Erik Azar, Mario Eguiluz Alebicto

Swift Data Structure and Algorithms

Master the most common algorithms and data structures, and learn how to implement them efficiently using the most up-to-date features of Swift 3

Packt>

Swift Data Structure and Algorithms

Master the most common algorithms and data structures, and learn how to implement them efficiently using the most up-to-date features of Swift 3

Erik Azar
Mario Eguiluz Alebicto

BIRMINGHAM - MUMBAI

Swift Data Structure and Algorithms

Copyright © 2016 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 authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

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

First published: November 2016

Production reference: 1111116

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78588-450-4

www.packtpub.com

Credits

Authors

Erik Azar

Mario Eguiluz Alebicto

Reviewer

Doug Sparling

Commissioning Editor

Kunal Parikh

Acquisition Editor

Shweta Pant

Content Development Editor

Divij Kotian

Technical Editor

Prashant Mishra

Copy Editor

Safis Editing

Project Coordinator

Ritika Manoj

Proofreader

Safis Editing

Indexer

Rekha Nair

Graphics

Jason Monteiro

Production Coordinator

Shraddha Falebhai

About the Authors

Erik Azar is a computer scientist with over 20 years of professional experience of architecting and developing scalable, high-performance desktop, web, and mobile applications in the areas of network engineering, system management and security, and enterprise business services, having worked in diverse positions in companies ranging from startups to Fortune 500 companies. He has been developing applications on macOS and iOS since attending his first WWDC in 2007, when Apple announced the initial iPhone.

Erik is an expert developer and architect for Availity, LLC, based in Jacksonville, Florida, where he works with teams to deliver software solutions in the healthcare industry.

Erik has performed technical reviews for several Packt Publishing books on Java RESTful APIs and security, enjoying the experience so much he decided to write his first book for Packt Publishing.

When Erik is not being a geek, he enjoys spending time with his wife, Rebecca, and his three kids, and getting out to ride his motorcycle up and down the Florida coast.

I want to thank my children, Patrick, Kyra, and Cassandra; my parents; and especially my wife, Rebecca, for their support and encouragement while writing this book. I'd also like to thank Michael Privat and Robert Warner for their support, encouragement, and guidance on this project as well. Lastly, I want to thank Mario, Divij, Prashant, and our technical reviewers for all of their hard work and guidance working on this book. It's been a great experience working with all of you.

Mario Eguiluz Alebicto is a software engineer with over 10 years of experience in development. He started developing software with Java, switched later to Objective-C when the first iPhone delighted the world, and now he is also working with Swift. He founded his own startup to develop mobile applications for small companies and local shops. He has developed apps for different Fortune 500 companies and also for new disrupting startups since 2011. Now, he is working as a contractor in mobile applications, while writing technical and teaching materials whenever possible.

Apart from software development, Mario loves to travel, learn new things, play sports, and has considered himself a hardcore gamer since he was a child.

I want to thank my mother and sister for their love and unconditional support. Borja, you helped me so much when I needed it. Gloria, thanks for keeping me positive beyond my limits. Also want to thank Divij, Erik, and the entire team for their guidance and work on this book. You guys are awesome!

About the Reviewers

Doug Sparling works as a technical architect and software developer for Andrews McMeel Universal, a publishing and syndication company in Kansas City, MO. At AMU, he uses Go for web services, Python for backend services, Ruby on Rails and WordPress for website development, and Objective-C, Swift, and Java for native iOS and Android development. AMU's sites include `www.gocomics.com`, `www.uexpress.com`, `www.puzzlesociety.com`, and `www.dilbert.com`.

He also was the co-author of a Perl book, Instant Perl Modules, for McGraw-Hill, and a reviewer for other Packt Publishing books, including jQuery 2.0 Animation Techniques: Beginner's Guide and WordPress Web Application Development. Doug has also played various roles for Manning Publications as a reviewer, technical development editor, and proofer, working on books such as Go in Action, The Well-Grounded Rubyist 2nd Edition, iOS Development with Swift, and Programming for Musicians and Digital Artists.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

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

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

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

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

Table of Contents

Preface

This book aims to teach experienced developers how to leverage the latest Swift language features. With Swift, Apple's new programming language for macOS, iOS, watchOS, tvOS, and Linux, you can write software that is fast and helps promote safer coding practices. With Swift and Xcode playgrounds, Apple has made it easy for developers to learn about the best practices and new programming concepts. Apple has open sourced the Swift language, that is, they have made it available on a wide range of platforms now, not just the Apple ecosystem. By doing this, developers are now able to develop server-side code on multiple platforms, in addition to developing code for the traditional line of Apple products.

Today, so many consumers are dependent on their smartphones and Internet access, the effect being the amount of data is growing exponentially, and being able to process, sort, and search that data as quickly as possible is more important than ever. Understanding how data structures and algorithms affect the efficiency of processing huge amounts of data is the key to any successful application or software library.

You will learn about the important Swift features and the most relevant data structures and algorithms, using a hands-on approach with code samples in Xcode Playgrounds for each data structure and algorithm covered in this book. We teach you factors to consider when selecting one type or method over another. You also learn how to measure the performance of your code using asymptotic analysis, a concept commonly used in the software industry in order to choose the best algorithm and data structure for a particular use case.

With knowledge learned in this book, you will have the best tools available to help you develop efficient and scalable software for your applications.

We hope you enjoy reading the book and learning more about some of the advanced Swift features, while also understanding how slight modifications to your existing code can dramatically improve the performance of your applications.

What this book covers

`Chapter 1`, *Walking Across the Playground*, contains an introduction to data structures and algorithms, the Swift REPL, and how to enter Swift statements into it to produce results on the fly.

`Chapter 2`, *Working with Commonly Used Data Structures*, covers classes and structures, the implementation details for the array, dictionary, and set collection types, how Swift interoperates with Objective-C and the C system libraries, and protocol-oriented programming introduction.

`Chapter 3`, *Standing on the Shoulders of Giants*, covers how to conform to Swift protocols, how to implement a stack and queue structure, and implement several types so you can gain experience for choosing the right type based on the requirements of your application.

`Chapter 4`, *Sorting Algorithms*, covers algorithms, sorting algorithms and how to apply them using an array data structure, explore different algorithms that use comparison sorting and look at both simple sorting and divide-and-conquer strategies.

`Chapter 5`, *Seeing the Forest through the Tree*, explains the tree data structure, including a definition and its properties, an overview of different types of trees, such as binary trees, binary search trees (BST), B–trees, and splay trees with implementation details.

`Chapter 6`, *Advanced Searching Methods*, covers more advanced tree structures: red-black trees, AVL trees, Trie trees (Radix trees) and covers several Substring search algorithms.

`Chapter 7`, *Graph Algorithms*, explains graph theory and data structures for graphs, as well as depth-first search, breadth-first search, spanning tree, shortest path, and SwiftGraph.

`Chapter 8`, *Performance and Algorithm Efficiency*, shows you algorithm efficiency and how to measure it, Big-O notation, orders of common functions, and evaluating runtime complexity.

`Chapter 9`, *Choosing the Perfect Algorithm*, learn how to deal with problems that require algorithms and data structures by creating a high level solution, writing the implementation in Swift, calculating Big-O complexities of our solution to check if the algorithm behaves properly in a real-world situation, measuring and detecting bottlenecks, and modifying the solution to achieve better performance.

What you need for this book

The basic requirements for this book are as follows:

- Xcode, at least v8.1
- macOS Sierra 10.12 or OS X El Capitan 10.11.5 or later

Who this book is for

Swift Data Structures and Algorithms is intended for developers who want to learn how to implement and use common data structures and algorithms natively in Swift. Whether you are a self-taught developer without a formal technical background or you have a degree in Computer Science, this book will provide with the knowledge you need to develop advanced data structures and algorithms in Swift using the latest language features. An emphasis is placed on resource usage to ensure the code will run on a range of platforms from mobile to server. A previous background in an object-oriented language is helpful, but not required, as each concept starts with a basic introductory.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `initWith` portion of the name is removed from the method name."

A block of code is set as follows:

```
class MovieList {
    private var tracks = ["The Godfather", "The Dark Knight", "Pulp
    Fiction"]
    subscript(index: Int) -> String {
        get {
            return self.tracks[index]
        }
        set {
            self.tracks[index] = newValue
        }
    }
}
```

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

```
erik@iMac ~ swift
Welcome to Apple Swift version 3.0 (swiftlang-800.0.46.2 clang-
800.0.38). Type :help for assistance.
  1>
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In Xcode, go to **File** | **New** | **Playground**, and call it `B05101_6_RedBlackTree`."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the Search box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Swift-Data-Structure-and-Algorithms`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

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

Errata

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

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

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

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

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

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Walking Across the Playground

Swift is a powerful new programming language from Apple for **macOS**, **iOS**, **watchOS**, and **tvOS**. It has been rapidly climbing in popularity since its release at Apple's **WWDC 2014**, and within a year already broke through as one of the top 20 languages, placing at number 18, based on GitHub usage (`http://githut.info/`) and stack overflow discussions. In this book, we are going to look at the core data structures and algorithms provided in the Swift standard library. We will also look at native Swift implementations for other commonly used data structures and algorithms, such as queues, stacks, lists, and hash tables. Next, we'll look at sorting algorithms and compare the performance characteristics of different algorithms, as well as how the input size effects performance. We will move on to native Swift implementations for various tree data structures and algorithms, and advanced search methods. We then close the book by looking at implementations of various graphing algorithms and the approaches for calculating performance and algorithm efficiency.

In this chapter, we will cover what data structures and algorithms are and why they are so important. Selecting the correct data structure and algorithm for a particular problem could mean either success or failure for your application; and potentially to the long-term success of your product or company.

We are going to start off by discussing the importance of data structures and why it will benefit you to have knowledge of the differences between them. We will then move on to some concrete examples of the fundamental data structures. Next, we will review some of the most advanced data structures that are built on top of the fundamental types. Once that base has been set, we will get to experiment with a few data structures using the Swift **Read-Eval-Print-Loop** (**REPL**), which we'll talk about shortly. Finally, we will wrap up this chapter by introducing the topic of algorithm performance so you can begin thinking about the trade-offs between the different data structures and algorithms we will discuss later on in this book.

What is the importance of data structures?

Data structures are the building blocks that allow you to develop efficient, scalable, and maintainable systems. They provide a means of organizing and representing data that needs to be shared, persisted, sorted, and searched.

There's a famous saying coined by the British computer scientist David Wheeler:

> *"All problems in computer science can be solved by another level of indirection…"*

In software engineering, we use this level of indirection to allow us to build abstract frameworks and libraries. Regardless of the type of system that you are developing, whether it be a small application running on an embedded microcontroller, a mobile application, or a large enterprise web application, these applications are all based on data. Most modern application developments use APIs from various frameworks and libraries to help them create amazing new products. At the end of the day, these APIs, which provide a level of abstraction, boil down to their use of data structures and algorithms.

Data structures + algorithms = programs

Data abstraction is a technique for managing complexity. We use data abstraction when designing our data structures because it hides the internal implementation from the developer. It allows the developer to focus on the interface that is provided by the algorithm, which works with the implementation of the data structure internally.

Data structures and algorithms are patterns used for solving problems. When used correctly they allow you to create elegant solutions to some very difficult problems.

In this day and age, when you use library functions for 90% of your coding, why should you bother to learn their implementations? Without a firm technical understanding, you may not understand the trade-offs between the different types and when to use one over another, and this will eventually cause you problems.

> *"Smart data structures and dumb code works a lot better than the other way around."*
> *– Eric S. Raymond, The Cathedral and The Bazaar*

By developing a broad and deep knowledge of data structures and algorithms, you'll be able to spot patterns to problems that would otherwise be difficult to model. As you become experienced in identifying these patterns you begin seeing applications for their use in your day-to-day development tasks.

We will make use of Playgrounds and the Swift REPL in this section as we begin to learn about data structures in Swift.

Interactive Playgrounds

Xcode 8.1 has added many new features to Playgrounds and updated it to work with the latest syntax for **Swift 3.0**. We will use Playgrounds as we begin experimenting with different algorithms so we can rapidly modify the code and see how changes appear instantly.

The Swift REPL

We are going to use the Swift compiler from the command-line interface known as the Read-Eval-Print-Loop, or REPL. Developers who are familiar with interpretive languages such as Ruby or Python will feel comfortable in the command-line environment. All you need to do is enter Swift statements, which the compiler will execute and evaluate immediately. To get started, launch the `Terminal.app` in the `/Applications/Utilities` folder and type `swift` from the prompt in macOS Sierra or OS X El Capitan. Alternatively, it can also be launched by typing `xcrun swift`. You will then be in the REPL:

```
erik@iMac ~ swift
Welcome to Apple Swift version 3.0 (swiftlang-800.0.46.2 clang-
800.0.38). Type :help for assistance.
  1>
```

Statement results are automatically formatted and displayed with their type, as are the results of their variables and constant values:

```
erik@iMac ~ swift
Welcome to Apple Swift version 3.0 (swiftlang-800.0.46.2 clang-
800.0.38). Type :help for assistance.
  1> var firstName = "Kyra"
firstName: String = "Kyra"

  2> print("Hello, \(firstName)")
Hello, Kyra

  3> let lastName: String = "Smith"
lastName: String = "Smith"

  4> Int("2000")
$R0: Int? = 2000
```

Note that the results from line four have been given the name `$R0` by the REPL even though the result of the expression wasn't explicitly assigned to anything. This is so you can reference these results to reuse their values in subsequent statements:

```
  5> $R0! + 500
$R1: Int = 2500
```

The following table will come in handy as you learn to use the REPL; these are some of the most frequently used commands for editing and navigating the cursor:

Table 1.1 – Quick Reference

Keys	**Actions**
Arrow keys	Move the cursor left/right/up/down.
Control + F	Move the cursor right one character, same as the right arrow.
Control + B	Move the cursor left one character, same as the left arrow.
Control + N	Move the cursor to the end of the next line, same as the down arrow.
Control + P	Move the cursor to the end of the prior line, same as the up arrow.
Control + D	Delete the character under the cursor.
Option + Left	Move the cursor to the start of the prior word.
Option + Right	Move the cursor to the start of the next word.
Control + A	Move the cursor to the start of the current line.
Control + E	Move the cursor to the end of the current line.
Delete	Delete the character to the left of the cursor.
Esc <	Move the cursor to the start of the first line.
Esc >	Move the cursor to the end of the last line.
Tab	Automatically suggest variables, functions, and methods within the current context. For example, after typing the dot operator on a string variable you'll see a list of available functions and methods.

Fundamental data structures

As we discussed previously, you need to have a firm understanding of the strengths and weaknesses of the different data structures. In this section, we'll provide an overview of some of the main data structures that are the building blocks for more advanced structures that we'll cover in this book.

There are two fundamental types of data structures, which are classified based on arrays and pointers, respectively as:

- **Contiguous data structures**, as their name imply, it means storing data in contiguous or adjoining sectors of memory. These are a few examples: arrays, heaps, matrices, and hash tables.
- **Linked data structures** are composed of distinct sectors of memory that are bound together by pointers. Examples include lists, trees, and graphs.

You can even combine these two types together to create advanced data structures.

Contiguous data structures

The first data structures we will explore are contiguous data structures. These linear data structures are index-based, where each element is accessed sequentially in a particular order.

Arrays

The array data structure is the most well-known data storage structure and it is built into most programming languages, including Swift. The simplest type is the linear array, also known as a one-dimensional array. In Swift, arrays are a zero-based index, with an ordered, random-access collection of elements of a given type.

For one-dimensional arrays, the index notation allows indication of the elements by simply writing *ai*, where the index *i* is known to go from *0* to *n*:

$$a = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix}$$

For example, give the array:

$$a = (3\ 5\ 7\ 9\ 13)$$

Some of the entries are:

$$a_0 = 3,\ a_1 = 5,\ \ldots,\ a_4 = 13$$

Another form of an array is the multidimensional array. A matrix is an example of a multidimensional array that is implemented as a two-dimensional array. The index notation allows indication of the elements by writing *aij*, where the indexes denote an element in row *i* and column *j*:

$$a = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

Given the matrix:

$$a = \begin{pmatrix} 2 & 5 & 7 \\ 10 & 3 & 5 \\ 4 & 1 & 4 \end{pmatrix}$$

Some of the entries are:

$$a_{00} = 2,\ a_{11} = 3,\ \ldots,\ a_{22} = 4$$

Declaring an array

There are three forms of syntax in Swift for declaring an array: the full method that uses the `Array<Type>` form, the shorthand method that uses the square bracket `[Type]` form, and type inference. The first two are similar to how you would declare variables and constants. For the remainder of this book, we'll use the shorthand syntax.

To declare an array using the full method, use the following code:

```
var myIntArray: Array<Int> = [1,3,5,7,9]
```

To declare an array using the shorthand array syntax, use the following code:

```
var myIntArray: [Int] = [1,3,5,7,9]
```

To declare an array using the type inference syntax, use the following code:

```
var myIntArray = [1,3,5,7,9]
```

Type inference is a feature that allows the compiler to determine the type at compile time based on the value you provide. Type inference will save you some typing if you declare a variable with an initial type.

If you do not want to define any values at the time of declaration, use the following code:

```
var myIntArray: [Int] = []
```

To declare a multidimensional array use nesting pairs of square brackets. The name of the base type of the element is contained in the innermost pair of square brackets:

```
var my2DArray: [[Int]] = [[1,2], [10,11], [20, 30]]
```

You can create beyond two dimensions by continuing to nest the type in square brackets. We'll leave that as an exercise for you to explore.

Retrieving elements

There are multiple ways to retrieve values from an array. If you know the elements index, you can address it directly. Sometimes you may want to loop through, or iterate through the collection of elements in an array. We'll use the `for...in` syntax for that. There are other times when you may want to work with a subsequence of elements in an array; in this case we'll pass a range instead of an index to get the subsequence.

Directly retrieve an element using its index:

```
1> var myIntArray: [Int] = [1,3,5,7,9]
myIntArray: [Int] = 5 values {
  [0] = 1
  [1] = 3
  [2] = 5
  [3] = 7
  [4] = 9
}
  2> var someNumber = myIntArray[2]
someNumber: Int = 5
```

Iterating through the elements in an array:

```
1> var myIntArray: [Int] = [1,3,5,7,9]
myIntArray: [Int] = 5 values {
  [0] = 1
  [1] = 3
  [2] = 5
  [3] = 7
  [4] = 9
}
  2> for element in myIntArray {
  3.     print(element)
  4. }

1
3
5
7
9
```

Notice in the preceding examples that when we typed the `for` loop, after we hit *Enter,* on the new line instead of a > symbol we have a `.` and our text is indented. This is the REPL telling you that this code will only be evaluated inside of this code block.

Retrieving a subsequence of an array:

```
1> var myIntArray: [Int] = [1,3,5,7,9]
myIntArray: [Int] = 5 values {
  [0] = 1
  [1] = 3
  [2] = 5
  [3] = 7
  [4] = 9
}
2> var someSubset = myIntArray[2...4]
someSubset: ArraySlice<Int> = 3 values {
  [2] = 5
  [3] = 7
  [4] = 9
}
```

Directly retrieve an element from a two-dimensional array using its index:

```
1> var my2DArray: [[Int]] = [[1,2], [10,11], [20, 30]]
my2DArray: [[Int]] = 3 values {
  [0] = 2 values {
    [0] = 1
```

```
      [1] = 2
    }
    [1] = 2 values {
      [0] = 10
      [1] = 11
    }
    [2] = 2 values {
      [0] = 20
      [1] = 30
    }
}
  2> var element = my2DArray[0][0]
element: Int = 1
  3> element = my2DArray[1][1]
  4> print(element)
11
```

Adding elements

You can add elements to an array using several different methods, depending on whether you want to add an element to the end of an array or insert an element anywhere between the beginning and the end of the array.

Adding an element to the end of an existing array:

```
1> var myIntArray: [Int] = [1,3,5,7,9]
myIntArray: [Int] = 5 values {
  [0] = 1
  [1] = 3
  [2] = 5
  [3] = 7
  [4] = 9
}
  2> myIntArray.append(10)
  3> print(myIntArray)
[1, 3, 5, 7, 9, 10]
```

Inserting an element at a specific index in an existing array:

```
1> var myIntArray: [Int] = [1,3,5,7,9]
myIntArray: [Int] = 5 values {
  [0] = 1
  [1] = 3
  [2] = 5
  [3] = 7
  [4] = 9
}
```

```
  2> myIntArray.insert(4, at: 2)
  3> print(myIntArray)
[1, 3, 4, 5, 7, 9]
```

Removing elements

Similarly, you can remove elements from an array using several different methods, depending on whether you want to remove an element at the end of an array or remove an element anywhere between the beginning and end of the array.

Removing an element at the end of an existing array:

```
1> var myIntArray: [Int] = [1,3]
myIntArray: [Int] = 2 values {
  [0] = 1
  [1] = 3
}
  2> myIntArray.removeLast()
$R0: Int = 3
  3> print(myIntArray)
[1]
```

To remove an element at a specific index in an existing array:

```
1> var myIntArray: [Int] = [1,3,5,7,9]
myIntArray: [Int] = 5 values {
  [0] = 1
  [1] = 3
  [2] = 5
  [3] = 7
  [4] = 9
}
  2> myIntArray.remove(at: 3)
$R0: Int = 7
  3> print(myIntArray)
[1, 3, 5, 9]
```

Arrays are used to implement many other data structures, such as stacks, queues, heaps, hash tables, and strings, just to name a few.

Linked data structures

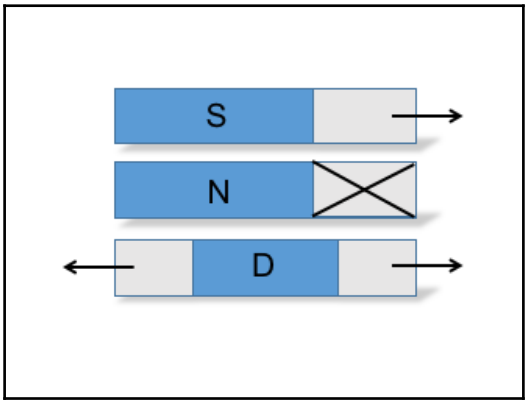
Linked structures are composed of a data type and bound together by pointers. A pointer represents the address of a location in the memory. Unlike other low-level programming languages such as C, where you have direct access to the pointer memory address, Swift, whenever possible, avoids giving you direct access to pointers. Instead, they are abstracted away from you.

We're going to look at linked lists in this section. A linked list consists of a sequence of nodes that are interconnected via their link field. In their simplest form, the node contains data and a reference (or link) to the next node in the sequence. More complex forms add additional links, so as to allow traversing forwards and backwards in the sequence. Additional nodes can easily be inserted or removed from the linked list.

Linked lists are made up of nodes, which are self-referential classes, where each node contains data and one or more links to the next node in the sequence.

In computer science, when you represent linked lists, arrows are used to depict references to other nodes in the sequence. Depending on whether you're representing a **singly** or **doubly** linked list, the number and direction of arrows will vary.

In the following example, nodes **S** and **D** have one or more arrows; these represent references to other nodes in the sequence. The **S** node represents a node in a singly linked list, where the arrow represents a link to the next node in the sequence. The **N** node represents a null reference and represents the end of a singly linked list. The **D** node represents a node that is in a doubly linked list, where the left arrow represents a link to the previous node, and the right arrow represents a link to the next node in the sequence.



Singly and doubly linked list data structures

We'll look at another linear data structure, this time implemented as a singly linked list.

Singly linked list

The linked list data structure is comprised of the four properties we defined previously, as shown in the following declaration:

```
class LinkedList<T> {
    var item: T?
    var next: LinkedList<T>?
}
```

We won't get into the full implementation of this class since we will cover a complete implementation in `Chapter 3`, *Standing on the Shoulders of Giants*.

Overview of data structures

The following is a table providing an overview of some of the most common and advanced data structures, along with their advantages and disadvantages:

Table 1.2 – Overview of Data Structures

Data Structure	**Advantages**	**Disadvantages**
Array	Very fast access to elements if index is known, fast insertion of new elements.	Fixed size, slow deletion, slow search.
Sorted array	Quicker search over non-sorted arrays.	Fixed size, slow insertion, slow deletion.
Queue	Provides FIFO (First In, First Out) access.	Slow access to other elements.
Stack	Provides LIFO (Last In, First Out).	Slow access to other elements.
List	Quick inserts and deletes.	Slow search.
Hash table	Very fast access if key is known, quick inserts.	Slow access if key is unknown, slow deletes, inefficient memory usage.
Heap	Very fast inserts and deletes, fast access to largest or smallest item.	Slow access to other items.

Trie (pronounced Try)	Very fast access, no collisions of different keys, very fast inserts and deletes. Useful for storing a dictionary of strings or doing prefix searches.	Can be slower than hash tables in some cases.
Binary tree	Very fast inserts, deletes, and searching (for balanced trees).	Deletion algorithm can be complex, tree shape depends on the order of inserts and can become degraded.
Red-black tree	Very fast inserts, deletes, and searching, tree always remains balanced.	Complex to implement because of all the operation edge conditions.
R-tree	Good for representing spatial data, can support more than two dimensions.	Does not guarantee good worst-case performance historically.
Graph	Models real-world situations.	Some algorithms are slow and complex.

Overview of algorithms

In studying algorithms, we often concern ourselves with ensuring their stingy use of resources. The time and space needed to solve a problem are the two most common resources we consider.

> *"Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output."*
> *– Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms 3rd Edition (2009)*

Specifically, we're interested in the asymptotic behavior of functions describing resource use in terms of some measure of problem size. We'll take a closer look at asymptotic behavior later in this chapter. This behavior is often used as a basis for comparison between methods, where we prefer methods whose resource use grows slowly as a function of the problem size. This means we should be able to solve larger problems quicker.

The algorithms we'll discuss in this book apply directly to specific data structures. For most data structures, we'll need to know how to:

- Insert new data items
- Delete data items
- Find a specific data item(s)
- Iterate over all data items
- Perform sorting on data items

Data types in Swift

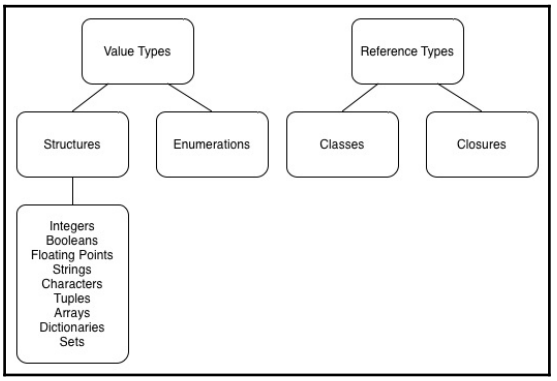
If you have programmed in other languages, such as C or its superset languages such as Objective-C or C++, you're probably familiar with the built-in primitive data types those languages provide. A primitive data type is generally a scalar type, which contains a single value. Examples of scalar data types are int, float, double, char, and bool. In Swift however, its primitive types are not implemented as scalar types. In this section, we'll discuss the fundamental types that Swift supports and how these are different from other popular languages.

Value types and reference types

Swift has two fundamental types: v**alue types** and r**eference types**. Value types are a type whose value is copied when it is assigned to a variable or constant, or when it is passed to a function. Value types have only one owner. Value types include structures and enumerations. All of Swift's basic types are implemented as structures.

Reference types, unlike value types, are not copied on assignment but are shared. Instead of a copy being made when assigning a variable or passing to a function, a reference to the same existing instance is used. Reference types have multiple owners.

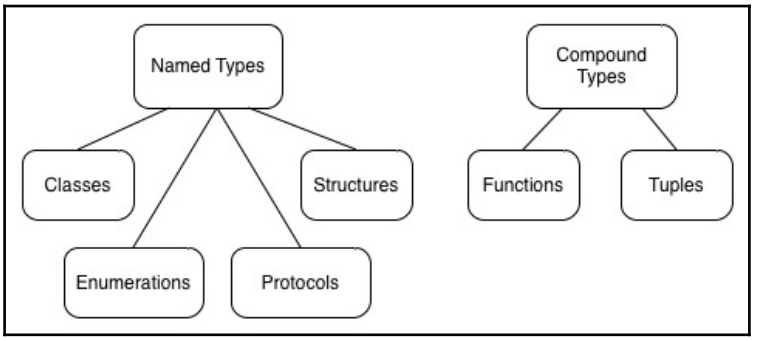
The Swift standard library defines many commonly used native types such as int, double, float, string, character, bool, array, dictionary, and set.



It's important to remember though that the preceding types, unlike in other languages, are not primitive types. They are actually **named types**, defined and implemented in the Swift standard library as structures. We'll discuss named types in the following section.

Named and compound types

In Swift, types are also classified as named types and **compound types**. A named type is a type that can be user-defined and given a particular name when it's defined. Named types include classes, structures, enumerations, and protocols. In addition to user-defined named types, the Swift standard library also includes named types that represent arrays, dictionaries, sets, and optional values. Named types can also have their behavior extended by using an extension declaration.



Compound types are types without a name, In Swift, the language defines two compound types: **function types** and **type types**. A compound type can contain both named types, and other compound types. As an example, the following tuple type contains two elements: the first is the named type `Int`, and the second is another compound type `(Float, Float)`:

```
(Int, (Float, Float))
```

Type aliases

Type aliases define an alternative name for existing types. The `typealias` keyword is similar to `typedef` in C-based languages. Type aliases are useful when working with existing types that you want to be more contextually appropriate to the domain you are working in. For example, the following associates the identifier `TCPPacket` with the type `UInt16`:

```
typealias TCPPacket = UInt16
```

Once you define a type alias you can use the alias anywhere you would use the original type:

```
1> typealias TCPPacket = UInt16
2> var maxTCPPacketSize = TCPPacket.max
maxTCPPacketSize: UInt16 = 65535
```

Collection types in the Swift standard library

Swift provides three types of collections: arrays, dictionaries, and sets. There is one additional type we'll also discuss, even though it's technically not a collection—tuples, which allow for the grouping of multiple values into a compound value. The values that are ordered can be of any type and do not have to be of the same type as each other. We'll look at these collection classes in depth in the next chapter.

Asymptotic analysis

When building a service, it's imperative that it finds information quickly, otherwise it could mean the difference between success or failure for your product. There is no single data structure or algorithm that offers optimal performance in all use cases. In order to know which is the best solution, we need to have a way to evaluate how long it takes an algorithm to run. Almost any algorithm is sufficiently efficient when running on a small number of inputs. When we're talking about measuring the cost or complexity of an algorithm, what we're really talking about is performing an analysis of the algorithm when the input sets are very large. Analyzing what happens as the number of inputs approaches infinity is referred to as asymptotic analysis. It allows us to answer questions such as:

- How much space is needed in the worst case?
- How long will an algorithm take to run with a given input size?
- Can the problem be solved?

For example, when analyzing the running time of a function that sorts a list of numbers, we're concerned with how long it takes as a function of the size of the input. As an example, when we compare sorting algorithms, we say the average insertion sort takes time $T(n)$, where $T(n) = c*n^2+K$ for some constants c and k, which represents a **quadratic running time**. Now compare that to merge sort, which takes time $T(n)$, where $T(n) = c*n*log_2(n)+k$ for some constants c and k, which represents a **linearithmic running time**.

We typically ignore smaller values of x since we're generally interested in estimating how slow an algorithm will be on larger data inputs. The asymptotic behavior of the merge sort function $f(x)$, such that $f(x) = c*x*log_2(x)+k$, refers to the growth of $f(x)$ as x gets larger.

Generally, the slower the asymptotic growth rate, the better the algorithm, although this is not a hard and fast rule. By this allowance, a linear algorithm, $f(x) = d*x+k$, is always asymptotically better than a linearithmic algorithm, $f(x) = c*x*log_2(x)+q$. This is because where c, d, k, and $q > 0$ there is always some x at which the magnitude of $c*x*log_2(x)+q$ overtakes $d*x+k$.

Order of growth

In estimating the running time for the preceding sort algorithms, we don't know what the constants *c* or *k* are. We know they are a constant of modest size, but other than that, it is not important. From our asymptotic analysis, we know that the log-linear merge sort is faster than the insertion sort, which is quadratic, even though their constants differ. We might not even be able to measure the constants directly because of CPU instruction sets and programming language differences. These estimates are usually only accurate up to a constant factor; for these reasons, we usually ignore constant factors in comparing asymptotic running times.

In computer science, **Big-O** is the most commonly used asymptotic notation for comparing functions, which also has a convenient notation for hiding the constant factor of an algorithm. Big-O compares functions expressing the upper bound of an algorithm's running time, often called the order of growth. It's a measure of the longest amount of time it could possibly take for an algorithm to complete. A function of a Big-O notation is determined by how it responds to different inputs. Will it be much slower if we have a list of 5,000 items to work with instead of a list of 500 items?

Let's visualize how insertion sort works before we look at the algorithm implementation.

Step 1	56	17	63	34	77	52	68	Assume first item is sorted
Step 2	17	56	63	34	77	52	68	17 < 56 so insert it
Step 3	17	56	63	34	77	52	68	63 > 56 so leave it where it is
Step 4	17	34	56	63	77	52	68	34 < 63; 34 < 56; 34 > 17 so insert it
Step 5	17	34	56	63	77	52	68	77 > 63 so leave it where it is
Step 6	17	34	52	56	63	77	68	52 < 77; 52 < 63; 52 < 56; 52 > 34 so insert it
Step 7	17	34	52	56	63	68	77	68 < 77; 68 > 63 so insert it

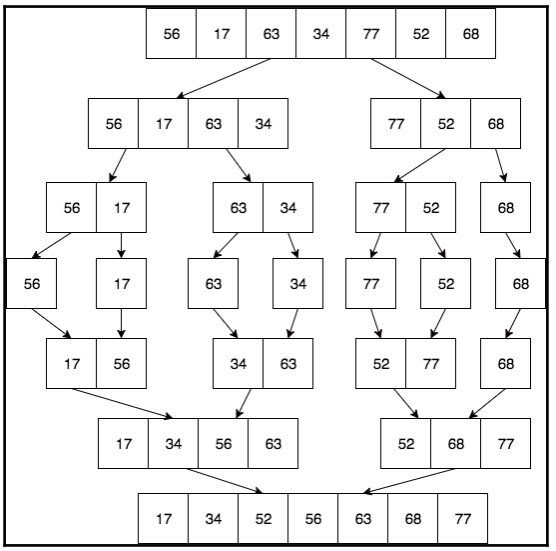
Given the list in Step 1, we assume the first item is in sorted order. In Step 2, we start at the second item and compare it to the previous item, if it is smaller we move the higher item to the right and insert the second item in first position and stop since we're at the beginning. We continue the same pattern in Step 3. In Step 4, it gets a little more interesting. We compare our current item, 34, to the previous one, 63. Since 34 is less than 63, we move 63 to the right. Next, we compare 34 to 56. Since it is also less, we move 56 to the right. Next, we compare 34 to 17, Since 17 is greater than 34, we insert 34 at the current position. We continue this pattern for the remaining steps.

Now let's consider an insertion sort algorithm:

```
func insertionSort( alist: inout [Int]){
    for i in 1..<alist.count {
        let tmp = alist[i]
        var j = i - 1
        while j >= 0 && alist[j] > tmp {
            alist[j+1] = alist[j]
            j = j - 1
        }
        alist[j+1] = tmp
    }
}
```

If we call this function with an array of 500, it should be pretty quick. Recall previously where we said the insertion sort represents a quadratic running time where $f(x) = c*n^2+q$. We can express the complexity of this function with the formula, $f(x) \epsilon O(x^2)$, which means that the function f grows no faster than the quadratic polynomial x^2, in the asymptotic sense. Often a Big-O notation is abused by making statements such as the complexity of $f(x)$ is $O(x^2)$. What we mean is that the worst case f will take $O(x^2)$ steps to run. There is a subtle difference between a function being in $O(x^2)$ and being $O(x^2)$, but it is important. Saying that $f(x) \epsilon O(x^2)$ does not preclude the worst-case running time of f from being considerably less than $O(x^2)$. When we say $f(x)$ is $O(x^2)$, we're implying that x^2 is both an upper and lower bound on the asymptotic worst-case running time.

Let's visualize how merge sort works before we look at the algorithm implementation:



In `Chapter 4`, *Sorting Algorithms*, we will take a detailed look at the merge sort algorithm so we'll just take a high-level view of it for now. The first thing we do is begin to divide our array, roughly in half depending on the number of elements. We continue to do this recursively until we have a list size of 1. Then we begin the combine phase by merging the sublists back into a single sorted list.

Now let's consider a merge sort algorithm:

```
func mergeSort<T:Comparable>(inout list:[T]) {
    if list.count <= 1 {
        return
    }
    func merge(var left:[T], var right:[T]) -> [T] {
        var result = [T]()
        while left.count != 0 && right.count != 0 {
            if left[0] <= right[0] {
                result.append(left.removeAtIndex(0))
            } else {
                result.append(right.removeAtIndex(0))
            }
        }
        while left.count != 0 {
            result.append(left.removeAtIndex(0))
        }
        while right.count != 0 {
```

```
                result.append(right.removeAtIndex(0))
            }
            return result
        }
        var left = [T]()
        var right = [T]()
        let mid = list.count / 2
        for i in 0..<mid {
            left.append(list[i])
        }
        for i in mid..<list.count {
            right.append(list[i])
        }
        mergeSort(&left)
        mergeSort(&right)
        list = merge(left, right: right)
    }
```

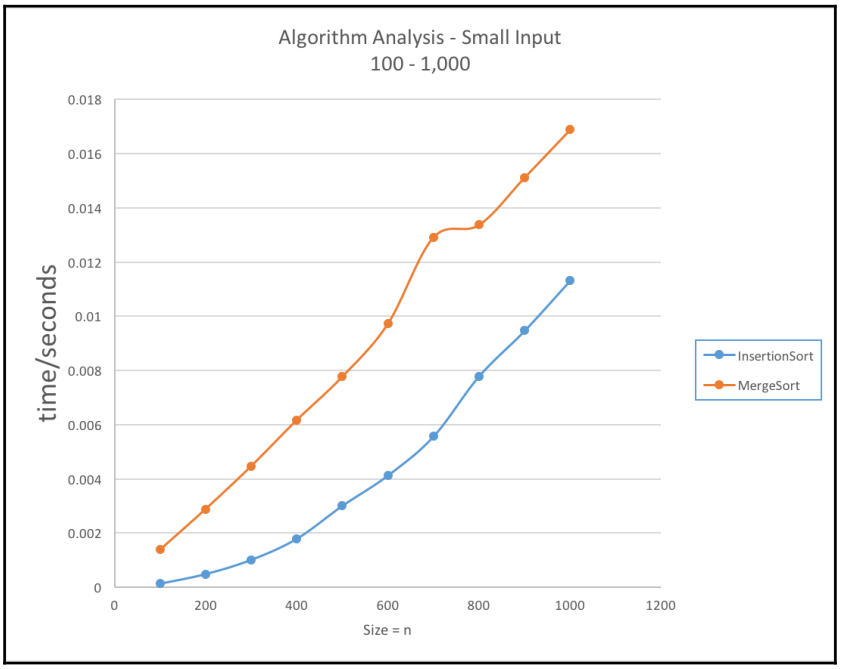
The source code for insertion sort and merge sort is provided as part of this book's source code download bundle. You can either run it in Xcode Playground or copy/paste it into the Swift REPL to experiment with it.

In Table 1.3, we can see with smaller sized inputs it appears at first glance that the insertion sort offers better performance over the merge sort:

Table 1.3 – Small Input Size: 100-1,000 items / seconds

n	T(n2)	Tm(n)
100	0.000126958	0.001385033
200	0.00048399	0.002885997
300	0.001008034	0.004469991
400	0.00178498	0.006169021
500	0.003000021	0.007772028
600	0.004121006	0.009727001
700	0.005564034	0.012910962
800	0.007784009	0.013369977
900	0.009467959	0.01511699
1,000	0.011316955	0.016884029

We can see from the following graph that the insertion sort performs quicker than the merge sort:



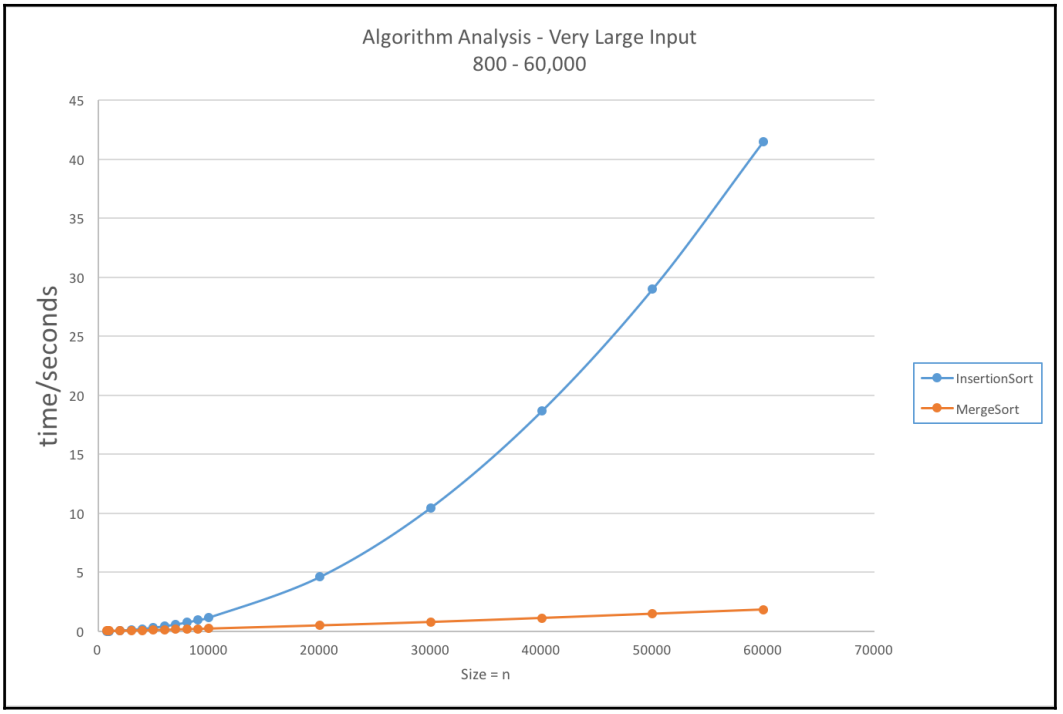
We can see in Table 1.4, what happens as our input gets very large using the insertion sort and merge sort we used previously:

Table 1.4 – Very Large Input Size: 2,000-60,000 items / seconds

n	T(n2)	Tm(n)
800	0.007784009	0.013369977
900	0.009467959	0.01511699
1,000	0.011316955	0.016884029
2,000	0.04688704	0.036652982
3,000	0.105984986	0.05787003
4,000	0.185739994	0.077836037
5,000	0.288598955	0.101580977
6,000	0.417855978	0.124255955

7,000	0.561426044	0.14714098
8,000	0.73259002	0.169996023
9,000	0.930015028	0.197144985
10,000	1.144114017	0.222028017
20,000	4.592146993	0.492881
30,000	10.45656502	0.784195006
40,000	18.64617997	1.122103989
50,000	29.000718	1.481712997
60,000	41.51619297	1.839293003

In this graph, the data clearly shows that the insertion sort algorithm does not scale for larger values of *n*:



Algorithmic complexity is a very important topic in computer science; this was just a basic introduction to the topic to raise your awareness on the subject. Towards the end of this book, we will cover these topics in greater detail in their own chapter.

Summary

This chapter started with a brief introduction on the importance of data structures and algorithms, and why it's important to develop both a broad and deep understanding of them to help you solve difficult problems.

Next, a brief introduction to the Swift REPL was provided where you learned how to enter Swift statements into it to produce results on the fly, and a quick reference of the most frequently used keyboard extensions was provided. We discussed the two fundamental data structures used in computer science and provided examples of the array and singly linked list classes; we'll expand into much greater details on these in the following chapters. We learned about data types in Swift and introduced the collection types available in the Swift standard library. We learned about asymptotic analysis toward the end of the chapter to help bring awareness of how different algorithms can dramatically affect performance.

In the next chapter, we will cover in depth the collection types available in the Swift standard library, followed by a close look at how bridging is performed by legacy Cocoa objects.

2
Working with Commonly Used Data Structures

The Swift language is truly powerful, but a powerful language is nothing if it doesn't have a powerful standard library to accompany it. The Swift standard library defines a base layer of functionality that you can use for writing your applications, including fundamental data types, collection types, functions and methods, and a large number of protocols.

We're going to take a close look at the Swift standard library, specifically looking at support for collection types, with a very low level examination of arrays, dictionaries, sets, and tuples.

The topics covered in this chapter are as follows:

- Using the Swift standard library
- Implementing subscripting
- Understanding immutability
- Interoperability between Swift and Objective-C
- Swift protocol-oriented programming

Using the Swift standard library

Users often treat a standard library as part of the language. In reality, philosophies on standard library design very widely, often with opposing views. For example, the C and C++ standard libraries are relatively small, containing only the functionality that every developer might reasonably require to develop an application. Conversely, languages such as Python, Java, and .NET have large standard libraries that include features, that tend to be separate in other languages, such as XML, JSON, localization, and e-mail processing.

In the Swift programming language, the Swift standard library is separate from the language itself, and is a collection of classes, structures, enumerations, functions, and protocols, which are written in the core language. The Swift standard library is currently very small, even compared to C and C++. It provides a base layer of functionality through a series of generic structures and enums, which also adopt various protocols that are also defined in the library. You'll use these as the building blocks for applications that you develop.

The Swift standard library also places specific performance characteristics on functions and methods in the library. These characteristics are guaranteed only when all participating protocols performance expectations are satisfied. An example is `Array.append()`, its algorithm complexity is amortized *O(1)*, (see `Chapter 1`, *Walking Across the Playground*, for an explanation of Order of Growth functions), unless the array's storage is shared with another live array; *O(count)* if array does not wrap a bridged **NSArray**; otherwise, the efficiency is unspecified.

In this section, we're going to take a detailed look at the implementation of arrays, dictionaries, sets, and tuples.

You'll want to make sure you have at least Xcode 8.1 installed to work with code in this section and to use the Playground examples.

Why structures?

If you're coming from a background working in languages such as Objective-C, C++, Java, Ruby, Python, or other object-oriented languages, you traditionally use classes to define the structure of your types. This is not the case in the Swift standard library; structures are used when defining the majority of the types in the library. If you're coming from an Objective-C or C++ background, this might seem especially odd and feel wrong, because classes are much more powerful than structures.

So why does Swift use structures, which are value types, when it also supports classes, which are reference types that support inheritance, deinitializers, and reference counting? It's exactly because of the limited functionality of structures compared to classes that Swift uses structures instead of classes for the building blocks of the standard library. Because structures are value types, they can have only one owner and are always copied when assigned to a new variable or passed to a function. This can make your code inherently safer because changes to the structure will not affect other parts of your application.

The preceding description refers to the copying of value types. The behavior you see in your code will always be as if a copy took place. However, Swift only performs an actual copy behind the scenes when it is absolutely necessary to do so. Swift manages all value copying to ensure optimal performance, and you should not avoid assignment to try to preempt this optimization.

Structures in Swift are far more powerful than in other C-based languages; they are very similar to classes. Swift structures support the same basic features as C-based structures, but Swift also adds support, which makes them feel more like classes.

Here are some of the features of Swift structures:

- In addition to an automatically generated memberwise initializer, they can have custom initializers
- They can have methods
- They can implement protocols

So this may leave you asking, when should I use a class over a structure? Apple has published guidelines you can follow when considering to create a structure. If one or more of the following conditions apply, consider creating a structure:

- Its primary purpose is to encapsulate a few simple data values
- You expect the encapsulated values to be copied rather than referenced when you pass around or assign an instance of the structure
- Any properties stored by the structure are value types, which would be expected to be copied instead of referenced
- The structure doesn't need to inherit properties or behavior from another existing type

In all other cases, create a class that will call instances of that class to be passed by the reference

Check out the Apple guidelines for choosing between classes and structures:
`https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ClassesAndStructures.html`

Declaring arrays in Swift

An array stores values of the same type in an ordered list. Arrays in Swift have a few important differences compared to arrays in Objective-C. The first is that elements in Swift arrays must be of the same type. If you're used to working with Objective-C NSArrays prior to Xcode 7 you may feel this is a disadvantage, but it actually is a significant advantage as it allows you to know exactly what type you get back when working with an array, allowing you to write more efficient code that leads to fewer defects. However, if you find you must have dissimilar types in an array, you can define the array to be of a protocol that is common to the other types, or defines the array of type `AnyObject`.

Another difference is that Objective-C array elements have to be a class type. In Swift, arrays are generic type collections; there are no limitations on what that type can be. You can create an array that holds any value type, such as int, float, string, or enums, as well as classes.

The last difference is that unlike arrays in Objective-C, Swift arrays are defined as a structure instead of a class.

In `Chapter 1`, *Walking Across the Playground*, we looked at the basic constructs for declaring an integer array and performing operations such as adding, removing, and deleting elements from an array. Let's have a closer, more detailed examination of how arrays are implemented in the standard library.

There are three array types available in Swift:

- Array
- ContiguousArray
- ArraySlice

Every `Array` class maintains a region of memory that stores the elements contained in the array. For array element types that are not a class or `@objc` protocol type, the array's memory region is stored in contiguous blocks. Otherwise, when an array's element type is a class or `@objc` protocol type, the memory region can be a contiguous block of memory, an instance of `NSArray`, or an instance of an `NSArray` subclass.

It may be more efficient to use a `ContiguousArray` if you're going to store elements that are a class or an `@objc` protocol type. The `ContiguousArray` class shares many of the protocols that `Array` implements, so most of the same properties are supported. The key differentiator between `Array` and `ContiguousArray` is that `ContiguousArray` does not provide support for bridging to Objective-C.

The `ArraySlice` class represents a subsequence of an `Array`, `ContiguousArray`, or another `ArraySlice`. Like `ContiguousArray`, `ArraySlice` instances also use contiguous memory to store elements, and they do not bridge to Objective-C. An `ArraySlice` represents a subsequence from a larger, existing array type. Because of this, you need to be aware of the side effects if you try storing an `ArraySlice` after the original array's lifetime has ended and the elements are no longer accessible. This could lead to memory or object leaks, thus Apple recommends that long-term storage of `ArraySlice` instances is discouraged.

When you create an instance of `Array`, `ContiguousArray`, or `ArraySlice`, an extra amount of space is reserved for storage of its elements. The amount of storage reserved is referred to as an array's capacity, which represents the potential amount of storage available without having to reallocate the array. Since Swift arrays share an exponential growth strategy, as elements are appended to an array, the array will automatically resize when it runs out of capacity. When you amortize the append operations over many iterations, the append operations are performed in constant time. If you know ahead of time an array will contain a large number of elements, it may be more efficient to allocate additional reserve capacity at creation time. This will avoid constant reallocation of the array as you add new elements. The following code snippet shows an example of declaring an initial capacity of `500` integer elements for the array `intArray`:

