2ND EDITION

C# Data Structures and Algorithms

Harness the power of C# to build a diverse range of efficient applications

MARCIN JAMRO

C# 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 or its dealers and distributors, will be held liable for any damages caused or alleged to have been 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.

Group Product Manager: Kunal Sawant

Publishing Product Manager: Teny Thomas

Book Project Manager: Manisha Singh

Senior Editor: Nithya Sadanandan

Technical Editor: Rajdeep Chakraborty

Copy Editor: Safis Editing

Indexer: Rekha Nair

Production Designer: Gokul Raj S.T

DevRel Marketing Coordinator: Shrinidhi Manoharan

Business Development Executive: Kriti Sharma

First published: April 2018

Second edition: February 2024

Production reference:1290124

Published by

Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN 978-1-80324-827-1

www.packtpub.com

Contributors

About the author

Marcin Jamro, PhD, DSc (dr hab. inż. Marcin Jamro) is a reliable entrepreneur, a helpful expert, and an experienced developer, with significant international experience. He held the role of CEO at a few technological companies, operated as CTO at companies in various countries, and also worked at Microsoft Corporation in Redmond, USA. Marcin shares his knowledge as an expert in international projects and invests in modern solutions.

He is the author of a few books on software engineering, as well as the author of numerous publications. The results of his research were presented and discussed at many scientific conferences. He has MCPD, MCTS, MCP, and CAE certificates. Marcin is a multiple laureate, finalist, and mentor in various competitions. He received the Primus Inter Pares medal and also, a few times, a scholarship of the Minister of Science and Higher Education for scientific achievements.

Marcin has significant experience in project development, especially with the C# language and .NET-based technologies. He has performed the role of lead architect on numerous complex software projects, including web and mobile applications, APIs, databases, and integration with external components.

You can learn more about the author at his website: https://marcin.com.

About the reviewer

Alexej Sommer is a professional software developer and technology enthusiast with expertise in a variety of technologies, mainly .NET and Azure. He is MCP/MCSD/Azure Developer/PSM I-certified.

Alexej is a former Microsoft Student Partner and Most Valuable Professional in the Windows Development category.

Currently, he occasionally participates as a speaker at IT conferences and meetups.

Table of Contents

Preface

1

Data Types

C# as a programming language

.NET-based console applications

Division of data types

Value types

Integral numbers

Floating-point numbers

Boolean values

Unicode characters

Constants

Enumerations

Value tuples

User-defined structs

Nullable value types

Reference types

Objects

Strings

Classes

Records

Interfaces

Delegates

Dynamics

Nullable reference types
Summary

2

Introduction to Algorithms

What are algorithms?
Definition
Real-world examples
Notations for algorithm representation
Natural language
Flowchart
Pseudocode
Programming language
Types of algorithms
Recursive algorithms
Divide and conquer algorithms
Back-tracking algorithms
Greedy algorithms
Heuristic algorithms
Dynamic programming
Brute-force algorithms
Computational complexity
Time complexity
Space complexity
Summary

3

Arrays and Sorting

Single-dimensional arrays

Example – month names

Multi-dimensional arrays

Example – multiplication table

Example – game map

Jagged arrays

Example – yearly transport plan

Sorting algorithms

Selection sort

Insertion sort

Bubble sort

Merge sort

Shell sort

Quicksort

Heap sort

Performance analysis

Summary

4

Variants of Lists

Simple lists

Array lists

Generic lists

Sorted lists

Example – address book

Linked lists

Singly linked lists

Doubly linked lists
Circular singly linked lists
Circular doubly linked lists
List-related interfaces
Summary

5

Stacks and Queues

Stacks
Example – reversing a word
Example – Tower of Hanoi
Queues
Example – call center with a single consultant
Example – call center with many consultants
Priority queues
Example – call center with priority support
Circular queues
Example – gravity roller coaster
Summary

6

Dictionaries and Sets

Hash tables
Example – phone book
Dictionaries
Example – product location
Example – user details

Sorted dictionaries
Example – encyclopedia
Hash sets
Example – coupons
Example – swimming pools
“Sorted” sets
Example – removing duplicates
Summary

7

Variants of Trees

Basic trees
Implementation
Example – hierarchy of identifiers
Example – company structure
Binary trees
Traversal
Implementation
Example – simple quiz
Binary search trees
Implementation
Example – BST visualization
Self-balancing trees
AVL trees
Red-black trees
Tries
Implementation
Example – autocomplete

Heaps
Summary

8

Exploring Graphs

The concept of graphs
Applications
Representations
Adjacency list
Adjacency matrix
Implementation
Node
Edge
Graph
Example – undirected and unweighted edges
Example – directed and weighted edges
Traversal
Depth-first search
Breadth-first search
Minimum spanning tree
Kruskal's algorithm
Prim's algorithm
Example – telecommunication cable
Coloring
Example – voivodeship map
Shortest path
Example – path in game
Summary

9

See in Action

The Fibonacci series

Minimum coin change

Closest pair of points

Fractal generation

Rat in a maze

A Sudoku puzzle

Title guess

A password guess

Summary

10

Conclusion

Classification

Arrays

Lists

Stacks

Queues

Dictionaries

Sets

Trees

Graphs

The last word

Index

Other Books You May Enjoy

Preface

Hello, I am Marcin!

It is so nice to meet you and to invite you to an amazing journey through various data structures and algorithms presented in this book. As you could already know, developing applications is certainly something exciting to work on, but it is also challenging, especially if you need to solve some complex problems. In such cases, you often need to take care of performance to ensure that the solution will work smoothly on devices with limited resources. Such a task could be really difficult and could require significant knowledge regarding not only the programming language but also data structures and algorithms. However, have you ever thought profoundly about them and their impact on the performance of your applications? If not, it is high time to take a look at this topic, and this book is a great place to start!

Did you know that replacing even one data structure with another could cause the performance results to increase hundreds of times or even more? Does it sound impossible? Maybe, but it is true! As an example, I would like to tell you a short story about one of the projects in which I was involved some time ago. The aim was to optimize the algorithm to find connections between blocks on a graphical diagram. Such connections should be automatically recalculated, refreshed, and redrawn as soon as any block has moved in the diagram. Of course, connections cannot go through blocks and cannot overlap other lines, and the number of crossings and direction changes should be limited. Depending on the size and the complexity of the diagram, the performance results differ significantly. However, while conducting tests, we received results in the range of 1 ms to almost 800 ms for the same test case. What was perhaps the most surprising aspect is that such a huge improvement was reached mainly by... changing the data structures of two sets.

Are you interested in knowing the influence of choosing a suitable data structure on the performance of your application? Do you want to know how you can increase the quality and performance of your solution by choosing the right accompanying algorithm? Are you curious about real-world scenarios where various data structures can be applied, as well as which algorithms could be used to solve some common problems? Unfortunately, the answer to such questions is not simple. However, within this book, you will find a lot of information about data structures and algorithms, presented in the context of the C# programming language, with many examples, code snippets, illustrations, and detailed explanations. Such content could help you to answer the aforementioned questions while developing the next great solutions, which could be used by many people all over the world! Are you ready to start your adventure with data structures and algorithms? If so, welcome on board of this book!

The book covers many data structures, starting with simple ones, namely arrays and a few of their variants, as representatives of random access data structures. Then, lists are introduced, together with their sorted, linked, and circular versions. The book also explains limited access data structures, based on stacks and queues, including a priority and a circular queue. Following this, we introduce you to the dictionary data structure, which allows you to map keys to values and perform fast lookup. The sorted variant of a dictionary is supported, as well. If you want to benefit from high-performance, set-related operations, you can use another data structure, namely a hash set. One of the most powerful constructs is a tree, which exists in many variants, including a binary tree, a binary search tree, a self-balancing tree, a trie, and a heap. The last data structure we analyze is a graph, supported by many interesting algorithmic topics, such as graph traversal, minimum spanning tree, node coloring, and finding the shortest path.

Arrays, lists, stacks, queues, dictionaries, hash sets, trees, tries, heaps, and graphs, as well as accompanying algorithms – a broad range of subjects await you on the next pages! Let's start the adventure and take the first step toward your mastery of data structures and algorithms, which hopefully will have a positive effect on your projects and on your career as a software developer!

Who this book is for

The book is aimed at developers who would like to learn about the data structures and algorithms that can be used with the C# language in various kind of applications, including web and mobile solutions. The topics presented here are suitable for programmers with various levels of experience, and even beginners will find interesting content. However, having at least basic knowledge of the C# programming language, such as about object-oriented programming, will be an added advantage.

It is worth noting that this book can contain some simplifications to make the subject easier to understand. What's more, some data structures are only mentioned briefly, without detailed explanations or examples. The book's aim is to interest you in the topic of data structures and algorithms, so you can further expand your knowledge with other books, research papers or on-line resources.

To easily understand the content, the book is equipped with many illustrations and examples. What is more, the source code for the accompanying projects is attached to the chapters and is available in the GitHub repository. Thus, you can easily run example applications and debug them without writing the code on your own.

It is worth mentioning that the code can be simplified and it can differ from the best practices. The examples can have significantly limited, or even no, security checks and functionalities. Before publishing your application using the content presented in the book, the application should be thoroughly tested to ensure that it works correctly in various circumstances, such as in the scenario of passing incorrect data.

What this book covers

Chapter 1, *Data Types*, introduces you to the topic of the data types available while developing applications in the C# programming language, both value and reference ones. You will learn about the built-in value types, such as integral or floating-point numbers, as well as constants, enumerations, value tuples, user-defined struct types, and nullable value types. As for the reference types, you will see object and string types, classes, records, interfaces, and delegate and dynamic types, as well as nullable reference types.

Chapter 2, *Introduction to Algorithms*, presents you with an algorithm definition and some real-world examples from your daily life. Then, you will learn a few notations for algorithms, namely natural language, a flowchart, pseudocode, and a programming language. Various types of algorithms are shown as well, including recursive, divide and conquer, back-tracking, greedy, heuristic, brute force, and dynamic programming. Computational complexity, including time complexity, is also presented and explained.

Chapter 3, *Arrays and Sorting*, covers scenarios storing data using some representatives of random access data structures, namely arrays. First, three variants of arrays are explained, that is, single-dimensional, multi-dimensional, and jagged. You'll also get to know seven popular sorting algorithms, namely selection sort, insertion sort, bubble sort, merge sort, Shell sort, quicksort, and heap sort. All of these algorithms are shown with illustrations, implementation code, and detailed explanations.

Chapter 4, *Variants of Lists*, deals with other representatives of random access data structures, namely a few variants of lists. Lists are similar to arrays, but make it possible to dynamically increase the size of the collection if necessary. A few variants of lists, including a singly linked list, a doubly linked list, a circular singly linked list, and a circular doubly linked list are introduced. Such data structures are presented in illustrations, are implemented using C# code, and are explained in detail.

Chapter 5, *Stacks and Queues*, explains how to use two variants of limited access data structures, namely stacks and queues, including priority and circular queues. The chapter shows how to perform push and pop operations on a stack, and also describes the enqueue and dequeue operations in the case of a queue. To aid your understanding of these topics, a few examples are presented, including the Tower of Hanoi game and an application that simulates a call center with multiple consultants and callers.

Chapter 6, *Dictionaries and Sets*, focuses on data structures that make it possible to map keys to values, perform fast lookups, and carry out various operations on sets. The chapter introduces you to

both non-generic and generic variants of a hash table, the sorted dictionary, and the high-performance solution to set operations allowing you to get union, intersection, subtraction, and symmetric difference. The "sorted" set concept is shown, as well.

Chapter 7, *Variants of Trees*, describes a few tree-related topics. First, it presents a basic tree, together with its implementation in C#, and examples showing it in action. The chapter also introduces you to binary trees, binary search trees, and self-balancing trees, namely AVL and red-black trees. Then, you'll see a trie that is a great approach for performing string-related operations. The remainder of the chapter is dedicated to a brief introduction to the topic of binary heaps as other tree-based structures.

Chapter 8, *Exploring Graphs*, contains a lot of information about graphs, starting with an explanation of their basic concepts, including nodes and a few variants of edges. The C#-based implementation of a graph is also covered. The chapter introduces you to two modes of graph traversal, namely depth-first and breadth-first search. Then, it presents the subject of minimum spanning trees using Kruskal's and Prim's algorithms, the node coloring problem, and finding the shortest path in a graph using Dijkstra's algorithm.

Chapter 9, *See in Action*, presents a few examples of various types of algorithms. You'll see a recursive way to calculate a number from the Fibonacci series, with dynamic programming optimization. Then, you'll learn a greedy approach to a minimum coin change problem, a divide-and-conquer method of getting the closest pair of points, a recursive way of generating fractals, a back-tracking and recursive way of solving rat in a maze and Sudoku puzzles, as well as a genetic algorithm and brute-force password guessing.

Chapter 10, *Conclusion*, is the conclusion of all the knowledge acquired from the previous chapters. It shows a brief classification of data structures, dividing them into two groups, namely linear and nonlinear. Finally, the chapter talks about the diversity of applications of various data structures. Each data structure is presented with a brief description and some of them are also shown with illustrations, to help you remember the information learned while reading the previous nine chapters.

To get the most out of this book

The book is aimed at programmers with various experience. However, beginners will also find some interesting content. Nevertheless, at least a basic knowledge of C#, such as object-oriented programming, will be an added advantage.

You need to have the **Visual Studio** IDE installed, in the version supporting **.NET 8.0**. You can download **Visual Studio 2022 Community** for free from https://visualstudio.microsoft.com/free-

developer-offers/. The detailed step-by-step instructions on how to install it can be found at https://learn.microsoft.com/en-us/visualstudio/install/install-visual-studio?view=vs-2022. Please keep in mind that the **.NET desktop development** option should be chosen in the **Workloads** tab while configuring the installation.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/C-Sharp-Data-Structures-and-Algorithms---Second-Edition If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, folder names, filenames, file extensions, pathnames, and user input. Here is an example: "The class contains three properties, namely **`Id`**, **`Name`**, and **`Role`**, as well as two constructors."

A block of code is set as follows:

```
int[,] numbers = new int[,]
{
    { 9, 5, -9 },
    { -11, 4, 0 },
    { 6, 115, 3 },
    { -12, -9, 71 },
    { 1, -6, -1 }
};
```

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

```
enum CurrencyEnum { Pln, Usd, Eur };
```

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

```
Name     Birth date   Temp. [C] -> Result
Marcin   09.11.1988        36.6 -> Normal
```

```
Adam     05.04.1995        39.1 -> High
Martyna  24.07.2003        35.9 -> Low
```

Bold: Indicates a new term, an important word, an important sentence, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "It sounds complicated, but fortunately, it is very simple. A jagged array could be understood as **a single-dimensional array, where each element is another array**."

TIPS OR IMPORTANT NOTES

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *C# Data Structures and Algorithms*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/978-1-80324-827-1

2. Submit your proof of purchase
3. That’s it! We’ll send your free PDF and other benefits to your email directly

1

Data Types

Welcome to the first chapter in which you'll start your amazing adventure with data structures and algorithms in the context of the **C# programming language**. First, we'll provide a short introduction to this language. You will get to know how broad its possibilities are, in how many scenarios you can apply this language, as well as some basic constructions that you can use. This isn't a C# course, so we won't be presenting various features one by one, and only a brief description will be provided.

The remaining part of this chapter is dedicated to **data types**, both built-in and user-defined, which you can use in your applications. First, you will learn what the difference is between value and reference types. Then, you will go through various available data types, starting with value types. Here, we will cover integral numeric types, floating-point numeric types, Boolean types, Unicode characters, constants, enumerations, value tuples, struct types, and nullable value types. Finally, we'll cover reference types, including object and string types, as well as classes, records, interfaces, and delegates, together with dynamic and nullable reference types.

As you can see, you have quite a long journey before you. However, if you get to know the basics well, it will be much easier for you to get the most out of the content presented in the remainder of this book. I, as the author, am keeping my fingers crossed for you – good luck!

In this chapter, we will cover the following topics:

- C# as a programming language
- .NET-based console applications
- The division of data types between value and reference types
- Value types
- Reference types

C# as a programming language

As a developer, you have probably heard about many programming languages, including **C#**, **Java**, **C++**, **C**, **PHP**, and **Ruby**. In all of them, you can use various data structures, as well as implement algorithms, to solve both basic and complex problems. However, each language has a specificity that's visible while implementing data structures and accompanying algorithms. As mentioned previously, this book only focuses on the C# programming language. This is also the main topic of this section.

The C# language, pronounced *C sharp*, is a **modern, general-purpose, strongly typed, and object-oriented programming language that can be used while developing a wide range of applications**, such as web, mobile, desktop, distributed, and embedded solutions, as well as even games! It cooperates with various additional technologies and platforms, including **ASP.NET Core**, **XAML**, and **Unity**. Therefore, when you learn the C# language, as well as get to know more about data structures and algorithms in the context of this programming language, you can use such skills to create more than one particular type of software. What's more, even if you decide to change your primary programming language to another, your knowledge regarding data structures and algorithms will still be useful. You might be wondering, *how is this possible?* The answer turns out to be very simple – you will understand how various data structures work, how you can implement them, as well as how you can apply them to solve various problems with specialized algorithms. But let's go back to the C# language.

The current version of the language is **C# 12**. It is worth mentioning its interesting history in terms of various versions of the language (including 2.0, 3.0, 5.0, and 8.0) in which new features were added to increase language possibilities. When you take a look at the release notes for particular versions, you will see how the language is being improved and expanded over time to be a powerful and convenient solution for developers. New features are pretty awesome and can significantly simplify your work and allow you to refactor the code so that it's shorter, as well as easier to understand and maintain. That's great work that's been done by the team developing C# that you can now benefit from while writing your code.

The syntax of the C# programming language is similar to other languages, such as Java or C++. For this reason, if you know such languages, you should easily be able to understand the code written in C#. As an example, similarly as in the languages mentioned previously, the code consists of statements that end with semicolons (`;`). The curly brackets, namely `{` and `}`, are used to group statements.

There are a few categories of statements, including the following:

- **Selection statements** containing **`if`** and **`switch`**. The **`if`** statement allows you to conditionally execute code, depending on the provided condition. The **`switch`** statement makes it possible to choose a statement list to execute using the **pattern match**.
- **Iteration statements**, including **`do-while`**, **`while`**, **`for`**, and **`foreach`**. They are related to **loops** and are used to execute a part of code many times while the condition is met.
- **Jump statements** containing **`break`**, **`continue`**, and **`goto`**. They are used to control the execution of loops, such as to break it or move to the next iteration.
- **Exception-handling statements**, including **`throw`**, **`try-catch`**, **`try-finally`**, and **`try-catch-finally`**. They are connected to handling exceptions that can be thrown in various parts of the code.

Other statements exist as well, such as `lock`, `yield`, `checked`, `unchecked`, and `fixed`. You will see some of the statements presented in the preceding list in the code snippets shown in the following chapters of this book, along with explanations.

Developing various applications in the C# language is also simplified by the availability of many additional great features, such as **Language Integrated Query** (**LINQ**), which allows developers to get data from various sources, including SQL databases and XML documents, consistently. There are also some approaches to shorten the required code, such as using Lambda expressions, pattern matching, properties, expression-bodied members, records, and string interpolations. It is worth mentioning automatic **garbage collection**, which greatly simplifies the task of releasing memory.

WHERE CAN YOU FIND MORE INFORMATION?

You can learn more about the newest version of the C# language at https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-12. The language history is shown at https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history. A set of detailed information about the language reference (including value and reference types) is available at https://learn.microsoft.com/en-us/dotnet/csharp/language-reference. You can find there information about available data types with a set of details, including the supported value ranges and precisions, which are presented later in this chapter.

Of course, the solutions mentioned previously are only a very limited subset of features that are available while developing in C#. You will see some others in the following parts of this book, along with examples and detailed descriptions.

.NET-based console applications

To keep things simple, while reading this book, you will create many console-based applications, but the data structures and algorithms could be used in other kinds of solutions as well. The console-based applications will be created in **Microsoft Visual Studio 2022 Community**. This **integrated development environment** (**IDE**) is a comprehensive solution for developing various kinds of projects and is equipped with many great features that simplify the development and testing of your applications.

Just after launching the IDE, we can proceed by creating a new project. To create one, follow these steps:

1. Click on **File** | **New** | **Project** in the main menu.
2. Choose **Console App** on the right in the **Create a new project** window.
3. Type the name of the project (**Project name**), select a location for the files (**Location**), and enter the name of the solution (**Solution name**). Then, press **Next**.
4. In the **Additional information** window, set the framework version to **.NET 8.0 (Long Term Support)** and ensure that **Do not use top-level statements** is unchecked. If you are ready, click on the **Create** button to automatically create the project and

generate the necessary files.

Congratulations! You've just created the first project. But what's inside?

Let's take a look at the **Solution Explorer** window, which presents the structure of the project. It is worth mentioning that the project is included in the solution with the same name. Of course, a solution could contain more than one project, which is a common scenario while developing more complex applications. You can see how it works if you browse the GitHub repository of this book. It contains one solution with over 40 projects.

DON'T HAVE SOLUTION EXPLORER?

*If you cannot find the **Solution Explorer** window, you can open it by choosing the **View | Solution Explorer** option from the main menu. Similarly, you could open other windows, such as **Output** or **Class View**. If you cannot find a suitable window (for example, **C# Interactive**) directly within the **View** option, you can find it in the **View | Other Windows** node.*

The automatically generated project contains the **Dependencies** element, which presents additional dependencies used by the project. It is worth noting that you could easily add references by choosing the **Add Project Reference**, **Add Shared Project Reference**, or **Add COM Reference** option from the context menu of the **Dependencies** element. Moreover, you can install additional packages using **NuGet Package Manager**, which can be launched by choosing **Manage NuGet Packages** from the **Dependencies** context menu.

WRITE FROM SCRATCH OR REUSE EXISTING PACKAGES?

It is a good idea to take a look at packages that are already available before writing the complex module on your own since a suitable package could already be available for developers. In such a case, you could not only shorten the development time but also reduce the chance of introducing mistakes. However, please check the license conditions and ensure that the external module is reliable.

The **`Program.cs`** file contains the main code in C#. You could adjust the behavior of the application by changing the following default implementation:

```
// See https://aka.ms/new-console-template
   for more information
Console.WriteLine("Hello, World!");
```

The initial content of this file contains just two lines. The first contains the comment, while the other writes the following text in the console when the program is launched:

```
Hello, World!
```

It looks so simple and easy to modify, doesn't it? This is true and the default implementation of this file has changed significantly over the last few years due to the functionality of **top-level statements**. With these statements, you avoid many lines of code in which the **`Program`** class is defined, together with the **`Main`** static method, where the logic of the simple program is placed.

So, what will the default code look like if you disable the top-level statements? Let's take a look at it:

```
namespace GettingStarted
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

The preceding code contains the definition of the **`Program`** class within the **`GettingStarted`** namespace. This class contains the **`Main`** static method, which is called automatically when the application is launched.

Before proceeding, let's take a look at the structure of the project in the file explorer, not in the **Solution Explorer** window. Are such structures the same?

HOW TO OPEN A PROJECT DIRECTORY

*You could open the directory with the project in the file explorer by choosing the **Open Folder in File Explorer** option from the context menu of the project node in the **Solution Explorer** window.*

First of all, you can see the **`bin`** and **`obj`** directories, which are generated automatically. Both contain **`Debug`** and **`Release`** directories, whose names are related to the configuration set in the IDE. After building the project, a subdirectory of the **`bin`** directory (that is, **`Debug`** or **`Release`**) contains the **`net8.0`** directory with **`.exe`**, **`.dll`**, and **`.pdb`** files. What's more, there is no **`Dependencies`** directory, but there is the **`.csproj`** file, which contains the XML-based configuration of the project. Similarly, the solution-based **`.sln`** configuration file is located in the solution's directory.

IGNORE SOME FILES AND DIRECTORIES USING GIT

*If you are using a **version control system**, such as **Git**, you should ignore the **`bin`** and **`obj`** directories, as well as the **`.csproj.user`** file. I strongly encourage you to use a version control system for various projects, as well as to **commit and push changes frequently**. If you can, you can also try to automate the process of testing and deployment, such as by introducing **continuous integration** and **continuous delivery** (**CI/CD**). The introduction of such procedures can have a very positive impact on the quality and stability of your great applications, regardless of their types.*

As you already know how to create projects for the examples we'll be covering in this book, let's focus on the available data types and their basic divisions, as well as write some code!

Division of data types

While developing applications in the C# language, you could use various data types, which are divided into two main groups, namely **value types** and **reference types**. The difference between

them is quite simple – **a variable of a value type directly contains data, while a variable of a reference type just stores a reference to data, which is located somewhere else**.

Here's an illustration of this:

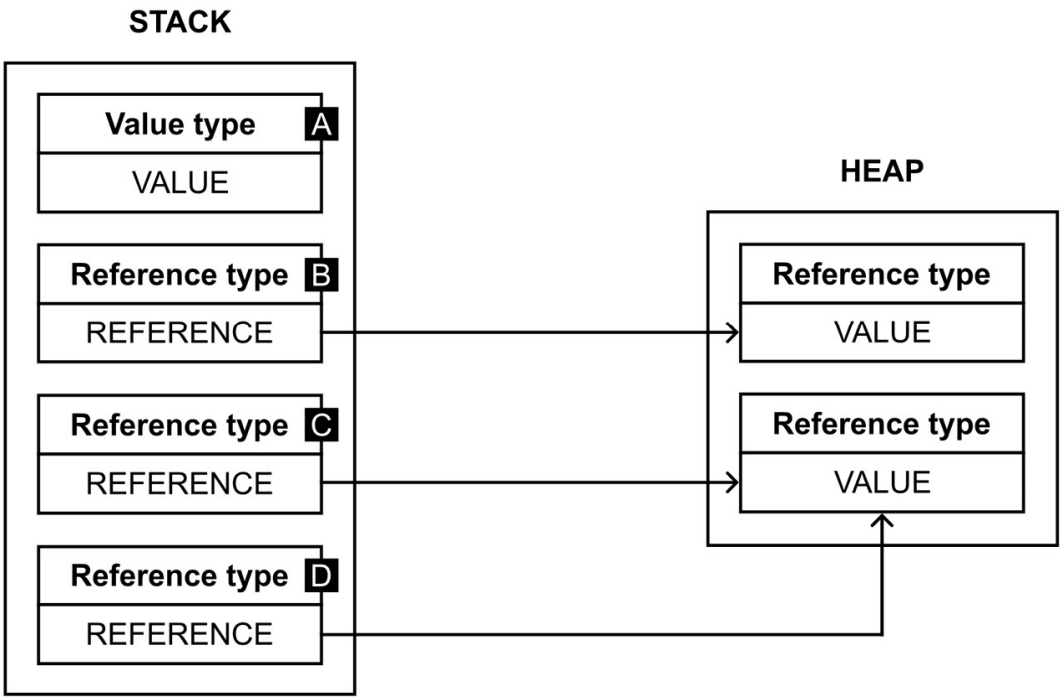

Figure 1.1 – The difference between value and reference types

As you can see, a variable of a **value type** (shown as **A**) stores its actual **value** directly in the **stack** memory, while a variable of a **reference type** only stores a **reference** here. The actual value is located in the **heap** memory. Therefore, it is possible to have two or more variables of a reference type that reference the same value, as indicated by the **C** and **D** boxes in the preceding figure.

BE CAREFUL – IT IS A SIMPLIFICATION!

Please remember that this is some kind of simplification because value types are not always stored on the stack. There are some scenarios when they are stored on the heap. If you are interested in this topic, you can read much more about it at https://tooslowexception.com/heap-vs-stack-value-type-vs-reference-type/.

Of course, a difference between value and reference types is important while programming and you should know which types belong to the groups mentioned previously. Otherwise, you could make mistakes in the code that could be quite difficult to find.

For instance, you should be careful while comparing two objects with the **equals operator** (==) since **two variables of a value type are equal if their data are equal, while two variables of a reference type are equal if they reference the same location**.

You should also take care while assigning a variable of a reference type to another variable and while passing a variable of a reference type as an argument to a method and updating its data. This is because the change can be reflected in other variables that are referencing the same object. In contrast, while using a value type, the variable value is copied while it's being passed as an argument to a method, returning a result from a method, or assigning it to another variable, so you only modify data in one location.

Does the division between value and reference types seem clear to you? If so, let's proceed to the next section, where the value types will be described in more detail, and some code snippets will be provided.

Value types

To give you a better understanding of data types, let's start by analyzing the first group, namely **value types**. They are further divided into the following categories:

- **Structs** encapsulating data and functionalities, which are divided into the following categories:
    - **Built-in value types**, also referred to as **simple types**. These are divided into:
        - **Integral numeric types**
        - **Floating-point numeric types**
        - **Boolean values**
        - **Unicode UTF-16 characters**
    - **Value tuples**
    - **User-defined struct types**
- **Constants**
- **Enumerations**

All of these groups will be described in this section, starting with the simple types.

Integral numbers

The first group of built-in value types are the **integral numeric types**, which allow you to store various **integer values**. Similar to other simple types, they can be used either as keywords or as types from the `System` namespace. Such types differ by the number of bytes used and whether signed or unsigned integral values are represented by them.

IMAGINE AN INTEGER VALUE

If you want to better visualize an integer value, you can find some examples around you – the publication year of this book, the number of legs of your table, and the number of keys on your keyboard. All of these are integer values, such as `2024`, `4`, and `84`. Yes, I counted the number of keys on my keyboard, especially for you!

The supported integral numeric types are as follows:

- **`Byte`** (the **`byte`** keyword), as 8-bit unsigned
- **`Sbyte`** (**`sbyte`**), as 8-bit signed
- **`Int16`** (**`short`**), as 16-bit signed
- **`Uint16`** (**`ushort`**), as 16-bit unsigned
- **`Int32`** (**`int`**), as 32-bit signed
- **`UInt32`** (**`uint`**), as 32-bit unsigned
- **`Int64`** (**`long`**), as 64-bit signed
- **`UInt64`** (**`ulong`**), as 64-bit unsigned
- **`System.IntPtr`** (**`nint`**), as 32- or 64-bit (platform-dependent) signed
- **`System.UintPtr`** (**`nuint`**), as 32- or 64-bit (platform-dependent) unsigned

As you can see, the types differ by the number of bytes for storing values and therefore by the range of available values. As an example, the **`byte`** data type supports values in the range from 0 to 255, **`sbyte`** from -128 to 127, **`short`** from -32,768 to 32,767, and **`uint`** from 0 to 4,294,967,295. Is it a huge number? Yes, it is. However, let's take a look at the range for **`ulong`**, which is from 0 to 18,446,744,073,709,551,615.

You can specify values for integral types in the following modes:

- **Decimal mode**: You do not use any prefix – for example, **`45`**.
- **Hexadecimal mode**: You apply **`0x`** or **`0X`** as a prefix – for example, **`0xff`** for 255.
- **Binary mode**: You apply **`0b`** or **`0B`** as a prefix – for example, **`0b1101110`** for 110. It can be also written as **`0b_0110_1110`** to improve the readiness of the number.

Here's an exemplary code snippet:

```
int a = -20;
byte b = 0x0f;
uint c = 0b01101110;
```

The last remark we'll make is about the default value for any built-in integral numeric type. You probably won't be surprised if I told you that it is **zero**.

Floating-point numbers

The second group of built-in value types are **floating-point numeric types**, which allow you to store **floating-point values**.

IMAGINE A FLOATING-POINT VALUE

If you want to better imagine a floating-point value, you can measure your current body temperature in Celsius degrees, get your height in centimeters, count the money that you currently have in your wallet, or take a look at your computer's processor frequency provided in GHz. All of these values are floating-point ones – for example, `36.6`, `184.8`, `105.34`, and `1.7`.

You can use three floating-point numeric types:

- **`Single`** (**`float`**) using 32 bits – for example, **`1.53f`** (**`f`** or **`F`** suffix)
- **`Double`** (**`double`**) using 64 bits – for example, **`1.53`** (no suffix or **`d`**/**`D`** suffix)
- **`Decimal`** (**`decimal`**) using 128 bits – for example, **`1.53M`** (**`m`** or **`M`** suffix)

Let's take a look at the following code:

```
float temperature = 36.6f;
double reading = -4.5178923;
decimal salary = 10000.47M;
```

As the number of used bits differs from 32 up to 128 bits, the range and precision of values differ significantly. Just take a look at the numbers:

- **`float`** stores numbers between $\pm 1.5 \times 10^{-45}$ and $\pm 3.4 \times 10^{38}$
- **`double`** stores numbers between $\pm 5.0 \times 10^{-324}$ and $\pm 1.7 \times 10^{308}$
- **`decimal`** stores numbers between $\pm 1.0 \times 10^{-28}$ and $\pm 7.9228 \times 10^{28}$

You might be surprised that even though **`decimal`** values use twice as many bits as the **`double`** type does, its range is significantly smaller. However, the **`decimal`** type is a good choice for monetary calculations.

In the end, remember that the default value for any floating-point numeric type is **zero**.

Boolean values

Regarding **Boolean values**, you can use the **`Boolean`** type or the **`bool`** keyword. This makes it possible to store a **logical value**. It represents one of two values, namely **`true`** or **`false`**. The default value is **`false`**.

IMAGINE A BOOLEAN VALUE

*If you want to visualize a Boolean value, answer the following questions: are you currently reading this book? Do you have at least 5 years of experience? Have you finished university? You can only answer these questions with yes (**`true`**) or no (**`false`**). No other answers are accepted. So, you can store such replies as Boolean variables.*

Let's take a look at the following code:

```
bool isTrue = true;
bool first = isTrue || false;  // true
bool second = isTrue && false; // false
bool third = 50 > 10;          // true
```

Sometimes, it is necessary to use the **three-valued Boolean logic**, which allows you to use not only `true` and `false` values, but also the **no-decision value** – that is, `null`. In such circumstances, you can benefit from the nullable Boolean type (`bool?`), which also supports three-valued logic. You'll learn more about nullable value types later.

Unicode characters

The last built-in value type we'll mention here is the **Unicode UTF-16 character**, which is represented by the `Char` type or the `char` keyword. It represents a single Unicode character.

IMAGINE A CHARACTER

If you want to understand what a Unicode character is, please write your first name on a piece of paper, separating the following letters. Each one is a character – for example, `M`, `a`, `r`, `c`, `i`, and `n`. You can use also a character to indicate the gender of a person – that is, `m` (for male), `f` (for female), and `o` (for other). As we are talking about UTF-16 encoding, a lot more values can be stored using a `char` variable, including symbols such as ©, ﷺ, or Ÿ. But that's not all – you can use also geometric symbols, such as ▶ and ◉, or even mathematical ones, such as ⅔ or ∑.

A `char` value can be specified using the following:

- **Character literal** – for example, `'a'` or `'M'`
- **Unicode escape sequence**, starting with `\u` – for example, `'\u25cf'` for ●
- **Hexadecimal escape sequence**, starting with `\x` – for example, `'\x107'` for ć

The exemplary code snippet is as follows:

```
char letter = 'M';
char bullet = '\u25cf';
char special = '\x107';
```

The default value for a `char` value is `\0` (`U+0000`).

Constants

It is worth noting that you can also define **constant values**. **Each constant value is an immutable value that cannot be changed**. Constant values can only be created from simple types.

IMAGINE A CONSTANT

If you want to remember constant values easily, think about some immutable values, such as the number of days in a week (`7`), the number of millimeters in a centimeter (`10`), the highest acceptable temperature value for a sensor (`90`), or the maximum number of iterations for your algorithm (`5`). None of these values can be changed after creation and can be defined as constants.

You can use the **`const`** keyword to create a constant value, as shown in the following line:

```
const int DaysInWeek = 7;
```

Another interesting fact is that the constant expressions for which all operands are constant values of simple types are evaluated at compile time. This has a positive impact on the performance of your application.

Enumerations

Apart from the already mentioned types, the value types contain **enumerations**. **Each has a set of named constants to specify the available set of values**.

IMAGINE AN ENUMERATION

If you want to better visualize an enumeration, try to specify available colors for your car while configuring it (for example, black, white, gray, red, and yellow), languages supported by your app (for example, English, Polish, and German), or currencies in which you accept payments (for example, PLN, USD, and EUR). In all of these scenarios, there is a precisely defined list of available values to select, so they are good representatives of enumerations.

An example definition is as follows:

```
enum CurrencyEnum { Pln, Usd, Eur };
```

The constants have automatically assigned values as integer numbers, starting with `0`. This means that the **`Pln`** constant is equal to `0`, while **`Eur`** is equal to `2`. What's more, the default value for the enumeration is `0`, which means that it is **`Pln`** in this case.

You can use the defined enumeration as a data type, as follows:

```
CurrencyEnum currency = CurrencyEnum.Pln;
switch (currency)
{
    case CurrencyEnum.Pln: /* Polish zloty */ break;
    case CurrencyEnum.Usd: /* American Dollar */ break;
    default: /* Euro */ break;
}
```

Please keep in mind that if you place the preceding code in the **`Program.cs`** file – that is, the line containing the enumeration definition and then a few lines of code with the **`switch`** statement – you will receive an error stating **Top-level statements must precede namespace and type declarations**. This means that the declaration of the enumeration declaration must be placed at the end of the code, as shown here:

```
CurrencyEnum currency = CurrencyEnum.Pln;
switch (currency)
{
    case CurrencyEnum.Pln: /* Polish zloty */ break;
    case CurrencyEnum.Usd: /* American Dollar */ break;
    default: /* Euro */ break;
}
enum CurrencyEnum { Pln, Usd, Eur };
```

This note is not related to enumerations only as you could receive a similar error while using other types, such as records or classes. So, please remember the rule and **place type declaration at the end**, even if they are presented in this book before the remaining code.

SHOULD YOU ADD ALL THE CODE TO ONE FILE?

*In simple exemplary applications, there's nothing wrong with placing all the code within one file. However, if you are developing something even a bit more complex, I strongly encourage you to divide the whole solution into suitable projects, as well as to **put various type declarations in separate files**. When you need to create types (for example, enumerations, classes, or records) while reading the remaining parts of this book, it is assumed that you add them to new files. Each file should be named the same as the type that is declared within it. From my point of view, writing code has some similarities to creating art, so let's try to **write beautiful code that is not only correct and tested but also greatly arranged and organized**!*

You can also benefit from more advanced features of enumerations, such as changing the underlying type or specifying values for particular constants. You can even do more and use the enumeration as a **bit field** – that is, as a set of **flags** – as presented here:

```
[Flags]
enum ActionEnum
{
    None    = 0b_0000_0000, // 0
    List    = 0b_0000_0001, // 1
    Details = 0b_0000_0010, // 2
    Add     = 0b_0000_0100, // 4
    Edit    = 0b_0000_1000, // 8
    Delete  = 0b_0001_0000, // 16
    Publish = 0b_0010_0000  // 32
}
```

Here, you can see the `ActionEnum` enumeration, which represents various actions that are allowed for users of the blog module, such as listing posts, showing details of a particular post, as well as adding, editing, deleting, and publishing a post. The constants have the following powers of two assigned, starting with `0` (`None`). The values are 2^0 (`1`), 2^1 (`2`), 2^2 (`4`), 2^3 (`8`), 2^4 (`16`), and 2^5 (`32`). These values are provided using the binary literal. Have you noticed that in each binary value, the `1`s are located in different places and everywhere there is only one `1`? Thanks to this, you can freely combine various flags, simply by using the `OR` binary operation, which is indicated by the `|` operator, as shown here:

```
ActionEnum guest = ActionEnum.List;
ActionEnum user = ActionEnum.List | ActionEnum.Details;
ActionEnum editor = ActionEnum.List | ActionEnum.Add
    | ActionEnum.Edit;
```

For example, if you apply `List` and `Details` permissions to a user, the combined permission is equal to `00000011`. The admin with full access to the system has combined permission equal to `00111111`. Simple and efficient, isn't it?

It's worth mentioning that enumerations allow you to replace some *magical strings* (such as `Pln` or `Usd`) with constant values. This has a very positive impact on code quality. What's more, it significantly simplifies refactoring, maintenance, and introducing changes in the code in the future.

Value tuples

The value tuple type is represented by the `System.ValueTuple` type and is **a lightweight data structure that allows you to group multiple data elements of specific types**. It has a very simple syntax, in which you just need to specify the types of all data members and, optionally, provide their names. All of the elements are public fields, so a tuple type is a **mutable value type**. What's more, **it is a very data-centric type, so you cannot even define methods within it**.

IMAGINE A VALUE TUPLE

If you want to understand a value tuple more easily, stop for a moment to think about the programming problem when you need to return a result from a method that consists of two or three values, such as a price in the chosen currency and the used conversion rate from the base currency, or a set of statistics consisting of minimum, maximum, and average values. As a solution, you can define a dedicated class, record, or struct and use it as a return type. However, this data structure will only be used once, and it will unnecessarily complicate the project and future changes. Another solution is to use `out` parameters for the method. However, such parameters cannot be used in all circumstances. To solve this problem, you can use a value tuple type and simply specify the types of data members that are returned from the method.

Even though the value tuple type seems to be straightforward, it can be used in various scenarios. Let's take a look at the following code snippet, where we use a value tuple and specify the type of data members returned from the method:

```
(int, int, double) result = Calculate(4, 8, 13);
Console.WriteLine($"Min = {result.Item1}
    / Max = {result.Item2} / Avg = {result.Item3:F2}");
```

You can easily access such data members by the automatically generated `Item1`, `Item2`, and `Item3` fields. The result is as follows:

```
Min = 4 / Max = 13 / Avg = 8.33
```

If you don't want to use `Item1`, `Item2`, and so on, you can change the code to specify the names of variables, as presented here:

```
(int min, int max, double avg) = Calculate(4, 8, 13);
Console.WriteLine($"Min = {min} / Max = {max}
    / Avg = {avg:F2}");
```

Here, we'll **deconstruct** a tuple by explicitly declaring the type and name of each field. If you execute the code, the result in the console will be the same as what it was previously. So far, you know how to get a result value that is a value tuple type, but how can you initialize it and return it from the method? Let's take a look:

```
(int, int, double) Calculate(params int[] numbers)
{
    if (numbers.Length == 0) { return (0, 0, 0); }
    int min = int.MaxValue;
    int max = int.MinValue;
    int sum = 0;
    foreach (int number in numbers)
    {
        if (number > max) { max = number; }
        if (number < min) { min = number; }
        sum += number;
    }
    return (min, max, (double)sum / numbers.Length);
}
```

You can further simplify the code and make it more readable by specifying an **alias** for this value tuple type. You can do so by using the following line of code:

```
using Statistics = (int Min, int Max, double Avg);
```

Then, it can be used in the remaining part of the code, as follows:

```
Statistics Calculate(params int[] numbers)
{
    /* (...) */
    return (min, max, (double)sum / numbers.Length);
}
```

You can call the `Calculate` method as follows:

```
Statistics result = Calculate(4, 8, 13);
Console.WriteLine($"Min = {result.Min} / Max = {result.Max}
    / Avg = {result.Avg:F2}");
```

Now that you understand how value tuples work, let's move on to the next data type.

User-defined structs

Apart from using the previously mentioned value types, you can create **data-centric struct types** (also named **structure types**) and use them in your applications.

IMAGINE A STRUCT

If you want to better visualize a data-centric struct type, think about the readings that are obtained from a weather station. A single reading consists of the current values of temperature, pressure, and humidity. You can specify a type for such readings as a user-defined struct type with three immutable data members, namely for temperature, pressure, and humidity. Such values cannot be changed once you've received the results from the weather station.

Structs have some of the same capabilities as classes, which we'll cover later. However, there are some differences – for example, a structure does not support inheritance. Despite the similarities to classes, you should only use structs in scenarios when a type does not provide behavior or provide it in a small amount.

This means that **a user-defined struct type should be quite small, data-centric, and immutable**. The last characteristic can be achieved by using the **`readonly`** modifier for the whole structure and all its data members, as shown here:

```
public readonly struct Price
{
    public Price(decimal amount, CurrencyEnum currency)
    {
        Amount = amount;
        Currency = currency;
    }
    public readonly decimal Amount { get; init; }
    public readonly CurrencyEnum Currency { get; init; }
    public override string ToString()
        => $"{Amount} {Currency}";
}
```

Here, we define the **`Price`** immutable struct type, which has two auto-implemented read-only properties, specified using the **`init`** accessor. This allows us to set a value for such properties during the object's construction and restrict later modifications. The struct type has also its own implementation of the **`ToString`** method, formatting the object as the amount and currency, separated by space. There is also a constructor with two parameters that sets the values of both properties.

You can make this code a bit shorter, as follows:

```
public readonly struct Price(
    decimal amount, CurrencyEnum currency)
{
    public readonly decimal Amount { get; init; } = amount;
    public readonly CurrencyEnum Currency { get; init; }
        = currency;
    public override string ToString()
        => $"{Amount} {Currency}";
}
```

Don't forget about the declaration of the **`CurrencyEnum`** enumeration, together with the **`public`** access modifier, as follows:

```
public enum CurrencyEnum { Pln, Usd, Eur };
```

The usage of the **`Price`** struct is quite simple:

```
Price priceRegular = new(100, CurrencyEnum.Pln);
Console.WriteLine(priceRegular);
```

The result that's shown in the console is as follows:

```
100 Pln
```

Since we're talking about struct types, it is worth noting the **`with`** **expression, which allows you to create a copy of a structure type instance, together with changing values of some properties and fields**. You can achieve this using the **object initializer syntax while specifying which members should be modified and what values should be assigned**. Let's take a look at the following code:

```
Price priceDiscount = priceRegular with { Amount = 50 };
```

Here, we produce a copy of the **`priceRegular`** instance and set the value of the **`Amount`** property to **`50`**. However, values of the remaining properties are the same as in the **`priceRegular`** instance, so **`Currency`** is set to **`Pln`**.

To conclude the topic of user-defined structs, it's worth remembering that the default value for each of them is created by setting all reference-type fields to **`null`** and all value-type fields to their default values.

Nullable value types

Now that we've come to the end of this section regarding value types, think about a scenario where you need to store either a particular value, such as a numeric one (for example, 154), or information that a value is not provided. One of the possible solutions is to use two variables. The first specifies whether the value is provided (the **`bool`** type), while the other stores a numeric value (for example, of the **`int`** type). However, is it possible to only use one variable instead of two? The answer is yes! To achieve this, you can use a **nullable value type**, which can have all the values of its underlying value type and the **`null`** value.

IMAGINE A NULLABLE VALUE TYPE

*If you want to understand nullable value types more easily, think about the scenario of calculating the age of a user of your portal. If the user provides you with their date of birth, the task is pretty simple. However, what about a situation when such a date is missing? You cannot calculate their age, so the age variable can be set to **`null`** instead of an integer value. Here, you can use the **`int?`** type.*

As for its implementation, you can use the **`?`** operator just after a type name or use the **`System.Nullable<T>`** structure, which has the same effect, as shown here:

```
int? age = 34;
float? note = 5.5f;
Nullable<bool> isAccepted = null;
```

If you want to check whether a variable of a nullable value type represents **`null`**, you can compare it with **`null`** or use the **`HasValue`** property. Then, you can get a value using the **`Value`** property, as

presented here:

```
if (age != null) { Console.WriteLine(age.Value); }
if (note.HasValue) { Console.WriteLine(note.Value); }
```

Another interesting feature related to nullable value types is the **null-coalescing operator** (`??`)**, which returns a non-nullable value, if one exists, or uses another value instead of null**. Another way to achieve the same result is to use the `GetValueOrDefault` method. Examples for both are shown here:

```
int chosenAge = age ?? 18;
float shownNote = note.GetValueOrDefault(5.0f);
```

If the `age` variable is not `null`, it is assigned to `chosenAge`. Otherwise, `18` is set. In the second line of code, the `note` value is assigned to `shownNote` if it is not `null`. Otherwise, `5.0f` is applied. Seems simple, doesn't it?

While talking about the null-coalescing operator (`??`), you should also take a look at the **null-coalescing assignment operator**, namely `??=`. **It allows you to assign the value of the right-hand side operand to the left-hand side operand if the first one is equal to null.** You can use this syntax instead of the `??` operator, as presented in the following code:

```
DateTime date = new(1988, 11, 9);
int? age = GetAgeFromBirthDate(date);
age ??= 18; // The same as: age = age ?? 18;
int? GetAgeFromBirthDate(DateTime date)
{
    double days = (DateTime.Now - date).TotalDays;
    return days > 0 ? (int)(days / 365) : null;
}
```

Since we're presenting various `null`-related operators, let's introduce the **null conditional operator** as well. **It is represented by the ?. operator and returns null if the left-hand side operand is null. Otherwise, it is used as a standard dot operator**. An example is as follows:

```
string? GetFormatted(float? number)
    => number?.ToString("F2");
```

The `GetFormatted` method returns `null` if `null` is provided as the `number` parameter. Otherwise, it returns the number formatted using the specified format.

Don't forget that a default value for a nullable value type represents `null`. This means that the `HasValue` property returns `false`.

Reference types

The second main group of types is **reference types**. As a quick reminder, a variable of a reference type does not directly contain data because it just stores a reference to data, which is located somewhere else. In this group, you can find four built-in types, namely `object`, `string`, `delegate`, and `dynamic`. Moreover, you can declare **classes**, **records**, and **interfaces**. **Nullable reference types** exist as well. All of these types will be described in this section. Let's get started!

Objects

The `Object` class (the `object` alias) is declared in the `System` namespace and performs an important role while developing applications in C#. Why? Because all other types in the **unified type system** of C# inherit directly or indirectly from `Object`. This means that built-in value types, built-in reference types, as well as user-defined value types and user-defined reference types, are derived from the `Object` class.

IMAGINE AN OBJECT

If you want to understand the object type more easily, think about it as "something." As everything is "something," everything is an object. Representatives of various value types and reference types are objects. Oh no – objects are everywhere around you! ;-)

Let's take a look at a set of methods that are available for all objects:

- `ToString` returns a string representation of the object
- `GetType` returns a type of the instance
- `Equals` checks whether the object is equal to a given object
- `GetHashCode` uses the hash function and returns its result

As the `Object` type is the base entity for all value types, this means that it is possible to convert a variable of any value type (for example, `int` or `float`) into the `object` type, as well as to convert a variable of the `object` type into a specific value type. Such operations are named **boxing** (the first) and **unboxing** (the other), as shown here:

```
int age = 28;
object ageBoxing = age;
int ageUnboxing = (int)ageBoxing;
```

Although the preceding code looks simple, you should be careful while unboxing. If you try to cast `ageBoxing` to `bool` instead of `int`, the code compiles without any errors. However, it fails at runtime with a `System.InvalidCastException` error. The additional message informs you that it is impossible to cast an object of the `System.Int32` type to the `System.Boolean` type.

Strings

There is often a necessity to store some text values. You can achieve this using the **`String`** built-in reference type from the **`System`** namespace, which is also available using the **`string`** keyword. The **`string`** type is **a sequence of Unicode characters**. It can have zero characters (the empty string) or one or more characters, or the **`string`** variable can be set to **`null`**.

IMAGINE A STRING

If you want to better visualize a string, take a look at this sentence. It is a string! Your first name is a string too. Close this book for a moment and take a look out of your window. The name of your street is a string. That's not all – even a car number plate is a string. It is one of the most common types that you'll use frequently while developing applications, so please read this chapter and this book carefully since all the text in this book is a string as well!

You can perform various operations on **`string`** objects, such as **concatenation**. You can also access a particular character using the **`[]`** operator, as shown here:

```
string firstName = "Marcin", lastName = "Jamro";
int year = 1988;
string note = firstName + " " + lastName.ToUpper()
   + " was born in " + year;
string initials = firstName[0] + "." + lastName[0] + ".";
```

First, the **`firstName`** variable is declared, and the **`Marcin`** value is assigned to it. Similarly, **`Jamro`** is set as a value of the **`lastName`** variable. In the third line, we concatenate five elements (using the **`+`** operator), namely the current value of **`firstName`**, the space, the current value of **`lastName`** converted into uppercase (by calling **`ToUpper`**), the **`was born in`** string (with additional spaces), and the current value of **`year`**. In the last line, the first characters of the **`firstName`** and **`lastName`** variables are obtained using the **`[]`** operator, as well as concatenated with two dots to form the initials – that is, **`M.J.`** – which are stored as a value of the **`initials`** variable.

The **`Format`** method can also be used for constructing this string, as shown here:

```
string note = string.Format("{0} {1} was born in {2}",
    firstName, lastName.ToUpper(), year);
```

In this example, we specify the **composite format string** with three format items, namely **`firstName`** (represented by **`{0}`**), uppercase **`lastName`** (**`{1}`**), and **`year`** (**`{2}`**). The objects to format are specified as the following parameters of the method.

It is also worth mentioning the **interpolated string**, which uses **interpolated expressions** to construct a string. To create a string using this approach, the **`$`** character should be placed before **`"`**, as shown in the following example:

```
string note = $"{firstName} {lastName.ToUpper()}
   was born in {year}";
```

Interpolated strings are equipped with even more features, such as specifying interpolated expressions with **alignments** (for example, a positive `,10` value for the minimum 10 characters and right alignment, as well as a negative `,-10` value for the minimum 10 characters and left alignment) or the **format strings** (such as `:F2` for a floating-point number with two digits after a comma or `:HH:mm` for presenting an hour with minutes).

Here's some example code:

```
string[] names = ["Marcin", "Adam", "Martyna"];
DateTime[] dates = [new(1988, 11, 9), new(1995, 4, 25),
    new(2003, 7, 24)];
float[] temperatures = [36.6f, 39.1f, 35.9f];
Console.WriteLine($"{"Name",-8} {"Birth date",10}
    {"Temp. [C]",11} -> Result");
for (int i = 0; i < names.Length; i++)
{
    string line = $"{names[i],-8} {dates[i],10:dd.MM.yyyy}
        {temperatures[i],11:F1} -> {
        temperatures[i] switch
        {
            > 40.0f => "Very high",
            > 37.0f => "High",
            > 36.0f => "Normal",
            > 35.0f => "Low",
            _ => "Very low"
        }
    }";
    Console.WriteLine(line);
}
```

This example presents a simple table with body temperatures for three people, namely `Marcin`, `Adam`, and `Martyna`, together with their birth dates. The alignments (for example, `-8`) and format strings (for example, `dd.MM.yyyy`) are used. What's more, the **switch expression** with **pattern matching** and **relational patterns** is applied to present additional information regarding whether the temperature is very high (`> 40.0f`), high (`> 37.0f`), normal (`> 36.0f`), low (`> 35.0f`), or very low. The latter is the final case and is also called the **discard pattern**. This is represented by _ and matches all other values.

When you execute this code, you'll see the following result:

```
Name     Birth date   Temp. [C] -> Result
Marcin   09.11.1988        36.6 -> Normal
Adam     05.04.1995        39.1 -> High
Martyna  24.07.2003        35.9 -> Low
```

As you can see, the C# language is equipped with various possibilities, even related to just the `string` type. What's more, you can combine different features, such as the interpolation string with the `switch` statement and pattern matching to create code that is easy to understand and maintain.

However, you should keep in mind that **`string`** is not a typical reference type and its behavior is a bit different than in the case of other reference types. You can see such a difference while comparing two **`string`** variables using the **`==`** operator. Here, two **`string`** instances are the same if they contain the same sequence of characters, so it is a similar behavior as in the case of value types.

Classes

As mentioned previously, C# is an object-oriented language and supports declaring classes together with various members, including **constructors**, **finalizers**, **constants**, **fields**, **properties**, **indexers**, **events**, **methods**, and **operators**, as well as **delegates**. Moreover, classes support **inheritance** and implementing **interfaces**. Static, abstract, sealed, and virtual members are also available. What's more, various members of the class could have a different accessibility level, specified using one of the following **access modifiers**: **`public`**, **`protected`**, **`internal`**, **`private`**, and **`file`**. These access modifiers are mentioned together with classes, but you should remember that they can be used for some other types as well.

IMAGINE A CLASS

If you want to visualize a class, think about a vehicle. Each vehicle has some properties, namely brand, model, color, length, width, height, and weight. A vehicle can perform some actions, such as taking a given distance. You can also define more specific variants of vehicles, such as a car, a plane, and a boat. Each has the same properties as the base vehicle, as well as has some additional ones, such as a number plate and a fuel type (for example, petrol, diesel, or electric) for a car. It also has the action of a door opening. You can also create instances of such classes – for example, you can create three instances of a car that differ by model and number plate, as well as create an instance of a plane. Then, you can perform actions on such instances.

An example class is shown here:

```
public class Person
{
    private string _location = string.Empty;
    public string Name { get; set; }
    public required int Age { get; set; }
    public Person() => Name = "---";
    public Person(string name)
    {
        Name = name;
    }
    public void Relocate(string location)
    {
        if (!string.IsNullOrEmpty(location))
        {
            _location = location;
        }
    }
    public float GetDistance(string location)
        => DistanceHelpers.GetDistance(_location, location);
}
```

The `Person` class contains the `_location` private field with the default value set to an empty string (`string.Empty`) and two public **auto-implemented properties** (`Name` and `Age`). You will use the properties quite often while writing the code examples shown in this book, so let's stop for a moment to explain them.

Each property is a member of a class that provides a mechanism for reading and writing using **accessors**:

- `get` to return the property value
- `set` to assign a new value for the property
- `init` to set a value during object construction and prevent modifications

By combining such accessors, a property can be put in one of the following instances:

- **Read-only** with the `get` accessor and without the `set` accessor
- **Write-only** with the `set` accessor and without the `get` accessor
- **Read-write** with both `get` and `set` accessors

Another interesting feature is a required variant of the property, which is specified by the `required` keyword placed just after the access modifier, as shown in the case of the `Age` property. It requires that the client code initializes the property, and it is a good idea to mark properties as `required` if they should be initialized at the beginning of using the class instance. It is also worth noting that a property has an accessibility level as one of the access modifiers, including `public` and `private`.

Let's take a closer look at the example class:

- It contains a default constructor that sets a value of the `Name` property to `---` using the **expression body definition**.
- It contains a constructor that takes one parameter and sets the value of the `Name` property.
- It contains the `Relocate` method, which updates the value of the private field.
- It contains the `GetDistance` method, which calls the `GetDistance` static method from the `DistanceHelpers` class and returns the distance between two cities provided in kilometers. The implementation of the helper class is not shown in the preceding code. Note that if you are curious about how you can create a real implementation of the mechanism for calculating the distance between cities, you can take a look at *Chapter 8*, *Exploring Graphs*, where such an application of graphs will be mentioned.

You can create an instance of the class using the `new` operator. Then, you can perform various operations on the object that's been created, such as calling a method, as shown here:

```
Person person = new("Martyna") { Age = 20 };
person.Relocate("Rzeszow");
float distance = person.GetDistance("Warsaw");
```

As the C# language is still being developed and improved, new amazing features were introduced in the following versions and are related to classes as well. For example, with the newest version, you

can use some concepts that make it possible to significantly limit the amount of code. The following code shows some of them:

```
public class Person(string name)
{
    private string _location = string.Empty;
    public string Name { get; set; } = name;
    public required int Age { get; set; }
    public void Relocate(string? location) =>
        _location = location ?? _location;
    public float GetDistance(string location) =>
        DistanceHelpers.GetDistance(_location, location);
}
```

The preceding code performs almost the same role as the previous variant, but it is even shorter. If you also like such improvements, keep reading – you will see various possibilities of the C# language in the remaining chapters of this book.

Let's proceed to the next section, which is dedicated to records.

Records

In the recent versions of the C# language, another great reference type was introduced: **records**. **They provide you with a built-in functionality for encapsulating data and using value-based equality**. What does this mean? **Two records are equal if their record type definitions are identical and when values for each field in both records are the same.** This differs from a class, where two instances of the same class are equal only when they are referencing the same data. A record can be defined using the `record` or `record class` keyword.

VALUE TYPE RECORDS EXIST AS WELL

It is worth noting that `record struct` construction exists as well. This construction represents a value type with similar functionality. However, in this book, I'll focus on the reference type version only. Of course, you can try another version on your own.

One great thing about records is that a smaller amount of code needs to be written because the compiler automatically generates the public `init`-only properties (called **positional properties**) for all the parameters provided in the record declaration (called **positional parameters**) itself. A primary constructor with the parameters matching the positional parameters on the record declaration is created as well, together with the `Deconstruct` method with a set of `out` parameters, each one representing a positional parameter. **This means that this data type is data-centric and is intended to be immutable, providing a short and clear syntax.**

IMAGINE A RECORD

If you want to visualize a record, stand up and take a look at yourself in your mirror, focusing on your beautiful T-shirt. It has some properties, such as size (for example, S, M, or L), color (for example, white or red), and brand. All of these properties are immutable, so you cannot change them as you cannot resize your favorite T-shirt since it is produced. So, as your nice T-shirt is data-centric and its properties are immutable, it is a good representative of a record. Smile to yourself in your mirror and come back to reading the first chapter of this book!

Let's take a look at the following example of a record declaration:

```
public record Dog(string Name, string Breed, int Height,
    float Weight, int Age);
```

That's all! Now, you have a record with five immutable properties (**`Name`**, **`Breed`**, **`Height`**, **`Weight`**, and **`Age`**), as well as a constructor with five parameters related to the positional parameters on the record declaration. You can use it to create a new instance, as shown in the following line of code:

```
Dog rex = new("Rex", "Schnauzer", 40, 11, 5);
```

Pretty simple and clear, isn't it? What's more, you can use the built-in formatting by using the compiler-generated **`ToString`** method, which is a nice feature while debugging because you can easily see the values of all properties. To see how it works, add the following line of code:

```
Console.WriteLine(rex);
```

The result is as follows:

```
Dog { Name = Rex, Breed = Schnauzer, Height = 40,
    Weight = 11, Age = 5 }
```

As you can see, the name of the record is shown, together with the names and values of the following properties. Please keep in mind that properties are defined with **`get`** and **`init`** accessors, so their values can be read and cannot be changed after they are initialized. So, the following line will cause a compiler error:

```
rex.Name = "Puppy";
```

If you want to change this behavior, you can do so with a record without positional parameters on the record declaration, but by defining the particular properties using the standard syntax, as shown here:

```
public record Dog
{
     public required string Name { get; set; }
     public required string Breed { get; set; }
     public required int Height { get; set; }
     public required float Weight { get; set; }
     public required int Age { get; set; }
}
```

Another useful feature is a clear syntax for **non-destructive mutation**, which allows you to create a copy of an instance with some modifications in the values of properties. You can do so using the **`with`** expression, as shown here:

```
Dog beauty = rex with { Name = "Beauty", Height = 35 };
```

Here, an instance representing **`Beauty`** is created, based on **`Rex`**. All of the values of the properties are taken from **`Rex`**, apart from **`Name`** and **`Height`**, as specified after the **`with`** keyword. You also need to remember that you can adjust both positional properties and properties created using the standard syntax that have the **`init`** or **`set`** accessor. In such a case, a **shallow copy** is created, so for value types, a copy is used, while for reference types, only a reference is copied, so both a source and a target property will reference the same instance of a reference type.

Let's take a look at the following line:

```
(string name, _, _, _, int age) = beauty;
```

Do you know what happened here? If not, let's recall how to deconstruct a value tuple. You can deconstruct a record type as well. In the preceding line, you only get values for **`Name`** and **`Age`** positional properties, ignoring others using **discards** represented by underscore characters, namely **`_`**. As you can see, records are equipped with a lot of useful features, but there are even more of them, such as the support for inheritance.

Interfaces

Previously, we mentioned classes. They can implement one or more **interfaces**. This means that such a class must implement all methods, properties, events, and indexers that are specified in all implemented interfaces.

IMAGINE AN INTERFACE

*If you want to remember what an interface is, think about various things that you have on yourself, including a shirt, pants, and a watch. As all of these things have different sets of properties, you can create a dedicated class for each of them. However, how can you indicate which things can be worn and which can be washed in a washing machine? You can mark various classes with special indicators, such as "wearable" and "washable." This is where interfaces come to the rescue! You can create **`IWearable`** and **`IWashable`** interfaces. Then, you can implement **`IWearable`** by **`Shirt`**, **`Pants`**, and **`Watch`**, as well as implement **`IWashable`** by **`Shirt`** and **`Pants`** only. You can also require that everything washable must have a property regarding a maximum temperature for washing in a washing machine. Looks nice, doesn't it? But that's not all – you can also create a class regarding a washing machine with the **`Wash`** method, which takes the **`IWashable`** parameter, so you can pass **`Shirt`** or **`Pants`**. No watches are allowed here! Is this magic?*

You can easily define interfaces in the C# language using the **`interface`** keyword:

```
public interface IDevice
{
    string Model { get; set; }
    string Number { get; set; }
    int Year { get; set; }
    void Configure(DeviceConfiguration configuration);
    bool Start();
```

```
    bool Stop();
}
```

The `IDevice` interface contains three properties representing the following:

- A device model (`Model`)
- A serial number (`Number`)
- A production year (`Year`).

What's more, it contains the signatures of three methods:

- `Configure` for device setup. Please note that the `DeviceConfiguration` class is missing here, so try to prepare it on your own.
- `Start` for starting the operation.
- `Stop` for stopping the operation.

When a class implements the `IDevice` interface, it should contain all of these properties and methods, as presented in the following code snippet:

```
public class Display
    : IDevice
{
    public string Model { get; set; }
    public string Number { get; set; }
    public int Year { get; set; }
    public int Diagonal { get; set; }
    public void Configure(
        DeviceConfiguration configuration) { (...) }
    public bool Start() { (...) }
    public bool Stop() { (...) }
}
```

As you can see, the `Display` class contains all of the properties and methods specified in the interface it implements. However, you can add more elements to the class according to your preferences, such as the `Diagonal` property in this example.

Delegates

The `delegate` reference type **specifies the required signature of a method**.

IMAGINE A DELEGATE

If you want to understand what a delegate type is, think about various ways of calculating the mean of three numbers. You can get it as an arithmetic mean, as a geometric mean, as a harmonic mean, or even as a root mean square or a power mean. However, in all of these cases, you need a function that takes three parameters (for three numbers) and returns a number as the result. In this case, you can understand a delegate as a template for a way of calculating any of the mentioned means. Then, you can prepare an exact implementation of each calculation.

The delegate could then be instantiated, as well as invoked, as shown here:

```
Mean arithmetic = (a, b, c) => (a + b + c) / 3;
Mean geometric = delegate (double a, double b, double c)
    { return Math.Pow(a * b * c, 1 / 3.0); };
Mean harmonic = Harmonic;
double a = arithmetic.Invoke(5, 6.5, 7);
double g = geometric.Invoke(5, 6.5, 7);
double h = harmonic.Invoke(5, 6.5, 7);
Console.WriteLine($"{a:F2} / {g:F2} / {h:F2}");
double Harmonic(double a, double b, double c) =>
    3 / ((1 / a) + (1 / b) + (1 / c));
delegate double Mean(double a, double b, double c);
```

In this example, the `Mean` delegate specifies the required signature of a method for calculating a mean value of three floating-point numbers. It is instantiated with the following:

- A **Lambda expression** (`arithmetic`)
- An **anonymous method** (`geometric`)
- A **named method** (`harmonic`)

Each delegate is invoked by calling the `Invoke` method. The results are presented in the console, as follows:

```
6.17 / 6.10 / 6.04
```

Here, the Lambda expression is shown, so it is a good idea to tell you a bit more about such a construction. It uses the `=>` operator, which separates the input parameters and the **Lambda body**. The parameters are placed on the left-hand side of the operator and are specified here as `(a, b, c)`. They are the same as the parameters of the delegate. On the right-hand side, there is the Lambda body, which calculates the result as a sum of inputs divided by 3. While developing applications in C#, you will use Lambda expressions quite often and it is a nice feature that can have a positive impact on your code quality.

Dynamics

Apart from the types we've already described, `dynamic` is available for developers. **It allows you to bypass type checking during compilation so that you can perform it during runtime.** Such a mechanism can be useful while you're accessing some types of **application programming interfaces** (**APIs**).

IMAGINE A DYNAMIC TYPE

If you want to better visualize a dynamic type, ask someone nearby to put a blindfold on you and then give you a set of instructions on how to move from one room to another and sit in a chair – for example, walk 5 steps forward, turn right, walk 10 steps forward, and then sit down. If the instructions are correct, you will go to the other room and sit comfortably in the chair. However, if anything is wrong, you won't know this by listening to all the instructions at the beginning, only when

you hit a wall or sit on the floor instead of in the chair. This situation is somewhat similar to using dynamic types. If the instructions are correct, the application will work correctly, but if something is wrong, it may hurt. ;-)

You will not use the **`dynamic`** type while reading this book. However, to provide a short introduction to this feature, take a look at the code:

```
dynamic posts = await GetPostsAsync();
foreach (dynamic post in posts)
{
    string title = post.title;
    Console.WriteLine($"Title: {title}");
}
Task<dynamic> GetPostsAsync() { /* ... */ }
```

Here, we get data on posts via an external API by calling the **`GetPostsAsync`** method. The result is assigned to the **`posts`** variable with the **`dynamic`** type. Thus, we bypass **static type checking** so that the compiler won't check whether a field or property exists, as well as whether a called method is available. This could be understood as something cool in that the compiler will not throw any errors and warnings. However, it's not, because errors will be shown during runtime as runtime exceptions. In some scenarios, using the **`dynamic`** type allows you to significantly limit the amount of code, but it should be used with caution.

STRONGLY TYPED FEATURES ARE COOL!

The strongly typed features of the C# language give you great development support. Both errors and warnings are useful as they help you make your code robust and more reliable. Remember that warnings are not something that should exist in the production version of your application. You should always try to decrease the number of warnings to zero, as well as take into account various hints provided by the IDE.

In the preceding code, the **`await`** keyword is used. It is related to **asynchronous programming** and the **`async`** keyword. The **`await`** operator is placed in the line where the **`GetPostsAsync`** method is called. This means that the evaluation of the code is suspended until the asynchronous operation of getting posts from an external API is completed. Then, the **`await`** operator returns the collection of posts.

ASYNCHRONOUS PROGRAMMING IS COOL TOO!

*Asynchronous programming is a very interesting and powerful topic and is crucial for the development of robust and highly efficient applications. This topic also involves the **task-based asynchronous pattern**, a few dedicated data structures (including **`ConcurrentQueue`**, which will be covered in Chapter 5, Stacks and Queues), as well as some **synchronization primitives**.*

Are you ready to proceed to the last type that will be described in this chapter? If so, let's go!

Nullable reference types

Now that we've come to the end of this section regarding reference types, let's take a look at **nullable reference types**. It is an interesting group of features that aims to prevent developers from running into situations when the runtime throws exceptions of the `System.NullReferenceException` type. It is thrown when you access one of the members of a variable using the dot operator (`.`), when a variable is `null`. When using a new feature, you can explicitly mark a reference type as nullable using the `?` operator, similarly as in the case of nullable value types.

IMAGINE A NULLABLE REFERENCE TYPE

If you want to understand what nullable reference types are, just remind yourself that reference types allow `null` values. So, why am I talking about nullable reference types? They are a special feature that tells you "Be careful, it could be null!" Using nullable reference types could sometimes seem to be an unnecessary complication for reference types. However, when using them for a longer period, you will see that they have a positive impact on your code's quality and allow you to pay more attention to null reference issues and therefore could limit the number of errors while the program is running. I recommend that you familiarize yourself with this feature, even if it can be quite cumbersome at the beginning. I like it! What about you?

This feature is equipped with **null-state static analysis**, which tracks the null state of references to decide whether it is not-null or maybe-null. The latter case indicates that the code may dereference `null`, so the mechanism emits a warning that can be solved by you. The first solution is to add a conditional statement that checks whether the value is not equal to `null`. Another way is to use the **null-forgiving operator**, also named **null-suppression** (`!`), when you are certain that the variable is not `null` here.

Let's take a look at an example:

```
Random random = new();
List<Measurement?> measurements = [];
for (int i = 0; i < 100; i++)
{
    Measurement? measurement = random.Next(3) != 0
        ? new(DateTime.Now, random.Next(1000) / 1000.0f)
        : null;
    measurements.Add(measurement);
    Console.WriteLine(IsValid(measurement)
        ? measurement!.ToString()
        : "-");
    await Task.Delay(100);
}
static bool IsValid(Measurement? measurement)
{
    return measurement != null
        && measurement.Value >= 0.0f
        && measurement.Value <= 1.0f;
}
public record Measurement(DateTime ReadAt, float Value);
```

In the second line, we create a list. Each of its elements contains the value of a measurement or `null`, if the measurement is not received correctly from some external device. Within the `for` loop, we simulate getting a measurement by using the `Random` class. Statistically, about 2/3 of total

measurements are retrieved correctly (the `Measurement` instance is then created) and the remaining 1/3 of total measurements are not retrieved (`null` is used instead). Such values are added to the `measurements` list. Then, we need to check whether the obtained reading is correct – that is, apart from being provided, its value is within the range of `<0.0, 1.0>`. We perform this check using the `IsValid` static method while taking the nullable `Measurement` instance as a parameter.

Finally, we just need to show information about a reading in the console. For the correct reading, we present a formatted value of the `Measurement` instance. Otherwise, we use a dash (`-`). However, if we write `measurement.ToString()`, we will receive a warning stating "*Dereference of a possibly null reference*" because the compiler does not know that we're checking whether the measurement is not `null` within the `IsValid` method. We can avoid this warning using the null-forgiving operator by adding the `!` sign just after the variable name, before calling the `ToString` method. A small explanation may also be required for the line containing the `Delay` method call. It is only used to simulate the real behavior of the device, from which we read a measurement each 100 ms.

As you can see, both nullable value types and nullable reference types provide you with very similar semantics (such as `bool?` for a nullable value type and `string?` for a nullable reference type), but they are implemented in other ways. A nullable value type uses `System.Nullable<T>` internally (for example, `System.Nullable<System.Boolean>` for `bool?`), while a non-nullable value type uses another type (just `System.Boolean` for `bool`). The nullable reference type uses the same type for both nullable and non-nullable variants. This means that `string?` and `string` are provided by the `System.String` class in both cases.

Summary

This was only the first chapter of this book, but it contained quite a lot of information that will be useful while you're reading the remaining ones. First, the **C# programming language** was briefly presented with a focus on showing various data types, both value and reference ones. You learned the difference between them and why understanding this difference is so important while developing applications.

Next, you saw various **value types**, including the built-in ones, such as integral numeric types, floating-point numeric types, Boolean type, and Unicode characters. Then, you learned about constants, enumerations, value tuples, user-defined struct types, and nullable value types. All of these were equipped with detailed descriptions, as well as some code examples to make understanding easier and faster.

Finally, you learned about the second group of types, namely **reference types**. Here, you saw the object and string types, classes, records, interfaces, as well as delegate and dynamic types. Then, you

learned about nullable reference types. Again, a lot of information was supported by explanations and some code fragments.

With this introduction, you should be ready to proceed to the next chapter and learn **what algorithms are and why they are so important**. Let's go!

2

Introduction to Algorithms

While reading the first chapter of this book, you learned about various data types. Now, it is high time to introduce the topic of algorithms. In this chapter, you will take a look at their **definition**, as well as some real-world **examples**, **notations**, and **types**. As you should take care of the performance of your applications, the subject of computational complexity of the algorithms, including time complexity, will also be presented and explained.

First, it is worth mentioning that the topic of algorithms is very broad and complex. You can easily find a lot of scientific publications about them on the internet, published by researchers from all over the world. The number of algorithms is enormous and it is almost impossible to even remember the names of all the commonly used ones. Of course, some algorithms are simple to understand and implement, while others are extremely complex and almost impossible to understand without deep knowledge of algorithmics, mathematics, and other dedicated field of science. There are also various classifications of algorithms by different key features, and there are a lot of types, including recursive, greedy, divide-and-conquer, back-tracking, and heuristic. However, for various algorithms, you can specify the computational complexity by stating how much time or space they require to operate with the increasing size of a processed input.

Does this sound overwhelming, complicated, and difficult? Don't worry. In this chapter, I will try to introduce the topic of algorithms in a way that everyone can understand, not only mathematicians or other scientists. For this reason, in this chapter, you will find some simplifications to make this topic simpler and easier to follow. However, the aim is to introduce you to this topic and **make you interested in algorithms**, not create another research publication or book with a lot of formal definitions and formulas. Are you ready? Let's get started!

In this chapter, we will cover the following topics:

- What are algorithms?
- Notations for algorithm representation
- Types of algorithms
- Computational complexity

What are algorithms?

Did you know you typically use algorithms every day and that you are already an author of some algorithms, even without writing any lines of code or drawing a diagram? If this sounds impossible, give me a few minutes and read this section to get to know how is it possible.

Definition

First, you need to know what an **algorithm** is. It is a **well-defined solution for solving a particular problem or performing a computation**. It is an ordered list of precise **instructions** that are performed in a given order and take a well-defined **input** into account (if any) to produce a well-defined **output**, as shown here:

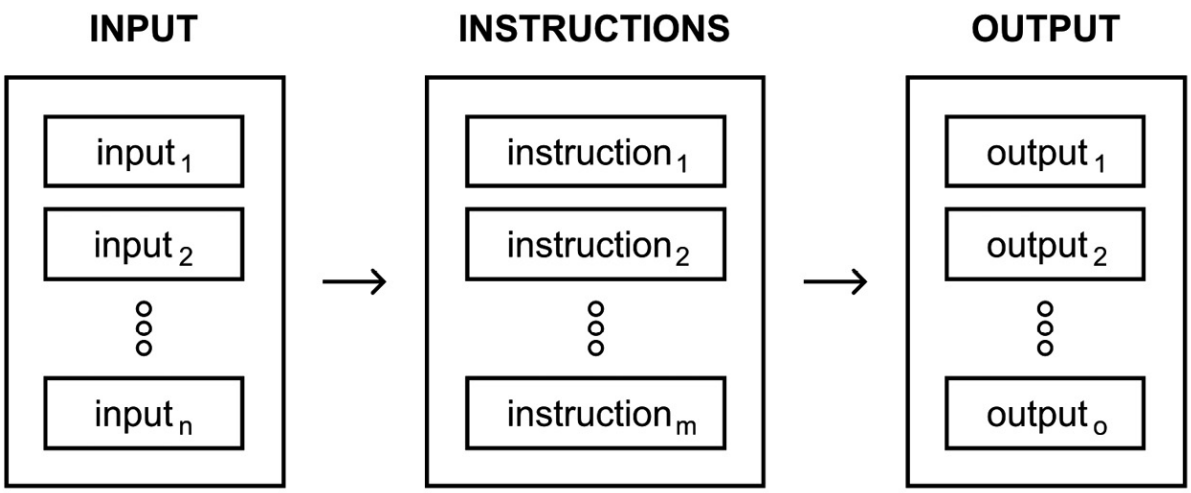

Figure 2.1 – Illustration of an algorithm

To be more precise, **an algorithm should contain a finite sequence of unambiguous instructions, which provides you with an effective and efficient way of solving the problem**. Of course, an algorithm can contain **conditional expressions**, **loops**, or **recursion**.

WHERE CAN YOU FIND MORE INFORMATION?

If you are interested in the topic of algorithms, you can find a lot of detailed information about them in many books, including Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Of course, there are also many resources available online, such as GeeksForGeeks (https://www.geeksforgeeks.org), The Algorithms (https://the-algorithms.com), and Algorithms, 4th Edition, by Robert Sedgewick and Kevin Wayne (https://algs4.cs.princeton.edu). A huge number of resources is also available if you browse the Algorithms topic on GitHub (https://github.com/topics/algorithms). I strongly encourage you to search for various resources, either in books or over the internet, and continue learning about algorithms when you've finished reading this book.

Real-world examples

With the definition of algorithms under your belt, you might be thinking, "*Come on – inputs, outputs, instructions… where I can find them?*" The answer turns out to be much simpler than you might expect because you can find such items almost everywhere, all the time!

Let's start with a **simple morning routine**. First, you wake up and take a look at your phone. If there are any notifications, you go through them and reply to urgent messages. For any unurgent items, you postpone them. Then, you go to the bathroom. If it is occupied, you wait until it is free, telling the person inside to hurry up. As soon as you are in the bathroom, you take a shower and brush your teeth. Finally, you choose suitable clothes according to the current weather and temperature. Surprise! Your morning routine is an algorithm. You can describe it as a set of instructions, which has some inputs, such as notifications and the current temperature, as well as outputs, such as chosen clothes. What's more, some of the instructions are conditional, such as only replying to urgent messages. Others can be executed in a loop, such as waiting until the bathroom is vacant.

The preceding morning routine also contains other algorithms, such as those for **unlocking a smartphone using face recognition**. It is an algorithm-based mechanism that you can use to ensure that only you can unlock your phone. What's more, even **organizing notifications on your phone** is the result of an algorithm that takes into account notifications as input, arranges them into groups, and sorts them suitably before presenting them to you.

At this point, you are dressed up and ready for a healthy and yummy breakfast. Imagine that you want to **prepare scrambled eggs** using your grandma's secret recipe. You need some ingredients, namely three eggs, salt, and pepper. As a result, you will have created an amazing dish for your perfect breakfast. First, you crack the eggs into a bowl and whisk them with a pinch of salt and pepper. Then, you melt butter in a non-stick skillet over medium-low heat. Next, you pour the egg mixture into the skillet and keep the eggs moving until there is no liquid egg. With that, your breakfast is ready. However, what is it if not a well-written and organized algorithm with a precise input and yummy output?

After breakfast, you need to go to work. So, you jump into your car and launch a navigation app on your smartphone to see **the fastest route to work** while taking the current traffic into account. This task is performed by complicated algorithms that can even involve **artificial intelligence** (**AI**), together with a computer-understandable representation of routes that use specialized data structures, as well as data obtained from other users. When combined, this forms traffic data. As you can see, the algorithm takes the complex input and performs various calculations to present you with an ordered list of route directions – for example, go to route A4, turn right to route S19, and follow this route until you reach your destination.

While at work, you need to prepare documents for your accountant, so you need to gather documents from colleagues, print some of them from emails, and then **sort all invoices by numbers**. How do you perform sorting? You take the first document from the stack and put it on the table. Then, you take the second document from the unsorted stack and put it either above, if the number is smaller than the first invoice, or below the previous one. Then, you take the third invoice and find a suitable place for it in the ordered stack. You perform this operation until there are no documents in the unsorted stack. Wow, another algorithm? Exactly! This is one of **sorting algorithms**. You'll learn about them in the next chapter.

It's time for a break at work! You launch your favorite social app and **receive suggestions for new friends**. However, how are they found and proposed to you? Yes, you're right – this is another algorithm that takes data from your profile and your activities, as well as the data of available users, as input, and returns a collection of best-suited suggestions for you. It can use many complex and advanced techniques, such as **machine learning** (**ML**) algorithms, which can learn and take your previous reactions into account. Just think for a second about the data structures that can be used in such cases. How you can organize your relationships with friends and how can you find out how many other people are between you and your favorite actor from Hollywood? It would be great to know that your friend knows Mary, who knows Adam, who is a friend of your idol, wouldn't it? Such a task can be accomplished using some graph-based structures, as you will see later in this book.

WILL YOU LEARN ABOUT AI ALGORITHMS IN THIS BOOK?

*Unfortunately, no. Due to the limited number of pages, various algorithms related to AI are not included in this book. However, note that it is a very interesting topic that involves many concepts, such as **ML** and **deep learning** (**DL**), which are used in many applications, including recommendation systems, speech-to-text, searching over extremely high amounts of data (the concept of **big data**), generating textual and graphical content, as well as controlling self-driving cars. To achieve these goals, a lot of interesting algorithms are used. I strongly encourage you to take a look at this topic on your own or choose one of Packt's books that focuses on AI-related topics.*

Are these examples enough? If not, just think about **choosing a movie in a cinema for the evening** while considering the AI-based suggestions of movies with geolocation-based data of cinemas, or **setting a clock alarm** depending on your plan for the next day. As you can see, **algorithms are everywhere and all of us use them, even if we do not realize it**.

So, if algorithms are so common and so useful, why don't we benefit from the huge collection of ones that are available or even write our own algorithms? There are still some problems that need to be solved using algorithms. I, as the author of this book, am keeping my fingers crossed for you to solve them!

Notations for algorithm representation

In the previous section, algorithms were presented in English. However, this is not the only way of specifying and documenting an algorithm. In this section, you will learn about four notations for algorithm representation, namely **natural language**, **flowchart, pseudocode**, and **programming language**.

To make this task easier to understand, you will specify the algorithm for calculating an **arithmetic mean** in all of these notations. As a reminder, the mean can be calculated using the following formula:

$$\mathbf{mean(a, n)} = \begin{cases} \text{null, if } n = 0 \\ \dfrac{\sum_{i=1}^{n} a_i}{n} = \dfrac{a_1 + a_2 + \ldots + a_n}{n}, \text{ otherwise} \end{cases}$$

Figure 2.2 – Formula for calculating an arithmetic mean

As you can see, two inputs are used, namely the provided numbers (*a*) and the total number of elements (*n*). If no numbers are provided, `null` is returned, indicating that no mean is available. Otherwise, you sum the numbers and divide them by the total number of elements to get the result.

Natural language

First, let's specify the algorithm using a natural language. It is a very easy way of providing information about algorithms, but it can be ambiguous and unclear. So, let's describe our algorithm in this way:

The algorithm reads the input, which represents the total number of elements from which an arithmetic mean will be calculated. If the entered number is equal to 0, the algorithm should return null. Otherwise, it should read the numbers in the amount equal to the expected total count. Finally, it should return the result as the sum of numbers divided by their count.

Quite simple and understandable, isn't it? You can use this notation for simple algorithms, but it can be useless for complex and advanced algorithms. Of course, some descriptions in the natural language are often useful, regardless of the complexity of an algorithm. They can give you a brief understanding of what the aim of the algorithm is, how it works, and what aspects should be taken into account while you're analyzing or implementing the algorithm.

Flowchart

Another way of presenting an algorithm is via a **flowchart**. A flowchart uses a set of graphical elements to prepare a diagram that specifies the algorithm's operation. Some of the available symbols are as follows:

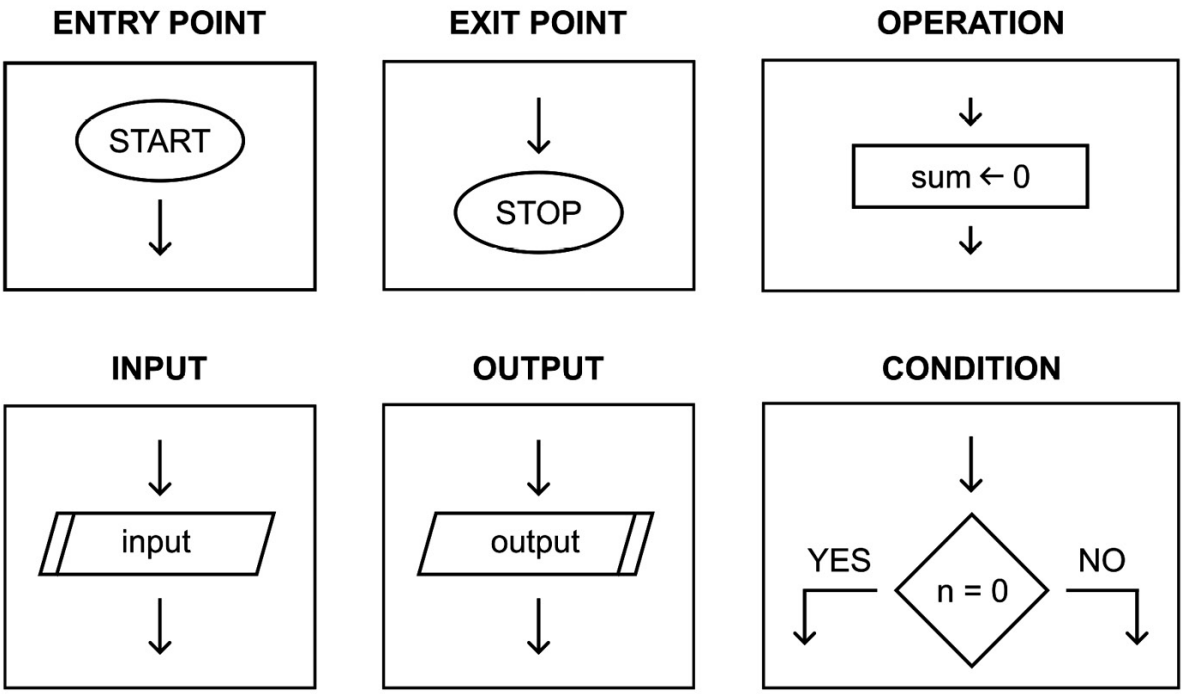

Figure 2.3 – The available symbols while designing a flowchart

The algorithm should contain the **entry point** and one or more **exit points**. It can also contain other blocks, including **operation**, **input**, **output**, or **condition**. The following blocks are connected with **arrows** that specify the order of execution. You can also draw **loops**.

Let's take a look at a flowchart for calculating the arithmetic mean:

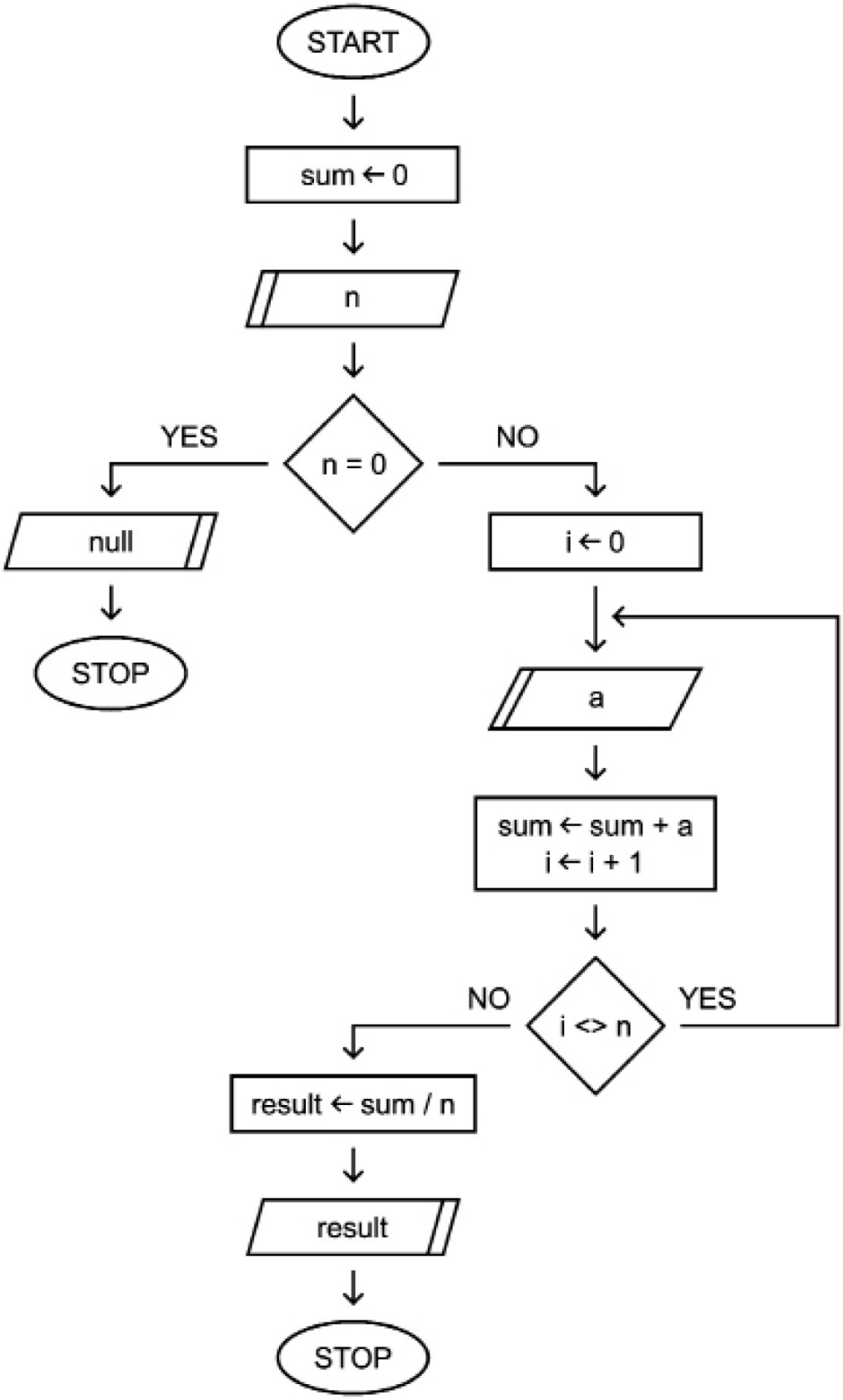
START
sum ← 0
n
n = 0
YES
NO
null
STOP
i ← 0
a
sum ← sum + a
i ← i + 1
i <> n
NO
YES
result ← sum / n
result
STOP

Figure 2.4 – Flowchart for calculating the arithmetic mean

The execution starts in the **START** block. Then, we assign **0** as a value of the **sum** variable, which stores the sum of all the entered numbers. Next, we read a value from the input and store it as a value of the **n** variable. This is the total number of elements used to calculate the arithmetic mean. Next, we check whether **n** is equal to **0**. If so, the **YES** branch is chosen, **null** is returned to the output, and the execution stops. If **n** is not equal to **0**, the **NO** branch is chosen and we assign **0** as a value of the **i** variable. It stores the number of elements already read from the input. Next, we read the number from the input and save it as a value of the **a** variable. The following operation block increases **sum** by the value of **a**, as well as increments the value of **i**.

The next block is a conditional one that checks whether **i** is not equal to **n**, which means that the required number of elements is not read from the input yet. If **i** is equal to **n**, the **NO** branch is chosen and a value of the **result** variable is set to a result of a division of **sum** by **n**. Then, the **result** variable is returned and the execution stops. An interesting construction is used when the conditional expression evaluates to **true**, which means that we need to read another input. Then, the loop is used and the execution comes back just before the input block for reading **a**. Thus, we can execute some operations multiple times, until the condition is met.

As you can see, a flowchart is a diagram that makes it possible to specify a way of algorithm operation in a more precise way than using natural language. It is an interesting option for simple algorithms, but it can be quite cumbersome in the case of advanced and complex ones, where it is impossible to present the whole operation within a diagram of a reasonably small size.

Pseudocode

The next notation we'll look at is **pseudocode**. It allows you to specify algorithms in another way, which is a bit similar to the code written in a programming language. Here, we use the English language to define inputs and outputs, as well as to present a set of instructions clearly and concisely, but without the syntax of any programming language.

Here's some example pseudocode for calculating the arithmetic mean:

```
INPUT:
n – total number of elements used for mean calculation.
a – the following numbers entered by a user.
OUTPUT:
result - arithmetic mean of the entered numbers.
INSTRUCTIONS:
sum <- 0
read n
if n = 0 then
   return null
```

```
endif
i <- 0
do
   read a
   sum <- sum + a
   i <- i + 1
while i <> n
result <- sum / n
return result
```

As you can see, the pseudocode provides us with a syntax that is easy to understand and follow, as well as quite close to a programming language. For this reason, it is a precise way of algorithm presentation and documentation that can be later used to transform it into a set of instructions in our chosen programming language.

Programming language

Now, let's look at the last form of algorithm notation: **programming language**. It is very precise, can be compiled and run. Thus, we can see the result of its operation and check it using a set of test cases. Of course, we can implement an algorithm in any programming language. However, in this book, you will see only examples in the C# language.

Let's take a look at the implementation of the mean calculation algorithm:

```
double sum = 0;
Console.Write("n = ");
int.TryParse(Console.ReadLine(), out int n);
if (n == 0) { Console.WriteLine("No result."); }
int i = 0;
do
{
    Console.Write("a = ");
    double.TryParse(Console.ReadLine(), out double a);
    sum += a;
    i++;
}
while (i != n);
double result = sum / n;
Console.WriteLine($"Result: {result:F2}");
```

The preceding code contains an **`if`** conditional statement and a **`do-while`** loop.

If we run the application, we need to enter the number of elements from which we would like to calculate the arithmetic mean. Then, we will be asked to enter the number **`n`** times. When the number of provided elements is equal to the expected value, the result is calculated and presented in the console, as follows:

```
n = 3
a = 1
a = 5
a = 10
Result: 5.33
```

That's all! Now, you know what algorithms are, where you can find them in your daily life, as well as how to represent algorithms using natural language, flowcharts, pseudocode, and programming languages. With this knowledge, let's proceed to learn about different types of algorithms, including recursive and heuristic algorithms.

Types of algorithms

As mentioned previously, algorithms are almost everywhere, and even intuitively, you use them each day while solving various tasks. There is an enormous amount of algorithms and a lot of their types, chosen according to different criteria. To simplify this topic, only a few types will be presented here, chosen from different classifications, to show you a variety and encourage you to learn more about them on your own. It is also quite common that the same algorithm is classified into a few groups.

Recursive algorithms

First, let's take a look at **recursive algorithms**, which are strictly connected with the idea of **recursion** and are the opposite of **iterative algorithms**. What does this mean? **An algorithm is recursive if it calls itself to solve smaller subproblems of the original problem**. The algorithm calls itself multiple times until the **base condition** is met.

This technique provides you with a powerful solution for solving problems, can limit the amount of code, and can be easy to understand and maintain. However, recursion has some drawbacks related to performance or the requirement for more space in the stack's memory, which could lead to stack overflow problems. Fortunately, you can prevent some of these issues using **dynamic programming**, an optimization technique that supports recursion.

Recursion can be used in several algorithms, including the following:

- Sorting an array with the **merge sort** and **quicksort** algorithms, which are implemented and presented in detail in *Chapter 3*, *Arrays and Sorting*
- Solving the **Towers of Hanoi** game, as depicted in *Chapter 5*, *Stacks and Queues*
- **Traversing a tree**, as described in *Chapter 7*, *Variants of Trees*
- Getting a number from the **Fibonacci series**, as shown in *Chapter 9*, *See in Action*
- **Generating fractals**, as shown in *Chapter 9*, *See in Action*
- Calculating a **factorial** (`n!`)
- Getting the **greatest common divisor of two numbers** using the Euclidean algorithm
- **Traversing the filesystem** with directories and subdirectories

Divide and conquer algorithms

Another group of algorithms is named **divide and conquer**. It is related to the **algorithmic paradigm of solving a problem by breaking it down into smaller subproblems (the "divide" step), calling them recursively until they are simple enough to be solved directly ("conquer"), and combining the results of subproblems to get the final result ("combine")**. This approach has many advantages, also taken from the pros of recursion, including ease of implementation, understanding, and maintenance. By dividing the problem into many subproblems, it supports **parallel computing**, which can lead to performance improvements. Unfortunately, this paradigm also has some disadvantages, including the necessity for a proper base case definition to terminate the execution of the algorithm. Performance issues, similar as in the case of recursive algorithms, can exist as well.

Divide and conquer is a popular approach for solving various algorithmic problems and you can see its implementations in a broad range of applications:

- Sorting an array with the **merge sort** and **quicksort** algorithms, which are implemented and presented in detail in *Chapter 3*, *Arrays and Sorting*
- **Finding the closest pair of points** located on the two-dimensional surface, which will be presented in *Chapter 9*, *See in Action*
- Calculating the **power of a number**
- Finding the **minimum and maximum values** in an array
- Calculating the **fast Fourier transform**
- **Multiplying large numbers** using Karatsuba's algorithm

Back-tracking algorithms

Next, we'll cover **back-tracking algorithms**. **They are used for solving problems that consist of a sequence of decisions, each depending on the decisions that have already been taken, incrementally building the solution. When you realize that the decisions that have been taken do not provide the correct solution, you backtrack**. Of course, you can support this approach with recursion to try various variants and therefore find a suitable solution, if one exists.

You can use this approach for many tasks, including the following:

- Solving the **rat in a maze** problem, as shown in *Chapter 9*, *See in Action*
- Solving **Sudoku**, as shown in *Chapter 9*, *See in Action*
- Solving **crosswords** by entering letters into empty spaces
- Solving the **eight queens** problem of placing eight queens on a chessboard and not allowing them to attack each other

- Solving the **knight's tour**, where you place a knight on the first block on a chessboard and move it so that it visits all blocks exactly once
- Generating **gray codes** to create bit patterns where the following ones differ by one bit only
- Solving the **m-coloring problem** for graph-related topics

Greedy algorithms

Now that we've covered the recursive, divide-and-conquer, and back-tracking algorithms, it's high time to present another type, namely greedy algorithms. **A greedy algorithm builds the solution piece by piece by choosing the best option in each step, not concerned about the overall solution, and being short-sighted in its operation**. For this reason, there is no guarantee that the final result is optimal. However, in many scenarios, using the local optimal solutions can lead to global optimal solutions or can be good enough.

Here are some examples:

- Finding the shortest path in a graph using **Dijkstra's algorithm**, as shown and explained in detail in *Chapter 8*, *Exploring Graphs*
- Calculating the minimum spanning tree in a graph with **Kruskal's algorithm** and **Prim's algorithm**, as shown in *Chapter 8*, *Exploring Graphs*
- Solving the **minimum coin change** problem, as explained in *Chapter 9*, *See in Action*
- The greedy approach to **Huffman coding** in **data compression algorithms**
- **Load balancing** and **network routing**

Heuristic algorithms

Now, it's time to add more "magic" to your algorithms via heuristics! **A heuristic algorithm calculates a near-optimal solution for an optimization problem and is especially useful for scenarios when the exact methods are not available or are too slow. Thus, you can see a significant speed boost, but with a decreased accuracy of the result.** Such an approach is popular and adequate for solving various real-world problems, often complex and big, and is applied in many different fields of science, even those regarding bioinformatics.

Heuristic algorithms have many applications and subtypes:

- **Genetic algorithms**, which are **adaptive heuristic search algorithms**, and can be used to guess the title of this book, as depicted in *Chapter 9*, *See in Action*
- Solving **vehicle routing problems** and the **traveling salesman problem** with the **Tabu Search algorithm**
- Solving the **Knapsack problem**, where you need to choose items of the maximum total value to be packed within the mass limit
- **Filtering and processing signals**

- **Detecting viruses**

Dynamic programming

Since we're talking about various types of algorithms, it is worth mentioning **dynamic programming**. It is a **technique that optimizes recursive algorithms by limiting the necessity of computing the same result multiple times**. This technique can be used in one of two approaches:

- The **top-down approach**, which uses **memoization** to save the results of subproblems. Therefore, the algorithm can use the value from the cache and does not need to recalculate the same results multiple times or does not need to call the method with the same parameters multiple times.
- The **bottom-up approach**, which uses **tabulation** to replace recursion with iteration. It limits the number of function calls and problems regarding stack overflow.

Both of these approaches can significantly decrease the time complexity and increase performance, and therefore speed up your algorithm. Every time you use recursion, it is a good idea to try to optimize it using dynamic programming. If you want to learn how to optimize calculating a number from the **Fibonacci series**, go to *Chapter 9*, *See in Action*.

You can also use dynamic programming to find the **shortest path between all pairs of vertices in a graph** by using the **Floyd-Warshall algorithm**, as well as in **Dijkstra's algorithm**. Another application is for solving the **Tower of Hanoi** mathematical game. Possibilities are even broader and you can also apply it to **artificial neural networks**.

Brute-force algorithms

While we're presenting various types of algorithms, we should also consider brute-force algorithms. **A brute-force algorithm is a general solution for solving a problem by checking all possible options and choosing the best one**. It is an approach that can have huge time complexity and its operation can take a lot of time, so it can be useless in real-world scenarios. However, a brute-force algorithm is often the first choice when you need to solve some algorithmic problem. There's nothing bad in doing this as you can learn more about the domain of the problem you wish to solve and see some results for simpler cases. Nevertheless, while developing an algorithm, it is a good idea to enhance it significantly by using other paradigms.

Here are some examples of where you can use brute-force algorithms:

- **Guessing a password**, where you check each possible password one after the other, as presented in *Chapter 9*, *See in Action*
- **Finding the minimum value in an unsorted array**, where you need to iterate through all items as there is no relationship between values in the array

- **Finding the best possible plan for a day**, placing various tasks between meetings, and trying to organize it in a way that you can start working late and ending early
- Solving the **traveling salesman** problem

After introducing a few types of algorithms, you can see that some of them provide you with a faster solution while others can have huge time complexity. But what does this mean? You will learn about computational complexity, especially time complexity, in the next section.

Computational complexity

In this final section, let's take a look at the **computational complexity** of algorithms, focusing on both **time complexity** and **space complexity**. Why is this so important? Because it can decide whether your algorithm can be used in real-world scenarios. As an example, which of the following do you prefer?

- *(A) Absolutely the best route directions to work, but you receive them after an hour, when you are already at work.*
- *(B) Good enough route directions to work, but you receive them within a few seconds, a moment after you enter your car.*

I am sure that you chose *B* – me too!

Time complexity

First, let's focus on **time complexity**, which indicates **the amount of time necessary to run an algorithm as a function of the input length**, namely *n*. You can specify it using **asymptotic analysis**. This includes **Big-O notation**, which is used to indicate **how much time the algorithm will take with the increasing size of the input**.

For example, if you search for the minimum value in an unsorted array of size *n*, you need to visit all elements so that the maximum number of operations is equal to *n*, which is written as $O(n)$. If the algorithm iterates through each item in a two-dimensional array of size $n \times n$, the time complexity is $O(n*n)$, so it is $O(n^2)$.

There are various time complexities, including the ones presented here:

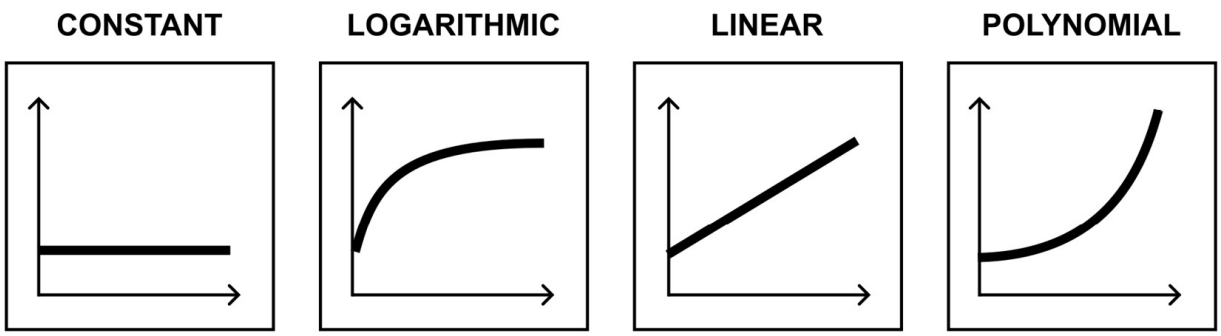

Figure 2.5 – Illustration of time complexities

The first is **O(1)** and is named the **constant time**. It indicates an algorithm whose execution time does not depend on the input size. The exemplary operations consistent with the *O(1)* constraint are getting an *i*-th element from an array, checking whether a hash set contains a given value, or checking whether a number is even or odd.

The next time complexity shown here is **O(log n)**, which is named the **logarithmic time**. In this case, the execution time is not constant, but it increases slower than in the linear approach. A well-known example of the *O(log n)* constraint is the problem of finding an item in a sorted array with binary search.

The third case is **O(n)** and is named the **linear time**. Here, the execution time increases linearly with the input length. You can take an algorithm for finding the minimum or maximum value in an unordered list or simply finding a given value in an unordered list as examples of the *O(n)* constraint.

The last time complexity shown here is the **polynomial time**, which is $\mathbf{O(n^m)}$, so it can be $O(n^2)$ (**quadratic time**), $O(n^3)$ (**cubic time**), and so on. In this case, the execution time increases much faster than in the case of the linear constraint. It can involve solutions that use nested loops. Examples include the bubble sort, insertion sort, and selection sort algorithms. We'll cover these in *Chapter 3*, *Arrays and Sorting*.

Of course, there are even more time complexities available, among which you will find **double logarithmic time**, **polylogarithmic time**, **fractional power time**, **linearithmic time**, **exponential time**, and **factorial time**.

Space complexity

Similar to time complexity, you can specify the **space complexity** using asymptotic analysis and the Big-O notation. Space complexity indicates **how much memory is necessary to run the algorithm with the increasing length of input**. You can use similar indicators, such as *O(1)*, *O(n)*, or $O(n^2)$.

WHERE CAN YOU FIND MORE INFORMATION?

In this chapter, only a very brief introduction to the subject of algorithms was presented. I strongly encourage you to try to broaden your knowledge regarding algorithms on your own. It is an extremely interesting and challenging topic. For example, you can learn more about various types of algorithms at https://www.techtarget.com/whatis/definition/algorithm and at https://www.geeksforgeeks.org/most-important-type-of-algorithms/, while about the computational complexity at https://en.wikipedia.org/wiki/Computational_complexity. I am keeping my fingers crossed for you success with algorithms!

Summary

You've just completed the second chapter of this book, which was all about data structures and algorithms in the C# language. This time, we focused on algorithms and indicated their crucial role in the development of various applications, regardless of their types.

First, you learned **what an algorithm is** and **where you can find algorithms** in your daily life. As you saw, algorithms are almost everywhere and you use and design them without even knowing it.

Then, you learned about **notations for algorithm representation**. There, you learned how to specify algorithms in a few ways, namely in a natural language, using a flowchart, via pseudocode, or directly in a programming language.

Next, you learned **about a few different types of algorithms**, starting with the recursive algorithms that call themselves to solve smaller subproblems. Then, you learned about **divide and conquer** algorithms, which divide the problem into three stages, namely divide, conquer, and combine. Next, you learned about **back-tracking** algorithms, which allow you to solve problems consisting of a sequence of decisions, each depending on a decision that's already been taken, together with the backtrack option if the decisions do not provide a correct solution. Then, you learned about **greedy** algorithms, which choose the best option in each step of their operation while not being concerned about the overall solution. Another group you learned about was **heuristic** algorithms for finding near-optimal solutions. Then, you learned that you can optimize recursive algorithms using **dynamic programming** and its top-down and bottom-up approaches. Finally, you learned about **brute-force** algorithms.

The final part of this chapter looked at **computational complexity** in terms of time and space complexity. Asymptotic analysis, together with Big-O notation, was presented.

In the next chapter, we'll cover **arrays** and various **sorting algorithms**. Are you ready to continue your adventure with data structures and algorithms in the C# language? If so, let's go!

3

Arrays and Sorting

As a developer, you have certainly stored various collections within your applications, such as data of users, books, and logs. One of the natural ways of storing such data is by using arrays. However, have you ever thought about their variants? For example, have you heard about jagged arrays? In this chapter, you will see arrays in action, together with examples and detailed descriptions.

You can use an array to **store many items of the same type**, such as `int`, `string`, as well as a user-defined class or record. Just keep in mind that **the number of elements in an array cannot be changed after initialization**. For this reason, you will not be able to easily add a new item at the end of the array or insert an element in a given position within the array while moving the remaining items one position further. If you need such features, you can use another data structure, namely a list and its variants, which will be described in the following chapter.

While developing applications in the C# language, you can benefit from a few variants of arrays, namely **single-dimensional arrays**, **multi-dimensional arrays**, and **jagged arrays**. In this chapter, you will also get to know seven **sorting algorithms**, namely **selection sort**, **insertion sort**, **bubble sort**, **merge sort**, **Shell sort**, **quicksort**, and **heap sort**. For each, you will see an illustration-based example, the implementation code, and a step-by-step explanation. You will also see their performance analysis, presented with charts.

We will cover the following topics in this chapter:

- Single-dimensional arrays
- Multi-dimensional arrays
- Jagged arrays
- Sorting algorithms

Single-dimensional arrays

Let's start with the simplest variant of arrays, namely single-dimensional ones. **Such an array stores a collection of items of the same type, which are accessible by an index**. It is important to remember that **indices of elements within arrays in C# are zero-based**. This means that the first element has an index equal to `0`, while the last one has an index equal to the length of the array minus one.

IMAGINE A SINGLE-DIMENSIONAL ARRAY

If you want to better imagine a single-dimensional array, take your eyes off this book for a moment and look at the chest of drawers or wardrobe in your room. A standard chest of drawers consists of several drawers and a single-dimensional array looks similar. It also has several elements (as drawers), which are accessible via the index. You cannot change the size of the array in the same way as you cannot change the number of drawers since the furniture is prepared. An array has one significant advantage over a chest of drawers, namely all its "drawers" are always working as expected.

An example of a single-dimensional array is shown in the following figure:

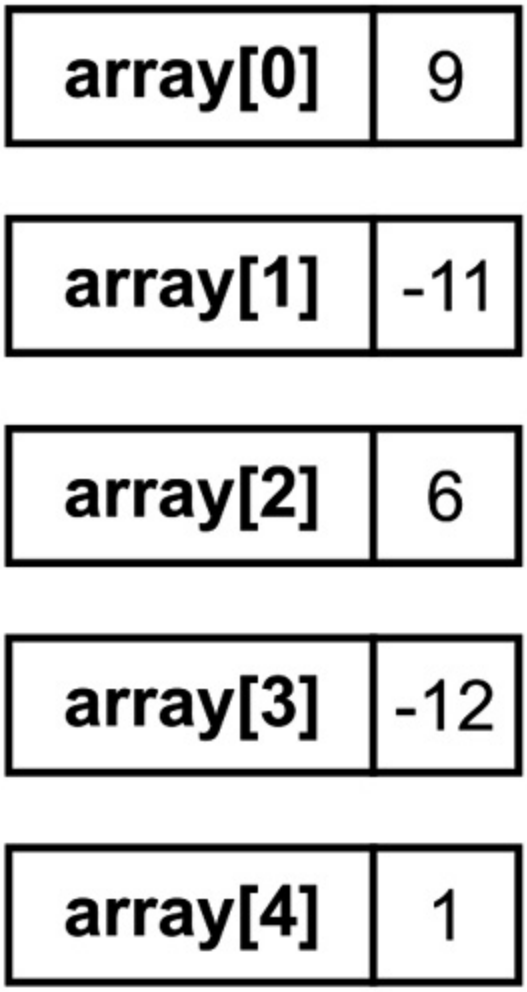

Figure 3.1 – Example of a single-dimensional array

It contains five elements with the following values: `9`, `-11`, `6`, `-12`, and `1`. The first element has an index equal to `0`, while the last one has an index equal to `4`.

To use a single-dimensional array, you need to declare and initialize it. The declaration is very simple because you just need to specify a type of element and a name, as follows:

```
type[] name;
```

The declaration of an array with integer values is shown in the following line:

```
int[] numbers;
```

So far, you know how to declare an array, but what about initializing one? To create an array with five elements and initialize them to default values, you can use the **`new`** operator, as shown here:

```
numbers = new int[5];
```

Of course, you can combine a declaration and initialization in the same line, as follows:

```
int[] numbers = new int[5];
```

Unfortunately, all the elements currently have default values – that is, zeros in the case of integer values. Thus, you need to set the values of particular elements. You can do this using the **`[]`** operator and an index of an element, as shown in the following code:

```
numbers[0] = 9;
numbers[1] = -11;
numbers[2] = 6;
numbers[3] = -12;
numbers[4] = 1;
```

Moreover, you can combine a declaration and initialization of array elements to specific values using one of the following variants:

```
int[] numbers = new int[] { 9, -11, 6, -12, 1 };
int[] numbers = { 9, -11, 6, -12, 1 };
```

Another approach involves using the **collection expression**, as follows:

```
int[] numbers = [9, -11, 6, -12, 1];
```

When you have the proper values of elements within an array, you can get values using the **`[]`** operator and by specifying the index, as shown in the following line of code:

```
int middle = numbers[2];
```

Here, you get a value of the third element (the index equal to **`2`**) from the **`numbers`** array and store it as a value of the **`middle`** variable.

The array has some properties that can be useful while developing applications. For example, the **`Length`** property makes it possible to get the size of the array, namely the number of elements stored within it. If you want to access the last item in the array, regardless of its size, you can use the following line of code:

```
int last = numbers[numbers.Length - 1];
```

You can simplify this with the **index operator**, as follows:

```
int last = numbers[^1];
```

It is worth noting that the second item from the end can be received by `[^2]`, the third by `[^3]`, and so on.

The other property is named **`Rank`** and returns the number of dimensions of the array. Usage of this property is shown in the following line of code:

```
int rank = numbers.Rank;
```

Apart from the already mentioned properties, you can use a set of static methods of the **`Array`** class, such as **`Exists`**, to check whether there is any element in the array that matches the given predicate. For example, you can easily verify whether the array contains any element whose value is greater than zero, as follows:

```
bool anyPositive = Array.Exists(numbers, e => e > 0);
```

Among other methods, you can use **`TrueForAll`** to check whether all elements meet the provided predicate, such as to ensure that there are no zeros in the array:

```
bool noZeros = Array.TrueForAll(numbers, e => e != 0);
```

You can also get the first element that matches the predicate. As an example, let's get the value of the first number that is smaller than zero using the **`Find`** method:

```
int firstNegative = Array.Find(numbers, e => e < 0);
```

If you want to get a new array with all the elements that meet the given condition, you can use the **`FindAll`** method. The following code shows how to get all negative numbers:

```
int[] negatives = Array.FindAll(numbers, e => e < 0);
```

When you do not want to specify the predicate and you just want to check whether the array contains a given element or not, you can use the **`IndexOf`** method, which returns an index of the first found occurrence of the value or **`-1`**, if not found:

```
int index = Array.IndexOf(numbers, -12);
```

Another interesting static method is **`ForEach`**. It allows you to perform some operations for all the elements in the array. As an example, you can use it to write the absolute value of each array element in the console, as shown in the following code:

```
Array.ForEach(numbers,
    e => Console.WriteLine(Math.Abs(e)));
```

As you can see, even for as simple a data structure as a single-dimensional array, you have a lot of useful built-in features. Let's continue learning them and take a look at the two next methods, namely

`Reverse` and **`Sort`**. According to their names, the first allows you to reverse the order of the elements, either for the whole array or only within some range. This is presented in the following line of code, which reverses the first three elements:

```
Array.Reverse(numbers, 0, 3);
```

The **`Sort`** method has even more variants. In its simplest form, it sorts the whole array. After running the following line, you'll get the array with the elements sorted from the smallest to the biggest:

```
Array.Sort(numbers);
```

If you want to fill the whole array with the same value or fill the range of elements with the same value, you can use a **`for`** loop and simply iterate through suitable indices and assign a given value. However, you can use the **`Fill`** method instead. The following line places **`3`** as a value of all elements in the array:

```
Array.Fill(numbers, 3);
```

Another method we can use is **`Clear`**, which makes it possible to clear the whole array or a range of its elements. For example, you can fill the whole array with the default value of the integer type, namely zeros, using the following line of code:

```
Array.Clear(numbers);
```

Now, let's take a look at **`Copy`**, which copies a range of elements from the source array to the destination array. You can use one out of a few variants, such as to specify indices from both arrays. As an example, let's copy **`3`** elements (specified as **`length`**) from the **`numbers`** array (as the source array), starting from the first element (source index set to **`0`**), and place them in the **`subarray`** array, starting from the first element (destination index set to **`0`**):

```
int[] subarray = new int[3];
Array.Copy(numbers, 0, subarray, 0, 3);
```

We've already covered a few available properties and methods, but it is worth mentioning some **extension methods** as well, such as **`Contains`** and **`Max`**.

HAVE YOU EVER HEARD ABOUT EXTENSION METHODS?

*If not, think of them as methods that are "added" to a particular existing type (both built-in or user-defined) and can be called in the same way as when they are defined directly as instance methods. The declaration of an extension method requires you to specify it within a static class as a static method with the first parameter indicating the type to which you want to "add" this method with the **`this`** keyword.*

You can use the **`Contains`** extension method to check whether the array contains an element passed as the parameter. As an example, let's learn how to ensure that the **`numbers`** array contains **`6`** as one of its elements:

```
bool contains = numbers.Contains(6);
```

The **Contains** method is not the only available extension method. Among others, you can find **All** and **Any**. The first (**All**) checks whether all of the elements match the given predicate, while the other (**Any**) verifies whether at least one element meets the condition. You can use them to ensure that there are no zeros in the array and check whether there is at least one positive element, as shown here:

```
bool noZeros = numbers.All(n => n != 0);
bool anyPositive = numbers.Any(n => n > 0);
```

What will you do to find the minimum or maximum value in the array? You will probably iterate through all elements and check whether each element is smaller (for searching the minimum) or bigger (in the case of the maximum) than the already found minimum or maximum value. If that is your solution, you are correct and it is a good way of solving the problem. However, you can make your code much shorter by using the **Min** and **Max** extension methods, as shown here:

```
int min = numbers.Min();
int max = numbers.Max();
```

Pretty simple, isn't it? Let's also take a look at the **Average** and **Sum** methods, which easily calculate the average value of all of the elements, as well as their sum:

```
double avg = numbers.Average();
int sum = numbers.Sum();
```

After this short introduction to single-dimensional arrays, it's high time to take a look at an example of how to apply such arrays in real-world scenarios.

WHERE CAN YOU FIND MORE INFORMATION?

You can find a lot of interesting information about arrays and their various variants in the context of the C# language at https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/.

Example – month names

To summarize what you've learned about single-dimensional arrays, let's use an array to store names of months, written in English. Such names should be obtained automatically, not by hardcoding them in the code.

The implementation is shown here:

```
using System.Globalization;
CultureInfo culture = new("en");
string[] months = new string[12];
for (int month = 1; month <= 12; month++)
{
    DateTime firstDay = new(DateTime.Now.Year, month, 1);
```

```
        string name = firstDay.ToString("MMMM", culture);
        months[month - 1] = name;
    }
    foreach (string m in months)
    {
        Console.WriteLine(m);
    }
```

First, you create a new instance of the `CultureInfo` class (from the `System.Globalization` namespace), passing `en` as a parameter, to later get the names of months in English. Then, you declare a new single-dimensional array and initialize it with default values. It contains `12` elements to store the names of all the months in a year. Then, the `for` loop is used to iterate through the numbers of all months – that is, from `1` to `12`. For each of them, a `DateTime` instance representing the first day in a particular month from the current year is created.

The name of the month is obtained by calling the `ToString` method on the `DateTime` instance, passing the proper format of the date (`MMMM`), as well as specifying the culture. Then, the name is stored in the array using the `[]` operator and an index of the element. It is worth noting that the index is equal to the current value of the `month` variable minus one. Such subtraction is necessary because the first element in the array has an index equal to zero, instead of one.

The next interesting part of the code is the `foreach` loop, which iterates through all elements of the array. For each of them, the name of the month is shown in the console:

```
January
February (...)
December
```

As mentioned earlier, single-dimensional arrays are not the only available variant. You will learn more about multi-dimensional arrays in the following section.

Multi-dimensional arrays

The arrays in the C# language do not need to have only one dimension. It is possible to create two-dimensional arrays as well. As you will see, multi-dimensional arrays are very useful and are frequently used while developing various applications.

IMAGINE A TWO-DIMENSIONAL ARRAY

If you want to imagine a two-dimensional array, take a break, close your eyes, and play Sudoku. If you don't know what this is, Sudoku is a popular game that requires you to fill empty cells of a 9x9 board with numbers from 1 to 9. However, each row, each column, and each 3x3 box can only contain unique numbers. Surprise – this board forms a two-dimensional array! You can point to any place on the board by specifying its row and column, the same as in the case of a two-dimensional array. And if you are a bit tired of solving such puzzles with a pencil and a piece of paper, take a look at Chapter 9, See in Action, where you will learn how to create an algorithm for solving a Sudoku puzzle!

An example two-dimensional array that stores integer values is shown here:

array[0,0]	9	array[0,1]	5	array[0,2]	-9
array[1,0]	-11	array[1,1]	4	array[1,2]	0
array[2,0]	6	array[2,1]	115	array[2,2]	3
array[3,0]	-12	array[3,1]	-9	array[3,2]	71
array[4,0]	1	array[4,1]	-6	array[4,2]	-1

Figure 3.2 – Example of a two-dimensional array

First, you need to declare and initialize a two-dimensional array with `5` rows and `3` columns, as shown in the following line of code:

```
int[,] numbers = new int[5, 3];
numbers[0, 0] = 9; (...)
```

You can combine a declaration with an initialization in a bit different way as well:

```
int[,] numbers = new int[,]
{
    { 9, 5, -9 },
    { -11, 4, 0 },
    { 6, 115, 3 },
    { -12, -9, 71 },
    { 1, -6, -1 }
};
```

A small explanation is necessary for the way you access particular elements from a two-dimensional array. Let's take a look at the following example:

```
int number = numbers[2, 1];
numbers[1, 0] = 11;
```

In the first line of code, the value from the third row (index equal to `2`) and second column (index equal to `1`) is obtained (that is, `115`) and set as a value of the **`number`** variable. The other line replaces `-11` with `11` in the second row and the first column.

Now that you've learned about one-dimensional and two-dimensional arrays, let's proceed to three-dimensional ones. Do you know how to understand this structure?

IMAGINE A THREE-DIMENSIONAL ARRAY

If you want to better imagine a three-dimensional array, launch a game in which you can create buildings from blocks. You place each of them in a specified location on the board, in X and Y coordinates. However, you can also build the next building floors, so you can specify the block's Z coordinate as well. In such circumstances, you operate in a three-dimensional world with three-dimensional arrays!

An example three-dimensional array is presented in the following figure:

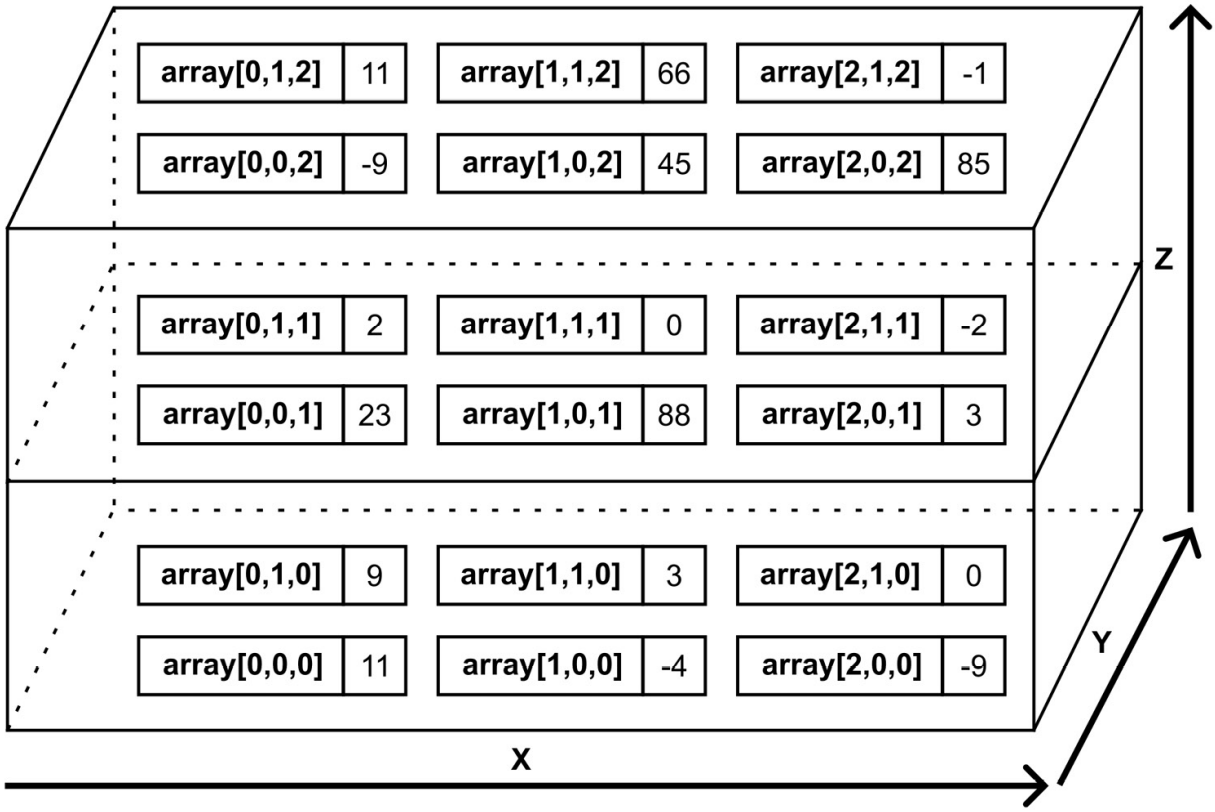

Figure 3.3 – Example of a three-dimensional array

If you want to create a three-dimensional array, you can use the following code:

```
int[,,] numbers = new int[3, 2, 3];
```

The remaining operations can be performed similarly as in the case of arrays with a different number of dimensions. Of course, you need to specify three indices while accessing a particular element of the array.

So far, you've learned about one-, two-, and three-dimensional arrays. But is it possible to use four-dimensional arrays? Of course!

IMAGINE A FOUR-DIMENSIONAL ARRAY

Imagining a four-dimensional array is not very easy, but let's try to do so! Once again, think about the three-dimensional game board we mentioned previously, but with content that changes depending on your level in the game. In this way, you can access a particular block in the three-dimensional world using X, Y, and Z coordinates. To get a target value, you need to use another dimension, namely by providing your current level. In this way, you will get different results depending on the fourth dimension. Not so difficult, right?

You can declare such an array using the following line of code:

```
int[,,,] numbers = new int[5, 4, 3, 2];
```

If you need more dimensions, you can apply them. However, please keep in mind that using more dimensions can be quite difficult to understand and your code can be more difficult to follow and maintain in the future.

With this introduction to the topic of multi-dimensional arrays out of the way, let's proceed to some examples. They will show you how to use such data structures in the real world.

Example – multiplication table

This first example shows basic operations being performed on a two-dimensional array to present a multiplication table. It stores the results of the multiplication of all integer values in the range from `1` to `10` in the array and present them in the console:

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

Let's take a look at the declaration and initialization of the array:

```
int[,] results = new int[10, 10];
```

Here, a two-dimensional array with `10` rows and `10` columns is created and its elements are initialized to default values – that is, to zeros. When the array is ready, you fill it with the results of the multiplication, as well as present the result in the console. Such a task can be performed using two **`for`** loops, as shown here:

```
for (int i = 0; i < results.GetLength(0); i++)
{
    for (int j = 0; j < results.GetLength(1); j++)
    {
        results[i, j] = (i + 1) * (j + 1);
        Console.Write($"{results[i, j],4}");
    }
```

```
    Console.WriteLine();
}
```

In the preceding code, you can see the **`GetLength`** method, which is called on the **`results`** array. This method returns the number of elements in a particular dimension – that is, the first (when passing `0` as the parameter) and the second (`1` as the parameter). In both cases, a value of `10` is returned, according to the values specified during the array's initialization. Another important part of the code is the way of setting the value of an element. To do so, you must provide two indices.

The multiplication results, after converting them into **`string`** values, have different lengths, from one character (as in the case of **`4`** as a result of `2*2`) to three (`100` from `10*10`). To improve their presentation, you need to write each result in **`4`** characters. Therefore, if an integer value takes less space, leading spaces should be added. As an example, `1` will be shown with three leading spaces (`___1`, where _ is a space), while `100` will be shown with only one space (`_100`). You can achieve this goal by using a proper composite format string (namely, **`,4`**) within the interpolated string.

Example – game map

Another example is a program that presents a map of a game. This map is a rectangle with 6 rows and 8 columns. Each element of the array specifies a type of terrain as grass, sand, water, or brick (also referred to as wall). Each place on the map should be shown in a particular color (such as green for grass), as well as using a custom character that depicts the terrain type (such as **≈** for water), as shown in the following figure:

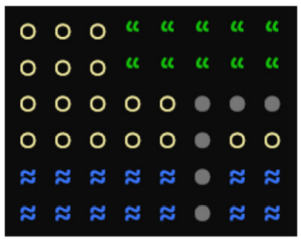

Figure 3.4 – Screenshot of the game map example

Let's start by creating two auxiliary methods that make it possible to get a particular color and character depending on the terrain's type (**`GetColor`** and **`GetChar`**, respectively). The code for these methods is as follows:

```
ConsoleColor GetColor(char terrain)
{
    return terrain switch
    {
        'g' => ConsoleColor.Green,
```

```
        's' => ConsoleColor.Yellow,
        'w' => ConsoleColor.Blue,
        _ => ConsoleColor.DarkGray
    };
}
char GetChar(char terrain)
{
    return terrain switch
    {
        'g' => '\u201c',
        's' => '\u25cb',
        'w' => '\u2248',
        _ => '\u25cf'
    };
}
```

As you can see, the code of the **`GetColor`** method is self-explanatory. However, the **`GetChar`** method returns a proper Unicode character depending on the character's value (**`g`**, **`s`**, **`w`**, or **`b`**). For example, in the case of water, the **`'\u2248'`** value is returned, which is a representation of the **≈** character.

Let's take a look at the remaining part of the code. Here, you configure the map, as well as present it in the console. The code is as follows:

```
using System.Text;
char[,] map =
{
    { 's', 's', 's', 'g', 'g', 'g', 'g', 'g' },
    { 's', 's', 's', 'g', 'g', 'g', 'g', 'g' },
    { 's', 's', 's', 's', 's', 'b', 'b', 'b' },
    { 's', 's', 's', 's', 's', 'b', 's', 's' },
    { 'w', 'w', 'w', 'w', 'w', 'b', 'w', 'w' },
    { 'w', 'w', 'w', 'w', 'w', 'b', 'w', 'w' }
};
Console.OutputEncoding = Encoding.UTF8;
for (int r = 0; r < map.GetLength(0); r++)
{
    for (int c = 0; c < map.GetLength(1); c++)
    {
        Console.ForegroundColor = GetColor(map[r, c]);
        Console.Write(GetChar(map[r, c]) + " ");
    }
    Console.WriteLine();
}
Console.ResetColor();
```

This code should not require additional comments or explanations. Just keep in mind that to use Unicode values in the console output, don't forget to choose the UTF-8 encoding by setting the **`Encoding.UTF8`** value for the **`OutputEncoding`** property. You can set the foreground color for the console using the **`ForegroundColor`** property. If you want to reset such a color to the default one, just call the **`ResetColor`** method, as presented in the last line.

So far, you've learned about both single- and multi-dimensional arrays, but one more variant remains to be presented in this book, namely jagged arrays. Let's continue reading to learn more about them.

Jagged arrays

The last variant of arrays to be described in this book is **jagged arrays**, also referred to as an **array of arrays**. It sounds complicated, but fortunately, it is very simple. A jagged array can be understood as **a single-dimensional array, where each element is another array**. Of course, such inner arrays can have different lengths or they can even be not initialized.

IMAGINE A JAGGED ARRAY

If you want to better imagine a jagged array, stop reading this book for a moment, open your calendar, and switch its view so that it presents the whole year. It contains 365 or 366 boxes, depending on the year. For each day, you have a different number of meetings. On some days, you have three meetings, while on others, only one or even zero. Your holidays are marked in the calendar and blocked for meetings. You can easily imagine an application of a jagged array in this case. Each day box is an element of this array and it contains an array with data of meetings organized on a particular day. If this day is during your holidays, a related item is not initialized. This makes a jagged array much easier to visualize.

An example jagged array is presented in the following figure:

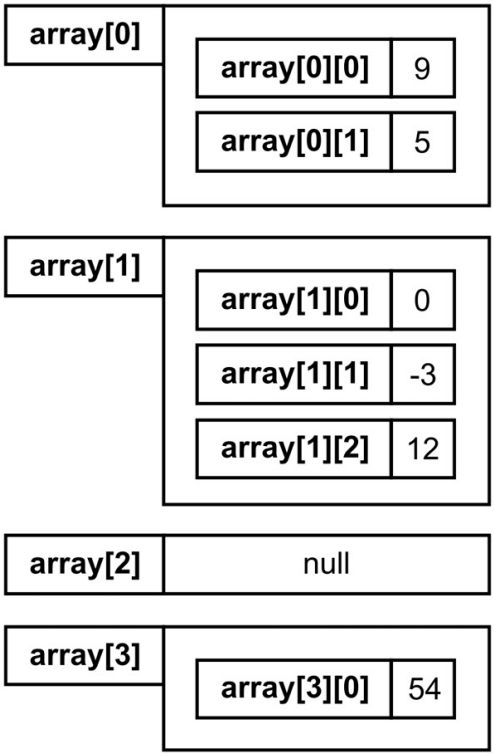

Figure 3.5 – Example of a jagged array

This jagged array contains four elements. The first has an array with two elements (`9` and `5`). The second element has an array with three elements (`0`, `-3`, and `12`). The third is not initialized (`null`), while the last one is an array with only one element (`54`).

Before proceeding to the example, it is worth mentioning the way of declaring and initializing a jagged array since it is a bit different from the arrays we've already described. Let's take a look at the following code snippet:

```
int[][] numbers = new int[4][];
numbers[0] = new int[] { 9, 5 };
numbers[1] = new int[] { 0, -3, 12 };
numbers[3] = new int[] { 54 };
```

This code can be simplified with a collection expression, as follows:

```
int[][] numbers = new int[4][];
numbers[0] = [9, 5];
numbers[1] = [0, -3, 12];
numbers[3] = [54];
```

In the first line, we declare a single-dimensional array with four elements. Each element is another single-dimensional array of integer values. When the first line is executed, the **`numbers`** array is initialized with default values, namely **`null`**. For this reason, we need to manually initialize particular elements, as shown in the following three lines of code. It is worth noting that the third element is not initialized.

You can also write the preceding code in a different way, as shown here:

```
int[][] numbers =
{
    new int[] { 9, 5 },
    new int[] { 0, -3, 12 },
    null!,
    new int[] { 54 }
};
```

That's not all – an even shorter variant is available:

```
int[][] numbers =
[
    [9, 5],
    [0, -3, 12],
    null!,
    [54]
];
```

How can you access a particular element from a jagged array? Let's see:

```
int number = numbers[1][2];
numbers[1][1] = 50;
```

The first line of code sets the value of the **`number`** variable to **`12`** – that is, to the value of the third element (index equal to **`2`**) from the array, which is the second element of the jagged array. The other line changes the value of the second element within the array, which is the second element of the jagged array, from **`-3`** to **`50`**.

Now that we've introduced jagged arrays, let's look at an example.

Example – yearly transport plan

In this example, you'll learn how to develop a program that creates a plan for your transportation for the whole year. For each day of each month, the application draws one of the available means of transport, such as by car, by bus, by subway, by bike, or simply on foot. In the end, the program presents the generated plan, as shown in the following screenshot:

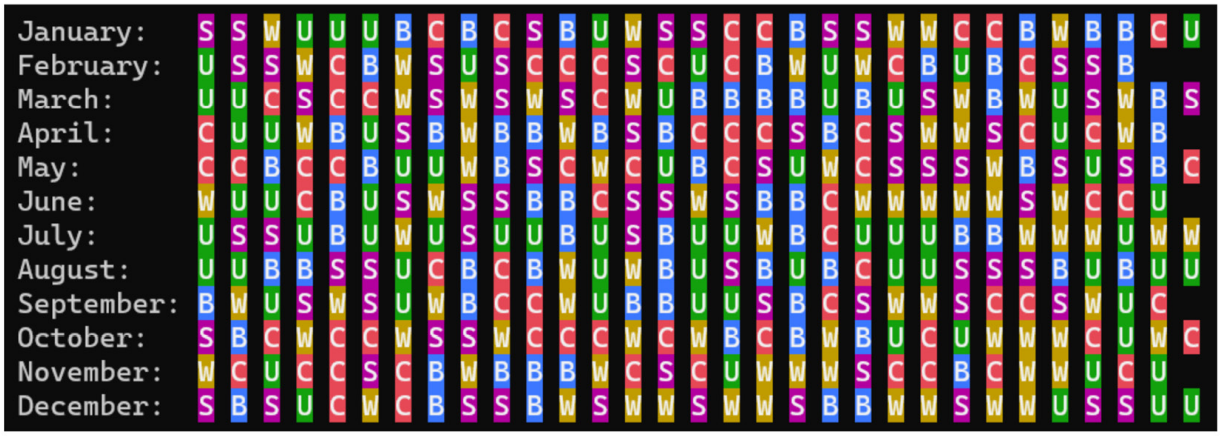

Figure 3.6 – Screenshot of the yearly transport plan example

First, let's declare the enumeration type with constants representing types of transport:

```
public enum MeanEnum { Car, Bus, Subway, Bike, Walk }
```

The next part of the code is as follows:

```
Random random = new();
int meansCount = Enum.GetNames<MeanEnum>().Length;
int year = DateTime.Now.Year;
MeanEnum[][] means = new MeanEnum[12][];
for (int m = 1; m <= 12; m++)
{
    int daysCount = DateTime.DaysInMonth(year, m);
    means[m - 1] = new MeanEnum[daysCount];
    for (int d = 1; d <= daysCount; d++)
    {
        int mean = random.Next(meansCount);
        means[m - 1][d - 1] = (MeanEnum)mean;
    }
}
```

First, a new instance of the `Random` class is created. This will be used to draw a suitable means of transport from the available ones. In the next line, we get the number of available transport types.

Then, the jagged array is created. It is assumed that it has `12` elements, representing all months in the current year.

Next, a `for` loop is used to iterate through all the months within the year. In each iteration, the number of days is obtained using the `DaysInMonth` static method of `DateTime`. Each element of the jagged array is a single-dimensional array with `MeanEnum` values. The length of such an inner array depends on the number of days in a month. For instance, it is set to `31` elements for January and `30` elements for April.

The next `for` loop iterates through all the days of the month. Within this loop, you draw a transport type and set it as a value of a suitable element within an array that is an element of the jagged array.

The next part of the code is related to presenting the plan in the console:

```
string[] months = GetMonthNames();
int nameLength = months.Max(n => n.Length) + 2;
for (int m = 1; m <= 12; m++)
{
    string month = months[m - 1];
    Console.Write($"{month}:".PadRight(nameLength));
    for (int d = 1; d <= means[m - 1].Length; d++)
    {
        MeanEnum mean = means[m - 1][d - 1];
        (char character, ConsoleColor color) = Get(mean);
        Console.ForegroundColor = ConsoleColor.White;
        Console.BackgroundColor = color;
        Console.Write(character);
        Console.ResetColor();
        Console.Write(" ");
    }
    Console.WriteLine();
}
```

First, a single-dimensional array with month names is created using the `GetMonthNames` method, which will be presented and described later. Then, a value of the `nameLength` variable is set to the maximum necessary length of text for storing the month name. To do so, the `Max` extension method is used to find the maximum length of text from the collection with names of months. The obtained result is increased by `2` to reserve space for a colon and a space.

A `for` loop is used to iterate through all the elements of the jagged array – that is, through all months. In each iteration, the month's name is presented in the console. The next `for` loop is used to iterate through all the items of the current element of the jagged array – that is, through all the days of the month. For each day, proper colors are set (for the foreground and background), and a suitable character is shown. Both a color and a character are returned by the `Get` method, taking the `MeanEnum` value as a parameter. This method will be shown a bit later.

Now, let's take a look at the implementation of the `GetMonthNames` method:

```
string[] GetMonthNames()
{
    CultureInfo culture = new("en");
    string[] names = new string[12];
    foreach (int m in Enumerable.Range(1, 12))
    {
        DateTime firstDay = new(DateTime.Now.Year, m, 1);
        string name = firstDay.ToString("MMMM", culture);
        names[m - 1] = name;
    }
    return names;
}
```

This code is self-explanatory, but let's focus on the line where we call the **`Range`** method. It returns a collection of integer values from **`1`** to **`12`**. Therefore, we can use it together with the **`foreach`** loop, instead of a simple **`for`** loop iterating from **`1`** to **`12`**. Just think about it as an alternative way of solving the same problem.

Finally, it is worth mentioning the **`Get`** method. It allows us to use one method instead of two, namely returning a character and a color for a given transport type. By returning data as a value tuple, the code is shorter and simpler, as shown here:

```
(char Char, ConsoleColor Color) Get(MeanEnum mean)
{
    return mean switch
    {
        MeanEnum.Bike => ('B', ConsoleColor.Blue),
        MeanEnum.Bus => ('U', ConsoleColor.DarkGreen),
        MeanEnum.Car => ('C', ConsoleColor.Red),
        MeanEnum.Subway => ('S', ConsoleColor.Magenta),
        MeanEnum.Walk => ('W', ConsoleColor.DarkYellow),
        _ => throw new Exception("Unknown type")
    };
}
```

Arrays are everywhere in this chapter! Now that we've learned about this data structure and its C# implementation-related topics, we can focus on some algorithms that are strictly related to arrays, namely sorting algorithms. Are you ready to get to know a few of them? If so, let's proceed to the next section.

Sorting algorithms

Many algorithms use arrays for a very broad range of applications. However, one of the most common tasks is **sorting an array to arrange its elements in the correct order, either ascending or descending**. Of course, you can sort data of various types, including numbers, strings, or even instances of user-defined classes. However, to keep things a bit simpler, here, we will only focus on sorting integer values.

IMAGINE A SORTING ALGORITHM

You benefit from the sorting procedure frequently in your daily life! For example, your inbox is sorted in a way to present the newest messages first (by sending date in descending order), your calendar presents a day plan sorted by hours (by event start date in ascending order), as well as your list of tasks shows entries from the most important to the least important (by priority in descending order). That's not all – at work, you sort documents by their issue date, then you choose a suitable road to home from the variants sorted by time to reach the destination, and in the evening, you change programs on the TV using a remote control according to the predefined order of channels.

Sorting algorithms involve many approaches and are also a popular subject of research. There are a lot of sorting types, including selection sort, insertion sort, bubble sort, merge sort, Shell sort, quicksort, and heap sort. These will be explained in detail in this chapter. However, these are not all of the available approaches. Various types differ in their performance results, which is one of the most important aspects that you should take into account while choosing your sorting implementation. This topic will be analyzed at the end of this chapter to give you some tips in this area.

WHERE CAN YOU FIND MORE INFORMATION?

Array sorting is a popular topic that's presented in various resources in books and research papers, as well as online. For example, you can read more about sorting algorithms presented in this chapter at Wikipedia, as well as you can take a look at some implementation codes at Wikibooks. You can browse for more information about the merge sort at https://en.wikipedia.org/wiki/Merge_sort and https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Merge_sort, about Shell sort at https://en.wikipedia.org/wiki/Shellsort and https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Shell_sort, about quicksort at https://en.wikipedia.org/wiki/Quicksort and https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Quicksort, and about heap sort at https://en.wikipedia.org/wiki/Heapsort and https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Heapsort. In a similar way, you can find information about other sorting algorithms. Of course, Wikipedia, together with Wikibooks, is not the only available source of content regarding such algorithms. There are a huge number of websites dedicated to this subject. Some of them also contain animations that show how various algorithms operate. This can help you visualize how they work.

Selection sort

Let's start with **selection sort**, which is one of the simplest sorting algorithms. **This algorithm divides the array into two parts, namely sorted and unsorted**. First, the sorted part is empty. In the following iterations, **the algorithm finds the smallest element in the unsorted part and exchanges it with the first element in the unsorted part**. Thus, the sorted part increases by one element. This sounds quite simple, doesn't it?

To better understand the selection sort algorithm, let's take a look at the following iterations for an array with nine elements (`-11`, `12`, `-42`, `0`, `1`, `90`, `68`, `6`, and `-9`), as shown in the following figure:

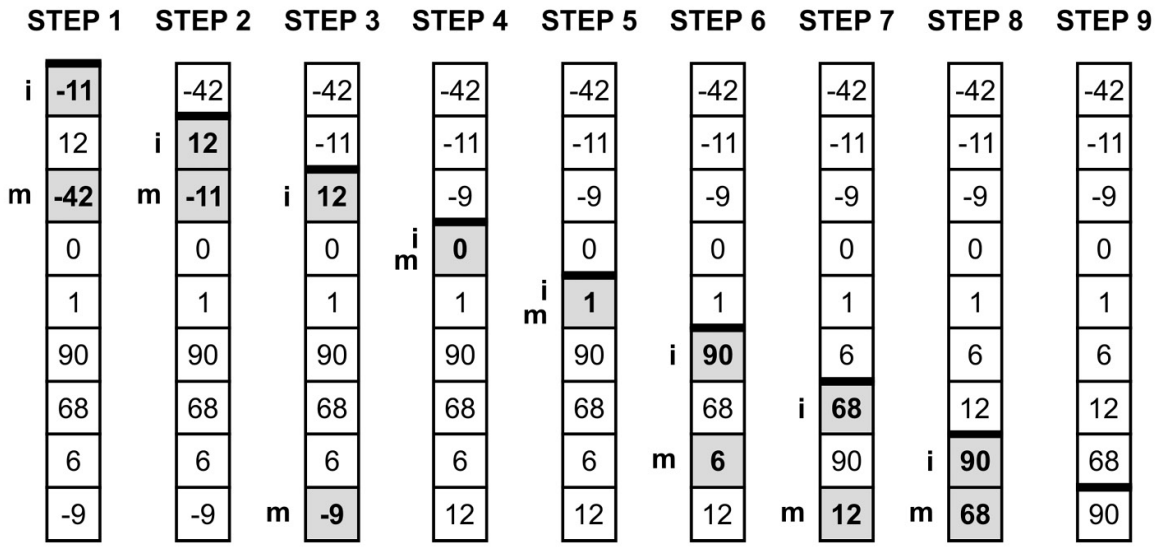

Figure 3.7 – Illustration of the selection sort algorithm

Bold lines are used to present the borders between the sorted and unsorted parts of the array. First (*Step 1*), the border is located just at the top of the array, which means that the sorted part is empty. Here, the algorithm finds the smallest value in the unsorted part (namely `-42`) and swaps it with the first element in this part (`-11`). The result is shown in *Step 2*, where the sorted part contains one element (`-42`), while the unsorted part consists of eight elements. In the next step, the algorithm finds `-11` as the smallest value in the unsorted part and swaps it with `12`, which is the first element in the unsorted part. As a result, the sorted part consists of two elements, namely `-42` and `-11`, while the unsorted part contains only seven elements, as shown in *Step 3*. The aforementioned steps are performed a few times until only one element is left in the unsorted part. The final result is shown in *Step 9*.

With that, you know how the selection sort algorithm works, but what role is performed by the `i` and `m` indicators shown on the left in the preceding diagram? They are related to the variables that are used in the implementation of this algorithm. So, it is time to see the code in the C# language!

The implementation is created within the `Sort` method, which takes the `a` array as the parameter and sorts it using selection sort:

```
void Sort(int[] a)
{
    for (int i = 0; i < a.Length - 1; i++)
    {
        int minIndex = i;
        int minValue = a[i];
        for (int j = i + 1; j < a.Length; j++)
        {
            if (a[j] < minValue)
            {
                minIndex = j;
```

```
                minValue = a[j];
            }
        }
        (a[i], a[minIndex]) = (a[minIndex], a[i]);
    }
}
```

A **`for`** loop is used to iterate through the elements until only one item is left in the unsorted part. Thus, the number of iterations of the loop is equal to the length of the array minus one (**`a.Length - 1`**). In each iteration, another **`for`** loop is used to find the smallest value in the unsorted part (**`minValue`**, from the **`i + 1`** index until the end of the array), as well as to store an index of the smallest value (**`minIndex`**, referred to as the **`m`** indicator in the preceding diagram). Finally, the smallest element in the unsorted part (with an index equal to **`minIndex`**) is swapped with the first element in the unsorted part (the **`i`** index).

That's all! Let's use the following code to test the implementation of the selection sort algorithm:

```
int[] array = [-11, 12, -42, 0, 1, 90, 68, 6, -9];
Sort(array);
Console.WriteLine(string.Join(" | ", array));
```

In the preceding code, an array is declared and initialized. Then, the **`Sort`** method is called, passing the **`array`** as a parameter. Finally, the **`string`** value is created by joining elements of the array, separated by **`|`**. The result is shown in the console:

```
-42 | -11 | -9 | 0 | 1 | 6 | 12 | 68 | 90
```

Since we're talking about various algorithms, one of the most important topics is computational complexity, especially time complexity. In the case of selection sort, both **the worst and the average time complexity is O(n^2)**. Why? Let's take a look at the code to answer this question. There are two **`for`** loops (one within the other), each iterating through many elements of the array, which contains *n* elements. For this reason, the complexity is indicated as *O(n^2)*.

A SMALL REMINDER ABOUT COMPUTATIONAL COMPLEXITY

You learned about computational complexity in the previous chapter. As a quick reminder, there are a few variants, such as for the worst or average case. This complexity can be interpreted as the number of basic operations that need to be performed by the algorithm, depending on the input size (n). The time complexity can be specified using Big O notation – for example, as O(n), O(n^2), O(n log(n)) or O(1). As an example, the O(n) notation indicates that the number of operations increases linearly with the input size (n).

With that, you've learned about selection sort. If you are interested in another approach to sorting, proceed to the next section, where insertion sort is presented.

Insertion sort

Insertion sort is another algorithm that makes it possible to sort a single-dimensional array simply. Here, **the array is divided into two parts, namely sorted and unsorted**. However, at the beginning, the first element is included in the sorted part. In each iteration, **the algorithm takes the first element from the unsorted part and places it in a suitable location within the sorted part, to leave the sorted part in the correct order**. Such operations are repeated until the unsorted part is empty.

As an example, let's take a look at an illustration of sorting an array with nine elements (`-11`, `12`, `-42`, `0`, `1`, `90`, `68`, `6`, and `-9`) using insertion sort:

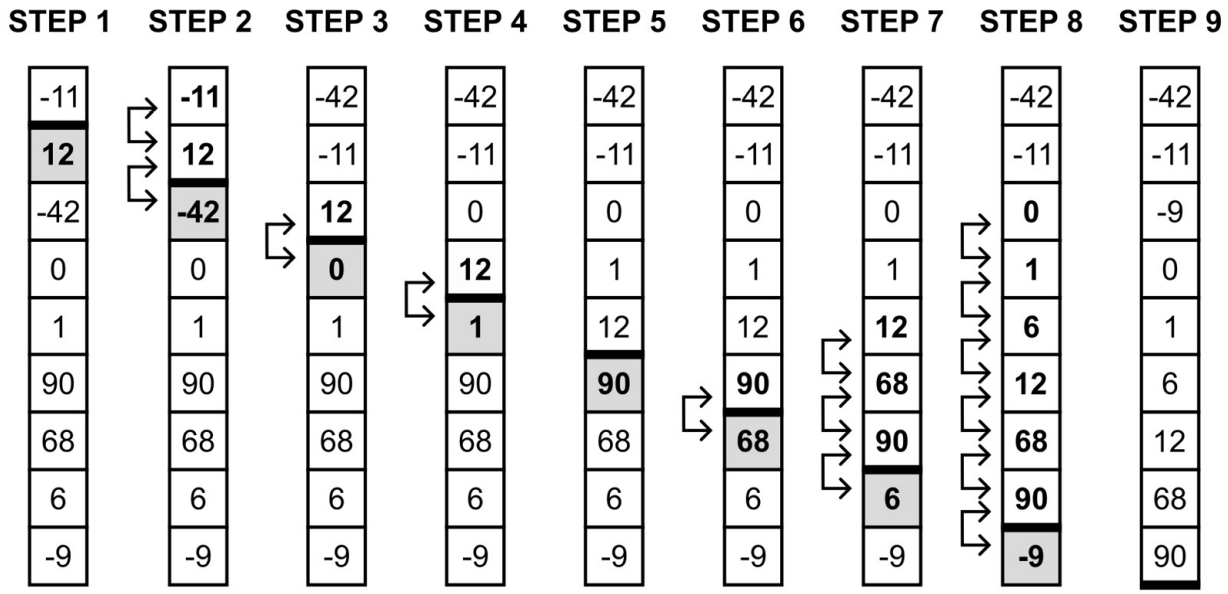

Figure 3.8 – Illustration of the insertion sort algorithm

First, only one element (namely `-11`) is located in the sorted part (*Step 1*). Then, you take the first element from the unsorted part (`12`). In this case, the location of this element does not need to be changed, so the sorted part is increased to two elements, namely `-11` and `12`. Then, you take `-42` as the first element in the unsorted part and you move it to the correct location in the sorted part. To do so, you need to perform two swap operations, as shown in *Step 2*. Thus, the length of the sorted part is increased to three elements, namely `-42`, `-11`, and `12`. In *Step 3*, you take `0` as the first element from the unsorted part and perform one swap operation to place it in the correct position, just before `12`, as presented in *Step 4*. At the same time, the size of the sorted part is increased to four already sorted elements, namely `-42`, `-11`, `0`, and `12`. Such operations are repeated until the unsorted part is empty (*Step 9*).

The implementation code for the insertion sort algorithm is very simple:

```
void Sort(int[] a)
```

```
{
    for (int i = 1; i < a.Length; i++)
    {
        int j = i;
        while (j > 0 && a[j] < a[j - 1])
        {
            (a[j], a[j - 1]) = (a[j - 1], a[j]);
            j--;
        }
    }
}
```

A `for` loop is used to iterate through all elements in the unsorted part. Thus, the initial value of the `i` variable is set to `1`, instead of `0`, because the unsorted part contains one element at the beginning. In each iteration of the `for` loop, a `while` loop is executed to move the first element from the unsorted part of the array (with the index equal to `i`) to the correct location within the sorted part, by swapping.

Finally, it is worth mentioning the time complexity of the insertion sort algorithm. Similarly, as in the case of the selection sort, both **the worst and average time complexity is O(n^2)**. If you take a look at the code, you will see two loops (`for` and `while`) placed one within the other, which could iterate multiple times, depending on the input size (*n*).

Bubble sort

The third sorting algorithm we'll cover is **bubble sort**. Its way of operation is very simple. **The algorithm just iterates through the array and compares adjacent elements. If they are located in an incorrect order, they are swapped.** It sounds very easy, doesn't it? Unfortunately, the algorithm is not efficient and its usage with large collections can cause performance-related problems.

To better understand how the algorithm works, let's take a look at the following figure, which shows how the algorithm operates in the case of sorting a single-dimensional array with nine elements (`-11`, `12`, `-42`, `0`, `1`, `90`, `68`, `6`, and `-9`):

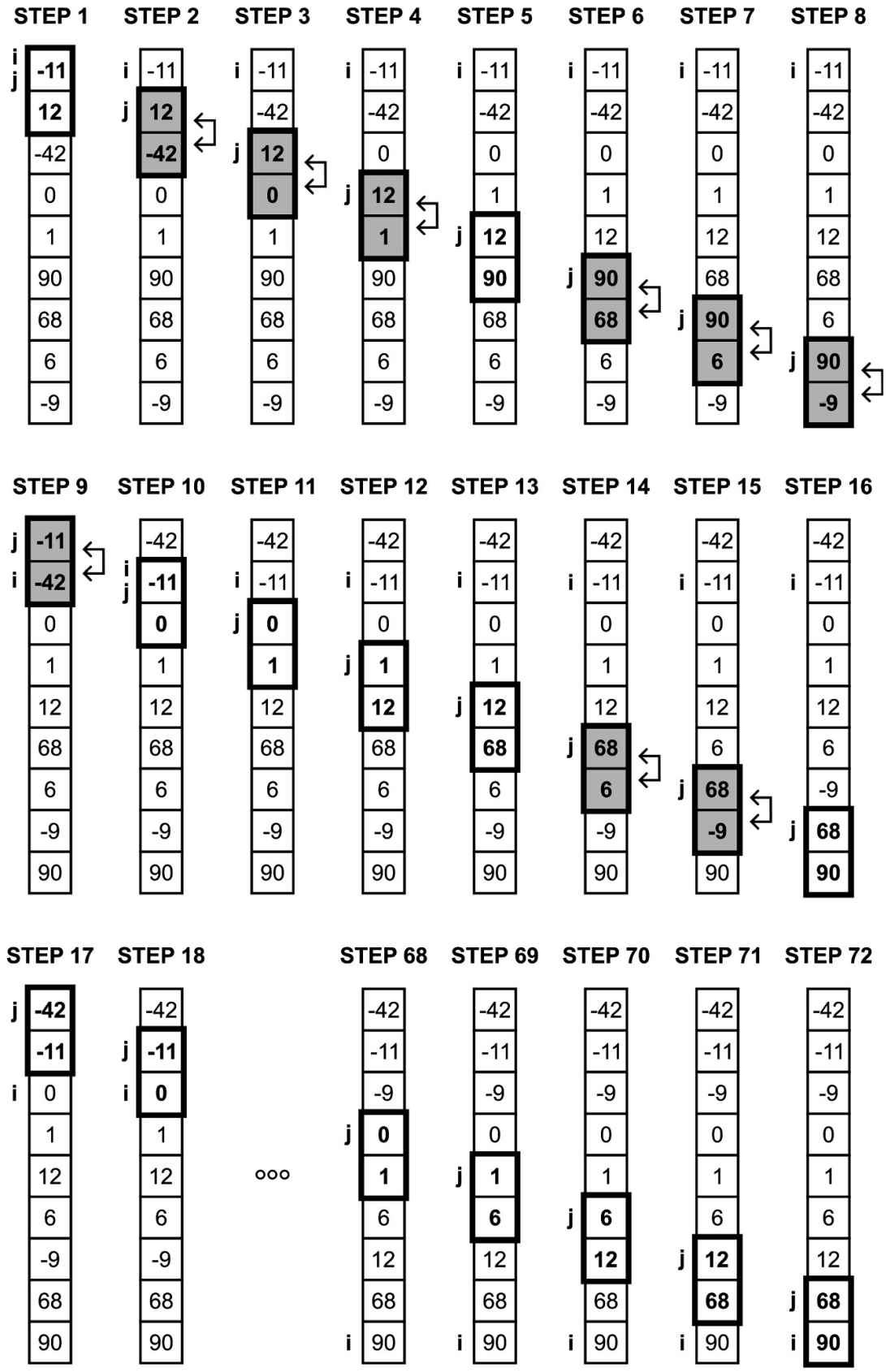
STEP 1
STEP 2
STEP 3
STEP 4
STEP 5
STEP 6
STEP 7
STEP 8
STEP 9
STEP 10
STEP 11
STEP 12
STEP 13
STEP 14
STEP 15
STEP 16
STEP 17
STEP 18
STEP 68
STEP 69
STEP 70
STEP 71
STEP 72
ooo

Figure 3.9 – Illustration of the bubble sort algorithm

In each step, the algorithm compares two adjacent elements in the array and swaps them, if necessary. For example, in *Step 1*, `-11` and `12` are compared. They are placed in the correct order, so it is not necessary to swap such elements. In *Step 2*, the next adjacent elements are compared (namely `12` and `-42`). This time, such elements are not placed in the correct order, so they are swapped. The aforementioned operations are performed many times. Finally, the array is sorted, as shown in *Step 72*.

The algorithm seems to be very easy, but what about its implementation? Is it also simple? Fortunately, yes! You just need to use two loops, compare adjacent elements, and swap them if necessary. That's all! Let's take a look at the following code snippet:

```
void Sort(int[] a)
{
    for (int i = 0; i < a.Length; i++)
    {
        for (int j = 0; j < a.Length - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                (a[j], a[j + 1]) = (a[j + 1], a[j]);
            }
        }
    }
}
```

Here, two `for` loops are used, together with a comparison and swapping. As mentioned previously, this algorithm is not efficient and its application can cause problems related to performance, especially in the case of large collections of data. However, it is possible to use a bit more efficient version of the bubble sort algorithm by introducing a simple modification. It is based on the assumption that **comparisons should be stopped when no changes are discovered during one iteration through the array**. The code is as follows:

```
void Sort(int[] a)
{
    for (int i = 0; i < a.Length; i++)
    {
        bool isAnyChange = false;
        for (int j = 0; j < a.Length - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                isAnyChange = true;
                (a[j], a[j + 1]) = (a[j + 1], a[j]);
            }
        }
        if (!isAnyChange) { break; }
    }
}
```

By introducing such a simple modification, the number of steps can decrease. In the preceding example, it decreases from 72 steps to 56 steps.

Before moving on to the next sorting algorithm, it is worth mentioning the time complexity of the bubble sort algorithm. As you may have already guessed, **both worst and average cases** are the same as in the case of the selection and insertion sort algorithms – that is, **O(n^2)**.

Merge sort

The fourth sorting algorithm operates in a significantly different way than the three already presented. This approach is named **merge sort**. **This algorithm recursively splits the array in half until the array contains only one element, which is sorted. Then, the algorithm merges the already sorted subarrays (starting with these with only one element) into the sorted array.** Finally, the whole array is sorted and the algorithm stops its operation.

To better understand the merge sort algorithm, let's take a look at the following iterations for an array with six elements (`-11`, `12`, `-42`, `0`, `90`, and `-9`):

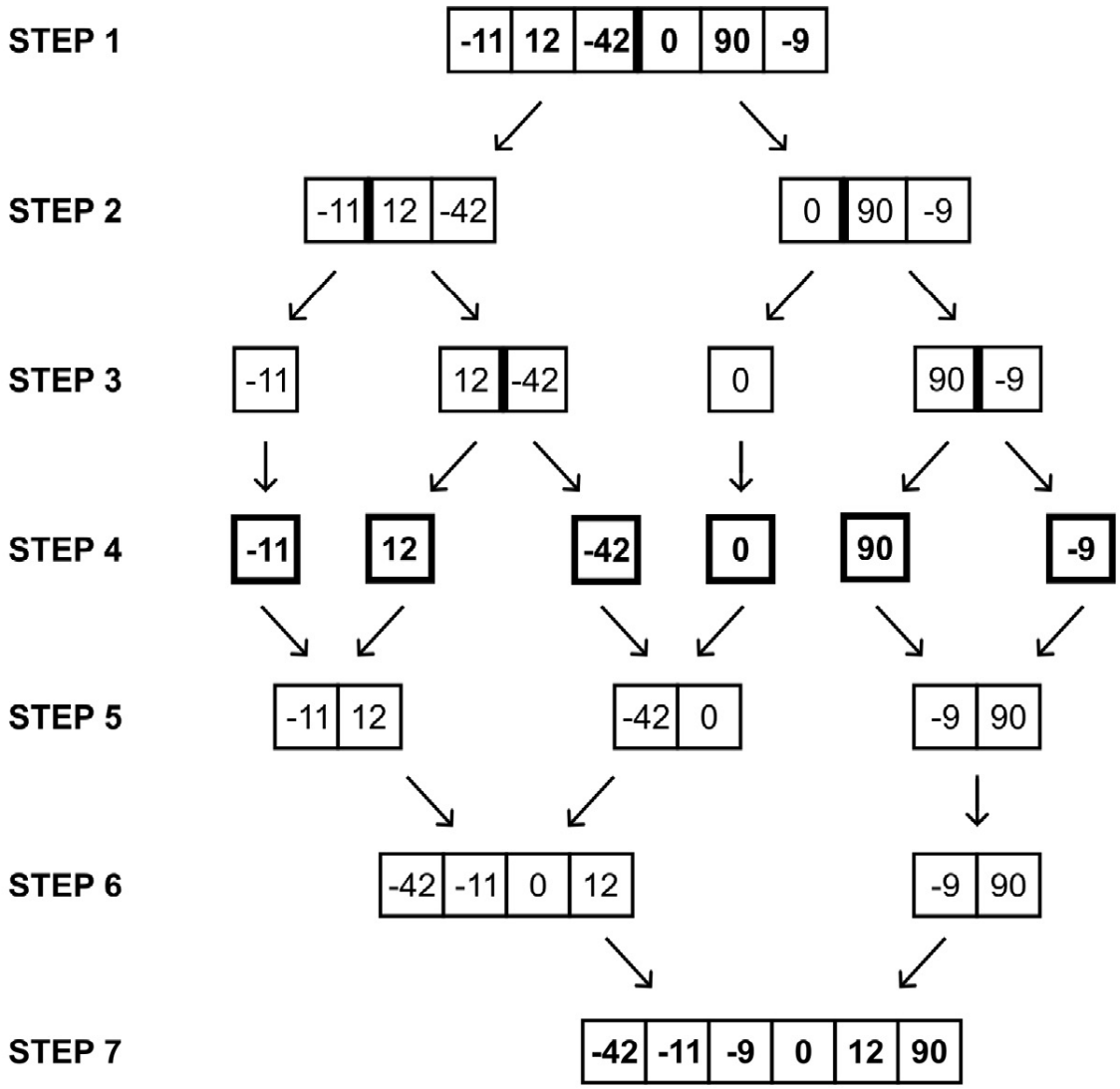

Figure 3.10 – Illustration of the merge sort algorithm

First (*Step 1*), you have the whole unsorted array, which you split into two parts, namely (`-11`, `12`, `-42`) and (`0`, `90`, `-9`), as shown in *Step 2*. In the next step, each of these subarrays is further split into (`-11`), (`12`, `-42`), (`0`), and (`90`, `-9`). In *Step 4*, you have the whole array divided into the subarrays with only one element each, namely (`-11`), (`12`), (`-42`), (`0`), (`90`), and (`-9`). Next, you merge all of these subarrays, together with sorting. Thus, in *Step 5*, you have three subarrays – that is, (`-11`, `12`), (`-42`, `0`), and (`-9`, `90`). Please keep in mind that these subarrays are already sorted. In *Step 6*, you need to merge and sort them further into (`-42`, `-11`, `0`, `12`) and (`-9`, `90`). Finally, you have the whole array sorted, namely (`-42`, `-11`, `-9`, `0`, `12`, `90`).

Does this seem simpler than just reading the textual description of the algorithm? If so, let's proceed to its implementation:

```
void Sort(int[] a)
{
    if (a.Length <= 1) { return; }
    int m = a.Length / 2;
    int[] left = GetSubarray(a, 0, m - 1);
    int[] right = GetSubarray(a, m, a.Length - 1);
    Sort(left);
    Sort(right);
    int i = 0, j = 0, k = 0;
    while (i < left.Length && j < right.Length)
    {
        if (left[i] <= right[j]) { a[k] = left[i++]; }
        else { a[k] = right[j++]; }
        k++;
    }
    while (i < left.Length) { a[k++] = left[i++]; }
    while (j < right.Length) { a[k++] = right[j++]; }
}
```

The `Sort` method is called **recursively** and takes the array that needs to be sorted as the parameter, named `a`. To stop infinitely calling this method recursively, you must specify the stop condition at the beginning. It simply checks whether the size of the array is not greater than 1. It is related to the assumption that you cannot further divide an array with one element only, because it is already sorted.

Next, you calculate an index of the middle element and store it as a value of `m`. In the following two lines, you call the auxiliary `GetSubarray` method, which creates a new array with only a part of elements, either from its left-hand side (with indices from `0` to `m-1`, stored as `left`) or the right-hand side (from `m` to the length of the array minus 1, stored as `right`). You will see its implementation after the explanation of the `Sort` method. Coming back to the explanation of the `Sort` method, you then recursively call the `Sort` method, passing the `left` and `right` subarrays.

The remaining part of the code is related to merging subarrays into the whole sorted array. Of course, this procedure is performed step by step, merging the subarrays into bigger and bigger subarrays until the whole array is sorted. You use a `while` loop to iterate through the `left` and `right` subarrays. You use three auxiliary variables, namely `i` as an index of the currently analyzed element from the `left` array, `j` from the `right` array, and `k` from the `a` array. Initially, all of them are set to `0`, so you keep an eye on the first element of the `left`, `right`, and `a` arrays.

Within the `while` loop, you check whether the current element from the `left` array (with the `i` index) is not greater than the current element from the `right` array (with the `j` index). If so, you place the current element from the `left` array as the first element in the `a` array. You also increase the `i` index, which means that the second element from the `left` array is the current one. If this condition is not met – that is, the current element from the `right` array is smaller than the current element from the `left` array – you use the current element from the `right` array as the first element in the `a` array and

increase the `j` index. Finally, you increase the `k` index to keep an eye on the second element from the `a` array. The `while` loop ends when you are out of bounds of either the `left` or `right` array.

What about when some elements haven't been analyzed yet from the `left` or `right` arrays? To handle such cases, you use two additional `while` loops. These allow you to place the remaining elements from either the `left` or `right` array on the remaining places in the `a` array. As you can see, the `Sort` method is equipped with a very simple way of merging two arrays into one, together with their sorting.

While explaining the algorithm's implementation, the `GetSubarray` auxiliary method was mentioned. So, let's show its code, together with a short explanation:

```
int[] GetSubarray(int[] a, int si, int ei)
{
    int[] result = new int[ei - si + 1];
    Array.Copy(a, si, result, 0, ei - si + 1);
    return result;
}
```

This method uses the `Copy` static method of the `Array` class to copy a part of the source array (`a`) to the declared and initialized here destination array (`result`). To perform this task, you need to take the correct number of elements, namely `ei - si + 1`. Here, `ei` stands for *end index* and `si` stands for *start index*. You need to copy elements between arrays starting with the `si` index in the source array (`a`) and store them starting from the `0` index in the destination array (`result`).

Of course, you can fill a subarray in different ways, such as using a `for` loop, which iterates through elements and copies them accordingly. If you want, you can prepare the alternative implementation on your own and then compare it during the performance tests, which you will see later in this chapter.

What about the time complexity? It's not very easy to specify it in the case of the merge sort algorithm compared to the other sorting algorithms I've presented. However, its time complexity is much better and can be indicated as **O(n log(n)) for both average and worst cases**. You will see what this means in practice while analyzing the performance results.

However, you still have some algorithms to learn about, so let's proceed to the next one.

Shell sort

A different approach to sorting is used in the **Shell sort** algorithm, whose name comes from its author's name. It is a variation of the already presented insertion sort. **The algorithm performs h-sorting to sort virtual subarrays consisting of elements with a distance equal to h, using the insertion sort. At the beginning, h is set to half of the array's length and is divided by 2 in each**

iteration, until it is equal to 1. This description can seem a bit complicated, but it is a surprisingly efficient algorithm with a very simple implementation.

First, let's take a look at a figure that should make this topic much simpler and easier to understand than just the plain text:

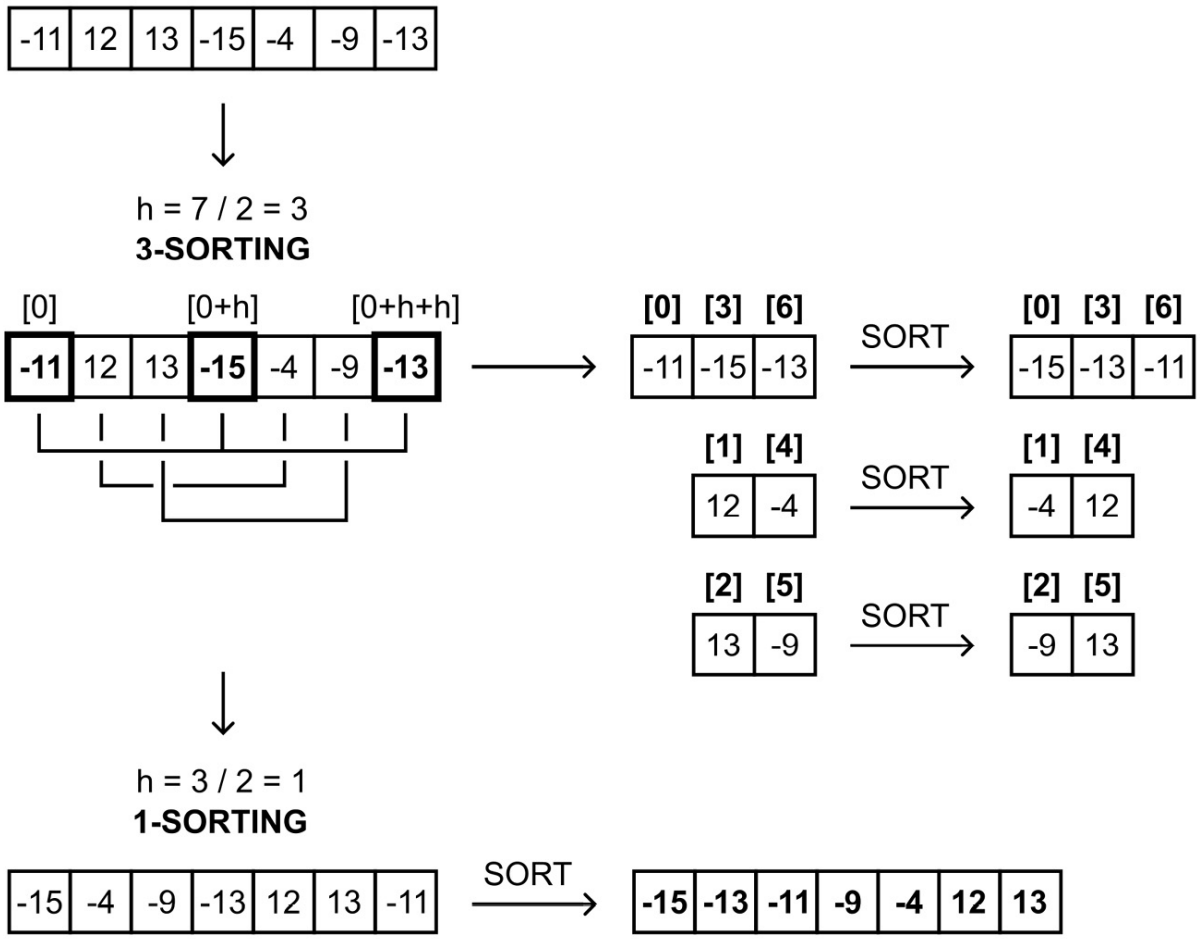

Figure 3.11 – Illustration of the Shell sort algorithm

As the source array contains 7 elements, the initial `h` value is set to `3`. So, now, it is time for **3-sorting**. You create virtual subarrays with elements whose indices differ by 3. This means that the following indices are used: (`0`, `3`, `6`), (`1`, `4`), and (`2`, `5`). The first virtual subarray consists of (`-11`, `-15`, `-13`), so you sort it and receive (`-15`, `-13`, `-11`). The second is (`12`, `-4`) and forms (`-4`, `12`) after sorting. The last is (`13`, `-9`) and is sorted into (`-9`, `13`). When 3-sorting is completed, you calculate the next *h* value, simply by dividing the current value by 2. The result is 1 and it is also the last *h*-sorting iteration, namely **1-sorting**. Now, you perform a simple insertion sort.

The illustration and description look pretty simple, don't they? Let's write some C# code to implement the Shell sort algorithm, as shown below:

```
void Sort(int[] a)
{
    for (int h = a.Length / 2; h > 0; h /= 2)
    {
        for (int i = h; i < a.Length; i++)
        {
            int j = i;
            int ai = a[i];
            while (j >= h && a[j - h] > ai)
            {
                a[j] = a[j - h];
                j -= h;
            }
            a[j] = ai;
        }
    }
}
```

The implementation consists of three loops. The first `for` loop is used to calculate proper values of `h`, starting with the array (`a`) length divided by `2`. It is further divided by `2` after each iteration and the last acceptable value is `1`.

The next `for` loop calculates the `i` index, starting with `h`, and increases it until the end of the array is reached. This part is used to perform the insertion sort on virtual subarrays.

Within the loop, you can use the `ai` variable to store the current value of the element with the `i` index, so you can replace it later with another value. Then, a `while` loop is used to shift elements in the virtual subarray to find the correct location for `ai`. Finally, you store the `ai` variable in the location indicated by the `j` variable.

As you can see, the implementation is very short and quite simple. What's more, this algorithm is efficient and can be used for sorting large collections of data, as you will see later in this chapter. But what about the time complexity? **In the worst case, it is $O(n^2)$**. However, its **average time complexity is about O(n log(n))**.

Quicksort

The sixth sorting algorithm described in this book is **quicksort**. It is one of the popular algorithms from the divide and conquer group and divides a problem into a set of smaller ones. How does it work?

The algorithm picks some value (for example, from the last element of the array) as a pivot. Then, it reorders the array in such a way that values lower than the pivot are placed before it (forming the lower subarray), while values greater than or equal to the pivot are placed after it (the higher subarray). Such a process is called partitioning. Next, the algorithm recursively sorts each of the aforementioned subarrays. Each subarray is further divided into the next two

subarrays, and so on. The recursive calls stop when there are one or zero elements in a subarray because in such a case, there is nothing to sort.

The preceding description may sound a bit complicated. However, the following figure and the algorithm's implementation should remove any doubts.

The following diagram shows how the quicksort algorithm sorts a single-dimensional array with nine elements (`-11`, `12`, `-42`, `0`, `1`, `90`, `68`, `6`, and `-9`):

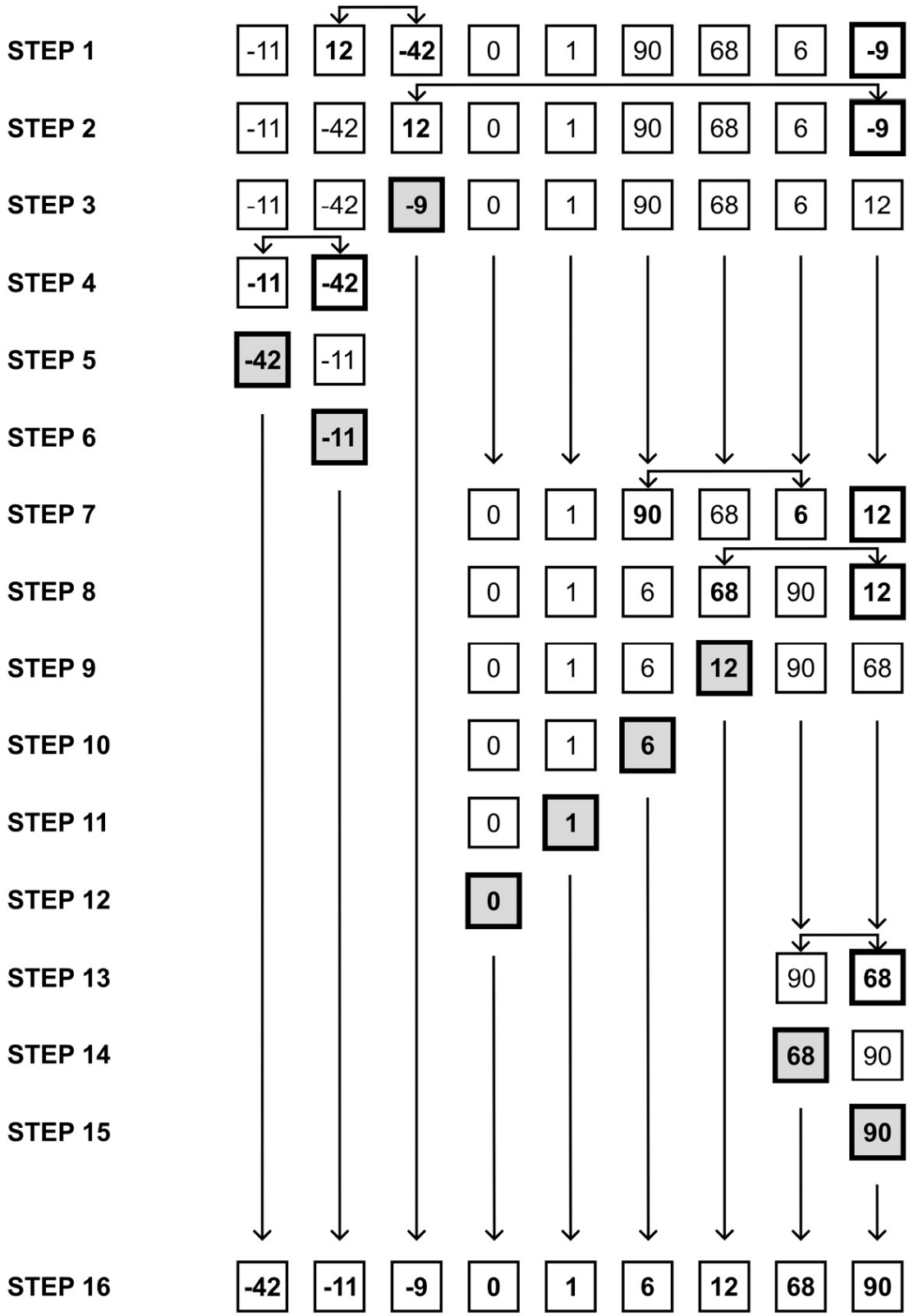

Figure 3.12 – Illustration of the quicksort algorithm

In our case, it is assumed that the pivot is chosen as a value of the last element of the subarray that is currently being sorted. In *Step 1*, `-9` is chosen as the pivot. Then, it is necessary to swap `12` with `-42` (*Step 1*), as well as `12` with `-9` (*Step 2*), to ensure that only values lower than the pivot (`-11`, `-42`) are in the lower subarray and only values greater than or equal to the pivot (`0`, `1`, `90`, `68`, `6`, `12`) are placed in the higher subarray (*Step 3*). Then, the algorithm is called recursively for both aforementioned subarrays, namely (`-11`, `-42`, from *Step 4*) and (`0`, `1`, `90`, `68`, `6`, `12`, from *Step 7*), so that they are processed in the same way as the input array.

As an example, *Step 7* shows that `12` is chosen as the pivot. After partitioning, the subarray is divided into two other subarrays, namely (`0`, `1`, `6`) and (`90`, `68`). For both, other pivot elements are chosen, namely `6` and `68`. After performing such operations for all the remaining parts of the array, you receive the result shown in *Step 16*.

It is worth mentioning that the pivot can be selected variously in other implementations of this algorithm. Now that you understand how the algorithm works, let's proceed to its implementation. It's no more complicated than the examples shown earlier, and it uses **recursion** to call the sorting method for subarrays. The main code is as follows:

