



Up to date for
Kotlin 1.3

Data Structures & Algorithms

in Kotlin

FIRST EDITION

Implementing Practical Data Structures in Kotlin

By the raywenderlich Tutorial Team

Irina Galata & Matei Suica

Data Structures & Algorithms in Kotlin

Irina Galata & Matei Suica

Copyright ©2019 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

About the Authors

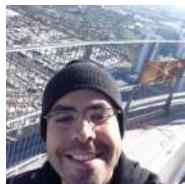


Irina Galata is an author of this book. She is a software engineer in Linz, Austria, working at Runtastic. She is passionate about programming and exploring new technologies. You can follow her on twitter @igalata13.

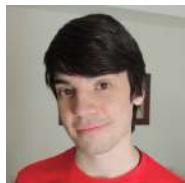


Matei Suica is an author of this book. He is a software developer that dreams about changing the world with his work. From his small office in Romania, Matei is trying to create an App that will help millions. When the laptop lid closes, he likes to go to the gym and read. You can find him on Twitter or LinkedIn: @mateisuica.

About the Editors



Bruno Lemgruber is the technical editor of this book. He is an iOS and Android developer who enjoys being challenged and working on projects that require him to work outside his comfort and knowledge set, as he continues to learn new languages and development techniques. Nowadays, he works in a bank from Brazil (@SICOOB_oficial) in the iOS team. He loves to drink craft beer when he has a free time! You can follow him on twitter @brunoteixeiralc.



Márton Braun is a technical editor of this book. He is a Kotlin enthusiast since the 1.0 of the language, and an aspiring writer, speaker, educator. He's working as an Android developer and teaches Kotlin and Android in university courses. Creator of the MaterialDrawerKt and Krate libraries. He occasionally gets addicted to StackOverflow, where he's one of the top contributors under the Kotlin tag.



Tammy Coron is an editor of this book. Tammy is an independent creative professional and the host of Roundabout: Creative Chaos. She's also a Development Editor at The Pragmatic Bookshelf, a Sr. Editor at Razeware, and a content contributor at Creative Bloq, Lynda.com, iMore, and raywenderlich.com.



Massimo Carli is the final pass editor of this book. Massimo has been working with Java since 1995 when he co-founded the first Italian magazine about this technology <http://www.mokabyte.it>. After many years creating Java desktop and enterprise application, he started to work in the mobile world. In 2001 he wrote his first book about J2ME. After many J2ME and Blackberry applications, Massimo then started to work with Android in 2008. The same year he wrote the first Italian book about Android, a best seller on Amazon.it. That was the first of a series of 10 books about Android and Kotlin. Massimo worked at Yahoo and Facebook and he's actually Senior Mobile Engineer at Spotify. He's a musical theatre lover and a supporter of the soccer team S.P.A.L.

About the Contributors

We'd like to acknowledge the work of the authors of *Data Structures & Algorithms in Swift*, the content of which served as the basis for this book.

- **Vincent Ngo**
- **Kelvin Lau**

We'd also like to acknowledge the efforts of the following contributors to the Swift Algorithm Club GitHub repo (<https://github.com/raywenderlich/swift-algorithm-club>), upon whose work portions of this book are based.

- **Matthijs Hollemans**, the original creator of the Swift Algorithm Club.

We'd also like to thank the following for their contributions to the repo:

- **Donald Pinckney**, Graph <https://github.com/donald-pinckney>
- **Christian Encarnacion**, Trie and Radix Sort <https://github.com/Thukor>
- **Kevin Randrup**, Heap <https://github.com/kevinrandrup>
- **Paulo Tanaka**, Depth First Search <https://github.com/paulot>
- **Nicolas Ameghino**, BST <https://github.com/nameghino>
- **Mike Taghavi**, AVL Tree
- **Chris Pilcher**, Breadth First Search

About the Artist



Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Table of Contents: Overview

Book License	14
Who This Book Is For	15
What You Need	16
Book Source Code & Forums	17
About the Cover	19
Section I: Introduction to Data Structures & Algorithms.....	20
Chapter 1: Kotlin & Kotlin Standard Library	21
Chapter 2: Complexity	39
Section II: Elementary Data Structures	52
Chapter 3: Linked List.....	53
Chapter 4: Stack Data Structures	85
Chapter 5: Queues	94
Section III: Trees.....	125
Chapter 6: Trees	127
Chapter 7: Binary Trees	140
Chapter 8: Binary Search Trees.....	152
Chapter 9: AVL Trees.....	173
Chapter 10: Tries	191
Chapter 11: Binary Search	204
Chapter 12: The Heap Data Structure	213

Chapter 13: Priority Queues	238
Section IV: Sorting Algorithms	251
Chapter 14: $O(n^2)$ Sorting Algorithms	253
Chapter 15: Merge Sort.....	268
Chapter 16: Radix Sort	279
Chapter 17: Heap Sort.....	289
Chapter 18: Quicksort	298
Section V: Graphs	316
Chapter 19: Graphs	318
Chapter 20: Breadth-First Search	344
Chapter 21: Depth-First Search	355
Chapter 22: Dijkstra's Algorithm	368
Chapter 23: Prim's Algorithm	387
Conclusion.....	403

Table of Contents: Extended

Book License	14
Who This Book Is For	15
What You Need	16
Book Source Code & Forums	17
About the Cover	19
Section I: Introduction to Data Structures & Algorithms	20
Chapter 1: Kotlin & Kotlin Standard Library	21
Introduction to Kotlin	22
The Kotlin Standard Library	30
Key points	38
Chapter 2: Complexity	39
Time complexity.....	40
Other time complexities.....	47
Comparing time complexity.....	47
Space complexity.....	48
Key points	51
Section II: Elementary Data Structures	52
Chapter 3: Linked List.....	53
Node	54
LinkedList	55
Adding values to the list	56
Removing values from the list	61
Kotlin collection interfaces.....	66
Becoming a Kotlin mutable collection	67

Challenges.....	76
Key points	84
Chapter 4: Stack Data Structures	85
Stack operations	86
Implementation	87
push and pop operations	87
Challenges.....	91
Key points	93
Chapter 5: Queues	94
Common operations	95
Example of a queue.....	96
List-based implementation.....	97
Doubly linked list implementation	100
Ring buffer implementation.....	104
Double-stack implementation.....	108
Challenges	115
Key points.....	124
Section III: Trees	125
Chapter 6: Trees	127
Terminology	128
Implementation	129
Traversal algorithms	131
Challenges	137
Key points.....	139
Chapter 7: Binary Trees	140
Implementation	141
Traversal algorithms	143
Challenges	147
Key points.....	151

Chapter 8: Binary Search Trees	152
Case study: array vs. BST	153
Implementation	157
Challenges	168
Key points	172
Chapter 9: AVL Trees	173
Understanding balance	174
Implementation	175
Challenges	187
Key points	190
Chapter 10: Tries	191
Example	192
Implementation	194
Challenges	202
Key points	203
Chapter 11: Binary Search	204
Example	205
Implementation	206
Challenges	208
Key points	212
Chapter 12: The Heap Data Structure	213
What is a heap?	214
The heap property	214
Heap applications	215
Common heap operations	216
Sorting and comparing	216
How do you represent a heap?	218
Inserting into a heap	220
Removing from a heap	223
Removing from an arbitrary index	227

Searching for an element in a heap	228
Heapify an array.....	229
Testing.....	231
Challenges	233
Key points.....	237
Chapter 13: Priority Queues.....	238
Applications	239
Common operations	239
Implementation	240
Challenges	245
Key points.....	250
Section IV: Sorting Algorithms	251
Chapter 14: $O(n^2)$ Sorting Algorithms	253
Bubble sort.....	254
Selection sort.....	257
Insertion sort	260
Generalization	262
Challenges	264
Key points.....	267
Chapter 15: Merge Sort.....	268
Implementation	270
Performance.....	274
Challenges	275
Key points.....	278
Chapter 16: Radix Sort.....	279
Example.....	280
Implementation	281
Challenges	284
Key points.....	288

Chapter 17: Heap Sort	289
Getting started	290
Example	290
Implementation	293
Performance	295
Challenges	296
Key points	297
Chapter 18: Quicksort	298
Example	299
Partitioning strategies	300
Effects of a bad pivot choice	307
Challenges	313
Key points	315
Section V: Graphs	316
Chapter 19: Graphs	318
Weighted graphs	319
Common operations	321
Defining a vertex	323
Defining an edge	323
Adjacency list	324
Implementation	325
Adjacency matrix	331
Implementation	332
Graph analysis	338
Challenges	340
Key points	343
Chapter 20: Breadth-First Search	344
Example	345
Implementation	348
Performance	349

Challenges	350
Key points.....	354
Chapter 21: Depth-First Search	355
DFS example	356
Implementation	360
Performance.....	362
Challenges	363
Key points.....	367
Chapter 22: Dijkstra's Algorithm	368
Example.....	369
Implementation	377
Trying out your code.....	382
Performance.....	383
Challenges	384
Key points.....	386
Chapter 23: Prim's Algorithm	387
Example.....	389
Implementation	393
Testing your code	397
Performance.....	398
Challenges	399
Key points.....	402
Conclusion	403

Book License

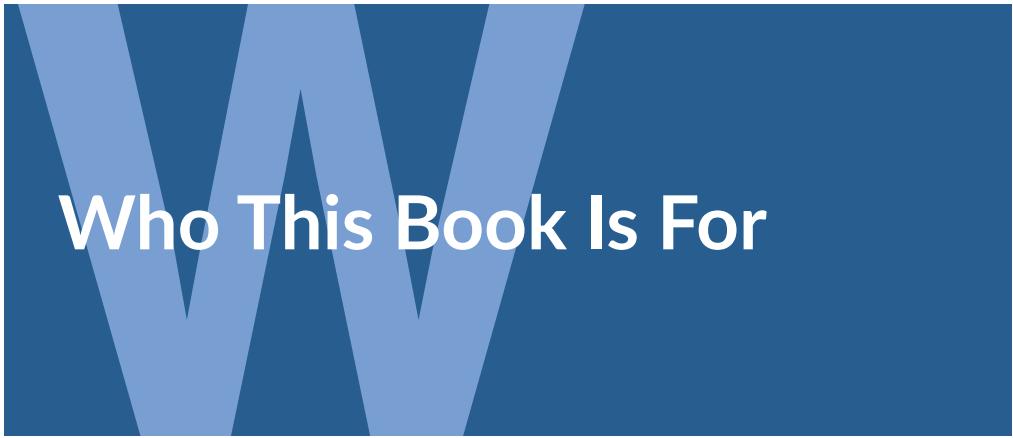
By purchasing *Data Structures & Algorithms in Kotlin*, you have the following license:

- You are allowed to use and/or modify the source code in *Data Structures & Algorithms in Kotlin* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Data Structures & Algorithms in Kotlin* in as many apps as you want, but must include this attribution line somewhere inside your app: “Artwork/images/designs: from *Data Structures & Algorithms in Kotlin*, available at www.raywenderlich.com”.
- The source code included in *Data Structures & Algorithms in Kotlin* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Data Structures & Algorithms in Kotlin* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.





Who This Book Is For

This book is for developers who are comfortable with Kotlin and want to ace whiteboard interviews, improve the performance of their code, and ensure their apps will perform well at scale.

If you're looking for more background on the Kotlin language, we recommend our book, *Kotlin Apprentice*, which goes into depth on the Kotlin language itself:

- <https://store.raywenderlich.com/products/kotlin-apprentice>

If you want to learn more about Android app development in Kotlin, we recommend working through our classic book, *Kotlin Apprentice*:

- <https://store.raywenderlich.com/products/kotlin-apprentice>



What You Need

To follow along with this book, you need:

- **IntelliJ IDEA Community Edition 2019.1.x:** Available at <https://www.jetbrains.com/idea/>. This is the environment in which you'll develop most of the sample code in this book.
- **Kotlin playground:** You can also use the Kotlin Playground available at the Kotlin home page at <https://play.kotlinlang.org>.





Book Source Code & Forums

If you bought the digital edition

The digital edition of this book comes with the source code for the starter and completed projects for each chapter. These resources are included with the digital edition you downloaded from store.raywenderlich.com.

If you bought the print version

You can get the source code for the print edition of the book here:

www.raywenderlich.com/store/data-structures-and-algorithms-in-kotlin-source-code/

Forums

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book or to submit any errors you may find.

Digital book editions

We have a digital edition of this book available in both ePUB and PDF, which can be handy if you want a soft copy to take with you, or you want to quickly search for a specific term within the book.

Buying the digital edition version of the book also has a few extra benefits: free updates each time we update the book, access to older versions of the book, and you can download the digital editions from anywhere, at anytime.



Visit our *Data Structures & Algorithms in Kotlin* store page here:

- <https://store.raywenderlich.com/products/data-structures-and-algorithms-in-kotlin>.

And if you purchased the print version of this book, you're eligible to upgrade to the digital editions at a significant discount! Simply email support@razeware.com with your receipt for the physical copy and we'll get you set up with the discounted digital edition version of the book.



About the Cover

Weaver birds are known for their intricate and sophisticated spherical nests, widely considered some of the most elegant animal-built structures in the world. Not only are these complex nests 100% waterproof, humans have never figured out how to reproduce these structures on their own.

Weaver birds rely on their elegant weaving techniques to build robust structures, just as you rely on elegant data structures and algorithms to create robust code. After reading Data Structures & Algorithms in Kotlin, you'll be able to *weave* structures and algorithms into your code to make your apps more performant and robust. Waterproof? Well, that's a different story!



Section I: Introduction to Data Structures & Algorithms

The chapters in this short but important section explain what's built into the Kotlin Standard Library and how you use it in building your apps. You'll learn why one algorithm may be better suited than another. You'll also learn what the Big-O notation is and how you can continue to answer the question: "Can we do better?"

Specifically, you'll learn:

- **Chapter 1: Kotlin & Kotlin Standard Library:** The Kotlin Standard Library refers to the framework that defines the core elements of the Kotlin language. Inside the Kotlin Standard Library, you'll find a variety of tools and data types to help build your Kotlin apps, including data structures.
- **Chapter 2: Complexity:** Answering the question, "Does it scale?" is all about understanding the complexity of an algorithm. The Big-O notation is the primary tool that you'll use to think about algorithmic performance in the abstract and independent hardware or language. This chapter will prepare you to think in these terms.

These fundamentals will set you on your way; before you know it, you'll be ready for the more advanced topics that follow.



1 Chapter 1: Kotlin & Kotlin Standard Library

By Matei řuică

Kotlin is a modern, multi-paradigm, programming language developed by JetBrains. It first appeared in 2011 and slowly evolved into one of the coolest languages available today.

One of the reasons developers love Kotlin so much is because it makes app development easier by providing a significant amount of out-of-the-box classes and utilities. Kotlin's classes and utilities are wrapped inside the **Kotlin Standard Library**, which contains the core components of the Kotlin language. Inside this library, you'll find a variety of tools and data types to help build your apps.

Before you start building your own custom data structures, it's essential to know about the primary data structures that the Kotlin Standard Library already provides.

In this chapter, you'll start by learning a few things about Kotlin like variables, data types, optionals, conditionals, loops and functions. You'll then focus on two specific data structures, `List` and `Map`, both of which are included with the Kotlin Standard Library. You'll end this chapter with a discussion about mutability in the context of these two data structures.



Introduction to Kotlin

To understand data structures and algorithms in Kotlin, you first need to understand the main features of the language. But don't worry: There's nothing overly complicated about Kotlin, especially if you have experience with other modern programming languages. However, regardless of your experience, there are a few things you need to know before diving deep into the details of data structures:

- How to declare variables and functions.
- How to create custom data types.
- How to manipulate data in loops and decision-making structures.

Once you get comfortable with the basics, you can move on to something a little more complicated: Generics. You can find most of the code of this chapter in the provided projects.

Ready to get started?

Variables and data types

A variable is a way to store information. Typically, a variable has a name and a data type. Variables can also have modifiers that add extra options or restrictions to it.

In Kotlin, there are two types of variables, `val` and `var`:

```
val name = "Bill Clinton"  
var country = "Romania"
```

The difference between `val` and `var` is that variables declared with `val` cannot be reassigned:

```
name = "Matei Suica" // compile error  
country = "Pakistan" // Ok
```

Since `name` was defined with `val` its value cannot be changed from "Bill Clinton", but since `country` was defined with `var`, its value can be updated.

With regard to data types: The Kotlin compiler can sometimes determine the data type of the variable. When it does, it's referred to as **type inference**, which most modern programming languages have.

The variables in the previous example are of type `String`. This is clear to the compiler because they were initialized when they were declared. Since nothing was



misleading about them, you did not have to declare the data type. However, this may not always be the case. For example, you can declare a variable that references a number but initializes it later:

```
var score: Int
```

In this example, since no initial value is set, its type cannot be inferred and so you must explicitly set the data type for `score`.

There are several data types in Kotlin that are already defined for you. The Kotlin Standard Library includes more than are covered here, but the basic types are:

- **Numbers:** `Double`, `Float`, `Long`, `Int`, `Short`, `Byte`.
- **Characters:** `Char`, `String`.
- **Other:** `Boolean`, `Array`.

As you work through this book, you'll encounter most of these data types. However, at this point, you don't need to study their specifics, only acknowledge their existence. Later, you'll create complex structures to store these kinds of data types.

Optionals and null-safety

Many programming languages, including Kotlin, have the concept of a `null` value. You can assign `null` to a variable whenever you want to signal that an object has no value.

For example, if you have a variable that can hold a `Car` but you've not yet created a `Car` object, the variable can hold a `null`:

```
var car: Car? = null
```

Upon object creation, you could easily reassign the variable:

```
car = Car("Mercedes-Benz")
```

The problem with the presence of `null` is that you might try to use it. Assuming that `Car` contains a `drive()` method, you might decide to try something like this:

```
car.drive()
```

When you have a value assigned, like `car = Car("Mercedes-Benz")`, you won't have an issue; however, if you try to do this with a `null` value, the program will crash. This is where the infamous **NPE**, Null-Pointer Exception, was born.



To prevent an NPE, Kotlin has a neat system baked into the language. Noticed the ? after the Car data type in the first declaration? That question mark changes the variable type to an **optional**. An optional tells the compiler that your object *could* contain a null value. This small detail triggers a chain reaction in the code.

For example, with an Optional, you cannot use:

```
car.drive()
```

Instead, you need to use the safe-call operator ?: :

```
car?.drive()
```

Using a safe-call operator means that this function will only execute if the object is not null.

To assign values to variables that hold null values, you can use ?:, which is also known as the Elvis operator:

```
val immutableCar: Car = car ?: Car("Porche")
```

This code does a lot in a single line:

1. Creates a variable immutableCar that cannot be reassigned.
2. The immutableCar is not an Optional now. You can be sure that there's a real car that you can drive() in that variable.
3. The immutableCar can be either the same as the car or a Porche if the car is null (not a real car).

These language features are nice, but there are cases where you don't want to play by the rules.

You know that your variable is not a null — even though it's an Optional — and you need to tell the compiler to use the value that it holds. For this, Kotlin has the not-null assertion operator !!. You can use it instead of the safe-call operator:

```
car!!.drive()
```

This calls drive() on the non-null value that the car holds; if, however, it holds a null, it'll throw an NPE. Therefore, you should think twice before using it. That could be why the JetBrains team made this operator a double-bang: Think! Twice!



Conditional statements

Programs in Kotlin execute linearly; in other words, one line at a time. While this is easy to follow and clean, it's not very useful. There are a lot of situations where making a decision or repeating a step can come in handy. Kotlin has structures that resolve both of these problems in a concise fashion.

For decision making, Kotlin has two constructs, `if-else` and `when`. Unlike other languages, there's no ternary operator in Kotlin; you have to use `if-else` to get the same result. Here's an example:

```
val a = 5
val b = 12
var max = -1

if (a > b) {
    max = a
} else {
    max = b
}

println(max) // prints 12
```

The code above makes a decision based on the condition inside the brackets.

If `a > b` is `true`, the first block of code is executed, and `max` takes the value of `a`. If the condition is `false`, then `max` takes the value of `b`.

In the structure, the `else` part is optional. You might want only to do something if the condition is `true` but not when it's `false`. Instead of leaving the `else` block empty, you can omit it.

`when` is much like a series of `if-else` that can handle many cases:

```
val groupSize = 3

when (groupSize) {
    1 -> println("Single")
    2 -> println("Pair")
    3 -> { // Note the block
        println("Trio")
    }
    else -> println("This is either nobody or a big crowd")
}
```

In this example, the `when` structure makes a decision based on the value of `groupSize`. It has some particular cases like 1, 2 or 3, and then an `else` clause that handles everything that isn't specified above.



For the `when` structure, the `else` can be optional if the compiler determines that you already handled all of the possible values. You won't dig into those right now because you need to learn more language features first.

Loops

There are two types of loops in Kotlin, `for` and `while`. Although you can do just fine only using `while`, in some situations, using `for` is easier and more elegant.

Let's start with the elegant one:

```
for (i in 1..3) {  
    println(i)  
}
```

`for` can iterate over any iterable collection of data. In this example, `1..3` creates an `IntRange` that holds the numbers from 1 to 3. The `i` variable takes each value, one at a time, and goes into the code block with it. In the block, `println()` is executed and the value of `i` goes into the standard output.

Here's a more generic example:

```
for (item in collection) println(item)
```

This prints all of the items in the `collection`.

The second type of loop is the `while` loop, which executes the same block of code as long as its condition remains `true`:

```
var x = 10  
while (x > 0) {  
    x--  
}
```

This code starts with `x` having the value of `10` and since `10` is greater than `0`, it executes the code inside the block. There, `x` decreases by 1, becoming `9`. Then, the loop goes back to the condition: "Is `9` greater than `0`?". This continues until eventually `x` gets to `0` and the condition becomes `false`.

There is a variation of `while` known as `do-while`. The `do-while` loop first executes the code and then checks for the condition to continue. This ensures that the block of code is executed at least once:

```
var x = 10  
do {
```

```
x--  
} while (x > 0)
```

One thing to notice with `while` loops is that you can easily create an infinite loop. If you do, your program will get stuck and eventually die in a pitiful `StackOverflowException` or something similar:

```
var x = 10  
while (x > 0) {  
    x++  
}  
println("The light at the end of the tunnel!")
```

This time, `x` gets incremented instead of decremented: 10, 11, 12, 13, and so on. It never gets less than or equal to 0, so the `while` loop has no reason to stop. In other words, you'll never get to see the light at the end of the tunnel.

Functions

Functions are an important part of any programming language, especially in Kotlin as it's a multi-paradigm programming language. Kotlin has a lot of functional programming features, so it treats functions with the respect they deserve!

In general, programming is based on small units of code that can be abstracted and reused. Functions are the smallest units of code that you can easily reuse. Here's an example of a function:

```
fun max(a: Int, b: Int): Int {  
    return if (a > b) a else b  
}
```

This is a simple function that compares two numbers and determines which is higher.

Functions are declared using the `fun` keyword, followed by the name of the function. By naming functions, you can then call them using their name, as you'll see momentarily.

Functions can also have a list of **parameters**. Each parameter has a name that you can use in the code block; parameters also include their data type. This function has two parameters, `a` and `b`; however, functions can also have more parameters or no parameters at all.

After a colon `:`, there's another data type; this is the function's return value data type. In other words, the result of this function will have that data type. A function



cannot have more than one return type; however, it's possible for it to have no return type at all. In that case, the function executes its block, usually handling some smaller or commonly shared operation.

Lastly, there's the function's code block. Because this function has a return type, it also needs to return a value. In this case, it returns either `a` or `b`, and since both are `Ints`, the return type is also `Int`.

Here's an example of a function that has no return type:

```
fun printMax(c: Int, d: Int) {  
    val maxValue = max(c, d)  
    println(maxValue)  
}
```

Again, because this function does not declare a return type, there's no need for a `return` keyword. So what's the point of this function if it contains no return value?

Well, if you look closely, you'll see that this function calls `max` by its name, and passes in two parameters, `c` and `d` (which `max` renames to `a` and `b`). From there, `printMax` takes the result of `max`, stores it into a variable named `maxValue` and prints it the console.

Note: This chapter does not cover higher-order functions and lambdas as these concepts are more complex. You will, however, touch on them in later chapters.

Generics

Generics are a great way to abstract your code whenever you can manipulate multiple data types in the same way.

Consider a class that emulates a box. A class is simply a collection of data and functions that are logically grouped to perform a set of specific tasks. When creating a class, think about how you might use it. In this case, with a box, you:

- Put something in it.
- Grab something out of it.
- Check if the box is empty.

Here's some code that can perform these tasks:

```
class Box {  
    var content: Any? = null  
  
    fun put(content: Any?) {  
        this.content = content  
    }  
  
    fun retrieve(): Any? {  
        return content  
    }  
  
    fun isEmpty(): Boolean {  
        return content == null  
    }  
}
```

This is a simple class that can store a value via `put()`, can retrieve a value via `retrieve()`, and can check if the box is empty via the `isEmpty()` method.

Since you want the box to store different kinds of objects, the type is set to `Any` since the `Any` class is the superclass of all objects in Kotlin.

This could work, but there's one drawback: Once you put something into the box, you lose the knowledge of the object's type since you had to use the `Any` type to store any kind of object.

To get a more specialized box, you could replace `Any` with the data type you need; for example, a `Cat` or a `Radio`. But you'd need to create a different type of `Box` for every type of object you'd want to store, i.e. you'd have to create `CatBox` and `RadioBox` separately.

Generics are an excellent way to keep the code abstract and let the objects specialize once instantiated. To abstract `Box`, you can write it like this:

```
class Box<T> {  
    var content: T? = null  
  
    fun put(content: T?) {  
        this.content = content  
    }  
  
    fun retrieve(): T? {  
        return content  
    }  
  
    fun isEmpty(): Boolean {  
        return content == null  
    }  
}
```

```
    }
```

Now, to benefit from a specialized box for this generic, you need to instantiate it:

```
val box = Box<Int>()
box.put(4)

val boolBox = Box<Boolean>()
boolBox.put(true)
boolBox.isEmpty()
```

Your box can handle any data type you want, and you'll be sure that whatever you put in it, has the same data type when you remove it from the box.

You can also apply generics at a function level, and there can be restrictions applied to the kind of data types the generic will accept. In Kotlin, there's a way to say "I want all functions to return this generic data type" or "I want only the input parameters to be this generic type".

There's a lot to learn about Generics, and you'll need to research it as you progress with your data structures and algorithms. But for now, you'll start with two of the most common generic data structures that are already provided by the Kotlin Standard Library.

The Kotlin Standard Library

With the Kotlin Standard Library you can get away without using any third-party libraries for most things. It contains useful classes and functions for text manipulation, math, streams, multithreading, annotations, collections and more.

There are many things to mention, but this book can't cover everything now, so keep your focus on the parts of the library that will help you with the algorithms.

Here are a few things to consider:

Package kotlin

This package contains many helpful higher-order functions. It also contains the definition of the most basic classes, exceptions and annotations. In this package, you'll find Any, Array, Int, ClassCastException and Deprecated to name a few. The most interesting things are the scoping functions defined in this package.



let

The `let` function helps you with null-checks and creates a new local scope to safely perform operations. Here's an example:

```
fun printCar(car: Car?) {
    val isCoupe = car?.let {
        (it.doors <= 2)
    }

    if (isCoupe == true) {
        println("Coupes are awesome")
    }
}
```

Inside `let`, `it` is the unwrapped value of `car`. Since you're using the safe-call operator `?.`, the code block won't run if `car` is `null`. That's how the compiler can unwrap it without complaining. As you might notice, `let` can return anything. In this case, it returns a Boolean telling you if the printed car was a coupé.

`let` uses the instance of the class as `this` inside the block, and the target object as `it`. This is helpful in a lot of situations. There are other functions that have a different approach.

run

`run` is similar to `let`, but it's more focused on the target object — the one you're using to call the function. Inside the block, `run` passes the target object as `this` and isolates the block from the outer scope.

```
fun printCar2(car: Car?) {
    val isCoupe = car?.run {
        (this.doors <= 2)
    }

    if (isCoupe == true) {
        println("Coupes are awesome")
    }
}
```

This is the same example, but now you're isolated inside `run`. The return value can still be anything.

These two functions are “transformational” functions. They’re called “transformational” because the object they return can be different from the object you call the function on. This is not the case with the following “mutating” functions.

also

If you try to replace `run` with `also`, you'll get compile errors. Unlike with `let` or `run` which return a transformation, the `also` function returns the original object.

Now, don't get tricked into thinking that `original` means that it's unmodified. It's just the same object. `also` uses `it` to refer to the object inside of the block and has access to the outer scope using `this`.

```
fun printCar3(car: Car?) {
    car?.also {
        it.doors = 4
    }.let {
        if (it?.doors != null && it.doors <= 2) {
            println("Coupes are awesome")
        }
    }
}
```

Since `also` returns the same `car` object, you can use it to mutate the object and then chain other calls to it. In this example, the check to see if the car is a coupe is within a `let` block, but since it was modified to have 4 doors within `also`, it won't print "Coupes are awesome"

apply

By now, you should be able to guess how `apply` works. It's an `also` that is isolated like a `run`. It returns the same object as the target, and it uses `this` inside the block:

```
fun printCar4(car: Car?) {
    car?.apply {
        doors = 4
    }.let {
        if (it?.doors != null && it.doors <= 2) {
            println("Coupes are awesome")
        }
    }
}
```

Again, the car has been updated to have 4 doors so it also won't print "Coupes are awesome"

These functions will come in handy from time to time, especially if you want to write clean and concise code.

There's one more function defined in **Standard.kt** that you'll see a lot. It's not the most useful one, but it's very common.

TODO

The JetBrains team decided to define `TODO` inside the Standard Kotlin Library to prevent one of the centuries-old habit of software developers: forgetting about **TODOs**.

Have a look at the definition of `TODO`:

```
public inline fun TODO(): Nothing = throw NotImplementedError()
```

`TODO()` throws an error when the code reaches one of these `TODOs`. This is a clever trick to prevent forgetting that you still have to write something. You'll see this every time IntelliJ generates a piece of code for you to implement. Just don't forget about it!

List

The second important package in the Kotlin Standard Library is **kotlin.collections**. You'll use it a lot in the following chapters and even more in real-life programming. For this introduction, you'll focus only on two basic collections, `List` and `Map`.

A `List` is a general-purpose, generic container for storing an ordered collection of elements; it's used commonly in many types of Kotlin programs.

You can create a `List` by using a helper function from the Kotlin Standard Library named `listOf()`. For example:

```
val places = listOf("Paris", "London", "Bucharest")
```

Note: Kotlin defines lists using interfaces. Each of these interface layers more capabilities on the list. For example, a `List` is an `Iterable`, which means that you can iterate through it at least once.

It's also a `Collection`, which means it can be traversed multiple times, non-destructively, and it can be accessed using a subscript operator `[]`.

For `List`, the positional access function `get()` guarantees access efficiency and it's the same as using the subscript operator.

Because Kotlin differentiates between mutable and immutable data structures, you'll want to create a `MutableList` to talk about all the operations lists have.



You'll learn more about mutability in Kotlin later in this chapter, but for now, just add another layer on top of your list, and create it like this:

```
val mutablePlaces = mutableListOf("Paris", "London",
    "Bucharest")
```

The Kotlin List is known as a **generic collection** because it can work with any type. In fact, most of the Kotlin standard library is built with generic code. Unlike the Java Collections that lose information about the type of collection, Kotlin's List is invariant. This means you cannot assign a `List<String>` to a `List<Any>`. Kotlin knows that these are different types of lists.

As with any data structure, there are certain notable traits of which you should be aware. The first of these is the notion of **order**.

Order

Elements in a list are explicitly **ordered**. Using the above `places` list as an example, Paris appears before London.

All of the elements in a list have a corresponding **zero-based**, integer index. For example, `places` from the above example has three indices, one corresponding to each element, starting with 0.

You can retrieve the value of an element in the list by writing the following:

```
places[0] // Paris
places[1] // London
places[2] // Bucharest
```

The order should not be taken for granted. Some data structures, such as Map, have a weaker concept of the order or no order at all. You can end up with a different order when you try to access elements out of different collections.

Random-access

Random-access is a trait that data structures can claim if they can handle element retrieval in a constant amount of time.

For example, getting "London" from `places` takes constant time. This means that there's no performance difference in accessing the first element, the 3rd element or any other element of the list. Again, this performance should not be taken for granted. Other data structures such as Linked Lists and Trees do not have constant time access.



For linked lists, the further the element is, the longer it takes to access it. You'll learn more about the complexity of the operations in the next chapter.

List performance

Aside from being a random-access collection, there are other areas of performance that are of interest on how well or poorly does the data structure fare when the amount of data it contains needs to grow. For lists, this varies on two factors.

Insertion location

The first factor is one in which you choose to insert the new element inside the list. The most efficient scenario for adding an element to a list is to append it at the end of the list:

```
mutablePlaces.add("Budapest")
println(mutablePlaces) // prints [Paris, London, Bucharest,
Budapest]
```

Inserting "Budapest" using `add()` places the string at the end of the list. This is a **constant-time** operation, meaning the time it takes to perform this operation stays the same no matter how large the list becomes.

However, there may come a time that you need to insert an element in a particular location, such as in the middle of the list. To help illustrate, consider the following analogy. You're standing in line for the movies. Someone new comes along to join the lineup. If they just go to the end of the line, nobody will even notice the newcomer. But, if the newcomer tried to insert himself into the middle of the line, he would have to convince half the lineup to shuffle back to make room. And if he were *terribly* rude, he may try to insert himself at the head of the line. This is the worst-case scenario because every single person in the lineup would need to shuffle back to make room for this new person in front!

This is exactly how lists work. Inserting new elements from anywhere aside from the end will force elements to shift back to make room for the new element:

```
mutablePlaces.add(0, "Kiev")
// [Kiev, Paris, London, Bucharest, Budapest]
```

To be precise, every element must shift back by one index. If we consider the number of items in the list to be n , this would take n steps. The time for this operation grows as the number of elements in the list grows. If the number of elements in the list doubles, the time required for this add operation will also double.

If inserting elements in front of a collection is a common operation for your program, you may want to consider a different data structure to hold your data.

Capacity

The second factor that determines the speed of insertion is the list's **capacity**.

Underneath the hood, Kotlin lists are allocated with a predetermined amount of space for its elements. If you try to add new elements to a list that is already at maximum capacity, the List must restructure itself to make more room for more elements.

This is done by copying all the current elements of the list in a new and bigger container in memory. However, this comes at a cost. Each element of the list has to be accessed and copied. This means that any insertion, even at the end, could take n steps to complete if a copy is made.

Note: Standard Library employs a strategy that minimizes the times this copying needs to occur. Each time it runs out of storage and needs to copy, it doubles the capacity.

Map

A Map is another generic collection that holds **key-value** pairs. For example, here's a map containing a user's name and a score:

```
val scores = mutableMapOf("Eric" to 9, "Mark" to 12, "Wayne" to 1)
```

There's no restriction on what type of object the Key is, but you should know that a Map uses the `hashCode()` function to store the data. Usually, the Key is one of the standard library data types which have the `hashCode()` function implemented. But if you want to use your own data type, you need to implement the function yourself.

It's not difficult to override `hashCode()`, you just have to investigate a little bit what are the most common strategies to get the best result out of the Map.

You can add a new entry to the map with the following syntax:

```
scores["Andrew"] = 0
```

This creates a new key-value pair in the map:

```
{Eric=9, Mark=12, Wayne=1, Andrew=0}
```

Maps are unordered, so you can't guarantee where new entries will be put. This is because maps put data into different *buckets*, depending on the result that the `hashCode()` function returns. The data in each bucket is ordered, but the general order of the data in the map is unpredictable.

It is possible to traverse through the key-values of a map multiple times as the Collection protocol affords. This order, while not defined, will be the same until the collection is changed.

The lack of explicit ordering disadvantage comes with some redeeming traits.

Unlike the list, maps don't need to worry about elements shifting around. Inserting into a map always takes a constant amount of time.

Note: Lookup operations also take a constant amount of time, which is significantly faster than finding a particular element in a list which requires a walk from the beginning of the list to the insertion point.

Mutable vs. immutable

As you've seen throughout the chapter, there's a distinction between mutable and immutable data structures in Kotlin.

When referring to the concept of a `List`, it's usually referring to the Kotlin's `MutableList`. Unlike `List`, `MutableList` also has functions for adding and removing elements. Kotlin doesn't allow a `List` to be changed in any way.

To change a data structure, you must express this intent by using the `Mutable` version of that data structure. These data structures have functions for adding and removing elements.

So why would you ever use the immutable version? **For safety.**

Whenever you need to pass your data structure as a parameter, and you want to be sure that the function doesn't produce a side effect, you should use an immutable collection as the parameter.

Consider this code:

```
fun noSideEffectList(names: List<String>) {
    println(names)
}

fun sideEffectList(names: MutableList<String>) {
    names.add("Joker")
}

fun mutableVsImmutable() {
    val people = mutableListOf("Brian", "Stanley", "Ringo")
    noSideEffectList(people) // [Brian, Stanley, Ringo]
    sideEffectList(people) // Adds a Joker to the list
    noSideEffectList(people) // [Brian, Stanley, Ringo, Joker]
}
```

The `sideEffectList` function adds a Joker to it. These kind of side-effects are usually the ones generating bugs. Avoiding them by using a `List` instead of a `MutableList` is preferred.

Key points

- Every data structure has advantages and disadvantages. Knowing them is key to writing performant software.
- Functions such as `add(Int, Any)` for `List` have performance characteristics that can cripple performance when used haphazardly. If you find yourself needing to use `add(Int, Any)` frequently with indices near the beginning of the list, you may want to consider using a different data structure such as the `Linked List`.
- `Map` trades the ability to maintain the order of its elements for fast insertion and searching.

Chapter 2: Complexity

By Matei řuică

How well will it scale?

This question is always asked sooner or later in the software development cycle and comes in several flavors.

From an architectural perspective, scalability refers to how flexible your app is as your features are increasing. From a database perspective, scalability is about the capability of a database to handle an increasing amount of data and users. For a web server, being scalable can mean that it can serve a high number of users accessing it at the same time. Regardless of what the question actually means, you need to study it and come up with a response as soon as possible. This way, you can avoid big problems down the line.

For algorithms, scalability refers to how the algorithm performs in terms of execution time and memory usage as the input size increases. With a small amount of data, any algorithm may still feel fast. However, as the amount of data increases, an expensive algorithm can become crippling.

So how bad can it get? Estimating this is an important skill for you to know.

In this chapter, you'll learn about the Big O notation for the different levels of scalability in two dimensions:

- Execution time.
- Memory usage.



Time complexity

With small amounts of data, even the most expensive algorithm can seem fast due to the speed of modern hardware. However, as data increases, the cost of an expensive algorithm becomes increasingly apparent.

Time complexity is a measure of the time required to run an algorithm as the input size increases. In this section, you'll go through the most common time complexities and learn how to identify them.

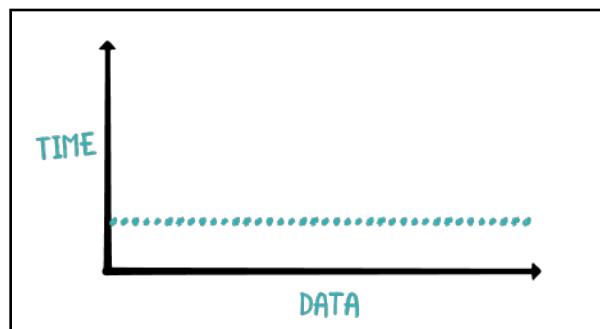
Constant time

A constant time algorithm is one that has the same running time regardless of the size of the input. Consider the following:

```
fun checkFirst(names: List<String>) {
    if (names.firstOrNull() != null) {
        println(names.first())
    } else {
        println("no names")
    }
}
```

The size of `names` does not affect the running time of this function. Whether `names` has 10 items or 10 million items, this function only checks the first element of the list.

Here's a visualization of this time complexity in a plot between time versus data size:



Constant time

As input data increases, the amount of time the algorithm takes does not change.

For brevity, programmers use a notation known as **Big O notation** to represent various magnitudes of time complexity. The Big O notation for constant time is $O(1)$. It's one unit of time, regardless of the input. This time doesn't need to be small, though. The algorithm can still be slow, but it's equally slow all of the time. :]

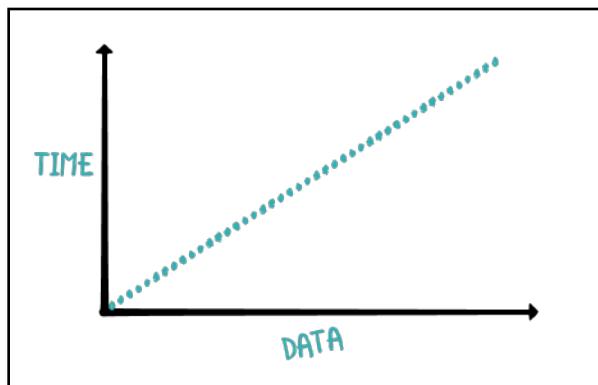
Linear Time

Consider the following snippet of code:

```
fun printNames(names: List<String>) {  
    for (name in names) {  
        println(name)  
    }  
}
```

This function prints all the names in a `String` list. As the input list increases in size, the number of iterations is increased by the same amount.

This behavior is known as **linear time** complexity:



Linear time

Linear time complexity is usually the easiest to understand. As the amount of data increases, the running time increases by the same amount. That's why you have the straight linear graph illustrated above. The Big O notation for linear time is $O(n)$.

Note: What about a function that has two loops over all of the data and a calls six different $O(1)$ methods? Is it $O(2n + 6)$?

Time complexity only gives a high-level shape of the performance. Loops that happen a set number of times are not part of the calculation. You'll need to abstract everything and consider only the most important thing that affects performance. All constants are dropped in the final Big O notation. In other words, $O(2n + 6)$ is surprisingly equal to $O(n)$.

Quadratic time

More commonly referred to as **n squared**, this time complexity refers to an algorithm that takes time proportional to the square of the input size.

Consider the following code:

```
fun multiplicationBoard(size: Int) {  
    for (number in 1..size) {  
        print(" | ")  
        for (otherNumber in 1..size) {  
            print("$number x $otherNumber = ${number * otherNumber} | "  
        )  
    }  
    println()  
}
```

If you call this function using a small number, like 2, you'll get the following output:

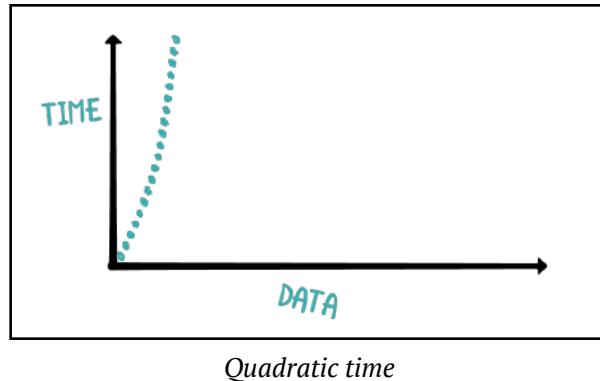
	1 x 1 = 1		1 x 2 = 2	
	2 x 1 = 2		2 x 2 = 4	

This time, the function prints all of the products of the numbers that are less than or equal to the input, starting with 1.

If the input is 10, it'll print the full multiplication board of 10×10 . That's 100 print statements. If you increase the input size by one, it'll print the product of 11 numbers with 11 numbers, resulting in 121 print statements.

Unlike the previous function, which operates in linear time, the n squared algorithm can quickly run out of control as the data size increases. Imagine printing the results for `multiplicationBoard(100_000)`!

Here's a graph illustrating this behavior:



Quadratic time

As the size of the input data increases, the amount of time it takes for the algorithm to run increases drastically. Thus, n squared algorithms don't perform well at scale.

The Big O notation for quadratic time is $O(n^2)$.

Note: No matter how inefficiently a linear time $O(n)$ algorithm is written, for a sufficiently large n , the linear time algorithm will always execute faster than a super optimized quadratic algorithm.

Although not a central concern of this book, optimizing for absolute efficiency can be crucial.

Companies put millions of dollars of R&D into reducing the slope of those constants that Big O notation ignores. For example, a GPU optimized version of an algorithm might run 100 \times faster than the naive CPU version while remaining $O(n)$.

Logarithmic time

So far, you've learned about the linear and quadratic time complexities wherein each element of the input is inspected at least once. However, there are scenarios in which only a subset of the input needs to be inspected, leading to a faster runtime.

Algorithms that belong to this category of time complexity are ones that can leverage some shortcuts by making some assumptions about the input data. For instance, if you had a **sorted** list of integers, what is the quickest way to find if a particular value exists?

A possible solution would be to inspect the array from start to finish to check every element before reaching a conclusion. Since you're inspecting each of the elements once, that would be a $O(n)$ algorithm.

Linear time is fairly good, but you can do better. Since the input array is sorted, there's an optimization that you can make. Consider the following code:

```
val numbers = listOf(1, 3, 56, 66, 68, 80, 99, 105, 450)

fun linearContains(value: Int, numbers: List<Int>): Boolean {
    for (element in numbers) {
        if (element == value) {
            return true
        }
    }
    return false
}
```

If you were checking if the number 451 existed in the list, this algorithm would have to iterate from the beginning to end, making a total of nine inspections for the nine values in the list. However, since the list is sorted, you can, right off the bat, drop half of the comparisons necessary by checking the middle value:

```
fun pseudoBinaryContains(value: Int, numbers: List<Int>):
Boolean {
    if (numbers.isEmpty()) return false

    val middleIndex = numbers.size / 2

    if (value <= numbers[middleIndex]) {
        for (index in 0..middleIndex) {
            if (numbers[index] == value) {
                return true
            }
        }
    } else {
        for (index in middleIndex until numbers.size) {
            if (numbers[index] == value) {
                return true
            }
        }
    }
    return false
}
```

The above function makes a small but meaningful optimization wherein it only checks half of the list to come up with a conclusion.

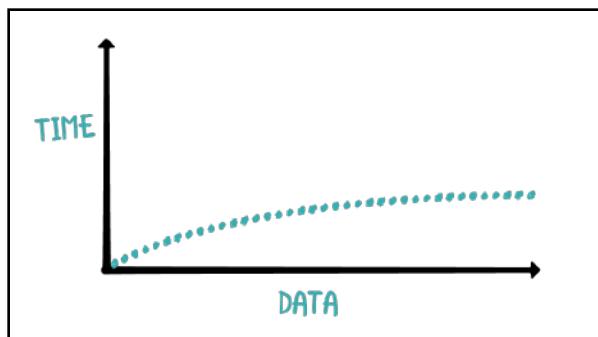


The algorithm first checks the middle value to see how it compares with the desired value. If the middle value is bigger than the desired value, the algorithm won't bother looking at the values on the right half of the list; since the list is sorted, values to the right of the middle value can only get bigger.

In the other case, if the middle value is smaller than the desired value, the algorithm won't look at the left side of the list. This optimization cuts the number of comparisons by half.

What if you could do this optimization repeatedly throughout this method? You'll find out in Chapter 11, "Binary Search".

An algorithm that can repeatedly drop half of the required comparisons will have logarithmic time complexity. Here's a graph illustrating how a logarithmic time algorithm would behave as input data increases:



Logarithmic time

As input data increases, the time it takes to execute the algorithm increases at a slower rate. If you look closely, you may notice that the graph seems to exhibit asymptotic behavior. This can be explained by considering the impact of halving the number of comparisons you need to do.

When you have an input size of 100, halving the comparisons means you save 50 comparisons. If the input size was 10,000, halving the comparisons means you save 5,000 comparisons.

You'll need just a few more halvings, and your input data will be around 50 again. The more data you have, the more the halving effect scales.

Algorithms in this category are few but are extremely powerful in situations that allow for it. The Big O notation for logarithmic time complexity is $O(\log n)$.

Note: Is it log base 2, log base 10, or the natural log?

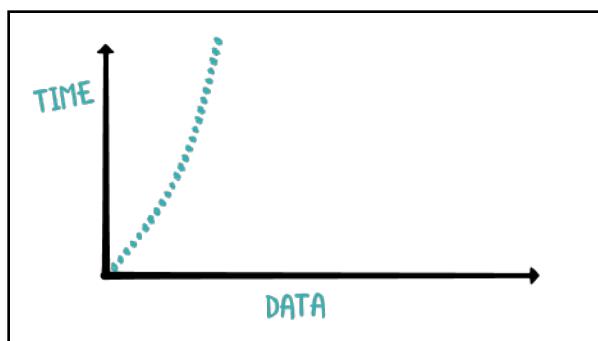
In the above example, log base 2 applies. However, since Big O notation only concerns itself with the shape of the performance, the actual base doesn't matter. The more input data you can drop after each pass, the faster the algorithm will be.

Quasilinear time

Another common time complexity you'll encounter is **quasilinear time**. Algorithms in this category perform worse than linear time but dramatically better than quadratic time. They are among the most common algorithms you'll deal with.

An example of a quasilinear time algorithm is Kotlin's `sort` method.

The Big-O notation for quasilinear time complexity is $O(n \log n)$ which is a multiplication of linear and logarithmic time. So quasilinear fits between logarithmic and linear time. It's a magnitude worse than linear time but still better than many of the other complexities that you'll see next. Here's the graph:



Quasilinear time

The quasilinear time complexity shares a similar curve with quadratic time. The key difference is that quasilinear complexity is more resilient to large data sets.

Other time complexities

The five complexities you've encountered are the ones that you'll encounter in this book. Other time complexities do exist, but are far less common and tackle more complex problems that are not discussed in this book. These time complexities include **polynomial time**, **exponential time**, **factorial time** and more.

It's important to note that time complexity is a high-level overview of performance, and it doesn't judge the speed of the algorithm beyond the general ranking scheme. This means that two algorithms can have the same time complexity, but one may still be much faster than the other. For small data sets, time complexity may not be an accurate measure of actual speed.

For instance, quadratic algorithms such as insertion sort can be faster than quasilinear algorithms, such as mergesort, if the data set is small. This is because the insertion sort does not need to allocate extra memory to perform the algorithm, while mergesort needs to allocate multiple arrays.

Note: For small data sets, the memory allocation can be expensive relative to the number of elements the algorithm needs to touch.

Comparing time complexity

Suppose you wrote the following code that finds the sum of numbers from 1 to n .

```
fun sumFromOne(n: Int): Int {  
    var result = 0  
    for (i in 1..n) {  
        result += i  
    }  
    return result  
}
```

If you try to call the function with `sumFromOne(10000)`, the code loops 10,000 times and returns `50005000`. It's $O(n)$ and will take a moment to run as it counts through the loop and prints results.

This can also be written using `reduce`:

```
fun sumFromOne(n: Int): Int {  
    return (1..n).reduce { sum, element -> sum + element }  
}
```

The time complexity of the version that uses `reduce` is also $O(n)$ since it essentially performs the same logic. It continuously adds each element to the sum and returns the total sum.

Finally, you can write:

```
fun sumFromOne(n: Int): Int {  
    return n * (n + 1) / 2  
}
```

This version of the function uses a trick that a famous mathematician, **Fredrick Gauss**, noticed while he was still in elementary school. The sum of a series of numbers starting from 1 up to n can be computed using simple arithmetic. This final version of the algorithm is $O(1)$ and tough to beat. A constant time algorithm is always preferred over a linear or logarithmic time algorithm since the time it takes to run will not change regardless of how large n gets.

Space complexity

The time complexity of an algorithm isn't the only performance metric against which algorithms are ranked. Another important metric is its space complexity, which is a measure of the amount of memory it uses.

Consider the following code:

```
fun printSorted(numbers: List<Int>) {  
    val sorted = numbers.sorted()  
    for (element in sorted) {  
        println(element)  
    }  
}
```

The above function creates a sorted copy of the list and prints it. To calculate the space complexity, you analyze the amount of memory the function allocates.

Since `numbers.sorted()` produces a new list with the same size of `numbers`, the space complexity of `printSorted` is $O(n)$. While this function is simple and elegant, there may be some situations in which you want to allocate as little memory as possible.

You could rewrite the above function like this:

```
fun printSorted(numbers: List<Int>) {
    // 1
    if (numbers.isEmpty()) return

    // 2
    var currentCount = 0
    var minValue = Int.MIN_VALUE

    // 3
    for (value in numbers) {
        if (value == minValue) {
            println(value)
            currentCount += 1
        }
    }

    while (currentCount < numbers.size) {
        // 4
        var currentValue = numbers.max()!!

        for (value in numbers) {
            if (value < currentValue && value > minValue) {
                currentValue = value
            }
        }

        // 5
        for (value in numbers) {
            if (value == currentValue) {
                println(value)
                currentCount += 1
            }
        }

        // 6
        minValue = currentValue
    }
}
```

Woah, that's a lot of code for something you've previously done in a couple of lines! But this implementation respects space constraints.

The overall goal is to iterate through the array multiple times, printing the next smallest value for each iteration.

Here's what this algorithm is doing:

1. Check for the case if the list is empty. If it is, there's nothing to print.
2. `currentCount` keeps track of the number of print statements made. `minValue` stores the last printed value.
3. The algorithm begins by printing all values matching the `minValue` and updates the `currentCount` according to the number of print statements made.
4. Using the `while` loop, the algorithm finds the lowest value bigger than `minValue` and stores it in `currentValue`.
5. The algorithm then prints all values of `currentValue` inside the array while updating `currentCount`.
6. `minValue` is set to `currentValue`, so the next iteration will try to find the next minimum value.

The above algorithm only allocates memory for a few variables. Since the amount of memory allocated is constant and does not depend on the size of the list, the space complexity is $O(1)$.

This is in contrast with the previous function, which allocates an entire list to create the sorted representation of the source array. The tradeoff here is that you sacrifice time and code readability to use as little memory is possible.

Note: While today's devices have a lot of space to store information, there was a time when everything an algorithm needed to store was bound to just a couple of KB.

Algorithms could be designed to take a longer time to finish but the memory constraint was a physical one, and it could not be broken.

Nowadays, the available memory is huge but so is the data you are handling, so algorithms still need to take space complexity into account.

Key points

- **Time complexity** is a measure of the time required to run an algorithm as the input size increases.
- **Space complexity** is a measure of the resources required for the algorithm to manipulate the input data.
- **Big O** notation is used to represent the general form of time and space complexity.
- Time and space complexity are high-level measures of scalability. They don't measure the actual speed of the algorithm itself.
- For small data sets, time complexity is usually irrelevant. A quasilinear algorithm can be slower than a linear algorithm.

Section II: Elementary Data Structures

This section looks at a few important data structures that form the basis of more advanced algorithms covered in future sections.

- **Chapter 3: Linked List:** A linked list is a collection of values arranged in a linear, unidirectional sequence. A linked list has several theoretical advantages over contiguous storage options such as the array, including constant time insertion and removal from the front of the list, and other reliable performance characteristics.
- **Chapter 4: Stack Data Structures:** The stack data structure is identical in concept to a physical stack of objects. When you add an item to a stack, you place it on top of the stack. When you remove an item from a stack, you always remove the topmost item. Stacks are useful, and also exceedingly simple. The main goal of building a stack is to enforce how you access your data.
- **Chapter 5: Queues:** Lines are everywhere, whether you're lining up to buy tickets to your favorite movie or waiting for a printer machine to print out your documents. These real-life scenarios mimic the queue data structure. Queues use first in, first out ordering. In other words, the first element that was enqueued will be the first to get dequeued. Queues are handy when you need to maintain the order of your elements to process later.

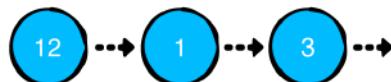
Continuing to study the fundamentals will strengthen your base knowledge.

Chapter 3: Linked List

By Matei řuică

A linked list is a collection of values arranged in a linear, unidirectional sequence. A linked list has several theoretical advantages over contiguous storage options such as the Kotlin Array or ArrayList:

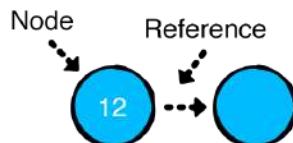
- Constant time insertion and removal from the front of the list.
- Reliable performance characteristics.



A linked list

As the diagram suggests, a linked list is a chain of **nodes**. Nodes have two responsibilities:

1. Hold a value.
2. Hold a reference to the next node. The absence of a reference to the next node, null, marks the end of the list.



A node holding the value 12

In this chapter, you'll implement a linked list and learn about the common operations associated with it. You'll also learn about the time complexity of each operation. Open the starter project for this chapter so that you can dive right into the code.

Node

Create a new Kotlin file in **src** and name it **Node.kt**. Add the following to the file:

```
data class Node<T>(var value: T, var next: Node<T>? = null) {
    override fun toString(): String {
        return if (next != null) {
            "$value -> ${next.toString()}"
        } else {
            "$value"
        }
    }
}
```

Navigate to the **Main.kt** file and add the following inside **main()**:

```
fun main() {
    "creating and linking nodes" example {
        val node1 = Node(value = 1)
        val node2 = Node(value = 2)
        val node3 = Node(value = 3)

        node1.next = node2
        node2.next = node3

        println(node1)
    }
}
```

You've just created three nodes and connected them:



A linked list containing values 1, 2, and 3

Once you run **Main.kt**, you'll see the following output in the console:

```
---Example of creating and linking nodes---
1 -> 2 -> 3
```

As far as practicality goes, this method of building lists is far from ideal. You can easily see that building long lists in this way is impractical. A common way to alleviate this problem is to build a `LinkedList` that manages the `Node` objects. You'll do just that!

LinkedList

In `src`, create a new file and name it `LinkedList.kt`. Add the following to the file:

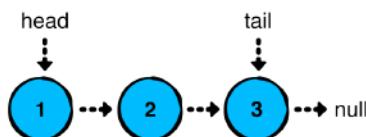
```
class LinkedList<T> {

    private var head: Node<T>? = null
    private var tail: Node<T>? = null
    private var size = 0

    fun isEmpty(): Boolean {
        return size == 0
    }

    override fun toString(): String {
        if (isEmpty()) {
            return "Empty list"
        } else {
            return head.toString()
        }
    }
}
```

A linked list has the concept of a **head** and **tail**, which refers to the first and last nodes of the list respectively:



The head and tail of the list

You'll also keep track of the size of the linked list in the `size` property. This might not seem useful yet, but it will come in handy later.

Adding values to the list

Next, you're going to provide an interface to manage the `Node` objects. You'll first take care of adding values. There are three ways to add values to a linked list, each having their own unique performance characteristics:

1. `push`: Adds a value at the front of the list.
2. `append`: Adds a value at the end of the list.
3. `insert`: Adds a value after a particular node of the list.

You'll implement each of these in turn and analyze their performance characteristics.

push operations

Adding a value at the front of the list is known as a `push` operation. This is also known as **head-first insertion**. The code for it is deliciously simple.

Add the following method to `LinkedList`:

```
fun push(value: T) {  
    head = Node(value = value, next = head)  
    if (tail == null) {  
        tail = head  
    }  
    size++  
}
```

In the case in which you're pushing into an empty list, the new node is both the `head` and `tail` of the list. Since the list now has a new node, you increment the value of `size`.

In `Main.kt`, add the following in `main()`:

```
"push" example {  
    val list = LinkedList<Int>()  
    list.push(3)  
    list.push(2)  
    list.push(1)  
  
    println(list)  
}
```

Your console output will show this:

```
---Example of push---
1 -> 2 -> 3
```

This is pretty cool, but you can do even better. You'll use the **fluent interface** pattern to chain multiple push calls. Go back to `push()` and add `LinkedList<T>` as its return type. Then, add a `return this` line at the end to return the list that you've just pushed an element into.

The method will now look like this:

```
fun push(value: T): LinkedList<T> {
    head = Node(value = value, next = head)
    if (tail == null) {
        tail = head
    }
    size++
    return this
}
```

In `main()`, you can now rewrite the previous example, making use of `push()`'s return value:

```
"fluent interface push" example {
    val list = LinkedList<Int>()
    list.push(3).push(2).push(1)
    println(list)
}
```

That's more like it! Now that you can add multiple elements to the start of the list with ease.

append operations

The next operation you'll look at is `append`. This adds a value at the end of the list, which is known as **tail-end insertion**.

In `LinkedList.kt`, add the following code just below `push()`:

```
fun append(value: T) {
    // 1
    if (isEmpty()) {
        push(value)
        return
    }
    // 2
```

```
tail?.next = Node(value = value)  
  
    // 3  
    tail = tail?.next  
    size++  
}
```

This code is relatively straightforward:

1. Like before, if the list is empty, you'll need to update both `head` and `tail` to the new node. Since `append` on an empty list is functionally identical to `push`, you invoke `push` to do the work for you.
2. In all other cases, you create a new node *after* the current `tail` node. `tail` will never be `null` here because you've already handled the case where the list is empty in the `if` statement.
3. Since this is tail-end insertion, your new node is also the tail of the list.

Go back to **Main.kt** and write the following at the bottom of `main()`:

```
"append" example {  
    val list = LinkedList<Int>()  
    list.append(1)  
    list.append(2)  
    list.append(3)  
  
    println(list)  
}
```

You'll see the following output in the console:

```
---Example of append---  
1 -> 2 -> 3
```

You can use the trick you learned for `push()` to get a fluid interface here too. It's up to you if you've liked it or not but imagine how you could chain pushes and appends in a world of endless possibilities. Or just have some fun with it. :]

insert operations

The third and final operation for adding values is `insert(afterNode: Node<T>)`. This operation inserts a value at a particular place in the list and requires two steps:

1. Finding a particular node in the list.
2. Inserting the new node after that node.

First, you'll implement the code to find the node where you want to insert your value.