```
// Create an array using full array syntax
var intArray = Array<Int>()

// Create an array using shorthand syntax
intArray = [Int]()

intArray.capacity             // contains 0
intArray.reserveCapacity(500)
intArray.capacity             // contains 508
```

You can see in the preceding example that our reserve capacity is actually larger than 500 integer elements. For performance reasons, Swift may allocate more elements than you request. But you can be guaranteed that at least the number of elements specified will be created.

When you make a copy of an array, a separate physical copy is not made during the assignment. Swift implements a feature called **copy-on-write**, which means that array elements are not copied until a mutating operation is performed when more than one array instances is sharing the same buffer. The first mutating operation may cost *O(n)* in time and space, where *n* is the length of the array.

Initializing array

The initialization phase prepares a struct, class, or enum for use through a method called `init`. If you're familiar with languages such as C++, Java, or C#, this type of initialization is performed in their class constructor, which is defined using the class's name.

If you're coming from Objective-C, the Swift initializers will behave a little differently from what you're used to. In Objective-C, the `init` methods will directly return the object they initialize, and callers will then check the return value when initializing a class and check for nil to see if the initialization process failed. In Swift, this type of behavior is implemented as a feature called **failable initialization**, which we'll discuss shortly.

There are four initializers provided for the three types of Swift arrays implemented in the standard library.

Additionally, you can use a dictionary literal to define a collection of one or more elements to initialize the array, the elements are separated by a comma (,):

```
// Create an array using full array syntax
var intArray = Array<Int>()

// Create an array using shorthand syntax
intArray = [Int]()

// Use array literal declaration
var intLiteralArray: [Int] = [1, 2, 3]
// [1, 2, 3]

//: Use shorthand literal declaration
intLiteralArray = [1, 2, 3]
// [1, 2, 3]

//: Create an array with a default value
intLiteralArray = [Int](count: 5, repeatedValue: 2)
// [2, 2, 2, 2, 2]
```

Adding and updating elements in an array

To add a new element to an array, you can use the `append(_:)` method. This will add new element to the end of the array:

```
var intArray = [Int]()
intArray.append(50)
// [50]
```

If you have an existing collection type, you can use the `append(_:)` method. This will append elements from new element to the end of the array:

```
intArray.append([60, 65, 70, 75])
// [50, 60, 65, 70, 75]
```

If you want to add elements at a specific index, you can use the `insert(newElement:at:)` method. This will add `newElement` at index `i`:

```
intArray.insert(newElement: 55, at: 1)
// [50, 55, 60, 65, 70, 75]
```

To add an element it requires `i <= count` otherwise you will receive a `fatal error: Array index out of range` message and execution will stop.

To replace an element at a specific index, you can use the subscript notation, providing the index of the element you want to replace:

```
intArray[2] = 63
// [50, 55, 63, 65, 70, 75]
```

Retrieving and removing elements from an array

There are several methods you can use to retrieve elements from an array. If you know the specific array index or sub-range of indexes you can use array subscripting:

```
// Initial intArray elements
// [50, 55, 63, 65, 70, 75]
// Retrieve an element by index
intArray[5]
// returns 75

// Retrieve an ArraySlice of elements by subRange
intArray[2..<5]
```

```
// Returns elements between index 2 and less than index 5
// [63, 65, 70]

// Retrieve an ArraySlice of elements by subRange
intArray[2...5]

// Returns elements between index 2 and index 5
// [63, 65, 70, 75]
```

You can also iterate over the array, examining each element in the collection:

```
for element in intArray {
    print(element)
}

// 50
// 55
// 63
// 65
// 70
// 75
```

You can also check whether an array has a specific element or pass a closure that will allow you to evaluate each element:

```
intArray.contains(55)
// returns true
```

Have a look at Apple's Swift documentation for a complete list of the methods available for arrays:
`https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/CollectionTypes.html`

Retrieving and initializing dictionaries

A dictionary is an unordered collection that stores associations between keys and values of the same type with no defined ordering. Each value is associated with a unique key that acts as an identifier for the value in the dictionary. A dictionary data structure works just like a real-world dictionary; to retrieve a definition, reference, translation, or some other type of information for a specific word, you open the dictionary and locate the word to find the information you're looking for. With a dictionary data structure, you store and retrieve values by associating them with a key.

A dictionary key type must conform to the `Hashtable` protocol.

Initializing a dictionary

Just like arrays, there are two ways you can declare a dictionary, by using either the full or literal syntax:

```
// Full syntax declaration
var myDict =  Dictionary<Int, String>()

// Shorthand syntax declaration
var myDict = [Int: String]()
```

Additionally, you can use a dictionary literal to define a collection of one or more key-value pairs to initialize the dictionary. The key and value are separated by a colon (:), and the pairs are separated by a comma (,).

If the keys and values have consistent types, you do not need to declare the type of dictionary in the declaration; Swift will infer it based on the key-value pairs used during initialization, which saves a few keystrokes and allows you to use the shorter form:

```
// Use dictionary literal declaration
var myDict: [Int: String] = [1: "One", 2: "Two", 3: "Three"]
// [2: "Two", 3: "Three", 1: "One"]

// Use shorthand literal declaration
var myDict = [1: "One", 2: "Two", 3: "Three"]
// [2: "Two", 3: "Three", 1: "One"]
```

Adding/modifying/removing a key-value pair

To add a new key-value pair, or to update an existing pair, you can use the `updateValue(_:forKey)` method or subscript notation. If the key is does not exist, a new pair will be added; otherwise, the existing pair is updated with the new value:

```
// Add a new pair to the dictionary
myDict.updateValue("Four", forKey: 4)
$R0: String? = nil
// [2: "Two", 3: "Three", 1: "One", 4: "Four"]
```

```
// Add a new pair using subscript notation
myDict[5] = "Five"
// [5: "Five", 2: "Two", 3: "Three", 1: "One", 4: "Four"]
```

Unlike the subscript method, the `updateValue(_:forKey:)` method will return the value that was replaced, or nil if a new key-value pair was added.

To remove a key-value pair you can use the `removeValue(forKey:)` method, providing the key to delete or setting the key to `nil` using subscript notation:

```
// Remove a pair from the dictionary - returns the removed pair
let removedPair = myDict.removeValue(forKey: 1)
removedPair: String? = "One"
// [5: "Five", 2: "Two", 3: "Three", 4: "Four"]


// Remove a pair using subscript notation
myDict[2] = nil
// [5: "Five", 3: "Three", 4: "Four"]
```

Unlike the subscript method, the `removeValue(forKey:)` method will return the value that was removed, or nil if the key doesn't exist.

Retrieving values from a dictionary

You can retrieve specific key-value pairs from a dictionary using subscript notation. The key is passed to the square bracket subscript notation; it's possible the key may not exist so subscripts will return an Optional. You can use either optional binding or forced unwrapping to retrieve the pair or determine if it doesn't exist. Do not use forced unwrapping though unless you're absolutely sure the key exists, or a runtime exception will be thrown:

```
// Example using Optional Binding
var myDict = [1: "One", 2: "Two", 3: "Three"]
if let optResult = myDict[4] {
    print(optResult)
}
else {
    print("Key Not Found")
}
```

```
// Example using Forced Unwrapping - only use if you know the key will
// exist
let result = myDict[3]!
print(result)
```

Instead of getting a specific value, you can iterate over the sequence of a dictionary and return a `(key, value)` tuple, which can be decomposed into explicitly named constants. In this example, `(key, value)` are decomposed into `(stateAbbr, stateName)`:

```
// Dictionary of state abbreviations to state names
let states = [ "AL" : "Alabama", "CA" : "California", "AK" : "Alaska", "AZ"
: "Arizona", "AR" : "Arkansas"]

for (stateAbbr, stateName) in states {
    print("The state abbreviation for \(stateName) is \(stateAbbr)")
}

// Output of for…in
The state abbreviation for Alabama is AL
The state abbreviation for California is CA
The state abbreviation for Alaska is AK
The state abbreviation for Arizona is AZ
The state abbreviation for Arkansas is AR
```

You can see from the output that items in the dictionary are not output in the order they were inserted. Recall that dictionaries are unordered collections, so there is no guarantee that the order pairs will be retrieved when iterating over them.

If you want to retrieve only the keys or values independently, you can use the keys or values properties on a dictionary.

These properties will return a `LazyMapCollection` instance over the collection. The elements of the result will be computed lazily each time they are read by calling the transform closure function on the base element. The key and value will appear in the same order as they would as a key-value pair, .0 member and .1 member respectively:

```
for (stateAbbr) in states.keys {
    print("State abbreviation: \(stateAbbr)")
}

//Output of for…in
State abbreviation: AL
State abbreviation: CA
State abbreviation: AK
State abbreviation: AZ
State abbreviation: AR
```

```
for (stateName) in states.values {
    print("State name: \(stateName)")
}

//Output of for…in
State name: Alabama
State name: California
State name: Alaska
State name: Arizona
State name: Arkansas
```

You can read more about the `LazyMapCollection` structure on Apple's developer site at `https://developer.apple.com/library/prerelease/ios/documentation/Swift/Reference/Swift_LazyMapCollection_Structure/`

There may be occasions when you want to iterate over a dictionary in an ordered manner. For those cases, you can make use of the global `sort(_:)` method. This will return an array containing the sorted elements of a dictionary as an array:

```
// Sort the dictionary by the value of the key
let sortedArrayFromDictionary = states.sort({ $0.0 < $1.0 })

// sortedArrayFromDictionary contains...
// [("AK", "Alaska"), ("AL", "Alabama"), ("AR", "Arkansas"),
// ("AZ", "Arizona"), ("CA", "California")]

for (key) in sortedArrayFromDictionary.map({ $0.0}) {
    print("The key: \(key)")
}

//Output of for…in
The key: AK
The key: AL
The key: AR
The key: AZ
The key: CA


for (value) in sortedArrayFromDictionary.map({ $0.1}) {
    print("The value: \(value)")
}

//Output of for…in
The value: Alaska
```

```
The value: Alabama
The value: Arkansas
The value: Arizona
The value: California
```

Let's walk through what is happening here. For the `sort` method, we are passing a closure that will compare the first arguments key from the key-value pair, `$0.0`, with the second arguments key from the key-value pair, `$0.1`;if the first argument is less than the second, it will be added to the new array. When the `sort` method has iterated over and sorted all of the elements, a new array of `[(String, String)]` containing the key-value pairings is returned.

Next, we want to retrieve the list of keys from the sorted array. On the `sortedArrayFromDictionary` variable we call `map({ $0.0})`;the transform passed to the map method will add the .0 element of each array element in `sortedArrayFromDictionary` to the new array returned by the map method.

The last step is to retrieve a list of values from the sorted array. We are performing the same call to the map method as we did to retrieve the list of keys; this time, we want the .1 element of each array element in `sortedArrayFromDictionary`, though. Like the preceding example, these values will be added to the new array returned by the map method.

But what if you wanted to base your sorting order on the dictionary value instead of the key? This is simple to do; you would just change the parameter syntax that is passed to the `sort` method. Changing it to `states.sort({ $0.1 < $1.1 })` will now compare the first arguments value from the key-value pair, `$0.1`, with the second arguments value from its key-value pair, `$1.1`, and add the lesser of the two to the new array that will be returned.

Declaring sets

A set is an unordered collection of unique, non-nil elements with no defined ordering. A set type must conform to the `Hashtable` protocol to be stored in a set. All of Swift's basic types are hashable by default. Enumeration case values that do not use associate values are also hashable by default. You can store a custom type in a set; you'll just need to ensure that it conforms to the `Hashable` protocol, as well as the `Equatable` protocol, since `Hashtable` conforms to it.

Sets can be used anywhere you would use an array when ordering is not important and you want to ensure only unique elements are stored. Additionally, access time has the potential of being more efficient than arrays. With an array, the worst case scenario when searching for an element is *O(n)*, where *n* is the size of the array, whereas accessing an element in a set is always constant time, *O(1)*, regardless of its size.

Initializing a set

Unlike the other collection types, sets cannot be inferred from an array literal alone and must be explicitly declared by specifying the `Set` type:

```
// Full syntax declaration
var stringSet =  Set<String>()
```

Because of Swift's type inference, you do not have to specify the type of the set that you're initializing; it will infer the type based on the array literal it is initialized with. Remember, though, that the array literal must contain the same types:

```
// Initialize a Set from an array literal
var stringSet: Set = ["Mary", "John", "Sally"]
print(stringSet.debugDescription)

// Out of debugDescription shows stringSet is indeed a Set type
"Set(["Mary", "John", "Sally"])"
```

Modifying and retrieving elements of a set

To add a new element to a set, use the `insert(_:)` method. You can check whether an element is already stored in a set using the `contains(_:)` method. Swift provides several methods for removing elements from a set; if you have an instance of the element you want to remove you can use the `remove(_:)` method, which takes an element instance. If you know the index of an element in the set you can use the `remove(at:)` method, which takes an instance of `SetIndex<Element>`. If the set count is greater than 0 you can use the `removeFirst()` method to remove the element and the starting index. Lastly, if you want to remove all elements from a set, use the `removeAll()` method or `removeAll(keepCapacity)` method; if `keepCapacity` is true, the current capacity will not decrease:

```
var stringSet: Set = ["Erik", "Mary", "Michael", "John", "Sally"]
// ["Mary", "Michael", "Sally", "John", "Erik"]

stringSet.insert("Patrick")
// ["Mary", "Michael", "Sally", "Patrick", "John", "Erik"]

if stringSet.contains("Erik") {
    print("Found element")
}
else {
    print("Element not found")
}
// Found element

stringSet.remove("Erik")
// ["Mary", "Sally", "Patrick", "John", "Michael"]

if let idx = stringSet.index(of: "John") {
    stringSet.remove(at: idx)
}
// ["Mary", "Sally", "Patrick", "Michael"]

stringSet.removeFirst()
// ["Sally", "Patrick", "Michael"]

stringSet.removeAll()
// []
```

You can iterate over a set the same way as you would the other collection types by using the `for…in` loop. The Swift set type is unordered, so you can use the `sort` method like we did for the dictionary type if you want to iterate over elements in a specific order:

```
var stringSet: Set = ["Erik", "Mary", "Michael", "John", "Sally"]
```

```
// ["Mary", "Michael", "Sally", "John", "Erik"]

for name in stringSet {
    print("name = \(name)")
}

// name = Mary
// name = Michael
// name = Sally
// name = John
// name = Erik

for name in stringSet.sorted() {
    print("name = \(name)")
}

// name = Erik
// name = John
// name = Mary
// name = Michael
// name = Sally
```

Set operations

The set type is modeled on the mathematical set theory, and it implements methods that support basic set operations for comparing two sets, as well as operations that perform membership and equality comparisons between two sets.

Comparison operations

The Swift set type contains four methods for performing common operations on two sets. The operations can be performed either by returning a new set, or using the operations alternative `InPlace` method to perform the operation in place on the source set.

The `union(_:)` and `formUnion(_:)` methods create a new set and update the source set with all the values from both sets, respectively.

The `intersection(_:)` and `formIntersection(_:)` methods create a new set and update the source set with values only common to both Sets, respectively.

The `symmetricDifference(_:)` and `formSymmetricDifference(_:)` methods create a new set and update the source set with values in either set, but not both, respectively.

The `subtracting(_:)` and `subtract(_:)` methods create a new set and update the source set with values not in the specified set, respectively:

```
let adminRole: Set = [ "READ", "EDIT", "DELETE", "CREATE", "SETTINGS",
"PUBLISH_ANY", "ADD_USER", "EDIT_USER", "DELETE_USER"]

let editorRole: Set = ["READ", "EDIT", "DELETE", "CREATE", "PUBLISH_ANY"]

let authorRole: Set = ["READ", "EDIT_OWN", "DELETE_OWN", "PUBLISH_OWN",
"CREATE"]

let contributorRole: Set = [ "CREATE", "EDIT_OWN"]

let subscriberRole: Set = ["READ"]

// Contains values from both Sets
let fooResource = subscriberRole.union(contributorRole)
// "READ", "EDIT_OWN", "CREATE"

// Contains values common to both Sets
let commonPermissions = authorRole.intersection(contributorRole)
// "EDIT_OWN", "CREATE"

// Contains values in either Set but not both
let exclusivePermissions = authorRole.symmetricDifference(contributorRole)
// "PUBLISH_OWN", "READ", "DELETE_OWN"
```

Membership and equality operations

Two sets are said to be equal if they contain precisely the same values, and since sets are unordered, the ordering of the values between sets does not matter.

Use the **==** operator, which is the is equal operator, to determine if two sets contain all of the same values

```
// Note ordering of the sets does not matter
var sourceSet: Set = [1, 2, 3]
var destSet: Set = [2, 1, 3]

var isequal = sourceSet == destSet
// isequal is true
```

Look at the following methods:

- `isSubset(of:)`: Use this method to determine if all of the values of a set are contained in a specified Set.
- `isStrictSubset(of:)`: Use this method to determine if a set is a subset, but not equal to the specified Set.
- `isSuperset(of:)`: Use this method to determine if a set contains all of the values of the specified Set.
- `isStrictSuperset(of:)`: Use this method to determine if a set is a superset, but not equal to the specified Set.
- `isDisjoint(with:)`: Use this method to determine if two sets have the same values in common:

```
let contactResource = authorRole
// "EDIT_OWN", "PUBLISH_OWN", "READ", "DELETE_OWN", "CREATE"

let userBob = subscriberRole
// "READ"

let userSally = authorRole
// "EDIT_OWN", "PUBLISH_OWN", "READ", "DELETE_OWN", "CREATE"

if userBob.isSuperset(of: fooResource){
    print("Access granted")
}
else {
    print("Access denied")
}
// "Access denied"

if userSally.isSuperset(of: fooResource){
    print("Access granted")
}
else {
    print("Access denied")
}
// Access granted

authorRole.isDisjoint(with: editorRole)
// false

editorRole.isSubset(of: adminRole)
// true
```

Characteristics of tuples

The tuples type is an advanced type introduced in Swift that was not available in Objective-C. While the tuples type is not a collection type such as an array, dictionary, or set, it does have similar characteristics. Tuples allow you to group one or more values of any type, but unlike the other collection types where those values must be of the same type, tuples can store values of different types. Since tuples aren't collections, they do not conform to the `SequenceType` protocol, so you cannot iterate over them like you would a collection type.

Tuples are used to store and pass around groups of data. This can be useful if you need to return multiple values from a method as a single value, but do not want to create a new structure type. You should not mean to be persisting beyond temporary scope, though. Apple's Swift documentation says the following about tuples:

> *"Tuples are useful for temporary groups of related values. They are not suited to the creation of complex data structures. If your data structure is likely to persist beyond a temporary scope, model it as a class or structure, rather than as a tuple."*
> *– Apple Inc, The Swift Programming Language (Swift 3)*

Unnamed tuples

You can create tuples with any number and combination of types. Let's create a tuple that contains an integer, string, and integer types:

```
let responseCode = (4010, "Invalid file contents", 0x21451fff3b)
// "(4010, "Invalid file contents", 142893645627)"
```

Because the type information can be inferred, the compiler is able to determine the value types we are setting. But let's see what happens if we try to set an initial value that would overflow an integer type:

```
let responseCode = (4010, "Invalid file contents", 0x8fffffffffffffff)
// We get a compiler error:
// Integer literal '10376293541461622783' overflows when stored into 'Int'
```

If you want to control the type used and not rely on the compiler inferring the type from the literal expression, you can explicitly declare the value type:

```
let responseCode: (Int, String, Double) = (4010, "Invalid file contents",
0x8fffffffffffffff)
// (4010, "Invalid file contents", 1.03762935414616e+19)

// You can verify the tuple types assigned
print(responseCode.dynamicType)
```

```
// (Int, String, Double)
```

There are two ways you can access the individual tuple values, by index, or by decomposing them into constants or variables:

```
let responseCode: (Int, String, Double) = (4010, "Invalid file contents",
0x8fffffffffffffff)
// (4010, "Invalid file contents", 1.03762935414616e+19)

// Using index
print(responseCode.0)   // 4010

// Using decomposition
let (errorCode, errorMessage, offset) = responseCode
print(errorCode)        // 4010
print(errorMessage)     // Invalid file contents
print(offset)           // 1.03762935414616e+19
```

Named tuples

Named tuples are just as their name implies, they allow you to name the tuple values. Using named tuples will allow you to write terser code and can be helpful for identifying what an index position is when returning a tuple from a method:

```
let responseCode = (errorCode:4010, errorMessage:"Invalid file contents",
offset:0x7fffffffffffffff)
// (4010, "Invalid file contents", 9223372036854775807)

print(responseCode.errorCode) // 4010
```

As with unnamed tuples, you can also explicitly declare the tuple type:

```
let responseCode: (errorCode:Int, errorMessage:String, offset:Double) =
(4010, "Invalid file contents", 0x8fffffffffffffff)
// (4010, "Invalid file contents", 1.03762935414616e+19)

print(responseCode.errorCode) // 4010
```

Using named tuples does not prevent you from simultaneously using a tuple index ID as if it was declared as an unnamed tuple.

As previously mentioned, tuples are most useful as providing temporary structured values that are returned from methods. By returning a tuple, you can return additional information that has traditionally required either defining a class or using a dictionary to hold multiple return values. In this example, we will see how tuples can be used to return a nested, structured tuple of values. We'll also see how using named tuples allow you to clearly access individual tuple index by name instead of using their index ID:

```
func getPartnerList() -> (statusCode:Int, description:String,
metaData:(partnerStatusCode:Int, partnerErrorMessage:String,
parterTraceId:String)) {
    //... some error occurred
    return (503, "Service Unavailable", (32323, "System is down for
    maintainance until 2015-11-05T03:30:00+00:00", "5A953D9C-7781-
    427C-BC00-257B2EB98426"))
}

var result = getPartnerList()
print(result)
//(503, "Service Unavailable", (32323, "System is down for maintainance
until
//2015-11-05T03:30:00+00:00", "5A953D9C-7781-427C-BC00-257B2EB98426"))

result.statusCode
// 503

result.description
// Service Unavailable

result.metaData.partnerErrorMessage
// System is down for maintainance until 2015-11-05T03:30:00+00:00

result.metaData.partnerStatusCode
// 32,323

result.metaData.parterTraceId
// 5A953D9C-7781-427C-BC00-257B2EB98426
```

Implementing subscripting

Subscripts can be defined for classes, structures, and enumerations. They are used to provide a shortcut to elements in collections, lists, and sequence types by allowing terser syntax. They can be used to set and get elements by specifying an index instead of using separate methods to set or retrieve values.

Subscript syntax

You can define a subscript that accepts one or more input parameters, the parameters can be of different types, and their return value can be of any type. Use the `subscript` keyword to define a subscript, which can be defined as read-only, or provide a getter and setter to access elements:

```
class MovieList {
    private var tracks = ["The Godfather", "The Dark Knight", "Pulp
Fiction"]
    subscript(index: Int) -> String {
        get {
            return self.tracks[index]
        }
        set {
            self.tracks[index] = newValue
        }
    }
}

var movieList = MovieList()

var aMovie = movieList[0]
// The Godfather

movieList[1] = "Forest Gump"
aMovie = movieList[1]
// Forest Gump
```

Subscript options

Classes and structures can return as many subscript implementations as needed. The support for multiple subscripts is known as **subscript overloading**, the correct subscript to be used will be inferred based on the subscript value types.

Understanding mutability and immutability

Swift doesn't require that you define separate **mutable** and **immutable** types. When you create an array, set, or dictionary variable using the `var` keyword, it will be created as a mutable object. You can modify it by adding, removing, or changing the value of items in the collection. If you create an array, set, or dictionary using the `let` keyword, you are creating a constant object. A constant collection type cannot be modified, either by adding or removing items, or by changing the value of items in the collection.

Mutability of collections

When working with collection types, you need to understand how Swift treats mutability between structs and classes. Some developers tend to get confused when working with constant class instances because their properties can still be modified.

When you create an instance of a structure and assign it to a constant, you cannot modify properties of that instance, even if they are declared as variables. This is not true for classes, though because they are reference types. When you create an instance of a class and assign it to a constant, you cannot assign another variable to that instance, but you can modify the properties of that instance:

```
struct Person {
    var firstName: String
    var lastName: String
    init(firstName: String, lastName: String){
        self.firstName = firstName
        self.lastName = lastName
    }
}

class Address {
    var street: String = ""
    var city: String = ""
    var state: String = ""
    var zipcode: String = ""
    init(street: String, city: String, state: String, zipcode:
    String){
        self.street = street
        self.city = city
        self.state = state
        self.zipcode = zipcode
    }
}
```

```
// Create a constant instance of a Person struct
let person = Person(firstName: "John:", lastName: "Smith")

// Generates a compile time error
person.firstName = "Erik"


// Create a constant instance of an Address class
let address = Address(street: "1 Infinite Loop", city: "Cupertino", state:
"CA", zipcode: "95014")

// This is valid and does not generate a compile time error,
// because Address is a reference type
address.city = "19111 Pruneridge Avenue"


// This does generate a compile time error, because the address instance is
// constant and cannot be modified to point to a different instance
address = Address(street: "19111 Pruneridge Avenue", city: "Cupertino",
state: "CA", zipcode: "95014")
```

Apple recommends creating immutable collections in all cases where the collection does not need to change. Doing so enables the compiler to optimize the performance of the code produced for collections.

Interoperability between Swift and Objective-C

Swift has been designed to be interoperable with Objective-C. You are able to use Swift APIs in your Objective-C projects, and Objective-C APIs in your Swift projects. Apple has greatly expanded the interoperability between the two languages since Swift was originally released; there are still certain features and functionalities that are not compatible with the new languages.

Even if you're new to iOS or macOS development and have never worked with Objective-C before, and do not intend to, you should still familiarize yourself with this section. There are so many existing Objective-C frameworks and class libraries that you're bound to have to deal with interoperability at some point in time.

We will not cover all of the specific areas of interoperability between the two languages in this book, there are other great books that have been published that focus on only core Swift language features that cover them in great detail. We'll look at the specifics of interoperability that you will need to deal with while designing algorithms and working with collection types.

Initialization

You can directly import any Objective-C framework or C library into Swift that supports modules, this includes all of the Objective-C system frameworks and common C libraries supplied with the system.

To use an Objective-C framework in Swift you only need to add the `import` framework name to make the framework accessible:

```
// This import will make all of the Objective-C classes in
// Foundation accessible
import Foundation
```

If you have not worked with Objective-C before let's explain the differences of how you interact with methods on a class instance. You'll often hear the terms methods, functions, and messages used interchangeably. While they are all performing the same action, that is, executing a set of instructions that are contained in a method, the way you interact with methods is different. In Objective-C, you do not call methods that are in Swift or other languages such as C# and Java. Objective-C has the concept of messages.

In Objective-C, a message consists of a receiver, a selector, and parameters. A receiver is the object you want to execute a method on. The selector is the name of the method. And the parameters are passed to the method executing on the object. The syntax would look something like `[myInstance fooMethod:2322 forKey:X]`: this is sending the fooMethod message to the `myInstance` object and passing two parameters, `2322` and `X`, to the `forKey` named parameter.

Let's look at an example using the `NSString` class:

```
NSString *postalCode = [[NSString alloc] initWithFormat:@"%d-%d",32259,
1234];
// postalCode = "32259-1234"
/*
postalCode - is the receiver
   length - is the selector we aren't passing any parameters
*/
int len = [postalCode length];
```

```
// len = 10
```

Note that length is just the name of the method and not the method itself.

With the Objective-C message passing model, it uses dynamic binding by default and does not perform compile-time binding. This allows messages to go unimplemented and the message will be resolved at runtime. At runtime, if an object doesn't support respond to a message, its inheritance chain is walked until an object is found that responds to the message. If one is not found, nil is returned; this behavior can be modified using compiler settings.

When you import an Objective-C framework into Swift, the class's `init` initializers are converted to `init` methods. Initializers that are prefixed with `initWith:` are imported as convenience initializers and the `With:` is removed from the selector name, the rest of the selector is defined using named parameters. Swift will import all class factory methods as convenience methods for consistency when creating objects.

Here is an example of the imported initializer for `NSString` `initWithFormat`:

```
public convenience init(format: NSString, _ args: CVarArgType...)
```

The `initWith` portion of the name is removed from the method name, and format is added as the first parameter. The _ represents an unnamed parameter; this is the variable argument parameter.

Let's contrast how we would create a new `NSString` instance in Swift versus Objective-C now that we understand how initializers and methods are imported:

```
var postalCode:NSString = NSString(format: "%d-%d", 32259, 1234)
var len = postalCode.length
// len = 10
```

The first difference you should notice is we do not have to call `alloc` or `init` to create the instance, Swift will do this for you automatically. Also, notice that `init` does not appear anywhere in our initializer.

You can also use the classes you develop in Swift directly in Objective-C. For your Swift class to be accessible in Objective-C, it must inherit from `NSObject` or any other Objective-C class. Swift classes conforming to this will have their class and members automatically available from Objective-C. If the class does not descend from `NSObject` you can still access its methods by using the `@objc` attribute. The `@objc` attribute can be applied to a Swift method, property, initializer, subscript, protocol, class, or enumeration. If you want to override the symbol name used in Objective-C you can use the `@objc(name)` attribute, where `name` is the new symbol name you want to define. You will then use that name when accessing the member:

```
@objc(ObjCMovieList)
class MovieList : NSObject {
    private var tracks = ["The Godfather", "The Dark Knight",
    "Pulp Fiction"]
    subscript(index: Int) -> String {
        get {
            return self.tracks[index]
        }
        set {
            self.tracks[index] = newValue
        }
    }
}
```

This example creates a new Swift class that inherits from `NSObject`. You do not have to include the `@objc` attribute; by default, the compiler will add it for you. However, in this example, we're changing the symbol name that will be used to access this class.

Swift type compatibility

Beginning in Swift 1.1, there is a built-in language featured called failable initialization, which allows you to define the `init` method. Prior to this feature, you would have been required to write factory methods to handle failures during object creation. By using the new failable initialization pattern, Swift is able to allow greater use of its uniform construction syntax and potentially remove confusion and duplication between initializers and the creation of factory methods.

This is an example of failable initialization using the `NSURLComponents` class:

```
// Note the '[' and ']' characters are invalid for a URI reg-name
// http://tools.ietf.org/html/rfc3986
if let url =  NSURLComponents(string: "http://[www].google.com") {
    // URL is valid and ready for use...
}
else {
    // One or more parts of the URL are invalid...
    // url is nil
}
```

I see the greatest value of the failable initialization feature if you're working with Objective-C and its frameworks. If you're writing pure Swift code, you should frugally use this feature and reserve its use for those situations where it is truly required; do not abuse its use. Later in this chapter, we'll look at examples of why you should not follow this as a general pattern when designing your Swift types and frameworks.

It's important to be aware of the traps you can fall into; just because something feels good because you're used to doing it a certain way, doesn't mean it's good for you. You need to be careful not to try reimplementing your Objective-C patterns in Swift, and instead take advantage of the new features Swift provides when designing your structures and classes.

Here is an example in Swift demonstrating why you should not use failable initialization in your own types:

```
import AppKit

public struct Particle {
    private var name: String
    private var symbol: String
    private var statistics: String
    private var image: NSImage
}

extension Particle{
    // Initializers
    init?(name: String, symbol: String, statistics: String,
    imageName: String){
        self.name = name
        self.symbol = symbol
        self.statistics = statistics
        if let image = NSImage(named: imageName){
            print("initialization succeeded")
            self.image = image
        }
        else {
```

```
            print("initialization failed")
            return nil
        }
    }
}


var quarkParticle = Particle(name: "Quark", symbol: "q", statistics:
"Fermionic", imageName: "QuarkImage.tiff")

// quarkParticle is nill because the image file named "QuarkImage"
// was not found when trying to initialize the object.
```

When developing your structures and classes, you should practice the **SOLID** principle – a mnemonic acronym introduced by Michael Feathers. SOLID stands for the first five basic principles of object-oriented design and programming, as follows:

- **Single Responsibility Principal** – a class should have only one responsibility, it should have one and only one potential reason to change
- **Open/Closed Principal** – software entities should be open for extension, but closed for modification
- **Liskov Substition Principal** – derived classes must be substitutable with their base classes
- **Interface Segregation Principal** – many client specific interfaces are better than a singular general purpose one
- **Dependency Inversion Principal** – depend on abstractions, not on concretions

In this example, we're violating the SOLID principals of Single Responsibility and Dependency Inversion because we're coupling our structure to know about `NSImage` classes, handling the creation of them during initialization, and opening up the potential to be affected by external changes in the `NSImage` class. We've also hidden from the user that we have a dependency on `NSImage` for representing images.

A better implementation is to get the initializer to handle the creation of the image and let the structure manage any errors within `NSImage` using an accessor function, as in this example:

```
import AppKit

public struct Particle {
    private var name: String
    private var symbol: String
    private var statistics: String
    private var image: NSImage
```

```
    public init(name: String, symbol: String, statistics: String,
    image: NSImage){
        self.name = name
        self.symbol = symbol
        self.statistics = statistics
        self.image = image
    }
}

extension Particle{
    public func particalAsImage() -> NSImage {
        return self.image
    }
}

var aURL = NSURL(string:
"https://upload.wikimedia.org/wikipedia/commons/thumb/6/62/Quark_structure_
pion.svg/2000px-Quark_structure_pion.svg.png")
let anImage = NSImage(contentsOfURL: aURL!)

var quarkParticle = Particle(name: "Quark", symbol: "q", statistics:
"Fermionic", image: anImage!)
let quarkImage = quarkParticle.particalAsImage()
```

Bridging collection classes

Swift provides bridging from the Foundation collection types, **NSArray**, **NSSet**, and **NSDictionary**, to the native Swift array, set, and dictionary types. This allows you to work with Foundation collection types and Swift collection types and use them interchangeably with many of the native Swift algorithms that work with collections.

NSArray to Array

Bridging from an NSArray with a parameterized type will create an array of type `[ObjectType]`. If the NSArray does not specify a parameterized type, then an array of `[AnyObject]` is created. Recall from earlier in this chapter that Objective-C arrays do not need to contain the same types like Swift arrays do. If you are working with a bridged NSArray of `[AnyObject]`, you need to handle those instances when the array contains different types. Fortunately, Swift provides methods for you to manage this using forced unwrapping and the type casting operator. If you do not know if an `[AnyObject]` array will contain different types, use the type casting operator since it will return nil, and you can safely perform alternate handling:

```
// Uses forced unwrapping of NSArray
let  nsFibonacciArray: NSArray = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
let swiftFibonacciArray: [Int] = nsFibonacciArray as! [Int]

// Uses type casting operator
if let swiftFibonacciArray: [Int] = nsFibonacciArray as? [Int] {
    // Use the swiftFibonacciArray array
}

// An example with NSArray containing different types
let mixedNSArray: NSArray = NSArray(array: [0, 1, "1", 2, "3", 5, "8", 13,
21, 34])

let swiftArrayMixed: [Int] = mixedNSArray as! [ Int]
// An exception was thrown because not all types were Int


if let swiftArrayMixed: [Int] = mixedNSArray as? [Int]{
    // The condition was false so this body is skipped
}
```

NSSet to set

Bridging from an NSSet with a parameterized type will create a set of type `Set<ObjectType>`. If the NSSet does not specify a parameterized type, then a Swift set of `Set<AnyObject>` is created. Use the same methods of forced unwrapping and the type casting operator that we discussed for arrays to work with sets.

NSDictionary to dictionary

Bridging from an NSDictionary with a parameterized type will create a dictionary of type `[ObjectType]`. If the NSDictionary does not specify a parameterized type, then a Swift dictionary of `[NSObject:AnyObject]` is created.

Swift protocol-oriented programming

In Swift, you should start with a protocol and not a class. Swift protocols define a list of methods, properties, and in some cases, related types and aliases, that a type supports. The protocol forms a contract with a promise that any type that conforms to it will satisfy the requirements of the protocol. Protocols are sometimes referred to as interfaces in other languages such as Java, C#, or Go.

Dispatching

Protocols in Swift are a superset of Objective-C protocols. In Objective-C, all methods are resolved via dynamic dispatch at runtime using **messaging**. Swift, on the other hand, makes use of multiple dispatch techniques; by default, it uses a **vtable**, which lists available methods in the class. A vtable is created at compile time and contains function pointers that are accessed by the index. The compiler will use the vtable as a lookup table for translating method calls to the appropriate function pointer. If a Swift class inherits from an Objective-C class, then it will use dynamic dispatch at runtime. You can also force Swift to use dynamic dispatching by marking class methods with the `@objc` attribute.

The Swift documentation says that, while the @objc attribute exposes your Swift API to the Objective-C runtime, it does not guarantee dynamic dispatch of a property, method, subscript, or initializer. The Swift compiler may still devirtualize or inline member access to optimize the performance of your code, bypassing the Objective-C runtime.

As hinted by Apple in the previous tip, there is a third method of dispatch that may be used. If the compiler has enough information, they will be dispatched statically; this will inline the method calls directly, or it can eliminate them entirely.

Protocol syntax

The syntax for declaring a protocol is very similar to declaring a class or structure:

```
protocol Particle {
var name: String { get }
    func particalAsImage() -> NSImage
}
```

Use the `protocol` keyword followed by the name of the protocol. When you define a protocol property you must specify if it is gettable or writable or both. To define a method provide its name, parameters, and return type. If the method will change member variables on a struct you must add the `mutating` keyword to the method definition.

A protocol can inherit from one or more other protocols. The following code is inheriting the CustomStringConvertible protocol:

```
protocol Particle: CustomStringConvertible {
}
```

You can combine multiple protocols into a single requirement using protocol composition. You can list as many protocols as you need within <> angle brackets, separated by commas. Note, however, that this does not define a new protocol, it's only a temporary local protocol that has the combined requirements of all the protocols in the composition.

Protocols as types

While protocols do not have an implementation, they are a type, and you can use them in places where a type is expected, including the following:

- A return type or parameter type in a function, method, or initializer
- A type of item in an array, set, or dictionary
- A type of variable, constant, or property

In Swift, any type you can name is a first-class citizen and will take advantage of all the OOP features. The real value comes from breaking out a customization point that you define in a protocol that a subclass can override. This allows for difficult logic to be reused while enabling open-ended flexibility and customization.

Protocol extensions

Protocol extensions allow you to extend functionality to an existing protocol, even if you do not have the source code. Extensions can add new methods, properties, and subscripts to existing types. Protocol extensions were introduced in Swift 2, which made it so you didn't have to write global functions to be able to extend an exiting type. Apple refactored a lot of the collection types in the standard library to take advantage of protocol extensions, getting rid of the majority of global functions.

With protocol extensions, you can also define your own default behaviors. Let's look at an example. It is extending the `Collection` protocol and adding the `encryptElements(_:)` method to all types that conform to it:

```
extension Collection {
    func encryptElements(salt: String) -> [Iterator.Element] {
        guard !salt.isEmpty else { return [] }
        guard self.count > 0 else {return [] }
        var index = self.startIndex
        var result: [Iterator.Element] = []
        repeat {
            // encrypt using salt...and add it to result
            let el = self[index]
            result.append(el)
            index = index.successor()
        } while (index != self.endIndex)
        return result
    }
}

var myarr = [String]()
myarr.append("Mary")

var result = myarr.encryptElements("test")
```

Note that there isn't an implementation on this sample for encrypting an element; it's merely meant to demonstrate how you can extend an existing protocol.

Examining protocols for use in collections

Let's look at a few of the protocols that are used to define the Swift Collection types. We'll examine more of these throughout this book, but this will give you an idea of how the Swift standard library uses protocols to design extensible types.

Array literal syntax

When defining an array, there are two forms of syntax you can use. The most common array syntax is to use the collection type name and the data type it will contain:

```
var myIntArray = Array<Int>()
```

Another method is to use array literal or what I'll call elegant syntax, which does not require specifying the collection type name:

```
var = myIntArray = [Int]()
```

The `ExpressibleByArrayLiteral` protocol allows structs, classes, and enums to be initialized using array-like syntax. This protocol requires one method to be implemented in order to comply with the protocol:

```
init(arrayLiteral elements: Self.Element...)
```

If we were to create our own collection type and wanted to support array literal syntax, we would start off defining our type as follows:

```
public struct ParticleList : ExpressibleByArrayLiteral {
   // ExpressibleByArrayLiteral
   init(arrayLiteral: Partible...)

