Up to date for iOS 11, Xcode 9 & Swift 4

Data Structures & Algorithms in Swift

FIRST EDITION

Implementing practical data structures with Swift 4

Kelvin **Lau** & Vincent **Ngo**

Data Structures

& Algorithms

in Swift

By Kelvin Lau & Vincent Ngo

Data Structures & Algorithms in Swift

Kelvin Lau & Vincent Ngo

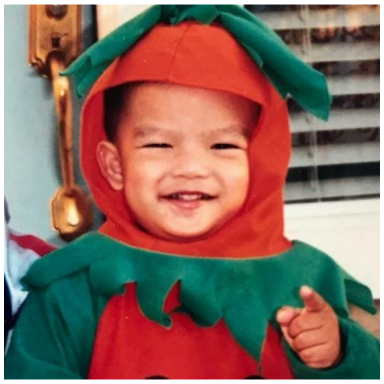
Notice of Rights

Notice of Liability

Trademarks

About the authors

Kelvin Lau is an author of this book. Kelvin is a physicist turned Swift iOS Developer. While he's currently entrenched with iOS development, he often reminisces of his aspirations to be part of the efforts in space exploration. Outside of programming work, he's an aspiring entrepreneur and musician. You can find him on Twitter: @kelvinlauKL



Vincent Ngo is an author of this book. A software developer by day, and an iOS-Swift enthusiast by night, he believes that sharing knowledge is the best way to learn and grow as a developer. Vincent starts every morning with a homemade green smoothie in hand to fuel his day. When he is not in front of a computer, Vincent is training to play in small golf tournaments, doing headstands at various locations while on a hiking adventure, or looking up how to make tamago egg. You can find him on Twitter: @vincentngo2.

About the editors

Steven Van Impe is the technical editor of this book. Steven is a computer science lecturer at the University College of Ghent, Belgium. When he's not teaching, Steven can be found on his bike, rattling over cobblestones and sweating up hills, or relaxing around the table, enjoying board games with friends. You can find Steven on Twitter as @svanimpe.

Chris Belanger is the editor of this book. Chris is the Editor in Chief at raywenderlich.com. He was a developer for nearly 20 years in various fields from e-health to aerial surveillance to industrial controls. If there are words to wrangle or a paragraph to ponder, he's on the case. When he kicks back, you can usually find Chris with guitar in hand, looking for the nearest beach. Twitter: @crispytwit.

Ray Fix is the final pass editor of this book. A passionate Swift educator, enthusiast and advocate, he is actively using Swift to create Revolve: a next generation iPad controlled research microscope at Discover Echo Inc. Ray is mostly-fluent in spoken and written Japanese and stays healthy by walking, jogging, and playing ultimate Frisbee. When he is not doing one of those things, he is writing and dreaming of code in Swift. You can find him on Twitter: @rayfix.

We’d also like to acknowledge the efforts of the following contributors to the Swift Algorithm Club GitHub repo, upon whose work portions of this book are based:

- 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

Book license

Early access edition

You're reading an an early access edition of Data Structures & Algorithms in Swift. This edition contains a sample of the chapters that will be contained in the final release.

We hope you enjoy the preview of this book, and that you'll come back to help us celebrate the full launch of Data Structures & Algorithms in Swift later in 2018!

The best way to get update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials that came out on raywenderlich.com that month, any important news like book updates or new books, and a list of our favorite development links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

Who this book is for

This book is for developers who are comfortable with Swift 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 Swift language, we recommend our book, the Swift Apprentice, which goes into depth on the Swift language itself:

- https://store.raywenderlich.com/products/swift-apprentice

If you want to learn more about iOS app development in Swift, we recommend working through our classic book, the iOS Apprentice:

- https://store.raywenderlich.com/products/ios-apprentice

What you need

To follow along with this book, you'll need the following:

- A Mac running macOS Sierra 10.12.6, at a minimum, with the latest point release and security patches installed. This is so you can install the latest version of the required development tool: Xcode.

- Xcode 9 or later. Xcode is the main development tool for writing code in Swift. You need Xcode 9 at a minimum, since that version includes Swift 4. You can download the latest version of Xcode for free from the Mac App Store, here: apple.co/1FLn51R.

If you haven't installed the latest version of Xcode, be sure to do that before continuing with the book. The code covered in this book depends on Swift 4 and Xcode 9 — you may get lost if you try to work with an older version.

Book source code and forums

You can get the source code for the book here:

www.raywenderlich.com/store/data-structures-algorithms-swift/source-code

There, you'll find all the code from the chapters for your use.

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.

About the cover

The legendary, elusive kraken has captured the imagination of sailors, artists and authors for hundreds of years. Most modern-day scientists believe that the creature known as the kraken is most likely the giant squid: a deep-water member of the Architeuthidae family that can grow up to 43 feet in length!

Little is known about the giant squid, due to its preference for cold, deep-water habitats. It’s much like the data structures and algorithms that lurk deep within software; although you may not yet understand how data structures and algorithms form the basis of scalable, high-performance solutions, this book will be your own personal Nautilus that will transport you 20,000 leagues under a sea of code!

Preface

The study of data structures is one about efficiency. Given a particular amount of data, what is the best way to store it to achieve a particular goal?

As a programmer, you regularly use a variety of collection types, such as arrays, dictionaries, and sets. These are data structures that hold a collection of data, each structure having its own performance characteristics.

As an example, consider the difference between an array and a set. Both are meant to hold a collection of elements, but trying to find a particular element in an array takes a lot longer than finding an element in a set. On the other hand, you can order the elements an array, but you can't order the elements of a set.

Data structures are a well-studied area, and the concepts are language agnostic; a data structure from C is functionally and conceptually identical to the same data structure in any other language, such as Swift. At the same time, the high-level expressiveness of Swift make it an ideal choice for learning these core concepts without sacrificing too much performance.

So why should you learn data structures and algorithms?

Interviews

When you interview for a software engineering position, chances are you'll be tested on data structures and algorithms. Having a strong foundation in data structures and algorithms is the "bar" for many companies with software engineering positions.

Work

Data structures are most relevant when working with large amounts of data. If you're dealing with a non-trivial amount of data, using the appropriate data structure will play a big role in the performance and scalability of your software.

Self-improvement

Knowing about the strategies used by algorithms to solve tricky problems gives you ideas for improvements you can make to your own code. Swift's Standard Library has small set of general purpose collection types; they definitely don't cover every case.

And yet, as you will see, these primitives can be used as a great starting point for building more complex and special purpose constructs. Knowing more data structures than just the standard array and dictionary gives you a bigger collection of tools you can use to build your own apps.

Swift Standard Library

Before you dive into the rest of this book, you'll first look at a few data structures that are baked into the Swift language. The Swift standard library refers to the framework that defines the core components of the Swift language. Inside, you'll find a variety of tools and types to help build your Swift apps.

In this chapter you'll focus on two data structures that the standard library provides right out of the box: `Array` and `Dictionary`.

Arrays

An array is a general-purpose, generic container for storing a collection of elements, and is used commonly in all sorts of Swift programs. You can create an array by using an array literal, which is a comma-separated list of values surrounded with square brackets. For example:

```
// An array of `String` elements
let people = ["Brian", "Stanley", "Ringo"]
```

This is a generic type in that it abstracts across any element type. The element type has no formal requirements; it can be anything. In the above example, the compiler type infers the elements to be `String` types.

Swift defines arrays using protocols. Each of these protocols layers more capability on the array. For example, an `Array` is a `Sequence` which means that you can iterate through it at least once. It is also a `Collection` which means it can be traversed multiple times, non-destructively, and it can be accessed using a subscript operator. Array is also a `RandomAccessCollection` which makes guarantees about efficiency.

For example, the `count` property is guaranteed to be a "cheap" constant-time operation written O(1). This means that no matter how big your array gets, computing `count` will always take the same amount of time.

Arrays are also useful for ordering the elements.

Order

Elements in an array are explicitly ordered. Using the above `people` array as an example, `"Brian"` comes before `"Stanley"`.

All elements in an array have a corresponding zero-based, integer index. For example, the `people` array from the above example has three indices, one corresponding to each element.

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

```
people[0] // "Brian"
people[1] // "Stanley"
people[2] // "Ringo"
```

Order is defined by the array data structure and should not be taken for granted. Some data structures, such as `Dictionary`, have a weaker concept of order.

Random-access

Random-access is a trait that data structures can claim if they can handle element retrieval in constant O(1) time. For example, getting `"Ringo"` from the `people` array takes constant time. Again, this performance should not be taken for granted. Other data structures such as linked lists and trees do not have constant time access.

Array performance

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

Insertion location

The first factor is where you choose to insert the new element inside the array. The most efficient scenario for adding an element to an array is to append it at the end of the array:

```
people.append("Charles")
print(people) // prints ["Brian", "Stanley", "Ringo", "Char
```

Inserting `"Charles"` using the `append` method will place the string at the end of the array. This is a constant-time operation, and is the most efficient way to add elements into an array. However, there may come a time that you need to insert an element in a particular location, such as in the very middle of the array. In such a scenario, this is an O(n) operation.

To help illustrate why that is the case, consider the following analogy. You're standing in line for the theater. Someone new comes along, to join the lineup. What's the easiest place to add people to the lineup? At the end of course!

If the newcomer tried to insert themselves into the middle of the line, they would have to convince half the lineup to shuffle back to make room.

And if they were terribly rude, they'd try to insert themselves 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 the array works. Inserting new elements from anywhere aside from the end of the array will force elements to shuffle backwards to make room for the new element:

```
people.insert("Andy", at: 0)
// ["Andy", "Brian", "Stanley", "Ringo", "Charles"]
```

To be precise, every element must shift backwards by one index, which takes n steps. Because of this it is considered a linear time or O(n) operation. If the number of elements in the array doubles, the time required for this `insert` 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.

The second factor that determines the speed of insertion is the array's `capacity`. If the space that the array pre-allocated (the capacity) is exceeded it must reallocate storage and copy over the current elements. This means that any given insertion, even at the end, could take n steps to complete if a copy is made. However, the standard library employs a subtle trick to prevent appends from being O(n) time. Each time it runs out of storage and needs to copy, it doubles the capacity. Doing this allows arrays to have an amortized cost that it is still constant time O(1).

Dictionary

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

```
var scores: [String: Int] = ["Eric": 9, "Mark": 12, "Wayne"
```

Dictionaries don’t have any guarantees of order, nor can you insert at a specific index. They also put a requirement on the `Key` type that it be `Hashable`. Fortunately almost all of the standard types are already `Hashable` and in the most recent versions Swift, adopting the `Hashable` protocol is now trivial.

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

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

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

```
["Eric": 9, "Mark": 12, "Andrew": 0, "Wayne": 1]
```

The `"Andrew"` key is inserted somewhere into dictionary. Dictionaries are unordered, so you can’t guarantee where new entries will be put. It is possible to traverse through the key-values of a dictionary multiple times as the `Collection` protocol affords. This order, while not defined, will be the same every time it is traversed until the collection is changed (mutated).

The lack of explicit ordering disadvantage comes with some redeeming traits. Unlike the array, dictionaries don’t need to worry about elements shifting around. Inserting into a dictionary is always O(1). Lookup operations are also done in O(1) time, which is significantly faster than the O(n) lookup time for a particular element in the array.

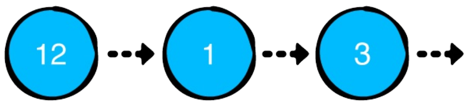
Where to go from here?

This chapter covers two of the most common data structures in Swift, and briefly highlights some trade-offs between the two. The rest of the book will look at other data structures that have unique performance characteristics that give them their edge in certain scenarios. Surprisingly, many of these other data structures can be built efficiently using these two simple standard library primitives.

Linked list

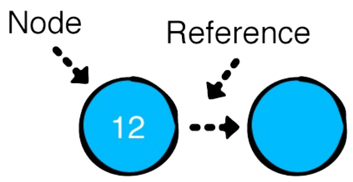
A linked list is a collection of values arranged in a linear unidirectional sequence. A linked list has several theoretical advantages over contiguous storage options such as the Swift Array:

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



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. A `nil` value represents the end of the list.



Open up the starter playground for this chapter so you can dive right into the code.

Node

Create a new Swift file in the Sources directory and name it

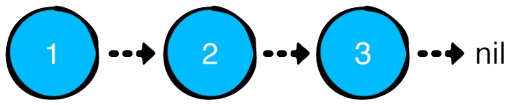
Create a new Swift file in the Sources directory and name it Node.swift. Add the following to the file:

```
public class Node<Value> {

  public var value: Value
  public var next: Node?

  public init(value: Value, next: Node? = nil) {
    self.value = value
    self.next = next
  }
}

extension Node: CustomStringConvertible {

  public var description: String {
    guard let next = next else {
      return "\(value)"
    }
    return "\(value) -> " + String(describing: next) + " "
  }
}
```

Navigate to the playground page and add the following:

```
example(of: "creating and linking nodes") {
  let node1 = Node(value: 1)
  let node2 = Node(value: 2)
  let node3 = Node(value: 3)

  node1.next = node2
  node2.next = node3

  print(node1)
}
```

You’ve just created three nodes and connected them:



In the console, you should see the following output:

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

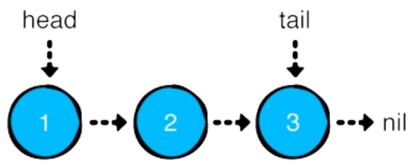
As far as practicality goes, the current method of building lists leaves a lot to be desired. You can easily see that building long lists this way is impractical. A common way to alleviate this problem is to build a `LinkedList` that manages the `Node` objects. You'll do just that!

LinkedList

In the Sources directory, create a new file and name it LinkedList.swift. Add the following to the file:

```
public struct LinkedList<Value> {

  public var head: Node<Value>?
  public var tail: Node<Value>?

  public init() {}

  public var isEmpty: Bool {
    return head == nil
  }
}

extension LinkedList: CustomStringConvertible {

  public var description: String {
    guard let head = head else {
      return "Empty list"
    }
    return String(describing: head)
    }
}
```

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



Adding values to the list

As mentioned before, 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(after:)`: Adds a value after a particular node of the list.

You'll implement each of these in the next section and analyze their performance characteristics.

push

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`:

```
public mutating func push(_ value: Value) {
    head = Node(value: value, next: head)
    if tail == nil {
      tail = head
    }
  }
```

In the case where you’re pushing into an empty list, the new node is both the `head` and `tail` of the list.

Head back to the playground page and add the following:

```
example(of: "push") {
  var list = LinkedList<Int>()
  list.push(3)
  list.push(2)
  list.push(1)

  print(list)
}
```

Your console output should show this:

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

append

The next operation you’ll look at is `append`. This is meant to add a value at the end of the list, and is known as tail-end insertion.

Head back into LinkedList.swift and add the following code just below `push`:

```
public mutating func append(_ value: Value) {

  // 1
  guard !isEmpty else {
    push(value)
    return
  }

  // 2
  tail!.next = Node(value: value)

  // 3
  tail = tail!.next
}
```

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 simply invoke `push` to do the work for you.

2. In all other cases, you simply create a new node after the `tail` node. Force unwrapping is guaranteed to succeed since you push in the `isEmpty` case with the above `guard` statement.

3. Since this is tail-end insertion, your new node is also the tail of the list.

Leap back into the playground and write the following at the bottom:

```
example(of: "append") {
  var list = LinkedList<Int>()
  list.append(1)
  list.append(2)
  list.append(3)

  print(list)
}
```

You should see the following output in the console:

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

insert(after:)

The third and final operation for adding values is `insert(after:)`. 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.

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

Back in LinkedList.swift, add the following code just below `append`:

```
public func node(at index: Int) -> Node<Value>? {
  // 1
  var currentNode = head
  var currentIndex = 0

  // 2
  while currentNode != nil && currentIndex < index {
    currentNode = currentNode!.next
    currentIndex += 1
  }

  return currentNode
}
```

`node(at:)` will try 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've reached the desired index. Empty lists or out-of-bounds

indexes will result in a `nil` return value.

Now you need to insert the new node.

Add the following method just below `node(at:)`:

```
// 1
@discardableResult
public mutating func insert(_ value: Value,
                            after node: Node<Value>)
                            -> Node<Value> {
  // 2
  guard tail !== node else {
    append(value)
    return tail!
  }
  // 3
  node.next = Node(value: value, next: node.next)
  return node.next!
}
```

Here's what you've done:

1. `@discardableResult` lets callers ignore the return value of this method without the compiler jumping up and down warning you about it.

2. In the case where this method is called with the `tail` node, you'll call the functionally equivalent `append` method. This will take care of updating `tail`.

3. Otherwise, you simply link up the new node with the rest of the list and return the new node.

Hop back to the playground page to test this out. Add the following to the bottom of the playground:

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

  print("Before inserting: \(list)")
  var middleNode = list.node(at: 1)!
  for _ in 1...4 {
    middleNode = list.insert(-1, after: middleNode)
  }
  print("After inserting: \(list)")
}
```

You should see the following output:

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

Performance analysis

Whew! You've 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(after:)**	**node(at:)**
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 main 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. `remove(at:)`: Removes a value anywhere in the list.

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

pop

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

Add the following method to `LinkedList`:

```
@discardableResult
public mutating func pop() -> Value? {
  defer {
    head = head?.next
    if isEmpty {
      tail = nil
    }
  }
  return head?.value
}
```

`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. ARC will remove the old node from memory once the method finishes, since there will be no more references attached to it. In the event that the list becomes empty, you set `tail` to `nil`.

Head back inside the playground page and test it out by adding the following code at the bottom:

```
example(of: "pop") {
  var list = LinkedList<Int>()
  list.push(3)
  list.push(2)
  list.push(1)

  print("Before popping list: \(list)")
  let poppedValue = list.pop()
  print("After popping list: \(list)")
  print("Popped value: " + String(describing: poppedValue))
}
```

You should see the following output:

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

removeLast

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'll have to do an arduous traversal. Add the following code just below `pop`:

```
@discardableResult
public mutating func removeLast() -> Value? {
  // 1
  guard let head = head else {
    return nil
  }
  // 2
  guard head.next != nil else {
    return pop()
  }
```

```
    // 3
    var prev = head
    var current = head

    while let next = current.next {
      prev = current
      current = next
    }
    // 4
    prev.next = nil
    tail = prev
    return current.value
  }
```

Here’s what’s going on:

1. If `head` is `nil`, there’s nothing to remove, so you return `nil`.

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’ll just delegate this work to it.

3. You keep searching for a next node until `current.next` is `nil`. This signifies that `current` is the last node of the list.

4. Since `current` is the last node, you simply disconnect it using the `prev.next` reference. You also make sure to update the `tail` reference.

Head back to the playground page and add the following to the bottom:

```
  example(of: "removing the last node") {
    var list = LinkedList<Int>()
    list.push(3)
    list.push(2)
    list.push(1)

    print("Before removing last node: \(list)")
    let removedValue = list.removeLast()
    print("After removing last node: \(list)")
    print("Removed value: " + String(describing: removedValue
  }
```

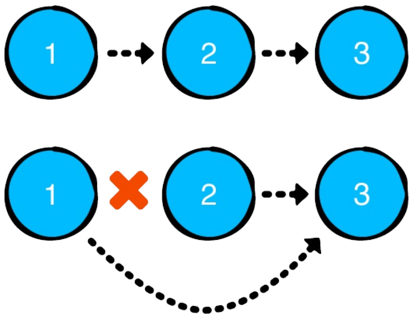
You should see the following at the bottom of the console:

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

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

remove(after:)

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



Navigate back to LinkedList.swift and add the following method below `removeLast`:

```
@discardableResult
public mutating func remove(after node: Node<Value>) -> Val
  defer {
    if node.next === tail {
      tail = node
    }
    node.next = node.next?.next
  }
  return node.next?.value
}
```

The unlinking of the nodes occurs in the `defer` block. Special care needs to be taken if the removed node is the tail node, since the `tail` reference will need to be updated.

Head back to the playground to try it out. You know the drill:

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

  print("Before removing at particular index: \(list)")
  let index = 1
  let node = list.node(at: index - 1)!
  let removedValue = list.remove(after: node)

  print("After removing at index \(index): \(list)")
  print("Removed value: " + String(describing: removedValue
}
```

You should 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: Optional(2)
```

Try adding more elements and play around with the value of index. Similar to `insert(at:)`, 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:

	pop	**removeLast**	**remove(after:)**
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 Swift semantics. In the next half of the chapter, you’ll focus on making the interface as Swifty as possible.

Swift collection protocols

The Swift standard library has a set of protocols that help define what’s expected of a particular type. Each of these protocols provides certain guarantees on characteristics and performance. Of these set of protocols, four are referred to as collection protocols.

Here’s a small sampler of what each protocol represents:

- Tier 1, Sequence: A sequence type provides sequential access to its elements. This axiom comes with a caveat: Using the sequential access may destructively consume the elements.

- Tier 2, Collection: A collection type is a sequence type that provides additional guarantees. A collection type is finite and allows for repeated nondestructive sequential access.

- Tier 3, BidirectionalColllection: A collection type can be a bidirectional collection type if it, as the name suggests, can allow for bidirectional travel up and down the sequence. This isn't possible for the linked list, since you can only go from the head to the tail, but not the other way around.

- Tier 4, RandomAccessCollection: A bidirectional collection type can be a random access collection type if it can guarantee that accessing an element at a particular index will take just as long as access an element at any other index. This is not possible for the linked list, since accessing a node near the front of the list is substantially quicker than one that is further down the list.

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 earn two qualifications from the Swift collection protocols. First, since a linked list is a chain of nodes, adopting the `Sequence` protocol makes sense. Second, since the chain of nodes is a finite sequence, it makes sense to adopt the `Collection` protocol.

Becoming a Swift collection

In this section, you'll look into implementing the `Collection` protocol. A collection type is a finite sequence and provides nondestructive sequential access. A Swift `Collection` also allows for access via a subscript , which is a fancy term for saying an index can be mapped to a value in the collection.

Here's an example of using the subscript of a Swift `Array`:

```
array[5]
```

The index of an array is an `Int` value, 5 in this example. The subscript operation is defined with the square brackets. Using the subscript with an index will return you a value from the collection.

Custom collection indexes

A defining metric for performance of the `Collection` protocol methods is the speed of mapping an `Index` to a value. Unlike other storage options such as the Swift `Array`, the linked list cannot achieve O(1) subscript operations using integer indexes. Thus, your goal is to define a custom index that contains a reference to its respective node.

In LinkedList.swift, add the following extension:

```
extension LinkedList: Collection {

  public struct Index: Comparable {

    public var node: Node<Value>?

    static public func ==(lhs: Index, rhs: Index) -> Bool {
      switch (lhs.node, rhs.node) {
      case let (left?, right?):
        return left.next === right.next
      case (nil, nil):
        return true
      default:
        return false
      }
    }

    static public func <(lhs: Index, rhs: Index) -> Bool {
      guard lhs != rhs else {
        return false
      }
      let nodes = sequence(first: lhs.node) { $0?.next }
      return nodes.contains { $0 === rhs.node }
    }
  }
}
```

You'll use this custom index to fulfill `Collection` requirements. Write the following inside the extension to complete it:

```
// 1
public var startIndex: Index {
  return Index(node: head)
}
// 2
public var endIndex: Index {
  return Index(node: tail?.next)
}
// 3
public func index(after i: Index) -> Index {
  return Index(node: i.node?.next)
}
// 4
public subscript(position: Index) -> Value {
  return position.node!.value
}
```

1. The `startIndex` is reasonably defined by the `head` of the linked list.

2. `Collection` defines the `endIndex` as the index right after the last accessible value, so you give it `tail?.next`.

3. `index(after:)` dictates how the index can be incremented. You simply give it an index of the immediate next node.

4. The `subscript` is used to map an `Index` to the value in the collection. Since you've created the custom index, you can easily achieve this in constant time by referring to the node's value.

That wraps up the procedures for adopting `Collection`. Navigate back to the playground page and take it for a test drive:

```
example(of: "using collection") {
  var list = LinkedList<Int>()
  for i in 0...9 {
    list.append(i)
  }

  print("List: \(list)")
  print("First element: \(list[list.startIndex])")
  print("Array containing first 3 elements: \(Array(list.pre
  print("Array containing last 3 elements: \(Array(list.suf

  let sum = list.reduce(0, +)
  print("Sum of all values: \(sum)")
}
```

You should see the following output:

```
---Example of using collection---
List: 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9
First element: 0
Array containing first 3 elements: [0, 1, 2]
Array containing last 3 elements: [7, 8, 9]
Sum of all values: 45
```

Value semantics and copy-on-write

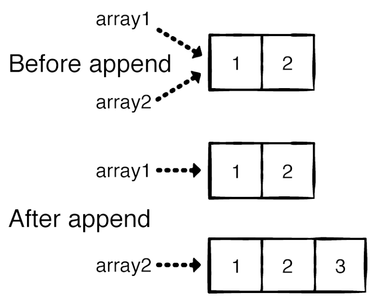
Another important quality of a Swift collections is that they have value semantics. This is implemented using copy-on-write, hereby known as COW. To illustrate this concept, you'll verify this behavior using arrays. Write the following at the bottom of the playground page:

```
example(of: "array cow") {
  let array1 = [1, 2]
  var array2 = array1
  print("array1: \(array1)")
  print("array2: \(array2)")

  print("---After adding 3 to array 2---")
  array2.append(3)
  print("array1: \(array1)")
  print("array2: \(array2)")
}
```

You should see the following output:

```
---Example of array cow---
array1: [1, 2]
array2: [1, 2]
---After adding 3 to array 2---
array1: [1, 2]
array2: [1, 2, 3]
```

The elements of `array1` are unchanged when `array2` is modified. Underneath the hood, `array2` makes a copy of the underlying storage when `append` is called:



Let's check whether or not your linked list has value semantics. Write the following at the bottom of the playground page:

```
example(of: "linked list cow") {
  var list1 = LinkedList<Int>()
  list1.append(1)
  list1.append(2)
  var list2 = list1
```

```
    print("List1: \(list1)")
    print("List2: \(list2)")

    print("After appending 3 to list2")
    list2.append(3)
    print("List1: \(list1)")
    print("List2: \(list2)")
  }
```

You should see the following output:

```
---Example of linked list cow---
List1: 1 -> 2
List2: 1 -> 2
After appending 3 to list2
List1: 1 -> 2 -> 3
List2: 1 -> 2 -> 3
```

Unfortunately, your linked list does not have value semantics! This is because your underlying storage uses a reference type (`Node`). This is a serious problem, as `LinkedList` is a struct and therefore should use value semantics. Implementing COW will fix this problem.

The strategy to achieve value semantics with COW is fairly straightforward. Before mutating the contents of the linked list, you want to perform a copy of the underlying storage and update all references (`head` and `tail`) to the new copy.

Head back to LinkedList.swift and add the following method to `LinkedList`:

```
private mutating func copyNodes() {
  guard var oldNode = head else {
    return
  }

  head = Node(value: oldNode.value)
  var newNode = head
  while let nextOldNode = oldNode.next {
    newNode!.next = Node(value: nextOldNode.value)
    newNode = newNode!.next

    oldNode = nextOldNode
  }

  tail = newNode
}
```

This method will replace the existing nodes of your linked list with newly allocated ones with the same value.

Now find all other methods in `LinkedList` marked with the `mutating` keyword and call `copyNodes` at the top of every method.

There are six methods in total:

- `push`
- `append`
- `insert(after:)`
- `pop`
- `removeLast`
- `remove(after:)`

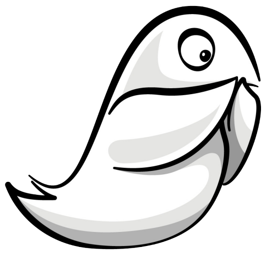
After you've completed the retrofits, the last `example` function call should yield the following output:

```
---Example of linked list cow---
List1: 1 -> 2
List2: 1 -> 2
After appending 3 to list2
List1: 1 -> 2
List2: 1 -> 2 -> 3
```

Which is what you want! Well, other than introducing a O(n) overhead on every mutating call...

Optimizing COW

The O(n) overhead on every mutating call is unacceptable.



There are two avenues that help alleviate this problem. The first is to avoid copying when the nodes only have one owner.

isKnownUniquelyReferenced

In the Swift Standard Library lives a function named `isKnownUniquelyReferenced`. This function can be used to determine whether or not an object has exactly one reference to it. Let's test this out in the linked list COW example.

In the last `example` function call, find the line where you wrote `var list2 = list` and update that to the following:

```
print("List1 uniquely referenced: \(isKnownUniquelyReference
var list2 = list1
print("List1 uniquely referenced: \(isKnownUniquelyReference
```

You should see two new lines in the console:

```
List1 uniquely referenced: true
List1 uniquely referenced: false
```

Using `isKnownUniquelyReferenced`, you can check whether or not the underlying node objects are being shared! Since you've verified this behaviour, remove the two `print` statements. Your path is clear. Add the following condition to the top of `copyNodes`:

```
guard !isKnownUniquelyReferenced(&head) else {
  return
}
```

You can be pleased that COW is still very much in effect:

```
---Example of linked list cow---
List1: 1 -> 2
List2: 1 -> 2
After appending 3 to list2
List1: 1 -> 2
List2: 1 -> 2 -> 3
```

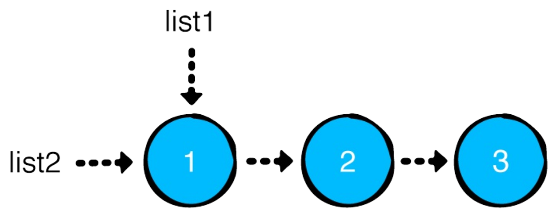
With this change, your linked list performance will reclaim its previous performance with the benefits of COW.

Sharing nodes

The second optimization is a partial sharing of nodes. As it turns out, there are certain scenarios where you can avoid a copy. A comprehensive evaluation of all the scenarios is beyond the scope of this book, but you'll try to get an understanding of how this works.

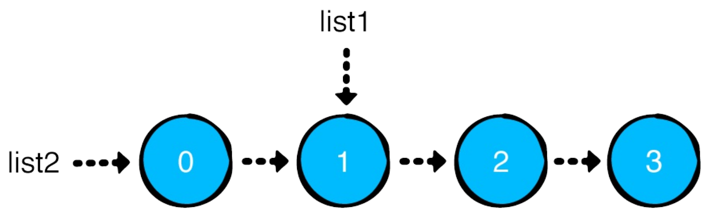
Take a look at the following example (no need to write this down):

```
var list1 = LinkedList<Int>()
(1...3).forEach { list1.append($0) }
var list2 = list1
```



Now consider the consequence of doing a push operation on `list2` with cow disabled:

```
list2.push(0)
```

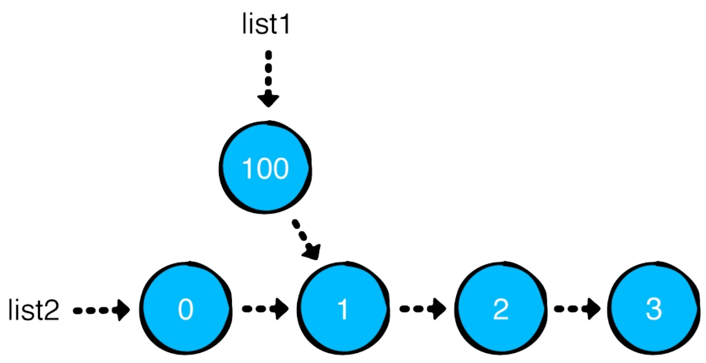


Is `list1` affected by `push` operation on `list2`? Not in this case! If you were to print the two lists, you'll get the following output:

```
List1: 1 -> 2 -> 3
List2: 0 -> 1 -> 2 -> 3
```

The result of pushing 100 to `list1` in this case is also safe:

```
list1.push(100)
```



If you were to print the two lists now, you'll get the following output:

```
List1: 100 -> 1 -> 2 -> 3
List2: 0 -> 1 -> 2 -> 3
```

The unidirectional nature of the linked list means that head first insertions can ignore the "COW tax"!

Where to go from here?

You've accomplished a lot in this chapter, and if you understood most of what you've read, you're in great shape.

Stacks

Stacks are everywhere. Here are some common examples of things you would stack:

- pancakes
- books
- paper
- 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 topmost item.

Stack

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 bad 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 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.

Stacks are used prominently in all disciplines of programming. To list a few:

- iOS uses the navigation stack to push and pop view controllers into and out of view.
- 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

Open up the starter playground for this chapter. In the Sources folder of your playground, create a file named Stack.swift. Inside the file, write the following:

```
public struct Stack<Element> {

  private var storage: [Element] = []

  public init() { }
}

extension Stack: CustomStringConvertible {

  public var description: String {
    let topDivider = "----top----\n"
    let bottomDivider = "\n-----------"

    let stackElements = storage
      .map { "\($0)" }
      .reversed()
      .joined(separator: "\n")
    return topDivider + stackElements + bottomDivider
  }
}
```

Choosing the right storage type for your stack is important. The array is an obvious choice, since it offers constant time insertions and deletions at one end via `append` and `popLast`. Usage of these two operations will facilitate the LIFO nature of stacks.

push and pop

Add the following two operations to your `Stack`:

```
public mutating func push(_ element: Element) {
  storage.append(element)
}

@discardableResult
public mutating func pop() -> Element? {
  return storage.popLast()
}
```

Fairly straightforward! Head back to the playground page and write the following:

```
example(of: "using a stack") {
  var stack = Stack<Int>()
  stack.push(1)
  stack.push(2)
  stack.push(3)
  stack.push(4)

  print(stack)

  if let poppedElement = stack.pop() {
    assert(4 == poppedElement)
    print("Popped: \(poppedElement)")
  }
}
```

You should see the following output:

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

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

Non-essential operations

There are a couple of nice-to-have operations that make a stack easier to use. Head back into Stack.swift and add the following to `Stack`:

```swift
public func peek() -> Element? {
 return storage.last
}

public var isEmpty: Bool {
  return peek() == nil
}
```

`peek` is an operation that is often attributed to the stack interface. The idea of `peek` is to look at the top element of the stack without mutating its contents.

Less is more

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

You might want to take an existing array 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, since you can write an initializer that just sets the underlying private storage. Add the following to your stack implementation:

```
public init(_ elements: [Element]) {
  storage = elements
}
```

Now add this example to the main playground:

```
example(of: "initializing a stack from an array") {
  let array = ["A", "B", "C", "D"]
  var stack = Stack(array)
  print(stack)
  stack.pop()
}
```

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

You can go a step further and make your stack initializable from an array literal. Add this to your stack implementation:

```
extension Stack: ExpressibleByArrayLiteral {
  public init(arrayLiteral elements: Element...) {
    storage = elements
  }
}
```

Now go back to the main playground page and add:

```
example(of: "initializing a stack from an array literal") {
  var stack: Stack = [1.0, 2.0, 3.0, 4.0]
  print(stack)
  stack.pop()
}
```

This creates a stack of `Doubles` and pops the top value 4.0. Again type inference saves you from having to type the more verbose `Stack<Double>`.

Where to go from here?

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, simply backtrack by popping from the stack and continuing until you escape or hit another dead end.

Queues

Lines are everywhere, whether you are 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 FIFO or first-in-first-out ordering, meaning 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.

Common operations

Let's establish a protocol for queues:

```
public protocol Queue {
  associatedtype Element
  mutating func enqueue(_ element: Element) -> Bool
  mutating func dequeue() -> Element?
  var isEmpty: Bool { get }
  var peek: Element? { get }
}
```

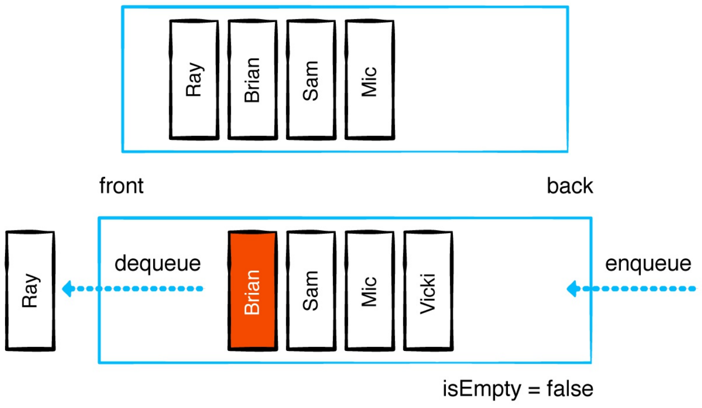
The protocol describes the common operations for a queue:

- `enqueue`: Insert an element at the back of the queue. Returns `true` if the operation was successful.
- `dequeue`: Remove the element at the front of the queue and return it.
- `isEmpty`: Check if the queue is empty.
- `peek`: Return 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 really need to know what the contents are in between. If you did, you would probably just use an array.

Example of a queue

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



The queue currently holds Ray, Brian, Sam, and Mic. Once Ray has received his ticket, he moves out of the line. By calling `dequeue()`, Ray is removed from the front of the queue.

Calling `peek` will return Brian since he is now at the front of the line.

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

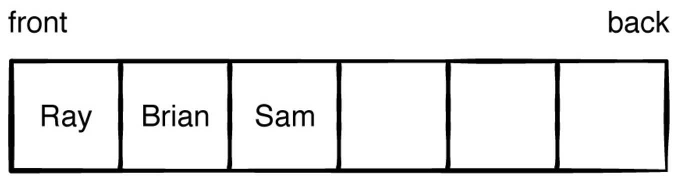
In the following sections, you will learn to create a queue in four different ways:

- Using an array
- Using a doubly linked list

- Using a ring buffer
- Using two stacks

Array-based implementation

The Swift Standard Library comes with core set highly-optimized, primitive data structures you can use to build higher level abstractions with. One of them is `Array`, a data structure that stores a contiguous, ordered list of elements. In this section, you will use an array to create a queue.



Open the starter playground. To the QueueArray page, add the following:

```
public struct QueueArray<T>: Queue {
  private var array: [T] = []
  public init() {}
}
```

Here you've defined a generic `QueueArray` struct that adopts the `Queue` protocol. Note that the associated type `Element` is inferred by the type parameter `T`.

Next, you'll complete the implementation of `QueueArray` to conform to the `Queue` protocol.

Leveraging arrays

Add the following code to `QueueArray`:

```
public var isEmpty: Bool {
  return array.isEmpty // 1
}

public var peek: T? {
  return array.first // 2
}
```

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

1. Check if the queue is empty.

2. Return the element at the front of the queue.

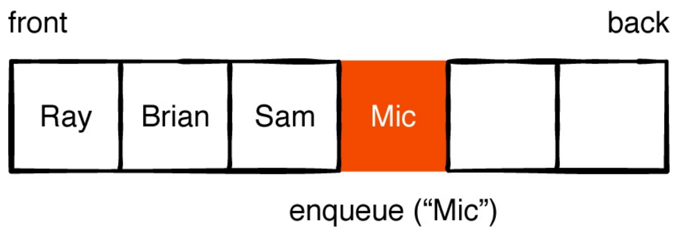
These operations are all O(1).

Enqueue

Adding an element to the back of the queue is easy. Just append an element to the array. Add the following:

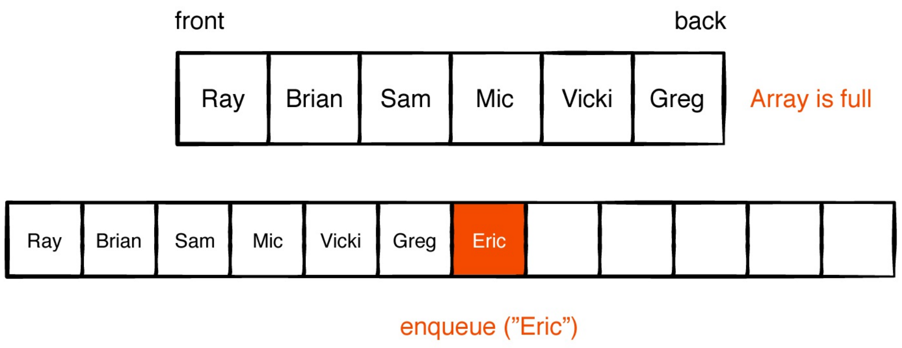
```
public mutating func enqueue(_ element: T) -> Bool {
  array.append(element)
  return true
}
```

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



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

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



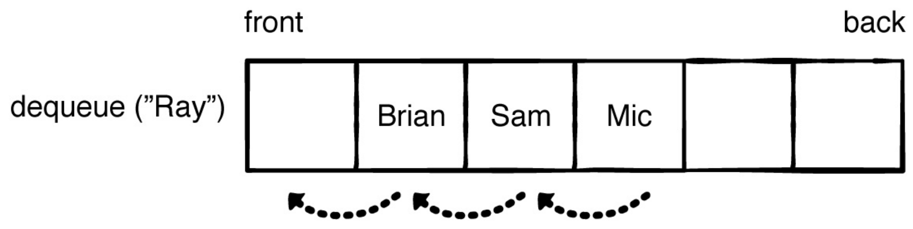
Resizing is an O(n) operation. Resizing requires the array to allocate new memory and copy all existing data over to the new array. Since this doesn't happen very often (thanks to doubling the size each time), the complexity still works out to be an ammortized O(1).

Dequeue

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