```
void Sort(int[] a)
{
    SortPart(a, 0, a.Length - 1);
}
```

The `Sort` method takes only one parameter, namely the array that should be sorted. It just calls the `SortPart` method, which makes it possible to **specify the lower and upper indices, which indicate which part of the array should be sorted**. The code of the `SortPart` method is shown here:

```
void SortPart(int[] a, int l, int u)
{
    if (l >= u) { return; }
    int pivot = a[u];
    int j = l - 1;
    for (int i = l; i < u; i++)
    {
        if (a[i] < pivot)
        {
            j++;
            (a[j], a[i]) = (a[i], a[j]);
        }
    }
    int p = j + 1;
    (a[p], a[u]) = (a[u], a[p]);
    SortPart(a, l, p - 1);
    SortPart(a, p + 1, u);
}
```

First, the method checks whether the array (or subarray) has at least two elements by comparing the values of the `l` (*lower index*) and `u` (*upper index*) variables. If not, you return from this method.

Otherwise, you perform the partitioning phase.

Here, the pivot is chosen as a value of the last element in the array (or subarray) and stored as a value of the `pivot` variable. Then, a `for` loop is used to rearrange the array using comparisons and swapping elements. You need to perform this stage to ensure that values lower than the pivot are placed before it, while values greater than or equal to the pivot are placed after it.

Finally, you store a new index of the pivot value as `p` and perform swapping to place it there. The `p` variable is also used to calculate lower and upper bounds for subarrays, namely as (`l`, `p-1`) and (`p+1`, `u`). Such ranges are then used while calling the `SortPart` method recursively for the lower and upper parts. That's all!

What about the time complexity? It has **O(n log(n)) average time complexity, despite having O(n^2) worst time complexity**. Does this look similar to Shell sort to you? If so, you are right! You are coming closer and closer to the end of this chapter, where you will see results from conducting performance tests on various sorting algorithms.

Heap sort

The last approach we'll cover is based on an interesting data structure known as a **binary heap**. To give you a brief introduction, **it is a tree-based structure where each node contains either zero, one, or two child nodes**. You'll learn more about trees and their variants later in this book.

It won't come as a surprise to you that the sorting solution is named **heap sort**. **First, the algorithm builds a max-heap from the array (the heapify operation). Then, it repeats a few steps until there is only one element in the heap:**

1. **Swap the first element (root with the maximum value) with the last element.**
2. **Remove the last element (which is currently the maximum value) from the heap.**
3. **Build the max-heap again.**

By performing these operations, you efficiently receive the sorted array.

As a new data structure must be introduced here, let's look at what the binary heap looks like and how the algorithm operates to sort the example array:

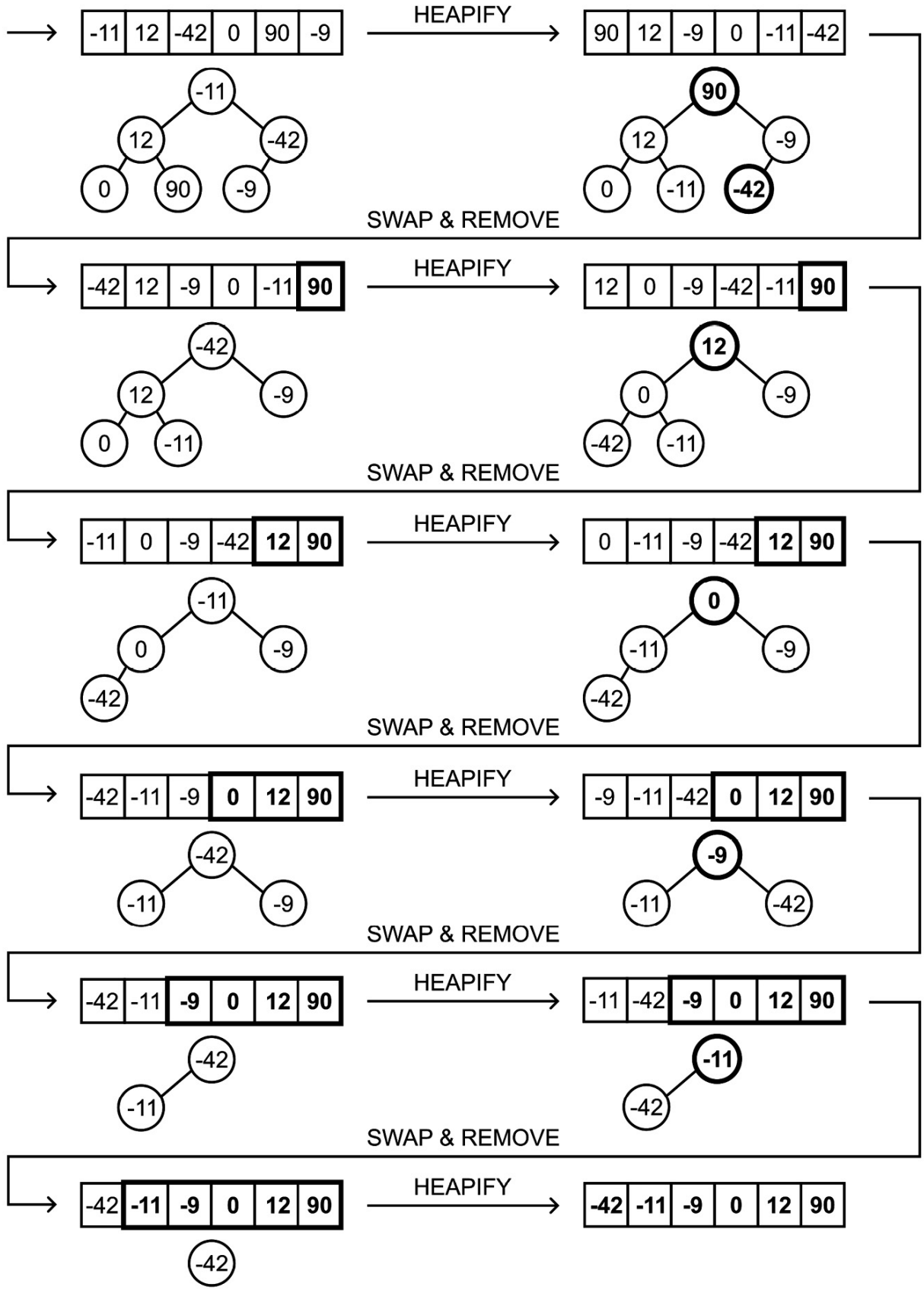
-11 12 -42 0 90 -9
HEAPIFY
90 12 -9 0 -11 -42
SWAP & REMOVE
-42 12 -9 0 -11 90
HEAPIFY
12 0 -9 -42 -11 90
SWAP & REMOVE
-11 0 -9 -42 12 90
HEAPIFY
0 -11 -9 -42 12 90
SWAP & REMOVE
-42 -11 -9 0 12 90
HEAPIFY
-9 -11 -42 0 12 90
SWAP & REMOVE
-42 -11 -9 0 12 90
HEAPIFY
-11 -42 -9 0 12 90
SWAP & REMOVE
-42 -11 -9 0 12 90
HEAPIFY
-42 -11 -9 0 12 90

Figure 3.13 – Illustration of the heap sort algorithm

The input array consists of six elements, namely `-11`, `12`, `-42`, `0`, `90`, and `-9`. You form a binary heap from it by placing the first element as a *root* and then by adding two of its child nodes: `12` and `-42`. You do not have more space at this level of the heap, so let's add the following two elements from the array (`0` and `90`) as child nodes to the node with `12`. The last element from the array is left. You must place it as a child node of the node with `-42`. As you can see, you can easily map an array to a binary heap data structure and use an array as a data structure to store the data of a binary heap.

INTERESTING PROPERTIES OF A BINARY HEAP

Remember that a root node in a binary heap, represented by an array, is available at ***`array[0]`****. If you need to access the data of a parent node of the i-th element, you can get it from* ***`array[(i-1)/2]`****. The left and right child nodes of the i-th element are available in* ***`array[(2*i)+1]`*** *and* ***`array[(2*i)+2]`****, respectively.*

The next operation, which takes an important role in the heap sort algorithm, is named **heapify.** It can look a bit complicated, but after a short explanation, it should be clear. This operation aims to convert a binary heap into a **max-heap**. This means that **each node can contain only the child nodes whose values are smaller than or equal to the node's value**. As an example, let's take a look at the first row of the preceding figure. By using the *heapify* operation, `90` is located as the *root*. It contains `12` and `-9` as nodes. The node with `12` contains child nodes with smaller values, namely `0` and `-11`. The node with `-9` contains only one element, which is also smaller than it, namely `-42`.

MAX-HEAP IS NOT THE ONLY OPTION

You can also use the heapify operation to form the ***min-heap****. It is similar to the max-heap, but each node needs to meet the condition that the values of its child nodes are greater than or equal to the parent node's value.*

Let's proceed to the second row of the preceding figure. Here, the last element of the array (`90`) is already sorted. This is the result of swapping the root (previously, `90`) with the last element in the array (previously, `-42`). Then, you must perform another *heapify* operation and receive the max-heap with `12` as the *root*. The aforementioned actions are repeated until the heap contains only one element. In the end, you receive the sorted array, as shown in the bottom-right corner of the preceding figure.

At this point, you should be ready to analyze the implementation code in the C# language:

```
void Sort(int[] a)
{
    for (int i = a.Length / 2 - 1; i >= 0; i--)
    {
        Heapify(a, a.Length, i);
    }
    for (int i = a.Length - 1; i > 0; i--)
    {
        (a[0], a[i]) = (a[i], a[0]);
```

```
        Heapify(a, i, 0);
    }
}
```

The **`Sort`** method contains two **`for`** loops. The first performs the initial *heapify* operation to prepare the *max-heap*. You can do so by calling the **`Heapify`** multiple times, namely in reverse order and on each node that is not a leaf. Then, you have the array with data forming the *max-heap*.

The second **`for`** loop is performed until there is at least one element in the heap. In each iteration, it swaps the *root* element (with an index equal to **`0`**) with the last element, which has an index equal to **`i`**. Then, you need to restore the *max-heap* property, which you do by calling the **`Heapify`** method, regarding the affected part of the heap.

Now, let's take a look at the code of the **`Heapify`** method:

```
void Heapify(int[] a, int n, int i)
{
    int max = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    max = l < n && a[l] > a[max] ? l : max;
    max = r < n && a[r] > a[max] ? r : max;
    if (max != i)
    {
        (a[i], a[max]) = (a[max], a[i]);
        Heapify(a, n, max);
    }
}
```

It takes three parameters, namely an array (**`a`**), the number of elements in a heap (**`n`**), as well as an index of an element (**`i`**), which is a root of a subtree that should be *heapified*. First, you get an index of the maximum element (*root*, as **`max`**), as well as its left and right children (**`l`** and **`r`**, respectively). You can calculate indices according to the formulas presented earlier, namely **`2*i+1`** and **`2*i+2`**.

In the following two lines, you check whether the left child index (**`l`**) is still within the heap (**`l<n`**) and whether the element with this index (**`a[l]`**) is greater than the current root value (**`a[max]`**). If so, you update the root index (**`max`**). In the same way, you check the right child and adjust the **`max`** variable, if necessary.

In the next line, you check whether the *root* index changed during the mentioned operations. If so, this means that the current *root* is not the biggest value and you need to swap two elements in the array, namely representing the *root* (the **`i`** index) and the biggest value (the **`max`** index). Next, you recursively perform the *heapify* operation for the affected subtree, namely a tree with a new root value.

After this detailed explanation, it is worth mentioning the time complexity. It is very important in this case because the method is efficient and can be used successfully while sorting large data collections. **The time complexity is O(n log(n))**.

Despite learning about seven different sorting algorithms, please keep in mind that there are many more such algorithms available, including **block sort**, **tree sort**, **cube sort**, **strand sort**, and **cycle sort**. If you are interested in this topic, I strongly encourage you to take a look at them on your own. In the meantime, let's compare the algorithms we've covered in this chapter.

Performance analysis

To perform some tests, you need to configure your environment. So, let's start by preparing the code for running various sorting algorithms using the same input arrays.

Do you remember that each implementation presented in this chapter involves the `Sort` method, taking only one parameter (namely the `a` array)? Now, you can benefit from this assumption and create the `AbstractSort` abstract class, which requires you to implement this method while deriving from this class.

The code for the abstract class is as follows:

```
public abstract class AbstractSort
{
    public abstract void Sort(int[] a);
}
```

Then, you need to prepare a separate class for each sorting algorithm (such as `SelectionSort` or `HeapSort`) according to the following template:

```
public class SelectionSort
    : AbstractSort
{
    public override void Sort(int[] a) { (...) }
}
```

Since all the classes representing sorting algorithms derive from the base abstract class (`AbstractSort`)you can easily create a list containing their instances:

```
List<AbstractSort> algorithms = new()
{
    new SelectionSort(),
    new InsertionSort(),
    new BubbleSort(),
    new MergeSort(),
    new ShellSort(),
    new QuickSort(),
    new HeapSort()
};
```

The most interesting part of the code is shown here:

```
for (int n = 0; n <= 100000; n += 10000)
{
    Console.WriteLine($"\nRunning tests for n = {n}:");
```

```
    List<(Type Type, long Ms)> milliseconds = [];
    for (int i = 0; i < 5; i++)
    {
        int[] array = GetRandomArray(n);
        int[] input = new int[n];
        foreach (AbstractSort algorithm in algorithms)
        {
            array.CopyTo(input, 0);
            Stopwatch stopwatch = Stopwatch.StartNew();
            algorithm.Sort(input);
            stopwatch.Stop();
            Type type = algorithm.GetType();
            long ms = stopwatch.ElapsedMilliseconds;
            milliseconds.Add((type, ms));
        }
    }
    List<(Type, double)> results = milliseconds
        .GroupBy(r => r.Type)
        .Select(r =>
            (r.Key, r.Average(t => t.Ms))).ToList();
    foreach ((Type type, double avg) in results)
    {
        Console.WriteLine($"{type.Name}: {avg} ms");
    }
}
```

Here, you use a **`for`** loop to choose suitable values of **`n`**, which is the length of the input array used for sorting. You start with an array with zero elements (**`n`** = **`0`**) and end with hundreds of thousands of elements (**`n`** = **`100000`**), increasing the size by **`10000`** in each iteration. The values of **`n`** will be **`0`**, **`10000`**, **`20000`**, and **`30000`**, up to **`100000`**.

In each iteration, you create a new instance of the list (**`milliseconds`**) . Each its element stores a tuple consisting of two elements, namely a type of the sorting algorithm class (**`Type`**) and elapsed milliseconds of execution (**`Ms`**). Then, you use another **`for`** loop to perform such tests **`5`** times. In each of them, you get a random array (**`array`**) with a given size by calling **`GetRandomArray`**, which will be used as a template for each test. Next, you declare and initialize the input array (**`input`**).

The next part involves a **`foreach`** loop and iterates through all instances of classes deriving from **`AbstractSort`**. For each of them, you create an input array by copying elements from **`array`** to **`input`**. Then, you start the stopwatch and call the **`Sort`** method. As soon as it has finished running, you stop the stopwatch and add results to the **`milliseconds`** list.

The last part of the code is related to calculating the average result for each sorting algorithm and its presentation in the console. To do so, you use a few extension methods, such as **`GroupBy`**, **`Select`**, and **`Average`**, as well as a **`foreach`** loop.

The **`GetRandomArray`** method was mentioned earlier, so let's take a look at it:

```
int[] GetRandomArray(long length)
{
    Random random = new();
    int[] array = new int[length];
```

```
    for (int i = 0; i < length; i++)
    {
        array[i] = random.Next(-100000, 100000);
    }
    return array;
}
```

It uses the **`Random`** class to get a random integer value in the range of `<-100,000, 100,000)`. The whole array is filled with such random values.

At this point, your environment is ready and you can perform tests! So, let's run the code and see the results. I received the following values:

N	SELECTION SORT [ms]	INSERTION SORT [ms]	BUBBLE SORT [ms]	MERGE SORT [ms]	SHELL SORT [ms]	QUICK SORT [ms]	HEAP SORT [ms]
10 000	57	140	267	1	0	0	1
20 000	200	443	1 078	6	2	2	2
30 000	422	994	2 439	3	3	4	5
40 000	837	2 173	5 164	5	6	6	6
50 000	1 293	3 238	7 983	5	6	9	7
60 000	1 875	4 657	11 503	7	10	12	9
70 000	2 545	6 390	15 761	9	10	18	12
80 000	3 312	8 309	20 442	10	12	23	15
90 000	4 252	10 578	26 895	12	13	25	15
100 000	5 494	13 977	32 863	13	12	28	16

Figure 3.14 – Results of analyzing the performance of the sorting algorithms

Apart from the table and its data, let's take a look at the chart:

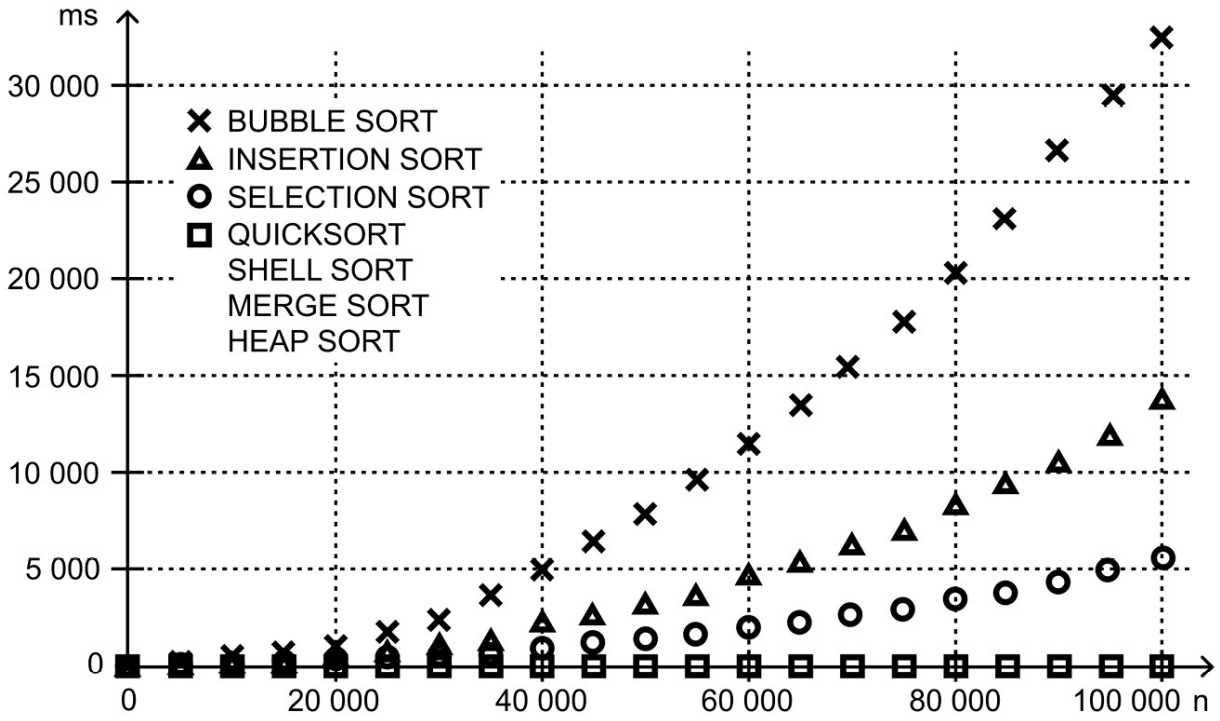

Figure 3.15 – Comparison of the sorting algorithms' performance results

As you can see, the worst results are received for bubble sort, then the insertion sort and selection sort algorithms. For an array with 100,000 elements, they need almost 33 seconds (bubble sort), almost 14 seconds (insertion sort), and more than 5 seconds (selection sort). Such values look extremely high compared to the results of merge sort, Shell sort, quicksort, and heap sort. These algorithms needed between 12 and 28 milliseconds! Does this seem surprising? It shouldn't if you recall time complexity.

Let's remember the average time complexity for the mentioned algorithms:

- $O(n^2)$: Selection sort, insertion sort, and bubble sort
- $O(n \log(n))$: Merge sort, Shell sort, quicksort, and heap sort

Oh, so it seems that such time complexities really have an importance! ;-) If you earlier had any doubts, it's the high time to take attention to the algorithms you use in your applications. You should choose them carefully and optimize the solution to handle various amounts of data that need to be processed.

DON'T FORGET ABOUT PERFORMANCE

Taking care of performance is important not only for sorting but for all operations that you perform in your mobile applications, web applications, APIs, and long-running background services. Let's try to write efficient code and test it not only by meeting the functional requirements but also by taking care of non-functional ones, such as those related to performance.

In the previous chart, you almost cannot see any data regarding the algorithms with *O(n log(n))* time complexity, so let's prepare another set of tests. Now, you can choose only these algorithms and increase the maximum number of *n* to one million! You can see my results in the following chart:

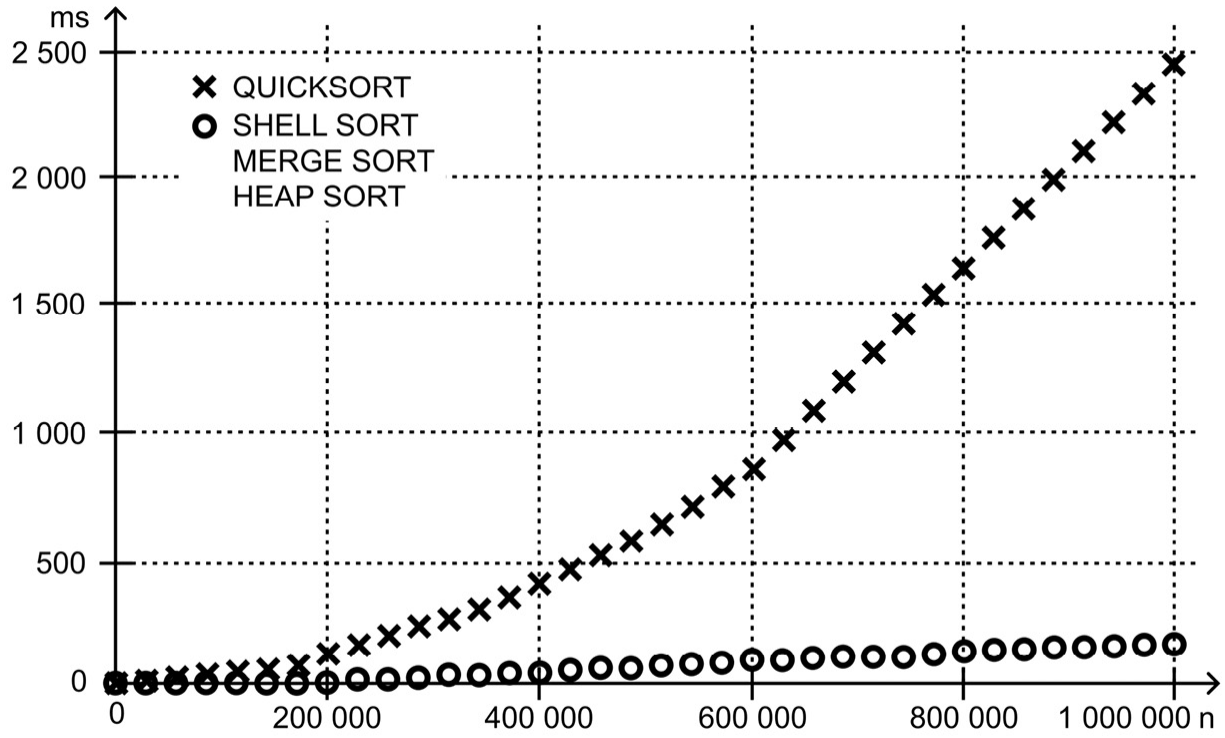

Figure 3.16 – Comparison of the sorting algorithms' performance results

There are some differences here, especially between quicksort and the remaining ones, namely Shell sort, merge sort, and heap sort. However, such changes are only visible with quite a huge input size and can be caused by the implementation details. All of the sorting algorithms with *O(n log(n))* time complexity are good solutions for sorting and can handle various amounts of data. It is also worth noting that these results were received on my device, so you may get different results. However, the relationship between the received number of elapsed milliseconds should be consistent.

Summary

Arrays are among the most common data structures that are used while developing various kinds of applications, such as mobile, web, or distributed ones. However, this topic is not as easy as it seems to be because even arrays can be divided into a few variants, namely **single-dimensional** and **multi-dimensional**, such as two-dimensional and three-dimensional, as well as **jagged arrays**, also referred to as arrays of arrays.

While talking about arrays, don't forget about **sorting algorithms**, which are one of the most popular algorithms used with this data structure. There are plenty of sorting algorithms that differ by their concept, application, implementation details, and performance results. In this chapter, you learned about seven different sorting algorithms, namely **selection sort**, **insertion sort**, **bubble sort**, **merge sort**, **Shell sort**, **quicksort**, and **heap sort**. Each of them was described, visualized in figures, and written in C# code.

At the end of this chapter, you saw how important time complexity is and how big an impact it can have on **performance results** while you're using algorithms with different computational complexity, such as $O(n^2)$ and $O(n \log(n))$. You learned how to prepare a simple environment for the performance tests and run them to get the results. They were later shown in a table, as well as in charts, together with an explanation.

Are you ready to learn other data structures? If so, proceed to the next chapter, where you'll learn about various **variants of lists**, namely simple, generic, sorted, as well as singly, doubly, and circular linked. You will see their implementation and a few examples of how you can use them in real-world examples.

4

Variants of Lists

In the previous chapter, you learned about arrays and their types. Of course, an array is not the only way of storing data. Another popular and even more powerful group of data structures contains various variants of **lists**. In this chapter, you will see such data structures in action, together with illustrations, explanations, and descriptions.

First, you will see a **simple list** as an array list and a generic list, in which you can easily add and remove elements according to your needs. Then, you will get to know **sorted lists**, which keep an order of elements. Next, you will learn about four variants of the **linked list**, namely a singly linked list, a doubly linked list, a circular singly linked list, and a circular doubly linked list. Finally, you will familiarize yourself with three list-related interfaces that you can use while developing applications. Does this sound a bit complicated? If so, don't worry. You will be guided throughout.

We will cover the following topics in this chapter:

- Simple lists
- Sorted lists
- Linked lists
- List-related interfaces

Simple lists

Arrays are really useful data structures and they are applied in many algorithms. However, in some cases, their application could be complicated due to their nature, which does not allow you to increase or decrease the length of the already-created array. What should you do if you do not know the total number of elements to store in the collection? Do you need to create a very big array and just not use unnecessary elements? Such a solution does not sound good, does it? A much better approach is to use a data structure that makes it possible to dynamically increase and decrease the size of the collection if necessary.

IMAGINE A SIMPLE LIST

If you want to better visualize a simple list and distinguish it from an array, close your eyes for a moment and try to think back to when you were just a few years old and Christmas was approaching. You and your family were preparing a chain to hang on the Christmas tree. You took another piece of paper, passed it through the last piece of the chain, and glued the new piece of the chain together. In this way, your chain grew by another element, and you could add more and more elements to the chain, basically endlessly. Well, maybe the limitation was the amount of paper and glue or your tiredness.

A list works somewhat similarly, where you can easily add new elements. You can also remove them, just like you can remove a piece of chain and then glue it back together and you can still hang it on your beautiful Christmas tree!

Array lists

The first data structure that allows you to **dynamically increase or decrease its size** is the **array list**. It is represented by the **`ArrayList`** class from the **`System.Collections`** namespace. You can use this class to store big collections of data, to which you can easily add new elements when necessary. Of course, you can also remove them, count items, and find an index of a particular value stored within the array list. How can you do this? Let's take a look at the following code:

```
using System.Collections;
ArrayList arrayList = new() { 5 };
arrayList.Add(6);
arrayList.AddRange(new int[] { -7, 8 });
arrayList.AddRange(new object[] { "Marcin", "Kate" });
arrayList.Insert(5, 7.8);
```

In the first line, a new instance of the **`ArrayList`** class is created and **`5`** is added as the first element. This can be simplified, as shown here:

```
ArrayList arrayList = [5];
```

Then, you use the **`Add`**, **`AddRange`**, and **`Insert`** methods to add new elements to the array list. The difference between them is as follows:

- **`Add`** adds a new item at the end of the list
- **`AddRange`** adds a collection of elements at the end of the array list
- **`Insert`** places an element in a specified location within the collection

When the preceding code is executed, the array list contains the following elements: **`5`**, **`6`**, **`-7`**, **`8`**, **`"Marcin"`**, **`7.8`**, and **`"Kate"`**. Please keep in mind that all the items stored within the array list are of the **`object`** type. Thus, you can place various types of data in the same collection at the same time.

DO YOU NEED TO SPECIFY A TYPE?

*Using **`object`** instead of a particular type is not always a good idea. So, if you want to specify a type of each element stored within the list, you can use the generic **`List`** class, which will be described just after **`ArrayList`**. I encourage you to use a strongly typed version of a collection whenever possible.*

It is worth mentioning that you can easily access a particular element within the array list using an index, as shown in the following two lines of code:

```
object first = arrayList[0]!;
int third = (int)arrayList[2]!;
```

Let's take a look at casting to **int** in the second line. Such casting is necessary because the array list stores **object** values. As in the case of arrays, the zero-based indices are used while accessing particular elements within the collection. When you run the preceding lines of code, **first** will be equal to **5**, while **third** will be equal to **-7**.

Of course, you can use a **foreach** loop to iterate through all items, as follows:

```
foreach (object element in arrayList)
{
     Console.WriteLine(element);
}
```

That's not all – the **ArrayList** class has a set of properties and methods that you can use while developing applications utilizing the aforementioned data structure. To start with, let's take a look at the **Count** and **Capacity** properties:

```
int count = arrayList.Count;
int capacity = arrayList.Capacity;
```

The first property (**Count**) returns the number of elements stored currently in the array list, while the other property (**Capacity**) indicates how many elements can be stored within it. If you check the value of the **Capacity** property after adding new elements to the array list, you will see that this value is automatically increased to prepare a place for new items. This is shown in the following figure, which presents the difference between **Count** (marked as **A**) and **Capacity** (**B**):

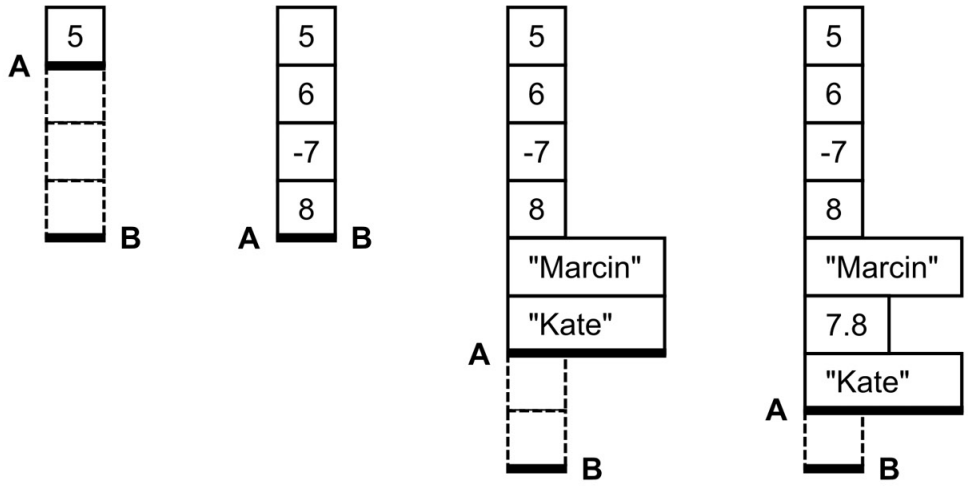

Figure 4.1 – The difference between Count and Capacity

The next common and important task is checking whether the array list contains an element with a particular value. You can perform this operation by calling the **Contains** method, as shown in the following line of code:

```
bool containsMarcin = arrayList.Contains("Marcin");
```

If the specified value is found in the array list, **`true`** is returned. Otherwise, **`false`** is returned. But how can you find an index of this element? To do so, you can use the **`IndexOf`** or **`LastIndexOf`** method, as shown in the following line of code:

```
int minusIndex = arrayList.IndexOf(-7);
```

The **`IndexOf`** method returns an index of the first occurrence of the element in the array list, while **`LastIndexOf`** returns an index of the last occurrence. If a value is not found, **`-1`** is returned by the methods. Thus, you can use **`IndexOf`** to check whether the array list contains a given element. If the result is smaller than zero, this means that the element is not available. On the other hand, if the result is greater than or equal to zero, it indicates that the item is found, as presented in the following line:

```
bool containsAnn = arrayList.IndexOf("Ann") >= 0;
```

Apart from adding some items to the array list, you can easily remove the already added elements using the **`Remove`**, **`RemoveAt`**, **`RemoveRange`**, and **`Clear`** methods, as shown here:

```
arrayList.Remove(5);
arrayList.RemoveAt(1);
arrayList.RemoveRange(1, 2);
arrayList.Clear();
```

The difference between the mentioned methods is as follows:

- **`Remove`** removes the first occurrence of a given value
- **`RemoveAt`** removes an item with a provided index
- **`RemoveRange`** removes a given number of elements starting from some index
- **`Clear`** removes all elements

Among other methods, it is worth mentioning **`Reverse`**, which reverses the order of the elements within the array list, as well as **`ToArray`**, which returns an array with all items stored in the **`ArrayList`** instance.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding an array list at https://learn.microsoft.com/en-us/dotnet/api/system.collections.arraylist.

Generic lists

As you can see, the **`ArrayList`** class contains a broad range of features, but it has a significant drawback – that is, it is not a strongly typed list. If you want to benefit from a strongly typed list, you

can use the generic `List` class, which represents the collection whose size can be increased and decreased as necessary. This class is available in the `System.Collections.Generic` namespace.

The generic `List` class contains many properties and methods that are useful while developing applications that store data. Many members are named the same as in the `ArrayList` class. An example is the following two properties:

- `Count`, which returns the current number of elements in the list
- `Capacity`, which indicates how many elements can be currently stored in the list

There are also many similar methods, including the following ones:

- `Add` adds an item at the end of the list
- `AddRange` adds a collection of elements at the end of the list
- `Insert` places an element in a specified location within the list
- `InsertRange` places a collection of items in a specified location in the list
- `Contains` checks whether the list contains a given element
- `IndexOf` returns an index of the first occurrence of a given item
- `LastIndexOf` returns an index of the last occurrence of a given item
- `Remove` removes the first occurrence of a given value
- `RemoveAt` removes an item with a provided index
- `RemoveRange` removes a given number of elements starting from some index
- `Clear` removes all elements from the list
- `Reverse` reverses the order of items within the list
- `ToArray` returns an array with all items stored in the list

You can get a particular element from the list using the `[]` operator with an index.

Apart from the already-described features, you can use a comprehensive set of extension methods from the `System.Linq` namespace. Some of them are as follows:

- `Min` finds the minimum value in the list
- `Max` finds the maximum value in the list
- `Sum` returns a sum of all elements in the list
- `Average` calculates the average value of elements in the list
- `All` checks whether all elements in the list satisfy a condition
- `Any` verifies whether at least one element in the list satisfies a condition
- `ElementAtOrDefault` returns an element at a given index in the collection or a default value if the index is out of bounds
- `Distinct` returns a collection with only unique elements, namely without duplicates

- **`OrderBy`** and **`OrderByDescending`** order all elements in the list in ascending or descending order, as well as return the ordered collection
- **`Skip`** returns a collection bypassing a given number of elements in the list
- **`Take`** returns a given number of elements from the list

After this theoretical introduction, let's see such methods in action! First, let's get the minimum, maximum, sum, and average values from the list, as shown here:

```
List<int> list = [6, 90, -20, 0, 4, 1, 8, -20, 41];
int min = list.Min();
int max = list.Max();
int sum = list.Sum();
double avg = list.Average();
```

A value of **`min`** is equal to **`-20`**, **`max`** is equal to **`90`**, **`sum`** is equal to **`110`**, and **`avg`** is near **`12.22`**.

Now, let's check out some conditions on the list elements:

```
bool allPositive = list.All(x => x > 0);
bool anyZero = list.Any(x => x == 0);
```

Here, **`allPositive`** is equal to **`false`**, while **`anyZero`** to **`true`**.

The next part of the code is shown in the following block:

```
int existingElement = list.ElementAtOrDefault(5);
int nonExistingElement = list.ElementAtOrDefault(100);
```

Here, you use the **`ElementAtOrDefault`** method to get a value of the element with an index equal to **`5`** and **`100`**. In the first case, **`1`** is returned and stored as a value of the **`existingElement`** variable. When you try to get an element with the index equal to **`100`**, a default value for **`int`** is used instead and returned, namely **`0`**.

The next extension method is named **`Distinct`** and can be used as follows:

```
List<int> unique = list.Distinct().ToList();
```

It returns a unique collection of the **`IEnumerable<int>`** type, which you can convert into **`List<int>`** by calling the **`ToList`** extension method. The resulting list contains **`6`**, **`90`**, **`-20`**, **`0`**, **`4`**, **`1`**, **`8`**, and **`41`**.

Let's order the list using the **`OrderBy`** extension method, as follows:

```
List<int> ordered = list.OrderBy(x => x).ToList();
```

The target list consists of **`-20`**, **`-20`**, **`0`**, **`1`**, **`4`**, **`6`**, **`8`**, **`41`**, and **`90`**.

Another interesting group of methods consists of **`Skip`** and **`Take`**, as shown here:

```
List<int> skipped = list.Skip(4).ToList();
List<int> taken = list.Take(3).ToList();
```

The **`Skip`** method skips **`4`** elements and returns the collection with the remaining elements, namely **`4`**, **`1`**, **`8`**, **`-20`**, and **`41`**.

The **`Take`** method simply takes **`3`** first elements – that is, **`6`**, **`90`**, and **`-20`**.

Do you have any idea how to combine **`Skip`** with **`Take`** in some real-world examples? If not, just think about the **pagination** mechanism, which you can find on many websites. It allows you to navigate between pages of data, where each page contains a specified number of elements. How you can get such items for a given page? The answer is as follows:

```
int page = 1;
int size = 10;
List<int> items = list
    .Skip((page - 1) * size)
    .Take(size)
    .ToList();
```

Of course, these are not the only features available for developers while creating applications using generic lists in the C# language. I strongly encourage you to discover more possibilities on your own. Next, we'll look at two examples that show how to use a generic list in practice.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a generic list at https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.

Example – average value

The first example utilizes the generic **`List`** class to store floating-point values (of the **`double`** type) entered by the user. After typing a number, the average value is calculated and presented in the console. The program stops the operation when an incorrect value is entered. The code is as follows:

```
List<double> num = [];
do
{
    Console.Write("Enter the number: ");
    string numStr = Console.ReadLine() ?? string.Empty;
    if (!double.TryParse(numStr, out double n)) { break; }
    num.Add(n);
    Console.WriteLine($"Average value: {num.Average()}");
}
while (true);
```

First, an instance of the **`List`** class is created. Then, within the infinite loop (**`do-while`**), the program waits until the user enters a number. If it is correct, the entered value is added to the list (by calling **`Add`**), and an average value from elements in the list is calculated (by calling **`Average`**). The result is shown in the console:

```
Enter the number: 10.5
Average value: 10.5 (...)
```

```
Enter the number: 15.5
Average value: 9.375
```

In this section, you saw how to use a list that stores **`double`** values. However, can it also store instances of user-defined classes or records? Of course! You will see how to achieve this goal in the next example.

Example – list of people

This second example shows you how to use a list to create a very simple database of people. For each, a name, an age, and a country are stored. When the program is launched, some data of people are added to the list. Then, such data is sorted and presented in the console.

Let's start with the declaration of the **`Person`** record:

```
public record Person(string Name, int Age, string Country);
```

The record contains three properties, namely **`Name`**, **`Age`**, and **`Country`**, which stores a country code. In the main part of the code, you create a new instance of the **`List`** class and add the data of a few people with different names, ages, and countries, as shown here:

```
List<Person> people =
[
    new("Marcin", 35, "PL"),
    new("Sabine", 25, "DE"),
    new("Mark", 31, "PL")
];
```

In the next line, you sort the list by names of people in ascending order:

```
List<Person> r = people.OrderBy(p => p.Name).ToList();
```

This line can be simplified using the collection expression, as follows:

```
List<Person> r = [.. people.OrderBy(p => p.Name)];
```

Then, you iterate through all the results using a **`foreach`** loop:

```
foreach (Person p in r)
{
    string line = $"{p.Name} ({p.Age}) from {p.Country}.";
    Console.WriteLine(line);
}
```

After running the program, the following results will be presented:

```
Marcin (35) from PL.
Mark (31) from PL.
Sabine (25) from DE.
```

That's all! Now, let's talk a bit more about the LINQ expressions, which can be used not only to order elements but also to filter items based on the provided criteria, and even more.

As an example, let's take a look at the following query, which is using the **method syntax**:

```
List<string> names = people
    .Where(p => p.Age <= 30)
    .OrderBy(p => p.Name)
    .Select(p => p.Name)
    .ToList();
```

It selects the names (the `Select` clause) of all people whose age is lower than or equal to `30` years (the `Where` clause), ordered by names (the `OrderBy` clause). The query is then executed and the results are returned as a list (`ToList`).

The same task can be accomplished using the **query syntax**, as shown in the following example, combined with calling the `ToList` method:

```
List<string> names = (from p in people
                      where p.Age <= 30
                      orderby p.Name
                      select p.Name).ToList();
```

You saw how to use the `ArrayList` class and the generic `List` class to store data in collections, the size of which could be dynamically adjusted. However, this is not the end of list-related topics within this chapter. Are you ready to get to know another data structure that maintains the elements in the sorted order? If so, let's proceed to the next section, which focuses on sorted lists.

Sorted lists

So far, you've learned how to store data using simple lists. However, do you know that you can even use a data structure that ensures that the elements are sorted all the time? If not, let's get to know the `SortedList` generic class (from the `System.Collections.Generic` namespace), **which is a collection of key-value pairs, sorted by keys, without the necessity for you to sort them on your own**. It's worth mentioning that all keys must be unique and cannot be equal to `null`.

IMAGINE A SORTED LIST

If you want to imagine a sorted list, think about a business holder in which you put business cards that you have received from other people. Since you like order and want to always be able to quickly find a business card for a specific person, you make sure that they are all arranged in alphabetical order, by last name. What a terrible waste of time, especially if you have dozens of business cards and suddenly you have to put in a card for Mrs. Ana Ave. Oh, no... almost all the business cards have to be moved. What can help you at this point is a sorted list! On its basis, your magic business card holder works, which automatically inserts a new business card into the appropriate place in the business card holder. Thanks to this, you always have order and you do not have to waste time constantly taking out and inserting business cards. Congratulations!

You can easily add an element to a sorted list using the `Add` method, as well as remove a specified item using the `Remove` method. Among other methods, it is worth noting `ContainsKey` and

ContainsValue for checking whether the collection contains an item with a given key or value, as well as **IndexOfKey** and **IndexOfValue** for returning an index of an element by its key or value.

As a sorted list stores key-value pairs, you have also access to the **Keys** and **Values** properties. Particular keys and values can be easily obtained using the **[]** operator together with an index. As you can see, this data structure is quite similar to the ones that have already been presented. However, it has some significant differences. So, let's take a look at an example that will show you how to use this data structure. You will also see differences in code compared with the previously described **List** class.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a sorted list at https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2.

Example – address book

This example uses the **SortedList** class to create a very simple address book, which is sorted by names of people. For each person, the following data is stored: **Name**, **Street**, **PostalCode**, **City**, and **Country**. The declaration of the **Person** record is shown in the following code:

```
public record Person(
    string Name,
    string Street,
    string PostalCode,
    string City,
    string Country);
```

In the main part of the code, you create a new instance of **SortedList**. You need to specify types for keys and values, namely **string** and **Person**. Within the following part of the code, you also initialize the sorted list with data for **Marcin** and **Martyna**:

```
SortedList<string, Person> people = new()
{
    { "Marcin Jamro", new("Marcin Jamro",
      "Polish Street 1/23", "35-001", "Rzeszow", "PL") },
    { "Martyna Kowalska", new("Martyna Kowalska",
      "World Street 5", "00-123", "Warsaw", "PL") }
};
```

Then, you can easily add data to the sorted list by calling the **Add** method, passing two parameters, namely a key (that is, a name), and a value (that is, an instance of the **Person** record), as shown in the following code snippet regarding **Mark**:

```
people.Add("Mark Smith", new("Mark Smith",
    "German Street 6", "10000", "Berlin", "DE"));
```

When all the data is stored within the collection, you can easily iterate through its elements (namely through key-value pairs) using a `foreach` loop. It is worth mentioning that a type of the variable that's used in the loop is `KeyValuePair<string, Person>`. However, you can use a value tuple to get access to a key (`k`) and a value (`p`):

```
foreach ((string k, Person p) in people)
{
    Console.WriteLine($"{k}: {p.Street}, {p.PostalCode}
        {p.City}, {p.Country}.");
}
```

When the program is launched, you receive the following result in the console:

```
Marcin Jamro: Polish Street 1/23, 35-001 Rzeszow, PL.
Mark Smith: German Street 6, 10000 Berlin, DE.
Martyna Kowalska: World Street 5, 00-123 Warsaw, PL.
```

As you can see, the collection is automatically sorted by names, which are used as keys for the sorted list. However, you need to remember that keys must be unique, so you cannot add more than one person with the same full name in this example.

Linked lists

While using the `List` generic class, you can easily get access to particular elements of the collection using indices. However, when you get a single element, how can you move to the next element of the collection? Is it possible? To do so, you may consider the `IndexOf` method to get an index of the element. Unfortunately, it returns an index of the first occurrence of a given value in the collection, so it will not always work as expected in this scenario. Fortunately, **linked lists** exist and can help you with this problem! In this section, you will learn about a few variants.

Singly linked lists

A **singly linked list** is a data structure in which **each list element contains a pointer to the next element**. Thus, you can easily **move from any element to the next one, but you cannot go back**. Of course, the last element in the list has an empty pointer to the next element because there is nothing more located in the list.

IMAGINE A SINGLY LINKED LIST

If you want to better visualize a singly linked list, think about how to represent the phases of human development. Life after birth consists of the neonatal period, infancy, post-infanthood, preschool, school, adolescence, adulthood, and old age. From each phase, you can only go to the next one and you can never go back, even if you try very, very hard. It's similar to a linked list, where you can easily move from a given item to the next item, but you don't have any data to return to the

item that brought you here. But it would be nice to be able to go back a dozen or so years and repeat some phase of development, right? Unfortunately, there is no "back" button here. :-)

Here's an example of a singly linked list:

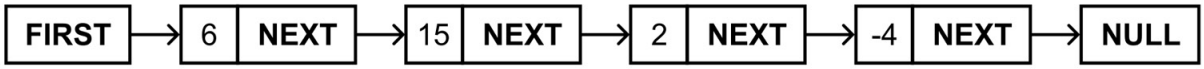

Figure 4.2 – Illustration of a singly linked list

Is it possible to further expand this data structure so that you can both go forward and backward from a given list element? Of course! Let's take a look.

Doubly linked lists

A **doubly linked list** is another data structure that **allows you to navigate both forward and backward from each list item**. It can be created based on the singly linked list by adding a second pointer, namely to the previous element.

IMAGINE A DOUBLY LINKED LIST

If you want to better imagine a doubly linked list, open a text editor and start describing your day in it. Whenever you make a mistake, you press the "undo" button and you see the earlier version. You can also press "redo" and suddenly, you see what was in the document just before you undone the changes. Of course, you can perform such an operation many times, and the system remembers many previous and next operations. This is how you can think of a doubly linked list. In each element of the list, you can easily go to both the next element (equivalent to a "redo" operation) and the previous element (equivalent to a "back" operation). Just look how easy it is to find applications for various data structures in everyday life!

The following figure illustrates a doubly linked list:

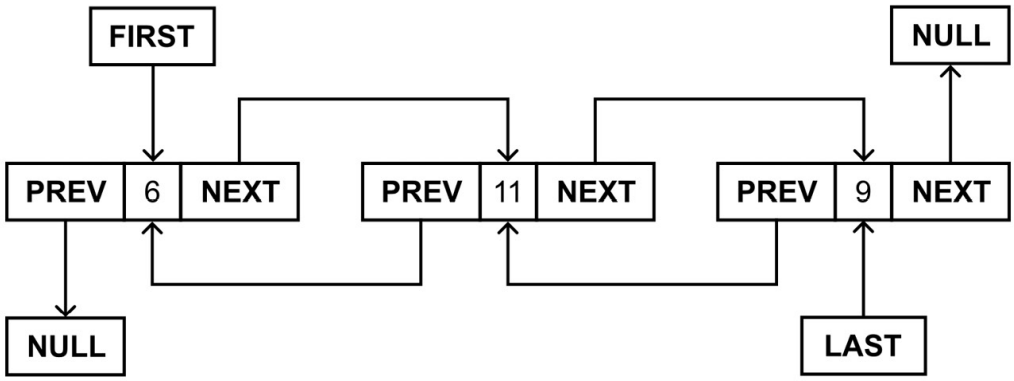

Figure 4.3 – Illustration of a doubly linked list

As you can see, the **FIRST** box indicates the first element in the list. Each item has two properties that point to the previous and next element (**PREV** and **NEXT**, respectively). If there is no previous element, the **PREV** property is equal to **null**. Similarly, when there is no next element, the **NEXT** property is set to **null**. Moreover, the doubly linked list contains the **LAST** box that indicates the last element.

Do you need to implement such a data structure on your own if you want to use it in your C#-based applications? Fortunately, no! It is already available as the **LinkedList** generic class in the **System.Collections.Generic** namespace. While creating an instance of this class, you need to specify the type parameter that indicates a type of a value stored in each element in the list, such as **int** or **string**. Each element (also referred to as a *node*) is represented by an instance of the **LinkedListNode** generic class, such as **LinkedListNode<int>** or **LinkedListNode<string>**.

Some additional explanation is necessary for the methods of adding new nodes to the doubly linked list. For this purpose, you can use a set of methods:

- **AddFirst** adds an element at the beginning of the list
- **AddLast** adds an element at the end of the list
- **AddBefore** adds an element before the specified node in the list
- **AddAfter** adds an element after the specified node in the list

All these methods return an instance of the **LinkedListNode** class. Moreover, there are some other methods:

- **Contains** checks whether the specified value exists in the list
- **Remove** removes a node from the list
- **Clear** removes all elements from the list

After this short introduction, let's take a look at an example that shows how to apply the doubly linked list, implemented as the **LinkedList** class, in practice.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a linked list at https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1.

Example – book reader

As an example, you will prepare a simple application that allows a user to read a book by changing pages. The user should be able to move to the next page (if it exists) after pressing the *N* key, and go back to the previous page (if it exists) after pressing the *P* key. The content of the current page, together with the page number, should be shown in the console, as presented in the screenshot:

```
                                        - 3 -

As a developer, you have certainly stored various collections within your applications, su
ch as data of users, books, and logs. One of the natural ways of storing such data is by u
sing arrays. However, have you ever thought about their variants? For example, have you he
ard about jagged arrays? In this chapter, you will see arrays in action, together with exa
mples and detailed descriptions.

Quote from "C# Data Structures and Algorithms"
by Marcin Jamro, published by Packt in 2024.

< PREV [P]                                                                  [N] NEXT >
```

Figure 4.4 – Screenshot of the book reader example

Let's start with a declaration of the **`Page`** record, as shown in the following code:

```
public record Page(string Content);
```

This record represents a single page and contains the **`Content`** property. Then, you create a few instances of the **`Page`** class, representing six pages of the book:

```
Page p1 = new("Welcome to (...)");
Page p2 = new("While reading (...)");
Page p3 = new("As a developer (...)");
Page p4 = new("In the previous (...)");
Page p5 = new("So far, you (...)");
Page p6 = new("The current (...)");
```

When the instances have been created, you can construct the doubly linked list using a few addition-related methods, as shown in the following lines of code:

```
LinkedList<Page> pages = new();
pages.AddLast(p2);
LinkedListNode<Page> n4 = pages.AddLast(p4);
pages.AddLast(p6);
pages.AddFirst(p1);
pages.AddBefore(n4, p3);
pages.AddAfter(n4, p5);
```

In the first line, a new empty list is created. Then, the given operations are performed:

1. Add the second page at the end (**`[2]`**).
2. Add the fourth page at the end (**`[2, 4]`**).
3. Add the sixth page at the end (**`[2, 4, 6]`**).
4. Add the first page at the beginning of the list (**`[1, 2, 4, 6]`**).
5. Add the third page before the fourth page (**`[1, 2, 3, 4, 6]`**).
6. Add the fifth page after the fourth page (**`[1, 2, 3, 4, 5, 6]`**).

The next part of the code is responsible for presenting a page in the console, as well as for navigating between pages after pressing the appropriate keys. The code is as follows:

```
LinkedListNode<Page> c = pages.First!;
int number = 1;
while (c != null)
{
    Console.Clear();
    string page = $"- {number} -";
    int spaces = (90 - page.Length) / 2;
    Console.WriteLine(page.PadLeft(spaces + page.Length));
    Console.WriteLine();
    string content = c.Value.Content;
    for (int i = 0; i < content.Length; i += 90)
    {
        string line = content[i..];
        line = line.Length > 90 ? line[..90] : line;
        Console.WriteLine(line.Trim());
    }
    Console.WriteLine($"\nQuote from (...)");
    Console.Write(c.Previous != null
        ? "< PREV [P]" : GetSpaces(14));
    Console.Write(c.Next != null
        ? "[N] NEXT >".PadLeft(76) : string.Empty);
    Console.WriteLine();
    ConsoleKey key = Console.ReadKey(true).Key;
    if (key == ConsoleKey.N && c.Next != null)
    {
        c = c.Next;
        number++;
    }
    else if (key == ConsoleKey.P && c.Previous != null)
    {
        c = c.Previous;
        number--;
    }
}
```

In the first line, the value of the `c` variable is set to the first node in the doubly linked list. Generally speaking, the `c` variable represents the page that is currently presented in the console. Then, the initial value for the page number is set to `1` (the `number` variable). However, the most interesting and complicated part of the code is shown in the `while` loop.

Within the loop, the current content of the console is cleared and the string for presenting the page number is properly formatted to display. Before and after it, the `-` characters are added. Moreover, leading spaces are inserted (using the `PadLeft` method) to prepare the string that is centered horizontally.

Then, the content of the page is divided into lines of no more than `90` characters and written in the console. To divide a string, the `Length` property and the **range operator** are used, such as in `content[i..]`. Similarly, additional information is presented in the console. Then, `PREV` and `NEXT` captions are shown, if a previous or a next page is available.

CAN YOU IMPROVE THIS EXAMPLE?

This example divides the text into a few lines while not taking spaces into account. I encourage you to modify the code so that it supports more user-friendly text wrapping. Good luck!

In the following part of the code, the program waits until the user presses any key and does not present it in the console (by passing **`true`** as a parameter of **`ReadKey`**). When the user presses *N*, the **`c`** variable is set to the next node, using the **`Next`** property. Of course, the operation should not be performed when the next page is unavailable. The *P* key is handled similarly, which causes the user to be navigated to the previous page. It is worth mentioning that the number of the page (the **`number`** variable) is modified alongside changing the value of the **`c`** variable.

Finally, the code of the auxiliary **`GetSpaces`** method is shown:

```
string GetSpaces(int number) => string.Join(
    null, Enumerable.Range(0, number).Select(n => " "));
```

This prepares and returns the **`string`** variable with the specified number of spaces. Of course, there are several ways to perform this task. However, in this book, I wanted to show you various approaches, even those that are not so typical. The aim is to show you various ways of achieving your goal and making your horizons as broad as possible.

With this, you should be ready to continue your adventure regarding lists. In the next section, you'll learn about circular lists and their two subtypes.

Circular singly linked lists

In the previous two sections, you learned about linked lists. As you should remember, in a singly linked list, you can navigate between the nodes using the **`Next`** property. However, the **`Next`** property of the last node is set to **`null`**. Do you know that you can easily expand this approach to create a **circular singly linked list**, where **the last node points to the first element, creating a list that can be iterated endlessly?**

IMAGINE A CIRCULAR SINGLY LINKED LIST

If you want to better imagine a circular singly linked list, think for a moment about a screensaver showing photos from a specific folder. After a certain period of inactivity, your screen starts showing photos, one after the other. When the last photo is displayed, the first one from the catalog is shown automatically. Of course, you can't control the photos yourself, because any interaction with the keyboard or mouse turns off the screensaver. A circular singly linked list works similarly. Here, only information about the next list element is saved, without the possibility of going back. The last element of the list takes you to the very beginning. It's so easy once you can imagine a real-life case, right? Now, move your mouse to make the screensaver disappear, and get back to learning more about data structures and algorithms!

The following figure illustrates a circular singly linked list:

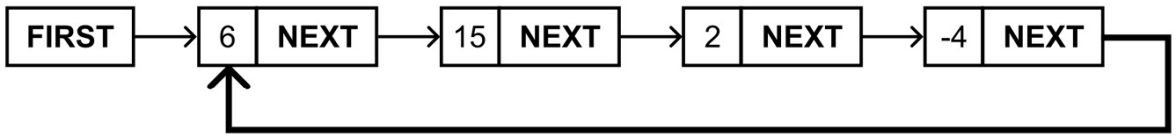

Figure 4.5 – Illustration of a circular singly linked list

After this short introduction to the topic of circular singly linked lists, it is time to take a look at the implementation code. As this data structure is not available by default while developing in C#, you will learn how to implement it on your own, based on a linked list. Let's start with the following code snippet:

```
using System.Collections;
public class CircularLinkedList<T>
    : LinkedList<T>
{
    public new IEnumerator GetEnumerator() =>
        new CircularEnumerator<T>(this);
}
```

The implementation can be created as a generic class that extends **`LinkedList`**, as shown in the preceding code. It is worth mentioning the implementation of the **`GetEnumerator`** method, which uses the **`CircularEnumerator`** class. By creating it, you will be able to endlessly iterate through all the elements of a circular linked list using a **`foreach`** loop. The code of **`CircularEnumerator`** is as follows:

```
public class CircularEnumerator<T>(LinkedList<T> list)
    : IEnumerator<T>
{
    private LinkedListNode<T>? _current = null;
    public T Current => _current != null
        ? _current.Value
        : default!;
    object IEnumerator.Current => Current!;
    public bool MoveNext()
    {
        if (_current == null)
        {
            _current = list?.First;
            return _current != null;
        }
        else
        {
            _current = _current.Next
                ?? _current!.List?.First;
            return true;
        }
    }
    public void Reset()
    {
        _current = null;
    }
    public void Dispose() { }
}
```

The `CircularEnumerator` class implements the `IEnumerator` interface. This class declares the `private` field, which represents the current node (`_current`) in the iteration over the list. It also contains two properties, namely `Current` and `IEnumerator.Current`, which are required by the `IEnumerator` interface.

One of the most important parts of the code is the `MoveNext` method. This checks whether the current element is equal to `null`. If so, it tries to get the first element from the list and starts iterating from it. If it does not exist, the method returns `false` since there are no items in the list. If the current element is not equal to `null`, it changes the current element to the next one or the first node in the list, if the next node is unavailable. In the `Reset` method, you just set a value of the `_current` field to `null`.

Finally, you create the `Next` extension method that navigates to the first element while trying to get the next element from the last item in the list. To simplify the implementation, such a feature will be available as a method, instead of the `Next` property. The code is shown here:

```
public static class CircularLinkedListExtensions
{
    public static LinkedListNode<T>? Next<T>(
        this LinkedListNode<T> n)
    {
        return n != null && n.List != null
            ? n.Next ?? n.List.First
            : null;
    }
}
```

The method checks whether the node exists and whether the list is available. In such a case, it returns a value of the `Next` property of the node (if such a value is not equal to `null`) or returns a reference to the first element in the list using the `First` property.

That's all! You've just completed the C#-based implementation of a circular singly linked list that you can use in various applications. But how? Let's take a look at the following example, which uses this data structure.

Example – spin the wheel

This example simulates a game in which a user spins a wheel at a random speed. The wheel rotates slower and slower until it stops. Then, the user can spin it again, from the previous stop position, as shown in the following figure:

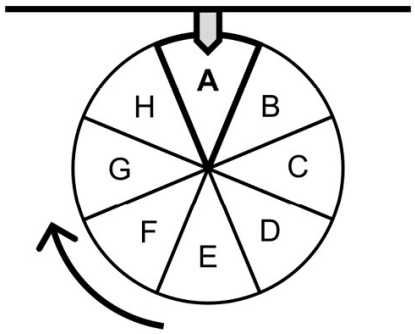

Figure 4.6 – Illustration of the spin the wheel example

Let's proceed to the first part of the code:

```
CircularLinkedList<string> categories = new();
categories.AddLast("Sport");
categories.AddLast("Culture");
categories.AddLast("History");
categories.AddLast("Geography");
categories.AddLast("People");
categories.AddLast("Technology");
categories.AddLast("Nature");
categories.AddLast("Science");
```

First, a new instance of the **`CircularLinkedList`** class is created, which represents a circular singly linked list with **`string`** elements. Then, eight values are added, namely **`Sport`**, **`Culture`**, **`History`**, **`Geography`**, **`People`**, **`Technology`**, **`Nature`**, and **`Science`**.

The following part of the code performs the most important operations:

```
bool isStopped = true;
Random random = new();
DateTime targetTime = DateTime.Now;
int ms = 0;
foreach (string category in categories)
{
    if (isStopped)
    {
        Console.WriteLine("Press [Enter] to start.");
        ConsoleKey key = Console.ReadKey().Key;
        if (key == ConsoleKey.Enter)
        {
            ms = random.Next(1000, 5000);
            targetTime = DateTime.Now.AddMilliseconds(ms);
            isStopped = false;
            Console.WriteLine(category);
        }
        else { return; }
    }
    else
    {
        int remaining = (int)(targetTime
            - DateTime.Now).TotalMilliseconds;
        int waiting = Math.Max(100, (ms - remaining) / 5);
        await Task.Delay(waiting);
```

```
            if (DateTime.Now >= targetTime)
            {
                Console.ForegroundColor = ConsoleColor.Red;
                isStopped = true;
            }
            Console.WriteLine(category);
            Console.ResetColor();
        }
    }
```

First, a few variables are declared:

- **`isStopped`**, which indicates whether the wheel is currently stopped
- **`random`**, for drawing random values of wheel spin in milliseconds
- **`targetTime`**, which is the target time when the wheel should stop
- **`ms`**, which is the last drawn number of milliseconds for wheel-spinning

Then, the **`foreach`** loop is used to iterate through all the elements within a circular singly linked list. If there are no **`break`** or **`return`** instructions within such a loop, it will execute indefinitely due to the nature of a circular linked list. If the last item is reached, the first element in the list is taken automatically in the next iteration.

In this loop, you check whether the wheel is currently stopped or has not been started yet. If so, the message is presented to the user and the program waits until the *Enter* key is pressed. In such a situation, the new spinning operation is configured by drawing the total time of spinning, setting the expected stop time, indicating that the wheel is not stopped, as well as writing the current category. When the user presses any other key, the program stops its execution.

If the wheel is currently not stopped, you calculate the remaining number of milliseconds and the waiting time. This formula makes it possible to provide smaller times at the beginning (the wheel spins faster) and bigger times at the end (the wheel spins slower). Then, the program waits for the specified number of milliseconds.

At the end, you check whether the target time is reached. If so, the foreground color is changed to red and you indicate that the wheel is stopped. Then, the currently chosen category on the spinning wheel is presented in the console.

When you run the application, you will get the result similar to the following one:

```
Press [Enter] to start.
Sport
Culture
History
Geography
People
Technology
Nature
Science
Sport
Culture
History
Geography
People
Technology
Nature
Science
Press [Enter] to start.
```

Figure 4.7 – Screenshot of the spin the wheel example

With that, we've looked at an example that uses a circular singly linked list. Are you curious whether you can expand it further to create a circular doubly linked list?

Circular doubly linked lists

The last data structure we'll cover in this chapter is named the **circular doubly linked list**. It is similar to the circular singly linked list but **allows you not only to iterate indefinitely in the forward direction but also in the backward direction**. You can achieve this by adding pointers to previous elements for each item in the list. Of course, you also need to point to the last element in the list as the previous element of the first one in the list.

IMAGINE A CIRCULAR DOUBLY LINKED LIST

If you want to better visualize a circular doubly linked list, grab your camera and start browsing the gallery of photos you've taken. Here, you can easily go from the first photo to the last one by clicking "back." You can also go from the last photo to the first one by clicking "next." Of course, you can also switch between subsequent photos in the photo gallery by clicking "back" and "next." There is no issue with you taking another photo, at which point it will be added to the collection of photos you've already taken. Take a look for yourself! This is how a circular doubly linked list works. Snap, photo taken, and... let's move on!

A circular doubly linked list is presented in the following diagram:

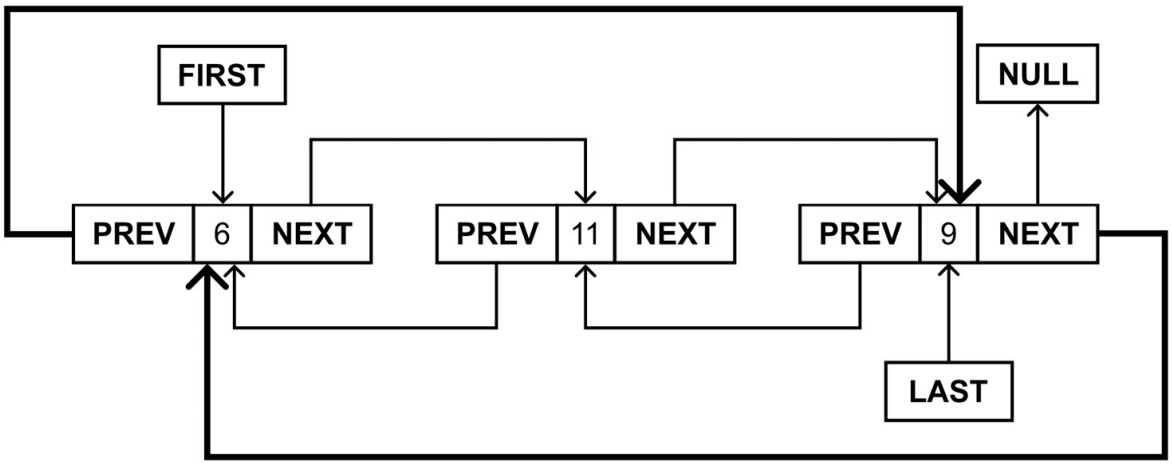

Figure 4.8 – Illustration of a circular doubly linked list

Here, the **PREV** property of the first node navigates to the last one, while the **NEXT** property of the last node navigates to the first. This data structure can be useful in some specific cases, as you will see while developing a real-world example.

After this short introduction to the topic of circular doubly linked lists, it is time to take a look at the implementation code. If you use the code that's already been prepared for the circular singly linked list, you only need to add one extension method, as shown here:

```
public static class CircularLinkedListExtensions
{
    public static LinkedListNode<T>? Next<T>(
        this LinkedListNode<T> n)
    {
        return n != null && n.List != null
            ? n.Next ?? n.List.First
            : null;
    }
    public static LinkedListNode<T>? Prev<T>(
        this LinkedListNode<T> n)
    {
        return n != null && n.List != null
            ? n.Previous ?? n.List.Last
            : null;
    }
}
```

The **Prev** method checks whether the node exists and whether the list is available. In such a case, it returns a value of the **Previous** property of the node (if such a value is not equal to **null**) or returns a reference to the last element in the list using the **Last** property. That's all! Let's take a look at the example.

Example – art gallery

This example is a viewer of drawings presented in the console. Does this sound strange? It could be, but let's try to create some console-based art!

REAL ART IN THE CONSOLE EXISTS!

The topic of creating console-based graphics is quite popular and some amazing art has already been created by various authors! If you are curious about this topic, just search for ASCII arts in your web browser. Will you join this community with your drawings? If so, please share them with me as well!

When a user presses *the right* or *left arrow*, the drawing is changed to the next or the previous one, respectively. As a result, the following art can be viewed in the console:

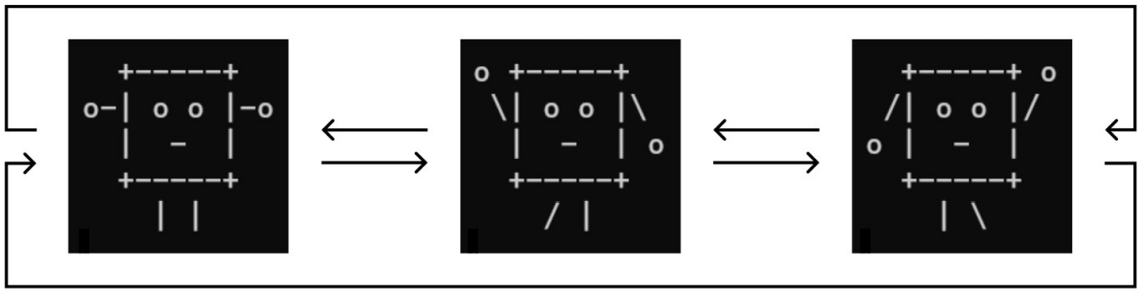

Figure 4.9 – Screenshots of the art gallery example

The code uses the **`CircularLinkedList`** class, as shown here:

```
string[][] arts = GetArts();
CircularLinkedList<string[]> images = new();
foreach (string[] art in arts) { images.AddLast(art); }
LinkedListNode<string[]> node = images.First!;
ConsoleKey key = ConsoleKey.Spacebar;
do
{
    if (key == ConsoleKey.RightArrow)
    {
        node = node.Next()!;
    }
    else if (key == ConsoleKey.LeftArrow)
    {
        node = node.Prev()!;
    }
    Console.Clear();
    foreach (string line in node.Value)
    {
        Console.WriteLine(line);
    }
}
while ((key = Console.ReadKey().Key) != ConsoleKey.Escape);
```

You create a circular doubly linked list consisting of a few elements. Each stores an array of strings. Such an array represents a particular image, namely the following rows forming the art. When you populate the list with data of all images, you store a reference to the first image as **`node`**. Then, you

use a **`do-while`** loop that is executed until the *Escape* button is pressed. If the user presses the right arrow, you update the **`node`** variable using the **`Next`** method. If the left arrow is pressed, the **`Prev`** method is used instead. In each iteration, you clear the console and print the art so that you can receive a simple animation of a dancing figure.

If you are curious how such images are defined, take a look at the following code:

```
string[][] GetArts() => [
    [
        "  +-----+  ",
        "o-| o o |-o",
        "  |  -  |  ",
        "  +-----+  ",
        "    | |    "
    ],
    [
        "o +-----+  ",
        " \\| o o |\\ ",
        "  |  -  | o",
        "  +-----+  ",
        "    / |    "
    ],
    [
        "  +-----+ o",
        " /| o o |/ ",
        "o |  -  |  ",
        "  +-----+  ",
        "    | \\    "
    ]
];
```

With that, you've learned how to use a circular doubly linked list. In the final section of this chapter, we'll learn about three list-related interfaces.

List-related interfaces

While developing applications in C#, you frequently use various collections, including lists. For this reason, it is worth mentioning three common interfaces:

- **`IEnumerable`**
- **`ICollection`**
- **`IList`**

The order of them is important because **`IEnumerable`** is the base interface for **`ICollection`** and **`IList`**, while **`ICollection`** is the base interface for **`IList`**. However, what is inside such interfaces? Let's take a look!

`IEnumerable` only provides you with a **simple iteration over a collection**. For this reason, it exposes the **`GetEnumerator`** method.

The `ICollection` interface adds the following methods for **manipulating the collection**:

- **`Add`** adds a given item to the collection
- **`Clear`** removes all the items from the collection
- **`Contains`** checks whether a given item exists in the collection
- **`Remove`** removes the first occurrence of a given item from the collection

It also exposes the `Count` and `IsReadOnly` properties, as well as the `CopyTo` method for copying the collection to an array.

The last interface I'll mention here is `IList`. It allows you to **access items within the collection by an index**. Thus, the interface contains the indexer for getting or setting an item at a specified index in the collection, as well as methods:

- **`IndexOf`** returns an index of a given item in the collection
- **`Insert`** inserts a given item at a specified index in the collection
- **`RemoveAt`** removes an item at a specified index in the collection

As an example, do you know that the `LinkedList` generic class implements both generic and non-generic variants of the `ICollection` and `IEnumerable` interfaces? I strongly encourage you to take a look at other collections to see what interfaces are implemented by them. You can see this by clicking on the collection name in your code (such as `List` or `ArrayList`) and choosing the **Go To Definition** option from the context menu or simply by pressing *F12*.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding the mentioned interfaces at: https://learn.microsoft.com/en-us/dotnet/api/system.collections.ienumerable, https://learn.microsoft.com/en-us/dotnet/api/system.collections.icollection, and https://learn.microsoft.com/en-us/dotnet/api/system.collections.ilist.

Next, I'll summarize this chapter.

Summary

This chapter was dedicated to **lists**, which are among the most common data structures that are used while developing various kinds of applications. However, this topic is not very easy because there are various variants of lists, including simple, sorted, and linked ones. Even this structure can be further divided as you saw while reading this chapter.

First, you learned about **simple lists**, which can be implemented as an **array list** or as a **generic list**. One of the most important differences between them is that the array list is not strongly typed, while

the generic list is. You learned about various properties and methods available for these data structures, together with extension methods. Such information was supported by code snippets.

Next, you learned about **sorted lists**, which ensures a proper order of elements available in the collection. It is a bit of a different data structure but can be useful in various development scenarios. You learned how to use it while creating an address book, which is always sorted by names.

Finally, you took a closer look at linked lists, starting with **singly linked lists** and **doubly linked lists**. They allow you to navigate between elements, either in only one direction or in both directions. Such a data structure can be easily extended to a **circular linked list**, either in singly or doubly linked variants. Therefore, you can benefit from the features of suitable structures without the significant development effort.

The available types of data structures can sound quite complicated. However, in this chapter, you saw detailed descriptions of particular data structures, together with illustrations and C#-based implementations. These should have helped simplify things for you and can be used as a basis for your future projects.

Are you ready to learn other data structures? If so, proceed to the next chapter and read about **stacks** and **queues**!

5

Stacks and Queues

So far, you learned a lot about arrays and lists. However, these structures are not the only ones available. Among others, there is also a group of more specialized data structures called **limited access data structures**.

What does this mean? To explain the name, let's return to the topic of arrays for the moment, which belong to the group of **random access data structures**. The difference between them is only one word - that is, *limited* or *random*. As you already know, arrays allow you to store data and get access to various elements using indices. Thus, you can easily get the first, the middle, the nth, or the last element from an array. For this reason, it can be named a random access data structure.

However, what does *limited* mean? The answer is very simple. With a limited access data structure, **you cannot access every element from the structure**. Thus, **the way of getting elements is strictly specified**. For example, you can get only the first or the last element, but you cannot get the nth element from the data structure. Popular representatives of limited access data structures are stacks and queues, which are topics mentioned in this chapter.

You will see the application of a stack, as well as a few variants of queues, including a regular one, a priority queue, and a circular queue. To make understanding easier, the text is supported with illustrations and code snippets with detailed explanations.

In this chapter, the following topics will be covered:

- Stacks
- Queues
- Priority queues
- Circular queues

Stacks

To begin, let's talk about a **stack**. It is a data structure that allows you to **add a new element only at the top** (referred to as a **push** operation) and **to get an element only by removing it from the top** (a **pop** operation). For this reason, a stack is consistent with the **LIFO** principle, which stands for **Last-In First-Out**.

IMAGINE A STACK

If you want to better visualize a stack, let's close the book for a moment, go to the kitchen, and take a look at a pile of plates, each placed on top of the other. You can only add a new plate to the top of the pile, and you can only get a plate from the top of the pile. You cannot remove the seventh plate without taking the previous six from the top, and you cannot add a plate to the middle of the pile. So, the last added plate (last-in) will be removed from the pile first (first-out). And do not even try to get a plate from the middle of the pile, as you don't want to break plates! The stack operates similarly. It allows you to add a new element only at the top (a push operation) and to get an element only by removing it from the top (a pop operation).

A diagram of a stack with *push* and *pop* operations is shown as follows:

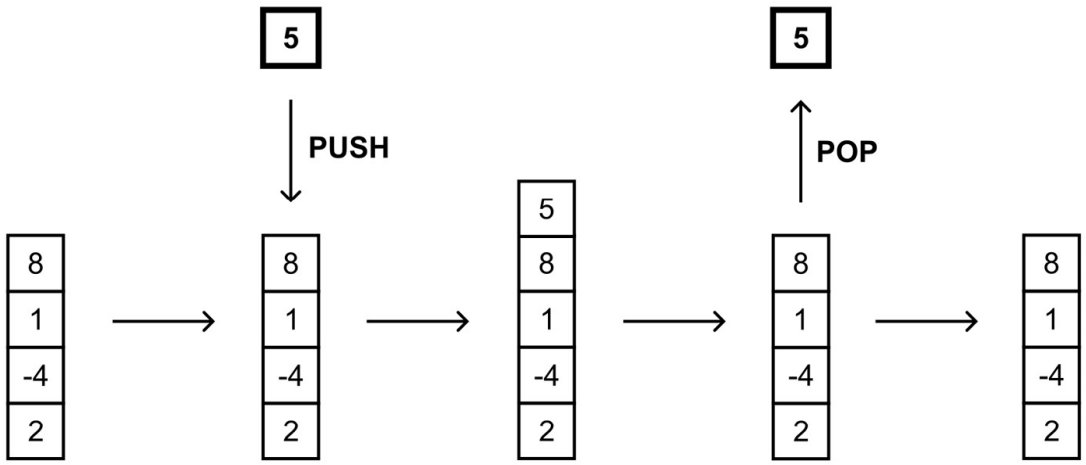

Figure 5.1 – Illustration of a stack

It seems to be very easy, doesn't it? It really is, and you can benefit from various features of stacks using the built-in generic `Stack` class. It is worth remembering that it is located in the `System.Collections.Generic` namespace.

Let's mention three methods from this class:

- `Push` inserts an element at the top of the stack
- `Pop` removes an element from the top of the stack and returns it
- `Peek` returns an element from the top of the stack without removing it

You also have access to other methods, such as for removing all elements from the stack (`Clear`) or for checking whether a given element is available in the stack (`Contains`). You can get the number of elements currently in the stack using the `Count` property.

WHAT ABOUT THE PERFORMANCE?

It is worth noting that the `Push` method is either an O(1) operation, if the capacity does not need to increase, or O(n) otherwise, where n is the number of elements in the stack. Both `Pop` and `Peek` are O(1) operations.

As the time complexity looks very promising, it is high time to take a look at some examples showing stacks in action.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a stack at https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1

Example – reversing a word

For the first example, let's try to reverse a word using a stack. You can do this by iterating through characters that form a string, adding each at the top of the stack, and then removing all elements from the stack. At the end, you receive the reversed word, as shown in the following diagram presenting how to reverse **MARCIN**:

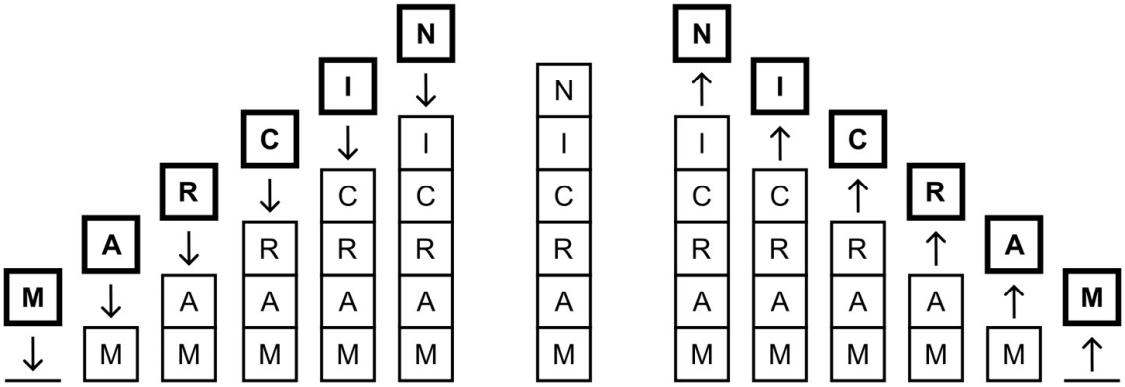

Figure 5.2 – Illustration of the reversing a word example

The implementation is shown in the following code snippet:

```
string text = "MARCIN";
Stack<char> chars = new();
foreach (char c in text) { chars.Push(c); }
while (chars.Count > 0) { Console.Write(chars.Pop()); }
```

Here, a new instance of the **Stack** class is created. In this scenario, the stack can contain only **char** elements. Then, you iterate through all characters using a **foreach** loop and insert each character at the top of the stack by calling the **Push** method. The remaining part of the code consists of a **while** loop, which is executed until the stack is empty. This condition is checked using the **Count** property. In each iteration, the top element is removed from the stack (by calling **Pop**) and written in the console (using the **Write** static method of the **Console** class).

After running the code, you will receive the following result:

```
NICRAM
```

Example – Tower of Hanoi

The next example is a significantly more complex application of stacks. It is related to the mathematical game *Tower of Hanoi*. The game requires three rods, onto which you can put discs. Each disc has a different size. At the beginning, all discs are placed on the first rod, forming a stack, ordered from the smallest (at the top) to the biggest (at the bottom). It is presented in the following diagram (on the left):

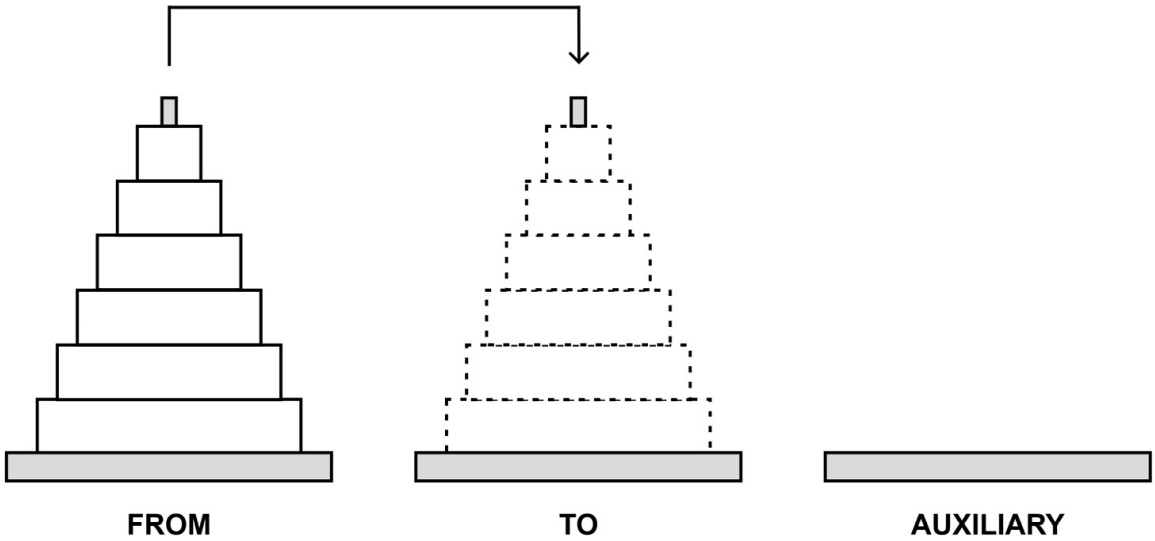

Figure 5.3 – Illustration of the Tower of Hanoi example

The aim of the game is to **move all the discs from the first rod** (`FROM`) **to the second one** (`TO`). However, during the whole game, you **cannot place a bigger disc on a smaller one**. Moreover, you **can only move one disc at a time**, and, of course, you **can only take a disc from the top of any rod**.

How could you move discs between the rods to comply with the aforementioned rules? The problem can be divided into sub-problems:

- **Moving one disc**: Such a case is trivial, and you just need to move a disc from `FROM` to `TO`, without using the `AUXILIARY` rod.
- **Moving two discs**: You move one disc from `FROM` to `AUXILIARY`. Then, you move the remaining disc from `FROM` to `TO`. At the end, you move a disc from `AUXILIARY` to `TO`.
- **Moving three discs**: You start by moving two discs from `FROM` to `AUXILIARY`, using the mechanism described earlier. The operation involves `TO` as the auxiliary rod. Then, you move the remaining disc from `FROM` to `TO`, and then move two discs from `AUXILIARY` to `TO`, using `FROM` as the auxiliary rod.

As you can see, you can solve the problem of **moving n discs** by moving *n-1* discs from `FROM` to `AUXILIARY`, using `TO` as the auxiliary rod. Then, you should move the remaining disc from `FROM` to `TO`.

At the end, you just need to move *n-1* discs from **AUXILIARY** to the **TO** rod, using **FROM** as the auxiliary rod.

Now that you know the basic rules, let's proceed to the code. First, let's focus on the **Game** class, which contains the logic related to the game:

```
public class Game
{
    public Stack<int> From { get; private set; }
    public Stack<int> To { get; private set; }
    public Stack<int> Auxiliary { get; private set; }
    public int DiscsCount { get; private set; }
    public int MovesCount { get; private set; }
    public event EventHandler<EventArgs>? MoveCompleted;
}
```

The class contains five properties, representing the following:

- Three rods (**From**, **To**, **Auxiliary**)
- The overall number of discs (**DiscsCount**)
- The number of performed moves (**MovesCount**)

The **MoveCompleted** event is declared as well. It is fired after each move to inform that the user interface should be refreshed. Therefore, you can show the proper content, illustrating the current state of the rods.

Apart from the properties and the event, the class also has the following constructor:

```
public Game(int discsCount)
{
    DiscsCount = discsCount;
    From = new Stack<int>();
    To = new Stack<int>();
    Auxiliary = new Stack<int>();
    for (int i = 0; i < discsCount; i++)
    {
        int size = discsCount - i;
        From.Push(size);
    }
}
```

The constructor takes only one parameter, namely the number of discs (**discsCount**), and sets it as a value of the **DiscsCount** property. Then, new instances of the **Stack** class are created, and references to them are stored in the **From**, **To**, and **Auxiliary** properties. At the end, a **for** loop is used to create the necessary number of discs and to add elements to the first stack (**From**), using the **Push** method.

It is worth noting that **From**, **To**, and **Auxiliary** stacks only store integer values (**Stack<int>**). Each integer value represents the size of a particular disc. Such data is crucial due to the rules of moving discs between rods.

One of the most interesting and important parts of the code is the **`MoveAsync`** recursive method. It takes four parameters, namely the number of discs and references to three stacks. However, what happens in the **`MoveAsync`** method? Let's look inside:

```
public async Task MoveAsync(int discs, Stack<int> from,
    Stack<int> to, Stack<int> auxiliary)
{
    if (discs == 0) { return; }
    await MoveAsync(discs - 1, from, auxiliary, to);
    to.Push(from.Pop());
    MovesCount++;
    MoveCompleted?.Invoke(this, EventArgs.Empty);
    await Task.Delay(250);
    await MoveAsync(discs - 1, auxiliary, to, from);
}
```

As **`MoveAsync`** is called recursively, it is necessary to specify an exit condition to prevent the method from being called infinitely. In this case, the method will not call itself when the value of the **`discs`** parameter is equal to **`0`**.

Otherwise, the **`MoveAsync`** method is called, but the order of stacks is changed. Then, the element is removed from the stack represented by the second parameter (**`from`**), and inserted at the top of the stack represented by the third parameter (**`to`**).

In the following lines, the number of moves (**`MovesCount`**) is incremented and the **`MoveCompleted`** event is fired. It is responsible for refreshing the user interface. Then, the algorithm stops for 250 milliseconds to show the following steps of the operation in a way well visible to a user.

At the end, the **`MoveAsync`** method is called again, with another configuration of rod order. By calling this method several times, the discs will be moved from the first (**`From`**) rod to the second (**`To`**) rod. The operations performed in the **`MoveAsync`** method are consistent with the description of the problem of moving *n* discs between rods, as explained in the introduction to this example.

When the class with the logic regarding the *Tower of Hanoi* game is created, let's see how to create a user interface that allows you to present the following moves of the algorithm. Such a task is accomplished by the **`Visualization`** class:

```
public class Visualization
{
    private readonly Game _game;
    private readonly int _columnSize;
    private readonly char[,] _board;
    public Visualization(Game game)
    {
        _game = game;
        _columnSize = Math.Max(6,
            GetDiscWidth(_game.DiscsCount) + 2);
        _board = new char[_game.DiscsCount,
            _columnSize * 3];
```