}
```

Let's take a look at a more complete example of what the `ParticleList` implementation would look like to support the array literal syntax:

```
struct Particle {
    var name: String
    var symbol: String
    var statistics: String
}

struct PatricleList: ExpressibleByArrayLiteral {
    private let items: [Particle]
    init(arrayLiteral: Particle...) {
        self.items = arrayLiteral
    }
}

var p1 = Particle(name: "Quark", symbol: "q", statistics: "Fermionic")
var p2 = Particle(name: "Lepton", symbol: "l", statistics: "Fermionic")
var p3 = Particle(name: "Photon", symbol: "Y", statistics: "Bosonic")

var particleList = [p1, p2, p3]
```

```
// particleList containts:
//  [name "Quark", symbol "q", statistics "Fermionic",
//    {name "Lepton", symbol "l", statistics "Fermionic"},
//    {name "Photon", symbol "Y", statistics "Bosonic"}]
```

Making an array enumerable

The `Sequence` and `IteratorProtocol` protocols provide the functionality that allow you to iterate over a collection using the `for…in` syntax. A type must comply with these in order to support enumeration, which supports iterating over a collection. Sequence provides many methods, but we'll only look at the ones required to support the `for…in` syntax for now.

Sequence/IteratorProtocol

The first two collection protocols are inextricably linked: a sequence (a type that conforms to Sequence) represents a series of values, while an iterator (conforming to `IteratorProtocol`, of course) provides a way to use the values in a sequence, one at a time, in sequential order. The `Sequence` protocol only has one requirement: every sequence must provide an iterator from its `makeIterator()` method.

Summary

In this chapter, we've learned about the difference between classes and structures and when you would use one type rather than another, as well as the characteristics of value types and reference types and how each type is allocated at runtime. We went into some of the implementation details for the array, dictionary, and set collection types that are implemented in the Swift standard library. And while not actually a collection type, we also discussed tuples and the two different types Swift supports.

We discussed how Swift interoperates with Objective-C and the C system libraries, and how Swift has added a feature called failable initialization so it can provide backward compatibility with Objective-C. Then we discussed some of the differences in how you call methods in Swift versus sending a message to an Objective-C receiver, as well as the different types of dispatching methods Swift supports. We saw how Swift supports bridging between the native Swift and Objective-C types and how you can guard against collections in Objective-C that may contain different types within a collection.

We then finished talking about protocol-oriented programming and how it forms the basis for the types in the Swift standard library. We only touched the surface on protocols in this chapter, specifically looking at protocols used for the standard library collection types. In later chapters, we'll look at more of the protocols that the Swift standard library defines and how we can leverage them in our structures and classes that we'll be developing.

In the next chapter, we're going to build on what we have learned, and start developing implementations for other advanced data types, such as stacks, queues, heaps, and graphs.

3
Standing on the Shoulders of Giants

The Swift standard library provides the building blocks by providing basic collection types you can use to build your applications. In this chapter, we're going to build on what we learned in the previous chapter and design several commonly used data structures that are used in computer science. Suppose you're receiving a steady stream of data and you want to ensure that it gets processed in the order it arrived. Well, if you use an array data structure, you would need to ensure the data is appended at the end of the array and read from the beginning of it. What if you needed to insert data based on the priority of the data type?

In this chapter, we're going to learn how to visualize data structures so we'll have a clear picture of what they look like in our head. This will prove helpful when you're working through your application requirements; if you can visualize in your head how the data is stored you'll be able to select the best algorithm to implement the requirement.

The topics covered in this chapter are as follows:

- Learning about core Swift protocols for implementing common language constructs
- Implementing array based stack and linked list based stack structures
- Implementing various queue structures
- Learning about linked lists

Iterators, sequences, and collections

The Swift standard library provides several built-in collection types that we looked at back in `Chapter 2`, *Working with Commonly Used Data Structures*. The Swift language runtime provides several language features for working with collections, such as subscripts for shortcuts to access collection elements, and `for…in` control flow for iterating over collections. By conforming to the built-in protocols, `IteratorProtocol`, `Sequence`, and `Collection`, your custom collection types will gain the same type of access the common language constructs use when using subscript and `for…in` syntax as native Swift collection types.

Iterators

An iterator is any type that conforms to the `IteratorProtocol` protocol. The sole purpose of `IteratorProtocol` is to encapsulate the iteration state of a collection by providing the `next()` method, which iterates over a collection, returning the next element in a sequence or nil if the end has been reached.

The `IteratorProtocol` protocol is defined as follows:

```
public protocol IteratorProtocol {
    /// The type of element traversed by the iterator.
    associatedtype Element
    /// Returns: The next element in the underlying sequence if a
    /// next element exists; otherwise, `nil`.
    public mutating func next() -> Self.Element?
}
```

Later in this chapter, we'll implement our own type that will conform to the `IteratorProtocol` protocol.

Sequences

A sequence is anything that conforms to the `Sequence` protocol. Types that conform to sequence can be iterated with a `for…in` loop. You can think of a sequence as a factory iterator that will return an `IteratorProtocol` based on the type of sequence a collection type contains.

The `Sequence` protocol defines many methods; the two definitions we're interested in right now are defined as follows:

```
public protocol Sequence {
    /// A type that provides the sequence's iteration interface and
    /// encapsulates its iteration state.
    associatedtype Iterator : IteratorProtocol
    /// Returns an iterator over the elements of this sequence.
    public func makeIterator() -> Self.Iterator
}
```

The first definition is to define an `associatedtype`. Associated types allow you to declare one or more types that are not known until the protocol is adopted. In Swift, this is how we can implement generics. This definition allows you to specify the actual iterator type when you define your sequence type.

The next definition defines the `makeIterator()` method. Generally, you will not call this method directly. The Swift runtime will call this automatically when using a `for...in` statement.

A complete list of sequence methods can be found at Apple's documentation portal at `https://developer.apple.com/reference/swift/sequence#protocol-requirements`

Collections

A Collection is anything that conforms to the `Collection` protocol. A Collection provides a multi-pass sequence with addressable positions, meaning that you can save the index of an element as you iterate over the collection, then revisit it by providing the index later on.

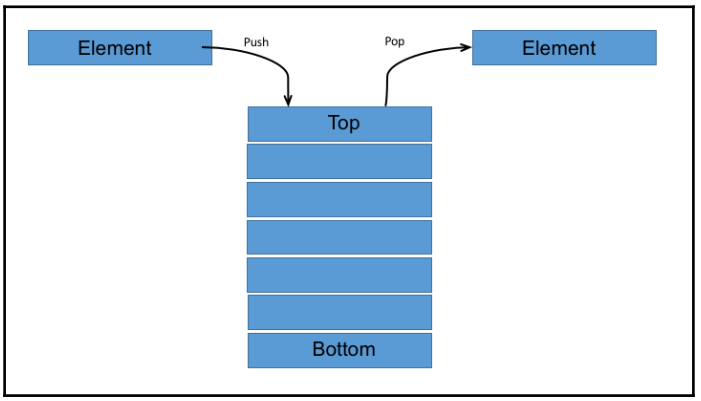
The `Collection` protocol is a `Sequence` and also conforms to the `Indexable` protocol. The minimum requirements for conformance to the `Collection` protocol is your type must declare at least the following four definitions:

- The `startIndex` and `endIndex` properties
- The `index(after:)` method used to advance an index in the collection
- A `subscript` that provides at least read-only access to your type's elements

Later in this chapter, we'll implement our own collection when we look at queues.

Stack

A stack is a **Last In First Out** (**LIFO**) data structure. You can think of a LIFO structure resembling a stack of plates; the last plate added to the stack is the first one removed. A stack is similar to an array but provides a more limited, controlled method of access. Unlike an array, which provides random access to individual elements, a stack implements a restrictive interface that tightly controls how elements of a stack are accessed.



Stack data structure

A stack implements the following three methods:

- `push()` – Adds an element to the bottom of a stack
- `pop()` – Removes and returns an element from the top of a stack
- `peek()` – Returns the top element from the stack, but does not remove it

Common implementations can also include helper operations such as the following:

- `count` – Returns the number of elements in a stack
- `isEmpty()` – Returns true if the stack is empty, and false otherwise
- `isFull()` – If a stack limits the number of elements, this method will return true if it is full and false otherwise

A stack is defined by its interface, which defines what operations can be performed on it. A stack does not define the underlying data structure you use to implement one; common data structures are an array or linked listed, depending on the performance characteristics required.

Applications

Common applications for stacks are expression evaluation and syntax parsing, converting an integer number to a binary number, backtracking algorithms, and supporting undo/redo functionality using the Command design pattern.

Implementation

We're going to take advantage of Swift Generics as we design our stack type for this chapter. This will provide flexibility so we can store any type we want.

Our stack type will implement five methods: `push()`, `pop()`, `peek()`, `isEmpty()`, and the `count` property. The backing storage is going to be an array, and you'll see as we begin implementing it that array provides built-in methods that will make working with it as a stack helpful.

We'll begin by defining our `Stack` structure:

```
public struct Stack<T> {
    private var elements = [T]()
    public init() {}
    public mutating func pop() -> T? {
        return self.elements.popLast()
    }

    public mutating func push(element: T){
        self.elements.append(element)
    }

    public func peek() -> T? {
        return self.elements.last
    }

    public func isEmpty: Bool {
        return self.elements.isEmpty
    }

    public var count: Int {
        return self.elements.count
    }
}
```

Our stack type introduced the `mutating` keyword. Sometimes it is necessary for a method to modify the instance it belongs to, such as modifying value types such as structures. To allow the method to modify data contained in the structure, you must place the `mutating` keyword before the method name. You can read more about mutating method requirements in Apple's documentation at `https://developer.apple.com/library/prerelease/content/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html#//apple_ref/doc/uid/TP40014097-CH25-ID271`

We are using generics for our type, which will be determined at compile time the type that is stored in our Stack. We are using an array to store elements in our Stack. The array type provides a `popLast()` we're able to leverage when implementing our `pop()` method to remove the topmost element from the array.

Let's look at a few examples of how we can use our stack structure:

```
var myStack = Stack<Int>()

myStack.push(5)
// [5]
myStack.push(44)
// [5, 44]
myStack.push(23)
// [5, 44, 23]

var x = myStack.pop()
// x = 23
x = myStack.pop()
// x = 44
x = myStack.pop()
// x = 5
x = myStack.pop()
// x = nil
```

You will find the stack implementations in the `B05101_3_Standing_On_The_Shoulders_Of_Giants.playground` defined as ArrayStack and StackList.

Protocols

Our stack type will use extensions to extend its behavior and functionality. At the beginning of this chapter, we discussed iterators and sequences. We'll add these to our stack type so we can conform to them and allow the built-in language constructs to be used when working with our stack type.

We'll begin by adding a couple of convenience protocols. The first two we'll add are `CustomStringConvertible` and `CustomDebugStringConvertible`. These allow you to return a friendly name when printing out a types value:

```
extension Stack: CustomStringConvertible, CustomDebugStringConvertible {
    public var description: String {
        return self.elements.description
    }
    public var debugDescription: String {
        return self.elements.debugDescription
    }
}
```

Next, we would like our stack to behave like an array when we initialize it, so we'll make it conform to `ExpressibleByArrayLiteral`. This will allow us to use angle bracket notation, such as `var myStack = [5, 6, 7, 8]`:

```
extension Stack: ExpressibleByArrayLiteral {
    public init(arrayLiteral elements: T...) {
        self.init(elements)
    }
}
```

Next, we'll extend `Stack` so it conforms to the `IteratorProtocol` protocol. This will allow us to return an iterator based on the type of element our stack contains:

```
public struct ArrayIterator<T> : IteratorProtocol {
    var currentElement: [T]
    init(elements: [T]){
        self.currentElement = elements
    }
    mutating public func next() -> T? {
        if (!self.currentElement.isEmpty) {
            return self.currentElement.popLast()
        }
        return nil
    }
}
```

Since our stack implementation is using an array for its internal storage, we define an instance variable named `currentElement` as `[T]`, where `T` is defined when `ArrayIterator` is adopted so it will hold the array that is passed to the initializer.

We then implement the `next()` method that is responsible for returning the next element in the sequence. If we've reached the end of the array, we return nil. This will signal to the Swift runtime that there are no more elements available when iterating over it in a `for...in` loop.

Next, we want to extend support to conform to the `Sequence` protocol by implementing the `makeIterator()` method. Inside this implementation is where we wire up the `ArrayIterator` type we just defined:

```
extension Stack: Sequence {
    public func makeIterator() -> ArrayIterator<T> {
        return ArrayIterator<T>(elements: self.elements)
    }
}
```

The Swift runtime will call the `makeIterator()` method to initialize the `for...in` loop. We'll return a new instance of our `ArrayIterator` that is initialized when the backing array is used by our stack instance.

The last addition we'll make is to add another initializer that will allow us to initialize a new stack from an existing one. Note that we need to call `reversed()` on our array when creating a copy. That's because `Sequence.makeIterator()` is called by the array initializer and pops elements off our stack. If we don't reverse them, we'll end up creating an inverted stack from our original, which is not what we want:

```
public init<S : Sequence>(_ s: S) where S.Iterator.Element == T {
    self.elements = Array(s.reversed())
}
```

Now we can do the following:

```
// Use array literal notation
var myStack = [4,5,6,7]

// Use the new initializer we defined
var myStackFromStack = Stack<Int>(myStack)
// [4, 5, 6, 7]

myStackFromStack.push(55)
// [4, 5, 6, 7, 55]

myStack.push(70)
```

```
// [4, 5, 6, 7, 70]
```

Note that a copy of the existing stack is used when creating the `myStackFromStack` instance. When we add `55` to `myStackFromStack`, the `myStack` instance is not updated and only contains items added explicitly to it.

We still have room for additional improvements. Instead of writing a lot of boilerplate code for our iterator and sequencer implementations, we can leverage work already done by the Swift standard library. We'll introduce three new types that will allow us to optimize our stack so it also supports lazy evaluation:

- `AnyIterator<Element>` – An abstract `IteratorProtocol` base type. Use as a sequence type associated IteratorProtocol type.
- `AnySequence<Base>` – Creates a sequence that has the same elements as base. When used below the call to .lazy, it will return a `LazySequence` type.
- `IndexingIterator<Elements>` – An iterator for an arbitrary collection. This is also the default iterator for any collection that doesn't declare its own.

Let's update our `Sequence` extension to use the preceding types:

```
extension Stack: Sequence {
    public func makeIterator() -> AnyIterator<T> {
      return AnyIterator(IndexingIterator(_elements:
      self.elements.lazy.reversed()))
    }
}
```

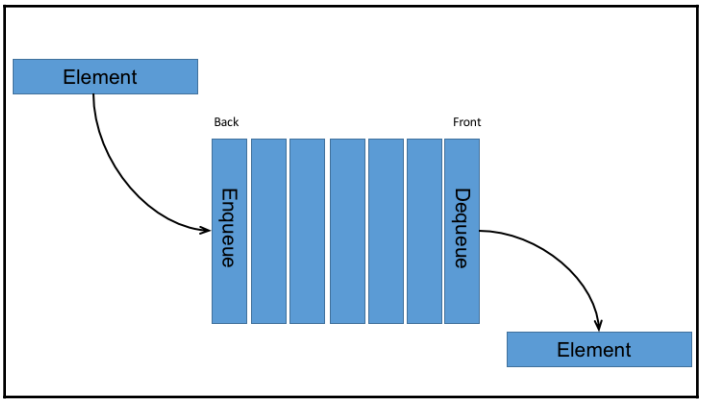
You can view the complete code included with this book.

> *"Extensions are the method used to add new functionality to an existing class, structure, enumeration, or protocol type. Extensions can add computed properties and computed type properties, define instance methods and type methods, provide new initializers, define subscripts, define and use new nested types, and make an existing type conform to a protocol."*
> *– Apple Swift Documentation*

Queue

A queue is a **First In First Out** (**FIFO**) data structure. To visualize a FIFO, imagine you're standing in line for the checkout at the grocery store. When the first person (head) in line reaches the cashier, she rings up their purchases, they pay and collect their groceries and leave (pop); the second person in line is now first in line, and we repeat the process.

When a new customer stands (push) in line behind the last person in line, they are now in the tail position.



Queue data structure

A queue implements the following seven operations:

- `enqueue()` – Adds an element to the back of the queue
- `dequeue()` – Removes and returns the first element from the queue
- `peek()` – Returns the first element from the queue, but does not remove it
- `clear()` – Resets the queue to an empty state
- `count` – Returns the number of elements in the queue
- `isEmpty()` – Returns true if the queue is empty, and false otherwise
- `isFull()` – Returns true if the queue is full, and false otherwise

Common implementations can also include helper methods such as the following:

- `capacity` – A read/write property for retrieving or setting the queue capacity
- `insert(_:atIndex)` – A method that inserts an element at a specified index in the queue
- `removeAtIndex(_)` – A method that removes an element at the specified index

What is the difference between `Array.capacity` and `Array.count`? The `Array.capacity` returns how many elements the array can hold, whereas `Array.count` returns how many elements are currently contained in the array.

Applications

The use of queues is common when you need to process items in the order they are received. Consider writing a **point-of-sale** (**POS**) system for a restaurant. It's very important that you process those orders in the order they were received, otherwise you'll have some very upset customers on your hands. Your POS system would write orders to the back of the queue, and then read orders from the front of the queue on different terminals used by the cooks. This will allow the cooks to work on the orders as they were added to the system.

Implementation

As we did with our stack implementation, we're going to take advantage of Swift generics as we design our queue data structures for this chapter, since this will make it flexible for storing any type you want.

Our queue type will implement six methods: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`, `isFull()`, and `clear()`, and one property, `count`. The backing storage will use an array.

We'll begin by defining our `Queue` structure:

```
public struct Queue<T> {
    private var data = [T]()
    /// Constructs an empty Queue.
    public init() {}

    /// Removes and returns the first `element` in the queue.
    /// -returns:
    /// -If the queue not empty, the first element of type `T`.
    /// -If the queue is empty, 'nil' is returned.
    public mutating func dequeue() -> T? {
        return data.removeFirst()
    }

    /// Returns the first `element` in the queue without
    /// removing it.
    /// -returns:
    /// -If the queue not empty, the first element of type `T`.
    /// -If the queue is empty, 'nil' is returned.
    public func peek() -> T? {
        return data.first
    }

    /// Appends `element` to the end of the queue.
    /// -complexity: O(1)
    /// -parameter element: An element of type `T`
```

```
    public mutating func enqueue(element: T) {
        data.append(element)
    }

    /// MARK:- Helpers for a Circular Buffer
    /// Resets the buffer to an empty state
    public mutating func clear() {
        data.removeAll()
    }

    /// Returns the number of elements in the queue.
    /// `count` is the number of elements in the queue.
    public var count: Int {
        return data.count
    }

    /// Returns the capacity of the queue.
    public var capacity: Int {
        get {
            return data.capacity
        }
        set {
            data.reserveCapacity(newValue)
        }
    }

    /// Check if the queue is full.
    /// -returns: `True` if the queue is full, otherwise
    /// it returns `False`.
    public func isFull() -> Bool {
        return count == data.capacity
    }

    /// Check if the queue is empty.
    /// - returns: `True` if the queue is empty, otherwise
    /// it returns `False`.
    public func isEmpty() -> Bool {
        return data.isEmpty
    }
}
```

The implementation is pretty simple, we're basically wrapping an array and providing assessor methods to allow it to behave like a queue. The array will dynamically resize itself as capacity is reached.

Let's look at a few examples of how we can use our `Queue` structure:

```
var queue = Queue<Int>()

queue.enqueue(100)
queue.enqueue(120)
queue.enqueue(125)
queue.enqueue(130)

let x = queue.dequeue()
// x = 100

// Lets look at the next element but not remove it
let y = queue.peek()
// y = 120

let z = queue.dequeue()
// y = 120
```

Protocols

Let's add a couple of convenience protocols. The first two we'll add are `CustomStringConvertible` and `CustomDebugStringConvertible`. These allow you to return a friendly name when printing out a types value:

```
extension Queue: CustomStringConvertible, CustomDebugStringConvertible {

    public var description: String {
        return data.description
    }
    public var debugDescription: String {
        return data.debugDescription
    }
}
```

Next, we would like our queue to behave like an array when we initialize it, so we'll make it conform to `ExpressibleByArrayLiteral`. This will allow us to use angle bracket notation such as `var queue: Queue<Int> = [1,2,3,4,5]`. We also need to provide a new initializer that takes a sequence that contains elements that match the type defined for the `Queue` instance we're initializing:

```
/// Constructs a Queue from a sequence.
public init<S: Sequence>(_ elements: S) where
S.Iterator.Element == T {
    data.append(contentsOf: elements)
}

extension Queue: ExpressibleByArrayLiteral {
    /// Constructs a queue using an array literal.
    public init(arrayLiteral elements: T...) {
        self.init(elements)
    }
}
```

We may also want to use our queue in a `for...in` loop as we would other collection types. Let's make it conform to the `Sequence` protocol so it returns a lazy loaded sequence:

```
extension Queue: Sequence {
    /// Returns an *iterator* over the elements of this
    /// *sequence*.
    /// -Complexity: O(1).
    public func generate() -> AnyIterator<T> {
        AnyIterator(IndexingIterator(_elements: data.lazy))
    }
}
```

Another useful protocol to implement for a collection type is the `MutableCollection`. It allows you to use subscript notation to set and retrieve elements of the queue. Since using subscript notation allows the client to specify the indexes, we need to make sure we validate them first, so we will also implement a `checkIndex()` method:

```
/// Verifies `index` is within range
private func checkIndex(index: Int) {
    if index < 0 || index > count {
        fatalError("Index out of range")
    }
}

extension Queue: MutableCollection {
    public var startIndex: Int {
        return 0
    }
```

```
    public var endIndex: Int {
        return count - 1
    }

/// Returns the position immediately after the given index.
public func index(after i: Int) -> Int {
        return data.index(after: i)
    }

    public subscript(index: Int) -> T {
        get {
            checkIndex(index)
            return data[index]
        }
        set {
            checkIndex(index)
            data[index] = newValue
        }
    }
}
```

Let's use the protocols we just implemented so you understand how they work:

```
// Use ArrayLiteral notation
var q1: Queue<Int> = [1,2,3,4,5]

// Create a new queue using the initializer that takes a SequenceType from
q1
var q2 = Queue<Int>(q1)

let q1x = q1.dequeue()
// q1x = 1

q2.enqueue(55)
// q2 = [1,2,3,4,5,55]

// For..in uses the SequenceType protocol
for el in q1 {
    print(el)
}
```

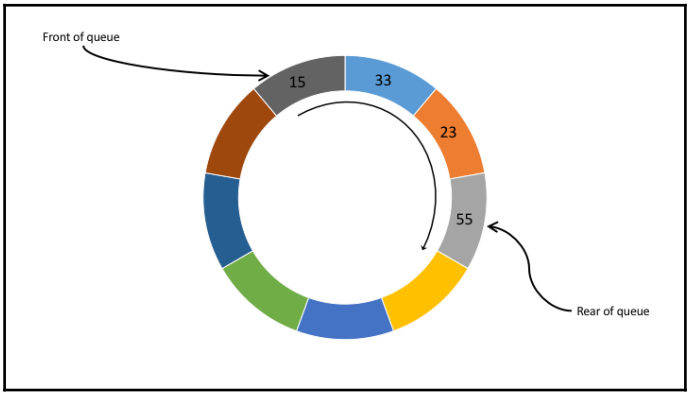
You can view the complete code included with this book.

Circular buffer

A circular buffer is a fixed-size data structure that contains two indices, a head index and a tail index that connects to the beginning of the buffer. When the buffer is full, the head index will loop back to 0. Their main purpose is to accept incoming data until their capacity is full, and then overwrite older elements.

Circular buffers are useful when you need a FIFO data structure. They are similar to the queue data structure, except the tail index wraps to the front of the buffer to form a circular data structure.

Since circular buffers are a fixed size, as they become full the older elements will be overwritten. Because of their fixed size, it is more efficient to use an array data structure internally to store the data instead of a linked list. Generally, once you create a circular buffer, the size will not increase, so the buffer memory size should stay pretty static. An implementation could include the capability to resize the buffer and move the existing elements to the newly created buffer. If you need to frequently resize the buffer, it may be more efficient to implement it using a linked list instead of an array.



Circular buffer data structure

A circular buffer implements the following six methods and two properties:

- `push()` – Adds an element to the end of a buffer
- `pop()` – Returns and removes the front element from the buffer
- `peek()` – Returns the front element from the buffer, but does not remove it
- `clear()` – Resets the buffer to an empty state
- `isEmpty()` – Returns true if the buffer is empty, and false otherwise
- `isFull()` – Returns true if the buffer is full, and false otherwise

- `count` – Returns the number of elements in the buffer
- `capacity` – A read/write property for returning or setting the buffer capacity

Common implementations can also include helper methods such as the following:

- `insert(_:atIndex)` – A method that inserts an element at a specified index in the buffer
- `removeAtIndex(_)` – A method that removes an element at the specified index

Applications

The use of circular buffers is common for performing video or audio processing, for example, if you're developing a video capturing application that is recording a live stream. Since writing to disk, either local or over the network, is a slow operation, you can write the incoming video stream to a circular buffer; we'll call this thread the producer. Then, within another thread, which we'll call the consumer, you would read elements from the buffer and write them to some form of durable storage.

Another similar example is when processing an audio stream. You could write the incoming stream to the buffer and have another thread that applies audio filtering before writing it to durable storage or playback.

Implementation

As we did with our queue implementation, we're going to take advantage of Swift Generics as we design our `CircularBuffer` data structure for this chapter, since this will make it flexible for storing any type you want.

Our `CircularBuffer` type will initially implement six methods: `push()`, `pop()`, `peek()`, `isEmpty()`, `isFull()`, and `clear()`, and one property, `count`. The backing storage will use an array.

We'll begin by defining our `CircularBuffer` structure:

```
public struct CircularBuffer<T> {
    fileprivate var data: [T]
    fileprivate var head: Int = 0, tail: Int = 0
    private var internalCount: Int = 0


    private var overwriteOperation:
```

```
CircularBufferOperation =
CircularBufferOperation.Overwrite

/// Constructs an empty CircularBuffer.
public init() {
    data = [T]()
    data.reserveCapacity
    (Constants.defaultBufferCapacity)
}

/// Construct a CircularBuffer of `count` elements
/// -remark: If `count` is not a power of 2 it will be
/// incremented to the next closest power of 2 of its value.
public init(_ count:Int, overwriteOperation:
CircularBufferOperation = .Overwrite){
    var capacity = count
    if (capacity < 1) {
        capacity = Constants.defaultBufferCapacity
    }

    // Ensure that `count` is a power of 2
    if ((capacity & (~capacity + 1)) != capacity) {
        var b = 1
        while (b < capacity) {
            b = b << 1
        }
        capacity = b
    }

    data = [T]()
    data.reserveCapacity(capacity)
    self.overwriteOperation = overwriteOperation
}

/// Constructs a CircularBuffer from a sequence.
public init<S: Sequence>(_ elements: S, size: Int)
where S.Iterator.Element == T {
    self.init(size)
    elements.forEach({ push(element: $0) })
}

/// Removes and returns the first `element` in the buffer.
/// -returns:
/// -If the buffer isn't empty, the first element of type `T`.
/// -If the buffer is empty, 'nil' is returned.
public mutating func pop() -> T? {
    if (isEmpty()){
        return nil
```

```
        }

        let el = data[head]
        head = incrementPointer(pointer: head)
        internalCount -= 1
        return el
    }

    /// Returns the first `element` in the buffer without
    /// removing it.
    /// -returns: The first element of type `T`.
    public func peek() -> T? {
        if (isEmpty()){
            return nil
        }
        return data[head]
    }

    /// Appends `element` to the end of the buffer.
    /// The default `overwriteOperation` is
    /// `CircularBufferOperation.Overwrite`, which overwrites
    /// the oldest elements first if the buffer capacity is full.
    /// If `overwriteOperation` is
    /// `CircularBufferOperation.Ignore`, when the capacity is
    /// full newer elements will not be added to the buffer
    /// until exisint elements are removed.
    /// -complexity: O(1)
    /// -parameter element: An element of type `T`
    public mutating func push(element: T) {
        if (isFull()){
            switch(overwriteOperation){
                case .Ignore:
                // Do not add new elements until the count
                is less than the capacity
                return
                case .Overwrite:
                pop()
            }
        }

        if (data.endIndex < data.capacity) {
            data.append(element)
        }
        else {
            data[tail] = element
        }

        tail = incrementPointer(pointer: tail)
```

```
    internalCount += 1
}

/// Resets the buffer to an empty state
public mutating func clear() {
    head = 0
    tail = 0
    internalCount = 0
    data.removeAll(keepingCapacity: true)
}

/// Returns the number of elements in the buffer.
/// `count` is the number of elements in the buffer.
public var count: Int {
    return internalCount
}

/// Returns the capacity of the buffer.
public var capacity: Int {
    get {
        return data.capacity
    }
    set {
        data.reserveCapacity(newValue)
    }
}

/// Check if the buffer is full.
/// -returns: `True` if the buffer is full, otherwise
/// it returns `False`.
public func isFull() -> Bool {
    return count == data.capacity
}

/// Check if the buffer is empty.
/// -returns: `True` if the buffer is empty,
/// otherwise it returns `False`.
public func isEmpty() -> Bool {
    return (count < 1)
}

/// Increment a pointer value by one.
/// - remark: This method handles wrapping the
/// incremented value if it would be beyond the last
/// element in the array.
fileprivate func incrementPointer(pointer: Int) -> Int
{
    return (pointer + 1) & (data.capacity - 1)
```

```
    }

    /// Decrement a pointer value by 1.
    /// - remark: This method handles wrapping the
    /// decremented value if it would be before the first
    /// element in the array.
    fileprivate func decrementPointer(pointer: Int) -> Int
    {
        return (pointer - 1) & (data.capacity - 1)
    }

    /// Converts a logical index used for subscripting to
    /// the current internal array index for an element.
    fileprivate func
    convertLogicalToRealIndex(logicalIndex: Int) -> Int {
        return (head + logicalIndex) & (data.capacity - 1)
    }

    /// Verifies `index` is within range
    fileprivate func checkIndex(index: Int) {
        if index < 0 || index > count {
            fatalError("Index out of range")
        }
    }
}
```

Let's start taking a closer look at this code. We'll begin by reviewing the two initializers. The default initializer will create an array with a capacity of eight elements. The second initializer, `init(_:)`, allows you to specify the capacity used to initialize the buffer. Note that `count` should be a power of 2. If it is not it will be incremented to the next closest power of 2 value.

Let's look at a few examples of how we can use our CircularBuffer structure:

```
var circBuffer = CircularBuffer<Int>(4)

circBuffer.push(element: 100)
circBuffer.push(element: 120)
circBuffer.push(element: 125)
circBuffer.push(element: 130)

let x = circBuffer.pop()
// x = 100

// Lets look at the next element but not remove it
let y = circBuffer.peek()
// y = 120
```

```
let z = circBuffer.pop()
// y = 120

circBuffer.push(element: 150)
circBuffer.push(element: 155)
circBuffer.push(element: 160)
// Our capacity is only 4 so this element overwrote 125
```

The default behavior so far is to overwrite older elements once the buffer is full. There may be times when that is not what you want. Let's modify the structure to support an option to either ignore new elements or overwrite the oldest elements when the buffer is full.

We'll add a new `enum` to define the behavior of a full buffer:

```
/// Control the buffer full behavior when adding new elements
public enum CircularBufferOperation {
    case Ignore, Overwrite
}
```

Now let's add a private property to track which operation we're using. Then we'll modify our initializer and push methods to use that property:

```
public struct CircularBuffer<T> {

    private var overwriteOperation:
    CircularBufferOperation =
    CircularBufferOperation.Overwrite

    public init(_ count:Int, overwriteOperation:
    CircularBufferOperation = .Overwrite){
        var capacity = count
        if (capacity < 1) {
            capacity = Constants.defaultBufferCapacity
        }

        // Ensure that `count` is a power of 2
        if ((capacity & (~capacity + 1)) != capacity) {
            var b = 1
            while (b < capacity) {
                b = b << 1
            }
            capacity = b
        }

        data = [T]()
        data.reserveCapacity(capacity)
        self.overwriteOperation = overwriteOperation
```

```
    }

    public mutating func push(element: T) {
        if (isFull()){
            switch(overwriteOperation){
                case .Ignore:
                // Do not add new elements until the count
                // is less than the capacity
                return
                case .Overwrite:
                pop()
            }
        }
        if (data.endIndex < data.capacity) {
            data.append(element)
        }
        else {
            data[tail] = element
        }
        tail = incrementPointer(tail)
        internalCount += 1
    }
}
```

Now the default behavior is to overwrite the oldest elements once the buffer is full. To change that, you use the custom initializer to specify that it should ignore new elements when the buffer is full. You can check the `count` property before calling `push(_:)` to see if the buffer is at capacity yet:

```
var circBufferIgnore = CircularBuffer<Int>(count: 4, overwriteOperation:
CircularBufferOperation.Ignore)

let cnt = circBufferIgnore.count
```

Protocols

Let's add a couple of convenience protocols. The first two we'll add are `CustomStringConvertible` and `CustomDebugStringConvertible`. These allow you to return a friendly name when printing out a types value:

```
extension CircularBuffer: CustomStringConvertible,
CustomDebugStringConvertible {

public var description: String {
        return data.description
    }
```

```
    public var debugDescription: String {
        return data.debugDescription
    }
}
```

Next, we would like our `CircularBuffer` to behave like an array when we initialize it, so we'll make it conform to `ExpressibleByArrayLiteral`. This will allow us to use angle bracket notation such as `var myCircBuffer: CircularBuffer<Int> = [5, 6, 7, 8]`. We also need to provide a new initializer that takes a `Sequence` that contains elements that match the type defined for the `CircularBuffer` instance we're initializing:

```
/// Constructs a CircularBuffer from a sequence.
public init<S: Sequence>(_ elements: S, size: Int) where S.Iterator.Element
== T {
        self.init(size)
        elements.forEach({ push(element: $0) })
    }

extension CircularBuffer: ExpressibleByArrayLiteral {
    /// Constructs a circular buffer using an array literal.
    public init(arrayLiteral elements: T...) {
        self.init(elements, size: elements.count)
    }
}
```

We may also want to use our `CircularBuffer` in a `for...in` loop as we would other collection types. Let's make it conform to the `Sequence` protocol so it returns a lazy loaded sequence. You will notice the `makeIterator()` method is more complicated than what we implemented for the `Stack` structure. Since our `CircularBuffer` loops back to the front once the capacity is full, we need to account for that when returning our iterator. We first check our head and tail pointers to see if our tail has wrapped to the front of our array. If it has, we need to first copy from the head pointer to the end of the array. We then check count of our buffer. If we have remaining elements that need to be copied, we increment the starting position in our range and copy to the capacity of the new array:

```
extension CircularBuffer: Sequence {
    /// Returns an *iterator* over the elements of this *sequence*.
    /// -Complexity: O(1).
    public func makeIterator() -> AnyIterator<T> {
        var newData = [T]()
        if count > 0 {
            newData = [T](repeatingValue: data[head],
            count: count)
            if head > tail {
                // number of elements to copy for firsthalf
```

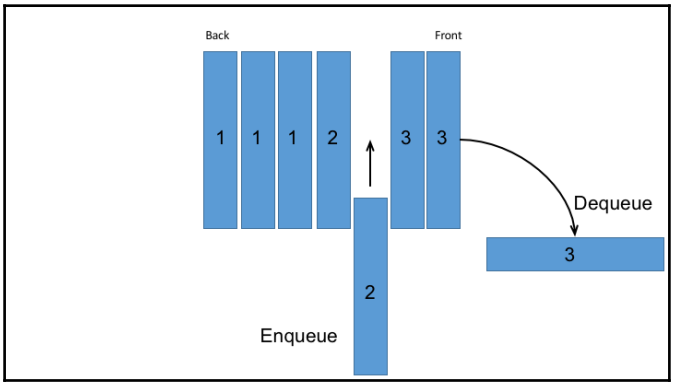
```
                let front = data.capacity - head
                newData[0..<front] = data[head..
                <data.capacity]
                if front < count {
                    newData[front + 1..<newData.capacity]
                    = data[0..<count - front]
                }
            }
            else {
                newData[0..<tail - head] = data[head..
                <tail]
            }
        }

        return AnyIterator(IndexingIterator(newData.lazy))
    }
}
```

You can view the complete code included with this book.

Priority queue

A Priority queue is like a regular queue, except each element has a priority assigned to it. Elements that have a higher priority are dequeued before lower priority elements. Instead of writing my own version of a `PriorityQueue` for this book, I'm going to reference an implementation by David Kopec. David's Swift `PriorityQueue` is a popular implementation, and he's done a great job of keep it up to date with all of the frequent Swift language changes. You can find the complete code for `PriorityQueue` included in this book. I'll also include the GitHub URL to his project at the end of this section.

The `PriorityQueue` is a pure Swift implementation of a generic priority queue data structure. It features a straightforward interface and can be used with any type that implements Comparable. It utilizes comparisons between elements rather than numeric priorities to determine order. It uses a classic binary heap implementation with *O(log n)* complexity for pushes (enqueue) and pops (dequeue). It also conforms to `Sequence` like the previous structures we've implemented, so it has support for iterating using the standard `for...in` notation.



Priority queue data structure

The `PriorityQueue` implements the following six operations:

- `push()` – Adds an element to the priority queue, O(log n)
- `pop()` – Returns and removes the element with the highest (or lowest if ascending) priority or nil if the priority, queue is empty, O(log n)
    - `peek()` – Returns the element with the highest (or lowest if ascending) priority, or nil if the priority queue is empty, O(log n)
- `clear()` – Resets the priority queue to an empty state
- `count` – Returns the number of elements in the priority queue
- `isEmpty` – Returns true if the priority queue has zero elements, and false otherwise

Applications

A Priority queue is useful whenever you to need to control the order in which elements are processed from a queue. Several popular algorithms use a priority queue in their implementation:

- **Best-first search algorithm:** Like the A* search algorithm, it finds the shortest path between two nodes of a weighted graph – a priority queue is used to keep track of unexplored routes
- **Prim algorithm:** This finds the minimum spanning tree for a weighted undirected graph

A common business application use case is a hospital information system for an emergency room. The application would add patients to the doctor's queue once their paperwork has been processed. There may be a patient that comes in with a severe or life-threatening injury. The hospital needs to get those patients in front of a doctor as soon as possible, so a priority queue would be used to set their priority to a higher level.

Implementation

Let's review the implementation for `PriorityQueue`; you'll find it is very similar to our `Queue` structure:

```
public struct PriorityQueue<T: Comparable> {
    fileprivate var heap = [T]()
    private let ordered: (T, T) -> Bool
    public init(ascending: Bool = false, startingValues:
    [T] = []) {
        if ascending {
            ordered = { $0 > $1 }
        } else {
            ordered = { $0 < $1 }
        }

        // Based on "Heap construction" from Sedgewick p 323
        heap = startingValues
        var i = heap.count/2 - 1
        while i >= 0 {
            sink(i)
            i -= 1
        }
    }

    /// How many elements the Priority Queue stores
```

```
public var count: Int { return heap.count }

/// true if and only if the Priority Queue is empty
public var isEmpty: Bool { return heap.isEmpty }

/// Add a new element onto the Priority Queue. O(log n)
/// -parameter element: The element to be inserted
/// into the Priority Queue.
public mutating func push(_ element: T) {
    heap.append(element)
    swim(heap.count - 1)
}

/// Remove and return the element with the highest
/// priority (or lowest if ascending). O(log n)
/// -returns: The element with the highest priority in the
/// Priority Queue, or nil if the PriorityQueue is empty.
public mutating func pop() -> T? {
    if heap.isEmpty { return nil }
    if heap.count == 1 { return heap.removeFirst() }
    /// added for Swift 2 compatibility
    /// so as not to call swap() with two instances of
    /// the same location
    swap(&heap[0], &heap[heap.count - 1])
    let temp = heap.removeLast()
    sink(0)
    return temp
}

/// Removes the first occurrence of a particular item.
/// Finds it by value comparison using ==. O(n)
/// Silently exits if no occurrence found.
/// -parameter item: Item to remove the first occurrence of.
public mutating func remove(_ item: T) {
    if let index = heap.index(of: item) {
        swap(&heap[index], &heap[heap.count - 1])
        heap.removeLast()
        swim(index)
        sink(index)
    }
}

/// Removes all occurrences of a particular item. Finds
/// it by value comparison using ==. O(n)
/// Silently exits if no occurrence found.
/// -parameter item: The item to remove.
public mutating func removeAll(_ item: T) {
    var lastCount = heap.count
```

```
            remove(item)
            while (heap.count < lastCount) {
                lastCount = heap.count
                remove(item)
            }
        }

        /// Get a look at the current highest priority item,
        /// without removing it. O(1)
        /// -returns: The element with the highest priority
        /// in the PriorityQueue, or nil if the PriorityQueue is empty.
        public func peek() -> T? {
            return heap.first
        }

        /// Eliminate all of the elements from the Priority Queue.
        public mutating func clear() {
            #if swift(>=3.0)
                heap.removeAll(keepingCapacity: false)
            #else
                heap.removeAll(keepCapacity: false)
            #endif
        }

        /// Based on example from Sedgewick p 316
        private mutating func sink(_ index: Int) {
            var index = index
            while 2 * index + 1 < heap.count {
                var j = 2 * index + 1
                if j < (heap.count - 1) && ordered(heap[j],
                heap[j + 1]) { j += 1 }
                if !ordered(heap[index], heap[j]) { break }
                swap(&heap[index], &heap[j])
                index = j
            }
        }

        /// Based on example from Sedgewick p 316
        private mutating func swim(_ index: Int) {
            var index = index
            while index > 0 && ordered(heap[(index - 1) / 2],
            heap[index]) {
                swap(&heap[(index - 1) / 2], &heap[index])
                index = (index - 1) / 2
            }
        }
    }
```

The sink and swim methods were inspired by the book Algorithms by Sedgewick and Wayne, Fourth Edition, Section 2.4.

The structure accepts any type that conforms to the `Comparable` protocol. The single initializer allows you to optionally specify the sorting order and a list of starting values. The default sorting order is descending and the default starting values are an empty collection:

```
/// Initialization
var priorityQueue = PriorityQueue<String>(ascending: true)

/// Initializing with starting values
priorityQueue = PriorityQueue<String>(ascending: true, startingValues:
["Coldplay", "OneRepublic", "Maroon 5", "Imagine Dragons", "The Script"])

var x = priorityQueue.pop()
/// Coldplay

x = priorityQueue.pop()
/// Imagine Dragons
```

Protocols

The `PriorityQueue` conforms to sequence, collection, and `IteratorProtocol`, so you can treat it like any other Swift sequence and collection:

```
extension PriorityQueue: IteratorProtocol {
    public typealias Element = T
    mutating public func next() -> Element? { return pop() }
}

extension PriorityQueue: Sequence {
    public typealias Iterator = PriorityQueue
    public func makeIterator() -> Iterator { return self }
}
```

This allows you to use Swift standard library functions on a `PriorityQueue` and iterate through a `PriorityQueue` like this:

```
for x in priorityQueue {
    print(x)
}

// Coldplay
```

```
// Imagine Dragons
// Maroon 5
// OneRepublic
// The Script
```

`PriorityQueue` also conforms to `CustomStringConvertible` and `CustomDebugStringConvertible`. These allow you to return a friendly name when printing out a types value:

```
extension PriorityQueue: CustomStringConvertible,
CustomDebugStringConvertible {
    public var description: String { return
    heap.description }
    public var debugDescription: String { return
    heap.debugDescription }
}
```

To allow `PriorityQueue` to be used as an array using subscript notation, it conforms to protocol:

```
extension PriorityQueue: Collection {
    public typealias Index = Int
    public var startIndex: Int { return heap.startIndex }
    public var endIndex: Int { return heap.endIndex }
    public subscript(i: Int) -> T { return heap[i] }
    #if swift(>=3.0)
    public func index(after i: PriorityQueue.Index) ->
    PriorityQueue.Index {
        return heap.index(after: i)
    }
    #endif
}
```

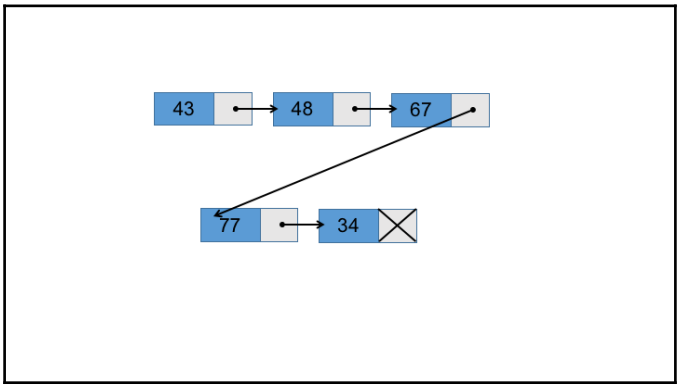
Using subscript notation, you get the following:

```
priorityQueue = PriorityQueue<String>(ascending: true, startingValues:
["Coldplay", "OneRepublic", "Maroon 5", "Imagine Dragons", "The Script"])

var x = priorityQueue[2]
// Maroon 5
```

You can view the most up to date version of `PriorityQueue` in David Kopac's GitHub repository here: `https://github.com/davecom/SwiftPriorityQueue`

StackList

The last data structure we'll cover in this chapter is the linked list. A linked list is an ordered set of elements where each element contains a link to its successor.



Linked list data structure

Linked lists and arrays are similar; they both contain a set of elements. Arrays are allocated in a contiguous range of memory, whereas linked lists are not. This can be an advantage if you have a large dataset you need to work with but you do not know its size ahead of time. Because linked list nodes are allocated individually, they do not allow random access to the elements they contain. If you need to access the fifth element of a linked list, you need to start at the beginning and follow the next pointer of each node until you reach it. Linked lists do support fast insertion and deletion though, *O(1)*.

There are additional linked list types that are useful for different requirements:

- **Doubly linked list** – When you want to be able to walk up and down a linked list. Each node contains two links; a next link that points to its successor, and a previous link that points to its predecessor.
- **Circular linked list** – When you want the last node to point to the beginning of the linked list. With a single or doubly linked list, the end node and first and end nodes respectively have a nil value to signify there are no more nodes. For a circular linked list, the end node's next link will point to the first node in the list.

To demonstrate using a linked list, we will implement a stack data structure that uses a linked list for internal storage. Our StackList will implement the following four methods and one property:

- `push()` – Adds an element to the bottom of a stack
- `pop()` – Removes and returns an element from the top of a stack
- `peek()` – Returns the top element from the stack, but does not remove it
- `isEmpty()` – Returns true if the stack is empty, and false otherwise
- `count` – Returns the number of elements in a stack

Applications

A linked list is generally used to implement other data structures. Instead of using an array in our stack and queue implementations, we could have used a linked list.

If you have a requirement that performs a lot of insertions and deletions and the data has the potential to be very large, consider using a linked list. Since each node in a linked list is not in contiguous memory, indexing a list may not perform well unless the linked list is implemented to optimally manage this. Another disadvantage is a small amount of memory overhead required to manage the links between nodes.

Implementation

The StackList implementation will have the same interface as the `Stack` structure we implemented at the beginning of the chapter.

StackList implements four methods, `push()`, `pop()`, `peek()`, `isEmpty()`, and the `count` property. The backing storage is going to be a linked list though, instead of an array.

We'll begin by defining our `StackList` structure:

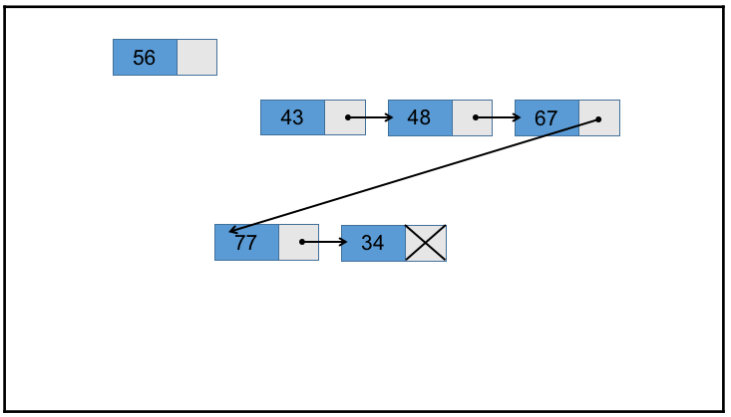
```
public struct StackList<T> {
    fileprivate var head: Node<T>? = nil
    private var _count: Int = 0
    public init() {}

    public mutating func push(element: T) {
        let node = Node<T>(data: element)
        node.next = head
        head = node
        _count += 1
```

```
        }
        public mutating func pop() -> T? {
            if isEmpty() {
                return nil
            }
            // Get the item of the head node.
            let item = head?.data
            // Remove the head node.
            head = head?.next
            // decrement number of elements
            _count -= 1
            return item
        }

        public func peek() -> T? {
            return head?.data
        }

        public func isEmpty() -> Bool {
            return count == 0
        }

        public var count: Int {
            return _count
        }
    }


    private class Node<T> {
        fileprivate var next: Node<T>?
        fileprivate var data: T
        init(data: T) {
            next = nil
            self.data = data
        }
    }
```

As you can see, this implementation is very similar to the `Stack` structure.

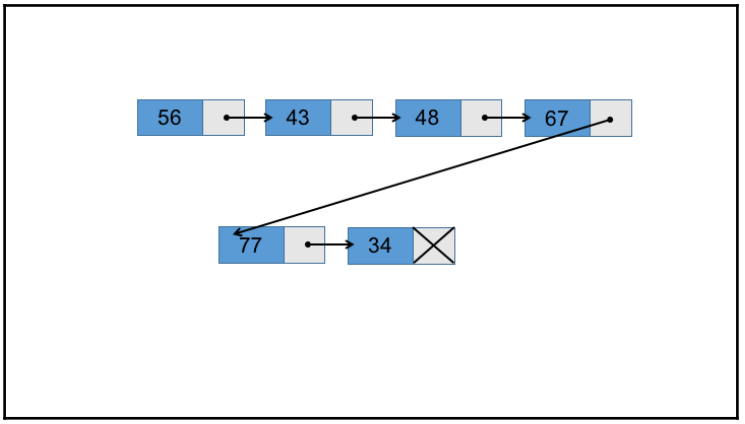
We are using generics for our type, which will be determined at compile time the type that is stored in our StackList. The interesting methods that handle the linked list operations are `push()`, `pop()`, and `peek()`. Let's take a look at each of these methods.

The push method will take a new element, passing it to the `Node` initializer. The `Node.init(_:)` initializer assigns the element to the data property for the new node.



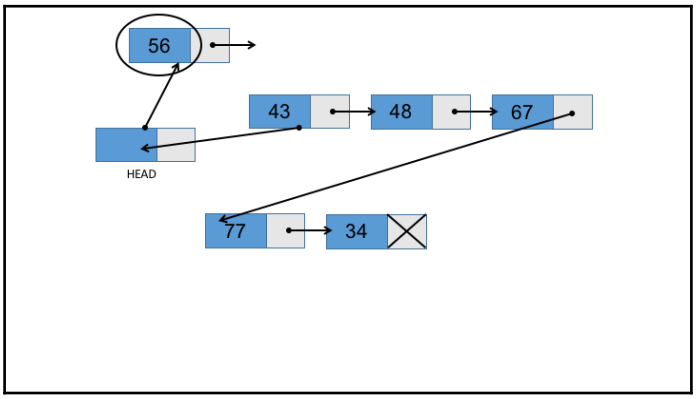
Linked list data structure – new node to insert

Next, we want to insert the new node into the front of our linked list. To do this, we take the existing head node and assign it to the new node's next pointer. Then we assign the new node to the StackLists head pointer and increment the count of elements in our linked list:



Linked list data structure – insert new node

The pop method will return nil if there are no elements. Otherwise, we retrieve the data element from the head node. Since we're returning this element, we want to remove it from the linked list, so we assign the head.next pointer to the `StackList.head` pointer. Then we decrement our count and return the element:



Linked list data structure – remove node

Let's look at a few examples of how we can use our `Stack` structure:

```
var myStack = Stack<Int>()

myStack.push(element: 34)
// [34]
myStack.push(element: 77)
// [77, 34]
myStack.push(element: 67)
// [67, 77, 34]

var x = myStack.pop()
// x = 67
x = myStack.pop()
// x = 77
x = myStack.pop()
// x = 34
x = myStack.pop()
// x = nil
```

Protocols

The StackList is using several of the same protocols as the other data structures we've implemented in this chapter.

The first two we'll add are `CustomStringConvertible` and `CustomDebugStringConvertible` to allow returning a friendly name when printing out a types value:

```
extension StackList: CustomStringConvertible, CustomDebugStringConvertible
{
    public var description: String {
        var d = "["
        var lastNode = head
        while lastNode != nil {
            d = d + "\(lastNode?.data)"
            lastNode = lastNode?.next
            if lastNode != nil {
                d = d + ","
            }
        }
        d = d + "]"
        return d
    }

    public var debugDescription: String {
        var d = "["
        var lastNode = head
        while lastNode != nil {
            d = d + "\(lastNode?.data)"
            lastNode = lastNode?.next
            if lastNode != nil {
                d = d + ","
            }
        }
        d = d + "]"
        return d
    }
}
```

Next, we would like our StackList to behave like an array when we initialize it, so we'll make it conform to `ExpressibleByArrayLiteral`. This will allow us to use angle bracket notation such as `var myStackList = [5, 6, 7, 8]`:

```
extension StackList: ExpressibleByArrayLiteral {
    /// MARK: ExpressibleByArrayLiteral protocol conformance
    /// Constructs a circular buffer using an array literal.
```

```
    public init(arrayLiteral elements: T...) {
        for el in elements {
            push(element: el)
        }
    }
}
```

Next, we'll extend StackList so it conforms to the `IteratorProtocol` and `Sequence` protocols. This will allow us to return an iterator based on the type of element our StackList contains. The `NodeIterator` structure will receive an instance of head on initialization, which we save. This will allow us to iterate through the elements in the linked list returning the next element each time the `next()` method is called:

```
public struct NodeIterator<T>: IteratorProtocol {
    public typealias Element = T
    private var head: Node<Element>?
    fileprivate init(head: Node<T>?){
        self.head = head
    }
    mutating public func next() -> T? {
        if (head != nil) {
            let item = head!.data
            head = head!.next
            return item
        }
        return nil
    }
}

extension StackList: Sequence {
    public typealias Iterator = NodeIterator<T>
    /// Returns an iterator over the elements of this sequence.
    public func makeIterator() -> NodeIterator<T> {
        return NodeIterator<T>(head: head)
    }
}
```

The last addition we'll make is to add another initializer that will allow us to initialize a new stack from an existing one:

```
public init<S : Sequence>(_ s: S) where S.Iterator.Element
== T {
    for el in s {
        push(element: el)
    }
}
```

Now we can do the following:

```
// Use array literal notation
var myStackList = [4,5,6,7]

// Use the new initializer we defined
var myStackListFromStackList = Stack<Int>(myStackList)
// [4, 5, 6, 7]

myStackListFromStackList.push(55)
// [4, 5, 6, 7, 55]

myStackList.push(70)
// [4, 5, 6, 7, 70]
```

You can view the complete code included with this book.

Summary

In this chapter, we've learned about a few of the common Swift protocols you can implement to provide a consistent and friendly development experience for other developers using your data structures. By conforming to these protocols, other developers will intuitively know how to use common Swift language constructs when working with them.

We then learned how to implement a `Stack` structure using an array and linked list. We also learned about queues and implemented several types so you can gain experience for choosing the right type based on the requirements for your application.

By this point you should have the confidence and knowledge to extend the data structures we have implemented, as well as be able to implement your own customized versions of them if they do not suit your needs.

In `Chapter 4`, *Sorting Algorithms*, we're going to build on the data structures we have implemented in this chapter. We'll learn about sorting algorithms and how to implement them with the various data structures we've just learned about.

4
Sorting Algorithms

So far we have learned about different data structures and their performance characteristics. In this chapter, we will begin learning about algorithms, which are the fundamental ways of processing data. Algorithms take a sequence of data as input, process that data, and then return a value or set of values as output.

In this chapter, we discuss sorting algorithms and how to apply them using an array data structure we learned about in the previous chapters. We will explore different algorithms that use comparison sorting and look at both simple sorting and divide and conquer strategies where the entire sorting process can be done in memory. As in `Chapter 2`, *Working with Commonly Used Data Structures*, we place importance on visualizing the algorithms we will learn about in this chapter.

The topics covered in this chapter are:

- Implementing insertion sort algorithm
- Implementing merge sort algorithm
- Implementing quick sort algorithm

We'll first look at the insertion sort that uses a simple sorting strategy. It is best suited for small datasets. The remaining algorithms in this chapter are based on the divide and conquer design pattern, which is a top-down strategy used to implement sorting algorithms. The divide and conquer design pattern uses a recursive algorithm, which takes a given problem, and subdivides it into smaller problems that are similar to the given problem, then solves them independently, combining the independent subproblems and returning the result. There are other strategies such as dynamic programming, greedy, and backtracking but we will not go into details in this book and leave that as a topic for you to do further research on.

The insertion sort

The insertion sort is a simple and popular sorting algorithm. Since it has *O(n2)* average runtime it is very inefficient for sorting larger datasets. However, it is an algorithm of choice when the data is nearly sorted or when the dataset is small. Given those two conditions, it can potentially outperform sorting algorithms that are *O(n log(n))* time complexity, such as merge sort.

The algorithm

The insertion sort algorithm performs in-place sorting and works with any element type that conforms to the comparable protocol. The element type must conform to comparable because we need to compare the individual elements against each other. It will make *N-1* iterations, where *i = 1* through *N-1*. The algorithm leverages the fact that elements 0 through *i-1* have already been put in sorted order.

Let's look at the algorithm:

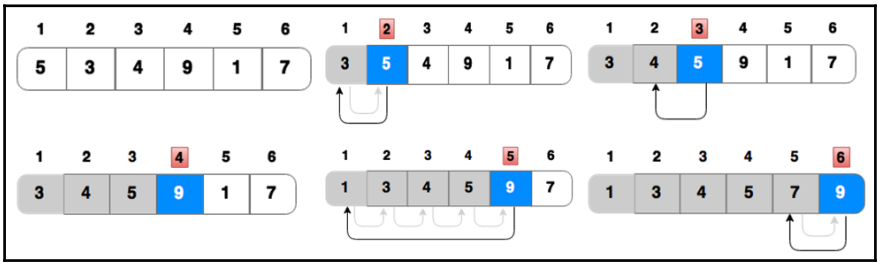
```
1  public func insertionSort<T: Comparable>(_ list: inout [T] ) {
2
3      if list.count <= 1 {
4          return
5      }
6
7      for i in 1..<list.count {
8          let x = list[i]
9          var j = i
10         while j > 0 && list[j - 1] > x {
11             list[j] = list[j - 1]
12             j -= 1
13         }
14         list[j] = x
15     }
16 }
```

Analysis of the insertion sort

On line 1, the `insertionSort` function is defined so that type `T` must conform to the `Comparable` protocol. This is required because we need to compare individual elements of the list array against each other. On line 3, the function returns if we only have one element.

We have nested loops on line 7 and 10 that each perform *N* iterations. Because of this insertion sort is *O(n2)*. On line 7, we iterate from element `i = 1` through `list.count - 1`. On each iteration, the element at index `i` is saved on line 8 so that it can be inserted later into its correct sorted location. On line 9, `j` is initialized to the current `i` index. This allows the algorithm to start the inner loop at the highest sorted index. On line 10, we ensure that `j` is greater than `0` and if the element at index `list[j-1]` is greater than the current element `x`, it is shifted to the right. When an element at `list[j-1]` is found to be less than `x` or `j=0`, we have found the correct position to insert `x` into. This repeats until we've iterated over each element *N* in list.

Given the following input the elements would move as follows:



Use cases of the insertion sort

The insertion sort can be used when you have elements that are mostly sorted and just need changing slightly every now and again.

An example would be a card game. A player receives a hand of cards and puts them in order. When the player is dealt a new card, it is added to their existing hand. The insertion sort can efficiently re-sort that list of cards.

Another example would be if you're designing a 2D game where all sprites are sorted by their *y*-position, and they are drawn over each other. In a list, you can store items like buildings, trees, and rocks that don't move along with mobile items such as people walking or cars moving slowly. You can then perform the insertion sort pass in each frame to iterate through the list, swapping neighboring elements as needed.

Optimizations

Given that the insertion sort is a quadratic algorithm, there isn't much point in trying to optimize it for larger datasets, you're better off looking at another sorting algorithm such as merge sort or quick sort that we'll review next in this chapter.

Merge sort

Merge sort is a divide and conquer algorithm that has a lower order running time than the insertion sort. The merge sort algorithm works by using recursion; it will repeatedly divide an unsorted list into two halves. When the list has a single item or it is empty it is considered sorted; this is called the base case. The majority of the sorting work is performed in the `merge` function, which is responsible for combining the two halves back together. The `merge` function uses a temporary array of equal size to the input array during the merge process so it has a higher order auxiliary space of *O(n)*. Because of this, merge sort is generally better off implemented for sorting a linked list instead of an array. We'll look at both implementations so you can see the performance differences based on the dataset size.

The algorithm for array-based merge sort

There are three steps to the divide and conquer process for sorting a collection. They are:

- **Divide**: If the collection S is zero or one, then return it since it is already sorted. Otherwise split the collection into two sequences, S1 and S2, with S1 containing the first *N/2* elements of S, and S2 containing the remaining *N/2* elements.
- **Conquer**: Recursively sort sublists S1 and S2, if they are small enough then solve their base case.
- **Combine**: Merge the sorted S1 and S2 sublists into a sorted sequence and return the elements back.

We'll first look at an array-based implementation of the merge sort algorithm by examining the `mergeSort` function:

```
1  public func mergeSort<T: Comparable>(_ list: [T]) -> [T] {
2
3      if list.count < 2 {
4          return list
5      }
6
7      let center = (list.count) / 2
```

```
8       return merge(mergeSort([T](list[0..<center])), rightHalf:
        mergeSort([T](list[center..<list.count])))
9   }
```

The `mergeSort` function is called recursively, each time splitting the list in half until it contains zero or one element.

Here is the `merge` function for the array-based implementation:

```
1   private func merge<T: Comparable>(_ leftHalf: [T], rightHalf: [T])
    -> [T] {
2
3       var leftIndex = 0
4       var rightIndex = 0
5       var tmpList = [T]()
6       tmpList.reserveCapacity(leftHalf.count + rightHalf.count)
7
8       while (leftIndex < leftHalf.count && rightIndex <
        rightHalf.count) {
9           if leftHalf[leftIndex] < rightHalf[rightIndex] {
10              tmpList.append(leftHalf[leftIndex])
11              leftIndex += 1
12          }
13          else if leftHalf[leftIndex] > rightHalf[rightIndex] {
14              tmpList.append(rightHalf[rightIndex])
15              rightIndex += 1
16          }
17          else {
18              tmpList.append(leftHalf[leftIndex])
19              tmpList.append(rightHalf[rightIndex])
20              leftIndex += 1
21              rightIndex += 1
22          }
23      }
24
25      tmpList += [T](leftHalf[leftIndex..<leftHalf.count])
26      tmpList += [T](rightHalf[rightIndex..<rightHalf.count])
27      return tmpList
28
29  }
```

The `merge` function is called to combine the two sorted sequences, S1 and S2, back together and return the combined, sorted collection.

Analysis of merge sort

Let's start by first reviewing the `mergeSort` function. On line 1, the `mergeSort` function is defined where type `T` must conform to the `Comparable` protocol. This is required because we need to compare individual elements of the list array against each other. On line 3, if our base case of `list.count < 2` is reached, the list is returned.

On line 7, each time the `mergeSort` function is called, we divide the list length by 2. On line 8 we call the `mergeSort` function recursively, passing the `sublist S[0..<center]` and `S[center..<list.count]` respectively.

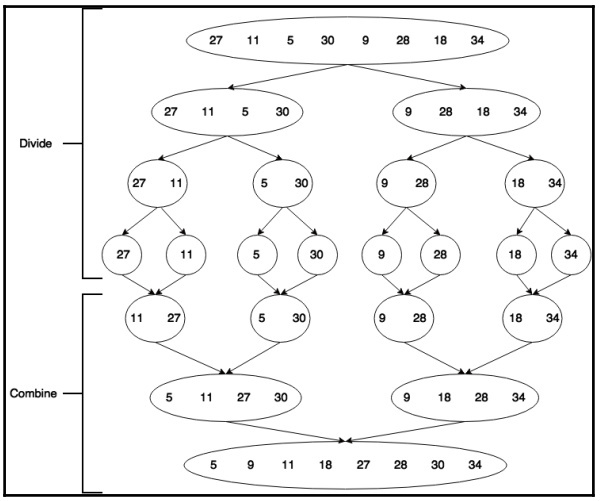
Let's take a look at the `merge` function next. On line 1, the `merge` function is defined where type `T` must conform to the `Comparable` protocol, just like the `mergeSort` function did. The function takes two sublists, `leftHalf` that corresponds to S1, and `rightHalf` that corresponds to S2. Lines 3 and 4 set our initial indexes to zero. On line 5, we create a temporary array that holds the combined sequences in sorted order.

On line 8, we loop through the sequences until either the left or right index are equal to their respective sequence count.

Our first comparison is on line 9; if the `leftHalf` element is less than the `rightHalf`, we add it to the temporary array on line 10, and the left index is also incremented once the element is added. If the `leftHalf` element is greater than the `rightHalf`, we will add the `rightHalf` element to the temporary array, then increment the right index. Otherwise the elements in both arrays are equal, so on line 18, we first add the `rightHalf` element and then on line 19, we add the `rightHalf` element to the temporary array, and both indexes are incremented.

On lines 25 and 26, we add any remaining elements from the `leftHalf` and `rightHalf` arrays, appending them to the temporary array that is returned.

Given the following input, the elements would move as follows:



The algorithm and analysis for linked list-based merge sort

For the linked list version of the merge sort algorithm, we'll slightly modify the `LinkedList` structure we defined in `Chapter 3`, *Standing on the Shoulders of Giants*, so that it exposes the head node property allowing us to directly modify the linked list.

Let's examine the `mergeSort` function first:

```
1  func mergeSort<T: Comparable>(list: inout LinkedList<T>) {
2
3      var left = Node<T>?()
4      var right = Node<T>?()
5
6      if list.head == nil || list.head?.next == nil {
7          return
8      }
9
10     frontBackSplit(list: &list, front: &left, back: &right)
11
12     var leftList = LinkedList<T>()
13     leftList.head = left
14
15     var rightList = LinkedList<T>()
```

```
16      rightList.head = right
17
18      mergeSort(list: &leftList)
19      mergeSort(list: &rightList)
20
21      list.head = merge(left: leftList.head, right: rightList.head)
22  }
```

On line 1, as with the array-based algorithm, the `merge` function is defined where type `T` must conform to the `Comparable` protocol. On lines 3 and 4, we define instances that will point to the sublists that are returned by the `frontBackSplit` function on line 10.

On line 6, with our linked list, since we are directly manipulating the internal linked list, we check the list's head and next pointers to see if they are nil. This is our base case and causes the recursion to end and returns the list. On line 10, we call `frontBackSplit` to divide the current list into two sequences; the function returns the node pointer that points to the first element for each sequence.

On lines 12 through 16, we create two new linked link instances and assign the `left` and `right` sequences respectively. On lines 18 and 19, we recursively call `mergeSort` to continue dividing the list until our base case is reached on line 6. On line 21, our base case is reached and we call `merge` to combine the two list sequences together in sorted order; we then assign the sorted linked list node pointer returned by that function to our lists header pointer.

The `merge` function:

```
1   private func merge<T: Comparable>(left: Node<T>?, right: Node<T>?)
    -> Node<T>? {
2
3       var result: Node<T>? = nil
4
5       if left == nil {
6           return right
7       }
8       else if right == nil {
9           return left
10      }
11
12      if left!.data <= right!.data {
13          result = left
14          result?.next = merge(left: left?.next, right: right)
15      }
16      else {
17          result = right
18          result?.next = merge(left: left, right: right?.next)
```

```
19      }
20
21      return result
22  }
```

On line 1, the `merge` function, we define type `T` so it must conform to the `Comparable` protocol, just like the `mergeSort` function did. The function takes two `Node<T>` references, `left` that corresponds to S1, and `right` that corresponds to S2. On lines 5 through 10, we check if either node reference is nil. If it is we return the opposing half's node reference.

On line 12, we perform a comparison between the linked list node data elements for each half of the sublists. If the left side's value is less than the right, we assign the left node reference to our result. We then call `merge` again, passing the left's node reference to the next node in the linked list. For the right parameter, we simply pass the current, right-half linked list reference. The return value from the `merge` function is assigned to the result's next data reference.

On line 16, if the right data element was greater, we assign the right node reference to our result. We then call `merge` again, passing the current left-half linked list reference, and pass the right node's reference to the next node in the linked list. The return value from the `merge` function is assigned to the result's next data reference. On line 21, the result reference that contains the current sorted and merged sublists is returned.

The `frontBackSplit` function:

```
1  private func frontBackSplit<T: Comparable>( list: inout
   LinkedList<T>, front: inout Node<T>?, back: inout Node<T>?){
2
3       var fast: Node<T>?
4       var slow: Node<T>?
5
6       if list.head == nil || list.head?.next == nil {
7           front = list.head
8           back = nil
9       }
10      else {
11          slow = list.head
12          fast = list.head?.next
13
14          while fast != nil {
15              fast = fast?.next
16              if fast != nil {
17                  slow = slow?.next
18                  fast = fast?.next
19              }
20          }
```

```
21
22          front = list.head
23          back = slow?.next
24          slow?.next = nil
25      }
26  }
```

Since we are working with a linked list, there is a little more work involved to partition our linked list in half. This function will walk through the linked list to divide it in half using the fast/slow pointer strategy.

On line 1, the function is defined using `inout` parameters for the list and left and right node references. This allows those structures to be modified within the `splitList` function and persist once the function returns. On lines 3 and 4, we define `fast` and `slow` node references. These are used when walking through the linked list, where the fast node is advanced twice as fast as the slow node.

Line 6 is similar to the array-based function when we check if the array count is less than two. In the linked list instance, if either the `list.head` or `list.head.next` node are nil, we set the left-half to the value of `list.head`, and the right-half is set to nil. Otherwise, on lines 11 and 12, we set the `slow` and `fast` node references to the first two nodes in the linked list respectively. On line 14, the `while` loop walks through the linked list fast node reference, continually checking it for nil. While it is not nil, the `fast` node is advanced twice for each `slow` node advance.

Once fast contains a nil value, the slow node reference will be just before the midpoint of the list. We use that to split the list in two by assigning the `slow.next` node to the back node reference, then setting the `slow.next` reference to nil. The left node reference is set pointing to the first node in the linked list by assigning the `list.head` node to the left node reference.

Performance comparison

The following table shows the performance characteristics with merge sort between using an array and linked list containing an `Int` type. As you'll see, as the number of elements increases, a linked list performs significantly faster and performance while running within a playground is substantially slower than the compiled code. We will first look at the performance inside a playground, then review the compiled code numbers.

Playground performance between array and linked list data structures

Data Set Size	Array	Linked list
50	0.265550017356873	0.864423990249634
500	8.38627701997757	12.72391396760094
1,000	23.3197619915009	29.36045503616633

Compiled code performance between array and linked list data structures

Data Set Size	Array	Linked list
500	0.0056149959564209	0.00161999464035034
20,000	0.270280003547668	0.101449966430664
40,000	0.567308008670807	0.217732012271881

Quick sort

Quick sort is another divide and conquer algorithm. It is a popular, fast sorting algorithm that can perform sorting in-place, so it is space efficient and has been used as the reference implementation in Java as well as the default library sort function in Unix. The algorithm works by dividing an initial array into two small subsequences, one with the lower sequences and another with the higher sequences, based on the pivot selected by a partitioning scheme. Its average running time is *O(n lg n)*, mainly due to its tight inner loop. It can have a worst case running time of *O(n2)*, but this can be minimized by ensuring the data is in a random order first. Additionally, ensuring that the correct pivot is selected will dramatically affect the algorithm's performance.

The algorithm – Lomuto's implementation

We'll begin our examination of the quick sort algorithm by looking at an implementation by Nico Lomuto, called the Lomuto Partitioning Scheme. This version of the algorithm is a little easier to understand and often taught in introductory computer science classes. The Lomuto algorithm is made up of two functions, `quicksort` and `partition`.

The `quickSort` function's purpose is to call the partition function and then recursively call itself to sort the `lo` and `hi` sides of the array subsequences. The `partition` function is responsible for rearranging the array subsequences in-place; this is the main function of the quick sort algorithm. In order to implement an efficient algorithm, it is important that you properly select the correct pivot value; we'll discuss this in more detail shortly. The Lomuto partitioning scheme always selects the `hi` element as the pivot.

The `quickSort` function is implemented as follows:

```
1  func quickSort<T: Comparable>(_ list: inout [T], lo: Int, hi: Int) {
2
3      if lo < hi {
4
5          let pivot = partition(&list, lo: lo, hi: hi)
6
7          quickSort(&list, lo: lo, hi: pivot - 1)
8          quickSort(&list, lo: pivot + 1, hi: hi)
9
10     }
11 }
```

The partition function is the key part of the quick sort algorithm; this example is using a naïve pivot selection:

```
1  func partition<T: Comparable>(_ list: inout [T], lo: Int, hi: Int)
   -> Int {
2
3      let pivot = list[hi]
4      var i = lo
5
6      for j in lo..<hi {
7          if list[j] <= pivot {
8              swap(&list, i, j)
9              i += 1
10         }
11     }
12     swap(&list, i, hi)
13     return i
14 }
```

Analysis of Lomuto's partitioning scheme

We'll review the `quickSort` function first.

On line 1, the `quickSort` function is defined where type `T` must conform to the `Comparable` protocol. This is required because we need to compare individual elements of the `list` array against each other. On line 3, we check if the `lo` index is less than the `hi` index; if it is, then on line 5, we call the partition function to begin sorting a subsequence of the array based on the current `lo` and `hi` index values. You will see later that the partition function will determine what value to use for the pivot for the recursive calls that follow.

On line 7, the leftmost array subsequences are sorted recursively based on the `pivot` value selected. When the recursion is unrolled, we call `quickSort` again on line 8 to complete sorting of the right array subsequences.

Let's turn our attention now to the `partition` function. The key to a performant quick sort algorithm is in the selection of the pivot. In this first iteration, we've chosen a naïve approach, one that selects the value from the end of the list.

On line 1, we see it has the same function definition as the `quickSort` function. The partition's sole purpose is to select the pivot value and to sort the array subsequences. On line 3, we select the value of the highest index in our list and we use this for comparison between index values later. On line 4, we initialize our lower bound array control index.

On lines 6 through 11, we iterate over our array comparing each element at each index from `lo` to `hi-1` against the pivot value. If the current element is less than the pivot value, we swap it with the current position of `i`, which initially starts at the `lo` index and is incremented each time a swap occurs. By swapping the `i` and `j` values, we push the larger elements to the right and the smaller elements to the left. On line 12, we've completed iterating over the list and now swap the `i` and `hi` elements; this will move our pivot element back in place. On line 13, we return the `i` value as the pivot value from the partition function.

Now that you understand how the quick sort algorithm is implemented using Lomuto's partitioning scheme, let's look at another implementation.

The algorithm – Hoare's implementation

The Lomuto partitioning scheme is a popular introductory algorithm because it's easy to follow, but not highly efficient. Tony Hoare developed the original `quicksort` algorithm in 1959. The Hoare version's partition function is a little more complicated than Lomuto's but it performs three times fewer swaps on average and efficiently creates a partition when an array's values are all equal.

The `quickSort` function is implemented as follows:

```
1  func quickSort<T: Comparable>(_ list: inout [T], lo: Int, hi: Int) {
2
3      if lo < hi {
4
5          let pivot = partition(&list, lo: lo, hi: hi)
6
7          quickSort(&list, lo: lo, hi: pivot)
8          quickSort(&list, lo: pivot + 1, hi: hi)
9
10     }
11 }
```

The partition function is implemented, requiring a minimum number of swaps:

```
1  private func partition<T: Comparable>(_ list: inout [T], lo: Int,
   hi: Int) -> Int {
2
3      let pivot = list[lo]
4      var i = lo - 1
5      var j = hi + 1
6
7      while true {
8          i += 1
9          while list[i] < pivot { i += 1 }
10         j -= 1
11         while list[j] > pivot { j -= 1}
12         if i >= j {
13             return j
14         }
15         swap(&list[i], &list[j])
16     }
17 }
```

Analysis of Hoare's partitioning scheme

We'll quickly review the `quickSort` function since there is only a slight modification over Lomuto's implementation.

On lines 7 and 8, we are creating two sequences of the array, `[lo...pivot]` and `[pivot + 1...hi]`. This is the only difference you'll find within this function.

The partition function is where the major difference occurs so let's turn our attention there.

On line 1, the same function definition is used. On line 3, we select the value of the lowest index, as opposed to the highest index in Lomuto's and we use this for comparison between index values later. On line 4, we store the value of our `lo` index and subtract `1` from it. This value will be used as we iterate over the array searching for elements less than our selected pivot. On line 5, we store the value of our `hi` index and add `1` to it. This value will be used as we iterate over the array searching for elements greater than our pivot.

On line 7, we begin an infinite loop. This loop is terminated once the array index pointers have either met or overlapped each other, at which time we return the index value of `j` that will point to an element of the next higher value from our current pivot. The return statement on line 13 will terminate the infinite while loop and return from the function. On line 15, if our index pointers have not met, we'll swap element `i` with element `j` and then repeat our while loop on line 7.

As you step through the code, you will notice a significant reduction in the number of swaps that occur. As you can see by running the code in the playground, it works and performs faster than the Lomuto partitioning scheme, so what is the problem with this type of partitioning scheme? The selection of our pivot is still naive, instead of selecting the highest array index, we are selecting the lowest. The another issue occurs when you're dealing with sorted or near sorted arrays. Instead of performing with an efficiency of *O(n log n)*, it is reduced to *O(n2)*. Let us look to the next section to understand the impact of not selecting the correct pivot value.

Choice of pivot

As mentioned in the implementation of the initial partition function, it is taking a naive approach by always selecting the highest index element. This can have a negative performance impact depending on how the data is already sorted. In cases where the array is already sorted or nearly sorted, this would produce a worst case complexity of *O(n2)* and this is not what we want. We want to choose a method that will avoid selecting the smallest or largest value. Selecting a random value could be an option but that does not guarantee we will select the best pivot.

Next, we'll examine the different approaches for selecting a pivot.

The wrong way – first or last element

As in the Lomuto partitioning scheme, a common method you will find when someone writes their own quick sort algorithm is they'll take a brute force approach to partitioning by selecting either the first or last element in the list. This approach can be fine as long as the list is in random order, and generally, that is not the case in most real-world implementations.

If the data is already sorted or nearly sorted, you will find that all of the remaining elements will go to either the S1 or S2 subsequences. The performance will be very slow since it will produce a worst-case complexity of *O(n2)*.

The wrong way – select random element

Another common method is to select a random element and use it as the pivot. While this approach is not as bad as selecting the first or last element, it does have its drawbacks. In most instances, this will perform well, even if the data is nearly sorted.

The downside is that using a random number generator is expensive, computationally. You also should ensure that your random number generator does produce random values.

The right way

A better approach to selecting a pivot is to use the **Median of Three** strategy. This will overcome the shortfalls of selecting a pivot randomly or selecting the first or last index. Using this method is much faster than selecting the median of the entire list of elements, and in instances where the data is sorted, it will prevent you from selecting the lowest or highest element.

When selecting the median element, we will also sort the left, right, and center elements. By doing this we know the element at the leftmost position in the subarray is less than our pivot, the rightmost position of our subarray is greater than our pivot.

Improved pivot selection for quick sort algorithm

We'll implement the quick sort algorithm to use the Median of Three strategy to improve overall performance. We'll introduce a new function, `getMedianOfThree`, that is responsible for selecting the pivot and ensuring the selected pivot is the lowest or highest value should the array already be sorted. We then refactor the `partition` method to accept the pivot value.

The new `quickSort` function is implemented as follows and differs slightly from the original by first getting the `median` value. It then calls the partition method to sort the subarrays:

```
1  public func quickSort<T: Comparable>(_ list: inout [T], lo: Int, hi:
   Int) {
2
3      if lo < hi {
4
5          let median = getMedianOfThree(&list, lo: lo, hi: hi)
6          let pivot = partition(&list, lo: lo, hi: hi, median: median)
7
8          quickSort (&list, lo: lo, hi: pivot)
9          quickSort &list, lo: pivot + 1, hi: hi)
10
11     }
12 }
```

The `median` value is then passed to our partition function. Let's take a look at what the `getMedianOfThree` function is doing:

```
1  private func getMedianOfThree<T: Comparable>(_ list: inout [T], lo:
   Int, hi: Int) -> T {
2
3      let center = lo + (hi - lo) / 2
4      if list[lo] > list[center] {
5          swap(&list[lo], &list[center])
6      }
7
8      if list[lo] > list[hi] {
9          swap(&list[lo], &list[hi])
10     }
11
```

```
12      if list[center] > list[hi] {
13          swap(&list[center], &list[hi])
14      }
15
16      swap(&list[center], &list[hi])
17
18      return list[hi]
19  }
```

On line 3, we are computing the center element position. We take into account whether the `hi` index is less than `lo`.

Nearly all binary searches and merge sorts are broken. You will find many example merge sort algorithms available on the Internet and in textbooks. However, the majority of them contain this major flaw. That is why the code on line 3 is so important as it will protect you from this. If you have any code that implements one of these algorithms, fix it now before it blows up. You can read more about the defect here: `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`

For the next three `if` statements, we swap values if the element on the left side is greater than the right. On lines 4 and 5, if the element at index `lo` is greater than the one at index `center`, we swap their positions. On lines 8 and 9, we perform the same steps as above, this time comparing the value at index `lo`, which may have been swapped during the last evaluation, to the element at index `hi`, and swap if the first one is greater.

On lines 12 and 13, repeating the steps from above, we do not compare our `center` index element to that of the element at index `hi`. On line 16, our last swapping operation is where we swap the `center` element with the one at the `hi` element. This process ensures that even if the list was sorted we have unordered it to an extent. On line 18, we return the element value at the `hi` index.

The partition function is unchanged from our Hoare's partitioning scheme.

Optimizations

We could continue to improve our algorithm based on the type of data we're working with. We'll leave these as areas for you to explore on your own, but good examples include the following:

- Switching over to the insertion sort if an array is very small

- Handling repeated elements using a linear-time partitioning scheme, such as the Dutch national flag problem that was proposed by Dijkstra
- Ensuring no more than *O(log n)* space is used by using tail call to recurse the larger side of the partition.

With Java 7, the core runtime switched its default sorting algorithm to use a dual pivot quick sort from the classical quick sort:
`http://epubs.siam.org/doi/pdf/10.1137/1.9781611972931.5`

Summary

In this chapter, we've learned about simple sorting and divide and conquer strategies where the entire sorting process can be performed in memory. You learned about the insertion sort which is good for working with very small datasets. We then moved on to two popular divide and conquer strategy sorting routines: merge sort and quick sort.

By now you should have a good understanding of how to implement these algorithms and customize them based on your specific performance requirements.

In the next chapter, we're going to learn about various tree data structures and algorithms.

5
Seeing the Forest through the Tree

In `Chapter 3`, *Standing on the Shoulders of Giants*, we learned about the most basic data structures and how to build them with Swift; queues, stacks, lists, hash tables, and heaps. Those data structures have helped us to learn the different types of basic data structures available. Right now, you are ready to learn more advanced topics. This chapter is going to introduce a new data structure: the tree.

Trees are a great data structure because, as we will see later, common operations such as search, insertion, and deletion are fast.

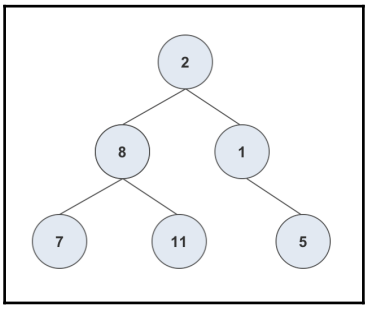
The topics covered in this chapter are as follows:

- Tree – definition and properties
- Overview of different types of tree
- Binary trees
- Binary search trees (BST)
- B-trees
- Splay trees

Tree – definition and properties

A tree is made of a set of nodes. Each node is a data structure that contains a key value, a set of children nodes, and a link to a parent node. There is only one node that has no parent: the root of the tree. A tree structure represents data in a hierarchical form, where the root node is on top of the tree and the child nodes are below it.

The tree has some constraints: a node cannot be referenced more than once, and no nodes point to the root, so a tree never contains a cycle:

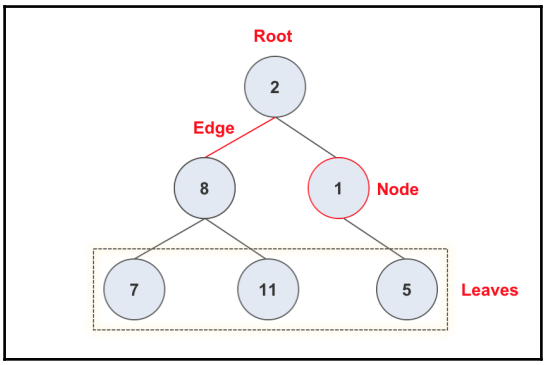


Basic tree data structure

Let's see some important terms when talking about tree data structures:

- **Root**: The node that is on the top of the tree and is the only node in the tree that has no parent.
- **Node**: A data structure that has a value key, and can contain a set of children and a reference to a parent node. If there is no reference to a parent node, the node is the root of the tree. If the node has no children, it is called a leaf.
- **Edge**: Represents the connection between a parent node and a child node.
- **Parent**: The node that is connected to another node and is directly above it in the hierarchy is called the parent of that node. Nodes have only one (or zero) parent.
- **Child**: A node that is connected to another node and is directly below it in the hierarchy is called a child node. Nodes can have zero or multiple children.
- **Sibling**: Nodes with the same parent node are called siblings.
- **Leaf**: A child node that has no further children below it. It is at the bottom of its subtree in the hierarchy. Leaves are also called external nodes. Nodes that are not leaves are called internal nodes.
- **Subtree**: All the descendants of a given node.
- **Height of a node**: The number of edges between a node and the furthest connected leaf. A tree with only the root node has a height of 0.
- **Height of a tree**: This is the height of the root node.
- **Depth**: The number of edges from the node to the root.
- **Level**: The level of a node is its depth + 1.
- **Tree traversal**: A process of visiting each node of a tree once. We will see different traversals later in this chapter.

Now let's review these concepts with the following tree example:



Tree data structure elements

- Root node: [2].
- Nodes: [2, 8, 1, 7, 11, 5].
- Leaves: [7, 11, 5].
- Height: There are two edges between the root [2] and the furthest connected leaf (which could be [7], [11], or [5] with same distance from the root). So the height of the tree is 2.
- Parent example: [8] is the parent of [7, 11].
- Children example: [7, 11] are the children of [8]. [5] is the child of [1].
- Subtrees: Starting in the root node [2], It has two subtrees: one is [8, 7, 11] and another one is [1, 5].
- Height of node [8]: 1.
- Depth of node [5]: 2.
- Level of root node: Depth + 1 = 0 + 1 = 1.

Overview of different types of tree

There are different types of tree data structures, each one with their own benefits and implementations. We are going to have a quick look over the most common ones so we can gain a good overview of the different types and choose which one to use in each case wisely.

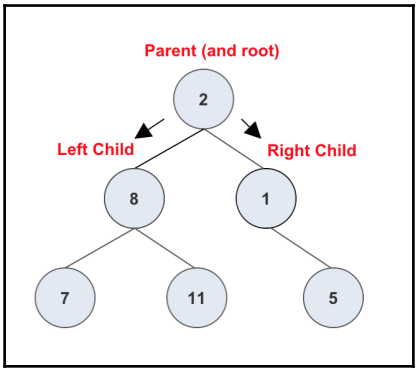
After the following introduction to the different types of trees, we will go deeper into the details, properties, uses, and implementations.

Binary tree

This is the most basic type of tree to start with. A binary tree is a tree data structure in which any node has at most two children.

In a binary tree, the data structure needs to store values and references inside each node, contain a value key, and potentially, a reference to the parent (except the root), a left reference to a child, and a right reference to another child.

When a node doesn't have a parent, a left child, or a right child, the reference to that element exists, but it contains NULL/nil value.



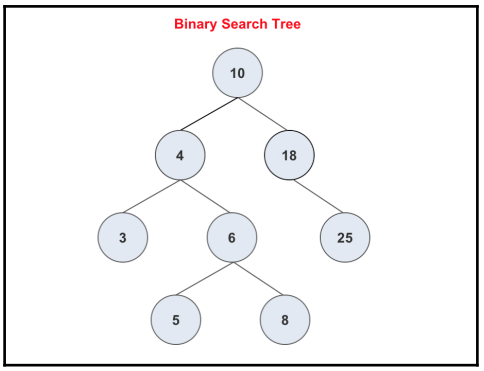
Binary tree data structure

Binary search tree

The binary search tree is a binary tree that fulfills the following conditions for every single node in the tree:

Given a node P in the tree:

- For every node L in the left subtree: L.value < P.value
- For every node R in the right subtree: R.value >= P.value



Binary search tree

This means that for every child node in the left subtree of a parent node, the key value of the children is always less than the parent key value. And, for every child node in the right subtree of a parent node, the key value of the children is always greater than the parent key value.

This property makes the binary search tree very efficient and useful in specific conditions, due to the fact that every node has at most two children and that it maintains a known order in the values of the subtrees. We will learn about this later in the chapter.

B-tree

B-trees are similar to balanced binary search trees but with one difference: B-trees can have more than two children per node.

As we will see later, B-trees are a common choice as a data structure for database systems and secondary file storage. B-trees are fast for processing large sets of data, which makes them useful for those scenarios.

Splay tree

Splay trees are a specific type of binary search tree with an additional benefit: recently accessed nodes are moved to the top of the tree. This property considerably reduces the time needed to access recently visited nodes on later occasions.

In order to accomplish this, splay trees adjust themselves after every access to a node. There is a process called **splaying** that uses tree rotations to rearrange the nodes and put the last accessed one on top of the tree.

So, we can say that splay trees optimize themselves in order to provide quick access to the most recently visited nodes.

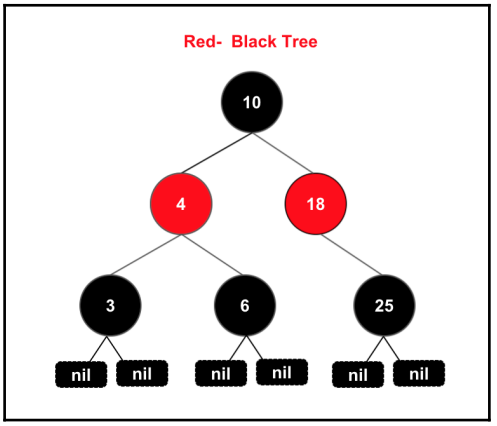
Red-black tree

Red-black trees are self-balancing binary search trees with a new parameter for every node; the color of that node.

The color of the node can be either red or black. So, the data structure needed for red-black tree nodes contains a key value, a color, the reference to a parent, and the references for the left and the right child.

Red-black trees need to satisfy the following color conditions:

- Every node must have a color: red or black
- The root is black
- All the NULL/nil leaves are black
- For any red node, both children are black
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.



Red-black tree

Red-black trees offer worst-case guarantees for key operations such as search, insertion, and deletions. This makes them a great candidate to be used in real-time processes and applications where having a known worst-case scenario is very useful.

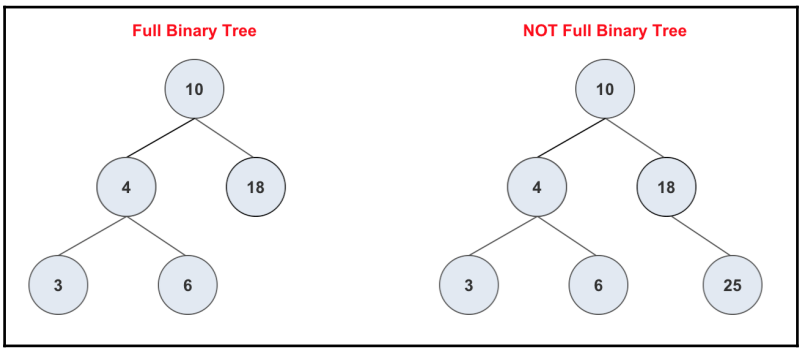
Binary trees

As we have seen before, a binary tree consists of a tree in which the maximum number of children per node is two. This property ensures us that every node has a finite number of children. Moreover, we can assign them known references, left and right children.

Types and variations

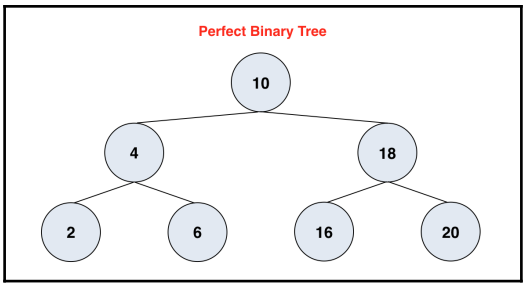
Before getting into the Swift implementation, let's define some different types of binary tree:

- **Full binary tree:** When for every node N in the tree, N has zero or two children (but never one).



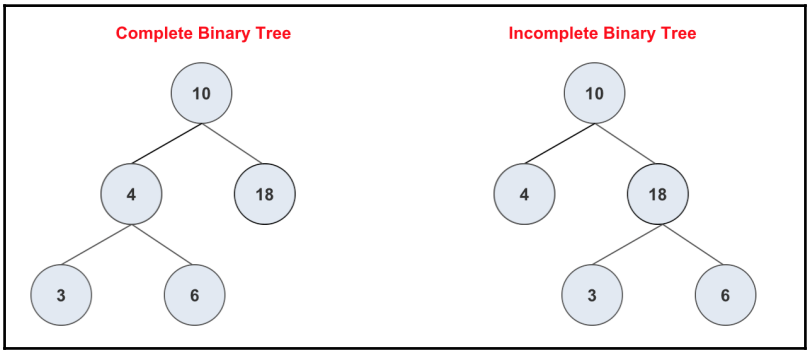
Full binary tree compared to a not full binary tree

- **Perfect binary tree**: All interior nodes have two children. All leaves have the same depth.



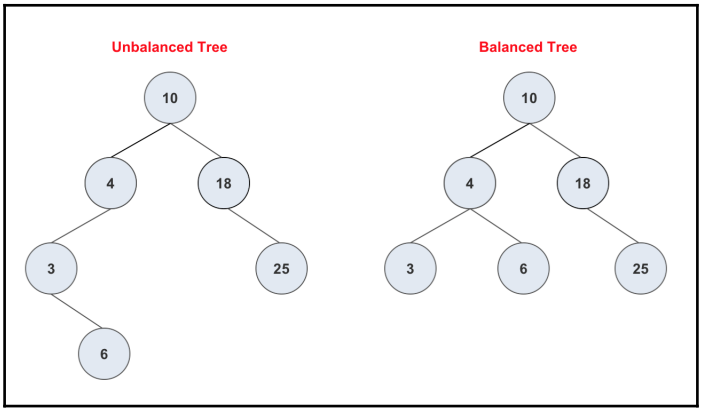
Perfect binary tree

- **Complete binary tree:** All levels are 100% filled by nodes except the last one, which can be not fully completed but in which the existent nodes are in the left side of the tree.



Complete binary tree

- **Balanced binary tree:** It has the minimum possible height for the leaf nodes.



Balanced binary tree

Code

For a binary tree implementation, the data structure needed to form a single node must have at least the following elements:

- A container for the key data value
- Two references to optional left and right children nodes
- A reference to a parent node

So, let's define a class that fulfills these requirements. In Xcode, go to **File** | **New** | **Playground**, and call it `B05101_05_Trees`. In the `Sources` folder, add a new file called `BinaryTreeNode.swift`. Add the following code inside it:

```
public class BinaryTreeNode<T:Comparable> {
    //Value and children vars
    public var value:T
    public var leftChild:BinaryTreeNode?
    public var rightChild:BinaryTreeNode?
    public weak var parent:BinaryTreeNode?