In `LinkedList.kt`, add the following code just below `append`:

```
fun nodeAt(index: Int): Node<T>? {  
    // 1  
    var currentNode = head  
    var currentIndex = 0  
  
    // 2  
    while (currentNode != null && currentIndex < index) {  
        currentNode = currentNode.next  
        currentIndex++  
    }  
  
    return currentNode  
}
```

`nodeAt()` tries to retrieve a node in the list based on the given index. Since you can only access the nodes of the list from the head node, you'll have to make iterative traversals. Here's the play-by-play:

1. You create a new reference to `head` and keep track of the current number of traversals.
2. Using a `while` loop, you move the reference down the list until you reach the desired index. Empty lists or out-of-bounds indexes will result in a `null` return value.

Now, you need to insert the new node.

Add the following method just below `nodeAt()`:

```
fun insert(value: T, afterNode: Node<T>): Node<T> {
    // 1
    if (tail == afterNode) {
        append(value)
        return tail!!
    }
    // 2
    val newNode = Node(value = value, next = afterNode.next)
    // 3
    afterNode.next = newNode
    size++
    return newNode
}
```

Here's what you've done:

1. In the case where this method is called with the `tail` node, you call the functionally equivalent `append` method. This takes care of updating `tail`.
2. Otherwise, you create a new node and link its `next` property to the next node of the list.
3. You reassign the `next` value of the specified node to link it to the new node that you just created.

To test things, go back to `Main.kt` and add the following to the bottom of `main()`:

```
"inserting at a particular index" example {
    val list = LinkedList<Int>()
    list.push(3)
    list.push(2)
    list.push(1)

    println("Before inserting: $list")
    var middleNode = list.nodeAt(1)!!
    for (i in 1..3) {
        middleNode = list.insert(-1 * i, middleNode)
    }
    println("After inserting: $list")
}
```

You'll see the following output:

```
---Example of inserting at a particular index---
Before inserting: 1 -> 2 -> 3
After inserting: 1 -> 2 -> -1 -> -2 -> -3 -> 3
```

Performance analysis

Whew! You made good progress so far. To recap, you've implemented the three operations that add values to a linked list and a method to find a node at a particular index.

	push	append	insert	nodeAt
Behaviour	insert at head	insert at tail	insert after a node	returns a node at given index
Time complexity	O(1)	O(1)	O(1)	O(i), where i is the given index

Next, you'll focus on the opposite action: removal operations.

Removing values from the list

There are three primary operations for removing nodes:

1. **pop**: Removes the value at the front of the list.
2. **removeLast**: Removes the value at the end of the list.
3. **removeAfter**: Removes a value anywhere in the list.

You'll implement all three and analyze their performance characteristics.

pop operations

Removing a value at the front of the list is often referred to as **pop**. This operation is almost as simple as `push()`, so dive right in.

Add the following method to `LinkedList`:

```
fun pop(): T? {
    if (!isEmpty()) size--
    val result = head?.value
    head = head?.next
}
```

```
if (isEmpty()) {  
    tail = null  
}  
  
return result  
}
```

`pop()` returns the value that was removed from the list. This value is optional since it's possible that the list is empty.

By moving the head down a node, you've effectively removed the first node of the list. The garbage collector will remove the old node from memory once the method finishes since there will be no more references attached to it. If the list becomes empty, you set `tail` to `null` as well.

To test, go to **Main.kt** and add the following code at the bottom inside `main()`:

```
"pop" example {  
    val list = LinkedList<Int>()  
    list.push(3)  
    list.push(2)  
    list.push(1)  
  
    println("Before popping list: $list")  
    val poppedValue = list.pop()  
    println("After popping list: $list")  
    println("Popped value: $poppedValue")  
}
```

You'll see the following output:

```
---Example of pop---  
Before popping list: 1 -> 2 -> 3  
After popping list: 2 -> 3  
Popped value: 1
```

removeLast operations

Removing the last node of the list is somewhat inconvenient.

Although you have a reference to the `tail` node, you can't chop it off without having a reference to the node before it. Thus, you need to traverse the whole list to find the node before the last.

Add the following code just below `pop()`:

```
fun removeLast(): T? {
    // 1
    val head = head ?: return null
    // 2
    if (head.next == null) return pop()
    // 3
    size--

    // 4
    var prev = head
    var current = head

    var next = current.next
    while (next != null) {
        prev = current
        current = next
        next = current.next
    }
    // 5
    prev.next = null
    tail = prev
    return current.value
}
```

Here's what's happening:

1. If `head` is `null`, there's nothing to remove, so you return `null`.
2. If the list only consists of one node, `removeLast` is functionally equivalent to `pop`. Since `pop` will handle updating the `head` and `tail` references, you can delegate this work to the `pop` function.
3. At this point, you know that you'll be removing a node, so you update the size of the list accordingly.
4. You keep searching for the next node until `current.next` is `null`. This signifies that `current` is the last node of the list.
5. Since `current` is the last node, you disconnect it using the `prev.next` reference. You also make sure to update the `tail` reference.

Go back to `Main.kt`, and in `main()`, add the following to the bottom:

```
"removing the last node" example {  
    val list = LinkedList<Int>()  
    list.push(3)  
    list.push(2)  
    list.push(1)  
  
    println("Before removing last node: $list")  
    val removedValue = list.removeLast()  
  
    println("After removing last node: $list")  
    println("Removed value: $removedValue")  
}
```

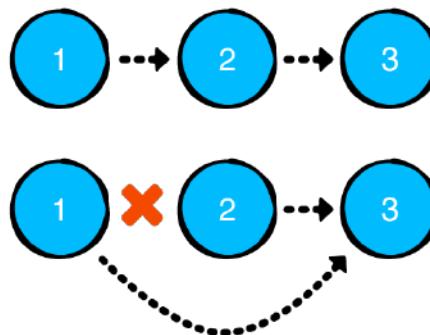
You'll see the following at the bottom of the console:

```
---Example of removing the last node---  
Before removing last node: 1 -> 2 -> 3  
After removing last node: 1 -> 2  
Removed value: 3
```

`removeLast()` requires you to traverse down the list. This makes for an $O(n)$ operation, which is relatively expensive.

Remove operations

The final remove operation is removing a node at a particular point in the list. This is achieved much like `insert()`. You'll first find the node immediately before the node you wish to remove and then unlink it.



Removing the middle node

Navigate back to **LinkedList.kt** and add the following method below `removeLast()`:

```
fun removeAfter(node: Node<T>): T? {
    val result = node.next?.value

    if (node.next == tail) {
        tail = node
    }

    if (node.next != null) {
        size--
    }

    node.next = node.next?.next
    return result
}
```

Special care needs to be taken if the removed node is the tail node since the `tail` reference will need to be updated.

Now, add the following example to `main()` to test `removeAfter()`. You know the drill:

```
"removing a node after a particular node" example {
    val list = LinkedList<Int>()
    list.push(3)
    list.push(2)
    list.push(1)

    println("Before removing at particular index: $list")
    val index = 1
    val node = list.nodeAt(index - 1)!!
    val removedValue = list.removeAfter(node)

    println("After removing at index $index: $list")
    println("Removed value: $removedValue")
}
```

You'll see the following output in the console:

```
---Example of removing a node after a particular node---
Before removing at particular index: 1 -> 2 -> 3
After removing at index 1: 1 -> 3
Removed value: 2
```

Try adding more elements and play around with the value of the index. Similar to `insert()`, the time complexity of this operation is $O(1)$, but it requires you to have a reference to a particular node beforehand.

Performance analysis

You've hit another checkpoint. To recap, you've implemented the three operations that remove values from a linked list:

	<code>pop</code>	<code>removeLast</code>	<code>removeAfter</code>
Behaviour	remove at head	remove at tail	remove the immediate next node
Time complexity	$O(1)$	$O(n)$	$O(1)$

At this point, you've defined an interface for a linked list that most programmers around the world can relate to. However, there's work to be done to adorn the Kotlin semantics. In the next half of the chapter, you'll focus on making the interface better by bringing it closer to idiomatic Kotlin.

Kotlin collection interfaces

The Kotlin Standard Library has a set of interfaces that help define what's expected of a particular type. Each of these interfaces provides certain guarantees on characteristics and performance. Of these set of interfaces, four are referred to as *collection interfaces*.

Here's a small sample of what each interface represents:

- **Tier 1, Iterable:** An iterable type provides sequential access to its elements via an `Iterator`.
- **Tier 2, Collection:** A collection type is an iterable type that provides additional functionality, allowing you to check if the collection contains a particular element or a collection of elements.
- **Tier 3, MutableIterable:** Provides a `MutableIterator`, which allows items to be removed from the given collection.

- **Tier 4, MutableCollection:** Unlike a simple Collection, a MutableCollection interface provides methods to alter the collection. For example, you can **add** and **remove** elements, and even **clear** the entire collection.

There's more to be said for each of these. You'll learn more about each of them when you need to conform to them.

A linked list can get to the tier 4 of the collection interfaces. Since a linked list is a chain of nodes, adopting the Iterable interface makes sense. And because you've already implemented adding elements and removing them, it's pretty clear we can go all the way to the MutableCollection interface.

Becoming a Kotlin mutable collection

In this section, you'll look into implementing the MutableCollection interface. A mutable collection type is a finite sequence and provides nondestructive sequential access but also allows you to modify the collection.

Iterating through elements

Reaching Tier 1 means implementing Iterable in the `LinkedList`. To make things easier, first make reading the `size` available outside the class. Modify the `size` property in `LinkedList.kt`, so that the property itself is public, but its setter remains private:

```
var size = 0
    private set
```

Then, add the Iterable interface to the class definition. The definition will now look like:

```
class LinkedList<T> : Iterable<T> {
    ...
}
```

This means that you're now required to add the following method to fulfill the Iterable interface:

```
override fun iterator(): Iterator<T> {
    return LinkedListIterator(this)
}
```

Right now, the compiler complains because it doesn't know what a `LinkedListIterator` is. Create a class named `LinkedListIterator` and make it implement the `Iterator` interface:

```
class LinkedListIterator<T> : Iterator<T> {  
    override fun next(): T {  
        TODO("not implemented")  
    }  
    override fun hasNext(): Boolean {  
        TODO("not implemented")  
    }  
}
```

To iterate through the linked list, you need to have a reference to the list. Add this parameter to the constructor:

```
class LinkedListIterator<T>(  
    private val list: LinkedList<T>  
) : Iterator<T> {  
    ...  
}
```

You can now start implementing the required methods, starting with the easier one, `hasNext()`. This method indicates whether your `Iterable` still has values to read. You'll need to keep track of the position that the iterator has in the collection, so create an `index` field:

```
private var index = 0
```

Then, you can easily check if the position you're at is less than the total number of nodes:

```
override fun hasNext(): Boolean {  
    return index < list.size  
}
```

For `next()`, the one that reads the values of the nodes, you can use another property to help you out. You'll want to keep track of the last node, so you can easily find the next one:

```
private var lastNode: Node<T>? = null
```

The `next()` function looks like:

```
override fun next(): T {  
    // 1
```

```

if (index >= list.size) throw IndexOutOfBoundsException()
// 2
lastNode = if (index == 0) {
    list.nodeAt(0)
} else {
    lastNode?.next
}
// 3
index++
return lastNode!! .value
}

```

Here are the crucial bits of this function, step-by-step:

1. You check that there are still elements to return. If there aren't, you throw an exception. This should never be the case if clients use the `Iterator` correctly, always checking with `hasNext()` before trying to read a value from it with `next()`.
2. If this is the first iteration, there is no `lastNode` set, so you take the first node of the list. After the first iteration, you can get the `next` property of the last node to find the next one.
3. Since the `lastNode` property was updated, you need to update the `index` too. You've now gone through one more iteration, so the `index` increments.

Now that you've implemented `Iterator`, you can do some really cool things. For example, you can iterate your linked list with a regular Kotlin `for` loop and print the double of each element in a list.

Add this to `main()`:

```

"printing doubles" example {
    val list = LinkedList<Int>()
    list.push(3)
    list.push(2)
    list.push(1)
    println(list)

    for (item in list) {
        println("Double: ${item * 2}")
    }
}

```

The output you'll get is:

```

---Example of printing doubles---
1 -> 2 -> 3

```

```
Double: 2  
Double: 4  
Double: 6
```

Cool, huh? A `for` loop helps you a lot but there are still things to implement if you want to have a fully featured `MutableCollection`.

Becoming a collection

Being a `Collection` requires more than just being an `Iterable` class. Change the definition of your `LinkedList` to implement `Collection`:

```
class LinkedList<T> : Iterable<T>, Collection<T> {  
    ...
```

Of course, you could now remove `Iterable` because a `Collection` is an `Iterable` anyway. You may also leave it there so that you can see the progress you're making.

The compiler will start complaining about the methods you need to implement. Here are some quick wins: You already have `isEmpty()` and `size`, so you can add the `override` keyword in front of them.

```
override var size = 0  
    private set  
  
override fun isEmpty(): Boolean {  
    return size == 0  
}
```

You still need to implement two more methods, but the good news is that you can use one to implement the other easily:

```
override fun contains(element: T): Boolean {  
    TODO("not implemented")  
}  
  
override fun containsAll(elements: Collection<T>): Boolean {  
    TODO("not implemented")  
}
```

Since you can now iterate through the list with `for`, the implementation of `contains` is straightforward:

```
override fun contains(element: T): Boolean {  
    for (item in this) {
```

```

        if (item == element) return true
    }
    return false
}

```

This method iterates through all elements of the list if needed, so it has a complexity of $O(n)$.

The second method is similar; it just works with a collection of elements.

```

override fun containsAll(elements: Collection<T>): Boolean {
    for (searched in elements) {
        if (!contains(searched)) return false
    }
    return true
}

```

As you'd probably guess, this is an inefficient method, it's $O(n^2)$. But if the Collection interface requires it, you need to provide it.

Mutating while iterating

To get to the 3rd tier, you need to make `LinkedList` a `MutableIterable`. If you add this interface to the list that `LinkedList` implements, you'll see the compiler complain again because `iterator()` doesn't return a `MutableIterator`. This time, start the other way around. Make `LinkedListIterator` implement the `MutableIterator<T>` interface:

```

class LinkedListIterator<T>(
    private val list: LinkedList<T>
) : Iterator<T>, MutableIterator<T> {
    ...
}

```

Again, `MutableIterator` is a broader interface than `Iterator`, so you can remove `Iterator` from the list of implemented interfaces.

You'll need to add the `remove()` method to comply with the new interface you've added:

```

override fun remove() {
    // 1
    if (index == 1) {
        list.pop()
    } else {
        // 2
    }
}

```

```

    val prevNode = list.nodeAt(index - 2) ?: return
    // 3
    list.removeAfter(prevNode)
    lastNode = prevNode
}
index--
}

```

Here's a breakdown of how this code uses the methods `LinkedList` already has:

1. The simplest case is when you're at the beginning of the list. Using `pop()` will do the trick.
2. If the iterator is somewhere inside the list, it needs to find the previous node. That's the only way to remove items from a linked list.
3. The iterator needs to step back so that `next()` returns the correct method the next time. This means reassigning the `lastNode` and decreasing the index.

Now, go to `LinkedList.kt` and add `MutableIterable` to the class definition:

```

class LinkedList<T>: Iterable<T>, Collection<T>,
MutableIterable<T> {
...
}
```

Modify `iterator()` to return a `MutableIterator`, which `LinkedListIterator` implements:

```

override fun iterator(): MutableIterator<T> {
    return LinkedListIterator(this)
}
```

Final step: Mutable collection

You've already completed the hardest steps, so this last step shouldn't be too bad.

First, add the `MutableCollection` interface to the definition of `LinkedList`:

```

class LinkedList<T>: Iterable<T>, Collection<T>,
MutableIterable<T>, MutableCollection<T> {
...
}
```

This will make you add six more methods:

```

override fun add(element: T): Boolean {
    TODO("not implemented")
```

```

    }

    override fun addAll(elements: Collection<T>): Boolean {
        TODO("not implemented")
    }

    override fun clear() {
        TODO("not implemented")
    }

    override fun remove(element: T): Boolean {
        TODO("not implemented")
    }

    override fun removeAll(elements: Collection<T>): Boolean {
        TODO("not implemented")
    }

    override fun retainAll(elements: Collection<T>): Boolean {
        TODO("not implemented")
    }
}

```

Three of them are relatively simple to implement. `add()`, `addAll()` and `clear()` are almost one-liners:

```

override fun add(element: T): Boolean {
    append(element)
    return true
}

override fun addAll(elements: Collection<T>): Boolean {
    for (element in elements) {
        append(element)
    }
    return true
}

override fun clear() {
    head = null
    tail = null
    size = 0
}

```

Since the `LinkedList` doesn't have a fixed size, `add()` and `addAll()` are always successful and need to return `true`. For the removal of elements, you'll use a different approach for iterating through your `MutableIterable` linked list. This way, you can benefit from your `MutableIterator`:

```

override fun remove(element: T): Boolean {
    // 1
}

```

```

    val iterator = iterator()
    // 2
    while (iterator.hasNext()) {
        val item = iterator.next()
        // 3
        if (item == element) {
            iterator.remove()
            return true
        }
    }
    // 4
    return false
}

```

This method is a little complex, so here's the step-by-step walkthrough:

1. Get an iterator that will help you iterate through the collection.
2. Create a loop that checks if there are any elements left, and gets the next one.
3. Check if the current element is the one you're looking for, and if it is, remove it.
4. Return a boolean that signals if an element has been removed.

With `removeAll()`, you can make use of `remove()`:

```

override fun removeAll(elements: Collection<T>): Boolean {
    var result = false
    for (item in elements) {
        result = remove(item) || result
    }
    return result
}

```

The return value of `removeAll` is `true` if any elements were removed.

The last method to implement is `retainAll()`, which should remove any elements in the list besides the ones passed in as the parameter. You'll need to approach this the other way around. Iterate through your list once and remove any element that is not in the parameter. Luckily, the parameter of `retainAll` is also a collection, so you can use all of the methods you implemented yourself, like `contains`:

```

override fun retainAll(elements: Collection<T>): Boolean {
    var result = false
    val iterator = this.iterator()
    while (iterator.hasNext()) {
        val item = iterator.next()
        if (!elements.contains(item)) {
            iterator.remove()
        }
    }
    return result
}

```

```
        result = true
    }
}
return result
}
```

Congrats! You finished the implementation, so it's time for some testing. Go back into **Main.kt** and add these at the end of `main()`:

```
"removing elements" example {
    val list: MutableCollection<Int> = LinkedList()
    list.add(3)
    list.add(2)
    list.add(1)

    println(list)
    list.remove(1)
    println(list)
}

"retaining elements" example {
    val list: MutableCollection<Int> = LinkedList()
    list.add(3)
    list.add(2)
    list.add(1)
    list.add(4)
    list.add(5)

    println(list)
    list.retainAll(listOf(3, 4, 5))
    println(list)
}

"remove all elements" example {
    val list: MutableCollection<Int> = LinkedList()
    list.add(3)
    list.add(2)
    list.add(1)
    list.add(4)
    list.add(5)

    println(list)
    list.removeAll(listOf(3, 4, 5))
    println(list)
}
```

As you'll see momentarily, your first challenge in this chapter is to check the output of each example to make sure it's correct. You'll also see the rest of the challenges after a quick summary.

Challenges

In these challenges, you'll work through five common scenarios for the linked list. These problems are relatively easy compared to most challenges, and they will serve to solidify your knowledge of data structures. You'll find the solutions to the challenges at the end of this chapter.

Challenge 1: Reverse a linked list

Create an extension function that prints out the elements of a linked list in reverse order. Given a linked list, print the nodes in reverse order. For example:

```
1 -> 2 -> 3  
// should print out the following:  
3  
2  
1
```

Solution 1

A straightforward way to solve this problem is to use recursion. Since recursion allows you to build a call stack, you need to call the `print` statements as the call stack unwinds.

Your first task is to define an extension function for `LinkedList`. Add the following helper function to your solution file:

```
fun <T> LinkedList<T>.printInReverse() {  
    this.nodeAt(0)?.printInReverse()  
}
```

This function forwards the call to the recursive function that traverses the list, node by node. To traverse the list, add this extension function for `Node`:

```
fun <T> Node<T>.printInReverse() {  
    this.next?.printInReverse()  
}
```

As you'd expect, this function calls itself on the next node. The terminating condition is somewhat hidden in the null-safety operator. If the value of `next` is `null`, the function stops because there's no next node on which to call `printInReverse()`. You're almost done; the next step is printing the nodes.

Printing

Where you add the `print` statement will determine whether you print the list in reverse order or not. Update the function to the following:

```
fun <T> Node<T>.printInReverse() {
    this.next?.printInReverse()
    // 1
    if (this.next != null) {
        print(" -> ")
    }
    // 2
    print(this.value.toString())
}
```

Any code that comes after the recursive call is called only after the base case triggers (i.e., after the recursive function hits the end of the list).

1. First, you check if you've reached the end of the list. That's the beginning of the reverse printing, and you'll not add an arrow there. The arrows start with the second element of the reverse output. This is just for pretty formatting.
2. As the recursive statements unravel, the node data gets printed.

Test it out!

Write the following at the bottom of `main()`:

```
"print in reverse" example {
    val list = LinkedList<Int>()
    list.add(3)
    list.add(2)
    list.add(1)
    list.add(4)
    list.add(5)

    println(list)
    list.printInReverse()
}
```

You'll see the following output:

```
---Example of print in reverse---
3 -> 2 -> 1 -> 4 -> 5
5 -> 4 -> 1 -> 2 -> 3
```

The time complexity of this algorithm is **$O(n)$** since you have to traverse each node of the list.

Challenge 2: The item in the middle

Given a linked list, find the middle node of the list. For example:

```
1 -> 2 -> 3 -> 4  
// middle is 3  
  
1 -> 2 -> 3  
// middle is 2
```

Solution 2

One solution is to have two references traverse down the nodes of the list where one is twice as fast as the other. Once the faster reference reaches the end, the slower reference will be in the middle. Write the following function:

```
fun <T> LinkedList<T>.getMiddle(): Node<T>? {  
    var slow = this.nodeAt(0)  
    var fast = this.nodeAt(0)  
  
    while (fast != null) {  
        fast = fast.next  
        if (fast != null) {  
            fast = fast.next  
            slow = slow?.next  
        }  
    }  
  
    return slow  
}
```

In the while declaration, you bind the next node to fast. If there's a next node, you update fast to the next node of fast, effectively traversing down the list twice. slow is updated only once. This is also known as the **runner technique**.

Try it out!

Write the following at the bottom of **Main.kt**:

```
"print middle" example {  
    val list = LinkedList<Int>()  
    list.add(3)  
    list.add(2)  
    list.add(1)  
    list.add(4)  
    list.add(5)  
  
    println(list)
```

```
    } println(list.getMiddle()?.value)
```

You'll see the following output:

```
---Example of print middle---
3 -> 2 -> 1 -> 4 -> 5
1
```

The time complexity of this algorithm is **$O(n)$** since you traversed the list in a single pass. The **runner technique** helps solve a variety of problems associated with the linked list.

Challenge 3: Reverse a linked list

To reverse a list is to manipulate the nodes so that the nodes of the list are linked in the opposite direction. For example:

```
// before  
1 -> 2 -> 3  
  
// after  
3 -> 2 -> 1
```

Solution 3

To reverse a linked list, you need to visit each node and update the `next` reference to point in the other direction. This can be a tricky task since you'll need to manage multiple references to multiple nodes. To do this, you would also need access to the `head` and `tail` of your liked list. Since you're implementing an extension function, you won't have access to these variables. Luckily, there's a simpler solution that has a small drawback discussed later.

You can easily reverse a list by using a recursive function that goes to the end of the list and then starts copying the nodes when it returns, in a new linked list. Here's how this function would look like:

```
private fun <T> addInReverse(list: LinkedList<T>, node: Node<T>)  
{  
    // 1  
    val next = node.next  
    if (next != null) {  
        // 2  
        addInReverse(list, next)  
    }  
    // 3  
    list.append(node.value)  
}
```

The following explains how this function works:

1. Get the next node of the list, starting from the one you've received as a parameter.
2. If there's a following node, recursively call the same function; however, now the starting node is the one after the current node.
3. When you reach the end, start adding the nodes in the reverse order.

O(n) time complexity, short and sweet! The only drawback is that you need a new list, which means that the space complexity is also **O(n)**.

To use this helper function conveniently on a `LinkedList`, add this extension function:

```
fun <T> LinkedList<T>.reversed(): LinkedList<T> {
    val result = LinkedList<T>()
    val head = this.elementAt(0)
    if (head != null) {
        addInReverse(result, head)
    }
    return result
}
```

This extension creates a new `LinkedList` and fills it with nodes by calling `addInReverse()`, passing in the first node of the current list.

Try it out!

Test `reversed()` by writing the following at the bottom of `main()`:

```
"reverse list" example {
    val list = LinkedList<Int>()
    list.add(3)
    list.add(2)
    list.add(1)
    list.add(4)
    list.add(5)

    println("Original: $list")
    println("Reversed: ${list.reversed()}")
}
```

You'll see the following output:

```
---Example of reverse list---
Original: 3 -> 2 -> 1 -> 4 -> 5
Reversed: 5 -> 4 -> 1 -> 2 -> 3
```

Challenge 4: Merging two linked lists

Create a function that takes two sorted linked lists and merges them into a single sorted linked list

Your goal is to return a new linked list that contains the nodes from two lists in sorted order. You may assume they are both sorted in ascending order. For example:

```
// list1  
1 -> 4 -> 10 -> 11  
  
// list2  
-1 -> 2 -> 3 -> 6  
  
// merged list  
-1 -> 1 -> 2 -> 3 -> 4 -> 6 -> 10 -> 11
```

Solution 4

The solution to this problem is to continuously pluck nodes from the two sorted lists and add them to a new list. Since the two lists are sorted, you can compare the next node of both lists to see which one should be the next one to add to the new list.

Setting up

You'll begin by checking the cases where one or both of the lists are empty. Create the following `mergeSorted` extension function:

```
fun <T : Comparable<T>> LinkedList<T>.mergeSorted(  
    otherList: LinkedList<T>  
) : LinkedList<T> {  
    if (this.isEmpty()) return otherList  
    if (otherList.isEmpty()) return this  
  
    val result = LinkedList<T>()  
  
    return result  
}
```

If one is empty, you return the other. You also introduce a new reference to hold a new `LinkedList`. The strategy is to merge the nodes in `this` and `otherList` into `result` in sorted order.

Next, you need to write a helper function that adds the current node to the result list and returns the next node. You'll use this function in your algorithm multiple times, so it's useful to have it extracted:

```
private fun <T : Comparable<T>> append(
    result: LinkedList<T>,
    node: Node<T>
): Node<T>? {
    result.append(node.value)
    return node.next
}
```

Merging

Add the following to `mergeSorted` immediately below the declaration for `result` and `right` above `return result`:

```
// 1
var left = nodeAt(0)
var right = otherList.nodeAt(0)
// 2
while (left != null && right != null) {
    // 3
    if (left.value < right.value) {
        left = append(result, left)
    } else {
        right = append(result, right)
    }
}
```

Here's how it works:

1. You start with the first node of each list.
2. The while loop continues until one of the lists reaches its end.
3. You compare the first nodes `left` and `right` to append to the `result`.

Since this loop depends on both `left` and `right`, it will terminate even if there are nodes left in either list.

Finally, add the following below the newly added code, and above `return result`, to handle the remaining nodes:

```
while (left != null) {
    left = append(result, left)
}

while (right != null) {
    right = append(result, right)
}
```

Try it out!

Write the following at the bottom of `main()`:

```
"merge lists" example {
    val list = LinkedList<Int>()
    list.add(1)
    list.add(2)
    list.add(3)
    list.add(4)
    list.add(5)

    val other = LinkedList<Int>()
    other.add(-1)
    other.add(0)
    other.add(2)
    other.add(2)
    other.add(7)

    println("Left: $list")
    println("Right: $other")
    println("Merged: ${list.mergeSorted(other)}")
}
```

You'll see the following output:

```
---Example of merge lists---
Left: 1 -> 2 -> 3 -> 4 -> 5
Right: -1 -> 0 -> 2 -> 2 -> 7
Merged: -1 -> 0 -> 1 -> 2 -> 2 -> 2 -> 3 -> 4 -> 5 -> 7
```

This algorithm has a time complexity of $O(m + n)$, where m is the # of nodes in the first list, and n is the # of nodes in the second list.

Key points

- Linked lists are linear and unidirectional. As soon as you move a reference from one node to another, you can't go back.
- Linked lists have a $O(1)$ time complexity for head-first insertions. Arrays have $O(n)$ time complexity for head-first insertions.
- Conforming to Kotlin collection interfaces, such as `Iterable` and `Collection`, offers a host of helpful methods for a reasonably small amount of requirements.

Chapter 4: Stack Data Structures

By Matei Ţuică

Stacks are everywhere. Some common examples of things you might stack:

- Pancakes.
- Books.
- Paper.
- Cash, especially cash. :]

The **stack** data structure is identical, in concept, to a physical stack of objects. When you add an item to a stack, you place it on top of the stack. When you remove an item from a stack, you always remove the top-most item.



Good news: A stack of pancakes. Bad news: You may only eat the top-most pancake.

Stack operations

Stacks are useful, and also exceedingly simple. The main goal of building a stack is to enforce how you access your data. If you had a tough time with the linked list concepts, you'll be glad to know that stacks are comparatively trivial.

There are only two essential operations for a stack:

- **push**: Adding an element to the top of the stack.
- **pop**: Removing the top element of the stack.

This means that you can only add or remove elements from one side of the data structure. In computer science, a stack is known as the **LIFO** (last-in first-out) data structure. Elements that are pushed in last are the first ones to be popped out.

If you want this in code, open then the started project, create the **Stack.kt** file into the **stack** package, and write the following code:

```
interface Stack<Element> {  
    fun push(element: Element)  
    fun pop(): Element?  
}
```

Note: The previous Stack interface is different from the Stack class provided by Kotlin (or Java) which extends the Vector class and provides methods we don't need here.

Stacks are used prominently in all disciplines of programming, such as:

- Android uses the *fragment stack* to push and pop fragments into and out of an Activity.
- Memory allocation uses stacks at the architectural level. Memory for local variables is also managed using a stack.
- *Search and conquer* algorithms, such as finding a path out of a maze, use stacks to facilitate backtracking.

Implementation

You can implement your Stack interface in different ways and choosing the right storage type is important. The `ArrayList` is an obvious choice since it offers constant time insertions and deletions at one end via `add` and `removeAt` with the last index as a parameter. Usage of these two operations will facilitate the **LIFO** nature of stacks.

In the same `Stack.kt` file you can then start your implementation with the following code:

```
class Stack<T : Any>() {
    private val storage = arrayListOf<T>()

    override fun toString() = buildString {
        appendln("----top----")
        storage.asReversed().forEach {
            appendln("$it")
        }
        appendln("-----")
    }
}
```

You defines a private property of type `ArrayList` for the data and you override the `toString` method in order to display its content for debug purposes. With this code, you'll get some errors because of the missing implementation of the `push` and `pop` operations but you're going to fix this soon.

push and pop operations

Add the following two operations to your Stack:

```
override fun push(element: Element) {
    storage.add(element)
}

override fun pop(): Element? {
    if (storage.size == 0) {
        return null
    }
    return storage.removeAt(storage.size - 1)
}
```

In the the push method you just append the value passed as parameter to the end of the `ArrayList` using it's add method. In the pop method you simply return null if the storage is empty or you remove and return the last element you have inserter.

It's time to see it working. Open `Main.kt` and write this code into the `main()`:

```
"using a stack" example {
    val stack = StackImpl<Int>().apply {
        push(1)
        push(2)
        push(3)
        push(4)
    }
    print(stack)
    val poppedElement = stack.pop()
    if (poppedElement != null) {
        println("Popped: $poppedElement")
    }
    print(stack)
}
```

You'll see the following output:

```
---Example of using a stack---
----top----
4
3
2
1
-----
Popped: 4
----top----
3
2
1
-----
```

`push` and `pop` both have an $O(1)$ time complexity.

Non-essential operations

Next, you'll add some nice-to-have operations that make stacks easier to use.

In `Stack.kt`, add the following code to the `Stack` interface :

```
fun peek(): Element?
val count: Int
    get
```

```
val isEmpty: Boolean  
    get() = count == 0
```

peek is an operation that's often attributed to the stack interface. The idea of peek is to look at the top element of the stack without mutating its contents. The count property returns the number of elements into the Stack and it's used to implement the isEmpty property.

You now need to add the implementation to the StackImpl class with this code:

```
override fun peek(): Element? {  
    return storage.lastOrNull()  
}  
  
override val count: Int  
    get() = storage.size
```

This allows you to have a cleaner code changing the implementation of the pop method like this:

```
override fun pop(): Element? {  
    if (isEmpty) {  
        return null  
    }  
    return storage.removeAt(count - 1)  
}
```

Less is more

You may have wondered if you could adopt the Kotlin collection interfaces for the stack. A stack's purpose is to limit the number of ways to access your data, and adopting interfaces such as Iterable would go against this goal by exposing all of the elements via iterators. In this case, less is more!

You might want to take an existing list and convert it to a stack so that the access order is guaranteed. Of course, it would be possible to loop through the array elements and push each element. However, you can write a **static factory method** that directly adds these elements to the Stack implementation.

Add the following code to StackImpl class

```
companion object {  
    fun <Element> create(items: Iterable<Element>):  
        Stack<Element> {  
        val stack = StackImpl<Element>()  
        for (item in items) {
```

```

        stack.push(item)
    }
    return stack
}
}

```

Now, add this example to the `main()` function:

```

"initializing a stack from a list" example {
    val list = listOf("A", "B", "C", "D")
    val stack = StackImpl.create(list)
    print(stack)
    println("Popped: ${stack.pop()}")
}

```

This code creates a stack of strings and pops the top element "D". Notice that the Kotlin compiler can type infer the element type from the list so you can use `Stack` instead of the more verbose `Stack<String>`.

You can go a step further and make your stack initializable by listing its elements, similar to `listOf()` and other standard library collection factory functions. Add this to `Stack.kt`, outside the `Stack` class definition:

```

fun <Element> stackOf(vararg elements: Element): Stack<Element>
{
    return StackImpl.create(elements.asList())
}

```

Now, go back to `main()` and add:

```

"initializing a stack from an array literal" example {
    val stack = stackOf(1.0, 2.0, 3.0, 4.0)
    print(stack)
    println("Popped: ${stack.pop()}")
}

```

This creates a stack of `Doubles` and pops the top value 4.0. Again, type inference saves you from having to specify the generic type argument of the `stackOf` function call.

Stacks are crucial to problems that *search* trees and graphs. Imagine finding your way through a maze. Each time you come to a decision point of left, right or straight, you can push all possible decisions onto your stack. When you hit a dead end, backtrack by popping from the stack and continuing until you escape or hit another dead end.

Challenges

A stack is a simple data structure with a surprisingly large amount of applications. Complete the following challenges to see what it can do.

Challenge 1: Reverse a LinkedList

Given a linked list, print the nodes in reverse order. You should not use recursion to solve this problem.

Solution 1

One of the prime use cases for stacks is to facilitate backtracking. If you push a sequence of values into the stack, sequentially popping the stack will give you the values in reverse order:

```
fun <T> LinkedList<T>.printInReverse() {
    val stack = StackImpl<T>()

    for (node in this) {
        stack.push(node)
    }

    var node = stack.pop()
    while (node != null) {
        println(node)
        node = stack.pop()
    }
}
```

Here's how it works:

1. Copy the content of the list into a stack, carefully putting the nodes on top of each other.
2. Remove and print the nodes from the stack one by one, starting from the top.

The time complexity of pushing the nodes into the stack is $O(n)$. The time complexity of popping the stack to print the values is also $O(n)$. Overall, the time complexity of this algorithm is $O(n)$.

Since you're allocating a container (the stack) inside the function, you also incur an $O(n)$ space complexity cost.

Challenge 2: The parentheses validation

Check for balanced parentheses. Given a string, check if there are (and) characters, and return true if the parentheses in the string are balanced. For example:

```
// 1  
h((e))llo(world)() // balanced parentheses  
  
// 2  
(hello world // unbalanced parentheses
```

Solution 2

To check if there are balanced parentheses in the string, you need to go through each character of the string. When you encounter an opening parenthesis, you'll push that into a stack. Vice versa, if you encounter a closing parenthesis, you should pop the stack.

Here's the code:

```
fun String.checkParentheses(): Boolean {  
    val stack = StackImpl<Char>()  
  
    for (character in this) {  
        when (character) {  
            '(' -> stack.push(character)  
            ')' -> if (stack.isEmpty) {  
                return false  
            } else {  
                stack.pop()  
            }  
        }  
    }  
    return stack.isEmpty  
}
```

Here's how it works:

1. Create a new stack and start going through your string, character by character.
2. Push every opening parenthesis into the stack.
3. Pop one item from the stack for every closing parenthesis, but if you're out of items on the stack, your string is already imbalanced, so you can immediately return from the function.

4. In the end, a balanced string is one that has popped all of the opening parentheses it's pushed (and not a single item more). That would leave the stack empty because you popped all the parentheses you pushed before.

The time complexity of this algorithm is $O(n)$, where n is the number of characters in the string. This algorithm also incurs an $O(n)$ space complexity cost due to the usage of the Stack data structure.

Key points

- Despite its simplicity, the stack is a key data structure for many problems.
- The only two essential operations for the stack are the **push** method for adding elements and the **pop** method for removing elements.

Chapter 5: Queues

By Matei Ţuică

We're all familiar with waiting in line. Whether you're in line to buy tickets to your favorite movie or waiting for a printer to print a file, these real-life scenarios mimic the **queue** data structure.

Queues use **FIFO** or **first in, first out** ordering, meaning the first element that was added will always be the first one removed. Queues are handy when you need to maintain the order of your elements to process later.

In this chapter, you'll learn all of the common operations of a queue, go over the various ways to implement a queue and look at the time complexity of each approach.



Common operations

First, establish an interface for queues. In the **base** package, create a file named **Queue.kt** and add the following code defining the Queue interface.

```
interface Queue<T> {  
  
    fun enqueue(element: T): Boolean  
  
    fun dequeue(): T?  
  
    val count: Int  
        get  
  
    val isEmpty: Boolean  
        get() = count == 0  
  
    fun peek(): T?  
}
```

This will be your starting point. From now on, everything you implement will obey the contract of this interface, which describes the core operations for a queue.

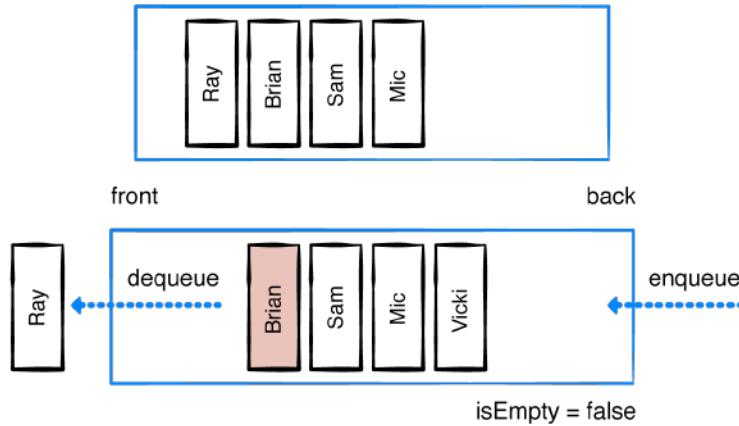
The core operations for a queue are:

- **enqueue**: Inserts an element at the back of the queue and returns `true` if the operation is successful.
- **dequeue**: Removes the element at the front of the queue and returns it.
- **isEmpty**: Checks if the queue is empty using the `count` property
- **peek**: Returns the element at the front of the queue *without* removing it.

Notice that the queue only cares about removal from the front and insertion at the back. You don't need to know what the contents are in between. If you did, you'd presumably use an array instead of a Queue.

Example of a queue

The easiest way to understand how a queue works is to look at a working example. Imagine a group of people waiting in line for a movie ticket.



This queue currently holds Ray, Brian, Sam and Mic. Once Ray receives his ticket, he moves out of the line. When you call `dequeue()`, Ray is removed from the *front* of the queue.

Calling `peek()` returns Brian since he's now at the front of the line.

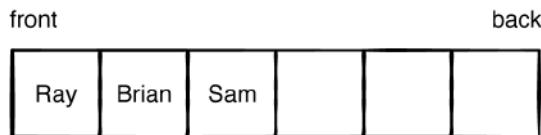
Now comes Vicki, who just joined the line to buy a ticket. When you call `enqueue("Vicki")`, Vicki gets added to the *back* of the queue.

In the following sections, you'll learn to create a queue in four different ways:

- Using an array based list
- Using a doubly linked list
- Using a ring buffer
- Using two stacks

List-based implementation

The Kotlin standard library comes with a core set of highly optimized data structures that you can use to build higher-level abstractions. One of these is the `ArrayList`, a data structure that stores a contiguous, ordered list of elements. In this section, you'll use an `ArrayList` to create a queue.



A simple Kotlin `ArrayList` can be used to model the queue.

Open the starter project. In the `list` package, create a file named `ArrayListQueue.kt` and add the following:

```
class ArrayListQueue<T> : Queue<T> {  
    private val list = arrayListOf<T>()  
}
```

Here, you defined a generic `ArrayListQueue` class that implements the `Queue` interface. Note that the interface implementation uses the same generic type `T` for the elements it stores.

Next, you'll complete the implementation of `ArrayListQueue` to fulfill the `Queue` contract.

Leveraging `ArrayList`

Add the following code to `ArrayListQueue`:

```
override val count: Int  
    get() = list.size  
  
override fun peek(): T? = list.getOrNull(0)
```

Using the features of `ArrayList`, you get the following for free:

1. Get the size of the queue using the same property of the list.
2. Return the element at the front of the queue, if there is any.

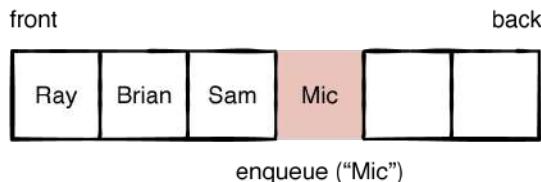
These operations are all $O(1)$.

Enqueue

Adding an element to the back of the queue is easy. You simply add the element to the end of the `ArrayList`. Add the following:

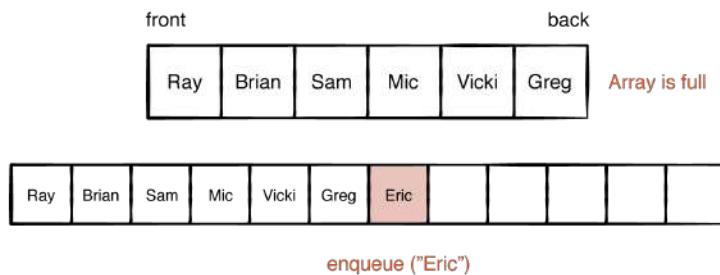
```
override fun enqueue(element: T): Boolean {
    list.add(element)
    return true
}
```

Regardless of the size of the list, enqueueing an element is an $O(1)$ operation. This is because the list has empty space at the back.



In the example above, notice that once you add Mic, the list has two empty spaces.

After adding multiple elements, the internal array of the `ArrayList` will eventually be full. When you want to use more than the allocated space, the array must resize to make additional room.



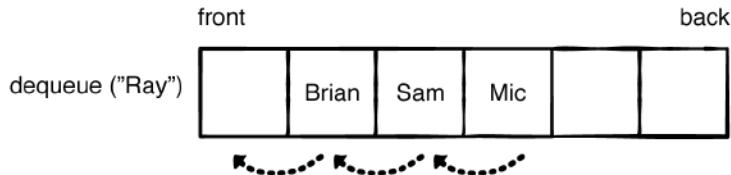
Resizing is an $O(n)$ operation. Resizing requires the list to allocate new memory and copy all existing data over to the new list. Since this doesn't happen very often (thanks to doubling the size each time), the complexity still works out to be an amortized $O(1)$.

Dequeue

Removing an item from the front requires a bit more work. Add the following:

```
override fun dequeue(): T? =  
    if (isEmpty) null else list.removeAt(0)
```

If the queue is empty, `dequeue()` simply returns `null`. If not, it removes the element from the front of the list and returns it.



Removing an element from the front of the queue is an $O(n)$ operation. To dequeue, you remove the element from the beginning of the list. This is always a linear time operation because it requires all of the remaining elements in the list to be shifted in memory.

Debug and test

For debugging purposes, you'll have your queue override `toString()`. Add the following at the bottom of the class:

```
override fun toString(): String = list.toString()
```

It's time to try out the queue that you just implemented. In `Main.kt`, add the following to the bottom of `main()`:

```
"Queue with ArrayList" example {  
    val queue = ArrayListQueue<String>().apply {  
        enqueue("Ray")  
        enqueue("Brian")  
        enqueue("Eric")  
    }  
    println(queue)  
    queue.dequeue()  
    println(queue)  
    println("Next up: ${queue.peek()}")  
}
```

This code puts Ray, Brian and Eric in the queue. It then removes Ray and peeks at Brian, but it doesn't remove him.

Strengths and weaknesses

Here's a summary of the algorithmic and storage complexity of the `ArrayList`-based queue implementation. Most of the operations are constant time except for `dequeue()`, which takes linear time. Storage space is also linear.

Array-Based Queue

Operations	Best case	Worse case
enqueue	$O(1)$	$O(1)$
dequeue	$O(n)$	$O(n)$
Space Complexity	$O(n)$	$O(n)$

You've seen how easy it is to implement a list-based queue by leveraging a Kotlin `ArrayList`. Enqueue is very fast, thanks to an $O(1)$ append operation.

There are some shortcomings to the implementation. Removing an item from the front of the queue can be inefficient, as removal causes all elements to shift up by one. This makes a difference for very large queues. Once the list gets full, it has to resize and may have unused space. This could increase your memory footprint over time. Is it possible to address these shortcomings? Let's look at a linked list-based implementation and compare it to an `ArrayListQueue`.

Doubly linked list implementation

Create a new file named `LinkedListQueue.kt` in the `linkedlist` package. In this package, you'll notice a `DoublyLinkedList` class. You should already be familiar with linked lists from Chapter 3, "Linked Lists". A doubly linked list is simply a linked list in which nodes also contain a reference to the previous node.



Start by adding a generic `LinkedListQueue` to the same package, with the following content:

```
class LinkedListQueue<T> : Queue<T> {

    private val list = DoublyLinkedList<T>()

    private var size = 0

    override val count: Int
        get() = size
}
```

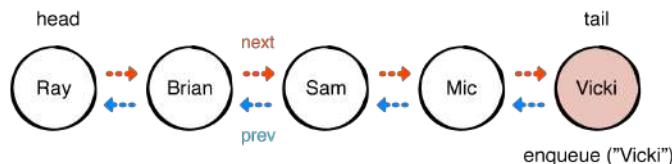
This implementation is similar to `ArrayListQueue`, but instead of an `ArrayList`, you create a `DoublyLinkedList`.

Next, you'll start implementing the `Queue` interface starting from the `count` property that the `DoublyLinkedList` doesn't provide.

Enqueue

To add an element to the back of the queue, add the following:

```
override fun enqueue(element: T): Boolean {
    list.append(element)
    size++
    return true
}
```



Behind the scenes, the doubly linked list will update its tail node's previous and next references to the new node. You also increment the size. This is an $O(1)$ operation.

Dequeue

To remove an element from the queue, add the following:

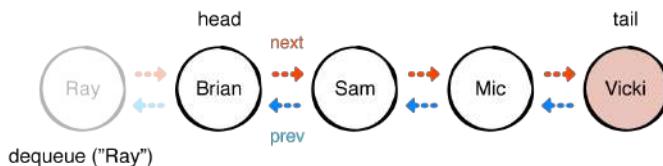
```
override fun dequeue(): T? {
    val firstNode = list.first ?: return null
```

```

    size--
    return list.remove(firstNode)
}

```

This code checks to see if the first element of the queue exists. If it doesn't, it returns null. Otherwise, it removes and returns the element at the front of the queue. In this case it also decrement the size.



Removing from the front of the list is also an $O(1)$ operation. Compared to the `ArrayList` implementation, you didn't have to shift elements one by one. Instead, in the diagram above, you simply update the next and previous pointers between the first two nodes of the linked list.

Checking the state of a queue

Similar to the `ArrayList` based implementation, you can implement `peek()` using the properties of the `DoublyLinkedList`. Add the following:

```
override fun peek(): T? = list.first?.value
```

Debug and test

For debugging purposes, add the following at the bottom of the class:

```
override fun toString(): String = list.toString()
```

This leverages the `DoublyLinkedList`'s existing implementation for `toString()`.

That's all there is to implementing a queue using a linked list. In `Main.kt`, you can try the following example:

```
"Queue with Doubly Linked List" example {
    val queue = LinkedListQueue<String>().apply {
        enqueue("Ray")
        enqueue("Brian")
        enqueue("Eric")
    }
}
```

```
    println(queue)
    queue.dequeue()
    println(queue)
    println("Next up: ${queue.peek()}")
}
```

This test code yields the same results as your `ArrayListQueue` implementation.

Strengths and weaknesses

Time to summarize the algorithmic and storage complexity of the implementation based on a doubly linked list.

Linked-List Based Queue

Operations	Best case	Worse case
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
Space Complexity	O(n)	O(n)

One of the main problems with `ArrayListQueue` is that dequeuing an item takes linear time. With the linked list implementation, you reduced it to a constant operation, $O(1)$. All you needed to do was update the node's previous and next pointers.

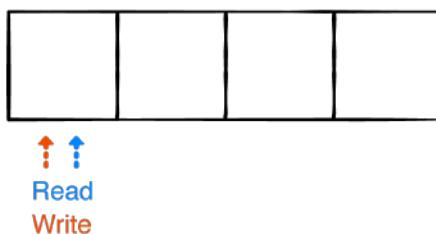
The main weakness with `LinkedListQueue` is not apparent from the table. Despite $O(1)$ performance, it suffers from high overhead. Each element has to have extra storage for the forward and back reference. Moreover, every time you create a new element, it requires a relatively expensive dynamic allocation. By contrast, `ArrayListQueue` does bulk allocation, which is faster.

Can you eliminate allocation overhead and preserve $O(1)$ dequeues? If you don't have to worry about your queue ever growing beyond a fixed size, you can use a different approach like the **ring buffer**. For example, you might have a game of *Monopoly* with five players. You can use a queue based on a ring buffer to keep track of whose turn is coming up next. You'll take a look at a ring buffer implementation next.

Ring buffer implementation

A ring buffer, also known as a **circular buffer**, is a fixed-size array. This data structure strategically wraps around to the beginning when there are no more items to remove at the end.

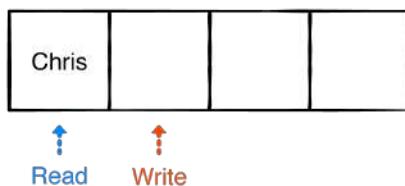
Going over a simple example of how a queue can be implemented using a ring buffer:



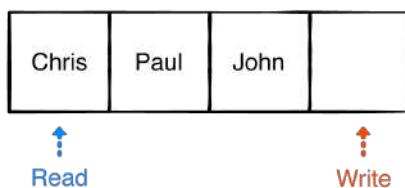
You first create a ring buffer that has a fixed size of **4**. The ring buffer has two “pointers” that keep track of two things:

1. The **read** pointer keeps track of the front of the queue.
2. The **write** pointer keeps track of the next available slot so that you can override existing elements that have already been read.

Enqueue an item:

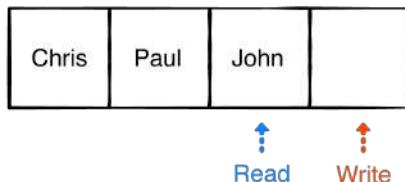


Each time that you add an item to the queue, the **write** pointer increments by one.
Add a few more elements:



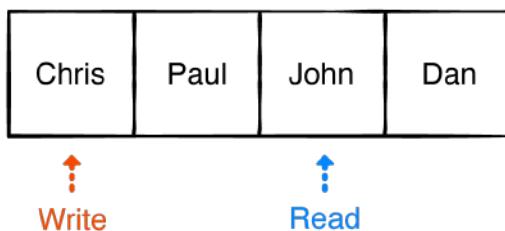
Notice that the **write** pointer moved two more spots and is ahead of the **read** pointer. This means that the queue is not empty.

Next, dequeue two items:



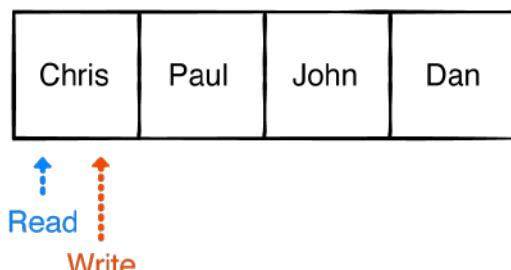
Dequeuing is the equivalent of reading a ring buffer. Notice how the **read** pointer moved twice.

Now, enqueue one more item to fill up the queue:



Since the **write** pointer reached the end, it simply wraps around to the starting index again.

Finally, dequeue the two remaining items:



The **read** pointer wraps to the beginning, as well.

As a final observation, notice that whenever the read and write pointers are at the same index, that means the queue is **empty**.

Now that you have a better understanding of how ring buffers make a queue, you're ready to implement one!

Go to the **ringbuffer** package and create a file named **RingBufferQueue.kt**. You'll notice a **RingBuffer** class inside this package, which you can look at to understand its internal mechanics.

In **RingBufferQueue.kt**, add the following:

```
class RingBufferQueue<T>(size: Int) : Queue<T> {  
  
    private val ringBuffer: RingBuffer<T> = RingBuffer(size)  
  
    override val count: Int  
        get() = ringBuffer.count  
  
    override fun peek(): T? = ringBuffer.first  
}
```

Here, you define a generic **RingBufferQueue**. Note that you must include a **size** parameter since the ring buffer has a fixed size.

To implement the **Queue** interface, you also need to implement **peek()** and the **count** property using the same from the **RingBuffer** class. Once you provide the **count** property, the **isEmpty** property is already defined in the **Queue** interface. Instead of exposing **ringBuffer**, you provide these helpers to access the front of the queue and to check if the queue is empty. Both of these are $O(1)$ operations.

Enqueue

Next, add the following method at the end of the **RingBufferQueue** class:

```
override fun enqueue(element: T): Boolean =  
    ringBuffer.write(element)
```

To append an element to the queue, you call **write()** on the **ringBuffer**. This increments the **write** pointer by one.

Since the queue has a fixed size, you must now return **true** or **false** to indicate whether the element has been successfully added. **enqueue()** is still an $O(1)$ operation.

Dequeue

To remove an item from the front of the queue, add the following:

```
override fun dequeue(): T? =  
    if (isEmpty) null else ringBuffer.read()
```

This code checks if the queue is empty and, if so, returns `null`. If not, it returns an item from the front of the buffer. Behind the scenes, the ring buffer increments the `read` pointer by one.

Debug and test

To easily see the contents of your buffer during debugging, add the following to `RingBufferQueue`:

```
override fun toString(): String = ringBuffer.toString()
```

This code creates a string representation of the queue by delegating to the underlying ring buffer.

That's all there is to it. Test your ring buffer-based queue by adding the following at the bottom of `Main.kt`, inside `main()`:

```
"Queue with Ring Buffer" example {  
    val queue = RingBufferQueue<String>(10).apply {  
        enqueue("Ray")  
        enqueue("Brian")  
        enqueue("Eric")  
    }  
    println(queue)  
    queue.dequeue()  
    println(queue)  
    println("Next up: ${queue.peek()}")  
}
```

This test code works like the previous examples dequeuing Ray and peeking at Brian.

Strengths and weaknesses

How does the ring-buffer implementation compare? Let's look at a summary of the algorithmic and storage complexity.

Ring-Buffer Based Queue

Operations	Best case	Worse case
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
Space Complexity	O(n)	O(n)

The ring-buffer-based queue has the same time complexity for enqueue and dequeue as the linked-list implementation. The only difference is the space complexity. The ring buffer has a fixed size, which means that enqueue can fail.

So far, you've seen three implementations: an array, a doubly linked-list and a ring-buffer.

Although they appear to be eminently useful, you'll next look at a queue implemented using two stacks. You'll see how its spatial locality is far superior to the linked list. It also doesn't need a fixed size like a ring buffer.

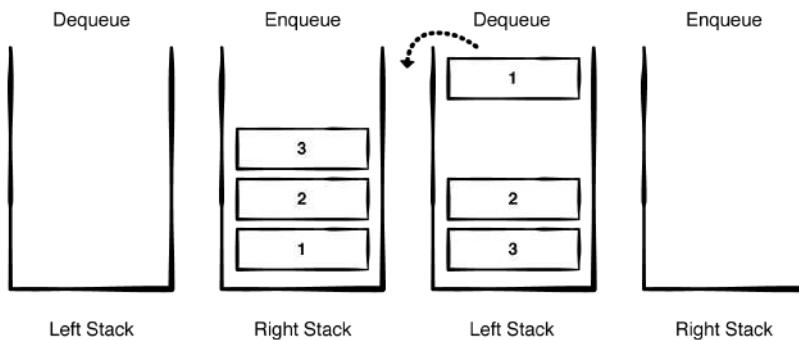
Double-stack implementation

Go to the **doublestack** package and start by adding a **StackQueue.kt** containing:

```
class StackQueue<T> : Queue<T> {
    private val leftStack = StackImpl<T>()
    private val rightStack = StackImpl<T>()
}
```

The idea behind using two stacks is simple. Whenever you enqueue an element, it goes in the **right** stack.

When you need to dequeue an element, you reverse the right stack and place it in the **left** stack so that you can retrieve the elements using FIFO order.



Leveraging the stacks

Implement the common features of a queue, starting with the following:

```
override val isEmpty: Boolean
    get() = leftStack.isEmpty && rightStack.isEmpty
```

To check if the queue is empty, simply check that both the left and right stack are empty. This means that there are no elements left to dequeue, and no new elements have been enqueued.

As you already know, there will be a time when you need to transfer the elements from the right stack into the left stack. That needs to happen whenever the left stack is empty. Add the following helper method:

```
private fun transferElements() {
    var nextElement = rightStack.pop()
    while (nextElement != null) {
        leftStack.push(nextElement)
        nextElement = rightStack.pop()
    }
}
```

With this code, you pop elements from the right stack and push them into the left stack. You already know from the previous chapter that stacks work in a LIFO way (last in, first out). You'll get them in reversed order without any additional work.

Next, add the following:

```
override fun peek(): T? {  
    if (leftStack.isEmpty) {  
        transferElements()  
    }  
    return leftStack.peek()  
}
```

You know that peeking looks at the top element. If the left stack is not empty, the element on top of this stack is at the front of the queue.

If the left stack is empty, you use `transferElements()`. That way, `leftStack.peek()` will always return the correct element or `null`. `isEmpty()` is still an $O(1)$ operation, while `peek()` is $O(n)$.

While this `peek()` implementation might seem expensive, it's amortized to $O(1)$ because each element in the queue only has to be moved from the right stack to the left stack once. If you have a lot of elements in the right stack, calling `peek()` will be $O(n)$ for just that one call when it has to move all of those elements. Any further calls will be $O(1)$ again.

Note: You could also make `peak()` operations precisely $O(1)$ for all calls if you implemented a method in `Stack` that let you look at the very bottom of the right stack. That's where the first item of the queue is if they're not all in the left stack, which is what `peak()` should return in that case.

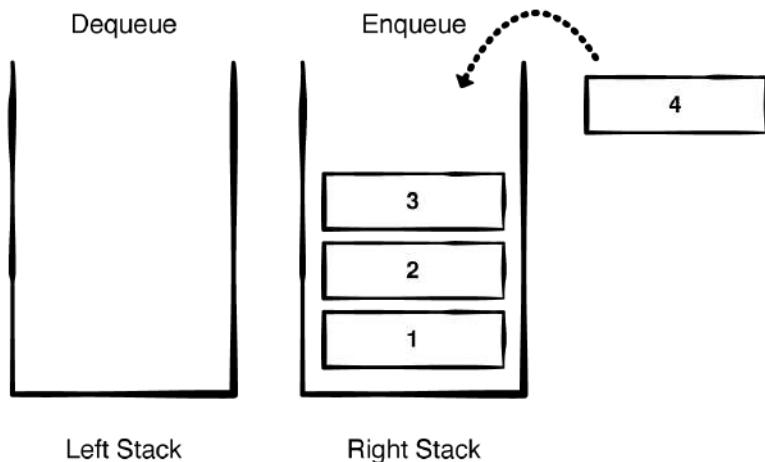
Enqueue

Next, add the method below:

```
override fun enqueue(element: T): Boolean {  
    rightStack.push(element)  
    return true  
}
```

Recall that the **right** stack is used to enqueue elements.

Previously, from implementing Stack, you know that pushing an element onto it is an $O(1)$ operation.