```
    }
}
```

It contains three private fields, namely storing a reference to data of the game (**`_game`**), the number of characters to present a single rod (**`_columnSize`**), as well as a two-dimensional array with visualization of all rods, shown in the console (**`_board`**). The constructor takes only one parameter and sets values for all private fields.

Column size is calculated using the **`GetDiscWidth`** auxiliary method:

```
private int GetDiscWidth(int size) => (2 * size) - 1;
```

You draw the current state of all three rods by calling **`Show`**, which is shown next:

```
public void Show(Game game)
{
    Console.Clear();
    if (game.DiscsCount <= 0) { return; }
    FillEmptyBoard();
    FillRodOnBoard(1, game.From);
    FillRodOnBoard(2, game.To);
    FillRodOnBoard(3, game.Auxiliary);
    Console.WriteLine(Center("FROM")
        + Center("TO") + Center("AUXILIARY"));
    DrawBoard();
    Console.WriteLine($"\nMoves: {game.MovesCount}");
    Console.WriteLine($"Discs: {game.DiscsCount}");
}
```

The method clears the current content of the console (by calling the **`Clear`** method). Then, it calls the **`FillEmptyBoard`** and **`FillRodOnBoard`** methods to clear content that should be shown in the console and then fill it with data of the current state of rods, one in each call of **`FillRodOnBoard`**. Next, you show captions for each rod, draw the board, as well as write the number of moves and discs.

To clear the content of the board, you just iterate through all elements in the two-dimensional array and set the value of each item to a space, as shown next:

```
private void FillEmptyBoard()
{
    for (int y = 0; y < _board.GetLength(0); y++)
    {
        for (int x = 0; x < _board.GetLength(1); x++)
        {
            _board[y, x] = ' ';
        }
    }
}
```

If you want to learn how to fill a part of the two-dimensional array that is related to a particular rod, let's take a look at the code of **`FillRodOnBoard`**:

```
private void FillRodOnBoard(int column, Stack<int> stack)
{
    int discsCount = _game.DiscsCount;
```

```
    int margin = _columnSize * (column - 1);
    for (int y = 0; y < stack.Count; y++)
    {
        int size = stack.ElementAt(y);
        int row = discsCount - (stack.Count - y);
        int columnStart = margin + discsCount - size;
        int columnEnd = columnStart + GetDiscWidth(size);
        for (int x = columnStart; x <= columnEnd; x++)
        {
            _board[row, x] = '=';
        }
    }
}
```

First, the left margin is calculated to add data in the correct section within the overall array - that is, within the correct range of columns. The main part of the method is the **`for`** loop, where the number of iterations is equal to the number of discs located in the stack. In each iteration, the size of the current disc is read using the **`ElementAt`** extension method (from the **`System.Linq`** namespace). Next, you calculate an index of a row, where the disc should be shown, as well as start and end indices for columns. Finally, a **`for`** loop is used to insert the equals sign (**`=`**) in proper locations in the array.

One of the auxiliary methods is **`Center`**. It aims to add additional spaces before and after the text, passed as the parameter, to center the text in the column:

```
private string Center(string text)
{
    int margin = (_columnSize - text.Length) / 2;
    return text.PadLeft(margin + text.Length)
        .PadRight(_columnSize);
}
```

The last used method is named **`DrawBoard`**. It simply iterates through all elements in the two-dimensional array and writes content in the console. The code is shown next:

```
private void DrawBoard()
{
    for (int y = 0; y < _board.GetLength(0); y++)
    {
        string line = string.Empty;
        for (int x = 0; x < _board.GetLength(1); x++)
        {
            line += _board[y, x];
        }
        Console.WriteLine(line);
    }
}
```

In the end, let's take a look at the main code, located in the **`Program.cs`** file:

```
Game game = new(10);
Visualization vis = new(game);
game.MoveCompleted += (s, e) => vis.Show((Game)s!);
await game.MoveAsync(game.DiscsCount,
    game.From, game.To, game.Auxiliary);
```

Here, a new instance of the `Game` class is created. The parameter indicates that `10` discs are used. In the next line, you create a new instance of the `Visualization` class responsible for showing the following steps of the game. You also specify that the `Show` method is called when the `MoveCompleted` event is fired. Finally, you call the `MoveAsync` method to start moving discs between rods.

You already added the necessary code to run the *Tower of Hanoi* mathematical game. Let's launch the application and see it in action! Just after starting the program, you see that all discs are located in the first rod (`FROM`). In the next step, the smallest disc is moved from the top of the first rod (`FROM`) to the top of the third rod (`AUXILIARY`), as shown in the following screenshot:

```
        FROM                    TO                AUXILIARY

         ====
        ======
       ========
      ==========
     ============
    ==============
   ================
  ==================
====================                              ==

Moves: 1
Discs: 10
```

Figure 5.4 – The second step in the Tower of Hanoi example

While making many other moves in the program, you can see how discs are moved between all three rods. One of the intermediate steps is as follows:

```
        FROM                    TO                  AUXILIARY




        ====
    ==============
   ================                                 ======
  ==================            ==                 ========
=====================      ============          ==========

Moves: 34
Discs: 10
```

Figure 5.5 – One of the intermediate steps in the Tower of Hanoi example

When the necessary moves are completed, all discs are moved from the first rod (`FROM`) to the second one (`TO`). The final result is presented next:

```
        FROM                    TO                  AUXILIARY
                                ==
                               ====
                              ======
                             ========
                            ==========
                           ============
                          ==============
                         ================
                        ==================
                       ====================

Moves: 1023
Discs: 10
```

Figure 5.6 – Final step in the Tower of Hanoi example

Finally, it is worth mentioning the number of moves necessary to complete the *Tower of Hanoi* game. In the case of 10 discs, the number of moves is 1,023. If you use only 3 discs, the number of moves is 7. Generally speaking, **the number of moves can be calculated with the formula 2^n-1**, where *n* is the number of discs.

That's all! In this section, you learned the first limited access data structure, namely a stack. Now, it is high time that you get to know more about queues.

Queues

A **queue** is a data structure that allows you **to add a new element only at the end of the queue** (referred to as an **enqueue** operation) and **to get an element only from the beginning of the queue** (a **dequeue** operation). For this reason, a queue is consistent with the **FIFO** principle, which stands for **First-In First-Out**.

IMAGINE A QUEUE

If you want to better imagine a queue, let's take a break from learning data structures and algorithms, wear your favorite jacket, and go to a shop in the vicinity. You buy your favorite ice cream, and you see five people waiting for checkout. Oh no... You are the last one, so you need to wait until the first, second, third, fourth, and fifth person pay. These lines in shops can be frustrating! In general, new people stand at the end of the line, and the next person is taken to the checkout from the beginning of the line. No one is allowed to choose a person from the middle and serve them in a different order. The queue data structure operates similarly. You can only add new elements at the end of the queue and remove an element from the beginning of the queue. So, people who come first (first-in) are served at the beginning (first-out).

The operation of a queue is presented in the following diagram:

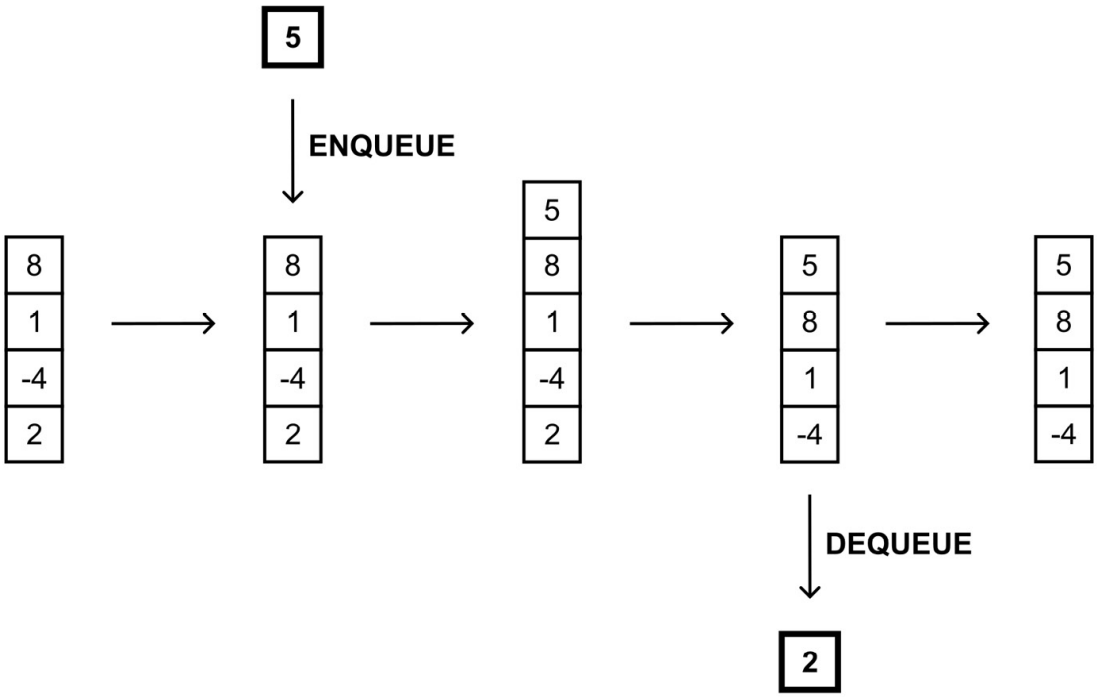

Figure 5.7 – Illustration of a queue

It is worth mentioning that a queue is a **recursive data structure**, similar to a stack. This means that **a queue can be either empty or consists of the first element and the rest of the queue, which also forms a queue**. Let's take a look at the following diagram, where the beginning of the queue is marked with a bold line:

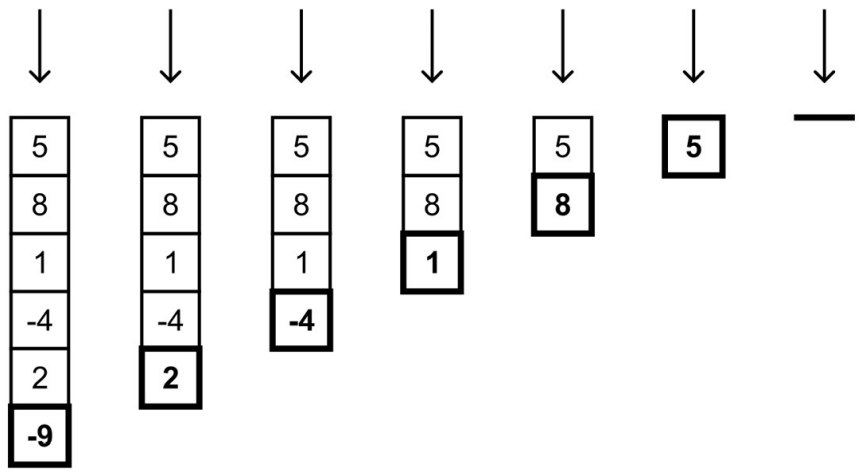

Figure 5.8 – A queue as a recursive data structure

The queue data structure seems to be very easy to understand, as well as being similar to a stack, apart from the way of removing an element. Does this mean that you can also use a built-in class to use a queue in your programs? Fortunately, yes! The available generic class is `Queue` from the `System.Collections.Generic` namespace.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a queue at https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1.

The `Queue` class contains the following set of methods:

- `Enqueue` adds an element at the end of the queue
- `Dequeue` removes an element from the beginning of the queue and returns it
- `Peek` returns an element from the beginning of the queue without removing it
- `Clear` removes all elements from the queue
- `Contains` checks whether the queue contains the given element

The `Queue` class also contains the `Count` property, which returns the total number of elements located in the queue. It can be used to check whether the queue is empty.

WHAT ABOUT THE PERFORMANCE?

*It is worth mentioning that the **`Enqueue`** method is either an O(1) operation, if the internal array does not need to be reallocated, or O(n) otherwise, where n is the number of elements in the queue. Both **`Dequeue`** and **`Peek`** are O(1) operations.*

The great performance results are supported by a very easy application of this data structure, as shown in the following part of the code:

```
List<int> items = [2, -4, 1, 8, 5];
Queue<int> queue = new();
items.ForEach(queue.Enqueue);
while (queue.Count > 0)
{
    Console.WriteLine(queue.Dequeue());
}
```

Here, you create a new list and a queue containing only integer values. Then, you add all elements from the list to the queue, using the **`Enqueue`** method. At the end, you use a **`while`** loop to dequeue all the elements, using the **`Dequeue`** method.

It is worth noting that in the third line, you do not use the lambda expression and simply use the name of the method. Of course, you can use the following form instead:

```
items.ForEach(i => queue.Enqueue(i));
```

The additional comment is necessary for scenarios where you want to use a queue concurrently from many threads. In such a case, it is necessary to choose the thread-safe variant of the queue, which is represented by the **`ConcurrentQueue`** generic class from the **`System.Collections.Concurrent`** namespace. This class contains a set of built-in methods to perform various operations on the queue, such as the following:

- **`Enqueue`** adds an element at the end of the queue
- **`TryDequeue`** tries to remove an element from the beginning and return it
- **`TryPeek`** tries to return an element from the beginning without removing it

Both **`TryDequeue`** and **`TryPeek`** have a parameter with the **`out`** keyword. If the operation is successful, such methods return **`true`**, and the result is returned as a value of the **`out`** parameter. Moreover, the **`ConcurrentQueue`** class also contains two properties, namely **`Count`** to get the number of elements stored in the collection and **`IsEmpty`** to return a value indicating whether the queue is empty.

WHERE CAN YOU FIND MORE INFORMATION?

*You can find content regarding the **`ConcurrentQueue`** class at https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1.*

After this short introduction, let's proceed to two examples representing a queue in the context of a call center, with many clients and one or many consultants.

Example – call center with a single consultant

This first example represents a simple approach to the call center solution, where there are **many clients** (with different identifiers), and **only one consultant**, who answers waiting calls in the same order in which they appear.

This scenario is shown next:

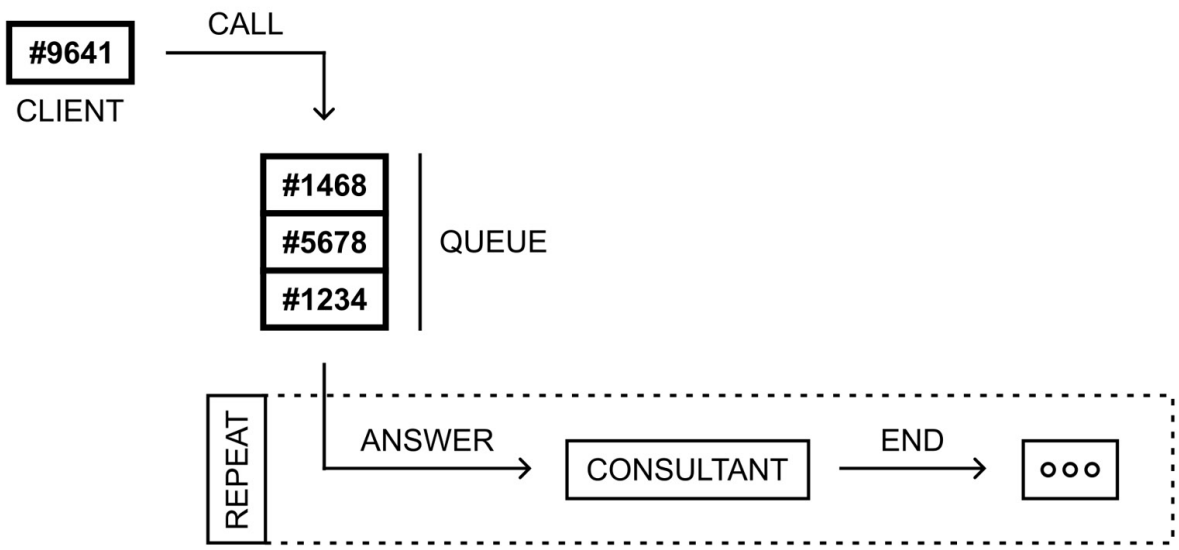

Figure 5.9 – Illustration of the call center with a single consultant example

As you can see in the preceding diagram, four calls are performed by clients. They are added to the queue with waiting phone calls, namely from clients `#1234`, `#5678`, `#1468`, and `#9641`. When a consultant is available, they answer the phone. When the call ends, the consultant can answer the next waiting call. According to this rule, the consultant will talk with clients in the following order: **#1234**, **#5678**, **#1468**, and **#9641**.

Let's take a look at the code of the first class, named `IncomingCall`, which represents a single incoming call performed by a client. Its code is as follows:

```
public class IncomingCall
{
    public int Id { get; set; }
    public int ClientId { get; set; }
    public DateTime CallTime { get; set; }
    public DateTime? AnswerTime { get; set; }
    public DateTime? EndTime { get; set; }
```

```
    public string? Consultant { get; set; }
}
```

The class contains six properties representing a unique identifier of a call (**`Id`**), a client identifier (**`ClientId`**), the date and time when the call was started (**`CallTime`**), when it was answered (**`AnswerTime`**), and when it was ended (**`EndTime`**), as well as the name of the consultant (**`Consultant`**).

The most important part of the code is related to the **`CallCenter`** class, which represents call-related operations. Its fragment is as follows:

```
public class CallCenter
{
    private int _counter = 0;
    public Queue<IncomingCall> Calls { get; private set; }
    public CallCenter() =>
        Calls = new Queue<IncomingCall>();
}
```

The **`CallCenter`** class contains the **`_counter`** field with an identifier of the last call, which is equal to the number of calls so far. The class also has the **`Calls`** property representing a queue (with **`IncomingCall`** instances), where data of waiting calls is stored. In the constructor, a new instance of the **`Queue`** generic class is created, and its reference is assigned to the **`Calls`** property.

Of course, the class contains some methods, such as **`Call`** with the following code:

```
public IncomingCall Call(int clientId)
{
    IncomingCall call = new()
    {
        Id = ++_counter,
        ClientId = clientId,
        CallTime = DateTime.Now
    };
    Calls.Enqueue(call);
    return call;
}
```

Here, you create a new instance of the **`IncomingCall`** class and set values of its properties, namely its identifier (together with pre-incrementing the **`_counter`** field), the client identifier (using the **`clientId`** parameter), and the call time. The created instance is added to the queue by calling the **`Enqueue`** method and returned.

The next method is **`Answer`**. It represents the operation of answering the call from the person waiting in the queue for the longest time. Such a call is represented by the element located at the beginning of the queue. The **`Answer`** method is shown next:

```
public IncomingCall? Answer(string consultant)
{
    if (!AreWaitingCalls()) { return null; }
    IncomingCall call = Calls.Dequeue();
    call.Consultant = consultant;
    call.AnswerTime = DateTime.Now;
```

```
    return call;
}
```

Within this method, you check whether the queue is empty. If so, the method returns **`null`**, which means that there are no phone calls that can be answered by the consultant. Otherwise, the call is removed from the queue (using the **`Dequeue`** method), and its properties are updated by setting the consultant's name (using the **`consultant`** parameter) and answer time (to the current date and time). At the end, the data of the call is returned.

Apart from the **`Call`** and **`Answer`** methods, you also implement the **`End`** method, which is called whenever the consultant ends a call with a particular client. In such a case, you only set the end time, as shown in the following piece of code:

```
public void End(IncomingCall call)
    => call.EndTime = DateTime.Now;
```

The last method in the **`CallCenter`** class is named **`AreWaitingCalls`**. It returns a value indicating whether there are any waiting calls in the queue, using the **`Count`** property of the **`Queue`** class. Its code is as follows:

```
public bool AreWaitingCalls() => Calls.Count > 0;
```

Let's proceed to the **`Program.cs`** file and its code:

```
Random random = new();
CallCenter center = new();
center.Call(1234);
center.Call(5678);
center.Call(1468);
center.Call(9641);
while (center.AreWaitingCalls())
{
    IncomingCall call = center.Answer("Marcin")!;
    Log($"Call #{call.Id} from client #{call.ClientId}
        answered by {call.Consultant}.");
    await Task.Delay(random.Next(1000, 10000));
    center.End(call);
    Log($"Call #{call.Id} from client #{call.ClientId}
        ended by {call.Consultant}.");
}
```

You create a new instance of the **`Random`** class (for getting random numbers), as well as an instance of the **`CallCenter`** class. Then, you simulate making a few calls by clients, namely with the following identifiers: **`1234`**, **`5678`**, **`1468`**, and **`9641`**. The most interesting part of the code is located in the **`while`** loop, which is executed until there are no waiting calls in the queue. Within the loop, the consultant answers the call (using the **`Answer`** method) and a log is generated (using the **`Log`** auxiliary method). Then, you wait for a random number of milliseconds (between **`1000`** and **`10000`**) to simulate the various lengths of a call. When this has elapsed, the call ends (by calling the **`End`** method), and a proper log is generated.

The last part of the code necessary for this example is the **Log** method:

```
void Log(string text) =>
    Console.WriteLine($"[{DateTime.Now:HH:mm:ss}] {text}");
```

When you run the example, you will receive a result similar to the following:

```
[13:10:53] Call #1 from client #1234 answered by Marcin.
[13:10:56] Call #1 from client #1234 ended by Marcin.
[13:10:56] Call #2 from client #5678 answered by Marcin.
[13:10:59] Call #2 from client #5678 ended by Marcin.
[13:10:59] Call #3 from client #1468 answered by Marcin.
[13:11:06] Call #3 from client #1468 ended by Marcin.
[13:11:06] Call #4 from client #9641 answered by Marcin.
[13:11:09] Call #4 from client #9641 ended by Marcin.
```

Congratulations! You just completed the first example regarding a queue data structure. If you want to learn more about the thread-safe version of the queue-related class, let's proceed to the next example.

Example – call center with many consultants

The example shown in the preceding section was intentionally simplified to make understanding a queue much simpler. However, it is high time you make it more related to real-world problems. In this section, you will see how to expand it to support many consultants, as shown in the following diagram:

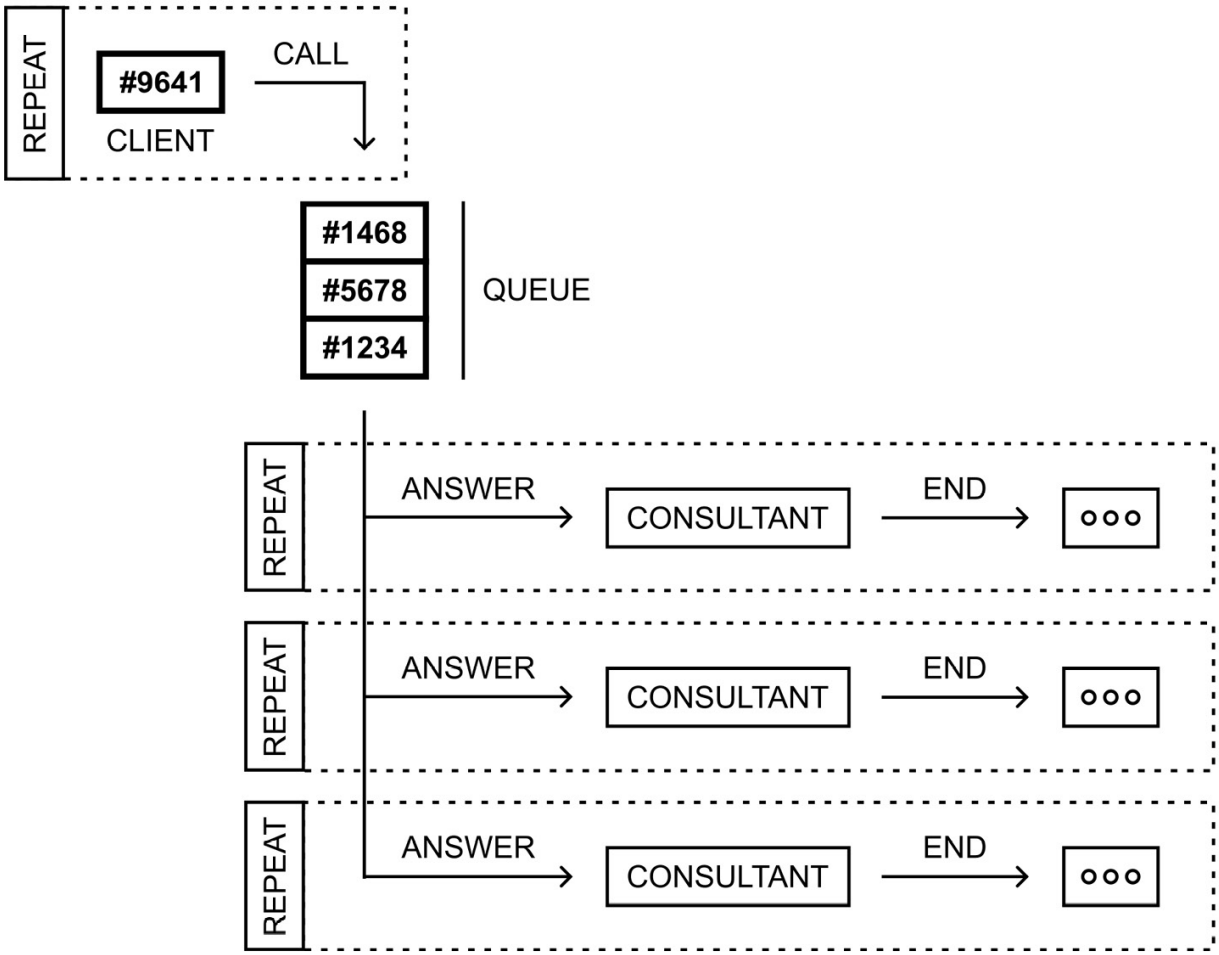

Figure 5.10 – Illustration of the call center with many consultants example

What is important is that both clients and consultants operate at the same time. If there are more incoming calls than available consultants, a new call will be added to the queue and will wait until there is a consultant who can answer the call. If there are too many consultants and few calls, the consultants will wait for a call. To perform this task, you create a few threads, which access the queue. Therefore, you use a thread-safe version of the queue, namely the **`ConcurrentQueue`** class.

Let's take a look at the code! First, you need to declare an **`IncomingCall`** class, the code of which is exactly the same as in the previous example. Various modifications are necessary in the **`CallCenter`** class, as presented next:

```
using System.Collections.Concurrent;
public class CallCenter
{
    private int _counter = 0;
    public ConcurrentQueue<IncomingCall> Calls
        { get; private set; }
    public CallCenter() => Calls =
```

```
        new ConcurrentQueue<IncomingCall>();
}
```

As the **Enqueue** method is available in both the **Queue** and **ConcurrentQueue** classes, no changes are necessary in the **Call** method.

However, the **Dequeue** method does not exist in **ConcurrentQueue**. For this reason, you need to modify the **Answer** method to use the **TryDequeue** method. It returns a value indicating whether the element is removed from the queue. The removed element is returned using the **out** parameter, as shown next:

```
public IncomingCall? Answer(string consultant)
{
    if (!Calls.IsEmpty
        && Calls.TryDequeue(out IncomingCall? call))
    {
        call.Consultant = consultant;
        call.AnswerTime = DateTime.Now;
        return call;
    }
    return null;
}
```

You can also slightly modify the **AreWaitingCalls** method to use the **IsEmpty** property instead of **Count**, presented as follows:

```
public bool AreWaitingCalls() => !Calls.IsEmpty;
```

No further modifications are necessary in the **CallCenter** class. However, more changes are required in the code located in **Program.cs**, as shown next:

```
Random random = new();
CallCenter center = new();
Parallel.Invoke(
  () => Clients(center),
  () => Consultant(center, "Marcin", ConsoleColor.Red),
  () => Consultant(center, "James", ConsoleColor.Yellow),
  () => Consultant(center, "Olivia", ConsoleColor.Green));
```

Here, just after the creation of the **CallCenter** instance, you start execution of four actions, namely representing clients and three consultants, using the **Invoke** static method of the **Parallel** class from the **System.Threading.Tasks** namespace. The lambda expressions are used to specify methods that are called, namely **Clients** for client-related operations and **Consultant** for consultant-related tasks. You also specify additional parameters, such as a name and a color for a given consultant.

The **Clients** method represents operations performed cyclically by many clients. Its code is shown in the following block:

```
void Clients(CallCenter center)
{
    while (true)
    {
        int clientId = random.Next(1, 10000);
```

```
        IncomingCall call = center.Call(clientId);
        Log($"Incoming call #{call.Id}
            from client #{clientId}");
        Log($"Waiting calls in the queue:
            {center.Calls.Count}");
        Thread.Sleep(random.Next(500, 2000));
    }
}
```

Within the **`while`** loop, you get a random number as an identifier of a client (**`clientId`**), and the **`Call`** method is called. The client identifier is logged, together with the number of waiting calls. At the end, the client-related thread is suspended for a random number of milliseconds in the range between 500 ms and 2,000 ms, to simulate the delay between another call made by the next client.

The following method is named **`Consultant`** and is executed on a separate thread for each consultant. The method takes three parameters, namely an instance of **`CallCenter`**, as well as a name and color for the consultant. The code is as follows:

```
void Consultant(CallCenter center, string name,
    ConsoleColor color)
{
    while (true)
    {
        Thread.Sleep(random.Next(500, 1000));
        IncomingCall? call = center.Answer(name);
        if (call == null) { continue; }
        Log($"Call #{call.Id} from client #{call.ClientId}
            answered by {call.Consultant}.", color);
        Thread.Sleep(random.Next(1000, 10000));
        center.End(call);
        Log($"Call #{call.Id} from client #{call.ClientId}
            ended by {call.Consultant}.", color);
    }
}
```

Within the **`while`** loop, the consultant waits for a random period, between 0.5 and 1 second. Then, they try to answer the first waiting call, using the **`Answer`** method. If there are no waiting calls, you skip to the next iteration. Otherwise, the log is presented in a color of the current consultant. Then, the thread is suspended for a random period of time between 1 and 10 seconds. After this time, the consultant ends the call, which is indicated by calling the **`End`** method, and a log is generated.

The last method is named **`Log`** and is similar to the previous example:

```
void Log(string text,
    ConsoleColor color = ConsoleColor.Gray)
{
    Console.ForegroundColor = color;
    Console.WriteLine(
        $"[{DateTime.Now:HH:mm:ss.fff}] {text}");
    Console.ResetColor();
}
```

When you run the program and wait for some time, you will receive a result similar to the one shown in the following screenshot:

```
[13:57:34.550] Incoming call #1 from client #513
[13:57:34.561] Waiting calls in the queue: 1
[13:57:35.311] Call #1 from client #513 answered by Olivia.
[13:57:35.326] Incoming call #2 from client #3349
[13:57:35.326] Waiting calls in the queue: 1
[13:57:35.362] Call #2 from client #3349 answered by Marcin.
[13:57:36.693] Incoming call #3 from client #1391
[13:57:36.695] Waiting calls in the queue: 1
[13:57:36.946] Call #3 from client #1391 answered by James.
[13:57:38.235] Incoming call #4 from client #6273
[13:57:38.236] Waiting calls in the queue: 1
[13:57:38.787] Incoming call #5 from client #4256
[13:57:38.789] Waiting calls in the queue: 2
[13:57:40.011] Incoming call #6 from client #9067
[13:57:40.012] Waiting calls in the queue: 3
[13:57:40.721] Incoming call #7 from client #7247
[13:57:40.722] Waiting calls in the queue: 4
[13:57:40.861] Call #2 from client #3349 ended by Marcin.
[13:57:41.413] Call #4 from client #6273 answered by Marcin.
[13:57:42.089] Incoming call #8 from client #3715
[13:57:42.090] Waiting calls in the queue: 4
[13:57:43.574] Call #3 from client #1391 ended by James.
[13:57:44.017] Incoming call #9 from client #5615
[13:57:44.017] Waiting calls in the queue: 5
[13:57:44.285] Call #5 from client #4256 answered by James.
[13:57:45.217] Call #1 from client #513 ended by Olivia.
[13:57:45.971] Incoming call #10 from client #1521
[13:57:45.973] Waiting calls in the queue: 5
[13:57:46.112] Call #6 from client #9067 answered by Olivia.
[13:57:46.505] Incoming call #11 from client #1347
[13:57:46.506] Waiting calls in the queue: 5
```

Figure 5.11 – Screenshot of the call center with many consultants example

You just completed two examples representing the application of a queue in the case of a call center scenario. Are you already a queue master?

TRY TO MODIFY PARAMETERS ON YOUR OWN

It is a good idea to modify various parameters of the program, such as the number of consultants, as well as delay times, especially the delay between following calls performed by clients. Then, you will see how the algorithm works in the case when there are too many clients, as well as too many or too few consultants.

However, how can you handle clients with priority support? In the current solution, they wait in the same queue as clients with the standard support plan. Do you need to create two queues and first take clients from the prioritized queue? If so, what should happen if you introduce another support plan? Do you need to add another queue and introduce such modifications in the code? Fortunately, no! You can use another data structure, namely a priority queue, as explained in detail in the following section.

Priority queues

A **priority queue** makes it possible to extend the concept of a queue by setting a **priority** for each element in the queue. **The elements are dequeued starting from those with the highest priority. Within each priority, you dequeue items in the same order as in a standard queue.** It is worth mentioning that the priority can be specified simply as an integer value. It depends on the implementation whether smaller or greater values indicate higher priority. In the book, it is assumed that the highest priority is equal to `0`, while lower priority is specified by `1`, `2`, `3`, and so on.

IMAGINE A PRIORITY QUEUE

If you want to better visualize a priority queue, close your eyes for a moment and imagine yourself going on the greatest vacation of your life. All passengers are already lining up at the gate, including you, but it turns out that right next to it, there is a much shorter queue for people who have a gold airline card. There are only 3 people in that line, and in yours there are over 100. These 3 people will be served first, and only then will the service of your queue begin. Well, that's how a priority queue works! You first serve all the highest priority items in the order they were added to the priority queue. Then, you return all lower priority items, also in the order they were added to the priority queue. Then, you take all items with an even lower priority, and so on, until all priorities are properly handled. And now the dream about holidays is over, it's time to get back to further learning data structures and algorithms!

A diagram of a priority queue is presented next:

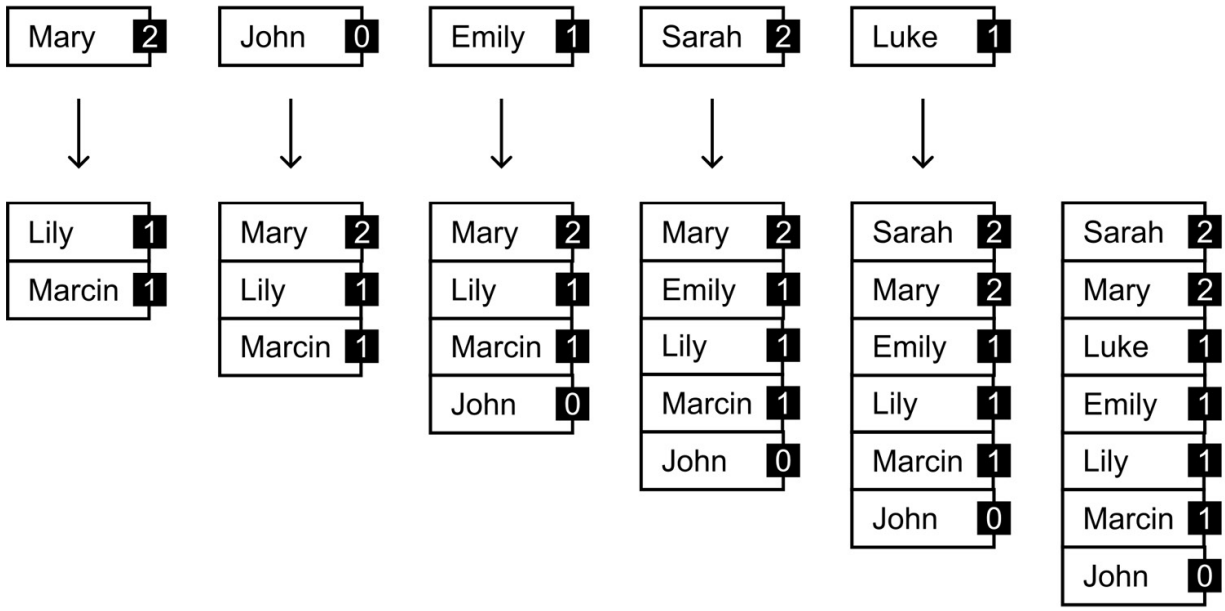

Figure 5.12 – Illustration of a priority queue

Let's analyze the diagram. First, the priority queue contains two elements with the same priority (equal to `1`), namely `Marcin` (first) and `Lily` (second). Then, `Mary` is added with the lowest priority (`2`), which means that this element is placed at the end of the queue. In the next step, `John` is added with the highest priority (`0`), so it is added at the beginning of the priority queue. The third column presents the addition of `Emily` with a priority equal to `1` - the same as `Marcin` and `Lily`. As `Emily` is added last, it is added after `Lily`. According to the aforementioned rules, you add the following elements - namely, `Sarah` with a priority set to `2` and `Luke` with a priority equal to `1`. The final order is shown on the right-hand side of the preceding diagram.

Of course, it is possible to implement a priority queue **on your own**. However, you can simplify this task by using the **built-in generic class**, namely `PriorityQueue` from the `System.Collections.Generic` namespace. The mentioned class requires you to specify two types, namely for the stored data and for the priority. The class contains some useful methods, such as the following:

- `Enqueue` adds an element to the priority queue
- `Dequeue` removes an element from the beginning and returns it
- `Clear` removes all elements from the priority queue
- `Peek` returns an element from the beginning of the queue without removing it

You can also get the number of elements in the queue using the `Count` property. The class contains a set of other methods as well - for example, `TryDequeue` and `TryPeek`.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a priority queue at https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.priorityqueue-2.

To make your horizons even broader, you will learn how to use **another implementation** of the priority queue, namely using one of the available **NuGet packages**, `OptimizedPriorityQueue`. More information about this package is available at https://www.nuget.org/packages/OptimizedPriorityQueue.

HOW TO INSTALL A NUGET PACKAGE?

*Do you know how you can add a NuGet package to your project? If not, select **Manage NuGet Packages** from the context menu of the project node in the **Solution Explorer** window. Then, choose the **Browse** tab in the opened window and type the name of the package in the **Search** box. Click on the name of the package and press **Install**. Confirm this operation and wait until the installation is ready.*

While the package is being installed, do you know that you can also be an author of a NuGet package that can be later used by developers from various regions of the world? If you create something great, please let me know! In the meantime, please keep in mind that you always should comply with the license terms of particular packages, and you should not fully trust all available packages, especially those with a smaller number of downloads. However, NuGet packages are a nice feature that can significantly simplify and speed up your work.

The `OptimizedPriorityQueue` library simplifies the application of a priority queue. Within it, the `SimplePriorityQueue` generic class is available, which contains some useful methods, such as the following:

- `Enqueue` adds an element to the priority queue
- `Dequeue` removes an element from the beginning of the queue and returns it
- `GetPriority` returns the priority of the element
- `UpdatePriority` updates the priority of the element
- `Contains` checks whether the element exists in the priority queue
- `Clear` removes all elements from the priority queue

You can get the number of elements currently available in the priority queue using the `Count` property. If you want to get an element from the beginning of the priority queue without removing it, you can use the `First` property. Moreover, the class contains a set of other methods, such as `TryDequeue` and `TryRemove`. As you can see, the names of some members of the class are even the same, as in the case of the `PriorityQueue` built-in class. Thus, you can easily change one implementation to another and check the impact of the implementation on the results or the performance of your solution.

WHAT ABOUT THE PERFORMANCE?

Both `Enqueue` and `Dequeue` methods are O(log n) operations.

If you want to see in action the priority queue depicted in the preceding diagram, you can use the following part of the code:

```
using Priority_Queue;
SimplePriorityQueue<string> queue = new();
queue.Enqueue("Marcin", 1);
queue.Enqueue("Lily", 1);
queue.Enqueue("Mary", 2);
queue.Enqueue("John", 0);
queue.Enqueue("Emily", 1);
queue.Enqueue("Sarah", 2);
queue.Enqueue("Luke", 1);
while (queue.Count > 0)
{
    Console.WriteLine(queue.Dequeue());
}
```

At the beginning, you create a new priority queue containing only **`string`** values. Then, you add all elements in the correct order, together with specifying their priority, using the **`Enqueue`** method. At the end, you use a **`while`** loop to dequeue all the elements, using the **`Dequeue`** method. Pretty simple and easy to understand, isn't it?

When you run the code, you will get the following result:

```
John
Marcin
Lily
Emily
Luke
Mary
Sarah
```

After this short introduction to the topic of priority queues, let's proceed to the example of a call center with priority support, which is described next.

Example – call center with priority support

As an example of a priority queue, let's present a simple approach to the call center solution, where there are many clients (with different identifiers), and only one consultant who answers waiting calls, first from clients with the priority support plan, and then from clients with the standard support plan.

This scenario is presented in the following diagram. Calls with standard priority are marked with –, while calls with priority support are indicated by ▵, as follows:

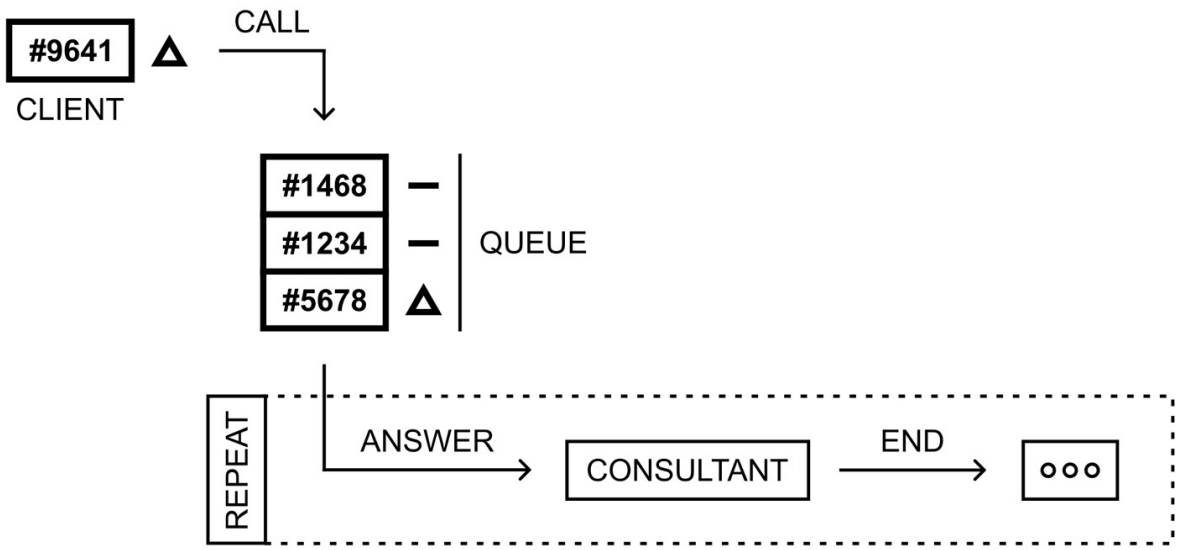

Figure 5.13 – Illustration of the call center with priority support example

The priority queue contains only three elements, which will be served in the following order: `#5678` (the priority support), `#1234`, and `#1468`. However, the call from the client with the `#9641` identifier causes the order to change to `#5678`, `#9641` (due to priority support), `#1234`, and `#1468`.

It is high time to write some code! Let's proceed to the implementation of the `IncomingCall` class:

```
public class IncomingCall
{
    public int Id { get; set; }
    public int ClientId { get; set; }
    public DateTime CallTime { get; set; }
    public DateTime? AnswerTime { get; set; }
    public DateTime? EndTime { get; set; }
    public string? Consultant { get; set; }
    public bool IsPriority { get; set; }
}
```

Here, there is only one change in comparison to the previously presented scenario of the simple call center application - namely, the `IsPriority` property is added. It indicates whether the current call has priority (`true`) or standard support (`false`).

Some modifications are also necessary for the `CallCenter` class, where a type of the `Calls` property is changed to `SimplePriorityQueue<IncomingCall>`, as shown next:

```
public class CallCenter
{
    private int _counter = 0;
    public SimplePriorityQueue<IncomingCall> Calls
        { get; private set; }
    public CallCenter() => Calls =
        new SimplePriorityQueue<IncomingCall>();
}
```

The following changes are necessary for the `Call` method:

```
public IncomingCall Call(int clientId, bool isPriority)
{
    IncomingCall call = new()
    {
        Id = ++_counter,
        ClientId = clientId,
        CallTime = DateTime.Now,
        IsPriority = isPriority
    };
    Calls.Enqueue(call, isPriority ? 0 : 1);
    return call;
}
```

Here, a value of the `IsPriority` property is set using the parameter. Moreover, while calling the `Enqueue` method, two parameters are used, not only the value of the element (an instance of the `IncomingCall` class), but also an integer value representing the priority, namely `0` in the case of priority support, or `1` otherwise.

No more changes are necessary in the methods of the `CallCenter` class, namely in `Answer`, `End`, and `AreWaitingCalls`, which are shown next for your convenience:

```
public IncomingCall? Answer(string consultant)
{
    if (!AreWaitingCalls()) { return null; }
    IncomingCall call = Calls.Dequeue();
    call.Consultant = consultant;
    call.AnswerTime = DateTime.Now;
    return call;
}
public void End(IncomingCall call) =>
    call.EndTime = DateTime.Now;
public bool AreWaitingCalls() => Calls.Count > 0;
```

Finally, let's take a look at the code located in the `Program.cs` file:

```
Random random = new();
CallCenter center = new();
center.Call(1234, false);
center.Call(5678, true);
center.Call(1468, false);
center.Call(9641, true);
while (center.AreWaitingCalls())
{
    IncomingCall call = center.Answer("Marcin")!;
    Log($"Call #{call.Id} from client #{call.ClientId} is
        answered by {call.Consultant}.", call.IsPriority);
    await Task.Delay(random.Next(1000, 10000));
    center.End(call);
    Log($"Call #{call.Id} from client #{call.ClientId} is
        ended by {call.Consultant}.", call.IsPriority);
}
void Log(string text, bool isPriority)
{
    Console.ForegroundColor = isPriority
        ? ConsoleColor.Red : ConsoleColor.Gray;
    Console.WriteLine($"[{DateTime.Now:HH:mm:ss}] {text}");
```

```
        Console.ResetColor();
    }
```

You may be surprised to learn that only small changes are necessary in this part of the code. The reason for this is that the logic regarding a used data structure is hidden in the `CallCenter` class. Within the `Program.cs` file, you call methods and use properties exposed by the `CallCenter` class. You just need to modify how you add calls to the queue (together with priorities), as well as adjust logs presented when a call is answered by the consultant, to choose a proper color based on the call's priority. That's all!

When you run the application, you will receive a result similar to the following:

```
[13:58:09] Call #2 from client #5678 is answered by Marcin.
[13:58:11] Call #2 from client #5678 is ended by Marcin.
[13:58:11] Call #4 from client #9641 is answered by Marcin.
[13:58:14] Call #4 from client #9641 is ended by Marcin.
[13:58:14] Call #1 from client #1234 is answered by Marcin.
[13:58:16] Call #1 from client #1234 is ended by Marcin.
[13:58:16] Call #3 from client #1468 is answered by Marcin.
[13:58:21] Call #3 from client #1468 is ended by Marcin.
```

Figure 5.14 – Screenshot of the call center with priority support example

As you can see, the calls are served in the correct order. This means that the calls from clients with priority support are served earlier than calls from clients with the standard support plan, even though such calls need to wait much longer to be answered.

Circular queues

At the end of this chapter, let's take a look at another data structure, namely a **circular queue**, also called a **ring buffer**. In this case, **a queue forms a circle**, internally uses an array, and the maximum number of elements that can be placed inside the queue is limited. You need to specify two variables that indicate indices of the **front** and **rear** elements. **The front one points to the element that will be dequeued first. The rear one points to the element that is the last in the queue**.

IMAGINE A CIRCULAR QUEUE

If you want to better imagine a circular queue, think back to your young years when you persuaded your parents to take you on a roller coaster. It consisted of 10 carriages, each with room for 2 people, so only 20 people could take part in 1 roller coaster ride. As this was a unique attraction, such a ride took place only once an hour. This meant that only 20 people were allowed to enter a queue for the roller coaster and no one else. As the departure date approached, people were admitted to it in the order in which they were admitted to the queue. And a circular queue works similarly! It has some

specific capacity, and nothing else can be enqueued to it. However, when you dequeue elements, new ones can be added in place of the previous ones. Well, it means that after an hour, you can fill a queue for the roller coaster with new people!

The mentioned data structure is presented in the following diagram:

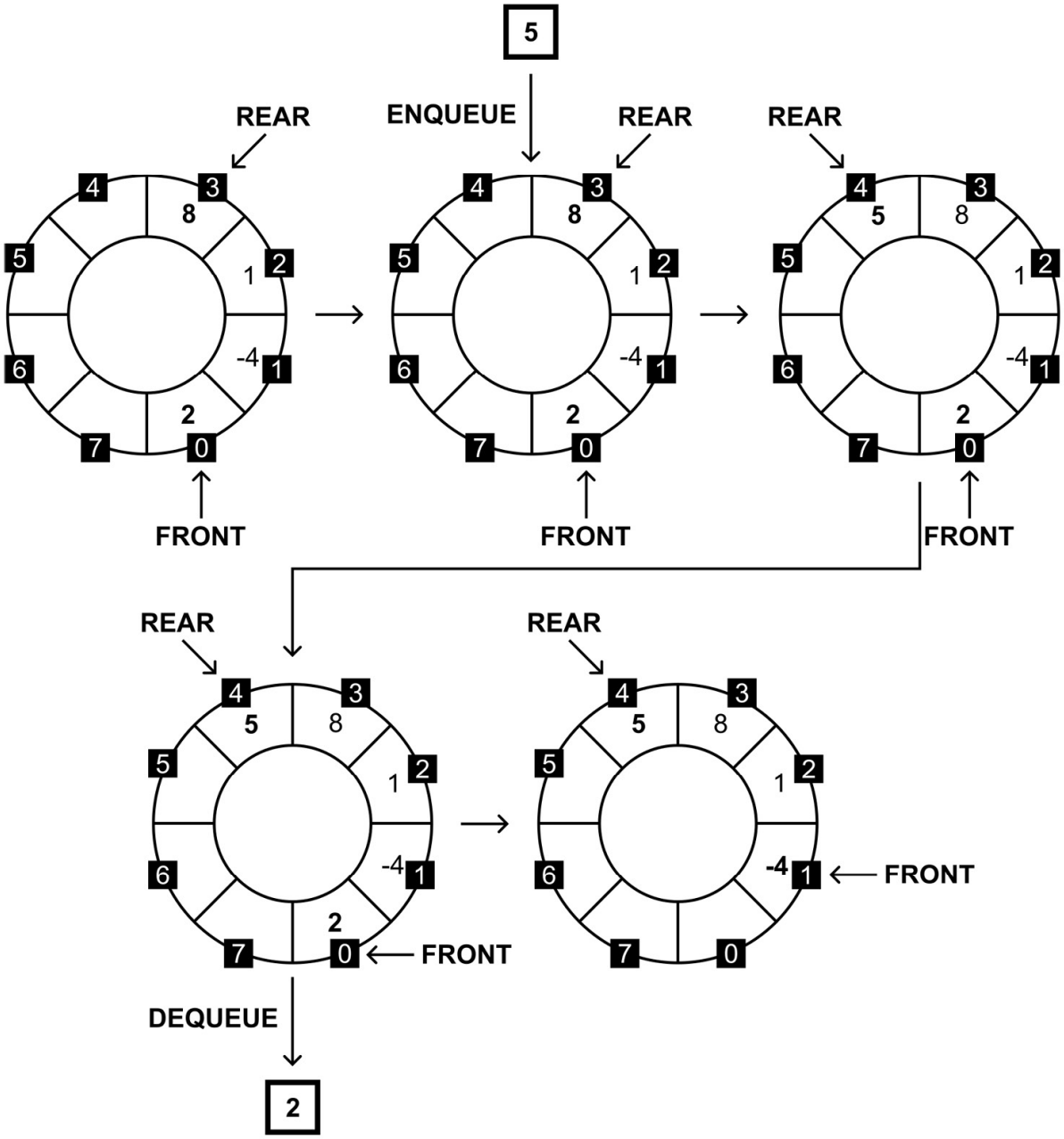

Figure 5.15 – Illustration of a circular queue

At the beginning, the circular queue is empty, so both **front** and **rear** indices are equal to `-1`. Then, you add `2`, `-4`, `1`, and `8` elements, and this state is shown in the first step in the preceding diagram. Here, the front index is equal to `0`, and the rear one to `3`.

In the next step, you perform an **enqueue operation that adds the item at the end of the queue**, as in the case of a regular queue. Thus, `5` is located at index `4`, as shown in the third step in the preceding diagram. Of course, the rear index is updated to `4` while the front index remains the same, namely `0`.

The following step shows a **dequeue operation that removes an item from the beginning of the queue**, so its aim is the same as in the case of a regular queue. In the example, `2` is returned, and the front index is changed to `1`. It means that currently, the circular queue stores `4` elements in the part of the array between `1` (the front index) and `4` (the rear index).

WHAT ABOUT THE PERFORMANCE?

The performance results are great in this case! Both the `Enqueue` and the `Dequeue` methods are O(1) operations, as you don't need to iterate through an array.

You can perform many more *enqueue* and *dequeue* operations to see how the content of the queue "rotates" within a circular queue. To do so, you need to implement this data structure. Let's write some code, starting with the `CircularQueue` class:

```
public class CircularQueue<T>(int size)
    where T : struct
{
    private readonly T[] _items = new T[size];
    private int _front = -1;
    private int _rear = -1;
    private int _count = 0;
    public int Count { get { return _count; } }
}
```

It is a generic class that uses the primary constructor taking the maximum number of elements in the queue as the `size` parameter. You can see four private fields:

- An array with the stored items (`_items`)
- An index of the front and rear elements in the queue (`_front` and `_rear`)
- The current number of elements located in the circular queue (`_count`)

The public read-only `Count` property is added as well, which returns the value of the `_count` field. If everything is clear for you, let's take a look at the `Enqueue` method:

```
public bool Enqueue(T item)
{
    if (_count == _items.Length) { return false; }
    if (_front < 0) { _front = _rear = 0; }
    else { _rear = ++_rear % _items.Length; }
    _items[_rear] = item;
    _count++;
    return true;
}
```

In the beginning, you need to check whether you have any space within the circular queue, so you compare the current number of elements in the queue (`_count`) with the length of the array storing

such data (**_items**). If these values are equal, you return **false** because there is no space, so you cannot enqueue any element.

The next line checks whether the circular queue is empty, which means that the front index is smaller than **0**. If so, both front and rear indices are set to **0**. It indicates that there is only one element in the circular queue, and it is pointed to by both these indices.

If there is already something in the queue, you increment a value of the rear index. If it is equal to the number of elements in the array, you assign **0** to it.

In the last three lines, you add the new item to the place indicated by the rear index (**_rear**), increment the counter storing the number of elements currently located in the queue (**_count**), as well as return **true** indicating that the enqueue operation was successful.

Let's now move to the **Dequeue** method, the code of which is shown next:

```
public T? Dequeue()
{
    if (_count == 0) { return null; }
    T result = _items[_front];
    if (_front == _rear) { _front = _rear = -1; }
    else { _front = ++_front % _items.Length; }
    _count--;
    return result;
}
```

Here, you check whether the circular queue is empty. If so, you return a **null** value. Otherwise, you save as **result** a value indicated by the front index. Such a value will be returned at the end of this method.

In the following lines, you check whether the front and rear indices are equal. It means that there is only one element in the queue. If so, you set both these indices to **-1**, which indicates that the circular queue is empty. Otherwise, you increment the front index. If it is equal to the number of elements in the array, you assign **0** to it.

In the last two lines, you just decrement the number of elements in the queue, as well as return the previously saved value (**result**).

Another method is named **Peek** and is presented as follows:

```
public T? Peek()
{
    if (_count == 0) { return null; }
    return _items[_front];
}
```

This method just returns the first item in the queue without removing it from the queue. Of course, it returns **null** if the queue is empty.

As you can see, the implementation of a circular queue is not difficult and requires a small number of lines of code. So, let's see it in action with the following code:

```
CircularQueue<int> queue = new(8);
queue.Enqueue(2);
queue.Enqueue(-4);
queue.Enqueue(1);
queue.Enqueue(8);
queue.Enqueue(5);
int item = queue.Dequeue();
Console.WriteLine(item);
```

The preceding lines perform the operations shown in the diagram presenting a circular queue (*Figure 5.15*). You create a new circular queue with places for eight elements of the `int` type. Then, you add `2`, `-4`, `1`, `8`, and `5` values and dequeue one element.

After the introduction to the topic of circular queues, it is high time to take a look at a real-world example.

Example – gravity roller coaster

Let's simulate the behavior of a gravity roller coaster located on a mountainside. There can be a maximum of 12 carts on this queue at the same time, which slide down the chute accelerated by gravity. After the participant enters the cart, it accelerates automatically, and there are several turns on its path. After reaching the foot of the mountain, the cart and the participant are pulled in using a simple pulley. The participant gets off at the same place where they boarded the cart, as shown next:

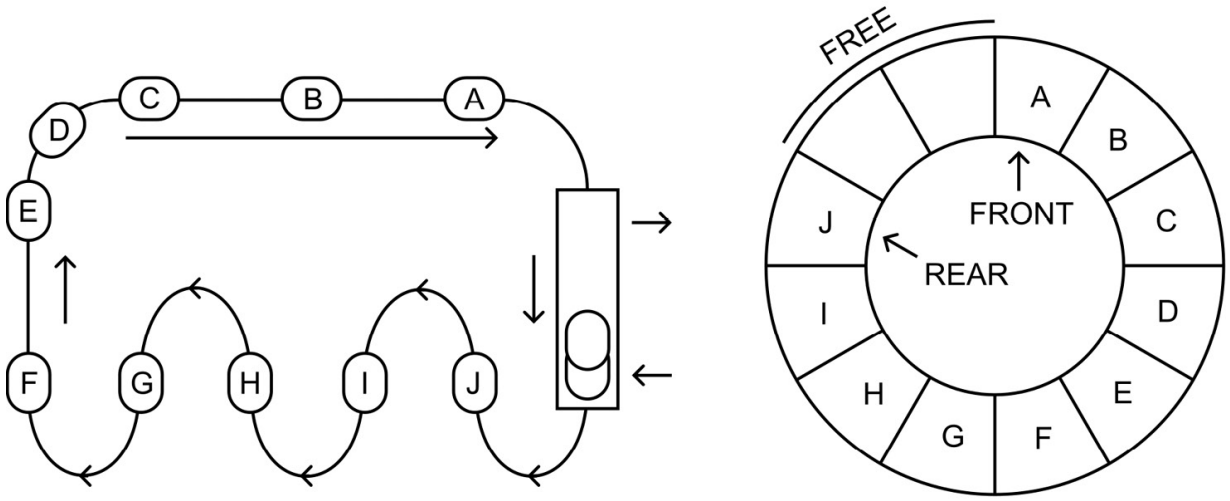

Figure 5.16 – Illustration of a gravity roller coaster example

You can simulate this example using a circular queue with the maximum size set to 12, which means that a maximum of 12 people can currently be on this gravity roller coaster. Another person will not be admitted until a seat becomes available. Entering the cart means performing an *enqueue* operation, and leaving the cart means performing a *dequeue* operation. It is also worth mentioning that it is not possible to change the order in which participants are served. Whoever enters the queue first will be let out first, which is consistent with the FIFO principle.

Let's take a look at the code, which is located in the **`Program.cs`** file:

```
using QueueItem = (System.DateTime StartedAt,
    System.ConsoleColor Color);
const int rideSeconds = 10;
Random random = new();
CircularQueue<QueueItem> queue = new(12);
ConsoleColor color = ConsoleColor.Black;
```

Here, you specify a **`QueueItem`** alias for the value tuple type consisting of cart entering time and the chosen color. Then, the ride length is set to 10 seconds, as well as some additional variables being created, including the circular queue and the last used color.

The next part of the code is shown here:

```
while (true)
{
    while (queue.Peek() != null)
    {
        QueueItem item = queue.Peek()!.Value;
        TimeSpan elapsed = DateTime.Now - item.StartedAt;
        if (elapsed.TotalSeconds < rideSeconds) { break; }
        queue.Dequeue();
        Log($"> Exits\tTotal: {queue.Count}", item.Color);
    }
    bool isNew = random.Next(3) == 1;
    if (isNew)
    {
        color = color == ConsoleColor.White
            ? ConsoleColor.DarkBlue
            : (ConsoleColor)(((int)color) + 1);
        if (queue.Enqueue((DateTime.Now, color)))
        {
            Log($"< Enters\tTotal: {queue.Count}", color);
        }
        else
        {
            Log($"! Not allowed\tTotal: {queue.Count}",
                ConsoleColor.DarkGray);
        }
    }
    await Task.Delay(500);
}
```

It contains an infinite **`while`** loop. Within it, you first check which items should be dequeued, which means that the ride time (that is, 10 seconds) elapsed for them. If so, you also log the message. Then, you draw a random number to decide whether a new item should be added to the circular queue in

this iteration of an infinite **`while`** loop. If so, you choose the next color from the **`ConsoleColor`** enumeration, try to enqueue a new item to the queue, as well as log the message. At the end of the iteration, you wait 500 milliseconds.

The code of the auxiliary **`Log`** method is presented here:

```
void Log(string text, ConsoleColor color)
{
    Console.ForegroundColor = color;
    Console.WriteLine($"{DateTime.Now:HH:mm:ss} {text}");
    Console.ResetColor();
}
```

When you run the code, you get the following result:

```
18:01:54 < Enters       Total: 1
18:01:55 < Enters       Total: 2
18:01:55 < Enters       Total: 3
18:01:56 < Enters       Total: 4
18:01:57 < Enters       Total: 5
18:01:58 < Enters       Total: 6
18:02:01 < Enters       Total: 7
18:02:04 > Exits        Total: 6
18:02:05 > Exits        Total: 5
18:02:05 > Exits        Total: 4
18:02:06 < Enters       Total: 5
18:02:06 > Exits        Total: 4
18:02:06 < Enters       Total: 5
18:02:07 > Exits        Total: 4
18:02:07 < Enters       Total: 5
18:02:08 > Exits        Total: 4
18:02:09 < Enters       Total: 5
18:02:09 < Enters       Total: 6
18:02:11 > Exits        Total: 5
18:02:13 < Enters       Total: 6
18:02:14 < Enters       Total: 7
18:02:15 < Enters       Total: 8
18:02:16 > Exits        Total: 7
18:02:16 < Enters       Total: 8
18:02:16 > Exits        Total: 7
18:02:17 > Exits        Total: 6
18:02:17 < Enters       Total: 7
18:02:18 < Enters       Total: 8
18:02:18 < Enters       Total: 9
18:02:19 > Exits        Total: 8
```

Figure 5.17 – Screenshot of the gravity roller coaster example

Congratulations – you now know how to use a few types of queues! You took a look at a regular one, a priority queue, as well as a circular one, together with examples. So, it is high time to summarize the chapter.

Summary

In this chapter, you learned about two limited access data structures, namely stacks and queues, including regular, priority, and circular ones. It is worth remembering that such data structures have strictly specified ways of accessing elements. All of them also have various real-world applications. Some of them were mentioned in this chapter.

First, you saw how a **stack** operates according to the LIFO principle. In this case, you can add an element at the top of the stack (a *push* operation), and remove an element from the top (a *pop* operation). The stack was shown in two examples, namely for reversing a word and for solving the *Tower of Hanoi* mathematical game.

In the following part of the chapter, you got to know a **queue** as a data structure, which operates according to the FIFO principle. In this case, *enqueue* and *dequeue* operations were presented. The queue was explained using two examples, both regarding the application simulating a call center. You learned how to use a thread-safe variant of a queue-related class, which is available while developing applications in the C# language.

The next data structure shown in this chapter is named a **priority queue** and is an extension of a queue that supports the priorities of particular elements. In the end, you learned about a **circular queue**, which expands the concept of a regular queue by forming a circle, where the first and rear elements are indicated by indices.

This is just the fifth chapter of this book, and you already learned a lot about various data structures and algorithms that are useful while developing applications in C#! Are you interested in increasing your knowledge by learning about **dictionaries** and **sets**? If so, let's proceed to the next chapter and learn more about them!

6

Dictionaries and Sets

This chapter focuses on data structures related to dictionaries and sets. Applying these data structures makes it possible **to map keys to values and perform fast lookup**, as well as **to make various operations on sets**. To simplify your understanding of dictionaries and sets, this chapter contains illustrations and code snippets, along with detailed descriptions.

First, you will learn about both non-generic and generic versions of a **dictionary**, which is a collection of pairs, each consisting of a key and a value. Then, a **sorted variant** of a dictionary will be presented. The remaining part of this chapter will show you how to use **hash sets**, together with a **"sorted" set** variant. Is it possible to have a "sorted" set? You'll learn more later in this chapter.

In this chapter, the following topics will be covered:

- Hash tables
- Dictionaries
- Sorted dictionaries
- Hash sets
- "Sorted" sets

Hash tables

Let's start with the first data structure, which is a **hash table**, also known as a **hash map**. **It allows you to map keys to particular values**. One of the most important assumptions of the hash table is the possibility of **a very fast lookup for a value based on the key**, which should be the *O(1)* operation.

IMAGINE A HASH TABLE OR A DICTIONARY

If you want to better imagine a hash table or a dictionary, it would be worth thinking about a collection containing a lot of data, where it is crucial to quickly check whether the dictionary contains a specific key, as well as quickly retrieve the value assigned to a given key. So, think about a system that allows you to determine which country a specific IP address comes from. As you know, there are many possible IP addresses, and your system must quickly obtain information from which country the user's request comes to select the default language version of the application. This is how a hash table and dictionary work! You use an IP address as a key (for example, 50.50.50.50) and a country code as a value (for example, PL). Thus, you can quickly find out from which country the user came to you, without manually browsing the entire collection. I come to you from Poland, which I cordially invite you to visit! It contains mountains, sea, lakes, and cities with a rich history. All this is waiting here for you!

To achieve a very fast lookup, a **hash function** is used. It takes the key to generate an index of a bucket, where the value can be found. For this reason, if you need to find a value of the key, you do not need to iterate through all the items in the collection since you can just use the hash function to easily locate a proper bucket and get the value. As you can see, the role of the hash function is critical and ideally, it should generate a unique result for all keys. However, the same result may be generated for different keys. Such a situation is called a **hash collision** and should be dealt with.

A way of mapping keys to particular values is shown in the following diagram:

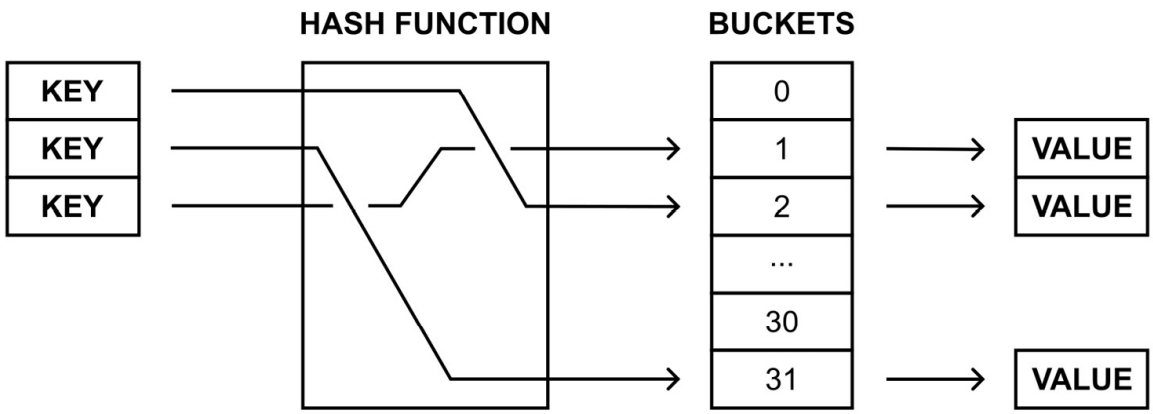

Figure 6.1 – Illustration of mapping keys to particular values

Due to the great performance of hash tables, they are frequently used in many real-world applications, such as for **associative arrays**, **database indices**, and **cache systems**.

The topic of implementing hash tables from scratch seems to be quite difficult, especially when it comes to using the hash function, handling hash collisions, as well as assigning particular keys to buckets. Fortunately, a suitable implementation is available while developing applications in the C# language, and its usage is very simple.

NON-GENERIC AND GENERIC VERSIONS

*There are two variants of the hash table-related classes, namely non-generic (**`Hashtable`**) and generic (**`Dictionary`**). The first will be described in this section, while the other will be described in the following section. If you can use the strongly typed generic version, I strongly recommend using it.*

Let's take a look at the **`Hashtable`** class from the **`System.Collections`** namespace. As mentioned previously, it stores a collection of pairs, where each contains a key and a value. A pair is represented by the **`DictionaryEntry`** instance.

Here's some example code that uses the **`Hashtable`** class:

```
using System.Collections;
Hashtable hashtable = new()
```

```
{
    { "Key #1", "Value #1" },
    { "Key #2", "Value #2" }
};
hashtable.Add("Key #3", "Value #3");
hashtable["Key #4"] = "Value #4";
```

As shown here, you can add items to a hash table in a few ways, namely by specifying keys and values while creating a new instance (such as for **`Key #1`** and **`Key #2`**), by using the **`Add`** method (**`Key #3`**) or by using the indexer (**`Key #4`**).

When you use the indexer to set a value for an already existing key, the value of this element is updated. A different behavior occurs while using the **`Add`** method because it throws an exception when an item with the same key already exists in the collection. You can handle this situation by using the **`try-catch`** statement, but there is a much better approach to check whether such an entry already exists – using the **`ContainsKey`** method. This will be shown a bit later.

It is worth mentioning that the **`null`** value is incorrect for the key of an element, but it is acceptable as a value of an element.

You can easily gain access to a particular element using the indexer. As the **`Hashtable`** class is a non-generic variant of hash table-related classes, you need to cast the returned result to the proper type (for example, **`string`**), as shown here:

```
string value = (string)hashtable["Key #1"]!;
```

If you want to get all the entries from the hash table, you can use a **`foreach`** loop to iterate through all pairs stored in the collection, as presented here:

```
foreach (DictionaryEntry entry in hashtable)
{
    Console.WriteLine($"{entry.Key}: {entry.Value}");
}
```

The variable that's used in the loop is of the **`DictionaryEntry`** type. Therefore, you can use its **`Key`** and **`Value`** properties to access the key and the value, respectively.

The **`Hashtable`** class is equipped with a few properties, such as those for getting the number of stored elements (**`Count`**), as well as returning the collection of keys and values (**`Keys`** and **`Values`**). You can use the following methods:

- **`Add`**, which adds a new element
- **`Remove`**, which removes an element
- **`Clear`**, which removes all elements
- **`ContainsKey`**, which checks whether the collection contains a given key
- **`ContainsValue`**, which checks whether the collection contains a given value

WHAT ABOUT PERFORMANCE?

A hash table is an efficient data structure. Retrieving a value by a key, checking whether the collection contains a given key, and removing an item by a key are the O(1) operations. As for addition, if the capacity does not need to be increased, it is the O(1) operation as well. Otherwise, it is the O(n) operation, where n is the number of items.

With this short introduction, let's take a look at an example.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a hash table at https://learn.microsoft.com/en-us/dotnet/api/system.collections.hashtable.

Example – phone book

As an example, let's say you create an application for a phone book. The **`Hashtable`** class is used to store entries where the person's name is a key and the phone number is a value, as shown here:

```
NAME ---> PHONE
Marcin -> 101-202-303
John ---> 202-303-404
Aline --> 303-404-505
```

This program demonstrates how to add elements to the collection, get the number of stored items, iterate through all of them, check whether an element with a given key exists, as well as how to get a value based on the key.

First, let's create a new instance of the **`Hashtable`** class, as well as initialize it with some entries, as shown in the following code:

```
Hashtable phoneBook = new()
{
    { "Marcin", "101-202-303" },
    { "John", "202-303-404" }
};
phoneBook["Aline"] = "303-404-505";
```

You can check whether there are no elements in the collection using the **`Count`** property and comparing its value with **`0`**, as presented here:

```
Console.WriteLine("Phone numbers:");
if (phoneBook.Count == 0)
{
    Console.WriteLine("Empty list.");
}
```

Then, you can iterate through all the pairs:

```
foreach (DictionaryEntry entry in phoneBook)
{
    Console.WriteLine($"{entry.Key}: {entry.Value}");
}
```

Finally, let's see how we can check whether a specific key exists in the collection, as well as how to get its value. The first task can be accomplished just by calling the **`ContainsKey`** method, which returns a value indicating whether a suitable element exists (**`true`**) or not (**`false`**). To get a value, you can use the indexer. Please keep in mind that you must cast the returned value to a suitable type, such as **`string`** in this example. This requirement is caused by the non-generic version of the hash table-related class. This code is as follows:

```
Console.Write("\nSearch by name: ");
string name = Console.ReadLine() ?? string.Empty;
if (phoneBook.ContainsKey(name))
{
    string number = (string)phoneBook[name]!;
    Console.WriteLine($"Phone number: {number}");
}
else
{
    Console.WriteLine("Does not exist.");
}
```

Your first program using the hash table is ready! After launching it, you should receive a result similar to the following:

```
Phone numbers:
Marcin: 101-202-303
Aline: 303-404-505
John: 202-303-404
Search by name: Aline
Phone number: 303-404-505
```

It is worth noting that the order of the pairs stored using the **`Hashtable`** class is not consistent with the order of their addition or keys. For this reason, if you need to present the sorted results, you need to sort the elements on your own or use another data structure, namely **`SortedDictionary`**, which is described later in this book.

Now, let's take a look at one of the most common classes used while developing in C#, namely **`Dictionary`**, which is a generic version of hash table-related classes.

Dictionaries

In the previous section, you learned about the **`Hashtable`** class, a non-generic variant of the hash table-related classes. However, it has a significant limitation, because it does not allow you to specify a type of a key and a value. Both the **`Key`** and **`Value`** properties of the **`DictionaryEntry`** class are of the **`object`** type. Therefore, you need to perform boxing and unboxing operations, even if all the keys and values are of the same type. If you want to benefit from the **strongly typed variant**, you can use the **`Dictionary`** generic class, which is the main subject of this section.

First of all, you should specify two types, namely a type of a key and a value, while creating an instance of the **`Dictionary`** class. Moreover, it is possible to define the initial content of the dictionary using the following code:

```
Dictionary<string, string> dictionary = new()
{
    { "Key #1", "Value #1" },
    { "Key #2", "Value #2" }
};
```

In the preceding code, a new instance of the **`Dictionary`** class is created. It stores **`string`**-based keys and values. Here, two entries exist in the dictionary, namely **`Key #1`** and **`Key #2`**. Their values are **`Value #1`** and **`Value #2`**.

Similar to the **`Hashtable`** class, you can also use the indexer to get access to a particular element within the collection, as shown in the following line of code:

```
string value = dictionary["Key #1"];
```

It is worth noting that casting to the **`string`** type is unnecessary because **`Dictionary`** is the strongly typed version of the hash table-related classes. Therefore, the returned value already has the proper type. If an element with the given key does not exist in the collection, **`KeyNotFoundException`** is thrown. To avoid problems, you can either check whether the element exists (by calling **`ContainsKey`**) or use the **`TryGetValue`** method.

You can add a new element or update a value of the existing one using the indexer:

```
dictionary["Key #3"] = "Value #3";
```

Similar to the non-generic variant, **`key`** cannot be equal to **`null`**, but **`value`** can be if it is allowed by the type of values stored in the collection.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a dictionary at https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2.

The **`Dictionary`** class is equipped with a few properties:

- **`Count`**, which gets the number of stored elements
- **`Keys`**, which returns the collection of keys
- **`Values`**, which returns the collection of values

You can also use some available methods:

- **`Add`**, which adds a new element to the dictionary
- **`Remove`**, which removes an element from the dictionary

- **`Clear`**, which removes all the elements from the dictionary
- **`ContainsKey`**, which checks whether the dictionary contains a key
- **`ContainsValue`**, which checks whether the dictionary contains a given value
- **`TryGetValue`**, which tries to get a value for a given key from the dictionary

As you can see, many properties and methods are almost the same as in the case of the **`Hashtable`** class. The consistency of naming allows you to easily use various classes without the necessity of learning everything from scratch.

WHAT ABOUT PERFORMANCE?

*You should remember that the performance of getting a value of an element (using an indexer or **`TryGetValue`**), updating an existing one (using an indexer), and checking whether the given key exists in the dictionary (**`ContainsKey`**) approaches the O(1) operation. However, the process of checking whether the collection contains a given value (**`ContainsValue`**) is the O(n) operation and requires you to search the entire collection for the particular value.*

If you want to iterate through all pairs stored in the collection, you can use a **`foreach`** loop. However, the variable that's used in the loop is an instance of the **`KeyValuePair`** generic class with **`Key`** and **`Value`** properties, allowing you to access the key and the value. This **`foreach`** loop is shown in the following code snippet:

```
foreach (KeyValuePair<string, string> pair in dictionary)
{
    Console.WriteLine($"{pair.Key}: {pair.Value}");
}
```

Here, you can also apply what you've learned about value tuples and the deconstruct operation. Thus, the preceding **`foreach`** loop can be simplified, as shown here:

```
foreach ((string k, string v) in dictionary)
{
    Console.WriteLine($"{k}: {v}");
}
```

As you can see, the C# language is being equipped with more and more useful features that make your code shorter, simpler, and easier to understand. You should keep an eye on the updates to the language. Good work, C# team – I am looking forward to more!

THREAD-SAFE VERSION

*Do you remember a thread-safe queue-related class from the previous chapter? If so, the situation looks quite similar in the case of **`Dictionary`** because the **`ConcurrentDictionary`** class (from the **`System.Collections.Concurrent`** namespace) is available.*

With this short introduction, let's start coding! In the following sections, you will find two real-world examples that use dictionaries for storing data.

Example – product location

The first example we'll look at is an application that helps employees of a shop to find a product's location. Let's imagine that each employee has a phone with your application on it, which is used to scan a barcode of the product, and the application tells them that the product should be located in area **`A1`** or **`C9`**. Sounds interesting, doesn't it?

As the number of products in the shop is often very high, it is necessary to find results very quickly. For this reason, the data of products, together with their locations, is stored in the hash table, using the generic **`Dictionary`** class. The key is the barcode (**`string`**), while the value is the area code (also **`string`**), as shown here:

```
BARCODE -------> AREA
5901020304050 -> A1
5910203040506 -> B5
5920304050607 -> C9
```

First, you create a new collection and add some data:

```
Dictionary<string, string> products = new()
{
    { "5901020304050", "A1" },
    { "5910203040506", "B5" },
    { "5920304050607", "C9" }
};
products["5930405060708"] = "D7";
```

The code shows two ways of adding elements to the collection, namely by passing their data while creating a new instance of the class and by using the indexer. A third solution also exists and uses the **`Add`** method, as shown in the following part of the code:

```
string key = "5940506070809";
if (!products.ContainsKey(key))
{
    products.Add(key, "A3");
}
```

Another solution uses the **`TryAdd`** method, as presented here:

```
if (!products.TryAdd(key, "B4"))
{
    Console.WriteLine("Cannot add.");
}
```

In the following part of the code, you present data of all the products that are available in the system. To do so, you use the **`foreach`** loop. Before that, you check whether there are any elements in the dictionary. If not, the proper message is presented to the user. Otherwise, keys and values from all pairs are presented in the console:

```
Console.WriteLine("All products:");
```

```
if (products.Count == 0) { Console.WriteLine("Empty."); }
foreach ((string k, string v) in products)
{
    Console.WriteLine($"{k}: {v}");
}
```

Now, let's take a look at the part of the code that makes it possible to find the location of the product by its barcode. To do so, you can use **`TryGetValue`** to check whether the element exists. If so, a message with the target location is presented in the console. Otherwise, other information is shown. The code is presented here:

```
Console.Write("\nSearch by barcode: ");
string barcode = Console.ReadLine() ?? string.Empty;
if (products.TryGetValue(barcode, out string? location))
{
    Console.WriteLine($"The product is in: {location}.");
}
else
{
    Console.WriteLine("The product does not exist.");
}
```

When you run the program, you see a list of all the products in the shop and the program asks you to enter the barcode. After typing it, you receive a message containing the area code. The result that's shown in the console should be similar to the following:

```
Cannot add.
All products:
5901020304050: A1
5910203040506: B5
5920304050607: C9
5930405060708: D7
5940506070809: A3
Search by barcode: 5901020304050
The product is in: A1.
```

You've just completed the first example! Let's proceed to the next one.

Example – user details

This second example shows you how to store more complex data in the dictionary. In this scenario, you'll create an application that shows details of a user based on their identifier, as shown here:

```
ID   ->  FIRST NAME | LAST NAME | PHONE NUMBER
100  ->  Marcin     | Jamro     | 101-202-303
210  ->  John       | Smith     | 202-303-404
303  ->  Aline      | Weather   | 303-404-505
```

The program starts with the data of three users. You should be able to enter an identifier and see details of the found user. Of course, the situation of the non-existence of a given user should be handled by presenting the proper information in the console.

First, let's add the **`Employee`** record, which stores data of an employee, namely first name, last name, and phone number. The code is as follows:

```
public record Employee(string FirstName, string LastName,
    string PhoneNumber);
```

Then, create a new instance of the **`Dictionary`** class and add data to it:

```
Dictionary<int, Employee> employees = new()
{
    { 100, new Employee("Marcin", "Jamro", "101-202-303") },
    { 210, new Employee("John", "Smith", "202-303-404") },
    { 303, new Employee("Aline", "Weather", "303-404-505") }
};
```

The most interesting operations are performed in the following **`do-while`** loop:

```
do
{
    Console.Write("Enter the identifier: ");
    string idString = Console.ReadLine() ?? string.Empty;
    if (!int.TryParse(idString, out int id)) { break; }
    if (employees.TryGetValue(id, out Employee? Employee))
    {
        Console.WriteLine(
            "Full name: {0} {1}\nPhone number: {2}\n",
            employee.FirstName,
            employee.LastName,
            employee.PhoneNumber);
    }
    else { Console.WriteLine("Does not exist.\n"); }
}
while (true);
```

First, the user is asked to enter an identifier of the employee, which is then parsed to the integer value. The loop is stopped when the provided identifier cannot be parsed to the integer value. Otherwise, the **`TryGetValue`** method is used to try to get details of the user. If the user is found (**`TryGetValue`** returns **`true`**), the details are presented in the console. Otherwise, an error message is shown.

When you run the application and enter some data, you will receive the following result:

```
Enter the identifier: 100
Full name: Marcin Jamro
Phone number: 101-202-303
Enter the identifier: 101
Does not exist.
```

That's all! You've completed two examples showing how to use dictionaries while developing applications in the C# language. Do you remember that another kind of dictionary was already mentioned, namely a sorted dictionary? Are you interested in finding out what it is and how you can use it in your programs? If so, move on to the next section.

Sorted dictionaries

Both non-generic and generic variants of the hash table-related classes do not keep the order of the elements. For this reason, if you need to present data from the collection sorted by keys, you need to sort them before presenting them. However, you can use another data structure, known as a **sorted dictionary**, to solve this problem and **keep keys sorted all the time**. Therefore, you can easily get the sorted collection if necessary.

IMAGINE A SORTED DICTIONARY

If you want to better imagine a sorted dictionary, remember the times from a dozen or so years ago, when the internet was not as popular and widespread as it is today, and at home there was a book on your shelf that allowed you to learn the meaning of a word in another language. How does it work? Let's assume that you have a Polish-English dictionary, thanks to which you can find out how to translate a specific word from Polish to English, such as cześć to hello. You open this book and look for words that start with the letter c. Found! Now, you are browsing through words starting with c to find the one you are interested in, namely cześć. Fortunately, it's not that complicated because all the words are listed in the dictionary in alphabetical order. And that's how a sorted dictionary works! You can easily view all the items in the dictionary in alphabetical order. You can also quickly check if the dictionary contains a specific key and what its value is. Today, you just enter a foreign word in a search engine and you instantly know what it means in your language, as well as in probably any other language in the world. You like this kind of technological progress, don't you?

The sorted dictionary is implemented as the **`SortedDictionary`** generic class, available in the **`System.Collections.Generic`** namespace. You can specify types for keys and values while creating a new instance of **`SortedDictionary`**. An item key cannot be equal to **`null`**, but its value can be if it is allowed by the type of values stored in the collection. Moreover, the class contains similar properties and methods to **`Dictionary`**.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a sorted dictionary at https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2.

The exemplary application is as follows:

```
SortedDictionary<string, string> dictionary = new()
{
    { "Key #1", "Value #1" },
    { "Key #2", "Value #2" }
};
dictionary.Add("Key #3", "Value #3");
dictionary["Key #4"] = "Value #4";
string value = dictionary["Key #1"];
dictionary.TryGetValue("Key #2", out string? result);
```

The class is equipped with various properties, such as getting the number of stored elements (**`Count`**), as well as returning the collection of keys and values (**`Keys`** and **`Values`**, respectively). Moreover, you can use the available methods, which include the following:

- **`Add`**, which adds a new element

- **`Remove`**, which removes an item
- **`Clear`**, which removes all elements
- **`ContainsKey`**, which checks whether the collection contains a particular key
- **`ContainsValue`**, which checks whether the collection contains a given value
- **`TryGetValue`**, which tries to get a value for a given key

If you want to iterate through all the pairs stored in the collection, you can use the **`foreach`** loop. The variable that's used in the loop is an instance of **`KeyValuePair`** with **`Key`** and **`Value`** properties, allowing you to access the key and the value.

WHAT ABOUT PERFORMANCE?

*Despite the automatic sorting advantages, the **`SortedDictionary`** class has some performance drawbacks compared to **`Dictionary`** because retrieval, insertion, and removal are the O(log n) operations, where n is the number of elements in the collection, instead of O(1).*

*Moreover, **`SortedDictionary`** is quite similar to **`SortedList`**, as described in Chapter 3, Arrays and Sorting. However, it differs in memory-related and performance-related results. The retrieval for both these classes is the O(log n) operation, but insertion and removal for unsorted data is O(log n) for **`SortedDictionary`** and O(n) for **`SortedList`**. Of course, more memory is necessary for **`SortedDictionary`** than for **`SortedList`**. As you can see, choosing a proper data structure is not an easy task and you should think carefully about the scenarios in which data structures are used and take into account both the pros and cons.*

With this short introduction, let's see the sorted dictionary in action.

Example – encyclopedia

As an example, let's create a simple encyclopedia where you can add entries and show their full content. The encyclopedia can contain millions of entries, so it is crucial to provide its users with the possibility of browsing entries in the correct order, alphabetically by keys, as well as finding entries quickly. For this reason, the sorted dictionary is a good choice.

The concept of the encyclopedia is shown here:

```
NAME     -> EXPLANATION
Lancut   -> A city located near Rzeszow, with a castle.
Rzeszow  -> A capital of the Subcarpathian voivodeship.
Warszawa -> A capital city of Poland.
Zakopane -> A city located in Tatra mountains in Poland.
```

When the program is launched, it presents a simple menu with two options, namely **`[A]dd`** and **`[L]ist`**. After pressing *A*, the application asks you to enter the key and explanation for the entry. If the provided data is correct, a new entry is added to the encyclopedia. If the user presses *L*, data of all

entries, sorted by keys, is presented in the console. When any other key is pressed, additional confirmation is shown and, if confirmed, the program exits.

Let's take a look at the code:

```
Console.WriteLine("Welcome to your encyclopedia!\n");
SortedDictionary<string, string> definitions = [];
do
{
    Console.WriteLine("\nChoose option ([A]dd, [L]ist): ");
    ConsoleKeyInfo keyInfo = Console.ReadKey(true);
    if (keyInfo.Key == ConsoleKey.A)
    {
        Console.Write("Enter the key: ");
        string key = Console.ReadLine() ?? string.Empty;
        Console.Write("Enter the explanation: ");
        string explanation = Console.ReadLine()
            ?? string.Empty;
        definitions[key] = explanation;
    }
    else if (keyInfo.Key == ConsoleKey.L)
    {
        foreach ((string k, string e) in definitions)
        {
            Console.WriteLine($"{k}: {e}");
        }
    }
    else
    {
        Console.WriteLine("Do you want to exit? Y or N.");
        if (Console.ReadKey().Key == ConsoleKey.Y)
        {
            break;
        }
    }
}
while (true);
```

First, a new instance of the **`SortedDictionary`** class is created, which represents a collection of pairs with **`string`**-based keys and **`string`**-based values. Then, the infinite **`do-while`** loop is used. Within it, the program waits until the user presses any key. If it is the *A* key, a key and an explanation of the entry are obtained from the values entered by the user. Then, a new entry is added to the dictionary using the indexer. Thus, if the entry with the same key already exists, it will be updated. In the case of pressing the *L* key, a **`foreach`** loop is used to show all entered entries. When any other key is pressed, another question is presented to the user and the program waits for confirmation. If the user presses *Y*, you break out of the loop.

When you run the program, you can enter a few entries, as well as present them. The result from the console can be similar to what's shown in the following block:

```
Welcome to your encyclopedia!
Choose option ([A]dd, [L]ist):
Enter the key: Zakopane
Enter the explanation: A city located in Tatra mountains in Poland.
```