    //Initialization
    public convenience init(value: T) {
        self.init(value: value, left: nil, right: nil, parent:nil)
    }
    public init(value:T, left:BinaryTreeNode?, right:BinaryTreeNode?,
    parent:BinaryTreeNode?) {
        self.value = value
        self.leftChild = left
        self.rightChild = right
        self.parent = parent
    }
}
```

So, at this point, we have the base class to conform a binary tree, `BinaryTreeNode`. We have made it generic `(<T>)` in order to allow any kind of value type inside the value property.

Our `BinaryTreeNode` class has a variable `value` to store the key data. Moreover, it has two `BinaryTreeNode` variables to store the left and the right children, and a reference to a parent node.

We added to the class 2 initializers, one with all the variables as parameters and the `convenience init` just with the mandatory one (param `value`).

Now, let's implement some operations such as inserting a new node or searching for a specific value. We are going to implement them in the next section, in the context of a binary search tree to make things more interesting.

Binary search trees

Binary search tree basic operations such as access, search, insertion, and deletion take between *O(n)* and *O(log(n))* time. Being both values the worst and the average scenarios. At the end, these times are going to depend on the height of the tree itself.

For example, for a complete binary search tree with *n* nodes, these operations could take *O(log(n))* time. But if a tree with the same number of nodes *n* is built like a linked list (just 1 child per node), having more levels/depth for the same *n* nodes, then the operations are going to take *O(n)* time.

In order to make basic operations such as insertion or search, we need to scan nodes from the root to the leaves. Because of this, we can infer that the height of the tree (the distance or nodes between the root and a leaf) will affect the time we spend performing basic operations.

Now, before jumping into the code of some operations, such as inserting and searching nodes in a binary search tree, lets recall the basic property.

Given a node P in the tree:

- For every node L in the left subtree: L.value <= P.value
- For every node R in the right subtree: R.value >= P.value

Inserting a node

With the binary search tree property in mind, let's implement a process that inserts a new node into the tree while maintaining the above binary search tree property.

In order to maintain the property, we must insert the new node recursively starting from the top of the tree (root node), and then going down to the left or right depending on the value of the node to insert.

To prevent the user from inserting the new node in the middle of the tree, breaking the binary search tree property by mistake, we will make sure that the user is always calling the `public func insertNodeFromRoot` that is taking care of it. Add this function to the `BinaryTreeNode` class:

```
public func insertNodeFromRoot(value:T) {
    // To maintain the binary search tree property, we must ensure that
    // we run the insertNode process from the root node
    if let _ = self.parent {
        // If parent exists, it is not the root node of the tree
        print("You can only add new nodes from the root node of the tree");
        return
    }
    self.addNode(value: value)
}
```

After ensuring that the new node is being inserted starting from the root node, we use `private func addNode`, which will recursively walk through the rest of the nodes to insert the new one in the proper position. Add the following function to the class:

```
private func addNode(value:T) {
    if value < self.value {
        // Value is less than root value: We should insert it
        // in the left subtree.
        // Insert it into the left subtree if it exists, if not,
        // create a new nodeand put it as the left child.
        if let leftChild = leftChild {
            leftChild.addNode(value: value)
        } else {
              let newNode = BinaryTreeNode(value: value)
              newNode.parent = self
              leftChild = newNode
          }
    } else {
          // Value is greater than root value: We should insert it
          // in the right subtree
          // Insert it into the right subtree if it exists, if not,
          // create a new node and put it as the right child.
          if let rightChild = rightChild {
              rightChild.addNode(value: value)
          } else {
                let newNode = BinaryTreeNode(value: value)
                newNode.parent = self
                rightChild = newNode
            }
      }
}
```

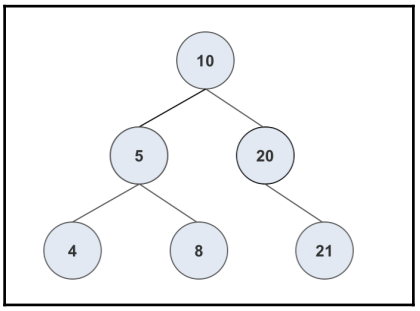
As the code shows, we start analyzing the current node value against the value of the new node. If the value of the new node is less than the value of the current node, we are going to call the same method, but within the left child of the current node. Otherwise, we will make the same, but within the right child.

In both cases, if the left or the right child doesn't exist at all, we will create a new child with the new value and we set it as the child itself.

You can test this code by creating a tree like this inside the playground file:

```
let rootNode = BinaryTreeNode(value: 10)
rootNode.insertNodeFromRoot(value: 20)
rootNode.insertNodeFromRoot(value: 5)
rootNode.insertNodeFromRoot(value: 21)
rootNode.insertNodeFromRoot(value: 8)
rootNode.insertNodeFromRoot(value: 4)
```

If you inspect the `rootNode` and its children in Xcode debugger, the resulting tree should look like this:



Resulting binary search tree

Tree walks (traversals)

Now that we have built a binary search tree, let's see some techniques to walk through it and visit each node in a particular order.

Inorder tree walk

Due to the binary search tree property, we can run the following algorithm, called **inorder** tree walk or inorder traversal, where the result is going to be the list of all the node values of the tree sorted in ascending order.

The algorithm will use recursion to visit each subtree of the root node in the following order:

- First the left subtree—then the node value—then the right subtree.

Knowing that in a BST:

- Left value < node value < right value

This will cause a sequence of ascending ordered values as the output. Add this class function to our `BinaryTreeNode` class:

```
//In-order tree walk with recursion
    public class func traverseInOrder(node:BinaryTreeNode?) {
        // The recursive calls end when we reach a Nil leaf
        guard let node = node else {
            return
        }

        // Recursively call the method again with the leftChild,
        // then print the value, then with the rigthChild
        BinaryTreeNode.traverseInOrder(node: node.leftChild)
        print(node.value)
        BinaryTreeNode.traverseInOrder(node: node.rightChild)
    }
```

Test it yourself with the created BST that you generated before. To run it, add this call at the end of the playground file:

```
BinaryTreeNode.traverseInOrder(node:rootNode)
```

The result should be as follows:

```
4
5
8
10
20
21
```

Looks good! There we have the nodes of the tree in ascending order.

In a similar way, let's see another two variants: **preorder** and **postorder**.

Preorder tree walk

The preorder algorithm visits a subtree in the following order: value—left subtree—right subtree.

So this time, pre means that we visit the node itself before the children. Let's see the implementation and the results. Add this new class function to the `BinaryTreeNode` class:

```
//Pre-order tree walk with recursion
    public class func traversePreOrder(node:BinaryTreeNode?) {
        //The recursive calls end when we reach a Nil leaf
        guard let node = node else {
            return
        }

        // Recursively call the method printing the node value and
        // visiting the leftChild and then the rigthChild
        print(node.value)
        BinaryTreeNode.traversePreOrder(node: node.leftChild)
        BinaryTreeNode.traversePreOrder(node: node.rightChild)
    }
```

To execute it, add this call at the end of the playground file:

```
BinaryTreeNode.traversePreOrder(node:rootNode)
```

Results (with the same BST tree that we generated before):

```
10
5
4
8
20
21
```

This type of tree walk is useful when we want to duplicate a BST node by node. Notice from the results how we can start cloning the tree from top to bottom left and then to the right, node by node.

Postorder tree walk

The post algorithm visits a subtree in the following order: left subtree—right subtree—value. This tree walk is useful when we want to delete a tree. You will notice in the output that we are going to print the nodes starting from the bottom-left and going right and up. In this way, we can delete all the references (if needed) without skipping a single node, which could waste memory space in some languages. Add a new class function to the `BinaryTreeNode` class:

```
//Post-order tree walk with recursion
    public class func traversePostOrder(node:BinaryTreeNode?) {
        //The recursive calls end when we reach a Nil leaf
        guard let node = node else {
            return
        }

        // Recursively call the method visiting the leftChild and
        // then the rigthChild, ending with the value of the node itself
        BinaryTreeNode.traversePostOrder(node: node.leftChild)
        BinaryTreeNode.traversePostOrder(node: node.rightChild)
        print(node.value)
    }
```

Run it with this call at the end of the playground file:

```
BinaryTreeNode.traversePostOrder(node:rootNode)
```

See the results:

```
4
8
5
21
20
10
```

The result is a list of nodes from bottom-left to right, and up until the root.

Searching

So, we already know how to build a binary search tree by inserting nodes in the correct order from the top and we also know how to visit the tree nodes in different orders recursively. Now let's learn how to perform a basic search.

Let's implement a method that searches for a node that contains a specific value as a key. If it doesn't exist, it will return nil inside an optional `BinaryTreeNode` object. Add this function to the `BinaryTreeNode` class:

```
public func search(value:T) -> BinaryTreeNode? {
    // If we find the value
    if value == self.value {
        return self
    }

    // If the value is less than the current node value ->
    // recursive search in the left subtree. If is bigger,
    // search in the right one.
    if value < self.value {
        guard let left = leftChild else {
            return nil
        }
        return left.search(value: value)
    } else {
          guard let right = rightChild else {
                  return nil
      }
    return right.search(value: value)
    }
}
```

Now test the function. Try to search for a non-existent value and for a value that does exist in the tree. At the end of the playground file, add the following:

```
//Search calls
print("Search result: " + "\(rootNode.search(value: 1)?.value)")
print("Search result: " + "\(rootNode.search(value: 4)?.value)")
```

As the result, you will get a nil value and an actual value:

```
Search result: nil
Search result: Optional(4)
```

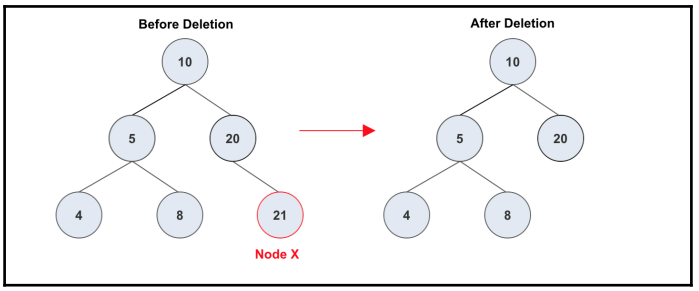
As we mentioned earlier, a search in a binary search tree is going to have a worst-case scenario of *O(log(n))* time for completed trees (or nearly completed trees) and *O(n)* for linear ones (similar to linked lists), where *n* is the height of the tree.

Deletion

Deletion is more complicated to implement than insert or search operations. The reason is that we must address different scenarios when we want to delete a node in the tree. Let's see some examples, and later, we will proceed to implement them.

Let's define the node x as the node to be deleted. The scenarios to analyze are as follows:

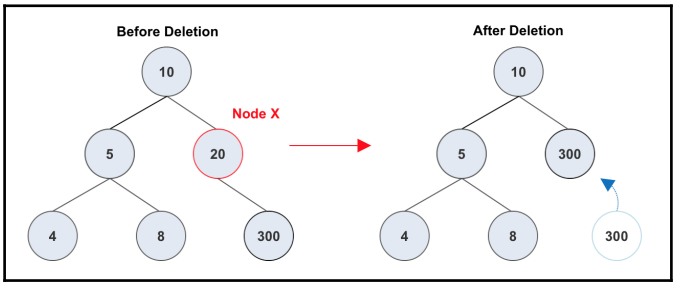
- When x has no children:



Binary search tree deletion of a node without children

In this scenario, we just assign the reference to nil, that the parent node has to node x. In this way, we are deleting x from the tree (no node has a reference to it anymore)

- When x has only one child:



Binary search tree deletion of a node with one child

In this case, we will use the reference to node x from its parent, to point to the child of node x. So, in the preceding example, our node x, whose value is **20**, was the right child of the node with value 10. Now, the right child reference of the node with the value **10** is going to point to the only child of node x, in our case, the node with value **300**.

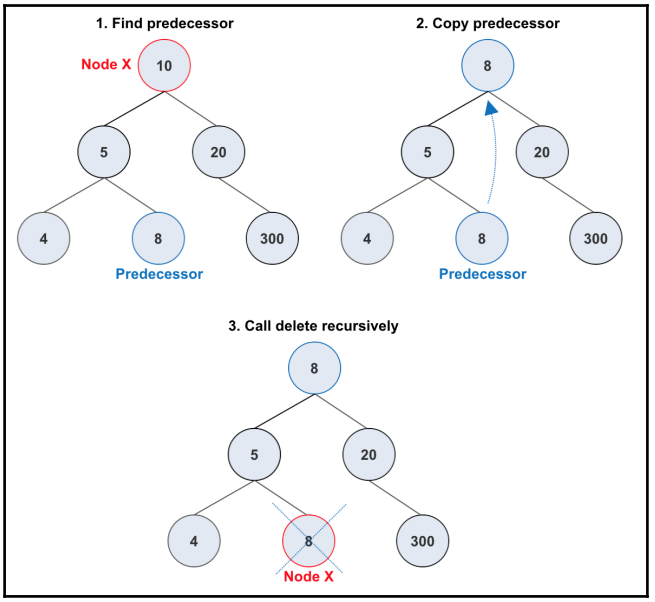
- When x has both children.

  This last scenario is a little more complicated. If we want to delete a node with both children, we need to take the following steps:

  1. Find the smallest child that is greater than its value (in-order successor), which is going to be the minimum value of its right subtree. In the same way, we can also use the biggest child that is less than its value (in-order predecessor), which will be the maximum of its left subtree.

  2. Move the position in the tree of that successor/predecessor to the position of the node to delete.

  3. Call the deletion process recursively in the successor/predecessor.



Binary search tree deletion of a node with two children

Let's see how the implementation looks. Add the following function to our `BinaryTreeNode` class:

```
//Deletion
    public func delete() {
        if let left = leftChild {

            if let _ = rightChild {
                // The node has 2 Children: left and right ->
                // Exchange with successor process
                self.exchangeWithSuccessor()
            } else {
                  // The node has 1 child (left) -> Connect
                  // self.parent to self.child. We need to know if
                  // the node to remove was the left of the right
                  // child of the parent first
                  self.connectParentTo(child: left)
              }

        } else if let right = rightChild {
              // The node has 1 child (right) -> Connect
              // self.parent to self.child. We need to know if
              // the node to remove was the left of the right child
              // of the parent first
              self.connectParentTo(child: right)
          } else {
                self.connectParentTo(child: nil)
            }

        // Delete node references
        self.parent = nil
        self.leftChild = nil
        self.rightChild = nil
    }

// Help us to exchange a node to be deleted for its successor
    private func exchangeWithSuccessor() {
        guard let right = self.rightChild , let left = self.leftChild
        else {
            return
        }
        let successor = right.minimum()
        successor.delete()

        successor.leftChild = left
        left.parent = successor

        if right !== successor {
```

```
            successor.rightChild = right
            right.parent = successor
        } else {
            successor.rightChild = nil
        }

        self.connectParentTo(child: successor)
    }

    private func connectParentTo(child:BinaryTreeNode?) {
        guard let parent = self.parent else {
            child?.parent = self.parent
            return
        }
        if parent.leftChild === self {
            parent.leftChild = child
            child?.parent = parent
        }else if parent.rightChild === self {
            parent.rightChild = child
            child?.parent = parent
        }
    }
```

We need to add some helper methods:

```
//MARK: Other methods
//Minimum value of the tree
    public func minimumValue() -> T {
        if let left = leftChild {
            return left.minimumValue()
        }else {
            return value
        }
    }

//Maximum value of the tree
    public func maximumValue() -> T {
        if let right = rightChild {
            return right.maximumValue()
        }else {
            return value
        }
    }

//Minimum node of the tree
    public func minimum() -> BinaryTreeNode {
        if let left = leftChild {
            return left.minimum()
```

```
            }else {
                return self
            }
        }

    //Maximum node of the tree
        public func maximum() -> BinaryTreeNode {
            if let right = rightChild {
                return right.maximum()
            }else {
                return self
            }
        }

    //Height
        public func height() -> Int {
            if leftChild == nil && rightChild == nil {
                return 0
            }
            return 1 + max(leftChild?.height() ?? 0, rightChild?.height() ?? 0)
        }

    //Depth
        public func depth() -> Int {
            guard var node = parent else {
                return 0
            }
            var depth = 1
            while let parent = node.parent {
                depth = depth + 1
                node = parent
            }
            return depth
        }
```

You can test deletion by adding this code to the end of the playground file:

```
//Deletion
rootNode.leftChild?.delete()
rootNode.rightChild?.delete()
BinaryTreeNode.traverseInOrder(node:rootNode)
```

We have built a basic binary tree structure in Swift. Then, we implemented basic operations: insertion and search. Last, we have implemented the three different scenarios to handle the deletion of a node from a binary search tree. Let's jump into a different type of tree structure now, B-trees.

B-trees,

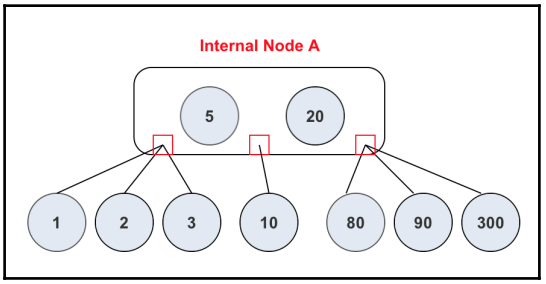
B-trees are similar to binary search trees in many ways, but they have two big differences: the number of children per node is not limited to two and the number of keys in the node is also variable (not just 1).

B-trees are self-balanced, rooted, sorted trees. They allow operations such as insert, search, deletion, and access in logarithmic time.

Each internal node has *n* keys. These keys are like dividing points between child nodes. So, for *n* keys, the internal node has *n+1* child nodes.

This feature makes B-trees suitable for different applications in fields such as databases and external storage. Having more than two children per node and multiple keys allows the B-tree to perform multiple comparisons for each internal node, so it has less tree height and therefore reduces the time complexity to access and search nodes.

As has been said, each internal node in a B-tree has a different number of keys inside. These keys are used to divide the subtrees below them in order. Look at the following example:



B-tree example

Internal node A has three children (subtrees). In order to separate them, it needs two keys [5, 20]. As you can imagine, it is easy to search for a specific node when you have a well-organized structure like this.

Let enumerate some mathematical properties for B-trees.

For a B-tree of order X:

- The root node may have 1 value and 0-2 children
- The other nodes have the following:
    - x/2 to x-1 ordered keys
    - x/2-1 to x children (subtrees)

- The worst height case is *O(log(n))*
- All the leaves have the same depth, which is the height of the tree

Splay trees

Splay trees are a specific type of binary search tree in which there is an operation called splaying, which grants the tree the ability to quickly access recently visited nodes.

The splay operation puts the last accessed node as the new root of the tree. Recently visited nodes always have a minimum height, therefore they are easy and quick to access again. We can say that splay trees optimize themselves by performing a mix of searches and tree rotations.

The average height is *O(log(n))* and the worst (and most unlikely) scenario is *O(n)*. The amortized time of each operation on a n-node tree is *O(log(n))*. The amortized time analysis is used when we don't always expect the worst scenario so we can consider different scenarios (not just the worst) in the overall time complexity of the algorithm.

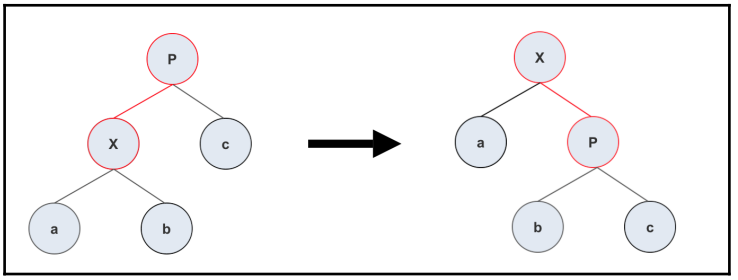
Two common uses of splay trees are caches and garbage collections. In both cases, we get the benefits of quick access to recently visited nodes, so this particular implementation of the binary search tree fits perfectly in both scenarios.

Splay operation

The splay operation has three different methods to move the node upwards to the root. At the same time as the splay operation does this, it also tries to make the tree more balanced by putting every node as close as possible to the root along the path of the splayed node. Let's see the three different splay operation types, assuming that we want to splay node x with parent p and grandparent g (if exists).

Simple rotation or zig

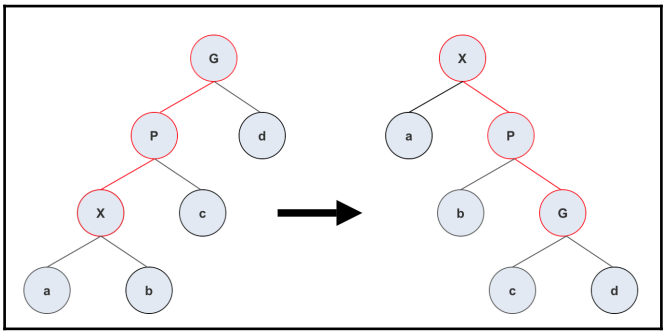
This step happens when **X** is the child of the root, when its parent **P** is the root of the tree. The tree is rotated in the edge from **X** to **P**:



B-tree zig rotation

Zig-Zig or Zag-Zag

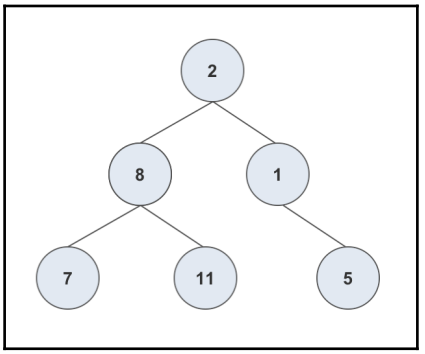
This is used when **P** is not the root and **X** and **P** are both right children or both left children. The tree is rotated about the edge from **P** to **G** and then about the edge from **X** to **P**. We have Zig-Zig and Zag-Zag rotations when we perform two consecutive rotations to the right, or two consecutive rotations to the left:



B-tree Zig-Zig rotation

Zig-Zag

This is used when **P** is not the root and **X** is the left child and **P** is the right child, or vice versa. First, we rotate the tree about the edge between **P** and **X** and then about the resulting edge between **G** and **X**:



B-tree Zig-Zag rotation

Summary

In this chapter, we've learned about a new data structure that you can add to the basic ones that you already know and use in your own project: the tree data structure, including the basic one and other types, such as binary search tree, B-tree, and splay tree. We also introduced red-black trees, which we will see in `Chapter 6`, *Advanced Searching Methods.*

We have seen how trees work, when they are useful, what type of tree is better depending on the problem to solve, and how to implement the most common one, the binary search tree.

Moreover, we have seen the basic operations and how to implement them in Swift: insert, search, and delete operations.

By the end of the chapter, we have reviewed the general characteristics of other types of tree such as B-trees and splay trees, both used in very specific situations. In the next chapter, we are going to go further and view more advanced tree structures.

6
Advanced Searching Methods

In `Chapter 5`, *Seeing the Forest through the Tree*, we introduced the tree data structure and some of its variants. After seeing the binary search trees, we made a quick overview of other types of advanced trees such as red-black and AVL trees. During this chapter, we are going to dive deeper and learn about trees that allow us to make advanced search methods.

The topics covered in this chapter are as follows:

- Red-black trees
- AVL trees
- Trie trees (Radix trees)
- A look at several substring search algorithms

Red-black trees

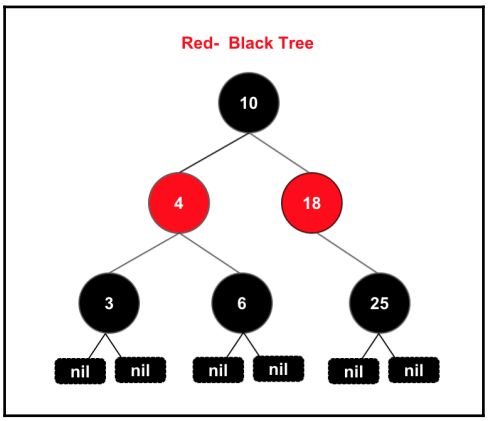
Red-black trees are similar to binary search trees with a new parameter for every node, the color of that node.

The color of the node can be either red or black. So, the data structure needed for red-black tree nodes contains a key value, a color, the reference to a parent, and the references to the left and right child.

Red-black trees need to satisfy the following color conditions:

1. Every node must have a color red or black
2. The root is black
3. All the NULL/nil leaves are black
4. For any red node, both children are black

5. For each node, all simple paths from the node to descendant leaves contains the same number of black nodes



Red-black tree data structure

Because of color condition number five (see the preceding figure), red-black trees offer worst-case guarantees for key operations such as search, insertion, and deletions that are proportional to its tree height. Unlike regular binary search trees, this makes red-black trees a great candidate to be used in real-time processes and applications where having a known worst-case scenario is very useful.

Red-black trees offer the following time and space complexities:

- **Search**: Average and worst time complexity = *O(log(n))*
- **Insertion**: Average and worst time complexity = *O(log(n))*
- **Deletion**: Average and worst time complexity = *O(log(n))*
- **Space**: Average and worst case = *O(n)*

n is the number of nodes in the tree.

Red-black tree node implementation

So now that we know the main characteristics of red-black trees, let's see how to implement the base class.

Then, we will progress to the insertion scenario, which is a little bit more complex than in the binary search tree case, because this time we have to maintain the five color conditions after the insertion of a new node in the tree.

Let's see the first version of the `RedBlackTreeNode` class and the `RedBlackTreeColor` enumeration that will help us with the color.

In Xcode, go to **File | New | Playground**, and call it `B05101_6_RedBlackTree`. In the `Sources` folder, add a new file called `RedBlackTreeNode.swift`. Add the following code inside it:

```
//Enumeration to model the possible colors of a node
public enum RedBlackTreeColor : Int {
    case red = 0
    case black = 1
}

public class RedBlackTreeNode<T:Comparable> {
    //Value and children-parent vars
    public var value:T
    public var leftChild:RedBlackTreeNode?
    public var rightChild:RedBlackTreeNode?
    public weak var parent:RedBlackTreeNode?
    //Color var
    public var color:RedBlackTreeColor

    //Initialization
    public convenience init(value: T) {
        self.init(value: value, left: nil, right: nil, parent:nil,
        color: RedBlackTreeColor.black)
    }

    public init(value:T, left:RedBlackTreeNode?,
    right:RedBlackTreeNode?, parent:RedBlackTreeNode?,
    color:RedBlackTreeColor) {
        self.value = value
        self.color = color
        self.leftChild = left
        self.rightChild = right
        self.parent = parent
    }
}
```

We defined a basic enumeration `RedBlackTreeColor` to help us model the two possible colors that a node can have: red and black.

We also defined the basic class to handle a node in the red-black tree: `RedBlackTreeNode`, which can contain the value, the children references, a parent reference, and a color.

When we initialize a red-black tree, we are going to use the `init` method, so the default color for the root is black (remember: the second color condition).

We are going to add two helper methods that are going to be very useful later in more complex operations. These methods help us accessing the potential uncle and the potential grandparent of a node. The uncle of a node is the sibling of its parent. The grandparent is the parent of its parent, as in the real world. Add these methods inside the `RedBlackTreeNode` class:

```
//MARK: Helper methods
//Returns the grandparent of the node, or nil
public func grandParentNode() -> RedBlackTreeNode? {
    guard let grandParentNode = self.parent?.parent else {
        return nil
    }
    return grandParentNode
}
// Returns the "uncle" of the node, or nil if doesn't exist. This is the
// sibling of its parent node
public func uncleNode() -> RedBlackTreeNode? {
    guard let grandParent = self.grandParentNode() else {
        return nil
    }
    if parent === grandParent.leftChild {
        return grandParent.rightChild
    }else {
        return grandParent.leftChild
    }
}

// Prints each layer of the tree from top to bottom with the node value
// and the color
public static func printTree(nodes:[RedBlackTreeNode]) {
    var children:[RedBlackTreeNode] = Array()

    for node:RedBlackTreeNode in nodes {
        print("\(node.value)" + " " + "\(node.color)")
        if let leftChild = node.leftChild {
            children.append(leftChild)
        }
        if let rightChild = node.rightChild {
            children.append(rightChild)
        }
    }

    if children.count > 0 {
        printTree(nodes: children)
    }
}
```

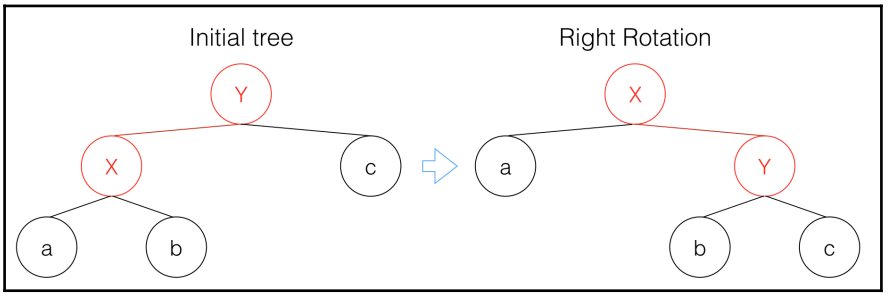
Rotations

Now, let's see the process that helps the red-black tree to balance (and therefore to maintain some of its color conditions): tree rotations.

Tree rotation is a mechanism that moves nodes of the tree to a different place in order to change the height of some of the nodes (and make it uniform among all the children). Let's see two different scenarios that we are going to use later in the insertion process: right rotation and left rotation.

Right rotation

We use a rotation to the right in the following scenario:



Right rotation in red-black trees

Here are the steps for right rotation:

1. Node **X** goes up to become the root of the new tree after the rotation (on the right side of the figure). Node **Y**, which was the parent of **X**, is now the right child (its value is greater, so it must be on the right subtree).
2. If node **Y** had a parent, we now assign that parent to node **X**.
3. The right child of node **X** is now the left child of its child node, **Y**.

Now, let's see how to implement this in Swift. Add a new method called `rotateRight()` to the `RedBlackTreeNode` class**:**

```
//MARK: Rotations
//Right
public func rotateRight() {
    guard let parent = parent else {
        return
    }
```

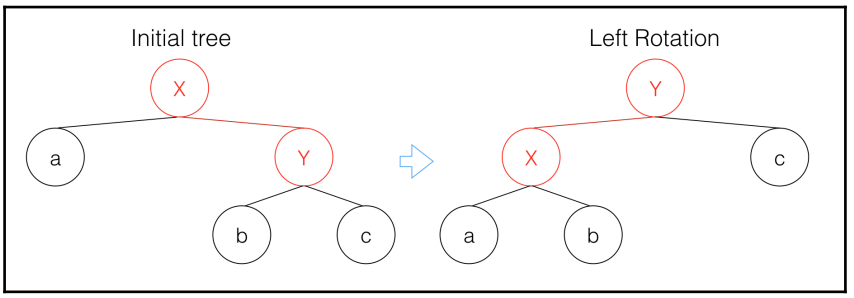
```
        //1.Let's store some temporary references to use them later
        let grandParent = parent.parent
        let newRightChildsLeftChild = self.rightChild
        var wasLeftChild = false
        if parent === grandParent?.leftChild {
            wasLeftChild = true
        }

        //2. My new right child is my old parent
        self.rightChild = parent
        self.rightChild?.parent = self

        //3. My new parent is my old grandparent
        self.parent = grandParent
        if wasLeftChild {
            grandParent?.leftChild = self
        }else {
            grandParent?.rightChild = self
        }

        //4. The left child of my new right child is my old right child
        self.rightChild?.leftChild = newRightChildsLeftChild
        self.rightChild?.leftChild?.parent = self.rightChild
    }
```

Following the code comments, we can see how we proceed with the steps described in the rotation to the right process:

1. Initial setup, where we store some references to be used later.
2. Node **Y**, the old parent of node **X**, is now the right child of node **X**.
3. We assign the old parent of Node **Y** as the new parent of node **X** (if it exists).
4. We assign the old right child of node **X** as the new left child of node **Y**.

Left rotation

Rotate to the left is the opposite version of rotate to the right. We use a rotation to the left in the following scenario:



Left rotation in red-black trees

Here are the steps to achieve left rotation:

1. Node **X** goes down to become the left child of the new tree after the rotation (right side of the figure). Node **Y**, which was the right child of **X**, is now its parent (node **Y** value is greater, so node **X** should be in the left subtree of node **Y**).
2. If node **X** had a parent, we now assign that parent to node **Y**.
3. The left child of node **Y** is now the right child of its child node, **X**.

Now, let's see how to implement this in Swift. Add a new method called `rotateLeft()` to the `RedBlackTreeNode` class**:**

```
//Left
public func rotateLeft() {
    guard let parent = parent else {
        return
    }

    //1.Let's store some temporary references to use them later
    let grandParent = parent.parent
    let newLeftChildsRightChild = self.leftChild
    var wasLeftChild = false
    if parent === grandParent?.leftChild {
        wasLeftChild = true
    }

    //2. My new left child is my old parent
    self.leftChild = parent
    self.leftChild?.parent = self
```

```
        //3. My new parent is my old grandparent
        self.parent = grandParent
        if wasLeftChild {
            grandParent?.leftChild = self
        }else {
            grandParent?.rightChild = self
        }

        //4. The right child of my new left child is my old left child
        self.leftChild?.rightChild = newLeftChildsRightChild
        self.leftChild?.rightChild?.parent = self.leftChild
    }
```

Following the code comments, we can see how to proceed to accomplish this with the steps described in the rotation to the left process:

1. The initial setup, where we store some references, is to be used later.
2. The old parent of node **Y** is now the left child of node **Y**.
3. We assign the old parent of Node **X** as the new parent of node **Y** (if it exists).
4. We assign the old left child of node **Y** as the new right child of node **X**.

Insertion

We have built a base class with some helper methods that represents a red-black tree node/tree. We also have an enumeration to handle the colors. We implemented two rotation methods: left and right. You are ready to learn about the insertion process.

The insertion process in the case of red-black trees is tricky, because we always need to maintain the five color conditions. The insertion process has different scenarios where it can impact those rules, so we have to make the process in a way that ensure the rules at all costs.

In order to simplify things, we are going to do the insertion in a two-step process:

1. Insert the node as we did in binary search trees, by setting the color red by default.
2. As it is possible that the first step destroyed one or more of the color rules, we will review the tree color structure and make modifications to fix those broken rules.

Let's add the following methods to the `RedBlackTreeNode` class**:**

```
    // MARK: Insertion
```

```
//Insert operation methods
public func insertNodeFromRoot(value:T) {
    // To maintain the binary search tree property, we must ensure that
    // we run the insertNode process from the root node
    if let _ = self.parent {
            // If parent exists, it is not the root node of the tree
            print("You can only add new nodes from the root
            node of the tree");
            return
    }
    self.addNode(value: value)
}

private func addNode(value:T) {
    if value < self.value {

        // Value is less than root value: We should insert it in the
        // left subtree.
        // Insert it into the left subtree if it exists, if not,
        // create a new node and put it as the left child.
        if let leftChild = leftChild {
            leftChild.addNode(value: value)
        } else {
            let newNode = RedBlackTreeNode(value: value)
            newNode.parent = self
            newNode.color = RedBlackTreeColor.red
            leftChild = newNode
            //Review tree color structure
            insertionReviewStep1 (node: newNode)
         }
    } else {
        // Value is greater than root value: We should insert it in the
        // right subtree
        // Insert it into the right subtree if it exists, if not,
        // create a new node and put it as the right child.
        if let rightChild = rightChild {
            rightChild.addNode(value: value)
        } else {
            let newNode = RedBlackTreeNode(value: value)
            newNode.parent = self
            newNode.color = RedBlackTreeColor.red
            rightChild = newNode

            //Review tree color structure
            insertionReviewStep1(node: newNode)
        }
    }
}
```

The tree color structure review should handle five different scenarios; we start with the first method, called `insertionReviewStep1`:

1. The node we are adding is the first one of the tree, so it becomes the root. We already know that the root should be black. This method ensures it. Therefore, add it:

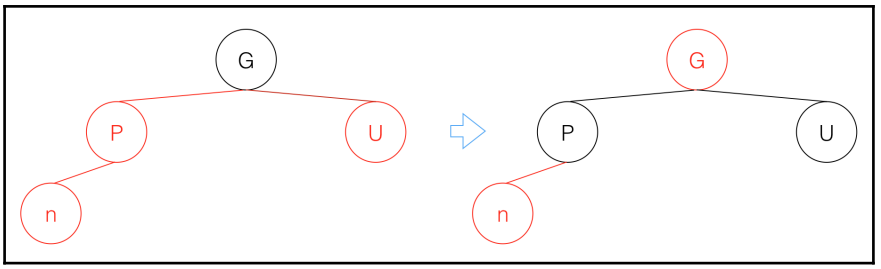
```
// 1. Root must be black
private func insertionReviewStep1(node:RedBlackTreeNode) {
    if let _ = node.parent {
        insertionReviewStep2(node: node)
    } else {
        node.color = .black
      }
}
```

2. In the second step, we are going to check if the parent node is red or black. If it is black, we have a valid tree (remember that we start with a valid tree, and we add a red leaf; if the parent is black, the tree is still valid). Add the following method:

```
// 2. Parent is black?
private func insertionReviewStep2(node:RedBlackTreeNode) {
    if node.parent?.color == .black {
        return
    }
    insertionReviewStep3(node: node)
}
```

3. In the third step, we are going to check if the parent and the uncle of the node are red. If so, we can switch their color to black, and the grandparent's color to red. This will make this part of the tree valid. However, setting the grandparent to red can break second color condition (the root is always black). To fix this, we can put the grandparent as it is, the node is to be added now, and call the same process again since step 1.

Assume in the following diagram that **n** is the new node, **P** is the parent, **U** is the uncle, and **G** is the grandparent:



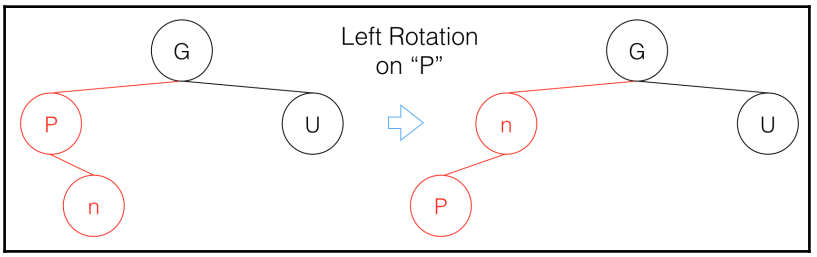
Red-black tree insertion step 3

Add this method:

```
// 3. Parent and uncle are red?
private func insertionReviewStep3(node:RedBlackTreeNode) {
    if let uncle = node.uncleNode() {
        if uncle.color == .red {
            node.parent?.color = .black
            uncle.color = .black
            if let grandParent = node.grandParentNode() {
                grandParent.color = .red
                insertionReviewStep1(node: grandParent)
            }
            return
        }
    }
    insertionReviewStep4(node: node)
}
```

4. When we arrive at step four, there is a possibility that the parent is red but the uncle is black. Also, we are going to assume that the node **n** is the right child of **P**, which is the left child of **G** (or vice versa).

Let's see what happens if we perform a left rotation on **P**:

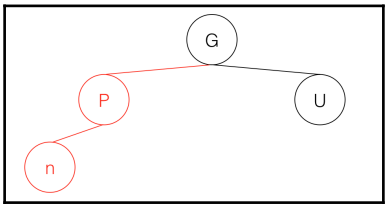


Red-black tree insertion step 4

After the rotation, **n** and **P** switched their roles. Now, **n** is the parent of **P**. So, when proceeding to the next step, we are going to exchange the labels of **n** and **P** in the next figure. Add this method:

```
// 4. Parent is red, uncle is black. Node is left child of a
// right child or right child of a left child
private func insertionReviewStep4(node:RedBlackTreeNode) {
    var node = node
    guard let grandParent = node.grandParentNode() else {
        return
    }
    if node === node.parent?.rightChild &&
    node.parent === grandParent.leftChild {
        node.parent?.rotateLeft()
        node = node.leftChild!
    } else if node === node.parent?.leftChild &&
      node.parent === grandParent.rightChild {
        node.parent?.rotateRight()
        node = node.rightChild!
      }
    insertionReviewStep5(node: node)
}
```

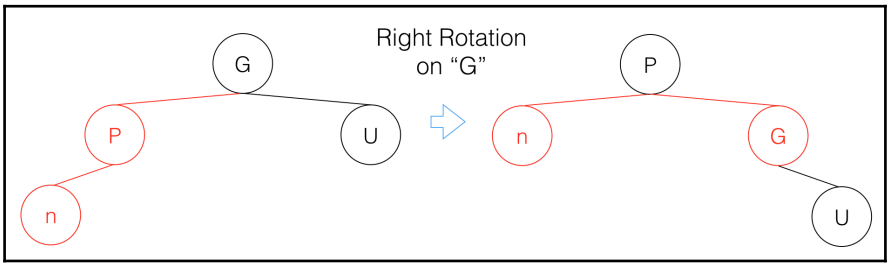
5. In the final step, we have something like this:



Red-black tree insertion before step 5

Where both node **n** and parent **P** are red and are left children of their respective parents (or both right children, in which we are going to do the opposite operations).

Look what happens if we perform a right rotation on **G**, and switch **P** and **G** colors:



Red-black tree insertion step 5

As you can see, we have now a valid red-black tree! Add the following method:

```
// 5. Parent is red, uncle is black. Node is left child of a
// left child or it is right child of a right child
    private func insertionReviewStep5(node:RedBlackTreeNode) {
        guard let grandParent = node.grandParentNode() else {
            return
        }
        node.parent?.color = .black
        grandParent.color = .red
        if node === node.parent?.leftChild {
            grandParent.rotateRight()
        } else {
            grandParent.rotateLeft()
        }
    }
```

As you can see, we have now a valid red-black tree. With this five-step process, we are ready to insert nodes and maintain the properties of a red-black tree. You can test it creating a new root and adding nodes. Add this code to the playground file in order to test the creation of a valid red-black tree:

```
let rootNode = RedBlackTreeNode.init(value: 10)
rootNode.insertNodeFromRoot(value: 12)
rootNode.insertNodeFromRoot(value: 5)
rootNode.insertNodeFromRoot(value: 3)
rootNode.insertNodeFromRoot(value: 8)
```

```
rootNode.insertNodeFromRoot(value: 30)
rootNode.insertNodeFromRoot(value: 11)
rootNode.insertNodeFromRoot(value: 32)
rootNode.insertNodeFromRoot(value: 4)
rootNode.insertNodeFromRoot(value: 2)

RedBlackTreeNode.printTree(nodes: [rootNode])
```

We learned the basics of red-black trees. We implemented the basic tree rotations, in order to achieve later the insertion process. Now, let's jump into AVL trees, which was the first self-balanced tree.

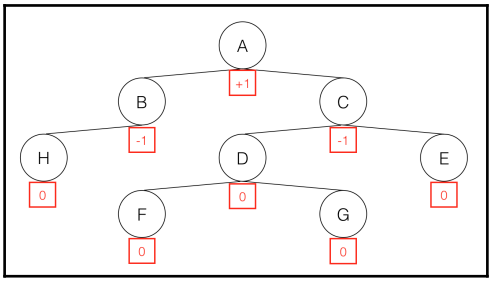
AVL trees

Invented by Georgy Adelson-Velski and Evgenii Landis, and named with their initials, AVL trees were the first self-balance binary search tree created.

AVL tree's special characteristic is if the height of a node subtree is *N*, the height of the other subtree of the same node must be in the range [*N-1, N+1*]. This means that heights of both children should differ at most one.

For example, if the height of the right subtree is 3, the height of the left subtree could be 2, 3, or 4. The difference between both heights is called the **Balance factor**:

Balance factor = Height(RightSubtree) – Height(LeftSubtree)



AVL tree example with balance factors of each node

In the preceding figure, the balance factor of a valid AVL tree is in the range [*-1, 1*] for every node. Leaves have a balance factor of 0.

- If Balance factor is < 0, the node is called **left heavy**
- If Balance factor is = 0, the node is called **balanced**

- If Balance factor is > 0, the node is called **right heavy**

If a child subtree doesn't satisfy this condition, a **rebalance** operation is executed.

Due to its rigid balanced structure, AVL trees ensure the following:

- **Search**: Average and worst time complexity = *O(log(n))*
- **Insertion**: Average and worst time complexity = *O(log(n))*
- **Deletion**: Average and worst time complexity = *O(log(n))*
- **Space**: Average and worst case = *O(n)*

n is the number of nodes in the tree.

Read operations such as search are faster here than in red-black trees because AVL trees are more balanced. However, modify operations such as insertion or deletion can be slower because they could need additional rebalance operations to maintain the balance factor in [*-1, 1*].

AVL tree node implementation

Let's create a new class to represent a node in a AVL tree. As explained before, AVL trees introduce the concept of the balance factor, so we are going to need a new property to store it for each node.

In Xcode, go to **File** | **New** | **Playground**, and call it `B05101_6_AVLTree`. In the `Sources` folder, add a new Swift class named `AVLTreeNode` with the following content:

```
public class AVLTreeNode<T:Comparable> {
    //Value and children-parent vars
    public var value:T
    public var leftChild:AVLTreeNode?
    public var rightChild:AVLTreeNode?
    public weak var parent:AVLTreeNode?
    public var balanceFactor:Int = 0

    //Initialization
    public convenience init(value: T) {
        self.init(value: value, left: nil, right: nil, parent:nil)
    }

    public init(value:T, left:AVLTreeNode?, right:AVLTreeNode?,
    parent:AVLTreeNode?) {
        self.value = value
        self.leftChild = left
```

```
            self.rightChild = right
            self.parent = parent
            self.balanceFactor = 0
        }
    }
```

As you can see, the general structure of the node is similar to the binary tree nodes that we already know. However, we added the `balanceFactor:Int` property to store the balance factor. By default, it is going to be equal to zero, because when we add a new AVL node to a tree, it doesn't have any subtrees, so the height is zero initially.

Before getting into basic operations of AVL trees, let's review the tree rotation technique. As we saw earlier in the chapter while explaining red-black trees, tree rotations are used to rebalance the structure or the tree, and we defined it as follows:

> *"Tree rotation is a mechanism that moves nodes of the tree to a different place in order to change the height of some of the nodes (and make it uniform among all the children)."*

AVL tree rotations

Rotations for AVL trees are very similar to red-black tree ones, but we have to take into account one variable more, each node has a `balanceFactor` property, and when we perform a rotation, its value could change.

AVL tree rotations have two steps:

1. The rotation itself.
2. Updating the `balanceFactor` of the nodes involved in the rotation process.

There are different types of rotation:

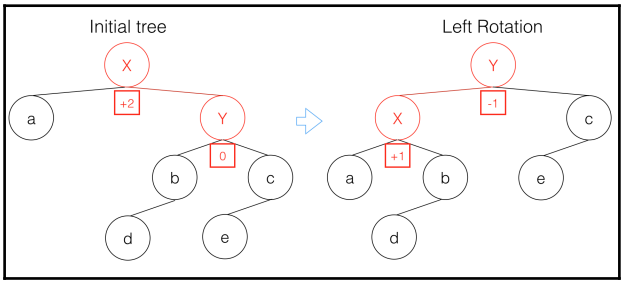
- Simple rotation left
- Simple rotation right
- Double rotation—left-right
- Double rotation—right-left

Let's implement them.

Simple rotation left

We use the simple rotation left when these conditions are met:

- Node **X** is the parent node of **Y**
- Node **Y** is the right child of node **X**
- Node **Y** is not left heavy (so the height is not less than 0)
- Node **X** has balance factor is **+2**



AVL tree simple left rotation before and after

The process is very similar to the one we saw before with red-black trees but with the addition of the balance factor update.

Let's add the `rotateLeft()` method to the `AVLTreeNode` class:

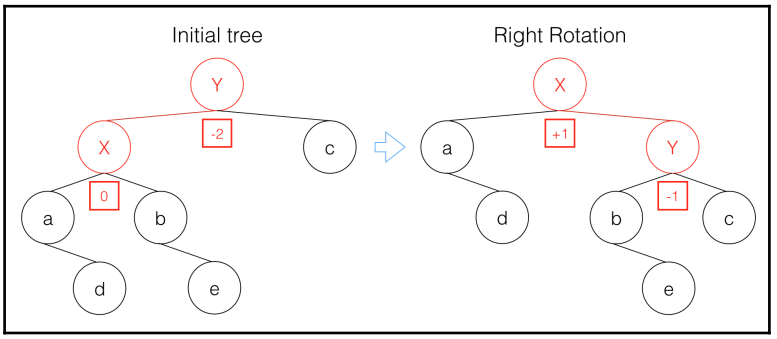
```
//Left
public func rotateLeft()  -> AVLTreeNode {
    guard let parent = parent else {
        return self
    }

    // Step 1: Rotation
    // 0.Let's store some temporary references to use them later
    let grandParent = parent.parent
    let newLeftChildsRightChild = self.leftChild
    var wasLeftChild = false
    if parent === grandParent?.leftChild {
        wasLeftChild = true
    }

    //1. My new left child is my old parent
    self.leftChild = parent
    self.leftChild?.parent = self

    //2. My new parent is my old grandparent
```

```
        self.parent = grandParent
        if wasLeftChild {
            grandParent?.leftChild = self
        } else {
            grandParent?.rightChild = self
          }

        //3. The right child of my new left child is my old left child
        self.leftChild?.rightChild = newLeftChildsRightChild
        self.leftChild?.rightChild?.parent = self.leftChild

        // Step 2: Height update
        if self.balanceFactor == 0 {
            self.balanceFactor = -1
            self.leftChild?.balanceFactor = 1
        } else {
            self.balanceFactor = 0
            self.leftChild?.balanceFactor = 0
        }
        return self
    }
```

You can see in the code comments the two steps involved that we talked about, the rotate step and the balance factors update step.

Simple rotation right

When we face the opposite case, we use the rotation to the right method.



AVL tree right rotation before and after

Let's add the `rotateRight()` method to the `AVLTreeNode` class:

```
    //Right
```

```
public func rotateRight()  -> AVLTreeNode {
    guard let parent = parent else {
        return self
    }

    // Step 1: Rotation
    // 0.Let's store some temporary references to use them later
    let grandParent = parent.parent
    let newRightChildsLeftChild = self.rightChild
    var wasLeftChild = false
    if parent === grandParent?.leftChild {
        wasLeftChild = true
    }

    //1. My new right child is my old parent
    self.rightChild = parent
    self.rightChild?.parent = self

    //2. My new parent is my old grandparent
    self.parent = grandParent
    if wasLeftChild {
        grandParent?.leftChild = self
    }else {
        grandParent?.rightChild = self
    }

    //3. The left child of my new right child is my old right child
    self.rightChild?.leftChild = newRightChildsLeftChild
    self.rightChild?.leftChild?.parent = self.rightChild