```
public mutating func dequeue() -> T? {
  return isEmpty ? nil : array.removeFirst()
}
```

If the queue is empty, `dequeue` simply returns `nil`. If not, it removes the element from the front of the array 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 array. This is always a linear time operation, because it requires all the remaining elements in the array to be shifted in memory.

Debug and test

For debugging purposes, you'll have your `queue` adopt the `CustomStringConvertible` protocol. Add the following at the bottom of the page:

```swift
extension QueueArray: CustomStringConvertible {
  public var description: String {
    return array.description
  }
}
```

Time to try out the queue you just implemented! Add the following to the bottom of the page:

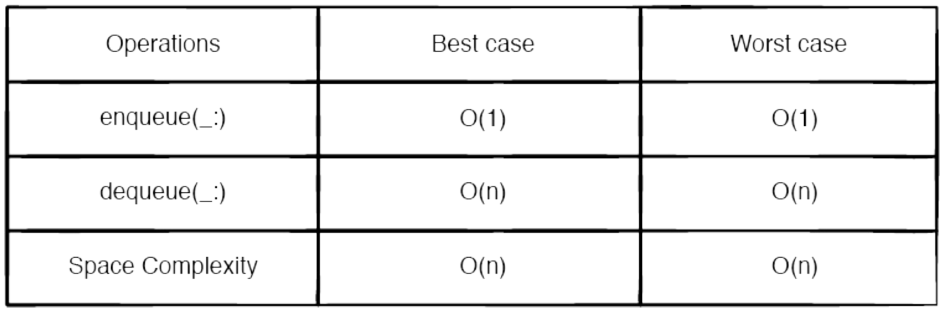
```swift
var queue = QueueArray<String>()
queue.enqueue("Ray")
queue.enqueue("Brian")
queue.enqueue("Eric")
queue.dequeue()
queue
queue.peek
```

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

Strengths and weaknesses

Here is a summary of the algorithmic and storage complexity of the array-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	Worst case
enqueue(_:)	O(1)	O(1)
dequeue(_:)	O(n)	O(n)
Space Complexity	O(n)	O(n)

You have seen how easy it is to implement an array-based queue by leveraging a Swift `Array`. 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 array 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 a `QueueArray`.

Doubly linked list implementation

Switch to the QueueLinkedList playground page. Within the page's Sources folder you will 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 `QueueLinkedList` to the very end of the page as shown below:

```
public class QueueLinkedList<T>: Queue {
  private var list = DoublyLinkedList<T>()
  public init() {}
}
```

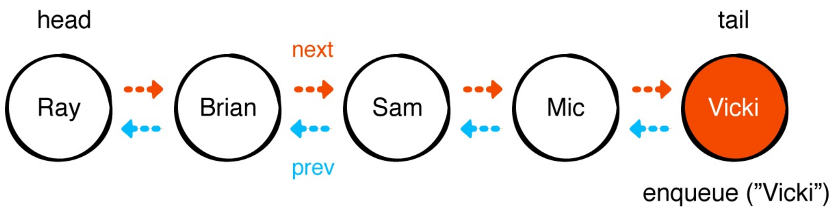
This implementation is similar to `QueueArray`, but instead of an array, you create a `DoublyLinkedList`.

Next, let's start conforming to the `Queue` protocol.

Enqueue

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

```
public func enqueue(_ element: T) -> Bool {
  list.append(element)
  return true
}
```



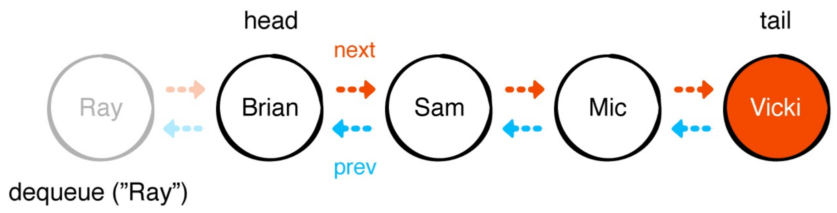
Behind the scenes, the doubly linked list will update its tail node's previous and next references to the new node. This is an O(1) operation.

Dequeue

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

```
public func dequeue() -> T? {
  guard !list.isEmpty, let element = list.first else {
    return nil
  }
  return list.remove(element)
}
```

This code checks to see if the list is not empty and the first element of the queue exists. If it doesn't, it returns `nil`. Otherwise, it removes and returns the element at the front of the queue.



Removing from the front of the list is also an O(1) operation. Compared to the array 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 array implementation, you can implement `peek` and `isEmpty` using the properties of the `DoublyLinkedList`. Add the following:

```
public var peek: T? {
  return list.first?.value
}

public var isEmpty: Bool {
  return list.isEmpty
}
```

Debug and test

For debugging purposes, you can add the following at the bottom of the page:

```swift
extension QueueLinkedList: CustomStringConvertible {
  public var description: String {
    return list.description
  }
}
```

This will leverage the `DoublyLinkedList`'s default implementation for the `CustomStringConvertible` protocol.

That's all there is to implementing a queue using a linked list! In the QueueLinkedList page of playground, you can try the example:

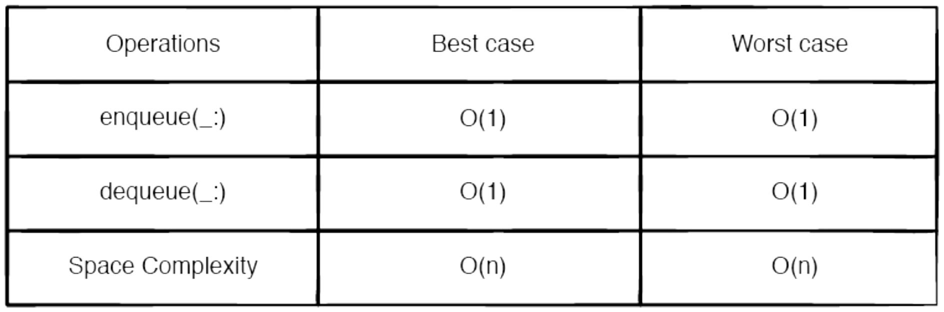
```swift
var queue = QueueLinkedList<String>()
queue.enqueue("Ray")
queue.enqueue("Brian")
queue.enqueue("Eric")
queue.dequeue()
queue
queue.peek
```

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

Strengths and weaknesses

Let's summarize of the algorithmic and storage complexity of the doubly-linked-list-based queue implementation.

Linked-List Based Queue



Operations	Best case	Worst case
enqueue(_:)	O(1)	O(1)
dequeue(_:)	O(1)	O(1)
Space Complexity	O(n)	O(n)

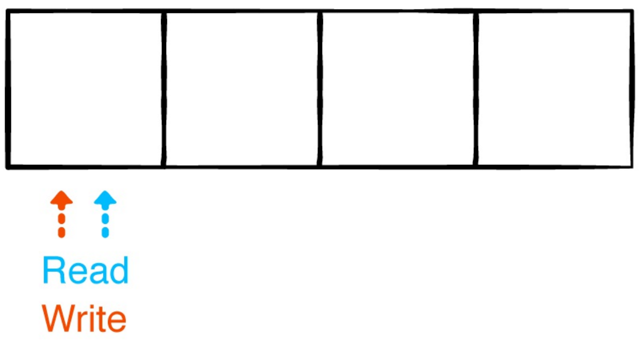
One of the main problems with `QueueArray` is dequeuing an item takes linear time. With the linked list implementation, you reduced it to a constant operation, O(1). All you needed to do was update the node's `previous` and `next` pointers.

The main weakness with `QueueLinkedList` 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 `QueueArray` does bulk allocation which is faster.

Can you eliminate allocation overhead and main 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

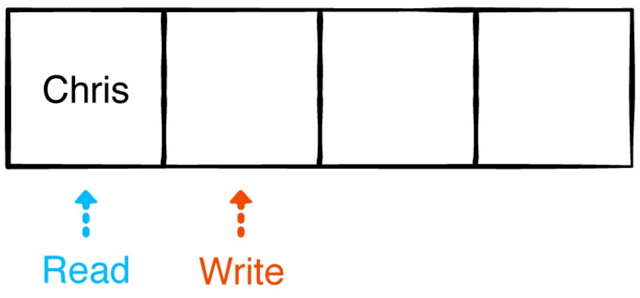
A ring buffer, also known as a circular buffer, is a fixed-size array. This data structure strategically wraps around to the beginning when there are no more items to remove at the end. Let's go over a simple example of how a queue can be implemented using a ring buffer.



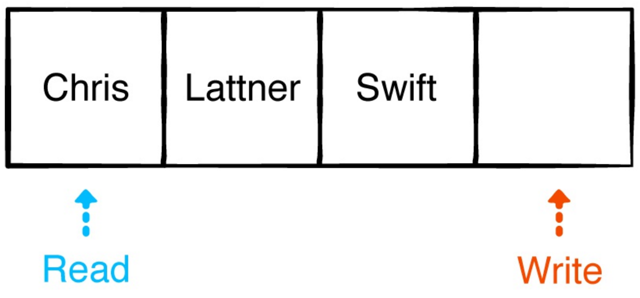
You first create a ring buffer, that has a fixed size of 4. The ring buffer has two pointers that keep track of two things:

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

Let's enqueue an item:

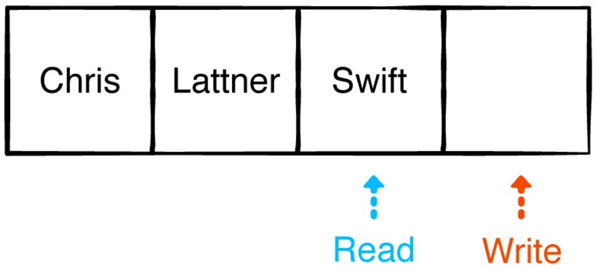


Each time you add an item to the queue, the write pointer increments by one. Let's add a few more elements:



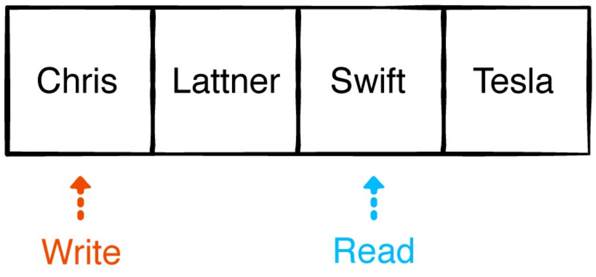
Notice that the write pointer moved two more spots and is ahead of the read pointer. This means that the queue is not empty.

Next, let's dequeue two items:



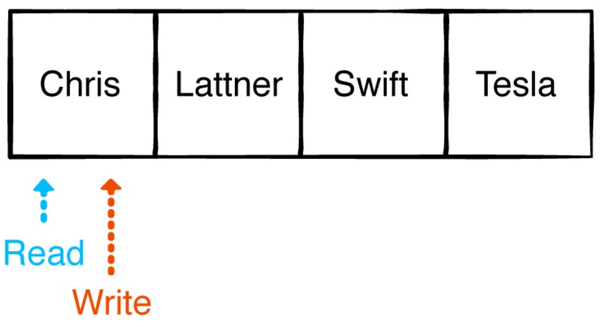
Dequeuing is the equivalent of reading a ring buffer. Notice how the read pointer moved twice.

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 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, let's implement one!

Go to the QueueRingBuffer playground page. Within the page's Sources folder you will notice a `RingBuffer` class.

> Note: If you want to learn more about the implementation of this class, head over to our Swift to get a full walkthrough at https://github.com/raywenderlich/swift-algorithm-club/tree/master/Ring%20Buffer.

In QueueRingBuffer page, add the following:

```swift
public struct QueueRingBuffer<T>: Queue {
  private var ringBuffer: RingBuffer<T>

  public init(count: Int) {
    ringBuffer = RingBuffer<T>(count: count)
  }
```

```
    public var isEmpty: Bool {
      return ringBuffer.isEmpty
    }

    public var peek: T? {
      return ringBuffer.first
    }
  }
```

Here you defined a generic `QueueRingBuffer`. Note that you must include a `count` parameter since the ring buffer has a fixed size.

To conform to the `Queue` protocol, you also created two properties `isEmpty` and `peek`. Instead of exposing `ringBuffer`, you provide helper variables 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 method below:

```
public mutating func enqueue(_ element: T) -> Bool {
  return ringBuffer.write(element)
}
```

To append an element to the queue, you simply 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:

```
public mutating func dequeue() -> T? {
  return isEmpty ? nil : ringBuffer.read()
}
```

This code checks if the queue is empty and if so, returns `nil`. 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 see your results in the playground, add the following:

```
extension QueueRingBuffer: CustomStringConvertible {
  public var description: String {
   return ringBuffer.description
  }
}
```

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 the page:

```
var queue = QueueRingBuffer<String>(count: 10)
queue.enqueue("Ray")
queue.enqueue("Brian")
queue.enqueue("Eric")
queue
queue.dequeue()
queue
queue.peek
```

This test code works just 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	Worst 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.

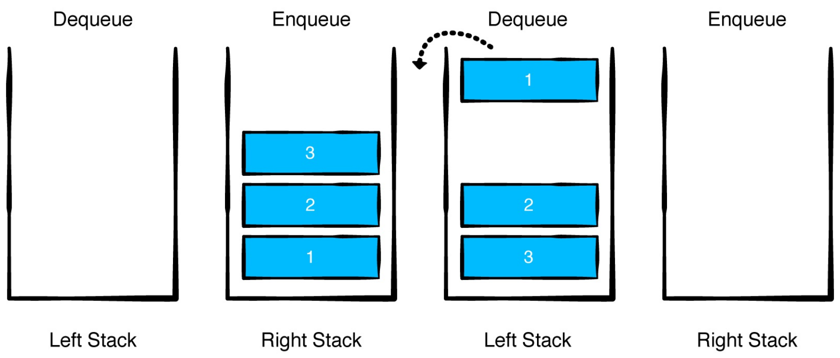
So far you have seen three implementations, a simple array, a doubly linked list, and a ring buffer. Although they appear to be eminently useful, you’ll next look at a queue implemented using two stacks. You will see how its spacial locality is far superior to the linked list. It also doesn’t need a fixed size like a ring buffer.

Double stack implementation

Open the QueueStack playground page and start by adding a generic `QueueStack` as shown below:

```
public struct QueueStack<T> : Queue {
  private var leftStack: [T] = []
  private var rightStack: [T] = []
  public init() {}
}
```

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 you can retrieve the elements using FIFO order.



Leveraging arrays

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

```
public var isEmpty: Bool {
  return leftStack.isEmpty && rightStack.isEmpty
}
```

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

Next add the following:

```
public var peek: T? {
  return !leftStack.isEmpty ? leftStack.last : rightStack.f
}
```

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, the right stack will be reversed and placed in the left stack. In this case, the element at the bottom of the right stack is next in the queue.

Note that the two properties `isEmpty` and `peek` are still O(1) operations.

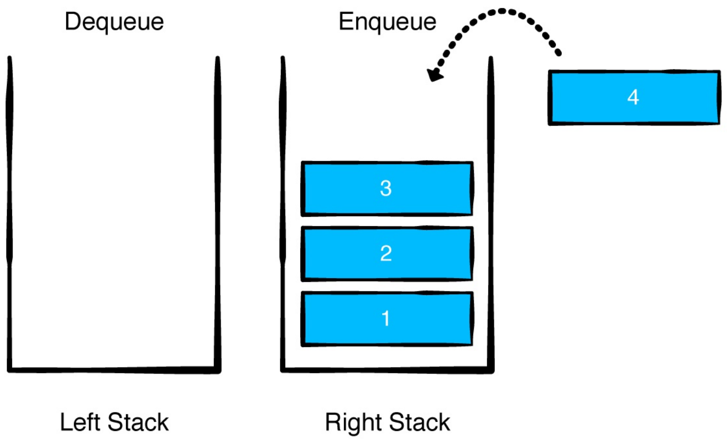
Enqueue

Next add the method below:

```
public mutating func enqueue(_ element: T) -> Bool {
  rightStack.append(element)
  return true
}
```

Recall that the right stack is used to enqueue elements. You simply push to the stack by appending to the array.

Previously from implementing the `QueueArray`, you know that appending an element is an O(1) operation.



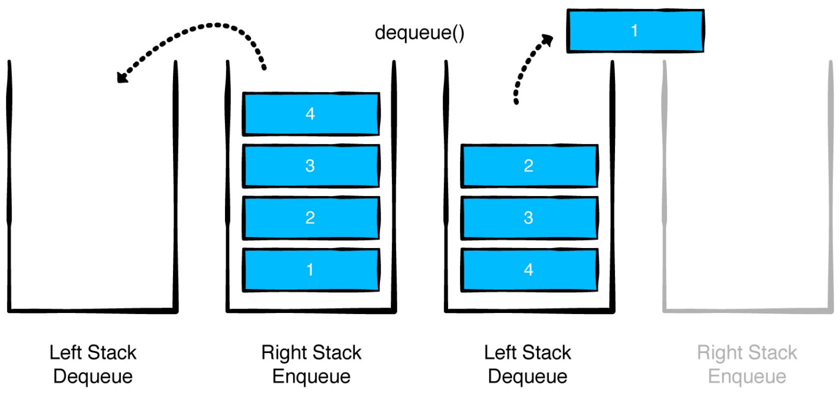
Dequeue

Removing an item from a two stack based implementation of a queue is tricky. Add the following method:

```
public mutating func dequeue() -> T? {
  if leftStack.isEmpty { // 1
```

```
    leftStack = rightStack.reversed() // 2
    rightStack.removeAll() // 3
  }
  return leftStack.popLast() // 4
}
```

1. Check to see if the left stack is empty.
2. If the left stack is empty, set it as the reverse of the right stack.



1. Invalidate your right stack. Since you have transferred everything to the left, just clear it.
2. Remove the last element from the left stack.

Remember, you only transfer the elements in the right stack when the left stack is empty!

Note: Yes, reversing the contents of an array is an O(n) operation. The overall dequeue cost is still amortized O(1). Imagine having a large number of items in both the left and right stack. If you dequeue all of the elements, first it will remove all of the elements from the left stack, then reverse-copy the right stack only once, and then continue removing elements off the left stack.

Debug and test

To see your results in the playground, add the following:

```
extension QueueStack: CustomStringConvertible {
  public var description: String {
    let printList = leftStack + rightStack.reversed()
    return printList.description
  }
}
```

Here you simply combine the left stack with the reverse of the right stack, and you print all the elements.

Let’s try out the double stack implementation:

```
var queue = QueueStack<String>()
queue.enqueue("Ray")
queue.enqueue("Brian")
queue.enqueue("Eric")
queue.dequeue()
queue
queue.peek
```

Just like all of the examples before, this code enqueues Ray, Brian and Eric, dequeues Ray and then peeks at Brian.

Strengths and weaknesses

Let’s look at a summary of the algorithmic and storage complexity of your two-stack-based implementation.

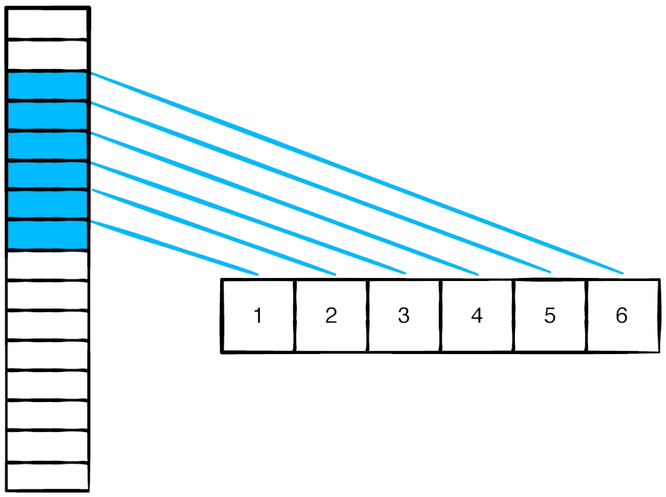
Double Stack Based Queue

Operations	Best case	Worst case
enqueue(_:)	O(1)	O(1)
dequeue(_:)	O(1)	O(1)
Space Complexity	O(n)	O(n)

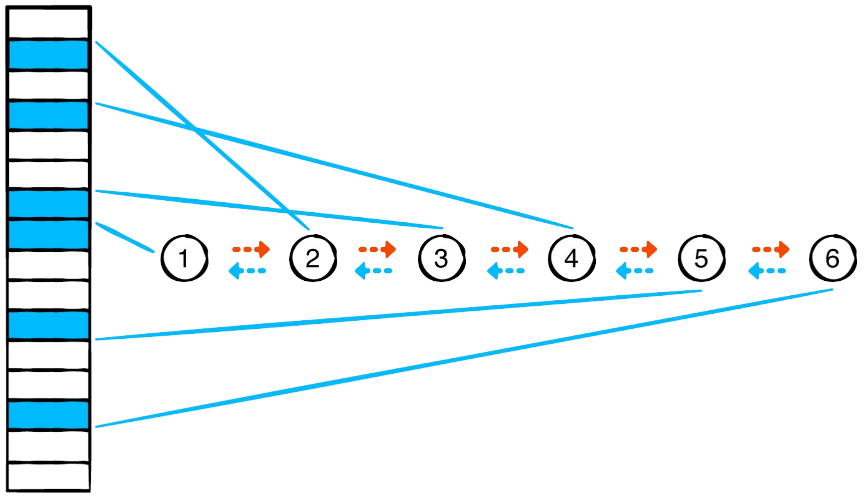
Compared to the array-based implementation, by leveraging two stacks, you were able to transform `dequeue(_:)` into an amortized O(1) operation.

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 spacial locality. This is because array elements are next to each other in memory blocks. So a large number of elements will be loaded in a cache on first access.



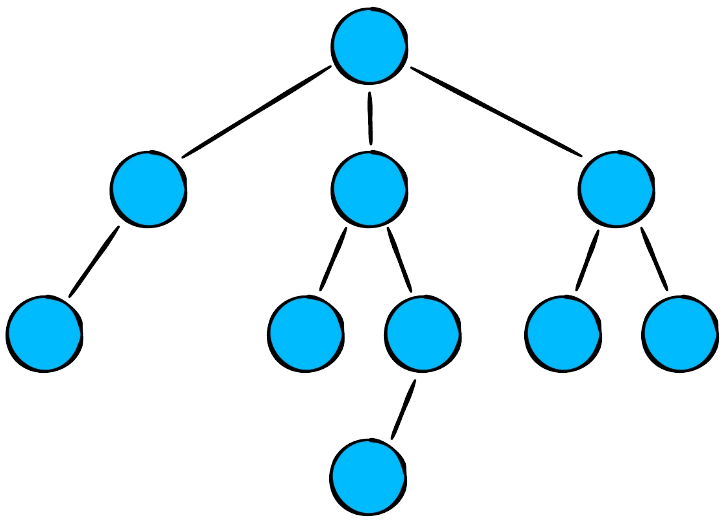
Compare this to a linked list where the elements aren't in contiguous blocks of memory. This could lead to more cache misses which will increase access time.



Where to go from here?

You’ve learned a lot in this chapter! You implemented four varieties of the queue and studied their strengths and weaknesses. You will see queues come up again and again in future chapters as you learn about more sophisticated algorithms.

Trees



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

- representing hierarchical relationships
- managing sorted data
- facilitating fast lookup operations

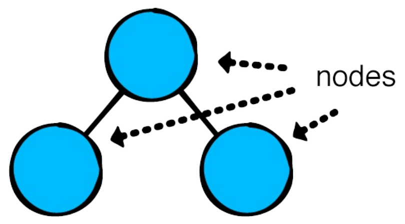
There are many types of trees, and they come in various shapes and sizes. In this chapter, you will learn the basics of using and implementing a tree.

Terminology

There are many terms associated with trees, so you will get acquainted with a couple right off the bat.

Node

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

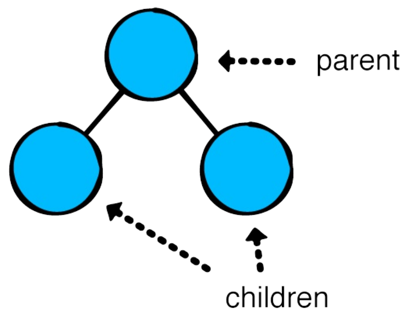


Each node encapsulates some data and keeps track of its children.

Parent and child

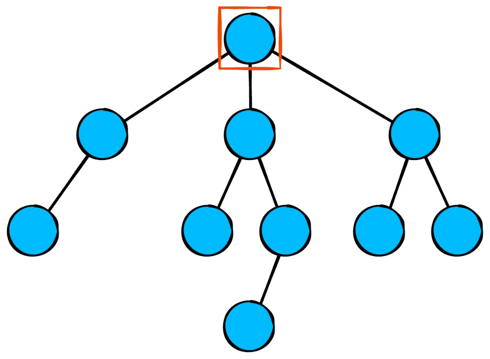
Trees are viewed starting from the top and branching towards the bottom, just like a real tree, only upside-down.

Every node (except for the topmost one) is connected to exactly one node above it. That node is called a parent node. The nodes directly below and connected to it are called its child nodes. In a tree, every child has exactly one parent. That's what makes a tree, a tree.



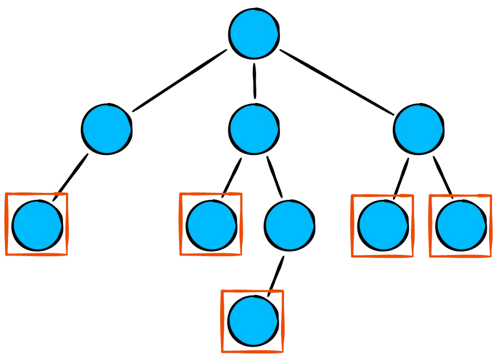
Root

The topmost node in the tree is called the root of the tree. It is the only node that has no parent:



Leaf

A node is a leaf if it has no children:



You will run into more terms later on, but this should be enough to get into the coding of trees.

Implementation

Open up the starter playground for this chapter to get started. A tree is made up of nodes, so your first task is to create a `TreeNode` class.

Create a new file named TreeNode.swift and write the following inside it:

```
public class TreeNode<T> {
  public var value: T
  public var children: [TreeNode] = []
  public init(_ value: T) {
    self.value = value
  }
}
```

Each node is responsible for a `value` and holds references to all its children using an array.

Next, add the following method inside the `TreeNode` class:

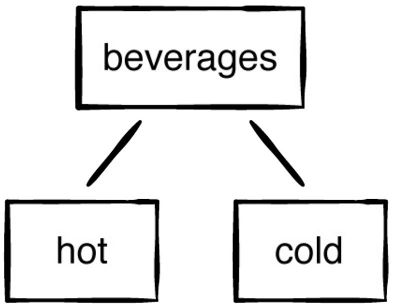
```
public func add(_ child: TreeNode) {
  children.append(child)
}
```

This method adds a child node to a node.

Time to give it a whirl. Head back to the playground page and write the following:

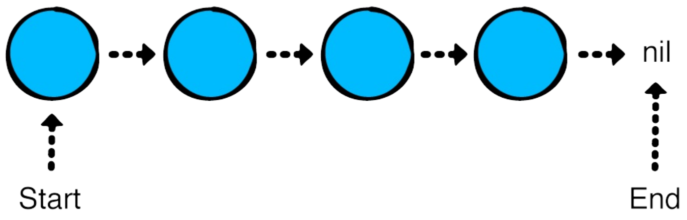
```
example(of: "creating a tree") {
  let beverages = TreeNode("Beverages")

  let hot = TreeNode("Hot")
  let cold = TreeNode("Cold")

  beverages.add(hot)
  beverages.add(cold)
}
```

Hierarchical structures are natural candidates for tree structures, so here you have defined three different nodes and organized them into a logical hierarchy. This arrangement corresponds to the following structure:

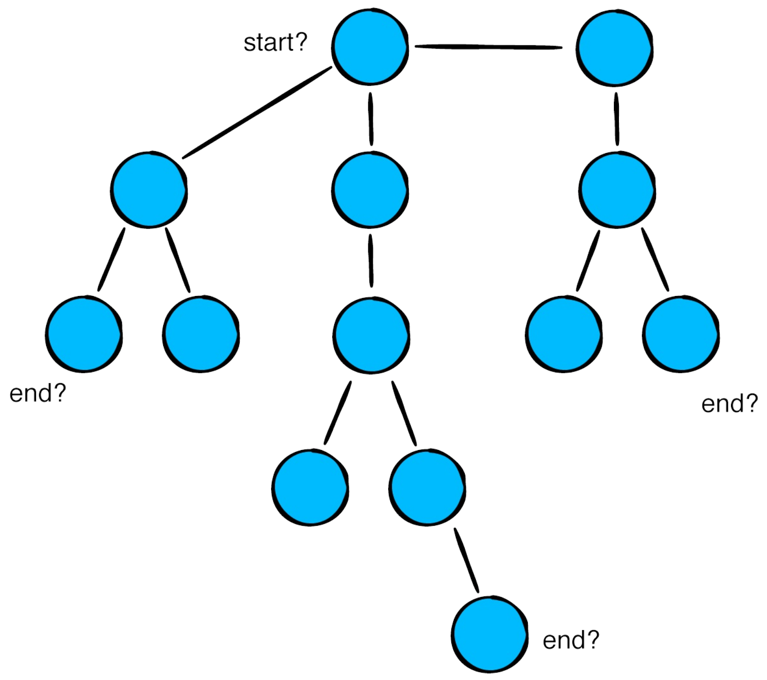


Traversal algorithms

Iterating through linear collections such as arrays or linked lists is straightforward. Linear collections have a clear start and end:



Iterating through trees is a bit more complicated:



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 the next section, you will look at depth-first traversal, a technique that starts at the root and visits nodes as deep as it can before backtracking.

Depth-first traversal

Write the following at the bottom of TreeNode.swift:

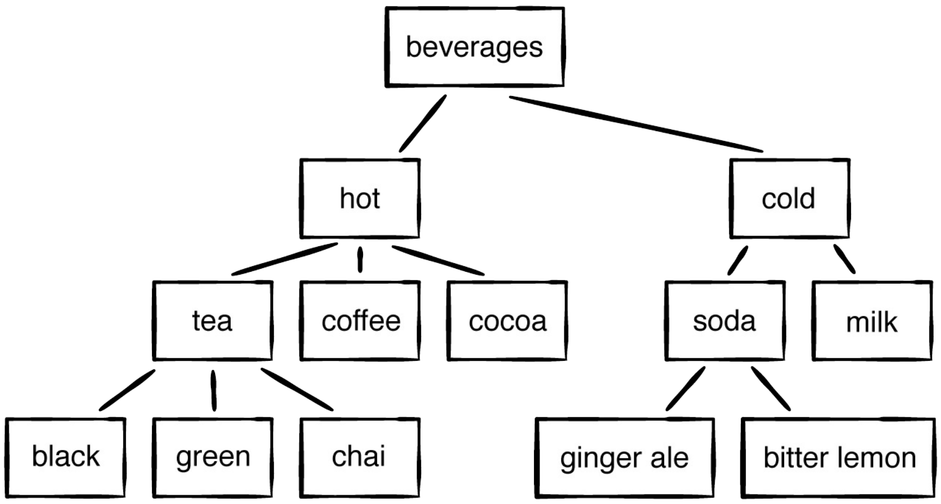
```swift
extension TreeNode {
  public func forEachDepthFirst(visit: (TreeNode) -> Void)
    visit(self)
    children.forEach {
      $0.forEachDepthFirst(visit: visit)
    }
  }
}
```

This simple code uses recursion process the next node. (You could use your own stack if you didn't want your implementation to be recurrsive.) Time to test it out.

Head back to the playground page and write the following:

```
func makeBeverageTree() -> TreeNode<String> {
  let tree = TreeNode("Beverages")

  let hot = TreeNode("hot")
  let cold = TreeNode("cold")

  let tea = TreeNode("tea")
  let coffee = TreeNode("coffee")
  let chocolate = TreeNode("cocoa")

  let blackTea = TreeNode("black")
  let greenTea = TreeNode("green")
  let chaiTea = TreeNode("chai")

  let soda = TreeNode("soda")
  let milk = TreeNode("milk")

  let gingerAle = TreeNode("ginger ale")
  let 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:



Next add this:

```
example(of: "depth-first traversal") {
  let tree = makeBeverageTree()
  tree.forEachDepthFirst { print($0.value) }
}
```

This produces the following depth first output:

```
---Example of: depth-first traversal---
Beverages
hot
tea
black
green
chai
coffee
cocoa
cold
soda
```

```
ginger ale
bitter lemon
milk
```

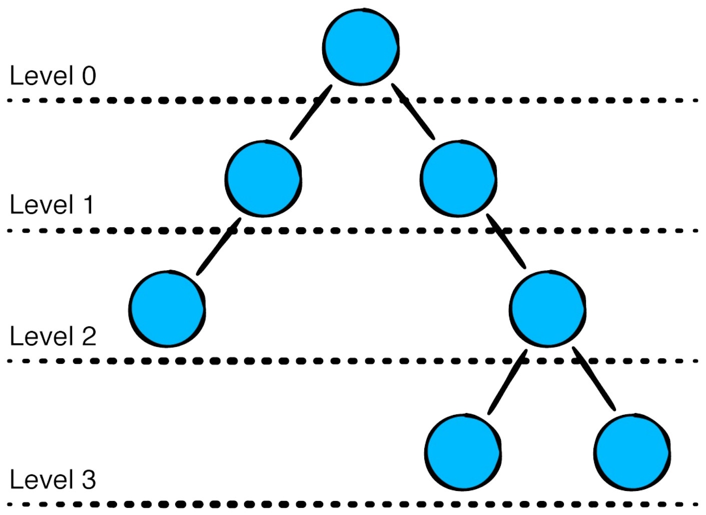
In the next section, you will look at level-order traversal, a technique that visits each node of the tree based on the depth of the nodes.

Level-order traversal

Write the following at the bottom of TreeNode.swift:

```
extension TreeNode {
  public func forEachLevelOrder(visit: (TreeNode) -> Void)
    visit(self)
    var queue = Queue<TreeNode>()
    children.forEach { queue.enqueue($0) }
    while let node = queue.dequeue() {
      visit(node)
      node.children.forEach { queue.enqueue($0) }
    }
  }
}
```

`forEachLevelOrder` visits each of the nodes in level-order:



Note how you used a queue (not a stack) to make sure the nodes are visited in the right level-order. A simple recursion (which implicitly uses a stack) would not have worked!

Head back to the playground page and write the following:

```
example(of: "level-order traversal") {
  let tree = makeBeverageTree()
  tree.forEachLevelOrder { print($0.value) }
}
```

In the console, you will see the following output:

```
---Example of: level-order traversal---
beverages
hot
cold
tea
coffee
cocoa
soda
milk
black
green
chai
ginger ale
bitter lemon
```

Search

You already have a method that iterates through all the nodes, so building a search algorithm shouldn't take long. Write the following at the bottom of TreeNode.swift:

```
extension TreeNode where T: Equatable {
  public func search(_ value: T) -> TreeNode? {
    var result: TreeNode?
    forEachLevelOrder { node in
      if node.value == value {
        result = node
      }
    }
    return result
  }
}
```

Head back to the playground page to test your code. To save some time, simply copy the previous example and modify it to test the `search` method:

```
example(of: "searching for a node") {
  // tree from last example

  if let searchResult1 = tree.search("ginger ale") {
    print("Found node: \(searchResult1.value)")
  }

  if let searchResult2 = tree.search("WKD Blue") {
    print(searchResult2.value)
  } else {
    print("Couldn't find WKD Blue")
  }
}
```

You will see the following console output:

```
---Example of: searching for a node---
Found node: ginger ale
Couldn't find WKD Blue
```

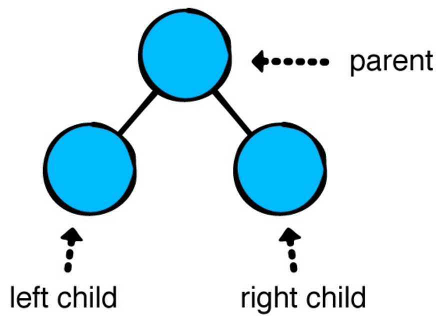
Here you used your level-order traversal algorithm. Since it visits all of the nodes, if there are multiple matches, the last match will win. This means that you will get different objects back depending on what traversal you use.

Where to go from here?

In this chapter you've learned about the general structure of a tree and a way to traverse through its nodes. In the following chapters, you'll learn about specialized trees that solve interesting problems.

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.

Implementation

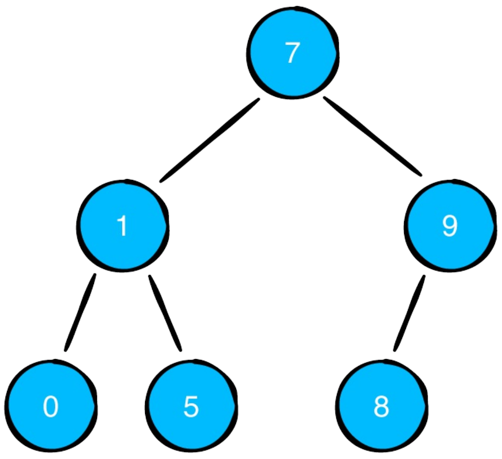
Open the starter project for this chapter. Create a new file and name it BinaryNode.swift. Add the following inside this file:

```swift
public class BinaryNode<Element> {

  public var value: Element
  public var leftChild: BinaryNode?
  public var rightChild: BinaryNode?

  public init(value: Element) {
    self.value = value
  }
}
```

In the main playground page, add the following:

```
var tree: BinaryNode<Int> {
  let zero = BinaryNode(value: 0)
  let one = BinaryNode(value: 1)
  let five = BinaryNode(value: 5)
  let seven = BinaryNode(value: 7)
  let eight = BinaryNode(value: 8)
  let nine = BinaryNode(value: 9)

  seven.leftChild = one
  one.leftChild = zero
  one.rightChild = five
  seven.rightChild = nine
  nine.leftChild = eight

  return seven
}
```

This defines a computed variable that returns the following 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.swift:

```
extension BinaryNode: CustomStringConvertible {

  public var description: String {
    return diagram(for: self)
  }

  private func diagram(for node: BinaryNode?,
                       _ top: String = "",
                       _ root: String = "",
                       _ bottom: String = "") -> String {
    guard let node = node else {
      return root + "nil\n"
    }
    if node.leftChild == nil && node.rightChild == nil {
      return root + "\(node.value)\n"
    }
    return diagram(for: node.rightChild,
                   top + "  ", top + "┌──", top + "│ ")
         + root + "\(node.value)\n"
         + diagram(for: node.leftChild,
                   bottom + "│ ", bottom + "└──", bottom + "
  }
}
```

`diagram` will recursively create a string representing the binary tree. To try it out, head back to the playground and write the following:

```
example(of: "tree diagram") {
  print(tree)
}
```

You should see the following console output:

```
---Example of tree diagram---
  ┌──nil
 ┌──9
 │ └──8
7
 │ ┌──5
 └──1
  └──0
```

You’ll be using this diagram for other binary trees in this book.

Traversal algorithms

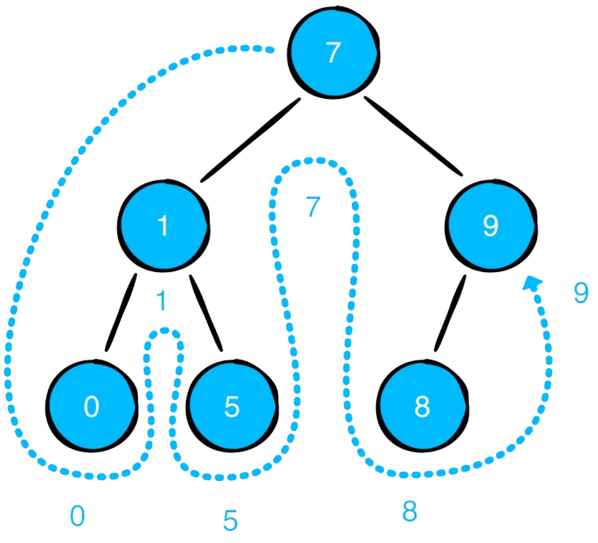
Previously, you looked at a level-order traversal of a tree. With a few tweaks, you can make this algorithm work for binary trees as well. However, instead of re-implementing level-order traversal, you’ll look at three traversal algorithms for binary trees: in-order, pre-order, and post-order traversals.

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 up BinaryNode.swift and add the following code to the bottom of the file:

```
extension BinaryNode {

  public func traverseInOrder(visit: (Element) -> Void) {
    leftChild?.traverseInOrder(visit: visit)
    visit(value)
    rightChild?.traverseInOrder(visit: visit)
  }
}
```

Following the rules laid out above, you first traverse to the leftmost node before visiting the value. You then traverse to the rightmost node.