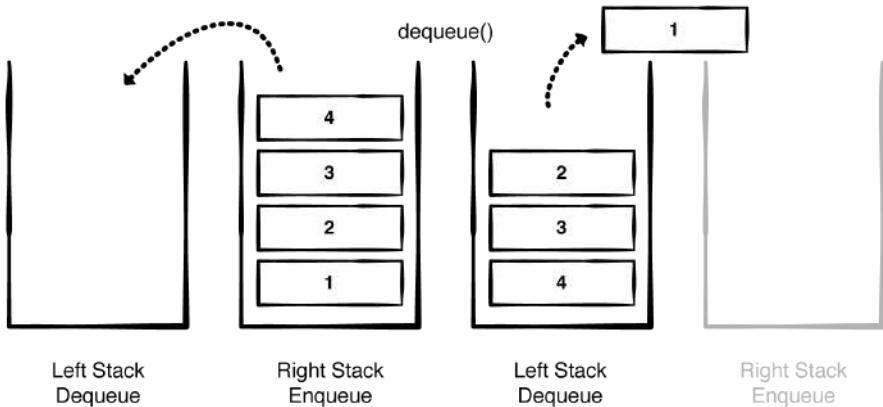
Dequeue

Removing an item from a two-stack-based implementation is as tricky as peeking. Add the following method:

```
override fun dequeue(): T? {  
    if (leftStack.isEmpty) { // 1  
        transferElements() // 2  
    }  
    return leftStack.pop() // 3  
}
```

Here's how it works:

1. Check to see if the left stack is empty.
2. If the left stack is empty, you need to transfer the elements from the right stack in reversed order.



- Remove the top element from the left stack.

Remember, you only transfer the elements in the right stack when the left stack is empty. This makes `dequeue()` an amortized $O(1)$ operation, just like `peek()`.

Debug and test

To see your results, add the following to `StackQueue`:

```
override fun toString(): String {
    return "Left stack: \n$leftStack \n Right stack:
\n$rightStack"
}
```

Here, you print the contents of the two stacks that represent your queue.

To try out the double-stack implementation, add the following to `main()`:

```
"Queue with Double Stack" example {
    val queue = StackQueue<String>().apply {
        enqueue("Ray")
        enqueue("Brian")
        enqueue("Eric")
    }
    println(queue)
    queue.dequeue()
    println(queue)
    println("Next up: ${queue.peek()}")
}
```

Similar to the previous examples, this code enqueues Ray, Brian and Eric, dequeues Ray and then peeks at Brian. Note how Eric and Brian ended up in the left stack and in reverse order as the result of the dequeue operation.

Strengths and weaknesses

Here's a summary of the algorithmic and storage complexity of your two-stack-based implementation.

Double Stack Based Queue

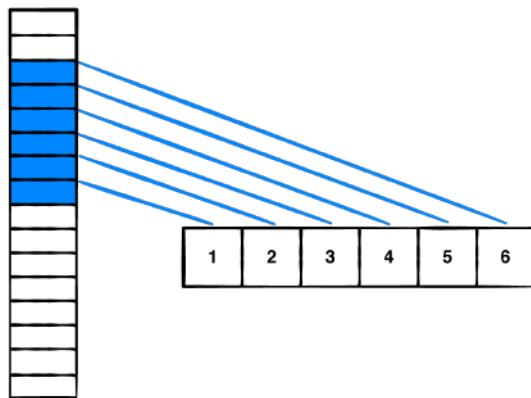
Operations	Best case	Worse case
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
Space Complexity	O(n)	O(n)

Compared to the list-based implementation, by leveraging two stacks, you were able to transform `dequeue()` into an amortized $O(1)$ operation.

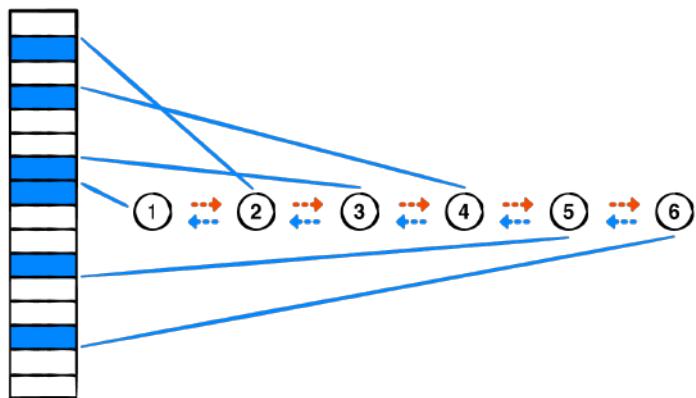
Moreover, your two-stack implementation is fully dynamic and doesn't have the fixed size restriction that your ring-buffer-based queue implementation has.

Finally, it beats the linked list in terms of spatial locality. This is because list elements are next to each other in memory blocks. So a large number of elements will be loaded in a cache on first access.

Compare the two images on the following page; one has elements in a contiguous array, the other has elements scattered all over memory:



Elements in a contiguous array.



Elements in a linked list, scattered all over memory.

In a linked list, elements aren't in contiguous blocks of memory. This could lead to more cache misses, which will increase access time.

Challenges

Think you have a handle on queues? In this section, you'll explore five different problems related to queues. This serves to solidify your fundamental knowledge of data structures in general.

Challenge 1: Explain differences

Explain the difference between a stack and a queue. Provide two real-life examples for each data structure.

Solution 1

Queues have a behavior of first in, first out. What comes in first must come out first. Items in the queue are inserted from the rear and removed from the front.

Queue Examples:

1. **Line in a movie theatre:** You would hate for people to cut the line at the movie theatre when buying tickets!
2. **Printer:** Multiple people could print documents from a printer, in a similar first-come-first-serve manner.

Stacks have a behavior of last-in-first-out. Items on the stack are inserted at the top and removed from the top.

Stack Examples:

1. **Stack of plates:** Placing plates on top of each other, and removing the top plate every time you use a plate. Isn't this easier than grabbing the one at the bottom?
2. **Undo functionality:** Imagine typing words on a keyboard. Most of the times, you would use undo for the last operation.

Challenge 2: What's the order?

Given the following queue:



Provide step-by-step diagrams showing how the following series of commands affects the queue:

```
enqueue("R")
enqueue("O")
dequeue()
enqueue("C")
dequeue()
dequeue()
enqueue("K")
}
```

Do this for the following queue implementations:

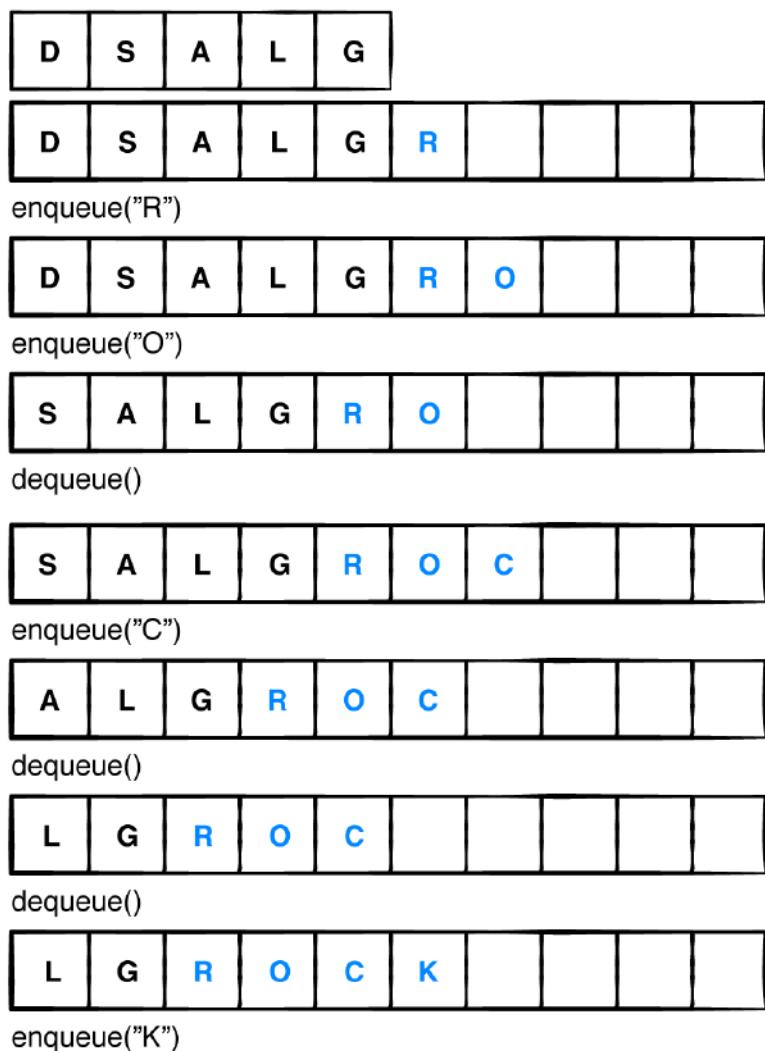
1. ArrayList
2. Linked list
3. Ring buffer
4. Double stack

Assume that the list and ring buffer have an initial size of 5.

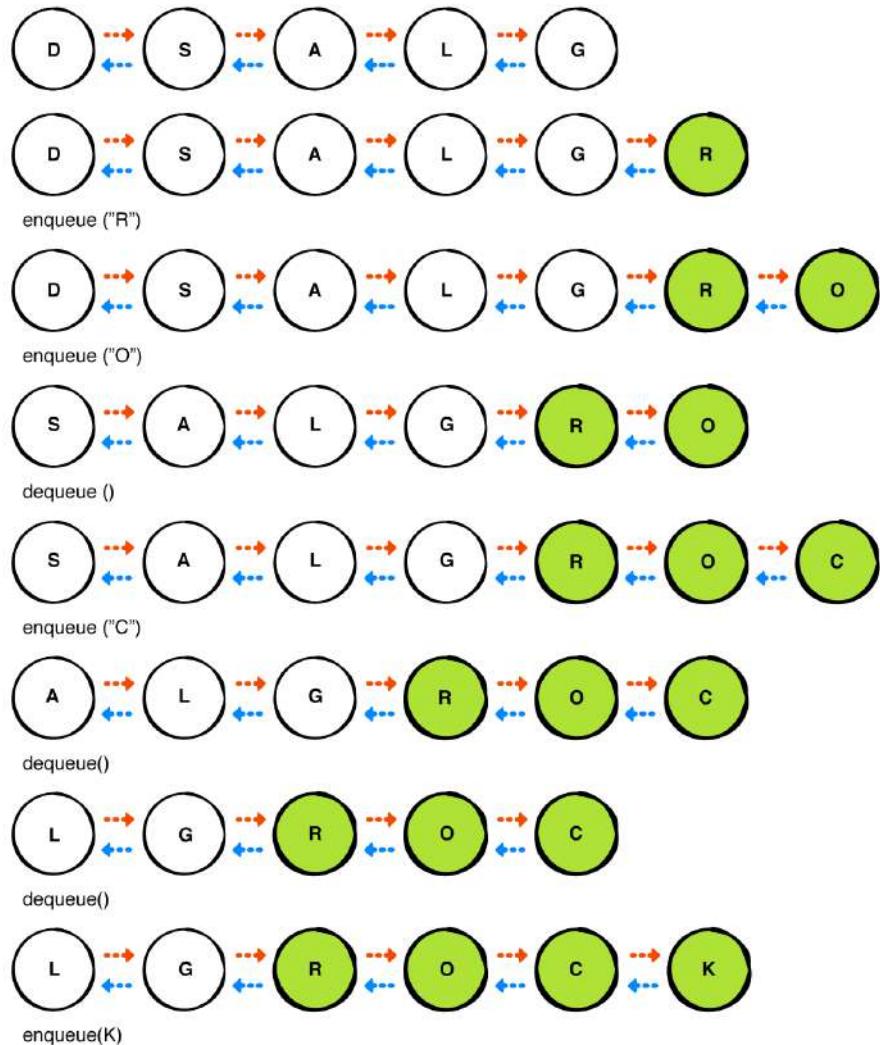
Solution 2

ArrayList

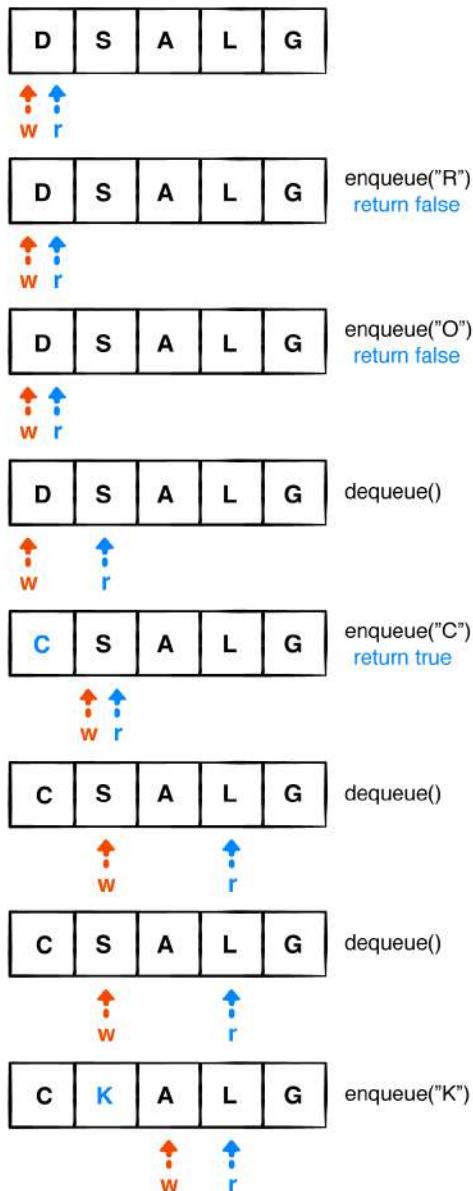
Keep in mind that whenever the underlying array is full, and you try to add a new element, a new array will be created with **twice** the capacity with existing elements being copied over.



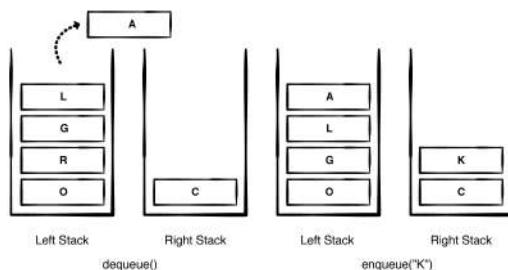
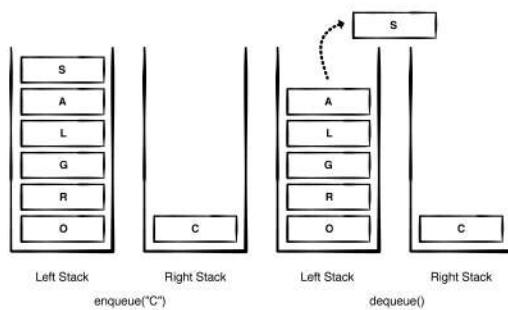
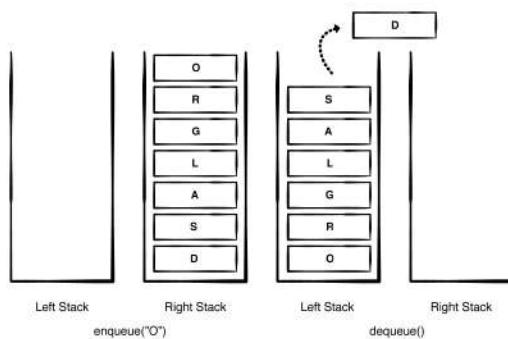
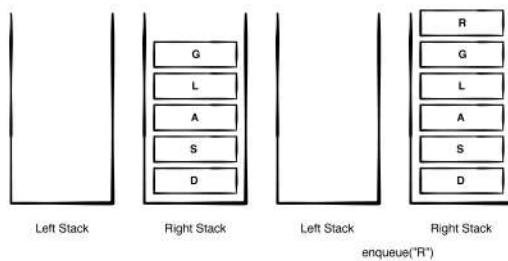
Linked list



Ring buffer



Double stack



Challenge 3: Monopoly

Imagine you're playing a game of Monopoly with your friends. The problem is that everyone always forgets whose turn it is! Create a Monopoly organizer that tells you whose turn it is. A great option is to create an extension function for Queue that always returns the next player. Here's how the definition could look:

```
fun <T> Queue<T>.nextPlayer(): T?
```

Solution 3

Creating a board game manager is straightforward. Your primary concern is whose turn it is. A queue data structure is a perfect choice to take care of game turns.

```
fun <T> Queue<T>.nextPlayer(): T? {  
    // 1  
    val person = this.dequeue() ?: return null  
    // 2  
    this.enqueue(person)  
    // 3  
    return person  
}
```

Here's how this works:

1. Get the next player by calling `dequeue`. If the queue is empty, return `null`, as the game has probably ended anyway.
2. `enqueue` the same person, this puts the player at the end of the queue.
3. Return the next player.

The time complexity depends on the queue implementation you select. For the array-based queue, it's overall $O(n)$ time complexity. `dequeue` takes $O(n)$ time because it has to shift the elements to the left every time you remove the first element.

Testing it out:

```
"Boardgame manager with Queue" example {  
    val queue = ArrayListQueue<String>().apply {  
        enqueue("Vincent")  
        enqueue("Remel")  
        enqueue("Lukiah")  
        enqueue("Allison")  
    }  
    println(queue)
```

```
println("===== boardgame ======")
queue.nextPlayer()
println(queue)
queue.nextPlayer()
println(queue)
queue.nextPlayer()
println(queue)
queue.nextPlayer()
println(queue)
}
```

Challenge 4: Reverse data

Implement a method to reverse the contents of a queue using an extension function.

Hint: The Stack data structure has been included in the project.

```
fun <T> Queue<T>.reverse()
```

Solution 4

A queue uses first in, first out whereas a stack uses last in, first out. You can use a stack to help reverse the contents of a queue. By inserting all of the contents of the queue into a stack, you basically reverse the order once you pop every element off the stack.

```
fun <T> Queue<T>.reverse() {
    // 1
    val aux = StackImpl<T>()

    // 2
    var next = this.dequeue()
    while (next != null) {
        aux.push(next)
        next = this.dequeue()
    }

    // 3
    next = aux.pop()
    while (next != null) {
        this.enqueue(next)
        next = aux.pop()
    }
}
```

For this solution, you added an extension function for any Queue implementation. It works the following way:

1. Create a stack.
2. dequeue all of the elements in the queue onto the stack.
3. pop all of the elements off the stack and insert them into the queue.
4. Return your reversed queue.

The time complexity is overall $O(n)$. You loop through the elements twice. Once for

removing the elements off the queue, and once for removing the elements off the stack.

Testing it out:

```
"Reverse queue" example {
    val queue = ArrayListQueue<String>().apply {
        enqueue("1")
        enqueue("21")
        enqueue("18")
        enqueue("42")
    }
    println("before: $queue")
    queue.reverse()
    println("after: $queue")
}
```

Key points

- Queue takes a FIFO strategy, an element added first must also be removed first.
- Enqueue inserts an element to the back of the queue.
- Dequeue removes the element at the front of the queue.
- Elements in an array are laid out in contiguous memory blocks, whereas elements in a linked list are more scattered with potential for cache misses.
- A ring buffer based queue implementation is useful for queues with a fixed size.
- Compared to other data structures, leveraging two stacks improves the dequeue() time complexity to an amortized $O(1)$ operation.
- The double-stack implementation beats linked list in terms of spatial locality.

Section III: Trees

Trees are another way to organize information, introducing the concept of children and parents. You'll look at the most common tree types and see how they can be used to solve specific computational problems.

The tree structures you'll learn about in this section include:

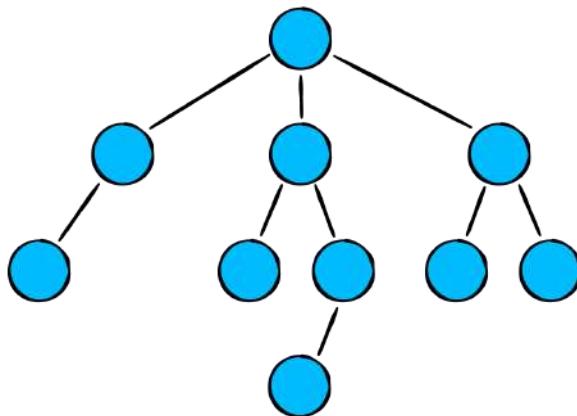
- **Chapter 6: Trees:** The tree is a data structure of profound importance. It's used to tackle many recurring challenges in software development such as representing hierarchical relationships, managing sorted data and facilitating fast lookup operations. There are many types of trees, and they come in various shapes and sizes.
- **Chapter 7: Binary Trees:** In the previous chapter, you looked at a basic tree where each node can have many children. A binary tree is a tree where each node has at most two children, often referred to as the left and right children. Binary trees serve as the basis for many tree structures and algorithms. In this chapter, you'll build a binary tree and learn about the three most important tree traversal algorithms.
- **Chapter 8: Binary Search Trees:** A binary search tree facilitates fast lookup, addition and removal operations. Each operation has an average time complexity of $O(\log n)$, which is considerably faster than linear data structures such as arrays and linked lists.
- **Chapter 9: AVL Trees:** In the previous chapter, you learned about the $O(\log n)$ performance characteristics of the binary search tree. However, you also learned that unbalanced trees can deteriorate the performance of the tree, all the way down to $O(n)$. In 1962, Georgy Adelson-Velsky and Evgenii Landis came up with the first self-balancing binary search tree: the AVL Tree.
- **Chapter 10: Tries.** The trie (pronounced as “try”) is a tree that specializes in storing data that can be represented as a collection, such as English words. The benefits of a trie are best illustrated by looking at it in the context of prefix matching, which is what you'll do in this chapter.

- **Chapter 11: Binary Search:** Binary search is one of the most efficient searching algorithms with a time complexity of $O(\log n)$. This is comparable with searching for an element inside a balanced binary search tree. To perform a binary search, the collection must be able to perform index manipulation in constant time and must be sorted.
- **Chapter 12: The Heap Data Structure:** A heap is a complete binary tree, also known as a binary heap, that can be constructed using an array. Heaps come in two flavors: Max heaps and Min heaps. Have you seen the movie *Toy Story* with the claw machine and the squeaky little green aliens? Imagine that the claw machine is operating on your heap structure and will always pick the minimum or maximum value depending on the flavor of heap.
- **Chapter 13: Priority Queues:** Queues are lists that maintain the order of elements using first in, first out (FIFO) ordering. A priority queue is another version of a queue that, instead of using FIFO ordering, dequeues elements in priority order. A priority queue is especially useful when you need to identify the maximum or minimum value given a list of elements.

Trees are a useful way to organize information when performance is critical. Adding them to your toolbelt will undoubtedly prove to be useful throughout your career.

Chapter 6: Trees

By Irina Galata



A tree

The **tree** is a data structure of profound importance. It's used to tackle many recurring challenges in software development, such as:

- Representing hierarchical relationships.
- Managing sorted data.
- Facilitating fast lookup operations.

There are many types of trees, and they come in various shapes and sizes. In this chapter, you'll learn the basics of using and implementing a tree.

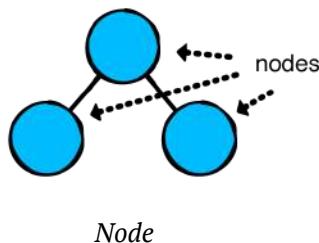


Terminology

There are many terms associated with trees, so it makes sense to get familiar with a few of them before starting.

Node

Like the linked list, trees are made up of **nodes**.



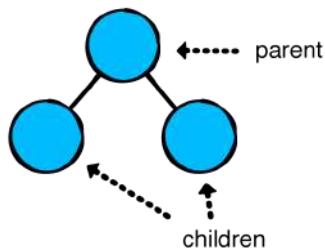
Node

Each node encapsulates some data and keeps track of its **children**.

Parent and child

Trees are viewed starting from the top and branching toward the bottom — just like a real tree, only upside-down.

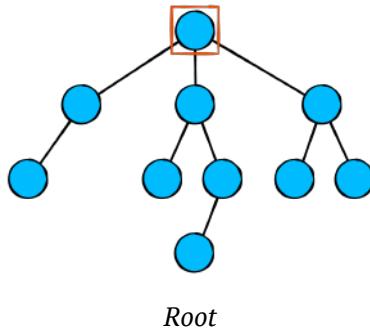
Every node, except for the first one, is connected to a single node above, which is referred to as a **parent** node. The nodes directly below and connected to the parent node are known as **child** nodes. In a tree, every child has exactly one parent. That's what makes a tree, well, a tree.



Parent and child

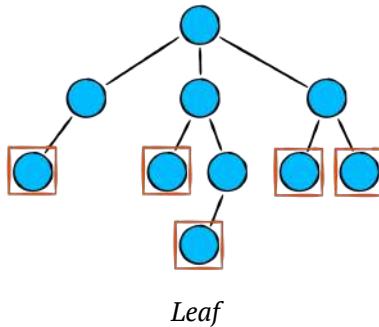
Root

The topmost node in the tree is called the **root** of the tree. It's the only node that has no parent:



Leaf

A node that has no children is called a **leaf**:



You'll run into more terms later, but this should be enough to start coding trees.

Implementation

To get started, open the starter project for this chapter.

A tree is made up of nodes, so your first task is to create a `TreeNode` class.

Create a new file named `TreeNode.kt` and add the following:

```
class TreeNode<T>(val value: T) {  
    private val children: MutableList<TreeNode<T>> =
```

```
    mutableListOf()
}
```

Each node is responsible for a value and holds references to all of its children using a mutable list.

Next, add the following method inside `TreeNode`:

```
fun add(child: TreeNode<T>) = children.add(child)
```

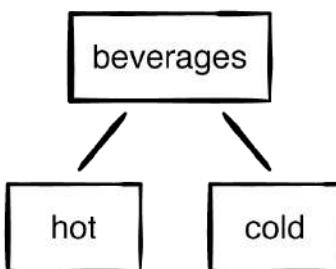
This method adds a child node to a node.

Time to give it a whirl. Go to the `main()` in the `Main.kt` file and add the following:

```
fun main() {
    val hot = TreeNode("Hot")
    val cold = TreeNode("Cold")

    val beverages = TreeNode("Beverages").run {
        add(hot)
        add(cold)
    }
}
```

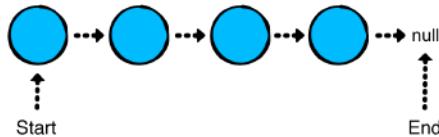
Hierarchical structures are natural candidates for tree structures. That being the case, you define three different nodes and organize them into a logical hierarchy. This arrangement corresponds to the following structure:



A small tree

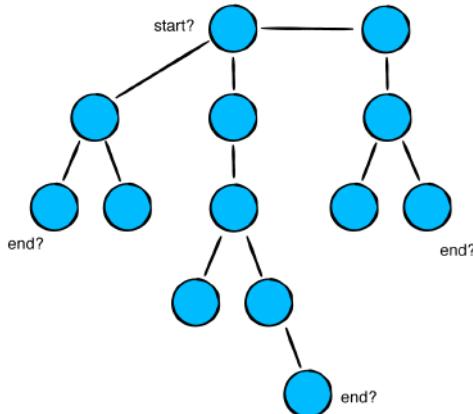
Traversal algorithms

Iterating through **linear collections** such as arrays or lists is straightforward. Linear collections have a clear start and end:



Traversing arrays or lists

Iterating through trees is a bit more complicated:



Traversing trees

Should nodes on the left have precedence? How should the depth of a node relate to its precedence? Your traversal strategy depends on the problem you're trying to solve.

There are multiple strategies for different trees and different problems. In all of these ways you can **visit** the node and use the information into them. This is why you add this definition into the `TreeNode.kt` file outside of the `TreeNode` class definition.

```
typealias Visitor<T> = (TreeNode<T>) -> Unit
```

In the next section, you'll look at **depth-first traversal**.

Depth-first traversal

Depth-first traversal starts at the root node and explores the tree as far as possible along each branch before reaching a leaf and then backtracking.

Add the following inside `TreeNode`:

```
fun forEachDepthFirst(visit: Visitor<T>) {
    visit(this)
    children.forEach {
        it.forEachDepthFirst(visit)
    }
}
```

This simple code uses recursion to process the next node.

You could use your own stack if you didn't want your implementation to be recursive. However, the recursive solution is more simple and elegant to code.

To test the recursive depth-first traversal function you just wrote, it's helpful to add more nodes to the tree. Go back to the playground page and add the following in `Main.kt`:

```
fun makeBeverageTree(): TreeNode<String> {
    val tree = TreeNode("Beverages")

    val hot = TreeNode("hot")
    val cold = TreeNode("cold")

    val tea = TreeNode("tea")
    val coffee = TreeNode("coffee")
    val chocolate = TreeNode("cocoa")

    val blackTea = TreeNode("black")
    val greenTea = TreeNode("green")
    val chaiTea = TreeNode("chai")

    val soda = TreeNode("soda")
    val milk = TreeNode("milk")

    val gingerAle = TreeNode("ginger ale")
    val bitterLemon = TreeNode("bitter lemon")

    tree.add(hot)
    tree.add(cold)

    hot.add(tea)
    hot.add(coffee)
    hot.add(chocolate)
```

```

    cold.add(soda)
    cold.add(milk)

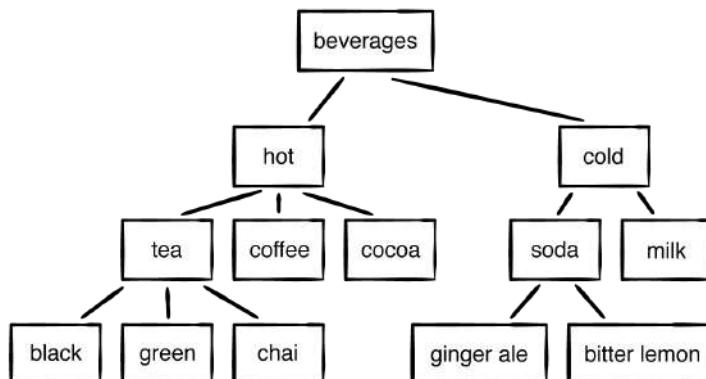
    tea.add(blackTea)
    tea.add(greenTea)
    tea.add(chaiTea)

    soda.add(gingerAle)
    soda.add(bitterLemon)

    return tree
}

```

This function creates the following tree:



A big tree

Next, replace the code in `main()` with the following:

```

fun main() {
    val tree = makeBeverageTree()
    tree.forEachDepthFirst { println(it.value) }
}

```

This code produces the following output that illustrates the order the depth-first traversal visits each node:

```

Beverages
hot
tea
black
green
chai
coffee
cocoa

```

```
cold
soda
ginger ale
bitter lemon
milk
```

In the next section, you'll look at **level-order traversal**.

Level-order traversal

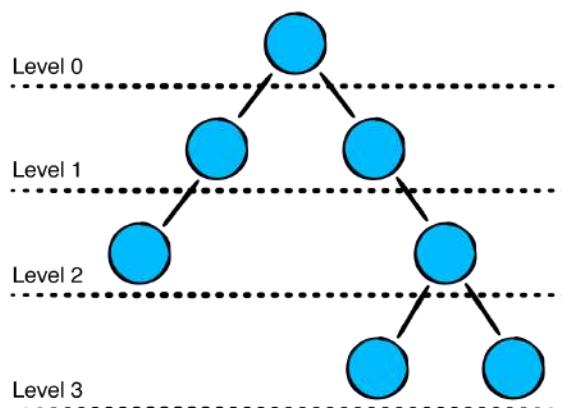
Level-order traversal is a technique that visits each node of the tree based on the depth of the nodes. Starting at the root, every node on a level is visited before going to a lower level.

Add the following inside `TreeNode`:

```
fun forEachLevelOrder(visit: Visitor<T>) {
    visit(this)
    val queue = Queue<TreeNode<T>>()
    children.forEach { queue.enqueue(it) }

    var node = queue.dequeue()
    while (node != null) {
        visit(node)
        node.children.forEach { queue.enqueue(it) }
        node = queue.dequeue()
    }
}
```

`forEachLevelOrder` visits each of the nodes in level-order:



Level-order traversal

Note how you use a queue to ensure that nodes are visited in the right level-order. You start visiting the current node and putting all its children into the queue. Then you start consuming the queue until it's empty. Every time you visit a node, you also put all its children into the queue. This ensures that all nodes at the same level are visited one after the other.

Open **Main.kt** and add the following:

```
fun main() {
    val tree = makeBeverageTree()
    tree.forEachLevelOrder { println(it.value) }
}
```

In the console, you'll see the following output:

```
beverages
hot
cold
tea
coffee
cocoa
soda
milk
black
green
chai
ginger ale
bitter lemon
```

Search

You already have a method that iterates through the nodes, so building a search algorithm won't take long.

Add the following inside `TreeNode`:

```
fun search(value: T): TreeNode<T>? {
    var result: TreeNode<T>? = null

    forEachLevelOrder {
        if (it.value == value) {
            result = it
        }
    }

    return result
}
```

To test your code, go back to `main()`. To save some time, copy the previous example and modify it to test the `search` method:

```
fun main() {
    val tree = makeBeverageTree()
    tree.search("ginger ale")?.let {
        println("Found node: ${it.value}")
    }

    tree.search("WKD Blue")?.let {
        println(it.value)
    } ?: println("Couldn't find WKD Blue")
}
```

You'll see the following console output:

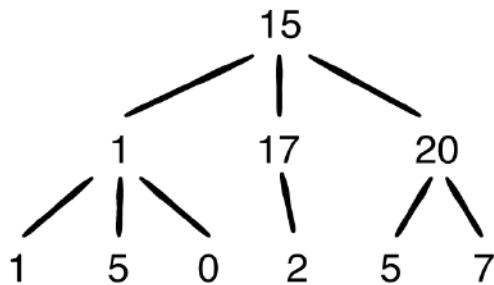
```
Found node: ginger ale
Couldn't find WKD Blue
```

Here, you used your level-order traversal algorithm. Since it visits all nodes, if there are multiple matches, the last match wins. This means that you'll get different objects back depending on what traversal you use.

Challenges

Challenge 1: Tree challenge

Print the values in a tree in an order based on their level. Nodes belonging to the same level should be printed on the same line. For example, consider the following tree:



Your algorithm should output the following in the console:

```
15
1 17 20
1 5 0 2 5 7
```

Hint: Consider using a Queue included for you in the starter project.

Solution 1

A straightforward way to print the nodes in level-order is to leverage the level-order traversal using a Queue data structure. The tricky bit is determining when a newline should occur.

Here's the solution:

```
fun printEachLevel() {
    // 1
    val queue = ArrayListQueue<TreeNode<T>>()
    var nodesLeftInCurrentLevel = 0
```

```
queue.enqueue(this)
// 2
while (queue.isEmpty.not()) {
    // 3
    nodesLeftInCurrentLevel = queue.count

    // 4
    while (nodesLeftInCurrentLevel > 0) {
        val node = queue.dequeue()
        node?.let {
            print("${node.value} ")
            node.children.forEach { queue.enqueue(it) }
            nodesLeftInCurrentLevel--
        } ?: break
    }

    // 5
    println()
}
}
```

And here's how it works:

1. You begin by initializing a Queue data structure to facilitate the level-order traversal. You also create `nodesLeftInCurrentLevel` to keep track of the number of nodes you'll need to work on before you print a new line.
2. Your level-order traversal continues until your queue is empty.
3. Inside the first `while` loop, you begin by setting `nodesLeftInCurrentLevel` to the current elements in the queue.
4. Using another `while` loop, you dequeue the first `nodesLeftInCurrentLevel` number of elements from the queue. Every element you dequeue is printed *without* establishing a new line. You also enqueue all the children of the node.
5. At this point, you generate the new line using `println()`. In the next iteration, `nodesLeftInCurrentLevel` is updated with the count of the queue, representing the number of children from the previous iteration.

This algorithm has a time complexity of $O(n)$. Since you initialize the Queue data structure as an intermediary container, this algorithm also uses $O(n)$ space.

Key points

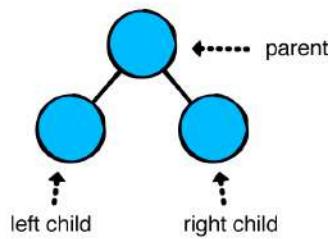
- Trees share some similarities to linked lists. However, a tree node can link to infinitely many nodes, whereas linked-list nodes may only link to one other node.
- Get comfortable with the tree terminology such as parent, child, leaf and root. Many of these terms are common and are used to help explain other tree structures.
- Traversals, such as depth-first and level-order traversals, aren't specific to only the general type of tree. They work on other types of trees as well, although their implementation is slightly different based on how the tree is structured.
- Trees are a fundamental data structure with different implementations. Some of these will be part of the next chapters.

7

Chapter 7: Binary Trees

By Irina Galata

In the previous chapter, you looked at a basic tree in which each node can have many children. A **binary tree** is a tree in which each node has at most **two** children, often referred to as the **left** and **right** children:



Binary Tree

Binary trees serve as the basis for many tree structures and algorithms. In this chapter, you'll build a binary tree and learn about the three most important tree traversal algorithms.



Implementation

Open the starter project for this chapter. Create a new file and name it **BinaryNode.kt**. You also define the `Visitor<T>` typealias. Add the following inside this file:

```
typealias Visitor<T> = (T) -> Unit

class BinaryNode<T>(val value: T) {

    var leftChild: BinaryNode<T>? = null
    var rightChild: BinaryNode<T>? = null

}
```

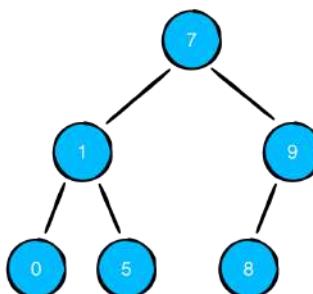
In `main()` in the **Main.kt** file, add the following:

```
fun main() {
    val zero = BinaryNode(0)
    val one = BinaryNode(1)
    val five = BinaryNode(5)
    val seven = BinaryNode(7)
    val eight = BinaryNode(8)
    val nine = BinaryNode(9)

    seven.leftChild = one
    one.leftChild = zero
    one.rightChild = five
    seven.rightChild = nine
    nine.leftChild = eight

    val tree = seven
}
```

This defines the following tree by executing the closure:



Example Binary Tree

Building a diagram

Building a mental model of a data structure can be quite helpful in learning how it works. To that end, you'll implement a reusable algorithm that helps visualize a binary tree in the console.

Note: This algorithm is based on an implementation by Károly Lőrentey in his book *Optimizing Collections*, available from <https://www.objc.io/books/optimizing-collections/>.

Add the following to the bottom of **BinaryNode.kt**:

```
override fun toString() = diagram(this)

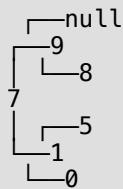
private fun diagram(node: BinaryNode<T>?,
                    top: String = "",
                    root: String = "",
                    bottom: String = ""): String {
    return node?.let {
        if (node.leftChild == null && node.rightChild == null) {
            "$root${node.value}\n"
        } else {
            diagram(node.rightChild, "$top ", "$top\u25bc", "$top| ") +
                root + "${node.value}\n" + diagram(node.leftChild,
                "$bottom| ", "$bottom\u25bc", "$bottom ")
        }
    } ?: "${root}null\n"
}
```

This method recursively creates a string representing the binary tree.

To try it out, open **main.kt** and add the following:

```
println(tree)
```

You'll see the following console output:



You'll use this diagram for other binary trees in this book.

Traversal algorithms

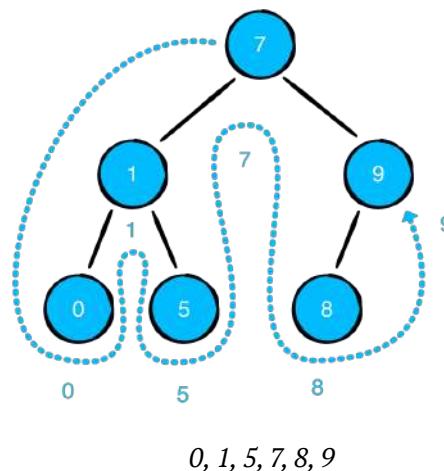
Previously, you looked at a level-order traversal of a tree. With a few tweaks, you can make this algorithm work for binary trees as well. However, instead of re-implementing level-order traversal, you'll look at three traversal algorithms for binary trees: **in-order**, **pre-order** and **post-order** traversals.

In-order traversal

In-order traversal visits the nodes of a binary tree in the following order, starting from the root node:

- If the current node has a left child, recursively visit this child first.
- Then visit the node itself.
- If the current node has a right child, recursively visit this child.

Here's what an in-order traversal looks like for your example tree:



You may have noticed that this prints the example tree in ascending order. If the tree nodes are structured in a certain way, in-order traversal visits them in ascending order. You'll learn more about binary search trees in the next chapter.

Open **BinaryNode.kt** and add the following code to the bottom of the file:

```
fun traverseInOrder(visit: Visitor<T>) {  
    leftChild?.traverseInOrder(visit)  
    visit(value)
```

```
    rightChild?.traverseInOrder(visit)  
}
```

Following the rules laid out above, you first traverse to the left-most node before visiting the value. You then traverse to the right-most node.

To test this, go to `main()`, and add the following at the bottom:

```
tree.traverseInOrder { println(it) }
```

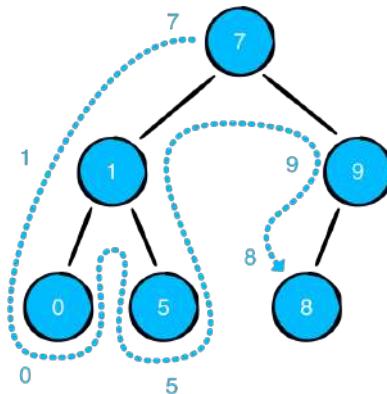
You should see the following in the console:

```
0  
1  
5  
7  
8  
9
```

Pre-order traversal

Pre-order traversal visits the nodes of a binary tree in the following order:

- Visits the current node first.
- Recursively visits the left and right child.



Pre-order traversal

Write the following immediately below the in-order traversal method:

```
fun traversePreOrder(visit: Visitor<T>) {  
    visit(value)  
    leftChild?.traversePreOrder(visit)  
    rightChild?.traversePreOrder(visit)  
}
```

Test it out with the following code in the `main` method:

```
tree.traversePreOrder { println(it) }
```

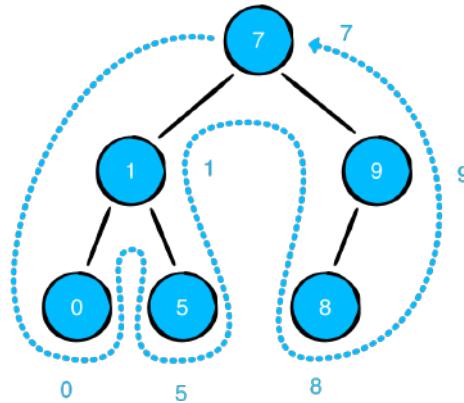
You'll see the following output in the console:

```
7  
1  
0  
5  
9  
8
```

Post-order traversal

Post-order traversal always visits the nodes of a binary tree in the following order:

- Recursively visits the left and right child.
- Only visits the current node after the left and right child have been visited recursively.



Post-order traversal

In other words, given any node, you'll visit its children before visiting itself. An interesting consequence of this is that the root node is always visited last.

Inside **BinaryNode.kt**, add the following below `traversePreOrder`:

```
fun traversePostOrder(visit: Visitor<T>) {  
    leftChild?.traversePostOrder(visit)  
    rightChild?.traversePostOrder(visit)  
    visit(value)  
}
```

Navigate back to `main()` and add the following to try it out:

```
tree.traversePostOrder { println(it) }
```

You'll see the following in the console:

```
0  
5  
1  
8  
9  
7
```

Each one of these traversal algorithms has both a time and space complexity of $O(n)$.

While this version of the binary tree isn't too exciting, you saw that you can use in-order traversal to visit the nodes in ascending order. Binary trees can enforce this behavior by adhering to some rules during insertion.

In the next chapter, you'll look at a binary tree with stricter semantics: the **binary search tree**.

Challenges

Binary trees are a surprisingly popular topic in algorithm interviews. Questions on the binary tree not only require a good foundation of how traversals work, but can also test your understanding of recursive backtracking. The challenges presented here offer an opportunity to put into practice what you've learned so far.

Open the starter project to begin these challenges.

Challenge 1: The height of the tree

Given a binary tree, find the height of the tree. The height of the binary tree is determined by the distance between the root and the furthest leaf. The height of a binary tree with a single node is zero since the single node is both the root and the furthest leaf.

Solution 1

A recursive approach for finding the height of a binary tree is as follows:

```
fun height(node: BinaryNode<T>? = this): Int {  
    return node?.let { 1 + max(height(node.leftChild),  
        height(node.rightChild)) } ?: -1  
}
```

You recursively call the height function. For every node you visit, you add one to the height of the highest child. If the node is `null`, you return `-1`.

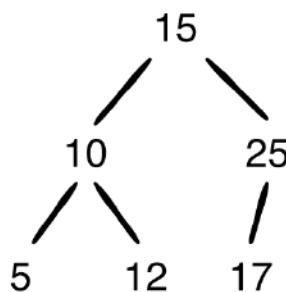
This algorithm has a time complexity of $O(n)$ since you need to traverse through all of the nodes. This algorithm incurs a space cost of $O(n)$ since you need to make the same n recursive calls to the call stack.

Challenge 2: Serialization of a Binary Tree

A common task in software development is serializing an object into another data type. This process is known as **serialization**, and it allows custom types to be used in systems that only support a closed set of data types.

An example of serialization is JSON. Your task is to devise a way to serialize a binary tree into a list, and a way to deserialize the list back into the same binary tree.

To clarify this problem, consider the following binary tree:



A particular algorithm may output the serialization as [15, 10, 5, null, null, 12, null, null, 25, 17, null, null, null]. The deserialization process should transform the list back into the same binary tree. Note that there are many ways to perform serialization. You may choose any way you wish.

Solution 2

There are many ways to serialize or deserialize a binary tree. Your first task when encountering this question is to decide on the traversal strategy.

For this solution, you'll explore how to solve this challenge by choosing the **pre-order** traversal strategy.

Traversal

Write the following code to **BinaryNode.kt**:

```
fun traversePreOrderWithNull(visit: Visitor<T>) {  
    visit(value)  
    leftChild?.traversePreOrderWithNull(visit) ?: visit(null)  
    rightChild?.traversePreOrderWithNull(visit) ?: visit(null)  
}
```

This is the pre-order traversal function. As the code suggests, pre-order traversal traverses each node and visit the node before traversing the children.

It's critical to point out that you'll need to also visit the `null` nodes since it's important to record those for serialization and deserialization.

As with all traversal functions, this algorithm goes through every element of the tree once, so it has a time complexity of $O(n)$.

Serialization

For serialization, you traverse the tree and store the values into an array. The elements of the array have type `T?` since you need to keep track of the `null` nodes. Add the following code to **BinaryNode.kt**:

```
fun serialize(node: BinaryNode<T> = this): MutableList<T?> {  
    val list = mutableListOf<T?>()  
    node.traversePreOrderWithNull { list.add(it) }  
    return list  
}
```

`serialize` returns a new array containing the values of the tree in pre-order.

The time complexity of the serialization step is $O(n)$. Because you're creating a new list, this also incurs an $O(n)$ space cost.

Deserialization

In the serialization process, you performed a pre-order traversal and assembled the values into an array. The deserialization process is to take each value of the array and reassemble it back to the tree.

Your goal is to iterate through the array and reassemble the tree in pre-order format. Write the following at the bottom of your playground page:

```
fun deserialize(list: MutableList<T?>): BinaryNode<T?>? {
    // 1
    val rootValue = list.removeAt(list.size - 1) ?: return null

    // 2
    val root = BinaryNode<T?>(rootValue)

    root.leftChild = deserialize(list)
    root.rightChild = deserialize(list)

    return root
}
```

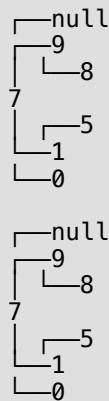
Here's how the code works:

1. This is the base case. If `removeAt` returns `null`, there are no more elements in the array, thus you'll end recursion here.
2. You reassemble the tree by creating a node from the current value and recursively calling `deserialize` to assign nodes to the left and right children. Notice this is very similar to the pre-order traversal, except, in this case, you're building nodes rather than extracting their values.

Your algorithm is now ready for testing. Write the following at the bottom of `main()`:

```
println(tree)
val array = tree.serialize()
println(tree.deserialize(array))
```

You'll see the following in your console:



Your deserialized tree mirrors the sample tree in the provided playground. This is the behavior you want.

However, as mentioned earlier, the time complexity of this function isn't desirable. Because you're calling `removeAt` as many times as there are elements in the array, this algorithm has an $O(n^2)$ time complexity. There's an easy way to remedy that.

Write the following function just after the `deserialize` function you created earlier:

```
fun deserializeOptimized(list: MutableList<T>): BinaryNode<T>?
{
    return deserialize(list.asReversed())
}
```

This is a function that first reverses the array before calling the previous `deserialize` function. In the other `deserialize` function, find the `removeAt(0)` call and change it to `list.removeAt(list.size - 1)`:

```
val rootValue = list.removeAt(list.size - 1) ?: return null
```

This small change has a big effect on performance. `removeAt(0)` is an $O(n)$ operation because, after every removal, every element after the removed element must shift left to take up the missing space. In contrast, `list.removeAt(list.size - 1)` is an $O(1)$ operation.

Finally, find and update the call of `deserialize` to use the new function that reverses the array:

```
println(tree.deserializeOptimized(array))
```

You'll see the same tree before and after the deserialization process. The time complexity for this solution is now $O(n)$ because you created a new reversed list and chose a recursive solution.

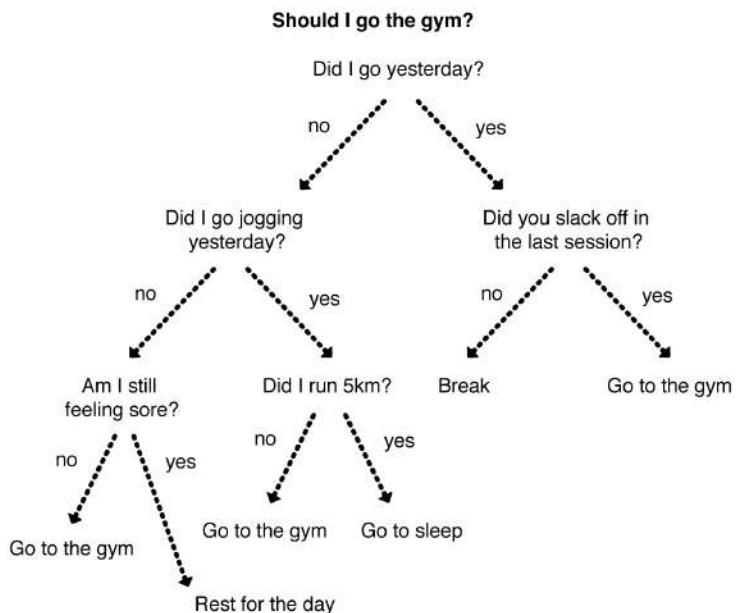
Key points

- The binary tree is the foundation to some of the most important tree structures. The binary search tree and AVL tree are binary trees that impose restrictions on the insertion/deletion behaviors.
- In-order, pre-order and post-order traversals aren't just important only for the binary tree; if you're processing data in any tree, you'll interface with these traversals regularly.

Chapter 8: Binary Search Trees

By Irina Galata

A **binary search tree**, or **BST**, is a data structure that facilitates fast lookup, insert and removal operations. Consider the following decision tree where picking a side forfeits all of the possibilities of the other side, cutting the problem in half.



Once you make a decision and choose a branch, there's no looking back. You keep going until you make a final decision at a leaf node. Binary trees let you do the same thing. Specifically, a binary search tree imposes two rules on the binary tree you saw in the previous chapter:

- The value of a **left child** must be less than the value of its **parent**.
- Consequently, the value of a **right child** must be greater than or equal to the value of its **parent**.

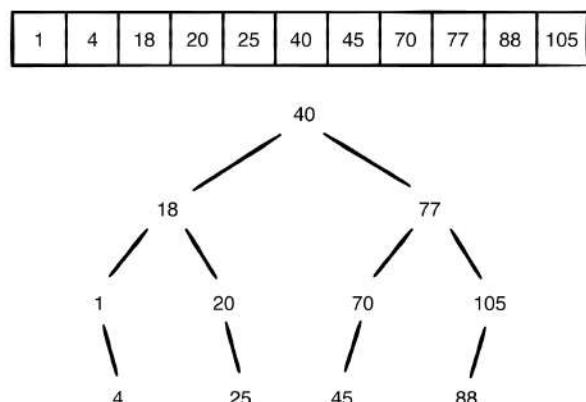
Binary search trees use this property to save you from performing unnecessary checking. As a result, lookup, insert and removal have an average time complexity of $O(\log n)$, which is considerably faster than linear data structures such as arrays and linked lists.

In this chapter, you'll learn about the benefits of the BST relative to an array and implement the data structure from scratch.

Case study: array vs. BST

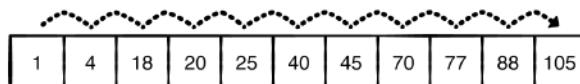
To illustrate the power of using a BST, you'll look at some common operations and compare the performance of arrays against the binary search tree.

Consider the following two collections:



Lookup

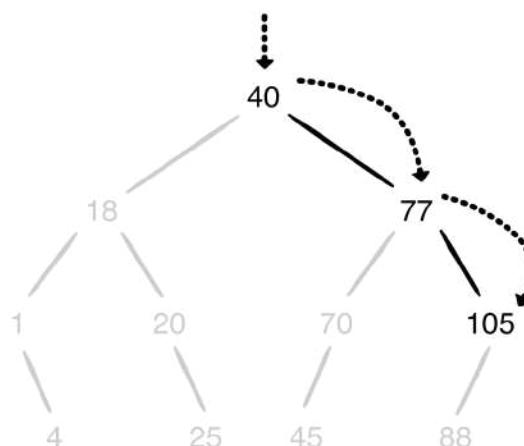
There's only one way to do element lookups for an unsorted array. You need to check every element in the array from the start.



Searching for 105

That's why `contains` is an $O(n)$ operation.

This is not the case for binary search trees.



Searching for 105

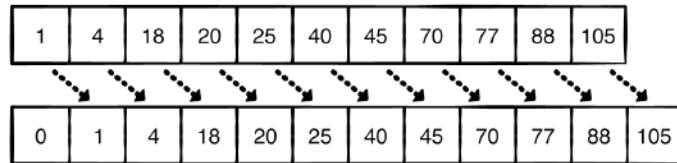
Every time the search algorithm visits a node in the BST, it can safely make these two assumptions:

- If the search value is **less than** the current value, it must be in the **left** subtree.
- If the search value is **greater than** the current value, it must be in the **right** subtree.

By leveraging the rules of the BST, you can avoid unnecessary checks and cut the search space in half every time you make a decision. That's why element lookup in a BST is an $O(\log n)$ operation.

Insertion

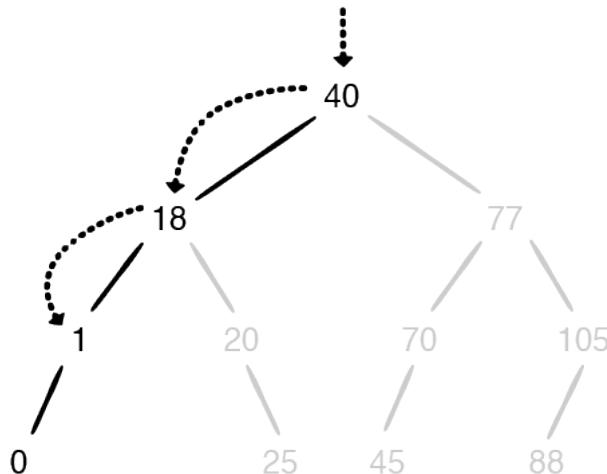
The performance benefits for the insertion operation follow a similar story. Assume you want to insert **0** into a collection.



Inserting 0 in sorted order

Inserting values into an array is like butting into an existing line: Everyone in the line behind your chosen spot needs to make space for you by shuffling back. In the above example, zero is inserted at the front of the array, causing all of the other elements to shift backward by one position. Inserting into an array has a time complexity of $O(n)$.

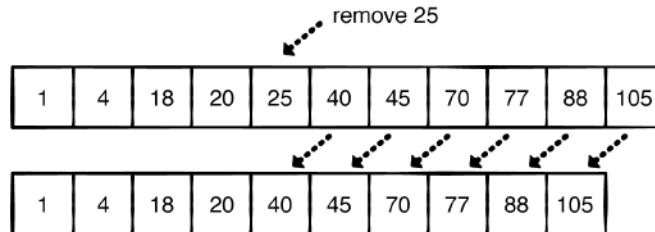
Insertion into a binary search tree is much more comforting.



By leveraging the rules for the BST, you only needed to make three hops to find the location for the insertion, and you didn't have to shuffle all of the elements around. Inserting elements in a BST is, again, an $O(\log n)$ operation.

Removal

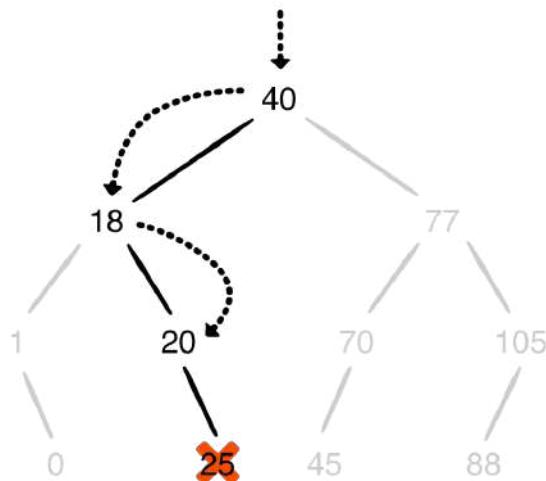
Similar to insertion, removing an element from an array also triggers a shuffling of elements.



Removing 25 from the array

This behavior also plays nicely with the line analogy. If you leave the middle of the line, everyone behind you needs to shuffle forward to take up the empty space.

Here's what removing a value from a BST looks like:



Nice and easy! There are complications to manage when the node you're removing has children, but you'll look into that later. Even with those complications, removing an element from a BST is still an $O(\log n)$ operation.

Binary search trees drastically reduce the number of steps for add, remove and lookup operations. Now that you have a grasp of the benefits of using a binary search tree, you can move on to the actual implementation.

Implementation

Open the starter project for this chapter. In it, you'll find the `BinaryNode` class you created in the previous chapter. Create a new file named `BinarySearchTree.kt` and add the following to it:

```
class BinarySearchTree<T: Comparable<T>>() {  
    var root: BinaryNode<T>? = null  
  
    override fun toString() = root?.toString() ?: "empty tree"  
}
```

By definition, binary search trees can only hold values that are `Comparable`.

Next, you'll look at the `insert` method.

Inserting elements

Following the rules of the BST, nodes of the left child must contain values less than the current node, whereas nodes of the right child must contain values greater than or equal to the current node. You'll implement `insert` while respecting these rules.

Add the following to `BinarySearchTree`:

```
fun insert(value: T) {  
    root = insert(root, value)  
}  
  
private fun insert(  
    node: BinaryNode<T>?,  
    value: T  
>: BinaryNode<T> {  
    // 1  
    node ?: return BinaryNode(value)  
    // 2  
    if (value < node.value) {  
        node.leftChild = insert(node.leftChild, value)  
    } else {  
        node.rightChild = insert(node.rightChild, value)  
    }  
}
```



```
// 3  
return node  
}
```

The first `insert` is exposed to users, while the second will be used as a private helper method:

1. This is a recursive method, so it requires a base case for terminating recursion. If the current node is `null`, you've found the insertion point and return the new `BinaryNode`.
2. This `if` statement controls which way the next `insert` call should traverse. If the new value is less than the current value, you call `insert` on the **left** child. If the new value is greater than or equal to the current value, you call `insert` on the **right** child.
3. Return the current node. This makes assignments of the form `node = insert(node, value)` possible as `insert` will either create `node` (if it was `null`) or return `node` (if it was not `null`).

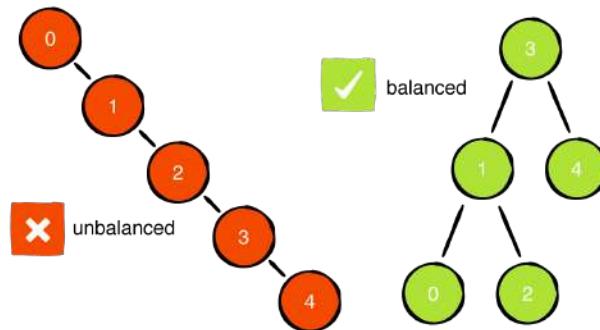
Go back to `main()` and add the following at the bottom:

```
"building a BST" example {  
    val bst = BinarySearchTree<Int>()  
    (0..4).forEach {  
        bst.insert(it)  
    }  
    println(bst)  
}
```

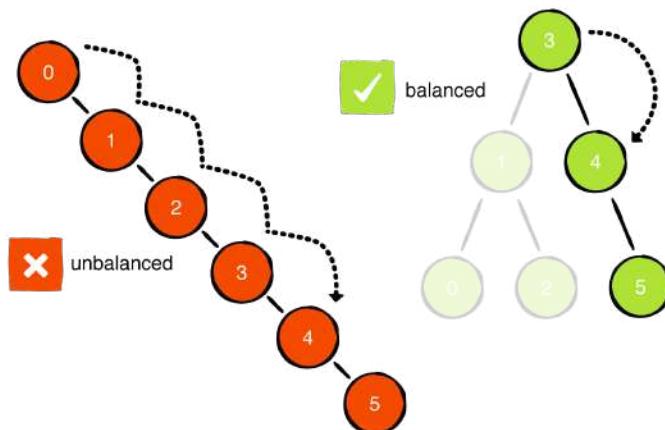
You'll see the following output:

```
---Example of building a BST---  
    4  
   / \  
   3   null  
  / \ / \  
 2   1   null  
 / \ / \ / \  
1   0   null  
/ \ / \ / \ / \  
0   null
```

That tree looks a bit *unbalanced*, but it does follow the rules. However, this tree layout has undesirable consequences. When working with trees, you always want to achieve a balanced format.