    // Step 2: Height update
    if self.balanceFactor == 0 {
        self.balanceFactor = 1
        self.leftChild?.balanceFactor = -1
    } else {
        self.balanceFactor = 0
        self.leftChild?.balanceFactor = 0
    }
    return self
}
```

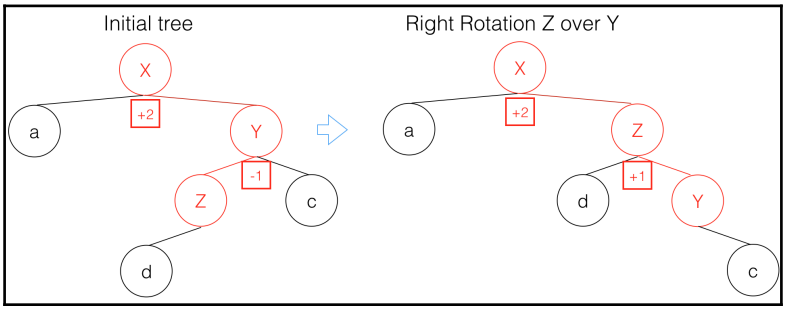
It is the opposite process that we saw with the rotate left scenario.

Double rotation – right-left

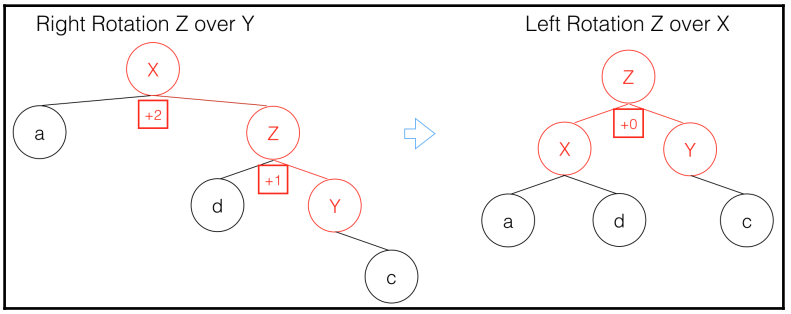
We have previously seen that to perform a simple left rotation, the following conditions are required:

- Node **X** is the parent node of **Y**
- Node **Y** is the right child of node **X**
- Node Y is not left heavy (so the height is not less than 0)
- Node **X** has balance factor is **+2**

But what happens when node **Y** is indeed left heavy? Then we need to perform a double rotation right-left:



AVL tree double rotation step 1 – right rotation of node Z over node Y



AVL tree double rotation step 2 – left rotation of node Z over node X

As you can see in the preceding figures, we need to perform two steps, and at the end, all the balance factors have valid values in the range [*-1, 1*].

Let's implement the method to achieve this. Add a method named `rotateRightLeft()` to the `AVLTreeNode` class:

```
//Right - Left
public func rotateRightLeft() -> AVLTreeNode {
    // 1: Double rotation
    _ = self.rotateRight()
    _ = self.rotateLeft()
    // 2: Update Balance Factors
    if (self.balanceFactor > 0) {
        self.leftChild?.balanceFactor = -1;
        self.rightChild?.balanceFactor = 0;
    }
    else if (self.balanceFactor == 0) {
        self.leftChild?.balanceFactor = 0;
        self.rightChild?.balanceFactor = 0;
    }
    else {
        self.leftChild?.balanceFactor = 0;
        self.rightChild?.balanceFactor = 1;
    }
    self.balanceFactor = 0;
    return self
}
```

As you can see, we perform two steps: the rotations and the balance factor updates.

Double rotation – left-right

This is the opposite case to the previous one; you can check the code in the project.

Let's build an invalid AVL tree and use rotations to fix it. Add this helper method to the `AVLTreeNode` class in order to build an initially incorrect AVL tree:

```
//Insert operation methods
public func insertNodeFromRoot(value:T) {
    // To maintain the binary search tree property, we must ensure
    // that we run the insertNode process from the root node
    if let _ = self.parent {
        // If parent exists, it is not the root node of the tree
        print("You can only add new nodes from the root node of the tree");
        return
    }
    self.addNode(value: value)
}
private func addNode(value:T) {
```

```
        if value < self.value {
            // Value is less than root value: We should insert it in
            // the left subtree.
            // Insert it into the left subtree if it exists, if not,
            // create a new node and put it as the left child.
            if let leftChild = leftChild {
                leftChild.addNode(value: value)
            } else {
                let newNode = AVLTreeNode(value: value)
                newNode.parent = self
                leftChild = newNode
            }
        } else {
            // Value is greater than root value: We should insert it in
            // the right subtree
            // Insert it into the right subtree if it exists, if not,
            // create a new node and put it as the right child.
            if let rightChild = rightChild {
                rightChild.addNode(value: value)
            } else {
                let newNode = AVLTreeNode(value: value)
                newNode.parent = self
                rightChild = newNode
            }
        }
    }

    // Prints each layer of the tree from top to bottom with the node
    // value and the balance factor
    public static func printTree(nodes:[AVLTreeNode]) {
        var children:[AVLTreeNode] = Array()

        for node:AVLTreeNode in nodes {
            print("\(node.value)" + " " + "\(node.balanceFactor)")
            if let leftChild = node.leftChild {
                children.append(leftChild)
            }
            if let rightChild = node.rightChild {
                children.append(rightChild)
            }
        }

        if children.count > 0 {
            printTree(nodes: children)
        }
    }
```

Now in the playground file, copy this code:

```
//: Create a non-balanced AVLTree
var avlRootNode = AVLTreeNode.init(value: 100)
avlRootNode.insertNodeFromRoot(value: 50)
avlRootNode.insertNodeFromRoot(value: 200)
avlRootNode.insertNodeFromRoot(value: 150)
avlRootNode.insertNodeFromRoot(value: 125)
avlRootNode.insertNodeFromRoot(value: 250)

avlRootNode.balanceFactor = 2
avlRootNode.rightChild?.balanceFactor = -1
avlRootNode.rightChild?.rightChild?.balanceFactor = 0
avlRootNode.rightChild?.leftChild?.balanceFactor = -1
avlRootNode.rightChild?.leftChild?.leftChild?.balanceFactor = 0
avlRootNode.leftChild?.balanceFactor = 0

print("Invalid AVL tree")
AVLTreeNode.printTree(nodes: [avlRootNode])

//: Perform rotations to fix it
if let newRoot = avlRootNode.rightChild?.leftChild?.rotateRightLeft() {
    avlRootNode = newRoot
}

//: Print each layer of the tree
print("Valid AVL tree")
AVLTreeNode.printTree(nodes: [avlRootNode])
```

As you can see, our rotation process works well and maintains the tree balanced with balance factors values between [*-1,1*], as expected.

Now that we now how to rebalance the AVL tree with these for techniques (two simple rotations and two double rotations), let's finish the content of AVL trees explaining the search and insertion process.

Search

Search in the AVL trees is the same as in a binary search tree. There is no difference when trying to search for a specific node in terms of methodology.

Insertion

However, the insertion process is more complicated. AVL trees are strictly balanced trees, and inserting a new node can break that balance. Once we put the new node in the correct subtree, we need to ensure that the balance factors of its ancestors are correct. This process is called **retracing**. If there is any balance factor with wrong values (not in the range [*-1 , 1*]), we will use rotations to fix it.

Let's see what the process looks like at a high level:

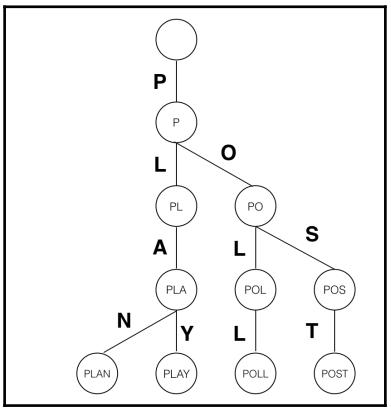
- We insert the new node **Z**, in a valid AVL tree, so all the balance factors are in the range [*-1, 1*].
- We are going to add node **Z** into a subtree of node **X**.
- We go from the bottom of the tree to the subtree, checking the balance factors. If some of them are incorrect, we are going to perform rotations to fix them. Then, we go up again until the balance factors are equal to 0 or until reaching the root.

Trie tree

Until now, we have seen different types of trees, such as binary trees, binary search trees, red-black trees, and AVL trees. In all these types of tree, the content of a node (a value or a key) is not related to the content of a previous node. A single node has a complete meaning, such as a value or number by itself.

But in some scenarios in real life, we need to store a series of data in which those values have common parts; think of it as the suffixes or prefixes in related words, in the alphabet, in a telephone directory.

Here is where a trie tree shines. They are ordered data structures where edges contain part of a key and its descendant nodes have common share part of the previous values. Check this example out:



Trie tree example – storing the words plan, play, poll, post

As you can see in the previous figure, each edge of the tree contains part of a key, and by adding every edge key from the top to a specific node (or leaf), we can build a complete key.

Some implementations use a nil leaf at the end of each subtree to indicate that there are no more nodes below it (so we can build a complete key at that point). Other implementations also use some indicator to warn the same but not only in the leaves, also in intern nodes. There are different variations of trie trees, and we are going to view one of them later, that is, radix trees.

Before that, let's enumerate the main characteristics of trie trees:

- The worst-case time complexity for search is *O(n)*, where *n* is the maximum length of a key. To have a known and limited worst-case is a benefit for a lot of applications.
- The height of a trie tree is also ***n*.**
- Compared to hash tables, trie trees don't need a hash function. They don't use any hash, so there are no key collisions. It is considered to be a replacement for hash tables.
- Values are ordered, so we can perform an alphabetical order, for example, in an easy way.
- The root node is always empty.

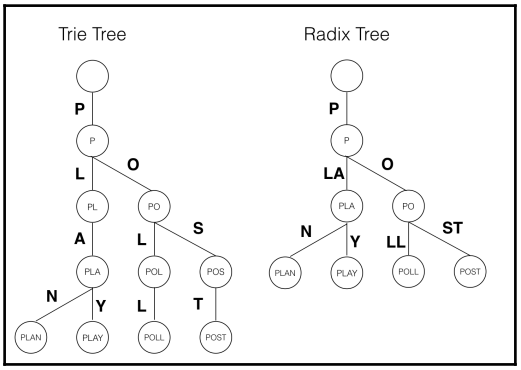
Some classic applications for trie trees are autocomplete functions, predictive text, or word games. These are applications where the values to find/use share a big part of their data (such as words). Using words in lowercase, without punctuation, without special characters, and so on, helps a lot to speed up the time and the memory complexity of trie trees.

Let's see now a different type of the trie data structure—radix trees.

Radix tree

In trie tree, we have seen that each edge contains a single letter or single part of a key. Radix trees are like a compressed version of trie trees, where the edges can contain more than a single letter, even an entire word (if we are using them for words/letters).

This is very effective, reducing the amount of memory and space the tree needs. Let's see an example:



Trie tree (left) and radix tree (right) for the same input

In the preceding figure, you can view the difference between a trie tree and a radix tree for the same input data, **PLAN**, **PLAY**, **POLL**, and **POST**. Note the following:

- The radix version of the trie uses fewer nodes; one of the purposes of the radix trees is to reduce the amount of memory used. This is because each key has more information (each edge), so we need fewer edges.
- We can perform this compression of single letters to partial words in edges when a node has a single child. Note the trie tree edges [**L** ->**A**], [**L** -> **L**], and [**S** -> **T**] in the preceding figure. In those cases, we can compress them into a single node/edge.

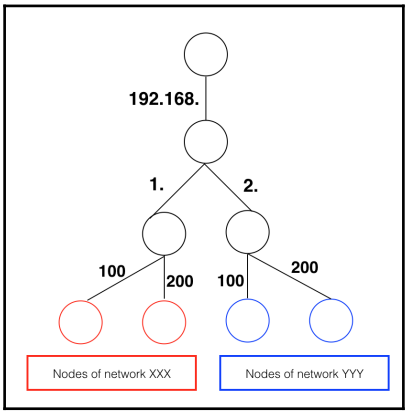
- Radix trees are more difficult to build than regular trie trees.

As an example of application, the IP routing system is a good fit. There are lot of values that share the same part of the key and only differ in the last digits. In this type of situations, a radix tree is very effective. Let's see an example.

Suppose we have a number of IP addresses to store in a data structure. IP addresses are sequences of numbers that represent a unique device in a network (this could be the Internet or a private network). IP addresses consist of four groups of numbers, from *0* to *255*, separated by dots. IP addresses of the same network share a big part of those groups (determined by the mask, but we are not going into detail here).

So for our example, let's imagine that we have two different local networks with two devices each:

- Local **network XXX** prefix (the common part of the network nodes) is `192.168.1.Z`, with the following nodes:
- Device A: `192.168.1.100`
- Device B: `192.168.1.200`
- Local **network YYY** prefix (the common part of the network nodes) is `192.168.2.Z`, with the following nodes:
- Device A: `192.168.2.100`
- Device B: `192.168.2.200`
- If we use a radix tree to determine if a device's IP address is from **network XXX** or **network YYY**, we can build this:



Radix tree application for looking up IP addresses

Note how easy it would be to determine in which part of the tree we have a node of the **network XXX** or **network YYY** following the path in the preceding figure. Taking into account that networks could have thousands of nodes, the radix trees allow us to compress the keys in the edges in order to use as few nodes as possible.

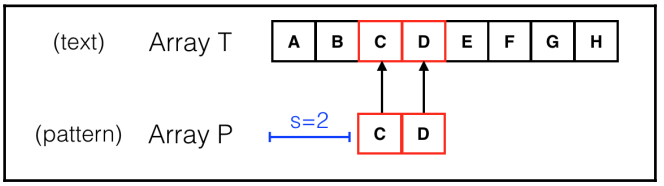
A look at several substring search algorithms

In software programming, it is very common to find a situation where we need to search for the occurrences of a specific pattern of characters in a bigger text. We are going to see some types of search algorithm that will help us with this task.

In order to explain them, first we are going to specify some assumptions:

- The text is defined as an array `T[1..n]`, with length *n*, which contains chars.
- The pattern that we are searching for is defined as an array `P[1..m]`, with length *m* and *m* <= *n*.
- Where the pattern **P** exists in **T**, we call it shift **s** in **T**. In other words, the pattern **P** occurs in the **s+1** position of the **T** array. So, this also implies that [*1* < *s* < *m-n*] and also T[*s+1* .. *s+m*] = P[*1* .. *m*].

Look at the following figure to understand these concepts better:



Text, pattern, and shift example

The objective of every string matching algorithm is to find the different **s** positions in the text, **T**.

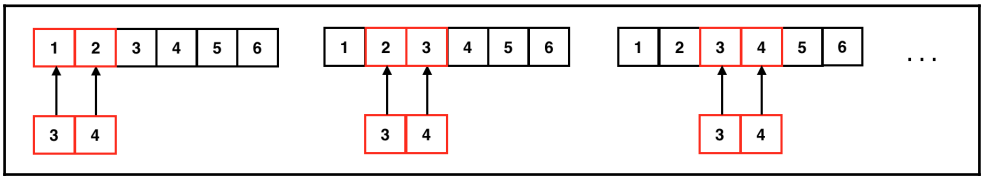
Substring search algorithm examples

Let's take a look at two different algorithms that will help us searching for subtrings in a text.

Naive (brute force) algorithm

The most easy to understand is the brute force algorithm. We are going to loop thought all the positions of the array **T**, comparing the pattern **P** against every position.

The implementation is very easy, but the time complexity in the worst case is *(n-m+1) m*.



In each step, we need m (two in the example) compare operations, one for 3 and one for 4 in every position. We need to check (n-m+1) positions. So the time complexity is (n-m+1) m.

Take a look at the following implementation. In Xcode, go to **File** | **New** | **Playground**, and call it `B05101_6_StringSearch`. In the `Sources` folder, add a new file called `StringSearch.swift`, and the copy this code into it:

```
public class StringSearch {
    // Brutce force using Array of chars
    public static func bruteForce(search pattern:[Character],
    in text:[Character]) {
        // Extract m and n
        let m = pattern.count - 1
        let n = text.count - 1

        // Search for the pattern in the text
        for index in 0...n - m {
            let substringToMatch = text[index...index+m]
            print(substringToMatch)

            if substringToMatch == pattern[0...m] {
                print("Pattern found")
            }
        }
    }
}
```

The method does the naive search with arrays of chars. Check the code to see the same method applied to strings.

You can test it with the following command. Copy this into the playground file:

```
StringSearch.bruteForce(search: ["3","4"], in: ["1","2","3","4","5","6"])
```

If you check the console log, you can see how we are iterating in the array of chars position by position:

```
["1", "2"]
["2", "3"]
["3", "4"]
Pattern found
["4", "5"]
["5", "6"]
```

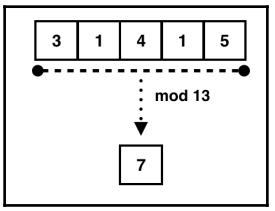
Now, we are going to see a complex algorithm that has a different approach.

The Rabin-Karp algorithm

This algorithm is based on the following numeric theory; the equivalence of two numbers modulo a third number.

The basic steps of the Rabin-Karp algorithm are as follows:

1. Transform the chars and strings into numerical form. For example, if our pattern **P** is the string **31415**, we are going to treat it as the digit **31415**. The same applies to the text **T** in groups of *m* digits (length of the pattern).
2. Calculate the modulo 13 of our pattern. The **mod 13** of **31415** is **7**. Store it in order to compare it later with the modulo 13 of numbers of the **Array T**.

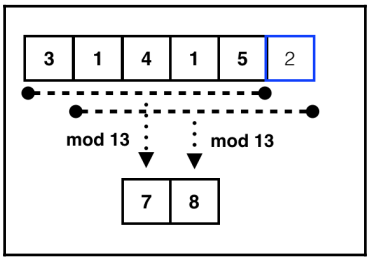


The mod 13 of our pattern = 7. We will search for groups of digits with the same mod 13

3. Now, we are going to loop through **Array T** with a group of numbers of the same length as our pattern, calculating the **mod 13** in every step.

4. If the **mod 13** of any group of digits is equal to **7**, it means that we have a potential result. We need to check it digit by digit because different digits can have the same **mod 13**. So after a match, we make an additional compare digit by digit between the group of numbers and the pattern.
5. If the **mod 13** is not equal, we go back to step four.
6. We are going to advance the pointer one position to the right. As we are dealing with digits, we can do the following. We are going to swift the high-order digit (top left) for the new low order digit in the array (next on the right). We can calculate the **mod 13** for the new group of digits like this:

Imagine that in our example, after the **31415**, the next digit is a **2**:



The **mod 13** of our pattern **31415** is **7**. The **mod 13** of the next group **14152** is **8**. We shift the high order digit for the next digit on the right (the new low order digit)

So, initially, we have our pointer in the digit group **31415**. The **mod 13** of this group is **7**. Now, we move one step to the right, so the digit **3** (high order on the left) is out, and the new low order digit **2** is in. To calculate the new **mod 13**, we can do the following:

*14152 = (31415 – 3*10000) *10 + 2* (mod 13 operations)

*14152 = (7 – 3*3) *10 + 2* (mod 13)

14152 = 8 (mod 13)

Add this method, called `rabinKarpNumbers`, to the `StringSearch` class to implement the Rabin-Karp algorithm. This implementation looks for a pattern of numbers in a text full of numbers:

```
public static func rabinKarpNumbers(search pattern:String, in text:String,
modulo:Int, base:Int) {
    // 1. Initialize
    // Put the pattern and the text into arrays of strings ->
    // So "123" will be ["1","2","3"]
    let patternArray = pattern.characters.map { String($0) }
```

```
let textArray = text.characters.map { String($0) }
let n = textArray.count
let m = patternArray.count
let h = (base ^^ (m-1)) % modulo
var patternModulo = 0
var lastTextModulo = 0

// 2. Calculate pattern modulo and the modulo of the first
// digits of the text (that we will use later to calculate
// the following ones with modulo arithmetic properties)
for i in 0...m-1 {
    guard let nextPatternDigit = Int(patternArray[i]),
    let nextTextDigit = Int(textArray[i]) else {
        print("Error")
        return
    }
    patternModulo = (base * patternModulo + nextPatternDigit) %
    modulo
    lastTextModulo = (base * lastTextModulo + nextTextDigit) %
    modulo
}
// 3. Check for equality and calculate succesive positions modulos
for s in 0...n - m - 1 {
    // Check last calculated modulo with the modulo of the
    // pattern
    if patternModulo == lastTextModulo {

        // We have a modulo equality. Now we check for
        // the same digits equality
        // (different digits could have the same modulo,
        // so we need this double check)
        let substringToMatch =
        textArray[s...s + m - 1].joined(separator: "")
        if pattern == substringToMatch {
            print("Pattern occurs at shift: " + "\(s)")
        } else {
            print("Same modulo but not same pattern: " + "\(s)")
          }
    }

    // Now calculate the modulo of the next group of digits
    if s < n - m {
        guard let highOrderDigit = Int(textArray[s]),
        let lowOrderDigit = Int(textArray[s + m]) else {
            print("Error")
            return
        }
```

```
                // To calculate the next modulo, we have to subtract the
                // modulo of the high order digit and add in a next step
                // the modulo of the new low order digit
                //1. Subtract previous high order digit modulo
                var substractedHighOrderDigit = (base*(lastTextModulo -
                highOrderDigit * h)) % modulo
                if substractedHighOrderDigit < 0 {
                    // If the modulo was negative we turn it positive
                    // (this is because '%' operator in swift is remainder,
                    // not modulo)
                    substractedHighOrderDigit = substractedHighOrderDigit +
                    modulo
                }
                //2. Add the new low order digit modulo
                var next = (substractedHighOrderDigit + lowOrderDigit) %
                modulo;
                if (next < 0) {
                    // If the modulo was negative we turn it positive
                    // (this is because '%' operator in swift is remainder,
                    // not modulo)
                    next = (next + modulo);
                }
                lastTextModulo = next
            }
        }
    }
```

You will need to add a helper infix operator to operate with powers. Above the `StringSearch` class, add the following code:

```
import Foundation

precedencegroup PowerPrecedence { higherThan: MultiplicationPrecedence }
infix operator ^^ : PowerPrecedence
func ^^ (radix: Int, power: Int) -> Int {
    return Int(pow(Double(radix), Double(power)))
}
```

Now, try this new search method with the following code in the playground file (remember that this implementation is for strings of numbers, not letters):

```
let text = "2359023141526739921"
    let pattern = "31415"
    let modulo = 13
    let base = 10
    StringSearch.rabinKarpNumbers(search: pattern, in: text, modulo:
modulo, base: base)
```

As we extract from the code and code comments, this algorithm is more complicated to implement than a brute force algorithm.

This algorithm also introduces a new process (delay) to take into account. As you have seen, before starting searching for the pattern, we need to perform some operations (modulo 13). Some algorithms invest time even before searching for the results. We call this delay the preprocessing time.

Take a look at the following table, where we expose the running times and complexities of some string search algorithms:

Algorithm	**Preprocessing time**	**Matching time**
Brute force	0	$O((n - m + 1)\, m)$
Rabin-Karp	$\Theta(m)$	$O((n - m + 1)\, m)$
Finite automaton	$O(m \lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

More complex algorithms, such as the Finite automaton and Knuth-Morris-Pratt, have great performance. With the basics explained in this chapter, you are now able to dive deeper yourself by learning and implementing different algorithms, such as those ones.

Summary

In this chapter, you've learned about new advanced data structures, such as red-black trees, AVL trees, and trie trees. You also learned how to perform common operations on them, such as single and double rotations. We have seen in which specific cases we can benefit from them.

At the end of the chapter, we reviewed the general characteristics of substring search algorithms by showing the most common and basic concepts. Now you are going to be able to study in depth more complex string search algorithms with these a knowledge of fundamentals.

In the next chapter, you are going to learn about graph algorithms and the data structures used to implement them.

7
Graph Algorithms

Graph algorithms and graph theory were discovered a long time ago, but nowadays they are a common practice in a lot of scenarios. The different algorithms that involve graphs are used in lots of applications in our day to day operations. Think of how social networks recommend new friends to you (which are always related to you in some way, such as a friend of a friend of mine) or how a GPS is able to find the shortest path between an origin and a destination. These kinds of problem can be solved with graph algorithm techniques, and we are going to discuss some of them throughout this chapter.

The topics covered in this chapter are as follows:

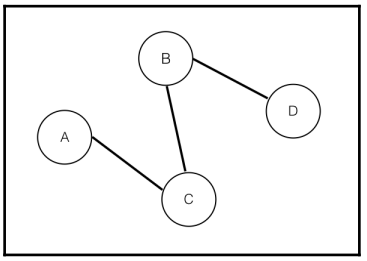
- Learning about graph theory
- Data structures for graphs
- Depth first search
- Breadth first search
- Spanning tree
- Shortest path
- SwiftGraph

Graph theory

Any graph consists of the following:

- A collection of vertices
- A collection of edges

A vertex is a single node that represents an entity (it will depend on the problem to solve). An edge is a connection between two vertices. Consider the following image:



A simple graph example with four vertices and three edges.

How is a graph used in the real world? Imagine that we work on a social network app and we want a way to represent how people are connected in our network. We can achieve this by using a graph, where vertices represent profiles, and edges represent the connections between them.

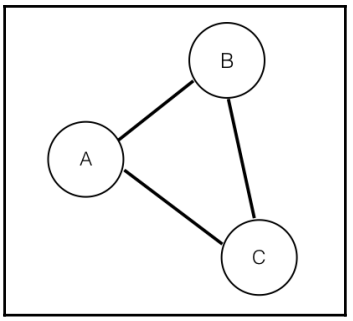
In order to cover as many scenarios as possible, there are different types of graphs, each one serving a different purpose based on their own properties. Let's see some of the most common ones.

Types of graphs

Using a different type of graph for the proper scenario is the best way to achieve a solution to a specific problem with the help of a graph. We are now going to look at the most common graph types.

Undirected graph

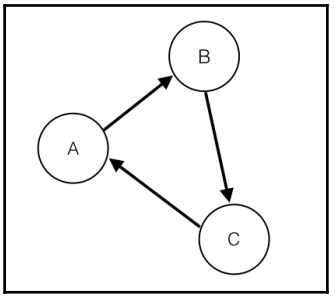
Each edge between vertices is bi-directional. This means that an edge between vertices **A** and **B** represents two paths: from **A** to **B** and from **B** to **A**. A good example of an undirected graph could be the representation of a group of friends, where each node is friends with another, in both directions.



An undirected graph. Each edge is a two-way path between connected vertices.

Directed graph

In this case, edges represent a one-way path between vertices. In order to specify the direction (**A** -> **B** or **A** <-**B**), an arrow is usually used to represent each edge. To use an example of a directed graph, imagine that we want to make a round road trip from point **A** to **B**, then from **B** to **C**, and then coming back to **A**. We can represent this with a directed graph:

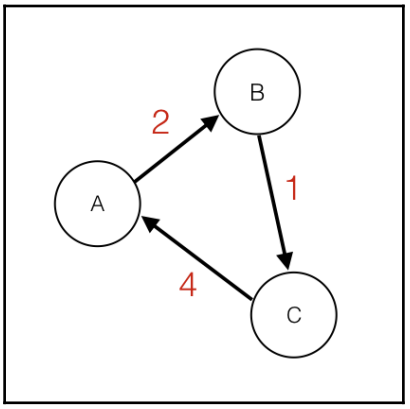


Directed graph

Each edge is a one-way path between connected vertices. Here we can go from **A** to **B**, from **B** to **C**, and from **C** to **A**. No other paths exist.

Weighted graph

This type of graph has extra information in each edge. Usually, this extra info represents a weight or a cost to cross that path between two vertices. Imagine that we want to know how many time units we are going to spend doing a road trip visiting points **A**->**B**->**C**->**A**. We can represent this like in the following figure, and it will be easy to calculate the total amount of time units as the sum of each edge cost as we cross them. That adds up to 7:



A weighted graph

Each edge has information about the cost to use it between origin and destination vertices.

Now that we know what a graph is and which types exist, let's see how to represent them in a different way (not just with vertices and edges as you have already seen).

Graph representations

There are different ways to store a graph in a form other than circles and lines, which are not the best way for computers and compilers. We can represent the vertices and edges of a graph in multiple ways. Let's see some of the most common ones.

Object-oriented approach – structs/classes

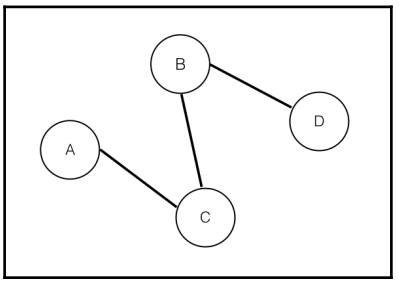
We can apply object-oriented concepts in order to represent the graph with the help of structs and/or classes.

For example, we can define a struct to represent the vertex entity, with some property to store any value (name, weight, or whatever we need), and we can define another struct to represent the edge entity, which will store two pointers or references to the connecting vertices. We can add more properties as needed (such as weights for the edge struct).

This solution takes *O(m+n)* space, where *m* is the number of vertices and *n* is the number of edges. Common operations would take *O(m)* time to finish, because they will require a scan of all the list of struct/classes, taking linear time to do it.

Adjacency list

With adjacency list we have a list of all the vertices, and each one contains a list of its connected vertices. Imagine that we have the following graph:



Graph example with four vertices and three edges

Its adjacency list is as follows:

- A: [C]
- B: [C, D]
- C: [A, B]
- D: [B]

With this solution it is very quick to check if a vertex is connected with another one. We just need to check in its list.

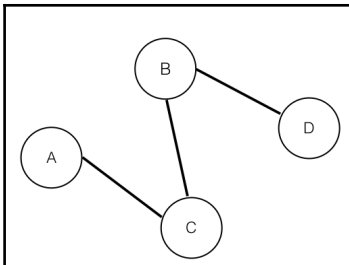
Look how the edges appear twice in the list, one per each vertex. For example, for the edge that connects **A** <-> **C**, we have [**C**] in the list of **A** and we have [**A**..] in the list of **C** too.

We need *O(m+n)* space to store it.

Adjacency matrix

This representation is very useful to check if an edge exists, but it uses more space than the previous ones.

We set up a matrix where rows and columns are the vertices. If there is an edge connecting two vertices, we put a 1 in that column-row pair. If not, we put a 0:



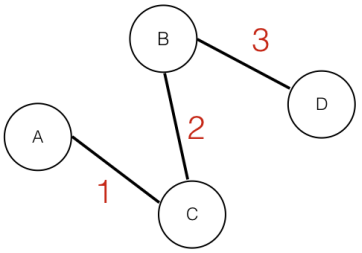
	A	B	C	D
A	0	0	1	0
B	0	0	1	1
C	1	1	0	0
D	0	1	0	0

The graph and the corresponding adjacency matrix

Incidence matrix

In this case, we build a matrix, but with the vertices in the rows and the edges in the columns. For each column, we put a 1 in the two vertices that are connected by that edge. Therefore, each column will have two ones, and the rest will be zeros.

By reading each row, we will see quickly in which edges the vertex is present:



	1	2	3
A	1	0	0
B	0	1	1
C	1	1	0
D	0	0	1

The graph and the corresponding incidence matrix

These are different ways to represent a graph more conveniently, in order to use them in a computer; let's see which data structures can be used to handle this with Swift.

Data structures

We are going to implement the vertex and edge entities using an approach with structs, generics, and protocols. We are going to start building the basic blocks (vertex, edge), and then we will create further data structures on top of them (adjacency list, graph), and so on.

Vertex

We are going to add a new struct to represent the vertex entity. Create a new playground file in Xcode and name it `B05101_7_AdjacencyList`. In the `Sources` folder, add a new Swift file and name it `Vertex.swift`. Add the following code to this file:

```
public struct Vertex<T:Equatable>:Equatable {
    public var data:T
    public let index:Int
}
```

We have defined the `Vertex` struct and we have indicated that it uses some generic type, `T`. We also defined that the generic `T` must be `Equatable`. Conforming to the `Equatable` protocol helps us to make comparisons later.

What do we need to store enough information to represent a vertex? You can see that we defined two properties: some data of generic type `T`, and an index. We will use the index later in the implementation.

Right now you will get some compiler errors. We still need to implement some mandatory methods of the `Equatable` protocol. Add this method below the struct definition:

```
public func == <T: Equatable> (lhs: Vertex<T>, rhs: Vertex<T>) -> Bool {
    guard lhs.data == rhs.data else {
        return false
    }
    return true
}
```

Conforming to the protocol makes it very easy to compare two vertices against each other. Now we can use == to do that. As we have defined in the method, we will compare their data property (which is Equatable too) in order to know if two vertices are equal.

Edge

Now, let's create a new file called `Edge` in the `Sources` folder in order to implement that entity. In the new file `Edge.swift` file, add the following definition for the struct:

```
public struct Edge<T:Equatable>:Equatable {
    public let from:Vertex<T>
    public let to:Vertex<T>
}

public func == <T: Equatable> (lhs: Edge<T>, rhs: Edge<T>) -> Bool {
    guard lhs.from == rhs.from else {
        return false
    }
    guard lhs.to == rhs.to else {
        return false
    }
    return true
}
```

As with `Vertex`, we have made `Edge` generic and `Equatable` (where the generic type is also `Equatable`!).

An `Edge` stores two properties: both are a `Vertex` of generic type `T`. Both represent the vertices that are connected by an edge.

Now that we have a `Vertex` and an `Edge`, let's implement some graph representation to work with both of them. We have seen some of them before in this chapter. Let's do an adjacency list.

Adjacency list

Create a new file in the `Sources` folder called `AdjacencyList.swift`. We are going to create a helper struct called `VertexEdgesList` inside. Add the following code to the `AdjacencyList.swift` file:

```
public struct VertexEdgesList<T:Equatable & Hashable> {
// Each VertexEdgesList contains the vertex itself and its connected
// vertices stored in an array of edges
    public let vertex:Vertex<T>
    public var edges:[Edge<T>] = []
    public init(vertex: Vertex<T>) {
        self.vertex = vertex
    }
```

```
        public mutating func addEdge(edge: Edge<T>) {
            // Check if the edge exists
            if self.edges.count > 0 {
                let equalEdges = self.edges.filter() {
                    existingEdge in
                    return existingEdge == edge
                }
                if equalEdges.count > 0 {
                    return
                }
            }
            self.edges.append(edge)
        }
    }
```

Remember that the adjacency list representation of a graph consists of a series of lists for each vertex, in which each list contains the vertices connected to that vertex.

So here we have implemented a struct to store each vertex with its corresponding list of connected vertices (in the form of edges).

Now, using this struct, let's create the adjacency list graph. The definition of `VertexEdgesList` is as follows:

```
    public struct AdjacencyListGraph<T:Equatable & Hashable> {

        public var adjacencyLists:[VertexEdgesList<T>] = []

        public var vertices:[Vertex<T>] {
            get {
                var vertices = [Vertex<T>]()
                for list in adjacencyLists {
                    vertices.append(list.vertex)
                }
                return vertices
            }
        }

        public var edges:[Edge<T>] {
            get {
                var edges = Set<Edge<T>>()
                for list in adjacencyLists {
                    for edge in list.edges {
                        edges.insert(edge)
                    }
                }
                return Array(edges)
            }
```

```
        }

        public init(){}
    }
```

We have created a new struct that contains an array of adjacency lists. It also has two calculated properties that allow us to get all the vertices and all the edges of the graph.

Right now, it should give you some compiler errors. This is because in the variable that gives us all the edges `public var edges`, we have used a `Set` to store inside unique edges and avoid duplicates. In order to do so, the `Edge` struct should implement the `Hashable` protocol, which gives the struct a way to calculate a unique hash for each edge. We are going to add it in a form of an extension to the `Edge` struct and to the `Vertex` struct, too.

In the `Edge` struct, change the initial line to the following:

```
public struct Edge<T:Equatable & Hashable>:Equatable {
```

And add the following extension:

```
extension Edge: Hashable {
    public var hashValue: Int {
        get {
            let stringHash = "\(from.index)->\(to.index)"
            return stringHash.hashValue
        }
    }
}
```

As you can see, in order to conform to the `Hashable` protocol, we must implement the `hashValue` var. We need to give it a (best effort) unique value. We do so by building a string with the `from` and the `to` indexes and passing it to a `hashValue`.

Now, go to the `Vertex` struct and change the initial line to the following:

```
public struct Vertex< T:Equatable & Hashable>:Equatable {
```

Add the following extension:

```
extension Vertex: Hashable {
    public var hashValue: Int {
        get {
            return "\(index)".hashValue
        }
    }
}
```

Finally, make sure that the initial line of `VertexEdgesList` and `AdjacencyListGraph` also reflect this so the generic `T` must be Hashable. Their initial lines should look like this:

`VertexEdgesList`:

```
private struct VertexEdgesList<T:Equatable & Hashable> {
    AdjacencyList:
    public struct AdjacencyListGraph<T:Equatable & Hashable> {
```

Now you should correctly compile all the files.

So we have created entities for a `Vertex`, an `Edge`, an `AdjacencyList`, and a graph represented by an array of adjacency lists. We have created the base; let's add some functionality to our `AdjacencyListGraph`. Add the following methods inside the `AdjacencyListGraph` struct.

The following is the method to add a new vertex to the graph:

```
public mutating func addVertex(data:T) -> Vertex<T> {
    // Check if the vertex exists
    for list in adjacencyLists {
        if list.vertex.data == data {
            return list.vertex
        }
    }

    // Create it, update the graph and return it
    let vertex:Vertex<T> = Vertex(data: data, index: adjacencyLists.count)
    let adjacencyList = VertexEdgesList(vertex: vertex)
    adjacencyLists.append(adjacencyList)
    return vertex
}
```

We first check if the vertex exists, and if it doesn't, we create it and add it to the graph (inside a new adjacency list).

Let's create a method to add edges:

```
public mutating func addEdge(from:Vertex<T>, to:Vertex<T>) -> Edge<T> {
    let edge = Edge(from: from, to: to)
    let list = adjacencyLists[from.index]

    // Check if the edge already exists
    if list.edges.count > 0 {
        for existingEdge in list.edges {
            if existingEdge == edge {
                return existingEdge
```

```
                }
            }
            adjacencyLists[from.index].edges.append(edge)
        } else {
              adjacencyLists[from.index].edges = [edge]
          }
        return edge
    }
```

As before, we check first if the edge already exists. Then, we create it and update the adjacency lists as needed.

Let's create some vertices and edges and print the result to test our adjacency list graph.

Put this code into the playground file:

```
//Create our Adjacency List Graph
var adjacencyList:AdjacencyListGraph<String> = AdjacencyListGraph<String>()

//Add some vertices
let vertexA = adjacencyList.addVertex(data: "A")
let vertexB = adjacencyList.addVertex(data: "B")
let vertexC = adjacencyList.addVertex(data: "C")
let vertexD = adjacencyList.addVertex(data: "D")

//Add some edges
let edgeAB = adjacencyList.addEdge(from: vertexA, to: vertexB)
let edgeBC = adjacencyList.addEdge(from: vertexB, to: vertexC)
let edgeCD = adjacencyList.addEdge(from: vertexC, to: vertexD)

//Print all
print(adjacencyList)
```

Check the printed statements. You can see the vertices and edges stored properly in the adjacency list form.

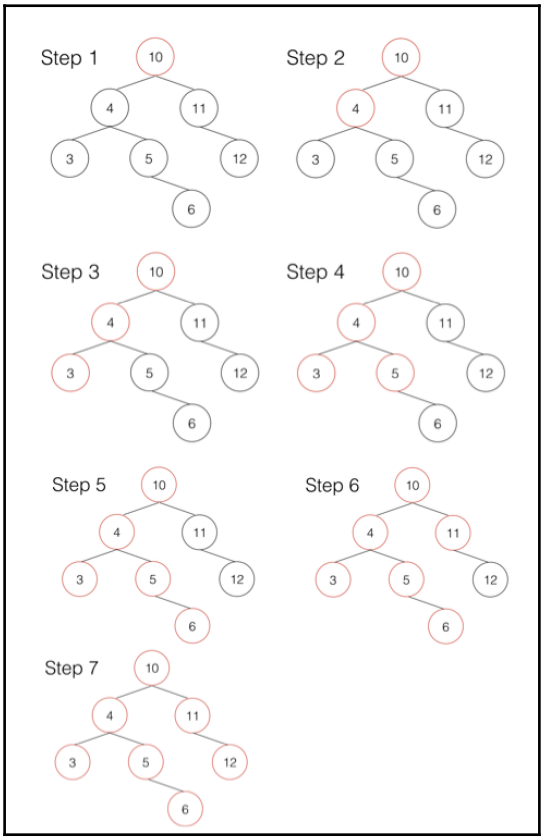
Right now, we have built a graph with vertices and edges. Let's see now how to traverse it in order to visit, search, or order the vertices in different ways.

Depth first search

When we have a graph (such as a tree or a binary tree) with information, it is very useful (and common) in the real world to visit the vertices/nodes of the graph looking for some info.

Depth First Search (**DFS)** is one of the most famous techniques to do this. This type of traversal visits nodes from top to bottom with only one condition: when visiting a node, you must visit the first (left) child of it, then the node itself, then the following child (to the right). Let's see an example with a binary search tree (which is a graph). Remember that a binary search tree is an ordered tree in which each node has at most two children.

Take a look at the following example:



DFS example

Try to apply this recursive pseudocode to the previous figure (in your mind), starting from the root node:

```
public func depthFirstSearch(node:TreeNode) {
    depthFirstSearch(node.leftChild)
    print(node.value)
    depthFirstSearch(node.rightChild)
```

```
}
```

As you can see, we have a binary search tree, and in step 1, we start at the root: node with value **10**. Step 2 visits the first child (from left to right), which is the node with value **4**. Then, as this node has children too, we have to visit them first (again, from left to right): so in step 3, we are visiting the node with value **3**.

Now, notice that there is no left child, so we have to print the node value itself (node with value **3**). Next, as there are no more children to visit (right one), we go backward to the first node unvisited. So in step 4, we have gone up until vertex **4**, print its value, and then we visit its right child, node **5**.

What will happen next? node **5** has no left child, so we print the node value itself and then we can go deeper so we visit the node with value 6. Then, we print the node value and we go up until we have a vertex unvisited. So, let's write down the order of the nodes printed with the DFS:

```
-> 3, 4, 5, 6, 10, 11, 12
```

Have you seen this? We have visited the nodes of the graph in growing order! This is one of the uses of DFS: to get the nodes in order.

Does this sound familiar to you? It should! We saw this in `Chapter 5`, *Seeing the Forest through the Tree*, when talking about in-order traversal and binary search trees. Let's remember what it looks like in Swift (don't create it again, we are just reviewing it).

Binary search tree node:

```
public class BinaryTreeNode<T:Comparable> {
    //Value and children vars
    public var value:T
    public var leftChild:BinaryTreeNode?
    public var rightChild:BinaryTreeNode?
    public var parent:BinaryTreeNode?