Head back to the playground page to test this out. Add the following at the bottom of the page:

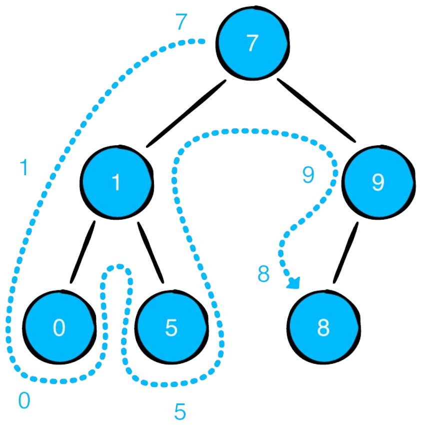
```
example(of: "in-order traversal") {
  tree.traverseInOrder { print($0) }
}
```

You should see the following in the console:

```
---Example of in-order traversal---
0
1
5
7
8
9
```

Pre-order traversal

Pre-order traversal always visits the current node first, then recursively visits the left and right child:



Write the following just below your in-order traversal method:

```
public func traversePreOrder(visit: (Element) -> Void) {
  visit(value)
  leftChild?.traversePreOrder(visit: visit)
  rightChild?.traversePreOrder(visit: visit)
}
```

Test it out with the following code:

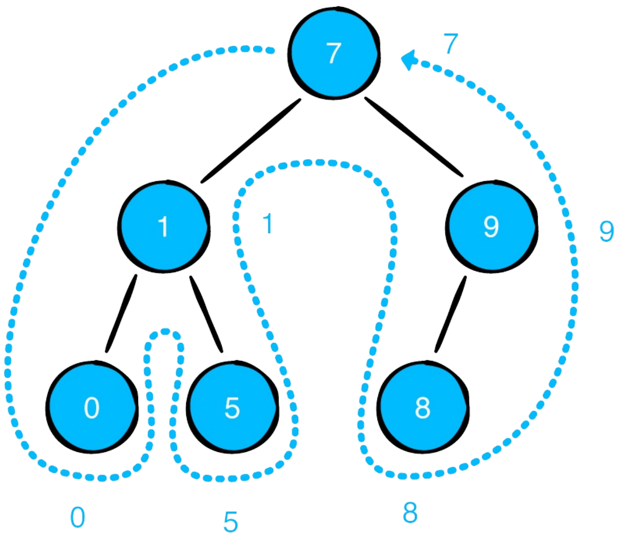
```
example(of: "pre-order traversal") {
  tree.traversePreOrder { print($0) }
}
```

You should see the following output in the console:

```
---Example of pre-order traversal---
7
1
0
5
9
8
```

Post-order traversal

Post-order traversal only visits the current node after the left and right child have been visited recursively.



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.

Back inside BinaryNode.swift, write the following below `traversePreOrder`:

```
public func traversePostOrder(visit: (Element) -> Void) {
  leftChild?.traversePostOrder(visit: visit)
  rightChild?.traversePostOrder(visit: visit)
  visit(value)
}
```

Navigate back to the playground page to try it out:

```
example(of: "post-order traversal") {
  tree.traversePostOrder { print($0) }
}
```

You should see the following in the console:

```
---Example of post-order traversal---
0
5
1
8
9
7
```

Where to go from here?

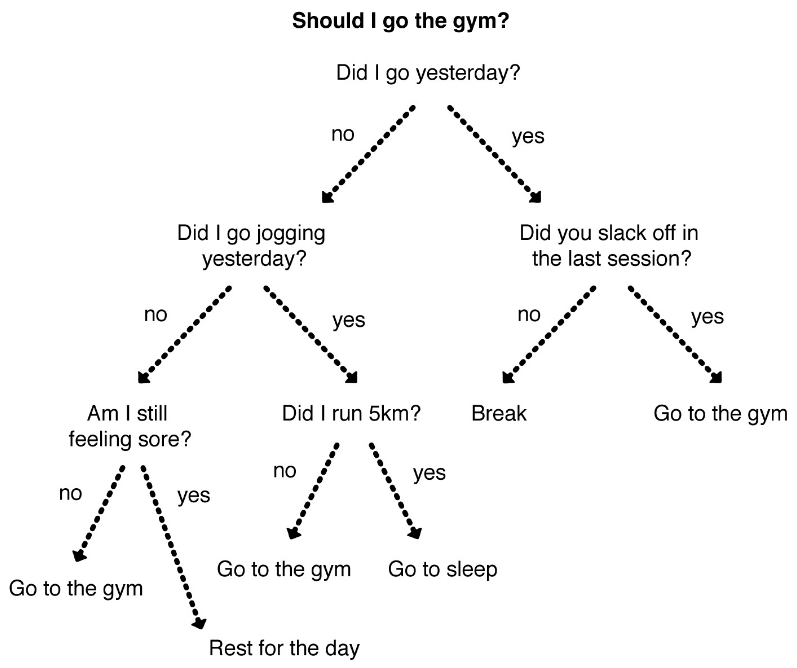
Each one of these traversal algorithms has both a time and space complexity of O(n). While this version of the binary tree isn't too interesting, you saw that in-order traversal can be used to visit the nodes in ascending order. Binary trees can enforce this behavior by adhering to some rules during insertion. In the next chapter, you'll look at a binary tree with stricter semantics: the binary search tree.

Binary Search Trees

A binary search tree (or BST) is a data structure that 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. A binary search tree achieves this performance by imposing 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.
- The value of a right child must be greater than or equal to the value of its parent.

These rules let the tree act like a decision tree:

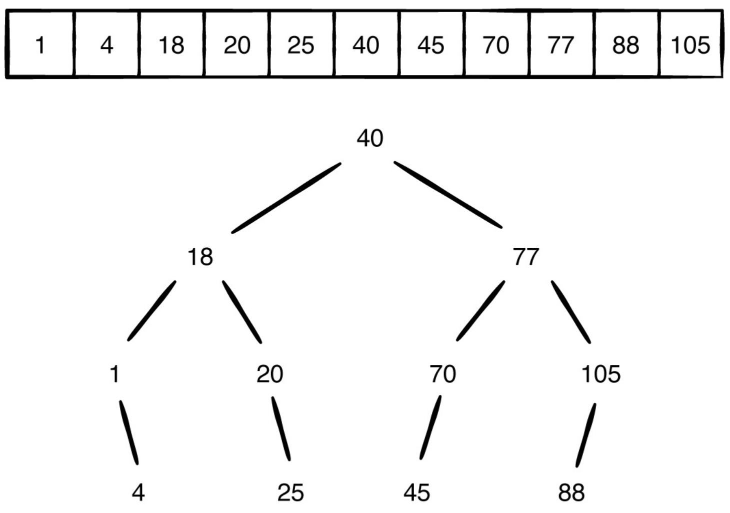


Picking a side forfeits all the possibilities of the other side. Binary search trees use this property to save you from performing unnecessary checking.

Case study: array vs. BST

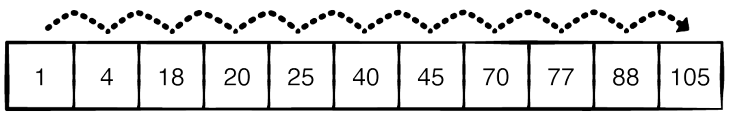
To illustrate the power of binary search trees, you'll look at some common operations and compare the performance of arrays against the binary search tree.

Consider the following two collections:



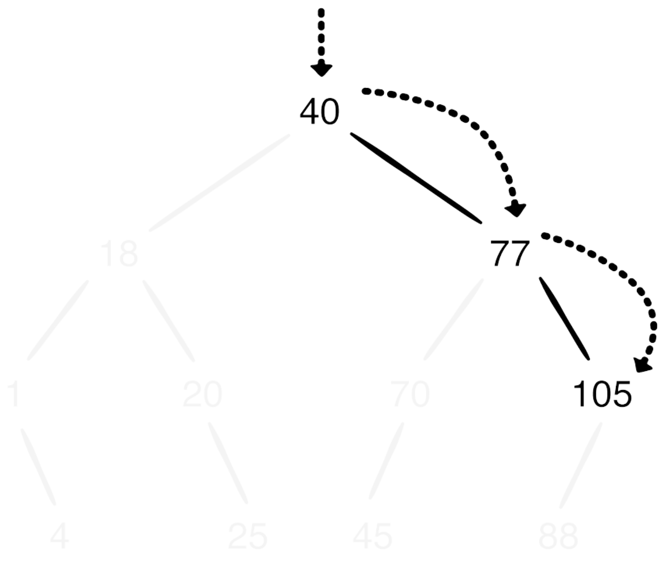
Lookup

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:



That's why `array.contains(:)` is an O(n) operation.

This is not the case for binary search trees:



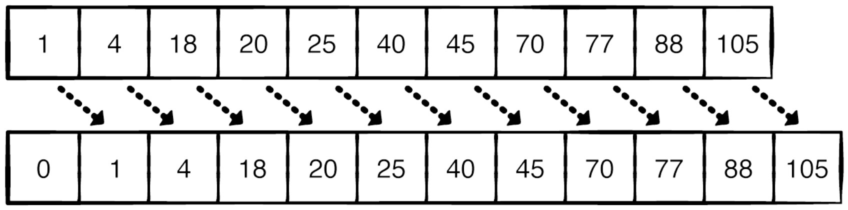
Every time the search algorithm visits a node in the BST, it can safely make these two assumptions:

- If the search value is less than the current value, it must be in the left subtree.
- If the search value 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

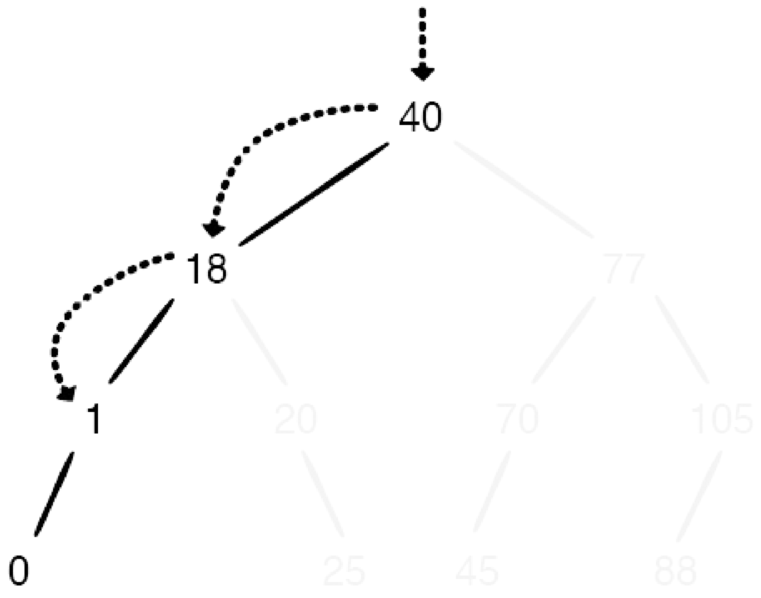
The performance benefits for the insertion operation follow a similar story. Assume you want to insert 0 into a collection:



Inserting 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 in front of the array, causing all other elements to shift backwards by one position. Inserting into an array has a time complexity of O(n).

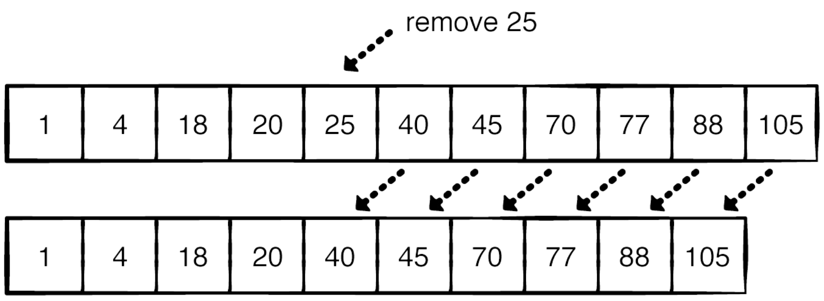
Insertion into a binary search tree is much more comforting:



By leveraging the rules for the BST, you only needed to make three traversals to find the location for the insertion, and you didn't have to shuffle all the elements around! Inserting elements in a BST is again an O(log n) operation.

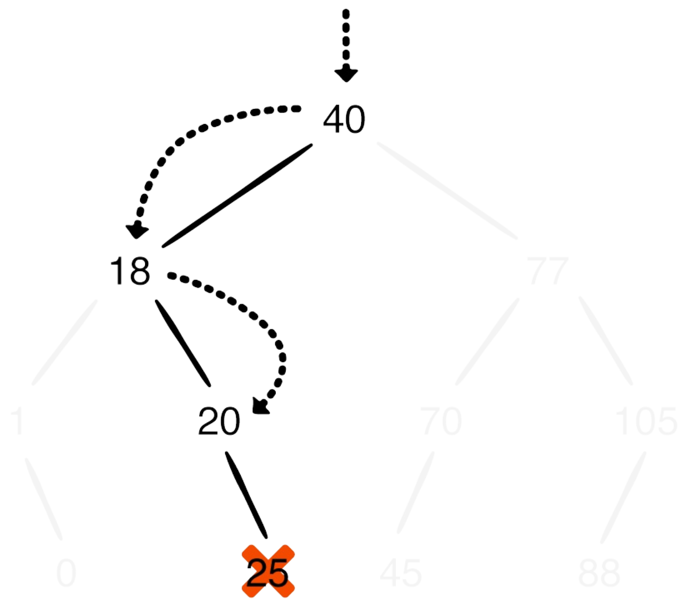
Removal

Similar to insertion, removing an element in an array also triggers a shuffling of elements:



This behavior also plays nicely with the lineup 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 up the starter project for this chapter. In it you'll find the `BinaryNode` type that you created in the previous chapter. Create a new file named BinarySearchTree.swift and add the following inside the file:

```swift
public struct BinarySearchTree<Element: Comparable> {

  public private(set) var root: BinaryNode<Element>?

  public init() {}
}

extension BinarySearchTree: CustomStringConvertible {

  public var description: String {
    return root?.description ?? "empty tree"
  }
}
```

By definition, binary search trees can only hold values that are `Comparable`.

Next, you'll look at the `insert` method.

Inserting elements

In accordance with the rules of the BST, nodes of the left child must contain values less than the current node. Nodes of the right child must contain values greater than or equal to the current node. You'll implement the `insert` method while respecting these rules.

Add the following to BinarySearchTree.swift:

```
extension BinarySearchTree {

  public mutating func insert(_ value: Element) {
    root = insert(from: root, value: value)
  }

  private func insert(from node: BinaryNode<Element>?, value
      -> BinaryNode<Element> {
    // 1
    guard let node = node else {
      return BinaryNode(value: value)
    }
    // 2
    if value < node.value {
      node.leftChild = insert(from: node.leftChild, value: 
    } else {
      node.rightChild = insert(from: node.rightChild, value
    }
    // 3
    return node
  }
}
```

The first `insert` method is exposed to users, while the second one 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 `nil`, you've found the insertion point and you return the new `BinaryNode`.

1. 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'll call `insert` on the right child.

2. Return the current node. This makes assignments of the form `node = insert(from: node, value: value)` possible as `insert` will either create `node` (if it was `nil`) or return `node` (it it was not `nil`).

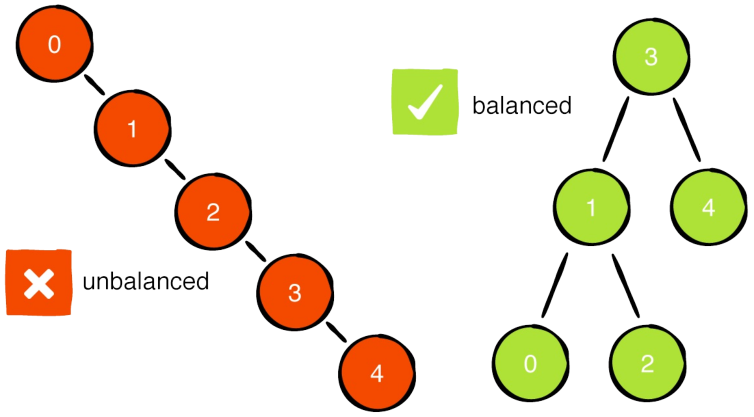
Head back to the playground page and add the following at the bottom:

```
example(of: "building a BST") {
  var bst = BinarySearchTree<Int>()
  for i in 0..<5 {
    bst.insert(i)
  }
  print(bst)
}
```

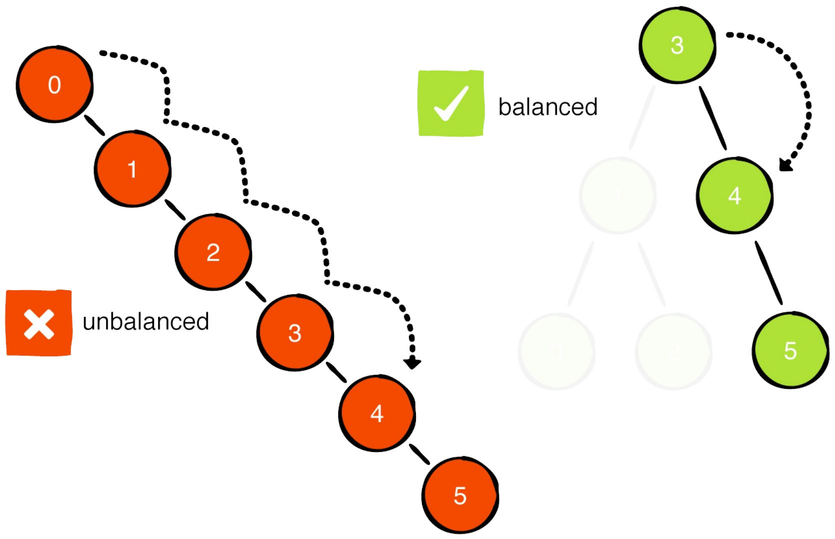
You should see the following output:

```
---Example of: building a BST---
    ┌──4
   ┌──3
   │ └──nil
  ┌──2
  │ └──nil
 ┌──1
 │ └──nil
0
└──nil
```

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've 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 computed variable at the top of the playground page:

```
var exampleTree: BinarySearchTree<Int> {
  var bst = BinarySearchTree<Int>()
  bst.insert(3)
  bst.insert(1)
  bst.insert(4)
  bst.insert(0)
  bst.insert(2)
  bst.insert(5)
  return bst
}
```

Replace your `example` function with the following:

```
example(of: "building a BST") {
  print(exampleTree)
}
```

You should see the following in the console:

```
---Example of: building a BST---
 ┌──5
┌──4
│ └──nil
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 you learned about in the previous chapter.

Add the following to the bottom of BinarySearchTree.swift:

```
extension BinarySearchTree {
  public func contains(_ value: Element) -> Bool {
    guard let root = root else {
      return false
    }
    var found = false
    root.traverseInOrder {
      if $0 == value {
        found = true
      }
    }
    return found
  }
}
```

Next, head back to the playground page to test this out:

```
example(of: "finding a node") {
  if exampleTree.contains(5) {
    print("Found 5!")
  } else {
    print("Couldn't find 5")
  }
}
```

You should 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. Back in BinarySearchTree.swift, update the `contains` method to the following:

```
public func contains(_ value: Element) -> Bool {
  // 1
  var current = root
  // 2
  while let node = current {
    // 3
    if node.value == value {
      return true
    }
    // 4
    if value < node.value {
      current = node.leftChild
    } else {
      current = node.rightChild
    }
  }
  return false
}
```

Here’s how this works:

1. Start off by setting `current` to the `root` node.
2. While `current` is not `nil`, 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 the right child.

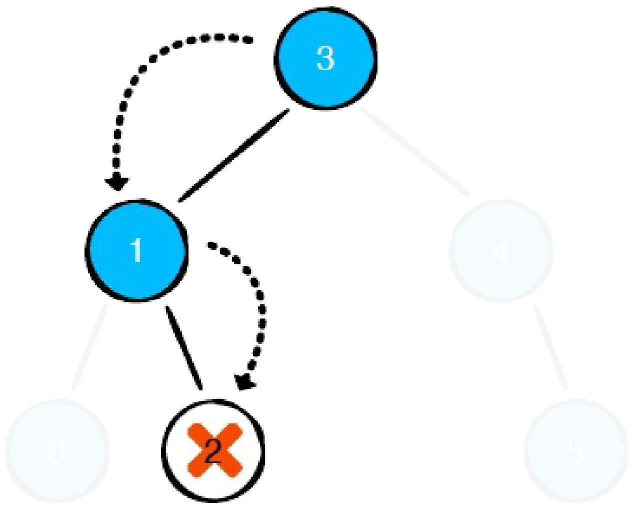
In a balanced binary search tree, this implementation of `contains` is an O(log n) operation.

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:

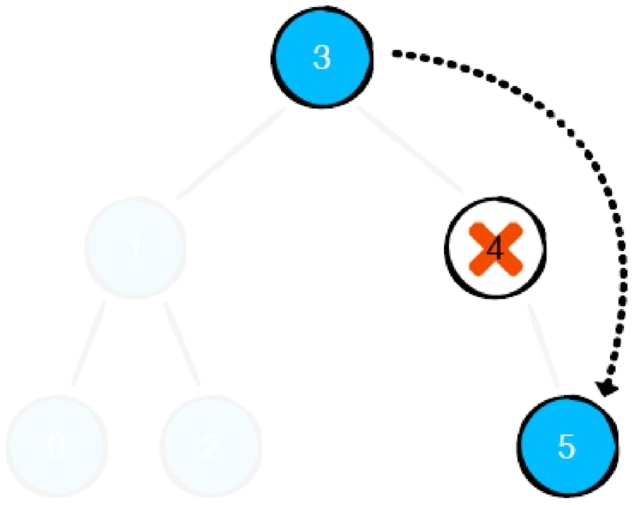


Simply detaching the leaf node is enough.

For non-leaf nodes however, there are extra steps to be taken.

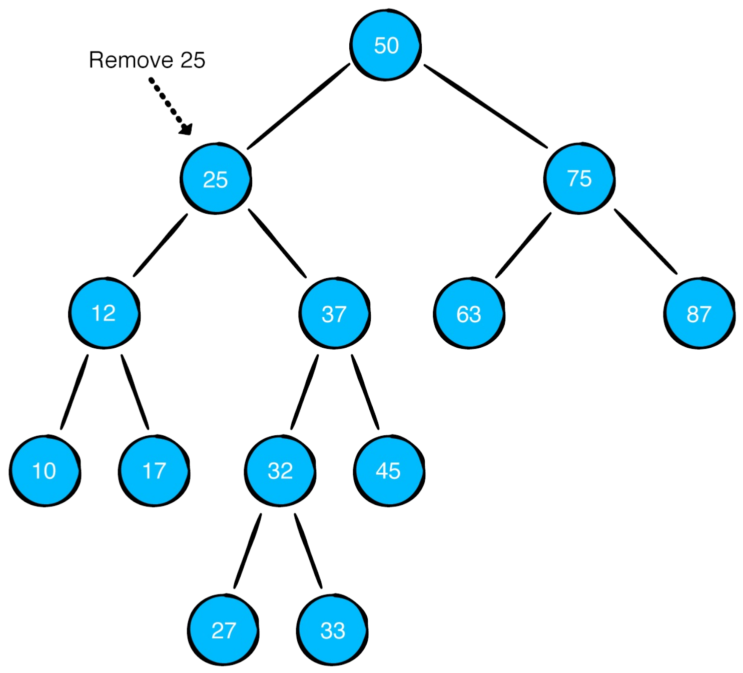
Case 2: Nodes with one child

When removing nodes with one child, you'll need to reconnect that one child with the rest of the tree:

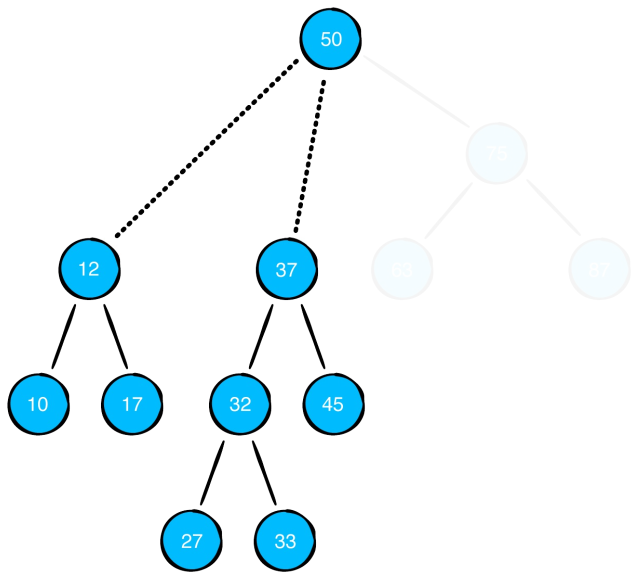


Case 3: Nodes with two children

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 you have the following tree and you want to remove the value 25:

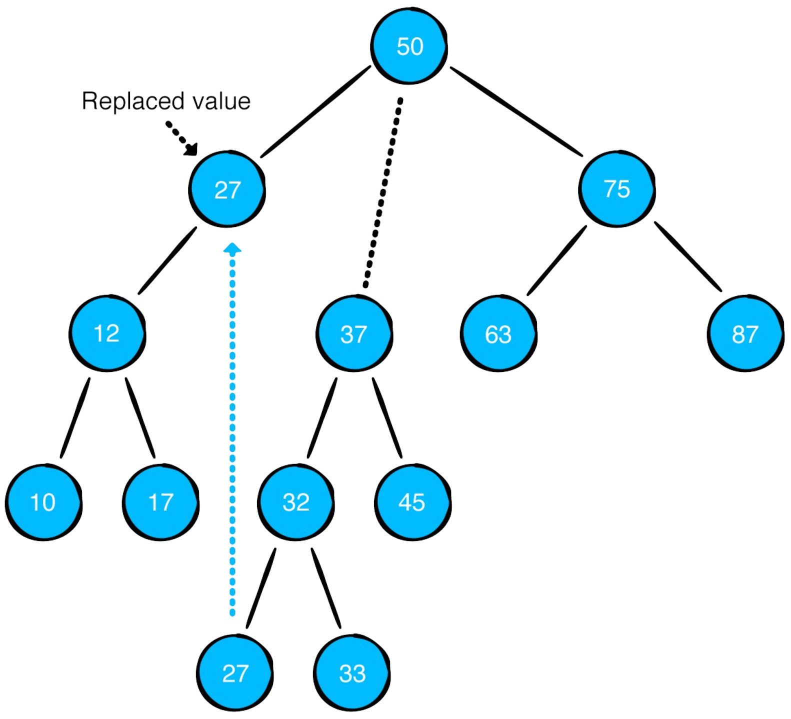


Simply deleting the node presents a dilemma:



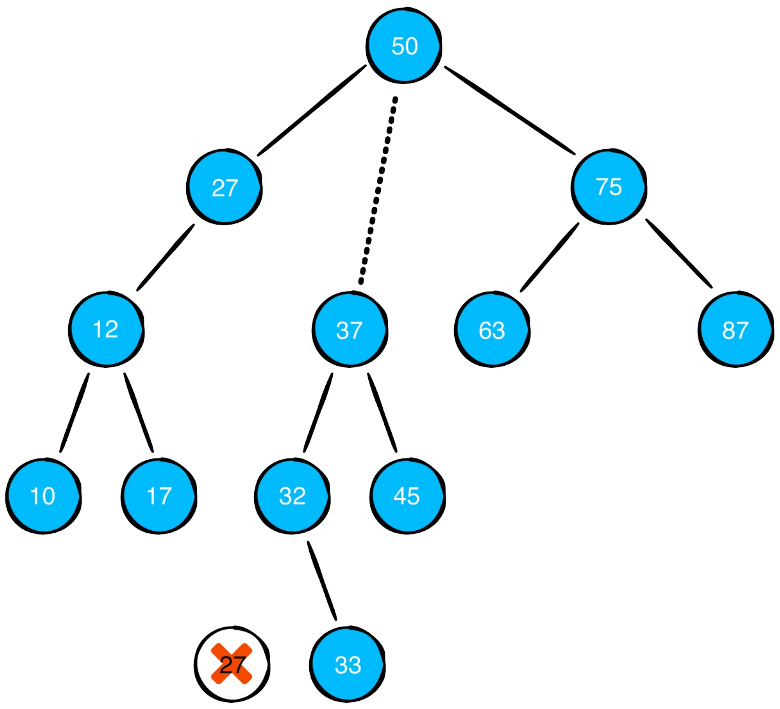
You have two child nodes (12 and 37) to reconnect, but the parent node only has space for one child. To solve this problem, you'll implement a clever workaround by performing a swap.

When removing a node with two children, replace the node you removed with 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 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 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

Open up BinarySearchTree.swift to implementing `remove`. Add the following code at the bottom of the file:

```
private extension BinaryNode {

  var min: BinaryNode {
    return leftChild?.min ?? self
  }
}

extension BinarySearchTree {

  public mutating func remove(_ value: Element) {
    root = remove(node: root, value: value)
  }

  private func remove(node: BinaryNode<Element>?, value: Ele
    -> BinaryNode<Element>? {
    guard let node = node else {
      return nil
    }
    if value == node.value {
      // more to come
    } else if value < node.value {
      node.leftChild = remove(node: node.leftChild, value: v
    } else {
      node.rightChild = remove(node: 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`. You've also added a recursive `min` property to `BinaryNode` to find the minimum node in a subtree.

The different removal cases are handled in the `if value == node.value` clause:

```
// 1
if node.leftChild == nil && node.rightChild == nil {
  return nil
}
// 2
if node.leftChild == nil {
  return node.rightChild
}
// 3
if node.rightChild == nil {
  return node.leftChild
}
// 4
node.value = node.rightChild!.min.value
node.rightChild = remove(node: node.rightChild, value: node
```

1. In the case where the node is a leaf node, you simply return `nil`, 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 where 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.

Head back to the playground page and test `remove` by writing the following:

```
example(of: "removing a node") {
  var tree = exampleTree
  print("Tree before removal:")
  print(tree)
  tree.remove(3)
  print("Tree after removing root:")
  print(tree)
}
```

You should see the following output in the console:

```
---Example of: removing a node---
Tree before removal:
  ┌──5
┌──4
│ └──nil
3
│ ┌──2
└──1
  └──0

Tree after removing root:
 ┌──5
4
│ ┌──2
└──1
 └──0
```

Where to go from here?

The BST is a powerful data structure that can delivers great performance when managing sorted data. In this chapter, you learned about the `insert`, `remove`, and `contains` methods of the binary search tree. You also learned about its Achilles' heel: the performance of operations on a BST can degrade to O(n) if the tree becomes unbalanced.

In the next chapter, you'll learn about a self-balancing binary search tree: the AVL tree.

AVL Trees

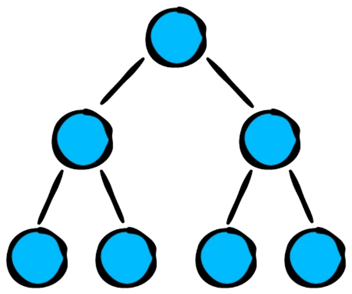
In the previous chapter, you learned about the O(log n) performance characteristics of the binary search tree. However, you also learned that unbalanced trees can deteriorate the performance of the tree, all the way down to O(n). In 1962, Georgy Adelson-Velsky and Evgenii Landis came up with the first self-balancing binary search tree: the AVL Tree.

Understanding balance

A balanced tree is the key to optimizing the performance of the binary search tree. In this section, you'll learn about the three main states of balance.

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.

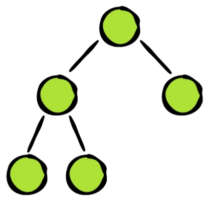


Not only is the tree perfectly symmetrical, the nodes at the bottom level are completely filled. This is the requirement for being perfectly balanced.

"Good-enough" balance

Although achieving perfect balance is ideal, it is rarely possible. A perfectly balanced tree has to contain the exact number of nodes to fill every level to the bottom, so it can only be perfect with a particular number of elements.

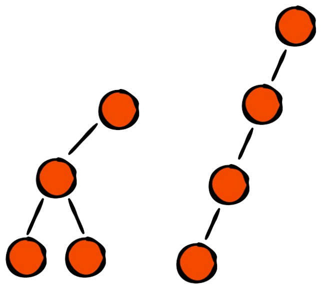
As an example, a tree with 1, 3, or 7 nodes can be perfectly balanced, but a tree with 2, 4, 5, or 6 cannot be perfectly balanced, since the last level of the tree will not be filled.



The definition of a balanced tree is that every level of the tree must be filled, except for the bottom level. 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.



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.

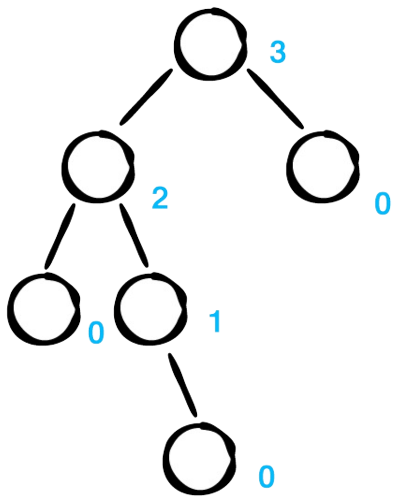
Implementation

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 be adding is the balancing component. Open up the starter project to begin.

Measuring balance

To keep a binary tree balanced, you'll 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:



Open the starter playground for this chapter and add the following property to `AVLNode` in the compiled sources folder:

```
public var height = 0
```

You'll use the relative heights of a node's children to determine whether a particular node is balanced.

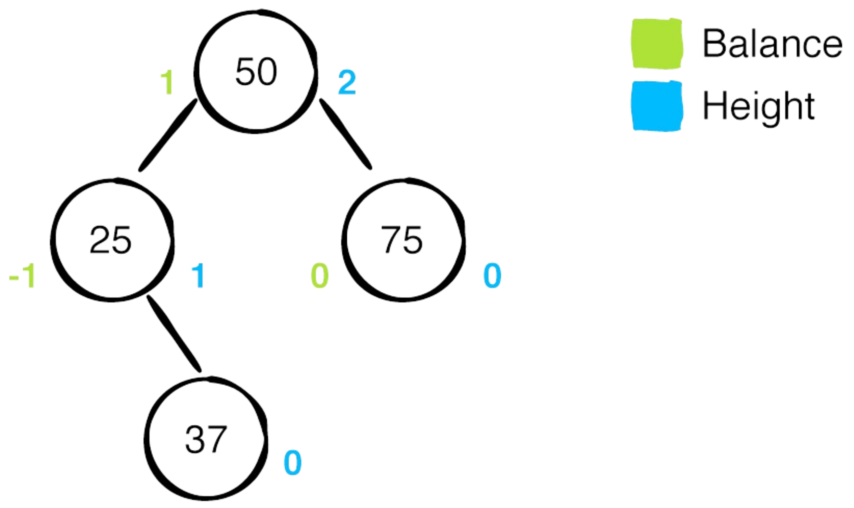
The height of the left and right children of each node must differ by at most 1. This is known as the balance factor.

Write the following just below the `height` property of `AVLNode`:

```
public var balanceFactor: Int {
  return leftHeight - rightHeight
}

public var leftHeight: Int {
  return leftChild?.height ?? -1
}

public var rightHeight: Int {
  return rightChild?.height ?? -1
}
```

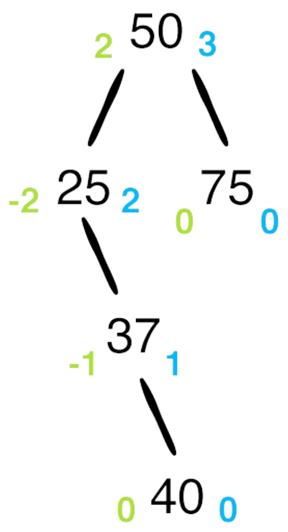
The `balanceFactor` computes the height difference of the left and right child. If a particular child is `nil`, its height is considered to be `-1`.

Here's an example of an AVL tree:



This is a balanced tree, where 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:



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.

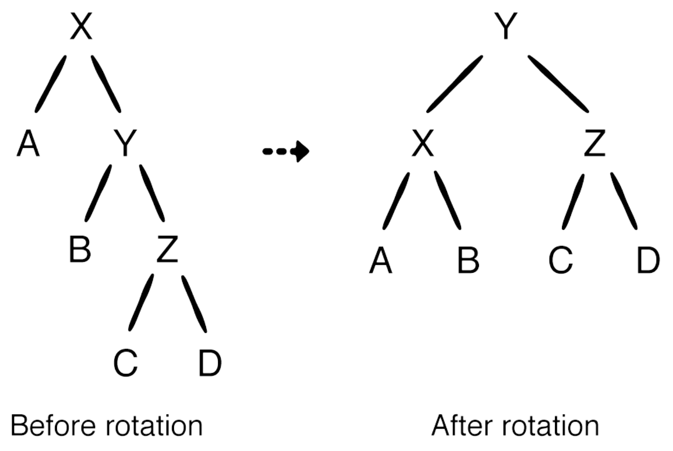
That's where rotations come in.

Rotations

The procedures used to balance a binary search tree are known as rotations. There are four rotations in total, for the four different ways that a tree can become unbalanced. These are known as left rotation, left-right rotation, right rotation, and right-left rotation.

Left rotation

The imbalance caused by inserting 40 into the tree can be solved by a left rotation. A generic left rotation of node x looks like this:



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 1 level after the rotation.

Add the following method to AVLTree, just below `insert(from:value:)`:

```
private func leftRotate(_ node: AVLNode<Element>) -> AVLNod
  // 1
  let 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) +
  // 5
  return pivot
}
```

Here are the steps needed to perform a left rotation:

1. The right child is chosen as the pivot. This node will replace the rotated node as the root of the subtree (it will move up a level).

1. The node to be rotated will become the left child of the pivot (it moves down a level). This means the current left child of the pivot must be moved elsewhere.

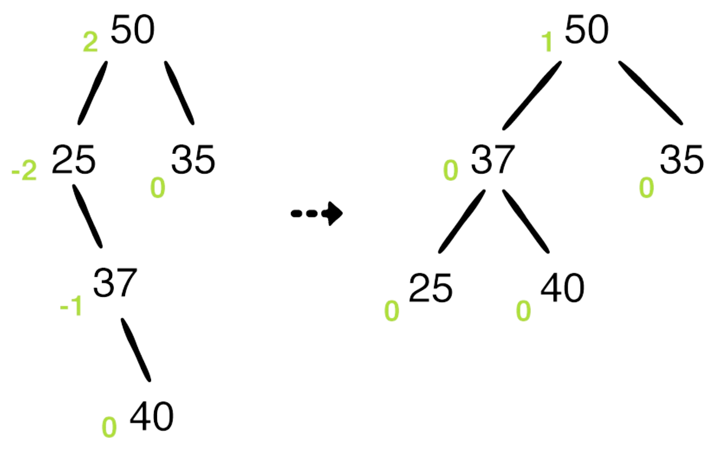
   In the generic example shown in the earlier image, this is node b. Because b is smaller than y but greater than x, it can replace y as the right child of x. So you update the rotated node's `rightChild` to the pivot's `leftChild`.

2. The pivot's `leftChild` can now be set to the rotated node.

3. You update the heights of the rotated node and the pivot.

4. Finally, you return the pivot so it can replace the rotated node in the tree.

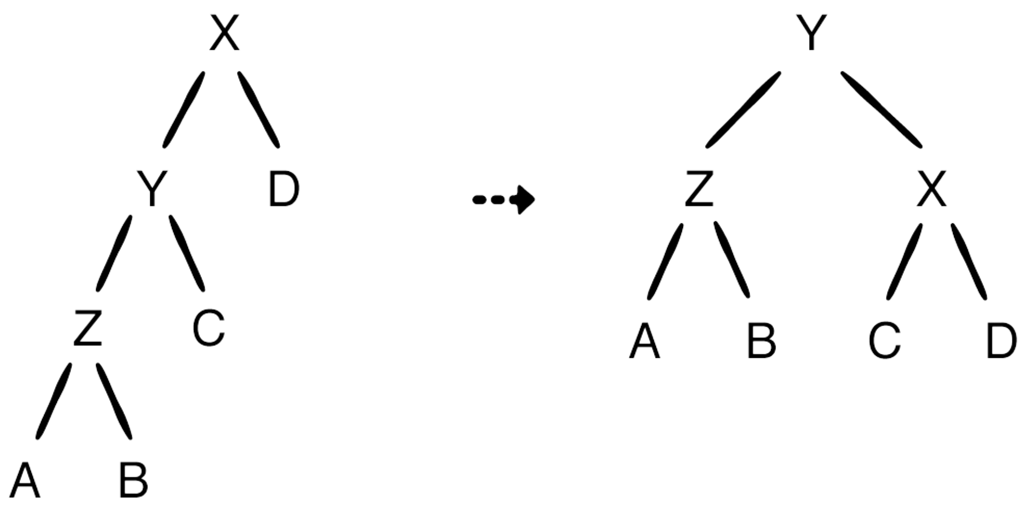
Here are the before and after effects of the left rotation of 25 from the previous example:



Before left rotate on 25 After left rotate on 25

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:



Before right rotate of x

Before right rotate of x

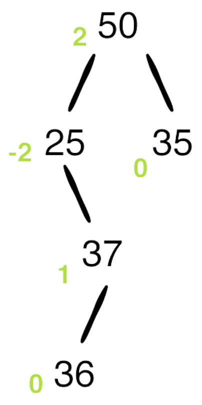
To implement this, add the following code just after `leftRotate`:

```
private func rightRotate(_ node: AVLNode<Element>) -> AVLNo
  let 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) +
  return pivot
}
```

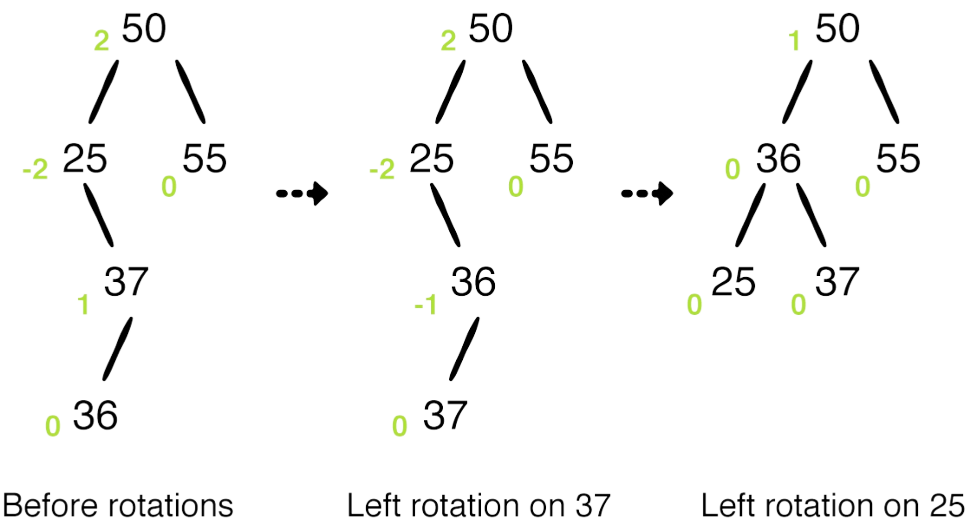
This is nearly identical to the implementation of `leftRotate`, except the references to the left and right children have been swapped.