An unbalanced tree affects performance. If you insert 5 into the unbalanced tree you created, it becomes an $O(n)$ operation.



You can create structures known as *self-balancing trees* that use clever techniques to maintain a balanced structure, but we'll save those details for the next chapter. For now, you'll simply build a sample tree with a bit of care to keep it from becoming unbalanced.

Add the following variable at the start of `main()`:

```
val exampleTree = BinarySearchTree<Int>().apply {  
    insert(3)  
    insert(1)
```

```
insert(4)
insert(0)
insert(2)
insert(5)
}
```

Then, update the previous example with this code:

```
"building a BST" example {
    println(exampleTree)
}
```

You'll see the following in the console:

```
---Example of building a BST---
      5
     / \
    4   null
   / \
  3   2
   / \
  1   0
```

Much nicer!

Finding elements

Finding an element in a BST requires you to traverse through its nodes. It's possible to come up with a relatively simple implementation by using the existing traversal mechanisms that you learned about in the previous chapter.

Add the following to the bottom of `BinarySearchTree`:

```
fun contains(value: T): Boolean {
    root ?: return false

    var found = false
    root?.traverseInOrder {
        if (value == it) {
            found = true
        }
    }

    return found
}
```

Next, go back to `main()` to test this out:

```
"finding a node" example {
    if (exampleTree.contains(5)) {
        println("Found 5!")
    } else {
        println("Couldn't find 5")
    }
}
```

You'll see the following in the console:

```
---Example of finding a node---
Found 5!
```

In-order traversal has a time complexity of $O(n)$, thus this implementation of `contains` has the same time complexity as an exhaustive search through an unsorted array. However, you can do better!

Optimizing `contains`

You can rely on the rules of the BST to avoid needless comparisons. Inside `BinarySearchTree.kt`, update `contains` to the following:

```
fun contains(value: T): Boolean {
    // 1
    var current = root

    // 2
    while (current != null) {
        // 3
        if (current.value == value) {
            return true
        }

        // 4
        current = if (value < current.value) {
            current.leftChild
        } else {
            current.rightChild
        }
    }

    return false
}
```

Here's how it works:

1. Start by setting `current` to the root node.
2. While `current` is not `null`, check the current node's value.
3. If the value is equal to what you're trying to find, return `true`.
4. Otherwise, decide whether you're going to check the left or right child.

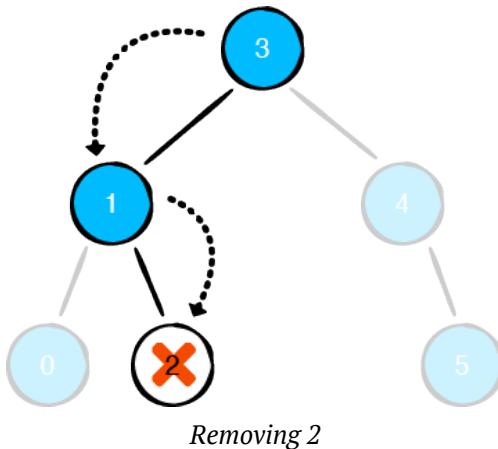
This implementation of `contains` is an $O(\log n)$ operation in a balanced binary search tree.

Removing elements

Removing elements is a little more tricky, as there are a few different scenarios you need to handle.

Case 1: Leaf node

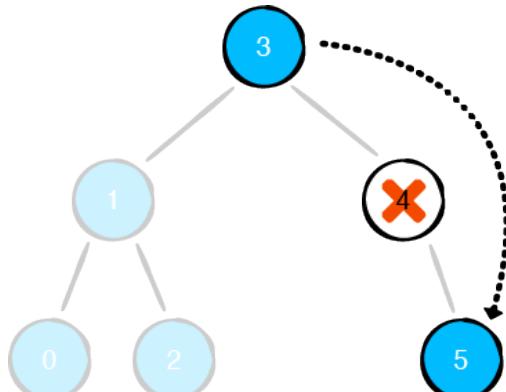
Removing a leaf node is straightforward; simply detach the leaf node.



For non-leaf nodes, however, there are extra steps you must take.

Case 2: Nodes with one child

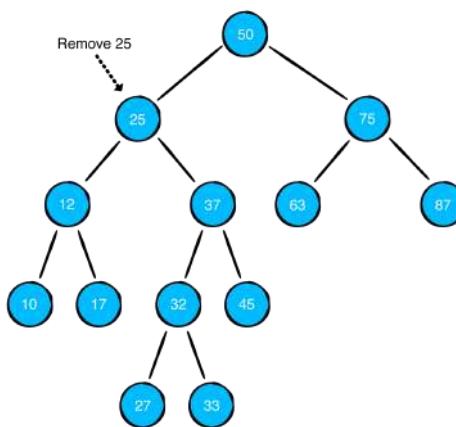
When removing nodes with one child, you need to reconnect that one child with the rest of the tree.



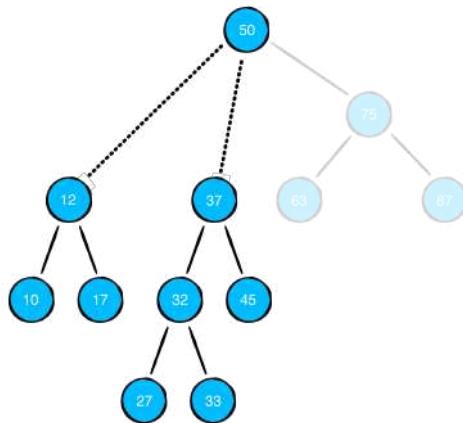
Removing 4, which has 1 child

Case 3: Nodes with two children

Nodes with two children are a bit more complicated, so a more complex example tree will serve better to illustrate how to handle this situation. Assume that you have the following tree and that you want to remove the value 25:

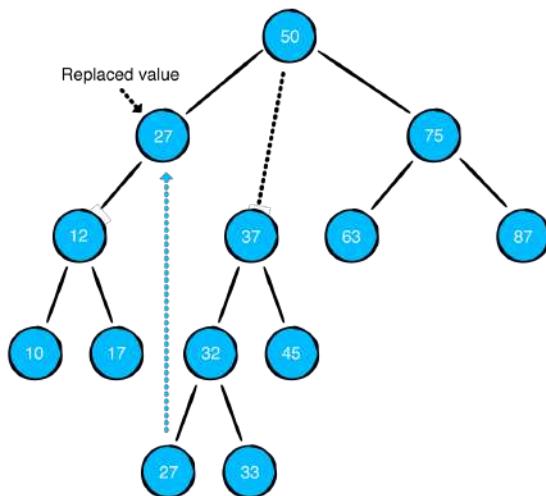


Simply deleting the node presents a dilemma.



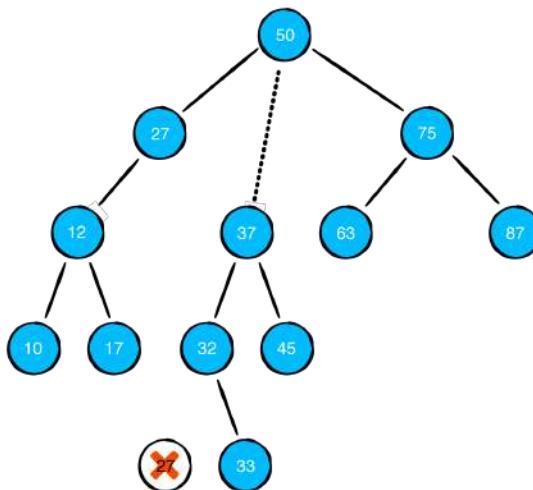
You have two child nodes (12 and 37) to reconnect, but the parent node only has space for one child. To solve this problem, you'll implement a clever workaround by performing a swap.

When removing a node with two children, replace the node you removed with the smallest node in its *right* subtree. Based on the rules of the BST, this is the leftmost node of the right subtree:



It's important to note that this produces a valid binary search tree. Because the new node was the smallest node in the right subtree, all of the nodes in the right subtree will still be greater than or equal to the new node. And because the new node came from the right subtree, all of the nodes in the left subtree will be less than the new node.

After performing the swap, you can simply remove the value you copied, which is just a leaf node.



This will take care of removing nodes with two children.

Implementation

Add the following code to **BinaryNode.kt**:

```

val min: BinaryNode<T>?
  get() = leftChild?.min ?: this
  
```

This recursive `min` property in `BinaryNode` will help you find the minimum node in a subtree.

Open **BinarySearchTree.kt** to implement `remove`. Add the following code at the bottom of the class:

```

fun remove(value: T) {
  root = remove(root, value)
}
  
```

```

private fun remove(
    node: BinaryNode<T>?,
    value: T
): BinaryNode<T>? {
    node ?: return null

    when {
        value == node.value -> {
            // more to come
        }
        value < node.value -> node.leftChild =
remove(node.leftChild, value)
        else -> node.rightChild = remove(node.rightChild, value)
    }
    return node
}

```

This should look familiar to you. You’re using the same recursive setup with a private helper method as you did for `insert`. The different removal cases are handled in the `value == node.value` branch:

```

// 1
if (node.leftChild == null && node.rightChild == null) {
    return null
}
// 2
if (node.leftChild == null) {
    return node.rightChild
}
// 3
if (node.rightChild == null) {
    return node.leftChild
}
// 4
node.rightChild?.min?.value?.let {
    node.value = it
}

node.rightChild = remove(node.rightChild, node.value)

```

Here’s what’s happening:

1. In the case in which the node is a leaf node, you simply return `null`, thereby removing the current node.
2. If the node has no left child, you return `node.rightChild` to reconnect the right subtree.
3. If the node has no right child, you return `node.leftChild` to reconnect the left subtree.

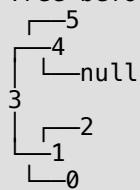
4. This is the case in which the node to be removed has both a left and right child. You replace the node's value with the smallest value from the right subtree. You then call remove on the right child to remove this swapped value.

Go back to `main()` and test `remove` by writing the following:

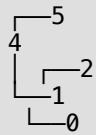
```
"removing a node" example {
    println("Tree before removal:")
    println(exampleTree)
    exampleTree.remove(3)
    println("Tree after removing root:")
    println(exampleTree)
}
```

You'll see the following output in the console:

```
---Example of removing a node---
Tree before removal:
```



```
Tree after removing root:
```



Challenges

Think you have searching of binary trees down cold? Try out these three challenges to lock down the concepts.

Challenge 1 : Is it a BST?

Create a function that checks if a binary tree is a binary search tree.

Solution 1

A binary search tree is a tree where every left child is less than or equal to its parent, and every right child is greater than its parent. An algorithm that verifies whether a tree is a binary search tree involves going through all the nodes and checking for this property.

Write the following in **BinaryNode.kt** in the **BinaryNode** class:

```
val isBinarySearchTree: Boolean
    get() = isBST(this, min = null, max = null)

// 1
private fun isBST(tree: BinaryNode<T>?, min: T?, max: T?):
Boolean {
// 2
    tree ?: return true

// 3
    if (min != null && tree.value <= min) {
        return false
    } else if (max != null && tree.value > max) {
        return false
    }

// 4
    return isBST(tree.leftChild, min, tree.value) &&
isBST(tree.rightChild, tree.value, max)
}
```

Here's how it works:

1. `isBST` is responsible for recursively traversing through the tree and checking for the BST property. It needs to keep track of progress via a reference to a `BinaryNode` and also keep track of the `min` and `max` values to verify the BST property.
2. This is the base case. If `tree` is `null`, then there are no nodes to inspect. A `null`

node is a binary search tree, so you'll return `true` in that case.

3. This is essentially a bounds check. If the current value exceeds the bounds of the `min` and `max` values, the current node does not respect the binary search tree rules.
4. This line contains the recursive calls. When traversing through the left children, the current value is passed in as the `max` value. This is because nodes in the left side cannot be greater than the parent. Vice versa, when traversing to the right, the `min` value is updated to the current value. Nodes in the right side must be greater than the parent. If any of the recursive calls evaluate `false`, the `false` value will propagate to the top.

The time complexity of this solution is $O(n)$ since you need to traverse through the entire tree once. There is also a $O(n)$ space cost since you're making n recursive calls.



Challenge 2 : Equality between BSTs

Override `equals()` to check whether two binary search trees are equal.

Solution 2

Overriding `equals()` is relatively straightforward. For two binary trees to be equal, both trees must have the same elements in the same order. This is how the solution looks:

```
// 1
override fun equals(other: Any?): Boolean {
    // 2
    return if (other != null && other is BinaryNode<*>) {
        this.value == other.value &&
            this.leftChild == other.leftChild &&
            this.rightChild == other.rightChild
    } else {
        false
    }
}
```

Here's an explanation of the code:

1. `equals` recursively checks two nodes and their descendants for equality.
2. Here, you check the value of the left and right nodes for equality. You also recursively check the left children and the right children for equality.

Inside **BinaryNode.kt**, update the `BinaryNode` class declaration to make `T` type comparable:

```
class BinaryNode<T: Comparable<T>>(var value: T)
```

The time complexity of this function is **$O(n)$** . The space complexity of this function is **$O(n)$** .

Challenge 3 : BSTs with same elements?

Create a method that checks if the current tree contains all of the elements of another tree.

Solution 3

Your goal is to create a method that checks if the current tree contains all of the elements of another tree. In other words, the values in the current tree must be a superset of the values in the other tree. The solution looks like this:

```
fun contains(subtree: BinarySearchTree<T>): Boolean {  
    // 1  
    val set = mutableSetOf<T>()  
    root?.traverseInOrder {  
        set.add(it)  
    }  
  
    // 2  
    var isEqual = true  
    subtree.root?.traverseInOrder {  
        isEqual = isEqual && set.contains(it)  
    }  
    return isEqual  
}
```

Here's how it works:

1. Inside `contains`, you begin by inserting all of the elements of the current tree into a set.
2. `isEqual` will store the result. For every element in the subtree, you check if the value is contained in the set. If at any point `set.contains(it)` evaluates to `false`, you'll make sure `isEqual` stays `false` even if subsequent elements evaluate to `true` by assigning `isEqual && list.contains(it)` to itself.

The time complexity for this algorithm is $O(n)$. The space complexity for this algorithm is $O(n)$.

Key points

- The binary search tree is a powerful data structure for holding sorted data.
- Average performance for `insert`, `remove` and `contains` in a BST is $O(\log n)$.
- Performance will degrade to $O(n)$ as the tree becomes unbalanced. This is undesirable, so you'll learn about a self-balancing binary search tree known as the AVL tree in the next chapter.

9 Chapter 9: AVL Trees

By Irina Galata

In the previous chapter, you learned about the $O(\log n)$ performance characteristics of the binary search tree. However, you also learned that *unbalanced* trees could deteriorate the performance of the tree, all the way down to $O(n)$.

In 1962, Georgy Adelson-Velsky and Evgenii Landis came up with the first *self-balancing* binary search tree: the **AVL tree**. In this chapter, you'll dig deeper into how the balance of a binary search tree can impact performance and implement the AVL tree from scratch.

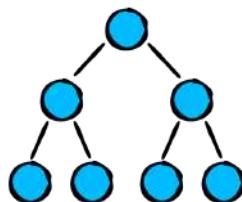


Understanding balance

A balanced tree is the key to optimizing the performance of the binary search tree. There are three main states of balance. You'll look at each one.

Perfect balance

The ideal form of a binary search tree is the **perfectly balanced** state. In technical terms, this means every level of the tree is filled with nodes from top to bottom.

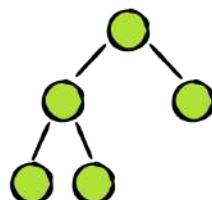


A perfectly balanced tree

Not only is the tree perfectly symmetrical, but the nodes at the bottom level are also completely filled. Note that perfect balanced trees can just have a specific number of nodes. For instance 1, 3 or 7 are possible number of nodes because they can fill 1, 2 or 3 levels respectively. This is the requirement for being perfectly balanced.

“Good-enough” balance

Although achieving perfect balance is ideal, it's rarely possible because it also depends on the specific number of nodes. A tree with 2, 4, 5 or 6 cannot be perfectly balanced since the last level of the tree will not be filled.

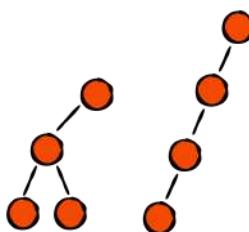


A balanced tree

Because of this, a different definition exists. A **balanced tree** must have all its levels filled, except for the bottom one. In most cases of binary trees, this is the best you can do.

Unbalanced

Finally, there's the **unbalanced** state. Binary search trees in this state suffer from various levels of performance loss depending on the degree of imbalance.



Some unbalanced trees

Keeping the tree balanced gives the **find**, **insert** and **remove** operations an $O(\log n)$ time complexity. AVL trees maintain balance by adjusting the structure of the tree when the tree becomes unbalanced. You'll learn how this works as you progress through the chapter.

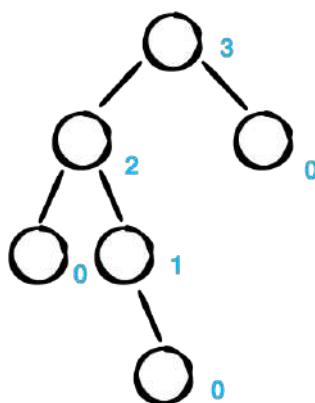
Implementation

Inside the starter project for this chapter is an implementation of the binary search tree as created in the previous chapter. The only difference is that all references to the binary search tree have been renamed to AVL tree.

Binary search trees and AVL trees share much of the same implementation; in fact, all that you'll add is the balancing component. Open the starter project to begin.

Measuring balance

To keep a binary tree balanced, you need a way to measure the balance of the tree. The AVL tree achieves this with a **height** property in each node. In tree-speak, the **height** of a node is the **longest** distance from the current node to a leaf node:



Nodes marked with heights

With the starter project for this chapter open the **AVLNode.kt** file, add the following property to the **AVLNode** class:

```
var height = 0
```

You'll use the *relative* heights of a node's children to determine whether a particular node is balanced.

The height of the left and right children of each node must differ at most by 1. This is known as the **balance factor**.

Write the following immediately below the **height** property of **AVLNode**:

```
var height = 0

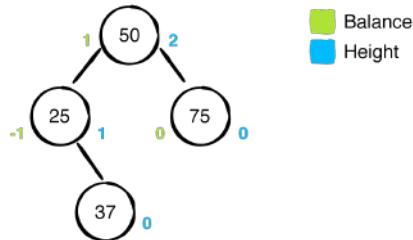
val leftHeight: Int
    get() = leftChild?.height ?: -1

val rightHeight: Int
    get() = rightChild?.height ?: -1

val balanceFactor: Int
    get() = leftHeight - rightHeight
```

The `balanceFactor` computes the height difference of the left and right child. If a particular child is `null`, its height is considered to be `-1`.

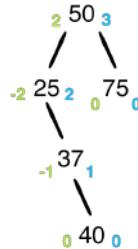
Here's an example of an AVL tree:



AVL tree with balance factors and heights

This is a balanced tree — all levels except the bottom one are filled. The blue numbers represent the height of each node, while the green numbers represent the `balanceFactor`.

Here's an updated diagram with **40** inserted:



Unbalanced tree

Inserting **40** into the tree turns it into an unbalanced tree. Notice how the `balanceFactor` changes. A `balanceFactor` of **2** or **-2** is an indication of an unbalanced tree.

Although more than one node may have a bad balancing factor, you only need to perform the balancing procedure on the bottom-most node containing the invalid balance factor: the node containing **25**.

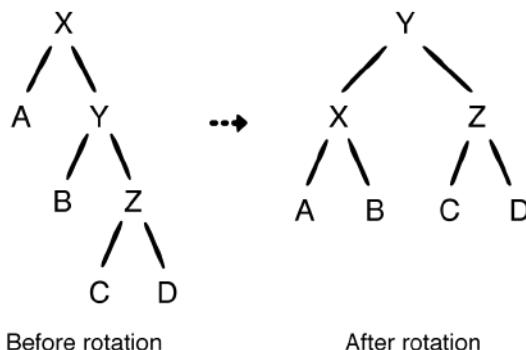
That's where rotations come in.

Rotations

The procedures used to balance a binary search tree are known as **rotations**. There are four rotations in total, one for each way that a tree can become unbalanced. These are known as **left** rotation, **left-right** rotation, **right** rotation and **right-left** rotation.

Left rotation

You can solve the imbalance caused by inserting **40** into the tree using a **left rotation**. A generic left rotation of node **X** looks like this:



Left rotation applied on node X

Before going into specifics, there are two takeaways from this before-and-after comparison:

- In-order traversal for these nodes remains the same.
- The *depth* of the tree is reduced by one level after the rotation.

Add the following method to `AVLTree`:

```

private fun leftRotate(node: AVLNode<T>): AVLNode<T> {
    // 1
    val pivot = node.rightChild!!
    // 2
    node.rightChild = pivot.leftChild
    // 3
    pivot.leftChild = node
    // 4
    node.height = max(node.leftHeight, node.rightHeight) + 1
    pivot.height = max(pivot.leftHeight, pivot.rightHeight) + 1
}

```

```
// 5
return pivot
}
```

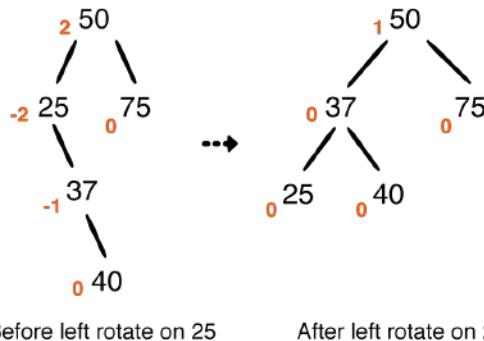
Here are the steps needed to perform a left rotation:

1. The right child is chosen as the **pivot**. This node replaces the rotated node as the root of the subtree (it moves up a level).
2. The node to be rotated becomes the left child of the pivot (it moves down a level). This means that the current left child of the pivot must be moved elsewhere.

In the generic example shown in the earlier image, this is node **b**. Because **b** is smaller than **y** but greater than **x**, it can replace **y** as the right child of **x**. So you update the rotated node's `rightChild` to the pivot's `leftChild`.

3. The pivot's `leftChild` can now be set to the rotated node.
4. You update the heights of the rotated node and the pivot.
5. Finally, you return the pivot so that it can replace the rotated node in the tree.

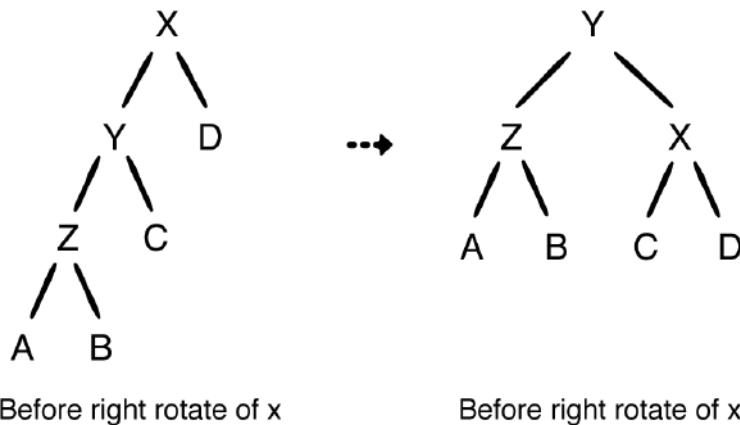
Here are the before-and-after effects of the left rotation of **25** from the previous example:



Right rotation

Right rotation is the symmetrical opposite of left rotation. When a series of left children is causing an imbalance, it's time for a right rotation.

A generic right rotation of node **X** looks like this:



To implement this, add the following code just after `leftRotate()`:

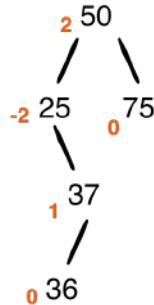
```
private fun rightRotate(node: AVLNode<T>): AVLNode<T> {
    val pivot = node.leftChild!!
    node.leftChild = pivot.rightChild
    pivot.rightChild = node
    node.height = max(node.leftHeight, node.rightHeight) + 1
    pivot.height = max(pivot.leftHeight, pivot.rightHeight) + 1
    return pivot
}
```

This is nearly identical to the implementation of `leftRotate()`, except the references to the left and right children have been swapped.

Right-left rotation

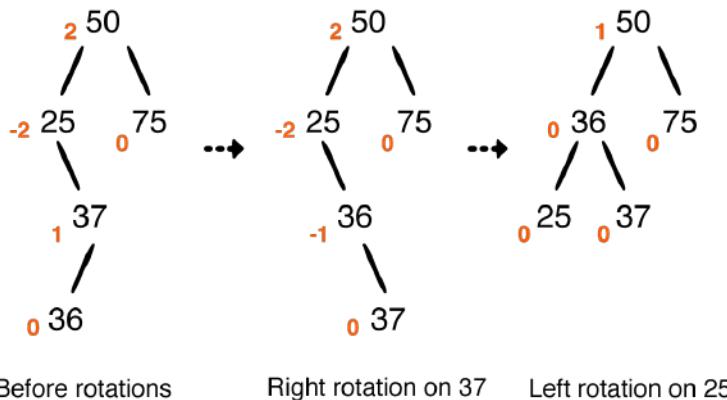
You may have noticed that the left and right rotations balance nodes that are all left children or all right children. Consider the case in which **36** is inserted into the original example tree.

The right-left rotation:



Inserted 36 as left child of 37

Doing a left rotation, in this case, won't result in a balanced tree. The way to handle cases like this is to perform a right rotation on the right child *before* doing the left rotation. Here's what the procedure looks like:



The right-left rotation

1. You apply a right rotation to 37.
2. Now that nodes 25, 36 and 37 are all right children, you can apply a left rotation to balance the tree.

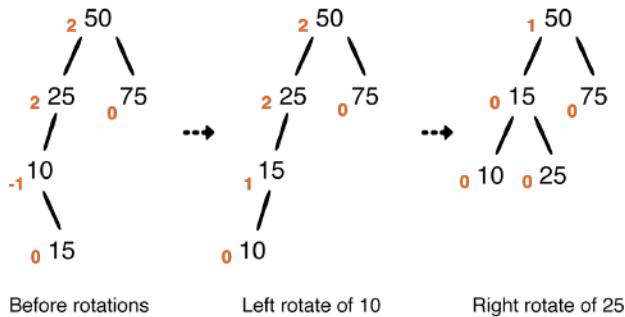
Add the following code immediately after `rightRotate()`:

```
private fun rightLeftRotate(node: AVLNode<T>): AVLNode<T> {
    val rightChild = node.rightChild ?: return node
    node.rightChild = rightRotate(rightChild)
    return leftRotate(node)
}
```

Don't worry just yet about when this is called. You'll get to that in a second. You first need to handle the last case, left-right rotation.

Left-right rotation

Left-right rotation is the symmetrical opposite of the right-left rotation. Here's an example:



The left-right rotation

1. You apply a left rotation to node **10**.
2. Now that nodes **25**, **15** and **10** are all left children, you can apply a right rotation to balance the tree.

Add the following code immediately after `rightLeftRotate()`:

```
private fun leftRightRotate(node: AVLNode<T>): AVLNode<T> {
    val leftChild = node.leftChild ?: return node
    node.leftChild = leftRotate(leftChild)
    return rightRotate(node)
}
```

That's it for rotations. Next, you'll figure out when to apply these rotations at the correct location.

Balanced

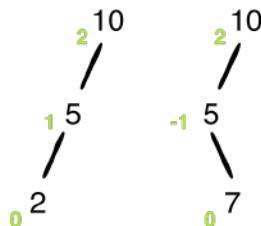
The next task is to design a method that uses `balanceFactor` to decide whether a node requires balancing or not. Write the following method below `leftRightRotate()`:

```
private fun balanced(node: AVLNode<T>): AVLNode<T> {
    return when (node.balanceFactor) {
        2 -> {
            }
        -2 -> {
            }
        else -> node
    }
}
```

There are three cases to consider.

1. A `balanceFactor` of **2** suggests that the left child is heavier (that is, contains more nodes) than the right child. This means that you want to use either right or left-right rotations.
2. A `balanceFactor` of **-2** suggests that the right child is heavier than the left child. This means that you want to use either left or right-left rotations.
3. The default case suggests that the particular node is balanced. There's nothing to do here except to return the node.

You can use the sign of the `balanceFactor` to determine if a single or double rotation is required:



Right rotate, or left right rotate?

Update the `balanced` function to the following:

```
private fun balanced(node: AVLNode<T>): AVLNode<T> {
    return when (node.balanceFactor) {
        2 -> {
            if (node.leftChild?.balanceFactor == -1) {
                leftRightRotate(node)
            } else {
                rightRotate(node)
            }
        }
        -2 -> {
            if (node.rightChild?.balanceFactor == 1) {
                rightLeftRotate(node)
            } else {
                leftRotate(node)
            }
        }
        else -> node
    }
}
```

`balanced()` inspects the `balanceFactor` to determine the proper course of action. All that's left is to call `balanced()` at the proper spot.

Revisiting insertion

You've already done the majority of the work. The remainder is fairly straightforward. Update `insert()` to the following:

```
private fun insert(node: AVLNode<T>?, value: T): AVLNode<T>? {
    node ?: return AVLNode(value)
    if (value < node.value) {
        node.leftChild = insert(node.leftChild, value)
    } else {
        node.rightChild = insert(node.rightChild, value)
    }
    val balancedNode = balanced(node)
    balancedNode?.height = max(balancedNode?.leftHeight ?: 0,
        balancedNode?.rightHeight ?: 0) + 1
    return balancedNode
}
```

Instead of returning the node directly after inserting, you pass it into `balanced()`. This ensures every node in the call stack is checked for balancing issues. You also update the node's height.

Time to test it. Go to **Main.kt** and add the following to `main()`:

```
"repeated insertions in sequence" example {
    val tree = AVLTree<Int>()

    (0..14).forEach {
        tree.insert(it)
    }

    print(tree)
}
```

You'll see the following output in the console:

```
---Example of repeated insertions in sequence---
    14
    └── 13
        ├── 12
        └── 11
            ├── 10
            │   ├── 9
            │   └── 8
            └── 7
                ├── 6
                │   ├── 5
                │   │   ├── 4
                │   │   └── 3
                │   └── 2
                └── 1
                    └── 0
```

Take a moment to appreciate the uniform spread of the nodes. If the rotations weren't applied, this would have become a long, unbalanced chain of right children.

Revisiting remove

Retrofitting the `remove` operation for self-balancing is just as easy as fixing `insert`. In `AVLTree`, find `remove` and replace the final `return` statement with the following:

```
val balancedNode = balanced(node)
balancedNode.height = max(
    balancedNode.leftHeight,
    balancedNode.rightHeight
) + 1
return balancedNode
```

Go back to `main()` in `Main.kt` and add the following code:

```
val tree = AVLTree<Int>()
tree.insert(15)
tree.insert(10)
tree.insert(16)
tree.insert(18)
print(tree)
tree.remove(10)
print(tree)
```

You'll see the following console output:

```
---Example of removing a value---
    18
   / \
  16  null
 / \
15  10
      |
     18
    / \
   16  15
```

Removing **10** caused a left rotation on **15**. Feel free to try out a few more test cases of your own.

Whew! The AVL tree is the culmination of your search for the ultimate binary search tree. The self-balancing property guarantees that the `insert` and `remove` operations function at optimal performance with an $O(\log n)$ time complexity.

Challenges

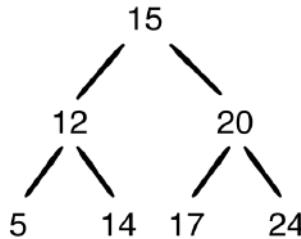
Here are three challenges that revolve around AVL trees. Solve these to make sure you understand the concepts.

Challenge 1: Count the leaves

How many **leaf** nodes are there in a perfectly balanced tree of height 3? What about a perfectly balanced tree of height h ?

Solution 1

A perfectly balanced tree is a tree where all of the leaves are at the same level, and that level is completely filled:



Recall that a tree with only a root node has a height of zero. Thus, the tree in the example above has a height of two. You can extrapolate that a tree with a height of three would have **eight** leaf nodes.

Since each node has two children, the number of leaf nodes doubles as the height increases. Therefore, you can calculate the number of leaf nodes using a simple equation:

```
fun leafNodes(height: Int): Int {  
    return 2.0.pow(height).toInt()  
}
```

Challenge 2: Count the nodes

How many **nodes** are there in a perfectly balanced tree of height 3? What about a perfectly balanced tree of height h ?

Solution 2

Since the tree is perfectly balanced, you can calculate the number of nodes in a perfectly balanced tree of height three using the following:

```
fun nodes(height: Int): Int {  
    var totalNodes = 0  
    (0..height).forEach { currentHeight ->  
        totalNodes += 2.0.pow(currentHeight).toInt()  
    }  
    return totalNodes  
}
```

Although this certainly gives you the correct answer, there's a faster way. If you examine the results of a sequence of height inputs, you'll realize that the total number of nodes is one less than the number of leaf nodes of the next level.

The previous solution is $O(\text{height})$ but here's a faster version of this in $O(1)$:

```
fun nodes(height: Int): Int {  
    return 2.0.pow(height + 1).toInt() - 1  
}
```



Challenge 3: Some refactoring

Since there are many variants of binary trees, it makes sense to group shared functionality in an abstract class. The traversal methods are a good candidate.

Create a `TraversableBinaryNode` abstract class that provides a default implementation of the traversal methods so that concrete subclasses get these methods for free. Have `AVLNode` extend this class.

Solution 3

First, create the following abstract class:

```
abstract class TraversableBinaryNode<Self : TraversableBinaryNode<Self, T>, T>(var value: T) {  
  
    var leftChild: Self? = null  
    var rightChild: Self? = null  
  
    fun traverseInOrder(visit: Visitor<T>) {  
        leftChild?.traverseInOrder(visit)  
        visit(value)  
        rightChild?.traverseInOrder(visit)  
    }  
  
    fun traversePreOrder(visit: Visitor<T>) {  
        visit(value)  
        leftChild?.traversePreOrder(visit)  
        rightChild?.traversePreOrder(visit)  
    }  
  
    fun traversePostOrder(visit: Visitor<T>) {  
        leftChild?.traversePostOrder(visit)  
        rightChild?.traversePostOrder(visit)  
        visit(value)  
    }  
}
```

Finally, add the following at the bottom of `main()`:

```
"using TraversableBinaryNode" example {  
    val tree = AVLTree<Int>()  
    (0..14).forEach {  
        tree.insert(it)  
    }  
    println(tree)  
    tree.root?.traverseInOrder { println(it) }  
}
```

You'll see the following results in the console:

```
---Example of using TraversableBinaryNode---
    14
   / \
  13  12
 / \  /
11  10 9
|  | \
10  9  8
|  |
7  6
|  |
6  5
|  |
5  4
|  |
3  2
|  |
1  0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

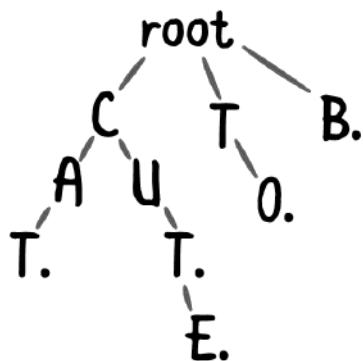
Key points

- A self-balancing tree avoids performance degradation by performing a balancing procedure whenever you add or remove elements in the tree.
- AVL trees preserve balance by readjusting parts of the tree when the tree is no longer balanced.

Chapter 10: Tries

By Irina Galata

The **trie** (pronounced *try*) is a tree that specializes in storing data that can be represented as a collection, such as English words:



A trie containing the words *CAT*, *CUT*, *CUTE*, *TO*, and *B*

Each character in a string is mapped to a node. The last node in each string is marked as a terminating node (a dot in the image above). The benefits of a trie are best illustrated by looking at it in the context of prefix matching.

In this chapter, you'll first compare the performance of the trie to the array. You'll then implement the trie from scratch.



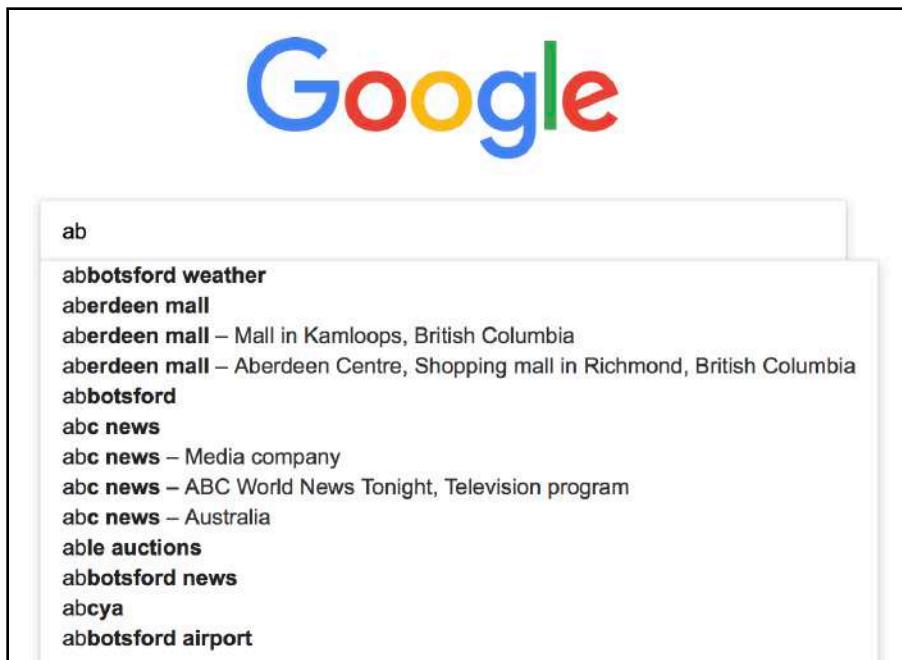
Example

You are given a collection of strings. How would you build a component that handles prefix matching? Here's one way:

```
class EnglishDictionary {  
  
    private val words: ArrayList<String> = ...  
  
    fun words(prefix: String) = words.filter  
    { it.startsWith(prefix) }  
  
}
```

`words()` goes through the collection of strings and returns the strings that match the prefix.

If the number of elements in the `words` array is small, this is a reasonable strategy. But if you're dealing with more than a few thousand words, the time it takes to go through the `words` array will be unacceptable. The time complexity of `words()` is $O(k * n)$, where k is the longest string in the collection, and n is the number of words you need to check.



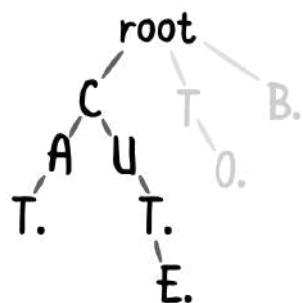
Imagine the number of words Google needs to parse

The trie data structure has excellent performance characteristics for this type of problem; like a tree with nodes that support multiple children, each node can represent a single character.

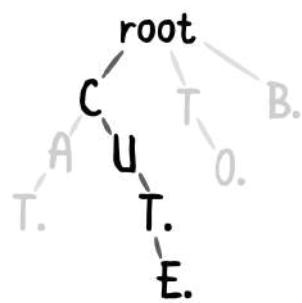
You form a word by tracing the collection of characters from the root to a node with a special indicator — a terminator — represented by a black dot. An interesting characteristic of the trie is that multiple words can share the same characters.

To illustrate the performance benefits of the trie, consider the following example in which you need to find the words with the prefix CU.

First, you travel to the node containing C. This quickly excludes other branches of the trie from the search operation:

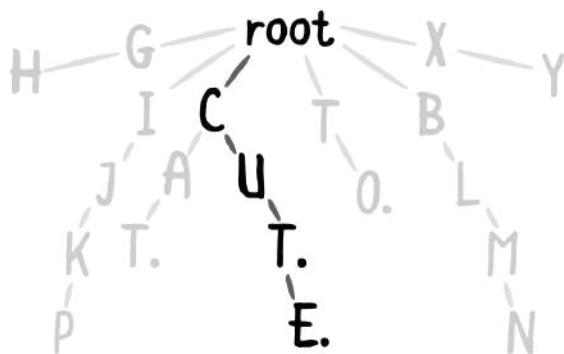


Next, you need to find the words that have the next letter, U. You traverse to the U node:



Since that's the end of your prefix, the trie returns all collections formed by the chain of nodes from the U node. In this case, the words CUT and CUTE are returned. Imagine if this trie contained hundreds of thousands of words.

The number of comparisons you can avoid by employing a trie is substantial.



Implementation

Open up the starter project for this chapter.

TrieNode

You'll begin by creating the node for the trie. Create a new file named **TrieNode.kt**. Add the following to the file:

```
class TrieNode<Key>(var key: Key?, var parent: TrieNode<Key>?) {  
    val children: HashMap<Key, TrieNode<Key>> = HashMap()  
    var isTerminating = false  
}
```

This interface is slightly different compared to the other nodes you've encountered:

1. `key` holds the data for the node. This is optional because the root node of the trie has no key.
2. A `TrieNode` holds a reference to its parent. This reference simplifies `remove()` later on.
3. In binary search trees, nodes have a left and right child. In a trie, a node needs to hold multiple different elements. You've declared a `children` map to help with that.

4. As discussed earlier, `isTerminating` acts as an indicator for the end of a collection.

Trie

Next, you'll create the trie itself, which will manage the nodes. Create a new file named **Trie.kt**. Add the following to the file:

```
class Trie<Key> {  
    private val root = TrieNode<Key>(key = null, parent = null)  
}
```

The `Trie` class can store collections containing `Keys`.

Next, you'll implement four operations for the trie: `insert`, `contains`, `remove` and a prefix match.

Insert

Tries work with lists of the `Key` type. The trie takes the list and represents it as a series of nodes in which each node maps to an element in the list.

Add the following method to `Trie`:

```
fun insert(list: List<Key>) {  
    // 1  
    var current = root  
  
    // 2  
    list.forEach { element ->  
        if (current.children[element] == null) {  
            current.children[element] = TrieNode(element, current)  
        }  
        current = current.children[element]!!  
    }  
  
    // 3  
    current.isTerminating = true  
}
```

Here's what's going on:

1. `current` keeps track of your traversal progress, which starts with the root node.

2. A trie stores each element of a list in separate nodes. For each element of the list, you first check if the node currently exists in the `children` map. If it doesn't, you create a new node. During each loop, you move `current` to the next node.
3. After iterating through the `for` loop, `current` should be referencing the node representing the end of the list. You mark that node as the terminating node.

The time complexity for this algorithm is $O(k)$, where k is the number of elements in the list you're trying to insert. This is because you need to traverse through or create each node that represents each element of the new list.

Contains

`contains` is similar to `insert`. Add the following method to `Trie`:

```
fun contains(list: List<Key>): Boolean {
    var current = root

    list.forEach { element ->
        val child = current.children[element] ?: return false
        current = child
    }

    return current.isTerminating
}
```

Here, you traverse the trie in a way similar to `insert`. You check every element of the list to see if it's in the tree. When you reach the last element of the list, it must be a terminating element. If not, the list wasn't added to the tree and what you've found is merely a subset of a larger list.

The time complexity of `contains` is $O(k)$, where k is the number of elements in the list that you're looking for. This is because you need to traverse through k nodes to find out whether or not the list is in the trie.

To test `insert` and `contains`, navigate to `main()` and add the following code:

```
"insert and contains" example {
    val trie = Trie<Char>()
    trie.insert("cute".toList())
    if (trie.contains("cute".toList())) {
        println("cute is in the trie")
    }
}
```

`String` is not a collection type in Kotlin, but you can easily convert it to a list of characters using the `toList` extension.



After running `main()`, you'll see the following console output:

```
---Example of insert and contains---
cute is in the trie
```

You can make storing `Strings` in a trie more convenient by adding some extensions. Create a file named **Extensions.kt**, and add the following:

```
fun Trie<Char>.insert(string: String) {
    insert(string.toList())
}

fun Trie<Char>.contains(string: String): Boolean {
    return contains(string.toList())
}
```

These extension functions are only applicable to tries that store lists of characters. They hide the extra `toList()` calls you need to pass in a `String`, allowing you to simplify the previous code example to this:

```
"insert and contains" example {
    val trie = Trie<Char>()
    trie.insert("cute")
    if (trie.contains("cute")) {
        println("cute is in the trie")
    }
}
```

Remove

Removing a node in the trie is a bit more tricky. You need to be particularly careful when removing each node since nodes can be shared between multiple different collections. Write the following method immediately below `contains`:

```
fun remove(collection: CollectionType) {
    // 1
    var current = root

    collection.forEach {
        val child = current.children[it] ?: return
        current = child
    }

    if (!current.isTerminating) return

    // 2
    current.isTerminating = false
```

```
// 3
val parent = current.parent
while (current.children.isEmpty() && !current.isTerminating) {
    parent?.let {
        it.children[current.key] = null
        current = it
    }
}
```

Here's how it works:

1. This part should look familiar, as it's basically the implementation of `contains`. You use it here to check if the collection is part of the trie and to point `current` to the last node of the collection.
2. You set `isTerminating` to `false` so that the current node can be removed by the loop in the next step.
3. This is the tricky part. Since nodes can be shared, you don't want to carelessly remove elements that belong to another collection. If there are no other children in the current node, it means that other collections do not depend on the current node.

You also check to see if the current node is a terminating node. If it is, then it belongs to another collection. As long as `current` satisfies these conditions, you continually backtrack through the `parent` property and remove the nodes.

The time complexity of this algorithm is $O(k)$, where k represents the number of elements of the collection that you're trying to remove.

Sticking to strings, it's time to add another extension in **Extensions.kt**:

```
fun Trie<Char>.remove(string: String) {
    remove(string.toList())
}
```

Go back to `main()` and add the following to the bottom:

```
"remove" example {
    val trie = Trie<Char>()

    trie.insert("cut")
    trie.insert("cute")

    println("\n*** Before removing ***")
    assert(trie.contains("cut"))
    println("\"cut\" is in the trie")
```

```

    assert(trie.contains("cute"))
    println("\\"cute\" is in the trie")

    println("\n*** After removing cut ***")
    trie.remove("cut")
    assert(!trie.contains("cut"))
    assert(trie.contains("cute"))
    println("\\"cute\" is still in the trie")
}

```

You'll see the following output in the console:

```

---Example of: remove---

*** Before removing ***
"cut" is in the trie
"cute" is in the trie

*** After removing cut ***
"cute" is still in the trie

```

Prefix matching

The most iconic algorithm for the trie is the prefix-matching algorithm. Write the following at the bottom of Trie:

```

fun collections(prefix: List<Key>): List<List<Key>> {
    // 1
    var current = root

    prefix.forEach { element ->
        val child = current.children[element] ?: return emptyList()
        current = child
    }

    // 2
    return collections(prefix, current)
}

```

Here's how it works:

1. You start by verifying that the trie contains the prefix. If not, you return an empty list.
2. After you've found the node that marks the end of the prefix, you call a recursive helper method to find all of the sequences after the current node.



Next, add the code for the helper method:

```
private fun collections(prefix: List<Key>, node: TrieNode<Key>?): List<List<Key>> {
    // 1
    val results = mutableListOf<List<Key>>()

    if (node?.isTerminating == true) {
        results.add(prefix)
    }

    // 2
    node?.children?.forEach { (key, node) ->
        results.addAll(collections(prefix + key, node))
    }

    return results
}
```

This code works like so:

1. You create a `MutableList` to hold the results. If the current node is a terminating node, you add the corresponding prefix to the results.
2. Next, you need to check the current node's children. For every child node, you recursively call `collections()` to seek out other terminating nodes.

`collection()` has a time complexity of $O(k*m)$, where k represents the longest collection matching the prefix and m represents the number of collections that match the prefix.

Recall that arrays have a time complexity of $O(k*n)$, where n is the number of elements in the collection.

For large sets of data in which each collection is uniformly distributed, tries have far better performance as compared to using arrays for prefix matching.

Time to take the method for a spin. Add a handy extension first, in `Extensions.kt`:

```
fun Trie<Char>.collections(prefix: String): List<String> {
    return collections(prefix.toList()).map
    { it.joinToString(separator = "") }
}
```

This extension maps the input string into a list of characters, and then maps the lists in the result of the `collections()` call back to strings. Neat!

Navigate back to `main()` and add the following:

```
"prefix matching" example {
    val trie = Trie<Char>().apply {
        insert("car")
        insert("card")
        insert("care")
        insert("cared")
        insert("cars")
        insert("carbs")
        insert("carapace")
        insert("cargo")
    }

    println("\nCollections starting with \"car\"")
    val prefixedWithCar = trie.collections("car")
    println(prefixedWithCar)

    println("\nCollections starting with \"care\"")
    val prefixedWithCare = trie.collections("care")
    println(prefixedWithCare)
}
```

You'll see the following output in the console:

```
---Example of prefix matching---

Collections starting with "car"
[car, carapace, carbs, cars, card, care, cared, cargo]

Collections starting with "care"
[care, cared]
```

Challenges

Challenge 1: Adding more features

The current implementation of the trie is missing some notable operations. Your task for this challenge is to augment the current implementation of the trie by adding the following:

1. A `lists` property that returns all of the lists in the trie.
2. A `count` property that tells you how many lists are currently in the trie.
3. An `isEmpty` property that returns `true` if the trie is empty, `false` otherwise.

Solution 1

For this solution, you'll implement `lists` as a computed property. It'll be backed by a private property named `storedLists`.

Inside `Trie.kt`, add the following new properties:

```
private val storedLists: MutableSet<List<Key>> = mutableSetOf()  
  
val lists: List<List<Key>>  
    get() = storedLists.toList()
```

`storedLists` is a set of the lists currently contained by the trie. Reading the `lists` property returns a list of these tries, which is created from the privately maintained set.

Next, inside `insert()`, find the line `current.isTerminating = true` and add the following immediately below it:

```
storedLists.add(list)
```

In `remove()`, find the line `current.isTerminating = false` and add the following immediately above that line:

```
storedLists.remove(list)
```

Adding the `count` and `isEmpty` properties is straightforward now that you're keeping track of the lists:

```
val count: Int
```

```
get() = storedLists.count()  
  
val isEmpty: Boolean  
    get() = storedLists.isEmpty()
```

Key points

- Tries provide great performance metrics in regards to prefix matching.
- Tries are relatively memory efficient since individual nodes can be shared between many different values. For example, “car”, “carbs”, and “care” can share the first three letters of the word.

Chapter 11: Binary Search

Irina Galata

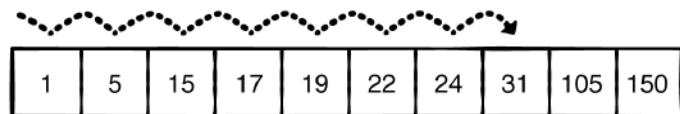
Binary search is one of the most efficient searching algorithms with a time complexity of $O(\log n)$. This is comparable with searching for an element inside a **balanced** binary search tree.

Two conditions need to be met before you can use binary search:

- The collection must be able to perform index manipulation in constant time.
Kotlin collections that can do this include the `Array` and the `ArrayList`.
- The collection must be **sorted**.

Example

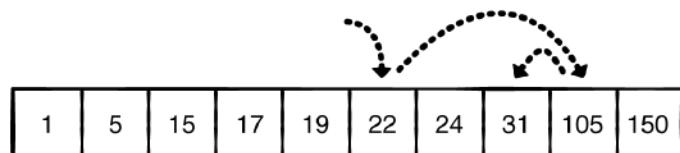
The benefits of binary search are best illustrated by comparing it with linear search. The `ArrayList` type uses linear search to implement its `indexOf()` method. This means that it traverses through the entire collection or until it finds the element.



Linear search for the value 31.

Binary search handles things differently by taking advantage of the fact that the collection is already sorted.

Here's an example of applying binary search to find the value **31**:

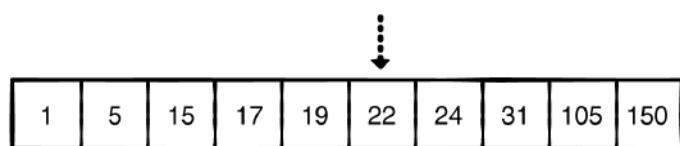


Binary search for the value 31.

Instead of eight steps to find 31, it only takes three. Here's how it works:

Step 1: Find middle index

The first step is to find the middle index of the collection, like so:



Step 2: Check the element at the middle index

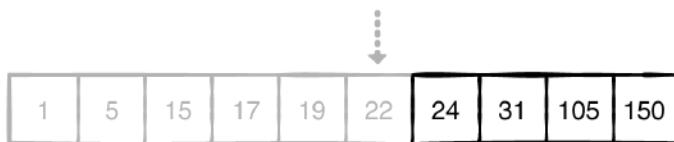
The next step is to check the element stored at the middle index. If it matches the value you're looking for, you return the index. Otherwise, you'll continue to Step 3.

Step 3: Recursively call binary Search

The final step is to recursively call binary search. However, this time, you'll only consider the elements exclusively to the **left** or **right** of the middle index, depending on the value you're searching for. If the value you're searching for is less than the middle value, you search the left subsequence. If it's greater than the middle value, you search the right subsequence.

Each step effectively removes half of the comparisons you would otherwise need to perform.

In the example where you're looking for the value **31** (which is greater than the middle element **22**), you apply binary search on the right subsequence.



You continue these three steps until you can no longer split the collection into left and right halves, or until you find the value inside the collection.

Binary search achieves an $O(\log n)$ time complexity this way.

Implementation

Open the starter project for this chapter. Create a new file named **BinarySearch.kt**. Add the following to the file:

```
// 1
fun <T : Comparable<T>> ArrayList<T>.binarySearch(
    value: T,
    range: IntRange = indices // 2
): Int? {
    // more to come
}
```

Things are fairly simple, so far:

1. You want **binarySearch** to be available on any **ArrayList**, so you define it as a generic extension function.

- Binary search is recursive, so you need to be able to pass in a range to search. The parameter range is made optional by giving it a default value; this lets you start the search without having to specify a range. In this case, the indices property of ArrayList is used, which covers all valid indexes of the collection.

Next, implement binarySearch:

```
// 1
if (range.first > range.last) {
    return null
}

// 2
val size = range.last - range.first + 1
val middle = range.first + size / 2

return when {
    // 3
    this[middle] == value -> middle
    // 4
    this[middle] > value -> binarySearch(value, range.first until
    middle)
    else -> binarySearch(value, (middle + 1)..range.last)
}
```

Here are the steps:

- First, you check if the range contains at least one element. If it doesn't, the search has failed and you return null.
- Now that you're sure you have elements in the range, you find the middle index in the range.
- You then compare the value at this index with the value you're searching for. If they match, you return the middle index.
- If not, you recursively search either the left or right half of the collection, excluding the middle item in both cases.

That wraps up the implementation of binary search. Go back to main() to test it out:

```
"binary search" example {
    val array = arrayListOf(1, 5, 15, 17, 19, 22, 24, 31, 105,
    150)

    val search31 = array.indexOf(31)
    val binarySearch31 = array.binarySearch(31)

    println("indexOf(): $search31")
```

```
    } println("binarySearch(): $binarySearch31")
```

You'll see the following output in the console:

```
---Example of binary search---
indexOf(): 7
binarySearch(): 7
```

This represents the index of the value you're looking for.

Binary search is a powerful algorithm to learn, and it comes up often in programming interviews. Whenever you read something along the lines of “Given a sorted array...”, consider using the binary search algorithm. Also, if you’re given a problem that looks like it’s going to be $O(n^2)$ to search, consider doing some upfront sorting. With upfront sorting, you can use binary searching to reduce complexity to the cost of the sort at $O(n \log n)$.

Challenges

Challenge 1: Find the range

Write a function that searches a **sorted** `ArrayList` and finds the range of indices for a particular element. For example:

```
val array = arrayListOf(1, 2, 3, 3, 3, 4, 5, 5)
val indices = array.findIndices(3)
println(indices)
```

`findIndices` should return the range `2..4`, since those are the start and end indices for the value 3.

Solution 1

An unoptimized but elegant solution is quite simple:

```
fun <T : Comparable<T>> ArrayList<T>.findIndices(
    value: T
): IntRange? {
    val startIndex = indexOf(value)
    val endIndex = lastIndexOf(value)

    if (startIndex == -1 || endIndex == -1) {
```

```

        return null
    }

    return startIndex.. endIndex
}

```

The time complexity of this solution is $O(n)$, which may not seem to be a cause for concern. However, you can optimize the solution to an $O(\log n)$ time complexity solution.

Binary search is an algorithm that identifies values in a **sorted** collection, so keep that in mind whenever the problem promises a sorted collection. The binary search you implemented in the theory chapter is not powerful enough to reason whether the index is a start or end index. You'll modify that binary search to accommodate for this new rule.

Write the following in **BinarySearch.kt**:

```

fun <T : Comparable<T>> ArrayList<T>.findIndices(
    value: T
): IntRange? {
    val startIndex = startIndex(value, 0..size) ?: return null
    val endIndex = endIndex(value, 0..size) ?: return null

    return startIndex until endIndex
}

private fun <T : Comparable<T>> ArrayList<T>.startIndex(
    value: T,
    range: IntRange
): Int? {
    // more to come
}

private fun <T : Comparable<T>> ArrayList<T>.endIndex(
    value: T,
    range: IntRange
): Int? {
    // more to come
}

```

This time, `findIndices` will use specialized binary searches. `startIndex` and `endIndex` will be the ones that do the heavy lifting with a customized binary search. You'll modify binary search so that it also inspects whether the adjacent value — depending on whether you're looking for the start or end index — is different from the current value.



Update the `startIndex` function to the following:

```
private fun <T : Comparable<T>> ArrayList<T>.startIndex(
    value: T,
    range: IntRange
): Int? {
    // 1
    val middleIndex = range.start + (range.last - range.start + 1) / 2

    // 2
    if (middleIndex == 0 || middleIndex == size - 1) {
        return if (this[middleIndex] == value) {
            middleIndex
        } else {
            null
        }
    }

    // 3
    return if (this[middleIndex] == value) {
        if (this[middleIndex - 1] != value) {
            middleIndex
        } else {
            startIndex(value, range.start until middleIndex)
        }
    } else if (value < this[middleIndex]) {
        startIndex(value, range.start until middleIndex)
    } else {
        startIndex(value, (middleIndex + 1)..range.last)
    }
}
```

Here's what you do with this code:

1. You start by calculating the middle value of the indices contained in `range`.
2. This is the base case of this recursive function. If the middle index is the first or last accessible index of the array, you don't need to call binary search any further. You'll determine whether or not the current index is a valid bound for the given value.
3. Here, you check the value at the index and make your recursive calls. If the value at `middleIndex` is equal to the value you're given, you check to see if the predecessor is also the same value. If it isn't, you know that you've found the starting bound. Otherwise, you'll continue by recursively calling `startIndex`.

The `endIndex` method is similar. Update the `endIndex` implementation to the following:

```
private fun <T : Comparable<T>> ArrayList<T>.endIndex(
    value: T,
    range: IntRange
): Int? {
    val middleIndex = range.start + (range.last - range.start + 1) / 2

    if (middleIndex == 0 || middleIndex == size - 1) {
        return if (this[middleIndex] == value) {
            middleIndex + 1
        } else {
            null
        }
    }

    return if (this[middleIndex] == value) {
        if (this[middleIndex + 1] != value) {
            middleIndex + 1
        } else {
            endIndex(value, (middleIndex + 1)..range.last)
        }
    } else if (value < this[middleIndex]) {
        endIndex(value, range.start until middleIndex)
    } else {
        endIndex(value, (middleIndex + 1)..range.last)
    }
}
```

Test your solution by writing the following in `main()`:

```
"binary search for a range" example {
    val array = arrayListOf(1, 2, 3, 3, 3, 4, 5, 5)
    val indices = array.findIndices(3)
    println(indices)
}
```

You'll see the following output in the console:

```
---Example of binary search for a range---
2..4
```

This improves the time complexity from the previous $O(n)$ to $O(\log n)$.

Key points

- Binary search is only a valid algorithm on sorted collections.
- Sometimes, it may be beneficial to sort a collection just to leverage the binary search capability for looking up elements.
- The `indexOf` method of arrays uses linear search, which has an $O(n)$ time complexity. Binary search has an $O(\log n)$ time complexity, which scales much better for large data sets.

Chapter 12: The Heap Data Structure

By Irina Galata

Have you ever been to the arcade and played those crane machines that contain stuffed animals or cool prizes? These machines make it extremely difficult to win. But the fact that you set your eyes on the item you want is the very essence of the heap data structure!

Have you seen the movie *Toy Story* with the claw and the little green squeaky aliens? Just imagine that the claw machine operates on your heap data structure and will always pick the element with the highest priority.



In this chapter, you'll focus on creating a heap, and you'll see how convenient it is to fetch the minimum and maximum element of a collection.



What is a heap?

A heap is a complete binary tree data structure also known as a **binary heap** that you can construct using an array.

Note: Don't confuse these heaps with memory heaps. The term heap is sometimes confusingly used in computer science to refer to a pool of memory. Memory heaps are a different concept and are not what you're studying here.

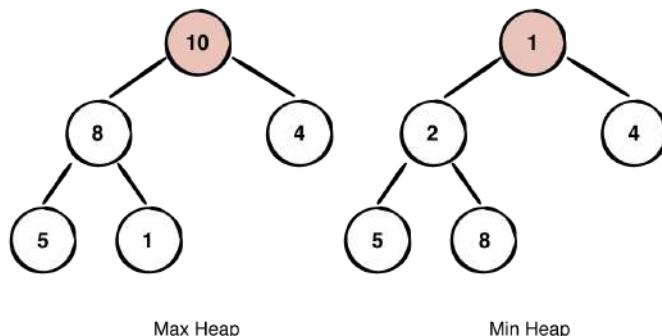
Heaps come in two flavors:

1. **Maxheap**, in which elements with a **higher** value have a higher priority.
2. **Minheap**, in which elements with a **lower** value have a higher priority.

Note: It's important to say that the concept of heap is valid for every type of object that can be compared to others of the same type. In this chapter you'll see mostly **Ints** but the same concepts are true for all **Comparable** types or, as you'll see later, if a **Comparator** is provided.

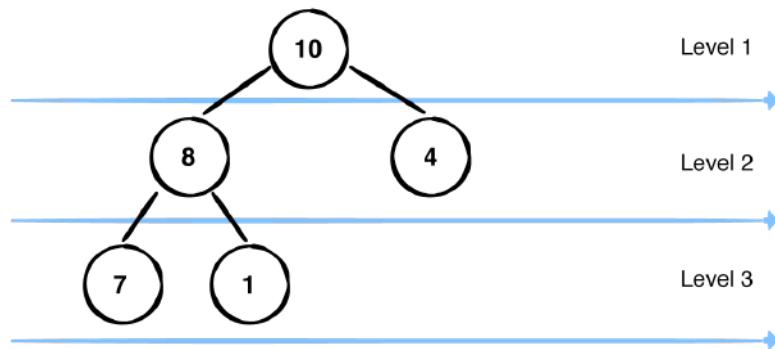
A heap has an important characteristic that must always be satisfied. This is known as the **heap invariant** or **heap property**.

The heap property



In a **maxheap**, parent nodes must always contain a value that is *greater than or equal to* the value in its children. The root node will always contain the highest value.

In a **minheap**, parent nodes must always contain a value that is *less than or equal to* the value in its children. The root node will always contain the lowest value.



Another important property of a heap is that it's a **complete** binary tree. This means that every level must be filled, except for the last level. It's like a video game wherein you can't go to the next level until you have completed the current one.

Heap applications

Some useful applications of a heap include:

- Calculating the minimum or maximum element of a collection.
- Heap sort.
- Implementing a priority queue.
- Supporting graph algorithms, like Prim's or Dijkstra's, with a priority queue.

Note: You'll learn about each of these concepts in later chapters.

Common heap operations

Open the empty starter project for this chapter. Start by defining the following basic Collection type:

```
interface Collection<Element> {  
    val count: Int  
        get  
  
    val isEmpty: Boolean  
        get() = count == 0  
  
    fun insert(element: Element)  
  
    fun remove(): Element?  
  
    fun remove(index: Int): Element?  
}
```

Here you have a generic Collection interface with the basic property count which returns the number of elements and the boolean property isEmpty which just tests if the count is 0. It also contains the classical operations of inserting and deletion. Given that you can define the Heap interface like this.