```
Choose option ([A]dd, [L]ist):
Enter the key: Rzeszow
Enter the explanation: A capital of the Subcarpathian voivodeship.
Choose option ([A]dd, [L]ist):
Rzeszow: A capital of the Subcarpathian voivodeship.
Zakopane: A city located in Tatra mountains in Poland.
Choose option ([A]dd, [L]ist):
Do you want to exit? Y or N.
```

So far, you've learned about three dictionary-related classes, namely `Hashtable`, `Dictionary`, and `SortedDictionary`. To make understanding them easier, a few examples were presented, together with detailed explanations.

However, do you know that some other data structures store just keys, without values? You will learn about these in the next sections.

Hash sets

In many algorithms, it is necessary to perform operations on sets with various data. However, what is a **set**? **A set is a collection of distinct objects without duplicated elements and without a particular order.** Therefore, you can only get to know whether a given element is in the set or not. These sets are strictly connected with mathematical models and operations, such as union, intersection, subtraction, and symmetric difference. A set can store various data, such as integer or string values. Of course, you can also create a set with instances of a user-defined class or record, as well as add and remove elements from the set at any time.

IMAGINE A HASH SET

If you want to better visualize a hash set, think for a moment about the game of chance, which is popular in many countries and involves selecting a few numbers that are then drawn from among many available ones. Depending on how many numbers you got from those drawn, you receive a prize. Of course, the chance of matching all the drawn numbers is very, very small. Now, you may be wondering where these collections are here. I'm in a hurry to answer this! There are three sets here, namely a set of all available numbers, a set of randomly drawn numbers, and a set of numbers selected by you. Each set cannot contain any duplicates. Of course, both the set of randomly drawn numbers and the set of numbers you choose are subsets of the set of all available numbers. How can you check how many numbers you have chosen correctly? It is very easy! Just perform the "intersection" operation on two sets, namely the set of randomly selected numbers and the set of numbers you selected, to obtain the result set. Now, all you have to do is keep your fingers crossed that the number of elements in this result set matches the number of elements in the set of drawn numbers. If that happens, then... you may become very rich because you matched all the drawn numbers. If so, congratulations!

Two exemplary sets are presented in the following figure:

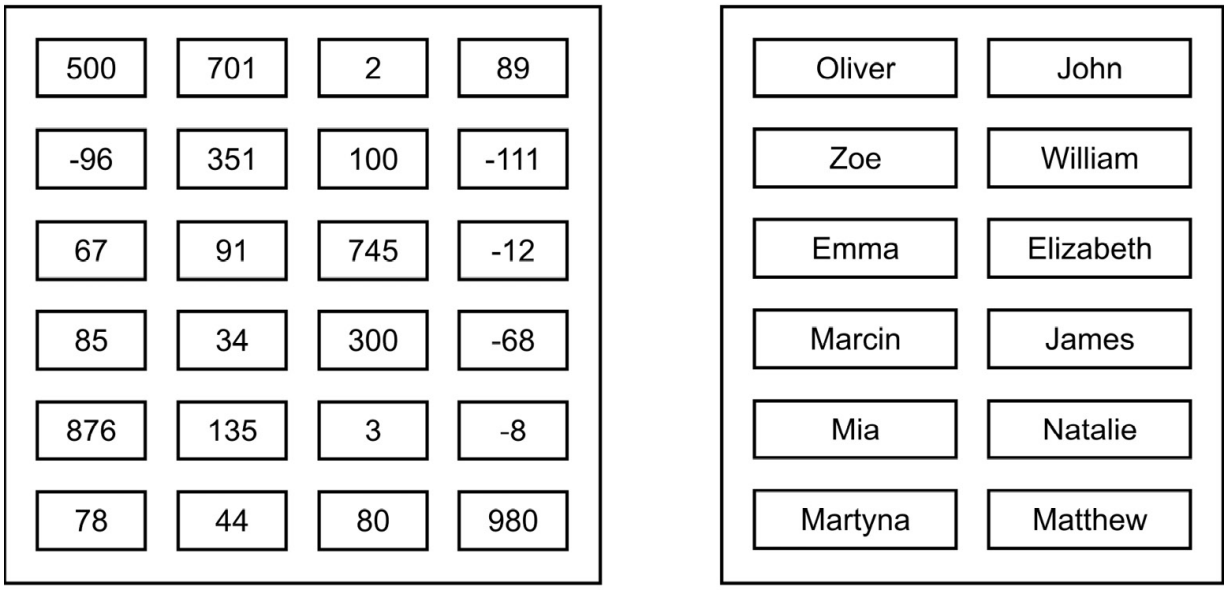

Figure 6.2 – Illustration of sets with integer and string values

Before seeing sets in action, it is a good idea to remind you of some basic operations that can be performed on two sets, named **A** and **B**.

The **union** (shown on the left of the following figure as **A∪B**) is **a set with all elements that belong to A or B**. The **intersection** (presented on the right as **A∩B**) **contains only the elements that belong to both A and B**:

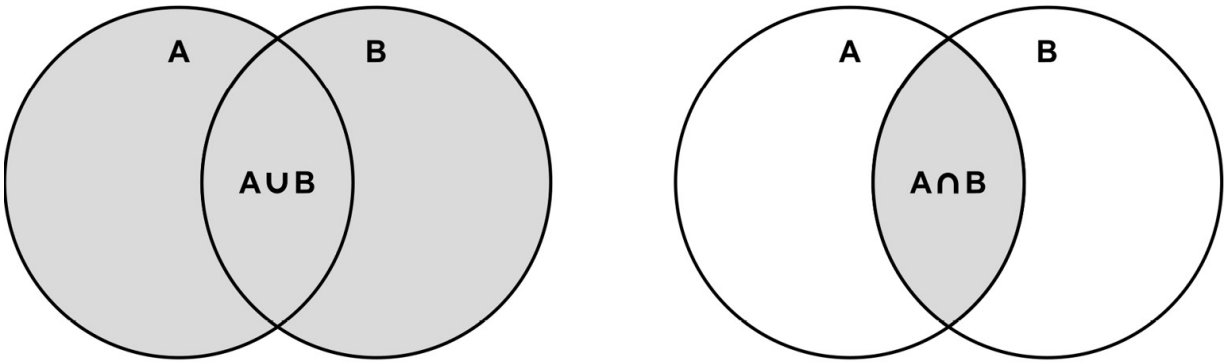

Figure 6.3 – Illustration of set union and intersection

Another common operation is set **subtraction**. **The result set of A \ B contains elements that are the members of A and not the members of B**. In the following figure, two examples are shown, namely **A \ B** and **B \ A**:

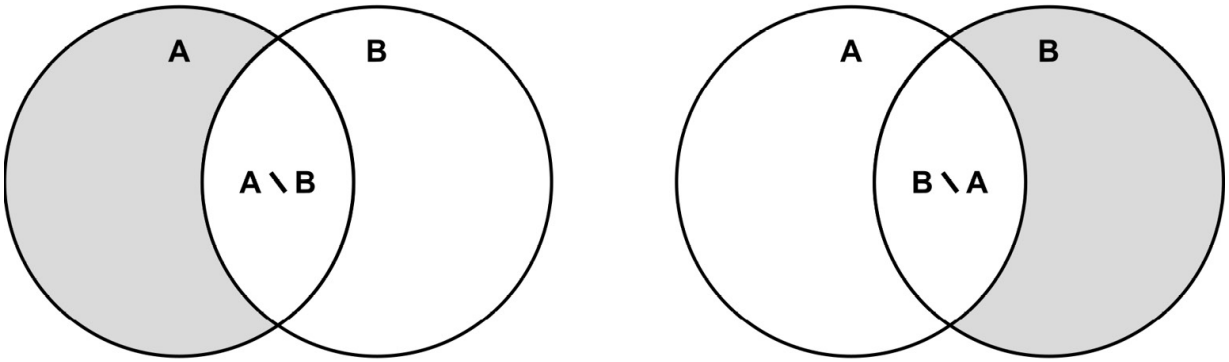

Figure 6.4 – Illustration of set subtraction

While performing various operations on sets, it is also worth mentioning the **symmetric difference**, shown in the following figure as **A** Δ **B**. The final set can be interpreted as a union of two sets, namely (**A** \ **B**) and (**B** \ **A**). Therefore, **it contains elements that belong to only one set, either A or B**. The elements that belong to both sets are excluded from the result:

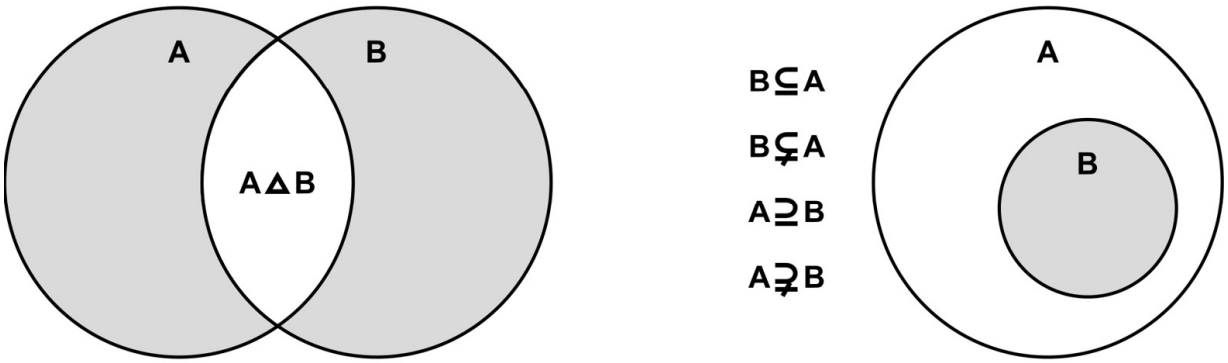

Figure 6.5 – Illustration of set symmetric difference and the relationships between sets

Another important topic is the **relationship** between sets. **If every element of B belongs to A, this means that B is a subset of A**. This is shown in the preceding diagram, on the right. At the same time, **A is a superset of B**. Moreover, **if B is a subset of A, but B is not equal to A, B is a proper subset of A**, and **A is a proper superset of B**.

While developing various kinds of applications in the C# language, you can benefit from high-performance operations provided by the `HashSet` class from the `System.Collections.Generic` namespace. The class contains a few properties, including `Count`, which returns the number of elements in the set.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a hash set at https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1.

Moreover, you can use many methods to perform operations of sets. The first group of methods makes it possible to modify the current set (on which the method is called) to create the union (`UnionWith`), the intersection (`IntersectWith`), the subtraction (`ExceptWith`), and the symmetric difference (`SymmetricExceptWith`) with the set passed as the parameter.

You can also check the relationship between two sets, such as checking whether the current set (on which the method is called) is a subset (`IsSubsetOf`), a superset (`IsSupersetOf`), a proper subset (`IsProperSubsetOf`) or a proper superset (`IsProperSupersetOf`) of the set passed as the parameter.

Furthermore, you can verify whether two sets contain the same elements (`SetEquals`) or whether two sets have at least one common element (`Overlaps`).

Apart from these operations, you can add a new element to the set (`Add`), remove a particular element (`Remove`), remove all elements (`Clear`), and check whether the given element exists in the set (`Contains`).

WHAT ABOUT PERFORMANCE?

Hash sets make it possible to perform a quick lookup for a given item. Thus, checking whether the set contains an item and removing an item are O(1) operations. As for addition, it is an O(1) operation if it does not need to increase the internal array. If resizing is necessary, it turns out to be the O(n) operation, where n is the number of items.

With this introduction, try to put what you've learned into practice. Now, let's proceed to two examples that show how you can apply hash sets in your applications.

Example – coupons

The first example represents a system that checks whether a one-time coupon has already been used. If so, a suitable message is presented to the user. Otherwise, the system informs the user that the coupon is valid. It is then marked as used and cannot be used again. Due to the high number of coupons, it is necessary to choose a data structure that allows you to quickly check whether an element exists in a collection. For this reason, a hash set has been chosen as a data structure for storing identifiers of used coupons.

Let's take a look at the code:

```
HashSet<int> usedCoupons = [];
do
{
    Console.Write("Enter the number: ");
    string number = Console.ReadLine() ?? string.Empty;
    if (!int.TryParse(number, out int coupon)) { break; }
    if (usedCoupons.Contains(coupon))
```

```
        {
            Console.WriteLine("Already used.");
        }
        else
        {
            usedCoupons.Add(coupon);
            Console.WriteLine("Thank you!");
        }
    }
    while (true);
```

First, a new instance of **`HashSet`** that stores integer values is created. Then, the majority of operations are performed within the **`do-while`** loop. Here, the program waits until the user enters the coupon identifier. If it cannot be parsed to the integer value, you break out of the loop. Otherwise, you check whether the set already contains the coupon identifier (using the **`Contains`** method). If so, the suitable information is presented in the console. If it does not exist, you add it to the collection of used coupons (using the **`Add`** method) and inform the user about the result.

When you break out of the loop, you just need to show the complete list of identifiers of the used coupons. You can do so using a **`foreach`** loop, iterating over the set, and writing its elements in the console, as shown in the following code:

```
Console.WriteLine("\nUsed coupons:");
foreach (int coupon in usedCoupons)
{
    Console.WriteLine(coupon);
}
```

Now, you can launch the application, enter some data, and see how it works:

```
Enter the number: 100
Thank you!
Enter the number: 101
Thank you! (...)
Enter the number: 101
Already used.
Enter the number:
Used coupons:
100
101
500
345
```

This is the end of the first example. Let's proceed to the next one, where you will see a more complex solution that uses a hash set.

Example – swimming pools

This example presents the system for a SPA center with four swimming pools, namely recreation, competition, thermal, and for kids. Each visitor receives a special wristband that allows them to enter

all the pools. However, it is necessary to scan the wristband upon each user entering any pool. Your program uses such data to create various statistics.

A hash set has been chosen as a data structure for storing unique numbers of wristbands that are scanned at the entrance to each swimming pool. Four sets are used, one per pool. Moreover, they are grouped in the dictionary to simplify and shorten the code, as well as make future modifications easier. To simplify testing the application, the initial data is set randomly. Then, you create statistics, namely the number of visitors by pool type, the most popular pool, the number of people who visited at least one pool, as well as the number of people who visited all the pools.

Let's start with the **`PoolTypeEnum`** enumeration, which represents possible types of swimming pools:

```
enum PoolTypeEnum
{
    Recreation,
    Competition,
    Thermal,
    Kids
};
```

Then, open **`Program.cs`** and add the **`random`** variable. This will be used to fill the hash set with some random values. The line of code is as follows:

```
Random random = new();
```

In the next part of the code, you create a new instance of **`Dictionary`**. This contains four entries. Each key is of the **`PoolTypeEnum`** type and each value is of the **`HashSet<int>`** type – that is, a set with integer values. The code is shown here:

```
Dictionary<PoolTypeEnum, HashSet<int>> tickets = new()
{
    { PoolTypeEnum.Recreation, new() },
    { PoolTypeEnum.Competition, new() },
    { PoolTypeEnum.Thermal, new() },
    { PoolTypeEnum.Kids, new() }
};
```

After that, you fill the sets with random values, as shown here:

```
for (int i = 1; i < 100; i++)
{
    foreach ((PoolTypeEnum p, HashSet<int> t) in tickets)
    {
        if (random.Next(2) == 1) { t.Add(i); }
    }
}
```

To do so, you use two loops, namely **`for`** and **`foreach`**. The first iterates **`100`** times and simulates **`100`** wristbands. Within it, there is a **`foreach`** loop, which iterates through all available pool types. For

each of them, you randomly check whether a visitor entered a particular swimming pool. If so, the identifier is added to the proper set.

The remaining code is related to generating various statistics. First, let's present the number of visitors by pool type. Such a task is very easy because you just need to iterate through the dictionary, as well as write the pool type and the number of elements in the set (using the **`Count`** property), as shown here:

```
Console.WriteLine("Number of visitors by a pool type:");
foreach ((PoolTypeEnum p, HashSet<int> t) in tickets)
{
    Console.WriteLine($"- {p}: {t.Count}");
}
```

The next part finds the swimming pool with the maximum number of visitors. This is done using a few extension methods, namely to order the results by the number of elements in the set, in descending order (**`OrderByDescending`**), to choose only a pool type (**`Select`**), and to take only the first element (**`FirstOrDefault`**). Then, you just present the result. The code for doing this is shown here:

```
PoolTypeEnum maxVisitors = tickets
    .OrderByDescending(t => t.Value.Count)
    .Select(t => t.Key)
    .FirstOrDefault();
Console.WriteLine($"{maxVisitors} - the most popular.");
```

Next, you want to get the number of people who visited at least one pool. You perform this task by creating a union of all sets and getting the count of the final set. First, you create a new set and fill it with identifiers regarding the recreation swimming pool. In the following lines of code, you call the **`UnionWith`** method to create a union with the following three sets:

```
HashSet<int> any = new(tickets[PoolTypeEnum.Recreation]);
any.UnionWith(tickets[PoolTypeEnum.Competition]);
any.UnionWith(tickets[PoolTypeEnum.Thermal]);
any.UnionWith(tickets[PoolTypeEnum.Kids]);
Console.WriteLine($"{any.Count} people visited
    at least one pool.");
```

The last statistic is the number of people who visited all pools during one visit to the SPA center. You just need to create the intersection of all sets and get the count of the final set. To do so, you create a new set and fill it with identifiers regarding the recreation swimming pool. Then, you call the **`IntersectWith`** method to create an intersection with the following three sets. Finally, you get the number of elements in the set using the **`Count`** property and present the results, as follows:

```
HashSet<int> all = new(tickets[PoolTypeEnum.Recreation]);
all.IntersectWith(tickets[PoolTypeEnum.Competition]);
all.IntersectWith(tickets[PoolTypeEnum.Thermal]);
all.IntersectWith(tickets[PoolTypeEnum.Kids]);
Console.WriteLine($"{all.Count} people visited all pools.");
```

That's all! When you run the application, you may receive the following result:

```
Number of visitors by a pool type:
- Recreation: 60
- Competition: 40
- Thermal: 47
- Kids: 45
Recreation - the most popular.
91 people visited at least one pool.
10 people visited all pools.
```

You've just completed two examples regarding hash sets. It's a good idea to try to modify the code and add new features on your own to learn more about this data structure. In the next section, we'll look at "sorted" sets.

"Sorted" sets

The previously described `HashSet` class can be understood as a dictionary that stores only keys, without values. So, if there is the `SortedDictionary` class, maybe there is also the `SortedSet` class? There is! However, can a set be "sorted"? Why is "sorted" written with quotation marks? The answer turns out to be very simple. By definition, a set stores a collection of distinct objects without duplicated elements and without a particular order. If a set does not support order, how can it be "sorted"? For this reason, **a "sorted" set can be understood as a combination of HashSet and SortedList, not a set itself.**

IMAGINE A "SORTED" SET

If you want to better imagine a "sorted" set, recall the previous example related to the game of chance. To facilitate manual comparison of results, both the set of randomly drawn numbers and the set of numbers selected by you can be "sorted" and shown in ascending order. This is where a "sorted" set comes in handy. It's very simple and clear, don't you think?

A "sorted" set can be used if you want to have a **sorted collection of distinct objects without duplicated elements**. The suitable class is named `SortedSet` and is available in the `System.Collections.Generic` namespace.

WHERE CAN YOU FIND MORE INFORMATION?

You can find content regarding a "sorted" set at https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedset-1.

It has a set of methods, similar to those already described in the case of the `HashSet` class, including `UnionWith`, `IntersectWith`, `ExceptWith`, `SymmetricExceptWith`, `Overlaps`, `IsSubsetOf`, `IsSupersetOf`, `IsProperSubsetOf`, and `IsProperSupersetOf`.

It contains additional properties for returning the minimum and maximum values (`Min` and `Max`, respectively). It is also worth mentioning the `GetViewBetween` method since it returns a `SortedSet`

instance with values from the given range.

WHAT ABOUT PERFORMANCE?

The "sorted" set is an interesting data structure in terms of its performance. It is a kind of a trade-off between functionalities and performance. Thus, checking whether the collection contains an item as well as removing any item from the collection are O(log n) operations. For this reason, you should expect worse performance results compared to the data structures described earlier.

Let's proceed to a simple example to see how to use a "sorted" set in code.

Example – removing duplicates

As an example, you will create a simple application that removes duplicates from the list of names. Of course, the comparison of names should be case-insensitive, so it is not allowed to have both **`Marcin`** and **`marcin`** in the same collection.

To do this, we can use the following code:

```
List<string> names =
[
    "Marcin", "Mary", "James", "Albert", "Lily",
    "Emily", "marcin", "James", "Jane"
];
SortedSet<string> sorted = new(
    names,
    Comparer<string>.Create((a, b) =>
        a.ToLower().CompareTo(b.ToLower())));
foreach (string name in sorted)
{
    Console.WriteLine(name);
}
```

First, a list of names is created and initialized with nine elements, including **`Marcin`** and **`marcin`**. Then, a new instance of the **`SortedSet`** class is created and two parameters are passed to the constructor, namely the list of names and the case-insensitive comparer. Finally, the collection is iterated through so that you can write names in the console.

When you run the application, you'll see the following result:

```
Albert
Emily
James
Jane
Lily
Marcin
Mary
```

Do you know that you can use another variant of the **`SortedSet`** constructor and pass only the first parameter, namely the list, without the comparer? In such a case, the default comparer will be used and will be case-sensitive.

Congratulations – you've just completed the last example shown in this chapter!

Summary

This chapter focused on hash tables, dictionaries, and sets. All of these collections are interesting data structures that can be used in various scenarios during the development of many applications. By presenting such collections with detailed descriptions, performance explanations, and examples, you saw that choosing a proper data structure is not a trivial task and requires analyzing performance-related topics.

First, you learned how to use two variants of a **hash table**, namely non-generic and generic. The huge advantage of these is a very fast lookup for a value based on the key, which is the close *O(1)* operation. To achieve this goal, the hash function is used. Moreover, the **sorted dictionary** was introduced as an interesting solution to solve the problem of unsorted items in the collection and to keep keys sorted all the time.

Afterward, the high-performance solution to **set operations** was presented. A set can be understood as a collection of distinct objects without duplicated elements and without a particular order. The class that was shown makes it possible to perform various operations on sets, such as union, intersection, subtraction, and symmetric difference. Then, the concept of the **"sorted" set** was introduced as a sorted collection of distinct objects without duplicated elements.

Do you want to dive deeper into the topic of data structures and algorithms while developing applications in the C# language? If so, proceed to the next chapter, where **trees** are presented.

7

Variants of Trees

In the previous chapters, you learned about many data structures, starting with simple ones such as arrays. Now, it is time for you to get to know a significantly more complex group of data structures, namely **trees**.

At the beginning of this chapter, a **basic tree** will be presented, together with its implementation in the C# language, and with some examples showing it in action. Then, a **binary tree** will be introduced, with a detailed description of its implementation and an example of its application. A **binary search tree** (**BST**) is another tree variant and is one of the most popular types of trees, used in many algorithms. You will also cover **self-balancing trees**, namely **AVL** and **red-black trees** (**RBTs**). Then, you will see a **trie** as a specialized data structure for performing operations on strings. The remaining part of the chapter is dedicated to a short introduction to the topic of **heaps**.

Arrays, lists, stacks, queues, dictionaries, sets, and now trees. Are you ready to increase the level of difficulty and learn the next data structures? If so, let's start reading!

In this chapter, the following topics will be covered:

- Basic trees
- Binary trees
- Binary search trees
- Self-balancing trees
- Tries
- Heaps

Basic trees

Let's start with introducing trees. What are they? Do you have any ideas about how such a data structure should look? If not, let's take a look at the following diagram, which depicts a tree with captions regarding its particular elements:

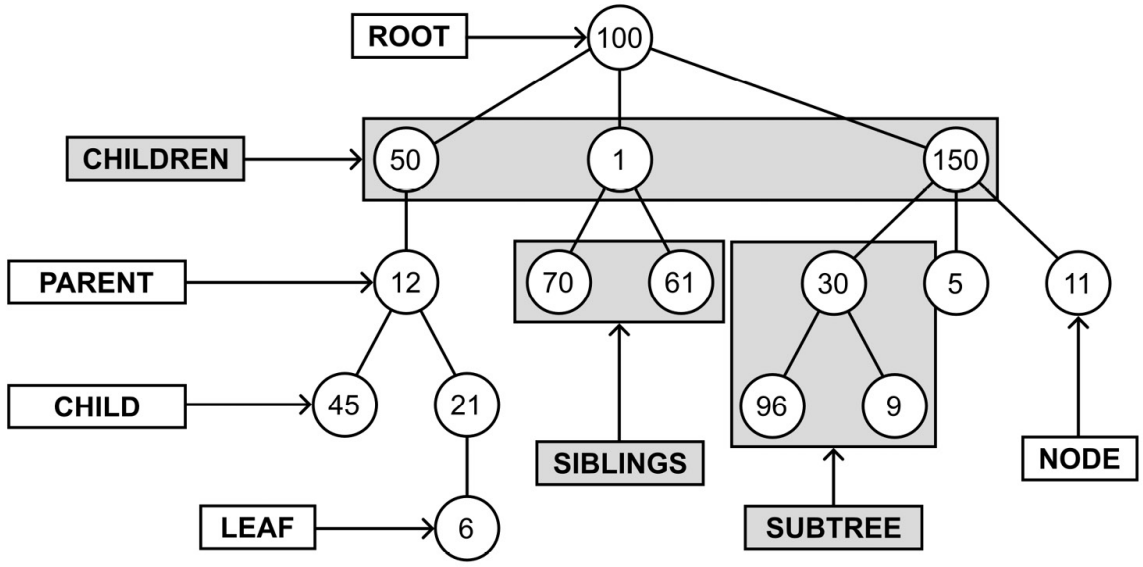

Figure 7.1 – Illustration of a tree

A tree consists of multiple **nodes**, including one **root** (**100** in the diagram). The root does not contain a **parent** node, while all other nodes do. For example, the parent element of node **1** is **100**, while node **96** has node **30** as the parent.

Moreover, each node can have any number of **child** nodes, such as three **children** (that is, **50**, **1**, and **150**) in the case of the **root**. The child nodes of the same node can be named **siblings**, as in the case of nodes **70** and **61**. A node without children is named a **leaf**, such as **45** and **6** in the diagram.

Let's take a look at the rectangle with three nodes (that is, **30**, **96**, and **9**). Such a part of the tree can be called a **subtree**. Of course, you can find many subtrees in the tree.

IMAGINE A TREE

*If you want to better imagine a tree, look at the structure of a slightly larger company, where at the very top of the hierarchy there is the **chief executive officer** (**CEO**), to whom the **chief operating officer** (**COO**), **chief marketing officer** (**CMO**), **chief financial officer** (**CFO**), and **chief technology officer** (**CTO**) are assigned. As sales is one of the key topics in the company's operations, regional directors report to the COO, and for each of them, between three and five sales specialists are assigned. Look for yourself – you have a tree in your mind right now! Its root is the CEO, which has four children (COO, CMO, CFO, and CTO), which can have further child nodes to create subsequent levels of the hierarchy. Sales specialists who no longer have any subordinates are named leaves.*

Let's briefly talk about the minimum and maximum numbers of children of a node. In general, such numbers are not limited, and each node can contain zero, one, two, three, or even more children. However, in practical applications, the number of children is often limited to two, as you will see soon.

Implementation

The C#-based implementation of a basic tree seems to be quite obvious and not complicated. To do so, you declare two classes, representing a single node and a whole tree, as described in this section.

Node

The first class is named **`TreeNode`** and is declared as a generic class to provide a developer with the ability to specify a type of data stored in each node. Thus, you can create a strongly typed solution, which eliminates the necessity of casting objects to target types. The code is as follows:

```
public class TreeNode<T>
{
    public T? Data { get; set; }
    public TreeNode<T>? Parent { get; set; }
    public List<TreeNode<T>> Children { get; set; } = [];
    public int GetHeight()
    {
        int height = 1;
        TreeNode<T> current = this;
        while (current.Parent != null)
        {
            height++;
            current = current.Parent;
        }
        return height;
    }
}
```

The class contains three properties:

- The data stored in the node (**`Data`**)
- A reference to the parent node (**`Parent`**)
- A collection of references to child nodes (**`Children`**)

Apart from the properties, the **`TreeNode`** class contains the **`GetHeight`** method, which returns the height of the node – that is, the distance from this node to the root node. The implementation of this method is very simple because it just uses a **`while`** loop to go up from the node until there is no parent element, which means that the root is reached.

Tree

The next necessary class is named **`Tree`**. It represents the whole tree, as follows:

```
public class Tree<T>
{
    public TreeNode<T>? Root { get; set; }
}
```

The class contains only one property, **`Root`**. You can use this property to get access to the root node, and then you can use its **`Children`** property to obtain data of its child nodes. Then, you can take a look

at each of them and get data of their child nodes, as well. By repeating such operations, you can get data from all nodes located in the tree.

It is worth noting that both **`TreeNode`** and **`Tree`** classes are generic, and the same type is used in the case of these classes. For instance, if tree nodes should store **`string`** values, the **`string`** type should be used for instances of **`Tree`** and **`TreeNode`** classes.

Example – hierarchy of identifiers

Do you want to see how to use a tree in a C#-based application? Let's take a look at our first example. The aim is to construct a tree with a few nodes, as shown in the following diagram. Only the group of nodes with a bolder border will be presented in the code. However, it is a good idea to adjust the code to construct the whole tree by yourself:

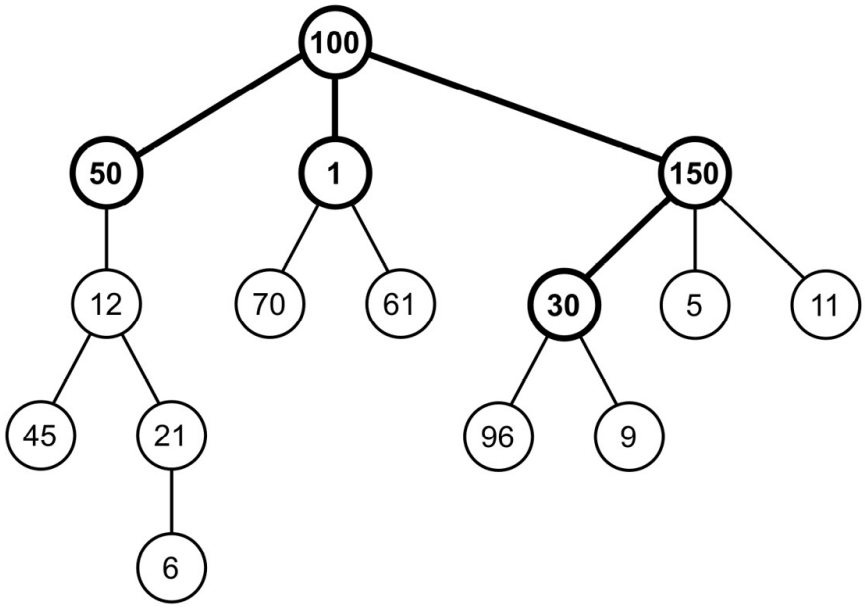

Figure 7.2 – Illustration of the hierarchy of identifiers example

Here, each node stores an integer value, so **`int`** is the type used for both the **`Tree`** and **`TreeNode`** classes. The following code should be placed in the **`Program.cs`** file:

```
Tree<int> tree = new() { Root = new() { Data = 100 } };
tree.Root.Children =
[
    new() { Data = 50, Parent = tree.Root },
    new() { Data = 1, Parent = tree.Root },
    new() { Data = 150, Parent = tree.Root }
];
tree.Root.Children[2].Children =
```

```
[
    new() { Data = 30, Parent = tree.Root.Children[2] }
];
```

The code looks quite simple, doesn't it? At the beginning, a new instance of the `Tree` class is created and the root node is configured by creating a new instance of the `TreeNode` class and setting a value of the `Data` property to `100`.

In the following lines, the child nodes of the root node are specified, namely the nodes with values equal to `50`, `1`, and `150`. For each of them, a value of the `Parent` property is set to a reference to the previously added root node.

The remaining part of the code shows how to add a child node for a given node, namely for the third child of the root node – that is, the node with a value equal to `150`. Here, only one node is added: the one with the value set to `30`. Of course, you need to specify a reference to the parent node, as well.

That's all! You created the first program that uses trees. Now, you can run it, but you will not see any output in the console. If you want to see how the data of nodes is organized, you can debug the program and see the values of variables while debugging.

Example – company structure

In the previous example, you saw how to use integer values as data stored in each node in a tree. However, it is also possible to store instances of user-defined classes in nodes. In this example, you will see how to create a tree presenting the structure of a company, divided into three main departments: development, research, and sales.

Within each department, there can be another structure, such as in the case of the development team. Here, **John Smith** is head of development. He is a boss for **Chris Morris**, who is a manager for two junior developers, **Eric Green** and **Ashley Lopez**. The latter is also a supervisor of **Emily Young**, who is a developer intern.

An example tree is shown in the following diagram:

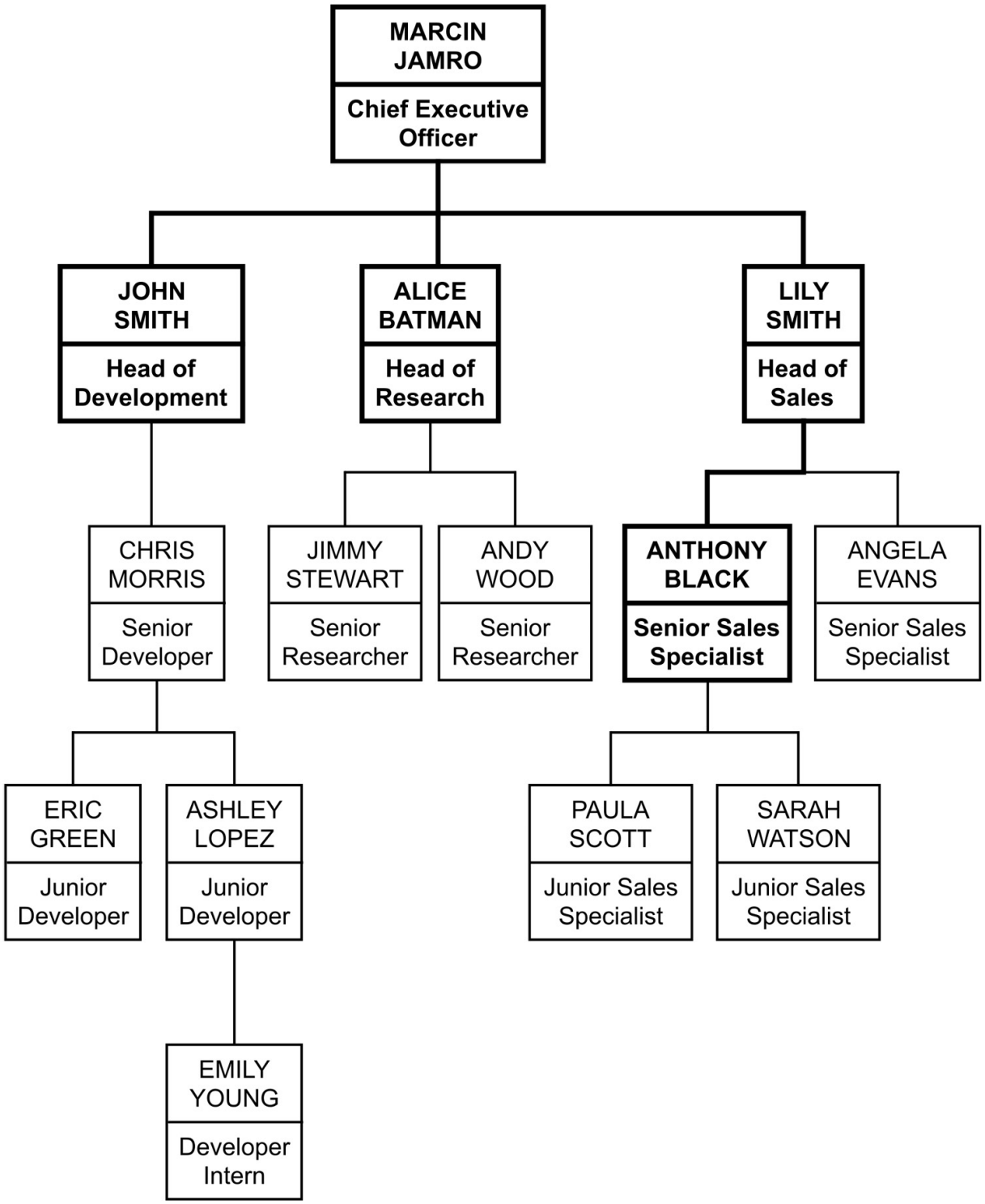

Figure 7.3 – Illustration of the company structure example

As you can see, each node should store more information than just an integer value. There should be a name and a role. Such data is stored as values of properties in an instance of the `Person` record,

which is shown next:

```
public record Person(string Name, string Role);
```

Apart from creating a new record, it is also necessary to add some code:

```
Tree<Person> company = new()
{
    Root = new()
    {
        Data = new Person("Marcin Jamro",
            "Chief Executive Officer"),
        Parent = null
    }
};
company.Root.Children =
[
    new() { Data = new Person("John Smith",
        "Head of Development"), Parent = company.Root },
    new() { Data = new Person("Alice Batman",
        "Head of Research"), Parent = company.Root },
    new() { Data = new Person("Lily Smith",
        "Head of Sales"), Parent = company.Root }
];
company.Root.Children[2].Children =
[
    new() { Data = new Person("Anthony Black",
        "Senior Sales Specialist"),
        Parent = company.Root.Children[2] }
];
```

In the first line, a new instance of the `Tree` class is created. It is worth mentioning that the `Person` record is used as a type specified while creating new instances of the `Tree` and `TreeNode` classes. Thus, you can easily store more than one simple data type for each node. The remaining lines of code look similar to the first example for basic trees. Here, you also specify the root node (for the `Chief Executive Officer` role), then configure its child elements (`John Smith`, `Alice Batman`, and `Lily Smith`), and set a child node for one of the existing nodes, namely the node for the `Head of Sales` role.

Does it look simple and straightforward? In the next section, you will see a more restricted, but very important and well-known variant of trees: a binary tree.

Binary trees

Generally speaking, each node in a basic tree can contain any number of children. However, in the case of **binary trees**, **a node cannot contain more than two children**. It means that **it can contain zero, one, or two child nodes**. Such a requirement has an important impact on the shape of a binary tree, as shown in the following two diagrams presenting binary trees:

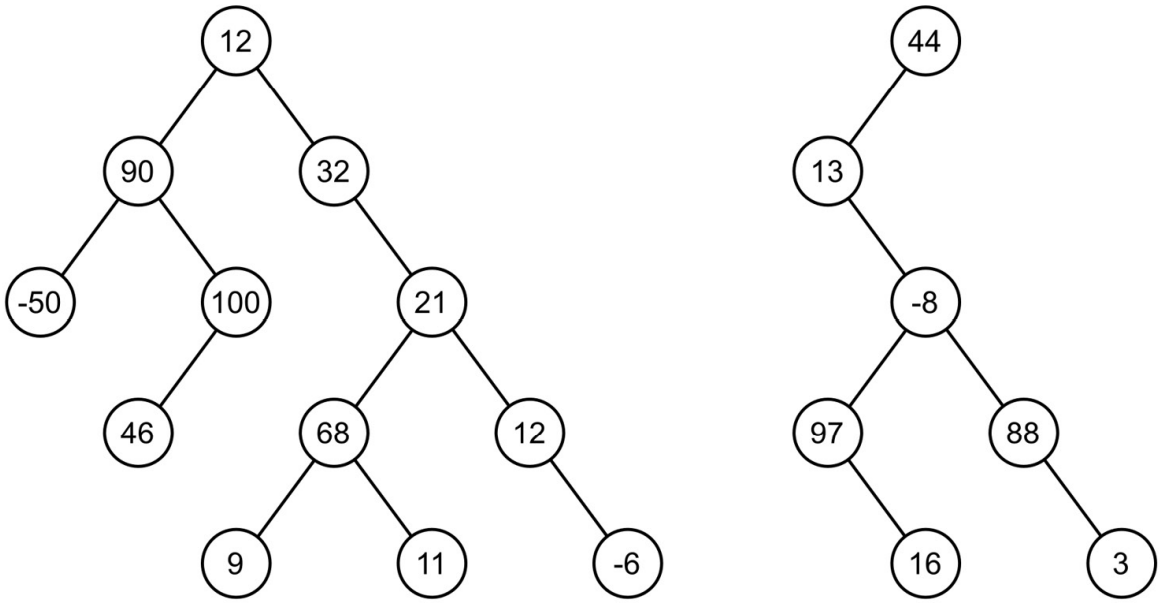

Figure 7.4 – Illustration of binary trees

As already mentioned, a node in a binary tree can contain at most two children. For this reason, they are referred to as the **left child** and the **right child**. In the case of the binary tree shown on the left-hand side of the preceding diagram, node **21** has two children, namely **68** as the left child and **12** as the right child, while node **100** has only a left child.

Traversal

Have you thought about how you can iterate through all the nodes in a tree? How can you specify an order of nodes during **traversal** of a tree? There are three common approaches, namely **pre-order**, **in-order**, and **post-order**, as shown next:

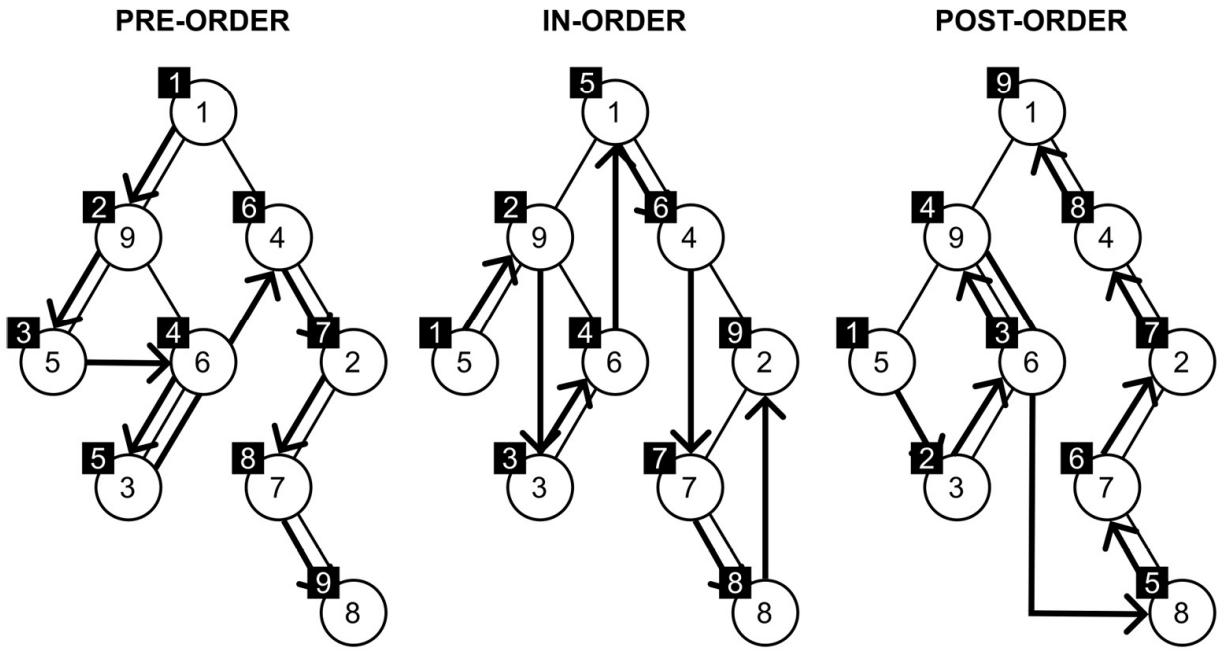

Figure 7.5 – Pre-order, in-order, and post-order traversal

As you can see in the diagram, there are visible differences between the approaches. However, do you have any idea how you can apply pre-order, in-order, or post-order traversals for binary trees? Let's explain all of these approaches in detail.

Pre-order

If you want to traverse a binary tree with the **pre-order** approach, you first visit the root node. Then, you visit the left child. Finally, the right child is visited. Of course, such a rule does not apply only to the root node, but to any node in a tree. For this reason, you can understand the order of pre-order traversal as **first visiting the current node, then its left child (the whole left subtree using the pre-order approach recursively), and finally its right child (the right subtree in a similar way)**.

The explanation can sound a bit complicated, so let's take a look at a simple example regarding the tree shown on the left of the preceding diagram. First, the root node (that is, **1**) is visited. Then, you analyze its left child node. For this reason, the next visited node is the current node, **9**. The next step is the pre-order traversal of its left child. Thus, **5** is visited. As this node does not contain any children, you can return to the stage of traversing when **9** is the current node. It has already been visited, as has its left child node, so it is time to proceed to its right child. Here, you first visit the current node, **6**, and follow to its left child, **3**. You can apply the same rules to continue traversing the tree. The final order is **1**, **9**, **5**, **6**, **3**, **4**, **2**, **7**, **8**.

If it still sounds a bit confusing, the following diagram should remove any doubts:

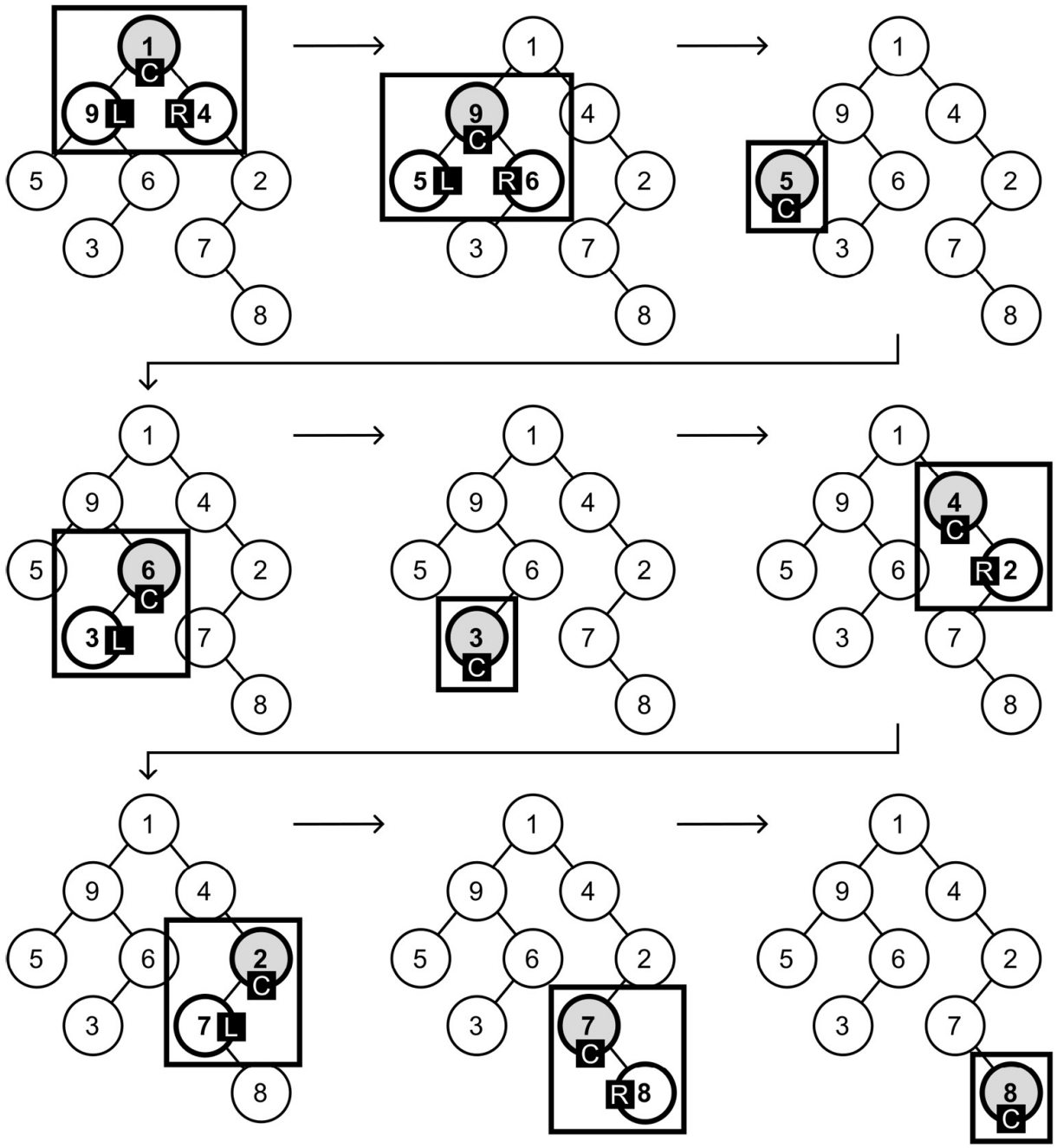

Figure 7.6 – Detailed diagram of pre-order traversal

The diagram presents the following steps of the pre-order traversal with additional indicators: **C** for the **current node**, **L** for the **left child**, and **R** for the **right child**.

In-order

The second traversal mode is called **in-order**. It differs from the pre-order approach in the order that nodes are visited: **the left child, the current node, and then the right child**.

If you take a look at the example shown in the diagram with all three traversal modes, you can see that the first visited node is **5**. Why? At the beginning, the root node is analyzed, but it is not visited because the in-order traversal starts with the left child node. Thus, it analyzes node **9**, but it also has a left child, **5**, so you proceed to this node. As this node does not have any children, the current node (**5**) is visited. Then, you return to the step when the current node is **9**, and, as its left child was already visited, you also visit the current node. Next, you follow to the right child, but it has a left child, **3**, which should be visited first. According to the same rules, you visit the remaining nodes in the binary tree. The final order is **5**, **9**, **3**, **6**, **1**, **4**, **7**, **8**, **2**.

Post-order

The last traversal mode is named **post-order** and supports the following order of node traversal: **the left child, the right child, and then the current node**.

Let's analyze the post-order example shown on the right-hand side of the diagram. At the beginning, the root node is analyzed but it is not visited because the post-order traversal starts with the left child node. Thus, you proceed to node **9**, then **5**, which you visit first. Next, you need to analyze the right child of node **9**. However, node **6** has the left child (**3**), which should be visited earlier. For this reason, after **5**, you visit **3**, and then **6**, followed by **9**. What is interesting is that the root node of the binary tree is visited at the end. The final order is **5**, **3**, **6**, **9**, **8**, **7**, **2**, **4**, **1**.

WHAT ABOUT THE PERFORMANCE?

If you want to check whether a binary tree contains a given value, you need to check each node, traversing the tree using one of three available modes: pre-order, in-order, or post-order. This means that the lookup time is linear, namely O(n).

After this short introduction, let's proceed to the C#-based implementation.

Implementation

The implementation of a binary tree is simple, especially if you use the already described code for the basic tree. Let's start with a class representing a node.

Node

A node in a binary tree is represented by an instance of **`BinaryTreeNode`**, which inherits from the **`TreeNode`** generic class. In the **`BinaryTreeNode`** class, it is necessary to hide the **`Children`** definition from the base class, as well as declare two properties, **`Left`** and **`Right`**, which represent both possible children of a node. The relevant part of the code is as follows:

```
public class BinaryTreeNode<T>
    : TreeNode<T>
{
    public new BinaryTreeNode<T>?[] Children { get; set; }
        = [null, null];
```

```
    public BinaryTreeNode<T>? Left
    {
        get { return Children[0]; }
        set { Children[0] = value; }
    }
    public BinaryTreeNode<T>? Right
    {
        get { return Children[1]; }
        set { Children[1] = value; }
    }
}
```

Moreover, you need to ensure that the array with child nodes contains exactly two items, initially set to **`null`**. Thus, if you want to add a child node, a reference to it should be placed as the first or the second element of the array from the **`Children`** property. Therefore, such an array always has exactly two elements, and you can access the first or the second one without any exception. If such an element is set to any node, a reference to it is returned. Otherwise, **`null`** is returned.

Tree

The next necessary class is named **`BinaryTree`**. It represents the whole binary tree. By using the generic class, you can easily specify the type of data stored in each node. The first part of the implementation of the **`BinaryTree`** class is as follows:

```
public class BinaryTree<T>
{
    public BinaryTreeNode<T>? Root { get; set; }
    public int Count { get; set; }
}
```

The **`BinaryTree`** class contains two properties:

- **`Root`** indicates the root node (instance of the **`BinaryTreeNode`** class)
- **`Count`** stores the total number of nodes placed in the tree

Of course, these are not the only members of the class because it is also equipped with a set of methods regarding traversing the tree. The first traversal method is **pre-order**. As a reminder, it first visits the current node, then its left child, followed by the right child. The code of the **`TraversePreOrder`** method is as follows:

```
private void TraversePreOrder(BinaryTreeNode<T>? node,
    List<BinaryTreeNode<T>> result)
{
    if (node == null) { return; }
    result.Add(node);
    TraversePreOrder(node.Left, result);
    TraversePreOrder(node.Right, result);
}
```

The method takes two parameters: the current node (**`node`**) and the list of already visited nodes (**`result`**). The recursive implementation is very simple. First, you check whether the node exists by

ensuring that the parameter is not equal to **`null`**. Then, you add the current node to the collection of visited nodes, start the same traversal method for the left child, and then start it for the right child.

Similar implementation is possible for the **in-order** and post-order traversal modes. Let's start with the code of the **`TraverseInOrder`** method, as follows:

```
private void TraverseInOrder(BinaryTreeNode<T>? node,
    List<BinaryTreeNode<T>> result)
{
    if (node == null) { return; }
    TraverseInOrder(node.Left, result);
    result.Add(node);
    TraverseInOrder(node.Right, result);
}
```

Here, you call the **`TraverseInOrder`** method for the left child, add the current node to the list of visited nodes, and start the in-order traversal for the right child.

The next method is related to the **post-order** traversal mode, as follows:

```
private void TraversePostOrder(BinaryTreeNode<T>? node,
    List<BinaryTreeNode<T>> result)
{
    if (node == null) { return; }
    TraversePostOrder(node.Left, result);
    TraversePostOrder(node.Right, result);
    result.Add(node);
}
```

The code is similar to the already described methods, but, of course, another order of visiting nodes is applied. Here, you start with the left child, then you visit the right child, followed by adding the current node to the list of visited nodes.

Finally, let's add a public method for traversing the tree in various modes, which calls the private methods presented earlier. The relevant code is as follows:

```
public List<BinaryTreeNode<T>> Traverse(TraversalEnum mode)
{
    List<BinaryTreeNode<T>> nodes = [];
    if (Root == null) { return nodes; }
    switch (mode)
    {
        case TraversalEnum.PreOrder:
            TraversePreOrder(Root, nodes);
            break;
        case TraversalEnum.InOrder:
            TraverseInOrder(Root, nodes);
            break;
        case TraversalEnum.PostOrder:
            TraversePostOrder(Root, nodes);
            break;
    }
    return nodes;
}
```

The method takes only one parameter, namely a value of the **`TraversalEnum`** enumeration, which chooses the proper mode from pre-order, in-order, and post-order. The **`Traverse`** method uses a **`switch`** statement to call a suitable private method, depending on the value of the parameter. The mentioned enumeration is as follows:

```
public enum TraversalEnum { PreOrder, InOrder, PostOrder }
```

The last method from the **`BinaryTree`** class is **`GetHeight`**. It returns the height of the tree, which can be understood as the maximum number of steps to travel from any leaf node to the root. The implementation is as follows:

```
public int GetHeight() => Root != null
    ? Traverse(TraversalEnum.PreOrder)
        .Max(n => n.GetHeight())
    : 0;
```

After the introduction to the topic of binary trees, let's see an example where this data structure is used for storing questions and answers in a simple quiz.

Example – simple quiz

As an example of a binary tree, a simple quiz application will be prepared. The quiz consists of a few questions and answers, shown depending on previously taken decisions. The application presents the question, waits until the user presses *Y* (yes) or *N* (no), and proceeds to the next question or shows the answer.

The structure of the quiz is created in the form of a binary tree, as follows:

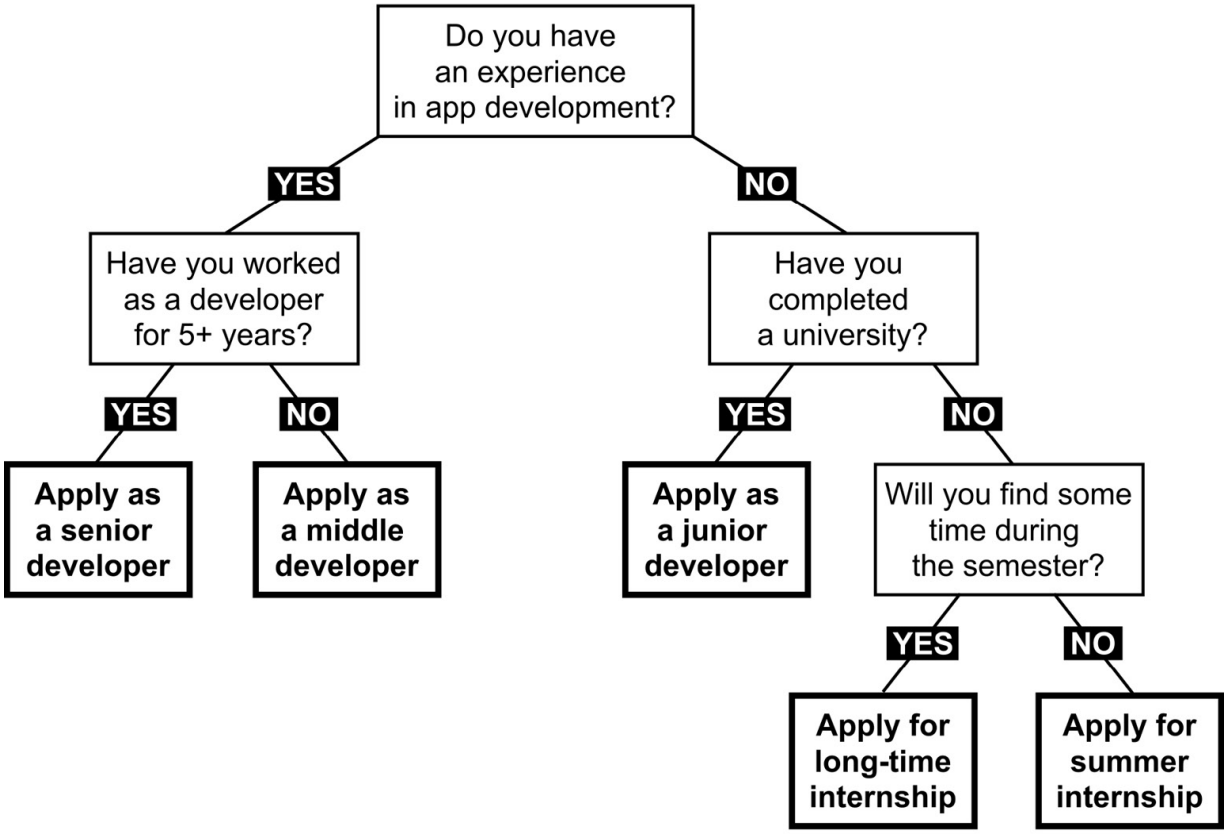

Figure 7.7 – Illustration of the simple quiz example

At the beginning, the user is asked whether they have any experience in application development. If so, the program asks whether they have worked as a developer for more than 5 years. In the case of a positive answer, the result regarding applying to work as a senior developer is presented. Of course, other answers and questions are shown in the case of different decisions taken by the user.

The implementation of the simple quiz requires the **`BinaryTree`** and **`BinaryTreeNode`** classes, which were presented and explained earlier. Each node stores only a **`string`** value as data, representing either a question or an answer.

Let's take a look at the main part of the code:

```
BinaryTree<string> tree = GetTree();
BinaryTreeNode<string>? node = tree.Root;
while (node != null)
{
    if (node.Left != null && node.Right != null)
    {
        Console.WriteLine(node.Data);
        node = Console.ReadKey(true).Key switch
        {
            ConsoleKey.Y => node.Left,
            ConsoleKey.N => node.Right,
```

```
            _ => node
        };
    }
    else
    {
        Console.WriteLine(node.Data);
        node = null;
    }
}
```

In the first line, the **`GetTree`** method is called to construct a tree with questions and answers. Such a method will be shown next. Then, the root node is taken as the current node, for which the following operations are taken until an answer is reached.

At the beginning, you check whether the left and right child nodes exist – that is, whether it is a question and not an answer. Then, the textual content is written in the console, and the program waits until the user presses a key. If it is equal to *Y*, the current node's left child is used as the current node. In the case of pressing *N*, the current node's right child is used instead.

When decisions taken by the user cause an answer to be shown, it is presented in the console, and **`null`** is assigned to the **`node`** variable. Therefore, you break out of the **`while`** loop.

As mentioned, the **`GetTree`** method is used to construct a binary tree with questions and answers. Its code is presented as follows:

```
BinaryTree<string> GetTree()
{
    BinaryTree<string> tree = new();
    tree.Root = new BinaryTreeNode<string>()
    {
        Data = "Do you have an experience
            in app development?",
        Children =
        [
            new BinaryTreeNode<string>()
            {
                Data = "Have you worked as a developer
                    for 5+ years?",
                Children =
                [
                    new() { Data = "Apply as
                        a senior developer" },
                    new() { Data = "Apply as
                        a middle developer" }
                ]
            },
            new BinaryTreeNode<string>()
            {
                Data = "Have you completed a university?",
                Children =
                [
                    new() { Data = "Apply as
                        a junior developer" },
                    new BinaryTreeNode<string>()
                    {
                        Data = "Will you find some time
```

```
                                during the semester?",
                            Children =
                            [
                                new() { Data = "Apply for
                                    long-time internship" },
                                new() { Data = "Apply for
                                    summer internship" }
                            ]
                        }
                    ]
                }
            ]
        };
        tree.Count = 9;
        return tree;
    }
```

At the beginning, a new instance of the **`BinaryTree`** generic class is created, and you assign a new instance of **`BinaryTreeNode`** to the **`Root`** property.

What is interesting is that even while creating questions and answers programmatically, you create some kind of a tree-like structure because you use the **`Children`** property and specify items directly within such constructions. Therefore, you do not need to create many local variables for all questions and answers. It is worth noting that a question-related node is an instance of the **`BinaryTreeNode`** class with two child nodes (for *yes* and *no* decisions), while an answer-related node is a leaf, so it does not contain any child nodes.

IMPORTANT NOTE

*In the presented solution, the values of the **`Parent`** property of the **`BinaryTreeNode`** instances are not set. If you want to use them or get the height of a node or a tree, you should set them on your own.*

The simple quiz application is ready! You can build the project, launch it, and answer a few questions to see the results. Then, let's close the program and proceed to the next section, where a variant of the binary tree data structure is presented.

Binary search trees

A binary tree is an interesting data structure that allows the creation of a hierarchy of elements, with the restriction that each node can contain at most two children, but without any rules about relationships between the nodes. For this reason, if you want to check whether a binary tree contains a given value, you need to check each node, traversing the tree using one of three available modes: pre-order, in-order, or post-order. This means that the lookup time is linear, namely *O(n)*.

What about a situation where there are some precise rules regarding relations between nodes in a tree? Let's imagine a scenario where you know that the left subtree contains nodes with values smaller than the root's value, while the right subtree contains nodes with values greater than the

root's value. Then, you can compare the searched value with the current node and decide whether you should continue searching in the left or right subtree. Such an approach can significantly limit the number of operations necessary to check whether the tree contains a given value. It seems quite interesting, doesn't it?

This approach is applied in the **binary search tree (BST)** data structure. It is a kind of binary tree that introduces two strict rules regarding relations between nodes in the tree. **The rules state that for any node, the values of all nodes in its left subtree must be smaller than its value, and the values of all nodes in its right subtree must be greater than its value.**

CAN YOU ADD DUPLICATES TO BSTS?

In general, a BST can contain two or more elements with the same value. However, within this book, a simplified version is given, which does not accept more than one element with the same value.

How does it look in practice? Let's take a look at the following diagram of BSTs:

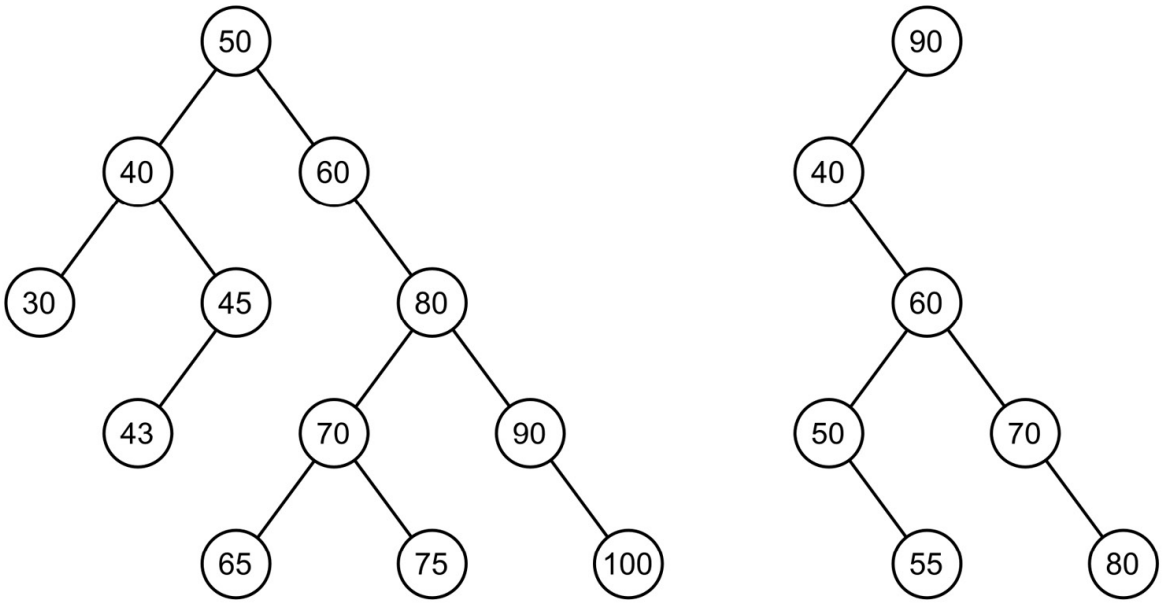

Figure 7.8 – Illustration of binary search trees.

The tree shown on the left-hand side contains 12 nodes. Let's check whether it complies with the BST rule. You can do so by analyzing each node in the tree, except leaf nodes. Let's start with the root node (with value **50**), which contains four descendant nodes in the left subtree (**40**, **30**, **45**, **43**), all smaller than **50**. The root node contains seven descendant nodes in the right subtree (**60**, **80**, **70**, **65**, **75**, **90**, **100**), all greater than **50**. That means that the BST rule is satisfied for the root node. While checking the BST rule for node **80**, you see that the values of all descendant nodes in the left subtree (**70**, **65**, **75**) are smaller than **80**, while the values in the right subtree (**90**, **100**) are greater than **80**.

You should perform the same verification for all other nodes. Similarly, you can confirm that the BST from the right-hand side of the diagram adheres to the rules.

However, two such BSTs significantly differ in their **topology**. Both have the same height, but the number of nodes is different, namely 12 and 7. The one on the left seems to be **fat**, while the other is rather **skinny**. Which one is better? To answer this question, let's think about the algorithm for searching a value in the tree. As an example, the process of searching for the value **43** is shown in the following diagram:

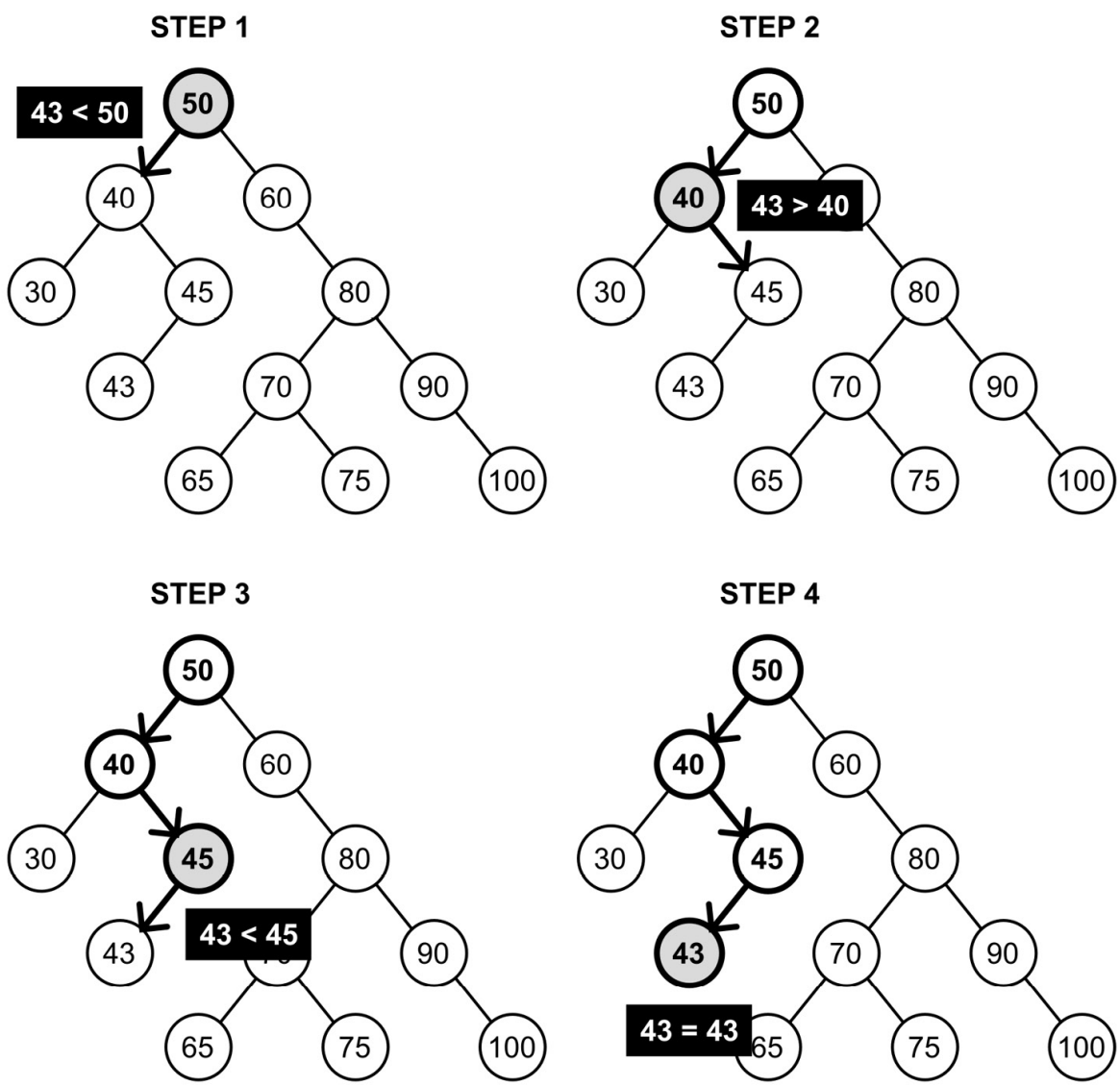

Figure 7.9 – Searching for a given value in a BST

At the beginning (**Step 1**), you take a value of the root node (that is, **50**) and check whether the given value (**43**) is smaller or greater. It is smaller, so you proceed to search in the left subtree (**Step 2**). Thus, you compare **43** with **40**. This time, the right subtree is chosen because **43** is greater than **40**. Next, **43** is compared with **45** (**Step 3**), and the left subtree is chosen. Here, you compare **43** with **43**, and the given value is found (**Step 4**). If you take a look at the tree, you will see that only four comparisons are necessary.

WHAT ABOUT THE PERFORMANCE?

*The shape of a tree has a great impact on the lookup performance. Of course, **it is much better to have a fat tree with limited height than a skinny tree with a bigger height**. The performance boost is caused by making decisions as to whether searching should be continued in the left or right subtree, without the necessity of analyzing the values of all nodes. If nodes do not have both subtrees, the positive impact on the performance will be limited. In the worst case, when each node contains only one child, the search time is even linear. However, in the ideal BST, the lookup time is an O(log n) operation.*

After this short introduction, let's proceed to the implementation in the C# language. At the end, you will see an example that shows how to use this data structure in practice.

Implementation

The implementation of a BST is a bit more difficult than the previously described variants of trees. For example, it requires you to prepare operations of insertion and removal of nodes from a tree, which do not break the rule regarding the arrangement of elements in the BST. You will see a solution shortly.

Tree

The whole tree is represented by an instance of the **`BinarySearchTree`** class, which inherits from the **`BinaryTree`** generic class, as in the following code snippet:

```
public class BinarySearchTree<T>
    : BinaryTree<T>
    where T : IComparable
{
}
```

It is worth mentioning that the type of data, stored in each node, should be comparable. For this reason, it has to implement the **`IComparable`** interface. Such a requirement is necessary because the algorithm needs to know the relationships between values.

Of course, it is not the final version of the implementation of the **`BinarySearchTree`** class. You will very soon learn how to add new features, such as lookup, insertion, and removal of nodes.

Lookup

Now, let's take a look at the **`Contains`** method, which checks whether the tree contains a node with a given value. Of course, this method takes into account the BST rule regarding the arrangement of nodes to limit the number of comparisons. The code is presented in the following block:

```
public bool Contains(T data)
{
    BinaryTreeNode<T>? node = Root;
    while (node != null)
    {
        int result = data.CompareTo(node.Data);
        if (result == 0) { return true; }
        else if (result < 0) { node = node.Left; }
        else { node = node.Right; }
    }
    return false;
}
```

The method takes only one parameter, namely the value to find in the tree. Inside the method, a **`while`** loop exists. Within it, the searched value is compared with the value of the current node. If they are equal (the comparison returns `0`), the value is found, and **`true`** is returned to inform that the search is completed successfully. If the searched value is smaller than the value of the current node, the algorithm continues searching in the left subtree. Otherwise, the right subtree is used instead.

HOW TO COMPARE OBJECTS?

*The **`CompareTo`** method is provided by the implementation of the **`IComparable`** interface from the **`System`** namespace. Such a method makes it possible to compare two values. If they are equal, `0` is returned. If the object on which the method is called is bigger than the parameter, a value higher than `0` is returned. Otherwise, a value lower than `0` is returned.*

The loop is executed until the node is found or there is no suitable child node to follow.

Insertion

The next necessary operation is the insertion of a node into a BST. Such a task is a bit more complicated because you need to find a place for adding a new element that will not violate BST rules. Let's take a look at the code of the **`Add`** method:

```
public void Add(T data)
{
    BinaryTreeNode<T>? parent = GetParentForNewNode(data);
    BinaryTreeNode<T> node = new()
    {
        Data = data,
        Parent = parent
    };
    if (parent == null)
    {
        Root = node;
    }
    else if (data.CompareTo(parent.Data) < 0)
    {
        parent.Left = node;
    }
    else
```

```
        {
            parent.Right = node;
        }
        Count++;
    }
```

The method takes one parameter, namely a value that should be added to the tree. Within the method, you find a parent element (using the **`GetParentForNewNode`** auxiliary method, shown a bit later), where a new node should be added as a child. Then, a new instance of the **`BinaryTreeNode`** class is created, and the values of its **`Data`** and **`Parent`** properties are set.

In the following part of the method, you check whether the found parent element is equal to **`null`**. It means that there are no nodes in the tree, and a new node should be added as the root, which is well visible in the line, where a reference to the node is assigned to the **`Root`** property.

The next comparison checks whether the value for addition is smaller than the value of the parent node. In such a case, a new node should be added as the left child of the parent node. Otherwise, the new node is placed as the right child of the parent node. At the end, the number of elements stored in the tree is incremented.

Let's take a look at the auxiliary method for finding the parent element for a new node:

```
    private BinaryTreeNode<T>? GetParentForNewNode(T data)
    {
        BinaryTreeNode<T>? current = Root;
        BinaryTreeNode<T>? parent = null;
        while (current != null)
        {
            parent = current;
            int result = data.CompareTo(current.Data);
            if (result == 0) { throw new ArgumentException(
                $"The node {data} already exists."); }
            else if (result < 0) { current = current.Left; }
            else { current = current.Right; }
        }
        return parent;
    }
```

This method takes one parameter, namely a value of the new node. Within this method, you declare two variables representing the currently analyzed node (**`current`**) and the parent node (**`parent`**). Such values are modified in a **`while`** loop until the algorithm finds a proper place for the new node.

In the loop, you store a reference to the current node as the potential parent node. Then, you check whether the value for addition is equal to the value of the current node. If so, an exception is thrown because it is not allowed to add more than one element with the same value to the analyzed version of the BST. If the value for addition is smaller than the value of the current node, the algorithm continues searching for a place for the new node in the left subtree. Otherwise, the right subtree of the current node is used. At the end, the value of the **`parent`** variable is returned to indicate the found location for the new node.

Removal

You now know how to create a new BST, add some nodes to it, as well as check whether a given value already exists in the tree. However, can you also remove an item from a tree? Of course! The main method regarding the removal of a node from the tree is named **`Remove`** and takes only one parameter, namely the value of the node that should be removed. The implementation of the **`Remove`** method is as follows:

```
public void Remove(T data) => Remove(Root, data);
```

As you can see, the method just calls another method, also named **`Remove`**. The implementation of this private method is more complicated and is as follows:

```
private void Remove(BinaryTreeNode<T>? node, T data)
{
    if (node == null)
    {
        return;
    }
    else if (data.CompareTo(node.Data) < 0)
    {
        Remove(node.Left, data);
    }
    else if (data.CompareTo(node.Data) > 0)
    {
        Remove(node.Right, data);
    }
    else
    {
        if (node.Left == null || node.Right == null)
        {
            BinaryTreeNode<T>? newNode =
                node.Left == null && node.Right == null
                    ? null
                    : node.Left ?? node.Right;
            ReplaceInParent(node, newNode!);
            Count--;
        }
        else
        {
            BinaryTreeNode<T> successor =
                FindMinimumInSubtree(node.Right);
            node.Data = successor.Data;
            Remove(successor, successor.Data!);
        }
    }
}
```

At the beginning, the method checks whether the current node (the **`node`** parameter) exists. If not, you exit from the method.

Then, the **`Remove`** method tries to find the node to remove. That is achieved by comparing the value of the current node with the value for removal and calling the **`Remove`** method recursively for either the left or right subtree of the current node.

The most interesting operations are performed in the following part of the method. Here, you need to handle four scenarios of node removal, namely the following:

- Removing a leaf node
- Removing a node with only a left child
- Removing a node with only a right child
- Removing a node with both left and right children

In the case of **removing a leaf node**, you just update a reference to the deleted node in the parent element. Therefore, there will be no reference from the parent node to the deleted node, and it cannot be reached while traversing the tree.

Removing a node with only a left child is also simple because you only need to replace a reference to the deleted node (in the parent element) with the node that is a left child of the deleted node. This scenario is shown in the following diagram, which presents how to remove node **80** with only the left child:

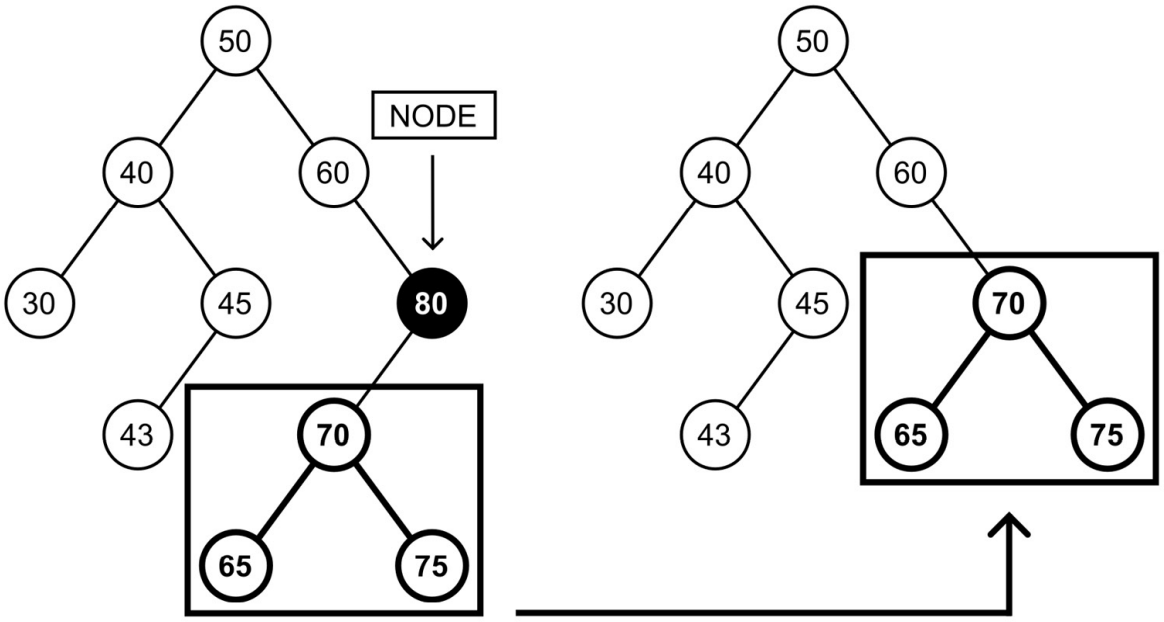

Figure 7.10 – Removing a node with only a left child from a BST

The case of **removing a node with only a right child** is very similar to the second case. Thus, you just replace a reference to the deleted node (in the parent element) with the node that is a right child of the deleted node.

All those three cases are handled in the code in a similar way, by calling the `ReplaceInParent` auxiliary method, the code of which is as follows:

```
private void ReplaceInParent(BinaryTreeNode<T> node,
    BinaryTreeNode<T> newNode)
{
    if (node.Parent != null)
    {
        BinaryTreeNode<T> parent =
            (BinaryTreeNode<T>)node.Parent;
        if (parent.Left == node) { parent.Left = newNode; }
        else { parent.Right = newNode; }
    }
    else { Root = newNode; }
    if (newNode != null) { newNode.Parent = node.Parent; }
}
```

The method takes two parameters: the node for removal (**`node`**) and the node that should replace it in the parent node (**`newNode`**). For this reason, if you want to remove a leaf node, you just pass **`null`** as the second parameter because you do not want to replace the removed node with anything else. In the case of removing a node with only one child, you pass a reference to the left or right child.

If the node for removal is not the root, you check whether it is the left child of the parent. If so, a proper reference is updated. It means that the new node is set as the left child of the parent node of the node for removal. In a similar way, the method handles the scenario when the node for removal is the right child of the parent. If the node for removal is the root, the node for replacing is set as the root. At the end, you check whether the new node is not equal to **`null`**. It means that you are not removing a leaf node. In such a case, you set a value of the **`Parent`** property to indicate that the new node should have the same parent as the node for removal.

A bit more complicated scenario is **the removal of a node with both child nodes**. In such a case, you find a node with the minimum value in the right subtree of the node for removal. Then, you swap the value of the node for removal with the value of the found node. Finally, you just need to call the **`Remove`** method recursively for the found node. The relevant part of the code is shown in the following code snippet, copied here from the **`Remove`** private method for your convenience:

```
BinaryTreeNode<T> successor =
    FindMinimumInSubtree(node.Right);
node.Data = successor.Data;
Remove(successor, successor.Data!);
```

The last auxiliary method is named **`FindMinimumInSubtree`** and is as follows:

```
private BinaryTreeNode<T> FindMinimumInSubtree(
    BinaryTreeNode<T> node)
{
    while (node.Left != null) { node = node.Left; }
    return node;
}
```

The method takes the root of the subtree, where the minimum value should be found, as the parameter. Within the method, a **`while`** loop is used to get the leftmost element. When there is no left

child, the current value of the `node` variable is returned.

WHERE YOU CAN FIND MORE INFORMATION?

There is a lot of information about BSTs in books, research papers, as well as over the internet. However, the presented implementation of a BST is based on the code shown at https://en.wikipedia.org/wiki/Binary_search_tree, where you can also find more information about this data structure. I strongly encourage you to be curious about various data structures and algorithms and to broaden your knowledge.

The preceding code does not look very difficult, does it? However, how does it work in practice? Let's take a look at a diagram depicting the removal of a node with two children:

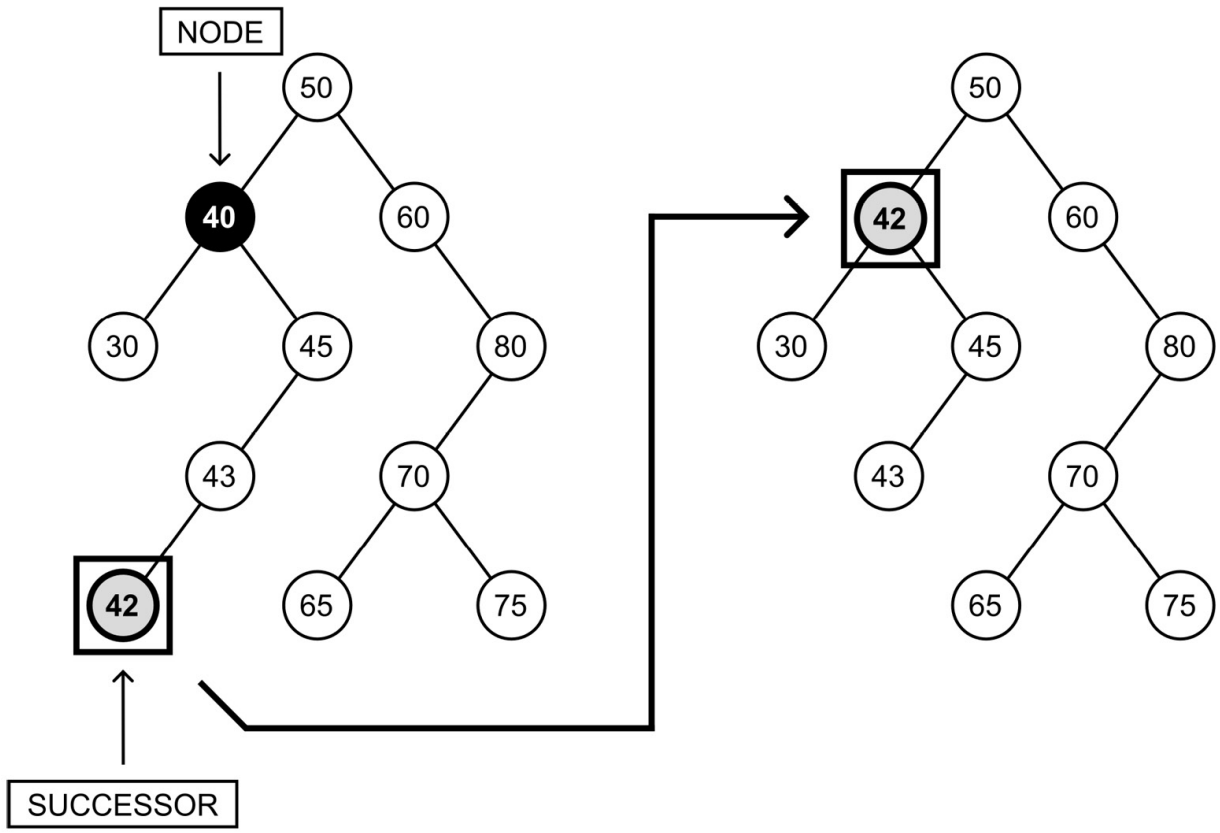

Figure 7.11 – Removing a node with two children in a BST

The diagram shows how to remove the node with **40** as the value. To do so, you need to find the successor. It is the node with the minimum value in the right subtree of the node for removal. The successor is node **42**, which replaces node **40**.

Example – BST visualization

While reading the section regarding BSTs, you learned a lot about this data structure. So, it is high time to create an example program to see this variant of trees in action. The application will show you how to create a BST, add some nodes (both manually and using the previously presented method for insertion), remove nodes, traverse the tree, as well as visualize the tree in the console.

At the beginning, a new tree (with nodes storing integer values) is prepared by creating a new instance of the **`BinarySearchTree`** class. It is configured manually by adding three nodes, together with indicating proper references for children and parent elements. The relevant part of the code is as follows:

```
BinarySearchTree<int> tree = new();
tree.Root = new BinaryTreeNode<int>() { Data = 100 };
tree.Root.Left = new() { Data = 50, Parent = tree.Root };
tree.Root.Right = new() { Data = 150, Parent = tree.Root };
tree.Count = 3;
Visualize(tree, "BST with 3 nodes (50, 100, 150):");
```

Then, you use the **`Add`** method to add some nodes to the tree and visualize the current state of the tree using the **`Visualize`** method, as follows:

```
tree.Add(75);
tree.Add(125);
Visualize(tree, "BST after adding 2 nodes (75, 125):");
```

Let's add five more nodes with the following code:

```
tree.Add(25);
tree.Add(175);
tree.Add(90);
tree.Add(110);
tree.Add(135);
Visualize(tree, "BST after adding 5 nodes
    (25, 175, 90, 110, 135):");
```

The next set of operations is related to the removal of various nodes from the tree, together with visualization of particular changes. The part of the code is as follows:

```
tree.Remove(25);
Visualize(tree, "BST after removing the node 25:");
tree.Remove(50);
Visualize(tree, "BST after removing the node 50:");
tree.Remove(100);
Visualize(tree, "BST after removing the node 100:");
```

At the end, all three traversal modes are presented. The suitable code is as follows:

```
foreach (TraversalEnum mode
    in Enum.GetValues<TraversalEnum>())
{
    Console.Write($"\n{mode} traversal:\t");
    Console.Write(string.Join(", ",
        tree.Traverse(mode).Select(n => n.Data)));
}
```

Another interesting task is the development of a tree visualization in the console. Such a feature is really useful because it allows a comfortable and fast way of observing the tree without the necessity of debugging the application in the IDE and expanding the following elements in the tooltip with the current values of variables. However, presenting a tree in the console is not a trivial task. Fortunately, you do not need to worry about it because you will learn how to implement such a feature in this section.

First, let's take a look at the **`Visualize`** method:

```
void Visualize(BinarySearchTree<int> tree, string caption)
{
    char[,] console = Initialize(tree, out int width);
    VisualizeNode(tree.Root, 0, width / 2,
        console, width);
    Console.WriteLine(caption);
    Draw(console);
}
```

The method takes two parameters, namely an instance of the **`BinarySearchTree`** class representing the whole tree, and a caption that should be shown above the visualization. Within the method, an array with characters that should be presented in the console is initialized using the **`Initialize`** auxiliary method, shown a bit later. Then, you call the **`VisualizeNode`** recursive method to fill various parts of the array with data regarding particular nodes existing in the tree. At the end, the caption and the board (represented by the array) are written in the console.

The **`Initialize`** method creates the aforementioned array, as presented in the following code snippet:

```
const int ColumnWidth = 5;
char[,] Initialize(BinarySearchTree<int> tree,
    out int width)
{
    int height = tree.GetHeight();
    width = (int)Math.Pow(2, height) - 1;
    char[,] console =
        new char[height * 2, ColumnWidth * width];
    for (int y = 0; y < console.GetLength(0); y++)
    {
        for (int x = 0; x < console.GetLength(1); x++)
        {
            console[y, x] = ' ';
        }
    }
    return console;
}
```

The two-dimensional array contains the number of rows equal to the height of the tree multiplied by **`2`** to have space also for lines connecting nodes with parents. The number of columns is calculated according to the formula *columnwidth* * 2^{height} - 1, where *columnwidth* is the **`ColumnWidth`** constant value and *height* is the height of the tree.

These values can be simpler to understand if you take a look at the result:

```
                              100
             +-------------------+-------------------+
            50                                     150
      +---------+---------+                  +---------+---------+
     25                  75                 125                 175
                      +----+             +----+----+
                          90           110       135
```

Figure 7.12 – Screenshot of the BST visualization example

In the **`Visualize`** method, **`VisualizeNode`** is called. Are you interested to learn about how it works and how you can present not only the values of nodes but also lines? If so, let's take a look at its code, which is as follows:

```
void VisualizeNode(BinaryTreeNode<int>? node, int row,
    int column, char[,] console, int width)
{
    if (node == null) { return; }
    char[] chars = node.Data.ToString().ToCharArray();
    int margin = (ColumnWidth - chars.Length) / 2;
    for (int i = 0; i < chars.Length; i++)
    {
        int col = ColumnWidth * column + i + margin;
        console[row, col] = chars[i];
    }
    int columnDelta = (width + 1) /
        (int)Math.Pow(2, node.GetHeight() + 1);
    VisualizeNode(node.Left, row + 2,
        column - columnDelta, console, width);
    VisualizeNode(node.Right, row + 2,
        column + columnDelta, console, width);
    DrawLineLeft(node, row, column, console, columnDelta);
    DrawLineRight(node, row, column, console, columnDelta);
}
```

The **`VisualizeNode`** method takes five parameters, including the current node for visualization (**`node`**), the index of a row (**`row`**), and the index of a column (**`column`**). Within the method, there is a check for whether the current node exists. If it does, the value of the node is obtained as a **`char`** array, the margin is calculated, and the **`char`** array (with a character-based representation of the value) is written in the buffer (the **`console`** variable) within a **`for`** loop.

In the following lines of code, the **`VisualizeNode`** method is called for the left and right child nodes of the current node. Of course, you need to adjust the index of the row (by adding **`2`**) and the index of the column (by adding or subtracting the calculated value).

At the end, lines are drawn by calling the **`DrawLineLeft`** and **`DrawLineRight`** methods. The first is presented in the following code snippet:

```
void DrawLineLeft(BinaryTreeNode<int> node, int row,
    int column, char[,] console, int columnDelta)
{
    if (node.Left == null) { return; }
    int sci = ColumnWidth * (column - columnDelta) + 2;
    int eci = ColumnWidth * column + 2;
    for (int x = sci + 1; x < eci; x++)
    {
        console[row + 1, x] = '-';
    }
    console[row + 1, sci] = '+';
    console[row + 1, eci] = '+';
}
```

The method also takes five parameters:

- The current node for which the line should be drawn (**`node`**)
- The index of a row (**`row`**)
- The index of a column (**`column`**)
- An array as a screen buffer (**`console`**)
- A delta value calculated in the **`VisualizeNode`** method

At the beginning, you check whether the current node contains a left child because only then is it necessary to draw the left part of the line. If so, you calculate the start (**`sci`**, which stands for *start column index*) and end (**`eci`** as *end column index*) indices of columns and fill the proper elements of the array with dashes. At the end, a plus sign is added to the array in the place where the drawn line will be connected with the right line of another element and on the other side of the line.

In almost the same way, you draw the right line for the current node. Of course, you need to adjust the code regarding calculating column start and end indices. The final version of the code of the **`DrawLineRight`** method is as follows:

```
void DrawLineRight(BinaryTreeNode<int> node, int row,
    int column, char[,] console, int columnDelta)
{
    if (node.Right == null) { return; }
    int sci = ColumnWidth * column + 2;
    int eci = ColumnWidth * (column + columnDelta) + 2;
    for (int x = sci + 1; x < eci; x++)
    {
        console[row + 1, x] = '-';
    }
    console[row + 1, sci] = '+';
    console[row + 1, eci] = '+';
}
```

At the end, let's see the `Draw` method that shows the board in the console. It just iterates through all elements of the array and writes them in the console, as follows:

```
void Draw(char[,] console)
{
    for (int y = 0; y < console.GetLength(0); y++)
    {
        for (int x = 0; x < console.GetLength(1); x++)
        {
            Console.Write(console[y, x]);
        }
        Console.WriteLine();
    }
}
```

That's all! You wrote the whole code necessary to build the project, launch the program, and see it in action. Just after launching, you will see the first BST, as follows:

```
BST with 3 nodes (50, 100, 150):
      100
  +----+----+
 50        150
```

Figure 7.13 – Screenshot of the BST visualization example, step 1

After adding the next two nodes, `75` and `125`, the BST looks a bit different:

```
BST after adding 2 nodes (75, 125):
                100
       +---------+---------+
      50                  150
       +----+         +----+
           75        125
```

Figure 7.14 – Screenshot of the BST visualization example, step 2

Then, you perform an insertion operation for the next five elements. These operations have a very visible impact on the tree shape, as presented in the console:

```
BST after adding 5 nodes (25, 175, 90, 110, 135):
                                100
                +-------------------+-------------------+
               50                                      150
       +---------+---------+                   +---------+---------+
      25                 75                  125                 175
                          +----+          +----+----+
                              90         110       135
```

Figure 7.15 – Screenshot of the BST visualization example, step 3

After adding 10 elements, the program shows the impact of removing a particular node on the shape of the tree. To start, let's remove the leaf node with `25` as the value:

```
BST after removing the node 25:
                                100
                +-------------------+-------------------+
               50                                      150
                +---------+                   +---------+---------+
                         75                  125                 175
                          +----+          +----+----+
                              90         110       135
```

Figure 7.16 – Screenshot of the BST visualization example, step 4

Then, the program removes a node with only one child node, namely the right one. What is interesting is that the right child also has a right child. However, the presented algorithm works properly in such conditions, and you receive the following result:

```
BST after removing the node 50:
                                100
                +-------------------+-------------------+
               75                                      150
                +---------+                   +---------+---------+
                         90                  125                 175
                                          +----+----+
                                         110       135
```

Figure 7.17 – Screenshot of the BST visualization example, step 5

The last removal operation is the most complicated one because it requires you to remove the node with both children, and it also performs the role of the root. In such a case, the leftmost element from the right subtree of the root is found and replaces the node for removal, as shown in the final view of the tree:

```
BST after removing the node 100:
                                 110
                 +------------------+------------------+
                75                                    150
                 +---------+                  +---------+---------+
                         90                  125                 175
                                              +----+
                                                  135
```

Figure 7.18 – Screenshot of the BST visualization example, step 6

One more set of operations is left, namely traversal of the tree in pre-order, in-order, and post-order. The application presents the following results:

```
Pre-order traversal:    110, 75, 90, 150, 125, 135, 175
In-order traversal:     75, 90, 110, 125, 135, 150, 175
Post-order traversal:   90, 75, 135, 125, 175, 150, 110
```

The created application looks quite impressive, doesn't it? You created not only the implementation of a BST from scratch but also prepared the platform for its visualization in the console. Great job!

IS IT ALREADY SORTED?

Let's take one more look at the results of the in-order approach. As you can see, it gives you the nodes sorted in ascending order in the case of a BST.

However, can you see a potential problem with the created solution? What about a scenario where you remove nodes only from the given area of a tree or when you insert already sorted values? It could mean that a fat tree, with a proper **breadth-depth ratio**, could become a skinny one. In the worst case, it could even be depicted as a list, where all nodes have only one child. Do you have any idea how to solve the problem of unbalanced trees and keep them balanced all the time? If not, next, you will find some information on how to achieve this goal.

Self-balancing trees

In this section, you will get to know two variants of a **self-balancing tree**, which **keeps the tree balanced all the time while adding and removing nodes.** However, why is it so important? As already mentioned, the lookup performance depends on the shape of the tree. In the case of improper

organization of nodes, forming a list, the process of searching for a given value can be an *O(n)* operation. With a correctly arranged tree, the performance can be significantly improved with *O(log n)*.

Do you know that a BST can very easily become an **unbalanced tree**? Let's make a simple test of adding the following nine numbers to the tree, from **1** to **9**. Then, you will receive a tree with the shape shown in the following diagram on the left. However, the same values can be arranged in another way, as a **balanced tree**, with a significantly better breadth-depth ratio, which is shown on the right:

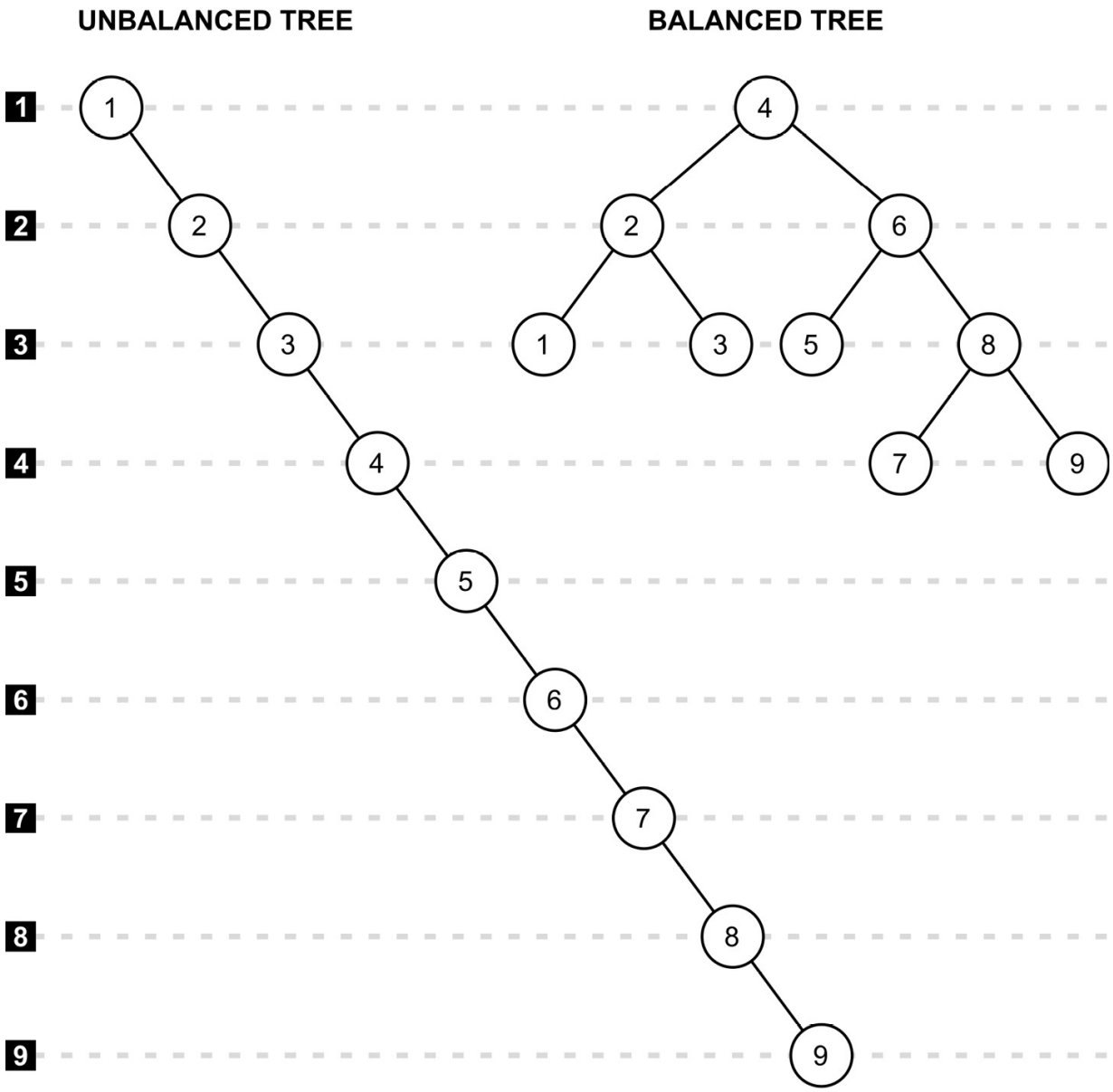

Figure 7.19 – Difference between an unbalanced and a balanced tree

You now know what unbalanced and balanced trees are, as well as what is the aim of self-balancing trees. However, what is an AVL tree or a red-black tree? How do they work? What rules should be taken into account while using these data structures? You will find answers to these questions next.

AVL trees

An **AVL tree** is named after its inventors, namely Adelson-Velsky and Landis. It is **a binary search tree with the additional requirement that, for each node, the height of its left and right subtrees cannot differ by more than one**. Of course, that rule must be maintained after adding and removing nodes from a tree. The important role is performed by **rotations**, used to fix incorrect arrangements of nodes.

WHAT ABOUT THE PERFORMANCE?

While talking about AVL trees, it is crucial to indicate the performance of this data structure. In this case, both average and worst-case scenarios of insertion, removal, and lookup are O(log n), so there is significant improvement in the worst-case scenarios in comparison with a BST.

The implementation of AVL trees, including various rotations necessary to keep the balanced state of a tree, is not trivial and will require quite a long explanation. Due to the limited number of pages in the book, its implementation is not presented here. Fortunately, you can use one of the available NuGet packages that support such tree-based data structures to benefit from AVL trees in your applications.

Red-black trees

A **red-black tree** (**RBT**) is the next variant of self-balancing binary search trees. As a variant of BSTs, this data structure requires that standard BST rules are maintained. Moreover, the following rules must be taken into account:

- **Each node must be colored either red or black**. Thus, you need to add additional data for a node that stores a color.
- **All nodes with values cannot be leaf nodes**. For this reason, `NIL` pseudo-nodes should be used as leaves in the tree, while all other nodes are internal ones. Moreover, all `NIL` pseudo-nodes must be black.
- **If a node is red, both its children must be black**.
- For any node, **the number of black nodes on the route to a descendant leaf** (that is, the `NIL` pseudo-node) **must be the same**.

A proper RBT is presented in the following diagram:

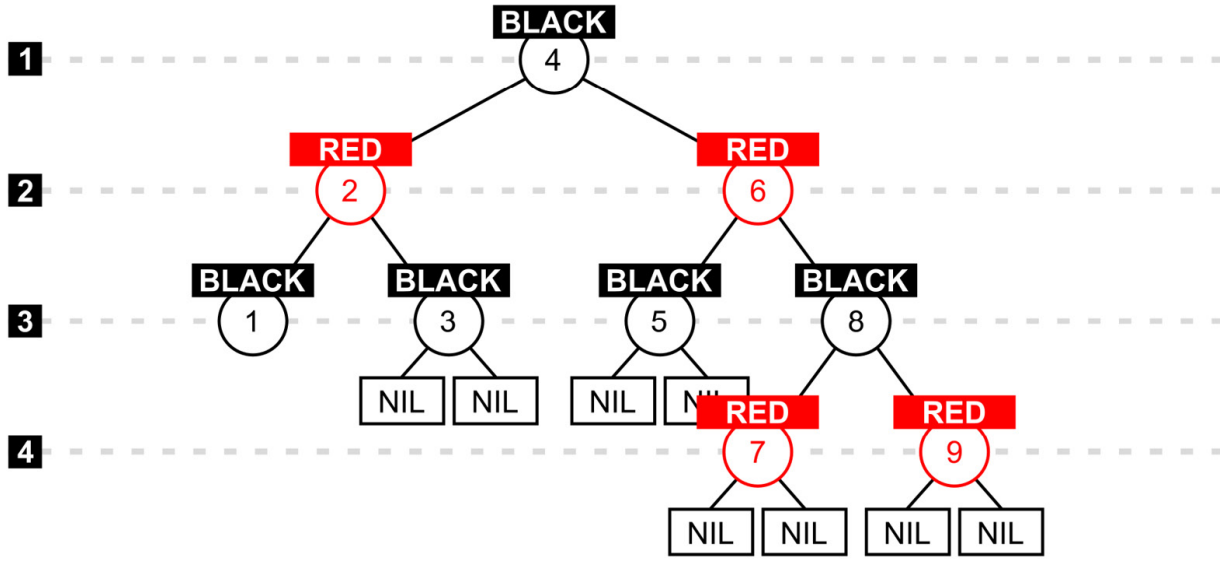

Figure 7.20 – Illustration of a red-black tree

The tree consists of nine nodes, each colored red or black. It is worth mentioning the `NIL` pseudo-nodes, which are added as leaf nodes. If you again take a look at the set of rules listed previously, you can confirm that all such rules are maintained in this case.

Similarly to AVL trees, RBTs also must maintain rules after adding or removing a node. In this case, the process of restoring the RBT properties is even more complicated because it involves both **recoloring** and **rotations**.

WHAT ABOUT THE PERFORMANCE?

While talking about this variant of self-balancing BSTs, it is also worth noting the performance. In both average and worst-case scenarios, insertion, removal, and lookup are O(log n) operations, so they are the same as in the case of AVL trees and much better in worst-case scenarios in comparison with BSTs.

Fortunately, you do not need to know and understand the internal details, which are quite complex, to benefit from this data structure and apply it to your projects. As already mentioned in the case of AVL trees, you can also use one of the available NuGet packages for RBTs.

WHERE CAN YOU FIND MORE INFORMATION?

The topic of trees is much broader than shown in this chapter. For this reason, if you are interested in such a subject, I strongly encourage you to search for more information on your own. You can also find some content on Wikipedia, such as at https://en.wikipedia.org/wiki/Binary_tree and https://en.wikipedia.org/wiki/Binary_search_tree. Self-balancing trees are covered at https://en.wikipedia.org/wiki/AVL_tree and https://en.wikipedia.org/wiki/Red-black_tree. The topic of tries and binary heaps (mentioned later in this chapter) is presented as well at https://en.wikipedia.org/wiki/Trie and https://en.wikipedia.org/wiki/Binary_heap.

You already learned some basic information about self-balancing trees, namely AVL trees and RBTs. So, let's take a look at another tree-based structure, namely a trie, which is a great solution for string-related operations.

Tries

A tree is a powerful data structure that is used in various scenarios. One of them is related to processing strings, such as for **autocomplete** and **spellchecker** features that you certainly know from many systems. If you want to implement it in your application, you can benefit from another tree-based data structure, namely a **trie**. It is used to store strings and to perform prefix-based searching.

A trie is a tree with one root node, where each node represents a string and each edge indicates a character. A trie node contains references to the next nodes as an array with 26 elements, representing 26 chars from the alphabet (from a to z). When you go from the root to each node, you receive a string, which is either a saved word or its substring.

WHY EXACTLY 26 ELEMENTS?

Here, we use 26 elements representing 26 chars because it is the exact number of basic characters between ***a*** *and* ***z*** *in the alphabet, without any special characters existing in various languages. Of course, in your implementation, you can expand this set with other characters, such as* ***ą****,* ***ę****, or* ***ś*** *from Polish, as well as with even digits or some special characters, such as a dash. Choosing a proper set of characters depends on the scenario in which this data structure will be used.*

Does it sound complicated? It could, so let's take a look at the following diagram, which should remove any doubts:

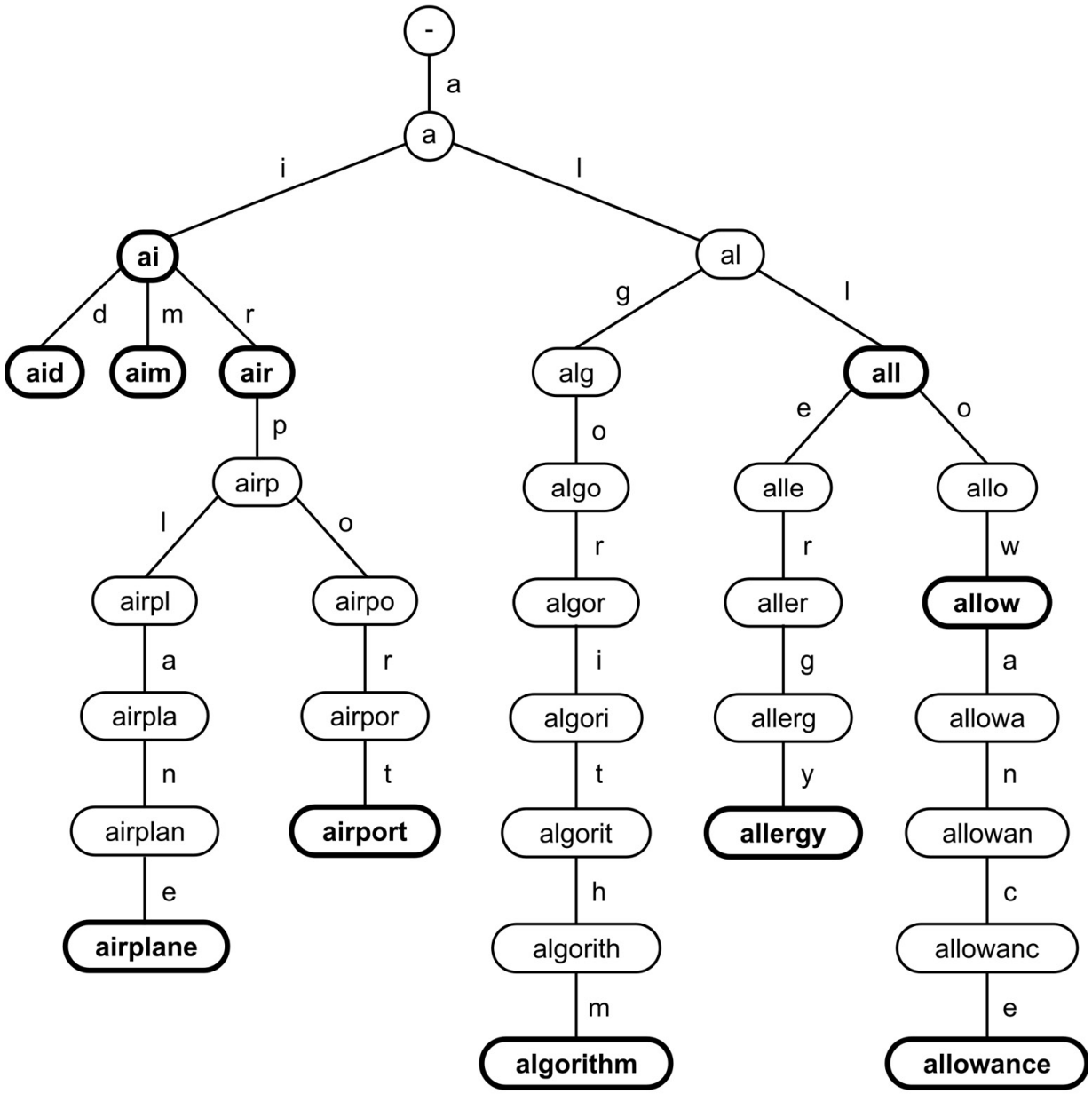

Figure 7.21 – Illustration of a trie

The diagram depicts a trie that stores the following words: **ai**, **aid**, **aim**, **air**, **airplane**, **airport**, **algorithm**, **all**, **allergy**, **allow**, **allowance**. As you can see, there is a root node (marked with **-**) that contains only one child, namely for the **a** substring. This node contains two child nodes, regarding the **ai** word and the **al** substring. The **ai** node has three children, namely representing **aid**, **aim**, and **air** words. In a similar way, you can analyze the whole trie. Please keep in mind that words are marked with bolder lines while substrings are shown with lighter ones.

WHAT ABOUT THE PERFORMANCE?

The searching and insertion in the case of a trie are O(n) operations, where n indicates a word length. So, a trie is an efficient data structure for string-based operations.

Implementation

After this short introduction, let's move to something more exciting – coding!

Node

Please take a look at the following implementation of a class representing a node:

```
public class TrieNode
{
    public TrieNode[] Children { get; set; }
        = new TrieNode[26];
    public bool IsWord { get; set; } = false;
}
```

The `TrieNode` class contains two properties. The first is named `Children` and is an array with `26` elements. Each of them represents a particular letter from an alphabet, starting from `a` (index equal to `0`) and ending with `z` (index equal to `25`). If there is another word with the same prefix, a reference to the next node is located in a suitable element of the `Children` array. The second property is named `IsWord` and indicates whether the current node is the last char from a word. It means that you can get this word by moving from the root element to this node.

Trie

The next part of the code shows the implementation of a class representing a trie:

```
public class Trie
{
    private readonly TrieNode _root = new();
}
```

Here, there is a private field representing the root element. Of course, you need to add some methods to make it operational. First, let's implement a method that checks whether a given word exists in the trie. Its code is as follows:

```
public bool DoesExist(string word)
{
    TrieNode current = _root;
    foreach (char c in word)
    {
        TrieNode child = current.Children[c - 'a'];
        if (child == null) { return false; }
        current = child;
    }
    return current.IsWord;
}
```

At the beginning, you save a reference to the root element as the current node. Then, you iterate through the following characters that form the word. For each character (represented by the `c`

variable), you get a proper node (**`child`**). If it is **`null`**, it means that the word does not exist in the trie. Otherwise, you save the child element as the current one. When the **`foreach`** loop ends, the current node represents a node of the last character, so you just need to return the value of the **`IsWord`** property.

The next method allows you to insert a word into a trie, as shown here:

```
public void Insert(string word)
{
    TrieNode current = _root;
    foreach (char c in word)
    {
        int i = c - 'a';
        current.Children[i] = current.Children[i] ?? new();
        current = current.Children[i];
    }
    current.IsWord = true;
}
```

The preceding code is a bit similar to that already described. However, there is one important difference in the **`foreach`** loop. Here, you create a new child node if it does not exist for any of the chars forming the word. At the end, you indicate that the node represents the word by setting the value of the **`IsWord`** property to **`true`**.

As already mentioned, a trie is a data structure that allows you to perform **prefix-based searching** in an efficient way. So, let's implement it:

```
public List<string> SearchByPrefix(string prefix)
{
    TrieNode current = _root;
    foreach (char c in prefix)
    {
        TrieNode child = current.Children[c - 'a'];
        if (child == null) { return []; }
        current = child;
    }
    List<string> results = [];
    GetAllWithPrefix(current, prefix, results);
    return results;
}
```

The method takes one parameter, namely the prefix of the searched words. At the beginning, you iterate through all characters of the prefix to get a reference to the last character forming the prefix. If a child node is not found at any phase, you return an empty list, which means that there are no results.

Otherwise, you create a **`List<string>`** instance to store the result, and then you call the **`GetAllWithPrefix`** method, the code of which is shown next:

```
private void GetAllWithPrefix(TrieNode node,
    string prefix, List<string> results)
{
    if (node == null) { return; }
```

```
        if (node.IsWord) { results.Add(prefix); }
        for (char c = 'a'; c <= 'z'; c++)
        {
            GetAllWithPrefix(node.Children[c - 'a'],
                prefix + c, results);
        }
    }
```

You check whether the current node is **`null`**. If so, you return from the method. Otherwise, you verify whether the current node forms a word. If so, you add it to **`results`**. Next, you iterate through all alphabet characters, namely from **`a`** to **`z`**, and call the same method recursively to find the next words and add them to the list with **`results`**.

As you can see, the basic implementation of a trie is not a complicated task and can be done with clear and short code. However, how can you test a trie in action? Let's see:

```
Trie trie = new();
trie.Insert("algorithm");
trie.Insert("aid");
trie.Insert("aim");
trie.Insert("air");
trie.Insert("ai");
trie.Insert("airport");
trie.Insert("airplane");
trie.Insert("allergy");
trie.Insert("allowance");
trie.Insert("all");
trie.Insert("allow");
bool isAir = trie.DoesExist("air");
List<string> words = trie.SearchByPrefix("ai");
foreach (string word in words)
{
    Console.WriteLine(word);
}
```

The preceding code forms a trie, as shown in *Figure 7.21*, with 11 words starting with **`a`**, such as **`algorithm`** and **`allow`**. You add such words with the **`Insert`** method. Then, you check whether the **`air`** word exists with the **`DoesExist`** method. Next, you get all words that start with the **`ai`** prefix and write them in the console:

```
ai
aid
aim
air
airplane
airport
```

At the end of the section regarding tries, let's talk about the space complexity of this data structure. As you can see, you need to store 26 references to child nodes for each trie node, and there can be a lot of situations where only one or two references are set. For instance, you can take a look at the **`algorithm`** word, where a lot of space is wasted. It would be much better to optimize it in some way to make the whole tree smaller.

Fortunately, it is possible to use another data structure that is named a **radix tree** or a **compressed trie**, which is **a space-optimized version of a trie**. The difference is quite simple: namely, **you merge with the parent each node that is the only child of this parent**. Of course, **edges can represent a substring** in such a case.

If you want to see what a radix tree looks like for the same input data as in the diagram of a trie, take a look at the following diagram:

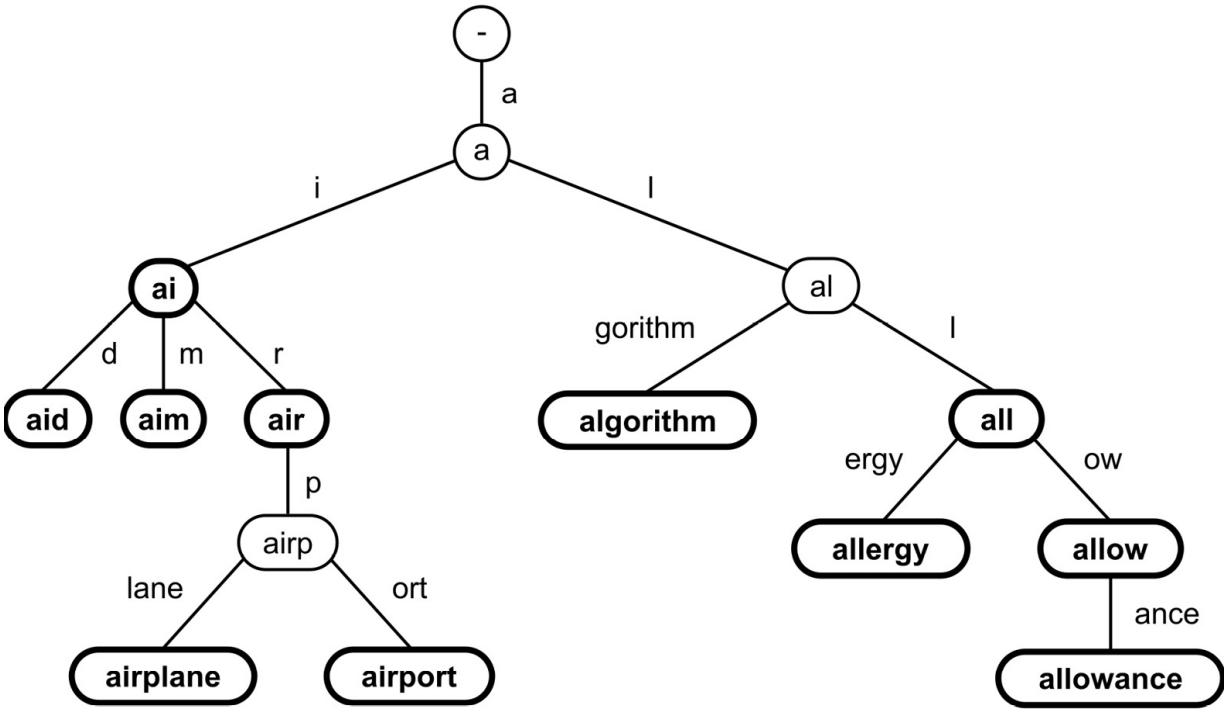

Figure 7.22 – Illustration of a radix tree

Looks much simpler, doesn't it? For example, let's analyze the path from the root node to **algorithm**. Here, you use only three edges, namely **a**, **l**, and **gorithm**.

TRY TO IMPLEMENT IT ON YOUR OWN

Based on the preceding diagram and the implementation of a trie, I encourage you to try to implement a radix tree on your own. You should also prepare a method for searching a word in such a data structure. Good luck!

Example – autocomplete

As an example of a trie application, you will create an **autocomplete** feature that suggests names of countries based on the prefix entered by a user. To do so, you first need to create a `Countries.txt` file

with names of countries, as well as add it to the project as content that will be automatically copied to the output directory.

HOW TO ADD A FILE TO THE PROJECT?

*You should right-click on the project node in the **Solution Explorer** window and choose the **Add | New Item** option. Then, type the name of the file, followed by the **`.txt`** extension. After confirmation, the file is created. If you want to mark this file as a content file and automatically copy it to the output directory, you should click on the file and change two properties in the **Properties** window. First, change **Build Action** to **Content**. Then, set **Copy to Output Directory** to **Copy always**.*

A part of the file with country names is as follows:

```
Afghanistan
Albania
Algeria (...)
Pakistan
Palau
Panama
Papua New Guinea
Paraguay
Peru
Philippines
Poland
Portugal (...)
Zambia
Zimbabwe
```

Of course, a lot of country names are omitted in the preceding code snippet. However, when the file with country names is ready, you need to read its content and form a trie, as presented in the following code block:

```
using System.Text.RegularExpressions;
Trie trie = new();
string[] countries =
    await File.ReadAllLinesAsync("Countries.txt");
foreach (string country in countries)
{
    Regex regex = new("[^a-z]");
    string name = country.ToLower();
    name = regex.Replace(name, string.Empty);
    trie.Insert(name);
}
```

At the beginning, you create a new instance of the **`Trie`** class. Then, you read all lines from the **`Countries.txt`** file and store them in the **`countries`** array.

The remaining part of the code consists of a **`foreach`** loop that iterates through all country names. For each of them, you make it lowercase and remove all chars other than **`a-z`**. Such a task is performed with a regular expression and the **`Regex`** class from the **`System.Text.RegularExpressions`** namespace.

When the trie is ready, you use a **`while`** loop, as shown next:

```
string text = string.Empty;
while (true)
```

```
{
    Console.Write("Enter next character: ");
    ConsoleKeyInfo key = Console.ReadKey();
    if (key.KeyChar < 'a' || key.KeyChar > 'z') { return; }
    text = (text + key.KeyChar).ToLower();
    List<string> results = trie.SearchByPrefix(text);
    if (results.Count == 0) { return; }
    Console.WriteLine(
        $"\nSuggestions for {text.ToUpper()}:");
    results.ForEach(r => Console.WriteLine(r.ToUpper()));
    Console.WriteLine();
}
```

Inside the **`while`** loop, you wait until the user presses any key. If this key is other than **`a-z`**, the program ends its operation. Otherwise, you append the entered char to the prefix that is used for searching all country names that start with this prefix. If the number of results is equal to zero, the application ends its operation. Otherwise, you use the **`ForEach`** extension method to write each suggestion on a separate line.

As you can see, a trie provides you with a powerful and efficient mechanism for implementing an autocomplete feature. But what does it look like in practice? Let's take a look at the following output regarding searching for **`POLAND`**:

```
Enter next character: p
Suggestions for P:
PAKISTAN
PALAU
PANAMA
PAPUANEWGUINEA
PARAGUAY
PERU
PHILIPPINES
POLAND
PORTUGAL
Enter next character: o
Suggestions for PO:
POLAND
PORTUGAL
Enter next character: l
Suggestions for POL:
POLAND
Enter next character: e
```

At the beginning, you can see names of countries that start with **`P`**. After typing **`O`**, you limit the results to countries whose names start with **`PO`**. In the same way, you further increase the prefix and get fewer and fewer results.

Let's proceed to the last part of this chapter, which is related to heaps. What are they, and why are they featured in a chapter about trees?

Heaps

A **heap** is another variant of a tree, which you already got to know in *Chapter 3*, *Arrays and Sorting*. There, you used a heap in the heap sort algorithm for sorting an array. For this reason, in the current chapter, you will see only a brief summary of this data structure. However, I strongly encourage you not to leave this topic and learn much more about heaps on your own, as they are powerful and popular data structures.

As you already know, a binary heap exists in two versions: **min-heap** and **max-heap**. For each of them, an additional property must be satisfied:

- **For min-heap**: The value of each node must be greater than or equal to the value of its parent node
- **For max-heap**: The value of each node must be less than or equal to the value of its parent node

These rules perform a very important role because they dictate that **the root node always contains the smallest value (in the min-heap) or the largest value (in the max-heap)**. You benefited from this assumption while sorting. Do you remember?

A binary heap must also adhere to the **complete binary tree** rule, which requires that **each node cannot contain more than two children and all levels of a tree must be fully filled, except the last one, which must be filled from left to right** and can have some space on the right.

Let's take a look at the following two binary heaps:

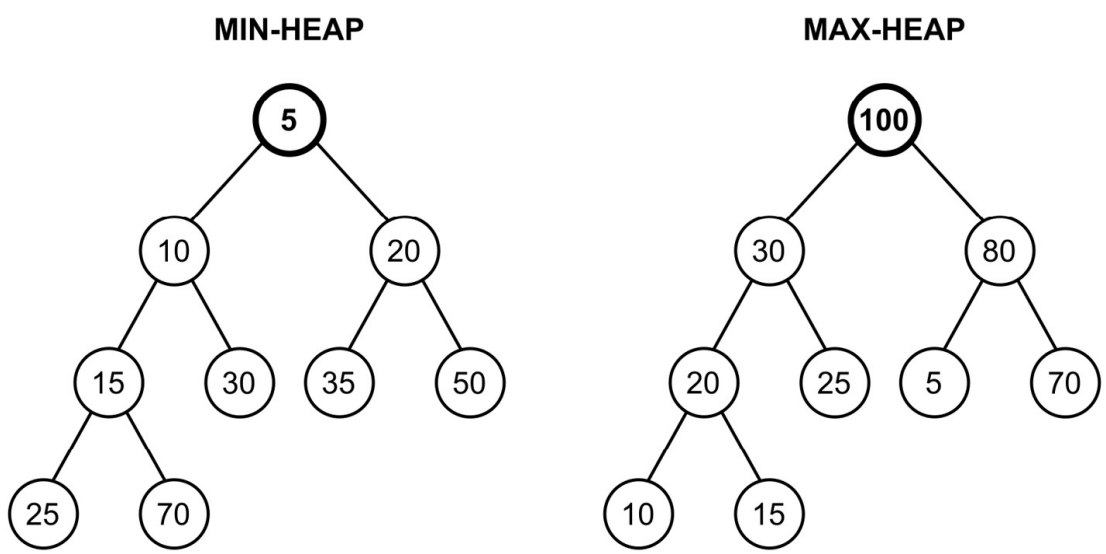

Figure 7.23 – Illustration of a min-heap and a max-heap

You can easily check whether both heaps adhere to all the rules. As an example, let's verify the heap property for the node with a value equal to **20** from the min-heap variant (shown on the left). The

node has two children with values of **35** and **50**, which are both greater than **20**. In the same way, you can check the remaining nodes in the heap.

The binary tree rule is also maintained, as each node contains at most two children. The last requirement is that each level of the tree is fully filled except the last one, which does not need to be fully filled, but must contain nodes from left to right. In the min-heap example, three levels are fully filled (with one, two, and four nodes), while the last level contains two nodes (**25** and **70**), placed on the two leftmost positions. In the same way, you can confirm that the max-heap (shown on the right) is configured properly.

At the end of this short introduction to the topic of heaps, and especially to binary heaps, it is worth mentioning the broad range of applications. First of all, this data structure is a convenient way of implementing a **priority queue** with the operation of inserting a new value and removing the smallest value (in the min-heap) or the largest value (in the max-heap). Moreover, a heap is used in the **heap sort algorithm**, as well as in **graph algorithms**.

A binary heap can either be implemented from scratch or you can use some of the already available implementations as NuGet packages. One of the solutions is named `PommaLabs.Hippie` and can be easily installed on the project using the **NuGet Package Manager**. The mentioned library contains an implementation of a few variants of heaps, including binary heaps, **binomial heaps**, and **Fibonacci heaps**.

Trees were everywhere in this chapter, and heaps are also representatives of this data structure! As you already learned a lot about trees, let's proceed to the *Summary* section.

Summary

The current chapter was the longest so far in the book. However, it contained a lot of information about variants of trees. Such data structures perform a very important role in many algorithms, and it is good to learn more about them, as well as to know how to use them in your applications. For this reason, this chapter contained not only short theoretical introductions but also diagrams, explanations, and code samples.

At the beginning, **the concept of a tree** was described. As a reminder, a tree consists of **nodes**, including one **root**. The root does not contain a parent node, while all other nodes do. Each node can have any number of **child nodes**. The child nodes of the same node can be named **siblings**, while a node without children is named a **leaf**.

Various variants of trees follow this structure. The first one described in the chapter is a **binary tree**. In this case, a node can contain at most two children. However, the rules for **binary search trees** are

even more strict. For any node in such trees, the values of all nodes in its left subtree must be smaller than the value of the node, while the values of all nodes in its right subtree must be greater than the value of the node. BSTs have a broad range of applications and provide developers with significant improvements in the lookup performance. However, it is possible to easily make a tree unbalanced while adding sorted values to the tree. Thus, the positive impact on the performance can be limited.

Fortunately, **self-balancing trees** exist and remain balanced all the time while adding or removing nodes. Two variants were presented: AVL trees and red-black trees. An **AVL tree** has an additional requirement that, for each node, the height of its left and right subtrees cannot differ by more than one. An **RBT** introduces the concept of coloring nodes, either red or black, as well as `NIL` pseudo-nodes. Moreover, it is required that if a node is red, both its children must be black, and for any node, the number of black nodes on the route to a descendant leaf must be the same.

Then, you learned a lot about **tries** and saw their great performance regarding processing strings, such as for autocomplete or spellchecker features. Each trie is a tree with one root node, where each node represents a string and each edge indicates a character. A trie node contains references to the next nodes as an array with elements representing possible characters. When you go from the root to each node, you receive a string, which is either a saved word or its substring. Within this part, a **radix tree** was mentioned as well, which is a space-optimized version of a trie.

The remaining part of the chapter was related to **binary heaps**. As a reminder, a heap is another variant of a tree, which exists in two versions, **min-heap** and **max-heap**. It is worth noting that the value of each node must be greater than or equal to (for min-heaps) or less than or equal to (for max-heaps) the value of its parent node.

Let's proceed to **graphs**, which are the subject of the next chapter!

8

Exploring Graphs

In the previous chapter, you learned about trees. However, did you know that such data structures also belong to graphs? But what is a graph and how can you use one in your applications? You'll find the answers to these and many other questions in this chapter!

First, basic information about graphs will be presented, including an explanation of **nodes** and **edges**. As graphs are data structures that are commonly used in practice, you will also see some of their applications, such as for storing data of friends on social media or for finding a road in a city. Then, the topic of graph **representation** will be covered, namely using an adjacency list and matrix.

After this short introduction, you will learn how to implement a graph in the C# language. Moreover, you will learn about two modes of graph **traversal**, namely **depth-first search** (**DFS**) and **breadth-first search** (**BFS**). For both of them, the code and a detailed description will be shown.

Next, you will learn about the subject of **minimum spanning trees** (**MSTs**), as well as two algorithms for their creation, namely Kruskal's and Prim's. Such algorithms will be presented as descriptions, code snippets, and illustrations. Moreover, an example real-world application will be provided.

Another interesting graph-related problem is the **coloring** of nodes, which will be taken into account in the following part of this chapter. Finally, the topic of finding the **shortest path** in a graph will be analyzed using Dijkstra's algorithm.

As you can see, the topic of graphs involves many interesting problems and only some of them will be mentioned in this book. However, the chosen subjects are suitable for presenting various graph-related aspects in the context of the C# language. Are you ready to dive into the topic of graphs? If so, start reading this chapter!

In this chapter, the following topics will be covered:

- The concept of graphs
- Applications
- Representations
- Implementation
- Traversal
- Minimum spanning tree

- Coloring
- Shortest path

The concept of graphs

Let's start with the question *what is a graph?* Broadly speaking, **a graph is a data structure that consists of nodes (also called vertices) and edges. Each edge connects two nodes**. A graph data structure does not require any specific rules regarding connections between nodes, as shown in the following diagram:

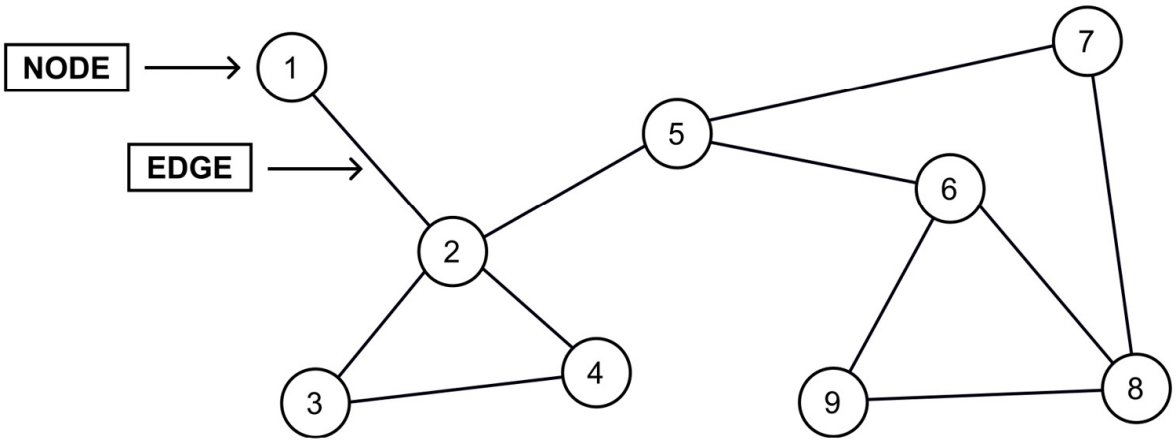

Figure 8.1 – Illustration of a graph

This concept seems very simple, doesn't it? Let's try to analyze the preceding graph to eliminate any doubts. It contains **9 nodes** with numbers between **1** and **9** as values. Such nodes are connected by **11 edges**, such as between nodes **2** and **4**. Moreover, a graph can contain **cycles** – for example, with nodes indicated by **2**, **3**, and **4** – as well as separate groups of nodes, which are not connected.

However, what about the topic of parent and child nodes, which you know from learning about trees? As there are no specific rules about connections in a graph, such concepts are not used in this case.

IMAGINE A GRAPH

If you want to better visualize a graph, take your eyes off this book for a moment and look at a map showing the most important roads in your country, such as highways or expressways. Each fragment of such a road connects two towns and has a certain length. Once you have drawn such a structure on a piece of paper, you will see that thanks to it, you can find a route between two towns, along with the total distance of the entire route. Did you know you just created a graph? Individual towns are nodes, and the lines connecting them are edges. The distance between the two towns is the edge weight. It's so simple when you can relate theory to practice, isn't it? Now, it's high time to put the map aside and focus on learning about the last data structure that will be covered in this book, namely the graph.

Some more comments are necessary for edges in a graph. In the preceding diagram, you can see a graph where all the nodes are connected with **undirected edges** – that is, **bidirectional edges**. They indicate that **it is possible to travel between nodes in both directions** – for example, from node **2** to **3** and from node **3** to **2**. Such edges are presented graphically as **straight lines**. When a graph contains undirected edges, it is an **undirected graph**.

However, what about a scenario when you need to indicate that **traveling between nodes is possible only in one direction**? In such a case, you can use **directed edges** – that is, **unidirectional edges** – which are presented graphically as **straight lines with arrows indicating the direction of an edge**. If a graph contains directed edges, it can be named a **directed graph**.

WHAT ABOUT SELF-LOOPS?

*A graph can also contain **self-loops**. Each is an edge that connects a given node with itself. However, such a topic is outside the scope of this book and won't be taken into account in the examples shown in this chapter.*

An example directed graph is presented in the following diagram on the right, while an undirected one is shown on the left:

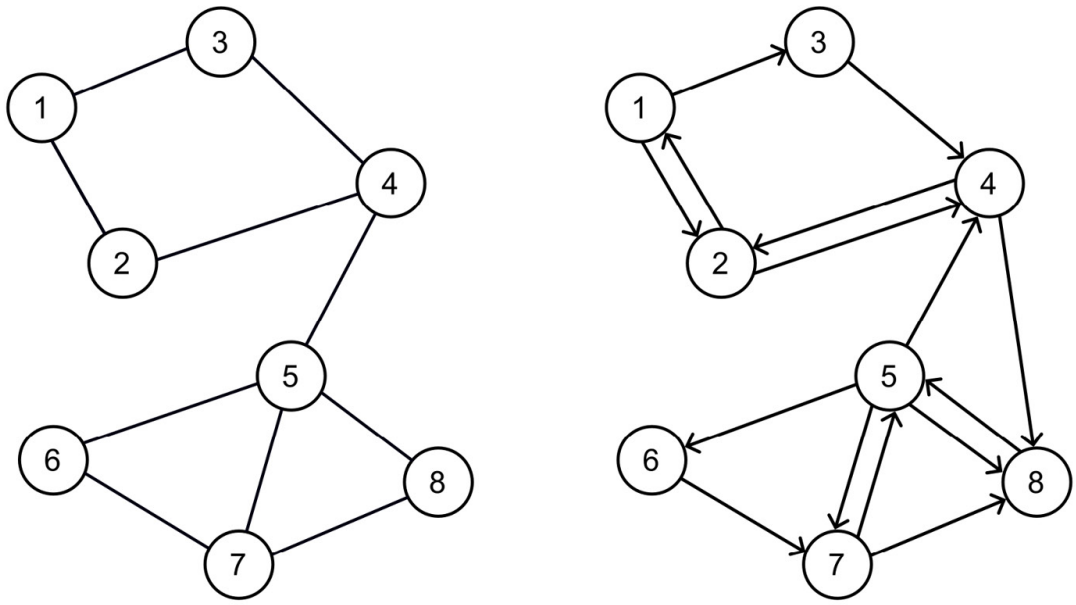

Figure 8.2 – The difference between undirected and directed graphs

As a short explanation, the directed graph (shown on the right in the preceding diagram) contains **8 nodes** connected by **15 unidirectional edges**. For example, they indicate that it is possible to travel between node **1** and **2** in both directions, but it is allowed to travel from the node **1** to **3** only in one direction, so it is impossible to reach node **1** from **3** directly.

The division between undirected and directed edges is not the only one. You can also specify **weights** (also referred to as **costs**) for particular edges to indicate the cost of traveling between nodes. Of course, such weights can be assigned to both undirected and directed edges. If weights are provided, an edge is named a **weighted edge**, and the whole graph is named a **weighted graph**. Similarly, if no weights are provided, **unweighted edges** are used in a graph. This graph is then called an **unweighted graph**.

Some example weighted graphs with undirected (on the left) and directed (on the right) edges are shown in the following diagram:

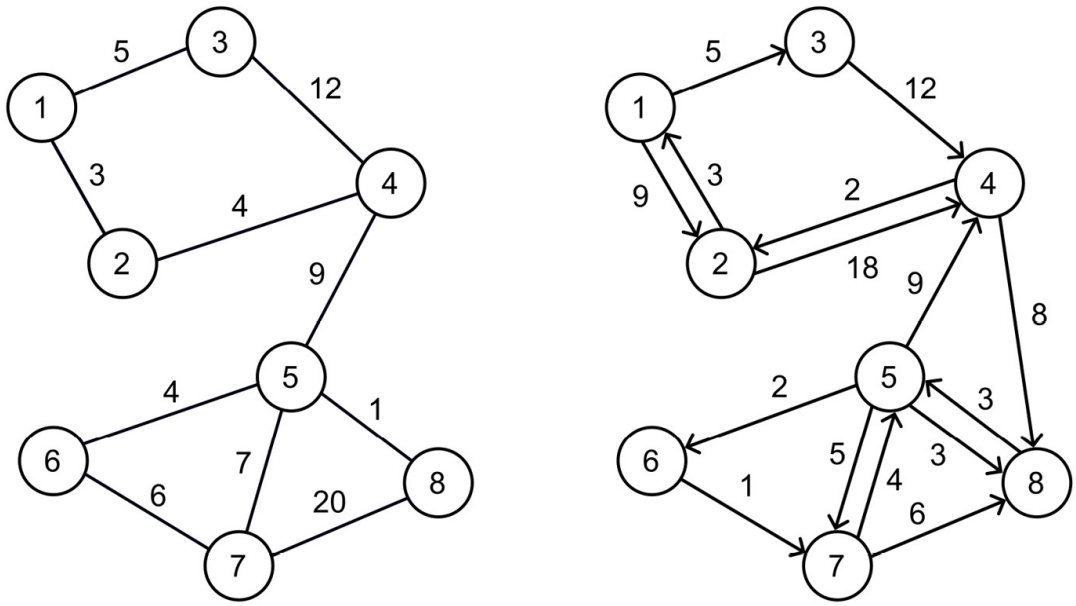

Figure 8.3 – Difference between weighted undirected and weighted directed graphs

This graphical presentation of a weighted edge shows the weight of an edge next to the line. For example, the cost of traveling from node **1** to **2**, as well as from node **2** to **1**, is equal to **3** in the case of the undirected graph, shown on the left in the preceding diagram. The situation is a bit more complicated in the case of the directed graph (on the right). Here, you can travel from node **1** to **2** with a cost equal to **9**, while traveling in the opposite direction (from node **2** to **1**) is much cheaper and costs only **3**.

Applications

At this point, you know some basic information about graphs, especially regarding nodes and various kinds of edges. However, why is the topic of graphs so important and why does it take up a whole chapter in this book? Could you use this data structure in your applications? The answer is obvious:

yes! Graphs are commonly used while solving algorithmic problems and have numerous real-world applications.

To start, let's think about a **structure of friends available on social media**. Each user has many contacts, but they also have many friends, and so on. What data structure should you choose to store such data? A graph is the simplest answer. In such a scenario, the nodes represent contacts, while the edges depict relationships between people. As an example, let's take a look at the following diagram of an undirected and unweighted graph:

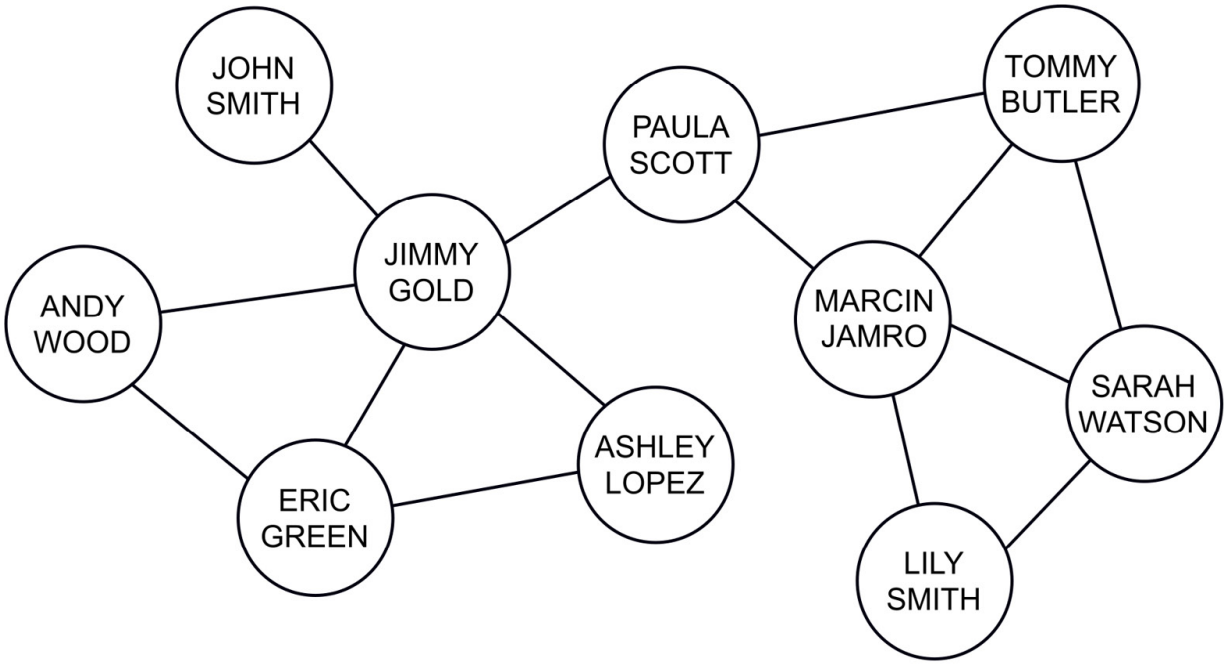

Figure 8.4 – Illustration of a graph representing a structure of friends

As you can see, **Jimmy Gold** has five contacts, namely **John Smith**, **Andy Wood**, **Eric Green**, **Ashley Lopez**, and **Paula Scott**. In the meantime, **Paula Scott** has two other friends: **Marcin Jamro** and **Tommy Butler**. By using a graph as a data structure, you can easily check whether two people are friends or whether they have a common contact.

Another common application of graphs involves the problem of **searching for the shortest path**. Let's imagine a program that should find a path between two points in the city, taking into account the time necessary for driving particular roads. In such a case, you can use a graph to present a map of a city, where nodes depict intersections and edges represent roads. Of course, you should assign weights to edges to indicate the time that's necessary to drive a given road. The topic of searching the shortest path can be understood as finding the list of edges from the source to the target node, with the minimum total cost. A diagram of a city map, based on a graph, is shown here:

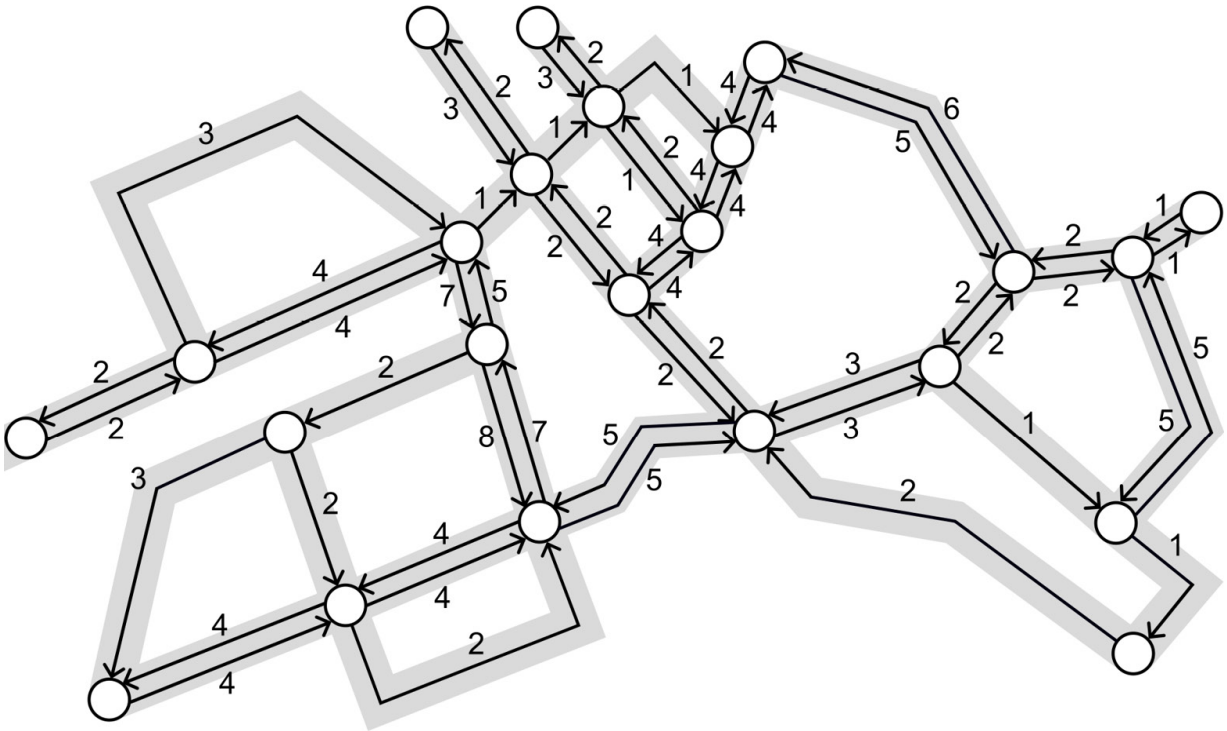

Figure 8.5 – Illustration of a graph representing a city map

As you can see, the directed and weighted graph was chosen. Directed edges make it possible to support both two-way and one-way roads, while weighted edges allow you to specify the time necessary to travel between two intersections.

Representations

At this point, you know what a graph is and when one can be used, but how can you represent one in the memory of a computer? There are two popular approaches to solve this problem, namely using an **adjacency list** and an **adjacency matrix**.

Adjacency list

The first approach requires you to **extend the data of a node by specifying a list of its neighbors**. Thus, you can easily get all the neighbors of a given node just by iterating through the adjacency list of a given node. Such a solution is space-efficient because you only store the data of adjacent edges. Let's take a look at the diagram:

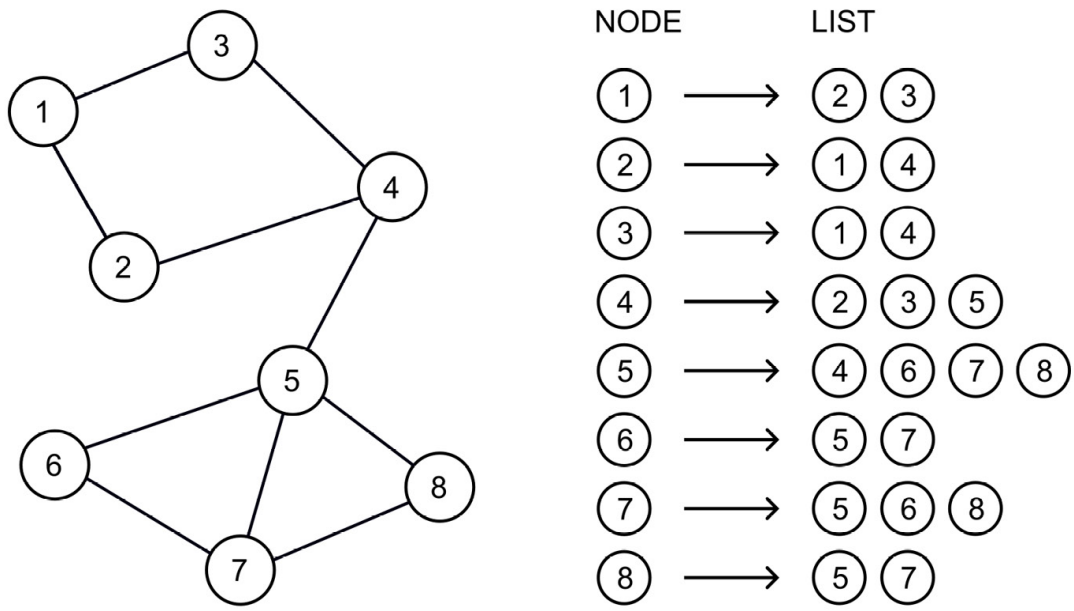

Figure 8.6 – Adjacency list representing an undirected and unweighted graph

This example graph contains 8 nodes and 10 edges. For each node, a list of adjacent nodes (that is, **neighbors**) is created, as shown on the right-hand side of the diagram. For example, node **1** has two neighbors, namely nodes **2** and **3**, while node **5** has four neighbors, namely nodes **4**, **6**, **7**, and **8**. As you can see, the representation based on the adjacency list for an undirected and unweighted graph is straightforward, as well as easy to use, understand, and implement.

But how does the adjacency list work in the case of a directed graph? The answer is obvious because the list that's assigned to each node just shows adjacent nodes that can be reached from the given node. Here's an example diagram:

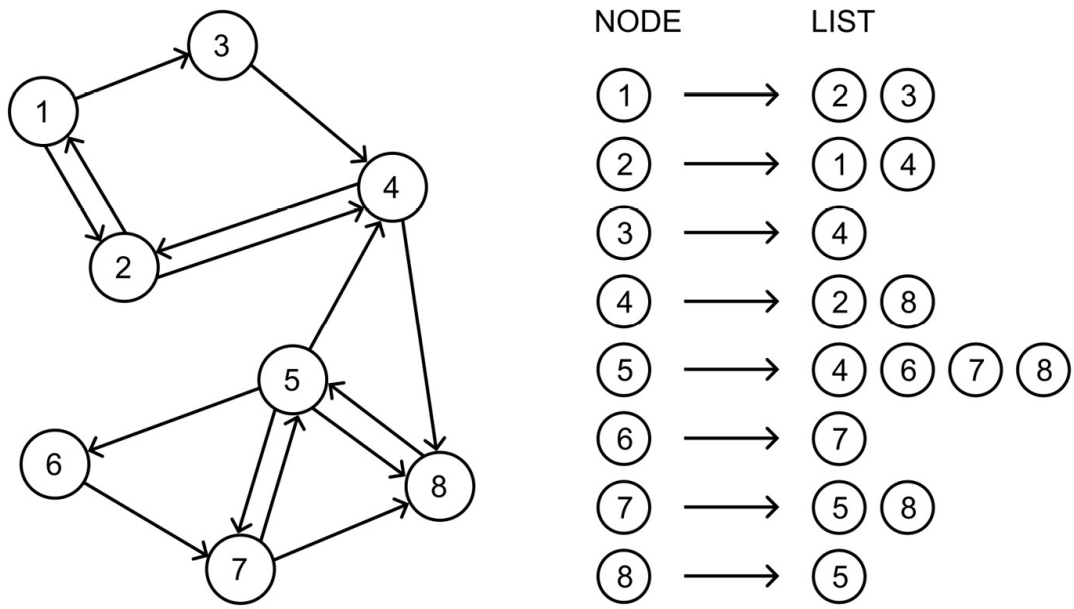

Figure 8.7 – Adjacency list representing a directed and unweighted graph

Let's take a look at node **3**. Here, the adjacency list contains only one element – that is, node **4**. Node **1** is not included, because it cannot be reached directly from node **3**.

A bit more clarification may be useful in the case of a weighted graph. In such a case, it is also necessary to store weights for particular edges. You can achieve this by extending data stored in the adjacency list, as shown in the following diagram:

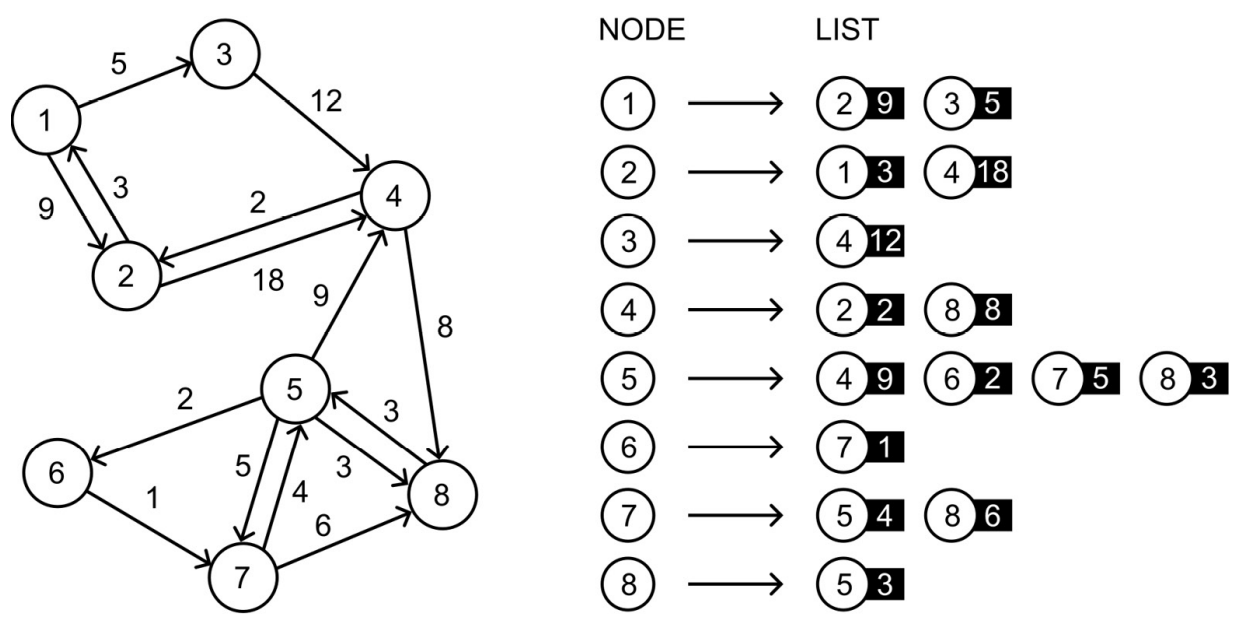

Figure 8.8 – Adjacency list representing a directed and weighted graph

For example, the adjacency list for node **7** contains two elements, namely regarding an edge to node **5** (with a weight equal to **4**) and to node **8** (with a weight equal to **6**).

Adjacency matrix

Another approach to graph representation involves the adjacency matrix, which uses **a two-dimensional array to show which nodes are connected by edges**. The matrix contains the same number of rows and columns, which is equal to the number of nodes. The main idea is to **store information about a particular edge in an element at a given row and column in the matrix**. The index of the row and the column depends on the nodes connected with the edge. For example, if you want to get information about an edge between nodes with indices **1** and **5**, you must check the element in the row with an index set to **1** and in the column with an index set to **5**.

Such a solution provides you with a **quick way of checking whether two particular nodes are connected by an edge**. However, it may require you to store significantly more data than the adjacency list, especially if the graph does not contain many edges between nodes.

To start, let's analyze the basic scenario of an undirected and unweighted graph. In such a case, the adjacency matrix may only store Boolean values. The `true` value that's placed in the element at the `i` row and the `j` column indicates that there is a connection between a node with an index equal to `i` and the node with an index set to `j`. If this sounds complicated, take a look at the following figure:

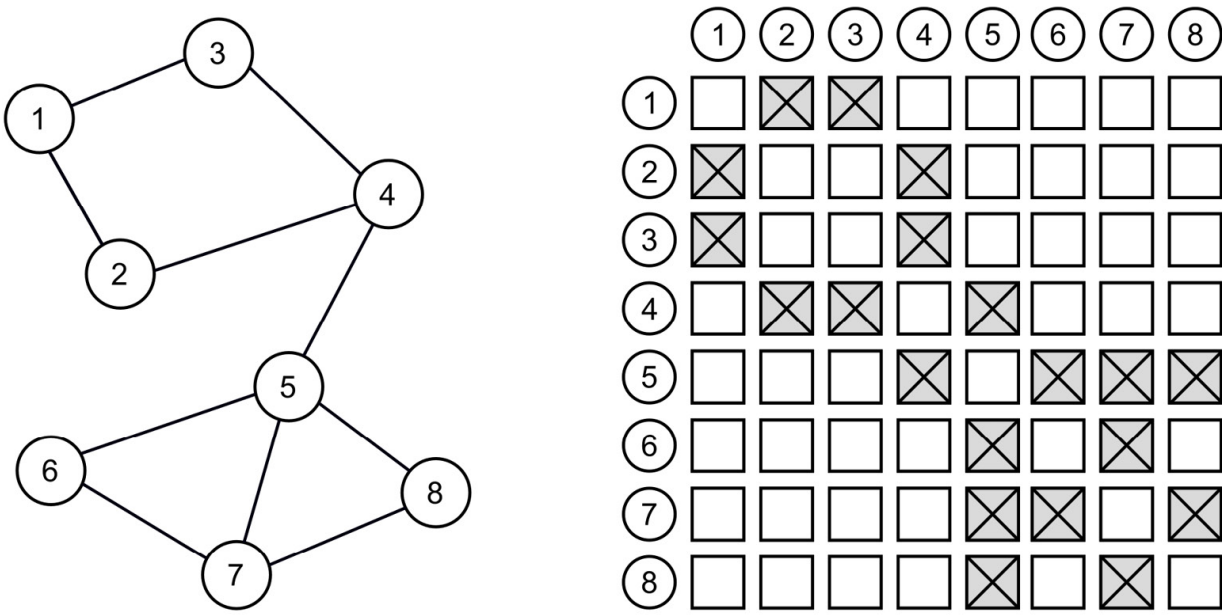

Figure 8.9 – Adjacency matrix representing an undirected and unweighted graph

Here, the adjacency matrix contains 64 elements (for 8 rows and 8 columns) because there are 8 nodes in the graph. The values of many elements in the array are set to `false`, which is represented by missing indicators. The remaining are marked with crosses, representing `true` values. For example, such a value in the element at the fourth row and third column means that there is an edge between nodes **4** and **3**, as shown in the preceding diagram.

SYMMETRIC ADJACENCY MATRIX

As the preceding graph is undirected, the adjacency matrix is symmetric. If there is an edge between nodes `i` and `j`, there is also an edge between nodes `j` and `i`.

The following example involves a directed and unweighted graph. In such a case, the same rules can be used, but the adjacency matrix does not need to be symmetric. Let's take a look at the illustration of the graph, presented together with the adjacency matrix:

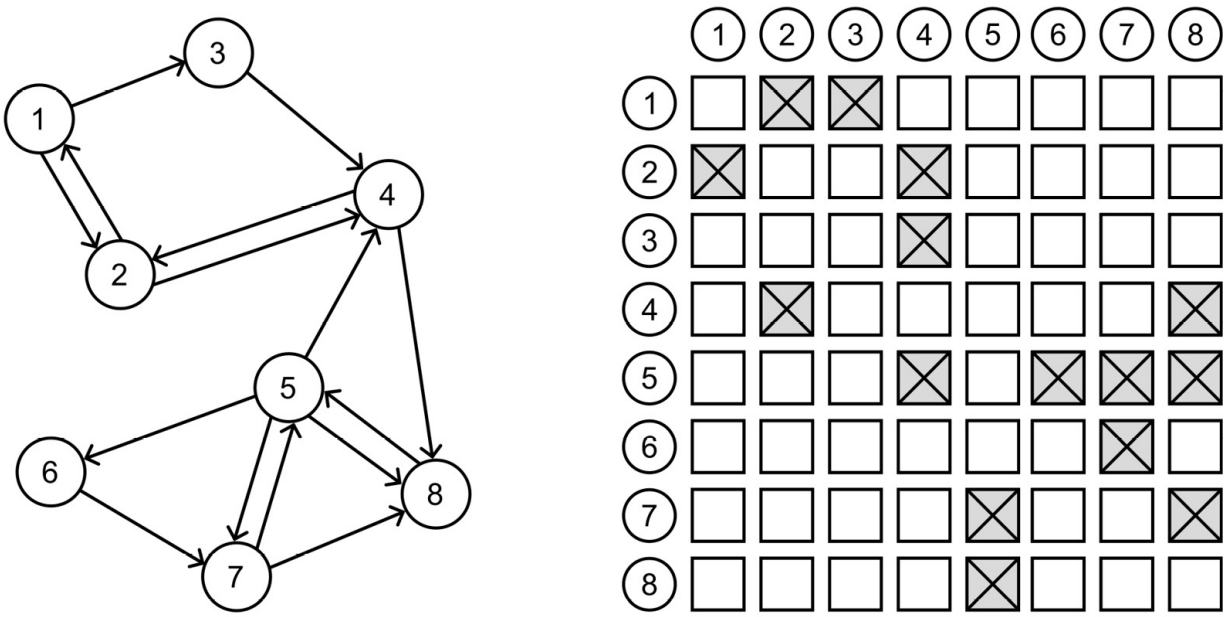

Figure 8.10 – Adjacency matrix representing a directed and unweighted graph

Within the shown adjacency matrix, you can find data of 15 edges, represented by 15 elements with `true` values, indicated by crosses in the matrix. For example, the unidirectional edge from node **5** to **4** is shown as the cross at the fifth row and the fourth column.

In both previous examples, you learned how to present an unweighted graph using an adjacency matrix. However, how you can store the data of the weighted graph, either undirected or directed? The answer is very simple – you just need to change the type of data stored in particular elements in

the adjacency matrix from Boolean to numeric. Thus, you can specify the weight of edges, as shown in the following figure:

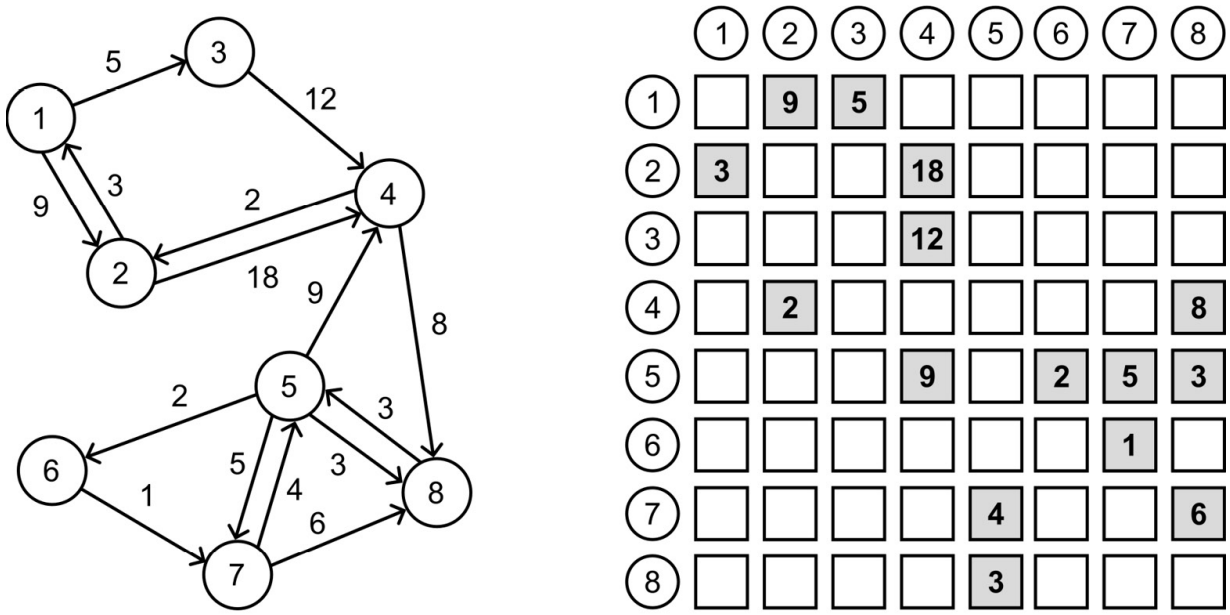

Figure 8.11 – Adjacency matrix representing a directed and weighted graph

To eliminate any doubt, let's take a look at the edge between nodes **5** and **6** with the weight set to **2**. Such an edge is represented by the element at the fifth row and the sixth column. The value of the element is equal to the cost of traveling between nodes.

Implementation

You already know some basic information about graphs, including nodes, edges, and two methods of representation, namely using an adjacency list and matrix. However, how you can use such a data structure in your applications? In this section, you will learn how to implement a graph using the C# language. To make your understanding of this content easier, two examples will be provided.

Node

To start with, let's take a look at the code of a generic class representing a single node in a graph. Such a class is named **`Node`** and its code is as follows:

```
public class Node<T>
{
    public int Index { get; set; }
    public required T Data { get; set; }
```

```
    public List<Node<T>> Neighbors { get; set; } = [];
    public List<int> Weights { get; set; } = [];
    public override string ToString() => $"Index: {Index}.
        Data: {Data}. Neighbors: {Neighbors.Count}.";
}
```

The class contains four properties. Since all of these elements perform important roles in the code snippets shown in this chapter, let's analyze them in detail:

- The first property (**`Index`**) stores an index of a particular node in a collection of nodes in a graph to simplify the process of accessing a particular element. Thus, it is possible to easily get an instance of the **`Node`** class by using an index.
- The next property is named **`Data`** and just stores some data in the node. It is worth mentioning that the type of such data is consistent with the type specified while creating an instance of the generic class.
- The **`Neighbors`** property represents the adjacency list for a particular node. Thus, it contains references to the **`Node`** instances representing adjacent nodes.
- The last property is named **`Weights`** and stores weights assigned to adjacent edges. In the case of a weighted graph, the number of elements in the **`Weights`** list is the same as the number of neighbors (**`Neighbors`**). If a graph is unweighted, the **`Weights`** list is empty.

Apart from the aforementioned properties, the class contains the overridden **`ToString`** method, which returns the textual representation of the object. Here, the string is returned in **`"Index: [index]. Data: [data]. Neighbors: [count]."`** format.

Edge

As mentioned in the short introduction to the topic of graphs, a graph consists of nodes and edges. As a node is represented by an instance of the **`Node`** class, the generic **`Edge`** class can be used to represent an edge. The suitable part of the code is as follows:

```
public class Edge<T>
{
    public required Node<T> From { get; set; }
    public required Node<T> To { get; set; }
    public int Weight { get; set; }
    public override string ToString() => $"{From.Data}
        -> {To.Data}. Weight: {Weight}.";
}
```

The class contains three properties, namely representing nodes adjacent to the edge (**`From`** and **`To`**), as well as the weight of the edge (**`Weight`**). Moreover, the **`ToString`** method is overridden to present some basic information about the edge.

Graph

The next class is named **Graph** and represents a whole graph, with either directed or undirected edges, as well as either weighted or unweighted edges. The implementation consists of various properties and methods. These are described in detail here.

Let's take a look at the basic version of the **Graph** class:

```
public class Graph<T>
{
    public required bool IsDirected { get; init; }
    public required bool IsWeighted { get; init; }
    public List<Node<T>> Nodes { get; set; } = [];
}
```

The class contains two properties indicating whether edges are directed (**IsDirected**) and weighted (**IsWeighted**). Moreover, the **Nodes** property is declared, which stores a list of nodes existing in the graph.

The next interesting member of the **Graph** class is the indexer, which takes two indices, namely indices of two nodes, to return an instance of the **Edge** class representing an edge between such nodes. The implementation is shown here:

```
public Edge<T>? this[int from, int to]
{
    get
    {
        Node<T> nodeFrom = Nodes[from];
        Node<T> nodeTo = Nodes[to];
        int i = nodeFrom.Neighbors.IndexOf(nodeTo);
        if (i < 0) { return null; }
        Edge<T> edge = new()
        {
            From = nodeFrom,
            To = nodeTo,
            Weight = i < nodeFrom.Weights.Count
                ? nodeFrom.Weights[i] : 0
        };
        return edge;
    }
}
```

Within the indexer, you get instances of the **Node** class representing two nodes (**nodeFrom** and **nodeTo**) according to the indices. As you want to find an edge from the first node (**nodeFrom**) to the second one (**nodeTo**), you need to try to find the second node in the collection of neighbor nodes of the first node using the **IndexOf** method. If such a connection does not exist, the **IndexOf** method returns a negative value, and **null** is returned by the indexer. Otherwise, you create a new instance of the **Edge** class and set the values of its properties, including **From** and **To**. If the data regarding the weight of particular edges is provided, the value of the **Weight** property of the **Edge** class is set as well.

At this point, you know how to store data of nodes in the graph, but how can you add a new node? To do so, you can implement the **AddNode** method, as follows:

```
public Node<T> AddNode(T value)
{
    Node<T> node = new() { Data = value };
    Nodes.Add(node);
    UpdateIndices();
    return node;
}
```

Within this method, you create a new instance of the **`Node`** class and set a value of the **`Data`** property, according to the value of the parameter. Then, the newly created instance is added to the **`Nodes`** collection, and the **`UpdateIndices`** method (described later) is called to update the indices of all the nodes stored in the collection. Finally, the **`Node`** instance, representing the newly added node, is returned.

You can remove the existing node as well. This operation is performed by the **`RemoveNode`** method, as shown in the following code snippet:

```
public void RemoveNode(Node<T> nodeToRemove)
{
    Nodes.Remove(nodeToRemove);
    UpdateIndices();
    Nodes.ForEach(n => RemoveEdge(n, nodeToRemove));
}
```

This method takes one parameter, namely an instance of the node that should be removed. First, you remove it from the collection of nodes. Then, you update the indices of the remaining nodes. Finally, you iterate through all the nodes in the graph to remove all edges that are connected with the node that has been removed.

As you already know, a graph consists of nodes and edges. Thus, the implementation of the **`Graph`** class should provide developers with a method for adding a new edge. Of course, it should support various variants of edges, either directed, undirected, weighted, or unweighted. The proposed implementation is as follows:

```
public void AddEdge(Node<T> from, Node<T> to, int w = 0)
{
    from.Neighbors.Add(to);
    if (IsWeighted) { from.Weights.Add(w); }
    if (!IsDirected)
    {
        to.Neighbors.Add(from);
        if (IsWeighted) { to.Weights.Add(w); }
    }
}
```

The **`AddEdge`** method takes three parameters, namely two instances of the **`Node`** class representing nodes connected by the edge (**`from`** and **`to`**), as well as the weight of the connection (**`w`**), which is set to **`0`** by default.

In the first line within the method, you add the **Node** instance representing the second node to the list of neighbor nodes of the first one. If the weighted graph is considered, a weight of the aforementioned edge is added as well.

The following part of the code is only taken into account when the graph is undirected. In such a case, you need to automatically add an edge in the opposite direction. To do so, you add the **Node** instance representing the first node to the list of neighbor nodes of the second one. If the edges are weighted, a weight of the aforementioned edge is added to the **Weights** list as well.

The process of removing an edge from the graph is supported by the **RemoveEdge** method. The code is as follows:

```
public void RemoveEdge(Node<T> from, Node<T> to)
{
    int index = from.Neighbors.FindIndex(n => n == to);
    if (index < 0) { return; }
    from.Neighbors.RemoveAt(index);
    if (IsWeighted) { from.Weights.RemoveAt(index); }
    if (!IsDirected)
    {
        index = to.Neighbors.FindIndex(n => n == from);
        if (index < 0) { return; }
        to.Neighbors.RemoveAt(index);
        if (IsWeighted) { to.Weights.RemoveAt(index); }
    }
}
```

This method takes two parameters, namely two nodes (**from** and **to**), between which there is an edge that should be removed. To start, you try to find the second node in the list of neighbor nodes of the first one. If it is found, you remove it. You should also remove the weight data if the weighted graph is considered. In the case of an undirected graph, you automatically remove a node in an opposite direction, namely between the **to** and **from** nodes.

The last public method is named **GetEdges** and makes it possible to get a collection of all the edges that are available in the graph. The proposed implementation is as follows:

```
public List<Edge<T>> GetEdges()
{
    List<Edge<T>> edges = [];
    foreach (Node<T> from in Nodes)
    {
        for (int i = 0; i < from.Neighbors.Count; i++)
        {
            int weight = i < from.Weights.Count
                ? from.Weights[i] : 0;
            Edge<T> edge = new()
            {
                From = from,
                To = from.Neighbors[i],
                Weight = weight
            };
            edges.Add(edge);
```

```
            }
        }
        return edges;
    }
```

First, a new list of edges is initialized. Then, you iterate through all the nodes in the graph using a **foreach** loop. Within it, you use a **for** loop to create instances of the **Edge** class. The number of instances should be equal to the number of neighbors of the current node (the **from** variable in the **foreach** loop). In the **for** loop, the newly created instance of the **Edge** class is configured by setting values of its properties, namely the first node (the **from** variable – that is, the current node from the **foreach** loop), the second node (to the currently-analyzed neighbor), and the weight. Then, the newly created instance is added to the collection of edges, represented by the **edges** variable. Finally, the result is returned.

In various methods, you use the **UpdateIndices** method. Its code is as follows:

```
private void UpdateIndices()
{
    int i = 0;
    Nodes.ForEach(n => n.Index = i++);
}
```

This method iterates through all the nodes in the graph and updates the values of the **Index** property to the consecutive number, starting from **0**. It is worth noting that the iteration is performed using the **ForEach** method, instead of using a **foreach** or **for** loop.

Now, you know how to create a basic implementation of a graph. The next step is to apply it to represent some example graphs.

Example – undirected and unweighted edges

Let's try to use the previous implementation to create an undirected and unweighted graph according to the following diagram:

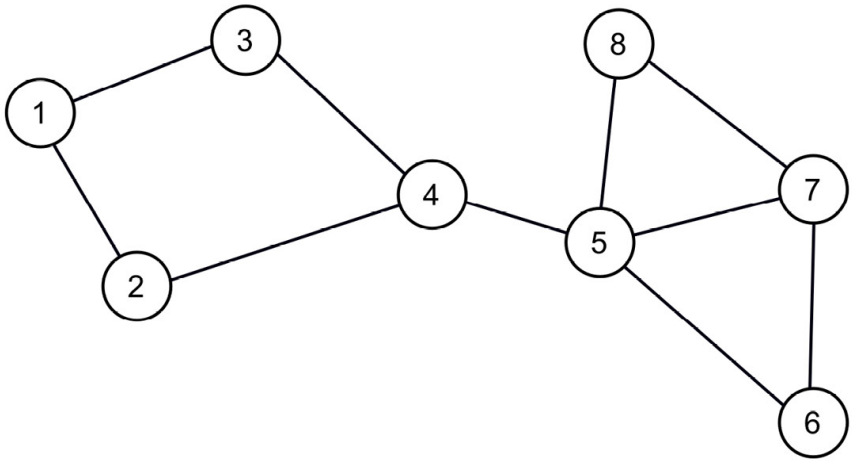

Figure 8.12 – Illustration of the undirected and unweighted edges example

As you can see, the graph contains 8 nodes and 10 edges. The implementation starts with the following line, which initializes a new undirected and unweighted graph:

```
Graph<int> graph = new()
    { IsDirected = false, IsWeighted = false };
```

Then, you add the necessary nodes and store references to them as new variables of the **`Node<int>`** type, as follows:

```
Node<int> n1 = graph.AddNode(1);
Node<int> n2 = graph.AddNode(2);
Node<int> n3 = graph.AddNode(3);
Node<int> n4 = graph.AddNode(4);
Node<int> n5 = graph.AddNode(5);
Node<int> n6 = graph.AddNode(6);
Node<int> n7 = graph.AddNode(7);
Node<int> n8 = graph.AddNode(8);
```

Finally, you only need to add edges between nodes, as shown in the preceding diagram. The necessary code is as follows:

```
graph.AddEdge(n1, n2);
graph.AddEdge(n1, n3);
graph.AddEdge(n2, n4);
graph.AddEdge(n3, n4);
graph.AddEdge(n4, n5);
graph.AddEdge(n5, n6);
graph.AddEdge(n5, n7);
graph.AddEdge(n5, n8);
graph.AddEdge(n6, n7);
graph.AddEdge(n7, n8);
```

That’s all! As you can see, configuring a graph is very easy using the proposed implementation of this data structure. Now, let’s proceed to a slightly more complex scenario with directed and weighted edges.

Example – directed and weighted edges

The following example involves a directed and weighted graph, as follows:

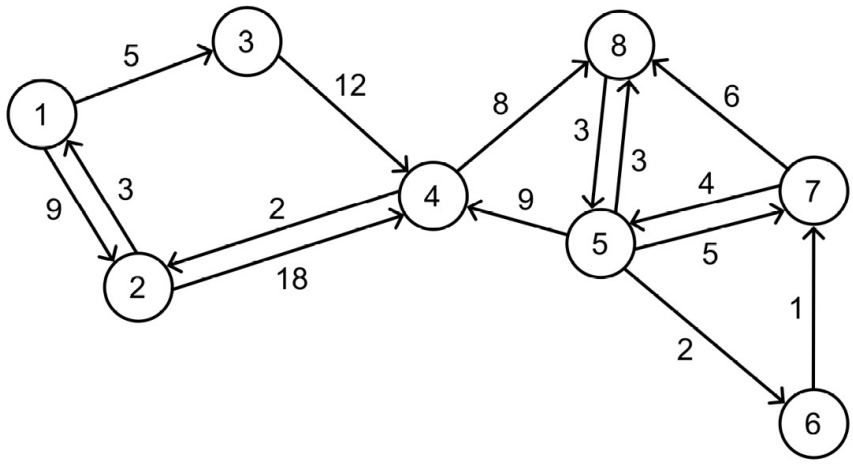

Figure 8.13 – Illustration of the directed and weighted edges example

The implementation is similar to the one described previously. However, some modifications are necessary. To start with, different values of the properties are used to indicate that a directed and weighted variant of the edges is being considered:

```
Graph<int> graph = new()
    { IsDirected = true, IsWeighted = true };
```

The part regarding adding nodes is the same as in the previous example:

```
Node<int> n1 = graph.AddNode(1);
Node<int> n2 = graph.AddNode(2);
Node<int> n3 = graph.AddNode(3);
Node<int> n4 = graph.AddNode(4);
Node<int> n5 = graph.AddNode(5);
Node<int> n6 = graph.AddNode(6);
Node<int> n7 = graph.AddNode(7);
Node<int> n8 = graph.AddNode(8);
```

Some changes are easily visible in the lines of code regarding the addition of edges. Here, you specify directed edges and their weights, as follows:

```
graph.AddEdge(n1, n2, 9);
graph.AddEdge(n1, n3, 5);
graph.AddEdge(n2, n1, 3);
graph.AddEdge(n2, n4, 18);
graph.AddEdge(n3, n4, 12);
graph.AddEdge(n4, n2, 2);
graph.AddEdge(n4, n8, 8);
graph.AddEdge(n5, n4, 9);
graph.AddEdge(n5, n6, 2);
graph.AddEdge(n5, n7, 5);
```