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 where 36 is inserted into the original example tree.



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:



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.

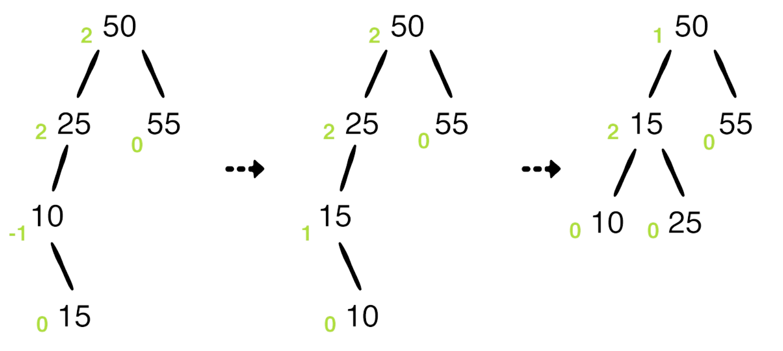
Add the following code just after `rightRotate`:

```
private func rightLeftRotate(_ node: AVLNode<Element>) -> A
  guard let rightChild = node.rightChild else {
    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 final case, left-right rotation.

Left-right rotation

Left-right rotation is the symmetrical opposite of the right-left rotation. Here's an example:



Before rotations Left rotate of 10 Right rotate of 25

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 just after `rightLeftRotate`:

```
private func leftRightRotate(_ node: AVLNode<Element>) -> A
  guard let leftChild = node.leftChild else {
    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.

Balance

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 func balanced(_ node: AVLNode<Element>) -> AVLNode<
  switch node.balanceFactor {
  case 2:
    // ...
  case -2:
    // ...
  default:
    return node
  }
}
```

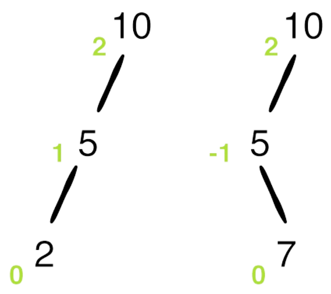
There are three cases to consider.

1. A `balanceFactor` of 2 suggests that the left child is “heavier” (that is, contains more nodes) than the right child. This means 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 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.

The sign of the balanceFactor can be used to determine if a single or double rotation is required:



Update the balanced function to the following:

```
private func balanced(_ node: AVLNode<Element>) -> AVLNode<
  switch node.balanceFactor {
  case 2:
    if let leftChild = node.leftChild, leftChild.balanceFac
      return leftRightRotate(node)
    } else {
      return rightRotate(node)
    }
  case -2:
    if let rightChild = node.rightChild, rightChild.balance
      return rightLeftRotate(node)
    } else {
      return leftRotate(node)
    }
  default:
    return node
  }
}
```

balanced inspects the balanceFactor to determine the proper course of action. All that's left is to call balance at the proper spot.

Revisiting insertion

You’ve already done the majority of the work. The remainder is fairly straightforward. Update `insert(from:value:)` to the following:

```
private func insert(from node: AVLNode<Element>?, value: El
  guard let node = node else {
    return AVLNode(value: value)
  }
  if value < node.value {
    node.leftChild = insert(from: node.leftChild, value: va
  } else {
    node.rightChild = insert(from: node.rightChild, value: 
  }
  let balancedNode = balanced(node)
  balancedNode.height = max(balancedNode.leftHeight, balanc
  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.

That’s all there is to it! Head into the playground page and test it out. Add the following to the playground:

```
example(of: "repeated insertions in sequence") {
  var tree = AVLTree<Int>()
  for i in 0..<15 {
    tree.insert(i)
  }
  print(tree)
}
```

You should 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 link 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:

```
let balancedNode = balanced(node)
balancedNode.height = max(balancedNode.leftHeight, balancedN
return balancedNode
```

Head back to the playground page and add the following code at the bottom of the file:

```
example(of: "removing a value") {
  var tree = AVLTree<Int>()
  tree.insert(15)
  tree.insert(10)
  tree.insert(16)
  tree.insert(18)
  print(tree)
  tree.remove(10)
  print(tree)
}
```

You should see the following console output:

```
---Example of: removing a value---
  ┌──18
 ┌──16
 │ └──nil
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.

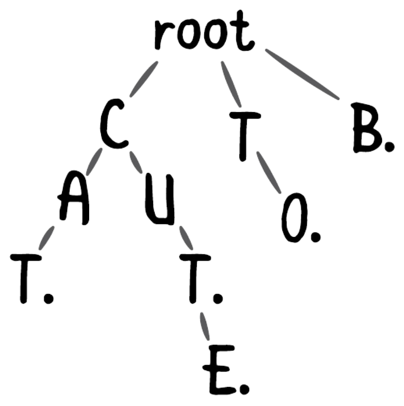
Where to go from here?

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.

While AVL trees were the first self-balancing implementations of a BST, others, such as the red-black tree and splay tree, have since joined the party. If you're interested, you check those out in the Swift Algorithm Club. Find them at at: https://github.com/raywenderlich/swift-algorithm-club/tree/master/Red-Black%20Tree and https://github.com/raywenderlich/swift-algorithm-club/tree/master/Splay%20Tree respectively.

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:



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.

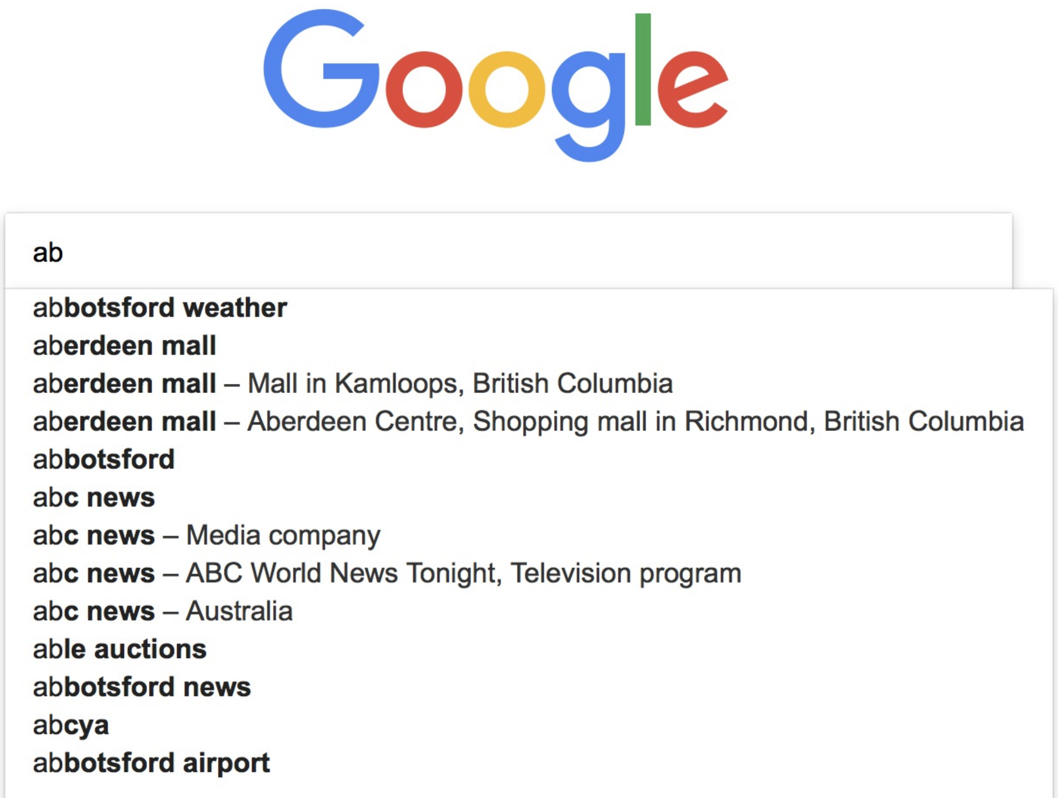
Example

You are given a collection of strings. How would you build a component that handles prefix matching? Here's one way:

```
class EnglishDictionary {

  private var words: [String]

  func words(matching prefix: String) -> [String] {
    return words.filter { $0.hasPrefix(prefix) }
  }
}
```

`words(matching:)` will go through the collection of strings and return 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(matching:)` is O(k*n), where k is the longest string in the collection, and n is the number of words you need to check.


ab

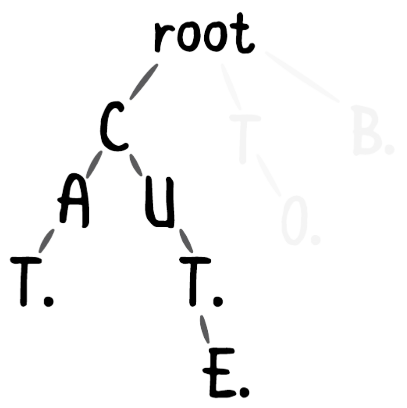
abbotsford weather
aberdeen mall
aberdeen mall – Mall in Kamloops, British Columbia
aberdeen mall – Aberdeen Centre, Shopping mall in Richmond, British Columbia
abbotsford
abc news
abc news – Media company
abc news – ABC World News Tonight, Television program
abc news – Australia
able auctions
abbotsford news
abcya
abbotsford airport

The trie data structure has excellent performance characteristics for this type of problem; as a tree with nodes that support multiple children, each node can represent a single character.

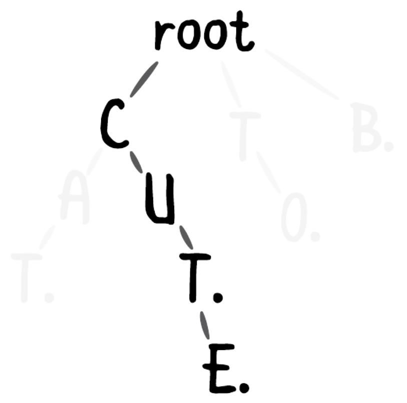
You form a word by tracing the collection of characters from the root to a node with a special indicator — a terminator — represented by a black dot. An interesting characteristic of the trie is that multiple words can share the same characters.

To illustrate the performance benefits of the trie, consider the following example where you need to find the words with the prefix `CU`.

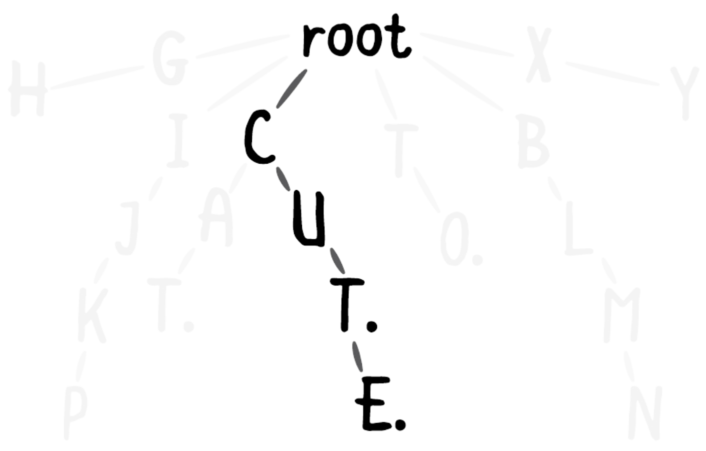
First you travel to the node containing `C`. That quickly excludes other branches of the trie from the search operation:



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 would return all collections formed by the chain of nodes from the `U` node. In this case, the words `CUT` and `CUTE` would be 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

As always, open up the starter playground for this chapter.

TrieNode

You'll begin by creating the node for the trie. In the Sources directory, create a new file named TrieNode.swift. Add the following to the file:

```
public class TrieNode<Key: Hashable> {

  // 1
  public var key: Key?

  // 2
  public weak var parent: TrieNode?

  // 3
  public var children: [Key: TrieNode] = [:]

  // 4
  public var isTerminating = false
  public init(key: Key?, parent: TrieNode?) {
    self.key = key
    self.parent = parent
  }
}
```

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 weak reference to its parent. This reference simplifies the `remove` method 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` dictionary 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. In the Sources folder, create a new file named Trie.swift. Add the following to the file:

```
public class Trie<CollectionType: Collection>
    where CollectionType.Element: Hashable {

  public typealias Node = TrieNode<CollectionType.Element>

  private let root = Node(key: nil, parent: nil)

  public init() {}
}
```

The `Trie` class is built for all types that adopt the `Collection` protocol, including `String`. In addition to this requirement, each element inside the collection must be `Hashable`. This is required because you'll be using the collection's elements as keys for the `children` dictionary in `TrieNode`.

Next, you'll implement four operations for the trie: `insert`, `contains`, `remove` and a prefix match.

Insert

Tries work with any type that conforms to `Collection`. The trie will take the collection and represent it as a series of nodes, where each node maps to an element in the collection.

Add the following method to `Trie`:

```
public func insert(_ collection: CollectionType) {
  // 1
  var current = root

  // 2
  for element in collection {
    if current.children[element] == nil {
      current.children[element] = Node(key: element, parent
    }
    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 collection in separate nodes. For each element of the collection, you first check if the node currently exists in the `children` dictionary. 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 collection. 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 collection you're trying to insert. This is because you need to traverse through or create each node that represents each element of the new collection.

Contains

`contains` is very similar to `insert`. Add the following method to `Trie`:

```
public func contains(_ collection: CollectionType) -> Bool
  var current = root
  for element in collection {
    guard let child = current.children[element] else {
      return false
    }
    current = child
  }
  return current.isTerminating
}
```

Here you traverse the trie in a way similar to `insert`. You check every element of the collection to see if it's in the tree. When you reach the last element of the collection, it must be a terminating element. If not, the collection was not added to the tree and what you've found is merely a subset of a larger collection.

The time complexity of `contains` is O(k), where k is the number of elements in the collection you're looking for. This is because you need to traverse through k nodes to find out whether or not the collection is in the trie.

To test out `insert` and `contains`, navigate to the playground page and add the following code:

```
example(of: "insert and contains") {
  let trie = Trie<String>()
  trie.insert("cute")
  if trie.contains("cute") {
    print("cute is in the trie")
  }
}
```

You should see the following console output:

```
---Example of: insert and contains---
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 two different collections. Write the following method just below `contains`:

```
public func remove(_ collection: CollectionType) {
  // 1
  var current = root
  for element in collection {
    guard let child = current.children[element] else {
      return
    }
    current = child
  }
  guard current.isTerminating else {
    return
  }
  // 2
  current.isTerminating = false
  // 3
  while let parent = current.parent,
        current.children.isEmpty && !current.isTerminating {
      parent.children[current.key!] = nil
      current = parent
  }
}
```

Taking it comment-by-comment:

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 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 you're trying to remove.

Head back to the playground page and add the following to the bottom:

```
example(of: "remove") {
  let trie = Trie<String>()
  trie.insert("cut")
  trie.insert("cute")

  print("\n*** Before removing ***")
  assert(trie.contains("cut"))
  print("\"cut\" is in the trie")
  assert(trie.contains("cute"))
  print("\"cute\" is in the trie")

  print("\n*** After removing cut ***")
  trie.remove("cut")
  assert(!trie.contains("cut"))
  assert(trie.contains("cute"))
  print("\"cute\" is still in the trie")
}
```

You should see the following output added to 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.swift:

```
public extension Trie where CollectionType: RangeReplaceabl

}
```

Your prefix matching algorithm will sit inside this extension, where `CollectionType` is constrained to `RangeReplaceableCollection`. This is required because the algorithm will need access to the `append` method of `RangeReplaceableCollection` types.

Next, add the following method inside the extension:

```
func collections(startingWith prefix: CollectionType) -> [Co
  // 1
  var current = root
  for element in prefix {
    guard let child = current.children[element] else {
      return []
    }
    current = child
  }

  // 2
  return collections(startingWith: prefix, after: current)
}
```

1. You start by verifying that the trie contains the prefix. If not, you return an empty array.

2. After you've found the node that marks the end of the prefix, you call a recursive helper method `collections(startingWith:after:)` to find all the sequences after the `current` node.

Next, add the code for the helper method:

```
private func collections(startingWith prefix: CollectionType
                         after node: Node) -> [CollectionTy
  // 1
  var results: [CollectionType] = []

  if node.isTerminating {
    results.append(prefix)
  }
  // 2
  for child in node.children.values {
    var prefix = prefix
    prefix.append(child.key!)
    results.append(contentsOf: collections(startingWith: pre
                                          after: child))
  }

  return results
}
```

1. You create an array to hold the results. If the current node is a terminating node, you add it to the results.

2. Next, you need to check the current node's children. For every child node, you recursively call `collections(startingWith:after:)` to seek out other terminating nodes.

`collection(startingWith:)` 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 where 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. Navigate back to the playground page and add the following:

```
example(of: "prefix matching") {
  let trie = Trie<String>()
  trie.insert("car")
  trie.insert("card")
  trie.insert("care")
  trie.insert("cared")
  trie.insert("cars")
  trie.insert("carbs")
  trie.insert("carapace")
  trie.insert("cargo")

  print("\nCollections starting with \"car\"")
  let prefixedWithCar = trie.collections(startingWith: "car'
  print(prefixedWithCar)

  print("\nCollections starting with \"care\"")
  let prefixedWithCare = trie.collections(startingWith: "ca
  print(prefixedWithCare)
}
```

You should see the following output in the console:

```
---Example of: prefix matching---

Collections starting with "car"
["car", "carbs", "care", "cared", "cars", "carapace", "cargo

Collections starting with "care"
["care", "cared"]
```

Where to go from here?

In this chapter, you learned about the trie, a tree structure that provides great performance metrics in regards to prefix matching. Tries are often featured in coding interviews, so study up!

Binary Search

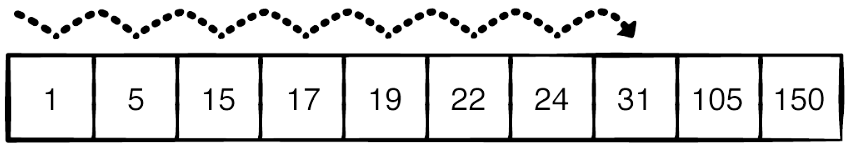
Binary search is one of the most efficient searching algorithms with a time complexity of O(log n). This is comparable with searching for an element inside a balanced binary search tree.

There are two conditions that need to be met before binary search may be used:

- The collection must be able to perform index manipulation in constant time. This means that the collection must be a `RandomAccessCollection`.
- The collection must be sorted.

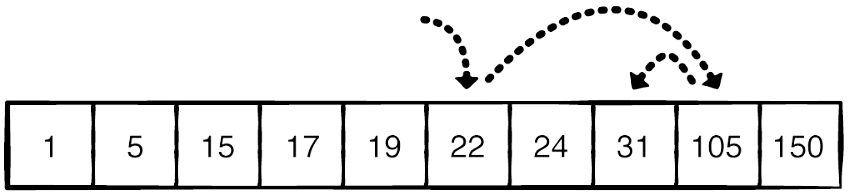
Example

The benefits of binary search are best illustrated by comparing it with linear search. Swift's `Array` type uses linear search to implement its `index(of:)` method. This means it traverses through the whole collection, or until it finds the element:



Binary search handles things differently by taking advantage of the fact that the collection is already sorted.

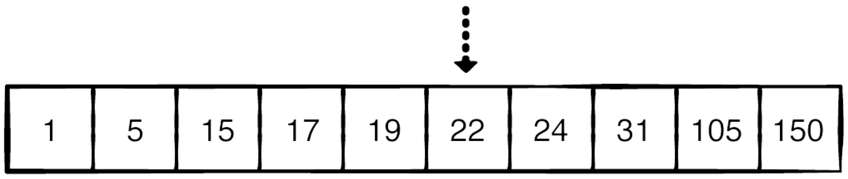
Here's an example of applying binary search to find the value 31:



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. This is fairly straightforward:



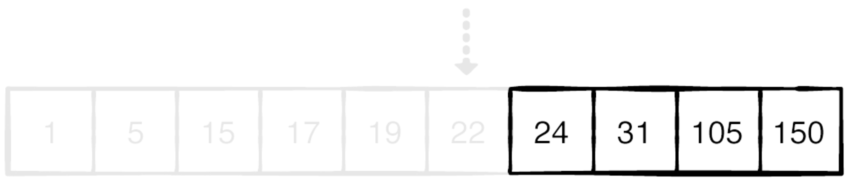
Step 2: Check the element at the middle index

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 to the 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 is 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 up 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 playground for this chapter. Create a new file in the Sources folder named BinarySearch.swift. Add the following to the file:

```
// 1
public extension RandomAccessCollection where Element: Compa
  // 2
  func binarySearch(for value: Element, in range: Range<Inde
      -> Index? {
    // more to come
  }
}
```

Things are fairly simple, so far:

1. Since binary search only works for types that conform to `RandomAccessCollection`, you add the method in an extension on

RandomAccessCollection. This extension is constrained as you need to be able to compare elements.

2. Binary search is recursive, so you need to be able to pass in a range to search. The parameter range is made optional so you can start the search without having to specify a range. In this case, where range is nil, the entire collection will be searched.

Next, implement binarySearch as follows:

```
// 1
let range = range ?? startIndex..<endIndex
// 2
guard range.lowerBound < range.upperBound else {
  return nil
}
// 3
let size = distance(from: range.lowerBound, to: range.upperI
let middle = index(range.lowerBound, offsetBy: size / 2)
// 4
if self[middle] == value {
  return middle
// 5
} else if self[middle] > value {
  return binarySearch(for: value, in: range.lowerBound..<mi
} else {
  return binarySearch(for: value, in: index(after: middle).
}
```

Here are the steps:

1. First you check if range was nil. If so, you create a range that covers the entire collection.

2. Then you check if the range contains at least one element. If it doesn't, the search has failed and you return nil.

3. Now that you're sure you have elements in the range, you find the middle index in the range.

4. You then compare the value at this index with the value you're searching for. If they match, you return the middle index.

5. If not, you recursively search either the left or right half of the collection.

That wraps up the implementation of binary search! Head back to the playground page to test it out. Write the following at the top of the playground page:

```
let array = [1, 5, 15, 17, 19, 22, 24, 31, 105, 150]

let search31 = array.index(of: 31)
let binarySearch31 = array.binarySearch(for: 31)

print("index(of:): \(String(describing: search31))")
print("binarySearch(for:): \(String(describing: binarySearcl
```

You should see the following output in the console:

```
index(of:): Optional(7)
binarySearch(for:): Optional(7)
```

This represents the index of the value you're looking for.

Where to go from here?

Binary search is a powerful algorithm to learn and 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 are given a problem that looks like it is going to be $O(n^2)$ to search, consider doing some up-front sorting so you can use binary searching to reduce it down to the cost of the sort at $O(n \log n)$.

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:

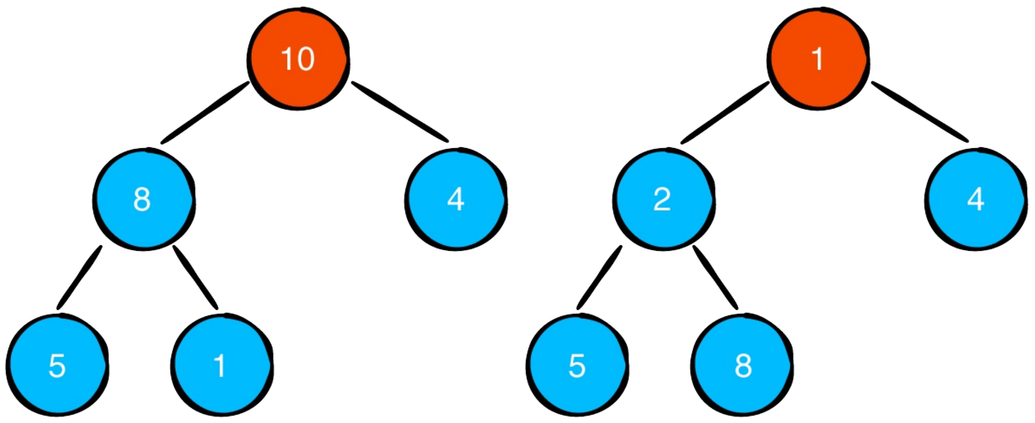
1. Max heap, where elements with a higher value have a higher priority.

2. Min heap, where elements with a lower value have a higher priority.

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 type of heap.

The heap property

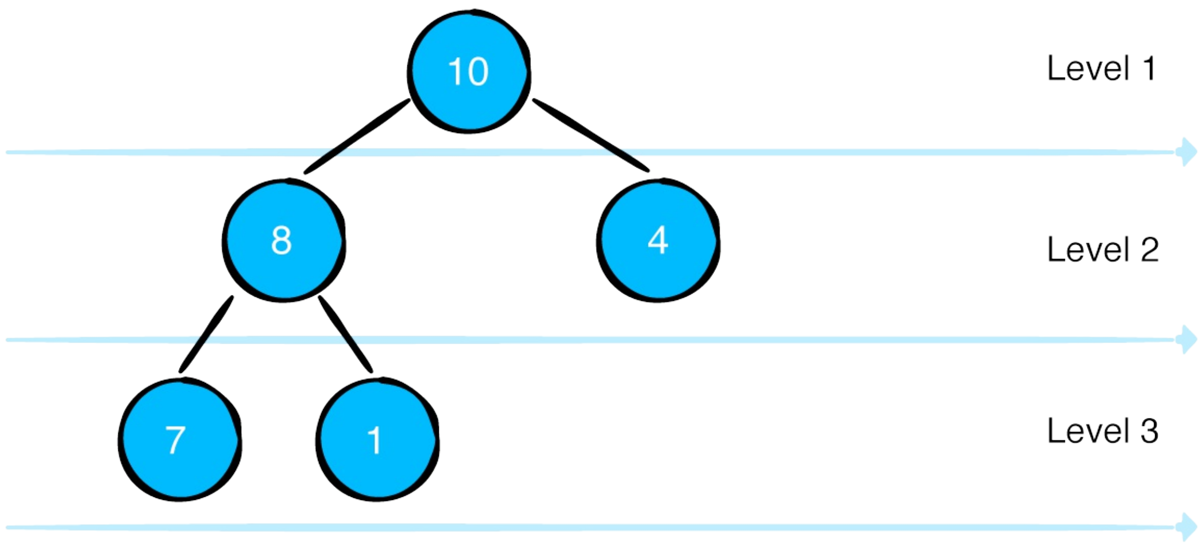
A heap has important characteristic that must always be satisfied. This is known as the heap invariant or heap property.



Max Heap Min Heap

In a max heap, 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 min heap, 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 is a complete binary tree. This means that every level must be filled, except for the last level. It's like a video game, where 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
- Constructing a priority queue
- Constructing graph algorithms like Prim's or Dijkstra's with a priority queue.

Note: You will learn about Priority Queues in Chapter 13, Heap Sort in Chapter 17 and Dijkstra's and Prim's algorithms in Chapter 22 and 23.

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 not what you are studying here.

In this chapter, you will focus on creating a heap, where you’ll see how convenient it is to fetch the minimum and maximum element of a collection.

Common heap operations

Open the empty starter playground for this chapter. Start by defining the following basic `Heap` type:

```
struct Heap<Element: Equatable> {

  var elements: [Element] = []
  let sort: (Element, Element) -> Bool

  init(sort: @escaping (Element, Element) -> Bool) {
    self.sort = sort
  }
}
```

This type contains an array to hold the elements in the heap and a sort function that defines how the heap should be ordered. By passing an appropriate function in the initializer, this type can be used to create both min and max heaps.

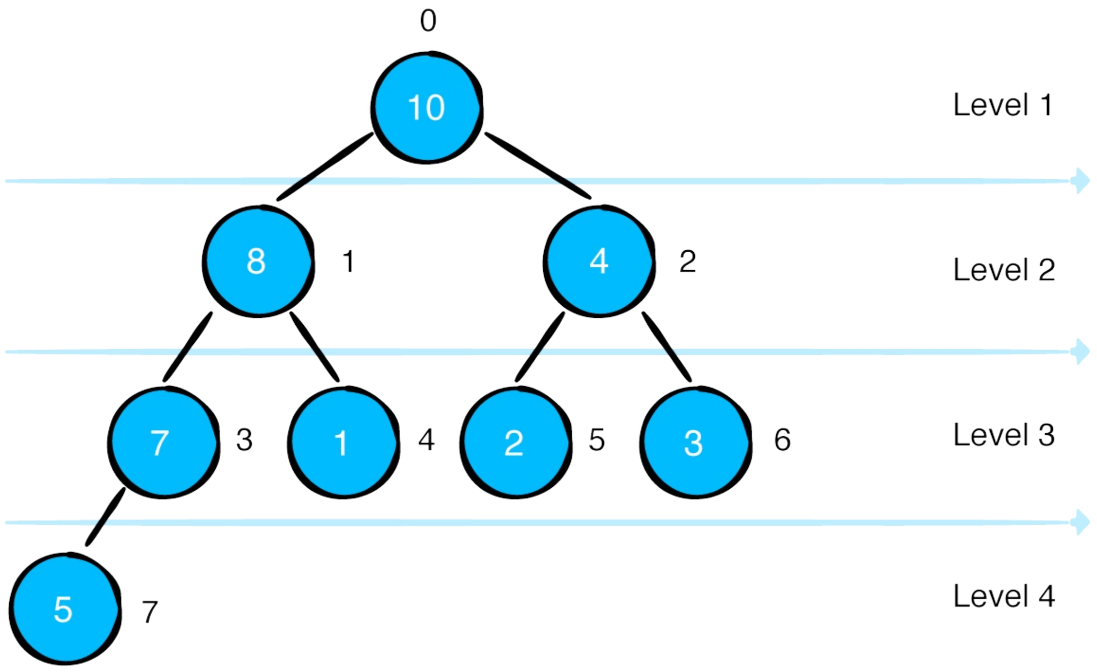
How do you represent a heap?

Trees hold nodes that store references to their children. In the case of a binary tree, these are references to a left and right child. Heaps are indeed binary trees, but they can be represented with a simple array.

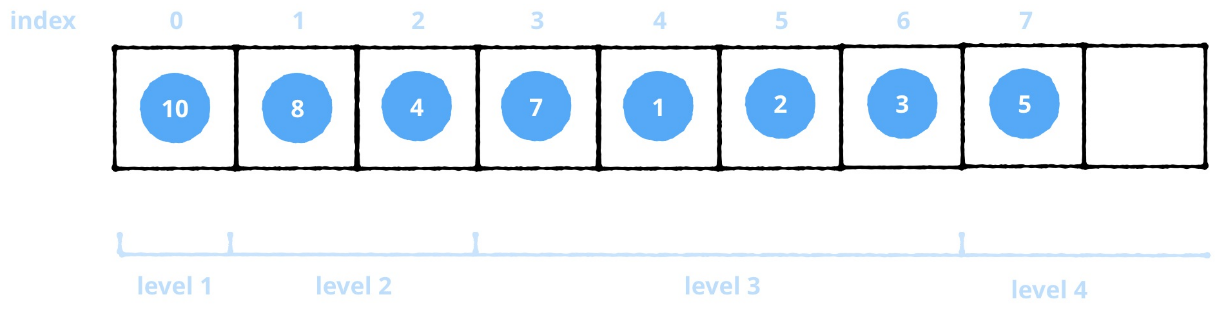
This seems like an unusual way to build a tree. But one of the benefits of this heap implementation is efficient time and space complexity, as the elements in the heap are all stored together in memory.

You will see later on that swapping elements will play a big part in heap operations. This is also easier to do with an array than with a binary tree data structure.

Let’s take a look at how heaps can be represented 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:

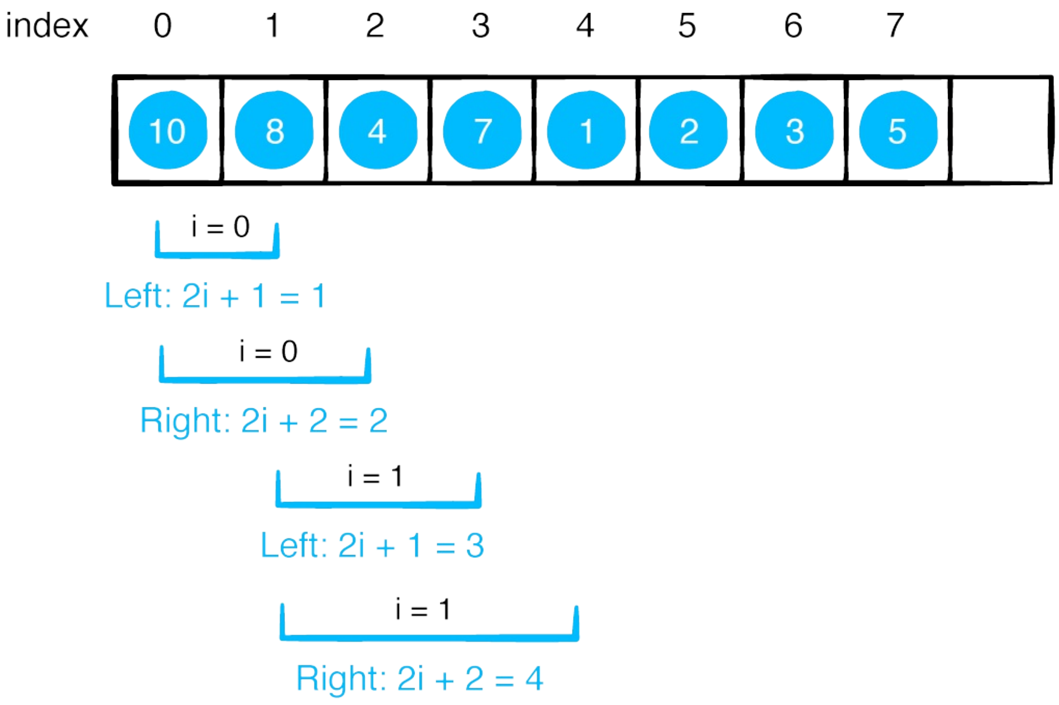


As you go up a level, you'll have twice as many nodes than in the level before.

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`:

- The left child of this node can be found at index `2i + 1`.
- The right child of this node can be found 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`, this child's parent node can be found at index `floor( (i - 1) / 2)`.

> Note: Traversing down an actual tree to get the left and right child of a node is a O(log n) operation. In an 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 `Heap`:

```
var isEmpty: Bool {
  return elements.isEmpty
}

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

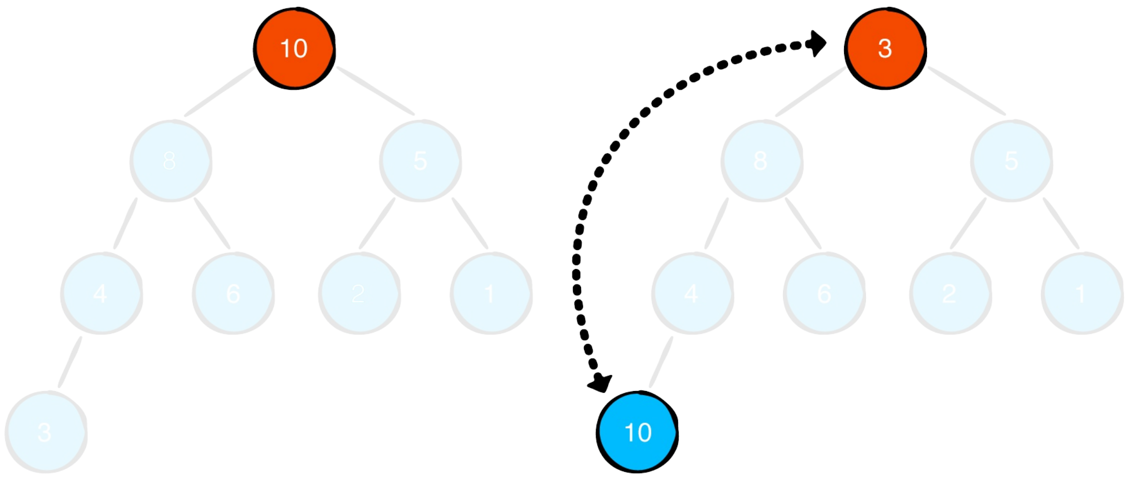
```
func peek() -> Element? {
  return elements.first
}

func leftChildIndex(ofParentAt index: Int) -> Int {
  return (2 * index) + 1
}

func rightChildIndex(ofParentAt index: Int) -> Int {
  return (2 * index) + 2
}

func parentIndex(ofChildAt index: Int) -> Int {
  return (index - 1) / 2
}
```

Now that you have a good understanding of how you can represent a heap using an array, you'll look at some important operations of a heap.

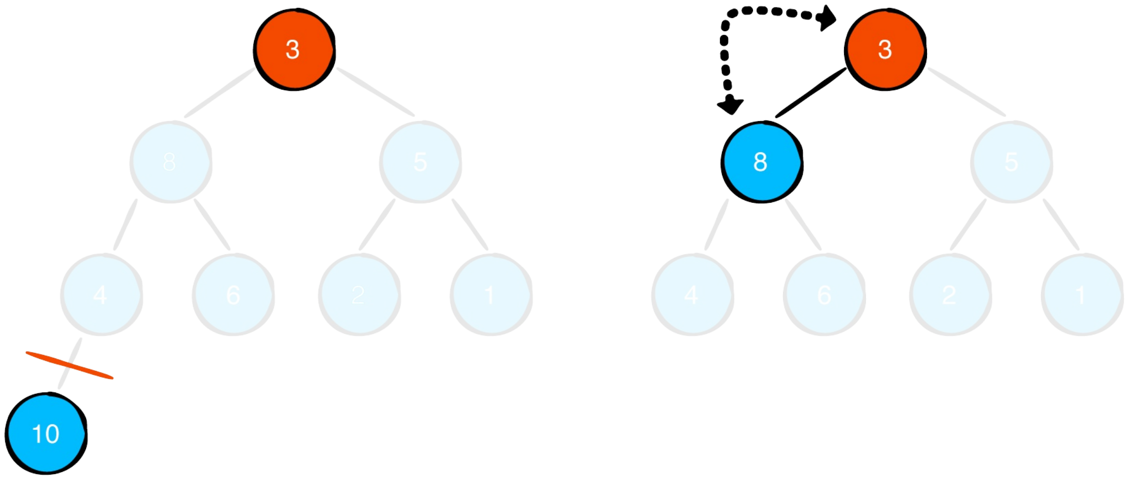
Removing from a heap

A basic remove operation simply removes the root node from the heap.

Take the following max heap:



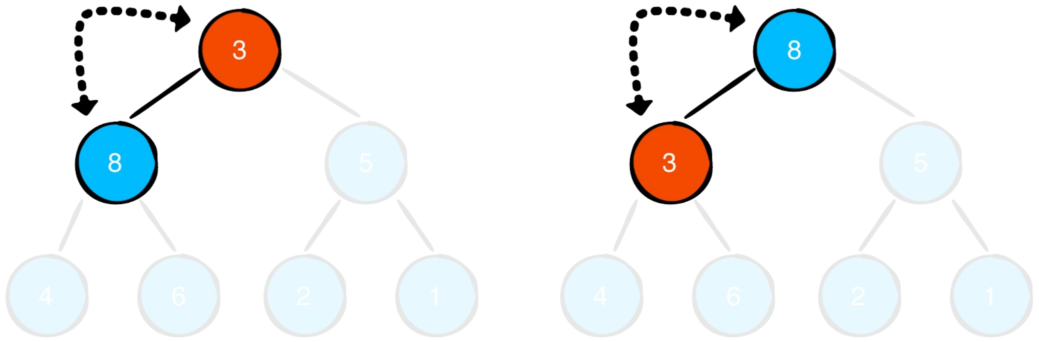
A remove operation will remove the maximum value at the root node. To do so, you must first swap the root node with the last element in the heap.



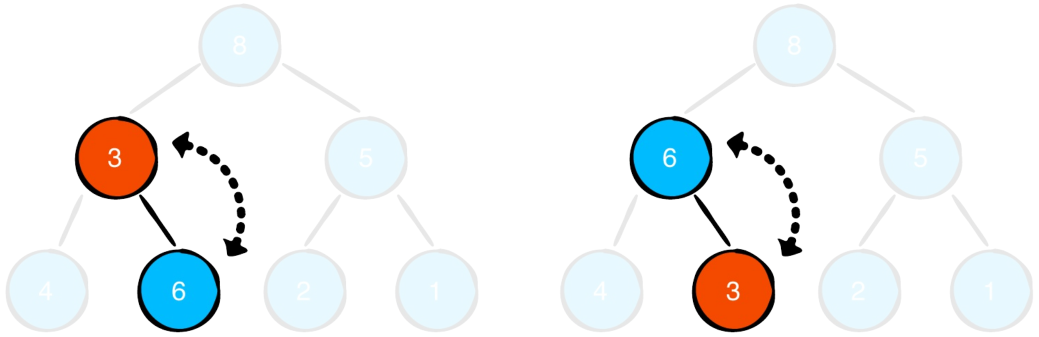
Once you've swapped the two elements, you can remove the last element and store its value so you can later return it.