```
interface Heap<Element> : Collection<Element> {  
  
    fun peek(): Element?  
}
```

The peek operation is a generalization of methods returning the min or the max depending on the implementation. Because of this you can usually find the same operation with name extract-min or extract-max.

Sorting and comparing

The heap properties imply there must be a way to **compare** each element and so a way to test if an element A is greater, smaller or equals than the element B. In Kotlin, as well as in Java, this can be achieved in 2 different ways:

- Element implements the Comparable<Element> interface
- You can provide a Comparator<Element> implementation

Implementing the `Comparable<Element>` interface, a type `Element` can only provide a single way of comparing instances of itself with others of the same type. If you use a `Comparator<Element>` you can choose different way of sorting simply using different `Comparator` implementations. In both cases you need to abstract the way you compare different instances. Because of this you can define the abstract class which contains the definition of the `compare` method you'll implement in different ways depending on the 2 different approaches. This method returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

```
abstract class AbstractHeap<Element>() : Heap<Element> {

    abstract fun compare(a: Element, b: Element): Int
}
```

In case of `Comparable` types you can define a `Heap` implementation like the following where the `Element` type implements `Comparable<Element>` and `compare` method invokes the related `compareTo` method.

```
class ComparableHeapImpl<Element : Comparable<Element>>() :
    AbstractHeap<Element>() {

    override fun compare(a: Element, b: Element): Int =
        a.compareTo(b)
}
```

In case you want to use a `Comparator<Element>` you can implement a `Heap` like this where the `compare` method delegates to the `Comparator<Element>` you pass as parameter.

```
class ComparatorHeapImpl<Element>(
    private val comparator: Comparator<Element>
) : AbstractHeap<Element>() {

    override fun compare(a: Element, b: Element): Int =
        comparator.compare(a, b)
}
```

In the previous code, you'll see some errors because of the missing implementation of the `peek` operation along with the operations from the `Collection` interface, but you'll fix everything very soon. Anyway, depending on the `Comparator<Element>` or the `Comparable<Element>` implementation you can create different minheaps and maxheaps. Before this you need some theory.

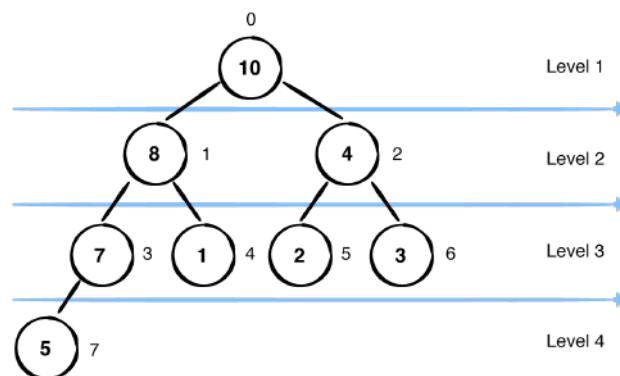
How do you represent a heap?

Trees hold nodes that store references to their children. In the case of a binary tree, these are references to a left and a right child.

Heaps are indeed binary trees, but you can represent them with a simple array. This seems like an unusual way to build a tree, but one of the benefits of this heap implementation is **efficient time** and **space complexity**, as the elements in the heap are all stored together in memory.

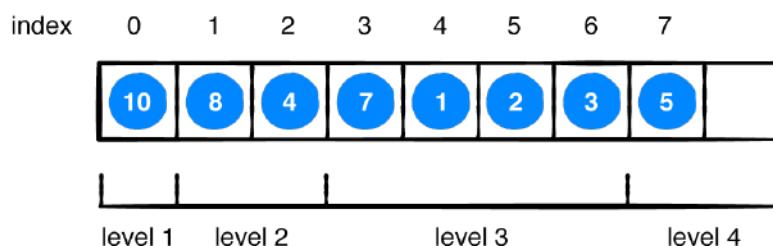
You'll see later on that **swapping** elements plays a big part in heap operations. This is also easier to do with an array than with a binary tree data structure.

It's time to look at how you can represent heaps using an array. Take the following binary heap:



To represent the heap above as an array, you would simply iterate through each element level-by-level from left to right.

Your traversal would look something like this:

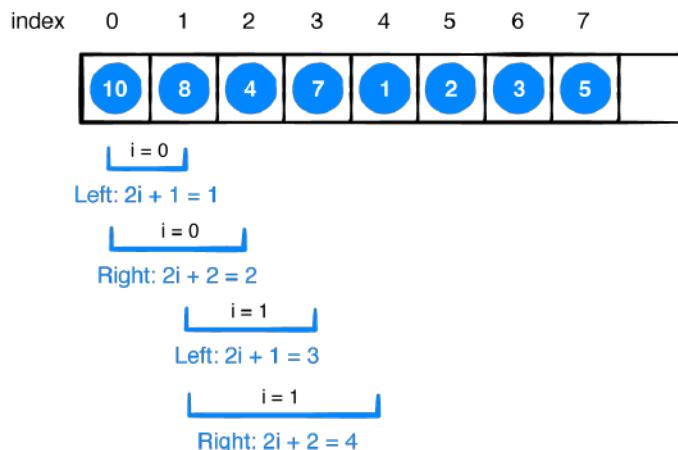


As you go up a level, you'll have twice as many nodes than in the level before.

It's now easy to access any node in the heap. You can compare this to how you'd access elements in an array: Instead of traversing down the left or right branch, you can simply access the node in your array using simple formulas.

Given a node at a zero-based index i :

- You can find the **left child** of this node at index $2i + 1$.
- You can find the **right child** of this node at index $2i + 2$.



You might want to obtain the parent of a node. You can solve for i in this case. Given a child node at index i , you can find this child's parent node at index $(i - 1) / 2$. Just remember this is an operation between `Int`s which returns an `Int`; in other languages you can call it the `floor` operation.

Note: Traversing down an actual binary tree to get the left and right child of a node is an $O(\log n)$ operation. In a random-access data structure, such as an array, that same operation is just $O(1)$.

Next, use your new knowledge to add some properties and convenience methods to the `AbstractHeap` class:

```
var elements: ArrayList<Element> = ArrayList<Element>()
override val count: Int
```

```
    get() = elements.size

    override fun peek(): Element? = elements.first()

    private fun leftChildIndex(index: Int) = (2 * index) + 1

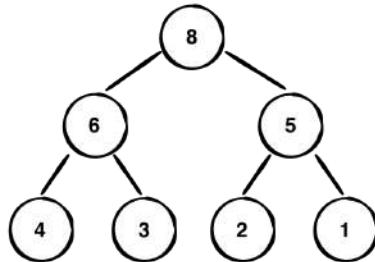
    private fun rightChildIndex(index: Int) = (2 * index) + 2

    private fun parentIndex(index: Int) = (index - 1) / 2
```

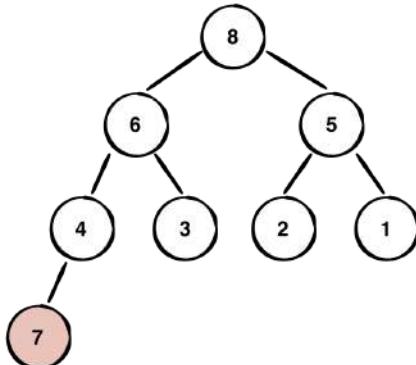
Now that you have a better understanding of how you can represent a heap using an array, you'll look at some important operations of a heap.

Inserting into a heap

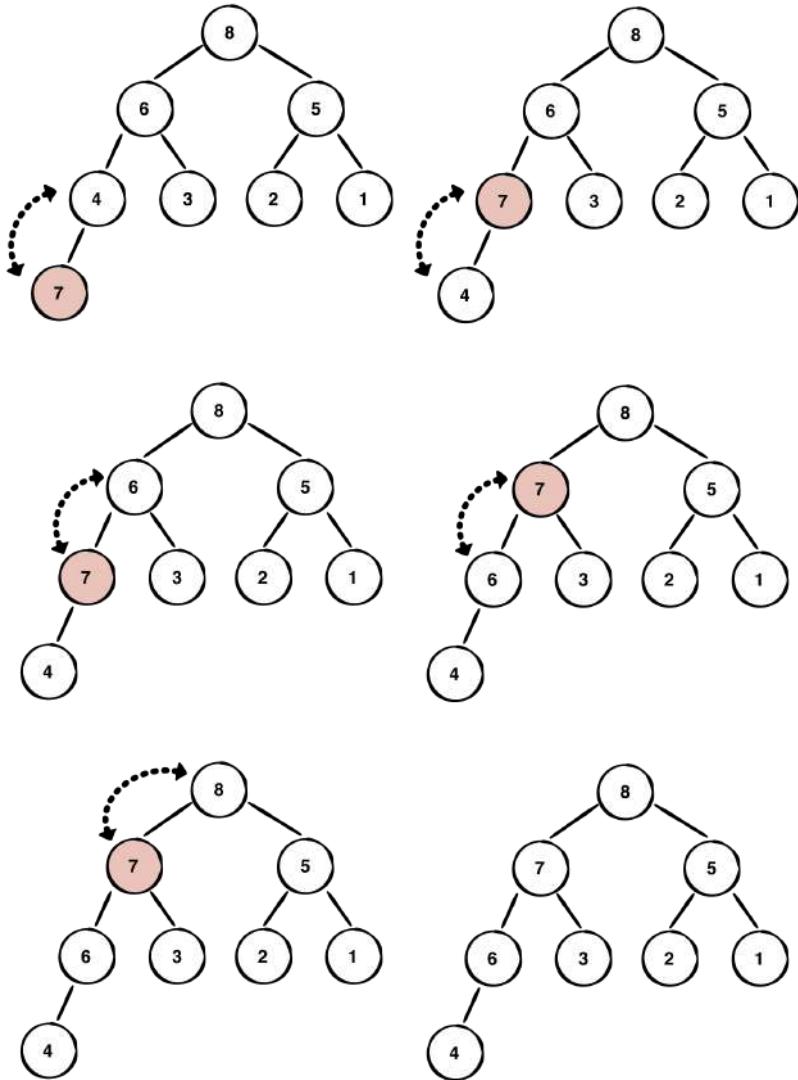
Suppose you insert a value of 7 to the heap below:



First, you add the value to the end of the heap:



Next, you must check the max heap's property. In order to do this you have to **sift up** since the node that you just inserted might have a higher priority than its parents. It does so by comparing the current node with its parent and swapping them if needed.



Your heap has now satisfied the max heap property.

Implementation of insert

Add the following code to `AbstractHeap`:

```
override fun insert(element: Element) {
    elements.add(element) // 1
    siftUp(count - 1) // 2
}

private fun siftUp(index: Int) {
    var child = index
    var parent = parentIndex(child)
    while (child > 0 && compare(elements[child], elements[parent]) > 0) {
        Collections.swap(elements, child, parent)
        child = parent
        parent = parentIndex(child)
    }
}
```

As you can see, the implementation is rather straightforward:

1. `insert` appends the element to the array and then performs a sift up.
2. `siftUp` swaps the current node with its parent, as long as that node has a higher priority than its parent.

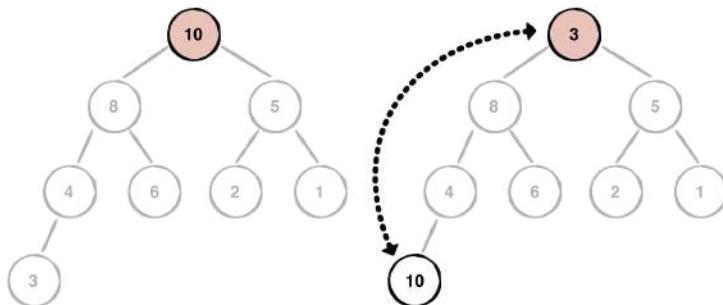
Complexity: The overall complexity of `insert()` is $O(\log n)$. Appending an element in an array takes only $O(1)$, while sifting up elements in a heap takes $O(\log n)$.

That's all there is to inserting an element in a heap but how can you remove an element?

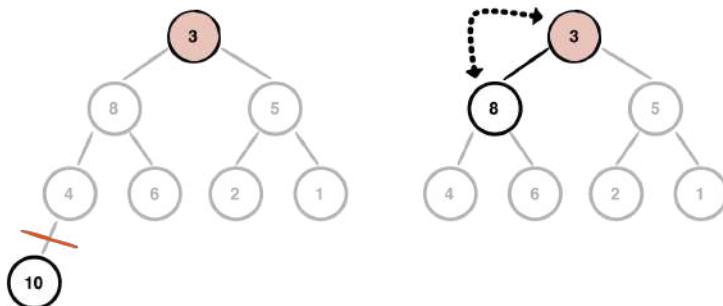
Removing from a heap

A basic remove operation removes the root node from the heap.

Take the following max heap:



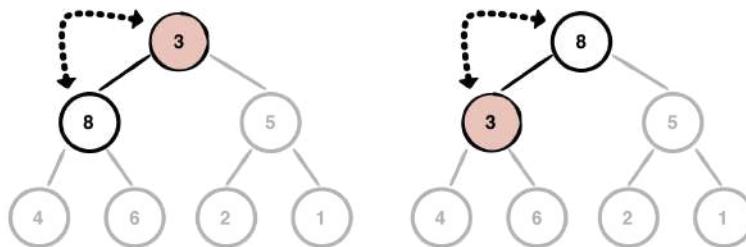
A remove operation will remove the maximum value at the root node. To do so, you must first swap the **root** node with the **last** element in the heap.



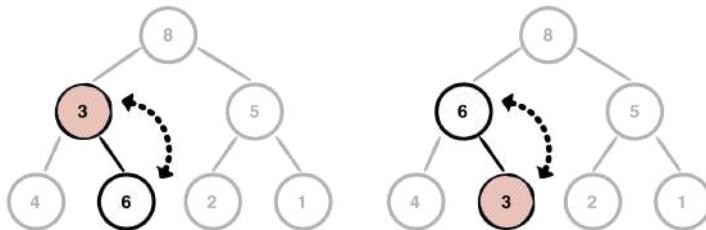
Once you've swapped the two elements, you can remove the last element and store its value so you can return it later.

Now, you must check the max heap's integrity. But first, ask yourself, “*Is it still a max heap?*”

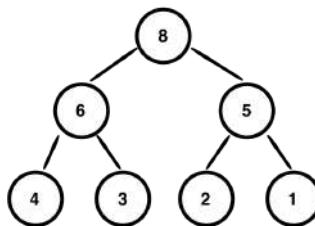
Remember: The rule for a max heap is that the value of every parent node must be larger than or equal to the values of its children. Since the heap no longer follows this rule, you must perform a **sift down**.



To perform a sift down, start from the current value **3** and check its left and right child. If one of the children has a value that is *greater* than the current value, you swap it with the parent. If both children have greater values, you swap the parent with the greater child value.



You have to continue to sift down until the node's value is not larger than the values of its children.



Once you reach the end, you're done, and the max heap's property has been restored.

Implementation of remove

Add the following code to `AbstractHeap`:

```
override fun remove(): Element? {
    if (isEmpty) return null // 1

    Collections.swap(elements, 0, count - 1) // 2
    val item = elements.removeAt(count - 1) // 3
    siftDown(0) // 4
    return item
}
```

Here's how this method works:

1. Check to see if the heap is empty. If it is, return `null`.
2. Swap the root with the last element in the heap.
3. Remove the last element (the maximum or minimum value) and return it.
4. The heap may not be a maxheap or minheap anymore, so you must perform a sift down to make sure it conforms to the rules.

To see how to sift down nodes, add the following method after `remove()`:

```
private fun siftDown(index: Int) {
    var parent = index // 1
    while (true) { // 2
        val left = leftChildIndex(parent) // 3
        val right = rightChildIndex(parent)
        var candidate = parent // 4
        if (left < count &&
            compare(elements[left], elements[candidate]) > 0) {
            candidate = left // 5
        }
        if (right < count &&
            compare(elements[right], elements[candidate]) > 0) {
            candidate = right // 6
        }
        if (candidate == parent) {
            return // 7
        }
        Collections.swap(elements, parent, candidate) // 8
        parent = candidate
    }
}
```

`siftDown()` accepts, as parameter, the index of the element to consider as the parent node to swap with one of the children until it finds the right position. Here's how the method works:

1. Store the parent index.
2. Continue sifting until you return.
3. Get the parent's left and right child index.
4. `candidate` is used to keep track of which index to swap with the parent.
5. If there's a left child, and it has a higher priority than its parent, make it the candidate.
6. If there's a right child, and it has an even greater priority, it will become the candidate instead.
7. If `candidate` is still `parent`, you have reached the end, and no more sifting is required.
8. Swap `candidate` with `parent` and set it as the new parent to continue sifting.

Complexity: The overall complexity of `remove()` is $O(\log n)$. Swapping elements in an array takes only $O(1)$, while sifting down elements in a heap takes $O(\log n)$ time.

Now that you know how to remove from the top of the heap, it's time to learn how to **add** to a heap.

Removing from an arbitrary index

Add the following method to `AbstractHeap` removing all the previous errors:

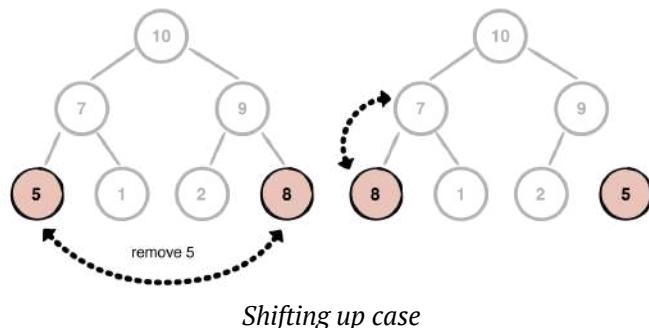
```
override fun remove(index: Int): Element? {
    if (index >= count) return null // 1

    return if (index == count - 1) {
        elements.removeAt(count - 1) // 2
    } else {
        Collections.swap(elements, index, count - 1) // 3
        val item = elements.removeAt(count - 1) // 4
        siftDown(index) // 5
        siftUp(index)
        item
    }
}
```

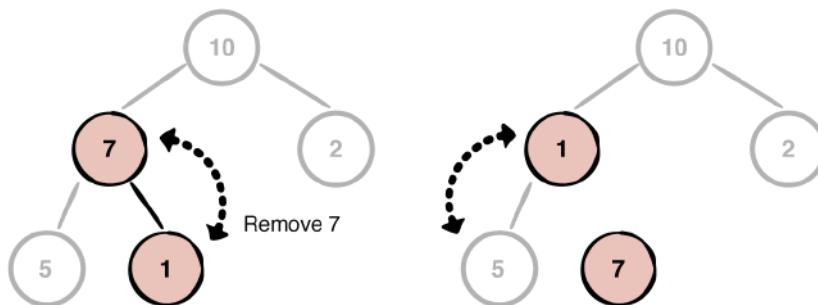
To remove any element from the heap, you need an index. Let's go over how this works:

1. Check to see if the index is within the bounds of the array. If not, return `null`.
2. If you're removing the last element in the heap, you don't need to do anything special. Simply remove and return the element.
3. If you're not removing the last element, first swap the element with the last element.
4. Then, return and remove the last element.
5. Finally, perform both a sift down and a sift up to adjust the heap.

So, why do you have to perform a sift down *and* a sift up?



Assume that you're trying to remove 5. You swap 5 with the last element, which is 8. You now need to perform a sift up to satisfy the max heap property.



Shifting down case

Now, assume that you're trying to remove 7. You swap 7 with the last element, which is 1. You now need to perform a sift down to satisfy the max heap property.

Complexity: Removing an arbitrary element from a heap is an $O(\log n)$ operation.

But how do you find the index of the element you want to delete?

Searching for an element in a heap

To find the index of the element that you want to delete, you must perform a search on the heap. Unfortunately, heaps are not designed for fast searches.

With a binary search tree, you can perform a search in $O(\log n)$ time, but since heaps are built using an array, and the node ordering in an array is different, you can't even perform a binary search.

Complexity: To search for an element in a heap is, in the worst-case, an $O(n)$ operation, since you may have to check every element in the array:

```
private fun index(element: Element, i: Int): Int? {
    if (i >= count) {
        return null // 1
    }
}
```

```
    }
    if (sort(element, elements[i])) {
        return null // 2
    }
    if (element == elements[i]) {
        return i // 3
    }
    val leftChildIndex = index(element, leftChildIndex(i))
    if (leftChildIndex != null) return leftChildIndex // 4

    val rightChildIndex = index(element, rightChildIndex(i))
    if (rightChildIndex != null) return rightChildIndex // 5

    return null // 6
}
```

Here's how this implementation works:

1. If the index is greater than or equal to the number of elements in the array, the search failed. Return `null`.
2. Check to see if the element that you're looking for has higher priority than the current element at index `i`. If it does, the element you're looking for cannot possibly be lower in the heap.
3. If the element is equal to the element at index `i`, return `i`.
4. Recursively search for the element starting from the left child of `i`.
5. Recursively search for the element starting from the right child of `i`.
6. If both searches failed, the search failed. Return `null`.

Note: Although searching takes $O(n)$ time, you have made an effort to optimize searching by taking advantage of the heap's property and checking the priority of the element when searching.

Heapify an array

In the previous implementations of the Heap data structure, you have used an `ArrayList`. Other implementation could use an `Array` setting a max dimension for it. Making an existing array following the heap properties is an operation usually called **heapify**.

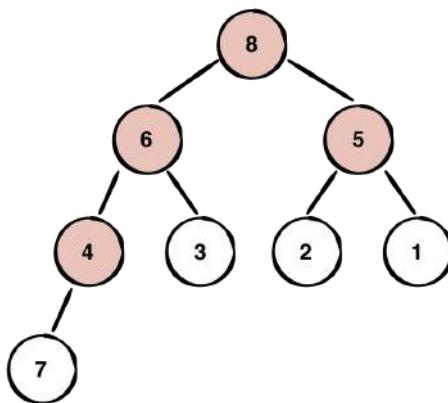


In order to implement this function add this code to `AbstractHeap`.

```
protected fun heapify(values: ArrayList<Element>) {
    elements = values
    if (!elements.isEmpty()) {
        (count / 2 downTo 0).forEach {
            siftDown(it)
        }
    }
}
```

If a non-empty array is provided, you use this as the elements for the heap. To satisfy the heap's property, you loop through the array backward, starting from the first non-leaf node, and sift down all parent nodes.

You loop through only half of the elements because there's no point in sifting down **leaf** nodes, only parent nodes.



$$\text{Number of parents} = \text{total number of elements} / 2 \\ 4 = 8 / 2$$

With this method you can add this code to `ComparableHeapImpl`:

```
companion object {
    fun <Element : Comparable<Element>> create(
        elements: ArrayList<Element>
    ): Heap<Element> {
        val heap = ComparableHeapImpl<Element>()
        heap.heapify(elements)
        return heap
    }
}
```

...and this code to `ComparatorHeapImpl`

```
companion object {
    fun <Element> create(
        elements: ArrayList<Element>,
        comparator: Comparator<Element>
    ): Heap<Element> {
        val heap = ComparatorHeapImpl(comparator)
        heap.heapify(elements)
        return heap
    }
}
```

This code allows you to define a **static factory method** and create a `Heap` implementation starting from a given array and test your implementations.

Testing

You now have all the necessary tools to create and test a `Heap`. You can start using this code in order to create a max-heap of `Comparable` objects represented by `Int` values.

```
fun main() {
    val array = arrayListOf(1, 12, 3, 4, 1, 6, 8, 7) // 1
    val priorityQueue = ComparableHeapImpl.create(array) // 2
    while (!priorityQueue.isEmpty) { // 3
        println(priorityQueue.remove())
    }
}
```

In the previous code you:

1. create an `ArrayList` of `Ints`
2. using the array in order to create a `ComparableHeapImpl`
3. remove and print the max value until the `Heap` is empty

Notice that the elements are removed largest to smallest, and the following numbers are printed to the console:

```
12
8
7
6
4
3
```

```
1  
1
```

If you want to create a min-heap you can replace the previous code in `main()` with the following:

```
fun main() {  
    val array = arrayListOf(1, 12, 3, 4, 1, 6, 8, 7) // 1  
    val inverseComparator = object : Comparator<Int> { // 2  
        override fun compare(o1: Int, o2: Int): Int =  
            o2.compareTo(o1)  
    }  
    val minHeap = ComparatorHeapImpl.create(array,  
        inverseComparator) // 3  
    while (!minHeap.isEmpty) { // 4  
        println(minHeap.remove())  
    }  
}
```

In this case you:

1. create an `ArrayList` of `Ints`
2. create an implementation of the `Comparator<Int>` which implements the inverse order for `Int`
3. using the array and the comparator in order to create a `ComparatorHeapImpl`
4. remove and print the value with highest priority (whose value this time is the lowest) until the `Heap` is empty

Running this code you'll get the the following output:

```
1  
1  
3  
4  
6  
7  
8  
12
```

Challenges

Challenge 1: Find the nth smallest value

Write a function to find the nth smallest integer in an unsorted array. For example:

```
val integers = arrayListOf(3, 10, 18, 5, 21, 100)
```

If $n = 3$, the result should be 10 .

Solution 1

There are many ways to solve for the nth smallest integer in an unsorted array. For example, you could choose a sorting algorithm you learned in this chapter, sort the array, and grab the element at the nth index.

Here's how you would obtain the nth smallest element using a minheap:

```
fun getNthSmallestElement(n: Int): Element? {
    var current = 1 // 1
    while (!isEmpty) { // 2
        val element = remove() // 3
        if (current == n) { // 4
            return element
        }
        current += 1 // 5
    }
    return null // 6
}
```

Here's how it works:

1. `current` tracks the nth smallest element.
2. As long as the heap is not empty, continue to remove elements.
3. Remove the root element from the heap.
4. Check to see if you reached the nth smallest element. If so, return the element.
5. If not, increment `current`.
6. Return `null` if the heap is empty.

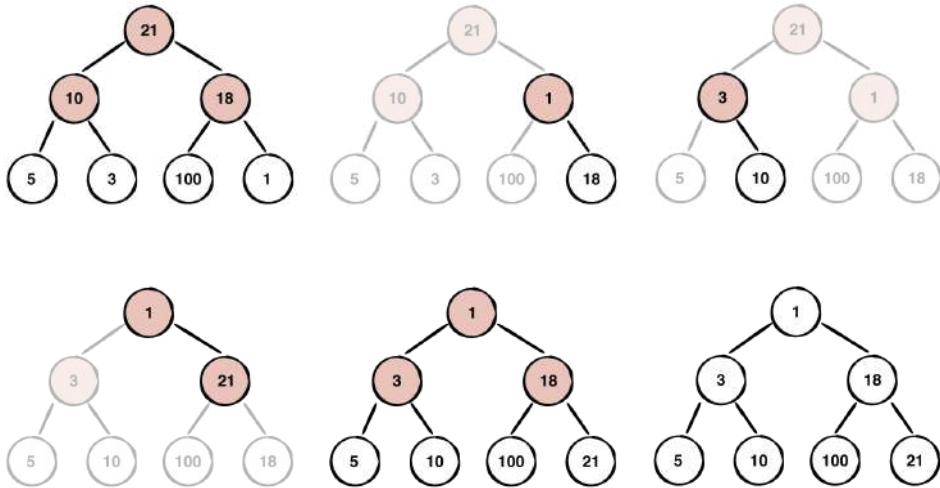
Building a heap takes $O(n)$. Every element removal from the heap takes $O(\log n)$. Keep in mind that you're also doing this n times. The overall time complexity is $O(n \log n)$.

Challenge 2: The min heap visualization

Given the following array list, visually construct a minheap. Provide a step-by-step diagram of how the minheap is constructed.

```
arrayListOf(3, 10, 18, 5, 21, 100)
```

Solution 2



Challenge 3: Heap merge

Write a method that combines two heaps.

Solution 3

Add this as an additional function for your `AbstractHeap` class after defining the same operation on the `Heap` interface:

```
override fun merge(heap: AbstractHeap<Element>) {
    elements.addAll(heap.elements)
    buildHeap()
}

private fun buildHeap() {
    if (!elements.isEmpty()) {
        (count / 2 downTo 0).forEach {
            siftDown(it)
        }
    }
}
```

To merge two heaps, you first combine both arrays which takes $O(m)$, where m is the length of the heap you are merging.

Building the heap takes $O(n)$. Overall the algorithm runs in $O(n)$.

Challenge 4: Min heap check

Write a function to check if a given array is a minheap.

Solution 4

To check if the given array is a minheap, you only need to go through the parent nodes of the binary heap. To satisfy the minheap, every parent node must be less than or equal to its left and right child node.

Here's how you can determine if an array is a minheap:

```
override fun isMinHeap(): Boolean {
    if (isEmpty) return true // 1
    (count / 2 - 1 downTo 0).forEach {
        // 2
        val left = leftChildIndex(it) // 3
        val right = rightChildIndex(it)
        if (left < count &&
            compare(elements[left], elements[it]) < 0) { // 4
            return false
        }
        if (right < count
            && compare(elements[right], elements[it]) < 0) { // 5
            return false
        }
    }
    return true // 6
}
```

Here's how it works:

1. If the array is empty, it's a minheap.
2. Go through all of the parent nodes in the array in reverse order.
3. Get the left and right child index.
4. Check to see if the left element is less than the parent.
5. Check to see if the right element is less than the parent.
6. If every parent-child relationship satisfies the minheap property, return `true`.

The time complexity of this solution is $O(n)$. This is because you still have to go through every element in the array.

Key points

- Here's a summary of the algorithmic complexity of the heap operations you implemented in this chapter:

Heap Data Structure

Operations	Time Complexity
remove	$O(\log n)$
insert	$O(\log n)$
search	$O(n)$
peek	$O(1)$

Heap operation time complexity

- The heap data structure is good for maintaining the highest or lowest priority element.
- Every time you insert or remove items from the heap, you must check to see if it satisfies the rules of the priority.

Chapter 13: Priority Queues

By Irina Galata

Queues are lists that maintain the order of elements using *first in, first out* (FIFO) ordering. A **priority queue** is another version of a queue. However, instead of using FIFO ordering, elements are dequeued in priority order.

A priority queue can have either a:

- **Max-priority:** The element at the front is always the largest.
- **Min-priority:** The element at the front is always the smallest.

A priority queue is especially useful when you need to identify the maximum or minimum value within a list of elements.

In this chapter, you'll learn the benefits of a priority queue and build one by leveraging the existing queue and heap data structures that you studied in previous chapters.



Applications

Some useful applications of a priority queue include:

- **Dijkstra's algorithm:** Uses a priority queue to calculate the minimum cost.
- **A* pathfinding algorithm:** Uses a priority queue to track the unexplored routes that will produce the path with the shortest length.
- **Heap sort:** Many heap sorts use a priority queue.
- **Huffman coding:** Useful for building a compression tree. A min-priority queue is used to repeatedly find two nodes with the smallest frequency that don't yet have a parent node.

Priority queues have many more applications and practical uses; the list above represents only a handful.

Common operations

In Chapter 5, “Queues”, you established the following interface for queues:

```
interface Queue<T> {  
    fun enqueue(element: T): Boolean  
    fun dequeue(): T?  
    val count: Int  
        get  
    val isEmpty: Boolean  
        get() = count == 0  
    fun peek(): T?  
}
```

A priority queue has the same operations as a normal queue, so only the implementation will be different.

The priority queue will implement the Queue interface and the common operations:

- **enqueue:** Inserts an element into the queue. Returns true if the operation is successful.

- **dequeue**: Removes the element with the highest priority and returns it. Returns `null` if the queue is empty.
- **count**: Property for the number of items in the queue.
- **isEmpty**: Checks if the queue is empty. The implementation just checks if the `count` property is 0.
- **peek**: Returns the element with the highest priority without removing it. Returns `null` if the queue is empty.

You're ready to look at different ways to implement a priority queue.

Implementation

You can create a priority queue in the following ways:

1. **Sorted array**: This is useful to obtain the maximum or minimum value of an element in $O(1)$ time. However, insertion is slow and requires $O(n)$ because you have to search the right position for every element you insert.
2. **Balanced binary search tree**: This is useful in creating a double-ended priority queue, which features getting both the minimum and maximum value in $O(\log n)$ time. Insertion is better than a sorted array, also in $O(\log n)$.
3. **Heap**: This is a natural choice for a priority queue. A heap is more efficient than a sorted array because a heap only needs to be partially sorted. All heap operations are $O(\log n)$ except extracting the min value from a min priority heap is a lightning-fast $O(1)$. Likewise, extracting the max value from a max priority heap is also $O(1)$.

Next, you'll look at how to use a heap to create a priority queue.

To get started, open the starter project. Inside, you'll notice the following files:

1. **Heap.kt**: The heap data structure (from the previous chapter) that you'll use to implement the priority queue.
2. **Queue.kt**: Contains the interface that defines a queue.

Add the following abstract class:

```
// 1
abstract class AbstractPriorityQueue<T> : Queue<T> {
```

```
// 2
abstract val heap: Heap<T>
    get
    // more to come ...
}
```

Here's a closer look at the code:

1. `AbstractPriorityQueue` implements the `Queue` interface and is generic in the type `T`. It's an abstract class because you want to manage comparison using either `Comparable<T>` objects or an external `Comparator<T>` implementation.
2. You're going to use a `Heap<T>`, so you need an abstract property that the specific implementation will define.

To implement the `Queue` interface, add the following to `AbstractPriorityQueue`:

```
// 1
override fun enqueue(element: T): Boolean {
    heap.insert(element)
    return true
}

// 2
override fun dequeue() = heap.remove()

// 3
override val count: Int
    get() = heap.count

// 4
override fun peek() = heap.peek()
```

The heap is a perfect candidate for a priority queue. To implement the operations of a priority queue, you need to call various methods of a heap.

1. By calling `enqueue()`, you add the element into the heap using `insert()`, which guarantees to arrange data internally so that the one with the highest priority is ready to extract. The overall complexity of `enqueue()` is the same as `insert()`: $O(\log n)$.
2. By calling `dequeue()`, you remove the root element from the heap using `remove()`. The Heap guarantees to get the one with the highest priority. The overall complexity of `dequeue()` is the same as `remove()`: $O(\log n)$.
3. `count` uses the same property of the heap.

4. `peek()` delegates to the same method of the heap.

Using Comparable objects

`AbstractPriorityQueue<T>` implements the `Queue<T>` interface delegating to a `Heap<T>`. You can implement this using either `Comparable<T>` objects or a `Comparator<T>`. In this example, you'll use the former.

Add the following code to `PriorityQueue.kt`.

```
class ComparablePriorityQueueImpl<T : Comparable<T>> :  
    AbstractPriorityQueue<T>() {  
  
    override val heap = ComparableHeapImpl<T>()  
}
```

Here, you implement `heap` using a `ComparableHeapImpl<T>` object. The `ComparablePriorityQueueImpl<T>` needs an object that implements the `Comparable<T>` interface.

To test this implementation, add the following code to `Main.kt`:

```
"max priority queue" example {  
    // 1  
    val priorityQueue = ComparablePriorityQueueImpl<Int>()  
    // 2  
    arrayListOf(1, 12, 3, 4, 1, 6, 8, 7).forEach {  
        priorityQueue.enqueue(it)  
    }  
    // 3  
    while (!priorityQueue.isEmpty) {  
        println(priorityQueue.dequeue())  
    }  
}
```

In this example, you:

1. Create a `ComparablePriorityQueueImpl<Int>` using `Int` as generic type value which is `Comparable<Int>`.
2. Enqueue the value from an unsorted array into the priority queue.
3. Dequeue all of the values from the priority queue.

When you run the code, notice the elements are removed largest to smallest. The following is printed to the console:

```
---Example of max priority queue---
12
8
7
6
4
3
1
1
```

Using Comparator objects

Providing different `Comparator<T>` interface implementations allows you to choose the priority criteria.

Add the following code to `PriorityQueue.kt`.

```
class ComparatorPriorityQueueImpl<T>(
    private val comparator: Comparator<T>
) : AbstractPriorityQueue<T>() {

    override val heap = ComparatorHeapImpl(comparator)
}
```

Here, the only difference is the value provided to `heap`, which is now a `ComparatorHeapImpl<T>` and needs a `Comparator<T>` that you provide as a constructor parameter.

To test this implementation, add the following code to `main()` inside `Main.kt`:

```
"min priority queue" example {
    // 1
    val stringLengthComparator = object : Comparator<String> {
        override fun compare(o1: String?, o2: String?): Int {
            val length1 = o1?.length ?: -1
            val length2 = o2?.length ?: -1
            return length1 - length2
        }
    }
    // 2
    val priorityQueue =
        ComparatorPriorityQueueImpl(stringLengthComparator)
    // 3
    arrayListOf("one", "two", "three", "forty", "five", "six",
    "seven", "eight", "nine").forEach {
```

```
    priorityQueue.enqueue(it)
}
// 4
while (!priorityQueue.isEmpty) {
    println(priorityQueue.dequeue())
}
}
```

In this example, you:

1. Create a `Comparator<String>` implementation that compares `String` based on the length from the longest to the shortest.
2. Create a `ComparatorPriorityQueueImpl` using the previous comparator in the constructor.
3. Enqueue value from an unsorted array as `String` into the priority queue.
4. Dequeue all the values from the priority queue.

When you run the code, you'll see this output where the `String` objects are sorted from the longest to the shortest.

```
---Example of min priority queue---
forty
three
eight
seven
nine
five
two
one
six
```



Challenges

Challenge 1: Constructing ArrayList priority queues

You learned to use a heap to construct a priority queue by implementing the Queue interface. Now, construct a priority queue using an ArrayList:

```
interface Queue<T> {  
    fun enqueue(element: T): Boolean  
    fun dequeue(): T?  
    val count: Int  
        get  
    val isEmpty: Boolean  
        get() = count == 0  
    fun peek(): T?  
}
```

Solution 1

Recall that a priority queue dequeues element in priority order. It could either be a min or max priority queue. To make an array-based priority queue, you need to implement the Queue interface. Instead of using a heap, you can use an array list.

First, add the following code to **PriorityQueueArray.kt**:

```
// 1  
abstract class AbstractPriorityQueueArrayList<T> : Queue<T> {  
  
    // 2  
    protected val elements = ArrayList<T>()  
  
    // 3  
    abstract fun sort()  
  
    // more to come ...  
}
```

Here, you:

1. Define the `AbstractPriorityQueueArrayList<T>` abstract class implementing

the Queue<T> interface.

2. Define the elements property of type ArrayList<T> as protected so it can be accessed by the classes extending this.
3. The sort abstract function is the one you're going to implement in different ways depending on the usage of Comparable<T> objects or a Comparator<T>.

With this code, some of the Queue<T> operations come for free, so add the following code:

```
override val count: Int
    get() = elements.size

override fun peek() = elements.firstOrNull()
```

Here, you're assuming that the ArrayList<T> is always sorted, and if it's not empty, it always contains the element with the highest priority in position 0. This assumption allows you to implement the dequeue operation using this code:

```
override fun dequeue() =
    if (isEmpty) null else elements.removeAt(0)
```

It's important to know how the dequeue operation is $O(n)$ because the removal of an item in position 0 requires the shift of all of the other elements. A possible optimization, which you can try as exercise, is to put the element with the highest priority in the last position so that you don't have to shift any elements but instead reduce the size by 1.

Next, add the enqueue method. This is the one responsible for the sorting:

```
override fun enqueue(element: T): Boolean {
    // 1
    elements.add(element)
    // 2
    sort()
    // 3
    return true
}
```

To enqueue an element into an array-based priority queue, this code does the following:

1. Appends the element in the ArrayList.
2. Sorts the elements into the ArrayList using the sort function.

3. Returns true because the element was inserted with success.

The overall time complexity here is the complexity of the `sort` implementation, because the `add` operation of the `ArrayList` is $O(1)$.

Before implementing `sort()`, add this code so you can print the priority queue in a nice format:

```
override fun toString() = elements.toString()
```

You can now provide different realizations for the `AbstractPriorityQueueArrayList<T>` class and the `sort` operation.

To manage `Comparable<T>` objects, add the following code:

```
class ComparablePriorityQueueArrayList<T : Comparable<T>> : AbstractPriorityQueueArrayList<T>() {
    override fun sort() {
        Collections.sort(elements)
    }
}
```

Here, you implement `sort()` using the same method of the `Collections` class. The complexity, in this case, is $O(n \log n)$; it's the same if you want to use a `Comparator<T>`, which you can do using the following code:

```
class ComparatorPriorityQueueArrayList<T>(
    private val comparator: Comparator<T>
) : AbstractPriorityQueueArrayList<T>() {
    override fun sort() {
        Collections.sort(elements, comparator)
    }
}
```

Can you do better? Sure! If you always insert the new item in the right position, you have to shift all of the other elements — and this can be done in $O(n)$. You can now write this implementation for `Comparable<T>` objects:

```
class CustomPriorityQueueArrayList<T : Comparable<T>> : AbstractPriorityQueueArrayList<T>() {
    override fun sort() {
        var index = count - 2
        while (index >= 0 &&
            elements[index + 1].compareTo(elements[index]) > 0) {
            swap(index, index + 1)
            index--
        }
    }
}
```

```
private fun swap(i: Int, j: Int) {
    val tmp = elements[i]
    elements[i] = elements[j]
    elements[j] = tmp
}
```

This is an $O(n)$ operation since you have to shift the existing elements to the left by one until you find the right position.

Congratulations, you now have an array-based priority queue.

To test the priority queue, add the following code to `main()`:

```
"max priority array list based queue" example {
    val priorityQueue = CustomPriorityQueueArrayList<Int>()
    arrayListOf(1, 12, 3, 4, 1, 6, 8, 7).forEach {
        priorityQueue.enqueue(it)
    }
    priorityQueue.enqueue(5)
    priorityQueue.enqueue(0)
    priorityQueue.enqueue(10)
    while (!priorityQueue.isEmpty) {
        println(priorityQueue.dequeue())
    }
}
```

Challenge 2: Sorting

Your favorite concert was sold out. Fortunately, there's a waitlist for people who still want to go. However, the ticket sales will first prioritize someone with a military background, followed by seniority.

Write a `sort` function that returns the list of people on the waitlist by the appropriate priority. Person is provided below and should be put inside `Person.kt`:

```
data class Person(  
    val name: String,  
    val age: Int,  
    val isMilitary: Boolean)
```

Solution 2

Given a list of people on the waitlist, you would like to prioritize the people in the following order:

1. Military background.
2. Seniority, by age.

The best solution for this problem is to put the previous logic into a `Comparator<Person>` implementation and then use the proper priority queue implementation. In this way, you can give `Person` objects different priority providing different `Comparator<Person>` implementations.

Add this code to `Person.kt`:

```
object MilitaryPersonComparator : Comparator<Person> {  
    override fun compare(o1: Person, o2: Person): Int {  
        if (o1.isMilitary && !o2.isMilitary) {  
            return 1  
        } else if (!o1.isMilitary && o2.isMilitary) {  
            return -1  
        } else if (o1.isMilitary && o2.isMilitary) {  
            return o1.age.compareTo(o2.age)  
        }  
        return 0  
    }  
}
```



To test your priority sort function, try a sample data set by adding the following:

```
"concert line" example {  
    val p1 = Person("Josh", 21, true)  
    val p2 = Person("Jake", 22, true)  
    val p3 = Person("Clay", 28, false)  
    val p4 = Person("Cindy", 28, false)  
    val p5 = Person("Sabrina", 30, false)  
    val priorityQueue =  
        ComparatorPriorityQueueImpl(MilitaryPersonComparator)  
        arrayListOf(p1, p2, p3, p4, p5).forEach {  
            priorityQueue.enqueue(it)  
        }  
        while (!priorityQueue.isEmpty) {  
            println(priorityQueue.dequeue())  
        }  
}
```

Running the previous code, you'll get this output:

```
---Example of concert line---  
Jake  
Josh  
Cindy  
Clay  
Sabrina
```

Key points

- A priority queue is often used to find the element in priority order.
- The `AbstractPriorityQueue<T>` implementation creates a layer of abstraction by focusing on key operations of a queue and leaving out additional functionality provided by the heap data structure.
- This makes the priority queue's intent clear and concise. Its only job is to enqueue and dequeue elements, nothing else.
- The `AbstractPriorityQueue<T>` implementation is another good example of **Composition over (implementation) inheritance**.



Section IV: Sorting Algorithms

Putting lists in order is a classical computational problem. Sorting has been studied since the days of vacuum tubes and perhaps even before that. Although you may never need to write your own sorting algorithm — thanks to the highly optimized standard library — studying sorting has many benefits. You'll be introduced, for example, to the all-important technique of divide-and-conquer, stability, and best- and worst-case timing.

The sorting algorithms you'll cover in this section include:

- **Chapter 14: $O(n^2)$ Sorting Algorithms:** $O(n^2)$ time complexity doesn't have great performance, but the sorting algorithms in this category are easy to understand and useful in some scenarios. These algorithms are space-efficient and only require constant $O(1)$ additional memory space. In this chapter, you'll look at the bubble sort, selection sort and insertion sort algorithms.
- **Chapter 15: Merge Sort:** In this chapter, you'll study one of the most important sorting algorithm based on the divide and conquer principle. You'll learn how to split an array, sort it recursively and then merge the two parts together.
- **Chapter 16: Radix Sort:** So far, you've been relying on comparisons to determine the sorting order. In this chapter, you'll look at a completely different model of sorting. Radix sort is a non-comparative algorithm for sorting integers in linear time. There are multiple implementations of radix sort that focus on different problems. To keep things simple, you'll focus on sorting base 10 integers while investigating the least significant digit (LSD) variant of radix sort.
- **Chapter 17: Heap Sort:** Heap sort is another comparison-based algorithm that sorts an array in ascending order using a heap. This chapter builds on the heap concepts presented in Chapter 12, "The Heap Data Structure." Heap sort takes advantage of a heap being, by definition, a partially sorted binary tree.
- **Chapter 18: Quicksort:** Quicksort is another divide-and-conquer technique that introduces the concept of partitions and a pivot to implement high-performance sorting. You'll see that while it's extremely fast for some datasets, for others, it can



be a bit slow.

Studying sorting may seem a bit academic and disconnected to the real world of app development, but understanding the tradeoffs for these simple cases will lead you to a better understanding and let you analyze any algorithm.

Chapter 14: $O(n^2)$ Sorting Algorithms

By Matei řuică

$O(n^2)$ time complexity is not great performance, but the sorting algorithms in this category are easy to understand and useful in some scenarios. These algorithms are space efficient, and they only require constant $O(1)$ additional memory space. For small data sets, these sorts compare favorably against more complex sorts. It's usually not recommended to use $O(n^2)$ in production code, but you'll need to start somewhere, and these algorithms are a great place to start.

In this chapter, you'll look at the following sorting algorithms:

- Bubble sort.
- Selection sort.
- Insertion sort.

All of these are **comparison-based** sorting methods. In other words, they rely on a comparison method, such as the less than operator, to order elements. You measure a sorting technique's general performance by counting the number of times this comparison gets called.

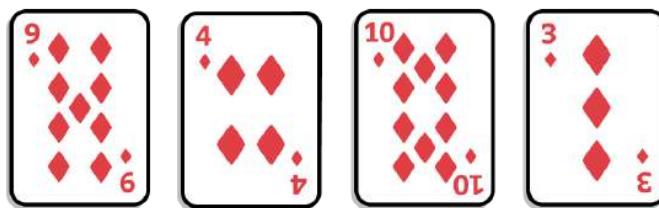


Bubble sort

One of the simplest sorts is the **bubble sort**. The bubble sort repeatedly compares adjacent values and swaps them, if needed, to perform the sort. The larger values in the set will, therefore, *bubble up* to the end of the collection.

Example

Consider the following hand of cards, and suppose you want to sort them in ascending order:

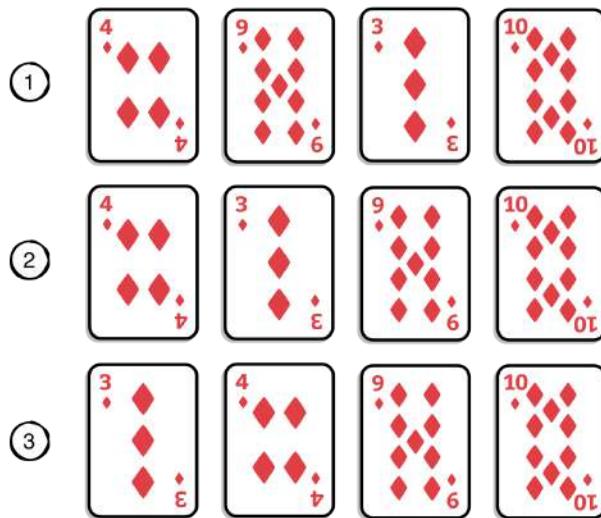


A single pass of the bubble sort algorithm consists of the following steps:

- Start at the beginning of the collection. Compare 9 with the next value in the array, which is 4. Since you're sorting in ascending order, and 9 is greater than 4, you need to swap these values. The collection then becomes [4, 9, 10, 3].
- Move to the next index in the collection. You're now comparing 9 and 10. These are in order, so there's nothing to do.
- Move to the next index in the collection. Compare 10 and 3. You need to swap these values. The collection then becomes [4, 9, 3, 10].
- You'll stop there as it's pointless to move to the next position because there's nothing left to sort.

A single pass of the algorithm seldom results in a complete ordering. You can see for yourself that the cards above are not yet completely sorted. 9 is greater than 3, but the 3 of diamonds still comes before the 9 of diamonds. It will, however, cause the largest value (10) to bubble up to the end of the collection.

Subsequent passes through the collection does the same for 9 and 4 respectively. Here's an illustration of the passes. You can see that after each pass, the collection has fewer cards in the wrong position.



The sort is only complete when you can perform a full pass over the collection without having to swap any values. At worst, this will require $n-1$ passes, where n is the count of members in the collection. For the cards example, you had four cards, so you needed three passes to make sure everything's in order.

Implementation

To get started, open the starter project for this chapter. Because you already know that you'll need to do a lot of swapping, the first thing you need to do is write an extension for `ArrayList` to swap elements.

Open `Utils.kt` and add:

```
fun <T> ArrayList<T>.swapAt(first: Int, second: Int) {
    val aux = this[first]
    this[first] = this[second]
    this[second] = aux
}
```

With this useful addition, start working on the `bubbleSort()` extension.

Since `ArrayList` is mutable, you're free to swap its elements. In `src › bubblesort`, create a new file named `BubbleSort.kt`. Add the following function:

```
fun <T : Comparable<T>> ArrayList<T>.bubbleSort(showPasses: Boolean = false) {
    // 1
    if (this.size < 2) return
    // 2
    for (end in (1 until this.size).reversed()) {
        var swapped = false
        // 3
        for (current in 0 until end) {
            if (this[current] > this[current + 1]) {
                // 4
                this.swapAt(current, current + 1)
                swapped = true
            }
        }
        // 5
        if (showPasses) println(this)
        // 6
        if (!swapped) return
    }
}
```

Here's how it works:

1. There's no need to sort the collection when it has less than two elements. One element is sorted by itself; zero elements don't require an order.
2. A single-pass will bubble the largest value to the end of the collection. Every pass needs to compare one less value than in the previous pass, so you shorten the array by one with each pass.
3. This loop performs a single pass starting from the first element and going up until the last element not already sorted. It compares every element with the adjacent value.
4. Next, the algorithm swaps the values if needed and marks this in `swapped`. This is important later because it'll allow you to exit the sort as early as you can detect the list is sorted.
5. This prints out how the list looks after each pass. This step has nothing to do with the sorting algorithm, but it will help you visualize how it works. You can remove it (and the function parameter) after you understand the sorting algorithm.



6. If no values were swapped this pass, the collection is assumed sorted, and you can exit early.

Try it out. Open **Main.kt** and write the following inside `main()`:

```
"bubble sort" example {
    val list = arrayListOf(9, 4, 10, 3)
    println("Original: $list")
    list.bubbleSort(true)
    println("Bubble sorted: $list")
}
```

Since you used the `showPasses` trick, you'll see the following output:

```
---Example of bubble sort---
Original: [9, 4, 10, 3]
[4, 9, 3, 10]
[4, 3, 9, 10]
[3, 4, 9, 10]
Bubble sorted: [3, 4, 9, 10]
```

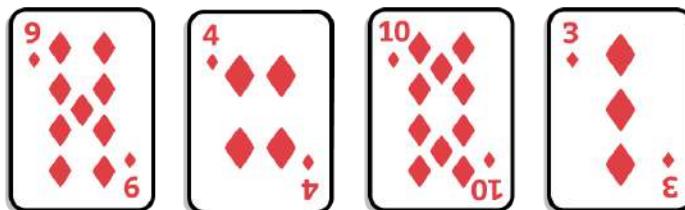
Bubble sort has a *best* time complexity of $O(n)$ if it's already sorted, and a *worst* and *average* time complexity of $O(n^2)$, making it one of the *least* appealing sorts.

Selection sort

Selection sort follows the basic idea of bubble sort but improves upon this algorithm by reducing the number of `swapAt` operations. Selection sort only swaps at the end of each pass. You'll see how that works in the following implementation.

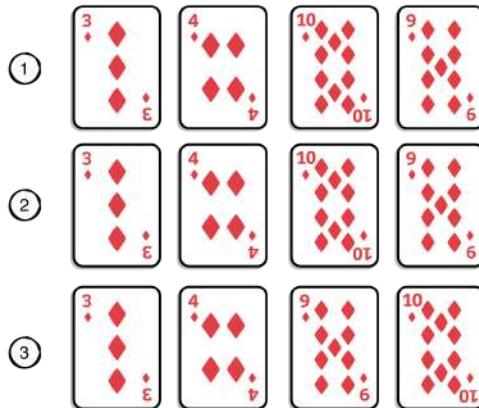
Example

Assume you have the following hand of cards:



During each pass, selection sort finds the lowest unsorted value and swaps it into place:

1. 3 is found as the lowest value, so it's swapped with 9.
2. The next lowest value is 4, and it's already in the right place.
3. Finally, 9 is swapped with 10.



Implementation

In `src > selectionsort`, create a new file named **SelectionSort.kt**. Since you added the `swapAt()` extension for the bubble sort, you'll leverage it here too.

Note: If you didn't already add `swapAt()`, go back and copy it into **Utils.kt**.

After you confirm that you can swap list elements, write the following inside the file:

```
fun <T : Comparable<T>> ArrayList<T>.selectionSort(showPasses: Boolean = false) {  
    if (this.size < 2) return  
    // 1  
    for (current in 0 until (this.size - 1)) {  
        var lowest = current  
        // 2  
        for (other in (current + 1) until this.size) {  
            if (this[lowest] > this[other]) {  
                lowest = other  
            }  
        }  
        swapAt(current, lowest)  
    }  
}
```

```

    }
    // 3
    if (lowest != current) {
        this.swapAt(lowest, current)
    }
    // 4
    if(showPasses) println(this)
}
}

```

Here's what's going on:

1. You perform a pass for every element in the collection, except for the last one. There's no need to include the last element because if all other elements are in their correct order, the last one will be as well.
2. In every pass, you go through the remainder of the collection to find the element with the lowest value.
3. If that element is not the current element, swap them.
4. This optional step shows you how the list looks after each step when you call the function with `showPasses` set to `true`. You can remove this and the parameter once you understand the algorithm.

Try it out. Go back to the main playground page and add the following:

```

"selection sort" example {
    val list = arrayListOf(9, 4, 10, 3)
    println("Original: $list")
    list.selectionSort(true)
    println("Bubble sorted: $list")
}

```

You'll see the following output in your console:

```

---Example of selection sort---
Original: [9, 4, 10, 3]
[3, 4, 10, 9]
[3, 4, 10, 9]
[3, 4, 9, 10]
Bubble sorted: [3, 4, 9, 10]

```

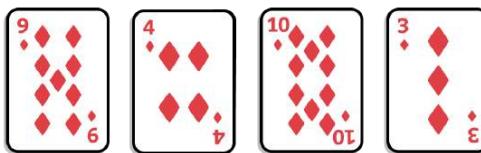
Like bubble sort, selection sort has a *worst* and *average* time complexity of $O(n^2)$, which is fairly dismal. Unlike the bubble sort, it also has the *best* time complexity of $O(n^2)$. Despite this, it performs better than bubble sort because it performs only $O(n)$ swaps — and the best thing about it is that it's a simple one to understand.

Insertion sort

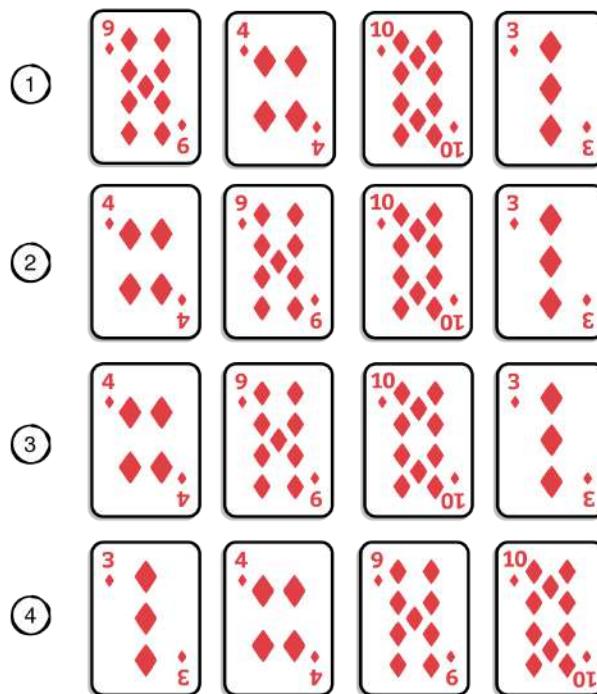
Insertion sort is a more useful algorithm. Like bubble sort and selection sort, insertion sort has an *average* time complexity of $O(n^2)$, but the performance of insertion sort can vary. The more the data is already sorted, the less work it needs to do. Insertion sort has a *best* time complexity of $O(n)$ if the data is already sorted.

Example

The idea of insertion sort is like how you'd sort a hand of cards. Consider the following hand:



Insertion sort will iterate *once* through the cards, from left to right. Each card is shifted to the left until it reaches its correct position.



1. You can ignore the first card, as there are no previous cards to compare it with.
2. Next, you compare 4 with 9 and shift 4 to the left by swapping positions with 9.
3. 10 doesn't need to shift, as it's in the correct position compared to the previous card.
4. Finally, 3 shifts to the front by comparing and swapping it with 10, 9 and 4, respectively.

It's worth pointing out that the best case scenario for insertion sort occurs when the sequence of values are already in sorted order, and no left shifting is necessary.

Implementation

In `src > selectionsort` of your starter project, create a new file named `InsertionSort.kt`. Write the following inside of the file:

```
fun <T : Comparable<T>> ArrayList<T>.insertionSort(showPasses: Boolean = false) {
    if (this.size < 2) return
    // 1
    for (current in 1 until this.size) {
        // 2
        for (shifting in (1..current).reversed()) {
            // 3
            if (this[shifting] < this[shifting - 1]) {
                this.swapAt(shifting, shifting - 1)
            } else {
                break
            }
        }
        // 4
        if (showPasses) println(this)
    }
}
```

Here's what you did:

1. Insertion sort requires you to iterate from left to right, once. This loop does that.
2. Here, you run backward from the current index so you can shift left as needed.
3. Keep shifting the element left as long as necessary. As soon as the element is in position, break the inner loop and start with the next element.
4. This is the same trick you used with the other sort algorithms; it shows you the passes. Remember that this is not part of the sorting algorithm.

Head back to `main()` in **Main.kt** and write the following at the bottom:

```
"insertion sort" example {  
    val list = arrayListOf(9, 4, 10, 3)  
    println("Original: $list")  
    list.insertionSort(true)  
    println("Bubble sorted: $list")  
}
```

You'll see the following console output:

```
---Example of insertion sort---  
Original: [9, 4, 10, 3]  
[4, 9, 10, 3]  
[4, 9, 10, 3]  
[3, 4, 9, 10]  
Bubble sorted: [3, 4, 9, 10]
```

Insertion sort is one of the fastest sorting algorithms when some of the data is already sorted, but this isn't true for *all* sorting algorithms. In practice, a lot of data collections will already be mostly — if not entirely — sorted, and an insertion sort will perform quite well in those scenarios.

Generalization

In this section, you'll generalize these sorting algorithms for list types other than `ArrayList`. Exactly which list types won't matter, as long as they're mutable since you need to be able to swap elements. The changes are small and simple but important. You always want your algorithms to be as generic as possible.

You'll go through the code in the exact order you've written it, starting with **Utils.kt**.

Open the file and swap the `swapAt()` definition with this one:

```
fun <T> MutableList<T>.swapAt(first: Int, second: Int)
```

Head back to **BubbleSort.swift** and update the function definition to the following:

```
fun <T : Comparable<T>> MutableList<T>.bubbleSort(showPasses: Boolean = false)
```

Are you starting to see a pattern here?

Because you didn't use any `ArrayList` specific methods in the algorithm, you can change the `ArrayList` usages with `MutableList`. Do the same with the selection sort, in `SelectionSort.kt`:

```
fun <T : Comparable<T>> MutableList<T>.selectionSort(showPasses: Boolean = false)
```

Finally, deal with the insertion sort. Open `InsertionSort.kt` and replace the extension function definition with this one:

```
fun <T : Comparable<T>> MutableList<T>.insertionSort(showPasses: Boolean = false)
```

Notice that you don't need to change the examples in `Main.kt`. That's because the `ArrayList` is a `MutableList`. Since your algorithms are now generic, they can handle any implementation of the `MutableList`.

Generalization won't always be so easy, but it's something you need to do. You want your algorithm to work with as many data structures as possible. With a bit of practice, generalizing these algorithms becomes a fairly mechanical process.

In the next chapters, you'll take a look at sorting algorithms that perform better than $O(n^2)$. Up next is a sorting algorithm that uses a classical algorithm approach known as **divide and conquer** — merge sort!

Challenges

Challenge 1: To the left, to the left

Given a list of Comparable elements, bring all instances of a given value in the list to the right side of the array.

Solution 1

The trick to this problem is to find the elements that need to be moved and shift everything else to the left. Then, return to the position where the element was before, and continue searching from there.