```
graph.AddEdge(n5, n8, 3);
graph.AddEdge(n6, n7, 1);
graph.AddEdge(n7, n5, 4);
graph.AddEdge(n7, n8, 6);
graph.AddEdge(n8, n5, 3);
```

You've just completed the basic implementation of a graph, shown in two examples. So, let's proceed to another topic, namely traversing a graph.

Traversal

One of the operations that's commonly performed on a graph is **traversal** – that is, **visiting all of the nodes in some particular order**. Of course, the aforementioned problem can be solved in various ways, such as using **DFS** or **BFS** approaches. It is worth mentioning that the traversal topic is strictly connected with the task of **searching for a given node in a graph**.

Depth-first search

The first graph traversal algorithm described in this chapter is named **DFS**. It tries to go as deep as possible. **First, it proceeds to the next levels of the nodes instead of visiting all the neighbors of the current node**. Its steps, in the context of the example graph, are as follows:

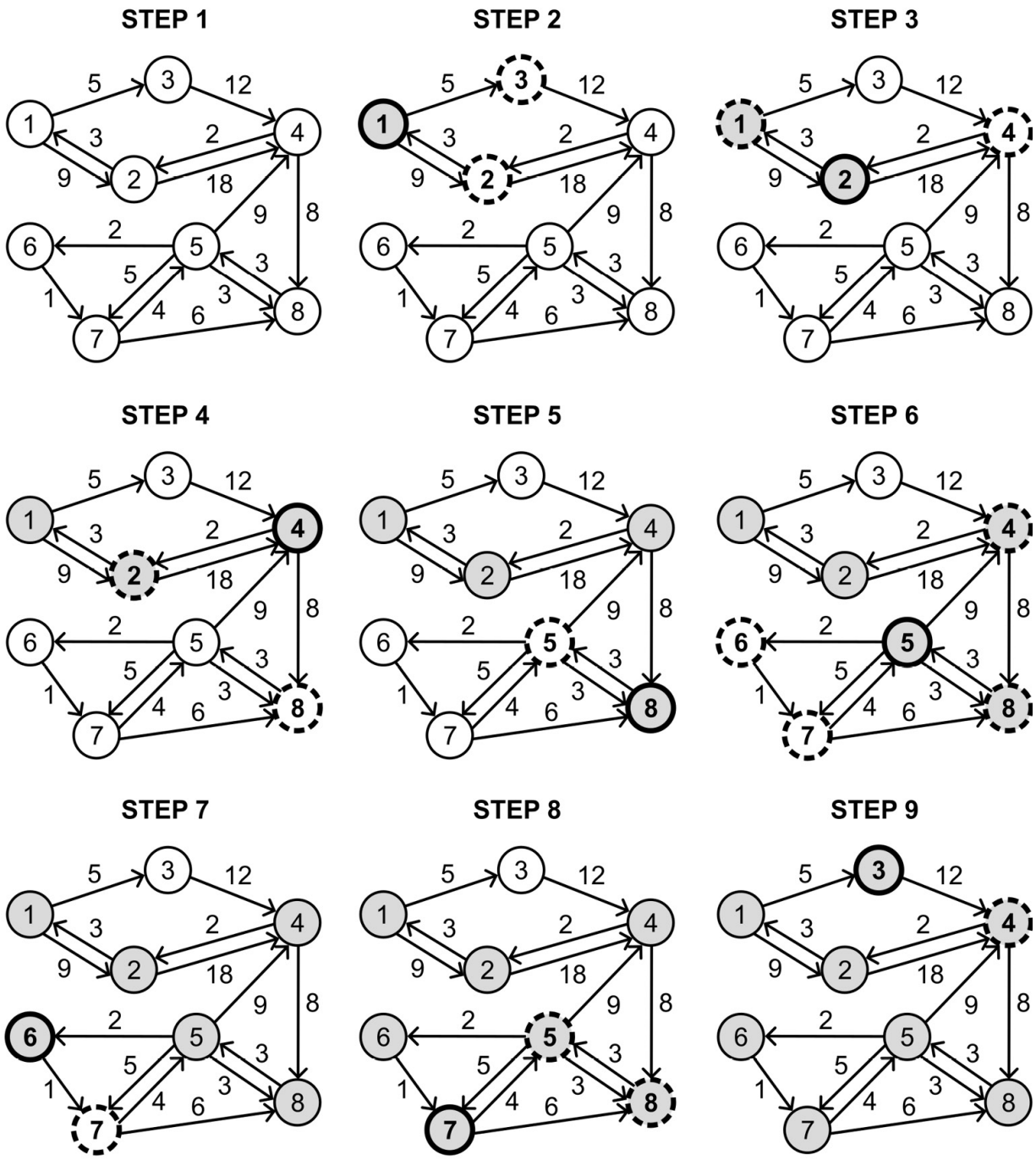

Figure 8.14 – Illustration of a DFS of a graph

Of course, it can be a bit difficult to understand how the DFS algorithm operates just by looking at the preceding diagram. For this reason, let's try to analyze its stages.

In **Step 1**, there's the graph with 8 nodes. In **Step 2**, node **1** is marked with a gray background (indicating that the node was already visited), as well as with a bolder border (indicating that it is the

node that is currently being visited). Moreover, an important role in the algorithm is performed by the neighbor nodes (shown as circles with dashed borders) of the current one. When you know the roles of particular indicators, it is clear that in **Step 2**, node **1** is visited. It has two neighbors, namely nodes **2** and **3**.

Then, the first neighbor (node **2**) is taken into account (**Step 3**) and the same operations are performed – that is, the node is visited and its neighbors (nodes **1** and **4**) are analyzed. As node **1** was visited, it is skipped. In **Step 4**, the first suitable neighbor of node **2** is taken into account, namely node **4**. It has two neighbors, namely node **2** (already visited) and **8**. Next, node **8** is visited (**Step 5**) and, according to the same rules, node **5** (**Step 6**). It has four neighbors, namely nodes **4** (already visited), **6**, **7**, and **8** (already visited). Thus, in **Step 7**, node **6** is taken into account. As it has only one neighbor (node **7**), it is visited next (**Step 8**).

Then, you check the neighbors of node **7**, namely nodes **5** and **8**. Both were already visited, so you return to the node with an unvisited neighbor. In this example, node **1** has one unvisited node, namely node **3**. When it is visited (**Step 9**), all nodes are traversed and no further operations are necessary.

Given this example, let's try to create the implementation in the C# language. To start, the code of the public **`DFS`** method (in the **`Graph`** class) is presented as follows:

```
public List<Node<T>> DFS()
{
    bool[] isVisited = new bool[Nodes.Count];
    List<Node<T>> result = [];
    DFS(isVisited, Nodes[0], result);
    return result;
}
```

The important role is performed by the **`isVisited`** array. It has the same number of elements as the number of nodes and stores values indicating whether a given node has already been visited. If so, the **`true`** value is stored. Otherwise, **`false`** is stored. The list of traversed nodes is represented as a list in the **`result`** variable. What's more, another variant of the **`DFS`** method is called here, passing three parameters:

- A reference to the **`isVisited`** array
- The first node to analyze
- The list for storing results

The code for the aforementioned variant of the **`DFS`** method is as follows:

```
private void DFS(bool[] isVisited, Node<T> node,
    List<Node<T>> result)
{
    result.Add(node);
    isVisited[node.Index] = true;
    foreach (Node<T> neighbor in node.Neighbors)
```

```
        {
            if (!isVisited[neighbor.Index])
            {
                DFS(isVisited, neighbor, result);
            }
        }
    }
```

First, the current node is added to the collection of traversed nodes, and the element in the **isVisited** array is updated. Then, you use the **foreach** loop to iterate through all the neighbors of the current node. For each of them, if they haven't already been visited, the **DFS** method is called recursively.

To finish, let's take a look at the code that can be placed in the **Program.cs** file. Its main parts are presented in the following code snippet:

```
Graph<int> graph = new()
    { IsDirected = true, IsWeighted = true };
Node<int> n1 = graph.AddNode(1); (...)
Node<int> n8 = graph.AddNode(8);
graph.AddEdge(n1, n2, 9); (...)
graph.AddEdge(n8, n5, 3);
List<Node<int>> nodes = graph.DFS();
nodes.ForEach(Console.WriteLine);
```

Here, you initialize a directed and weighted graph. It is worth noting that the missing lines of code (indicated by three dots) are the same as in the example where you created a graph with directed and weighted edges.

To start traversing the graph, you just need to call the **DFS** method, which returns a list of **Node** instances. Then, you can easily iterate through elements of the list to print some basic information about each node in the console:

```
Index: 0. Data: 1. Neighbors: 2.
Index: 1. Data: 2. Neighbors: 2.
Index: 3. Data: 4. Neighbors: 2.
Index: 7. Data: 8. Neighbors: 1.
Index: 4. Data: 5. Neighbors: 4.
Index: 5. Data: 6. Neighbors: 1.
Index: 6. Data: 7. Neighbors: 2.
Index: 2. Data: 3. Neighbors: 1.
```

That's all! As you can see, the algorithm tries to go as deep as possible and then goes back to find the next unvisited neighbor that can be traversed.

However, this algorithm is not the only approach to the problem of graph traversal. We'll cover another method and its implementation in the next section.

Breadth-first search

In the previous section, you learned about the DFS approach. Now, you will see another solution, namely **BFS**. Its main aim is to **visit all the neighbors of the current node and then proceed to the next level of nodes**.

If the previous description sounds a bit complicated, take a look at this diagram, which depicts the steps of the BFS algorithm:

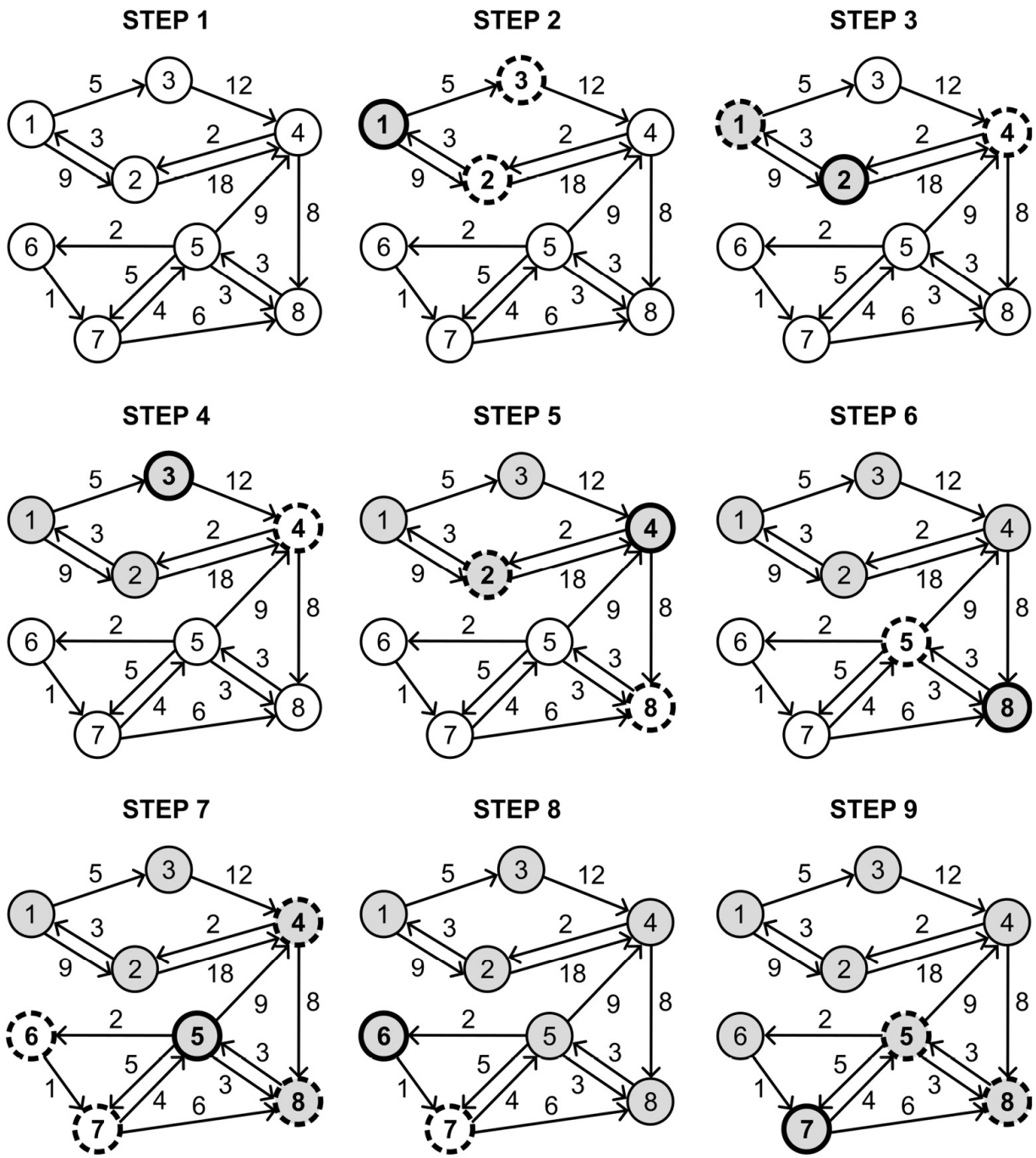

Figure 8.15 – Illustration of a BFS of a graph

The algorithm starts by visiting node **1** (**Step 2**). It has two neighbors, namely nodes **2** and **3**, which are visited next (**Step 3** and **Step 4**). As node **1** does not have more neighbors, the neighbors of its first neighbor (node **2**) are considered. As it has only one unvisited neighbor (node **4**), it is visited in the next step. According to the same method, the remaining nodes are visited in this order: **8**, **5**, **6**, **7**.

It sounds very simple, doesn't it? Let's take a look at the implementation:

```
public List<Node<T>> BFS() => BFS(Nodes[0]);
```

The **`BFS`** public method is added to the **`Graph`** class and is used to start the traversal of a graph. It calls the private **`BFS`** method, passing the first node as the parameter. Its code is as follows:

```
private List<Node<T>> BFS(Node<T> node)
{
    bool[] isVisited = new bool[Nodes.Count];
    isVisited[node.Index] = true;
    List<Node<T>> result = [];
    Queue<Node<T>> queue = [];
    queue.Enqueue(node);
    while (queue.Count > 0)
    {
        Node<T> next = queue.Dequeue();
        result.Add(next);
        foreach (Node<T> neighbor in next.Neighbors)
        {
            if (!isVisited[neighbor.Index])
            {
                isVisited[neighbor.Index] = true;
                queue.Enqueue(neighbor);
            }
        }
    }
    return result;
}
```

The important part of the code is performed by the **`isVisited`** array, which stores Boolean values indicating whether particular nodes were already visited. The array is initialized at the beginning of the **`BFS`** method, and the value of the element related to the current node is set to **`true`**, which indicates that this node was visited.

Then, the list for storing traversed nodes (**`result`**) and the queue for storing nodes that should be visited next (**`queue`**) are created. Just after the initialization of the queue, the current node is added to it.

The following operations are performed until the queue is empty: you get the first node from the queue (the **`next`** variable), add it to the collection of visited nodes, and iterate through the neighbors of the current node. For each of them, you check whether it has already been visited. If not, it is

marked as visited by setting a proper value in the `isVisited` array, and the neighbor is added to the queue for analysis in one of the next iterations of the `while` loop.

Finally, the list of the visited nodes is returned. If you want to test this algorithm, you can use the following code:

```
Graph<int> graph = new()
    { IsDirected = true, IsWeighted = true };
Node<int> n1 = graph.AddNode(1); (...)
Node<int> n8 = graph.AddNode(8);
graph.AddEdge(n1, n2, 9); (...)
graph.AddEdge(n8, n5, 3);
List<Node<int>> nodes = graph.BFS();
nodes.ForEach(Console.WriteLine);
```

The preceding code calls the `BFS` public method to traverse the graph according to the BFS algorithm. The last line is responsible for iterating through the results to present the data of the nodes in the console, as shown here:

```
Index: 0. Data: 1. Neighbors: 2.
Index: 1. Data: 2. Neighbors: 2.
Index: 2. Data: 3. Neighbors: 1.
Index: 3. Data: 4. Neighbors: 2.
Index: 7. Data: 8. Neighbors: 1.
Index: 4. Data: 5. Neighbors: 4.
Index: 5. Data: 6. Neighbors: 1.
Index: 6. Data: 7. Neighbors: 2.
```

You've just learned about two algorithms for traversing a graph, namely DFS and BFS. To make your understanding of such topics easier, this chapter contains detailed descriptions, illustrations, and examples. Now, let's proceed to another important topic, namely the minimum spanning tree, which has many real-world applications.

WHERE CAN YOU FIND MORE INFORMATION?

There are many online resources regarding traversing a graph. You can learn more about DFS at https://en.wikipedia.org/wiki/Depth-first_search, while you can find more information about the BFS algorithm and its implementation at https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/.

Minimum spanning tree

While talking about graphs, it is beneficial to introduce the subject of a **spanning tree**. What is it? **A spanning tree is a subset of edges that connects all nodes in a graph without cycles**. Of course, it is possible to have many spanning trees within the same graph. For example, let's take a look at the following diagram:

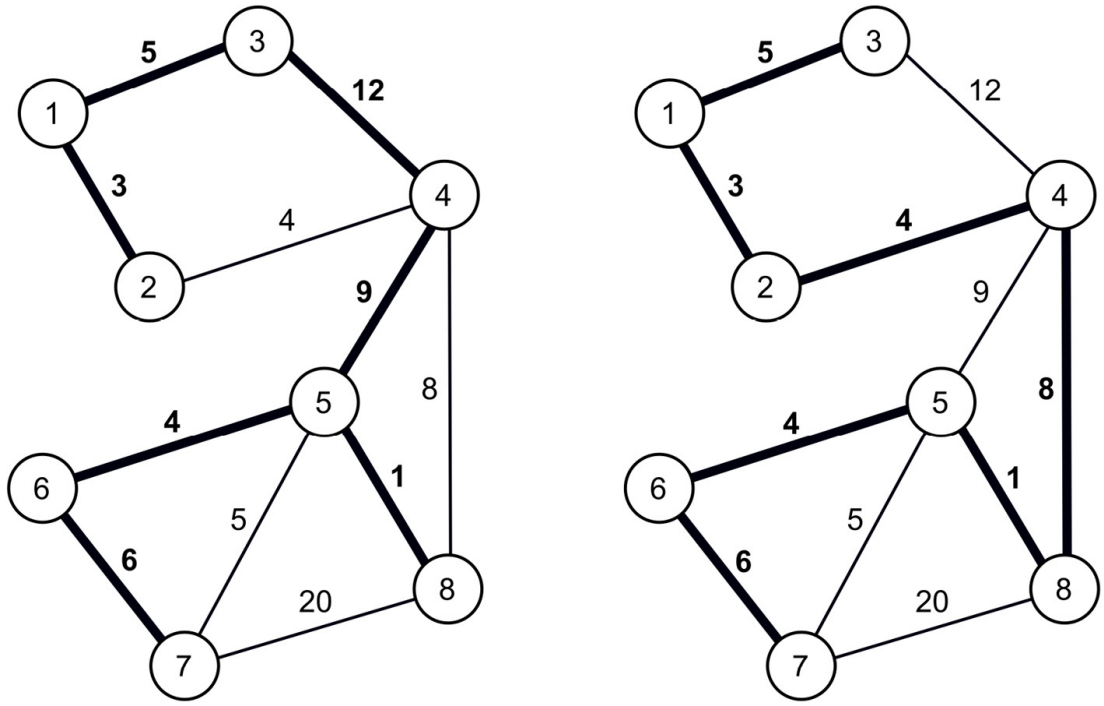

Figure 8.16 – Illustration of spanning trees within a graph

On the left-hand side is a spanning tree that consists of the following edges: (**1**, **2**), (**1**, **3**), (**3**, **4**), (**4**, **5**), (**5**, **6**), (**6**, **7**), and (**5**, **8**). The total weight is equal to 40. On the right-hand side, another spanning tree is shown. Here, the following edges are chosen: (**1**, **2**), (**1**, **3**), (**2**, **4**), (**4**, **8**), (**5**, **8**), (**5**, **6**), and (**6**, **7**). The total weight is equal to 31.

However, neither of the preceding spanning trees is the **minimum spanning tree (MST)** of this graph. What does it mean that a spanning tree is *minimum*? The answer is really simple: it is **a spanning tree with the minimum cost from all spanning trees available in the graph**. You can get the MST by replacing the edge (**6**, **7**) with (**5**, **7**). Then, the cost is equal to 30. It is also worth mentioning that the number of edges in a spanning tree is equal to the number of nodes minus one.

Why is the topic of MST so important? Let's imagine a scenario where you need to connect many buildings to a telecommunication cable. Of course, there are various possible connections, such as from one building to another, or using a hub. What's more, environmental conditions can have a serious impact on the cost of the investment due to the necessity of crossing a road or even a river. Your task is to successfully connect all buildings to the telecommunication cable at the lowest possible cost. How should you design the connections? To answer this question, you just need to create a graph, where nodes represent connectors and edges indicate possible connections. Then, you find the MST, and that's all!

DO YOU WANT SOME EXAMPLES?

The aforementioned problem of connecting many buildings to the telecommunication cable is presented in the example at the end of this section regarding the MST.

The next question is how to find the MST. There are various approaches to solving this problem, including the application of Kruskal's or Prim's algorithms. These are presented and explained in the following sections.

Kruskal's algorithm

One of the algorithms for finding the MST was discovered by **Kruskal**. Its operation is very simple to explain. **The algorithm takes an edge with the minimum weight from the remaining ones and adds it to the MST, but only if adding it does not create a cycle**. The algorithm stops when all the nodes are connected.

Let's take a look at a diagram that presents the steps of finding the MST using **Kruskal's algorithm**:

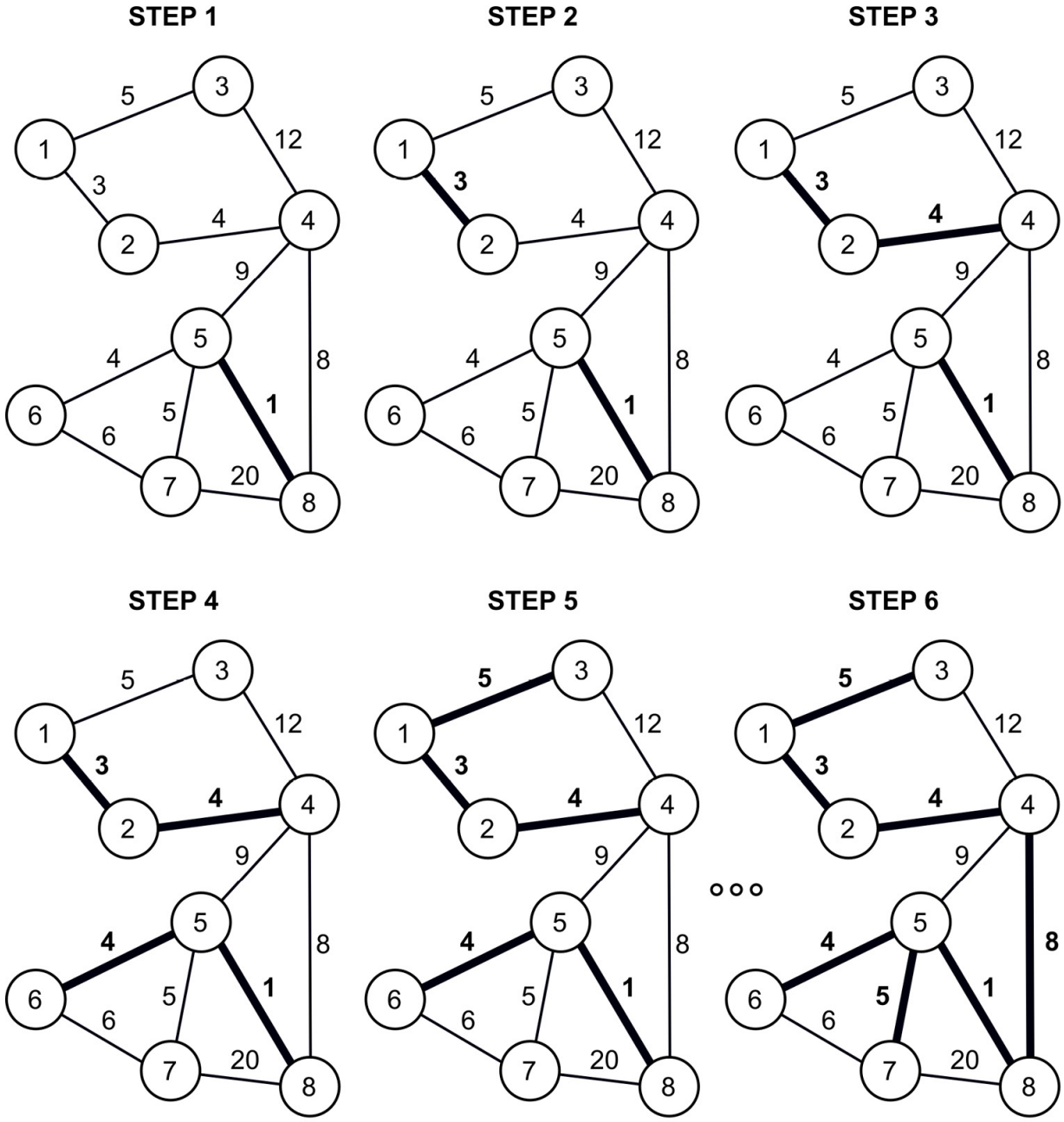

Figure 8.17 – Illustration of Kruskal's algorithm

In **Step 1**, edge (**5**, **8**) is chosen because it has the minimum weight, namely **1**. Then, the following edges are selected: (**1**, **2**) in **Step 2**, (**2**, **4**) in **Step 3**, (**5**, **6**) in **Step 4**, (**1**, **3**) in **Step 5**, as well as (**5**, **7**)

and (**4**, **8**) in **Step 6**. It is worth noting that before taking the (**4**, **8**) edge, (**6**, **7**) is considered due to its lower weight (6 instead of 8). However, adding it to the MST will introduce a cycle formed by the (**5**, **6**), (**6**, **7**), and (**5**, **7**) edges. For this reason, such an edge is ignored and the algorithm chooses (**4**, **8**). Finally, the number of edges in the MST is 7. The number of nodes is equal to 8, which means that the algorithm can stop operating and the MST has been found.

Let's take a look at its implementation. It involves the **`MSTKruskal`** method, which should be added to the **`Graph`** class. The proposed code is as follows:

```
public List<Edge<T>> MSTKruskal()
{
    List<Edge<T>> edges = GetEdges();
    edges.Sort((a, b) => a.Weight.CompareTo(b.Weight));
    Queue<Edge<T>> queue = new(edges);
    Subset<T>[] subsets = new Subset<T>[Nodes.Count];
    for (int i = 0; i < Nodes.Count; i++)
    {
        subsets[i] = new() { Parent = Nodes[i] };
    }
    List<Edge<T>> result = [];
    while (result.Count < Nodes.Count - 1)
    {
        Edge<T> edge = queue.Dequeue();
        Node<T> from = GetRoot(subsets, edge.From);
        Node<T> to = GetRoot(subsets, edge.To);
        if (from == to) { continue; }
        result.Add(edge);
        Union(subsets, from, to);
    }
    return result;
}
```

This method does not take any parameters. To start, a list of edges is obtained by calling the **`GetEdges`** method. Then, the edges are sorted in ascending order by weight. Such a step is crucial because you need to get an edge with the minimum cost in the following iterations of the algorithm. In the next line, a new queue is created and **`Edge`** instances are enqueued, using the constructor of the **`Queue`** class.

In the next block of code, an array with data of subsets is created. By default, each node is added to a separate subset. This is the reason why the number of elements in the **`subsets`** array is equal to the number of nodes. The subsets are used to check whether an addition of an edge to the MST causes the creation of a cycle.

Then, the list for storing edges from the MST is created (**`result`**). The most interesting part of the code is the **`while`** loop, which iterates until the correct number of edges is found in the MST. Within this loop, you get the edge with the minimum weight, just by calling the **`Dequeue`** method on the **`Queue`** instance. Then, you can check whether no cycles were introduced by adding the found edge to the MST. In such a case, the edge is added to the target list, and the **`Union`** method is called to union two subsets.

While analyzing the previous method, the **GetRoot** one is mentioned. It aims to update parents for subsets, as well as return the root node of the subset, as follows:

```
private Node<T> GetRoot(Subset<T>[] subsets, Node<T> node)
{
    int i = node.Index;
    ss[i].Parent = ss[i].Parent != node
        ? GetRoot(ss, ss[i].Parent) : ss[i].Parent;
    return ss[i].Parent;
}
```

The last private method is named **Union** and performs the *union* operation (by a rank) of two sets. It takes three parameters, namely an array of **Subset** instances and two **Node** instances, representing root nodes for subsets on which the *union* operation should be performed. The suitable part of the code is as follows:

```
private void Union(Subset<T>[] ss, Node<T> a, Node<T> b)
{
    ss[b.Index].Parent =
        ss[a.Index].Rank >= ss[b.Index].Rank
            ? a : ss[b.Index].Parent;
    ss[a.Index].Parent =
        ss[a.Index].Rank < ss[b.Index].Rank
            ? b : ss[a.Index].Parent;
    if (ss[a.Index].Rank == ss[b.Index].Rank)
    {
        ss[a.Index].Rank++;
    }
}
```

In the previous code snippets, you can see the **Subset** class, but what does it look like? Let's take a look at its declaration:

```
public class Subset<T>
{
    public required Node<T> Parent { get; set; }
    public int Rank { get; set; }
    public override string ToString() => $"Rank: {Rank}.
        Parent: {Parent.Data}. Index: {Parent.Index}.";
}
```

The class contains properties representing the parent node (**Parent**), as well as the rank of the subset (**Rank**). The class also contains the overridden **ToString** method, which presents some basic information about the subset in textual form.

WHERE CAN YOU FIND MORE INFORMATION?

*Did you know that the presented approach is representative of a **greedy algorithm**? The code shown here is based on the implementation available at https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/. You can find there a lot of interesting information about Kruskal's algorithm, as well as about many other algorithms regarding graphs, such as about a simple approach to coloring, which is also one of the topics waiting for you in the current chapter. GeeksForGeeks is a great resource for various algorithms with a huge collection of content, and it's something I highly recommend!*

Let's take a look at the usage of the **`MSTKruskal`** method:

```
Graph<int> graph = new()
    { IsDirected = false, IsWeighted = true };
Node<int> n1 = graph.AddNode(1);
Node<int> n2 = graph.AddNode(2);
Node<int> n3 = graph.AddNode(3);
Node<int> n4 = graph.AddNode(4);
Node<int> n5 = graph.AddNode(5);
Node<int> n6 = graph.AddNode(6);
Node<int> n7 = graph.AddNode(7);
Node<int> n8 = graph.AddNode(8);
graph.AddEdge(n1, n2, 3);
graph.AddEdge(n1, n3, 5);
graph.AddEdge(n2, n4, 4);
graph.AddEdge(n3, n4, 12);
graph.AddEdge(n4, n5, 9);
graph.AddEdge(n4, n8, 8);
graph.AddEdge(n5, n6, 4);
graph.AddEdge(n5, n7, 5);
graph.AddEdge(n5, n8, 1);
graph.AddEdge(n6, n7, 6);
graph.AddEdge(n7, n8, 20);
List<Edge<int>> edges = graph.MSTKruskal();
edges.ForEach(Console.WriteLine);
```

First, you initialize an undirected and weighted graph, as well as add nodes and edges. Then, you call the **`MSTKruskal`** method to find the MST using Kruskal's algorithm. Finally, you use the **`ForEach`** method to write the data of each edge from the MST in the console. The exemplary output is shown here:

```
8 -> 5. Weight: 1.
1 -> 2. Weight: 3.
2 -> 4. Weight: 4.
5 -> 6. Weight: 4.
1 -> 3. Weight: 5.
7 -> 5. Weight: 5.
8 -> 4. Weight: 8.
```

As mentioned previously, you will learn about two algorithms for finding the MST in this chapter. Now, it is high time to take a look at the second one, namely Prim's algorithm.

Prim's algorithm

Another solution to solve the problem of finding the MST is **Prim's algorithm**. **It uses two sets of nodes that are disjointed, namely the nodes located in the MST and the nodes that are not placed there yet**. In the following iterations, the algorithm **finds an edge with the minimum weight that connects a node from the first group with a node from the second group. The node of the edge, which is not already in the MST, is added to this set.**

The preceding description sounds quite simple, doesn’t it? Let’s see it in action by analyzing the diagram presenting the steps of finding the MST using Prim’s algorithm:

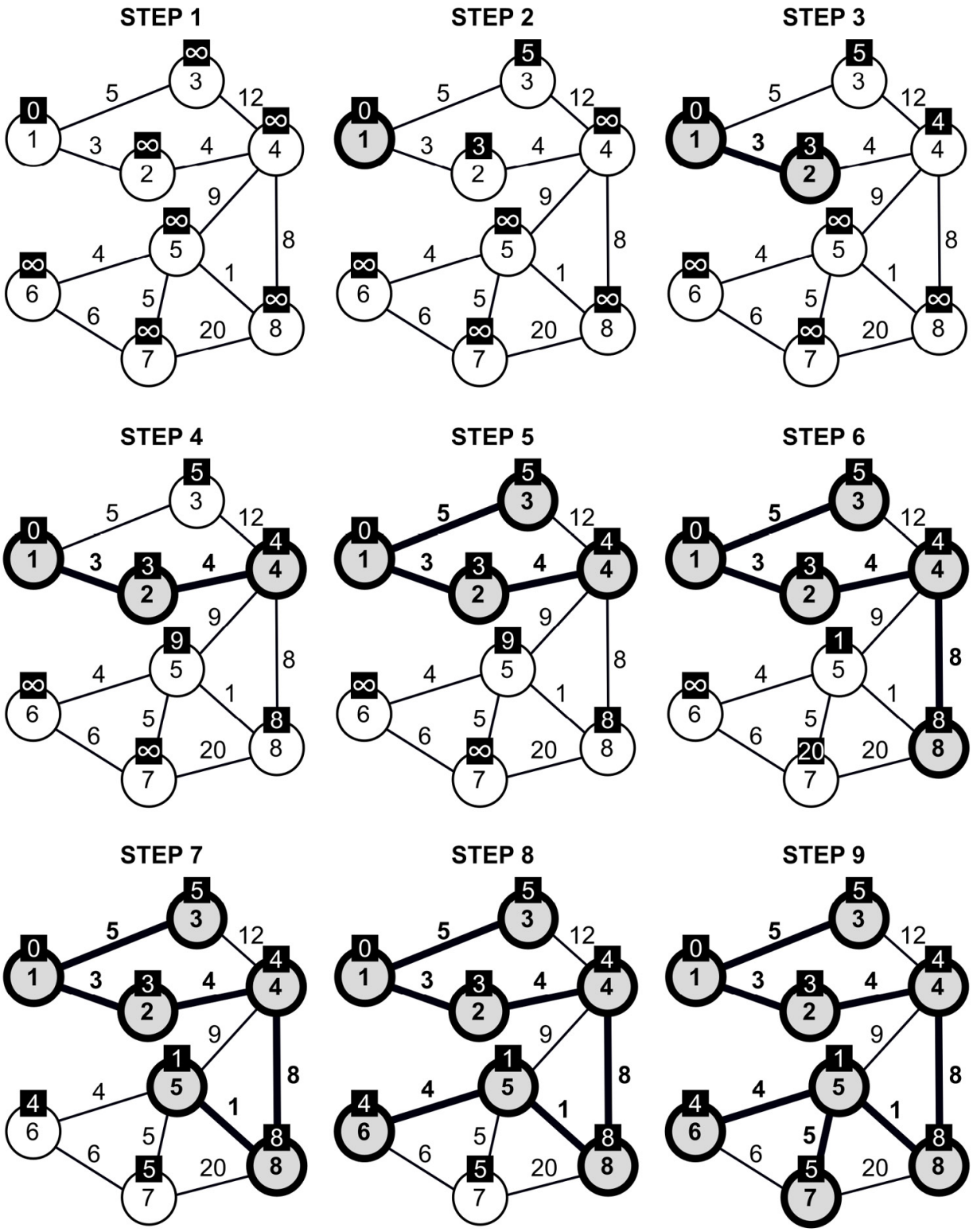

Figure 8.18 – Illustration of Prim's algorithm

Let's take a look at the additional indicators that have been added next to the nodes in the graph. They present the minimum weight necessary to reach such a node from any of its neighbors. By default, the starting node has such a value set to **0**, while all others are set to infinity, as presented in **Step 1**.

In **Step 2**, the starting node is added to the subset of nodes forming the MST, and the distance to its neighbors is updated, namely **5** for reaching node **3** and **3** for reaching node **2**. The values of the other nodes are still set to infinity.

In **Step 3**, the node with the minimum cost is chosen. In this case, node **2** is selected because the cost is equal to **3**. Its competitor (namely node **3**) has a cost equal to **5**. Next, you need to update the cost of reaching the neighbors of the current node, namely node **4** with the cost set to **4**.

The next chosen node is node **4** because it does not exist in the MST set and has the lowest reaching cost (**Step 4**). In the same way, you choose the next edges in the following order: (**1**, **3**) in **Step 5**, (**4**, **8**) in **Step 6**, (**8**, **5**) in **Step 7**, (**5**, **6**) in **Step 8**, and (**5**, **7**) in **Step 9**. Now, all the nodes are included in the MST and the algorithm can stop its operation.

Given this detailed description of the steps of the algorithm, let's proceed to the C#-based implementation. The majority of operations are performed in the `MSTPrim` method, which should be added to the `Graph` class:

```
public List<Edge<T>> MSTPrim()
{
    int[] previous = new int[Nodes.Count];
    previous[0] = -1;
    int[] minWeight = new int[Nodes.Count];
    Array.Fill(minWeight, int.MaxValue);
    minWeight[0] = 0;
    bool[] isInMST = new bool[Nodes.Count];
    Array.Fill(isInMST, false);
    for (int i = 0; i < Nodes.Count - 1; i++)
    {
        int mwi = GetMinWeightIndex(minWeight, isInMST);
        isInMST[mwi] = true;
        for (int j = 0; j < Nodes.Count; j++)
        {
            Edge<T>? edge = this[mwi, j];
            int weight = edge != null ? edge.Weight : -1;
            if (edge != null
                && !isInMST[j]
                && weight < minWeight[j])
            {
                previous[j] = mwi;
                minWeight[j] = weight;
            }
        }
    }
    List<Edge<T>> result = [];
    for (int i = 1; i < Nodes.Count; i++)
    {
        result.Add(this[previous[i], i]!);
```

```
    }
    return result;
}
```

The **MSTPrim** method does not take any parameters. It uses three auxiliary node-related arrays that assign additional data to the nodes of the graph:

- The first, namely **previous**, stores indices of the previous node, from which the given node can be reached. By default, the values of all elements are equal to **0**, except the first one, which is set to **-1**.
- The **minWeight** array stores the minimum weight of the edge for accessing the given node. By default, all elements are set to the maximum value of the **int** type, while the value for the first element is set to **0**.
- The **isInMST** array indicates whether the given node is already in the MST. To start with, the values of all the elements should be set to **false**.

The most interesting part of the code is located in the **for** loop. Within it, you'll find the index of the node from the set of nodes not located in the MST, which can be reached with the minimum cost. Such a task is performed by the **GetMinWeightIndex** method. Then, another **for** loop is used. Within it, you get an edge that connects nodes with the **mwi** index (this stands for *minimum weight index*) and **j**. You check whether the node is not already located in the MST and whether the cost of reaching the node is smaller than the previous minimum cost. If so, values of node-related elements in the **previous** and **minWeight** arrays are updated.

The remaining part of the code just prepares the final results. Here, you create a new instance of the list with the data of edges that form the MST. The **for** loop is used to get the data of the following edges and to add them to the **result** list.

While analyzing the code, the **GetMinWeightIndex** private method is mentioned. Its code is presented in the following block:

```
private int GetMinWeightIndex(
    int[] weights, bool[] isInMST)
{
    int minValue = int.MaxValue;
    int minIndex = 0;
    for (int i = 0; i < Nodes.Count; i++)
    {
        if (!isInMST[i] && weights[i] < minValue)
        {
            minValue = weights[i];
            minIndex = i;
        }
    }
    return minIndex;
}
```

The **GetMinWeightIndex** method just finds an index of the node, which is not located in the MST and can be reached with the minimum cost. To do so, you use a **for** loop to iterate through all the nodes. For each of them, you check whether the current node is not located in the MST and whether the cost

of reaching it is smaller than the already-stored minimum value. If so, the values of the **`minValue`** and **`minIndex`** variables are updated. Finally, the index is returned.

WHERE CAN YOU FIND MORE INFORMATION?

*Similar to Kruskal's algorithm, Prim's variant is also representative of a **greedy algorithm**. I strongly encourage you to search for even more interesting information about this algorithm in books, research papers, and on the internet. It is worth noting that the presented code is based on the implementation shown at https://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/.*

Let's take a look at the usage of the **`MSTPrim`** method:

```
Graph<int> graph = new()
    { IsDirected = false, IsWeighted = true };
Node<int> n1 = graph.AddNode(1); (...)
Node<int> n8 = graph.AddNode(8);
graph.AddEdge(n1, n2, 3); (...)
graph.AddEdge(n7, n8, 20);
List<Edge<int>> edges = graph.MSTPrim();
edges.ForEach(Console.WriteLine);
```

The missing parts of the code are the same as in the case of the exemplary code regarding Kruskal's algorithm. When you run the code, you will get the following result:

```
1 -> 2. Weight: 3.
1 -> 3. Weight: 5.
2 -> 4. Weight: 4.
8 -> 5. Weight: 1.
5 -> 6. Weight: 4.
5 -> 7. Weight: 5.
4 -> 8. Weight: 8.
```

Now that we've looked at various algorithms for finding the MST, let's proceed to an example.

Example – telecommunication cable

As mentioned in the introduction to the topic of MSTs, this problem has some important real-world applications, such as creating a plan of connections between buildings to supply all of them with a telecommunication cable with the smallest cost. Of course, there are various possible connections, such as from one building to another or using a hub. What's more, environmental conditions can have a serious impact on the cost of the investment due to the necessity of crossing a road or even a river.

For example, let's create a program that solves this problem in the context of a set of buildings, as shown in the following figure:

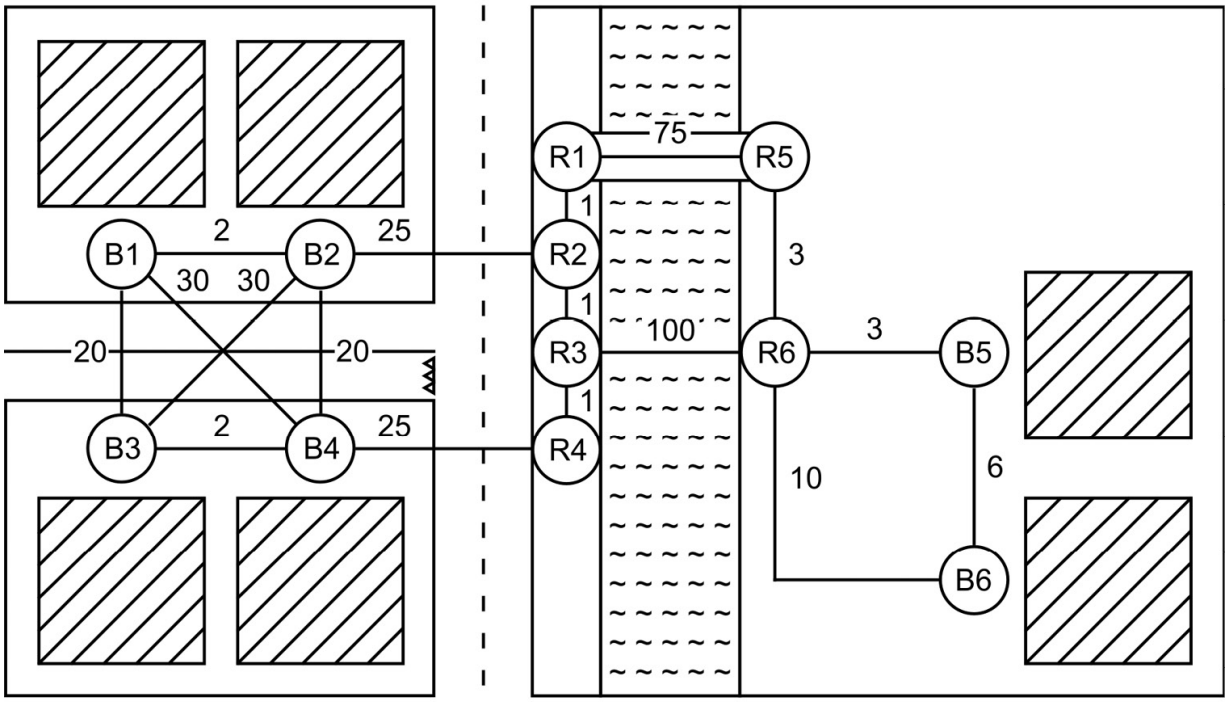

Figure 8.19 – Illustration of the telecommunication cable example

As you can see, the estate community consists of six buildings. The buildings are located on two sides of a small river with only one bridge. Moreover, two roads exist. Of course, there are different costs of connections between various points, depending both on the distance and the environmental conditions. For example, the direct connection between two buildings (**B1** and **B2**) has a cost equal to **2**, while using the bridge (between **R1** and **R5**) involves a cost equal to **75**. If you need to cross the river without a bridge (between **R3** and **R6**), the cost is even higher and equal to **100**.

Your task is to find the MST. Within this example, you will apply both Kruskal's and Prim's algorithms to solve this problem. To start, let's initialize the undirected and weighted graph, as well as add nodes and edges, as follows:

```
Graph<string> graph = new()
    { IsDirected = false, IsWeighted = true };
Node<string> nodeB1 = graph.AddNode("B1");
Node<string> nodeB2 = graph.AddNode("B2");
Node<string> nodeB3 = graph.AddNode("B3");
Node<string> nodeB4 = graph.AddNode("B4");
Node<string> nodeB5 = graph.AddNode("B5");
Node<string> nodeB6 = graph.AddNode("B6");
Node<string> nodeR1 = graph.AddNode("R1");
Node<string> nodeR2 = graph.AddNode("R2");
Node<string> nodeR3 = graph.AddNode("R3");
Node<string> nodeR4 = graph.AddNode("R4");
Node<string> nodeR5 = graph.AddNode("R5");
Node<string> nodeR6 = graph.AddNode("R6");
graph.AddEdge(nodeB1, nodeB2, 2);
```

```
graph.AddEdge(nodeB1, nodeB3, 20);
graph.AddEdge(nodeB1, nodeB4, 30);
graph.AddEdge(nodeB2, nodeB3, 30);
graph.AddEdge(nodeB2, nodeB4, 20);
graph.AddEdge(nodeB2, nodeR2, 25);
graph.AddEdge(nodeB3, nodeB4, 2);
graph.AddEdge(nodeB4, nodeR4, 25);
graph.AddEdge(nodeR1, nodeR2, 1);
graph.AddEdge(nodeR2, nodeR3, 1);
graph.AddEdge(nodeR3, nodeR4, 1);
graph.AddEdge(nodeR1, nodeR5, 75);
graph.AddEdge(nodeR3, nodeR6, 100);
graph.AddEdge(nodeR5, nodeR6, 3);
graph.AddEdge(nodeR6, nodeB5, 3);
graph.AddEdge(nodeR6, nodeB6, 10);
graph.AddEdge(nodeB5, nodeB6, 6);
```

Now, you just need to call the **`MSTKruskal`** method to use Kruskal's algorithm to find the MST. When the results are obtained, you can easily present them in the console, together with the total cost. The suitable part of the code is shown in the following block:

```
Console.WriteLine("Minimum Spanning Tree - Kruskal:");
List<Edge<string>> kruskal = graph.MSTKruskal();
kruskal.ForEach(Console.WriteLine);
Console.WriteLine("Cost: " + kruskal.Sum(e => e.Weight));
```

The results presented in the console are shown here:

```
Minimum Spanning Tree - Kruskal:
R4 -> R3. Weight: 1.
R3 -> R2. Weight: 1.
R2 -> R1. Weight: 1.
B1 -> B2. Weight: 2.
B3 -> B4. Weight: 2.
R6 -> R5. Weight: 3.
R6 -> B5. Weight: 3.
B6 -> B5. Weight: 6.
B1 -> B3. Weight: 20.
R2 -> B2. Weight: 25.
R1 -> R5. Weight: 75.
Cost: 139
```

If you visualize such results on the map, you'll find the following MST:

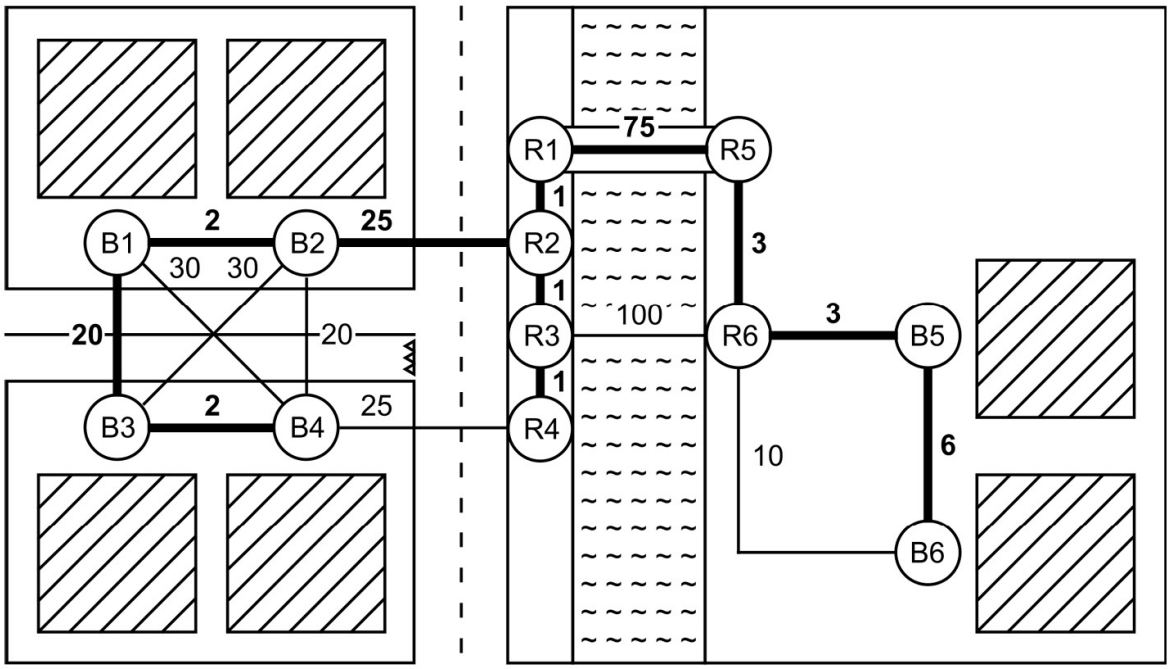

Figure 8.20 – Illustration of the result of the telecommunication cable example

Similarly, you can apply Prim's algorithm:

```
Console.WriteLine("\nMinimum Spanning Tree - Prim:");
List<Edge<string>> prim = graph.MSTPrim();
prim.ForEach(Console.WriteLine);
Console.WriteLine("Cost: " + prim.Sum(e => e.Weight));
```

The results are as follows:

```
Minimum Spanning Tree - Prim:
B1 -> B2. Weight: 2.
B1 -> B3. Weight: 20.
B3 -> B4. Weight: 2.
R6 -> B5. Weight: 3.
B5 -> B6. Weight: 6.
R2 -> R1. Weight: 1.
B2 -> R2. Weight: 25.
R2 -> R3. Weight: 1.
R3 -> R4. Weight: 1.
R1 -> R5. Weight: 75.
R5 -> R6. Weight: 3.
Cost: 139
```

You've just completed an example related to the real-world application of MSTs. Are you ready to proceed to another graph-related subject known as coloring?

Coloring

The topic of finding the MST is not the only graph-related problem. Among others, **node coloring** exists. It aims to **assign colors (numbers) to all nodes to comply with the rule that there cannot be an edge between two nodes with the same color**. Of course, the number of colors should be as low as possible. Such a problem has some real-world applications, such as for coloring a map. The implementation of the coloring algorithm, which is shown in this chapter, is quite simple and in some cases could use more colors than is necessary.

FOUR-COLOR THEOREM

Did you know that the nodes of each planar graph can be colored with no more than four colors? If you are interested in this topic, take a look at the ***four-color theorem*** *(http://mathworld.wolfram.com/Four-ColorTheorem.html). Since I am talking about a planar graph, you should understand that it is a graph whose edges do not cross each other while it is drawn on the plane.*

Let's take a look at the following diagram:

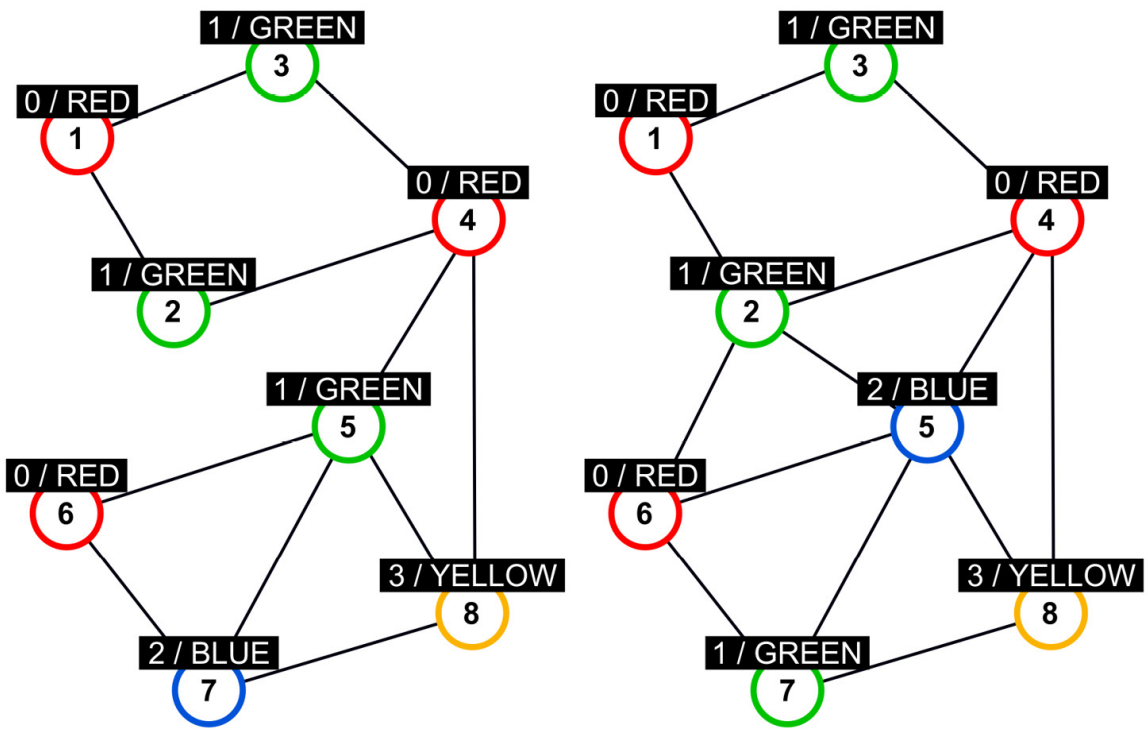

Figure 8.21 – Illustration of graph coloring

The left-hand side illustration presents a graph that is colored using four colors: red (index equal to **0**), green (**1**), blue (**2**), and yellow (**3**). As you can see, there are no nodes with the same colors connected by an edge. The graph shown on the right depicts the graph with two additional edges, namely (**2**, **6**) and (**2**, **5**). In such a case, the coloring has changed, but the number of colors remains the same.

The question is, how can you find colors for nodes to comply with the aforementioned rule? Fortunately, the algorithm is very simple and its implementation is presented here. Here is the code of the **`Color`** method, which should be added to the **`Graph`** class:

```
public int[] Color()
{
    int[] colors = new int[Nodes.Count];
    Array.Fill(colors, -1);
    colors[0] = 0;
    bool[] available = new bool[Nodes.Count];
    for (int i = 1; i < Nodes.Count; i++)
    {
        Array.Fill(available, true);
        foreach (Node<T> neighbor in Nodes[i].Neighbors)
        {
            int ci = colors[neighbor.Index];
            if (ci >= 0) { available[ci] = false; }
        }
        colors[i] = Array.IndexOf(available, true);
    }
    return colors;
}
```

The **`Color`** method uses two auxiliary node-related arrays. The first is named **`colors`** and stores indices of colors chosen for particular nodes. By default, the values of all elements are set to **`-1`**, except the first one, which is set to **`0`**. This means that the color of the first node is automatically set to the first color (for example, red). The other auxiliary array (**`available`**) stores information about the availability of particular colors.

The most crucial part of the code is the **`for`** loop. Within it, you reset the availability of colors by setting **`true`** as the value of all elements within the **`available`** array. Then, you iterate through the neighbor nodes of the current node to read their colors and mark such colors as unavailable by setting **`false`** as a value of a particular element in the **`available`** array. Then, you find the first available color for the current node and use it.

Let's take a look at the usage of the **`Color`** method:

```
Graph<int> graph = new()
    { IsDirected = false, IsWeighted = false };
Node<int> n1 = graph.AddNode(1);
Node<int> n2 = graph.AddNode(2);
Node<int> n3 = graph.AddNode(3);
Node<int> n4 = graph.AddNode(4);
Node<int> n5 = graph.AddNode(5);
Node<int> n6 = graph.AddNode(6);
Node<int> n7 = graph.AddNode(7);
Node<int> n8 = graph.AddNode(8);
graph.AddEdge(n1, n2);
graph.AddEdge(n1, n3);
graph.AddEdge(n2, n4);
graph.AddEdge(n3, n4);
graph.AddEdge(n4, n5);
graph.AddEdge(n4, n8);
```

```
graph.AddEdge(n5, n6);
graph.AddEdge(n5, n7);
graph.AddEdge(n5, n8);
graph.AddEdge(n6, n7);
graph.AddEdge(n7, n8);
int[] colors = graph.Color();
for (int i = 0; i < colors.Length; i++)
{
    Console.WriteLine(
        $"Node {graph.Nodes[i].Data}: {colors[i]}");
}
```

Here, you create a new undirected and unweighted graph, the same as shown in the preceding figure, on the left. Then, you add nodes and edges, as well as call the `Color` method to perform the node coloring. As a result, you receive an array with indices of colors for particular nodes. Then, you present the results in the console:

```
Node 1: 0
Node 2: 1
Node 3: 1
Node 4: 0
Node 5: 1
Node 6: 0
Node 7: 2
Node 8: 3
```

With this short introduction, you are ready to proceed to the real-world application, namely for coloring the voivodeship map.

Example – voivodeship map

Let's create a program that represents the map of voivodeships in Poland as a graph, and color such areas so that two voivodeships with common borders aren't the same color. Of course, you should try to limit the number of colors.

To start, let's think about the graph's representation. Here, nodes represent particular voivodeships, while edges represent common borders between voivodeships.

The map of Poland with the graph already colored is shown in the following diagram:

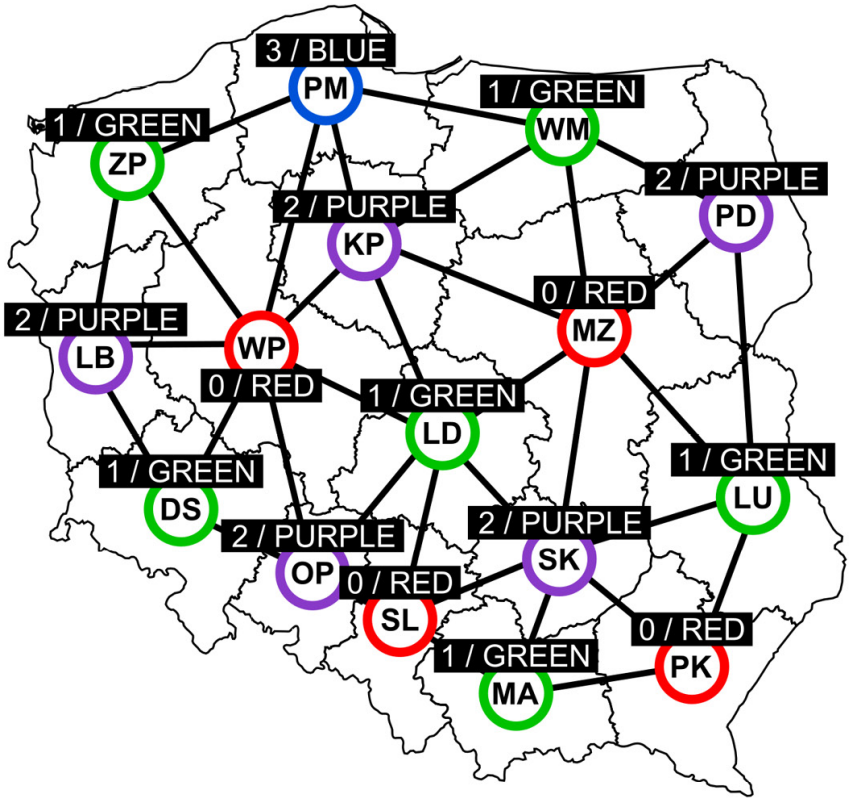

Figure 8.22 – Illustration of the voivodeship map example

Your task is just to color nodes in the graph using the previously described algorithm. To do so, you create an undirected and unweighted graph, add nodes representing voivodeships, and add edges to indicate common borders. The code is as follows:

```
Graph<string> graph = new()
    { IsDirected = false, IsWeighted = false };
List<string> borders =
[
    "PK:LU|SK|MA",
    "LU:PK|SK|MZ|PD",
    "SK:PK|MA|SL|LD|MZ|LU",
    "MA:PK|SK|SL",
    "SL:MA|SK|LD|OP",
    "LD:SL|SK|MZ|KP|WP|OP",
    "WP:LD|KP|PM|ZP|LB|DS|OP",
    "OP:SL|LD|WP|DS",
    "MZ:LU|SK|LD|KP|WM|PD",
    "PD:LU|MZ|WM",
    "WM:PD|MZ|KP|PM",
    "KP:MZ|LD|WP|PM|WM",
    "PM:WM|KP|WP|ZP",
    "ZP:PM|WP|LB",
    "LB:ZP|WP|DS",
    "DS:LB|WP|OP"
];
Dictionary<string, Node<string>> nodes = [];
```

```
foreach (string border in borders)
{
    string[] parts = border.Split(':');
    string name = parts[0];
    nodes[name] = graph.AddNode(name);
}
foreach (string border in borders)
{
    string[] parts = border.Split(':');
    string name = parts[0];
    string[] vicinities = parts[1].Split('|');
    foreach (string vicinity in vicinities)
    {
        Node<string> from = nodes[name];
        Node<string> to = nodes[vicinity];
        if (!from.Neighbors.Contains(to))
        {
            graph.AddEdge(from, to);
        }
    }
}
```

Then, the **`Color`** method is called on the **`Graph`** instance and the color indices for particular nodes are returned. Finally, you present the results in the console. The suitable part of the code is as follows:

```
int[] colors = graph.Color();
for (int i = 0; i < colors.Length; i++)
{
    Console.WriteLine(
        $"{graph.Nodes[i].Data}: {colors[i]}");
}
```

The results are as follows:

```
PK: 0
LU: 1
SK: 2
MA: 1
SL: 0
LD: 1
WP: 0
OP: 2
MZ: 0
PD: 2
WM: 1
KP: 2
PM: 3
ZP: 1
LB: 2
DS: 1
```

You just learned how to color nodes in a graph! However, this is not the end of the interesting topics regarding graphs that will be presented in this book. Next, we'll search for the shortest path in the graph.

Shortest path

A graph is a great data structure for storing data of various maps, such as cities and the distances between them. For this reason, one of the obvious real-world applications of graphs is **searching for the shortest path between two nodes, which takes into account a specific cost**, such as the distance, the necessary time, or even the amount of fuel required.

There are several approaches to the topic of searching for the shortest path in a graph. However, one of the common solutions is **Dijkstra's algorithm**, which makes it possible to **calculate the distance from a starting node to all nodes located in the graph**. Then, you can easily get not only the cost of the connection between two nodes but also find nodes that are between the start and end nodes.

Dijkstra's algorithm uses two auxiliary node-related arrays:

- One for storing an identifier of the previous node, which is the node from which the current node can be reached with the smallest overall cost
- One for storing the minimum distance (cost), which is necessary for accessing the current node

What's more, it uses the queue for storing nodes that should be checked. **During the consecutive iterations, the algorithm updates the minimum distances to particular nodes in the graph**. Finally, the auxiliary arrays contain the minimum distance (cost) to reach all the nodes from the chosen starting node, as well as information on how to reach each node using the shortest path.

WHERE CAN YOU FIND MORE INFORMATION?

You can find a lot of content about Dijkstra's algorithm on the internet. Just search for its name and you will see a huge number of results. As an example, you can find useful content related to the implementation presented in this chapter at https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

Let's take a look at the following diagram, which presents two various shortest paths that have been found using Dijkstra's algorithm. The left-hand side shows the path from node **8** to **1**, while the right-hand side shows the path from node **1** to **7**:

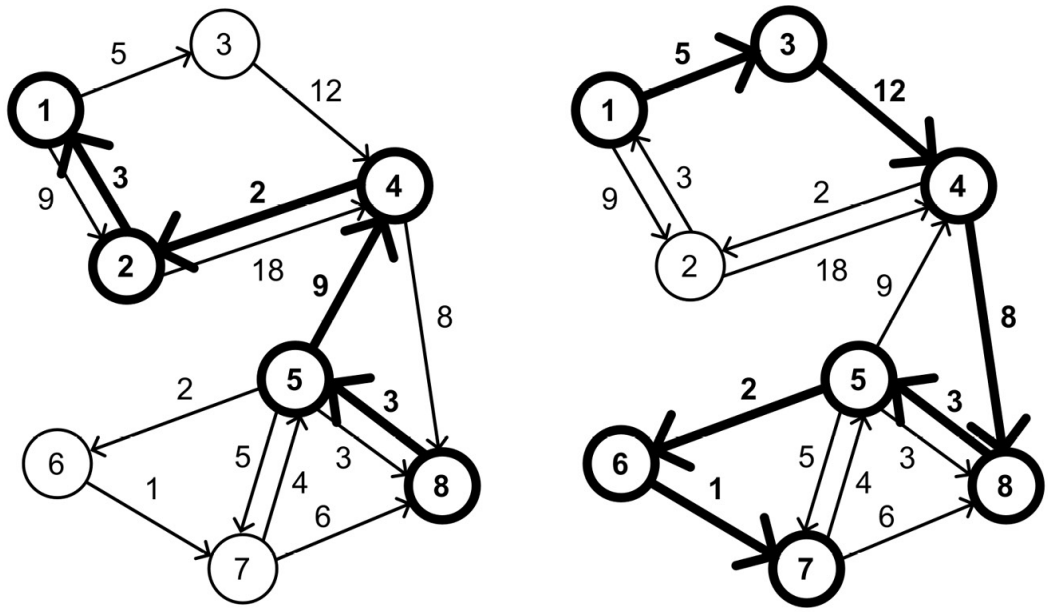

Figure 8.23 – Illustration of the shortest paths in a graph

It is high time that you see some C# code that can be used to implement Dijkstra's algorithm. The main role is performed by the **`GetShortestPath`** method, which should be added to the **`Graph`** class. The code is as follows:

```
using Priority_Queue; (...)
public List<Edge<T>> GetShortestPath(
    Node<T> source, Node<T> target)
{
    int[] previous = new int[Nodes.Count];
    Array.Fill(previous, -1);
    int[] distances = new int[Nodes.Count];
    Array.Fill(distances, int.MaxValue);
    distances[source.Index] = 0;
    SimplePriorityQueue<Node<T>> nodes = new();
    for (int i = 0; i < Nodes.Count; i++)
    {
        nodes.Enqueue(Nodes[i], distances[i]);
    }
    while (nodes.Count != 0)
    {
        Node<T> node = nodes.Dequeue();
        for (int i = 0; i < node.Neighbors.Count; i++)
        {
            Node<T> neighbor = node.Neighbors[i];
            int weight = i < node.Weights.Count
                ? node.Weights[i] : 0;
            int wTotal = distances[node.Index] + weight;
            if (distances[neighbor.Index] > wTotal)
            {
                distances[neighbor.Index] = wTotal;
                previous[neighbor.Index] = node.Index;
                nodes.UpdatePriority(neighbor,
                    distances[neighbor.Index]);
```

```
            }
        }
    }
    List<int> indices = [];
    int index = target.Index;
    while (index >= 0)
    {
        indices.Add(index);
        index = previous[index];
    }
    indices.Reverse();
    List<Edge<T>> result = [];
    for (int i = 0; i < indices.Count - 1; i++)
    {
        Edge<T> edge = this[indices[i], indices[i + 1]]!;
        result.Add(edge);
    }
    return result;
}
```

The **`GetShortestPath`** method takes two parameters, namely **`source`** and **`target`** nodes. First, it creates two node-related auxiliary arrays for storing the indices of previous nodes, from which the given node can be reached with the smallest overall cost (**`previous`**), as well as for storing the current minimum distances to the given node (**`distances`**). By default, the values of all elements in the **`previous`** array are set to **`-1`**, while in the **`distances`** array, they are set to the maximum value of the **`int`** type. Of course, the distance to the source node is set to **`0`**.

Then, you create a new priority queue and enqueue the data of all nodes. The priority of each element is equal to the current distance to such a node. Here, you use the same implementation of a priority queue, as presented in *Chapter 5*, *Stacks and Queues*, namely from the **`OptimizedPriorityQueue`** NuGet package.

The most interesting part of the code is the **`while`** loop, which is executed until the queue is empty. Within this **`while`** loop, you get the first node from the queue and iterate through all of its neighbors using a **`for`** loop. Inside such a loop, you calculate the distance to a neighbor by taking the sum of the distance to the current node and the weight of the edge. If the calculated distance is smaller than the currently stored value, you update the values regarding the minimum distance to the given neighbor, as well as the index of the previous node, from which you can reach the neighbor. It's worth noting that the priority of the element in the queue should be updated as well.

The remaining operations are used to resolve the path using the values stored in the **`previous`** array. To do so, you save the indices of the following nodes in the **`indices`** list. Then, you reverse it to achieve the order from the source node to the target one. Finally, you create the list of edges to present the result in a form that's suitable for returning from the method.

Let's take a look at the usage of the **`GetShortestPath`** method:

```
Graph<int> graph = new()
```

```
    { IsDirected = true, IsWeighted = true };
Node<int> n1 = graph.AddNode(1); (...)
Node<int> n8 = graph.AddNode(8);
graph.AddEdge(n1, n2, 9); (...)
graph.AddEdge(n8, n5, 3);
List<Edge<int>> path = graph.GetShortestPath(n1, n5);
path.ForEach(Console.WriteLine);
```

Here, you create a new directed and weighted graph, as well as add nodes and edges. The missing parts of the code are the same as in the case of the *directed and weighted edges* example. Then, you call the `GetShortestPath` method to search for the shortest path between nodes `1` and `5`. As a result, you receive a list of edges forming the shortest path. Then, you just iterate through all the edges and present the results in the console:

```
1 -> 3. Weight: 5.
3 -> 4. Weight: 12.
4 -> 8. Weight: 8.
8 -> 5. Weight: 3.
```

With this short introduction, together with a simple example, let's proceed to a more advanced and interesting application related to game development.

Example – path in game

The last example we'll cover in this chapter involves the application of Dijkstra's algorithm to find the shortest path in a game map. Let's imagine that you have a board with various obstacles. For this reason, the player can use only part of the board to move. Your task is to find the shortest path between two places located on the board.

To start, let's represent the board as a jagged array. The suitable part of the code is shown here:

```
using System.Text;
string[] lines = new string[]
{
    "00111000001111100000011111",
    "00111000001111100000011111",
    "00111000001111100000011111",
    "00000000000111000000011111",
    "00000011100000000000011111",
    "00010011100111000000011111",
    "11111111111111101111111100",
    "11111111111111101111111101",
    "11111111111111101111111100",
    "00000000000000001111111110",
    "00000000000000001111111100",
    "00011111110011000000001101",
    "00011111110011000000001100",
    "00011000000000001111111110",
    "11111000000000001111111100",
    "11111000110011001000100001",
    "11111000110011000010001000"
};
bool[][] map = new bool[lines.Length][];
```

```
for (int i = 0; i < lines.Length; i++)
{
    map[i] = lines[i]
        .Select(c => int.Parse(c.ToString()) == 0)
        .ToArray();
}
```

To improve the readability of code, the map is represented as an array of `string` values. Each row is presented as text, with the number of characters equal to the number of columns. The value of each character indicates the availability of the point. If it is equal to `0`, the position is available. Otherwise, it is not. The `string`-based map representation should then be converted into the Boolean jagged array. Such a task is performed by a few lines of code, as shown in the preceding snippet.

The next step is to create the graph, as well as add the necessary nodes and edges. The suitable part of the code is as follows:

```
Graph<string> graph = new()
    { IsDirected = false, IsWeighted = true };
for (int i = 0; i < map.Length; i++)
{
    for (int j = 0; j < map[i].Length; j++)
    {
        if (!map[i][j]) { continue; }
        Node<string> from = graph.AddNode($"{i}-{j}");
        if (i > 0 && map[i - 1][j])
        {
            Node<string> to = graph.Nodes
                .Find(n => n.Data == $"{i - 1}-{j}")!;
            graph.AddEdge(from, to, 1);
        }
        if (j > 0 && map[i][j - 1])
        {
            Node<string> to = graph.Nodes
                .Find(n => n.Data == $"{i}-{j - 1}")!;
            graph.AddEdge(from, to, 1);
        }
    }
}
```

First, you initialize a new undirected and weighted graph. Then, you use two `for` loops to iterate through all the places on the board. Within such loops, you check whether the given place is available. If so, you create a new node (`from`). Then, you check whether the node placed immediately above the current one is also available. If so, a suitable edge is added with a weight equal to `1`. Similarly, you can check whether the node placed on the left of the current one is available and add an edge if necessary.

Now, you just need to get the `Node` instances representing the source and the target nodes. You can do so by using the `Find` method and providing the textual representation of the node – for example, `0-0` or `16-24`. Then, you call the `GetShortestPath` method. In this case, the algorithm will try to find the shortest path between the node in the first row and column and the node in the last row and column. The code is presented in the following block:

```
Node<string> s = graph.Nodes.Find(n => n.Data == "0-0")!;
Node<string> t = graph.Nodes.Find(n => n.Data == "16-24")!;
List<Edge<string>> path = graph.GetShortestPath(s, t);
```

The last part of the code is related to presenting the map in the console:

```
Console.OutputEncoding = Encoding.UTF8;
for (int r = 0; r < map.Length; r++)
{
    for (int c = 0; c < map[r].Length; c++)
    {
        bool isPath = path.Any(e =>
            e.From.Data == $"{r}-{c}"
            || e.To.Data == $"{r}-{c}");
        Console.ForegroundColor = isPath
            ? ConsoleColor.White
            : map[r][c]
                ? ConsoleColor.Green
                : ConsoleColor.Red;
        Console.Write("\u25cf ");
    }
    Console.WriteLine();
}
Console.ResetColor();
```

To start, you set the proper encoding in the console to be able to present Unicode characters as well. Then, you use two **`for`** loops to iterate through all the places on the board. Inside such loops, you choose a color that should be used to represent a point in the console, either green (the point is available) or red (unavailable). If the currently-analyzed point is a part of the shortest path, the white color is set. Finally, you write the Unicode character representing a bullet. When the program's execution exits both loops, the console's color is reset.

When you run the application, you will see the following result:

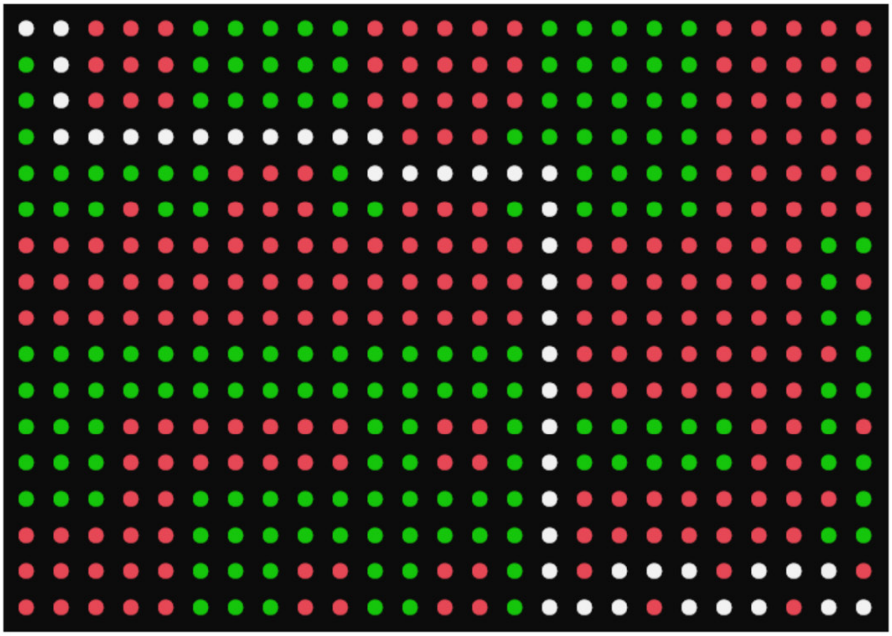

Figure 8.24 – Screenshot of the game map example

Great work! Now, let's summarize the topics that were covered in this chapter.

Summary

This chapter was related to one of the most important data structures available while developing applications: graphs. As you learned, a **graph** is a data structure that consists of **nodes** and **edges**. Each edge connects two nodes. What's more, there are various variants of edges, such as undirected and directed, as well as unweighted and weighted. All of them were described and explained in detail, and illustrations and code examples were provided. Two methods of graph representation, namely using an **adjacency list** and an **adjacency matrix**, were explained as well. You also learned how to implement a graph in the C# language.

While talking about graphs, it's important to present some **real-world applications**, especially due to the common use of such a data structure. For example, this chapter explained the structure of friends that are available on social media or the problem of searching for the shortest path in a city.

Among the topics that were covered in this chapter, you learned how to traverse a graph to visit all of the nodes in some particular order. Two approaches were presented, namely **DFS** and **BFS**. It's worth mentioning that the traversal topic can be also applied to searching for a given node in a graph.

Next, the subject of **spanning trees**, as well as **minimum spanning trees**, was introduced. As a reminder, a spanning tree is a subset of edges that connects all nodes in a graph without cycles, while

an MST is a spanning tree with the minimum cost from all spanning trees available in the graph. There are a few approaches to finding the MST, including **Kruskal's** and **Prim's algorithms**.

Then, you learned how to solve the problem of **coloring**, where you assigned colors (numbers) to all the nodes to comply with the rule that there cannot be an edge between two nodes with the same color.

The other problem was searching for the **shortest path between two nodes**, which takes into account a specific cost, such as the distance, the necessary time, or even the amount of fuel required. There are several approaches to the topic of searching for the shortest path in a graph. However, one of the common solutions is **Dijkstra's algorithm**, which makes it possible to calculate the distance from a starting node to all the nodes located in the graph. We covered this in detail in this chapter.

Now, it is high time to proceed to the next chapter, which focuses on **practical aspects of algorithms** from various groups, including recursive, greedy, back-tracking, and even genetic. Let's turn the page and see them in action!

9

See in Action

As you already know, algorithms are almost everywhere, and there are many types and classifications. They are supported by numerous data structures and some of them you learned while reading the previous chapters. After some theoretical parts, it is high time to keep on practicing, based on interesting examples. They are chosen from various types of algorithms, summarizing many subjects that you have got to know.

First, you will see how to calculate a given number from the **Fibonacci series** in a few variants that differ significantly in performance results, so you will get to know how you can optimize your code. Sometimes, even small changes can lead to huge performance improvements. Then, you will learn how to apply the greedy approach to solve the **minimum coin change** problem, as well as how to benefit from the divide-and-conquer algorithm to find the **closest pair of points** located on the two-dimensional surface. You will also see a beautiful **fractal** and the code that designs such graphics. The following examples will be related to applications of back-tracking with recursion to solve puzzles, namely **rat in a maze** and **Sudoku**. Coming closer to the end of the chapter, you will see how to apply a genetic algorithm to **guess a title** of this book, based on the rules of Darwinian theory of evolution and natural selection. The last example will be a brute-force algorithm for **guessing a secret password**.

As you can see, there are a lot of interesting examples just ahead of you, so be ready to write quite a lot of code and solve these tasks together. Let's start!

In this chapter, you will cover the following topics:

- The Fibonacci series
- Minimum coin change
- The closest pair of points
- Fractal generation
- Rat in a maze
- A Sudoku puzzle
- A title guess
- A password guess

The Fibonacci series

As the first example, let's take a look at calculating a given number from the **Fibonacci series**, using the following **recursive** function:

$$F_n = \begin{cases} 0, \text{ if } n = 0 \\ 1, \text{ if } n = 1 \\ F_{n-1} + F_{n-2}, \text{ if } n > 1 \end{cases}$$

Figure 9.1 – A formula for calculating a number from the Fibonacci series

Its interpretation is very simple:

- *F(0)* is equal to 0
- *F(1)* is equal to 1
- *F(n)* is a sum of *F(n-1)* and *F(n-2)*, which means that this number is a sum of the two preceding ones

As an example, *F(2)* is equal to the sum of *F(0)* and *F(1)*. Thus, it is equal to 1, while *F(3)* is equal to 2. It is worth mentioning that there are two base cases, namely for *n* equal to 0 and 1. For both of them, there is a specific value defined, namely 0 and 1.

The **recursive** implementation in the C# language is shown as follows:

```
long Fibonacci(int n)
{
    if (n == 0) { return 0; }
    if (n == 1) { return 1; }
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

As you see, the `Fibonacci` method calls itself twice with different values of parameters, namely smaller by 1 and 2 than the `n` parameter passed to the method. If you call the method passing `25`, you will receive 75025 as a result, as follows:

```
long result = Fibonacci(25);
```

Keep in mind that the presented recursive version for calculating a value of the Fibonacci function is very inefficient and will be very slow for larger input numbers.

You can significantly improve its performance using **dynamic programming**, either with top-down or bottom-up approaches. First, let's use the **top-down approach** with **memoization** to **cache the calculated results** for subproblems:

```
Dictionary<int, long> cache = [];
long Fibonacci(int n)
{
    if (n == 0) { return 0; }
    if (n == 1) { return 1; }
```

```
    if (cache.ContainsKey(n)) { return cache[n]; }
    long result = Fibonacci(n - 1) + Fibonacci(n - 2);
    cache[n] = result;
    return result;
}
```

You use the `Dictionary` class as a cache, where keys are values of `n` passed to the `Fibonacci` method and values are the calculated results, namely `Fibonacci(n)`. Within the method, you add the check on whether the cache contains a key equal to `n`. If so, you do not perform further operations and simply return the value from the cache. If the cache does not have such a key yet, you use the same approach as in the case of the recursive version and add the calculated result to the cache just before returning the result.

Is it worth introducing such changes? Let's see some numbers regarding execution time for the 50th number from the Fibonacci series. In the basic recursive version, it took more than 88 seconds on my machine. Introducing the top-down approach caused the same result received in... less than 1 millisecond. This solution is almost 100,000 times faster!

Now you know that dynamic programming can make a huge difference, let's take a look at the **bottom-up approach** for the Fibonacci number calculation:

```
long Fibonacci(int n)
{
    if (n == 0) { return 0; }
    if (n == 1) { return 1; }
    long a = 0;
    long b = 1;
    for (int i = 2; i <= n; i++)
    {
        long result = a + b;
        a = b;
        b = result;
    }
    return b;
}
```

Here, a bigger modification is introduced because you replace recursion with iteration. However, the code is very simple, as it consists of only one `for` loop that iterates from 2 until the given number and calculates the sum of the two preceding values. Of course, there are separate `if` conditions for the 0 and 1 values of the `n` parameter.

And what about the performance in this case? Let's compare calculating the 5,000th number from the Fibonacci series using both the top-down and bottom-down approaches. The top-down approach requires about 2 milliseconds, while the bottom-up still takes less than 1 millisecond on my laptop. Keep in mind that we are now talking about the 5,000th number from the Fibonacci series, and previously, the tests were made for only the 50th number. Incredible performance boost, isn't it?

RESULTS CAN DIFFER

The performance results are obtained on my computer and are calculated in a very simple way, even without repeating them several times. Of course, such results can be different in other circumstances, such as while using your machine. However, it is crucial to present some trend, not a precise result in milliseconds. This performance testing aims to show you a huge difference between a basic recursive version and any of the optimized versions with dynamic programming.

After the first example, let's proceed to solving the minimum coin change problem.

Minimum coin change

The second example shown in this chapter presents a **greedy algorithm** to solve the **minimum coin change** problem, for finding the minimum number of coins to receive the amount specified as the input.

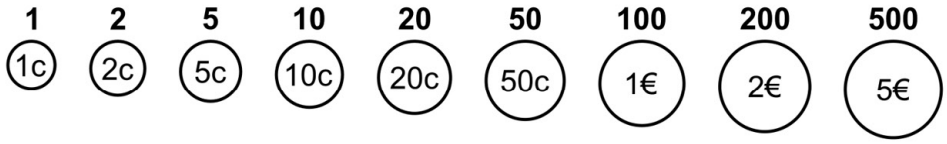

Figure 9.2 – Illustration of denominations in the case of the euro currency

For example, for the coin system consisting of 1, 2, 5, 10, 20, 50, 100, 200, and 500 denominations, if you want to get a value of 158, you need to pick 5 coins, namely 100, 50, 5, 2, and 1. The greedy approach is very simple because you just **pick the largest possible denomination not greater than the remaining amount**. You perform this operation until the remaining amount is equal to 0. As you see, the algorithm does not care about the overall solution and tries to choose the best solution at each step.

The C#-based implementation is shown here:

```
int[] den = [1, 2, 5, 10, 20, 50, 100, 200, 500];
List<int> coins = GetCoins(158);
coins.ForEach(Console.WriteLine);
List<int> GetCoins(int amount)
{
    List<int> coins = [];
    for (int i = den.Length - 1; i >= 0; i--)
    {
        while (amount >= den[i])
        {
            amount -= den[i];
            coins.Add(den[i]);
        }
    }
    return coins;
}
```

The most important role is performed by the `GetCoins` method, which takes one input, namely the amount to get. It returns a list of chosen coins. For example, if you call this method passing 158, you

will see 100, 50, 5, 2, and 1 in the console.

That's a quick example! Now, let's proceed to something a bit more complex.

Closest pair of points

Another example is an algorithm to **find the closest pair of points** located on the two-dimensional surface. It is an interesting algorithmic problem that can be solved using the **divide-and-conquer** paradigm.

Each point is represented by *x* and *y* coordinates, with values starting from (**0**, **0**) in the top-left corner of the surface. To find the closest pair of points, you first sort all points according to the *x* coordinate, as shown in the following diagrams, marked from **A** to **N**:

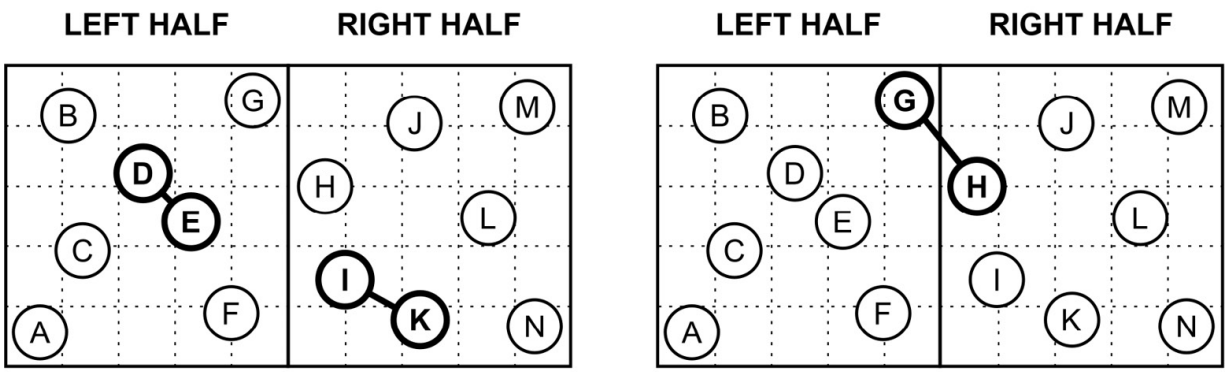

Figure 9.3 – Diagrams of the algorithm to find the closest pair of points

Then, you divide the surface into two halves. You can do this by calculating half of the points count, namely 7 in our example, and taking the first 7 points as the left half and the next 7 points as the right half.

Here is a task for **recursion**, so you recursively find the closest points in both halves and store data as r_l (points **D** and **E**) and r_r (points **I** and **K**). You choose the closer pair by comparing such distances and store the result as *r*, namely points **D** and **E** in our case.

That's not all – you also need to check the distance between points from the left and the right half, as presented in the preceding diagram, on the right. To do so, you get an array with data of all points that are closer to the middle point (in respect to the *x* coordinate only) than the *r* distance of the already found pair of points (**D** and **E** in our example). Then, you find the closest pair of points in this array (**G** and **H** in the example). Let's name the result *s*. To complete the task, choose which pair, from *r* (**D** and **E**) and *s* (**G** and **H**), is closer. Then, just return the result (**D** and **E** in our case).

The most important part of the code is shown as follows:

```
Result? FindClosestPair(Point[] points)
{
    if (points.Length <= 1) { return null; }
    if (points.Length <= 3) { return Closest(points); }
    int m = points.Length / 2;
    Result r = Closer(
        FindClosestPair(points.Take(m).ToArray())!,
        FindClosestPair(points.Skip(m).ToArray())!);
    Point[] strip = points.Where(p => Math.Abs(p.X
        - points[m].X) < r.Distance).ToArray();
    return Closer(r, Closest(strip));
}
```

First, there is the base condition that terminates execution when the array of points is empty or contains only one element. Then, you check whether the number of points in the array is less than or equal to 3. If so, you choose the closest pair of points in the collection just by checking all possible variants. Otherwise, you choose an index of the middle point and call the method recursively for the left and right halves. Then, you get points from both halves that are close enough to the middle point, taking only *x* coordinates into account. Next, you calculate the distance between all points in the **`strip`** array to get the closest pair from it. Finally, you just return the closer pair of points.