Now you must check the max heap's integrity. But first, ask yourself, "Is it still a max heap?"

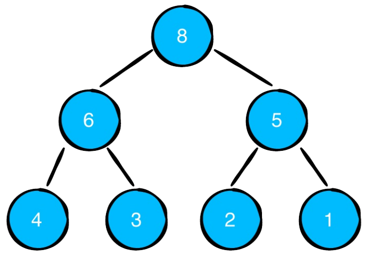
Remember: The rule for a max heap is that the value of every parent node must be larger than, or equal to, the values of its children. Since the heap no longer follows this rule, you must perform a sift down.



To perform a sift down, you 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 a greater value, you swap the parent with the child having the greater value.



Now 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 method to `Heap`:

```
mutating func remove() -> Element? {
  guard !isEmpty else { // 1
    return nil
  }
  elements.swapAt(0, count - 1) // 2
  defer {
    siftDown(from: 0) // 4
  }
  return elements.removeLast() // 3
}
```

Here's how this method works:

1. Check to see if the heap is empty. If it is, return `nil`.
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 max or min heap anymore, so you must perform a sift down to make sure it conforms to the rules.

Now to see how to sift down nodes. Add the following method after `remove()`:

```
mutating func siftDown(from index: Int) {
  var parent = index // 1
  while true { // 2
    let left = leftChildIndex(ofParentAt: parent) // 3
    let right = rightChildIndex(ofParentAt: parent)
    var candidate = parent // 4
```

```
      if left < count && sort(elements[left], elements[candida
        candidate = left // 5
      }
      if right < count && sort(elements[right], elements[cand
        candidate = right // 6
      }
      if candidate == parent {
        return // 7
      }
      elements.swapAt(parent, candidate) // 8
      parent = candidate
    }
  }
```

`siftDown(from:)` accepts an arbitrary index. This will always be treated as the parent node. Here's how the method works:

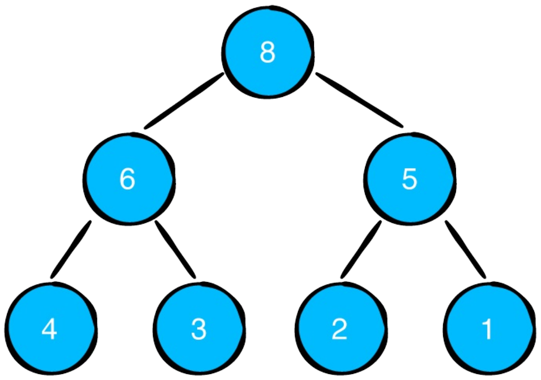
1. Store the `parent` index.
2. Continue sifting until you `return`.
3. Get the parent's left and right child index.
4. The `candidate` variable is used to keep track of which index to swap with the parent.
5. If there is a left child, and it has a higher priority than its parent, make it the candidate.
6. If there is 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.

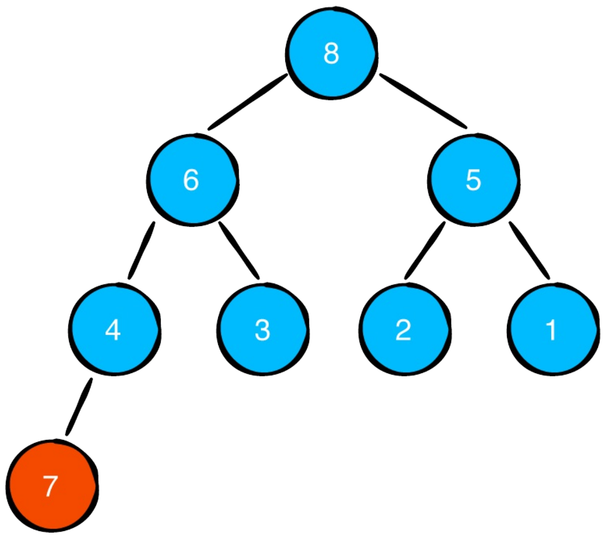
Now that you know how to remove from the top of the heap, how do you add to a heap?

Inserting into a heap

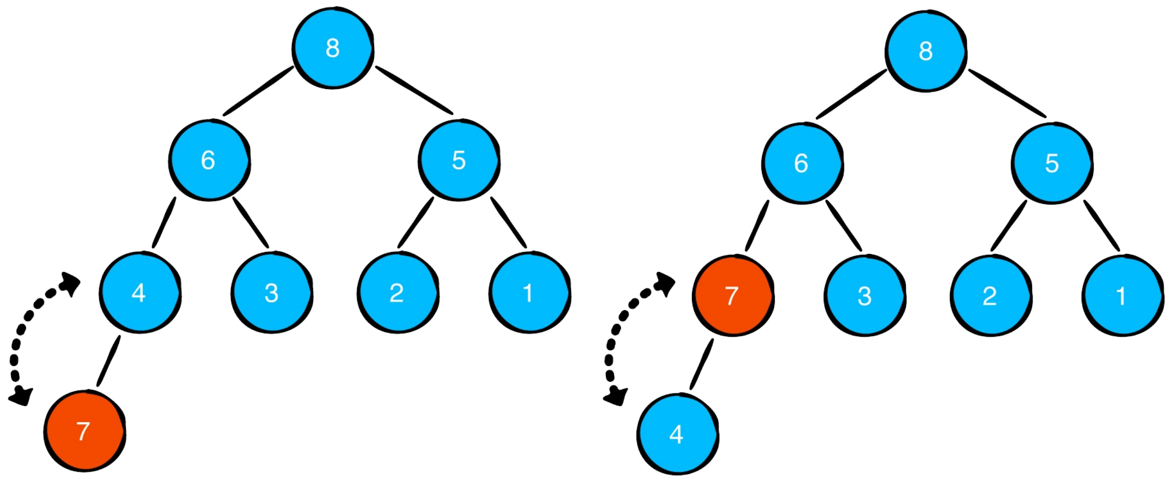
Let's say you insert a value of 7 to the heap below:

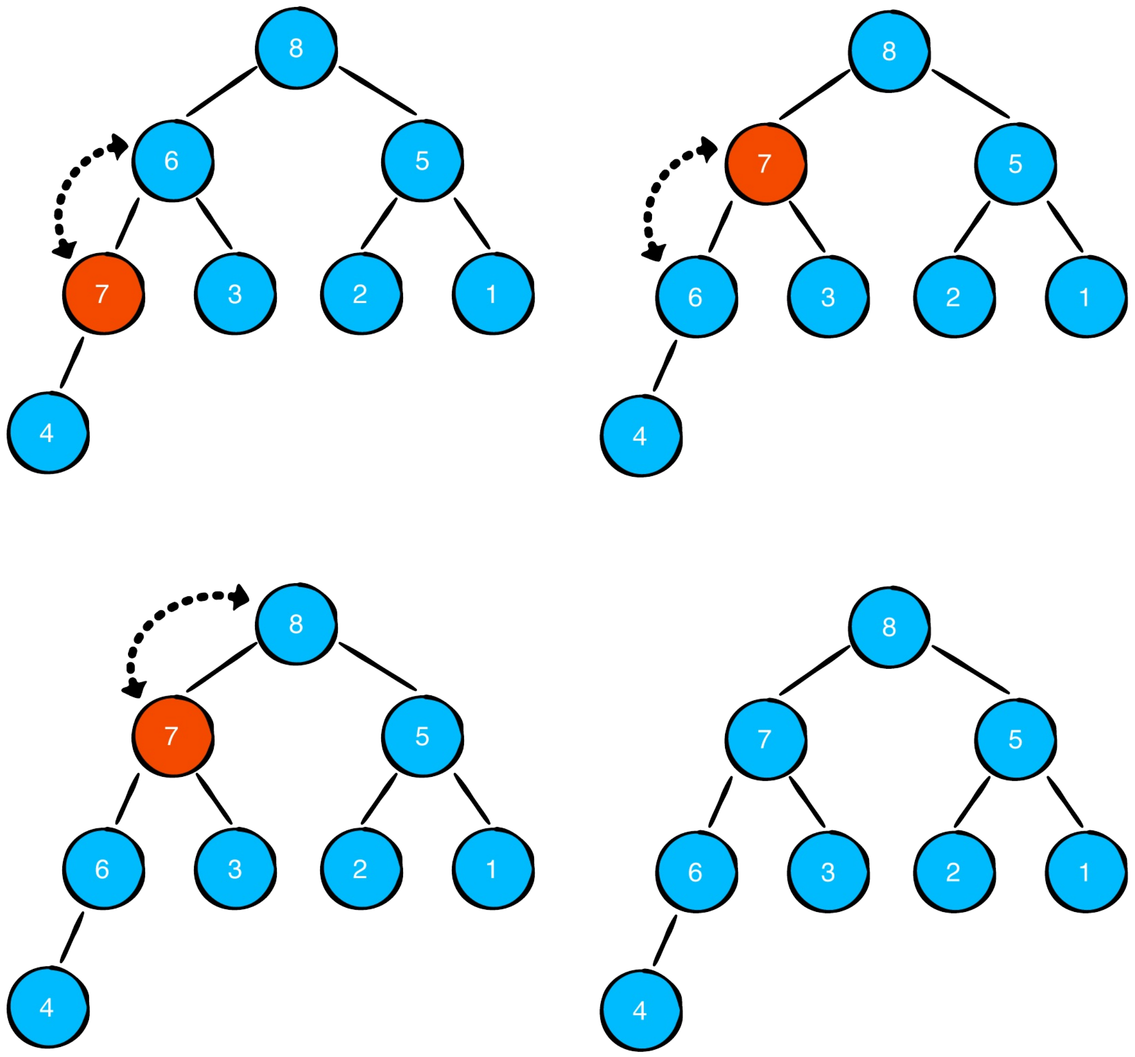


First, you add the value to the end of the heap:



Now you must check the max heap's property. Instead of sifting down, you must now sift up since the node you just inserted might have a higher priority than its parents. This sifting up works much like sifting down, 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 method to `Heap`:

```
mutating func insert(_ element: Element) {
  elements.append(element)
  siftUp(from: elements.count - 1)
}

mutating func siftUp(from index: Int) {
  var child = index
  var parent = parentIndex(ofChildAt: child)
  while child > 0 && sort(elements[child], elements[parent]
    elements.swapAt(child, parent)
    child = parent
    parent = parentIndex(ofChildAt: child)
  }
}
```

As you can see, the implementation is pretty straightforward:

- `insert` appends the element to the array and then performs a sift up.
- `siftUp` swaps the current node with its parent, as long as that node has a higher priority than its parent.

> Complexity: The overall compexity 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.

You have so far looked at removing the root element from a heap, and inserting into a heap. But what if you wanted to remove any arbitrary element from the heap?

Removing from an arbitrary index

Add the following to `Heap`:

```
mutating func remove(at index: Int) -> Element? {
  guard index < elements.count else {
    return nil // 1
  }
  if index == elements.count - 1 {
    return elements.removeLast() // 2
  } else {
    elements.swapAt(index, elements.count - 1) // 3
    defer {
      siftDown(from: index) // 5
      siftUp(from: index)
    }
    return elements.removeLast() // 4
  }
}
```

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 `nil`.

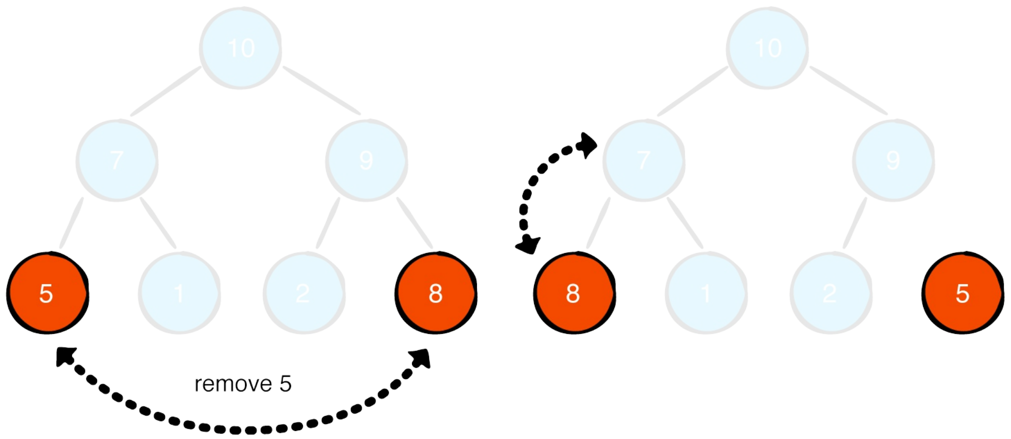
2. If you're removing the last element in the heap, you don't need to do anything special. Simply remove and return the element.

1. If you're not removing the last element, first swap the element with the last element.

2. Then return and remove the last element.

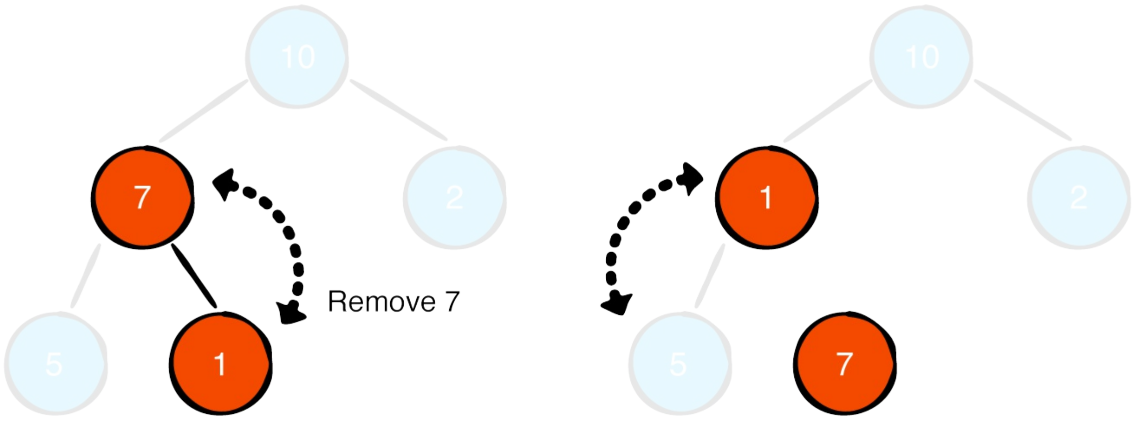
3. Finally, perform a sift down and a sift up to adjust the heap.

But — why do you have to perform both a sift down and a sift up?



Shifting Up Case

Assume you are 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 you are trying to remove 7. You swap 7 gets swapped with the last element, 1. You now need to perform a sift down to satisfy the max heap property.

Removing an arbitrary element from a heap is an O(log n) operation. But how do you actually find the index of the element you wish to delete?

Searching for an element in a heap

To find the index of the element you wish 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.

```
func index(of element: Element, startingAt i: Int) -> Int?
  if i >= count {
    return nil // 1
  }
  if sort(element, elements[i]) {
    return nil // 2
  }
  if element == elements[i] {
    return i // 3
  }
  if let j = index(of: element, startingAt: leftChildIndex(
    return j // 4
  }
  if let j = index(of: element, startingAt: rightChildIndex
    return j // 5
  }
  return nil // 6
}
```

Let's go over this implementation:

1. If the index is greater than the number of elements in the array, the search failed. Return `nil`.

2. Check to see if the element you are looking for has higher priority than the current element at index `i`. If it does, the element you are 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 `nil`.

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.

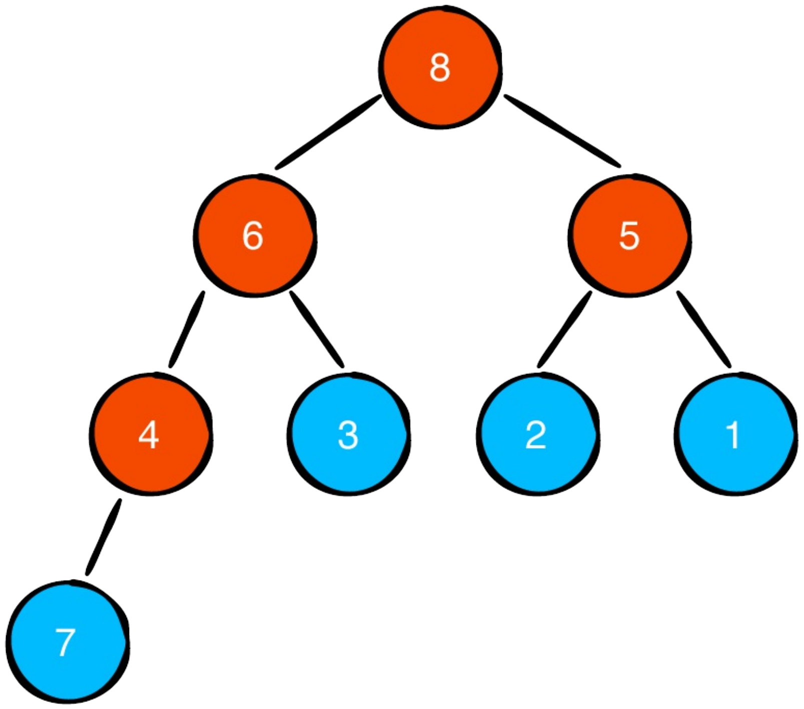
Building a heap

You now have all the necessary tools to represent a heap. To wrap up this chapter, you'll build a heap from an existing array of elements and test it out.

Update the initializer of `Heap` as follows:

```
init(sort: @escaping (Element, Element) -> Bool,
     elements: [Element] = []) {
  self.sort = sort
  self.elements = elements

  if !elements.isEmpty {
    for i in stride(from: elements.count / 2 - 1, through: (
      siftDown(from: i)
    }
    }
  }
```

The initializer now takes an additional parameter. 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 backwards, starting from the first non-leaf node, and sift down all parent nodes. You loop through only half of the elements, because there is no point in sifting down leaf nodes, only parent nodes.



Number of parents = total number of elements / 2
4 = 8 / 2

That's all there is to building a heap!

Testing

Time to try it out. Add the following to your playground:

```
var heap = Heap(sort: >, elements: [1,12,3,4,1,6,8,7])

while !heap.isEmpty {
  print(heap.remove()!)
}
```

This creates a max heap (because > is used as the sorting function) and removes elements one-by-one until it 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
```

Where to go from here?

Here is a summary of the algorithmic complexity of the heap operations you implemented in this chapter.

Heap Data Structure

Operations	Time Complexity
remove	0(log n)
insert	0(log n)
search	0(n)
peek	0(1)

The heap data structure is good for maintaining the highest or lowest value, depending on the type of heap. You also learned how to validate the heap by sifting elements up and down to satisfy the max or min heap property.

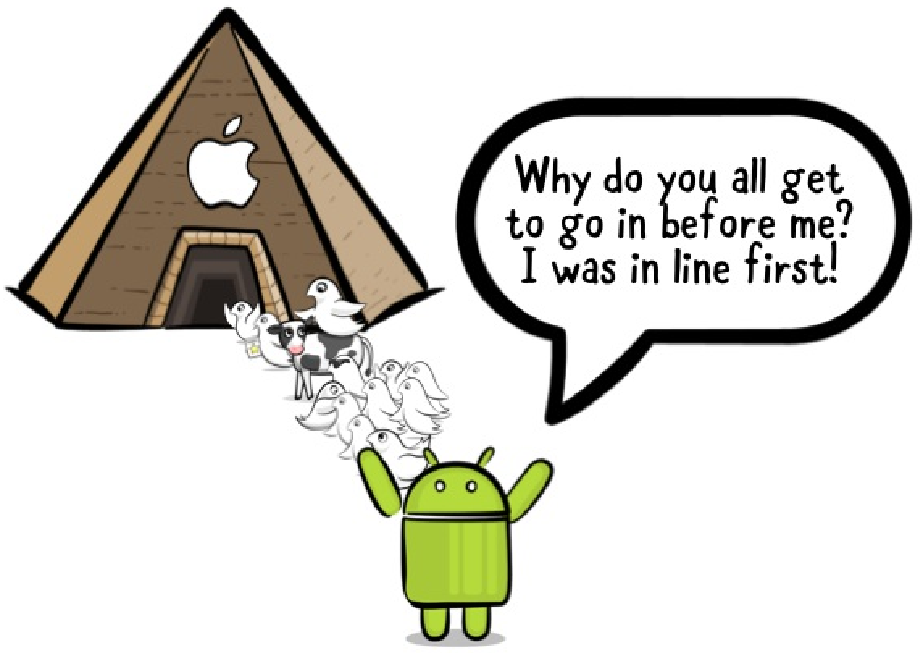
In the next few chapters, you will see other uses for heaps such as building priority queues and sorting a collection of objects.

Priority Queue

Queues are simply 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. For example, a priority queue can either be:

1. Max-priority, where the element at the front is always the largest.

2. Min-priority, where 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 given a list of elements.

Applications

Some useful applications of a priority queue include:

- Dijkstra’s algorithm, which uses a priority queue to calculate the minimum cost.
- A* pathfinding algorithm, which uses a priority queue to track the unexplored routes that will produce the path with the shortest length.
- Heap sort, which can be implemented using a priority queue.
- Huffman coding that builds a compression tree. A min-priority queue is used to repeatedly find two nodes with the smallest frequency that do not have a parent node yet.

These are just some of the use cases, but priority queues have many more applications as well.

Common operations

In Chapter 5, Queues, you established the following protocol for queues:

```swift
public protocol Queue {
  associatedtype Element
  mutating func enqueue(_ element: Element) -> Bool
  mutating func dequeue() -> Element?
  var isEmpty: Bool { get }
  var peek: Element? { get }
}
```

A priority queue has the same operations as a normal queue, so only the implementation will be different.

The priority queue will conform to the `Queue` protocol and implement the common operations:

- `enqueue`: Inserts an element into the queue. Returns `true` if the operation was successful.

- `dequeue`: Removes the element with the highest priority and return it. Returns `nil` if the queue was empty.
- `isEmpty`: Checks if the queue is empty.
- `peek`: Returns the element with the highest priority without removing it. Returns `nil` if the queue was empty.

Let's 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 will require O(n) since you have to insert it in order.

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 will look at how to use a heap to create a priority queue.

Open up the starter playground to get started. In the Sources folder you will notice the following files:

1. Heap.swift: The heap data structure (from the previous chapter) you will use to implement the priority queue.

2. Queue.swift: Contains the protocol that defines a queue.

In the main playground page, add the following:

```
struct PriorityQueue<Element: Equatable>: Queue { // 1

  private var heap: Heap<Element> // 2

  init(sort: @escaping (Element, Element) -> Bool,
       elements: [Element] = []) { // 3
    heap = Heap(sort: sort, elements: elements)
  }

  // more to come ...
}
```

Let's go over this code:

1. `PriorityQueue` will conform to the `Queue` protocol. The generic parameter `Element` must conform to `Equatable` as you need to be able to compare elements.

2. You will use this heap to implement the priority queue.

3. By passing an appropriate function into this initializer, `PriorityQueue` can be used to create both min and max priority queues.

To conform to the `Queue` protocol, add the following right after the `init(sort:elements:)` initializer:

```
var isEmpty: Bool {
  return heap.isEmpty
}

var peek: Element? {
  return heap.peek()
}

mutating func enqueue(_ element: Element) -> Bool { // 1
  heap.insert(element)
  return true
}

mutating func dequeue() -> Element? { // 2
  return heap.remove()
}
```

The heap is a perfect candidate for a priority queue. You simply need to call various methods of a heap to implement the operations of a priority queue!

1. From the previous chapter, you should understand that by calling `enqueue(_:)` you simply insert into the heap and the heap will sift up to validate itself. The overall complexity of `enqueue(_:)` is O(log n) .

2. By calling `dequeue(_:)` you remove the root element from the heap by replacing it with the last element in the heap and then sift down to validate the heap. The overall complexity of `dequeue()` is O(log n) .

Testing

Add the following to your playground:

```
var priorityQueue = PriorityQueue(sort: >, elements: [1,12,
while !priorityQueue.isEmpty {
  print(priorityQueue.dequeue()!)
}
```

You'll notice a priority queue has the same interface as a regular queue. The previous code creates a max priority queue. Notice that the elements are removed largest to smallest. The following numbers are printed to the console:

```
12
8
7
6
4
3
1
1
```

Where to go from here?

A priority queue is basically a heap that is useful when you need to find the maximum or minimum element immediately.

Not only did you learn how to implement a priority queue, you learned to apply the heap data structure and conformed to the queue protocol. Now that’s composition!

O(n^2) Sorting Algorithms

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; they only require constant O(1) additional memory space. For small data sets, these sorts compare very favorably against more complex sorts.

In this chapter, you'll be looking at the following sorting algorithms:

- Bubble sort
- Selection sort
- Insertion sort

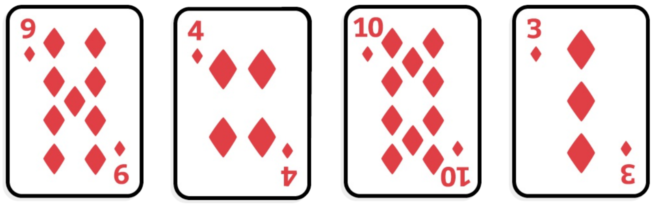
All of these are comparison-based sorting methods. They rely on a comparison method, such as the less-than operator, to order the elements. The number of times this comparison gets called is how you can measure a sorting technique's general performance.

Bubble sort

One of the simplest sorts is the bubble sort, which 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:

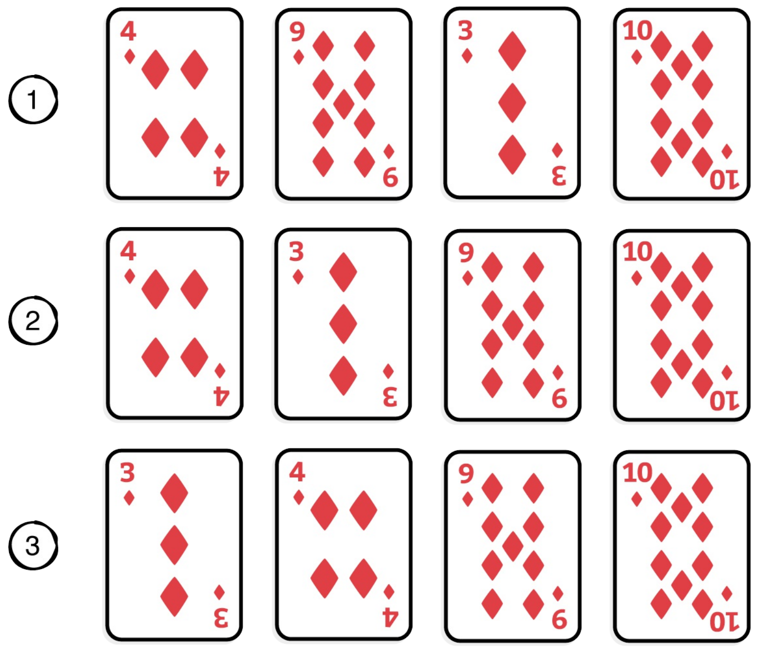


A single pass of the bubble sort algorithm would consist of the following steps:

- Start at the beginning of the collection. Compare 9 and 4. These values need to be swapped. The collection then becomes `[4, 9, 10, 3]`.
- Move to the next index in the collection. Compare 9 and 10. These are in order.
- Move to the next index in the collection. Compare 10 and 3. These values need to be swapped. The collection then becomes `[4, 9, 3, 10]`.

A single pass of the algorithm will seldom result in a complete ordering, which is true for this collection. It will, however, cause the largest value — 10 — to bubble up to the end of the collection.

Subsequent passes through the collection will do the same for 9 and 4 respectively:



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.

Implementation

Open up the Swift playground for this chapter to get started. In the Sources directory of your playground, create a new file named BubbleSort.swift. Write the following inside the file:

```swift
public func bubbleSort<Element>(_ array: inout [Element])
    where Element: Comparable {
  // 1
  guard array.count >= 2 else {
    return
  }
  // 2
  for end in (1..<array.count).reversed() {
    var swapped = false
    // 3
    for current in 0..<end {
      if array[current] > array[current + 1] {
        array.swapAt(current, current + 1)
        swapped = true
      }
    }
    // 4
    if !swapped {
      return
    }
  }
}
```

Here's the play-by-play:

1. There is no need to sort the collection if it has less than two elements.

2. A single pass 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 essentially shorten the array by one with each pass.

3. This loop performs a single pass; it compares adjacent values and swaps them if needed.

4. If no values were swapped this pass, the collection must be sorted, and you can exit early.

Try it out! Head back into the main playground page and write the following:

```
example(of: "bubble sort") {
  var array = [9, 4, 10, 3]
  print("Original: \(array)")
  bubbleSort(&array)
  print("Bubble sorted: \(array)")
}
```

You should see the following output:

```
---Example of bubble sort---
```

```
Original: [9, 4, 10, 3]
Bubble sorted: [3, 4, 9, 10]
```

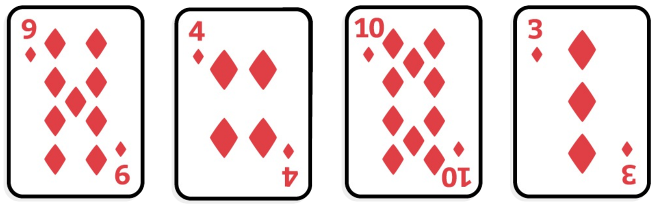
Bubble sort has a best time complexity of O(n) if it's already sorted, and a worst and average time complexity of O(n^2), making it one of the least appealing sorts in the known universe.

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 will only swap at the end of each pass. You'll see how that works in the following implementation.

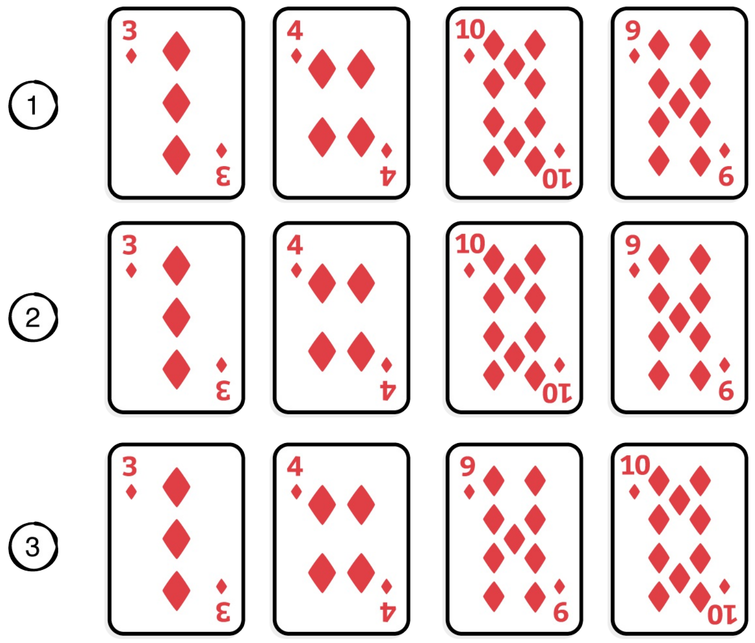
Example

Assume you have the following hand of cards:



During each pass, selection sort will find the lowest unsorted value and swap it into place:

1. First, 3 is found as the lowest value. It is swapped with 9.
2. The next lowest value is 4. It's already in the right place.
3. Finally, 9 is swapped with 10.



Implementation

In the Sources directory of your playground, create a new file named SelectionSort.swift. Write the following inside the file:

```
public func selectionSort<Element>(_ array: inout [Element]
    where Element: Comparable {
  guard array.count >= 2 else {
    return
  }
  // 1
  for current in 0..<(array.count - 1) {
    var lowest = current
    // 2
    for other in (current + 1)..<array.count {
      if array[lowest] > array[other] {
        lowest = other
      }
    }
    // 3
    if lowest != current {
      array.swapAt(lowest, current)
    }
  }
}
```

Here's what's going on:

1. You perform a pass for every element in the collection, except for the last one. There is no need to include the last element, since 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.

Try it out! Head back to the main playground page and add the following:

```
example(of: "selection sort") {
  var array = [9, 4, 10, 3]
  print("Original: \(array)")
  selectionSort(&array)
  print("Selection sorted: \(array)")
}
```

You should see the following output in your console:

```
---Example of selection sort---
Original: [9, 4, 10, 3]
Selection sorted: [3, 4, 9, 10]
```

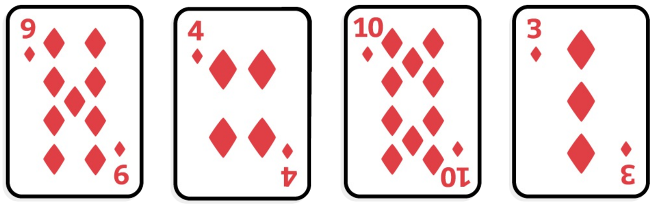
Just like bubble sort, selection sort has a best, worst, and average time complexity of $O(n^2)$, which is fairly dismal. It's a simple to understand, though, and it does perform better than bubble sort!

Insertion sort

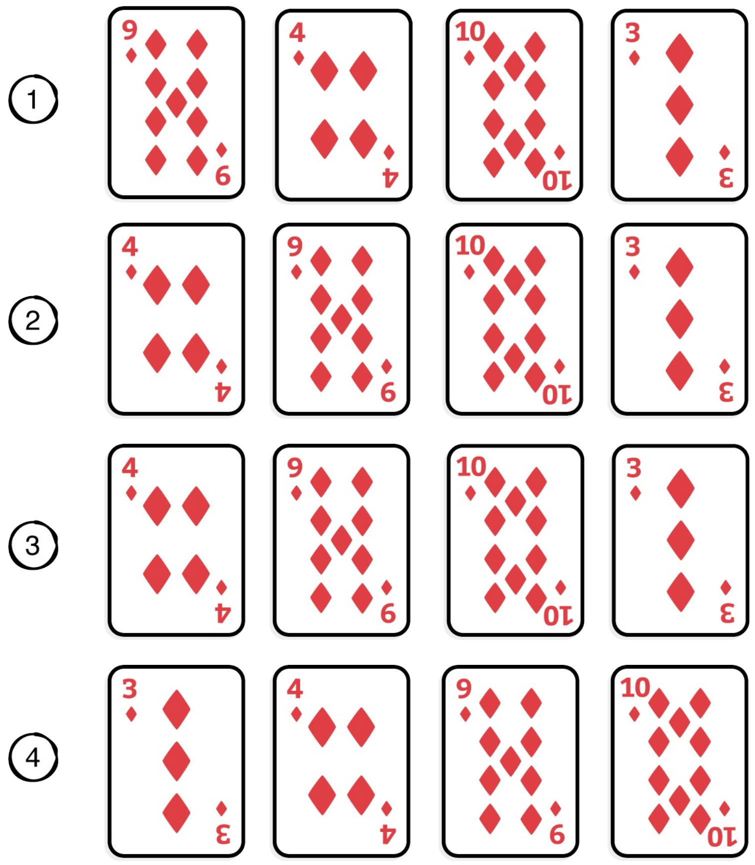
Insertion sort is a more useful algorithm. Like bubble sort and selection sort, insertion sort has an average time complexity of $O(n^2)$, but the performance of insertion sort can vary. The more the data is already sorted, the less work it needs to do. Insertion sort has a best time complexity of $O(n)$ if the data is already sorted. The Swift Standard Library sort algorithm uses a hybrid of sorting approaches with insertion sort being used for small (<20 element) unsorted partitions.

Example

The idea of insertion sort is similar to 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.

1. 10 doesn’t need to shift, as it's in the correct position compared to the previous card.
2. Finally, 3 is shifted all the way 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 the Sources directory of your playground, create a new file named InsertionSort.swift. Write the following inside the file:

```
public func insertionSort<Element>(_ array: inout [Element]
    where Element: Comparable {
  guard array.count >= 2 else {
    return
  }
  // 1
  for current in 1..<array.count {
    // 2
    for shifting in (1...current).reversed() {
      // 3
      if array[shifting] < array[shifting - 1] {
        array.swapAt(shifting, shifting - 1)
      } else {
        break
      }
    }
  }
}
```

Here's what you did above:

1. Insertion sort requires you to iterate from left to right once. This loop does that.

2. Here, you run backwards 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.

Head back to the main playground page and write the following at the bottom:

```
example(of: "insertion sort") {
  var array = [9, 4, 10, 3]
  print("Original: \(array)")
  insertionSort(&array)
  print("Insertion sorted: \(array)")
}
```

You should see the following console output:

```
---Example of insertion sort---
Original: [9, 4, 10, 3]
Insertion sorted: [3, 4, 9, 10]
```

Insertion sort is one of the fastest sorting algorithm, if the data is already sorted. That sounds obvious, but it isn't true for all sorting algorithms. In practice, a lot of data collections will already be largely — if not entirely — sorted, and insertion sort will perform quite well in those scenarios.

Generalization

In this section, you'll generalize these sorting algorithms for collection types other than `Array`. Exactly which collection types, though, depends on the algorithm:

- Insertion sort traverses the collection backwards when shifting elements. As such, the collection must be of type `BidirectionalCollection`.
- Bubble sort and selection sort really only traverse the collection front to back, so they can handle any `Collection`.
- In any case, the collection must be a `MutableCollection` as you need to be able to swap elements.

Head back to BubbleSort.swift and update the function to the following:

```
public func bubbleSort<T>(_ collection: inout T)
    where T: MutableCollection, T.Element: Comparable {
  guard collection.count >= 2 else {
      return
  }
  for end in collection.indices.reversed() {
    var swapped = false
    var current = collection.startIndex
    while current < end {
      let next = collection.index(after: current)
      if collection[current] > collection[next] {
        collection.swapAt(current, next)
        swapped = true
      }
      current = next
    }
    if !swapped {
      return
    }
  }
}
```

The algorithm stays the same; you simply update the loop to use the collection's indices. Head back to the main playground page to verify that bubble sort still works the way it should.

Selection sort can be updated as follows:

```
public func selectionSort<T>(_ collection: inout T)
    where T: MutableCollection, T.Element: Comparable {
  guard collection.count >= 2 else {
    return
  }
  for current in collection.indices {
    var lowest = current
    var other = collection.index(after: current)
    while other < collection.endIndex {
      if collection[lowest] > collection[other] {
        lowest = other
      }
      other = collection.index(after: other)
    }
    if lowest != current {
      collection.swapAt(lowest, current)
    }
  }
}
```

And insertion sort becomes:

```
public func insertionSort<T>(_ collection: inout T)
    where T: BidirectionalCollection & MutableCollection,
          T.Element: Comparable {
  guard collection.count >= 2 else {
    return
  }
  for current in collection.indices {
    var shifting = current
    while shifting > collection.startIndex {
      let previous = collection.index(before: shifting)
      if collection[shifting] < collection[previous] {
        collection.swapAt(shifting, previous)
      } else {
        break
      }
      shifting = previous
    }
  }
}
```

With just a bit of practice, generalizing these algorithms becomes a fairly mechanical process.

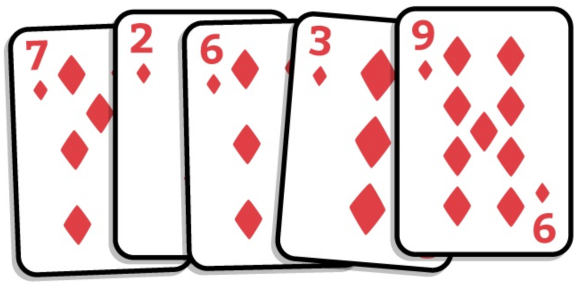
Where to go from here?

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!

Merge Sort

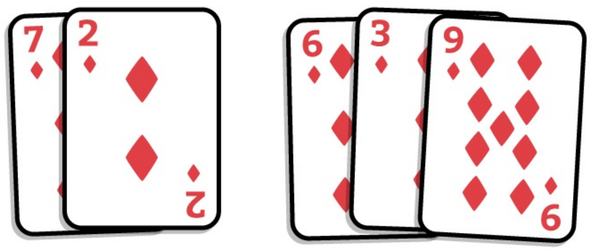
Merge sort is one of the most efficient sorting algorithms. With a time complexity of O(log n), it's one of the fastest of all general-purpose sorting algorithms. The idea behind merge sort is divide and conquer; to break up a big problem into several smaller, easier to solve problems and then combine those solutions into a final result. The merge sort mantra is to split first and merge after.

Assume you're given a pile of unsorted playing cards:

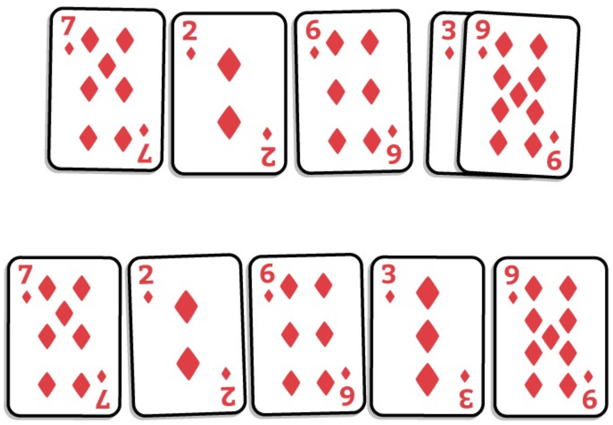


The merge sort algorithm works as follows:

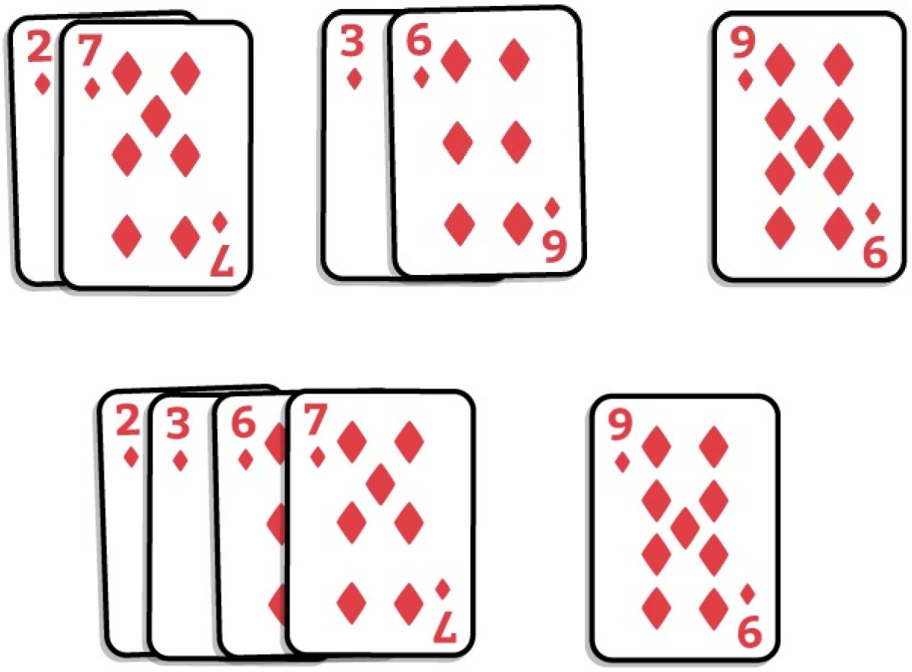
1. First, split the pile in half. You now have two unsorted piles:

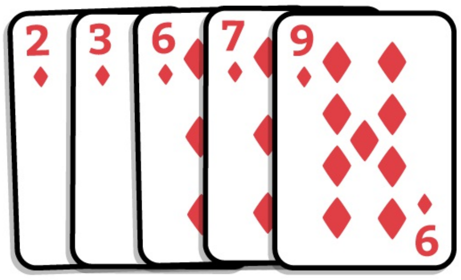


1. Now keep splitting the resulting piles until you can't split anymore. In the end, you will have one (sorted!) card in each pile:



1. Finally, merge the piles together in the reverse order in which you split them. During each merge, you put the contents in sorted order. This is easy because each individual pile has already been sorted:





Implementation

Open up the starter playground to get started.

Split

In the Sources folder in your playground, create a new file named MergeSort.swift. Write the following inside the file:

```
public func mergeSort<Element>(_ array: [Element])
    -> [Element] where Element: Comparable {
  let middle = array.count / 2
  let left = Array(array[..<middle])
  let right = Array(array[middle...])
  // ... more to come
}
```

Here you split the array into halves. Splitting once isn't enough, however; you have to keep splitting recursively until you can't split any more, which is when each subdivision contains just one element.=

To do this, update `mergeSort` as follows:

```
public func mergeSort<Element>(_ array: [Element])
    -> [Element] where Element: Comparable {
  // 1
  guard array.count > 1 else {
    return array
  }
  let middle = array.count / 2
  // 2
  let left = mergeSort(Array(array[..<middle]))
  let right = mergeSort(Array(array[middle...]))
  // ... more to come
}
```

You've made two changes here:

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 array only has one element.

2. You're now calling `mergeSort` on the left and right halves of the original array. As soon as you've split the array in half, you'll try to split again.

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` arrays together. To keep things clean, you will create a separate `merge` function for this.

The sole responsibility of the merging function is to take in two `sorted` arrays and combine them while retaining the sort order. Add the following just below the `mergeSort` function:

```
private func merge<Element>(_ left: [Element], _ right: [Ele
    -> [Element] where Element: Comparable {
  // 1
  var leftIndex = 0
  var rightIndex = 0
  // 2
  var result: [Element] = []
  // 3
  while leftIndex < left.count && rightIndex < right.count
    let leftElement = left[leftIndex]
    let rightElement = right[rightIndex]
```

```
      // 4
      if leftElement < rightElement {
        result.append(leftElement)
        leftIndex += 1
      } else if leftElement > rightElement {
        result.append(rightElement)
        rightIndex += 1
      } else {
        result.append(leftElement)
        leftIndex += 1
        result.append(rightElement)
        rightIndex += 1
      }
    }
    // 5
    if leftIndex < left.count {
      result.append(contentsOf: left[leftIndex...])
    }
    if rightIndex < right.count {
      result.append(contentsOf: right[rightIndex...])
    }
    return result
  }
```

Here’s what’s going on:

1. The `leftIndex` and `rightIndex` variables track your progress as you parse through the two arrays.

2. The `result` array will house the combined array.

1. Starting from the beginning, you compare the elements in the `left` and `right` arrays sequentially. If you’ve reached the end of either array, there’s nothing else to compare.

2. The smaller of the two elements goes into the `result` array. If the elements were equal, they can both be added.

3. The first loop guarantees that either `left` or `right` is empty. Since both arrays are sorted, this ensures that the leftover

elements are greater than or equal to the ones currently in `result`. In this scenario, you can append the rest of the elements without comparison.

Finishing up

Complete the `mergeSort` function by calling `merge`. Because you call `mergeSort` recursively, the algorithm will split and sort both halves before merging them together.