```
fun<T: Comparable<T>> MutableList<T>.rightAlign(element: T) {  
    // 1  
    if(this.size < 2) return  
    // 2  
    var searchIndex = this.size - 2  
    while(searchIndex >= 0) {  
        // 3  
        if(this[searchIndex] == element) {  
            // 4  
            var moveIndex = searchIndex  
            while(moveIndex < this.size - 1 &&  
                  this[moveIndex + 1] != element) {  
                swapAt(moveIndex, moveIndex + 1)  
                moveIndex++  
            }  
        }  
        // 5  
        searchIndex--  
    }  
}
```

Here's a breakdown of this moderately complicated function:

1. If there are less than two elements in the list, there's nothing to do.
2. You leave the last element alone and start from the previous one. Then, you go to the left (decreasing the index), until you reach the beginning of the list when the `searchIndex` is 0.
3. You're looking for elements that are equal to the one in the function parameter.
4. Whenever you find one, you start shifting it to the right until you reach another element equal to it or the end of the list. Remember, you already implemented `swapAt()`; don't forget to increment `moveIndex`.

5. After you're done with that element, move `searchIndex` to the left again by decrementing it.

The tricky part here is to understand what sort of capabilities you need. Since you need to make changes to the underlying storage, this function is only available to `MutableList` types.

To complete this algorithm efficiently, you need backward index traversal, which is why you can't use any generic `MutableCollection`.

Finally, you also need the elements to be `Comparable` to target the appropriate values.

The time complexity of this solution is $O(n)$.

Challenge 2: Duplicate finder

Given a list of Comparable elements, return the largest element that's a duplicate in the list.

Solution 2

Finding the biggest duplicated element is rather straightforward. To make it even easier, you can sort the list with one of the methods you've already implemented.

```
fun<T: Comparable<T>> MutableList<T>.biggestDuplicate(): T? {  
    // 1  
    this.selectionSort()  
    // 2  
    for(i in (1 until this.size).reversed()) {  
        // 3  
        if(this[i] == this[i - 1]) {  
            return this[i]  
        }  
    }  
    return null  
}
```

Here's the solution in three steps:

1. You first sort the list.
2. Start going through it from right to left since you know that the biggest elements are on the right, neatly sorted.
3. The first one that's repeated is your solution.

In the end, if you've gone through all of the elements and none of them are repeated, you can return `null` and call it a day.

The time complexity of this solution is $O(n^2)$ because you've used sorting.

Challenge 3: Manual reverse

Reverse a list of elements by hand. Do not rely on `reverse` or `reversed`; you need to create your own.

Solution 3

Reversing a collection is also quite straightforward. Using the double reference approach, you start swapping elements from the start and end of the collection, making your way to the middle.

Once you've hit the middle, you're done swapping, and the collection is reversed.

```
fun<T: Comparable<T>> MutableList<T>.rev() {  
    var left = 0  
    var right = this.size - 1  
  
    while (left < right) {  
        swapAt(left, right)  
        left++  
        right--  
    }  
}
```

For this solution, you need `MutableList` since you need to mutate the collection to reverse.

The time complexity of this solution is $O(n)$.

Key points

- n^2 algorithms often have a terrible reputation, but some of these algorithms usually have some redeeming points. `insertionSort` can sort in $O(n)$ time if the collection is already in sorted order and gradually scales down to $O(n^2)$.
- `insertionSort` is one of the best sorts in situations wherein you know ahead of time that your data is in sorted order.
- Design your algorithms to be as generic as possible without hurting the performance.

15

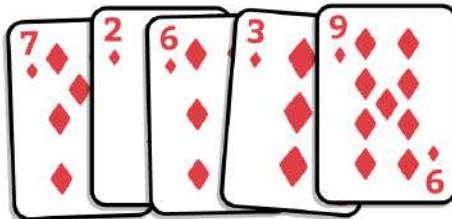
Chapter 15: Merge Sort

By Matei řuică

Merge sort is one of the most efficient sorting algorithms. With a time complexity of $O(n \log n)$, it's one of the fastest of all general-purpose sorting algorithms. The idea behind merge sort is **divide and conquer** — to break a big problem into several smaller, easier-to-solve problems, and then combine those solutions into a final result. The merge sort mantra is to *split first and merge after*.

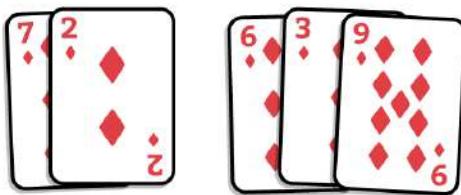


As an example, assume that you're given a pile of unsorted playing cards:

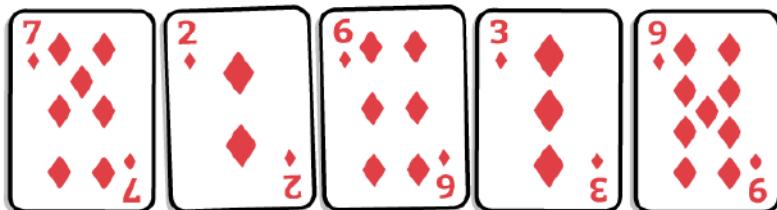
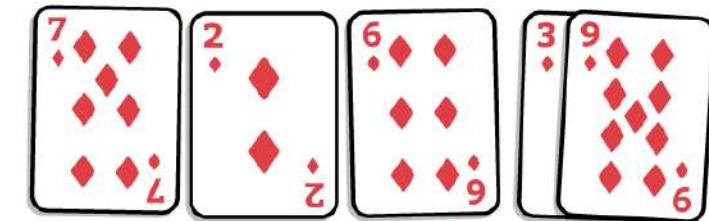


The merge sort algorithm works as follows:

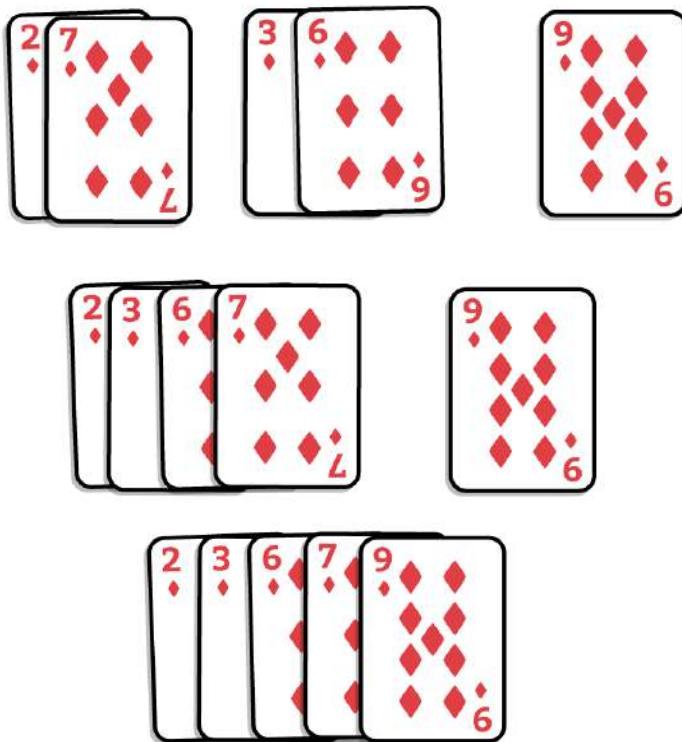
1. Split the pile in half, which gives you two unsorted piles:



2. Keep splitting the resulting piles until you can't split them anymore. In the end, you'll have one card in each pile. Because a single card is always sorted, you now have a bunch of sorted piles:



- Merge the piles in the reverse order in which you split them. During each merge, put the contents in sorted order. This is easy because each pile has already been sorted. You know that the smallest cards in any pile are on the left side:



In this chapter, you'll implement merge sort from scratch.

Implementation

The Merge sort consists of two main steps: split and merge. To implement them, open the starter project and start editing the `MergeSort.kt` file into the `mergesort` package.

Split

In the **MergeSort.kt** file copy the following code:

```
fun <T : Comparable<T>> List<T>.mergeSort(): List<T> {
    if (this.size < 2) return this
    val middle = this.size / 2
    val left = this.subList(0, middle)
    val right = this.subList(middle, this.size)
    // ... more to come
}
```

When it comes to sorting algorithms, any list that's shorter than two elements is already sorted. That's why the first `if` is your best chance to exit fast and finish with the sorting.

You then split the list into halves. Splitting once isn't enough — you have to keep splitting recursively until you can't split anymore, which is when each subdivision contains only a single element.

To do this, update `mergeSort` as follows:

```
fun <T : Comparable<T>> List<T>.mergeSort(): List<T> {
    // 1
    if (this.size < 2) return this
    val middle = this.size / 2

    // 2
    val left = this.subList(0, middle).mergeSort()
    val right = this.subList(middle, this.size).mergeSort()

    // ... still more to come
}
```

Here are the key elements of the code as it looks right now:

1. Recursion needs a **base case**, which you can also think of as an “exit condition”. In this case, the base case is when the list only has one element. Your previous *quick win* is now the cornerstone of the algorithm.
2. You’re calling `mergeSort` on each of the sub-lists. This recursion continues to try to split the lists into smaller lists until the “exit condition” is fulfilled. In your case, it will split until the lists contain only one element.

There’s still more work to do before your code will compile. Now that you’ve accomplished the splitting part, it’s time to focus on merging.

Merge

Your final step is to merge the `left` and `right` lists. To keep things clean, you'll create a separate `merge` function to handle this.

The sole responsibility of the `merge` function is to take in two `sorted` lists and combine them while retaining the sort order. Add the following immediately below `mergeSort`:

```
private fun <T : Comparable<T>> merge(left: List<T>, right: List<T>): List<T> {
    // 1
    var leftIndex = 0
    var rightIndex = 0
    // 2
    val result = mutableListOf<T>()
    // 3
    while (leftIndex < left.size && rightIndex < right.size) {
        val leftElement = left[leftIndex]
        val rightElement = right[rightIndex]
        // 4
        if (leftElement < rightElement) {
            result.add(leftElement)
            leftIndex += 1
        } else if (leftElement > rightElement) {
            result.add(rightElement)
            rightIndex += 1
        } else {
            result.add(leftElement)
            leftIndex += 1
            result.add(rightElement)
            rightIndex += 1
        }
    }
    // 5
    if (leftIndex < left.size) {
        result.addAll(left.subList(leftIndex, left.size))
    }
    if (rightIndex < right.size) {
        result.addAll(right.subList(rightIndex, right.size))
    }
    return result
}
```

Here's what's going on:

1. The `leftIndex` and `rightIndex` variables track your progress as you parse through the two lists.
2. The `result` list will house the combined lists.

3. Starting from the beginning, you compare the elements in the `left` and `right` lists sequentially. When you reach the end of either list, there's nothing else to compare.
4. The smaller of the two elements goes into the `result` list. If the elements are equal, they can both be added.
5. The first loop guarantees that either `left` or `right` is empty. Since both lists are sorted, this ensures that the leftover elements are greater than or equal to the ones currently in `result`. In this scenario, you can add the rest of the elements without comparison.

Finishing up

Complete `mergeSort` by calling `merge`. Because you call `mergeSort` recursively, the algorithm will split and sort both halves before merging them.

```
fun <T : Comparable<T>> List<T>.mergeSort(): List<T> {
    if (this.size < 2) return this
    val middle = this.size / 2
    val left = this.subList(0, middle).mergeSort()
    val right = this.subList(middle, this.size).mergeSort()

    return merge(left, right)
}
```

This is the final version of the merge sort algorithm. Here's a summary of the key procedures of merge sort:

1. The strategy of merge sort is to *divide and conquer* so that you solve many small problems instead of one big problem.
2. It has two core responsibilities: a method to divide the initial list recursively, as well as a method to merge two lists.
3. The merging function should take two sorted lists and produce a single sorted list.

Finally, it's time to see this in action. Head back to **Main.kt** and test your merge sort.

Add the following code in `main()`:

```
"merge sort" example {  
    val list = listOf(7, 2, 6, 3, 9)  
    println("Original: $list")  
    val result = list.mergeSort()  
    println("Merge sorted: $result")  
}
```

This outputs:

```
---Example of merge sort---  
Original: [7, 2, 6, 3, 9]  
Merge sorted: [2, 3, 6, 7, 9]
```

Performance

The best, worst and average time complexity of merge sort is $O(n \log n)$, which isn't too bad. If you're struggling to understand where $n \log n$ comes from, think about how the recursion works:

- As you recurse, you split a single list into two smaller lists. This means a list of size 2 will need one level of recursion, a list of size 4 will need two levels, a list of size 8 will need three levels, and so on. If you had a list of 1,024 elements, it would take 10 levels of recursively splitting in two to get down to 1024 single element lists. In general, if you have a list of size n , the number of levels is $\log_2(n)$.
- A single recursion level will merge n elements. It doesn't matter if there are many small merges or one large one; the number of elements merged will still be n at each level. This means the cost of a single recursion is $O(n)$.

This brings the total cost to $O(\log n) \times O(n) = O(n \log n)$.

The previous chapter's sort algorithms were **in-place** and used `swapAt` to move elements around. Merge sort, by contrast, allocates additional memory to do its work. How much? There are $\log_2(n)$ levels of recursion, and at each level, n elements are used. That makes the total $O(n \log n)$ in space complexity. Merge sort is one of the hallmark sorting algorithms. It's relatively simple to understand, and serves as a great introduction to how divide and conquer algorithms work. Merge sort is $O(n \log n)$, and this implementation requires $O(n \log n)$ of space. If you're clever with your bookkeeping, you can reduce the memory required to $O(n)$ by discarding the memory that is not actively being used.

Challenges

Challenge 1: Iterables merge

Write a function that takes two sorted iterables and merges them into a single iterable. Here's the function signature to start:

```
fun <T : Comparable<T>> merge(  
    first: Iterable<T>,  
    second: Iterable<T>  
) : Iterable<T>
```

Solution 1

The tricky part of this challenge is the limited capabilities of `Iterable`. Traditional implementations of this algorithm rely on the abilities of `List` types to keep track of indices.

Since `Iterable` types have no notion of indices, you'll make use of their iterator. The `Iterator` in Kotlin has a slight inconvenience that you need to fix first. If there are no more elements in the iterable and you try to get the next one using `next()`, you'll get a `NoSuchElementException`. To make it friendlier for your algorithm, write the following extension function first:

```
private fun <T> Iterator<T>.nextOrNull(): T? {  
    return if (this.hasNext()) this.next() else null  
}
```

You can now use `nextOrNull()` to safely get the next element. If the returned value is `null`, this means there's no next element, and the iterable is over. This will be important later on.

Now, for `merge()`. Add the following code to your file:

```
fun <T : Comparable<T>> merge(  
    first: Iterable<T>,  
    second: Iterable<T>  
) : Iterable<T> {  
  
    // 1  
    val result = mutableListOf<T>()  
    val firstIterator = first.iterator()  
    val secondIterator = second.iterator()  
  
    // 2
```

```

if (!firstIterator.hasNext()) return second
if (!secondIterator.hasNext()) return first

// 3
var firstEl = firstIterator.nextOrNull()
var secondEl = secondIterator.nextOrNull()

// 4
while (firstEl != null && secondEl != null) {
    // more to come
}
}

```

Setting up the algorithm involves the following steps:

1. Create a new container to store the merged iterable. It could be any class that implements `Iterable` but a `MutableList` is an easy to use choice, so go with that one. Then, grab the iterators of the first and second iterable. Iterators sequentially dispense values of the iterable via `next()`, but you'll use your own extension `nextOrNull()`.
2. Create two variables that are initialized as the first and second iterator's first value.
3. If one of the iterators didn't have a first value, it means the iterable it came from was empty, and you're done sorting. Simply return the other iterable.
4. This first `while` loop is where all of the comparisons are made to get the resulting iterable ordered. It only works while you still have values in both iterables.

Using the iterators, you'll decide which element should be appended into the `result` list by comparing the first and second next values. Write the following inside the `while` loop:

```

when {
    // 1
    firstEl < secondEl -> {
        result.add(firstEl)
        firstEl = firstIterator.nextOrNull()
    }
    // 2
    secondEl < firstEl -> {
        result.add(secondEl)
        secondEl = secondIterator.nextOrNull()
    }
    // 3
    else -> {

```

```

        result.add(firstEl)
        result.add(secondEl)
        firstEl = firstIterator.nextOrNull()
        secondEl = secondIterator.nextOrNull()
    }
}

```

This is the main component of the merging algorithm. There are three situations possible, as you can see in the when statement:

1. If the first value is less than the second, you'll append the first value in `result` and seed the next value to be compared with by invoking `nextOrNull()` on the first iterator.
2. If the second value is less than the first, you'll do the opposite. You seed the next value to be compared by invoking `nextOrNull()` on the second iterator.
3. If they're equal, you append both the `first` and `second` values and seed both next values.

This will continue until one of the iterators runs out of elements to dispense. In that scenario, the iterator with elements left only has elements that are equal to or greater than the current values in `result`.

To add the rest of those values, write the following at the end of `merge()`:

```

while (firstEl != null) {
    result.add(firstEl)
    firstEl = firstIterator.nextOrNull()
}
while (secondEl != null) {
    result.add(secondEl)
    secondEl = secondIterator.nextOrNull()
}

return result

```

Confirm that this function works by writing the following in `Main.kt`:

```

"merge iterables" example {
    val list1 = listOf(1, 2, 3, 4, 5, 6, 7, 8)
    val list2 = listOf(1, 3, 4, 5, 5, 6, 7, 7)

    val result = merge(list1, list2)
    println("Merge sorted: $result")
}

```

You'll see the following console output:

```
---Example of merge iterables---
Merge sorted: [1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 7, 8]
```

Key points

- Merge sort is in the category of the divide and conquer algorithms.
- There are many implementations of merge sort, and you can have different performance characteristics depending on the implementation.
- To do a comparison, in this chapter you sorted objects implementing the `Comparable<T>` interface but the same can be done providing a different implementation of `Comparator<T>`.

16

Chapter 16: Radix Sort

Matei řuică

So far, you've been relying on comparisons to determine the sorting order. In this chapter, you'll look at a completely different model of sorting known as radix sort.

Radix sort is a non-comparative algorithm for sorting integers in linear time. There are many implementations of radix sort that focus on different problems. To keep things simple, you'll focus on sorting base 10 integers while investigating the *least significant digit* (LSD) variant of radix sort.

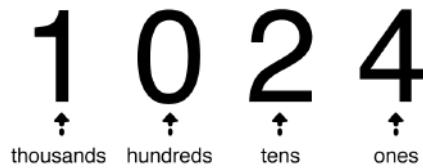


Example

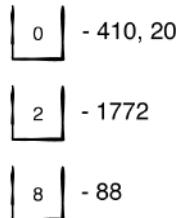
To show how radix sort works, you'll sort the following list:

```
var list = arrayListOf(88, 410, 1772, 20)
```

Radix sort relies on the positional notation of integers, as shown here:



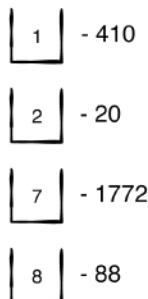
First, the list is divided into buckets based on the value of the least significant digit, the **ones** digit.



These buckets are then emptied in order, resulting in the following partially sorted list:

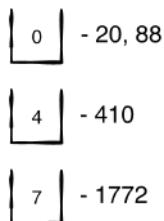
```
list = arrayListOf(410, 20, 1772, 88)
```

Next, repeat this procedure for the **tens** digit:



The relative order of the elements didn't change this time, but you've still got more digits to inspect.

The next digit to consider is the **hundreds** digit:

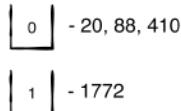


Note: For values that have no hundreds position or any other position, the digit is assumed to be zero.

Reassembling the list based on these buckets gives the following:

```
list = arrayListOf(20, 88, 410, 1772)
```

Finally, you need to consider the **thousands** digit:



Reassembling the list from these buckets leads to the final sorted list:

```
list = arrayListOf(20, 88, 410, 1772)
```

When many numbers end up in the same bucket, their relative ordering doesn't change. For example, in the zero bucket for the hundreds position, 20 comes before 88. This is because the previous step put 20 in a lower bucket than 80, so 20 ended up before 88 in the list.

Implementation

Open the starter project for this chapter. In `src > radixsort`, create a new file named `RadixSort.kt`.

Add the following to the file:

```
fun MutableList<Int>.radixSort() {  
    // ...  
}
```

Here, you've added `radixSort()` to `MutableList` of integers via an extension function. Start implementing `radixSort()` using the following:

```
fun MutableList<Int>.radixSort() {  
    // 1  
    val base = 10  
    // 2  
    var done = false  
    var digits = 1  
    while (!done) {  
        // ...  
    }  
}
```

This bit is straightforward:

1. You're sorting base 10 integers in this instance. Since you'll use this value many times in the algorithm, you store it in a constant `base`.
2. You declare two variables to track your progress. Radix sort works in many passes, so `done` serves as a flag that determines whether the sort is complete. The `digits` variable keeps track of the current digit you're looking at.

Next, you'll write the logic that sorts each element into buckets.

Bucket sort

Write the following **inside** the `while` loop:

```
// 1  
val buckets = arrayListOf<MutableList<Int>>().apply {  
    for(i in 0..9) {  
        this.add(arrayListOf())  
    }  
}  
// 2  
this.forEach {  
    number ->  
    val remainingPart = number / digits  
    val digit = remainingPart % base  
    buckets[digit].add(number)  
}
```

```
// 3
digits *= base

this.clear()
this.addAll(buckets.flatten())
```

Here's how it works:

1. You instantiate the buckets using a two-dimensional list. Because you're using base 10, you need ten buckets.
2. You place each number in the correct bucket.
3. You update digits to the next digit you want to inspect and update the list using the contents of `buckets.flatten()` flattens the two-dimensional list to a one-dimensional list, as if you're emptying the buckets into the list.

When do you stop?

Your `while` loop currently runs forever, so you'll need a terminating condition somewhere. You'll do that as follows:

1. At the beginning of the `while` loop, add `done = true`.
2. Inside the closure of `forEach`, add the following:

```
if (remainingPart > 0) {
    done = false
}
```

Since `forEach` iterates over all of the integers, as long as one of the integers still has unsorted digits, you'll need to continue sorting.

With that, you've learned about your first non-comparative sorting algorithm. Head back to `Main.kt` and add the following to test your code, inside `main()`:

```
"radix sort" example {
    val list = arrayListOf(88, 410, 1772, 20)
    println("Original: $list")
    list.radixSort()
    println("Radix sorted: $list")
}
```

You'll see the following console output:

```
---Example of radix sort---
Original: [88, 410, 1772, 20]
```

```
Radix sorted: [20, 88, 410, 1772]
```

Radix sort is one of the fastest sorting algorithms. The average time complexity of radix sort is $O(k \times n)$, where k is the number of significant digits of the largest number, and n is the number of integers in the list.

Radix sort works best when k is constant, which occurs when all numbers in the list have the same count of significant digits. Its time complexity then becomes $O(n)$. Radix sort also incurs an $O(n)$ space complexity, as you need space to store each bucket.

Challenges

Challenge 1: Most significant sort

The implementation discussed in the chapter used a *least significant digit* radix sort. Your task is to implement a *most significant digit (MSD)* radix sort.

This sorting behavior is called **lexicographical sorting** and is also used for `String` sorting.

For example:

```
var list = arrayListOf(500, 1345, 13, 459, 44, 999)
list.lexicographicalSort()
println(list) // outputs [13, 1345, 44, 459, 500, 999]
```

Solution 1

MSD radix sort is closely related to LSD radix sort, in that both use bucket sort. The difference is that MSD radix sort needs to curate subsequent passes of the bucket sort carefully. In LSD radix sort, bucket sort ran repeatedly using the whole list for every pass. In MSD radix sort, you run bucket sort with the entire list only once. Subsequent passes will sort each bucket recursively.

You'll implement MSD radix sort piece-by-piece, starting with the components on which it depends.



Digits

Add the following inside **Challenge1.kt**:

```
fun Int.digits(): Int {
    var count = 0
    var num = this
    while (num != 0) {
        count += 1
        num /= 10
    }
    return count
}

fun Int.digit(atPosition: Int): Int? {
    if(atPosition > digits()) return null
    var num = this
    val correctedPosition = (atPosition + 1).toDouble()
    while (num / (pow(10.0, correctedPosition.toInt())) != 0) {
        num /= 10
    }
    return num % 10
}
```

`digits` is a computed property that returns the number of digits that the `Int` has. For example, the value of 1024 has four digits.

`digit(atPosition:)` returns the digit at a given position. Like lists, the leftmost position is zero. Thus, the digit for position zero of the value 1024 is **1**. The digit for position three is **4**. Since there are only four digits, the digit for position five will return `null`.

The implementation of `digit()` works by repeatedly chopping a digit off the end of the number, until the requested digit is at the end. It's then extracted using the remainder operator.

Lexicographical sort

With the helper methods, you're now equipped to deal with MSD radix sort. Write the following at the bottom of the file:

```
fun MutableList<Int>.lexicographicalSort() {
    this.clear()
    this.addAll(msdRadixSorted(this, 0))
}

private fun msdRadixSorted(list: MutableList<Int>, position: Int): MutableList<Int> {
```

`lexicographicalSort()` is the user-facing API for MSD radix sort.
`msdRadixSorted()` is the meat of the algorithm and will be used to recursively apply MSD radix sort to the list.

Update `msdRadixSorted()` to the following:

```
private fun msdRadixSorted(list: MutableList<Int>, position: Int): MutableList<Int> {
    // 1
    val buckets = arrayListOf<MutableList<Int>>().apply {
        for(i in 0..9) {
            this.add(arrayListOf())
        }
    }
    // 2
    val priorityBucket = arrayListOf<Int>()
    // 3
    list.forEach { number ->
        val digit = number.digit(position)
        if(digit == null) {
            priorityBucket.add(number)
            return@forEach
        }
        buckets[digit].add(number)
    }
}
```

Here's how it works:

1. Similar to LSD radix sort, you instantiate a two-dimensional list for the buckets.
2. The `priorityBucket` is a special bucket that stores values with fewer digits than the current position. Values that go in the `priorityBucket` are sorted first.
3. For every number in the list, you find the digit of the current position and place the number in the appropriate bucket.

Next, you need to recursively apply MSD radix sort for each of the individual buckets. Write the following at the end of `msdRadixSorted()`:

```
priorityBucket.addAll(
    buckets.reduce { result, bucket ->
        if (bucket.isEmpty()) return@reduce result
        result.addAll(msdRadixSorted(bucket, position + 1))
        result
    }
)

return priorityBucket
```

This statement calls `reduce(into:)` to collect the results of the recursive sorts and appends them to the `priorityBucket`. That way, the elements in the `priorityBucket` always go first. You're almost done!

Base case

As with all recursive operations, you need to set a terminating condition that stops the recursion. Recursion should halt if the current position you're inspecting is greater than the number of significant digits of the largest value inside the list.

At the bottom of the file, write the following:

```
private fun MutableList<Int>.maxDigits(): Int {  
    return this.max()?.digits() ?: 0  
}
```

Next, add the following at the top of `msdRadixSorted`:

```
if(position > list.maxDigits()) return list
```

This ensures that if the position is equal or greater than the list's `maxDigits`, you'll terminate the recursion.

Take it out for a spin! Add the following to `Main.kt` so you can test the code:

```
"MSD radix sort" example {  
    val list= (0..10).map { (Math.random() *  
    10000).toInt() }.toMutableList()  
    println("Original: $list")  
    list.lexicographicalSort()  
    println("Radix sorted: $list")  
}
```

You should see a list of random numbers like this:

```
---Example of MSD radix sort---  
Original: [448, 3168, 6217, 7117, 1256, 3887, 3900, 3444, 4976,  
6891, 4682]  
Radix sorted: [1256, 3168, 3444, 3887, 3900, 448, 4682, 4976,  
6217, 6891, 7117]
```

Since the numbers are random, you won't get an identical list. The important thing to note is the lexicographical ordering of the values.

Key points

- Radix sort is a non-comparative sort that doesn't rely on comparing two values. Instead, it leverages bucket sort, which is like a sieve for filtering values. A helpful analogy is how some of the vending machines accept coins — the coins are distinguished by size.
- This chapter covered the least significant digit radix sort. Another way to implement radix sort is the most significant digit form. This form sorts by prioritizing the most significant digits over the lesser ones and is best illustrated by the sorting behavior of the `String` type.

Chapter 17: Heap Sort

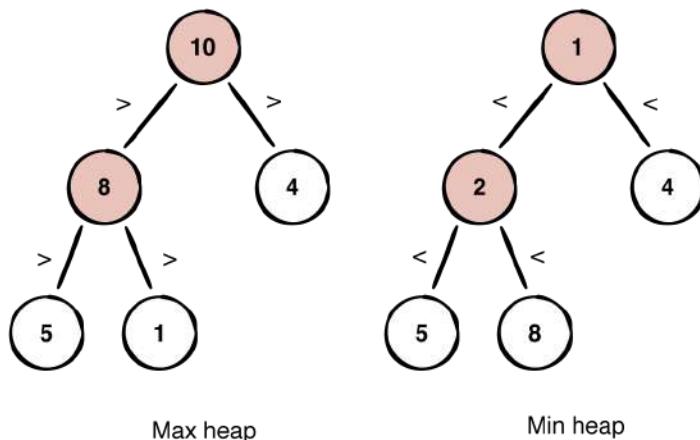
Matei řuică

Heapsort is another comparison-based algorithm that sorts an array in ascending order using a heap. This chapter builds on the heap concepts presented in Chapter 12, "Heap Data Structure."

Heapsort takes advantage of a heap being, by definition, a partially sorted binary tree with the following qualities:

1. In a max heap, all parent nodes are larger than their children.
2. In a min heap, all parent nodes are smaller than their children.

The diagram below shows a heap with parent node values underlined:



Getting started

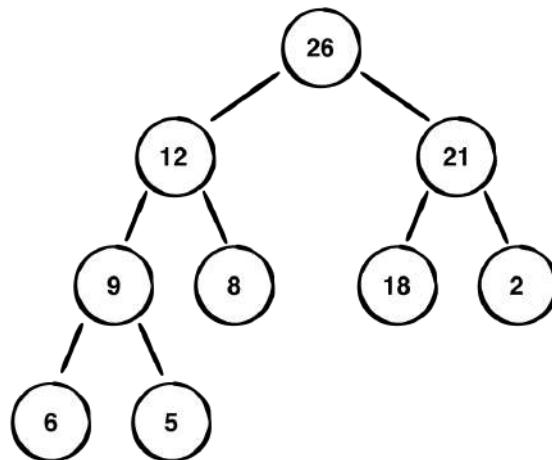
Open up the starter project. This project already contains an implementation of a heap. Your goal is to extend Heap so it can also sort. Before you get started, let's look at a visual example of how heap sort works.

Example

For any given unsorted array, to sort from lowest to highest, heap sort must first convert this array into a heap:

6	12	2	26	8	18	21	9	5
---	----	---	----	---	----	----	---	---

This conversion is done by sifting down all the parent nodes so that they end up in the right spot. The resulting heap is:



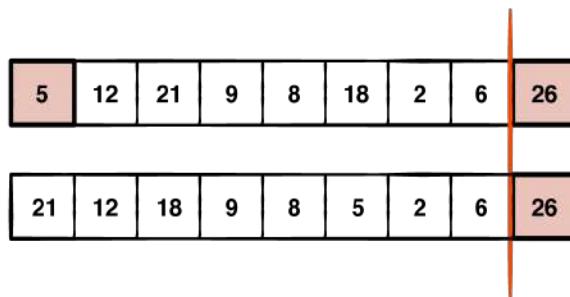
This corresponds with the following array:

26	12	21	9	8	18	2	6	5
----	----	----	---	---	----	---	---	---

Because the time complexity of a single sift-down operation is $O(\log n)$, the total time complexity of building a heap is $O(n \log n)$.

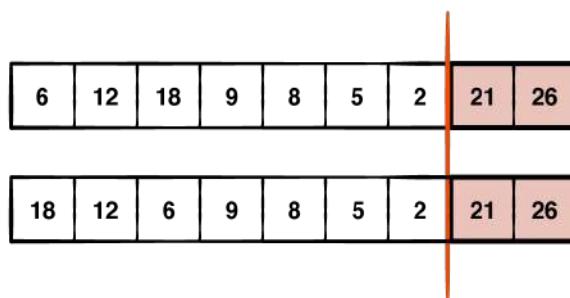
Let's look at how to sort this array in ascending order.

Because the largest element in a max heap is always at the root, you start by swapping the first element at index **0** with the last element at index **$n - 1$** . As a result of this swap, the last element of the array is in the correct spot, but the heap is now invalidated. The next step is, thus, to sift down the new root node **5** until it lands in its correct position.



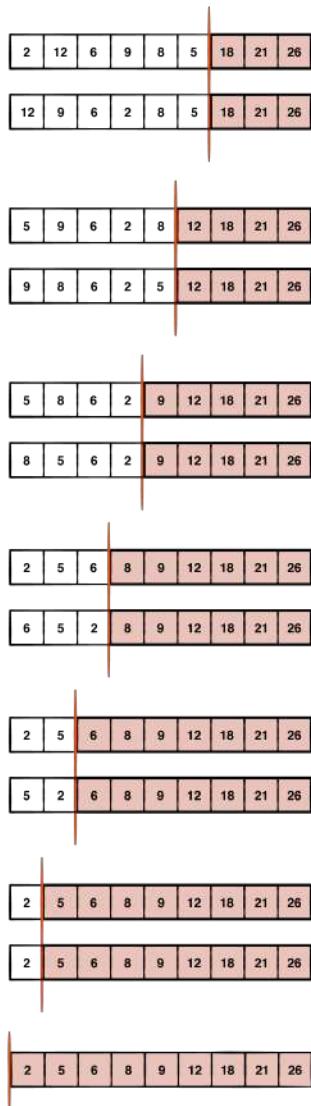
Note that you exclude the last element of the heap as you no longer consider it part of the heap, but of the sorted array.

As a result of sifting down **5**, the second largest element **21** becomes the new root. You can now repeat the previous steps, swapping **21** with the last element **6**, shrinking the heap and sifting down **6**.



Starting to see a pattern? Heap sort is very straightforward. As you swap the first and last elements, the larger elements make their way to the back of the array in the correct order. You simply repeat the swapping and sifting steps until you reach a heap of size 1.

The array is then fully sorted.



Note: This sorting process is very similar to selection sort from **Chapter 14**.

Implementation

Next, you'll implement this sorting algorithm. The actual implementation is simple. You'll be performing the sort on an `Array` and reusing the algorithms already implemented in `Heap` in the form of extension functions. You'll reorder the elements using `heapify()`. After that, the heavy lifting will be done by a `siftDown()` method.

First, create a new file in the `heapsort` package, called `HeapSort.kt`. You might remember that the `siftDown()` method in `Heap` needs to determine the index of the left and right children of an element at a given index. You will start by copying the functions that do just that, as top level functions in this new file:

```
private fun leftChildIndex(index: Int) = (2 * index) + 1  
private fun rightChildIndex(index: Int) = (2 * index) + 2
```

After that, you will copy the `siftDown()` method from `Heap`, and transform it into an extension function for `Array<T>`. Since this extension function can work with all kinds of `T` elements, you'll also need to add a `Comparator<T>` to the parameters of the function. The transformed `siftDown()` function will look like this:

```
fun <T> Array<T>.siftDown(  
    index: Int,  
    upTo: Int,  
    comparator: Comparator<T>  
) {  
    var parent = index  
    while (true) {  
        val left = leftChildIndex(parent)  
        val right = rightChildIndex(parent)  
        var candidate = parent  
        if (left < upTo &&  
            comparator.compare(this[left], this[candidate]) > 0  
        ) {  
            candidate = left  
        }  
        if (right < upTo &&  
            comparator.compare(this[right], this[candidate]) > 0  
        ) {  
            candidate = right  
        }  
        if (candidate == parent) {  
            return  
        }  
        this.swapAt(parent, candidate)  
        parent = candidate  
    }  
}
```

The differences between this function and the method that the **Heap** has is that you operate on `this` array instead of `elements`, and that your `compare()` calls are addressed to the `comparator` you get as a parameter. The algorithm itself remains the same, so if you can't wrap your head around this, you can take a look at the **Heap Data Structure** chapter again, which explains this function in detail.

Next, you'll need a `heapify()` function. As with `siftDown()`, it will be an extension function very similar to the one in `Heap`. This one will also have a `Comparator<T>` parameter, as it will have to call `siftDown()`. Copy this into `HeapSort.kt`:

```
fun <T> Array<T>.heapify(comparator: Comparator<T>) {
    if (this.isEmpty()) {
        (this.size / 2 downTo 0).forEach {
            this.siftDown(it, this.size, comparator)
        }
    }
}
```

The final step is to implement the actual sorting. This is simple enough, here's how:

```
fun <T> Array<T>.heapSort(comparator: Comparator<T>) {
    this.heapify(comparator)
    for (index in this.indices.reversed()) { // 1
        this.swapAt(0, index) // 2
        siftDown(0, index, comparator) // 3
    }
}
```

Here's what's going on:

1. You reorder the elements so that the array looks like a Heap.
2. You loop through the array, starting from the last element.
3. You swap the first element and the last element. This moves the largest unsorted element to its correct spot.
4. Because the heap is now invalid, you must sift down the new root node. As a result, the next largest element will become the new root.

Note that, in order to support heap sort, you've added an additional parameter `upTo` to the `siftDown()` method. This way, the sift down only uses the unsorted part of the array, which shrinks with every iteration of the loop.

Finally, give your new method a try. Add this to the `main()` function in **Main.kt**:

```
"Heap sort" example {  
    val array = arrayOf(6, 12, 2, 26, 8, 18, 21, 9, 5)  
    array.heapSort(ascending)  
    print(array.joinToString())  
}
```

This should print:

```
---Example of Heap sort---  
2, 5, 6, 8, 9, 12, 18, 21, 26
```

Performance

Even though you get the benefit of in-memory sorting, the performance of heap sort is $O(n \log n)$ for its best, worse and average cases. This is because you have to traverse the whole array once and, every time you swap elements, you must perform a sift down, which is an $O(\log n)$ operation.

Heap sort is also not a stable sort because it depends on how the elements are laid out and put into the heap. If you were heap sorting a deck of cards by their rank, for example, you might see their suite change order with respect to the original deck.

Challenges

Challenge 1: Fewest comparisons

When performing a heap sort in ascending order, which of these starting arrays requires the fewest comparisons?

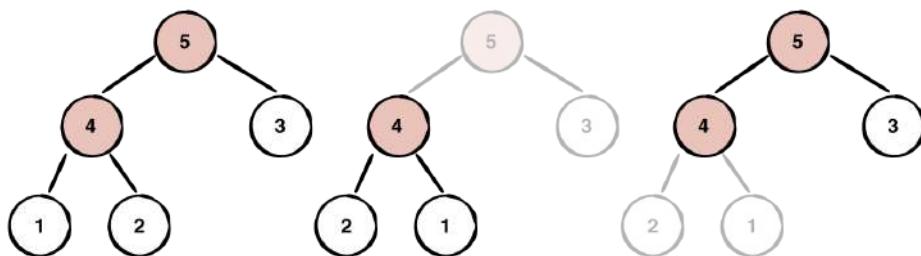
- [1,2,3,4,5]
- [5,4,3,2,1]

Solution 1

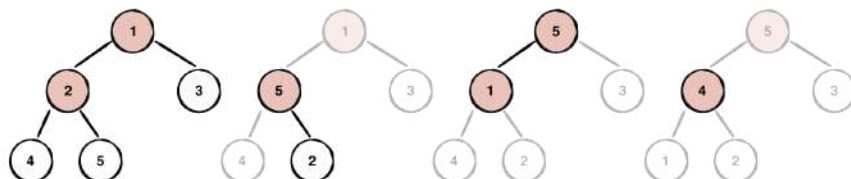
When sorting elements in ascending order using heap sort, you first need a max heap. What you really need to look at is the number of comparisons that happen when constructing the max heap.

[5,4,3,2,1] will yield the fewest number of comparisons, since it's already a max heap and no swaps take place.

When building a max heap, you only look at the parent nodes. In this case there are two parent nodes, with two comparisons.



[1,2,3,4,5] will yield the most number of comparisons. There are two parent nodes, but you have to perform three comparisons:



Challenge 2: Descending sort

The current example of heap sort sorts the elements in **ascending** order. How would you sort in **descending** order?

Solution 2

This is a simple one as well. You just need to swap the ascending comparator with a descending one when you create the `SortingHeap`. Looking at how `ascending` is defined, you can easily come up with a descending:

```
val descending = Comparator { first: Int, second: Int ->
    when {
        first < second -> 1
        first > second -> -1
        else -> 0
    }
}
```

If you haven't already noticed, you just need to change the signs for -1 and 1. That's it! Now you can create another example in `Main.kt` to see how it works:

```
"Heap sort descending" example {
    val array = arrayListOf(6, 12, 2, 26, 8, 18, 21, 9, 5)
    array.heapSort(descending)
    print(array)
}
```

The result is a descending sorted list:

```
---Example of Heap sort descending---
26, 21, 18, 12, 9, 8, 6, 5, 2
```

Key points

- Heap sort leverages the max-heap data structure to sort elements in an array.
- Heap sort sorts its elements by following a simple pattern: swap the first and last element, perform a `sift-down`, decrease the array size by one; then repeat.
- The performance of heap sort is $O(n \log n)$ for its best, worse and average cases.

18

Chapter 18: Quicksort

By Matei řuică

In the preceding chapters, you learned to sort a list using comparison-based sorting algorithms, such as merge sort and heap sort.

Quicksort is another comparison-based sorting algorithm. Much like merge sort, it uses the same strategy of **divide and conquer**. One important feature of quicksort is choosing a **pivot** point. The pivot divides the list into three partitions:

```
[ elements < pivot | pivot | elements > pivot ]
```

In this chapter, you'll implement quicksort and look at various partitioning strategies to get the most out of this sorting algorithm.



Example

Open the starter project. Inside **QuicksortNaive.kt**, you'll see a naive implementation of quicksort:

```
fun<T: Comparable<T>> List<T>.quicksortNaive(): List<T> {
    if (this.size < 2) return this // 1

    val pivot = this[this.size / 2] // 2
    val less = this.filter { it < pivot } // 3
    val equal = this.filter { it == pivot }
    val greater = this.filter { it > pivot }
    return less.quicksortNaive() + equal +
        greater.quicksortNaive() // 4
}
```

This implementation recursively filters the list into three partitions. Here's how it works:

1. There must be more than one element in the list. If not, the list is considered sorted.
2. Pick the **middle** element of the list as your pivot.
3. Using the pivot, split the original list into three partitions. Elements **less than**, **equal to** or **greater than** the pivot go into different buckets.
4. Recursively sort the partitions and then combine them.

Now, it's time to visualize the code above. Given the **unsorted** list below:

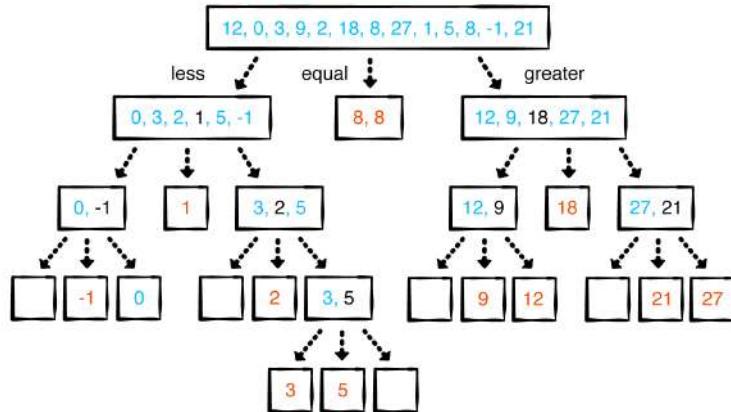
```
[12, 0, 3, 9, 2, 18, 8, 27, 1, 5, 8, -1, 21]
*
```

Your partition strategy in this implementation is to always select the **middle** element as the pivot. In this case, the element is **8**. Partitioning the list using this pivot results in the following partitions:

```
less: [0, 3, 2, 1, 5, -1]
equal: [8, 8]
greater: [12, 9, 18, 27, 21]
```

Notice that the three partitions aren't completely sorted yet. Quicksort will recursively divide these partitions into even smaller ones. The recursion will only halt when all partitions have either zero or one element.

Here's an overview of all the partitioning steps:



Each level corresponds with a recursive call to quicksort. Once recursion stops, the leafs are combined again, resulting in a fully sorted list:

```
[-1, 1, 2, 3, 5, 8, 8, 9, 12, 18, 21, 27]
```

While this naive implementation is easy to understand, it raises some issues and questions:

- Calling `filter` three times on the same list is not efficient?
- Creating a new list for every partition isn't space-efficient. Could you possibly sort in place?
- Is picking the middle element the best pivot strategy? What pivot strategy should you adopt?

Partitioning strategies

In this section, you'll look at partitioning strategies and ways to make this quicksort implementation more efficient. The first partitioning algorithm you'll look at is **Lomuto's algorithm**.

Lomuto's partitioning

Lomuto's partitioning algorithm always chooses the **last** element as the pivot. Time to look at how this works in code.

In your project, create a file named **QuicksortLomuto.kt** and add the following function declaration:

```
fun<T: Comparable<T>> MutableList<T>.partitionLomuto(  
    low: Int,  
    high: Int): Int {  
}
```

This function takes three arguments:

- **list** is the list you are partitioning.
- **low** and **high** set the range within the list you'll partition. This range will get smaller and smaller with every recursion.

The function returns the index of the pivot.

Now, implement the function as follows:

```
val pivot = this[high] // 1  
  
var i = low // 2  
for (j in low until high) { // 3  
    if (this[j] <= pivot) { // 4  
        this.swapAt(i, j) // 5  
        i += 1  
    }  
}  
this.swapAt(i, high) // 6  
return i // 7
```

Here's what this code does:

1. Set the pivot. Lomuto always chooses the last element as the pivot.
2. The variable **i** indicates how many elements are **less** than the pivot. Whenever you encounter an element that is less than the pivot, you swap it with the element at index **i** and increase **i**.
3. Loop through all the elements from **low** to **high**, but not including **high** since it's the pivot.
4. Check to see if the current element is less than or equal to the pivot.

5. If it is, swap it with the element at index i and increase i .
 6. Once done with the loop, swap the element at i with the pivot. The pivot always sits between the **less** and **greater** partitions.
 7. Return the index of the pivot.

While this algorithm loops through the list, it divides the list into four regions:

1. `this.subList(low, i)` contains all elements \leq pivot.
 2. `this.subList(i, j)` contains all elements $>$ pivot.
 3. `this.subList(j, high)` are elements you have not compared yet.
 4. `this[high]` is the pivot element.

$\text{values} \leq \text{pivot}$	$\text{values} > \text{pivot}$	not compared yet	pivot			
low	$i-1$	i	$j-1$	j	$\text{high}-1$	high

Step-by-step

Looking at a few steps of the algorithm will help you get a better understanding of how it works.

Given the **unsorted** list below:

```
[12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8]
```

First, the last element **8** is selected as the pivot:

```

[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, | 12
[ 12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, | 8 ]
    low                                high
      i
      j

```

Then, the first element **12** is compared to the pivot. It's not smaller than the pivot, so the algorithm continues to the next element:

```

[ 0  1  2  3  4  5  6  7  8  9  10 11 | 12
[ 12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, | 8 ]
    low           high
      i
      j

```

The second element **0** is smaller than the pivot, so it's swapped with the element currently at index **i** (**12**) and **i** is increased:

0	1	2	3	4	5	6	7	8	9	10	11	12	
[0,	12,	3,	9,	2,	21,	18,	27,	1,	5,	8,	-1,	
low													high
i													j

The third element **3** is again smaller than the pivot, so another swap occurs:

0	1	2	3	4	5	6	7	8	9	10	11	12	
[0,	3,	12,	9,	2,	21,	18,	27,	1,	5,	8,	-1,	
low													high
i													j

These steps continue until all but the pivot element have been compared. The resulting list is:

0	1	2	3	4	5	6	7	8	9	10	11	12	
[0,	3,	2,	1,	5,	8,	-1,	27,	9,	12,	21,	18,	
low													high
i													

Finally, the pivot element is swapped with the element currently at index **i**:

0	1	2	3	4	5	6	7	8	9	10	11	12	
[0,	3,	2,	1,	5,	8,	-1	8	9,	12,	21,	18,	
low													high
i													

Lomuto's partitioning is now complete. Notice how the pivot is between the two regions of elements less than or equal to the pivot and elements greater than the pivot.

In the naive implementation of quicksort, you created three new lists and filtered the unsorted lists three times. Lomuto's algorithm performs the partitioning in place. That's much more efficient!

With your partitioning algorithm in place, you can now implement quicksort adding the following to your **QuicksortLomuto.kt** file:

```
fun<T: Comparable<T>> MutableList<T>.quicksortLomuto(low: Int,
    high: Int) {
    if (low < high) {
        val pivot = this.partitionLomuto( low, high)
```

```
        this.quicksortLomuto( low, pivot - 1)
        this.quicksortLomuto( pivot + 1, high)
    }
```

Here, you apply Lomuto's algorithm to partition the list into two regions. You then recursively sort these regions. Recursion ends once a region has less than two elements.

You can try Lomuto's quicksort by adding the following to your **Main.kt** file, inside the `main()` function:

```
"Lomuto quicksort" example {
    val list = arrayListOf(12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8,
    -1, 8)
    println("Original: $list")
    list.quicksortLomuto(0, list.size - 1)
    println("Sorted: $list")
}
```

Hoare's partitioning

Hoare's partitioning algorithm always chooses the **first** element as the pivot. So, how does this work in code?

In your project, create a file named **QuicksortHoare.kt** and add the following function:

```
fun<T: Comparable<T>> MutableList<T>.partitionHoare(low: Int,
high: Int): Int {
    val pivot = this[low] // 1
    var i = low - 1 // 2
    var j = high + 1
    while (true) {
        do { // 3
            j -= 1
        } while (this[j] > pivot)
        do { // 4
            i += 1
        } while (this[i] < pivot)
        if (i < j) { // 5
            this.swapAt(i, j)
        } else {
            return j // 6
        }
    }
}
```

Here's how it works:

1. Select the first element as the pivot.
2. Indexes i and j define two regions. Every index before i will be **less than or equal to** the pivot. Every index after j will be **greater than or equal to** the pivot.
3. Decrease j until it reaches an element that is not greater than the pivot.
4. Increase i until it reaches an element that is not less than the pivot.
5. If i and j have not overlapped, swap the elements.
6. Return the index that separates both regions.

Note: The index returned from the partition does not necessarily have to be the index of the pivot element.

Step-by-step

Given the **unsorted** list below:

```
[ 12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8 ]
```

First, **12** is set as the pivot. Then i and j will start running through the list, looking for elements that are not less than (in the case of i) or greater than (in the case of j) the pivot. i will stop at element **12** and j will stop at element **8**:

```
[ 12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8 ]
    p
    i                               j
```

These elements are then swapped:

```
[ 8, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 12 ]
    i                               j
```

i and j now continue moving, this time stopping at **21** and **-1**:

```
[ 8, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 12 ]
    i                               j
```

Which are then swapped:

```
[ 8, 0, 3, 9, 2, -1, 18, 27, 1, 5, 8, 21, 12 ]
      i           j
```

Next, **18** and **8** are swapped, followed by **27** and **5**.

After this swap the list and indices are as follows:

```
[ 8, 0, 3, 9, 2, -1, 8, 5, 1, 27, 18, 21, 12 ]
      i           j
```

The next time you move *i* and *j*, they will overlap:

```
[ 8, 0, 3, 9, 2, -1, 8, 5, 1, 27, 18, 21, 12 ]
      j           i
```

Hoare's algorithm is now complete, and index *j* is returned as the separation between the two regions.

There are far fewer swaps here compared to Lomuto's algorithm. Isn't that nice?

You can now implement a `quicksortHoare` function:

```
fun<T: Comparable<T>> MutableList<T>.quicksortHoare(low: Int,
high: Int) {
    if (low < high) {
        val p = this.partitionHoare(low, high)
        this.quicksortHoare(low, p)
        this.quicksortHoare(p + 1, high)
    }
}
```

Try it out by adding the following in your `Main.kt`:

```
"Hoare quicksort" example {
    val list = arrayListOf(12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8,
    -1, 8)
    println("Original: $list")
    list.quicksortHoare(0, list.size - 1)
    println("Sorted: $list")
}
```

Effects of a bad pivot choice

The most important part of implementing quicksort is choosing the right partitioning strategy.

You've looked at three different partitioning strategies:

1. Choosing the middle element as a pivot.
2. **Lomuto**, or choosing the last element as a pivot.
3. **Hoare**, or choosing the first element as a pivot.

What are the implications of choosing a bad pivot?

Starting with the following unsorted list:

```
[8, 7, 6, 5, 4, 3, 2, 1]
```

If you use Lomuto's algorithm, the pivot will be the last element, 1. This results in the following partitions:

```
less: []
equal: [1]
greater: [8, 7, 6, 5, 4, 3, 2]
```

An ideal pivot would split the elements evenly between the **less than** and **greater than** partitions. Choosing the first or last element of an already sorted list as a pivot makes quicksort perform much like **insertion sort**, which results in a worst-case performance of $O(n^2)$. One way to address this problem is by using the **median of three** pivot selection strategy. Here, you find the median of the first, middle and last element in the list, and use that as a pivot. This prevents you from picking the highest or lowest element in the list.

Create a new file named **QuicksortMedian.kt** and add the following function:

```
fun<T: Comparable<T>> MutableList<T>.medianOfThree(low: Int,
high: Int): Int {
    val center = (low + high) / 2
    if (this[low] > this[center]) {
        this.swapAt(low, center)
    }
    if (this[low] > this[high]) {
        this.swapAt(low, high)
    }
    if (this[center] > this[high]) {
        this.swapAt(center, high)
    }
}
```

```

    }
    return center
}

```

Here, you find the median of `this[low]`, `this[center]` and `this[high]` by sorting them. The median will end up at index `center`, which is what the function returns.

Next, you'll implement a variant of Quicksort using this median of three:

```

fun<T: Comparable<T>> MutableList<T>.quickSortMedian(low: Int,
high: Int) {
    if (low < high) {
        val pivotIndex = medianOfThree(low, high)
        this.swapAt(pivotIndex, high)
        val pivot = partitionLomuto(low, high)
        this.quicksortLomuto(low, pivot - 1)
        this.quicksortLomuto(pivot + 1, high)
    }
}

```

This is a variation on `quicksortLomuto` that adds a median of three as a first step.

Try this by adding the following in your playground:

```

"Median of three quicksort" example {
    val list = arrayListOf(12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8,
    -1, 8)
    println("Original: $list")
    list.quickSortMedian(0, list.size - 1)
    println("Sorted: $list")
}

```

This is definitely an improvement, but can you do better?

Dutch national flag partitioning

A problem with Lomuto's and Hoare's algorithms is that they don't handle duplicates really well. With Lomuto's algorithm, duplicates end up in the *less than* partition and aren't grouped together. With Hoare's algorithm, the situation is even worse as duplicates can be all over the place.

A solution to organize duplicate elements is using **Dutch national flag partitioning**. This technique is named after the Dutch flag, which has three bands of colors: red, white and blue. This is similar to how you create three partitions. Dutch national flag partitioning is a good technique to use if you have a lot of duplicate elements.



Create a file named **QuicksortDutchFlag.kt** and add the following function:

```
fun<T: Comparable<T>> MutableList<T>.partitionDutchFlag(low: Int, high: Int, pivotIndex: Int): Pair<Int, Int> {
    val pivot = this[pivotIndex]
    var smaller = low // 1
    var equal = low // 2
    var larger = high // 3
    while (equal <= larger) { // 4
        if (this[equal] < pivot) {
            this.swapAt(smaller, equal)
            smaller += 1
            equal += 1
        } else if (this[equal] == pivot) {
            equal += 1
        } else {
            this.swapAt(equal, larger)
            larger -= 1
        }
    }
    return Pair(smaller, larger) // 5
}
```

You'll adopt the same strategy as Lomuto's partition by choosing the last element as the `pivotIndex`. Here's how it works:

1. Whenever you encounter an element that is less than the pivot, move it to index `smaller`. This means that all elements that come before this index are less than the pivot.
2. Index `equal` points to the next element to compare. Elements that are equal to the pivot are skipped, which means that all elements between `smaller` and `equal` are equal to the pivot.
3. Whenever you encounter an element that is greater than the pivot, move it to index `larger`. This means that all elements that come after this index are greater than the pivot.
4. The main loop compares elements and swaps them if needed. This continues until index `equal` moves past index `larger`, meaning all elements have been moved to their correct partition.
5. The algorithm returns indices `smaller` and `larger`. These point to the first and last elements of the middle partition.

Step-by-step

Looking at an example using the **unsorted** list below:

```
[ 12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8 ]
```

Since this algorithm is independent of a pivot selection strategy, you'll adopt lomuto and pick the last element **8**.

Note: I challenge you to try a different strategy, such as median of three.

Next, you set up the indices **smaller**, **equal** and **larger**:

```
[12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8]
  s
  e
                l
```

The first element to be compared is **12**. Since it's larger than the pivot, it's swapped with the element at index **larger** and this index is decremented.

Note that index **equal** is not incremented so the element that was swapped in **(8)** is compared next:

```
[8, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 12]
  s
  e
                l
```

Remember that the pivot you selected is still **8**. **8** is equal to the pivot, so you increment **equal**:

```
[8, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 12]
  s
    e
                l
```

0 is smaller than the pivot, so you swap the elements at **equal** and **smaller** and increase both pointers:

```
[0, 8, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 12]
  s
    e
                l
```

And so on.

Note how `smaller`, `equal` and `larger` partition the list:

- Elements in `this.subList(low, smaller)` are smaller than the pivot.
- Elements in `this.subList(smaller, equal)` are equal to the pivot.
- Elements in `this.subList(larger, high)` are larger than the pivot.
- Elements in `this.subList(equal, larger + 1)` haven't been compared yet.

To understand how and when the algorithm ends, let's continue from the second-to-last step:

```
[0, 3, -1, 2, 5, 8, 8, 27, 1, 18, 21, 9, 12]
      s
          e
            l
```

Here, **27** is being compared. It's greater than the pivot, so it's swapped with **1** and index `larger` is decremented:

```
[0, 3, -1, 2, 5, 8, 8, 1, 27, 18, 21, 9, 12]
      s
          e
            l
```

Even though `equal` is now equal to `larger`, the algorithm isn't complete.

The element currently at `equal` hasn't been compared yet. It's smaller than the pivot, so it's swapped with **8** and both indices `smaller` and `equal` are incremented:

```
[0, 3, -1, 2, 5, 1, 8, 8, 27, 18, 21, 9, 12]
      s
          e
            l
```

Indices `smaller` and `larger` now point to the first and last elements of the middle partition. By returning them, the function clearly marks the boundaries of the three partitions.

You're now ready to implement a new version of quicksort using Dutch national flag partitioning:

```
fun<T: Comparable<T>> MutableList<T>.quicksortDutchFlag(low:
    Int, high: Int) {
    if (low < high) {
```

```
    val middle = partitionDutchFlag(low, high, high)
    this.quicksortDutchFlag(low, middle.first - 1)
    this.quicksortDutchFlag(middle.second + 1, high)
}
}
```

Notice how recursion uses the `middleFirst` and `middleLast` indices to determine the partitions that need to be sorted recursively. Because the elements equal to the pivot are grouped together, they can be excluded from the recursion.

Try out your new quicksort by adding the following in your **Main.kt**:

```
"Dutch flag quicksort" example {
    val list = arrayListOf(12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8,
    -1, 8)
    println("Original: $list")
    list.quicksortDutchFlag( 0, list.size - 1)
    println("Sorted: $list")
}
```

That's it!

Challenges

Challenge 1: Using recursion

In this chapter, you learned how to implement quicksort recursively. Your challenge is to implement it iteratively. Choose any partition strategy you learned in this chapter.

Solution 1

You implemented quicksort recursively, which means you know what quicksort is all about. So, how you might do it iteratively? This solution uses Lomuto's partition strategy.

This function takes in a list and the range between `low` and `high`. You're going to leverage the stack to store pairs of `start` and `end` values.