    //Initialization
    public convenience init(value: T) {
        self.init(value: value, left: nil, right: nil, parent:nil)
    }

    public init(value:T, left:BinaryTreeNode?,
    right:BinaryTreeNode?,parent:BinaryTreeNode?) {
        self.value = value
        self.leftChild = left
        self.rightChild = right
        self.parent = parent
```

```
        }
    }
```

In the preceding code, we have implemented a node that contains a value, and some references to the left child, the right child, and the parent node. We also included two different inits, because this is a class and Swift requires us to create a init method manually (as opposed to structs).

Imagine that we add this method to the `depthFirstSearch` class recursive method (in-order traversal):

```
public class func depthFirstSearch(node:BinaryTreeNode?) {
    //The recursive calls end when we reach a Nil leaf
    guard let node = node else {
        return
    }

    // Recursively call the method again with the leftChild, then print
    // the value, then with the rigthChild
    BinaryTreeNode.depthFirstSearch (node: node.leftChild)
    print(node.value)
    BinaryTreeNode.depthFirstSearch (node: node.rightChild)
}
```

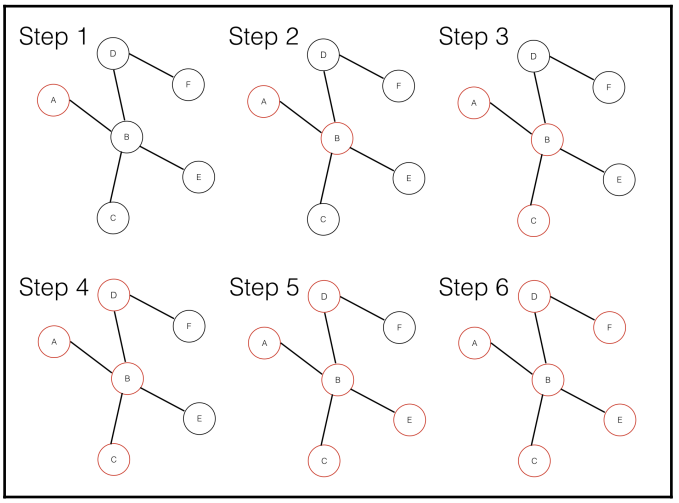
If you remember now, it is the same as the recursive method that we saw before.

But there is another famous technique that changes the visit order, and it is also very common to see: Breadth first search.

Breadth first search

Breadth first search (**BFS**) is a type of traversal that focuses on visiting the nodes of the same level (or neighbors) before going deeper into the graph (to the neighbors of the neighbors).

Another condition to take into account is that we should visit each node just once. Let's see an example with the following graph:



BFS example

In step 1, we visit node **A**. Then we pass to the first child of **A**: node **B**. We visit all the children of **B**: **C**, **D**, and **E**, before going into its grandchildren: F.

We have already implemented `Vertex`, `Edge`, and `AdjacencyList` structs. We are going to make an example of BFS based on a graph built with classes instead, so we will cover both approaches.

Let's start by implementing a graph node with a Swift class. In order to perform a BFS, we need a data structure that contains info about the following:

- Some value to identify the node (such as in the preceding figure: **A**, **B**, **C**, **D**, and so on).
- The list of connected nodes (or children, neighbors…it depends on the context).
- If the node has been visited or not. Remember that in BFS, we should visit each node only once.

So let's code the node with these three conditions in mind. Create a new playground file in Xcode and name it `B05101_7_BFS`. In the `Sources` folder, add a new Swift file and name it `BFSNode.swift`. Add the following code to this file:

```
public class BFSNode<T> {
    //Value, visit status and reference vars
```

```
    public var value:T
    public var neighbours:[BFSNode]
    public var visited:Bool

    //Initialization
    public init(value:T, neighbours:[BFSNode], visited:Bool) {
        self.value = value
        self.neighbours = neighbours
        self.visited = visited
    }

    //Helper method for the example
    public func addNeighbour(node: BFSNode) {
        self.neighbours.append(node)
        node.neighbours.append(self)
    }
}
```

We have created a class to represent the node of a graph for a BFS traversal. The node contains a value (generic type), an array of neighbors connected to itself, and a boolean value to mark the node as visited or not.

We have an `init` method and a method to connect nodes called `addNeighbour`.

Now that we have the base of the node class, we are going to implement a method to perform the BFS. In order to visit each node once and keep track of their neighbors, we are going to need the help of an additional data structure: a queue. A queue is a data structure in which the first element in is the first element out (FIFO).

As we start the BFS process, we are going to visit the first node. Then we are going to put all its neighbors in the queue. We are going to pop them one by one, marking them as visited and adding their own neighbors to the queue (the ones that are not visited). In this way, we are going to visit all the nodes once and in the order that we are looking for.

For clarity of the example, we are not going to implement a full queue in Swift here. We are going to use a plain array for the same purpose (see `Chapter 3`, *Standing on the Shoulders of Giants*, for details of how to implement the queue).

See the following implementation. Add this method inside the `BFSNode` class:

```
public static func breadthFirstSearch(first:BFSNode) {
    //Init the queue
    var queue:[BFSNode] = []
    //Starting with the root
    queue.append(first)
```

```
        //Start visiting nodes in the queue
        while queue.isEmpty == false {
            if let node = queue.first {
                //Print the value of the current node and mark it as visited
                print(node.value)
                node.visited = true
                //Add negihbours not visited to the queue
                for neighbour in node.neighbours {
                    if neighbour.visited == false {
                        queue.append(neighbour)
                    }
                }
                // Remove the already processed node and keep working
                // with the rest of the queue
                queue.removeFirst()
            }
        }
    }
```

As you can follow in the code, we use a queue (array) in order to visit each node in the right order. As we process one node, we put its neighbors in the queue, and we visit them later one by one, before going deeper into the graph.

In order to test it, let's implement the graph of the previous figure. Add the following code to the playground file:

```
let nodeA = BFSNode(value: "A", neighbours: [], visited: false)
let nodeB = BFSNode(value: "B", neighbours: [], visited: false)
let nodeC = BFSNode(value: "C", neighbours: [], visited: false)
let nodeD = BFSNode(value: "D", neighbours: [], visited: false)
let nodeE = BFSNode(value: "E", neighbours: [], visited: false)
let nodeF = BFSNode(value: "F", neighbours: [], visited: false)

nodeA.addNeighbour(node: nodeB)
nodeC.addNeighbour(node: nodeB)
nodeD.addNeighbour(node: nodeB)
nodeE.addNeighbour(node: nodeB)
nodeF.addNeighbour(node: nodeD)
```

Now, let's try our BFS method from `nodeA`:

```
BFSNode.breadthFirstSearch(first: nodeA)
```

Check the console log:

```
A
B
C
```

```
D
E
F
```

You can check that we are visiting the graph in BFS style, visiting each level's child before going deeper. You can try it again starting in any other node (node B is a good example) to see it again.

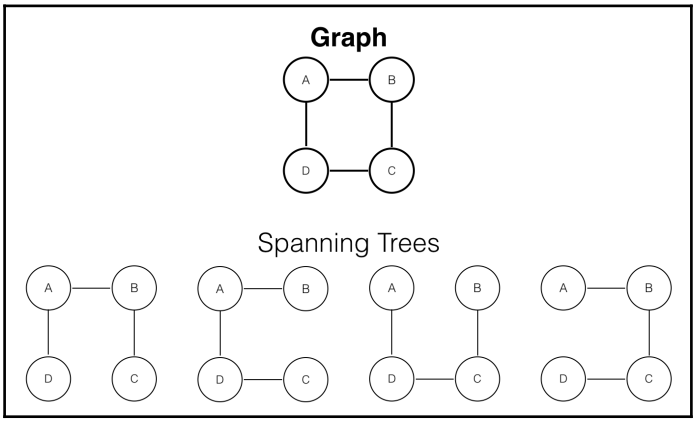
Note that the space and time complexities tend to *O(m)*, where m is the number of nodes in the graph, because we need a queue to store the m nodes, and every node is visited once.

One of the uses of BFS and DFS is the discovery of the spanning tree of a graph. Let's see what a spanning tree is and its uses in graph theory.

Spanning tree

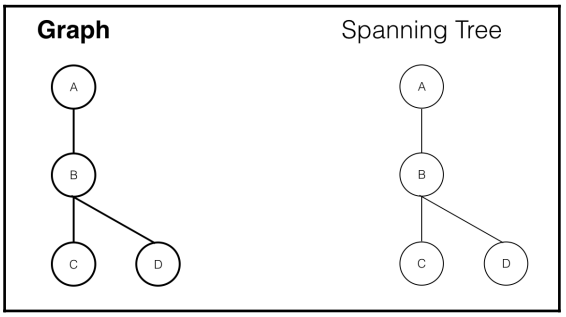
A spanning tree T of a graph G is a subgraph that is a tree and must contain all the vertices of G. In order to fulfill this condition, G must be a connected graph (that is, all vertices have at least one connection to another vertex).

Take a look at the following example of a graph and its spanning trees:



A graph and its spanning trees example

Note that there could be more than one spanning tree for any graph G. If graph G is a tree, then there is only one spanning tree, which is the tree itself:



A tree and its spanning tree example

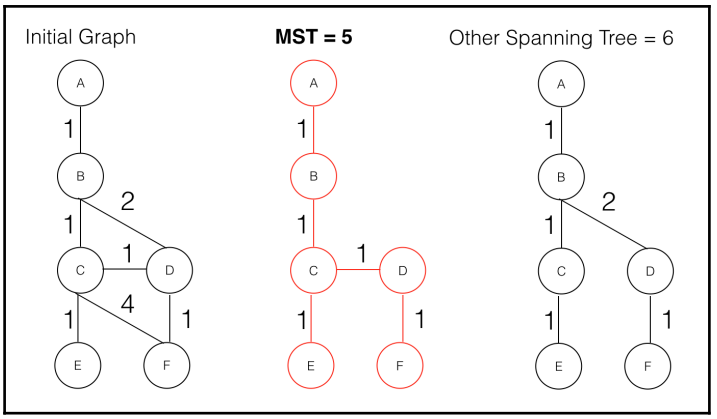
Remember the BFS and DFS algorithms? Well, both of them will give us one of the spanning trees of a graph.

Spanning tree applications include several examples, such as pathfinding algorithms (such as **Dijkstra** and **A***), speech recognition, Internet routing protocol techniques to avoid loops, and so on. Most of them make use at some point of the minimum spanning tree, which we are going to see next.

Minimum spanning tree

In some scenarios where we can use graphs to solve a problem, there is additional information beside vertices and edges. Each edge contains or represents a cost, a weight, or some type of length. In this situation, it is usually useful to know the minimum cost to reach all the vertices.

We can define the **minimum spanning tree** (**MST**) as follows. For a connected undirected graph G where edges have weight, with any number of potential spanning trees, the MST is the spanning tree with less total weight (being the total weight, the sum of all the edges' weights):



Minimum spanning tree example

In the preceding figure, we can see the initial graph on the left side. It is an undirected connected graph. Then, in the middle, we have the MST, which is the spanning tree in which the sum of all the weights (of the edges) is the minimum, in our case, **5**. On the right side, we have an example of another spanning tree of the same graph, which has a total weight of **6**.

How do we use this in the real world? Think about the basic problem of a graph that represents the different paths to reach from node **A** to node **D** (like a GPS system) going through a set of different nodes (**B** and **C**). Each node between them represents a checkpoint, and each edge connecting nodes represents the time to travel between checkpoints. What is the minimum travel time to reach **D** departing from **A**, passing by **B** and **C**? That is a great situation to use an MST. It will give us the path with the lowest cost between those vertices that includes all the vertices of the graph.

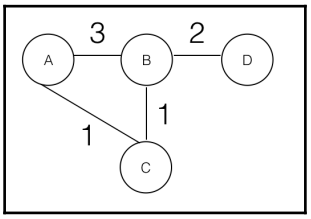
In order to develop a function in Swift to calculate the MST, let's see which algorithms can help us with this task.

Prim algorithm

It was Robert C. Prim in 1957 who gave his surname to **Prim's algorithm**, which takes an undirected connected graph and calculates its MST in linear time.

Other algorithms, such as Kruskal and Borûvka, also calculate the MST of a graph, but in these cases the initial graph is a forest, not an undirected connected graph, which is what we are going to try out now.

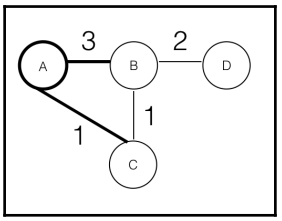
Let's see how Prim's algorithm works with a real example. Here's a graph G with the following vertices, edges, and weights:



Initial undirected connected graph

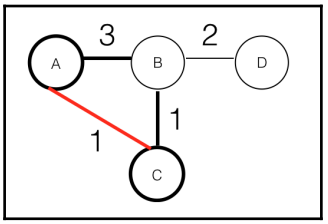
Here are the steps to calculate the MST:

1. Start with an arbitrary vertex of the graph. For clarity, we are going to start with vertex **A**. Initialize a tree with that vertex:



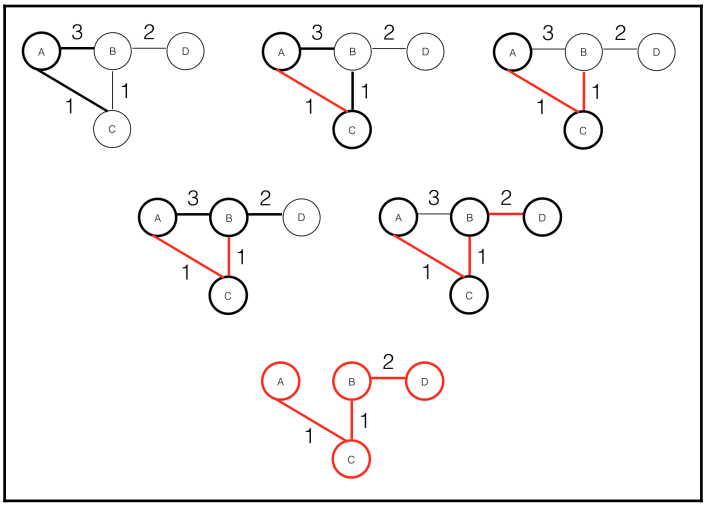
Prim Step 1: Start with an arbitrary vertex

2. For each edge of the selected node, take the edge with the minimum weight, which points to a node that is not visited. In our case, we get the edge from **A** to **C**. Then mark **C** as visited and save that edge as part of the MST:



Prim Step 2: Select the minimum value edge that points to an unvisited vertex

3. Proceed as in step 2 with the recently marked as the visited node, in our case, **C**. Do the same process again until you cover all the vertices of the graph:



Prim Step 3: Select the minimum value edge that points to an unvisited vertex

Create a new playground file in Xcode and name it `B05101_7_MST`. In the `Sources` folder, add a new Swift file and name it `MSTNode.swift`. In this file, add the following code:

```
public class MSTNode<T:Equatable & Hashable> {
        //Value, visit status and reference vars
        public var value:T
        public var edges:[MSTEdge<T>]
        public var visited:Bool
        //Initialization
        public init(value:T, edges:[MSTEdge<T>], visited:Bool) {
            self.value = value
            self.edges = edges
            self.visited = visited
        }
    }
```

As you can see, there is nothing new here, and it is almost exactly the same as the previously created `BFSNode` class. We created a class that contains some generic value (for the ID or label for example), an array of edges, and a boolean value to determine if the node has already been visited or not.

Now add a new file for the edge entity called `MSTEdge.swift` in the `Sources` folder and add the following code:

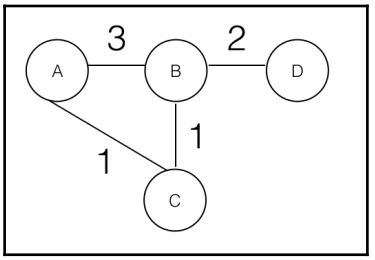
```
public class MSTEdge<T:Equatable & Hashable>:Equatable {
    public var from:MSTNode<T>
    public var to:MSTNode<T>
    public var weight:Double

    //Initialization
    public init(weight:Double, from:MSTNode<T>, to:MSTNode<T>) {
        self.weight = weight
        self.from = from
        self.to = to
        from.edges.append(self)
    }
}

public func == <T: Equatable> (lhs: MSTEdge<T>, rhs: MSTEdge<T>) -> Bool {
    guard lhs.from.value == rhs.from.value else {
        return false
    }
    guard lhs.to.value == rhs.to.value else {
        return false
    }
    return true
}

extension MSTEdge: Hashable {
    public var hashValue: Int {
        get {
            let stringHash = "\(from.value)->\(to.value)"
            return stringHash.hashValue
        }
    }
}
```

We made a class to represent the edges of a graph, with the references to the two connected nodes and a generic value to store the weight of the edge.
Now let's create a class to represent a graph.
Create a new file in the `Sources` folder named `MSTGraph.swift` with the following content:

```
public class MSTGraph<T:Hashable & Equatable> {
    public var nodes:[MSTNode<T>]
    public init(nodes:[MSTNode<T>]) {
        self.nodes = nodes
    }
```

```
}
```

We just created a Graph class to store the vertices and edges and call a method that will print the MST as soon as we implement it. Before doing that, let's build the following graph:



Initial undirected connected graph

Now, in the playground file, add the following code:

```
let nodeA = MSTNode(value: "A", edges: [], visited: false)
let nodeB = MSTNode(value: "B", edges: [], visited: false)
let nodeC = MSTNode(value: "C", edges: [], visited: false)
let nodeD = MSTNode(value: "D", edges: [], visited: false)

let edgeAB = MSTEdge(weight: 3, from: nodeA, to: nodeB)
let edgeBA = MSTEdge(weight: 3, from: nodeB, to: nodeA)
let edgeAC = MSTEdge(weight: 1, from: nodeA, to: nodeC)
let edgeCA = MSTEdge(weight: 1, from: nodeC, to: nodeA)
let edgeBC = MSTEdge(weight: 1, from: nodeB, to: nodeC)
let edgeCB = MSTEdge(weight: 1, from: nodeC, to: nodeB)
let edgeBD = MSTEdge(weight: 2, from: nodeB, to: nodeD)
let edgeDB = MSTEdge(weight: 2, from: nodeD, to: nodeB)

let graph = MSTGraph(nodes: [nodeA,nodeB,nodeC,nodeD])
```

Now, let's create the Prim algorithm implementation. Add the following method inside the `MSTGraph.swift` class:

```
public static func minimumSpanningTree(startNode:MSTNode<T>,
graph:MSTGraph<T>) {
    // We use an array to keep track of the visited nodes to process
    // their edges and select the minimum one (which was not visited
already):
    var visitedNodes:[MSTNode<T>] = []

    // Start by printing the initial node and add it to the visitedNodes
    // array, to process its edges:
    print(startNode.value)
```

```
        visitedNodes.append(startNode)
        startNode.visited = true

        //We loop until we have visited all the nodes of the graph
        while visitedNodes.count < graph.nodes.count {

        // First, we are going to extract all the edges where their
        // "to" node is not visited yet (to avoid loops):
        var unvistedEdges:[MSTEdge<T>] = []
        _ = visitedNodes.map({ (node) -> () in
            let edges = node.edges.filter({ (edge) -> Bool in
                edge.to.visited == false
            })
            unvistedEdges.append(contentsOf: edges)
        })

        // Now, from this array of edges, we are going to select the
        // one with less weight. We print it and add its "to" node to
        // the visitedNode array, to keep processing nodes in the next
        // iteration of the while loop:
        if let minimumUnvisitedEdge = unvistedEdges.sorted(by: { (edgeA,
        edgeB) -> Bool in
            edgeA.weight < edgeB.weight}).first {
                print("\(minimumUnvisitedEdge.from.value) <--------> \
                (minimumUnvisitedEdge.to.value)")
                minimumUnvisitedEdge.to.visited = true
                visitedNodes.append(minimumUnvisitedEdge.to)
            }
        }
    }
```

So as the code comments explain, we are doing the following process:

1. Start with the first node. Print it and add it to an array of nodes to process: `visitedNodes`.
2. From this array of nodes, `visitedNodes`, we want to extract all the edges that point to any unvisited node. We got an array of edges, `unvisitedEdges`.
3. Finally, from this array of edges that point to an unvisited node, `unvisitedEdges`, we want to get the edge with the least weight: `minimumUnvisitedEdge`. We print it and add the destination node of it (`unvisitedEdges.to`) to the visited nodes array to process the next minimum edge available.

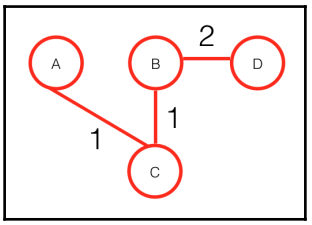
Now that we have the method, let's try it. Add this line at the end of the playground file:

```
MSTGraph.minimumSpanningTree(startNode: nodeA, graph: graph)
```

Now check the console:

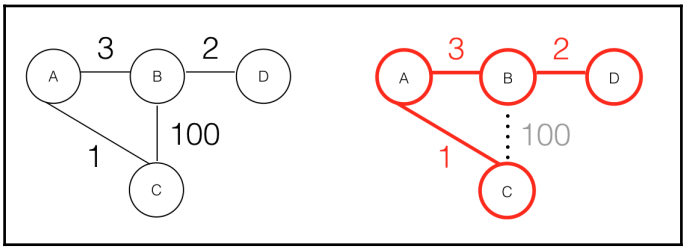
```
A
A <---------> C
C <---------> B
B <---------> D
```

So the MST of this graph is **A**->**C**->**B**->**D**. Let's remember the results of the graph when we explained it before in this chapter:



MST result

Well, it looks great! You can try changing the weights of the edges (take care, you have to change it in both directions) to see how the MST changes too. For example, what will happen if **B**<–>**C** now has a value of 100? Of course, the MST will change as follows:



New MST result

Try it in the playground file. Build the new graph and launch the method. Replace all the previous code with the following:

```
let nodeA = MSTNode(value: "A", edges: [], visited: false)
let nodeB = MSTNode(value: "B", edges: [], visited: false)
let nodeC = MSTNode(value: "C", edges: [], visited: false)
let nodeD = MSTNode(value: "D", edges: [], visited: false)
```

```
let edgeAB = MSTEdge(weight: 3, from: nodeA, to: nodeB)
let edgeBA = MSTEdge(weight: 3, from: nodeB, to: nodeA)
let edgeAC = MSTEdge(weight: 1, from: nodeA, to: nodeC)
let edgeCA = MSTEdge(weight: 1, from: nodeC, to: nodeA)
let edgeBC = MSTEdge(weight: 100, from: nodeB, to: nodeC)
let edgeCB = MSTEdge(weight: 100, from: nodeC, to: nodeB)
let edgeBD = MSTEdge(weight: 2, from: nodeB, to: nodeD)
let edgeDB = MSTEdge(weight: 2, from: nodeD, to: nodeB)

let graph = MSTGraph(nodes: [nodeA,nodeB,nodeC,nodeD])

MSTGraph.minimumSpanningTree(startNode: nodeA, graph: graph)
```

Now check the console:

```
A
A <--------> C
A <--------> B
B <--------> D
```

Well, we got the same new result! Our MST algorithm is working well.

We have developed a method to get the minimum path that includes all the vertices of the graph. What if we want to know which is the shortest path between just two nodes? This is what we are going to see in the next few pages.

Shortest path

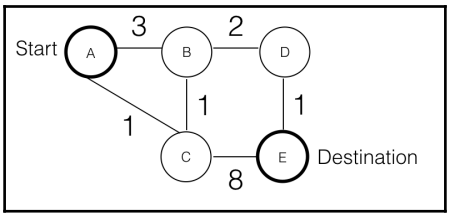
The shortest path in graph theory identifies the path across the nodes of a graph with the lowest cost to go from an origin to a destination. A classic real-life example could is a map routing application where the user wants to go from point **A** to point **B** as quickly as possible, minimizing the total weights during the path.

Depending on the type of graph (undirected, directed, mixed, and so on), different algorithms apply. In our case, we are going to study the Dijkstra algorithm. For Dijkstra, we are going to deal with a directed non-negative weight graph.

Dijkstra algorithm

Edsger W. Dijkstra conceived his algorithm to solve the shortest path for graphs between 1956–1959.

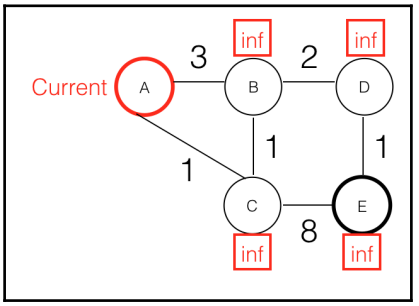
His algorithm finds the shortest path between two nodes, but other variants exist to find the shortest paths between an origin and all other nodes; this is called a shortest path tree. Let's see how it works, and then we will implement it in Swift. We are going to explain it with the following example graph. We want the shortest path between node **A** and node **E**:



Shortest path example

The steps are as follows:

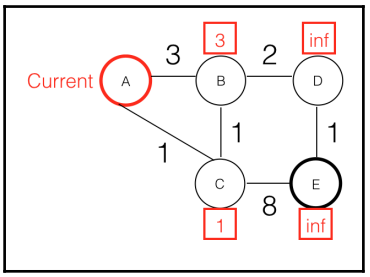
1. The algorithm starts by marking the first node as the current node. It puts all the nodes as unvisited inside a set. It also initializes every node with a temporary distance, infinitum or a maximum number:



Shortest path step 1

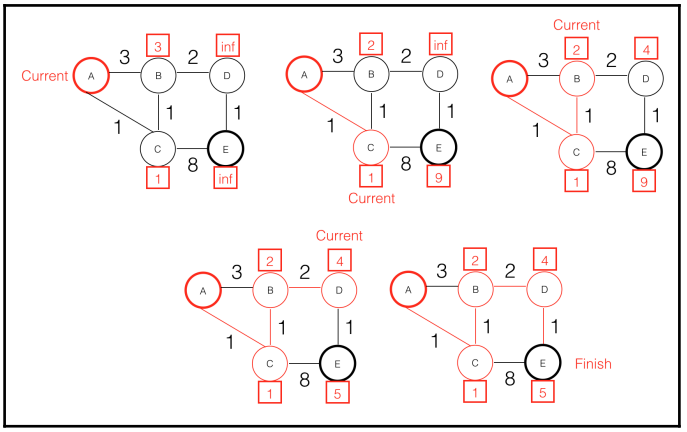
2. Then, for each unvisited neighbor of the current node, calculate the temporary distance from our current node to all its neighbors as the sum of the current node distance and edge weight to the neighbor for each case. If the result is smaller than the existing distance of the node, update it with this new value (we just found a shorter path).

Mark the current node as part of the shortest path:



Shortest path step 2

3. Now you can remove the current node from the set of unvisited nodes.
4. If the destination node has been marked as visited, the algorithm has finished.
5. If not, set the unvisited node with the lowest temporary distance as the current node and go back to step 2:



Shortest path step 5 and the algorithm result

Now let's implement this in Swift. Create a new playground file called `B05101_7_Dijkstra.playground`. In its `Sources` folders, create the following files: `DijkstraNode.swift`, `DijkstraEdge.swift`, and `DijkstraGraph.swift`.

Copy the following code into each file:

`DijkstraNode.swift`:

```
public class DijkstraNode<T:Equatable & Hashable>:Equatable {
```

```
    //Value, visit status and reference vars
    public var value:T
    public var edges:[DijkstraEdge<T>]
    public var visited:Bool

    //Shortest distance to this node from the origin
    public var distance:Int = Int.max

    //Previous node to this in the shortest path
    public var previous:DijkstraNode<T>?

    //Initialization
    public init(value:T, edges:[DijkstraEdge<T>], visited:Bool) {
        self.value = value
        self.edges = edges
        self.visited = visited
    }
}

public func == <T: Equatable> (lhs: DijkstraNode<T>, rhs: DijkstraNode<T>)
-> Bool {
    guard lhs.value == rhs.value else {
        return false
    }
    return true
}

extension DijkstraNode: Hashable {
    public var hashValue: Int {
        get {
            return value.hashValue
        }
    }
}
```

In the Dijkstra algorithm, we are going to use a `Set` to store the unvisited nodes. In order to do it, we need our nodes to be `Hashable` and `Equatable`.

As we will calculate some temporary distances for each node, we have created a new property to store it called distance. As you can see, we initialize the variable to Int.max, which is almost like setting it as infinitum, as Dijkstra asks us.

We also need to store a path to each node, the shortest one. So we have added another property, called previous.

Now,have a look at `DijkstraEdge.swift`:

```
public class DijkstraEdge<T:Equatable & Hashable>:Equatable {
    public var from:DijkstraNode<T>
    public var to:DijkstraNode<T>
    public var weight:Double

    //Initialization
    public init(weight:Double, from:DijkstraNode<T>, to:DijkstraNode<T>) {
        self.weight = weight
        self.from = from
        self.to = to
        from.edges.append(self)
    }
}

public func == <T: Equatable> (lhs: DijkstraEdge<T>, rhs: DijkstraEdge<T>)
->
Bool {
    guard lhs.from.value == rhs.from.value else {
        return false
    }
    guard lhs.to.value == rhs.to.value else {
        return false
    }
    return true
}

extension DijkstraEdge: Hashable {
    public var hashValue: Int {
        get {
            let stringHash = "\(from.value)->\(to.value)"
            return stringHash.hashValue
        }
    }
}
```

This should look familiar to you; it is the same class as MSTEdge.swift. No changes are needed this time, but for clarity of the playground, we created a new class.

Finally, let's copy this code into `DijkstraGraph.swift`:

```
public class DijkstraGraph<T:Hashable & Equatable> {
    public var nodes:[DijkstraNode<T>]

    public init(nodes:[DijkstraNode<T>]) {
        self.nodes = nodes
    }
```

```
public static func dijkstraPath(startNode:DijkstraNode<T>,
graph:DijkstraGraph<T>, finishNode:DijkstraNode<T>) {
    //Create a set to store all the nodes as unvisited
    var unvisitedNodes = Set<DijkstraNode<T>>(graph.nodes)
    //Mark it as visited and put its temporary distance to 0
    startNode.distance = 0

    //Assign the current node
    var currentNode:DijkstraNode<T> = startNode

    //Loop until we visit the finish node
    while (finishNode.visited == false) {
        // For each unvisited neighbour, calculate the
        // distance from the current node
        for edge in currentNode.edges.filter({ (edge) -> Bool in
            return edge.to.visited == false
        }) {
        // Calculate the temporary distance from the current
        // node to this neighbour
        let temporaryDistance = currentNode.distance +
        Int(edge.weight)

        // If it is less than the current distance of the
        // neighbour, we update it
            if edge.to.distance > temporaryDistance {
                edge.to.distance = temporaryDistance
                edge.to.previous = currentNode
            }
        }

        //Mark the node as visited
        currentNode.visited = true

        //Remove the current node from the set
        unvisitedNodes.remove(currentNode)

        if let newCurrent = unvisitedNodes.sorted(by: {
        (nodeA, nodeB) -> Bool in
            nodeA.distance  < nodeB.distance
        }).first {
            currentNode = newCurrent
        } else {
            break
        }
    }
    DijkstraGraph.printShortestPath(node: finishNode)
}
```

```
    public static func printShortestPath(node:DijkstraNode<T>) {
        if let previous = node.previous {
            DijkstraGraph.printShortestPath(node: previous)
        } else {
            print("Shortest path:")
        }
        print("->\(node.value)", terminator:"")
    }
}
```

As you can see in the comments, we have created the Dijkstra algorithm and a helper method to print the final shortest path.

In the `dijsktraPath` method, we have followed the steps described previously in this section:

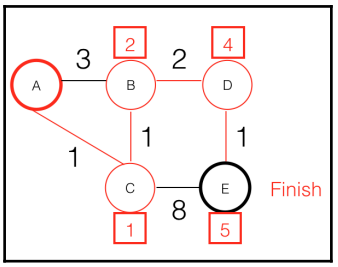
1. Create a set of unvisited nodes (all of them with infinitum as the distance, generated on initialization of the node).
2. Start with the initial node as the current node.
3. For each unvisited neighbor, update the shortest distance to it. If updated, update to the previous node, to keep track of the shortest path to this node. Mark the current node as visited.
4. Get the next unvisited node with the lowest distance and repeat the process from step 3 while we haven't visited the final node.
5. Print the path, starting from the last node.

In order to see this working, copy this code into the playground to execute it:

```
let nodeA = DijkstraNode(value: "A", edges: [], visited: false)
let nodeB = DijkstraNode(value: "B", edges: [], visited: false)
let nodeC = DijkstraNode(value: "C", edges: [], visited: false)
let nodeD = DijkstraNode(value: "D", edges: [], visited: false)
let nodeE = DijkstraNode(value: "E", edges: [], visited: false)

let edgeAB = DijkstraEdge(weight: 3, from: nodeA, to: nodeB)
let edgeBA = DijkstraEdge(weight: 3, from: nodeB, to: nodeA)
let edgeAC = DijkstraEdge(weight: 1, from: nodeA, to: nodeC)
let edgeCA = DijkstraEdge(weight: 1, from: nodeC, to: nodeA)
let edgeBC = DijkstraEdge(weight: 1, from: nodeB, to: nodeC)
let edgeCB = DijkstraEdge(weight: 1, from: nodeC, to: nodeB)
let edgeBD = DijkstraEdge(weight: 2, from: nodeB, to: nodeD)
let edgeDB = DijkstraEdge(weight: 2, from: nodeD, to: nodeB)
let edgeDE = DijkstraEdge(weight: 1, from: nodeD, to: nodeE)
let edgeED = DijkstraEdge(weight: 1, from: nodeE, to: nodeD)
let edgeCE = DijkstraEdge(weight: 8, from: nodeC, to: nodeE)
```

```
let edgeEC = DijkstraEdge(weight: 8, from: nodeE, to: nodeC)

let graph = DijkstraGraph(nodes: [nodeA,nodeB,nodeC,nodeD,nodeE])

DijkstraGraph.dijkstraPath(startNode: nodeA, graph: graph, finishNode:
nodeE)
```

We have created a graph with the following shortest path:



Dijkstra example

Check the playground console; you should get the same result:

```
Shortest path:
->A->C->B->D->E
```

We have learned how to implement graphs in Swift. But there are lots of different ways to do this. There is an open source implementation that you can check in order to see a different approach to what you have done. It is called SwiftGraph, and you can find it in GitHub.

SwiftGraph

SwiftGraph is a pure Swift (no cocoa) implementation of a graph data structure, appropriate for use on all platforms that Swift supports (iOS, macOS, Linux, and so on), created by David Kopec. It includes support for weighted, unweighted, directed, and undirected graphs. It uses generics to abstract away both the type Fof the vertices, and the type of the weights.

It includes copious in-source documentation, some unit tests, as well as utility functions for things such as BFS, DFS, and Dijkstra's algorithm. It has appeared as a dependency in multiple open source projects, but lacks robust testing with large datasets.

Check it out and compare it with your own implementation.

Summary

In this chapter, we've learned about graph theory, vertices, edges, and different searches, such as BFS and DFS. We have also learned how to implement this in Swift using approaches such as structs, protocols, classes, and so on. Finally, we have seen the uses and implementations of the spanning trees of a graph and the shortest path.

8
Performance and Algorithm Efficiency

In this chapter, we are going to see algorithms in terms of performance and efficiency. We have been implementing different algorithms in previous chapters, and now we are going to explain how we can measure them, in theory and in practice. We are going to use the help of asymptotic analysis and Big-O notation to classify and compare algorithms.

The topics covered are as follows:

- Algorithm efficiency
- Measuring efficiency
- Big-O notation
- Orders of common functions
- Evaluating runtime complexity

Algorithm efficiency

When we need to use an algorithm to solve a problem, the first question is, which algorithm is the best to solve this particular problem? Usually, we can have multiple alternatives that will solve our problem, but we need to choose the best one.

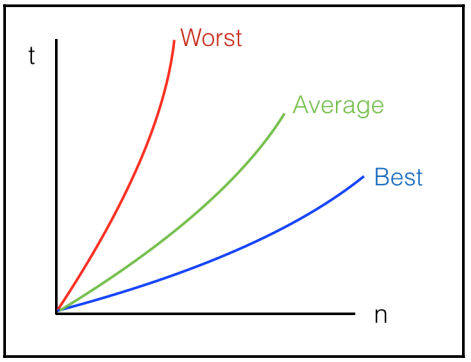
Here is where algorithm efficiency can help us decide which one is a better fit. The efficiency of an algorithm is divided into two main categories:

- **Space analysis**: Algorithms use different data structures and temporal variables to achieve their goal. We have seen how sort algorithms and others use structures such as arrays, stacks, queues, trees, sets, and so on. So when memory space is a constraint, space analysis is a critical criterion to check. This requirement was very important in the past, when memory was a big constraint, but it should not be overlooked now. For example, RAM memory in mobile applications is very important.
- **Time analysis**: Some algorithms are faster than others. Some algorithms are fast when the input data is small, but they start to slow down when the input is larger. Time analysis helps us to determine if an algorithm is fast enough for a specific amount of input data. Now that programs are not solely executed in super computers (think of low-end smart phones, tablets, even smart fridges!), the time needed to run an algorithm is very important. The user experience is a keystone for any software program; users tend to quit an app if it behaves slowly. So it is better to take into account the time requirements of an algorithm before choosing it for your development.

Let's see this with an example. Imagine that Algorithm A uses 100 KB of memory to work and lasts 100 ms to run with certain input data. Algorithm B, with the same amount of data, uses 900 KB of memory and lasts 20 ms. Which one is better? Well, it depends on the situation. Is memory a constraint? Then Algorithm A could take the advantage. Is speed a constraint? Then Algorithm B. But, does 80 ms affect the performance of your program too much? Or is the process executed in a background thread, not affecting the user experience at all? Then maybe it is better to use less memory and sacrifice speed a little, right? As you can see, in the end, which algorithm to use depends on the situation you are facing; it is not a matter of numbers. But we use algorithm efficiency and asymptotic analysis of algorithms to see the big picture and then choose wisely.

Best, worst, and average cases

When calculating the time and space complexity of an algorithm, there could be different inputs that produce different results (in time and in space). Imagine an algorithm that tries to sort a series of numbers. The time that the algorithm needs to sort the numbers is going to differ a lot depending on the input. For example, if we have this sequence, [1,2,3,4,5,7,6] and we want to sort it into [1,2,3,4,5,6,7], it is going to be more easy to do so if we have [7,6,5,4,3,2,1] as input, because the number of permutations that we need is less in the first case. This is an edge case, but it is useful to see that when we talk about time and space complexity, we can have different results. In order to simplify it, we will talk about best, worst, and average cases:



Best, worst, and average cases

Let's put some real scenarios where these cases happen. Imagine that your algorithm searches for an item inside an array:

- **Worst-case**: Item was not found. So your algorithm has checked all the elements.
- **Best-case**: Item was found on the first try.
- **Average-case**: Suppose that the item is potentially in any position of the array with the same probability.

Usually, the worst-case scenario is the one that we need to check the most, because it assures you a limit in the running time/memory requirements. So it is common to compare it, checking the worst-case running times. Also note that we are comparing grow rates, not just specific values.

When we calculate the best/worst/average case, we are measuring the efficiency of the algorithm. We are doing an asymptotic analysis and we can express it with the Big-O notation. Let's see what this is.

Measuring efficiency and the Big-O notation

Any algorithm is going to have its own running time and space complexity. As we have seen, these two variables are not fixed, and usually they depend on the input data. We have also seen that we can have a high level idea with the best, worst, and average complexities. In order to express them in an easy way, we are going to use asymptotic analysis and the Big-O notation.

Asymptotic analysis

Asymptotic analysis gives us the vocabulary and the common base to measure and compare an algorithm's efficiency and properties. It is widely used among developers to describe the running time and complexity of an algorithm.

Asymptotic analysis helps you to have a high-level picture of how an algorithm behaves in terms of memory and speed depending on the amount of data to process. Look at the following example.

Imagine a very simple algorithm that just prints the numbers of an array one by one:

```
let array = [1,2,3,4,5]
for number in array {
    print(number)
}
```

Suppose that the amount of time that the machine spends per instruction is *t*. So, how much time does this algorithm spend to finish? So we have the array initialization plus a loop, with a single instruction inside. If our array has five numbers, we will execute *5*t* instructions in the loop, plus the array initialization, which is *1t*. The algorithm will last *5t+t* (in time measures).

But, what if our input data (array) has 100 numbers? Then the running time will be *100t+t*. And with 10,000 numbers? Then it will be 10,000 t+ t. We can generalize it and say that for an array of size *n*, the running time is *nt+t* or just *n+1* units of time.

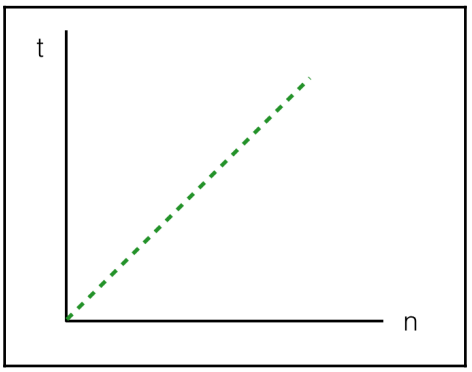
Usually, we have polynomials like this, or more complicated ones. It is common to remove the low order items of the equation and even the multiply factors of the bigger ones. Why? Because when *n* is very large, low order items tend to have less of an effect, and it is more easy to handle. We call this simplification Big-O notation. Look at the following examples:

- If running time is: $3n+5$ -> then Big-O is: $O(n)$
- If running time is: $n^2+10n+33$ -> then Big-O is: $O(n^2)$

- If running time is: $100n^5+n^3+2$ -> then Big-O is: $O(n^5)$

In the case where we are comparing two algorithms of the same Big-O, we should take those factors into account. Imagine that you are comparing Algorithm A with *O(n)* running time and Algorithm B with *O(2n)*. In this case, you should not simplify Algorithm B; it lasts twice the time, and you take this into consideration.

So in our case, where we had running time of *n+1*, the Big-O will be *O(n)*. As you can see, the running time depends on the input data, with a linear relationship:



Running time – linear. *n* is the number of elements in the input array

In asymptotic analysis, we say that this algorithm running time is Big-O of *n*, or abbreviated *O(n)* running time.

What about the space analysis? This algorithm uses just one data structure, an array. The size that we need in memory will be necessary to store all the array elements. If the array has *n* elements, the space needed will also be *n*. So, we again have a linear relationship for the space requirements: *O(n)* space.

Why is this useful? Well, because we can easily see how the algorithm will behave in running time and space requirements because we know that it is *O(n)* for both. We don't need to check the code or run any actual test! That is why the asymptotic analysis is used as a common vocabulary to classify algorithms.

Now we can enter in more detail. Do you remember that we have three different scenarios? Best, worst, and average complexities. Well, we have a different Big-O notation to express them:

- **Big-θ** (**Big-Theta**): We use it to express that the complexity is going to be limited by the worst-case and the best-case, within their bounds

- **Big-O**: We use it to express that the complexity is going to be less or equal to the worst-case
- **Big-Ω** (**Big-Omega**): We use it to express that the complexity is going to be at least more than the best-case

The most common one is the Big-O, because it gives us the upper limit of the complexity, which is good to know if we are breaking any time/space requirement. Now we are going to explain how to determine complexities based on common development blocks and then we will explain with more detail the different orders of Big-O functions that we are going to find.

How to calculate complexities

Depending on the statements and blocks of code used in an algorithm, it will have a different complexity. Let's see the most common lines of code and their effect on the complexity:

- Simple statements such as assignments or variable initializations, call for a function or basic arithmetic operations that will take *O(1)* per statement. So *k* simple statements will take *O(k)*. See the following example of *O(1)* statements:

```
let number = 5
let result = number + 4
var myString:String = "Hey"
```

- `if…else` blocks contain two pieces of code that are different. We are going to execute the code under the `if` or the code under the `else`. Usually we will take the worst complexity of them as the complexity for the whole `if…else` block. So, for example, if the if block is *O(1)* and the else block is *O(n)*, then the complexity of the entire `if…else` is going to be *O(n)*. The following example demonstrates this:

```
let array:[Int] = [1,2,3,4]
if array[0] == 1 {
    //O(1)
    print(array[0])
}else {
    //O(array.lenght)
    for number in array {
        print(number)
    }
}
```

- Loops with simple statements inside repeat themselves *n* times. So if the code inside the loop takes *O(k)* to execute, and the loop executes *n* times, the complexity will be *n*O(k)* or *O(n*k)*:

```
// n = 4
    let intArray:[Int] = [1,2,3,4]
    for number in intArray {
        // O(n) = 4 x O(1) = O(4)
        print(number)
    }
```

- Nested loops, where we have a loop inside another one, will grow exponentially. That is, if we have *O(n)* for a simple loop, if we add another loop inside, the complexity will be $O(n^2)$. And for each loop that we add inside another one, it will grow again. So for three nested loops, the complexity is $O(n^3)$, for four it is $O(n^4)$, and so on:

```
let intsArray:[Int] = [1,2,3,4] //O(n) = 1
var total = 0 //O(n) = 1
for number in intsArray {
    //O(n) = 4^2 = 16
    for nestedNumber in intsArray {
        total = total + number * nestedNumber
    }
}
//O(n) = 16 + 1 + 1
```

Let's see these complexities in more detail with example codes and their graphic representation, so you will have a clear picture of how to calculate the complexities of your algorithms.

Orders of common functions

When we compare the Big-O of two algorithms, we are comparing at the end how the running time and space requirements grow depending on the input data. We need to know how the algorithm will behave with any amount of data. Let's see the orders of common functions in ascending order.

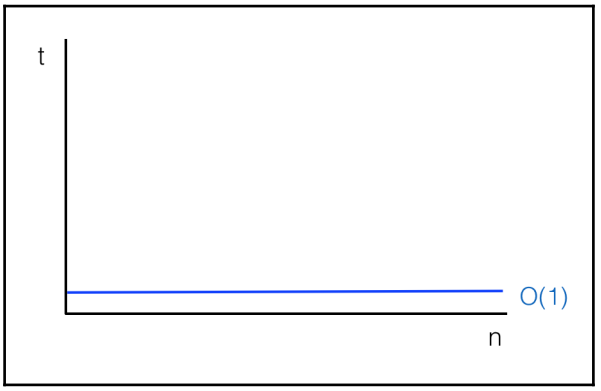
O(1)

When the running time is constant, always with the same value, we have *O(1)*. So the algorithm space/running time is not dependent on the input data. One example is the time needed to access an item in an array with the index. It uses just one instruction (at a high level) to do it. The pop function on a stack is another example of *O(1)* operations. The space complexity of the insertion sort also uses just one memory register, so it is *O(1)*.

Here is an example:

```
public func firstElement(array:[Int]) -> Int?
{
    return array.first
}
```

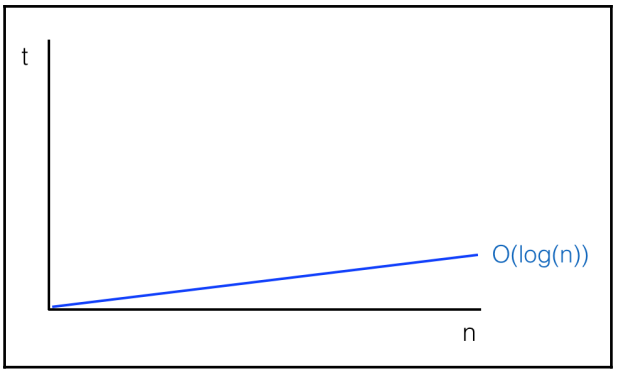
Here we have a very simplified function that receives an array of integers and returns the first one (if it exists). This is an example of an algorithm that takes *O(1)* time to execute. It has only one instruction and it is an access to an array, which also is *O(1):*



Big-O types and performance – O(1)

O(log(n))

Log(n) is between *O(1)* and *O(n)*. We have seen examples of *O(log(n))* when we saw the search and the insertion on red-black trees. It is considered a great complexity, and it is considered the theoretical limit to search in a dataset.



Big-O types and performance – O(log(n))

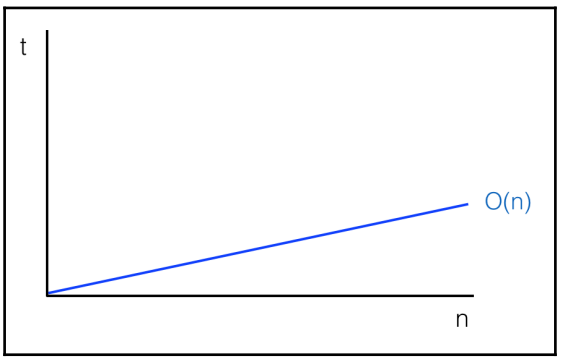
O(n)

This is also known as the linear complexity. *t* grows at the same time as *n*. The search, insertion, and deletion worst-cases of an array are *O(n)*. If it is the worst-case, you need to search the entire array for an item. If you want to insert/delete, usually you have to search first, so that is why we have *O(n)* for all those three scenarios.

Here is an example:

```
for number in array {
    print(number)
}
```

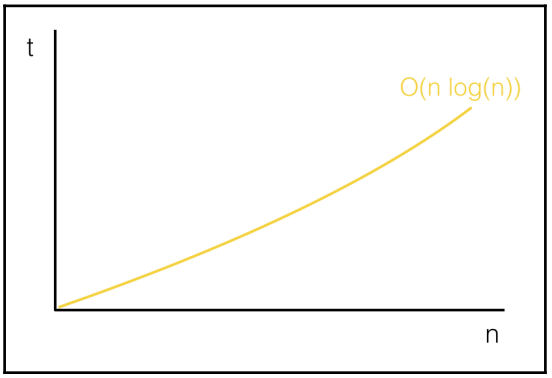
We are going to loop through the *n* elements of the array so we have *O(n)* complexity:



Big-O types and performance – O(n)

O(nlog(n))

This is worse than *O(n)*. An example is the sorting algorithm merge sort (worst-case):

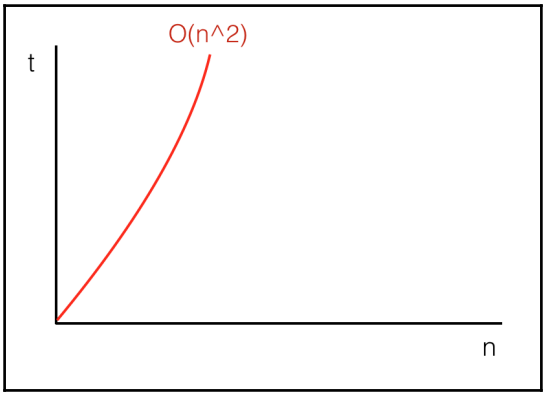


Big-O types and performance – O(nlog(n))

O(n^2)

This is called the quadratic function. It has a bad performance if we compare it with others. But there are still worse cases, such as $O(2^n)$.

We see this complexity when we have two loops nested. For each nested loop, we add one to the exponent, two loops: $O(n^2)$, three loops: $O(n^3)$, four loops: $O(n^4)$, and so on.

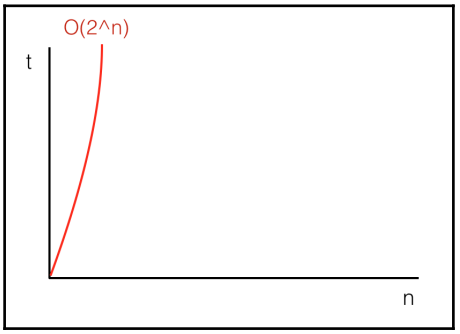


Big-O types and performance – $O(n^2)$

O(2^n)

When an algorithm's work doubles in each step, we have a $O(2^n)$. This is a very bad performance and it should be avoided. Look at the following example:

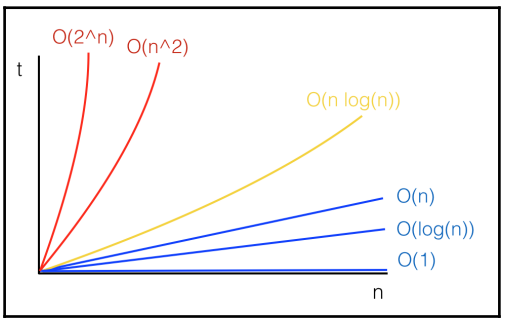
```
public func fibonacci(number:Int) -> Int {
    if number <= 1 {
        return number
    }

    return fibonacci(number: number-2) + fibonacci(number: number-1)
}
```



Big-O types and performance – $O(2^n)$

Graphic comparison

Take a look at the following figure, which has all the order functions together:



Big-O types and performance – red = bad, yellow = regular, blue = good

The $O(2^n)$ and $O(n^2)$ functions are considered to have bad performance in terms of complexity. The rest are considered to be good, so remember this figure, it is very useful when you have to choose between different algorithms. We'll see some real-world examples of this in the following section.

Evaluating runtime complexity

Now that you have the big picture in mind, let's look at some real data for the time performance of common Big-O functions so you will definitely understand the magnitude of the different orders. These running times are done in a faster computer where a simple instruction takes just one nanosecond. Take a look at the following table:

n / f(n)	log(n)	n	n log(n)	n^2	2^n	n!
10	0.003 µs	0.01 µs	0.033 µs	0.1 µs	1 µs	3.63 ms
20	0.004 µs	0.02 µs	0.086 µs	0.4 µs	1ms	77.1 years
30	0.005 µs	0.03 µs	0.147 µs	0.9 µs	1s	8.4×10^{15} years
40	0.005 µs	0.04 µs	0.213 µs	1.6 µs	18.3 min	
50	0.006 µs	0.05 µs	0.282 µs	2.5 µs	13 days.	
100	0.007 µs	0.1 µs	0.644 µs	10 µs	4×10^{13} years	
1,000	0.010 µs	1 µs	9.966 µs	1ms		
10,000	0.013 µs	10 µs	130 µs	100ms		
100,000	0.017 µs	0.10 ms	1.67 ms	10s		
1,000,000	0.020 µs	1ms	19.93 ms	16.7 min		
10,000,000	0.023 µs	0.01 s	0.23 s	1.16 days		
100,000,000	0.027 µs	0.1 s	2.66 s	115.7 days		
1,000,000,000	0.030 µs	1s	29.90 s	31.7 years		

Big-O types and performance – running times for different inputs

The grayed section is colored because the times are not practical in real-world applications. Any algorithm with a running time of more than 1 second is going to have an impact in your application. So by looking at the data provided by this table, some conclusions arise:

- For a very tiny n ($n < 10$), almost any order function works quick
- Algorithms that run on $log(n)$ can have a huge amount of data without becoming slow at all
- Linear and $n*log(n)$ algorithms have a great performance for huge inputs
- Quadratic functions (n^2) start to have bad performance for n>10,000
- Algorithms with $n!$ become slow for any n above 10

So now you have a high level picture, but also some real data to select an algorithm based on its Big-O order properly.

It is very handy to measure the time spent by your own code. Let's see how we can do this in Swift with a simple struct. Create a new playground in Xcode. In the Sources folder of the playground, create a new file and call it `Stopwatch.swift`. Add the following code:

```
import Foundation
public struct Stopwatch {
    public init() { }
    private var startTime: TimeInterval = 0.0;
    private var endTime: TimeInterval = 0.0;

    public mutating func start() {
        startTime = NSDate().timeIntervalSince1970;
    }

    public mutating func stop() -> TimeInterval {
        endTime = NSDate().timeIntervalSince1970;
        return endTime - startTime
    }

}
```

If you follow the code, we are creating a struct with two variables, `startTime` and `endTime`, both of them are `TimeInterval`. `TimeInterval` is a `Double`, so we will gain precision by working with it.

The `Stopwatch` struct has a default init and two methods, `start()`, to initiate the count, and `stop()`, which stops it and returns the elapsed time as `TimeInterval`.

Using it is very easy. In your playground, you are going to create an array of `Int` and then measure a simple loop with your new `Stopwatch` struct. Add the following code inside your playground file:

```
// Initialization
var timer = Stopwatch()

// Measure algorithm
timer.start()
for counter:Int in 1...1000 {
    let a = counter
}

// Print elapsed time
print("Elapsed time \(timer.stop())")
```

Now try changing the amount of elements of the for loop, in order to see how the elapsed time changes. Remember that this simple loop is an *O(n)* function, so it will accept big numbers. However, the playground process is going to add delays, so don't expect the values of the last table where we could input millions of data and still have a quick running time. In `chapter 9`, *Choosing the Perfect Algorithm*, we will make use of `Stopwatch` again to measure the time of different pieces of code.

This exercise will help you to understand in practice how the input data can impact the running time of your algorithm and then the performance of your software. Now you also have an easy method to measure pieces of code in the future.

There are more methods in Swift to measure running times, but we will not go into them in more detail:

- `NSDate`, which you have already used
- `CFAbsoluteTime`
- `ProcessInfo.systemUptime`
- `match_absolute_time`
- `clock()`
- `times()`

Summary

In this chapter, we've learned about algorithm efficiency and how to measure it. We now know about Big-O notation and the syntax used to describe how an algorithm behaves in time and space for any input data size. We have a clear picture of the different Big-O order functions and we can identify the order of different pieces of code. We have also seen how to measure any method in Swift using different functions. In the next chapter, we are going to put into practice what we know about algorithms in order to select the appropriate one for different scenarios.

9
Choosing the Perfect Algorithm

In this chapter, we are going to describe some problems/applications that exist nowadays and how to solve them with algorithms and data structures. We have been preparing ourselves during the last eight chapters, learning the basic and advanced concepts of data structures and algorithms and their corresponding implementations in Swift. Now we are going to describe scenarios that exist in the real world and we are going to solve them by applying the concepts that we have learned throughout this book.

Have you ever attached a URL link in any social network status with a limited count of characters? Have you ever noticed how some Internet applications change long and redundant URLs to tiny, shorter ones? They do it in order to save space and memory, but with an additional benefit for the user, that is, to save you some characters and to allow you to write more content. This is one of the example scenarios that we are going to explain in this chapter, how to build a URL shortening algorithm and which data structures to use.

Before giving more details, let's see the outline that we are going to follow to solve each task.

The scenarios that we are going to solve are as follows:

- Creating a URL shortener
- Creating a secure link checker

For both scenarios, we will perform the following steps to solve them:

1. Explain the problem.
2. Create a high-level first approach of the solution.
3. Write and describe the Swift implementation.
4. Calculate Big-O complexities of our solution to check if the algorithm behaves properly for a real-world situation.

These are the steps that will help us to resolve the problems of this chapter, and also the problems that we will face in our daily work with algorithms.

URL shortener

The first task we are going to address is to build a URL shortener library. But before anything else…what is a URL shortener, and how does it work?

Problems with long URL

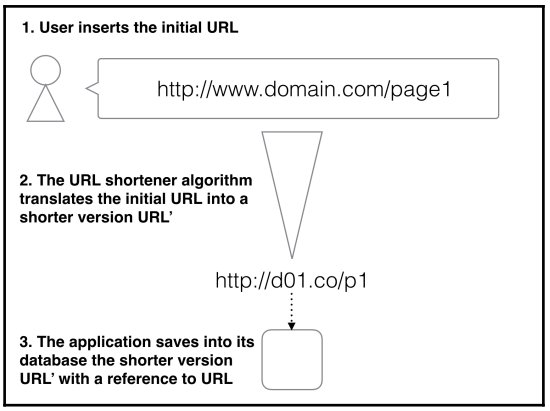
A URL is a unique address that identifies a webpage (or domain) on the Internet (the pretty and readable version of an IP Address). URLs such as `www.google.com` or `www.apple.com` seem to be easy, short, and straightforward to remember and use. However, the Internet today is full of content (more than we can even imagine) and each webpage needs its own unique URL. Moreover, we apply some organization to make URLs more structured and easy to use with the use of slashes, /, the slugs from blog posts, the date included in the URL, and so on. All of this information makes URLs longer, so it is not complicated to find URLs such as `www.domain.com/category/subcategory/year/month/day/long-blog-post-slug-or-title`.

And this is the most common pattern for blog sites, for example, which are one of the most shared content types in social networks (which, by the way, make use of URL shorteners).

This is why Internet services that are being used by people to share information between them, have created algorithms to make these long URLs shorter and store them with fewer storage requirements. And these algorithms are URL shorteners.

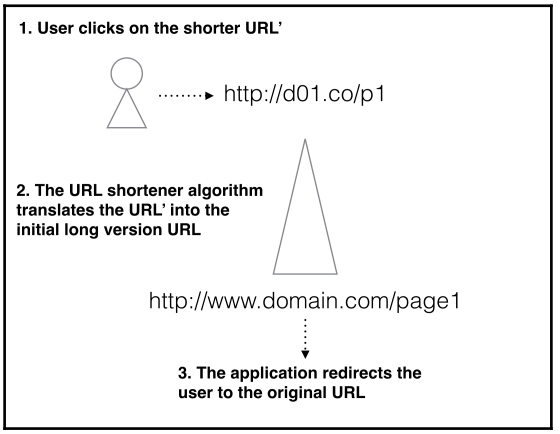
So basically, the process of a URL shortener is like the process in a dictionary to translate from one word into another, and vice versa. The same behavior that we have in a Spanish<->English dictionary, where we can translate *hola* to *hi* and *hi* to *hola*. We have two words that correspond one to each other, that have the same meaning, but with different length. URL shortening systems are the same: they can translate the example URL `http://www.domain.com/page1` into something like `http://d01.co/p1`, and vice versa. So here we have the two scenarios that the system has to handle:

1. Create a short, unique version of the URL (or almost unique version—more on this later):



Creating a shorter version of a URL

2. Translate the short version back into the initial one, by maintaining a reference to the original and indexing/searching for it correctly:



Searching and redirecting the user to the long version of the URL

This is why the URL shorteners need to perform the translation in both directions. They translate it in one direction when the user inserts the initial URL and the URL shortener creates the short version to store it. And later, when a user clicks on the short version of the URL, the URL shortener needs to make the opposite translation with the short version, fetch and redirect the user to the initial long (and proper) one.

But is character count saving the only benefit of URL shorteners? No, there are more:

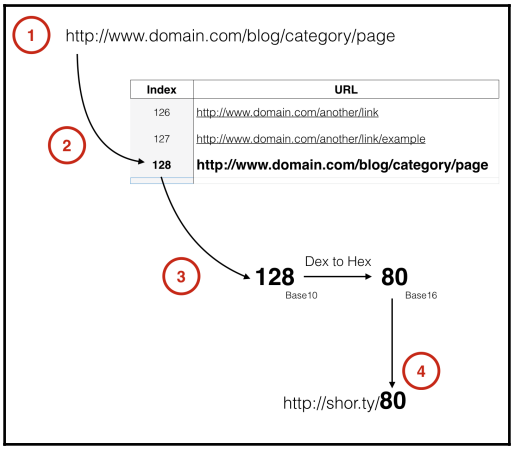
- URL shorteners act as an intermediary when a user clicks on a link, so they have the chance to capture a lot of data from the user web browser, such as geographic location, interests, time of visit, number of visits per link, and so on. This info is very valuable, and services that shorten URLs offer additional services with it.
- As they know which links and content are more often visited, they aggregate information and generate trends and predictive analysis, which is also very valuable information for marketing departments, for example.
- For mobile devices with small screens, showing long URLs is not very user friendly, so the use of short versions improves the user experience.

Now that we know what a URL shortener is, what its purpose is, and the problem(s) it solves, let's see how to implement one.

URL shortener solution approach

We are going to build a URL shortener. The domain of our URL shortener is going to be `http://shor.ty/`. Every short URL is going to be under that domain. So, for example, we could have this link: `http://shor.ty/Ax33`.

At a high level, we can divide our system into four separate pieces or steps, and implement them one by one. Take a look at the following figure, which describes the process:



URL shortner steps

So the four step process of our URL shortener system solution is as follows. Numbers correspond to the previous figure:

1. The system receives as input a long URL string: `http://www.domain.com/blog/category/page`.
2. The system is going to use some data structure to save the URL paired to a reference index. Some real-world services use tables in databases to store the URLs. They generate a table with auto-incremental index, ensuring that each new entry in the table (URL, in our case) will have a unique reference, the table-row ID. In our example figure, the index value 128 has been assigned to our URL. Note how the maximum length of the table index is going to determine the amount of URL that our system can handle. The more digits the index has, the more URLs we can store.
3. Now, in order to make an even shorter version, the system is going to change the base of the index for a greater one. Why? Think in the decimal system (base-10) against the hexadecimal system (base-16). The decimal system can represent 10 different values with just one digit (0-9). However, the hexadecimal system can represent 16 different values with the same amount of digits, 0-9 + A, B, C, D, E, F.

   In our example, we are modifying our index value of 128 (base-10) to an index value of 80 (base-16). Note how we have passed from a three character index to a two character one.

   So in our array, we are going to store a tuple like this (80, longURL).

4. In the last step, we just append the calculated short index to our domain to compound the final URL `http://shor.ty/80`. So finally, we have reduced the URL from `http://www.domain.com/blog/category/page` to `http://shor.ty/80`. That is a lot of characters gone!

When we need to translate it back, we just have to search in our array for a tuple with the value 80 in the first element, and return the second one, which is the longURL the user needs.

URL shortener Swift implementation

Now we are going to implement a struct in Swift that will act as a URL shortener, with a function to shorten a URL, and another function to expand a short URL into the original one. We are going to implement the URL shortener in two different ways, and we will measure and compare both of them to select the best one.

Method 1 – searching for the correct tuple

We are going to start with the implementation of method 1. Create a new playground file in Xcode and name it `B05101_9_URLShortener`. In the `Sources` folder, add a new Swift file and name it `URLShortener.swift`. In this file, add the following code:

```
import Foundation
public struct URLShortener {
    //Our short domain name. We will append the rest of the short
    //URL at the end of this String (the path)
    let domainName:String

    //Array to store tuples of (path, long url) -METHOD1- and array
    //to store just (index) -METHOD2-
    var urlArrayTuples:[(path:String,url:String)] = [] //METHOD1
    public var urlArray:[String] = [] //METHOD2
    //Public init
    public init(domainName:String) {
        self.domainName = domainName
    }
    //MARK: - Shorten and expand methods
    //METHOD 1
    //Function to receive a URL, use the index of the array
    //(transformed to Base16) as ID forming a tuple (Base16ID, URL)
    //and save it into an array. Returns the short URL as
    //"domainName" + Base16ID
    public mutating func shorten(url:String) -> String {
        //Save the position of the new URL, which will be the last
        //one of the array (we append new elements at the end)
        let index = urlArrayTuples.count

        //Swift native method to transform from Base 10 Int to Base
        //16 String
        let pathBase16String = String(index, radix: 16)
        //Create a new tuple (pathbase16ID, URL) and append it
        urlArrayTuples.append((pathBase16String , url))

        //Compound and return the shorten Url like domainName/path
        //-> http://short.ty/ + 1zxf31z
        return domainName + pathBase16String
    }

    //Function that receives a short URL and search in the array
    //for the tuple with that 'path' value to retrieve the 'url'
    //value of the tuple
    public func expand(url:String) -> String {
        let pathBase16String = url.components(separatedBy:
        "/").last!
```

```
            for tuple in urlArrayTuples {
                if (tuple.path == pathBase16String) {
                    //URL found
                    return tuple.url
                }
            }

            //URL not found
            return domainName + "error404.html"
        }
    }
```

We have implemented a new struct to represent our `URLShortener`. It has two properties, the domain name `domainName` and an Array to store URLs `urlArrayTuples`. It also has another array for method 2, which we will see later, `urlArray`. It has a public `init` to assign a specific string for the domain name.

It has `func shorten`, which receives a long URL and outputs a short one. This function applies the steps we have seen before:

1. Receives a URL as input.
2. Gets the last index where we will append our tuple.
3. Transforms the index in base-10 into a base-16 string, this will be the path. Then it stores the URL in an array as a tuple such as (path, longURL).
4. It outputs the new short URL, such as domainName + path.

There is another method called `func expand`**,** which does the opposite, from a short URL, search for the original (and long) one. It gets the path part of the short URL, and searches in the `urlArrayTuples` for the tuple with the same path value. Then, it returns the `url` value of that tuple, which is the long initial URL the user needs.

Finally, in `Chapter 8`, *Performance and Algorithm Efficiency*, we learned how to measure the elapsed time of a piece of code. Let's add the same `Stopwatch` struct and use it here. Add this code in the `URLShortener.swift` file, below our `URLShortener` struct:

```
//MARK - For measuring code
public struct Stopwatch {
    public init() { }
    private var startTime: TimeInterval = 0.0;
    private var endTime: TimeInterval = 0.0;

    public mutating func start() {
        startTime = NSDate().timeIntervalSince1970;
    }
```

```
    public mutating func stop() -> TimeInterval {
        endTime = NSDate().timeIntervalSince1970;
        return endTime - startTime
    }
}
```

Now we have all the pieces needed to shorten a URL and expand it back. Let's try and measure it! In the `B05101_9_URLShortener` playground file, copy this code to create a new `URLShortener` struct and measure tests:

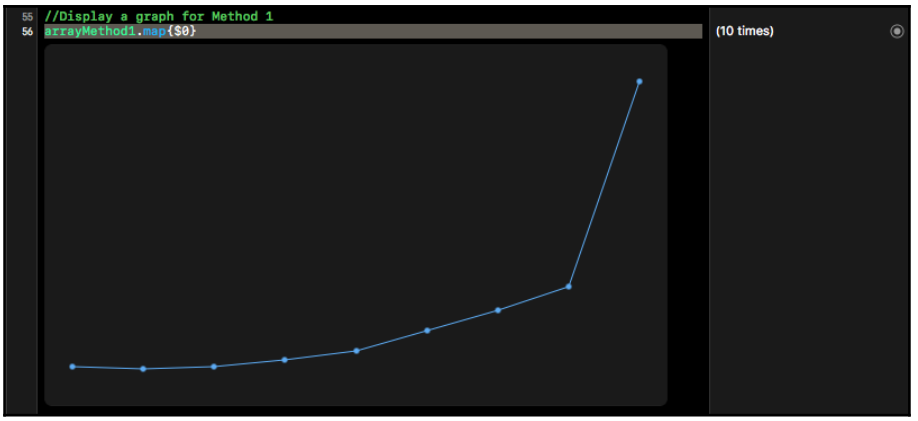
```
import Foundation
//Create URL shortener
var myShortenMachine = URLShortener(domainName:"http://shor.ty/")
var crono = Stopwatch()
//Fill it (both arrays, one for each method) with lots of long URLs
//to shorten
for i in 0...100000 {
    myShortenMachine.shorten(url:
    "http://www.test.com/blog/page/file/" + "\(i)")
    //myShortenMachine.shortenFast(url:
    "http://www.test.com/blog/page/file/" + "\(i)")
}

//Now search for specific URLs and measure the time spent to do it.
//We will store measures in 2 arrays to compare them at the end
var arrayMethod1:[TimeInterval] = []
var arrayMethod2:[TimeInterval] = []

//Method 1: Searching tuples in the array of (path,URL)
crono.start()
print(myShortenMachine.expand(url: "http://shor.ty/0"))
arrayMethod1.append((crono.stop()))
crono.start()
print(myShortenMachine.expand(url: "http://shor.ty/100"))
arrayMethod1.append(crono.stop())
crono.start()
print(myShortenMachine.expand(url: "http://shor.ty/500"))
arrayMethod1.append(crono.stop())
crono.start()
print(myShortenMachine.expand(url: "http://shor.ty/1000"))
arrayMethod1.append(crono.stop())
crono.start()
print(myShortenMachine.expand(url: "http://shor.ty/2000"))
arrayMethod1.append(crono.stop())
crono.start()
print(myShortenMachine.expand(url: "http://shor.ty/3000"))
arrayMethod1.append(crono.stop())
crono.start()
```

```
print(myShortenMachine.expand(url: "http://shor.ty/4000"))
arrayMethod1.append(crono.stop())
crono.start()
print(myShortenMachine.expand(url: "http://shor.ty/7000"))
arrayMethod1.append(crono.stop())
crono.start()
print(myShortenMachine.expand(url: "http://shor.ty/18500"))
arrayMethod1.append(crono.stop())

//Display a graph for Method 1
arrayMethod1.map{$0}
```

We have created a URL shortener struct and have stored 100,000 long URLs there. Then we have taken measure of how much time is needed to translate seven different short URLs into long ones. Some of these URLs are at the start of the array and some of them at the end, to take measures across it. Execute the playground, and check the resulting graph by clicking the circle/plus button icon in the top right of the line:

```
arrayMethod1.map{$0}
```

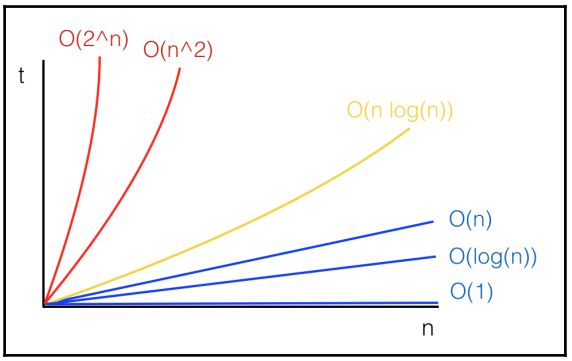
The result is as follows:



URLShortener – method 1 Time complexity

You know what this is, right? We saw it in `Chapter 8`, *Performance and Algorithm Efficiency*. It is the time complexity of our algorithm to shorten and expand URLs!

How does it look like? Remember the Big-O function orders:



O(n) time complexity orders

The good news is that our `URLShortener` seems to work fine, it shortens and expands URLs correctly. The bad news is that looking into the graph, its time complexity looks very similar to something between $O(n^2)$ and $O(nlog(n))$.

Let's see in `func shorten` and `func expand` if we can detect some look and how to improve our algorithms:

- `func shorten`: In this method, we have four single commands. All of them have $O(1)$ time complexity, there is no loop or nested loops, neither recursive calls. So this method is not the bottleneck.
- `func expand`**:** Here, we have something different. After extracting the path part of the URL, we are looping through the `urlArrayTuple` searching for the specific tuple. In the worst-case, which is the last one, we are going to search `urlArrayTuple.count` once. In our example, the array contains 100,000 tuples, so here we have a piece of code that grows up as our input (amount of URLs stored) grows up with $O(mn)$ complexity, where n is the number of tuples and m is the length of the path we are looking for.

Now that we know where the problem is thanks to our Big-O notation skills, let's try to modify our `URLShortener` a bit to reduce this.

Method 2 – accessing the correct array position by index

In method 1, we were storing the short URLs as tuples in the array and later we searched for the correct tuple with the path value. The path was a string (hexadecimal) that we have calculated from the index position (decimal) in the array. In that position, is where we have appended the tuple.

So, what if we do the following. When we receive a long URL, we put it directly in the last position of the array. With that position value, let's call it `decimalIndex`, we calculate the hexadecimal string to obtain a path (like before). But remember, we are not storing a tuple in the array with the hexadecimal value as the key to search for it. We are storing the URL directly in the `decimalIndex` position.

So later, when we receive a short URL and we need to fetch the initial long one, we can follow this process:

1. Extract the path of the URL, which is the hexadecimal string we have calculated when shortening the URL.
2. Translate it to a decimal number.
3. This decimal number is the exact position in the array where we have stored the URL! Fetch it and return it to the user.

Let see the code. Add the following methods to the struct `URLShortener` in the file `URLShortener.swift`:

```
//METHOD 2
    //Function to receive a URL, append the URL at the end of the
    //array, an use that position as the short URL path (transformed
    //into base16 first). Returns the short URL as "domainName" +
    //Base16ID.
    //So in this case, when we receive a short URL an want to get
    //the initial URL, we just translate back the 'path' part of the
    //URL to Base10 number so we have the specific index in the array
    //where we have stored the inital URL, there is no need for
    //searching
    public mutating func shortenFast(url:String) -> String {
        //Save the position of the new URL, which will be the last
        //one of the array (we append new elements at the end)
        let index = urlArray.count

        //Save the URL: Append it at the end
        urlArray.append(url)
```

```
        //Swift native method to transform from Base 10 Int to Base
        //16 String
        let indexBase16String = String(index, radix: 16)

        //Own method: Change the base of the index where we have
        //stored the Url (from Decimal to Hexadecimal) and pass it as
        //String (it could contain letters not just numbers!)
        //let indexBase16String =
        URLShortener.base10toBase16(number:index)

        //Compound and return the shorten Url
        return domainName + indexBase16String
    }

    // Function that receives a short URL, calculate the proper
    // index in base 10 and fetch the url for the array of urls
    // directly by its position in the array
    public func expandFast(url:String) -> String {
        let hexString = url.components(separatedBy: "/").last!

        // Swift native method to transform from Base 16 String to
        // Base 10 Int
        let decimalIndex = Int(hexString, radix: 16)!

        // Own method: To transform from Base 16 String to Base 10
        // Int
        // let decimalIndex = URLShortener.base16toBase10(hexString:
        // hexString)
        if let url = urlArray[safe: decimalIndex] {
            return url
        }
        return domainName + "error404.html"
    }
```

As you can see, now there is no loop in any of the functions. We just have operations with *O(1)* or *O(k)* time complexity. We have modified our expand algorithm so that instead of a loop with *O(nm)* complexity, we have a direct access to an array which is *O(1)*.

We need to add a helper method to retrieve elements of an array in a secure way. At the end of the file, below the Stopwatch struct, add this extension to Collection:

```
/// Returns the element at the specified index iff it is within
/// bounds, otherwise nil.
extension Collection {
    subscript (safe index: Index) -> Iterator.Element? {
        return index >= startIndex && index < endIndex ?
        self[index] : nil
    }
```

```
}
```

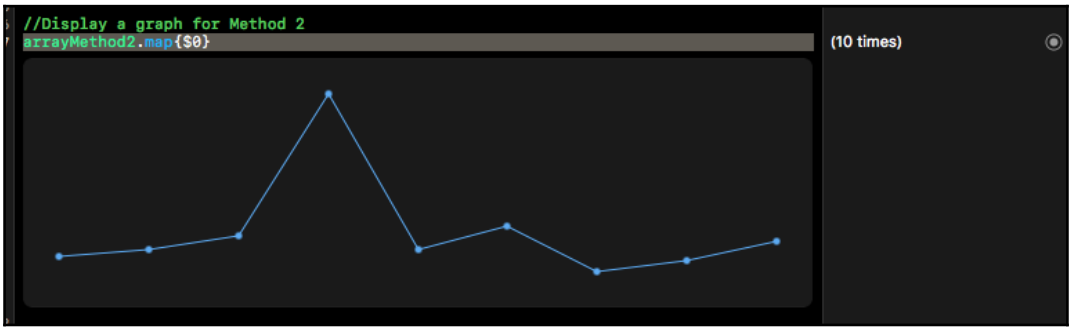
Now, the last step is to confirm all of this by testing it in the playground. In the playground file, uncomment this line (inside of the for loop):

```
myShortenMachine.shortenFast(url:
"http://www.test.com/blog/page/file/" + "\(i)")
```

And add this code below the tests that we have done for method 1, at the end of the file:

```
//Method 2: With direct access with array index
crono.start()
print(myShortenMachine.expandFast(url: "http://shor.ty/0"))
arrayMethod2.append((crono.stop()))
crono.start()
print(myShortenMachine.expandFast(url: "http://shor.ty/100"))
arrayMethod2.append(crono.stop())
crono.start()
print(myShortenMachine.expandFast(url: "http://shor.ty/500"))
arrayMethod2.append(crono.stop())
crono.start()
print(myShortenMachine.expandFast(url: "http://shor.ty/1000"))
arrayMethod2.append(crono.stop())
crono.start()
print(myShortenMachine.expandFast(url: "http://shor.ty/2000"))
arrayMethod2.append(crono.stop())
crono.start()
print(myShortenMachine.expandFast(url: "http://shor.ty/3000"))
arrayMethod2.append(crono.stop())
crono.start()
print(myShortenMachine.expandFast(url: "http://shor.ty/4000"))
arrayMethod2.append(crono.stop())
crono.start()
print(myShortenMachine.expandFast(url: "http://shor.ty/7000"))
arrayMethod2.append(crono.stop())
crono.start()
print(myShortenMachine.expandFast(url: "http://shor.ty/18500"))
arrayMethod2.append(crono.stop())

//Display a graph for Method 2
arrayMethod2.map{$0}
```

These are the same test scenarios, but using our new methods. Again, expand the graph of the last line of code and you should see something like this:



URLShortener – method 2, time complexity

As you can see, this time the results don't grow (just vary a bit) with the number of inputs. We have a constant algorithm of *O(1)*.

In order to see the real magnitude of the difference between method 1 and method 2, check these differences in time spent:

Time access	http://shor.ty/0	http://shor.ty/2000	http://shor.ty/7000	http://shor.ty/18500
Method 1 (s)	0,00155	0,001551	0,00813	0,03130
Method 2 (s)	0,000352	0,00037	0,000352	0,000323
Difference (ms)	**0,1198**	**0,1181**	**0,7778**	**3,0977**

Time differences between method 1 and method 2

In the last row, you can see the time difference between the access time of the same element in both methods. Notice how the bigger the input, the higher the difference, having more than 3 ms of difference in the last test. With more and more URLs, the difference will keep growing because of the Big-O of each method.

What about the space complexity? Well, in both methods we are using an array of the same capacity as the input, so we have an *O(n)* space complexity.

Our first problem, the URL shortener, is solved in an efficient way of *O(k)* and *O(n)* time and space complexity. Well done! Let's see the next scenario.

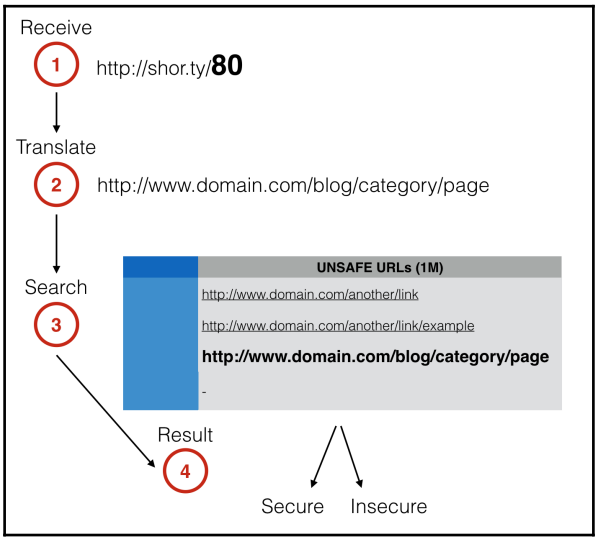
Searching in a huge amount of data

In this section, we are going to develop a new feature for our URL shortener. This feature will need to search for data in a huge table, which, as you can guess, could delay the response of the system. Let's see the feature we are going to add and the solutions that we are going to implement.

The huge blacklist problem

Now, imagine that our URL shortener is very successful. We want to offer a premium service for our users, We will have a new feature, when a user clicks one of our short URLs `http://shor.ty/4324`, we guarantee that the final website `http://www.blog.com/category/page/1` is safe for the user.

In order to achieve a secure service like this, we are going to call a web service from an Internet provider that has over 1 million blacklisted URLs in a table. So when we transform a short URL into the original one, we can check the blacklist table to see if the URL is secure or not and act upon it. The process is described in the following figure:



Method 1 to search for unsecured links

The huge blacklist solution approach

So with the general idea of the previous figure, if we break it down into small pieces, the solution needs the following steps:

1. Receive a short URL.
2. Translate it, as we already know how to get the original URL.
3. Search in the blacklist table to check if the original URL is unsafe.
4. Act upon it, give it to the user if safe, or display some warning if not.

The huge blacklist Swift implementation

Let's translate the four steps into the Swift implementation. Add the following code to the `URLShortener.swift` file, inside the `URLShortener` struct. First, add this property to the struct:

```
//Secure Bloom filter features
public var unsafeUrlsArray:[String] = []
```

In this array, we will simulate a list of 1 million blacklisted URLs. We will just copy the same content that we have in the `urlArray` (all the URLs) after populating it.

Now, add the following method to the struct, too:

```
// Function that receives a short URL, calculate the proper index in
// base 10 and fetch the url for the array of urls directly by its
// position in the array. Then, checks in the unsafe array if the url
// is secure or not
    public func expandSecure(url:String) -> String {
        let hexString = url.components(separatedBy: "/").last!

        //Swift native method to transform from Base 16 String to
        Base 10 Int
        let decimalIndex = Int(hexString, radix: 16)!

        //Own method: To transform from Base 16 String to Base 10
        Int
        //let decimalIndex = URLShortener.base16toBase10(hexString:
        hexString)

            if let url = urlArray[safe: decimalIndex] {
            //Check if the URL is not in the blacklisted array
            for unsafeUrl in unsafeUrlsArray {
                if url == unsafeUrl {
```

```
                    return domainName + "unsafeAddress.html"
                }
            }
            return url
        }
        return domainName + "error404.html"
    }
```

This method is similar to the function `expandFast`, which you already know. But it adds a final check: it searches in the `unsafeArray` if the URL that we get is present or not. If present, it means that it is an insecure URL so it returns a special address instead of the unsafe one.

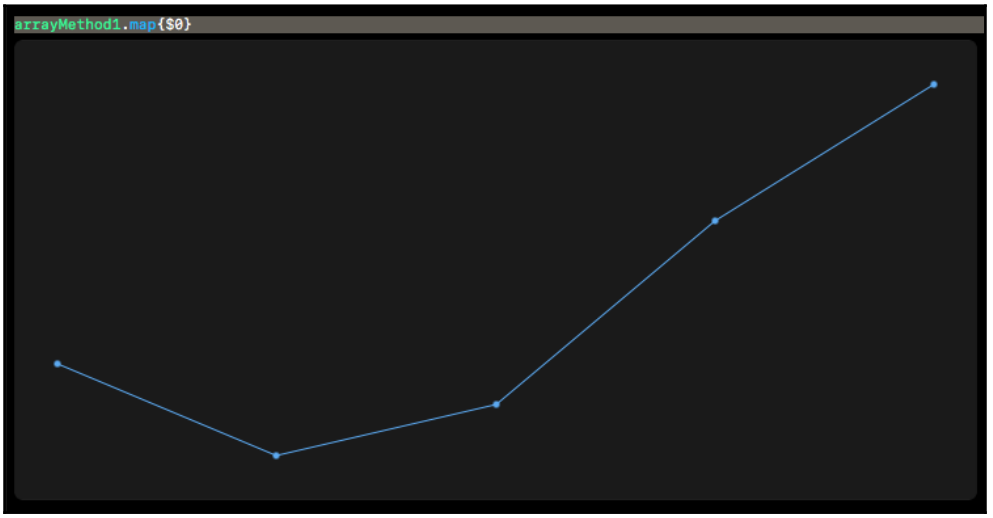
Let's test and measure how much time we spend fetching a URL and checking if it is unsafe with our modified algorithm.

Select **File** | **New** | **Playground** page. Rename the new page of the playground and name it `SecureFeature`. We are going to add 1 million URLs to our `URLShortener` and the same 1 million to the blacklist (all of them are going to be unsafe for our test). Then, we are going to measure how much time we need to fetch (with the safe/unsafe search add-on) URLs at some point of the list. Add the following code to the new playground page:

```
import Foundation
//Create URL shortener
var myShortenMachine = URLShortener(domainName:"http://shor.ty/")
var crono = Stopwatch()

//Blacklist urls
//myShortenMachine.blackList(url:
"http://www.test.com/blog/page/file/0")
//myShortenMachine.blackList(url:
"http://www.test.com/blog/page/file/100000")
//myShortenMachine.blackList(url:
"http://www.test.com/blog/page/file/1000000")

//Setup our URL list and for the example, add ALL to the unsafe
URLs list
for i in 0...1000000 {
    myShortenMachine.shortenFast(url:
    "http://www.test.com/blog/page/file/" + "\(i)")
}

myShortenMachine.unsafeUrlsArray =
myShortenMachine.urlArray.map{$0}

//Now search for specific URLs and measure the time spent to do it.
var arrayMethod1:[TimeInterval] = []
```

```
//Check for a blacklisted url
crono.start()
print(myShortenMachine.expandSecure(url: "http://shor.ty/0"))
arrayMethod1.append(crono.stop())
crono.start()
print(myShortenMachine.expandSecure(url: "http://shor.ty/2000"))
arrayMethod1.append(crono.stop())
crono.start()
print(myShortenMachine.expandSecure(url: "http://shor.ty/7000"))
arrayMethod1.append(crono.stop())
crono.start()
print(myShortenMachine.expandSecure(url: "http://shor.ty/18500"))
arrayMethod1.append(crono.stop())
crono.start()
print(myShortenMachine.expandSecure(url: "http://shor.ty/F4240"))
arrayMethod1.append(crono.stop())
arrayMethod1.map{$0}
```

Now let's see the graph of the measures stored in `arrayMethod1`, displayed by the `map` function:



Method 1 – search for unsecured links result graph

The data is as follows:

Time access	http://shor.ty/0	http://shor.ty/2000	http://shor.ty/7000	http://shor.ty/18500	http://shor.ty/F4240
Method 1 (s)	0,0013489	0,00048184	0,00095915	0,002686	0,0039761

Method 1 – search for unsecured links data results

As you can see, we have an algorithm that grows very quick when the input size grows. A case between $O(n^2)$ and $O(nlog(n))$. Why does this happen? Well, let's look again at our new `expandSecure` function. We have added a loop to search for unsafe URLs inside of the blacklist array. This is the code to do so:

```
//Check if the URL is not in the blacklisted array
for unsafeUrl in unsafeUrlsArray {
    if url == unsafeUrl {
        return domainName + "unsafeAddress.html"
    }
}
```

A loop that in the worst-case will execute 1 million times in our example case with 1 million unsafe URLs stored. And this will happen for each search! How can we improve this? We cannot access the array by the index like before, because we need to know if the URL is there or not, we don't have an index to access this time. How can we do this safe/unsafe check more quickly than with a brute force search? The answer is a Bloom filter.

Method 2 – the Bloom filter solution

Bloom filters are used to check the existence of an element in a set in a probabilistic space efficient way.

It has false positive matches, but it doesn't have false negatives. That means in plain language that a Bloom filter can tell if an element is definitely not present in a set and if an element is maybe present in a set. Bloom filters don't store the actual value.

Being a probabilistic and efficient data structure produces the false positives. The more memory-efficient we set up the Bloom filter, the more false positives we can produce. But it never produces false negatives. When a result is negative, it means that at 100% the element is not present in the analyzed set.

Bloom filters contain a bit array of zeros and ones and a variable number of hash algorithms. They work in the following way:

- Initially, all bits of the Bloom filter array are set to `0s`.
- When a new element is inserted or processed by the Bloom filter, it applies all its hash algorithms on the element, one by one.
- For each hash algorithm applied, we obtain a hash-result. It will be a number from [0 to `bitArray.count`]. We put a 1 in the bit of `bitArray`[hash-result] at that position [hash-result].
- This is repeated for each hash algorithm for the element and the proper bits are set to `1s`.
- To test the membership of an element, we just have to hash the element with all the hash algorithms and check if the proper bits were set to `1` in the `bitArray`. If all of them are set to `1`, the element may be present. If any bit fails to be 1, we know that the element is definitely not present.

So with a Bloom filter with the proper number of hashes and bit array size, we can know if an element is not present in a set, and with 99% of probability, if it is present. We can get benefit of the Bloom filter to our problem in order to check if a URL is present in the blacklist or not. If it is not, we are 100% sure that the URL is secure. If the Bloom filter is positive, and to discard a false positive we will search manually for the element (wasting more time, but having a correct result 100% of the time). Let's view the overall process:

1. **Preprocess**: Initially, we have to run all the unsecured URLs thought the Bloom filter once. Now we will have in memory an array of bits with the information about all the unsecured URLs. Each unsecured URL has set up some bits to `1` (some of them are common, that's why there could be false positives).
2. Now our process receives a short URL to translate.
3. The short URL is translated into the initial long URL.
4. We run the long URL through the Bloom filter to check existence in the blacklist.
5. If the Bloom filter result is negative, the URL is definitely not in the unsecured list.
6. If the Bloom filter is positive, we will search for it manually to discard a false positive and return the result to the user.

Let's implement a Bloom filter in our `URLShortener`. In the `Sources` folder, add a new Swift file and name it `BloomFilter.swift`. Add the following code to the new file:

```
public struct BloomFilter<T> {
    var arrayBits:[Bool] = Array(repeating: false, count: 17)
    var hashFunctions:[(T) -> Int]
```

```
    public init(hashFunctions:[(T) -> Int]) {
        self.hashFunctions = hashFunctions
    }

    // Execute each hash function of our filter agains the element
    // and returns an array of Ints as result
    private func calculeHashes(element:T) -> [Int]{
        return hashFunctions.map() {
            hashFunc in abs(hashFunc(element) % arrayBits.count)
        }
    }

    // Insert an element results in converting some bits of our
    // arrayBits into '1's, depending on the results of each hash
    // function
    public mutating func insert(element: T) {
        for hashValue in calculeHashes(element:element) {
            arrayBits[hashValue] = true
        }
    }

    //Check for existence of an element in the Bloom Filter
    public func exists(element:T) -> Bool {
        let hashResults = calculeHashes(element: element)
        //Check hashes agains the array of the filter
        let results = hashResults.map() { hashValue in
        arrayBits[hashValue] }
        //NO is 100% true. YES could be a false positive.
        let exists = results.reduce(true, { $0 && $1 })
        return exists
    }
}

//Hash functions see http://www.cse.yorku.ca/~oz/hash.html
func djb2(x: String) -> Int {
    var hash = 5381
    for char in x.characters {
        hash = ((hash << 5) &+ hash) &+ char.hashValue
    }
    return Int(hash)
}

func sdbm(x: String) -> Int {
    var hash = 0
    for char in x.characters {
        hash = char.hashValue &+ (hash << 6) &+ (hash << 16) &-hash
    }
```

```
        return Int(hash)
    }
```

The code is easy to follow, we use a struct to represent the Bloom filter, which has an array of bool to represent the bits `arrayBits` and a property to store multiple hash functions `hashFunctions`. It has an `init`, a method to insert a new element, and a method to check for the existence of an element (exists).

It uses two standard hash algorithms: `djb2` and `sdbm`. Now, in the `URLShortener.swift` file, add the following:

Add a new property to the `URLShortenerstruct`:

```
public var bloomFilter = BloomFilter<String>(hashFunctions:
[djb2, sdbm])
```

You can define it below this other property:

```
public var unsafeUrlsArray:[String] = []
```

Now, add a new function to blacklist a specific URL:

```
//Blacklist one url
public mutating func blackList(url:String) {
    self.bloomFilter.insert(element: url)
}
```

It just inserts a URL into the Bloom filter, to put the proper bits to `1`, and later, it checks for existence. To do it, add the following method:

```
// Check if a url is secure. If the Bloom Filter says that is not
// present in the insecure array, is secure 100%. If the Bloom Filter
// says it is present in the insecure array, could be a false
// positive, we search manually to confirmn it.
    public func isSecure(url:String) -> Bool {
        let initialUrl = self.expandFast(url: url)
        let exists = self.bloomFilter.exists(element: initialUrl)
        if exists == true {
            //Check if the URL is not in the blacklisted array
            for unsafeUrl in unsafeUrlsArray {
                if initialUrl == unsafeUrl {
                    return false
                }
            }
            return true
        } else {
              return true
          }
```

```
    }
```

Now, in order to test our new function, let's do some changes in the `SecureFeature` playground page.

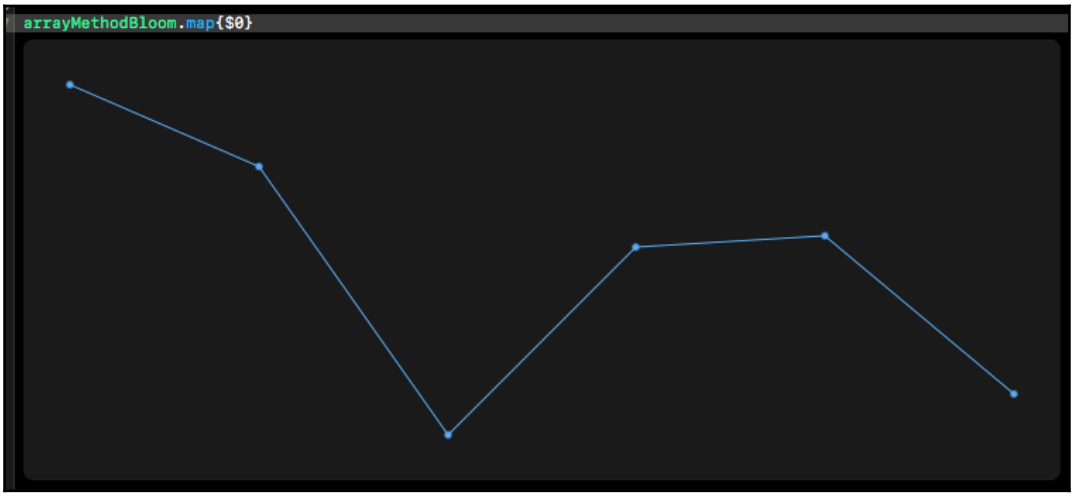
Uncomment these lines, in order to have three unsecured URLs in the Bloom filter ready to test:

```
myShortenMachine.blackList(url:
"http://www.test.com/blog/page/file/0")
myShortenMachine.blackList(url:
"http://www.test.com/blog/page/file/100000")
myShortenMachine.blackList(url:
"http://www.test.com/blog/page/file/1000000")
```

Now, at the bottom of the page, add this code to make the tests and measures and then represent them in a graph with the `map` function:

```
//Now perform the check with the Bloom Filter
var arrayMethodBloom:[TimeInterval] = []
crono.start()
myShortenMachine.isSecure(url: "http://shor.ty/0")
arrayMethodBloom.append(crono.stop())
crono.start()
myShortenMachine.isSecure(url: "http://shor.ty/2000")
arrayMethodBloom.append(crono.stop())
crono.start()
myShortenMachine.isSecure(url: "http://shor.ty/7000")
arrayMethodBloom.append(crono.stop())
crono.start()
myShortenMachine.isSecure(url: "http://shor.ty/18500")
arrayMethodBloom.append(crono.stop())
crono.start()
myShortenMachine.isSecure(url: "http://shor.ty/F4240")
arrayMethodBloom.append(crono.stop())
crono.start()
myShortenMachine.isSecure(url: "http://shor.ty/FFFFF")
arrayMethodBloom.append(crono.stop())
arrayMethodBloom.map{$0}
```

Run the playground. It could take a while if you have set 1,000,000 elements in the `for` loop. When it has finished, check the results graph:



Method 2 – Bloom filter graph

And the data results against method 1:

Time access	http://shor.ty/0	http://shor.ty/2000	http://shor.ty/7000	http://shor.ty/18500	http://shor.ty/F4240
Method 1 (s)	0,0013489	0,00048184	0,00095915	0,002686	0,0039761
Method 2 Bloom (s)	0,0002639	0,0002479	0,00019502	0,00023198	0,0002341
Difference (ms)	**0,1085**	**0,023394**	**0,076413**	**0,245402**	**0,3742**

Method 2 – Bloom filter versus method 1 data results

We can extract the following from the table:

- Method 1, based on brute force search, has a time complexity between $O(n^2)$ and $O(nlog(n))$, which is not good at all for big input sizes.
- Method 2, with Bloom filter, has much better time complexity $O(1)$, constant time. Great!

It is clear now that using a Bloom Filter makes our algorithm independent from the size of our unsecured links list. We will have preprocessing time spent calculating the Bloom filter bits for each URL, but after processing that one time per URL, checking for the existence of an element is constant time. In terms of space complexity, with the Bloom filter we need to add some memory to save the bit array, but it is worth it.

Summary

In this chapter, we have learned how to deal with problems that require algorithms and data structures by:

1. Creating a high-level approach of the solution.
2. Writing and describing the Swift implementation.
3. Calculating Big-O complexities of our solution to check if the algorithm behaves properly for a real-world situation.
4. Measuring and detecting bottlenecks.
5. Modifying them to achieve a better performance with an alternative solution.

Moreover, and in order to learn this, we have seen what a Bloom filter is and how to display graphs in the playground to analyze results more easily.

We have created two solutions for two problems, and after a Big-O analysis and with the help of algorithms and data structures, we have improved our solutions a lot and we have made our code more efficient. That was the goal of this chapter, and of the entire book itself. Congratulations!

Epilogue

After nine chapters, you have learned about Swift, data structures, and algorithms. Now it is time for you to put all these concepts in practice in real-world scenarios. When facing a complex task, remember the cons and benefits of the different data structures, take into account the time and space complexity of each algorithm, and measure your first solution in order to improve it as needed. You have learned the basic concepts to start making great solutions to complex problems with the help of data structures and algorithms in Swift. Now it's your turn!

Index

C

D

E

M

N

O

P

Q

R

S

T

U

V

W