```
public func mergeSort<Element>(_ array: [Element])
    -> [Element] where Element: Comparable {
  guard array.count > 1 else {
    return array
  }
  let middle = array.count / 2
  let left = mergeSort(Array(array[..< middle]))
  let right = mergeSort(Array(array[middle...]))
  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 array recursively, and a method to merge two arrays.

3. The merging function should take two sorted arrays and produce a single sorted array.

Finally - time to see this in action. Head back to the main playground page and test your merge sort with the following:

```
example(of: "merge sort") {
  let array = [7, 2, 6, 3, 9]
```

```
    print("Original: \(array)")
    print("Merge sorted: \(mergeSort(array))")
}
```

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 array into two smaller arrays. This means an array of size 2 will need 1 level of recursion, an array of size 4 will need 2 levels, an array of size 8 will need 3 levels, and so on. If you had an array of 1024 elements, it would take 10 levels of recursively splitting in two to get down to 1024 single element arrays. In general, if you have an array of size n, the number of levels is log2(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) × 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 log2(n) levels of recursion and at each level n elements are used. That makes the total O(n log n) in space complexity.

Where to go from here?

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 are really clever with your bookkeeping, you can reduce the memory required to O(n) by discarding the memory that is not actively being used.

Radix Sort

In this chapter, you'll look at a completely different model of sorting. So far, you've been relying on comparisons to determine the sorting order. Radix sort is a non-comparative algorithm for sorting integers in linear time.

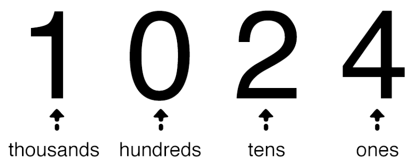
There are multiple implementations of radix sort that focus on different problems. To keep things simple, in this chapter you'll focus on sorting base 10 integers while investigating the least significant digit (LSD) variant of radix sort.

Example

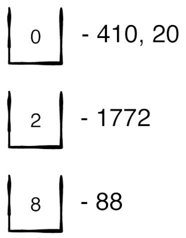
To show how radix sort works, you'll sort the following array:

```
var array = [88, 410, 1772, 20]
```

Radix sort relies on the positional notation of integers, as shown here:



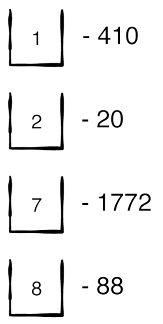
First, the array is divided into buckets based on the value of the least significant digit: the ones digit.



These buckets are then emptied in order, resulting in the following partially-sorted array:

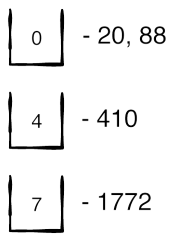
```
array = [410, 20, 1772, 88]
```

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:

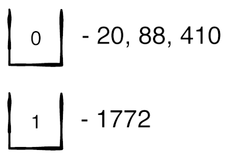


For values that have no hundreds position (or any other position without a value), the digit will be assumed to be zero.

Reassembling the array based on these buckets gives the following:

```
array = [20, 88, 410, 1772]
```

Finally, you need to consider the thousands digit:



Reassembling the array from these buckets leads to the final sorted array:

```
array = [20, 88, 410, 1772]
```

When multiple 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 array.

Implementation

Open up the starter project for this chapter. In the Sources directory, create a new file named RadixSort.swift. Add the following to the file:

```
extension Array where Element == Int {

  public mutating func radixSort() {

  }
}
```

Here you've added a `radixSort` method to arrays of integers via an extension. Start implementing the `radixSort` method using the following:

```
public mutating func radixSort() {
  // 1
  let base = 10
  // 2
  var done = false
```

```
    var digits = 1
    while !done {

    }
  }
```

This bit is fairly straightforward:

1. You're sorting base 10 integers in this instance. Since you'll be using this value multiple times in the algorithm, you store it in a constant `base`.

2. You declare two variables to track your progress. Radix sort works in multiple 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 (also known as Bucket sort).

Bucket Sort

Write the following inside the `while` loop:

```
// 1
var buckets: [[Int]] = .init(repeating: [], count: base)
// 2
forEach {
  number in
  let remainingPart = number / digits
  let digit = remainingPart % base
  buckets[digit].append(number)
}
// 3
digits *= base
self = buckets.flatMap { $0 }
```

Here's what you've written:

1. You instantiate the buckets using a two-dimensional array. Because you're using base 10, you need 10 buckets.

2. You place each number in the correct bucket.

3. You update `digits` to the next digit you wish to inspect and update the array using the contents of `buckets`. `flatMap` will flatten the two-dimensional array to a one-dimensional array, as if you're emptying the buckets into the array.

When do you stop?

Your `while` loop currently runs forever, so you'll need a terminating condition in there 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 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 the playground page and write the following to try out your code:

```
example(of: "radix sort") {
  var array = [88, 410, 1772, 20]
  print("Original array: \(array)")
  array.radixSort()
  print("Radix sorted: \(array)")
}
```

You should see the following console output:

```
---Example of: radix sort---
Original: [88, 410, 1772, 20]
Radix sorted: [20, 88, 410, 1772]
```

Where to go from here?

Radix sort is one of the fastest sorting algorithms. The average time complexity of radix sort is O(k × n), where k is the number of significant digits of the largest number, and n is the number of integers in the array.

Radix sort works best when k is constant, which occurs when all numbers in the array have the same count of significant digits. Its time complexity then becomes O(n). Radix sort also incurs a O(n) space complexity, as you need space to store each bucket.

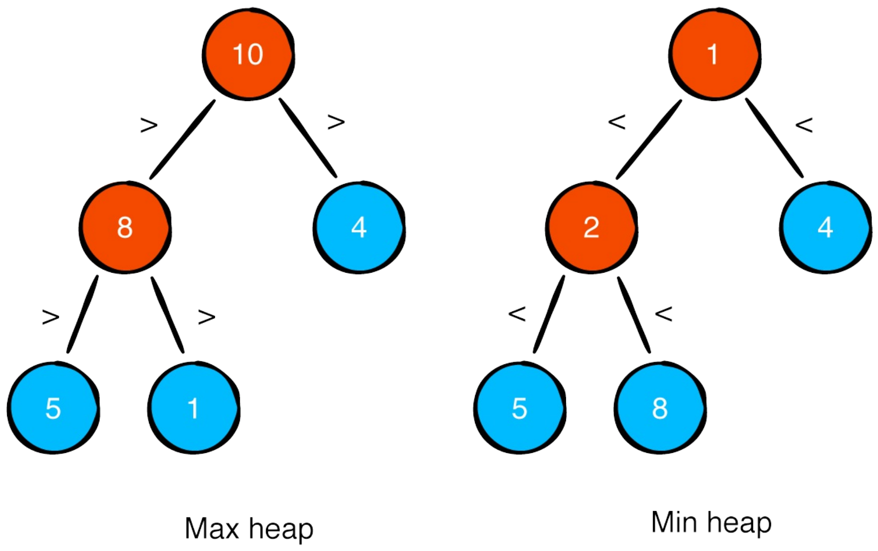
Heap Sort

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, "The 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:

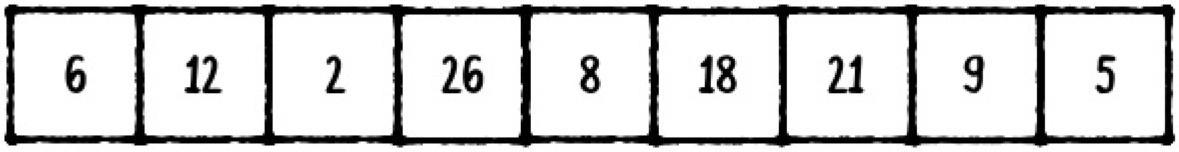


Getting started

Open up the starter playground. This playground already contains an implementation of a max 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.

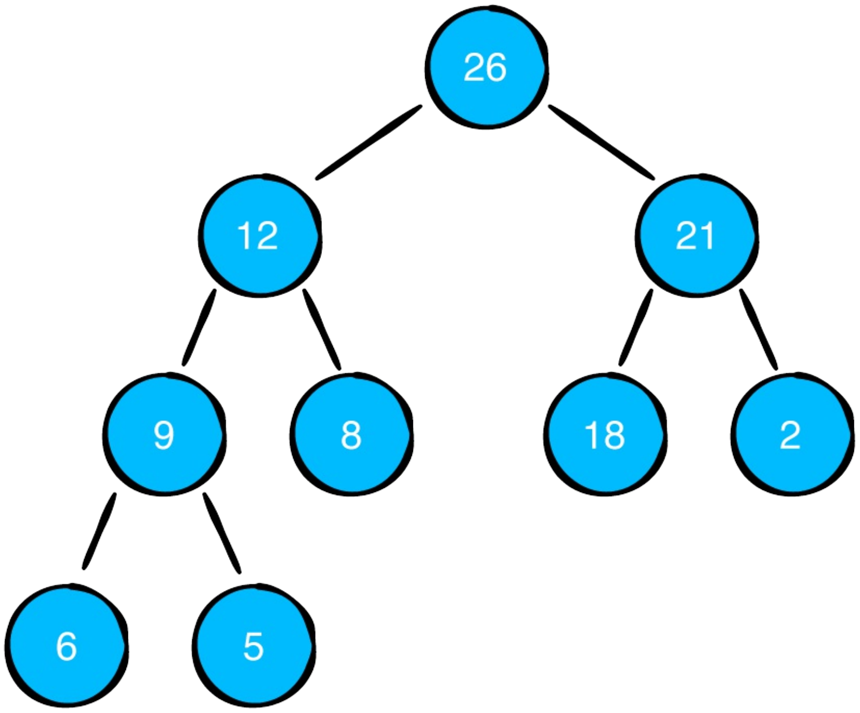
Example

For any given unsorted array, to sort from lowest to highest, heap sort must first convert this array into a max heap.

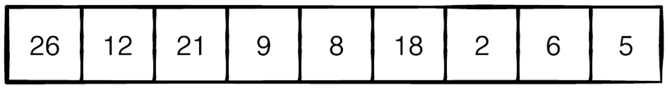


6	12	2	26	8	18	21	9	5

This conversion is done by sifting down all the parent nodes so they end up in the right spot. The resulting max heap is:



Which corresponds with the following array:

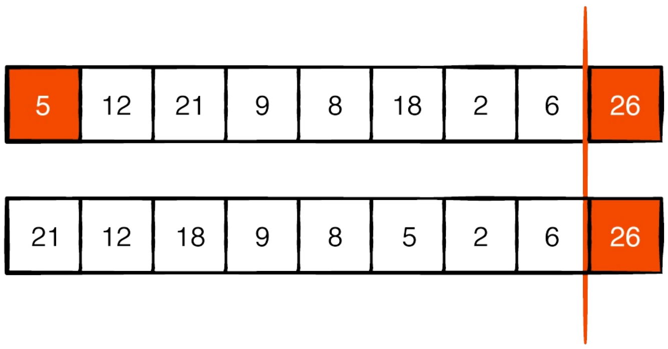


26	12	21	9	8	18	2	6	5

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).

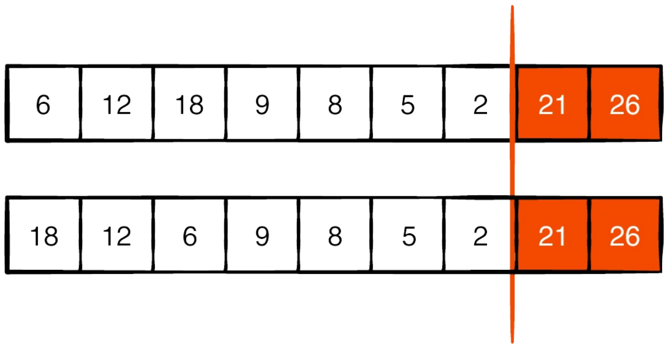
Let's look at how to sort this array in ascending order.

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 note 5 until it lands in its correct position.

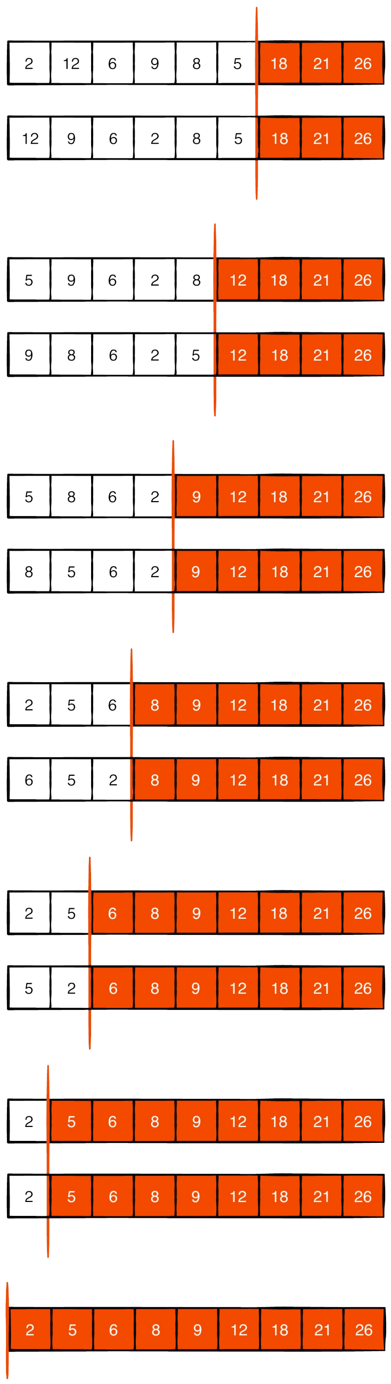


Note that you exclude the last element of the heap as we no longer consider it part of the heap, but of the sorted array.

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 very simple, as the heavy lifting is already done by the `siftDown` method:

```
extension Heap {
  func sorted() -> [Element] {
    var heap = Heap(sort: sort, elements: elements) // 1
    for index in heap.elements.indices.reversed() { // 2
      heap.elements.swapAt(0, index) // 3
      heap.siftDown(from: 0, upTo: index) // 4
    }
    return heap.elements
  }
}
```

Here's what's going on:

1. You first make a copy of the heap. After heap sort sorts the `elements` array, it is no longer a valid heap. By working on a copy of the heap, you ensure the heap remains valid.

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:

```
let heap = Heap(sort: >, elements: [6, 12, 2, 26, 8, 18, 21
print(heap.sorted())
```

This should print:

```
[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 list 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.

Where to go from here?

Heap sort is a natural application of the heap data structure and you now should have a solid grasp on how heap sorting works. You will use this as a fundamental building block in future chapters as you build your algorithm repertoire.

Quicksort

In the preceding chapters, you've learned to sort an array using comparison-based sorting algorithms, 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 array into three partitions:

```
[ elements < pivot | pivot | elements > pivot ]
```

In this chapter, you will implement Quicksort and look at various partitioning strategies to get the most out of this sorting algorithm.

Example

Open up the starter playground. A naïve implementation of Quicksort is provided in quicksortNaive.swift:

```
public func quicksortNaive<T: Comparable>(_ a: [T]) -> [T]
  guard a.count > 1 else { // 1
    return a
  }
  let pivot = a[a.count / 2] // 2
  let less = a.filter { $0 < pivot } // 3
  let equal = a.filter { $0 == pivot }
  let greater = a.filter { $0 > pivot }
  return quicksortNaive(less) + equal + quicksortNaive(grea
}
```

Let's look at how it works:

1. This is a recursive function, so you must have a base case. There must be more than one element in the array, otherwise there is

no need to sort.

2. The partitioning strategy here is to always pick the middle element of the array as the pivot.

3. Using the pivot, split the original array into three partitions. Elements less than, equal to or greater than the pivot go into different partitions.

4. Recursively sort the partitions and then combine them.

Let’s now visualize the code above. Given the unsorted array below:

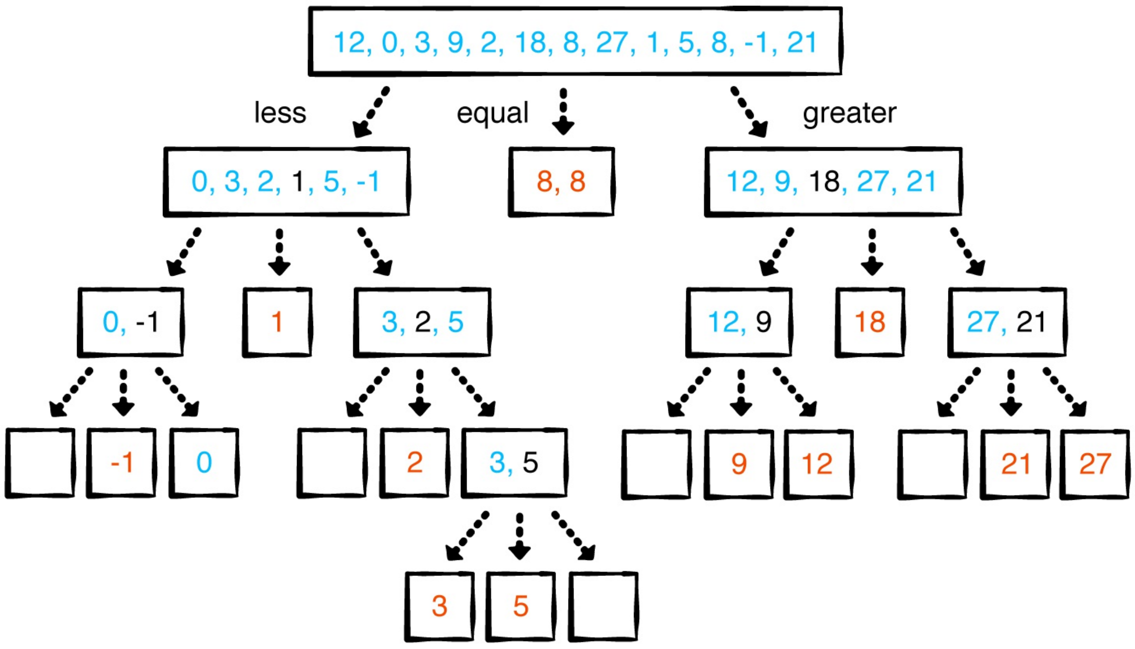
```
[12, 0, 3, 9, 2, 18, 8, 27, 1, 5, 8, -1, 21]
                    *
```

Your strategy in this implementation is to always select the middle element as the pivot. In this case, the element is 8. Partitioning the array 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 resulting partitions (the numbers in red) are combined again, resulting in a fully sorted array:

```
[-1, 1, 2, 3, 5, 8, 8, 9, 12, 18, 21, 27]
```

While this naïve implementation is easy to understand, it has some issues:

- Calling `filter` three times on the array is hardly efficient.
- Creating a new array for every partition isn't space efficient. Could you possibly sort in place?
- Is picking the middle element as a pivot a good strategy? What pivot strategy should you adopt?

Partitioning strategies

In this section, you will look at partitioning strategies and ways to make this Quicksort implementation more efficient. The first partitioning algorithm you will look at is Lomuto's algorithm.

Lomuto's partitioning

Lomuto's partitioning algorithm always chooses the last element as the pivot. Let's look at how this works in code.

In your playground, create a file called quicksortLomuto.swift and add the following function declaration:

```
public func partitionLomuto<T: Comparable>(_ a: inout [T],
                                           low: Int,
                                           high: Int) -> In
}
```

This function takes three arguments:

- `a` is the array you are partitioning.
- `low` and `high` set the range within the array you will 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:

```
let pivot = a[high] // 1

var i = low // 2
for j in low..<high { // 3
  if a[j] <= pivot { // 4
    a.swapAt(i, j) // 5
    i += 1
  }
}

a.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 array, it divides the array into four regions:

1. `a[low..<i]` contains all elements <= `pivot`.
2. `a[i...j-1]` contains all elements > `pivot`.
3. `a[j...high-1]` are elements you have not compared yet.
4. `a[high]` is the pivot element.

```
[ values <= pivot | values > pivot | not compared yet | piv
  low          i-1   i           j-1   j          high-1   hig
```

Step-by-step

Let’s look at a few steps of the algorithm to get a clear understanding of how it works.

Given the unsorted array 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 is 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 is 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, |  8  ]
 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, |  8  ]
 low                                         high
         i
            j
```

These steps continue until all but the pivot element have been compared. The resulting array 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, |  8  ]
 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, |  27  ]
 low                                          high
                         i
```

Lomuto's partitioning is now complete. Notice how the pivot is in between the two regions of elements less than or equal to the pivot and elements greater than the pivot.

In the naïve implementation of Quicksort, you created three new arrays and filtered the unsorted array 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:

```
public func quicksortLomuto<T: Comparable>(_ a: inout [T],
  if low < high {
    let pivot = partitionLomuto(&a, low: low, high: high)
    quicksortLomuto(&a, low: low, high: pivot - 1)
    quicksortLomuto(&a, low: pivot + 1, high: high)
  }
}
```

Here you simply apply Lomuto's algorithm to partition the array into two regions, then recursively sort these regions. Recursion ends once a region has less than two elements.

You can try out Lomuto's Quicksort by adding the following to your playground:

```
var list = [12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8]
quicksortLomuto(&list, low: 0, high: list.count - 1)
print(list)
```

Hoare's partitioning

Hoare's partitioning algorithm always chooses the first element as the pivot. Let's look at how this works in code.

In your playground, create a file named quicksortHoare.swift and add the following function:

```
public func partitionHoare<T: Comparable>(_ a: inout [T], lo
  let pivot = a[low] // 1
  var i = low - 1 // 2
  var j = high + 1

  while true {
    repeat { j -= 1 } while a[j] > pivot // 3
    repeat { i += 1 } while a[i] < pivot // 4

    if i < j { // 5
      a.swapAt(i, j)
    } else {
      return j // 6
    }
  }
}
```

Let's go over these steps:

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 lesser 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 array 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 array, looking for elements that are not lesser 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 array 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 ]
                                i
                            j
```

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:

```
public func quicksortHoare<T: Comparable>(_ a: inout [T], l
    if low < high {
      let p = partitionHoare(&a, low: low, high: high)
      quicksortHoare(&a, low: low, high: p)
      quicksortHoare(&a, low: p + 1, high: high)
    }
  }
```

Try it out by adding the following in your playground:

```
var list2 = [12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8]
quicksortHoare(&list2, low: 0, high: list.count - 1)
print(list2)
```

Effects of a bad pivot choice

The most important part of implementing Quicksort is choosing the right partitioning strategy.

You have 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?

Let’s start with the following unsorted array:

```
[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 array 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 array and use that as a pivot. This prevents you from picking the highest or lowest element in the array.

Let's look at an implementation. Create a new file named quicksortMedian.swift and add the following function:

```
public func medianOfThree<T: Comparable>(_ a: inout [T], lo
  let center = (low + high) / 2
  if a[low] > a[center] {
    a.swapAt(low, center)
  }
  if a[low] > a[high] {
    a.swapAt(low, high)
  }
  if a[center] > a[high] {
    a.swapAt(center, high)
  }
  return center
}
```

Here you find the median of `a[low]`, `a[center]` and `a[high]` by sorting them. The median will end up at index `center`, which is what the function returns.

Next, let's implement a variant of Quicksort using this median of three:

```
public func quickSortMedian<T: Comparable>(_ a: inout [T],
  if low < high {
    let pivotIndex = medianOfThree(&a, low: low, high: high
    a.swapAt(pivotIndex, high)
    let pivot = partitionLomuto(&a, low: low, high: high)
    quicksortLomuto(&a, low: low, high: pivot - 1)
    quicksortLomuto(&a, low: pivot + 1, high: high)
  }
}
```

This is simply a variation on `quicksortLomuto` that adds median of three as a first step.

Try this out by adding the following in your playground:

```
var list3 = [12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8]
quickSortMedian(&list3, low: 0, high: list3.count - 1)
print(list3)
```

This is definitely an improvement, but can we 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 lots of duplicate elements.

Let's look at how it's implemented. Create a file named quicksortDutchFlag.swift and add the following function:

```
public func partitionDutchFlag<T: Comparable>(_ a: inout [T
  let pivot = a[pivotIndex]
  var smaller = low // 1
  var equal = low // 2
  var larger = high // 3
  while equal <= larger { // 4
    if a[equal] < pivot {
      a.swapAt(smaller, equal)
      smaller += 1
      equal += 1
    } else if a[equal] == pivot {
      equal += 1
    } else {
      a.swapAt(equal, larger)
      larger -= 1
    }
  }
  return (smaller, larger) // 5
}
```

This is based on Lomuto's algorithm. Let's go over 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.

1. 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.

2. 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.

3. The algorithm returns indices `smaller` and `larger`. These point to the first and last elements of the middle partition.

Step-by-step

Let’s go over an example using the unsorted array 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 simply pick the last element 8. You could use median of three instead. 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 is larger than the pivot, it is 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
```

8 is equal to the pivot so you simply 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 array:

- Elements in `[low..<smaller]` are smaller than the pivot.
- Elements in `[smaller..<equal]` are equal to the pivot.
- Elements in `[larger>..high]` are larger than the pivot.
- Elements in `[equal...larger]` 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 is greater than the pivot, so it is 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 is smaller than the pivot, so it is 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:

```
public func quicksortDutchFlag<T: Comparable>(_ a: inout [T
  if low < high {
    let (middleFirst, middleLast) = partitionDutchFlag(&a,
    quicksortDutchFlag(&a, low: low, high: middleFirst - 1)
    quicksortDutchFlag(&a, low: middleLast + 1, high: 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 playground:

```
var list4 = [12, 0, 3, 9, 2, 21, 18, 27, 1, 5, 8, -1, 8]
quicksortDutchFlag(&list4, low: 0, high: list4.count - 1)
print(list4)
```

That's it!

Where to go from here?

In this chapter, you learned another divide and conquer sorting algorithm. Quicksort is all about choosing the right pivot for the job and then partitioning. You sort the array by breaking it down into smaller partitions and partition until you can go no further.

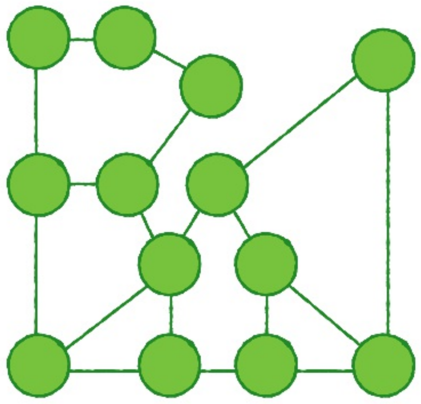
Remember that quicksort can perform its best in O(n log n), but when an array is nearly sorted, it could perform as bad as $O(n^2)$.

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 is made up of vertices connected by edges.

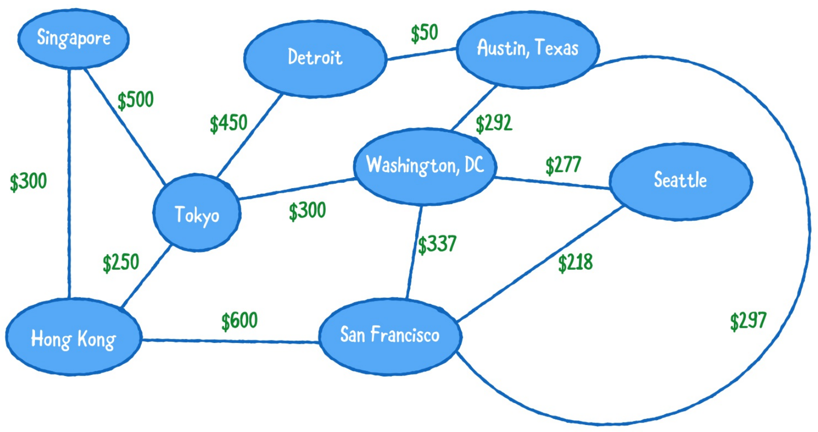
In the graph below, the vertices are represented by circles, and the edges are the lines that connect them.



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:



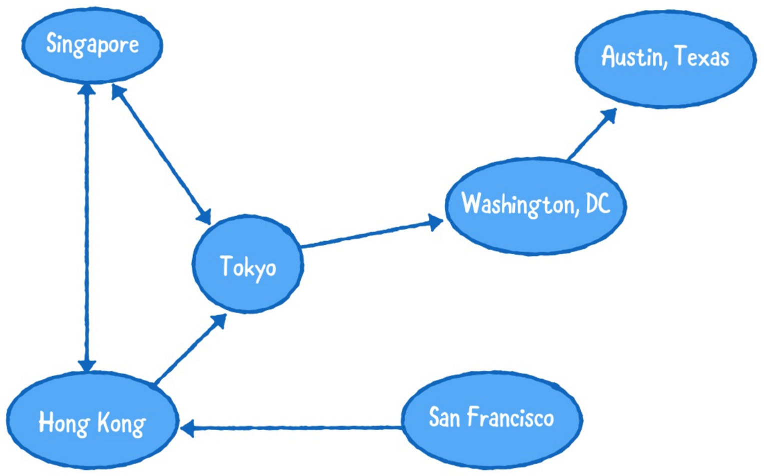
In this example, the vertices represent a state or country, while the edges represent a route from one place to another. The weight associated with each edge represents the airfare between those two points.

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.



You can tell a lot from this diagram:

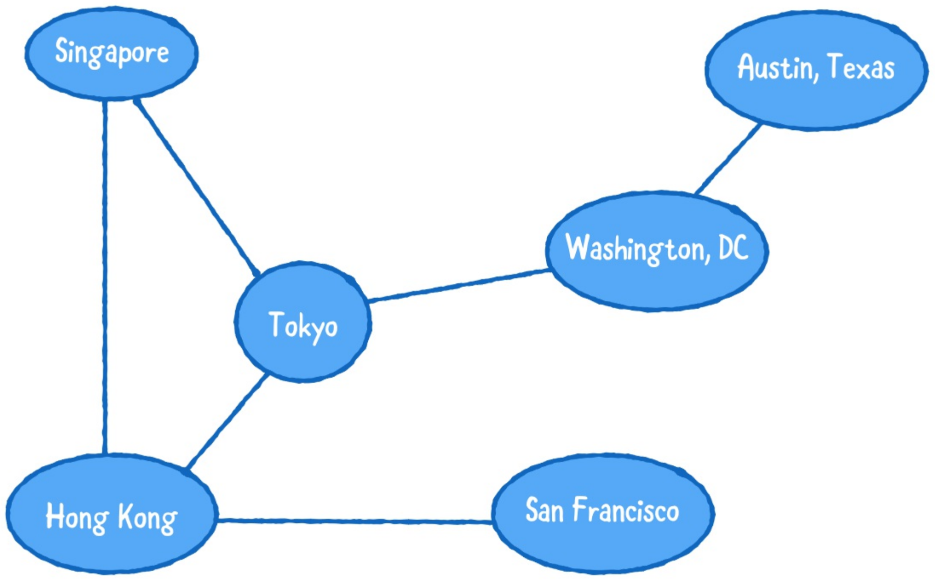
- There is a flight from Hong Kong to Tokyo.
- There is no direct flight from San Francisco to Tokyo.
- You can buy a roundtrip ticket between Singapore and Tokyo.
- There is no way to get from Tokyo to San Francisco.

Undirected graphs

You can think of an undirected graph as a directed graph where all edges are bidirectional.

In an undirected graph:

- Two connected vertices have edges going back and forth.
- The weight of an edge applies to both directions.



Common operations

Let's establish a protocol for graphs.

Open up the starter project for this chapter. Create a new file named Graph.swift and add the following inside the file:

```
public enum EdgeType {

  case directed
  case undirected
}

public protocol Graph {

  associatedtype Element

  func createVertex(data: Element) -> Vertex<Element>
  func addDirectedEdge(from source: Vertex<Element>,
                       to destination: Vertex<Element>,
                       weight: Double?)
  func addUndirectedEdge(between source: Vertex<Element>,
                         and destination: Vertex<Element>,
                         weight: Double?)
  func add(_ edge: EdgeType, from source: Vertex<Element>,
                             to destination: Vertex<Element
                             weight: Double?)
  func edges(from source: Vertex<Element>) -> [Edge<Element
  func weight(from source: Vertex<Element>,
              to destination: Vertex<Element>) -> Double?
}
```

This protocol describes the common operations for a graph:

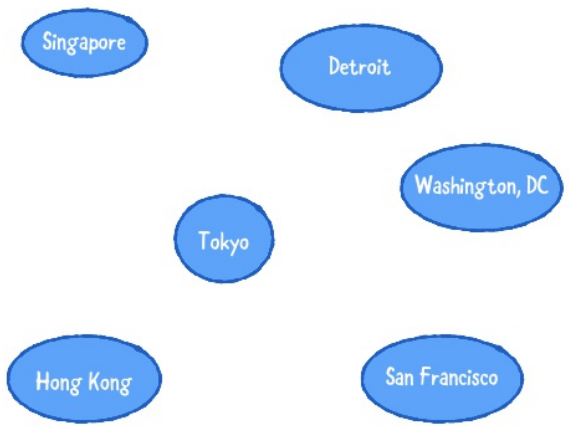
- `createVertex(data:)`: Creates a vertex and adds it to the graph.
- `addDirectedEdge(from:to:weight:)`: Adds a directed edge between two vertices.
- `addUndirectedEdge(between:and:weight:)`: Adds an undirected (or bidirectional) edge between two vertices.
- `add(from:to:)`: Uses `EdgeType` to add either a directed or undirected edge between two vertices.
- `edges(from:)`: Returns a list of outgoing edges from a specific vertex.
- `weight(from:to:)`: Returns the weight of the edge between two vertices.

In the following sections, you'll implement this protocol 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



Create a new file named Vertex.swift and add the following inside the file:

```
public struct Vertex<T> {

  public let index: Int
  public let data: T
}
```

Here you've defined a generic `Vertex` struct. A vertex has a unique index within its graph and holds a piece of data.

You'll use `Vertex` as the key type for a dictionary, so you need to conform to `Hashable`. Add the following extension to implement the requirements for `Hashable`:

```
extension Vertex: Hashable {

  public var hashValue: Int {
    return index.hashValue
  }

  public static func ==(lhs: Vertex, rhs: Vertex) -> Bool {
    return lhs.index == rhs.index
  }
}
```

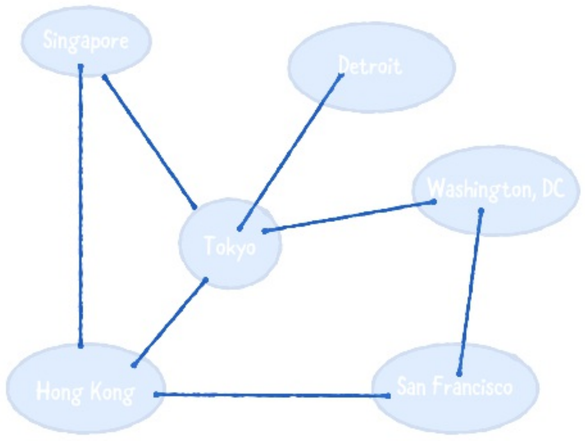
Because vertices have a unique `index`, you use the `index` property to implement `hashValue` and `==`.

Finally, you want to provide a custom string representation of `Vertex`. Add the following right after:

```
extension Vertex: CustomStringConvertible {

  public var description: String {
    return "\(index): \(data)"
  }
}
```

Defining an edge

To connect two vertices, there must be an edge between them!