```
fun<T: Comparable<T>>
    MutableList<T>.quicksortIterativeLomuto(low: Int, high: Int) {
    val stack = Stack<Int>() // 1
    stack.push(low) // 2
    stack.push(high)

    while (!stack.isEmpty) { // 3
        // 4
        val end = stack.pop() ?: continue
        val start = stack.pop() ?: continue
        val p = this.partitionLomuto(start, end) // 5
        if ((p - 1) > start) { // 6
            stack.push(start)
            stack.push(p - 1)
        }
        if ((p + 1) < end) { // 7
            stack.push(p + 1)
            stack.push(end)
        }
    }
}
```

Here's how the solution works:

1. Create a stack that stores indices.
2. Push the starting `low` and `high` boundaries on the stack to initiate the algorithm.
3. As long as the stack is not empty, quicksort is not complete.

4. Get the pair of `start` and `end` indices from the stack.
5. Perform Lomuto's partitioning with the current `start` and `end` index. Recall that Lomuto picks the last element as the pivot, and splits the partitions into three parts: elements that are less than the pivot, the pivot, and finally elements that are greater than the pivot.
6. Once the partitioning is complete, check and add the lower bound's `start` and `end` indices to later partition the lower half.
7. Similarly, check and add the upper bound's `start` and `end` indices to later partition the upper half.

You're simply using the stack to store a pair of `start` and `end` indices to perform the partitions.

Now, check to see if your iterative version of quicksort works:

```
"Iterative lomuto quicksort" example {
    val list = arrayListOf(12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8,
    -1, 8)
    println("Original: $list")
    list.quicksortIterativeLomuto( 0, list.size - 1)
    println("Sorted: $list")
}
```

Challenge 2: Provide an explanation

Explain when and why you would use merge sort over quicksort.

Solution 2

- Merge sort is preferable over quicksort when you need stability. Merge sort is a stable sort and guarantees $O(n \log n)$. This is not the case with quicksort, which isn't stable and can perform as bad as $O(n^2)$.
- Merge sort works better for larger data structures or data structures where elements are scattered throughout memory. Quicksort works best when elements are stored in a contiguous block.

Key points

- The naive partitioning creates a new list on every filter function; this is inefficient. All other strategies sort in place.
- **Lomuto's** partitioning chooses the last element as the pivot.
- **Hoare's** partitioning chooses the first element as its pivot.
- An ideal pivot would split the elements evenly between partitions.
- Choosing a bad pivot can cause quicksort to perform in $O(n^2)$.
- **Median of three** finds the pivot by taking the median of the first, middle and last element.
- **Dutch national flag** partitioning strategy helps to organize duplicate elements in a more efficient way.

Section V: Graphs

Graphs are an extremely useful data structure that can be used to model a wide range of things: webpages on the internet, the migration patterns of birds, protons in the nucleus of an atom. This section gets you thinking deeply (and broadly) about how to use graphs and graph algorithms to solve real-world problems. The chapters that follow will give the foundation you need to understand graph data structures. Like previous sections, every other chapter will serve as a Challenge chapter so you can practice what you've learned.

The graph-related topics covered include:

- **Chapter 19: Graphs:** What do social networks have in common with booking cheap flights around the world? You can represent both of these real-world models as graphs. A graph is a data structure that captures relationships between objects. It's made up of vertices connected by edges. In a weighted graph, every edge has a weight associated with it that represents the cost of using this edge. This lets you choose the cheapest or shortest path between two vertices.
- **Chapter 20: Breadth-First Search:** In the previous chapter, you explored how graphs can be used to capture relationships between objects. Several algorithms exist to traverse or search through a graph's vertices. One such algorithm is the breadth-first-search algorithm, which you can use to solve a wide variety of problems, including generating a minimum spanning tree, finding potential paths between vertices and finding the shortest path between two vertices.
- **Chapter 21: Depth-First Search:** In the previous chapter, you looked at breadth-first-search, where you had to explore every neighbor of a vertex before going to the next level. In this chapter, you'll look at depth-first search, which has applications for topological sorting, detecting cycles, pathfinding in maze puzzles and finding connected components in a sparse graph.
- **Chapter 22: Dijkstra's Algorithm:** Have you ever used the Google or Apple Maps app to find the shortest or fastest from one place to another? Dijkstra's algorithm is particularly useful in GPS networks to help find the shortest path between two



places. Dijkstra's algorithm is a greedy algorithm that constructs a solution step-by-step and picks the most optimal path at every step.

- **Chapter 23: Prim's Algorithm:** In previous chapters, you looked at depth-first and breadth-first search algorithms. These algorithms form spanning trees. In this chapter, you'll look at Prim's algorithm, a greedy algorithm used to construct a minimum spanning tree. A minimum spanning tree is a spanning tree with weighted edges where the total weight of all edges is minimized. You'll learn how to implement a greedy algorithm to construct a solution step-by-step and pick the most optimal path at every step.

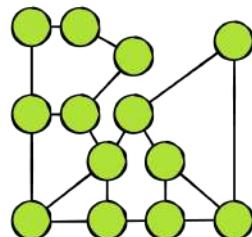
After completing this section, you'll have powerful tools at your disposal to model and solve important real-life problems using graphs.

Chapter 19: Graphs

By Irina Galata

What do social networks have in common with booking cheap flights around the world? You can represent both of these real-world models as **graphs**!

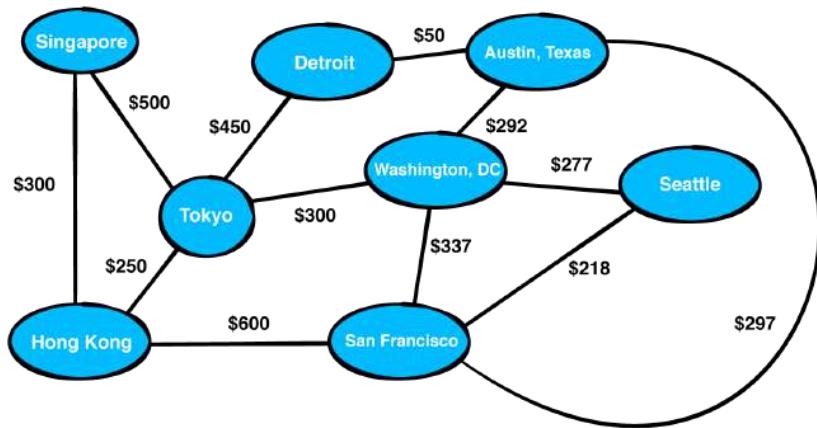
A graph is a data structure that captures relationships between objects. It's made up of **vertices** connected by **edges**. In the graph below, the vertices are represented by circles, and the edges are the lines that connect them.



Weighted graphs

In a **weighted graph**, every edge has a weight associated with it that represents the cost of using this edge. This lets you choose the cheapest or shortest path between two vertices.

Take the airline industry as an example, and think of a network with varying flight paths:



In this example, the vertices represent a state or country, while the edges represent a route from one place to another. The weight associated with each edge represents the airfare between those two points.

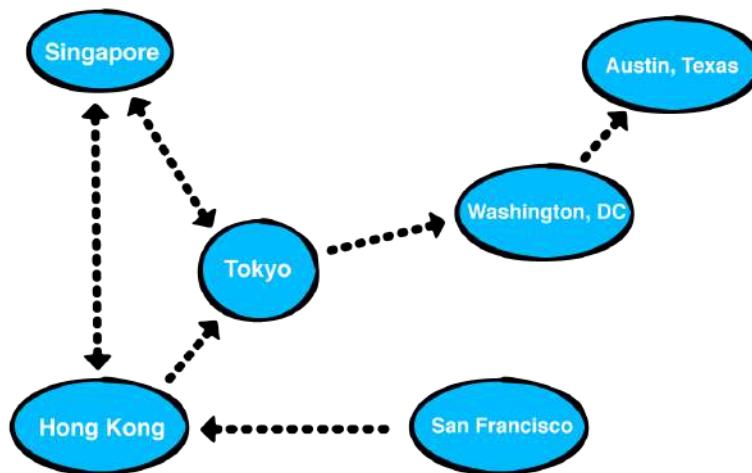
Using this network, you can determine the cheapest flights from San Francisco to Singapore for all those budget-minded digital nomads out there.

Directed graphs

As well as assigning a weight to an edge, your graphs can also have **direction**.

Directed graphs are more restrictive to traverse, as an edge may only permit traversal in one direction.

The diagram below represents a directed graph.



A directed graph

You can tell a lot from this diagram:

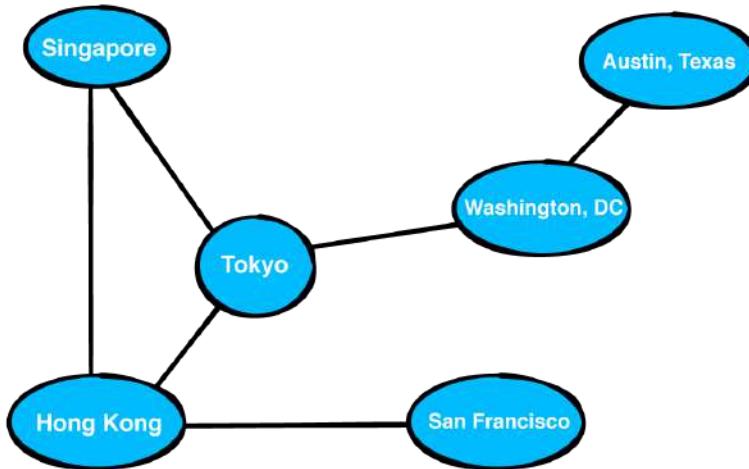
- There is a flight from Hong Kong to Tokyo.
- There is no direct flight from San Francisco to Tokyo.
- You can buy a roundtrip ticket between Singapore and Tokyo.
- There is no way to get from Tokyo to San Francisco.

Undirected graphs

You can think of an undirected graph as a directed graph where all of the edges are bi-directional.

In an undirected graph:

- Two connected vertices have edges going back and forth.
- The weight of an edge applies to both directions.



An undirected graph

Common operations

It's time to establish an interface for graphs.

Open the starter project for this chapter. Create a new file named **Graph.kt** and add the following inside the file:

```
interface Graph<T> {  
    fun createVertex(data: T): Vertex<T>  
    fun addDirectedEdge(source: Vertex<T>,  
                        destination: Vertex<T>,  
                        weight: Double?)
```

```
fun addUndirectedEdge(source: Vertex<T>,
                      destination: Vertex<T>,
                      weight: Double?)

fun add(edge: EdgeType,
        source: Vertex<T>,
        destination: Vertex<T>,
        weight: Double?)

fun edges(source: Vertex<T>): ArrayList<Edge<T>>

fun weight(source: Vertex<T>,
           destination: Vertex<T>): Double?

}

enum class EdgeType {
    DIRECTED,
    UNDIRECTED
}
```

This interface describes the common operations for a graph:

- **createVertex()**: Creates a vertex and adds it to the graph.
- **addDirectedEdge()**: Adds a directed edge between two vertices.
- **addUndirectedEdge()**: Adds an undirected (or bi-directional) edge between two vertices.
- **add()**: Uses EdgeType to add either a directed or undirected edge between two vertices.
- **edges()**: Returns a list of outgoing edges from a specific vertex.
- **weight()**: Returns the weight of the edge between two vertices.

In the following sections, you'll implement this interface in two ways:

- Using an adjacency list.
- Using an adjacency matrix.

Before you can do that, you must first build types to represent vertices and edges.

Defining a vertex



A collection of vertices – not yet a graph

Create a new file named **Vertex.kt** and add the following inside the file:

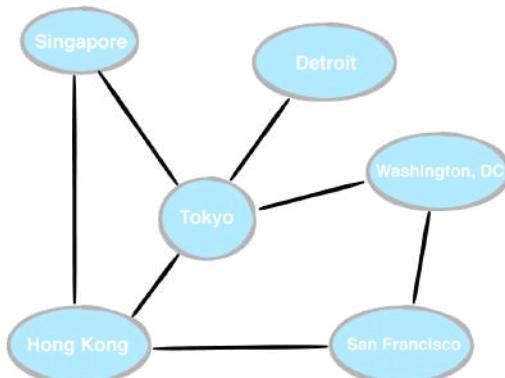
```
data class Vertex<T>(val index: Int, val data: T)
```

Here, you've defined a generic **Vertex** class. A vertex has a unique index within its graph and holds a piece of data.

You defined **Vertex** as a **data class** because it will be used as a key in a Map later, and a data class gives you `equals()` and `hashCode()` implementations, without having to write them yourself.

Defining an edge

To connect two vertices, there must be an edge between them.



Edges added to the collection of vertices

Create a new file named **Edge.kt** and add the following inside the file:

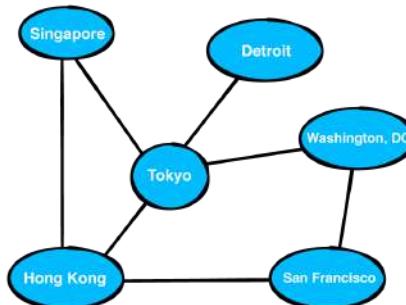
```
data class Edge<T>(
    val source: Vertex<T>,
    val destination: Vertex<T>,
    val weight: Double? = null
)
```

An Edge connects two vertices and has an optional weight.

Adjacency list

The first graph implementation that you'll learn uses an **adjacency list**. For every vertex in the graph, the graph stores a list of outgoing edges.

Take as an example the following network:



The adjacency list below describes the network of flights depicted above:

Vertices	Adjacency Lists		
Singapore	→ TOK	→ HK	
Hong Kong	→ TOK	→	→ SING
Tokyo	→ SG	→ DET	→ WDC
Detroit	→ TOK		
Washington DC	→ TOK	→ SF	
San Francisco	→ HK	→	→ WDC

There's a lot you can learn from this adjacency list:

1. Singapore's vertex has two outgoing edges. There's a flight from Singapore to Tokyo and Hong Kong.
2. Detroit has the smallest number of outgoing traffic.
3. Tokyo is the busiest airport, with the most outgoing flights.

In the next section, you'll create an adjacency list by storing a **map of arrays**. Each key in the map is a vertex, and in every vertex, the map holds a corresponding array of edges.

Implementation

Create a new file named **AdjacencyList.kt** and add the following:

```
class AdjacencyList<T> : Graph<T> {  
  
    private val adjacencies: HashMap<Vertex<T>,  
    ArrayList<Edge<T>>> = HashMap()  
  
    // more to come ...  
}
```

Here, you've defined an **AdjacencyList** that uses a map to store the edges.

You've already implemented the **Graph** interface but still need to implement its requirements. That's what you'll do in the following sections.

Creating a vertex

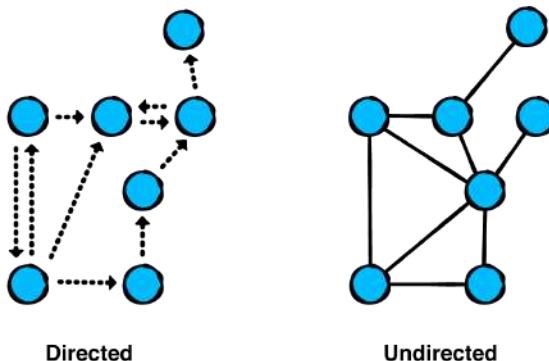
Add the following method to **AdjacencyList**:

```
override fun createVertex(data: T): Vertex<T> {  
    val vertex = Vertex(adjacencies.count(), data)  
    adjacencies[vertex] = ArrayList()  
    return vertex  
}
```

Here, you create a new vertex and return it. In the adjacency list, you store an empty list of edges for this new vertex.

Creating a directed edge

Recall that there are **directed** and **undirected** graphs.



Start by implementing the `addDirectedEdge` requirement. Add the following method:

```
override fun addDirectedEdge(source: Vertex<T>, destination: Vertex<T>, weight: Double?) {
    val edge = Edge(source, destination, weight)
    adjacencies[source]?.add(edge)
}
```

This method creates a new edge and stores it in the adjacency list.

Creating an undirected edge

You just created a method to add a directed edge between two vertices. How would you create an undirected edge between two vertices?

Remember that an undirected graph can be viewed as a bidirectional graph. Every edge in an undirected graph can be traversed in both directions. This is why you'll implement `addUndirectedEdge()` on top of `addDirectedEdge()`. Because this implementation is reusable, you'll add it as a default implementation in `Graph`.

In **Graph.kt**, update `addUndirectedEdge()`:

```
fun addUndirectedEdge(source: Vertex<T>, destination: Vertex<T>,
    weight: Double?) {
    addDirectedEdge(source, destination, weight)
    addDirectedEdge(destination, source, weight)
}
```

Adding an undirected edge is the same as adding two directed edges.

Now that you've implemented both `addDirectedEdge()` and `addUndirectedEdge()`, you can implement `add()` by delegating to one of these methods. Update the `add()` method of `Graph` as well:

```
fun add(edge: EdgeType, source: Vertex<T>, destination:
    Vertex<T>, weight: Double?) {
    when (edge) {
        EdgeType.DIRECTED -> addDirectedEdge(source, destination,
            weight)
        EdgeType.UNDIRECTED -> addUndirectedEdge(source,
            destination, weight)
    }
}
```

`add()` is a convenient helper method that creates either a directed or undirected edge. This is where interfaces with default methods can become very powerful. Anyone that implements the `Graph` interface only needs to implement `addDirectedEdge()` in order to get `addUndirectedEdge()` and `add()` for free.

Retrieving the outgoing edges from a vertex

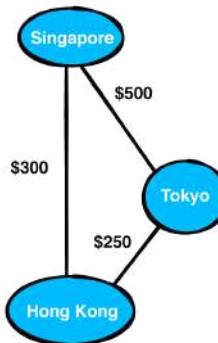
Back in `AdjacencyList.kt`, continue your work on implementing `Graph` by adding the following method:

```
override fun edges(source: Vertex<T>) =
    adjacencies[source] ?: arrayListOf()
```

This is a straightforward implementation: You either return the stored edges or an empty list if the `source` vertex is unknown.

Retrieving the weight of an edge

How much is the flight from Singapore to Tokyo?



Add the following immediately after edges():

```

override fun weight(source: Vertex<T>, destination: Vertex<T>): Double? {
    return edges(source).firstOrNull { it.destination == destination }?.weight
}
  
```

Here, you find the first edge from source to destination; if there is one, you return its weight.

Visualizing the adjacency list

Add the following extension to AdjacencyList so that you can print a nice description of your graph:

```

override fun toString(): String {
    return buildString { // 1
        adjacencies.forEach { (vertex, edges) -> // 2
            val edgeString = edges.joinToString(
                { it.destination.data.toString() }) // 3
            append("${vertex.data} --> [ $edgeString ]\n") // 4
        }
    }
}
  
```

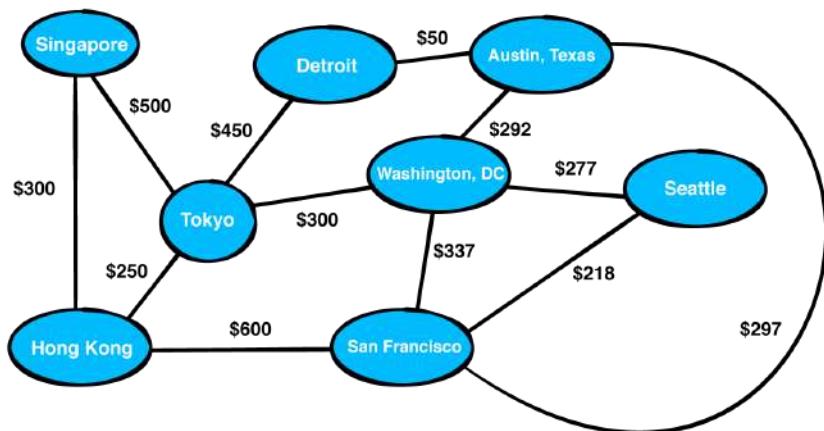
Here's what's going on in the previous code:

1. You'll be assembling the result using `buildString()`, which places you inside the scope of a `StringBuilder`, and returns whatever you've built.
2. You loop through every key-value pair in `adjacencies`.
3. For every vertex, you create a string representation of all its outgoing edges. `joinToString()` gives you a neat, comma-separated list of the items.
4. Finally, for every vertex, you append both the vertex itself and its outgoing edges to the `StringBuilder` that `buildString()` provides you with.

You've finally completed your first graph. You're ready to try it out by building a network.

Building a network

Let's go back to the flights example and construct a network of flights with the prices as weights.



Within `main()`, add the following code:

```

val graph = AdjacencyList<String>()

val singapore = graph.createVertex("Singapore")
val tokyo = graph.createVertex("Tokyo")
val hongKong = graph.createVertex("Hong Kong")
val detroit = graph.createVertex("Detroit")
val sanFrancisco = graph.createVertex("San Francisco")
  
```

```

val washingtonDC = graph.createVertex("Washington DC")
val austinTexas = graph.createVertex("Austin Texas")
val seattle = graph.createVertex("Seattle")

graph.add(EdgeType.UNDIRECTED, singapore, hongKong, 300.0)
graph.add(EdgeType.UNDIRECTED, singapore, tokyo, 500.0)
graph.add(EdgeType.UNDIRECTED, hongKong, tokyo, 250.0)
graph.add(EdgeType.UNDIRECTED, tokyo, detroit, 450.0)
graph.add(EdgeType.UNDIRECTED, tokyo, washingtonDC, 300.0)
graph.add(EdgeType.UNDIRECTED, hongKong, sanFrancisco, 600.0)
graph.add(EdgeType.UNDIRECTED, detroit, austinTexas, 50.0)
graph.add(EdgeType.UNDIRECTED, austinTexas, washingtonDC, 292.0)
graph.add(EdgeType.UNDIRECTED, sanFrancisco, washingtonDC,
337.0)
graph.add(EdgeType.UNDIRECTED, washingtonDC, seattle, 277.0)
graph.add(EdgeType.UNDIRECTED, sanFrancisco, seattle, 218.0)
graph.add(EdgeType.UNDIRECTED, austinTexas, sanFrancisco, 297.0)

println(graph)

```

You'll get the following output in your console:

```

Detroit ---> [ Tokyo, Austin, Texas ]
Hong Kong ---> [ Singapore, Tokyo, San Francisco ]
Singapore ---> [ Hong Kong, Tokyo ]
Washington, DC ---> [ Tokyo, Austin, Texas, San Francisco,
Seattle ]
Tokyo ---> [ Singapore, Hong Kong, Detroit, Washington, DC ]
San Francisco ---> [ Hong Kong, Washington, DC, Seattle, Austin,
Texas ]
Austin, Texas ---> [ Detroit, Washington, DC, San Francisco ]
Seattle ---> [ Washington, DC, San Francisco ]

```

Pretty cool, huh? This shows a visual description of an adjacency list. You can clearly see all of the outbound flights from any place.

You can also obtain other useful information such as:

- How much is a flight from Singapore to Tokyo?

```
println(graph.weight(singapore, tokyo))
```

- What are all the outgoing flights from San Francisco?

```

println("San Francisco Outgoing Flights:")
println("-----")
graph.edges(sanFrancisco).forEach { edge ->
    println("from: ${edge.source.data} to: ${edge.destination.data}")
}

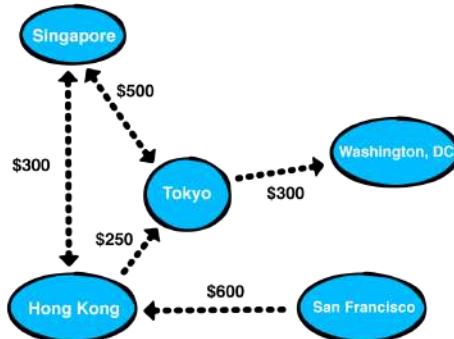
```

You have just created a graph using an adjacency list, wherein you used a map to store the outgoing edges for every vertex. Let's take a look at a different approach to how to store vertices and edges.

Adjacency matrix

An **adjacency matrix** uses a square matrix to represent a graph. This matrix is a two-dimensional array wherein the value of `matrix[row][column]` is the weight of the edge between the vertices at `row` and `column`.

Below is an example of a directed graph that depicts a flight network traveling to different places. The weight represents the cost of the airfare.



The following adjacency matrix describes the network for the flights depicted above.

Edges that don't exist have a weight of 0.

		Columns				
		Vertices				
0	Singapore	0	\$300	\$500	0	0
1	Hong Kong	\$300	0	\$250	0	0
2	Tokyo	\$500	0	0	\$300	0
3	Washington, DC	0	0	0	0	0
4	San Francisco	0	\$600	0	0	0

Compared to an adjacency list, this matrix is a little more challenging to read.

Using the array of vertices on the left, you can learn a lot from the matrix. For example:

- [0] [1] is 300, so there is a flight from Singapore to Hong Kong for \$300.
- [2] [1] is 0, so there is no flight from Tokyo to Hong Kong.
- [1] [2] is 250, so there is a flight from Hong Kong to Tokyo for \$250.
- [2] [2] is 0, so there is no flight from Tokyo to Tokyo!

Note: There's a pink line in the middle of the matrix. When the row and column are equal, this represents an edge between a vertex and itself, which is not allowed.

Implementation

Create a new file named **AdjacencyMatrix.kt** and add the following to it:

```
class AdjacencyMatrix<T> : Graph<T> {  
  
    private val vertices = arrayListOf<Vertex<T>>()  
    private val weights = arrayListOf<ArrayList<Double?>>()  
  
    // more to come ...  
}
```

Here, you've defined an **AdjacencyMatrix** that contains an array of vertices and an adjacency matrix to keep track of the edges and their weights.

Just as before, you've already declared the implementation of **Graph** but still need to implement the requirements.

Creating a Vertex

Add the following method to **AdjacencyMatrix**:

```
override fun createVertex(data: T): Vertex<T> {  
    val vertex = Vertex(vertices.count(), data)  
    vertices.add(vertex) // 1  
    weights.forEach {
```

```

        it.add(null) // 2
    }
    weights.add(arrayListOf()) // 3
    return vertex
}
override fun createVertex(data: T): Vertex<T> {
    val vertex = Vertex(vertices.count(), data)
    vertices.add(vertex) // 1
    weights.forEach {
        it.add(null) // 2
    }
    val row = ArrayList<Double?>(vertices.count())
    repeat(vertices.count()) {
        row.add(null)
    }
    weights.add(row) // 3
    return vertex
}

```

To create a vertex in an adjacency matrix, you:

1. Add a new vertex to the array.
2. Append a null weight to every row in the matrix, as none of the current vertices have an edge to the new vertex.

		Columns					
		0	1	2	3	4	5
Rows	0	0	\$300	\$500	0	0	0
	1	\$300	0	\$250	0	0	0
	2	\$500	0	0	\$300	0	0
	3	0	0	0	0	0	0
	4	0	\$600	0	0	0	0
							+

3. Add a new row to the matrix. This row holds the outgoing edges for the new vertex. You put a `null` value in this row for each vertex that your graph stores.

		Columns					
		0	1	2	3	4	5
Rows	0	0	\$300	\$500	0	0	0
	1	\$300	0	\$250	0	0	0
	2	\$500	0	0	\$300	0	0
	3	0	0	0	0	0	0
	4	0	\$600	0	0	0	0
	5	0	0	0	0	0	0

Creating edges

Creating edges is as simple as filling in the matrix. Add the following method:

```
override fun addDirectedEdge(
    source: Vertex<T>,
    destination: Vertex<T>,
    weight: Double?
) {
    weights[source.index][destination.index] = weight
}
```

Remember that `addUndirectedEdge()` and `add()` have a default implementation in the interface, so this is all you need to do.

Retrieving the outgoing edges from a vertex

Add the following method:

```
override fun edges(source: Vertex<T>): ArrayList<Edge<T>> {
    val edges = arrayListOf<Edge<T>>()
    (0 until weights.size).forEach { column ->
        val weight = weights[source.index] [column]
        if (weight != null) {
            edges.add(Edge(source, vertices[column], weight))
        }
    }
    return edges
}
```

To retrieve the outgoing edges for a vertex, you search the row for this vertex in the matrix for weights that are not null.

Every non-null weight corresponds with an outgoing edge. The destination is the vertex that corresponds with the column in which the weight was found.

Retrieving the weight of an edge

It's easy to get the weight of an edge; simply look up the value in the adjacency matrix. Add this method:

```
override fun weight(
    source: Vertex<T>,
    destination: Vertex<T>
): Double? {
    return weights[source.index] [destination.index]
}
```

Visualize an adjacency matrix

Finally, add the following extension so you can print a nice, readable description of your graph:

```
override fun toString(): String {
    // 1
    val verticesDescription = vertices.joinToString("\n") { "$
{it.index}: ${it.data}" }

    // 2
    val grid = arrayListOf<String>()
    weights.forEach {
```

```
var row = ""
(0 until weights.size).forEach { columnIndex ->
    if (columnIndex >= it.size) {
        row += "\u00d8\t\t"
    } else {
        row += it[columnIndex] ?.let { "$it\t" } ?: "\u00d8\t\t"
    }
}
grid.add(row)
}
val edgesDescription = grid.joinToString("\n")
// 3
return "$verticesDescription\n\n$edgesDescription"
}

override fun toString(): String {
// 1
    val verticesDescription = vertices
        .joinToString(separator = "\n") { "${it.index}: ${it.data}" }

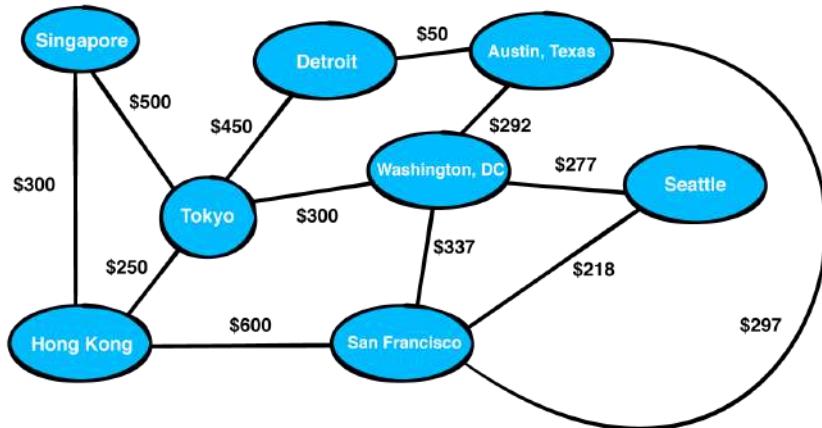
// 2
    val grid = weights.map { row ->
        buildString {
            (0 until weights.size).forEach { columnIndex ->
                val value = row[columnIndex]
                if (value != null) {
                    append("$value\t")
                } else {
                    append("\u00d8\t\t")
                }
            }
        }
    }
    val edgesDescription = grid.joinToString("\n")
// 3
    return "$verticesDescription\n\n$edgesDescription"
}
```

Here are the steps:

1. You first create a list of the vertices.
2. Then, you build up a grid of weights, row by row.
3. Finally, you join both descriptions together and return them.

Building a network

You'll reuse the same example from `AdjacencyList`:



Go to `main()` and replace:

```
val graph = AdjacencyList<String>()
```

With:

```
val graph = AdjacencyMatrix<String>()
```

`AdjacencyMatrix` and `AdjacencyList` have the same interface, so the rest of the code stays the same.

You'll get the following output in your console:

```

0: Singapore
1: Tokyo
2: Hong Kong
3: Detroit
4: San Francisco
5: Washington DC
6: Austin Texas
7: Seattle

      0       1       2       3       4       5       6       7
0   ø     500.0   300.0   ø     ø     ø     ø     ø
1  500.0     ø   250.0   450.0   ø   300.0   ø     ø
2  300.0   250.0     ø     ø   600.0   ø     ø     ø
3     ø   450.0     ø     ø     ø     ø   50.0   ø
4     ø     ø   600.0     ø     ø     ø   337.0  297.0
5     ø     ø     ø     ø     ø     ø     ø     ø
6     ø     ø     ø     ø     ø     ø     ø     ø
7     ø     ø     ø     ø     ø     ø     ø     ø
  
```

218.0							
\emptyset	300.0	\emptyset	\emptyset	337.0	\emptyset	292.0	
277.0							
\emptyset	\emptyset	\emptyset	50.0	297.0	292.0	\emptyset	\emptyset
\emptyset	\emptyset	\emptyset	\emptyset	218.0	277.0	\emptyset	\emptyset

San Francisco Outgoing Flights:

```
from: San Francisco to: Hong Kong
from: San Francisco to: Washington, DC
from: San Francisco to: Austin, Texas
from: San Francisco to: Seattle
```

In terms of visual beauty, an adjacency list is a lot easier to follow and trace than an adjacency matrix. Next, you'll analyze the common operations of these two approaches and see how they perform.

Graph analysis

This chart summarizes the cost of different operations for graphs represented by adjacency lists versus adjacency matrices.

Operations	Adjacency List	Adjacency Matrix
Storage Space	$O(V + E)$	$O(V^2)$
Add Vertex	$O(1)$	$O(V^2)$
Add Edge	$O(1)$	$O(1)$
Finding Edges and Weight	$O(V)$	$O(1)$

V represents vertices, and **E** represents edges.

An adjacency list takes less storage space than an adjacency matrix. An adjacency list simply stores the number of vertices and edges needed. As for an adjacency matrix, recall that the number of rows and columns is equal to the number of vertices. This explains the quadratic space complexity of $O(V^2)$.

Adding a vertex is efficient in an adjacency list: Simply create a vertex and set its key-value pair in the map. It's amortized as $O(1)$. When adding a vertex to an adjacency matrix, you're required to add a column to every row and create a new row for the new vertex. This is at least $O(V)$, and if you choose to represent your matrix with a contiguous block of memory, it can be $O(V^2)$.

Adding an edge is efficient in both data structures, as they are both constant time. The adjacency list appends to the array of outgoing edges. The adjacency matrix sets the value in the two-dimensional array.

Adjacency list loses out when trying to find a particular edge or weight. To find an edge in an adjacency list, you must obtain the list of outgoing edges and loop through every edge to find a matching destination. This happens in $O(V)$ time. With an adjacency matrix, finding an edge or weight is a constant time access to retrieve the value from the two-dimensional array.

Which data structure should you choose to construct your graph?

If there are few edges in your graph, it's considered a **sparse graph**, and an adjacency list would be a good fit. An adjacency matrix would be a bad choice for a sparse graph, because a lot of memory will be wasted since there aren't many edges.

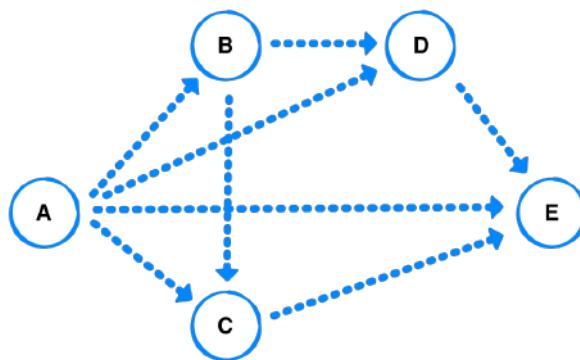
If your graph has lots of edges, it's considered a **dense graph**, and an adjacency matrix would be a better fit as you'd be able to access your weights and edges far more quickly.

- Adjacency matrix uses a square matrix to represent a graph.
- Adjacency list is generally good for **sparse graphs**, when your graph has the least amount of edges.
- Adjacency matrix is generally suitable for **dense graphs**, when your graph has lots of edges.

Challenges

Challenge 1: Find the distance between 2 vertices

Write a method to count the number of paths between two vertices in a directed graph. The example graph below has 5 paths from **A** to **E**:



Solution 1

The goal is to write a function that finds the number of paths between two vertices in a graph. One solution is to perform a depth-first traversal and keep track of the visited vertices.

```

fun numberOfPaths(
    source: Vertex<T>,
    destination: Vertex<T>
): Int {
    val numberOfPaths = Ref(0) // 1
    val visited: MutableSet<Vertex<Element>> = mutableSetOf() // 2
    paths(source, destination, visited, numberOfPaths) // 3
    return numberOfPaths.value
}
    
```

And to create a `Ref` class to pass the `Int` value by reference:

```

data class Ref<T>(var value: T)
    
```

Here, you do the following:

1. `numberOfPaths` keeps track of the number of paths found between the source and destination.
2. `visited` is an `ArrayList` that keeps track of all the vertices visited.
3. `paths` is a recursive helper function that takes in four parameters. The first two parameters are the source and destination vertices. `visited` tracks the vertices visited, and `numberOfPaths` tracks the number of paths found. These last two parameters are modified within `paths`.

Add the following immediately after `numberOfPaths()`:

```
fun numberOfPaths(source: Vertex, destination: Vertex)
```

```
fun paths(
    source: Vertex<T>,
    destination: Vertex<T>,
    visited: MutableSet<Vertex<T>>,
    pathCount: Ref<Int>
) {
    visited.add(source) // 1
    if (source == destination) { // 2
        pathCount.value += 1
    } else {
        val neighbors = edges(source) // 3
        neighbors.forEach { edge ->
            // 4
            if (edge.destination !in visited) {
                paths(edge.destination, destination, visited, pathCount)
            }
        }
    } // 5
    visited.remove(source)
}
```

To get the paths from the source to destination:

1. Initiate the algorithm by marking the source vertex as visited.
2. Check to see if the source is the destination. If it is, you have found a path, so increment the count by one.
3. If the destination has not been found, get all of the edges adjacent to the source vertex.
4. For every edge, if it has not been visited before, recursively traverse the neighboring vertices to find a path to the destination vertex.

5. Remove the source vertex from the visited list so that you can continue to find other paths to that node.

You're doing a depth-first graph traversal. You recursively dive down one path until you reach the destination, and back-track by popping off the stack. The time-complexity is $O(V + E)$.

Challenge 2: The small world

Vincent has three friends: Chesley, Ruiz and Patrick. Ruiz has friends as well: Ray, Sun and a mutual friend of Vincent's. Patrick is friends with Cole and Kerry. Cole is friends with Ruiz and Vincent. Create an adjacency list that represents this friendship graph. Which mutual friend do Ruiz and Vincent share?

Solution 2

```
val graph = AdjacencyList<String>()

val vincent = graph.createVertex("vincent")
val chesley = graph.createVertex("chesley")
val ruiz = graph.createVertex("ruiz")
val patrick = graph.createVertex("patrick")
val ray = graph.createVertex("ray")
val sun = graph.createVertex("sun")
val cole = graph.createVertex("cole")
val kerry = graph.createVertex("kerry")

graph.add(EdgeType.UNDIRECTED, vincent, chesley, 0.0)
graph.add(EdgeType.UNDIRECTED, vincent, ruiz, 0.0)
graph.add(EdgeType.UNDIRECTED, vincent, patrick, 0.0)
graph.add(EdgeType.UNDIRECTED, ruiz, ray, 0.0)
graph.add(EdgeType.UNDIRECTED, ruiz, sun, 0.0)
graph.add(EdgeType.UNDIRECTED, patrick, cole, 0.0)
graph.add(EdgeType.UNDIRECTED, patrick, kerry, 0.0)
graph.add(EdgeType.UNDIRECTED, cole, ruiz, 0.0)
graph.add(EdgeType.UNDIRECTED, cole, vincent, 0.0)

println(graph)
println("Ruiz and Vincent both share a friend name Cole")
```

Key points

- You can represent real-world relationships through **vertices** and **edges**.
- Think of **vertices** as objects and **edges** as the relationship between the objects.
- Weighted graphs associate a weight with every edge.
- Directed graphs have edges that traverse in one direction.
- Undirected graphs have edges that point both ways.
- Adjacency list stores a list of outgoing edges for every vertex.

20

Chapter 20: Breadth-First Search

By Irina Galata

In the previous chapter, you explored how you can use graphs to capture relationships between objects. Remember that objects are vertices, and the relationships between them are represented by edges.

Several algorithms exist to traverse or search through a graph's vertices. One such algorithm is the **breadth-first search** (BFS) algorithm.

You can use BFS to solve a wide variety of problems:

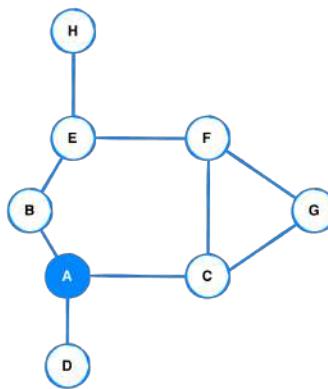
1. Generating a minimum-spanning tree.
2. Finding potential paths between vertices.
3. Finding the shortest path between two vertices.



Example

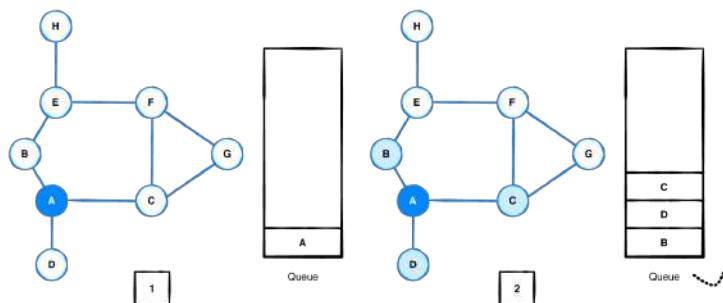
BFS starts by selecting any vertex in a graph. The algorithm then explores all neighbors of this vertex before traversing the neighbors of said neighbors and so forth. As the name suggests, this algorithm takes a **breadth-first** approach because it doesn't visit the children until all the siblings are visited.

To get a better idea of how things work, you'll look at a BFS example using the following undirected graph:



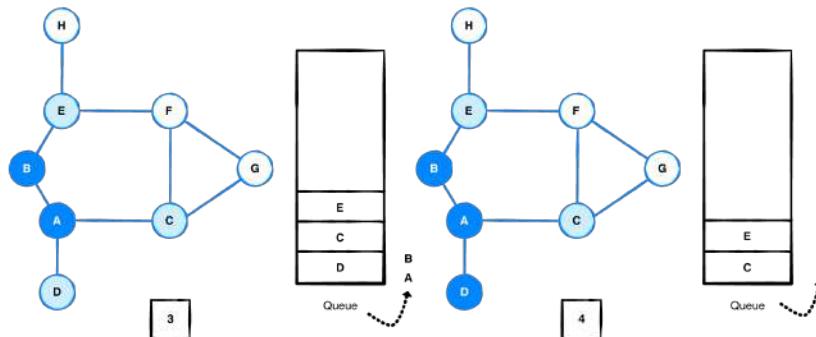
Note: Highlighted vertices represent vertices that have been visited.

To keep track of which vertices to visit next, you can use a queue. The first in, first out approach of the queue guarantees that all of a vertex's neighbors are visited before you traverse one level deeper.

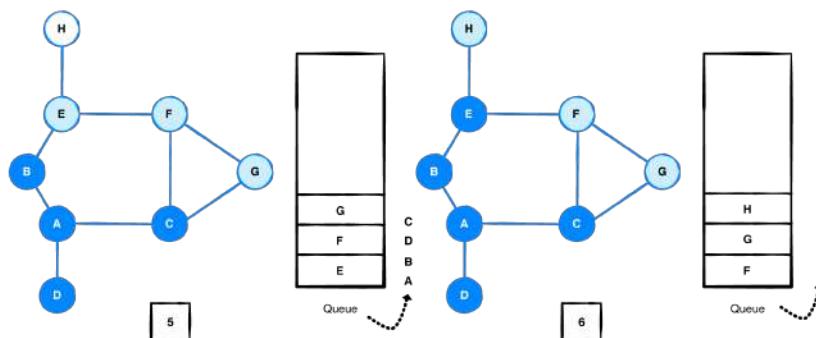


- To begin, pick a source vertex to start from. In this example, you choose A, which is added to the queue.
- As long as the queue is not empty, you can dequeue and visit the next vertex, in this case, A. Next, you need to add all A's neighboring vertices [B, D, C] to the queue.

Note: You only add a vertex to the queue when it has not yet been visited and is not already in the queue.



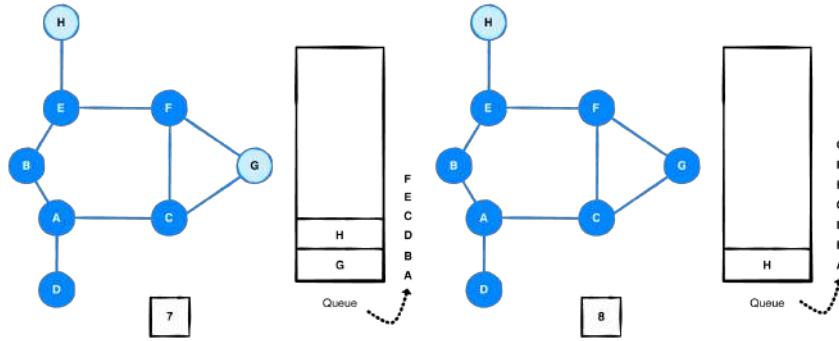
- The queue is not empty, so you need to dequeue and visit the next vertex, which is B. You then need to add B's neighbor E to the queue. A is already visited, so it doesn't get added. The queue now has [D, C, E].
- The next vertex to be dequeued is D. D does not have any neighbors that aren't visited. The queue now has [C, E].



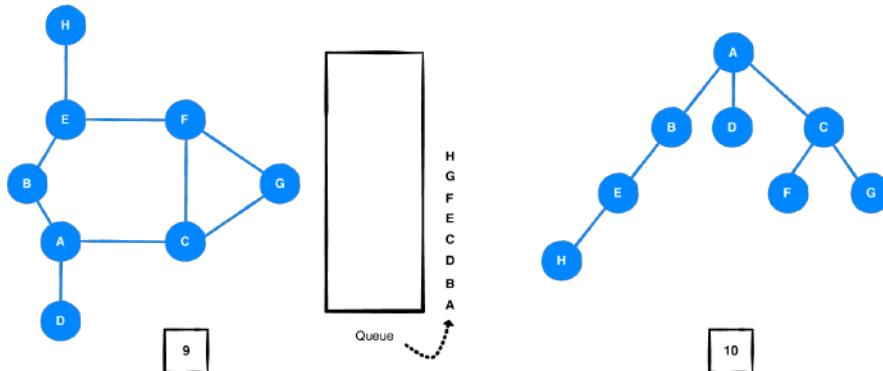
5. Next, you need to dequeue C and add its neighbors [F, G] to the queue. The queue now has [E, F, G].

Note that you have now visited all of A's neighbors, and BFS moves on to the second level of neighbors.

6. You need to dequeue E and add H to the queue. The queue now has [F, G, H]. Note that you don't add B or F to the queue because B is already visited and F is already in the queue.



7. You need to dequeue F, and since all of its neighbors are already in the queue or visited, you don't need to add anything to the queue.
8. Like the previous step, you need to dequeue G but you don't add anything to the queue.



9. Finally, you need to dequeue H. The breadth-first search is complete since the queue is now empty.

10. When exploring the vertices, you can construct a tree-like structure, showing the vertices at each level: First the vertex you started from, then its neighbors, then its neighbors' neighbors and so on.

Implementation

Open the starter playground for this chapter. This playground contains an implementation of a graph that was built in the previous chapter. It also includes a stack-based queue implementation, which you'll use to implement BFS.

In your starter project, you'll notice **Graph.kt**. Add the following method to the **Graph** class:

```
fun breadthFirstSearch(source: Vertex<T>): ArrayList<Vertex<T>>
{
    val queue = QueueStack<Vertex<T>>()
    val enqueued = ArrayList<Vertex<T>>()
    val visited = ArrayList<Vertex<T>>()

    // more to come ...

    return visited
}
```

Here, you defined the method **breadthFirstSearch()** which takes in a starting vertex. It uses three data structures:

1. **queue**: Keeps track of the neighboring vertices to visit next.
2. **enqueued**: Remembers which vertices have been enqueue, so you don't enqueue the same vertex twice.
3. **visited**: An array list that stores the order in which the vertices were explored.

Next, complete the method by replacing the comment with:

```
queue.enqueue(source) // 1
enqueued.add(source)

while (true) {
    val vertex = queue.dequeue() ?: break // 2

    visited.add(vertex) // 3

    val neighborEdges = edges(vertex) // 4
    neighborEdges.forEach {
```

```
        if (!enqueued.contains(it.destination)) { // 5
            queue.enqueue(it.destination)
            enqueued.add(it.destination)
        }
    }
}
```

Here's what's going on:

1. You initiate the BFS algorithm by first enqueueing the source vertex.
2. You continue to dequeue a vertex from the queue until the queue is empty.
3. Every time you dequeue a vertex from the queue, you add it to the list of visited vertices.
4. You then find all edges that start from the current vertex and iterate over them.
5. For each edge, you check to see if its destination vertex has been enqueued before, and if not, you add it to the code.

That's all there is to implementing BFS. It's time to give this algorithm a spin. Add the following code to the **main** method in the **Main.kt** file:

```
val vertices = graph.breadthFirstSearch(a)
vertices.forEach {
    println(it.data)
}
```

Take note of the order of the explored vertices using BFS:

```
A  
B  
C  
D  
E  
F  
G  
H
```

One thing to keep in mind with neighboring vertices is that the order in which you visit them is determined by how you construct your graph. You could have added an edge between A and C before adding one between A and B. In this case, the output would list C before B.

Performance



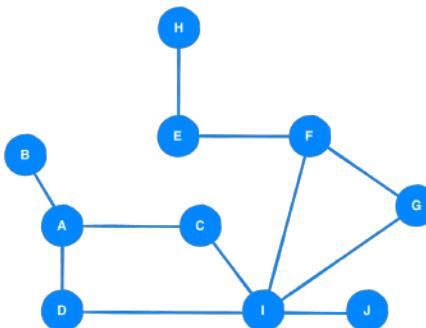
When traversing a graph using BFS, each vertex is enqueued once. This has a time complexity of $O(V)$. During this traversal, you also visit all of the edges. The time it takes to visit all edges is $O(E)$. This means that the overall time complexity for breadth-first search is $O(V + E)$.

The space complexity of BFS is $O(V)$ since you have to store the vertices in three separate structures: queue, enqueued and visited.

Challenges

Challenge 1: How many nodes?

For the following undirected graph, list the **maximum** number of items ever in the queue. Assume that the starting vertex is A.



Solution 1

The maximum number of items ever in the queue is **3**.

Challenge 2: What about recursion?

In this chapter, you went over an iterative implementation of breadth-first search. Now, write a recursive implementation.

Solution 2

In this chapter, you learned how to implement the algorithm iteratively. Let's look at how you would implement it recursively.

```
fun bfs(source: Vertex<T>): ArrayList<Vertex<T>> {
    val queue = QueueStack<Vertex<T>>() // 1
    val enqueued = arrayListOf<Vertex<T>>() // 2
    val visited = arrayListOf<Vertex<T>>() // 3

    queue.enqueue(source) // 4
    enqueued.add(source)

    bfs(queue, enqueued, visited) // 5

    return visited // 6
}
```

`bfs` takes in the source vertex to start traversing from:

1. `queue` keeps track of the neighboring vertices to visit next.
2. `enqueued` remembers which vertices have been added to the queue.
3. `visited` is a list that stores the order in which the vertices were explored.
4. Initiate the algorithm by inserting the source vertex.
5. Perform `bfs` recursively on the graph by calling a helper function.
6. Return the vertices visited in order.

The helper function looks like this:

```
private fun bfs(queue: QueueStack<Vertex<T>>, enqueued: ArrayList<Vertex<T>>, visited: ArrayList<Vertex<T>>) {
    val vertex = queue.dequeue() ?: return // 1

    visited.add(vertex) // 2

    val neighborEdges = edges(vertex) // 3
    neighborEdges.forEach {
        if (!enqueued.contains(it.destination)) { // 4
            queue.enqueue(it.destination)
        }
    }
}
```

```
        enqueueed.add(it.destination)
    }
}

bfs(queue, enqueueed, visited) // 5
```

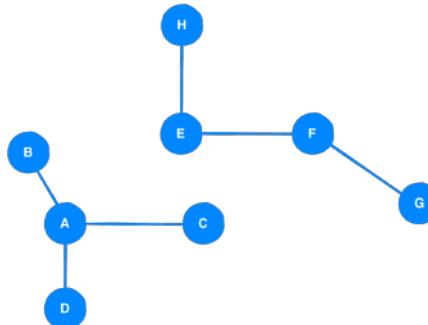
Here's how it works:

1. We start from the first node we dequeue from the queue of all vertices. Then we recursively continue to dequeue a vertex from the queue till it's empty.
2. Mark the vertex as visited.
3. For every neighboring edge from the current vertex.
4. Check to see if the adjacent vertices have been visited before inserting into the queue.
5. Recursively perform bfs until the queue is empty.

The overall time complexity for breadth-first search is $O(V + E)$.

Challenge 3: Detect disconnects

Add a method to Graph to detect if a graph is disconnected. An example of a disconnected graph is shown below:



Solution 3

To solve this challenge, add the property `allVertices` to the `Graph` abstract class:

```
abstract val allVertices: ArrayList<Vertex<T>>
```

Then, implement this property in `AdjacencyMatrix` and `AdjacencyList` respectively:

```
override val allVertices: ArrayList<Vertex<T>>
  get() = vertices
```

```
override val allVertices: ArrayList<Vertex<T>>
  get() = ArrayList(adjacencies.keys)
```

A graph is said to be disconnected if no path exists between two nodes.