As you see, the main part of the algorithm is pretty simple to implement and understand. So, let's talk about the rest, starting with the **`Point`** definition:

```
public record Point(int X, int Y)
{
    public float GetDistanceTo(Point p) =>
        (float)Math.Sqrt(Math.Pow(X - p.X, 2)
            + Math.Pow(Y - p.Y, 2));
};
```

The **`Result`** record is presented here:

```
public record Result(Point P1, Point P2, double Distance);
```

The main part of code uses two auxiliary methods, namely **`Closest`** and **`Closer`**. The first one searches for the closest pair of points, and its code is shown here:

```
Result Closest(Point[] points)
{
    Result result = new(points[0], points[0], double.MaxValue);
    for (int i = 0; i < points.Length; i++)
    {
        for (int j = i + 1; j < points.Length; j++)
        {
            double distance = points[i].GetDistanceTo(points[j]);
            if (distance < result.Distance)
            {
                result = new(points[i], points[j], distance);
            }
        }
    }
```

```
    return result;
}
```

The **`Closer`** method is presented in the following code snippet:

```
Result Closer(Result r1, Result r2) =>
    r1.Distance < r2.Distance ? r1 : r2;
```

Finally, let's take a look at a way of calling the described method:

```
List<Point> points =
[
    new Point(6, 45),   // A
    new Point(12, 8),   // B
    new Point(14, 31),  // C
    new Point(24, 18),  // D
    new Point(32, 26),  // E
    new Point(40, 41),  // F
    new Point(44, 6),   // G
    new Point(57, 20),  // H
    new Point(60, 35),  // I
    new Point(72, 9),   // J
    new Point(73, 41),  // K
    new Point(85, 25),  // L
    new Point(92, 8),   // M
    new Point(93, 43)   // N
];
points.Sort((a, b) => a.X.CompareTo(b.X));
Result? closestPair = FindClosestPair(points.ToArray());
if (closestPair != null)
{
    Console.WriteLine(
        "Closest pair: ({0}, {1}) and ({2}, {3})
         with distance: {4:F2}",
        closestPair.P1.X,
        closestPair.P1.Y,
        closestPair.P2.X,
        closestPair.P2.Y,
        closestPair.Distance);
}
```

You provide the collection of points, sort them by *x* coordinates, and call the **`FindClosestPair`** method, passing the whole array as a parameter. Finally, you show the following result in the console:

```
Closest pair: (24, 18) and (32, 26) with distance: 11.31
```

So, you get the same result as received when you analyzed the example at the beginning of this section. Good work – congratulations!

WHERE CAN YOU FIND MORE INFORMATION?

The examples shown in this chapter are representatives of various popular algorithmic problems, which you can receive even during interviews while recruiting for a job as a developer. These topics are also popular on the internet. For example, you can find more information about the aforementioned approach to the closest pair of points problem and its implementation at https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/. As GeeksForGeeks contains a huge number of various articles, you can also find entries there about some other problems mentioned in this chapter, together with some implementations, such as about the rat in a maze problem at

https://www.geeksforgeeks.org/rat-in-a-maze/ and about the Sudoku puzzle at https://www.geeksforgeeks.org/sudoku-backtracking-7/.

In my opinion, coding can be understood as a kind of art. Similar to painters who paint beautiful paintings, developers can write beautiful code. So, while we are talking about art, let's write beautiful code that will paint beautiful fractals!

Fractal generation

The **recursion** can be applied to many various algorithms, also related to computer graphics. For this reason, let's take a look at another example – **fractal generation** creating interesting patterns, such as the following:

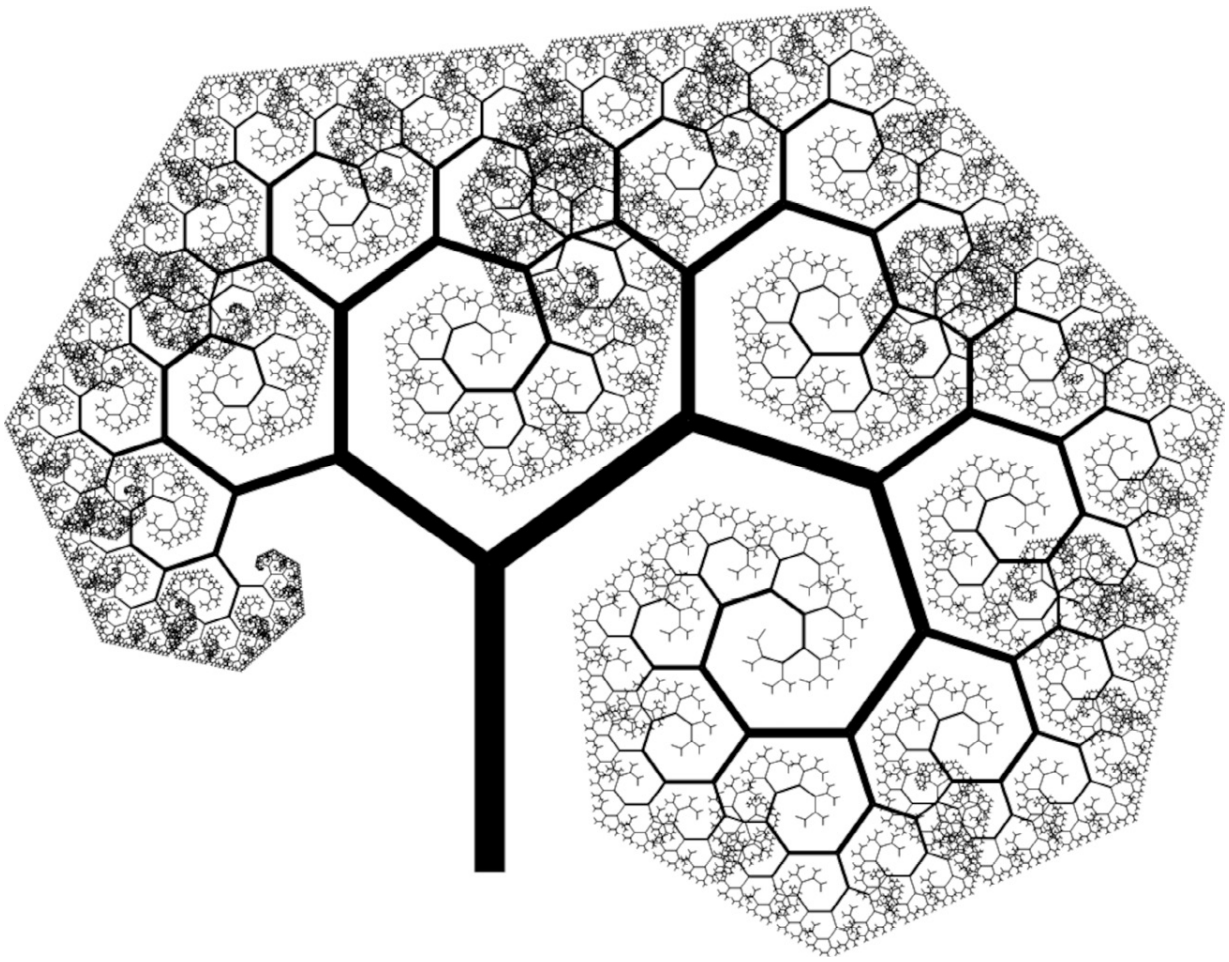

Figure 9.4 – An exemplary fractal generated using the recursive function

It's really beautiful, isn't it? Can you see some tree patterns in this image? If not, let's follow the bold line in the middle of the image (the tree *trunk*) and note that it is divided into two lines (*branches*), each rotated by a given degree. Then, follow one of these lines and see that it is divided according to

the same rule. This process is applied further and further until the specified number of levels is reached.

The description of this recursive algorithm in the natural language is quite easy, so let's take a look at code to calculate the coordinates of the start and end points of the following lines that together form the beautiful drawing. The code of the **`AddLine`** method is shown as follows:

```
void AddLine(int level, float x, float y,
    float length, float angle)
{
    if (level < 0) { return; }
    float endX = x + (float)(length * Math.Cos(angle));
    float endY = y + (float)(length * Math.Sin(angle));
    lines.Add(new(x, y, endX, endY));
    AddLine(level - 1, endX, endY, length * 0.8f,
        angle + (float)Math.PI * 0.3f);
    AddLine(level - 1, endX, endY, length * 0.6f,
        angle + (float)Math.PI * 1.7f);
}
```

The method takes a few parameters, namely the following:

- A level of pattern, starting with the non-negative number and leading to 0
- The *x* and *y* coordinates of the start point
- A length of the line
- Its angle, provided in radians

Within the method, you check the base condition, namely whether the level is smaller than 0. If not, you calculate *x* and *y* coordinates of the end point and add the line to the collection of lines (**`lines`**). At the end, you recursively call the **`AddLine`** method, passing different parameters. You decrease the level, pass the calculated end point coordinates as coordinates of a start point for the next line, decrease the length by 20% and 40% (depending on the branch), and also modify the angle.

It is worth noting that the preceding code uses the **`Line`** record, the code for which is as follows:

```
record Line(float X1, float Y1, float X2, float Y2)
{
    public float GetLength() =>
        (float)Math.Sqrt(Math.Pow(X1 - X2, 2)
            + Math.Pow(Y1 - Y2, 2));
}
```

The next part of the code is presented here:

```
using System.Drawing;
using System.Drawing.Drawing2D;
const int maxSize = 1000;
List<Line> lines = [];
AddLine(14, 0, 0, 500, (float)Math.PI * 1.5f);
```

Here, you specify the maximum width or height of the bitmap on which the fractal will be drawn (**`maxSize`**). Then, you prepare an empty list for the lines. In the last line, you call the **`AddLine`** method. You indicate that **`14`** levels of pattern will be added.

THE REQUIRED NUGET PACKAGE

*As you use elements from the **`System.Drawing`** and **`System.Drawing.Drawing2D`** namespaces, it is necessary to install an additional NuGet package, namely **`System.Drawing.Common`**.*

As soon as you have the collection of lines, you can calculate the minimum and maximum *x* and *y* coordinates, as well as the target *width* and *height*, as presented here:

```
float xMin = lines.Min(l => Math.Min(l.X1, l.X2));
float xMax = lines.Max(l => Math.Max(l.X1, l.X2));
float yMin = lines.Min(l => Math.Min(l.Y1, l.Y2));
float yMax = lines.Max(l => Math.Max(l.Y1, l.Y2));
float size = Math.Max(xMax - xMin, yMax - yMin);
float factor = maxSize / size;
int width = (int)((xMax - xMin) * factor);
int height = (int)((yMax - yMin) * factor);
```

The remaining part of code is related to printing the fractal on the bitmap:

```
using Bitmap bitmap = new(width, height);
using Graphics graphics = Graphics.FromImage(bitmap);
graphics.Clear(Color.White);
graphics.SmoothingMode = SmoothingMode.AntiAlias;
using Pen pen = new(Color.Black, 1);
foreach (Line line in lines)
{
    pen.Width = line.GetLength() / 20;
    float sx = (line.X1 - xMin) * factor;
    float sy = (line.Y1 - yMin) * factor;
    float ex = (line.X2 - xMin) * factor;
    float ey = (line.Y2 - yMin) * factor;
    graphics.DrawLine(pen, sx, sy, ex, ey);
}
bitmap.Save($"{DateTime.Now:HH-mm-ss}.png");
```

Within the presented code, you create a new instance of the **`Bitmap`** class with the specified size, as well as prepare the **`Graphics`** object to draw on this bitmap. Then, you paint the whole bitmap with a white color, set anti-aliasing, and specify a black pen for drawing.

The preceding piece of code involves the **`foreach`** loop. Within it, you calculate a line width, as well as the start and end coordinates. The last line in the loop simply draws the line. Finally, you save the prepared bitmap in the working directory in the file, whose name is created based on the current time.

DO YOU SEE WARNINGS?

*The prepared code shows some warnings in the IDE. They inform you about the availability of graphics-related features only on the Windows platform. You can hide such warnings by adding the line **`#pragma warning disable CA1416`** just before the preceding code, as well as adding the line **`#pragma warning restore CA1416`** just at the end. What's*

more, if you want to also draw graphics on other platforms, you can use other available NuGet packages, such as ***`SkiaSharp`****. I strongly encourage you to create this example with* ***`SkiaSharp`*** *as well.*

That's all! You can now adjust various parameters to paint beautiful fractals, even better than presented in the preceding figure. Some of the other results are shown here:

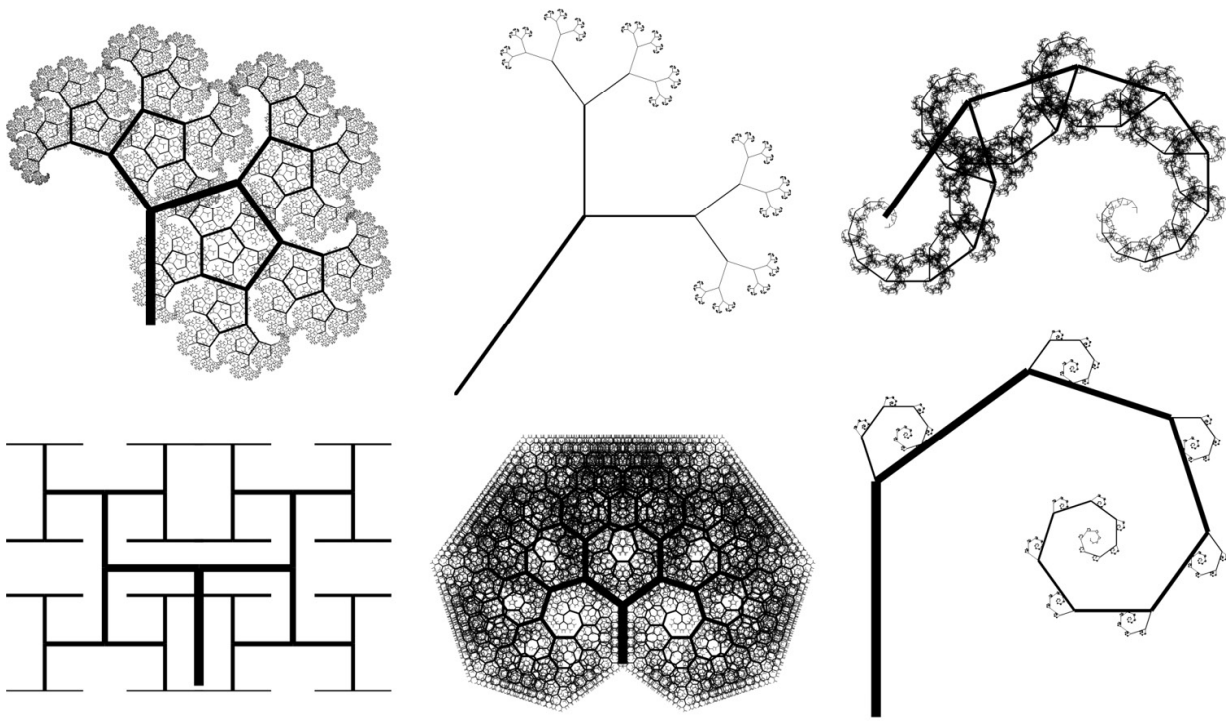

Figure 9.5 – Exemplary fractals generated using the recursive function

WHERE CAN YOU FIND MORE INFORMATION?

You can find a lot of content about fractals in the internet. However, an approach similar to presented here, is described at http://www.csharphelper.com/howtos/howto_curly_tree.html.

As soon as you are satisfied with the design of your fractal, let's move to the next section, where you will solve the *rat in a maze* puzzle.

Rat in a maze

Let's continue our adventure with examples by solving the **rat in a maze** problem with a **back-tracking algorithm**. The diagram is shown as follows:

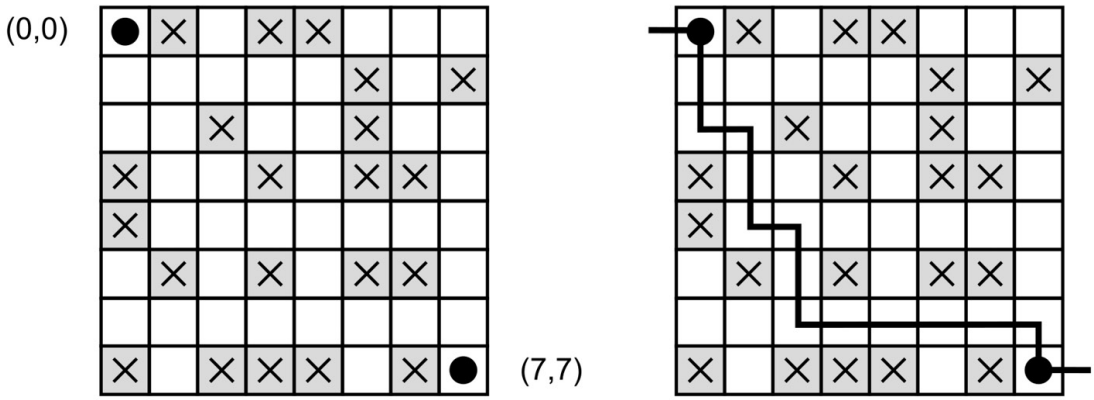

Figure 9.6 – Illustration of the rat in a maze example

Let's imagine that a rat is located in the top-left field on the board, which is marked as **(0, 0)** in the preceding figure, and we need to find a path to the exit, which is located in the bottom-right field and is marked as **(7, 7)**. Of course, some blocks are disabled (shown in gray) and the rat cannot go through them. To reach the target, the rat can go up, down, left, or right only using the available blocks.

You can solve this problem using the **recursion** to check possible paths leading the rat from the entry to the exit. If the currently calculated path does not reach the exit, you **backtrack** and try other variants.

The main part of the implementation is the `Go` method, as follows:

```
bool Go(int row, int col)
{
    if (row == size - 1
        && col == size - 1
        && maze[row, col])
    {
        solution[row, col] = true;
        return true;
    }
    if (row >= 0 && row < size
        && col >= 0 && col < size
        && maze[row, col])
    {
        if (solution[row, col]) { return false; }
        solution[row, col] = true;
        if (Go(row + 1, col)) { return true; }
        if (Go(row, col + 1)) { return true; }
        if (Go(row - 1, col)) { return true; }
        if (Go(row, col - 1)) { return true; }
        solution[row, col] = false;
        return false;
    }
    return false;
}
```

The method takes two parameters, namely **row** and **column**. It also uses three additional variables. The first is named **maze** and is a two-dimensional array representing the maze with available (filled with **true** values) and unavailable (**false**) fields for the rat. The second, namely **size**, stores the size of the maze, namely the number of rows, which is also equal to the number of columns. Another variable (**solution**) is similar to **maze**, but it stores the data of the currently checked path. The fields forming the solution are filled with **true** values, while others are filled with **false**.

At the beginning of the method, you check whether the rat already reached the exit. If so, you mark the final field as a part of the solution and return a value indicating that the rat completed its task and exited the maze. Otherwise, you check whether the rat is still within the maze and not on any unavailable field. If all of these conditions are met, you check whether this field is already a part of the path, and if so, you inform that this solution is incorrect.

If the rat is within the maze and on an available field that has not already been visited, you mark this field as a part of the solution and try to go down, right, up, and left by calling the **Go** method recursively. If none of these moves reaches the target (of course, also after the next steps), you indicate that the current field is not a part of the solution, which represents **back-tracking**. Then, you return a value indicating that the target has not been reached.

Next, take a look at the code that calls the **Go** method for the first time:

```
int size = 8;
bool t = true;
bool f = false;
bool[,] maze =
{
    { t, f, t, f, f, t, t, t },
    { t, t, t, t, t, f, t, f },
    { t, t, f, t, t, f, t, t },
    { f, t, t, f, t, f, f, t },
    { f, t, t, t, t, t, t, t },
    { t, f, t, f, t, f, f, t },
    { t, t, t, t, t, t, t, t },
    { f, t, f, f, f, t, f, t }
};
bool[,] solution = new bool[size, size];
if (Go(0, 0)) { Print(); }
```

The remaining code is related to the **Print** method:

```
void Print()
{
    for (int row = 0; row < size; row++)
    {
        for (int col = 0; col < size; col++)
        {
            Console.Write(solution[row, col] ? "x" : "-");
        }
        Console.WriteLine();
    }
}
```

The result is shown here:

```
x-------
x-------
xx------
-x------
-xx-----
--x-----
--xxxxxx
-------x
```

As we conclude this example, it is worth mentioning how the code is simple and short. Thus, you can define the solution to a problem in a clear way. However, keep in mind that if there is more than one path, an algorithm shows only one.

After helping the rat to find a path in a maze, let's move on to the next example, where you will learn how to automatically solve a Sudoku puzzle.

A Sudoku puzzle

Have you ever solved **Sudoku**? It is a very popular game that requires you to **fill empty cells of a 9x9 board with numbers from 1 to 9**. However, **each row, each column, and each 3x3 box must contain only unique numbers**. An exemplary starting board and a solved one are shown as follows:

	5		4		1			6
1			9	5		8		
9		4		6				1
6	2				5	3		4
	9			7		2		5
5		7					8	9
8			5	1	9			2
2	3				6	5		8
4	1		2		8	6		

7	5	**3**	4	**8**	1	**9**	**2**	6
1	**6**	**2**	9	5	**7**	8	**4**	**3**
9	**8**	4	**3**	6	**2**	**7**	**5**	1
6	2	**1**	**8**	**9**	5	3	**7**	4
3	9	**8**	**1**	7	**4**	2	**6**	5
5	**4**	7	**6**	**2**	**3**	**1**	8	9
8	**7**	**6**	5	1	9	**4**	**3**	2
2	3	**9**	**7**	**4**	6	5	**1**	8
4	1	**5**	2	**3**	8	6	**9**	**7**

Figure 9.7 – An example of non-solved and solved Sudoku puzzles

Now, you will learn how to solve Sudoku not with the usage of a pencil and a piece of paper but with an algorithm! You can perform this task using the **back-tracking** approach, trying to assign numbers to empty cells if, of course, they meet the conditions regarding unique numbers in each row, column, and box. If an entered number does not result in solving the whole puzzle, you assign another number and perform the check again. Let's take a look at the most important part of the code:

```
bool Solve()
{
    (int row, int col) = GetEmpty();
    if (row < 0 && col < 0) { return true; }
    for (int i = 1; i <= 9; i++)
    {
        if (IsCorrect(row, col, i))
        {
            board[row, col] = i;
            if (Solve()) { return true; }
            else { board[row, col] = 0; }
        }
    }
    return false;
}
```

At the beginning of the `Solve` method, you get coordinates of the first empty cell. If there are no empty cells, the `GetEmpty` method returns (**-1**, **-1**). Then, you return `true`, indicating that the game is solved.

Otherwise, you iterate through all possible numbers (namely from 1 to 9), using the `for` loop. In each iteration, you check whether the number can be correctly entered in this cell, using the `IsCorrect` method, ensuring that the number is unique in a row, a column, and a box. If so, you enter this number into the cell and **recursively** call the `Solve` method. If it returns `false`, indicating that this variant does not work, you backtrack by clearing the value entered in the cell, which means that it is empty and another variant needs to be used. If no variants lead to the solution, you return `false`.

The presented code uses two auxiliary methods, including `GetEmpty`, which searches for the first cell that is not already filled. Its code is as follows:

```
(int, int) GetEmpty()
{
    for (int r = 0; r < 9; r++)
    {
        for (int c = 0; c < 9; c++)
        {
            if (board[r, c] == 0) { return (r, c); }
        }
    }
    return (-1, -1);
}
```

The second auxiliary method is named `IsCorrect` and ensures that after entering a provided number in a given cell (with a specified row and column), the board still meets the criteria of the Sudoku game. Its code is presented here:

```
bool IsCorrect(int row, int col, int num)
{
    for (int i = 0; i < 9; i++)
    {
        if (board[row, i] == num) { return false; }
        if (board[i, col] == num) { return false; }
    }
```

```
    int rs = row - row % 3;
    int cs = col - col % 3;
    for (int r = rs; r < rs + 3; r++)
    {
        for (int c = cs; c < cs + 3; c++)
        {
            if (board[r, c] == num) { return false; }
        }
    }
    return true;
}
```

At the beginning, you check whether values are unique in a given row and column. The remaining part checks whether a particular 3x3 box contains only unique numbers.

The exemplary code for launching the Sudoku solving algorithm is as follows:

```
int[,] board = new int[,]
{
    { 0, 5, 0, 4, 0, 1, 0, 0, 6 },
    { 1, 0, 0, 9, 5, 0, 8, 0, 0 },
    { 9, 0, 4, 0, 6, 0, 0, 0, 1 },
    { 6, 2, 0, 0, 0, 5, 3, 0, 4 },
    { 0, 9, 0, 0, 7, 0, 2, 0, 5 },
    { 5, 0, 7, 0, 0, 0, 0, 8, 9 },
    { 8, 0, 0, 5, 1, 9, 0, 0, 2 },
    { 2, 3, 0, 0, 0, 6, 5, 0, 8 },
    { 4, 1, 0, 2, 0, 8, 6, 0, 0 }
};
if (Solve()) { Print(); }
```

Finally, let's take a look at the `Print` method:

```
void Print()
{
    for (int r = 0; r < 9; r++)
    {
        for (int c = 0; c < 9; c++)
        {
            Console.Write($"{board[r, c]} ");
        }
        Console.WriteLine();
    }
}
```

The result is shown here:

```
7 5 3 4 8 1 9 2 6
1 6 2 9 5 7 8 4 3
9 8 4 3 6 2 7 5 1
6 2 1 8 9 5 3 7 4
3 9 8 1 7 4 2 6 5
5 4 7 6 2 3 1 8 9
8 7 6 5 1 9 4 3 2
2 3 9 7 4 6 5 1 8
4 1 5 2 3 8 6 9 7
```

As you can see, a back-tracking algorithm can be successfully applied to solve both rat in a maze and Sudoku puzzles. You can achieve this goal with short and clear code that is also easy to understand.

So, after these examples, let's move on to the next section where you will see an interesting application of a genetic algorithm.

Title guess

It is high time to change a type of applied algorithm to a **heuristic** one, which has many applications and also subtypes. Here, we focus only on **genetic algorithms**, which are **adaptive heuristic search algorithms**. They are related to the Darwinian theory of evolution and natural selection. According to it, individuals in a population compete, and the **population evolves to create next generations that are better suited to survive**. The genetic algorithms operate on strings that evolve to receive possibly the highest value of **fitness**, complying with the **rule of survival** and **passing on the genes of the fittest parents**, also based on a **randomized data exchange**. The algorithm ends its operation when a suitable value of fitness is reached or when the maximum number of generations is reached.

WHERE CAN YOU FIND MORE INFORMATION?

You can find a lot of content about genetic algorithms in the internet, such as in the article published at https://link.springer.com/article/10.1007/s11042-020-10139-6. The simple approach to a genetic algorithm, which is shown in this chapter, is based on the solution presented at https://www.geeksforgeeks.org/genetic-algorithms/.

Let's take a look at an example of a genetic algorithm application to guess the title of this book. The first part of the code is as follows:

```
const string Genes = "abcdefghijklmnopqrstuvwxyz
    #ABCDEFGHIJKLMNOPQRSTUVWXYZ ";
const string Target = "C# Data Structures and Algorithms";
Random random = new();
int generationNo = 0;
List<Individual> population = [];
for (int i = 0; i < 1000; i++)
{
    string chromosome = GetRandomChromosome();
    population.Add(new(chromosome,
        GetFitness(chromosome)));
}
```

First, you create an initial population, with 1,000 individuals. Each individual has a random chromosome, represented by a random string whose length is equal to the target string, which is a title of the book. Let's go further:

```
List<Individual> generation = [];
while (true)
{
    population.Sort((a, b) =>
        b.Fitness.CompareTo(a.Fitness));
    if (population[0].Fitness == Target.Length)
    {
        Print();
        break;
```

```
    }
    generation.Clear();
    for (int i = 0; i < 200; i++)
    {
        generation.Add(population[i]);
    }
    for (int i = 0; i < 800; i++)
    {
        Individual p1 = population[random.Next(400)];
        Individual p2 = population[random.Next(400)];
        Individual offspring = Mate(p1, p2);
        generation.Add(offspring);
    }
    population.Clear();
    population.AddRange(generation);
    Print();
    generationNo++;
}
```

The most interesting part is located in the infinite **`while`** loop. Here, you sort the population from the best fitted to survive – that is, by fitness in decreasing order. To explain it in detail, fitness is equal to 0 when no chars in the chromosome string match the following chars in the target string. In turn, fitness is equal to 33 (i.e., the number of chars in the book title), when the chromosome string is equal to the target string. For this reason, if the first element from the population (namely the fittest) has a fitness equal to the target string length, it means that the solution is found, so you just print it and exit the loop.

Otherwise, you clear the list with data of a new generation and add 200 best-fitted individuals to it. This means that **20% of the best-fitted individuals are moved automatically to the next generation**. For the remaining 800 places in the new generation, you perform **crossover** and **randomly choose parents, from 40% of the best-fitted individuals, to generate new individuals**. Then, you replace the current population with the new generation and proceed to the next iteration.

It's worth mentioning the **`Individual`** record, the code for which is as follows:

```
record Individual(string Chromosome, int Fitness);
```

It contains two properties, namely **`Chromosome`** and **`Fitness`**. The first stores the string adjusted in the evolution, while the other is the number indicating how this particular individual is fit to survive. Of course, a higher value is better.

The **`Mate`** method is used to generate a new individual using two parents:

```
Individual Mate(Individual p1, Individual p2)
{
    string child = string.Empty;
    for (int i = 0; i < Target.Length; i++)
    {
        float r = random.Next(101) / 100.0f;
        if (r < 0.45f) { child += p1.Chromosome[i]; }
        else if (r < 0.9f) { child += p2.Chromosome[i]; }
        else { child += GetRandomGene(); }
```

```
    }
    return new Individual(child, GetFitness(child));
}
```

The most interesting part of this method is the **`for`** loop in which the chromosome of the child is created, according to the following rules:

- **Approximately 45% of genes are taken from the first parent**
- **Approximately 45% of genes are taken from the second parent**
- **The remaining 10% are randomized**

And how can you get a random single gene or generate a random whole chromosome? You just take a look at the code:

```
char GetRandomGene() => Genes[random.Next(Genes.Length)];
string GetRandomChromosome()
{
    string chromosome = string.Empty;
    for (int i = 0; i < Target.Length; i++)
    {
        chromosome += GetRandomGene();
    }
    return chromosome;
}
```

The next necessary method is named **`GetFitness`**, which simply returns the number of characters that matches the target book title. Its code is as follows:

```
int GetFitness(string chromosome)
{
    int fitness = 0;
    for (int i = 0; i < Target.Length; i++)
    {
        if (chromosome[i] == Target[i]) { fitness++; }
    }
    return fitness;
}
```

Finally, let's take a look at the **`Print`** method:

```
void Print() => Console.WriteLine(
    $"Generation {generationNo:D2}:
    {population[0].Chromosome} / {population[0].Fitness}");
```

When you run the code, the best-fitted individual from each generation is presented, as shown in the following output:

```
Generation 00: UvWvvtycVTYAsJYxXZpanLkj#rDrmDIEI / 4
Generation 01: sXDGuQQDPnbjpRvWZs evqRNlg#yiwIPL / 5
Generation 02: j#TvvtmKToXuTjxBegpaCLkmNsornzg R / 7
Generation 03: fZCUBIT QrnuzwuWTskTOf bezodQwhmM / 8
Generation 04: CyDwafZZpinLziuPgs yID AevGrGf bs / 9
Generation 05: C# ZaBawSWwLoturSXOcIq wLeSgQOhme / 12 (...)
Generation 10:  Sboats ttrDcterus Mnt jmvGrifhms / 17 (...)
Generation 15: C kData ltrCkteres entbAagorZthmD / 21 (...)
```

```
Generation 20: C#VDatahStrdcturessanU Al#orithmd / 26 (...)
Generation 25: CZ Data StrunturOs awd Algorithms / 29 (...)
Generation 30: C# Data Structures Qjd Algorithms / 31 (...)
Generation 35: C# Data Structures and Algorothms / 32 (...)
Generation 37: C# Data Structures and Algorithms / 33
```

Is this *magic*? No, it's just the algorithm you wrote that manages the following generations and evolves the individuals, giving you the expected result.

A password guess

As an example of a **brute-force algorithm**, let's create a program to **generate all possible passwords and trying to guess your secret one**, which consists of small letters and digits only. The program starts with passwords of a length equal to 2 and proceeds until 8.

The first part of the code is presented here:

```
using System.Diagnostics;
using System.Text;
const string secretPassword = "csharp";
int charsCount = 0;
char[] chars = new char[36];
for (char c = 'a'; c <= 'z'; c++)
{
    chars[charsCount++] = c;
}
for (char c = '0'; c <= '9'; c++)
{
    chars[charsCount++] = c;
}
```

First, you specify a secret password, which you will try to guess in the remaining part of the code. Then, you create an array with available characters, namely small letters and digits. At the end of this code snippet, the `charsCount` variable stores the number of available characters.

The most interesting part of the code is the `for` loop, where each iteration represents a particular length of a password, between two and eight chars. The code is presented here:

```
for (int length = 2; length <= 8; length++)
{
    Stopwatch sw = Stopwatch.StartNew();
    int[] indices = new int[length];
    for (int i = 0; i < length; i++) { indices[i] = 0; }
    bool isCompleted = false;
    StringBuilder builder = new();
    long count = 0;
    while (!isCompleted)
    {
        builder.Clear();
        for (int i = 0; i < length; i++)
        {
            builder.Append(chars[indices[i]]);
        }
        string guess = builder.ToString();
```

```
            if (guess == secretPassword)
            {
                Console.WriteLine("Found.");
            }
            count++;
            if (count % 10000000 == 0)
            {
                Console.WriteLine($" > Checked: {count}.");
            }
            indices[length - 1]++;
            if (indices[length - 1] >= charsCount)
            {
                for (int i = length - 1; i >= 0; i--)
                {
                    indices[i] = 0;
                    indices[i - 1]++;
                    if (indices[i - 1] < charsCount) { break; }
                    if (i - 1 == 0 && indices[0] >= charsCount)
                    {
                        isCompleted = true;
                        break;
                    }
                }
            }
        }
        sw.Stop();
        int seconds = (int)sw.ElapsedMilliseconds / 1000;
        Console.ForegroundColor = ConsoleColor.White;
        Console.WriteLine($"{length} chars: {seconds}s");
        Console.ResetColor();
    }
```

The **`indices`** array has a length equal to the value of the **`length`** variable. Each item stores a current index from the **`chars`** array, indicating the char that is currently placed on the *i*-th location in the string. In each iteration of the **`while`** loop, you change values in the **`indices`** array until all possible combinations of the indices are used.

Furthermore, you save the guessed password in the **`guess`** variable, and here, it can be either printed on the console or hashed and compared with the hashed password that you want to guess. As this is only a demonstration of a brute-force algorithm, it does not stop its operation when the password is guessed. Thus, you can get more performance results and observe what impact the password length has on the required time for guessing.

As you can see, the brute-force approach is very simple, but what about performance? In the preceding code, you can see the usage of **`Stopwatch`**, so you can get some results. Generating all possible variants of a password consisting of two chars takes less than 1 millisecond. For three- and four-char passwords, the time is also very small, much less than 100 milliseconds. For five-char passwords, the time goes up to about two seconds, while generating passwords of a length equal to six chars takes almost a minute. If you add a mechanism to hash a password and compare it with the target hash, also taking into account that passwords can also contain capital letters and many other chars, the brute-force algorithm seems to be simply impractical in the case of longer passwords.

It is worth mentioning that the presented performance results were received on my computer and can be different on other devices. They are shown only to indicate a trend that as a password length increases, the time necessary to guess it is significantly longer with each added character. Thus, it is also a useful tip that you should always use a complicated password that contains small and capital letters, digits, and special characters. Of course, the length of the password is also important.

Summary

You just completed the ninth chapter of this book, which examined data structures and algorithms in the context of the C# language. This time, we focused on practical examples of algorithms, with code snippets, detailed descriptions, and also brief indications of which types of algorithms the aforementioned examples belong to.

First, you learned how to implement a simple algorithm to calculate a given number from the **Fibonacci series** in three variants. You saw a simple recursive approach as well as top-down and bottom-up approaches to dynamic programming.

The next example showed the greedy approach to solve the **minimum coin change** problem. It was followed by the divide-and-conquer algorithm to find the **closest pair of points** located on a two-dimensional surface. The fourth example presented a recursive way of **generating fractals** and drawing them on a bitmap.

The following two examples were related to back-tracking algorithms to solve the **rat in a maze** and the **Sudoku** puzzles. These examples used recursion as well.

Another interesting approach involved a genetic algorithm as a subtype of a heuristic algorithm. It was used to **guess the title of the book**, with the rules of the Darwinian theory of evolution and natural selection.

The last example used a brute-force algorithm to **guess a secret password**, by checking all possible variants of passwords. You saw that with the increasing password length, the time necessary to guess it increased significantly.

Now, it is high time to proceed to the overall summary to take a look at all of the data structures that have been presented in the book so far. Let’s turn the page and proceed to the last chapter!

10

Conclusion

As you saw while reading the book, there are many data structures with many configuration variants. Thus, **choosing a proper data structure is not an easy task**, which could have a significant impact on the performance of the developed solution. Even the topics mentioned in this book form quite a long list of described data structures. For this reason, it is a good idea to classify them in some way.

Within this chapter, the described data structures are grouped into linear and non-linear categories. Each element in a **linear data structure** can be logically adjacent to the following or the previous element. In the case of a **nonlinear data structure**, a single element can be logically adjacent to numerous others, not necessarily only one or two.

As it is the last chapter of the book, we will also summarize all of the gathered knowledge. Each data structure will be presented with a brief description, and some of them will be also shown with illustrations to help you remember this information.

In this chapter, the following topics will be covered:

- Classification
- Arrays
- Lists
- Stacks
- Queues
- Dictionaries
- Sets
- Trees
- Graphs
- The last word

Classification

I will start with a classification of the data structures shown already in the book. The classification divides all structures into linear and non-linear ones.

A **linear data structure** means that **each element can be logically adjacent to the following or the previous element**. There are several data structures that follow this rule, such as arrays, lists, stacks,

and queues. Of course, you should also take care of various subtypes of the mentioned data structures, such as four variants of a linked list, which is a subtype of a list.

A **non-linear data structure** indicates that **a single element can be logically adjacent to numerous others, not necessarily only one or two**. They can be freely distributed throughout the memory. Of course, graph-based data structures, including trees, are included in this group. Trees include binary trees, tries, and heaps, while a binary search tree is a subtype of a binary tree. In a similar way, you can describe the relationships of other data structures presented and explained in this book.

The mentioned classification is presented in the following diagram:

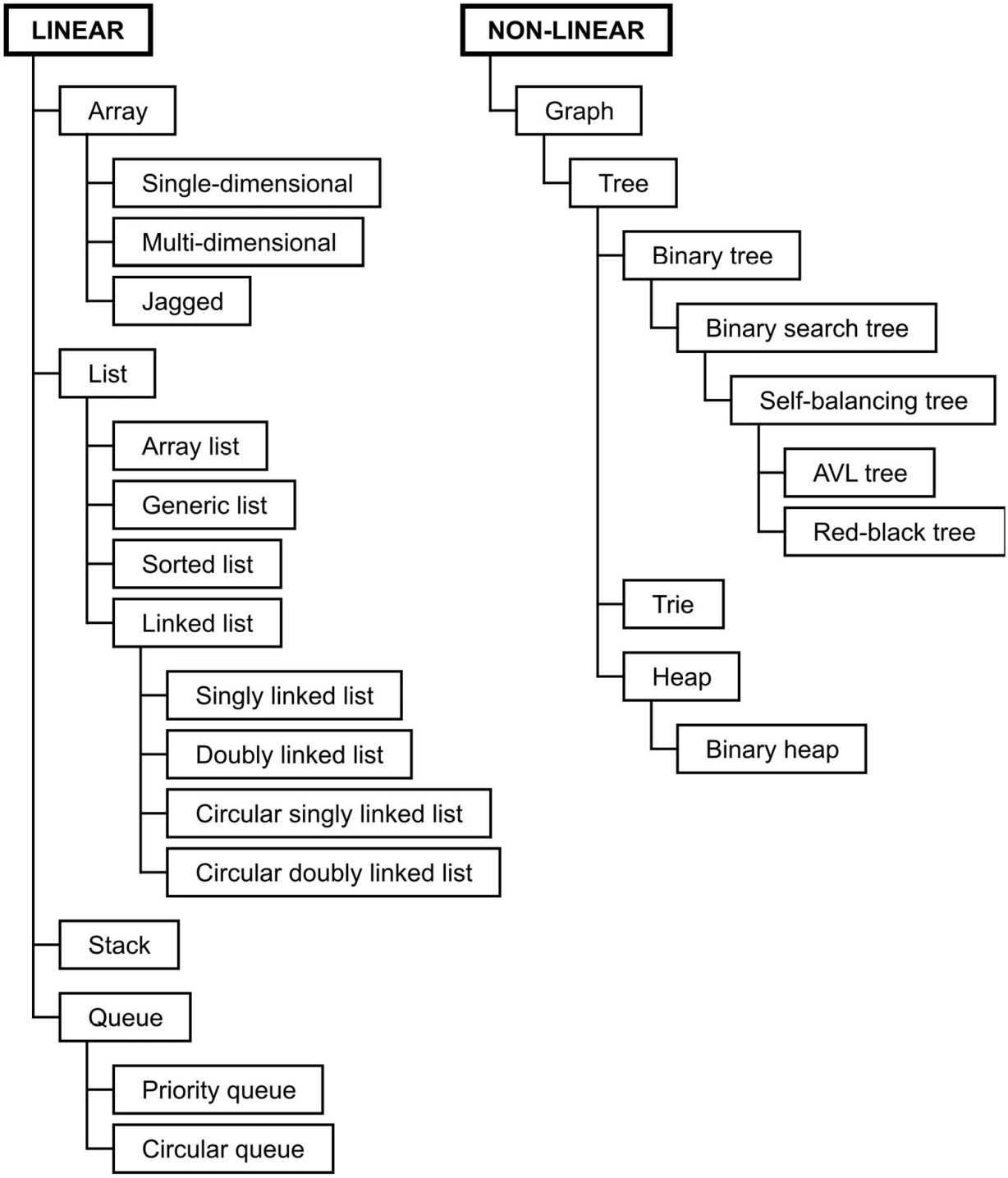

Figure 10.1 – Classification of data structures into linear and non-linear ones

Do you remember all of the data structures shown in the book? Due to the high number of described topics, it is a good idea to take a look at the following data structures once again. The associated

algorithms will be mentioned, as well. The remaining part of this chapter is a brief summary with information about some real-world applications.

Arrays

Let's start with an **array**, which was the main topic of *Chapter 3*, *Arrays and Sorting*. You can use this data structure to **store many data of the same type**, such as `int`, `string`, or a user-defined class. The important assumption is that the number of elements in an array cannot be changed after initialization. Moreover, arrays belong to **random access data structures**. This means that you can use indices to get access to the first, the middle, the *n*-th, or the last element from the array.

You can benefit from a few variants of arrays – namely, **single-dimensional**, **multi-dimensional**, and **jagged arrays**, also referred to as an **array of arrays**. All of these variants are shown in the following illustration:

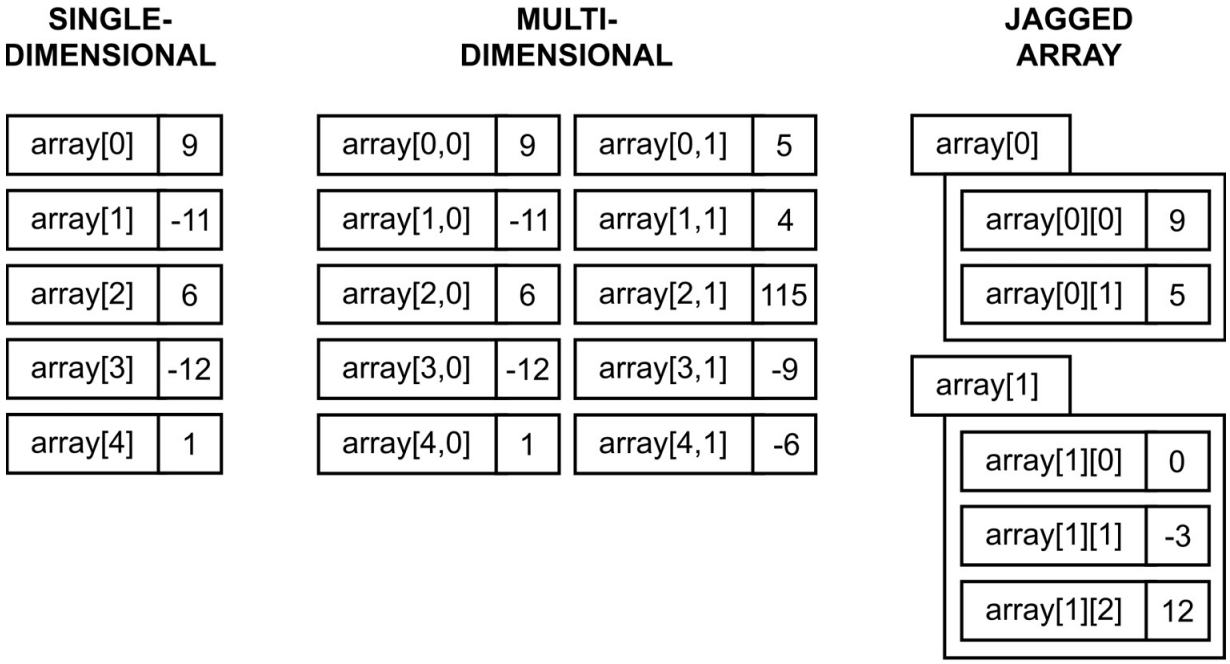

Figure 10.2 – Variants of arrays

There are a lot of applications for arrays and, as a developer, you have probably already used this data structure many times. In this book, you saw how you can use it to store various data, such as the **names of months**, the **multiplication table**, or even a **map of a game**. In the last case, you created a two-dimensional array with the same size as a map, where each element specified a certain type of terrain, such as grass.

There are many algorithms that perform operations on arrays. However, one of the most common tasks is sorting an array to arrange its elements in the correct order, either ascending or descending. This book focuses on seven algorithms, namely **selection sort**, **insertion sort**, **bubble sort**, **merge sort**, **Shell sort**, **quicksort**, and **heap sort**. Each of them was described and presented in the illustration, as well as written in the C# code, together with a detailed explanation.

Lists

The next group of data structures are **lists**. They were described in *Chapter 4*, *Variants of Lists*. Lists are similar to arrays but make it possible to **dynamically increase the size of the collection**, if necessary. It is worth mentioning that the built-in implementation is available for the array list (`ArrayList`), as well as its generic (`List`) and sorted (`SortedList`) variants. The latter can be understood as a collection of key-value pairs, always sorted by keys.

There are a few other variants of lists, including a **singly linked list**, a **doubly linked list**, a **circular singly linked list**, and a **circular doubly linked list**. The first variant makes it possible to easily navigate from one element to the next one. However, it can be further expanded by allowing navigating in forward and backward directions, forming the doubly linked list. In the circular doubly linked list, the first node navigates to the last one in the case of backward direction, while the last node navigates to the first in the forward direction. It is worth noting that there is a built-in implementation of the doubly linked list (`LinkedList`). You can quite easily extend it to behave as any circular linked list, either as a circular singly or circular doubly linked list.

Various variants of lists are shown in the following illustration:

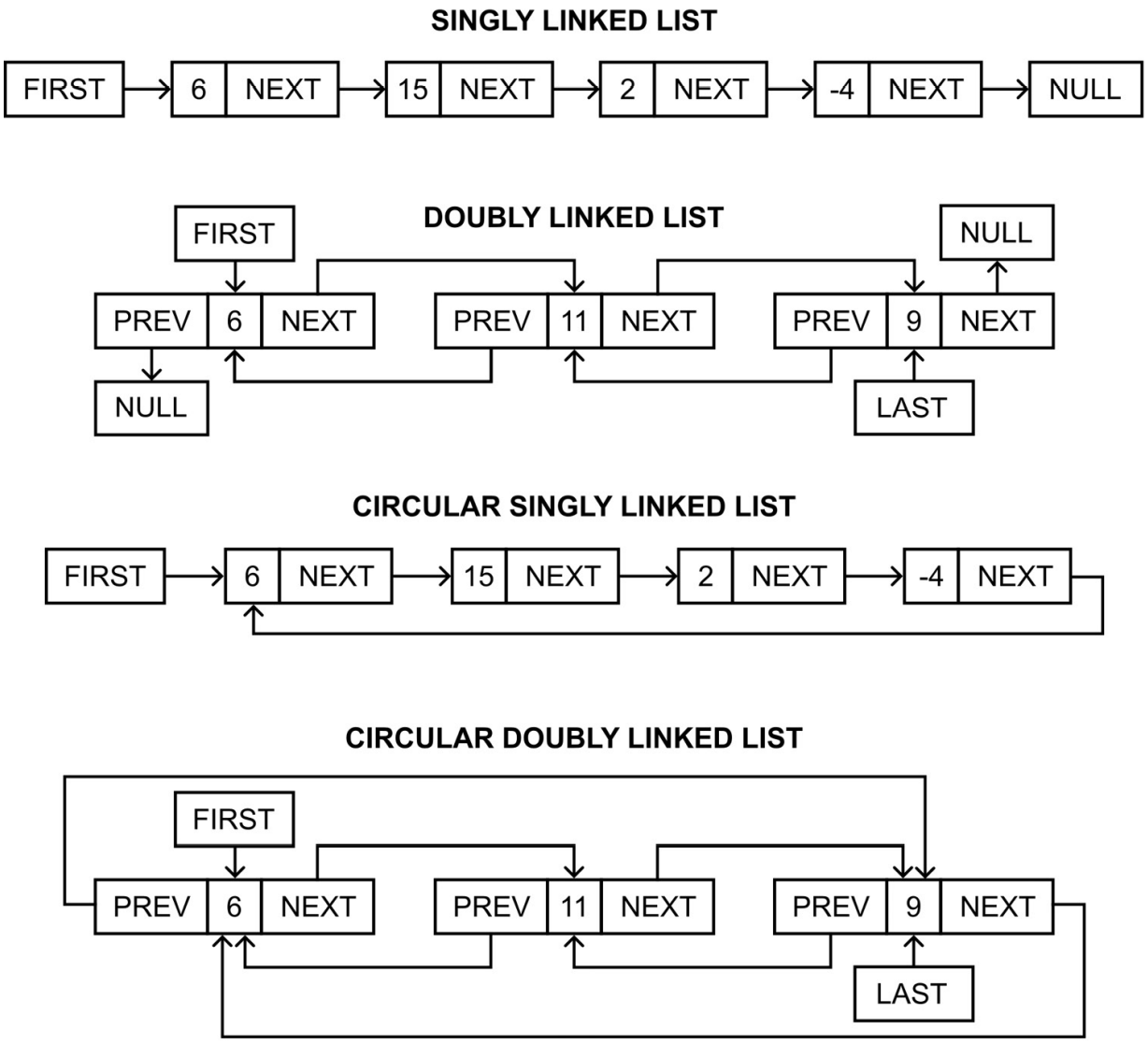

Figure 10.3 – Variants of lists

There are a lot of applications for the lists to solve diverse problems in various kinds of applications. In this book, you saw how to utilize the list to store some floating-point values and calculate the average value, how to use this data structure to create a simple **database of people**, and how to develop an automatically sorted **address book**. Moreover, you prepared an exemplary application that allows a user to **read the book** by changing the pages, as well as a game, in which the user **spins the wheel** with random power. The wheel rotates slower and slower until it stops. Then, the user can spin it again, from the previous stop position, which illustrates a circular linked list.

Stacks

Chapter 5, *Stacks and Queues*, focused on stacks and queues. Now, let's recap a **stack**, which is representative of **limited access data structures**. This name means that you cannot access every element from the structure. So, the way of getting elements is strictly specified. In the case of a stack, you can only add a new element at the top (the **push** operation) and get an element by removing it from the top (the **pop** operation). For this reason, a stack is consistent with the **LIFO** principle, which means **Last-In First-Out**. The built-in implementation as the `Stack` class is available, as well.

The illustration of a stack is shown as follows:

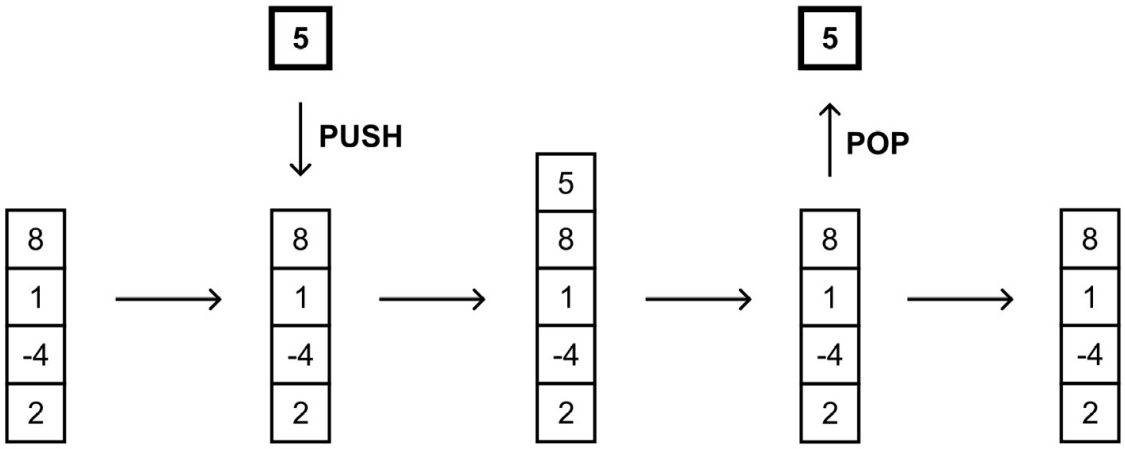

Figure 10.4 – Illustration of a stack

A stack has many real-world applications. One of the mentioned examples is related to a **pile of many plates**, each placed on top of the other. You can only add a new plate at the top of the pile, and you can only get a plate from the top of the pile. You cannot remove the seventh plate without taking the previous six from the top, and you cannot add a plate to the middle of the pile. You also saw how to use a stack to **reverse a word** and how to apply it to solve the mathematical game **Tower of Hanoi**. That's not all, because applications of stacks are much broader, such as for calculating mathematical expressions provided in the **reverse Polish notation**.

Queues

Another leading subject of *Chapter 5*, *Stacks and Queues*, was a **queue**. This is also a representative of limited access data structures. While using a queue, you can only add new elements at the end (the **enqueue** operation) and remove the element from the queue only from the beginning of the queue (the **dequeue** operation). For this reason, this data structure is consistent with the **FIFO** principle, which stands for **First-In First-Out**. The built-in implementation as the `Queue` class is available for you, as well.

The illustration of a queue is shown as follows:

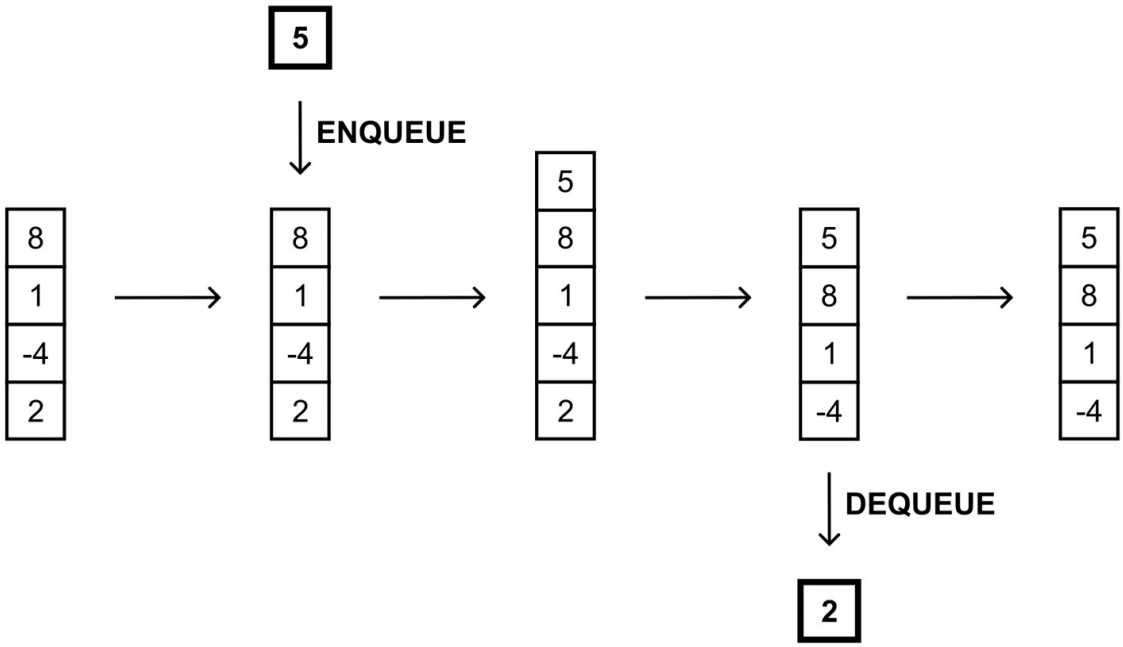

Figure 10.5 – Illustration of a queue

It is also possible to use a **priority queue**, which extends the concept of a queue by setting the priority for each element. Thus, the dequeue operation returns the element with the highest priority, which was added earliest to the queue. When all elements with the highest priority are dequeued, the priority queue handles elements with the next highest priority and dequeues such elements from those added earliest.

Another variant of a queue is a **circular queue**, also called a **ring buffer**, which was also presented and explained in the book. Here, a queue forms a circle, internally uses an array, and the maximum number of elements that can be placed inside the queue is limited. You specify indices for front and rear elements in this case.

There are many real-world applications of a queue. For example, a queue can be used to represent a **line of people** waiting in a shop at a checkout. New people stand at the end of the line, and the next person is taken to the checkout from the beginning of the line. You are not allowed to choose a person from the middle and serve them. Moreover, you saw a few examples of the solution of a **call center**, where there are many clients and one consultant, many clients and many consultants, or many clients (with different plans, either standard or priority support) and only one consultant, who answers the waiting calls. Another group of queue applications was shown while presenting **graph-based algorithms**. A queue was used in the **breadth-first search** algorithm for traversing a graph or

for searching for a given value in a graph. A priority queue was applied in **Dijkstra's algorithm** for searching the shortest path in a graph.

Dictionaries

The topic of *Chapter 6*, *Dictionaries and Sets*, was related to dictionaries and sets. First, let's recap a **dictionary**, which allows **mapping keys to values and performing fast lookups**. A dictionary uses a hash function and can be understood as a collection of pairs, each consisting of a key and a value.

There are two built-in versions of a dictionary – non-generic (`Hashtable`) and generic (`Dictionary`). The sorted variant of a dictionary (`SortedDictionary`) is available, as well. All of them were described in detail.

The mechanism of a hash table is presented in the following illustration:

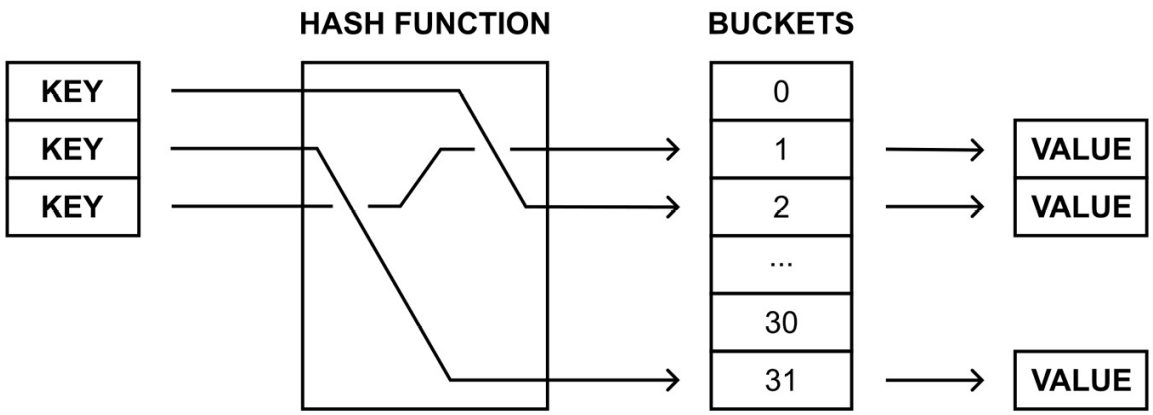

Figure 10.6 – Illustration of mapping keys to particular values

Due to the great performance of the hash table, such a data structure is frequently used in many real-world applications, such as for **associative arrays**, **database indices**, or **cache systems**. Within this book, you saw how to create a **phone book** to store entries where a person's name is a key and a phone number is a value. Among other examples, you developed an application that helps employees of shops **find the location** of where a product should be placed, and you applied the sorted dictionary to create a simple **encyclopedia**, where a user can add entries.

Sets

Another data structure from *Chapter 6*, *Dictionaries and Sets*, is a **set**, which is a **collection of distinct objects without duplicated elements and without any particular order**. Therefore, you can only get to know whether a given element is in the set or not. The sets are strictly connected with

mathematical models and operations, such as **union**, **intersection**, **subtraction**, and **symmetric difference**.

The exemplary sets, storing data of various types, are shown as follows:

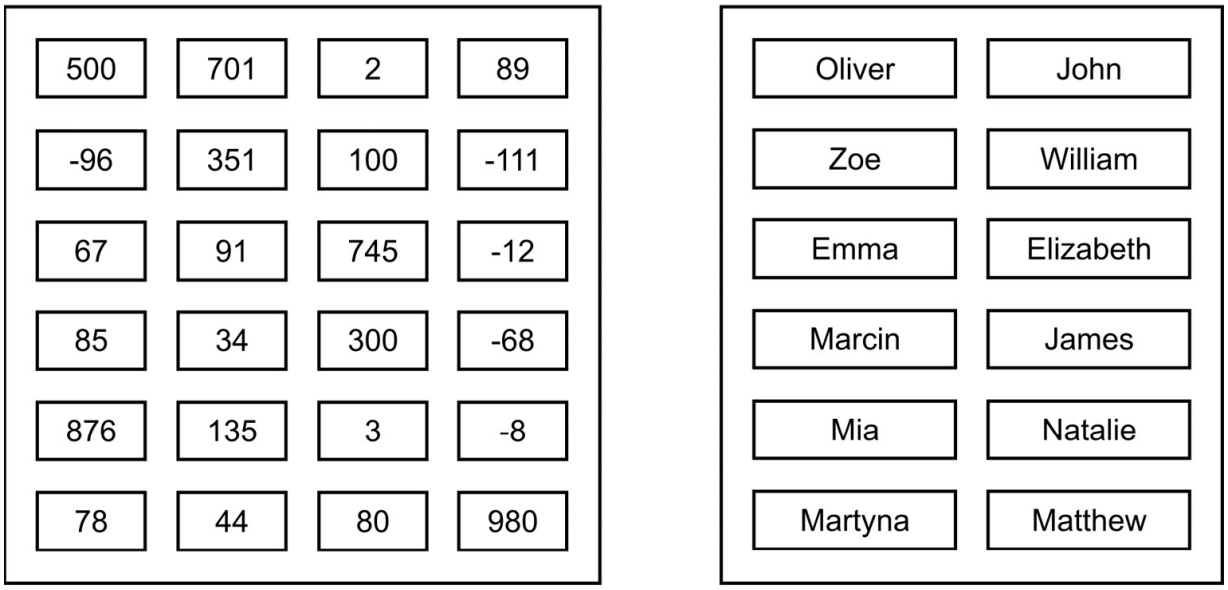

Figure 10.7 – Illustration of sets with integer and string values

While developing applications in the C# language, you can benefit from high-performance set-related operations provided by the `HashSet` class. As an example, you saw how to create a system that handles **one-time promotional coupons** and allows you to check whether the scanned one was already used. Another example was the **reporting service** for the system of a SPA center with four swimming pools. By using sets, you calculated statistics, such as the number of visitors to a pool, the most popular pool, and the number of people who visited at least one pool.

Trees

The next topic is about **trees**, which were the subject of *Chapter 7*, *Variants of Trees*. A tree consists of **nodes** with one **root**. The root contains no **parent** node, while all other nodes do. Moreover, each node can have any number of **child nodes**. The child nodes of the same parent can be called **siblings**, while a node without children is called a **leaf**.

An exemplary tree is shown here:

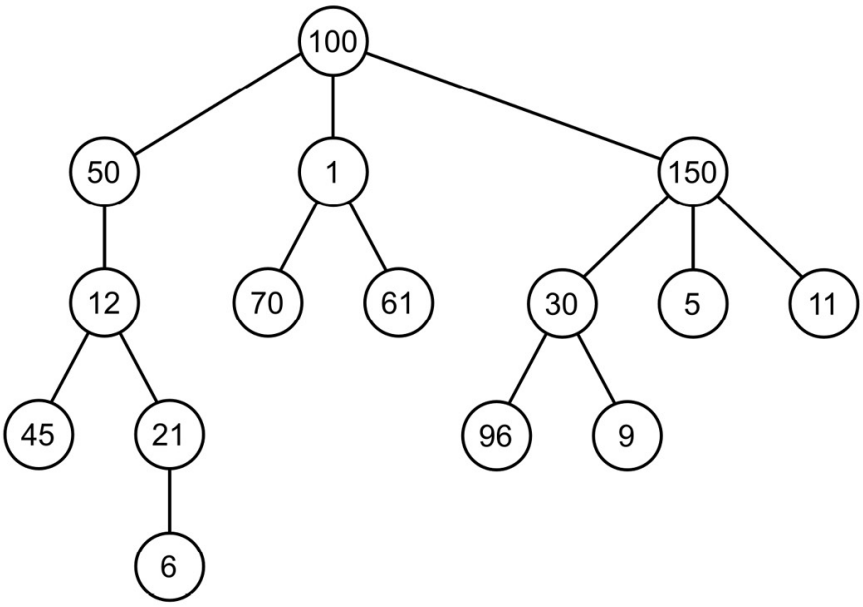

Figure 10.8 – Illustration of a tree

A tree is a data structure that is great for the representation of various data, such as the **structure of a company**, divided into a few departments, where each has its own structure. You also saw an example where a tree was used to arrange a **simple quiz** consisting of a few questions and answers, which are shown depending on the previously taken decisions.

Generally speaking, each node in a tree can contain any number of children. However, in the case of **binary trees**, a node cannot contain more than two children – that is, it can contain no child nodes, or only one or two. However, there are no rules about relationships between the nodes. The exemplary binary trees are shown here:

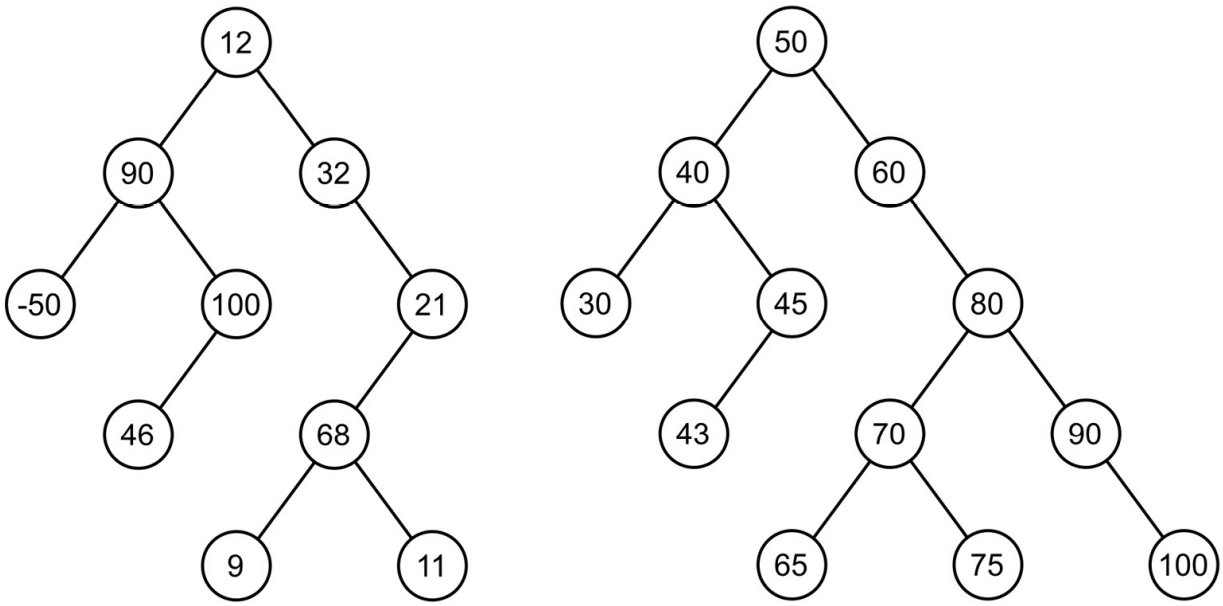

Figure 10.9 – Illustration of a binary tree and a binary search tree

If you want to use a **binary search tree** (**BST**), the next rule is introduced. It states that, for any node, the values of all nodes in its left subtree must be smaller than its value, and that the values of all nodes in its right subtree must be greater than its value. The exemplary BST is presented on the right-hand side of the preceding illustration.

Another group of trees is called **self-balancing trees**, which keep a tree balanced all the time while adding and removing nodes. Their application is very important because it allows you to form the correctly arranged tree, which has a positive impact on performance. There are several variants of self-balancing trees, but **AVL trees** and **red-black trees** (**RBTs**) are some of the most popular.

One of the tree applications is related to processing strings, such as for **auto-complete** and **spell-checker** features. Here, you can benefit from another tree-based data structure – namely, a **trie**. It is used to store strings and to perform prefix-based searching. A trie is a tree with one root node, where each node represents a string and each edge indicates a character. A trie node contains references to the next nodes – for example, as an array with 26 elements, representing the 26 characters from the alphabet (from `a` to `z`). When you go from the root to each node, you receive a string, which is either a saved word or its substring, as presented in the following illustration:

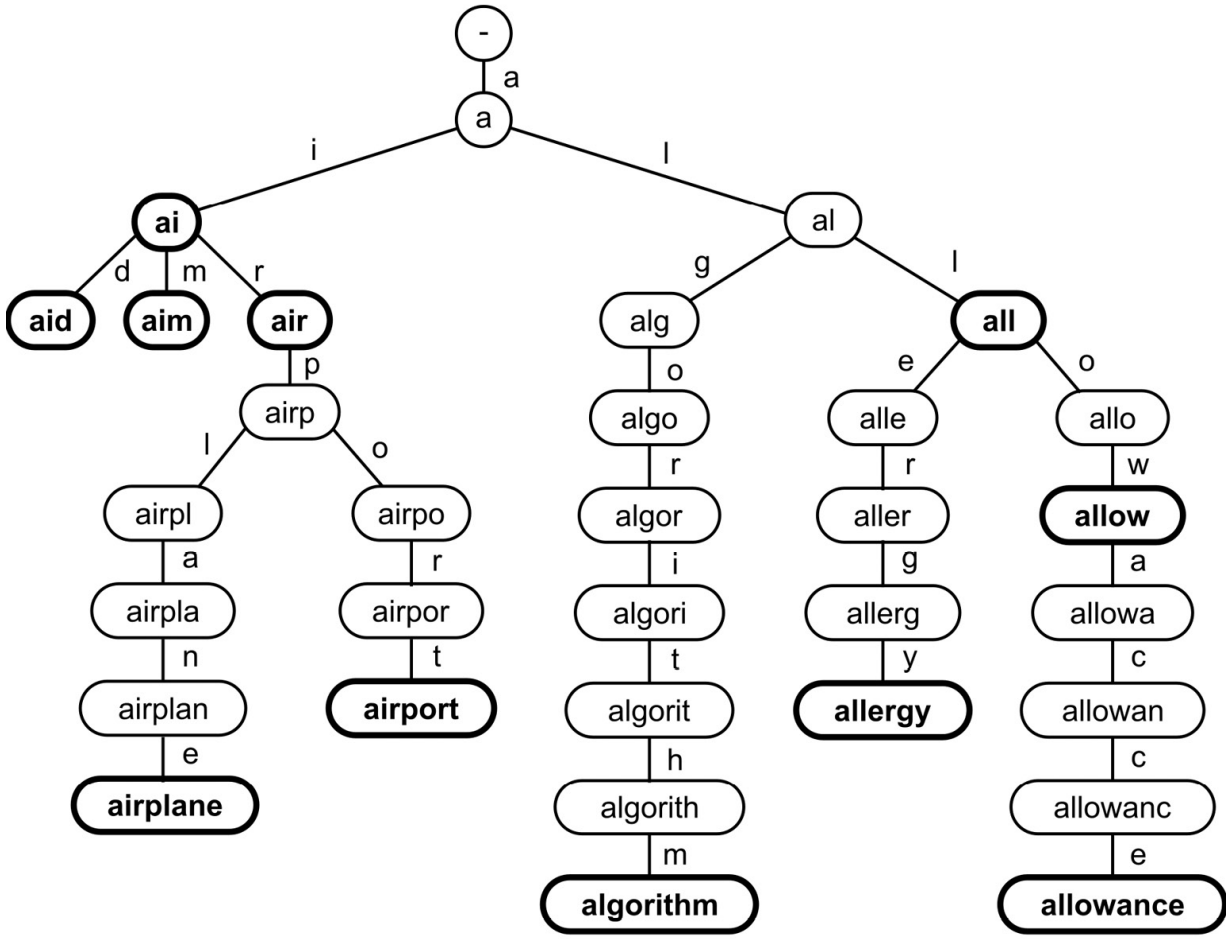

Figure 10.10 – Illustration of a trie

A **heap** is another subtype of a tree and exists in many variants, including a **binary heap**. It contains two versions – namely, **min-heap** and **max-heap**. For each of them, the additional property must be satisfied. For the min-heap, the value of each node must be greater than or equal to the value of its parent node. Thus, the root node contains the smallest value. For the max-heap, the value of each node must be less than or equal to the value of its parent node. Therefore, the root node always contains the largest value. The exemplary binary heaps are shown as follows:

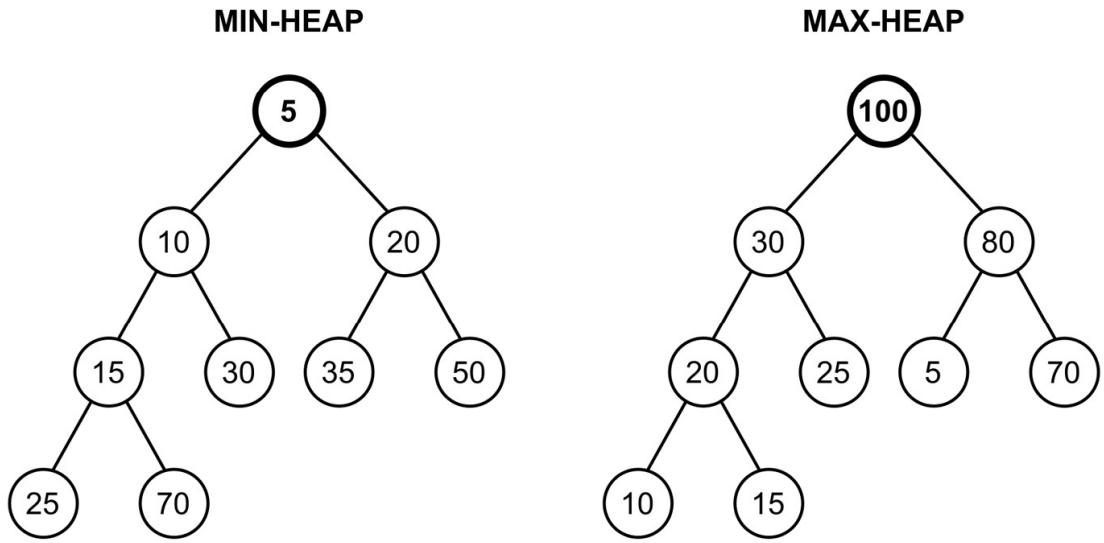

Figure 10.11 – Illustration of a min-heap and max-heap

A heap is a convenient data structure for implementing a **priority queue**. Another interesting application is the sorting algorithm, named **heap sort**, which was presented and explained in the chapter regarding arrays and sorting.

Graphs

Chapter 8, *Exploring Graphs*, was related to **graphs** – a very popular data structure with a broad range of applications. As a reminder, a graph is a data structure that consists of **nodes** and **edges**. Each edge connects two nodes. There are a few variants of edges in a graph, such as undirected and directed, as well as unweighted and weighted. A graph can be represented as an adjacency list or as an adjacency matrix.

All of these topics were described in the book, together with the problem of graph **traversal** with breadth-first search and depth-first search algorithms, finding the **minimum spanning tree** with Kruskal's and Prim's algorithms, **node coloring**, and **finding the shortest path** in a graph with Dijkstra's algorithm.

The exemplary graphs are shown in the following illustration:

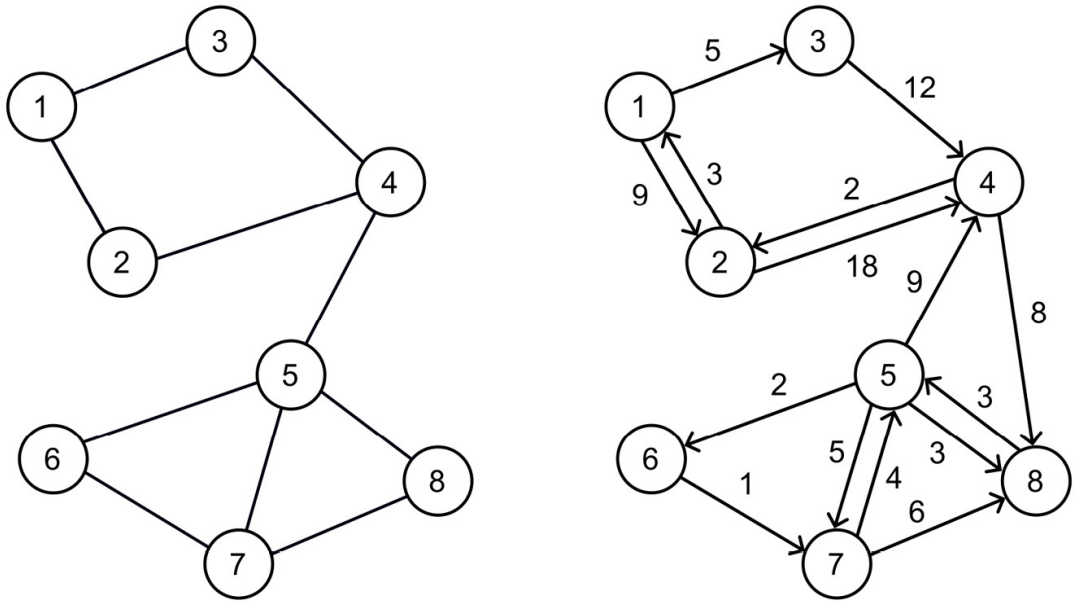

Figure 10.12 – Illustration of graphs

A graph data structure is commonly used in various applications. It is also a great way to represent diverse data, such as the **structure of friends** available on a social media site. Here, the nodes can represent contacts, while edges represent relationships between people. Thus, you can easily check whether two contacts know each other or how many people should be involved to arrange a meeting between two particular people.

Another common application of graphs involves the problem of **finding a path**. As an example, you can use a graph to find a path between two points in the city, taking into account the distance or time necessary for driving. You can use a graph to represent a map of a city, where nodes are intersections and edges represent roads. You can assign weights to edges to indicate the necessary distance or time for driving a given road.

There are many other applications related to graphs. For instance, the minimum spanning tree can be used to create a **plan of connections between buildings** to supply all of them with a telecommunication cable at the smallest cost. The node coloring problem was used in the book for **coloring voivodeships** on a map of Poland according to the rule that two voivodeships that have common borders cannot have the same color. Another shown example involves Dijkstra's algorithm for finding **the shortest path in a game map**, taking into account various obstacles.

The last word

You just reached the end of the last chapter of the book. First, the classification of data structures was presented, taking into account linear and non-linear data structures. In the first group, you can find arrays, lists, stacks, and queues, while the second group involves graphs and their subtypes, including trees and heaps. In the following part of this chapter, the diversity of applications of various data structures was taken into account. You saw a short summary of each described data structure, as well as information about some problems that can be solved with the use of a particular data structure, such as a queue or a graph. To make the content easier to understand, as well as to remind you of the various topics from the previous chapters, the summary was equipped with brief descriptions and illustrations of data structures.

In the introduction to this book, I invited you to start your adventure with data structures and algorithms. While reading the following chapters, writing hundreds of lines of code, and debugging, you had a chance to familiarize yourself with various data structures, starting with arrays and lists, through stacks, queues, dictionaries, and sets, and ending with trees and graphs. I hope that this book is only the first step in your long, challenging, and successful adventure with data structures and algorithms.

I would like to thank you for reading this book. If you have any questions or problems regarding the described content, please do not hesitate to contact me directly using the contact information shown at https://marcin.com. While visiting my website, you can also find answers to many questions that you can ask during your development career. Please also tell me what topics are missing from this book that you want to learn about for the next edition of this book or from another of my books. I really hope that you will benefit from the presented content. I would like to wish you all the best in your career as a software developer, and I hope that you have many successful projects! I will be very happy if you let me know about your great projects, especially if they are inspired by the content of this book. Good luck and keep in touch!

Index

As this ebook edition doesn't have fixed pagination, the page numbers below are hyperlinked for reference only, based on the printed edition of this book.

Symbols

.NET-based console applications 3-5

A

access modifiers 22

accessors 23

adaptive heuristic search algorithms 47

adjacency list 249-251

adjacency matrix 252-254

  symmetric adjacency matrix 253

algorithm representation notation 38, 39

  flowchart 39-41

  programming language 44

  pseudocode 42, 43

algorithms 36

  defining 36

  example 37, 38

algorithm types 44

  back-tracking algorithm 45, 46

  brute-force algorithm 47

  divide and conquer algorithm 45

  dynamic programming 47

  greedy algorithm 46

heuristic algorithm 46
recursive algorithm 44
application programming interfaces (APIs) 30
arithmetic mean 39
array list 96-98
reference link 99
array of arrays 64, 330
arrays 330, 331
art gallery 119-121
artificial intelligence (AI) 37
ASP.NET Core 2
asymptotic analysis 48
auto-implemented properties 23
average value 102
AVL tree 227

B

back-tracking algorithm 45, 46, 313
balanced tree 226
basic tree 191
company structure 195-197
hierarchy of identifiers 194, 195
implementation 193
Node class 193
nodes 192
root 192
Tree class 194
bidirectional edges 245

big data 38

Big-O notation 48

binary heap 85, 87

binary search tree (BST) 191, 208-211, 338

duplicates, adding 208

implementation 211

item, removing 214-218

lookup 211, 212

node, inserting 212-214

reference link 217

tree 211

visualization, example 218-225

binary trees 191, 197, 198

example 205-208

node 202

traversal 198, 199

tree 202-204

bit field 13

book reader

example 109-111

Boolean values 10

bottom-up approach 47

boxing 20

breadth-depth ratio

reference link 226

breadth-first search (BFS) 266-269

brute-force algorithm 47

example 48

bubble sort algorithm 74, 76

built-in generic class 149

C

C#

using, as programming language 2, 3

C# 12 2

chief executive officer (CEO) 192

chief financial officer (CFO) 192

chief marketing officer (CMO) 192

chief operating officer (COO) 192

chief technology officer (CTO) 192

child node 192

circular doubly linked list 118, 119

art gallery 119-121

circular queue 155-159

gravity roller coaster 159-162

performance 157

circular singly linked list 112-114

spin the wheel 115 - 117

classes 22-25

closest pair of points

finding 303-307

composite format string 21

compressed trie 235

computational complexity 48, 72

space complexity 49

time complexity 48, 49

constants 11

constant time 49

constant values 11

continuous integration and continuous delivery (CI/CD) 6

cubic time 49

D

data structures classification 328-330

linear data structure 328

non-linear data structure 328

data types 7

reference types 6

value types 6

deep learning (DL) 38

delegates 29, 30

depth-first (DFS) 262-265

dictionaries 170-172, 335

product location 173, 174

user details 175, 176

Dijkstra's algorithm 290

auxiliary node-related arrays 290

directed edges 245

directed graph 245

discard pattern 22

discards 27

divide and conquer algorithm 45

double logarithmic time 49

doubly linked list 107, 108

book reader 109-111

methods 108

dynamic programming 47

bottom-up approach 47

top-down approach 47

dynamics 30, 31

E

enumerations 12-14

equals operator (==) 7

exception-handling statements 3

exponential time 49

expression body definition 24

extension methods 55

F

factorial time 49

Fibonacci series 300-302

FIFO principle 136

floating-point numbers 9

flowchart 39-41

four-color theorem 284

four-dimensional array 60

fractal generation 307-311

fractional power time 49

G

garbage collection 3

generic list 99 - 102

average value 102

people list 103, 104

reference link 99

genetic algorithms 47

graph coloring 284-287

voivodeship map 287-290

graph implementation 254

directed and weighted edges 261, 262

Edge 255

Graph 256-259

Node 254, 255

undirected and unweighted edges 260, 261

graph representations

adjacency list 249-252

adjacency matrix 252-254

graphs 244-247, 340-341

applications 247-249

cycles 244

edges 244

nodes (vertices) 244

representations 249

greedy algorithm 46, 280

example 46

H

hash collision 166

hash function 166

hash map 165

hash set 180-183

coupons 183, 184

swimming pools 185 - 187

hash table 165-168

phone book 68-170

hash table-related classes

generic 167

non-generic 167

heapify operation 85

heaps 239

max-heap 239

min-heap 239

heap sort algorithm 85-89

heuristic algorithm 46

h-sorting 80

I

immutable value 11

index 52

in-order approach 201

insertion sort algorithm 73, 74

integer value 8

integral numbers 8

modes 9

integral numeric types 8

integrated development environment (IDE) 3

interfaces 27 - 29

interpolated expressions 21

interpolated string 21

intersection 181

iteration statements 2

iterative algorithms 44

J

jagged arrays 64-66

example 65

yearly transport plan 66-69

jump statements 3

K

Kruskal's algorithm 271-276

L

Lambda body 30

Language Integrated Query (LINQ) 3

leaf 192

left child 198

LIFO principle 126

limited access data structures 125

linear data structure 328

linearithmic time 49

linear time 49

linked list 106

circular doubly linked list 118, 119

circular singly linked list 112-114

doubly linked list 107, 108

singly linked list 106, 107

LinkedListNode class

methods 108

list-related interfaces 121, 122

lists 95, 331, 332

logarithmic time 49

loops 2

M

machine learning (ML) 38

max-heap 85-87

memoization 47, 301

merge sort algorithm 77-80

method syntax 104

Microsoft Visual Studio 2022 Community 3

min-heap 87

minimum coin change problem 302, 303

minimum spanning trees (MSTs) 243, 269-271

Kruskal's algorithm 271-276

Prim's algorithm 276-281

telecommunication cable 281-284

multi-dimensional arrays 58-61

game map 62-64

multiplication table 61, 62

N

named constants 12

natural language 39

neighbors 250

node 108

no-decision value 10

node coloring 284

non-destructive mutation 27

non-linear data structure 328

NuGet packages 149

installing 150

nullable reference types 32, 33

nullable value types 17-19

null-coalescing assignment operator 18

null-coalescing operator (??) 18

null conditional operator 19

null-state static analysis 32

O

object initializer syntax 17

objects 19

OptimizedPriorityQueue library 150

P

pagination 101

parallel computing 45

parent node 192

partitioning 82

password guess 322, 324

pattern match 2

people list 103, 104

polylogarithmic time 49

polynomial time 49

positional parameters 25

positional properties 25

prefix-based searching 233

Prim's algorithm 276-281

priority queue 148-240

call center with priority support 151-154

diagram 149

programming language 43, 44

project directory 5

pseudocode 42, 43

Q

quadratic time 49

query syntax 104

queue 136, 333, 334

call center with many consultants 143-147

call center with single consultant 139-142

circular queue 155-159

dequeue operation 136

enqueue operation 136

operations 136-138

performance 138

priority queue 148-151

recursive data structure 137

quicksort algorithm 82-85

R

radix tree 235

random access data structures 125

range operator 111

rat in a maze puzzle 312-314

records 25-27

recursive algorithms 44

recursive data structure 137

red-black trees (RBTs) 191, 228-230

reference types 7, 19

  classes 22-25

  delegates 29, 30

  dynamics 30, 31

  interfaces 28, 29

  nullable reference types 32, 33

  objects 20

  records 25-27

  strings 20-22

relational patterns 22

right child 198

ring buffer 155, 334

S

selection sort algorithm 70-72

selection statements 2

self-balancing tree 191, 226, 227,338

  AVL tree 227

  RBTs 228

self-loops 245

set 180, 335, 336

Shell sort algorithm 80-82

shortest path 290 - 294

  example 294-297

siblings 336

simple list 95, 96

  array list 96-98

  generic list 99-102

simple types 7

single-dimensional arrays 52-56

  example 52

  month names 56, 57

singly linked list 106, 107

sorted dictionaries 176-178

  encyclopedia 178-180

sorted list 104, 105

  address book 105, 106

sorted sets 188

  duplicates, removing 189

sorting algorithm 69, 70

  bubble sort 74-77

  heap sort 85-89

  insertion sort 73, 74

  merge sort 77-80

  performance analysis 89 - 94

  quicksort 82-85

selection sort 70-72
shell sort 80-82
space complexity 49
spanning tree 269
spin the wheel
example 115-117
stack 125, 333
benefits 126
performance 127
push and pop operations diagram 126
reversing word 127, 128
Tower of Hanoi 128-135
static type checking 31
strings 20, 21, 22
structs 16
subtraction 182
subtree 192
Sudoku puzzle 314-317
switch expression 22
symmetric difference 182

T

tabulation 47
task-based asynchronous pattern 31
three-dimensional array 59
three-valued Boolean logic 10
time complexity 48, 49
title guess 318-321

top-down approach 47

top-level statements 5

Tower of Hanoi 128-135

moving discs problem 129

traversal

BFS 266-269

DFS 262-265

directed and weighted edges 262

traversal, binary trees

in-order approach 201

post-order approach 201, 202

pre-order approach 199, 200

trees 336-340

trie 191, 230, 231

autocomplete 236-239

class, implementing 232-236

implementation 232

node 232

two-dimensional array 58

U

unbalanced tree 226

unboxing 20

undirected edges 245

undirected graph 245

Unicode characters 10, 11

unified type system 19

union 181

Unity 2

unweighted edge 246

unweighted graph 246

user-defined structs 16, 17

V

value tuples 14, 15

value type records 25

value types 6, 7

  Boolean values 10

  constants 8, 11

  enumerations 8, 12-14

  floating-point numbers 9

  integral numbers 8

  nullable value types 17-19

  structs 7

  Unicode characters 10, 11

  user-defined structs 16, 17

  value tuples 14, 15

version control system

  Git 6

W

warnings 311

weighted edge 246

weighted graph 246

weights (costs) 246

with expression 17

X

XAML 2

www.packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Refactoring with C#

Matt Eland

ISBN: 978-1-83508-998-9

- Understand technical debt, its causes and effects, and ways to prevent it
- Explore different ways of refactoring classes, methods, and lines of code
- Discover how to write effective unit tests supported by libraries such as Moq
- Understand SOLID principles and factors that lead to maintainable code
- Use AI to analyze, improve, and test code with the GitHub Copilot Chat
- Apply code analysis and custom Roslyn analyzers to ensure that code stays clean
- Communicate tech debt and code standards successfully in agile teams

Clean Code with C#

Jason Alls

ISBN: 978-1-83763-519-1

- Master the art of writing evolvable and adaptable code
- Implement the fail-pass-refactor methodology using a sample C# console application
- Develop custom C# exceptions that provide meaningful information
- Identify low-quality C# code in need of refactoring
- Improve code performance using profiling and refactoring tools
- Create efficient and bug-free code using functional programming techniques
- Write cross-platform code using MAUI
- Develop cloud-deployable microservices for versatile applications

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *C# Data Structures and Algorithms*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/978-1-80324-827-1

2. Submit your proof of purchase

3. That's it! We'll send your free PDF and other benefits to your email directly