Create a new file named Edge.swift and add the following inside the file:

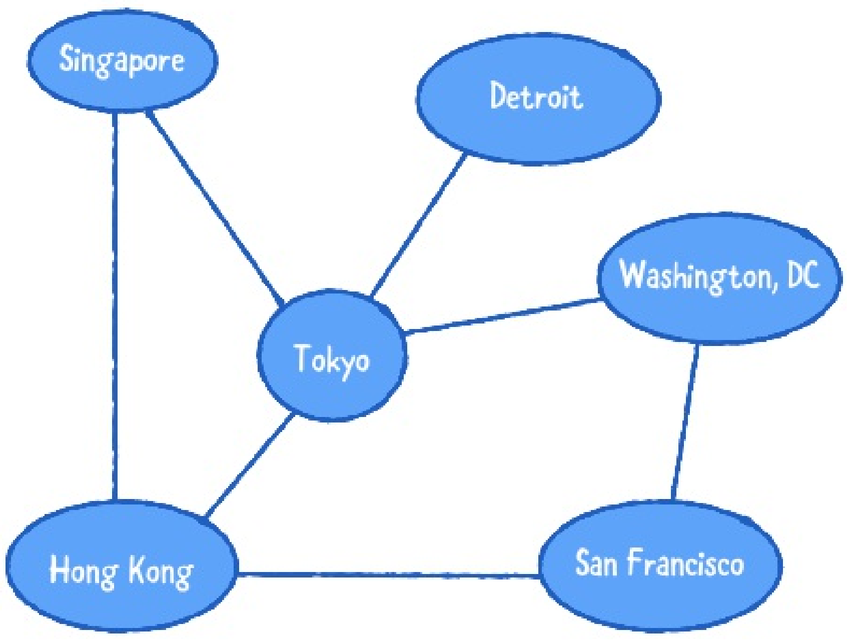
```
public struct Edge<T> {

  public let source: Vertex<T>
  public let destination: Vertex<T>
  public let weight: Double?
}
```

An `Edge` connects two vertices and has an optional weight. Simple, isn't it?

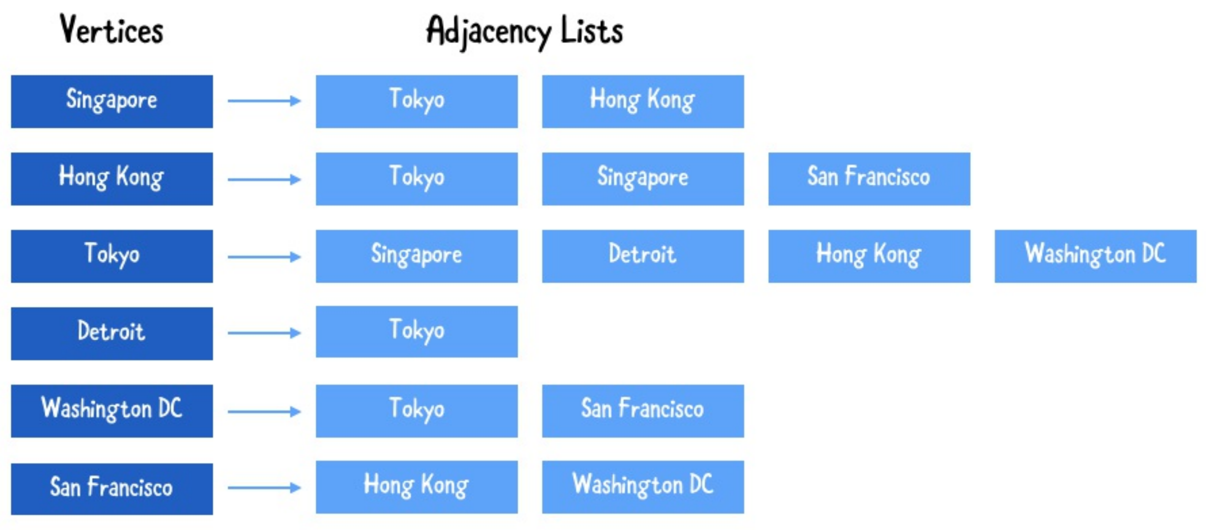
Adjacency list

The first graph implementation 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 network below:



The adjacency list below describes the network for the network of flights depicted above:



There is a lot you can learn from this adjacency list:

1. Singapore's vertex has two outgoing edges. There is 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 will create an adjacency list by storing a dictionary of arrays. Each key in the dictionary is a vertex, and in every vertex the dictionary holds a corresponding array of edges.

Implementation

Create a new file named AdjacencyList.swift, and add the following:

```
public class AdjacencyList<T>: Graph {

  private var adjacencies: [Vertex<T>: [Edge<T>]] = [:]

  public init() {}

  // more to come ...
}
```

Here you've defined an `AdjacencyList` that uses a dictionary to store the edges. You've already adopted the `Graph` protocol but still need to implement its requirements. That's what you'll do in the following sections.

Creating a vertex

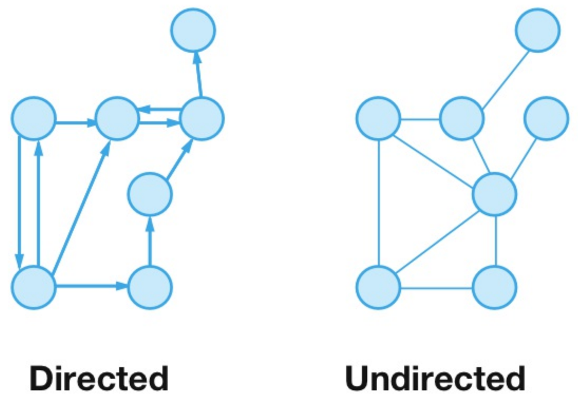
Add the following method to `AdjacencyList`:

```
public func createVertex(data: T) -> Vertex<T> {
  let vertex = Vertex(index: adjacencies.count, data: data)
  adjacencies[vertex] = []
  return vertex
}
```

Here you create a new vertex and return it. In the adjacency list, you store an empty array 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:

```
public func addDirectedEdge(from source: Vertex<T>,
                            to destination: Vertex<T>,
                            weight: Double?) {
  let edge = Edge(source: source,
                  destination: destination,
                  weight: weight)
  adjacencies[source]?.append(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 protocol extension on `Graph`.

In Graph.swift, add the following extension:

```
extension Graph {

  public func addUndirectedEdge(between source: Vertex<Eleme
                                and destination: Vertex<Elem
                                weight: Double?) {
    addDirectedEdge(from: source, to: destination, weight: 
    addDirectedEdge(from: destination, to: source, weight: 
  }
}
```

Adding an undirected edge is the same as adding two directed edges.

Now that you've implemented both `addDirectionalEdge` and `addUndirectedEdge`, you can implement `add` by delegating to one of these methods. In the same protocol extension, add:

```
public func add(_ edge: EdgeType, from source: Vertex<Elemen
                                  to destination: Vertex<Ele
                                  weight: Double?) {
  switch edge {
  case .directed:
    addDirectedEdge(from: source, to: destination, weight: w
  case .undirected:
    addUndirectedEdge(between: source, and: destination, wei
  }
}
```

The `add` method is a convenient helper method that creates either a directed or undirected edge.

This is where protocols can become very powerful! Anyone that adopts the `Graph` protocol only needs to implement `addDirectedEdge` in order to get `addUndirectedEdge` and `add` for free!

Retrieving the outgoing edges from a vertex

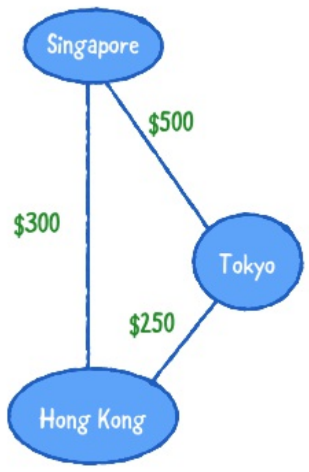
Back in AdjacencyList.swift, continue your work on conforming to `Graph` by adding the following method:

```
public func edges(from source: Vertex<T>) -> [Edge<T>] {
  return adjacencies[source] ?? []
}
```

This is a straightforward implementation: you either return the stored edges, or an empty array if the `source` vertex is unknown.

Retrieving the weight of an edge

How much is the flight from Singapore to Tokyo?



Add the following right after `edges(from:)`:

```
public func weight(from source: Vertex<T>,
                   to destination: Vertex<T>) -> Double? {
  return edges(from: source)
           .first { $0.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 you can print a nice description of your graph:

```
extension AdjacencyList: CustomStringConvertible {

  public var description: String {
    var result = ""
    for (vertex, edges) in adjacencies { // 1
      var edgeString = ""
      for (index, edge) in edges.enumerated() { // 2
        if index != edges.count - 1 {
          edgeString.append("\(edge.destination), ")
        } else {
                edgeString.append("\(edge.destination)")
             }
          }
          result.append("\(vertex) ---> [ \(edgeString) ]\n") /
       }
       return result
    }
 }
```

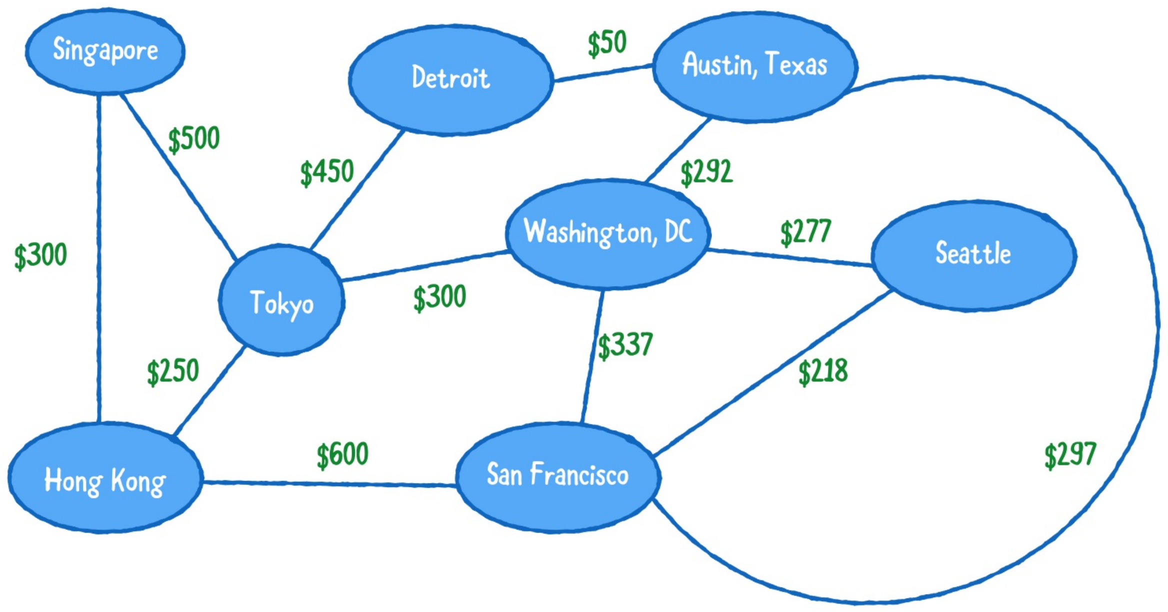
Here's what's going on in the code above:

1. You loop through every key-value pair in `adjacencies`.
2. For every vertex, you loop through all its outgoing edges and add an appropriate string to the output.
3. Finally, for every vertex you print both the vertex itself and its outgoing edges.

You have finally completed your first graph! Let's now 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 the main playground page, add the following code:

```
let graph = AdjacencyList<String>()

let singapore = graph.createVertex(data: "Singapore")
let tokyo = graph.createVertex(data: "Tokyo")
let hongKong = graph.createVertex(data: "Hong Kong")
let detroit = graph.createVertex(data: "Detroit")
let sanFrancisco = graph.createVertex(data: "San Francisco"
let washingtonDC = graph.createVertex(data: "Washington DC"
let austinTexas = graph.createVertex(data: "Austin Texas")
let seattle = graph.createVertex(data: "Seattle")

graph.add(.undirected, from: singapore, to: hongKong, weigh
graph.add(.undirected, from: singapore, to: tokyo, weight: 
graph.add(.undirected, from: hongKong, to: tokyo, weight: 2
graph.add(.undirected, from: tokyo, to: detroit, weight: 45
graph.add(.undirected, from: tokyo, to: washingtonDC, weigh
graph.add(.undirected, from: hongKong, to: sanFrancisco, we
graph.add(.undirected, from: detroit, to: austinTexas, weig
graph.add(.undirected, from: austinTexas, to: washingtonDC,
graph.add(.undirected, from: sanFrancisco, to: washingtonDC
graph.add(.undirected, from: washingtonDC, to: seattle, wei
graph.add(.undirected, from: sanFrancisco, to: seattle, wei
graph.add(.undirected, from: austinTexas, to: sanFrancisco,
print(graph.description)
```

You should get the following output in your playground:

```
2: Hong Kong ---> [ 0: Singapore, 1: Tokyo, 4: San Francisc
4: San Francisco ---> [ 2: Hong Kong, 5: Washington DC, 7: S
5: Washington DC ---> [ 1: Tokyo, 6: Austin Texas, 4: San Fr
6: Austin Texas ---> [ 3: Detroit, 5: Washington DC, 4: San
7: Seattle ---> [ 5: Washington DC, 4: San Francisco ]
0: Singapore ---> [ 2: Hong Kong, 1: Tokyo ]
1: Tokyo ---> [ 0: Singapore, 2: Hong Kong, 3: Detroit, 5: W
3: Detroit ---> [ 1: Tokyo, 6: Austin Texas ]
```

Pretty cool, huh? This shows a visual description of an adjacency list. You can clearly see all the outbound flights from any place!

You can also obtain other useful information such as:

- How much is a flight from Singapore to Tokyo?

```
graph.weight(from: singapore, to: tokyo)
```

- What are all the outgoing flights from San Francisco?

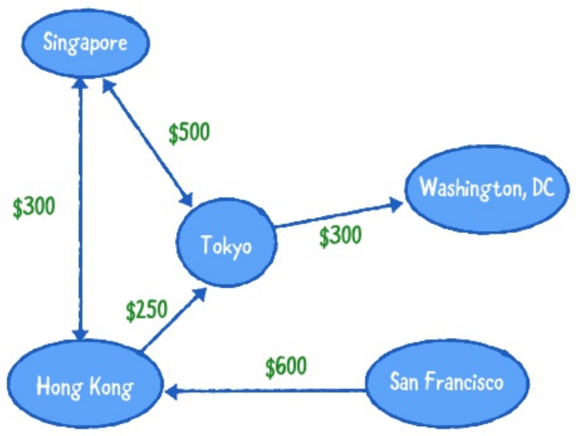
```
print("San Francisco Outgoing Flights:")
print("--------------------------------")
for edge in graph.edges(from: sanFrancisco) {
  print("from: \(edge.source) to: \(edge.destination)")
}
```

You have just created a graph using an adjacency list, where you used a dictionary to store the outgoing edges for every vertex. Let's take a look at a different approach to how to store vertices and edges.

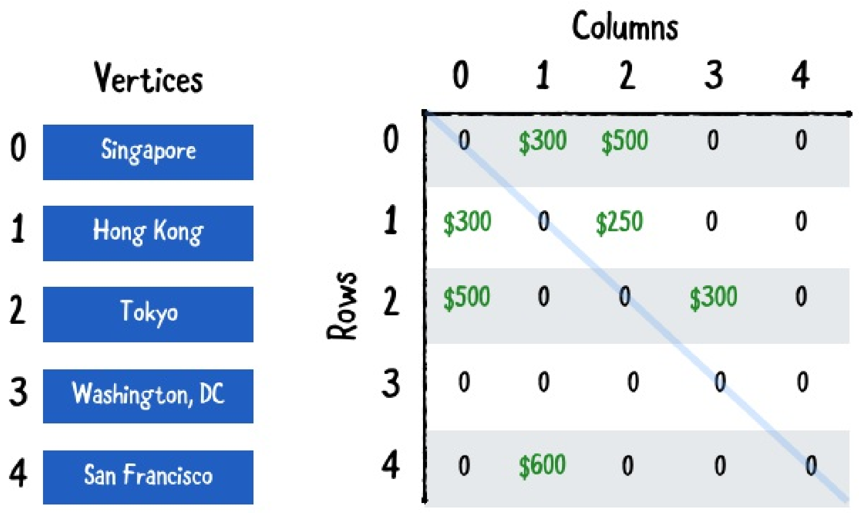
Adjacency matrix

An adjacency matrix uses a square matrix to represent a graph. This matrix is a two-dimensional array where 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.



	Vertices
0	Singapore
1	Hong Kong
2	Tokyo
3	Washington, DC
4	San Francisco

Rows \ Columns	0	1	2	3	4
0	0	$300	$500	0	0
1	$300	0	$250	0	0
2	$500	0	0	$300	0
3	0	0	0	0	0
4	0	$600	0	0	0

Compared to an adjacency list, this matrix is a little harder 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 is a blue 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.swift and add the following to it:

```
public class AdjacencyMatrix<T>: Graph {

  private var vertices: [Vertex<T>] = []
  private var weights: [[Double?]] = []

  public init() {}

  // 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 conformance to `Graph` but still need to implement the requirements.

Creating a Vertex

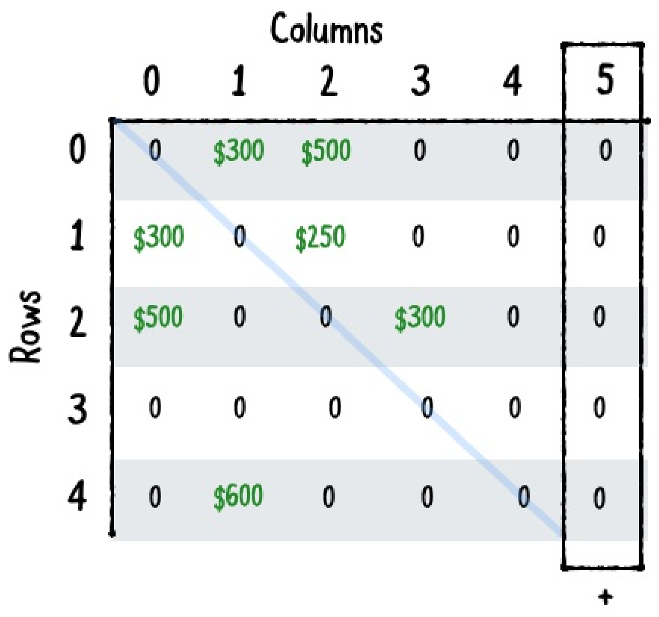
Add the following method to `AdjacencyMatrix`:

```
public func createVertex(data: T) -> Vertex<T> {
  let vertex = Vertex(index: vertices.count, data: data)
  vertices.append(vertex) // 1
  for i in 0..<weights.count { // 2
    weights[i].append(nil)
  }
  let row = [Double?](repeating: nil, count: vertices.count
  weights.append(row)
  return vertex
}
```

To create a vertex in an adjacency matrix, you:

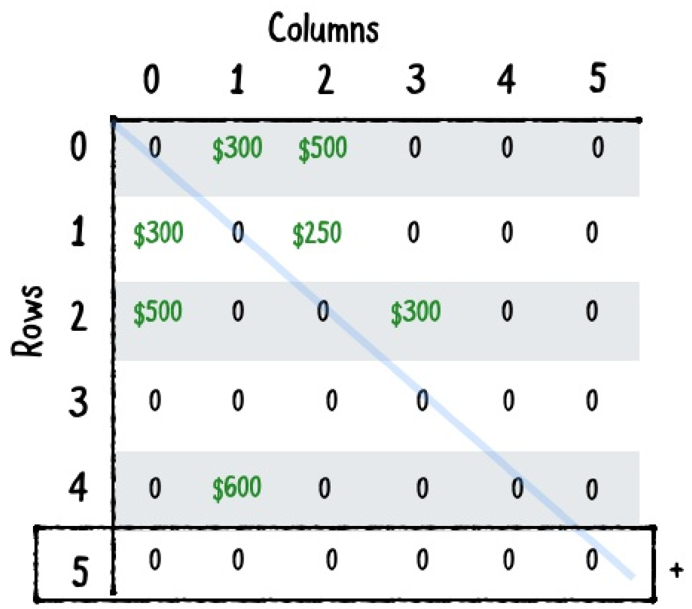
1. Add a new vertex to the array.

2. Append a `nil` weight to every row in the matrix, as none of the current vertices have an edge to the new vertex.



$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & \$300 & \$500 & 0 & 0 & 0 \\ 1 & \$300 & 0 & \$250 & 0 & 0 & 0 \\ 2 & \$500 & 0 & 0 & \$300 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & \$600 & 0 & 0 & 0 & 0 \end{array}$$

1. Add a new row to the matrix. This row holds the outgoing edges for the new vertex.



$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & \$300 & \$500 & 0 & 0 & 0 \\ 1 & \$300 & 0 & \$250 & 0 & 0 & 0 \\ 2 & \$500 & 0 & 0 & \$300 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & \$600 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Creating edges

Creating edges is as simple as filling in the matrix. Add the following method:

```
public func addDirectedEdge(from source: Vertex<T>,
                            to destination: Vertex<T>, weigl
  weights[source.index][destination.index] = weight
}
```

Remember that `addUndirectedEdge` and `add` have a default implementation in the protocol extension, so this is all you need to do!

Retrieving the outgoing edges from a vertex

Add the following method:

```
public func edges(from source: Vertex<T>) -> [Edge<T>] {
  var edges: [Edge<T>] = []
  for column in 0..<weights.count {
    if let weight = weights[source.index][column] {
      edges.append(Edge(source: source,
                        destination: vertices[column],
                        weight: weight))
    }
  }
  return edges
}
```

To retrieve the outgoing edges for a vertex, you search the row for this vertex in the matrix for weights that aren not `nil`. Every non-`nil` 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' i's very easy to get the weight of an edge; simply look up the value in the adjacency matrix. Add this method:

```
public func weight(from source: Vertex<T>,
                   to destination: Vertex<T>) -> Double? {
```

```
  return weights[source.index][destination.index]
}
```

Visualize an adjacency matrix

Finally, add the following extension so you can print out a nice, readable description of your graph:

```
extension AdjacencyMatrix: CustomStringConvertible {

  public var description: String {
    // 1
    let verticesDescription = vertices.map { "\($0)" }
                                      .joined(separator: "\
    // 2
    var grid: [String] = []
    for i in 0..<weights.count {
      var row = ""
      for j in 0..<weights.count {
        if let value = weights[i][j] {
          row += "\(value)\t"
        } else {
          row += "ø\t\t"
        }
      }
      grid.append(row)
    }
    let edgesDescription = grid.joined(separator: "\n")
    // 3
    return "\(verticesDescription)\n\n\(edgesDescription)"
  }
}
```

Here are the steps:

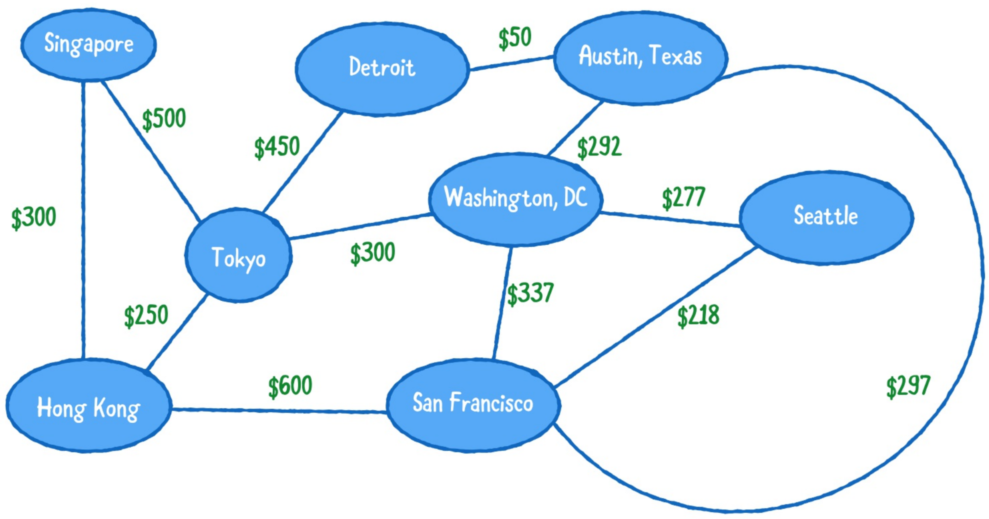
1. You first create a list of the vertices.

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 will be reusing the same example from `AdjacencyList`:



Go to the main playground page and replace:

```
let graph = AdjacencyList<String>()
```

with:

```
let graph = AdjacencyMatrix<String>()
```

`AdjacencyMatrix` and `AdjacencyList` conform to the same protocol `Graph`, so the rest of the code stays the same.

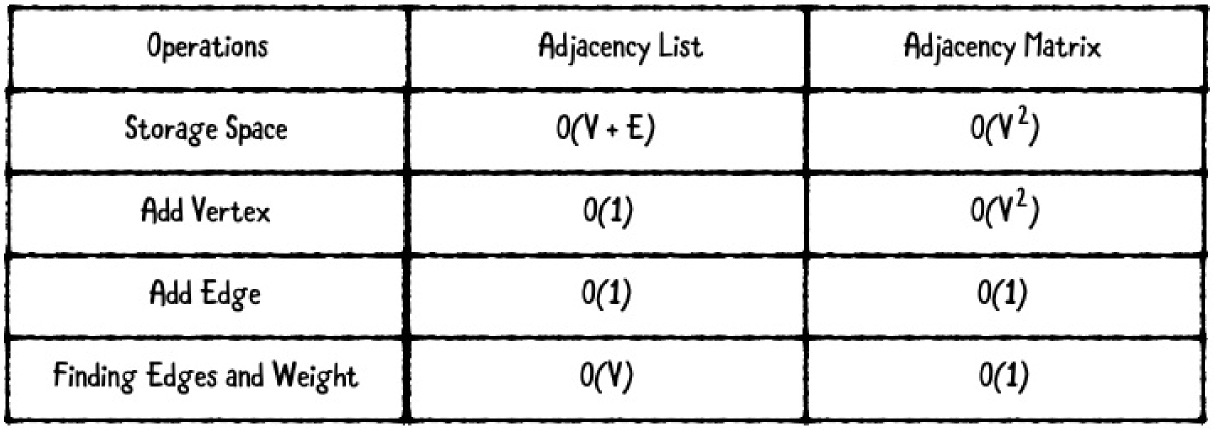
You should get the following output in your playground:

```
0: Singapore
1: Tokyo
2: Hong Kong
3: Detroit
4: San Francisco
5: Washington DC
6: Austin Texas
7: Seattle
```

```
ø       500.0   300.0   ø       ø       ø       ø       ø
500.0   ø       250.0   450.0   ø       300.0   ø       ø
300.0   250.0   ø       ø       600.0   ø       ø       ø
ø       450.0   ø       ø       ø       ø       50.0    ø
ø       ø       600.0   ø       ø       337.0   297.0   218
ø       300.0   ø       ø       337.0   ø       292.0   277
ø       ø       ø       50.0    297.0   292.0   ø       ø
ø       ø       ø       ø       218.0   277.0   ø       ø
San Francisco Outgoing Flights:
--------------------------------
from: 4: San Francisco to: 2: Hong Kong
from: 4: San Francisco to: 5: Washington DC
from: 4: San Francisco to: 6: Austin Texas
from: 4: San Francisco to: 7: Seattle
```

In terms of visual beauty, an adjacency list is a lot easier to follow and trace than an adjacency matrix. Let's analyze the common operations of these two approaches and see how they perform.

Graph analysis



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 dictionary. It is amortized as O(1). When adding a vertex to an adjacency matrix, you are 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, 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 simply 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 is 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 lots 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.

Where to go from here?

In this chapter you learned about different types of graphs representations and how these can be applied to real world applications. You also constructed graphs in two different ways, using an adjacency list or an adjacency matrix.

You have only scratched the surface of graphs so far. There are many more algorithms that can be applied to graph data structures. You will go through some of these in the upcoming chapters!

Breadth-First Search

In the previous chapter, you explored how graphs can be used to capture relationships between objects. Remember that objects are just 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.

BFS can be used to solve a wide variety of problems:

1. Generating a minimum spanning tree.

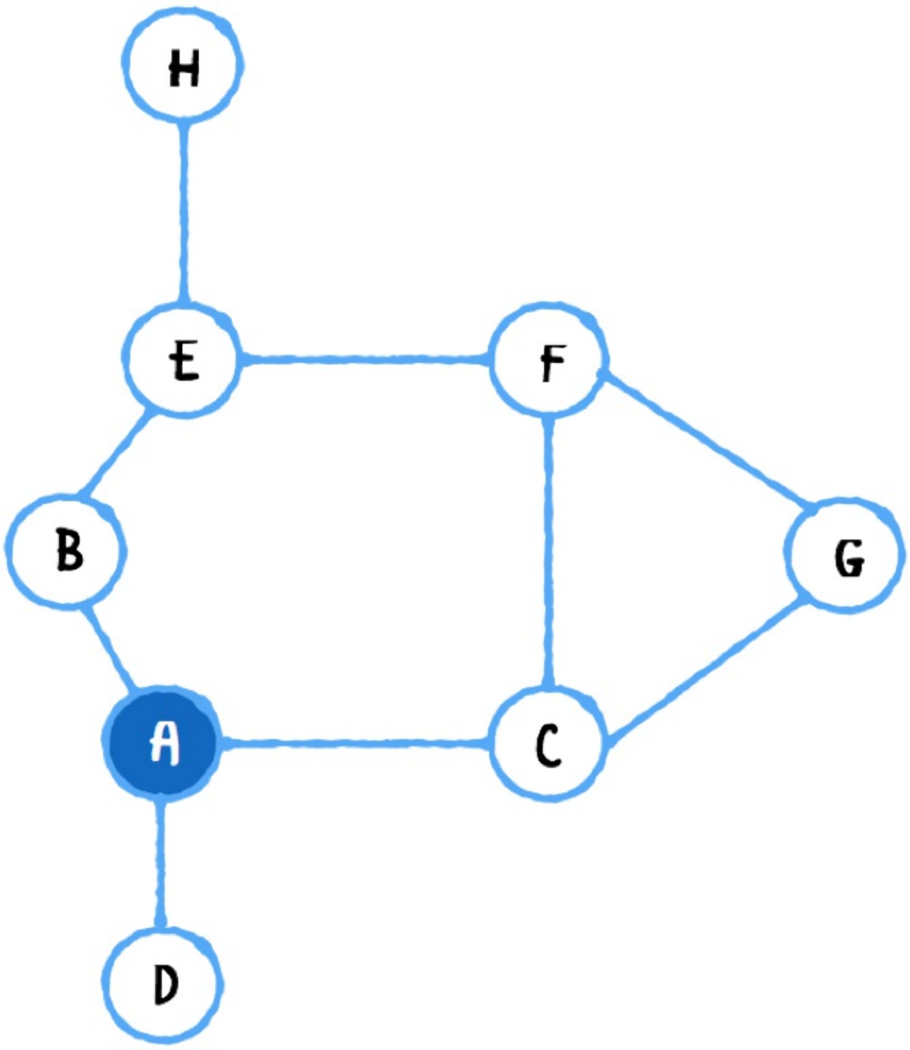
2. Finding potential paths between vertices.

3. Finding the shortest path between two vertices.

Example

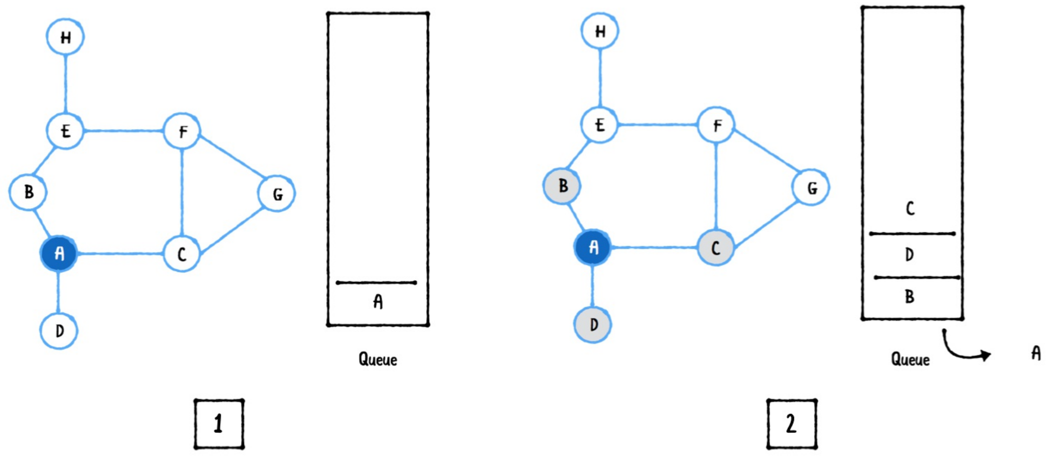
BFS starts off by selecting any vertex in a graph. The algorithm then explores all neighbors of this vertex before traversing the neighbors of said neighbors, and so forth. As the name suggests, this algorithm takes a breadth-first approach.

Let's go through a BFS example using the undirected graph below:



Note: Highlighted vertices represent vertices that have been visited.

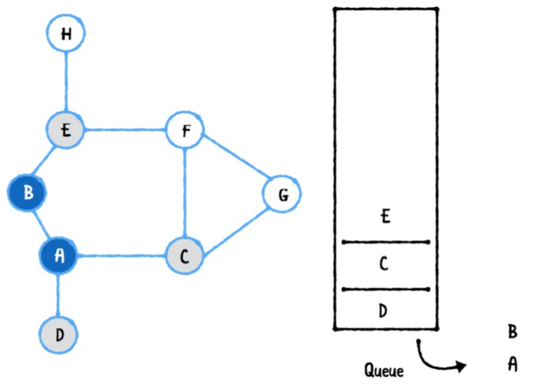
You will use a queue to keep track of which vertices to visit next. The first in first out approach of the queue guarantees that all of a vertex's neighbors are visited before you traverse one level deeper.



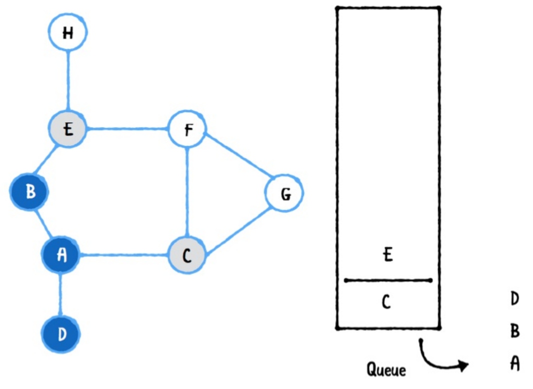
1. To begin, you pick a source vertex to start from. Here you have chosen `A`, which is added to the queue.

2. As long as the queue is not empty, you dequeue and visit the next vertex, in this case `A`. Next, you add all of `A`'s neighboring vertices `[B, D, C]` to the queue.

Note: It's important to note that you only add a vertex to the queue when it has not yet been visited and is not already in the queue.



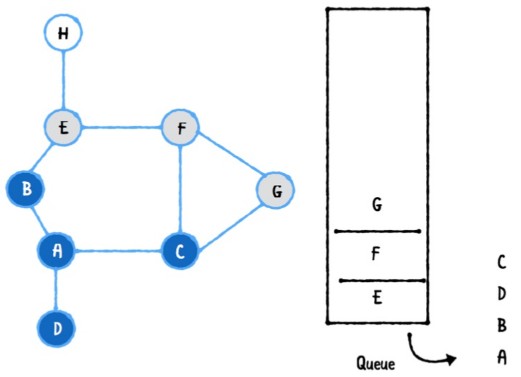
3



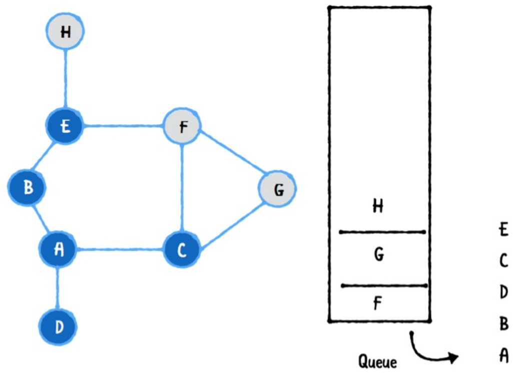
4

1. The queue is not empty, so you dequeue and visit the next vertex, which is `B`. You then add `B`'s neighbor `E` to the queue. `A` is already visited so it does not get added. The queue now has `[D, C, E]`.

2. 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

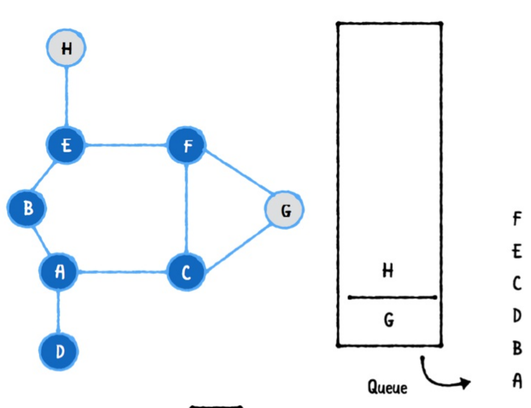


6

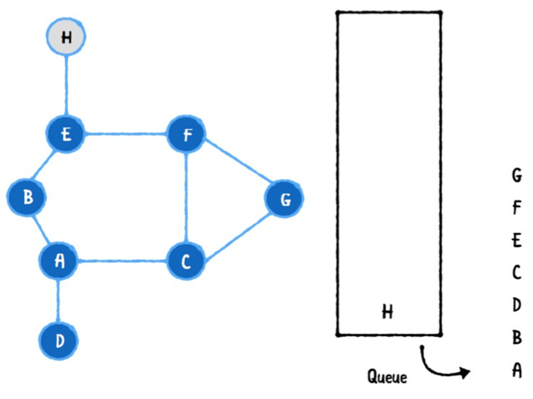
1. Next, you dequeue `C` and add its neighbors `[F, G]` to the queue. The queue now has `[E, F, G]`.

Note that you have now visited all of `A`'s neighbors! BFS now moves on to the second level of neighbors.

1. You 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.



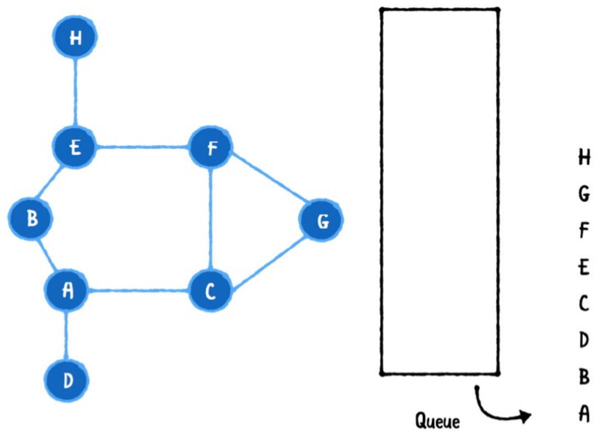
7



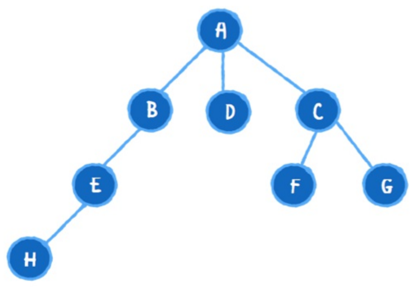
8

1. You dequeue F, and since all its neighbors are already in the queue or visited, you don't add anything to the queue.

2. Just like the previous step, you deque G and don't add anything to the queue.



9



10

1. Finally, you dequeue `H`. The breadth-first search is complete since the queue is now empty!

2. 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 up 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 will use to implement BFS.

In your main playground file, you will notice a pre-built sample graph. Add the following below:

```
extension Graph {

  func breadthFirstSearch(from source: Vertex<Element>)
      -> [Vertex<Element>] {
    var queue = QueueStack<Vertex<Element>>()
    var enqueued: Set<Vertex<Element>> = []
    var visited: [Vertex<Element>] = []

    // more to come

    return visited
  }
}
```

Here you've defined a method `breadthFirstSearch(from:)` that 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 enqueued before so you don't enqueue the same vertex twice.

3. `visited` is an array that stores the order in which the vertices were explored.

Next, complete the method by replacing the comment with:

```
queue.enqueue(source) // 1
enqueued.insert(source)

while let vertex = queue.dequeue() { // 2
  visited.append(vertex) // 3
  let neighborEdges = edges(from: vertex) // 4
  neighborEdges.forEach { edge in
    if !enqueued.contains(edge.destination) { // 5
      queue.enqueue(edge.destination)
      enqueued.insert(edge.destination)
    }
  }
}
```

Here's what's going on:

1. You initiate the BFS algorithm by first enqueuing 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. Then, you 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! Let's give this algorithm a spin. Add the following code:

```
let vertices = graph.breadthFirstSearch(from: a)
vertices.forEach { vertex in
  print(vertex)
}
```

Notice the order of the explored vertices using breadth first search:

```
0: A
1: B
2: C
3: D
4: E
5: F
6: G
7: 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 breadth-first search, each vertex is enqueued once. This has a time complexity of O(V) . During this traversal, you also visit all the the edges. The time it takes to visit all edges is O(E) . This means 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`.

Where to go from here?

Breadth-first search is an algorithm for traversing or searching a graph. 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 or path. Also it's good for generating a minimum spanning tree as you will see in Chapter 23.

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 will look at depth-first search, another algorithm for traversing or searching a graph.