```
fun isDisconnected(): Boolean {
  val firstVertex = allVertices.firstOrNull() ?: return false // 1
  val visited = breadthFirstSearch(firstVertex) // 2
  allVertices.forEach { // 3
    if (!visited.contains(it)) return true
  }
  return false
}
```

Here's how it works:

1. If there are no vertices, treat the graph as connected.
2. Perform a breadth-first search starting from the first vertex. This will return all the visited nodes.
3. Go through every vertex in the graph and check to see if it has been visited before.

The graph is considered disconnected if a vertex is missing in the visited list.

Key points

- Breadth-first search (BFS) is an algorithm for traversing or searching a graph.
- BFS explores all of the current vertex's neighbors before traversing the next level of vertices.
- It's generally good to use this algorithm when your graph structure has a lot of neighboring vertices or when you need to find out every possible outcome.
- The queue data structure is used to prioritize traversing a vertex's neighboring edges before diving down a level deeper.

21

Chapter 21: Depth-First Search

By Irina Galata

In the previous chapter, you looked at breadth-first search (BFS) in which you had to explore every neighbor of a vertex before going to the next level. In this chapter, you'll look at **depth-first search (DFS)**, another algorithm for traversing or searching a graph.

There are many applications for DFS:

- Topological sorting.
- Detecting a cycle.
- Pathfinding, such as in maze puzzles.
- Finding connected components in a sparse graph.

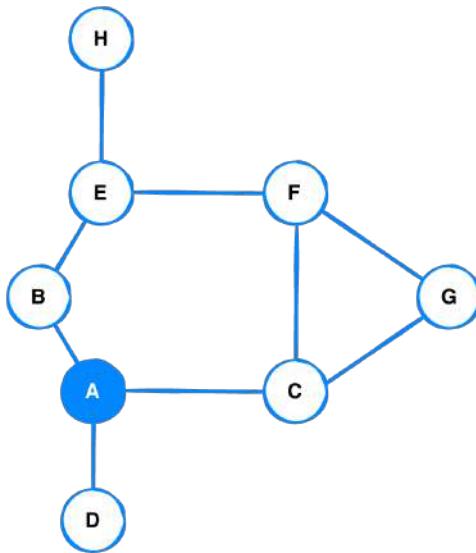
To perform a DFS, you start with a given source vertex and attempt to explore a branch as far as possible until you reach the end. At this point, you'd backtrack (move a step back) and explore the next available branch until you find what you're looking for or until you've visited all the vertices.



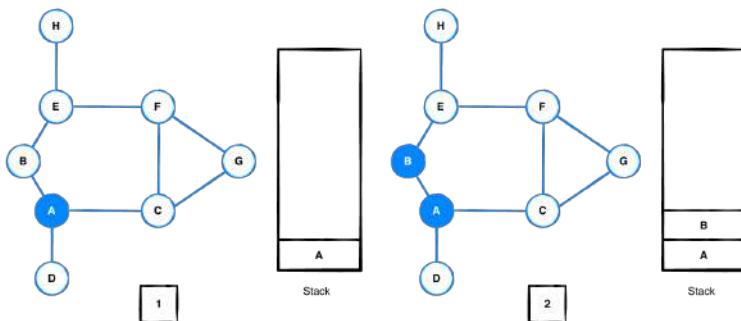
DFS example

The example graph below is the same as the previous chapter.

Using the same graph helps you to see the difference between BFS and DFS.

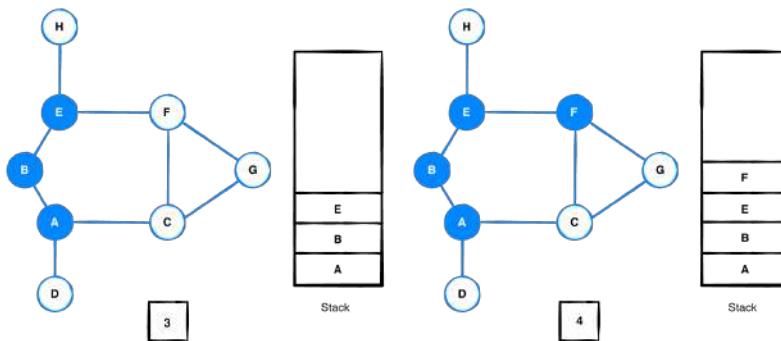


You'll use a stack to keep track of the levels you move through. The stack's LIFO approach helps with backtracking. Every push on the stack means that you move one level deeper. When you reach a dead end, you can pop to return to a previous level.



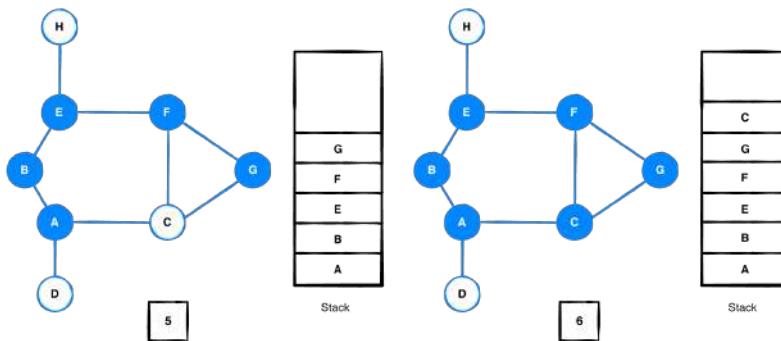
1. As in the previous chapter, you choose A as a starting vertex and add it to the stack.
2. As long as the stack is not empty, you visit the top vertex on the stack and push the first neighboring vertex that you haven't yet visited. In this case, you visit A and push B.

Recall from the previous chapter that the order in which you add edges influences the result of a search. In this case, the first edge added to A was an edge to B, so B is pushed first.



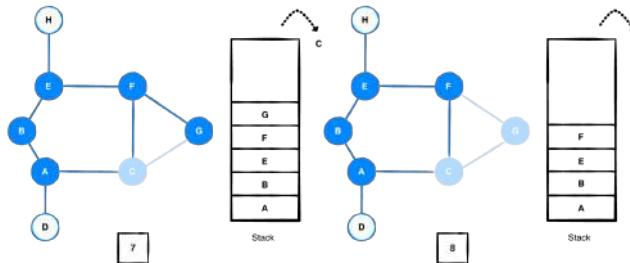
3. You visit B and push E because you already visited A.
4. You visit E and push F.

Note that, every time you push on the stack, you advance farther down a branch. Instead of visiting every adjacent vertex, you continue down a path until you reach the end and then backtrack.



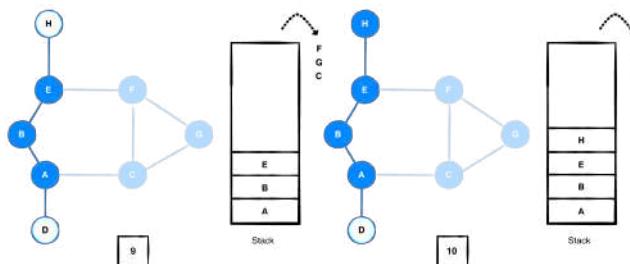
5. You visit F and push G.

6. You visit G and push C.



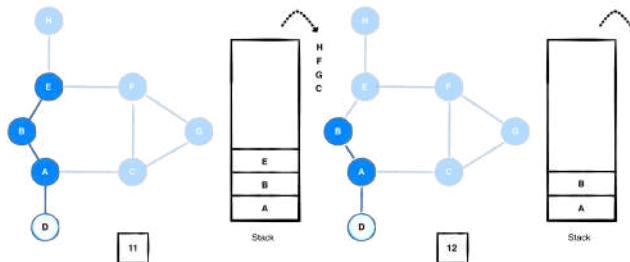
7. The next vertex to visit is C. It has neighbors [A, F, G], but you already visited these. You reached a dead end, so it's time to backtrack by popping C off the stack.

8. This brings you back to G. It has neighbors [F, C], but you also already visited these. Another dead end, pop G.

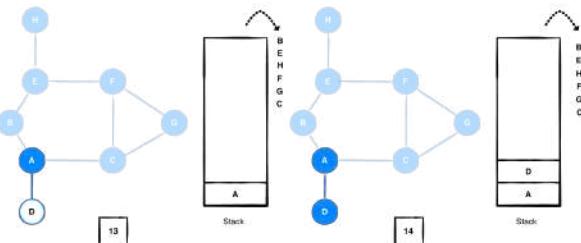


9. F also has no unvisited neighbors remaining, so pop F.

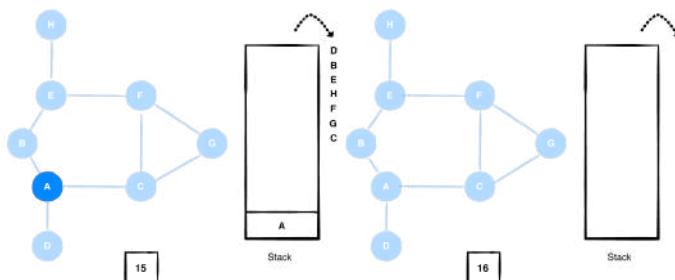
10. Now, you're back at E. Its neighbor H is still unvisited, so you push H on the stack.



11. Visiting H results in another dead end, so pop H.
12. E also doesn't have any available neighbors, so pop it.

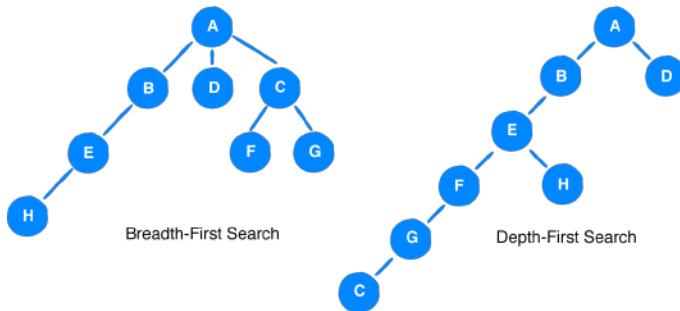


13. The same is true for B, so pop B.
14. This brings you back to A, whose neighbor D still needs a visit, so you push D on the stack.



15. Visiting D results in another dead end, so pop D.
16. You're back at A, but this time, there are no available neighbors to push, so you pop A. The stack is now empty, and the DFS is complete.

When exploring the vertices, you can construct a tree-like structure, showing the branches you've visited. You can see how deep DFS went when compared to BFS.



Implementation

Open the starter project for this chapter. This project contains an implementation of a graph, as well as a stack that you'll use to implement DFS.

Look at **Main.kt**, and you'll see a pre-built sample graph. This is the graph you'll be working with.

To implement DFS, add the following inside Graph:

```
fun depthFirstSearch(source: Vertex<T>): ArrayList<Vertex<T>> {  
    val stack = StackImpl<Vertex<T>>()  
    val visited = arrayListOf<Vertex<T>>()  
    val pushed = mutableSetOf<Vertex<T>>()  
  
    stack.push(source)  
    pushed.add(source)  
    visited.add(source)  
  
    // more to come ...  
  
    return visited  
}
```

With this code you define `depthFirstSearch()`, a new method that takes in a starting vertex and returns a list of vertices in the order they were visited. It uses three data structures:

1. **stack:** Used to store your path through the graph.

2. **pushed**: Remembers which vertices were already pushed so that you don't visit the same vertex twice. It's a `MutableSet` to ensure fast $O(1)$ lookup.
3. **visited**: A list that stores the order in which the vertices were visited.

In the first step you insert the vertex passed as parameter to the three data structures. You do this because this is the first to be visited and it's the starting point in order to navigate the neighbors.

Next, complete the method by replacing the comment with:

```
outer@ while (true) {  
    if (stack.isEmpty) break  
  
    val vertex = stack.peek()!! // 1  
    val neighbors = edges(vertex) // 2  
  
    if (neighbors.isEmpty()) { // 3  
        stack.pop()  
        continue  
    }  
  
    for (i in 0 until neighbors.size) { // 4  
        val destination = neighbors[i].destination  
        if (destination !in pushed) {  
            stack.push(destination)  
            pushed.add(destination)  
            visited.add(destination)  
            continue@outer // 5  
        }  
    }  
    stack.pop() // 6  
}
```

Here's what's going on:

1. You continue to check the top of the stack for a vertex until the stack is empty. You've labeled this loop `outer` so that you have a way to continue to the next vertex, even within nested loops.
2. You find all the neighboring edges for the current vertex.
3. If there are no edges, you pop the vertex off the stack and continue to the next one.
4. Here, you loop through every edge connected to the current vertex and check to see if the neighboring vertex has been seen. If not, you push it onto the stack and add it to the `visited` list.

It may seem a bit premature to mark this vertex as visited — you haven't popped it yet — but since vertices are visited in the order in which they are added to the stack, it results in the correct order.

5. Now that you've found a neighbor to visit, you continue the outer loop and move to the newly pushed neighbor.
6. If the current vertex did not have any unvisited neighbors, you know that you reached a dead end and can pop it off the stack.

Once the stack is empty, the DFS algorithm is complete. All you have to do is return the visited vertices in the order you visited them.

To test your code, add the following to `main()`:

```
val vertices = graph.depthFirstSearch(a)
vertices.forEach {
    println(it.data)
}
```

If you run the `main()` in the **Main.kt** file, you'll see the following output for the order of the visited nodes using a DFS:

```
A
B
E
H
F
C
G
D
```

Performance

DFS visits every vertex at least once. This has a time complexity of $O(V)$.

When traversing a graph in DFS, you have to check all neighboring vertices to find one that's available to visit. The time complexity of this is $O(E)$ because in the worst case, you have to visit every edge in the graph.

Overall, the time complexity for depth-first search is $O(V + E)$.

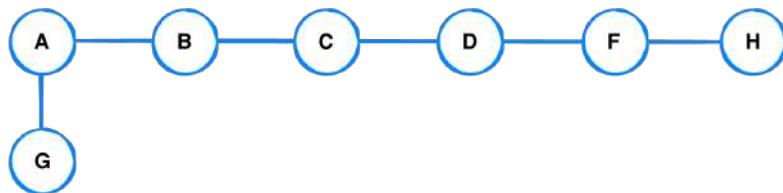
The space complexity of depth-first search is $O(V)$ because you have to store vertices in three separate data structures: `stack`, `pushed` and `visited`.



Challenges

Challenge 1: Depth First Search

For each of the following two examples, which traversal (depth-first or breadth-first) is better for discovering if a path exists between the two nodes? Explain why.



- Path from A to F.
- Path from A to G.

Solution 1

- Path from A to F: Use depth-first, because the path you're looking for is deeper in the graph.
- Path from A to G: Use breadth-first, because the path you're looking for is near the root.

Challenge 2: Depth First Search

In this chapter, you went over an iterative implementation of depth-first search. Write a recursive implementation.

Solution 2

Let's look at how you can implement DFS recursively.

```
fun depthFirstSearchRecursive(start: Vertex<T>):  
    ArrayList<Vertex<T>> {  
        val visited = arrayListOf<Vertex<T>>() // 1  
        val pushed = mutableSetOf<Vertex<T>>() // 2  
  
        depthFirstSearch(start, visited, pushed) // 3  
  
        return visited  
    }
```

Here's what's happening:

1. `visited` keeps track of the vertices visited in order.
2. `pushed` keeps tracks of which vertices have been visited.
3. Perform depth-first search recursively by calling a helper function.

The helper function looks like this:

```
fun depthFirstSearch(  
    source: Vertex<T>,  
    visited: ArrayList<Vertex<T>>,  
    pushed: MutableSet<Vertex<T>>  
) {  
    pushed.add(source) // 1  
    visited.add(source)  
  
    val neighbors = edges(source)  
    neighbors.forEach { // 2  
        if (it.destination !in pushed) {  
            depthFirstSearch(it.destination, visited, pushed) // 3  
        }  
    }  
}
```

Here's how it works:

1. Insert the source vertex into the queue, and mark it as visited.
2. For every neighboring edge...
3. As long as the adjacent vertex has not been visited yet, continue to dive deeper down the branch recursively.

Overall, the time complexity for depth-first search is $O(V + E)$.

Challenge 3: Depth First Search Challenge

Add a method to Graph to detect if a **directed** graph has a cycle.

Solution 3

A graph is said to have a cycle when there's a path of edges and vertices leading back to the same source.

```
fun hasCycle(source: Vertex<T>): Boolean {
    val pushed = arrayListOf<Vertex<T>>() // 1
    return hasCycle(source, pushed) // 2
}
```

Here's how it works:

1. `pushed` is used to keep track of all the vertices visited.
2. Recursively check to see if there's a cycle in the graph by calling a helper function.

The helper function looks like this:

```
fun hasCycle(source: Vertex<T>, pushed: MutableSet<Vertex<T>>):
Boolean {
    pushed.add(source) // 1

    val neighbors = edges(source) // 2
    neighbors.forEach {
        if (it.destination !in pushed && hasCycle(it.destination,
pushed)) { // 3
            return true
        } else if (it.destination in pushed) { // 4
            return true
        }
    }

    pushed.remove(source) // 5
    return false // 6
}
```

And it works like this:

1. To initiate the algorithm, first, insert the source vertex.
2. For every neighboring edge...
3. If the adjacent vertex has not been visited before, recursively dive deeper down a branch to check for a cycle.

4. If the adjacent vertex has been visited before, you've found a cycle.
5. Remove the source vertex so that you can continue to find other paths with a potential cycle.
6. No cycle has been found.

You're essentially performing a depth-first graph traversal by recursively diving down one path until you find a cycle. You're backtracking by popping off the stack to find another path. The time-complexity is $O(V + E)$.

Key points

- Depth-first search (DFS) is another algorithm to traverse or search a graph.
- DFS explores a branch as far as possible until it reaches the end.
- Leverage a stack data structure to keep track of how deep you are in the graph.
Only pop off the stack when you reach a dead end.

Chapter 22: Dijkstra's Algorithm

By Irina Galata

Have you ever used the Google or Apple Maps app to find the shortest distance or fastest time from one place to another? **Dijkstra's algorithm** is particularly useful in GPS networks to help find the shortest path between two places.

Dijkstra's algorithm is a greedy algorithm. A **greedy** algorithm constructs a solution step-by-step, and it picks the most optimal path at every step. In particular, Dijkstra's algorithm finds the shortest paths between vertices in either directed or undirected graphs. Given a vertex in a graph, the algorithm will find all shortest paths from the starting vertex.

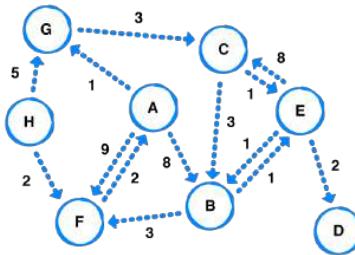
Some other applications of Dijkstra's algorithm include:

1. Communicable disease transmission: Discover where biological diseases are spreading the fastest.
2. Telephone networks: Routing calls to highest-bandwidth paths available in the network.
3. Mapping: Finding the shortest and fastest paths for travelers.



Example

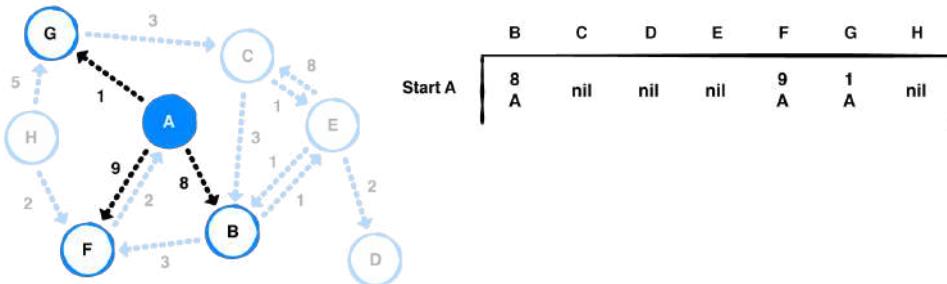
All the graphs you've looked at thus far have been undirected graphs. Let's change it up a little and work with a directed graph! Imagine the directed graph below represents a GPS network:



The vertices represent physical locations, and the edges between the vertices represent one way paths of a given cost between locations.

First pass

In Dijkstra's algorithm, you first choose a **starting vertex**, since the algorithm needs a starting point to find a path to the rest of the nodes in the graph. Assume the starting vertex you pick is **vertex A**.



From **vertex A**, look at all outgoing edges. In this case, you've three edges:

- A to B, has a cost of **8**.
- A to F, has a cost of **9**.
- A to G, has a cost of **1**.

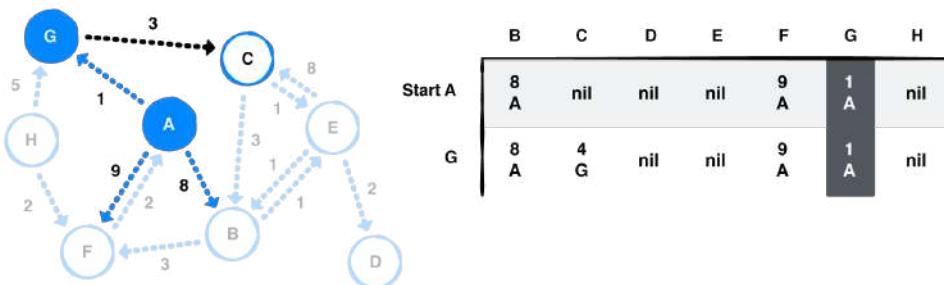
The remainder of the vertices will be marked as `nil`, since there is no direct path to them from **A**.

As you work through this example, the table on the right of the graph will represent a history or record of Dijkstra's algorithm at each stage. Each pass of the algorithm will add a row to the table. The last row in the table will be the final output of the algorithm.

Second pass

	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil

In the next cycle, Dijkstra's algorithm looks at the **lowest-cost** path you've thus far. **A** to **G** has the smallest cost of **1**, and is also the shortest path to get to **G**. This is marked with a dark fill in the output table.



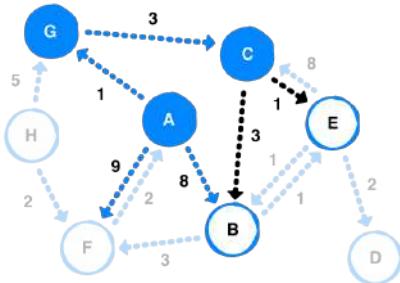
Now, from the lowest-cost path, **vertex G**, look at all the outgoing edges. There's only one edge from **G** to **C**, and its total cost is **4**. This is because the cost from **A** to **G** to **C** is **1 + 3 = 4**.

Every value in the output table has two parts: the total cost to reach that vertex, and the last neighbor on the path to that vertex. For example, the value **4 G** in the column for vertex **C** means that the cost to reach **C** is **4**, and the path to **C** goes through **G**. A value of `nil` indicates that no path has been discovered to that vertex.

Third pass

	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil
G	8 A	4 G	nil	nil	9 A	1 A	nil

In the next cycle, you look at the next-lowest cost. According to the table, the path to C has the smallest cost, so the search will continue from C. You fill column C because you've found the shortest path to get to C.



	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil
G	8 A	4 G	nil	nil	9 A	1 A	nil
C	7 C	4 G	nil	5 C	9 A	1 A	nil

Look at all of C's outgoing edges:

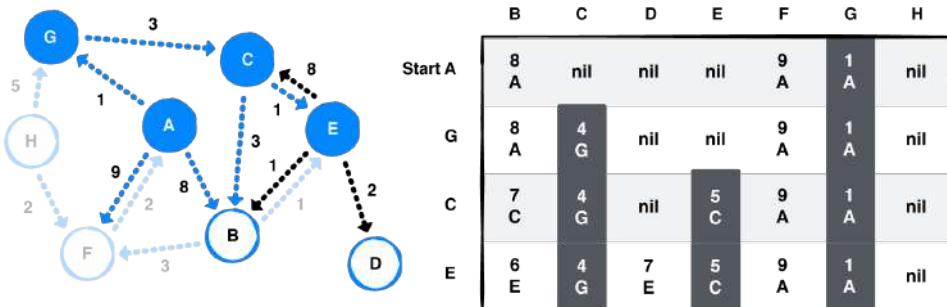
- C to E has a total cost of $4 + 1 = 5$.
- C to B has a total cost of $4 + 3 = 7$.

You've found a lower-cost path to B, so you replace the previous value for B.

Fourth pass

	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil
G	8 A	4 G	nil	nil	9 A	1 A	nil
C	7 C	4 G	nil	5 C	9 A	1 A	nil

Now, in the next cycle, ask yourself what is the next-lowest cost path? According to the table, **C** to **E** has the smallest total cost of 5, so the search will continue from **E**.



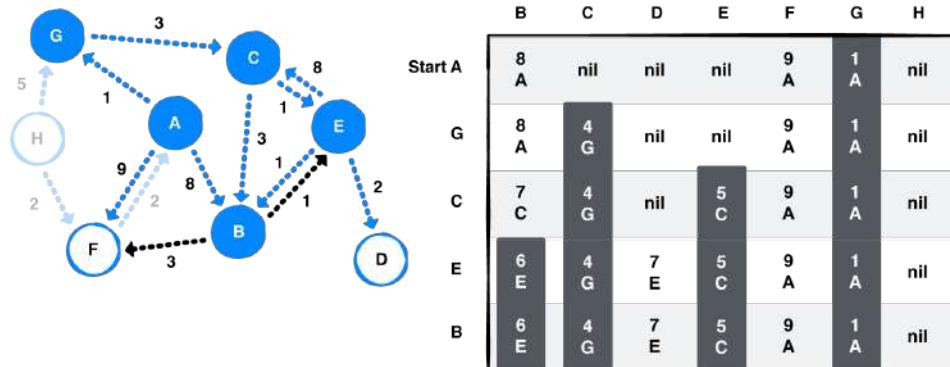
You fill column **E** because you've found the shortest path. Vertex **E** has the following outgoing edges:

- **E** to **C** has a total cost of $5 + 8 = 13$. Since you've found the shortest path to **C** already, disregard this path.
- **E** to **D** has a total cost of $5 + 2 = 7$.
- **E** to **B** has a total cost of $5 + 1 = 6$. According to the table, the current shortest path to **B** has a total cost of 7. You update the shortest path from **E** to **B**, since it has a smaller cost of 6.

Fifth pass

	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil
G	8 A	4 G	nil	nil	9 A	1 A	nil
C	7 C	4 G	nil	5 C	9 A	1 A	nil
E	6 E	4 G	7 E	5 C	9 A	1 A	nil

Next, you continue the search from **B**.



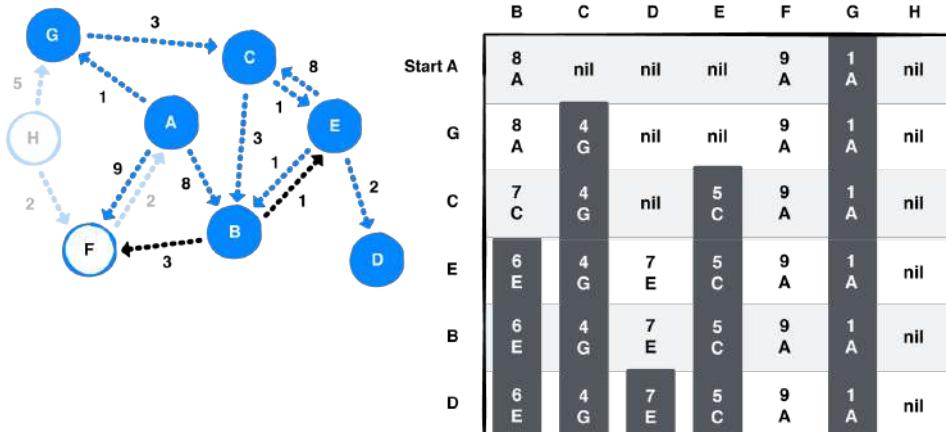
B has these outgoing edges:

- **B** to **E** has a total cost of $6 + 1 = 7$, but you've already found the shortest path to **E**, so disregard this path.
- **B** to **F** has a total cost of $6 + 3 = 9$. From the table, you can tell that the current path to **F** from **A** also has a cost of **9**. You can disregard this path since it isn't any shorter.

Sixth pass

	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil
G	8 A	4 G	nil	nil	9 A	1 A	nil
C	7 C	4 G	nil	5 C	9 A	1 A	nil
E	6 E	4 G	7 E	5 C	9 A	1 A	nil
B	6 E	4 G	7 E	5 C	9 A	1 A	nil

In the next cycle, you continue the search from **D**.

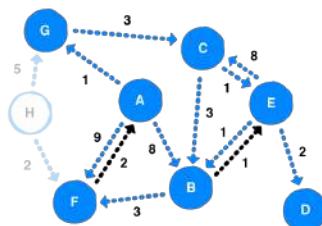


However **D** has no outgoing edges, so it's a dead end. You simply record that you've found the shortest path to **D** and move on.

Seventh pass

	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil
G	8 A	4 G	nil	nil	9 A	1 A	nil
C	7 C	4 G	nil	5 C	9 A	1 A	nil
E	6 E	4 G	7 E	5 C	9 A	1 A	nil
B	6 E	4 G	7 E	5 C	9 A	1 A	nil
D	6 E	4 G	7 E	5 C	9 A	1 A	nil

F is next up.

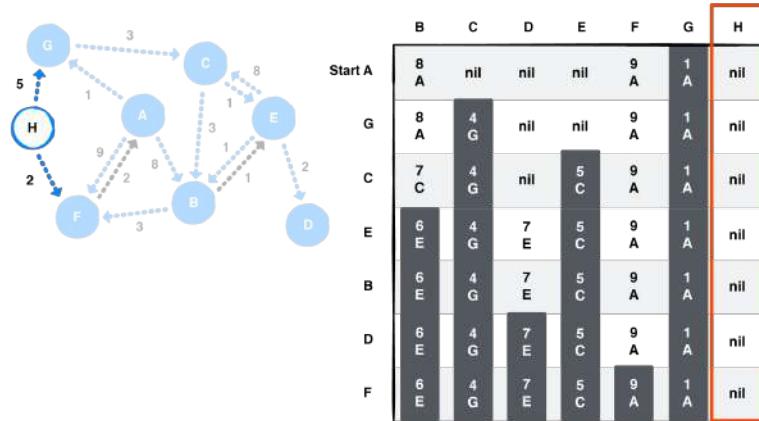


	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil
G	8 A	4 G	nil	nil	9 A	1 A	nil
C	7 C	4 G	nil	5 C	9 A	1 A	nil
E	6 E	4 G	7 E	5 C	9 A	1 A	nil
B	6 E	4 G	7 E	5 C	9 A	1 A	nil
D	6 E	4 G	7 E	5 C	9 A	1 A	nil
F	6 E	4 G	7 E	5 C	9 A	1 A	nil

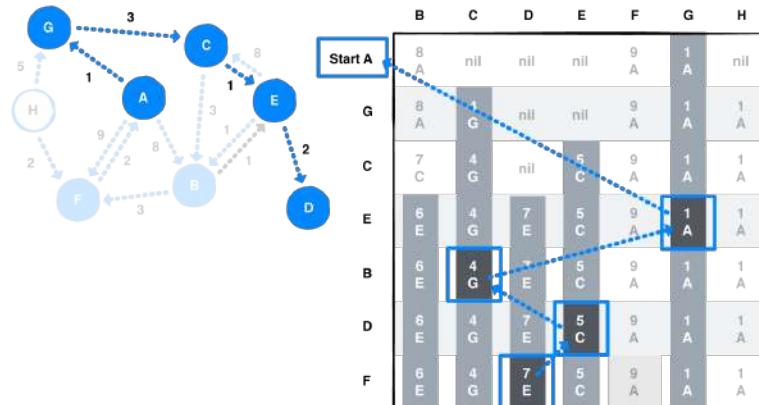
F has one outgoing edge to A with a total cost of **9 + 2 = 11**. You can disregard this edge since A is the starting vertex.

Eighth pass

You've covered every vertex except for **H**. **H** has two outgoing edges to **G** and **F**. However, there's no path from **A** to **H**. This is why the whole column for **H** is **nil**.



This completes Dijkstra's algorithm, since all the vertices have been visited!



You can now check the final row for the shortest paths and their costs. For example, the output tells you the cost to get to **D** is 7. To find the path, you simply backtrack. Each column records the previous vertex the current vertex is connected to. You should get from **D** to **E** to **C** to **G** and finally back to **A**. Let's look at how you can build this in code.

Implementation

Open up the starter playground for this chapter. This playground comes with an adjacency list graph and a priority queue, which you'll use to implement Dijkstra's algorithm.

The priority queue is used to store vertices that have not been visited. It's a min-priority queue so that, every time you dequeue a vertex, it gives you vertex with the current tentative shortest path.

Create a new file named **Dijkstra.kt** and add the following inside the file:

```
class Dijkstra<T>(private val graph: AdjacencyList<T>){  
    // to be continued ...  
}
```

Next, add the following class below the **Dijkstra** class:

```
class Visit<T>(val type: VisitType, val edge: Edge<T>? = null)  
  
enum class VisitType {  
    START, // 1  
    EDGE // 2  
}
```

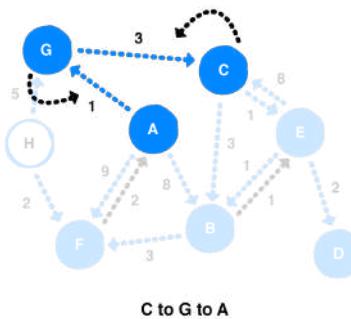
Here, you defined an enum named **Visit**. This keeps track of two states:

1. The vertex is the starting vertex.
2. The vertex has an associated edge that leads to a path back to the starting vertex.

Helper methods

Before building Dijkstra, let's create some helper methods that will help create the algorithm.

Tracing back to the start



You need a mechanism to keep track of the total weight from the current vertex back to the start vertex. To do this, you'll keep track of a map named paths that stores a Visit state for every vertex.

Add the following method to class Dijkstra:

```
private fun route(destination: Vertex<T>, paths: HashMap<Vertex<T>, Visit<T>>): ArrayList<Edge<T>> {
    var vertex = destination // 1
    val path = arrayListOf<Edge<T>>() // 2

    loop@ while (true) {
        val visit = paths[vertex] ?: break

        when(visit.type) {
            VisitType.EDGE -> visit.edge?.let { // 3
                path.add(it) // 4
                vertex = it.source // 5
            }
            VisitType.START -> break@loop // 6
        }
    }

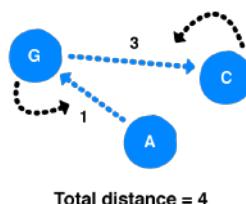
    return path
}
```



This method takes in the destination vertex along with a dictionary of existing paths, and it constructs a path that leads to the destination vertex. Going over the code:

1. Start at the destination vertex.
2. Create a list of edges to store the path.
3. As long as you've not reached the start case, continue to extract the next edge.
4. Add this edge to the path.
5. Set the current vertex to the edge's source vertex. This moves you closer to the start vertex.
6. Once the while loop reaches the start case, you've completed the path and return it.

Calculating total distance



Once you've the ability to construct a path from the **destination** back to the **start** vertex, you need a way to calculate the total weight for that path.

Add the following method to class `Dijkstra`:

```
private fun distance(destination: Vertex<T>, paths: HashMap<Vertex<T>, Visit<T>>): Double {
    val path = route(destination, paths) // 1
    return path.sumByDouble { it.weight ?: 0.0 }
}
```

This method takes in the destination vertex and a dictionary of existing paths, and it returns the total weight. Going over the code:

1. Construct the path to the destination vertex.
2. `sumByDouble` sums the weights of all the edges.

Now that you've established your helper methods, let's implement Dijkstra's algorithm.

Generating the shortest paths

After the `distance` method, add the following:

```
fun shortestPath(start: Vertex<T>): HashMap<Vertex<T>, Visit<T>>
{
    val paths: HashMap<Vertex<T>, Visit<T>> = HashMap()
    paths[start] = Visit(VisitType.START) // 1

    // 2
    val distanceComparator = Comparator<Vertex<T>>({ first, second
        -> (distance(second, paths) - distance(first, paths)).toInt()
    })
    // 3
    val priorityQueue =
        ComparatorPriorityQueueImpl(distanceComparator)
    // 4
    priorityQueue.enqueue(start)

    // to be continued ...
}
```

This method takes in a `start` vertex and returns a dictionary of all the paths. Within the method you:

1. Define `paths` and initialize it with the `start` vertex.
2. Create a `Comparator` which uses distances between vertices for sorting
3. Use the previous `Comparator` and create a min-priority queue to store the vertices that must be visited.
4. Enqueue the `start` vertex as the first vertex to visit.

Complete your implementation of `shortestPath` with:

```
while (true) {
    val vertex = priorityQueue.dequeue() ?: break // 1
    val edges = graph.edges(vertex) // 2

    edges.forEach {
        val weight = it.weight ?: return@forEach // 3

        if (paths[it.destination] == null
            || distance(vertex, paths) + weight <
```

```
        distance(it.destination, paths) { //4
            paths[it.destination] = Visit(VisitType.EDGE, it)
            priorityQueue.enqueue(it.destination)
        }
    }

return paths
```

Going over the code:

1. You continue Dijkstra's algorithm to find the shortest paths until you've visited all the vertices have been visited. This happens once the priority queue is empty.
2. For the current vertex, you go through all its neighboring edges.
3. You make sure the edge has a weight. If not, you move on to the next edge.
4. If the destination vertex has not been visited before or you've found a cheaper path, you update the path and add the neighboring vertex to the priority queue.

Once all the vertices have been visited, and the priority queue is empty, you return the map of shortest paths back to the start vertex.

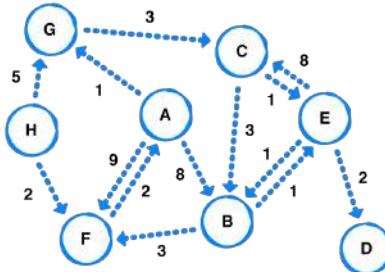
Finding a specific path

Add the following method to class `Dijkstra`:

```
fun shortestPath(destination: Vertex<T>, paths:
    HashMap<Vertex<T>,
    Visit<T>>): ArrayList<Edge<T>> {
    return route(destination, paths)
}
```

This simply takes the `destination` vertex and the map of shortest and returns the path to the `destination` vertex.

Trying out your code



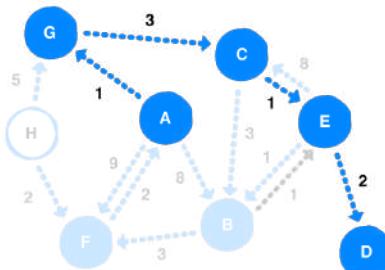
Navigate to the `main()` function, and you'll notice the graph above has been already constructed using an adjacency list. Time to see Dijkstra's algorithm in action.

Add the following code to the `main()` function:

```
val dijkstra = Dijkstra(graph)
val pathsFromA = dijkstra.shortestPath(a) // 1
val path = dijkstra.shortestPath(d, pathsFromA) // 2
path.forEach { // 3
    println("${it.source.data} --|${it.weight ?: 0.0}|--> + " +
        "${it.destination.data}")
}
```

Here, you simply create an instance of `Dijkstra` by passing in the graph network and do the following:

1. Calculate the shortest paths to all the vertices from the start vertex **A**.
2. Get the shortest path to **D**.
3. Print this path.



This outputs:

```
E --|2.0|--> D
C --|1.0|--> E
G --|3.0|--> C
A --|1.0|--> G
```

Performance

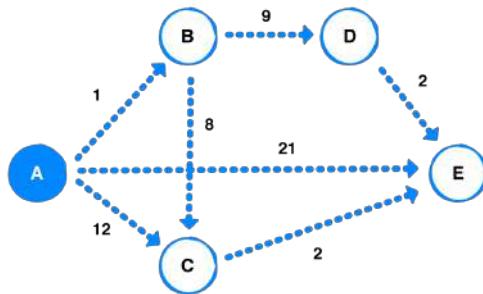
In Dijkstra's algorithm, you constructed your graph using an adjacency list. You used a min-priority queue to store vertices and extract the vertex with the minimum path. This has an overall performance of $O(\log V)$. This's because the heap operations of extracting the minimum element or inserting an element both take $O(\log V)$.

If you recall from the breadth-first search chapter, it takes $O(V + E)$ to traverse all the vertices and edges. Dijkstra's algorithm is somewhat similar to breadth-first search, because you have to explore all neighboring edges. This time, instead of going down to the next level, you use a min-priority queue to select a single vertex with the shortest distance to traverse down. That means it is $O(1 + E)$ or simply $O(E)$. So, combining the traversal with operations on the min-priority queue, it takes $O(E \log V)$ to perform Dijkstra's algorithm.

Challenges

Challenge 1: Running Dijkstra's

Given the following graph, step through Dijkstra's algorithm to produce the shortest path to every other vertex starting from **vertex A**. Provide the final table of the paths as shown in the previous chapter.



Solution 1

	B	C	D	E
Start	A	12 A	null	21 A
B	A	9 B	10 B	21 A
C	A	9 B	10 B	11 C
D	A	9 B	10 B	11 C
E	A	9 B	10 B	11 C

- Path to B: A - (1) - B
- Path to C: A - (1) - B - (8) - C
- Path to D: A - (1) - B - (9) - D
- Path to E: A - (1) - B - (8) - C - (2) - E

Challenge 2: Collect Dijkstra's data

Add a method to class `Dijkstra` that returns a dictionary of all the shortest paths to all vertices given a starting vertex. Here's the method signature to get you started:

```
fun getAllShortestPath(source: Vertex<T>): HashMap<Vertex<T>,  
    Visit<T>> {  
    val paths = HashMap<Vertex<T>, Visit<T>>()  
  
    // Implement solution here ...  
  
    return paths  
}
```

Solution 2

This function is part of `Dijkstra.kt`. To get the shortest paths from the source vertex to every other vertex in the graph, do the following:

```
fun getAllShortestPath(source: Vertex<T>): HashMap<Vertex<T>,  
    ArrayList<Edge<T>>> {  
    val paths = HashMap<Vertex<T>, ArrayList<Edge<T>>>() // 1  
    val pathsFromSource = shortestPath(source) // 2  
  
    graph.vertices.forEach { // 3  
        val path = shortestPath(it, pathsFromSource)  
        paths[it] = path  
    }  
  
    return paths // 4  
}
```

1. The map stores the path to every vertex from the source vertex.
2. Perform Dijkstra's algorithm to find all the paths from the source vertex.
3. For every vertex in the graph, generate the list of edges between the source vertex to every vertex in the graph.
4. Return the map of paths.

Key points

- Dijkstra's algorithm finds a path to the rest of the nodes given a starting vertex.
- This algorithm is useful for finding the shortest paths between different endpoints.
- Visit state is used to track the edges back to the start vertex.
- The priority queue data structure helps to always return the vertex with the shortest path.
- Hence, it is a greedy algorithm!



Chapter 23: Prim's Algorithm

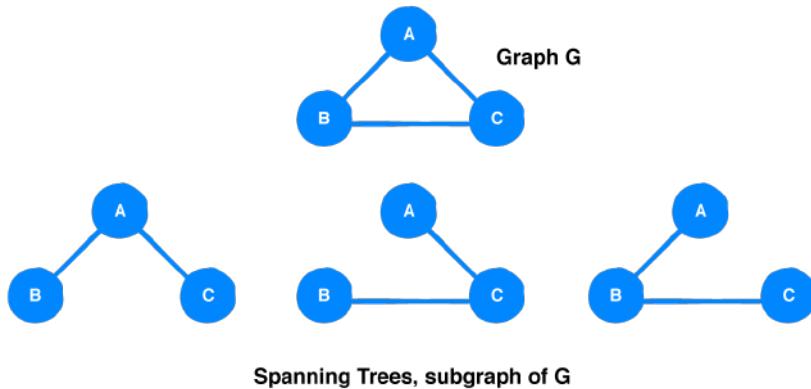
By Irina Galata

In previous chapters, you've looked at depth-first and breadth-first search algorithms. These algorithms form **spanning trees**.

A **spanning tree** is a subgraph of an undirected graph, containing all of the graph's vertices, connected with the fewest number of edges. A spanning tree cannot contain a cycle and cannot be disconnected.



Here's an example of some spanning trees:

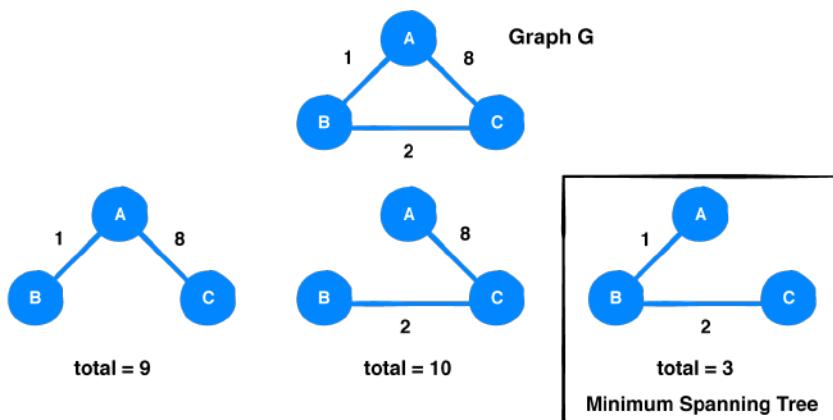


From this undirected graph that forms a triangle, you can generate three different spanning trees in which you require only two edges to connect all vertices.

In this chapter, you will look at **Prim's algorithm**, a greedy algorithm used to construct a **minimum spanning tree**. A **greedy** algorithm constructs a solution step-by-step and picks the most optimal path at every step.

A minimum spanning tree is a spanning tree with weighted edges in which the total weight of all edges is minimized. For example, you might want to find the cheapest way to lay out a network of water pipes.

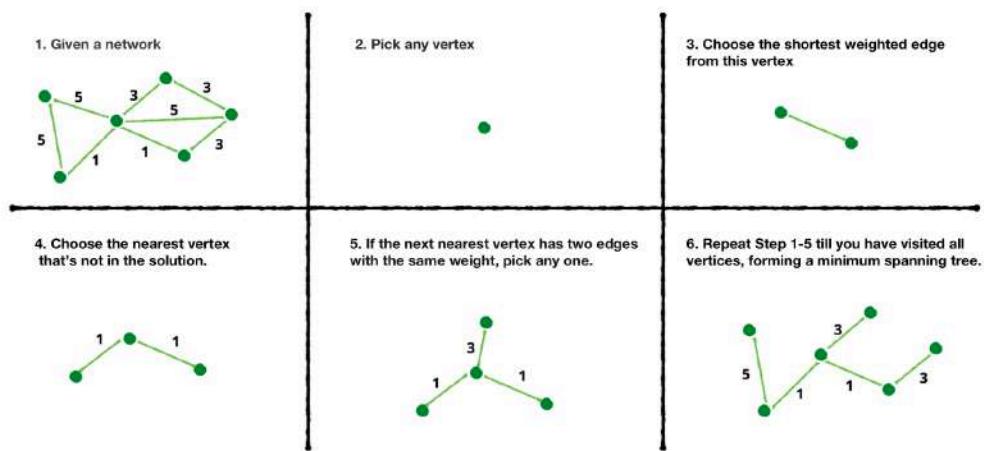
Here's an example of a minimum spanning tree for a weighted undirected graph:



Notice that only the third subgraph forms a minimum spanning tree, since it has the minimum total cost of 3.

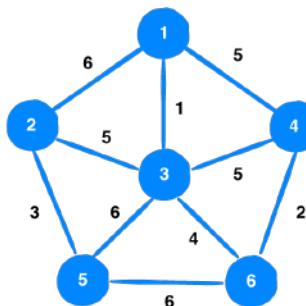
Prim's algorithm creates a minimum spanning tree by choosing edges one at a time. It's greedy because, every time you pick an edge, you pick the smallest weighted edge that connects a pair of vertices.

There are six steps to finding a minimum spanning tree with Prim's algorithm:

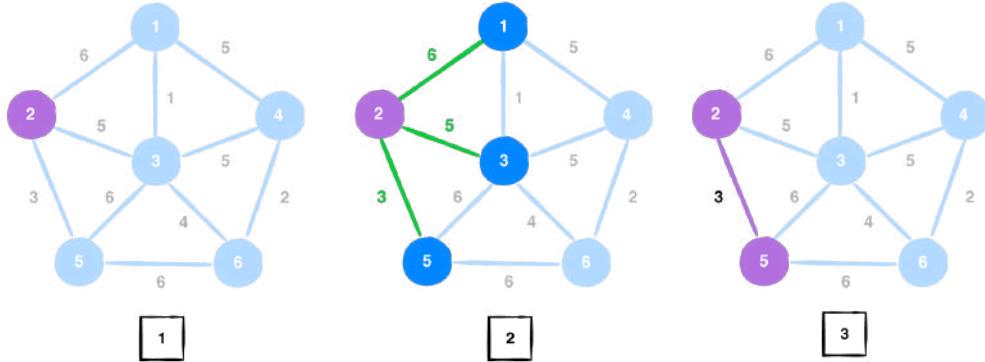


Example

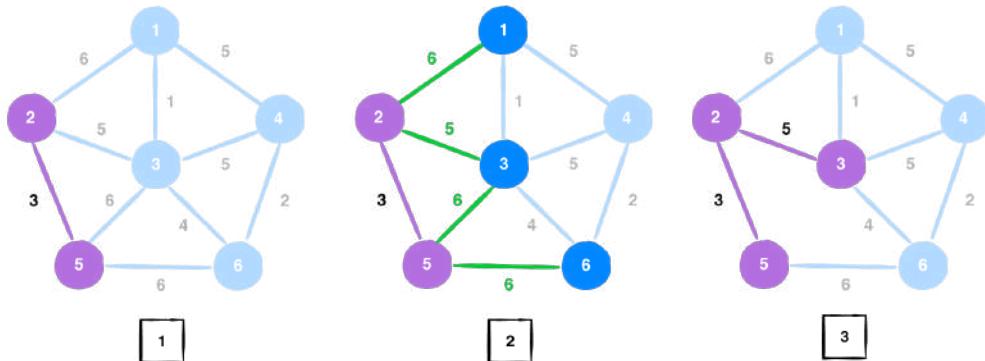
Imagine the graph below represents a network of airports. The vertices are the airports, and the edges between them represent the cost of fuel to fly an airplane from one airport to the next.



Let's start working through the example:

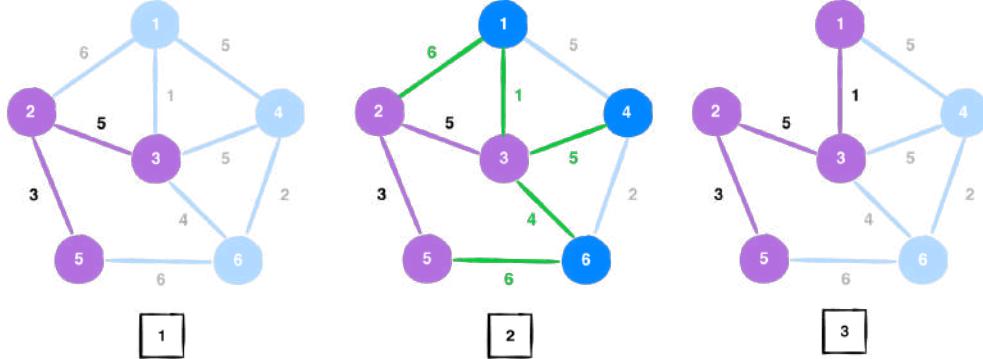


1. Choose any vertex in the graph. Let's assume you chose **vertex 2**.
2. This vertex has edges with weights [6, 5, 3]. A greedy algorithm chooses the smallest-weighted edge.
3. Choose the edge that has a weight of 3 and is connected to **vertex 5**.

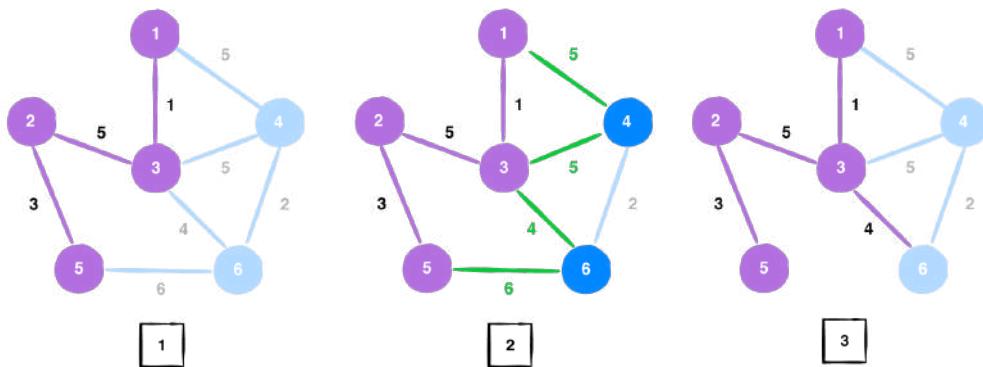


1. The explored vertices are {2, 5}.
2. Choose the next shortest edge from the explored vertices. The edges are [6, 5, 6, 6]. You choose the edge with weight 5, which is connected to **vertex 3**.

- Notice that the edge between **vertex 5** and **vertex 3** can be removed since both are already part of the spanning tree.

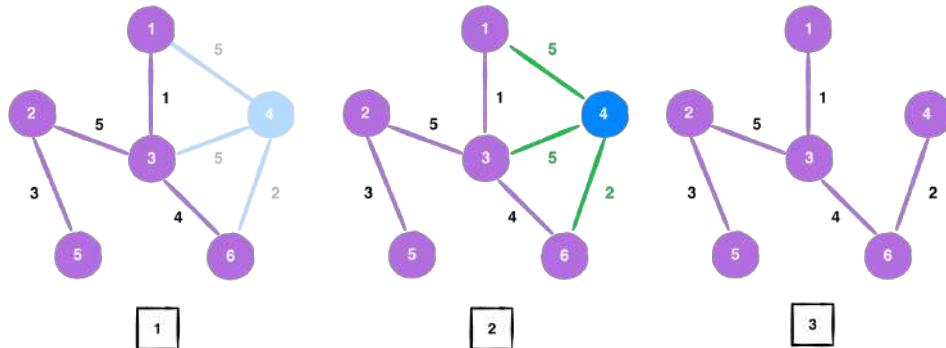


- The explored vertices are **{2, 3, 5}**.
- The next potential edges are **[6, 1, 5, 4, 6]**. You choose the edge with weight **1**, which is connected to **vertex 1**.
- The edge between **vertex 2** and **vertex 1** can be removed.



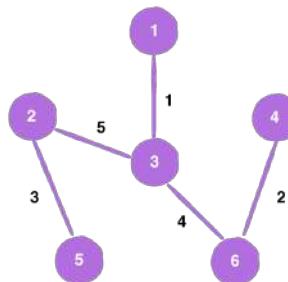
- The explored vertices are **{2, 3, 5, 1}**.
- Choose the next shortest edge from the explored vertices. The edges are **[5, 5, 4, 6]**. You choose the edge with weight **4**, which is connected to **vertex 6**.

3. The edge between **vertex 5** and **vertex 6** can be removed.



1. The explored vertices are $\{2, 5, 3, 1, 6\}$.
2. Choose the next shortest edge from the explored vertices. The edges are $[5, 5, 2]$. You choose the edge with weight 2, which is connected to **vertex 4**.
3. The edges $[5, 5]$ connected to **vertex 4** from **vertex 1** and **vertex 3** can be removed.

Note: If all edges have the same weight, you can pick any one of them.



This is the minimum spanning tree from our example produced from Prim's algorithm.

Next, let's see how to build this in code.

Implementation

Open up the starter project for this chapter. This project comes with an adjacency list graph and a priority queue, which you will use to implement Prim's algorithm.

The priority queue is used to store the edges of the explored vertices. It's a min-priority queue so that every time you dequeue an edge, it gives you the edge with the smallest weight.

Start by defining an object called `Prim`. Create it in the `Prim.kt` file:

```
object Prim
```

Helper methods

Before building the algorithm, you'll create some helper methods to keep you organized and consolidate duplicate code.

Copying a graph

To create a minimum spanning tree, you must include all vertices from the original graph. Open up `AdjacencyList.kt` and add the following to class `AdjacencyList`:

```
val vertices: Set<Vertex<T>>
    get() = adjacencies.keys
```

This property returns the set of vertices that your `AdjacencyList` is currently storing.

Next, add the following method:

```
fun copyVertices(graph: AdjacencyList<T>) {
    graph.vertices.forEach {
        adjacencies[it] = arrayListOf()
    }
}
```

This copies all of a graph's vertices into a new graph.



Finding edges

Besides copying the graph's vertices, you also need to find and store the edges of every vertex you explore. Open up **Prim.kt** and add the following to **Prim**:

```
private fun <T> addAvailableEdges(
    vertex: Vertex<T>,
    graph: Graph<T>,
    visited: Set<Vertex<T>>,
    priorityQueue: AbstractPriorityQueue<Edge<T>>
) {
    graph.edges(vertex).forEach { edge -> // 1
        if (edge.destination !in visited) { // 2
            priorityQueue.enqueue(edge) // 3
        }
    }
}
```

This method takes in four parameters:

1. The current `vertex`.
2. The `graph`, wherein the current `vertex` is stored.
3. The vertices that have already been visited.
4. The priority queue to add all potential edges.

Within the function, you do the following:

1. Look at every edge adjacent to the current `vertex`.
2. Check to see if the `destination` vertex has already been visited.
3. If it has not been visited, you add the edge to the priority queue.

Now that we have established our helper methods, let's implement Prim's algorithm.

Producing a minimum spanning tree

Add the following method to `Prim`:

```
fun <T> produceMinimumSpanningTree(
    graph: AdjacencyList<T>
): Pair<Double, AdjacencyList<T>> { // 1
    var cost = 0.0 // 2
    val mst = AdjacencyList<T>() // 3
    val visited = mutableSetOf<Vertex<T>>() // 4
    val comparator = Comparator<Edge<T>> { first, second -> // 5
        val firstWeight = first.weight ?: 0.0
        val secondWeight = second.weight ?: 0.0
        (secondWeight - firstWeight).roundToInt()
    }
    val priorityQueue = ComparatorPriorityQueueImpl(comparator) // 6
    // to be continued
}
```

Here's what you have so far:

1. `produceMinimumSpanningTree` takes an undirected graph and returns a minimum spanning tree and its cost.
2. `cost` keeps track of the total weight of the edges in the minimum spanning tree.
3. This is a graph that will become your minimum spanning tree.
4. `visited` stores all vertices that have already been visited.
5. You create the `Comparator<Edge<T>>` to use for the priority queue.
6. This is a min-priority queue to store edges.

Next, continue implementing `produceMinimumSpanningTree` with the following:

```
mst.copyVertices(graph) // 1
val start = graph.vertices.firstOrNull() ?: return Pair(cost,
mst) // 2
visited.add(start) // 3
addAvailableEdges(start, graph, visited, priorityQueue) // 4
```

This code initiates the algorithm:

1. Copy all the vertices from the original graph to the minimum spanning tree.
2. Get the starting vertex from the graph.
3. Mark the starting vertex as visited.
4. Add all potential edges from the start vertex into the priority queue.

Finally, complete `produceMinimumSpanningTree` with:

```
while (true) {
    val smallestEdge = priorityQueue.dequeue() ?: break // 1
    val vertex = smallestEdge.destination // 2
    if (visited.contains(vertex)) continue // 3

    visited.add(vertex) // 4
    cost += smallestEdge.weight ?: 0.0 // 5

    mst.add(EdgeType.UNDIRECTED, smallestEdge.source,
        smallestEdge.destination, smallestEdge.weight) // 6

    addAvailableEdges(vertex, graph, visited, priorityQueue) // 7
}

return Pair(cost, mst) // 8
```

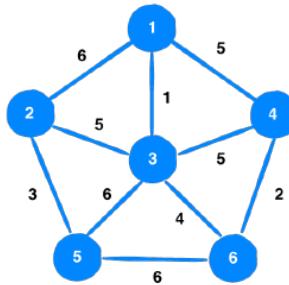
Going over the code:

1. Continue Prim's algorithm until the queue of edges is empty.
2. Get the destination vertex.
3. If this vertex has been visited, restart the loop and get the next smallest edge.
4. Mark the destination vertex as visited.
5. Add the edge's weight to the total cost.
6. Add the smallest edge into the minimum spanning tree you are constructing.
7. Add the available edges from the current vertex.
8. Once the `priorityQueue` is empty, return the minimum cost, and minimum spanning tree.



Testing your code

Navigate to the `main()` function, and you'll see the graph on the next page has been already constructed using an adjacency list.

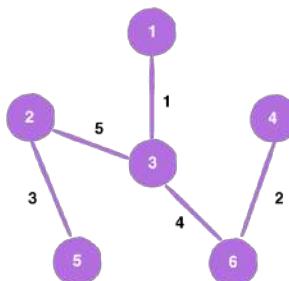


It's time to see Prim's algorithm in action. Add the following code:

```
val (cost, mst) = Prim.produceMinimumSpanningTree(graph)
println("cost: $cost")
println("mst:")
println(mst)
```

This constructs a graph from the example section. You'll see the following output:

```
cost: 15.0
mst:
3 ----> [ 1, 6, 2 ]
4 ----> [ 6 ]
1 ----> [ 3 ]
5 ----> [ 2 ]
2 ----> [ 3, 5 ]
6 ----> [ 3, 4 ]
```



Performance

In the algorithm above, you maintain three data structures:

1. An adjacency list graph to build a minimum spanning tree. Adding vertices and edges to an adjacency list is $O(1)$.
2. A Set to store all vertices you have visited. Adding a vertex to the set and checking if the set contains a vertex also have a time complexity of $O(1)$.
3. A min-priority queue to store edges as you explore more vertices. The priority queue is built on top of a heap and insertion takes $O(\log E)$.

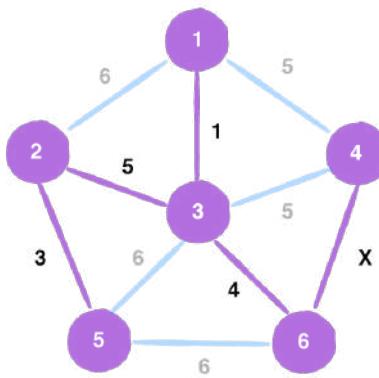
The worst-case time complexity of Prim's algorithm is $O(E \log E)$. This is because, each time you dequeue the smallest edge from the priority queue, you have to traverse all the edges of the destination vertex ($O(E)$) and insert the edge into the priority queue ($O(\log E)$).



Challenges

Challenge 1: Discover the edge weight

Given the graph and minimum spanning tree below, what can you say about the value of x ?

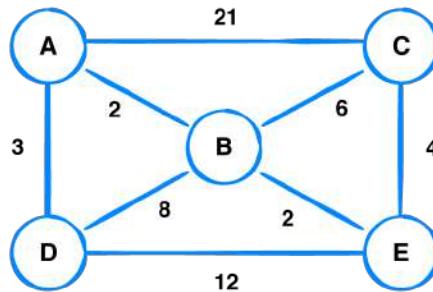


Solution 1

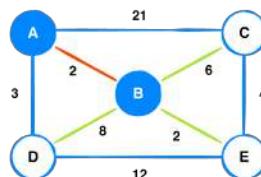
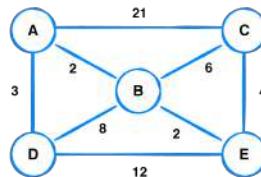
The value of x is no more than 5.

Challenge 2: One step at the time

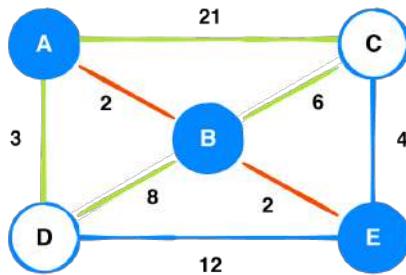
Given the graph below, step through Prim's algorithm to produce a minimum spanning tree, and provide the total cost. Start at vertex B. If two edges share the same weight, prioritize them alphabetically.



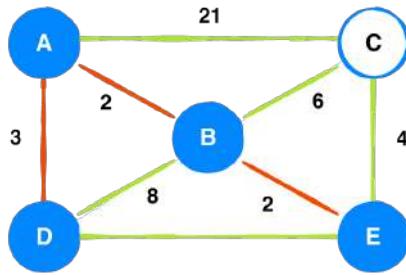
Solution 2



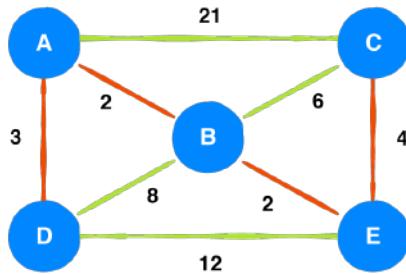
```
Edges [A:2, D:8, C:6, E:2]  
Edges part of MST: [A:2]  
Explored [A, B]
```



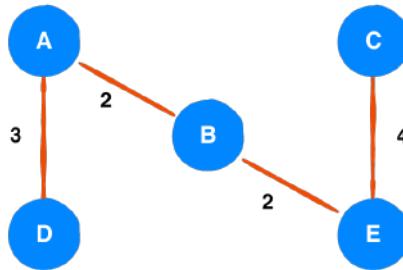
```
Edges [D:8, C:6, E:2, D:3, C:21]
Edges part of MST: [A:2, E:2]
Explored [A, B, E]
```



```
Edges [D:8, C:6, D:3, C:21, D:12, C:4]
Edges part of MST: [A:2, E:2, D:3]
Explored [A, B, E, D]
```



```
Edges [C:6, C:21, C:4]
Edges part of MST: [A:2, E:2, D:3, C:4]
Explored [A, B, E, D, C]
```



```
Edges [A:2, E:2, D:3, C:4]
Explored [A, B, E, D, C]
Total Cost: 11
```

Key points

- You can leverage three different data structures: Priority queue, set, and adjacency lists to construct Prim's algorithm.
- Prim's algorithm is a greedy algorithm that constructs a **minimum spanning tree**.
- A spanning tree is a subgraph of an undirected graph that contains all the vertices with the fewest number of edges.



Conclusion

We hope that by reading this book, you learned a lot about data structures and algorithms in Kotlin. We also hope that you had some fun in the process! Knowing when and why to apply data structures and algorithms goes beyond acing that whiteboard interview. With the knowledge you've gained here, you can easily and efficiently solve many data manipulation or graph analysis issues put in front of you.

If you have any questions or comments about this book and the material within its pages, please stop by our forums at <http://forums.raywenderlich.com> and look for the particular forum category for this book.

Thank you for purchasing this book. Your continued support is what makes the tutorials, books, videos, conferences and other things we do at raywenderlich.com possible, and we truly appreciate it.

Wishing you all the best in your continued algorithmic adventures,

The *Data Structures & Algorithms in Kotlin* Team



Learn Data Structure and Algorithms in Kotlin!

Data Structure and Algorithms are fundamental tools every developer should have. In this book you'll learn how to implement the main data structure in Kotlin and how to use them solving a great set of algorithms.

Who This Book Is For

This book is for intermediate Kotlin or Android developers who already know the basics of the language and want to improve their knowledge.

Topics Covered in Data Structures & Algorithms in Kotlin:

- ▶ **Introduction to Kotlin:** If you're new to Kotlin you can learn the main construct and start writing code soon.
- ▶ **Understand complexity:** When you study algorithm you need a way to compare their performance in time and space. Here you'll learn what the Big-O notation is.
- ▶ **Elementary Data Structure:** You'll learn how to implement Linked List, Stack and Queues in Kotlin.
- ▶ **Trees:** In this book you'll learn everything you need about Trees and in particular about Binary Trees, AVL Trees along with Binary Search and much more.
- ▶ **Sorting Algorithms:** Sorting algorithms are very famous and important. In this chapter you'll implement the main sorting algorithms using the tools provided by Kotlin.
- ▶ **Graphs:** Have you ever heard the name Dijkstra and the way to calculate the shortest path between two different points? In chapter you'll learn all about Graphs and how to use them in order to solve the most useful and important algorithms.

About the Tutorial Team

The Tutorial Team is a group of app developers and authors who write tutorials at the popular website raywenderlich.com. We take pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun.

If you've enjoyed the tutorials we've written in the past, you're in for a treat. The tutorials we've written for this book are some of our best yet — and this book contains detailed technical knowledge you simply won't be able to find anywhere else.