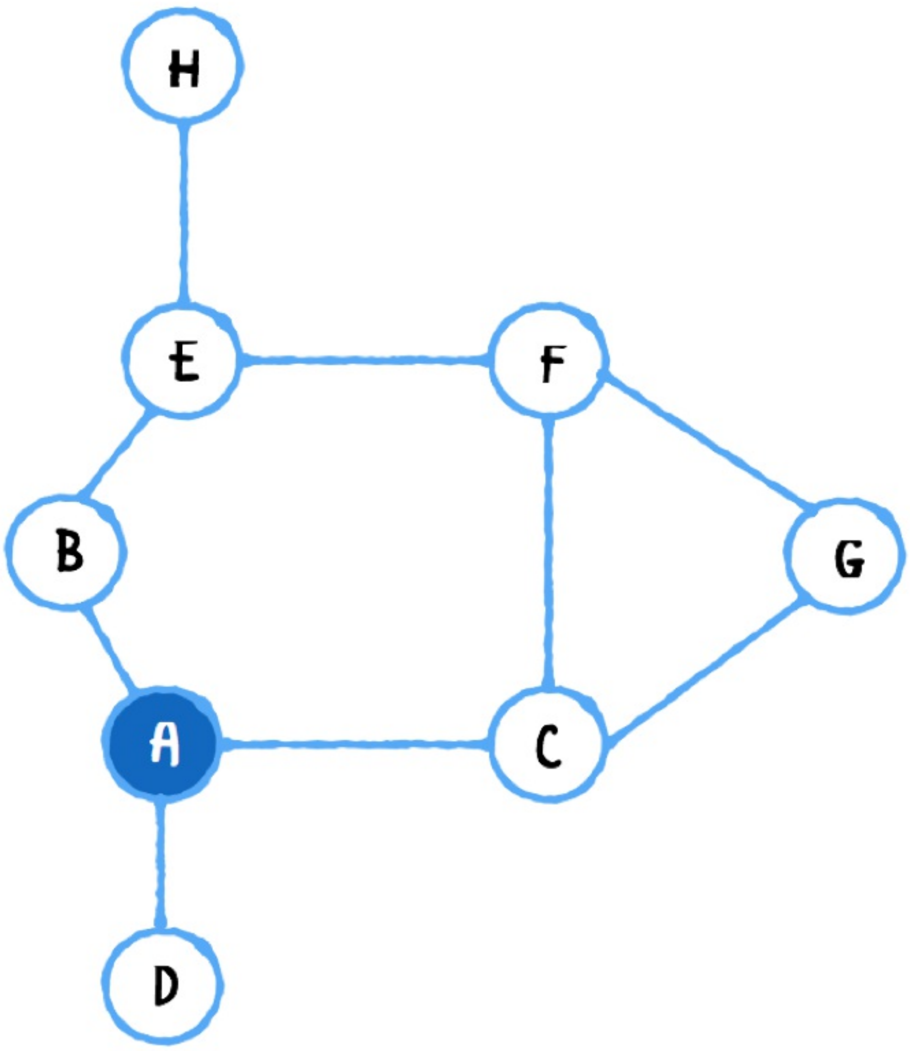
There are a lot of applications for depth-first search:

- Topological sorting
- Detecting a cycle
- Path finding, such as in maze puzzles
- Finding connected components in a sparse graph

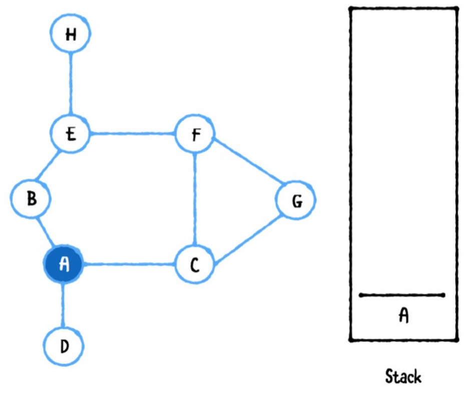
To perform a depth-first search (DFS), you start with a given source vertex and attempt to explore a branch as far as possible until you reach the end. At this point, you would backtrack (move a step back) and explore the next available branch until you find what you are looking for, or you've visited all the vertices.

Example

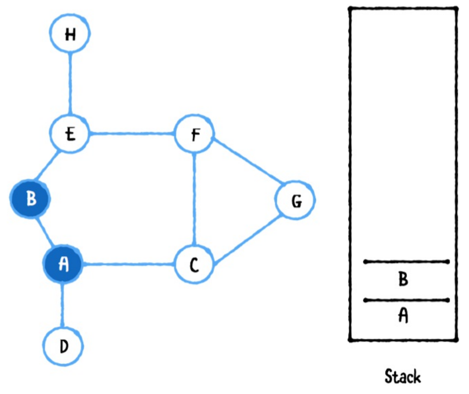
Let's go through a DFS example. The example graph below is exactly the same as the previous chapter. This is so you can see the difference between BFS and DFS.



You will use a stack to keep track of the levels you move through. The stack's last-in-first-out approach helps with backtracking. Every push on the stack means you move one level deeper. You can pop to return to a previous level if you reach a dead end.



1

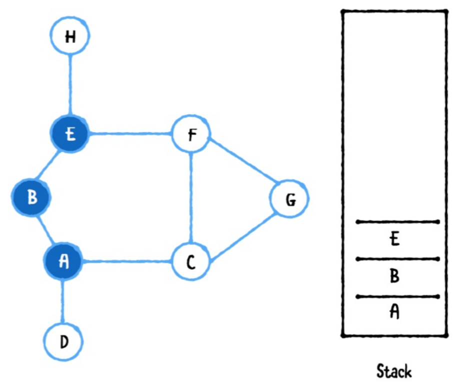


2

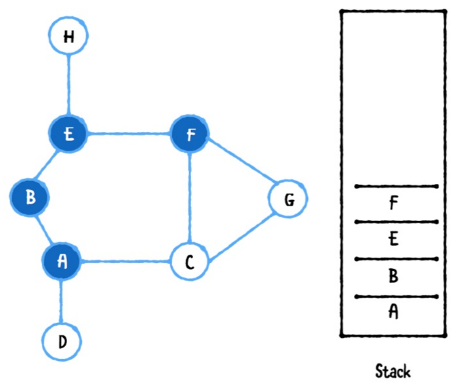
1. As in the previous chapter, you choose `A` as a starting vertex and add it to the stack.

2. As long as the stack is not empty, you visit the top vertex on the stack and push the first neighboring vertex that has yet to be 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.



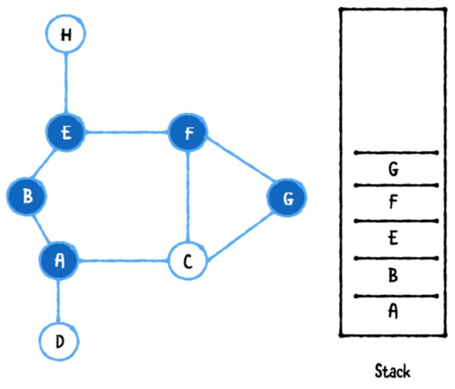
3



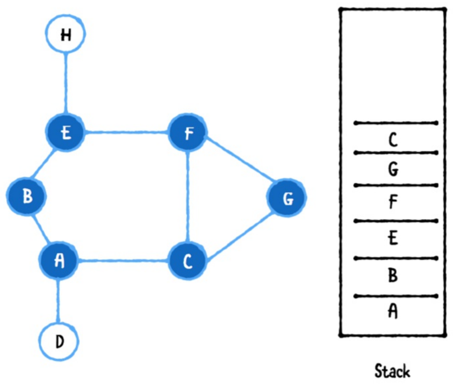
4

1. You visit B and push E because A is already visited.
2. You visit E and push F.

Note that every time you push on the stack, you advance further down a branch. Instead of visiting every adjacent vertex, you simply continue down a path until you reach the end and then backtrack.

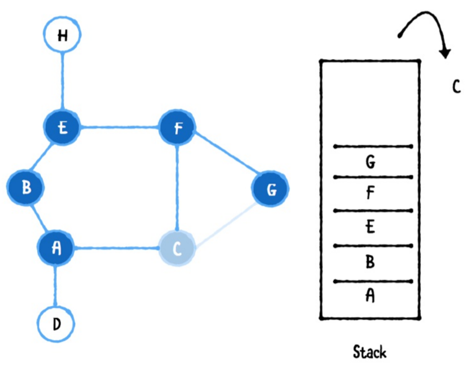


5

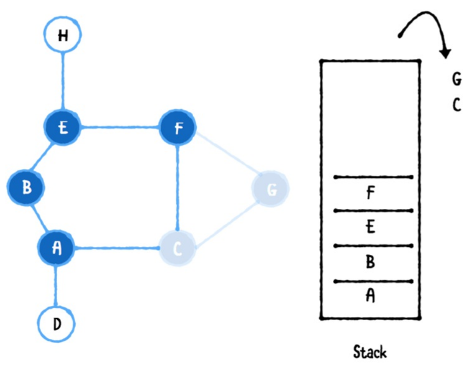


6

1. You visit F and push G.
2. You visit G and push C.



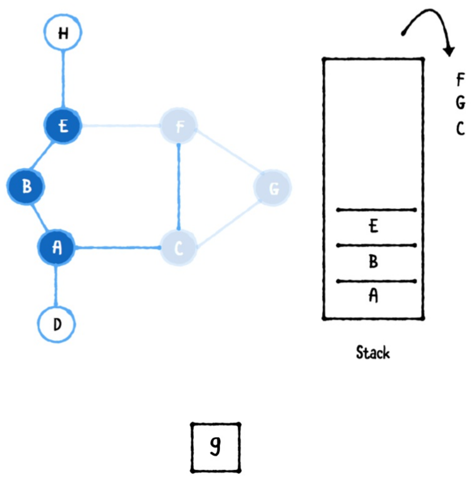
7

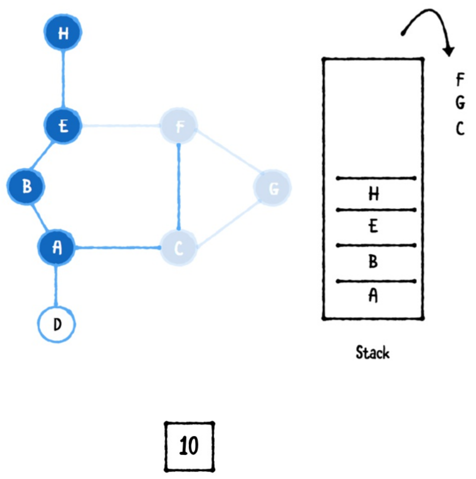


8

1. The next vertex to visit is `C`. It has neighbors `[A, F, G]`, but all of these have been visited. You have reached a dead end, so it's time to backtrack by popping `C` off the stack.

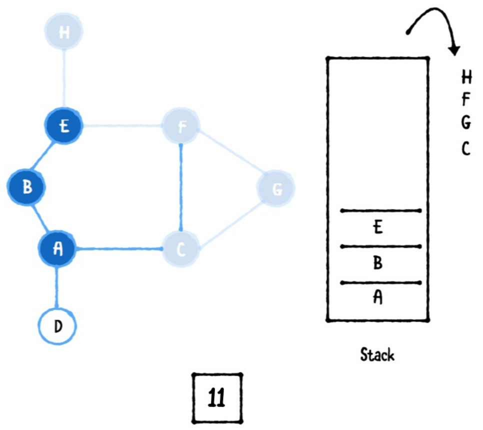
2. This brings you back to `G`. It has neighbors `[F, C]`, but all of these have been visited. Another dead end, pop `G`.

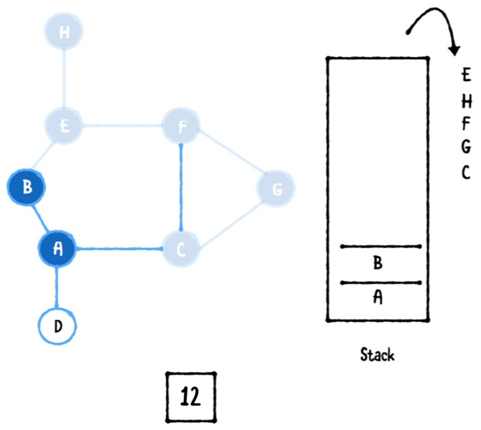




1. `F` also has no unvisited neighbors remaining, so pop `F`.

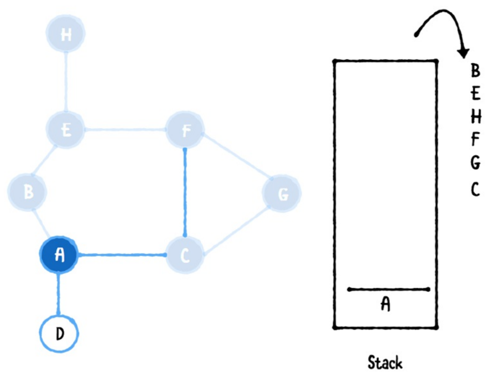
2. Now you're back at `E`. Its neighbor `H` is still unvisited, so you push `H` on the stack.



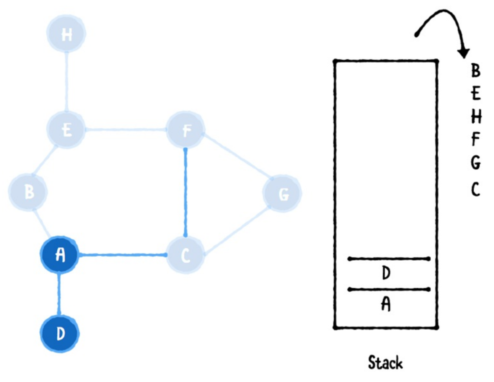


1. Visiting `H` results in another dead end, so pop `H`.

2. `E` also doesn't have any available neighbors, so pop it.



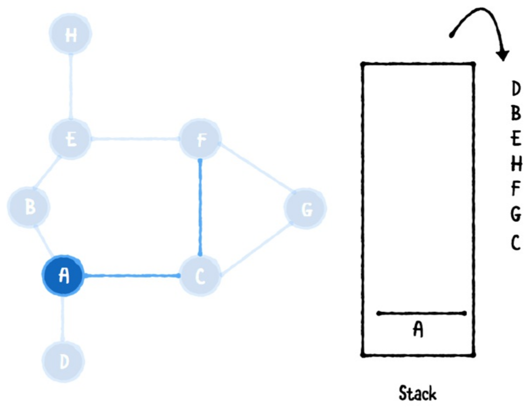
13



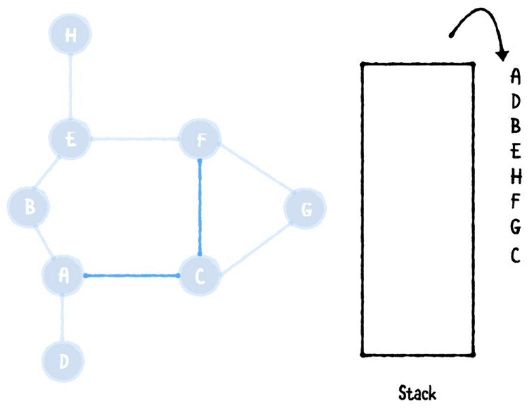
14

1. The same is true for `B`, so pop `B`.

2. This brings you all the way back to `A`, whose neighbor `D` still needs to be visited, so you push `D` on the stack.



15

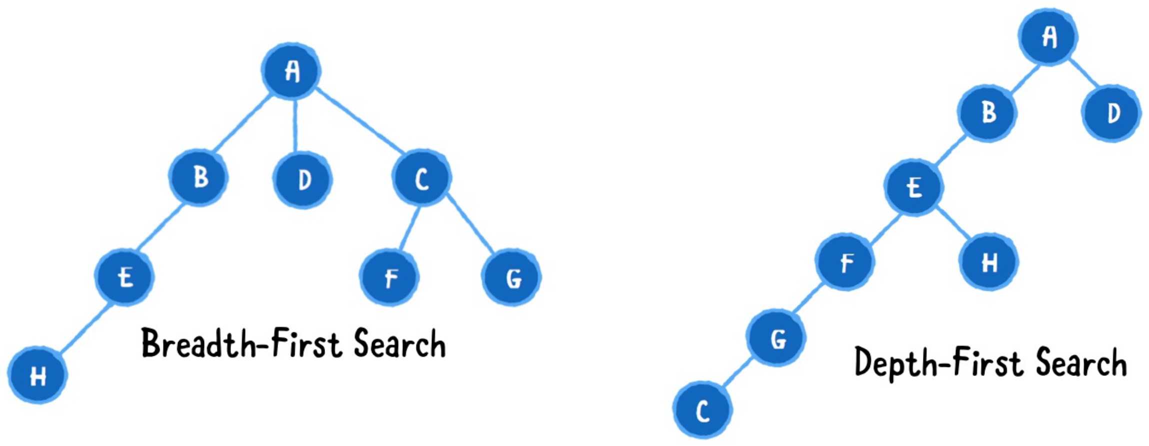


16

1. Visiting `D` results in another dead end, so pop `D`.

2. 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 depth-first search 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, compared to BFS.



Implementation

Open up the starter playground for this chapter. This playground contains an implementation of a graph that as well as a stack which you'll use to implement DFS.

In your main playground file, you will notice a pre-built sample graph. Add the following below:

```
extension Graph {

  func depthFirstSearch(from source: Vertex<Element>)
      -> [Vertex<Element>] {
    var stack: Stack<Vertex<Element>> = []
    var pushed: Set<Vertex<Element>> = []
    var visited: [Vertex<Element>] = []

    stack.push(source)
    pushed.insert(source)
    visited.append(source)

    // more to come ...
    return visited
  }
}
```

Here you've defined a method `depthFirstSearch(from:)`, which takes in a starting vertex and returns a list of vertices in the order they were visited. It uses three data structures:

1. `stack` is used to store your path through the graph.

1. `pushed` remembers which vertices have been pushed before so you don't visit the same vertex twice. It is a `Set` to ensure fast O(1) lookup.

2. `visited` is an array that stores the order in which the vertices were visited.

To start the algorithm, you add the `source` vertex to all three.

Next, complete the method by replacing the comment with:

```
outer: while let vertex = stack.peek() { // 1
  let neighbors = edges(from: vertex) // 2
  guard !neighbors.isEmpty else { // 3
    stack.pop()
    continue
  }
  for edge in neighbors { // 4
    if !pushed.contains(edge.destination) {
      stack.push(edge.destination)
      pushed.insert(edge.destination)
      visited.append(edge.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 have labeled this loop `outer` so 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` array. It may seem a bit premature to mark this vertex as visited (you haven't peeked at 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 you've 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 try out your code, add the following to the playground:

```
let vertices = graph.depthFirstSearch(from: a)
vertices.forEach { vertex in
  print(vertex)
}
```

Notice the order of the visited nodes using depth-first search:

```
0: A
1: B
4: E
5: F
6: G
2: C
7: H
3: D
```

Performance

Depth-first search will visit every single 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 available to visit. The time complexity of this is O(E) , because in the worst case, you have to visit every single 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) since you have to store vertices in three separate data structures: `stack`, `pushed` and `visited`.

Where to go from here?

Depth-first search is another algorithm to traverse or search a graph. You have learned to leverage a stack to build DFS, how to use the stack to keep track of how deep you are in a graph, and why it's a good idea to keep a history of vertices in the event you need to backtrack.

In this chapter, you created an iterative version of DFS. However, a recursive implementation is also possible. Check out the recursive version on the Swift Algorithm Club repo at https://github.com/raywenderlich/swift-algorithm-club/tree/master/Depth-First%20Search.

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. A greedy algorithm constructs a solution step-by-step, and picks the most optimal path at every step. In particuar 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.

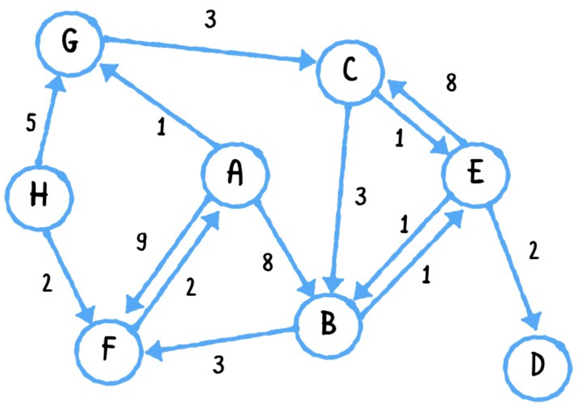
2. Telephone networks: Routing calls to highest-bandwidth paths available in the network.

3. Mapping: Finding the shortest and fastest paths are important to pretty much everyone who travels.

Example

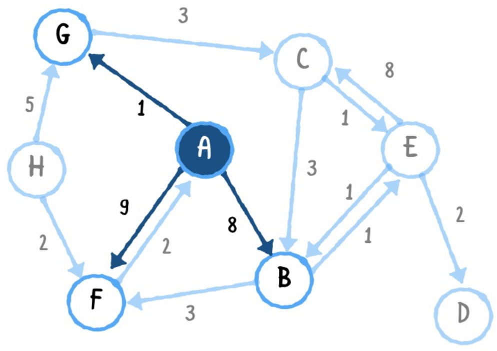
All the graphs you have looked at thus far have been undirected graphs. Let’s change it up a little and work with a directed graph!

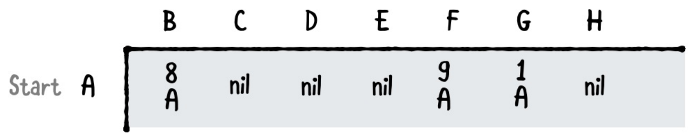
Imagine the directed graph below represents a GPS network. The vertices represent physical locations, and the edges between the vertices represent one way paths of a given cost between locations.



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.

First pass





	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil

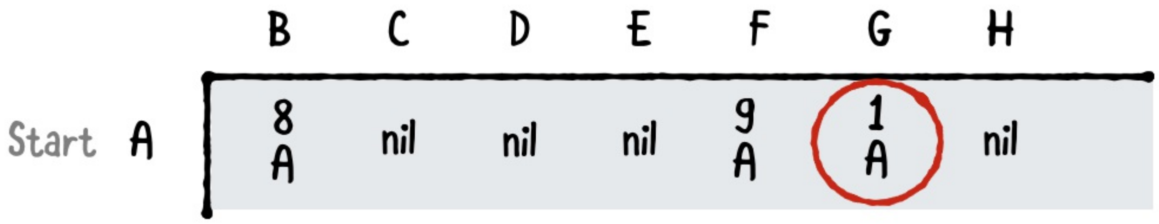
From vertex A, look at all outgoing edges. In this case, you have three edges:

- 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.

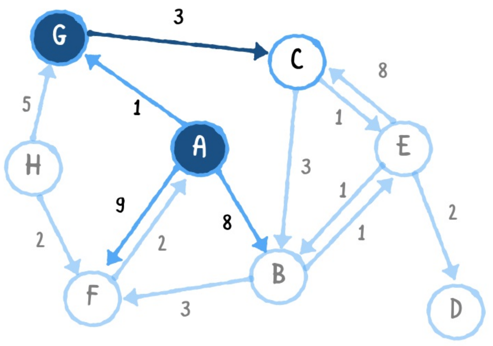
As you work through this example, the table on the right of the graph will represent a history, or record, of Dijkstra's algorithm at each stage. Each pass of the algorithm will add a row to the table. The last row in the table will be the final output of the algorithm.

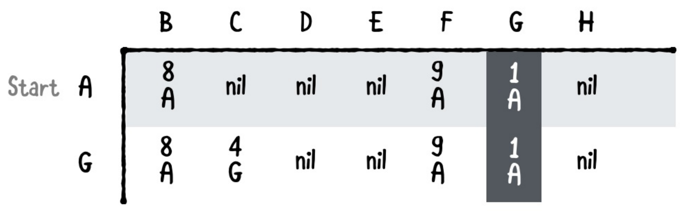
Second pass



	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 have 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.



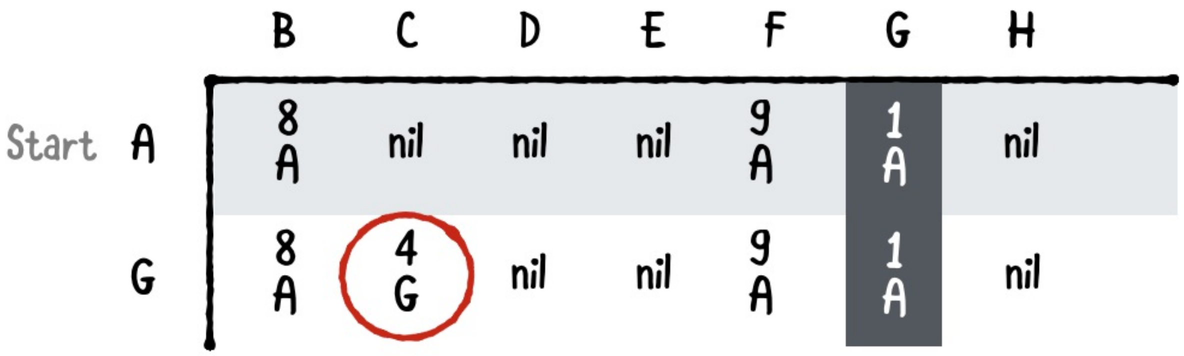


	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil
G	8 A	4 G	nil	nil	9 A	1 A	nil

Now from the lowest cost path, vertex G, look at all the outgoing edges. There is 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.

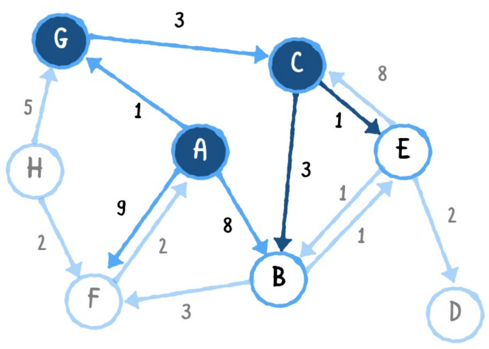
Every value in the output table has two parts: the total cost to reach that vertex, and the last neighbor on the path to that vertex. For example, the value 4 G in the column for vertex C means that the cost to reach C is 4, and the path to C goes through G. A value of `nil` indicates that no path has been discovered to that vertex.

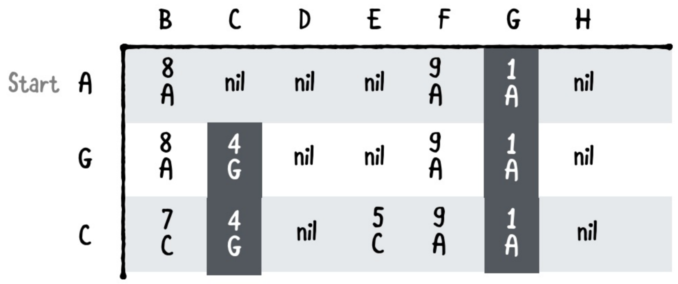
Third pass



		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:

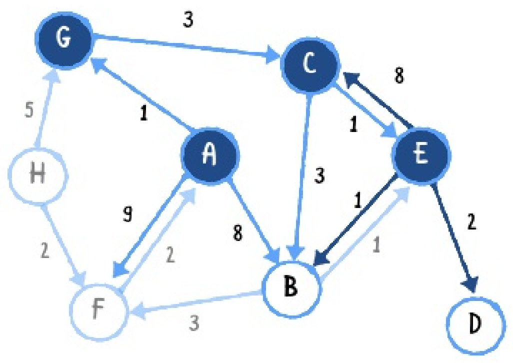
- C to E has a total cost of 4 + 1 = 5.
- C to B has a total cost of 4 + 3 = 7.

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.



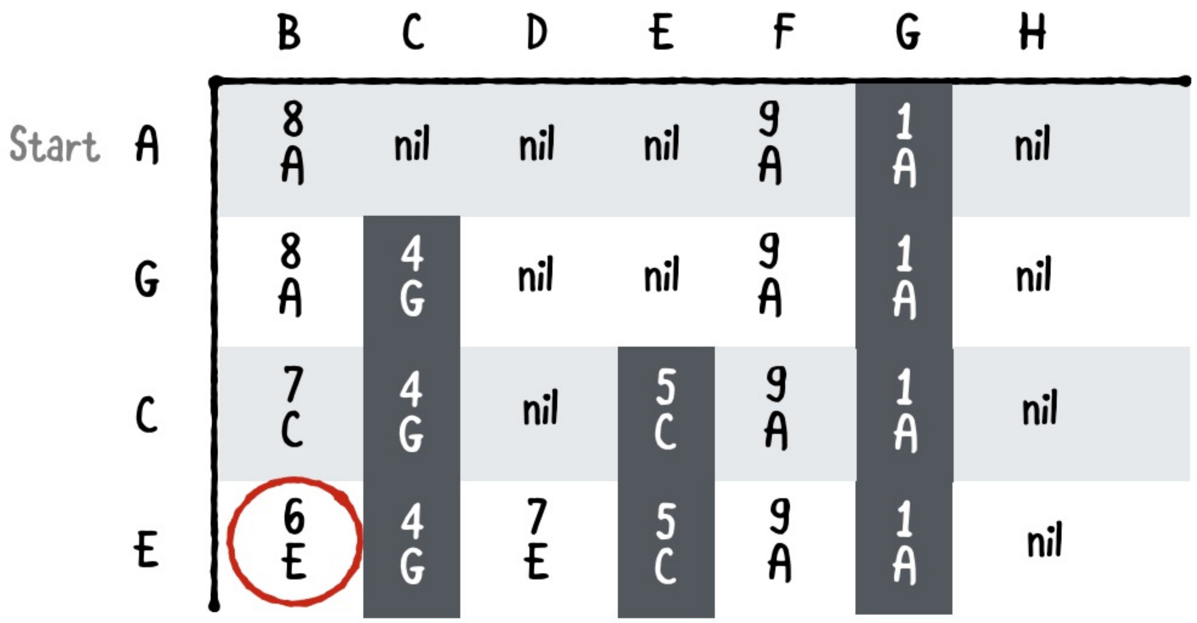
		B	C	D	E	F	G	H
Start	A	8 A	nil	nil	nil	9 A	1 A	nil
	G	8 A	4 G	nil	nil	9 A	1 A	nil
	C	7 C	4 G	nil	5 C	9 A	1 A	nil
	E	6 E	4 G	7 E	5 C	9 A	1 A	nil

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 have found the shortest path to C already, disregard this path.
- E to D has a total cost of 5 + 2 = 7.

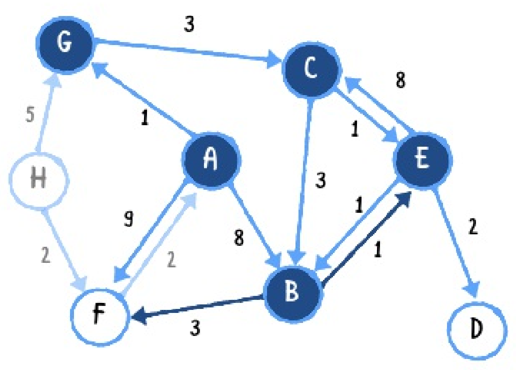
- E to B has a total cost of 5 + 1 = 6. According to the table, the current shortest path to B has a total cost of 7. You update the shortest path to be from E to B since it has a smaller cost of 6.

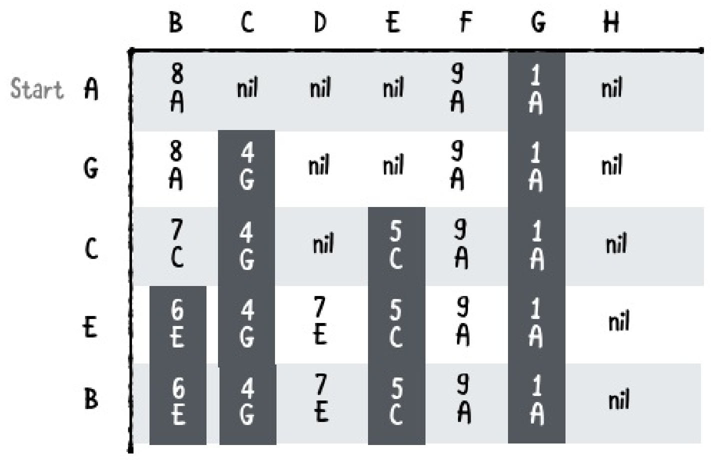
Fifth pass



	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.



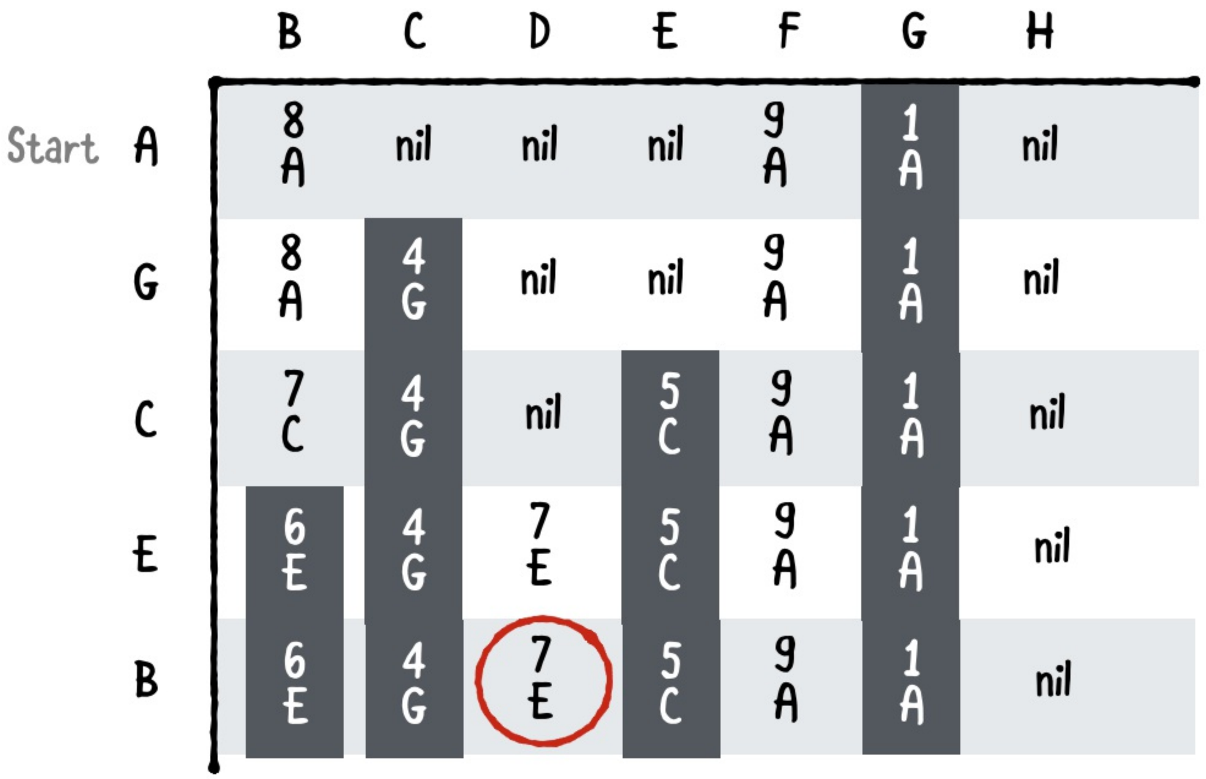


	B	C	D	E	F	G	H
Start A	8 A	nil	nil	nil	9 A	1 A	nil
G	8 A	4 G	nil	nil	9 A	1 A	nil
C	7 C	4 G	nil	5 C	9 A	1 A	nil
E	6 E	4 G	7 E	5 C	9 A	1 A	nil
B	6 E	4 G	7 E	5 C	9 A	1 A	nil

B has these outgoing edges:

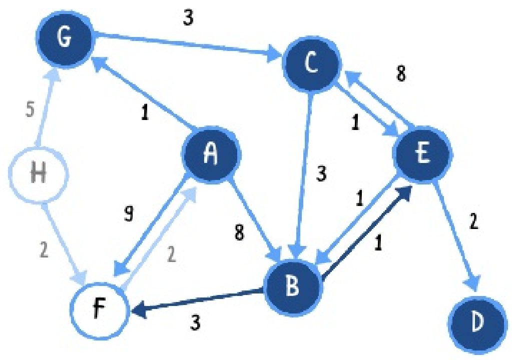
- B to E has a total cost of 6 + 1 = 7, but you've already found the shortest path to E, so disregard this path.
- B to F has a total cost of 6 + 3 = 9. From the table, you can tell that the current path to F from A also has a cost of 9. You can disregard this path since it isn't any shorter.

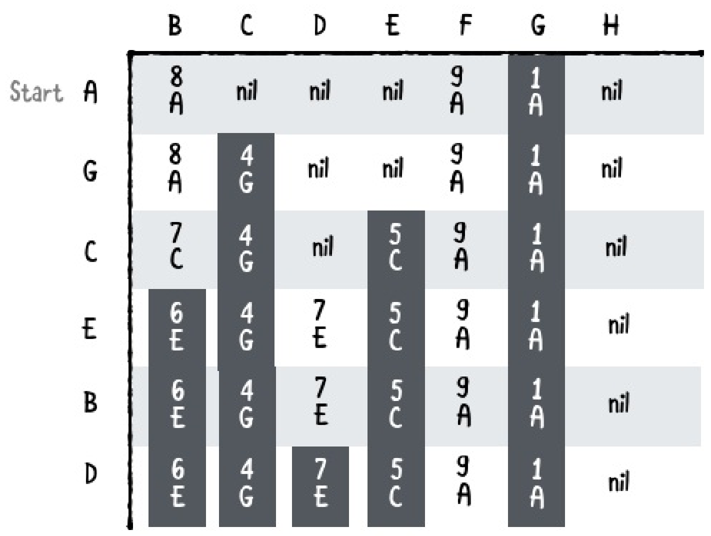
Sixth pass



		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.

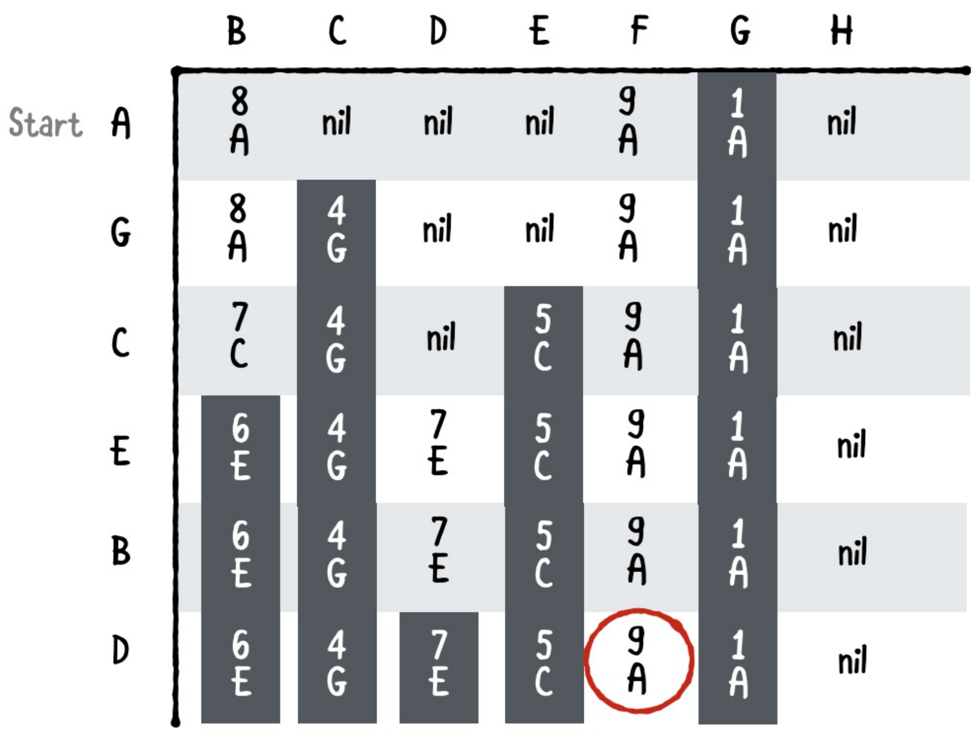




		B	C	D	E	F	G	H
Start	A	8 A	nil	nil	nil	9 A	1 A	nil
	G	8 A	4 G	nil	nil	9 A	1 A	nil
	C	7 C	4 G	nil	5 C	9 A	1 A	nil
	E	6 E	4 G	7 E	5 C	9 A	1 A	nil
	B	6 E	4 G	7 E	5 C	9 A	1 A	nil
	D	6 E	4 G	7 E	5 C	9 A	1 A	nil

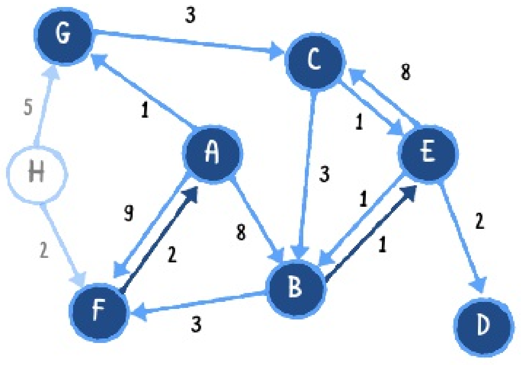
However D has no outgoing edges, so it’s a dead end. You simply record that you’ve found the shortest path to D and move on.

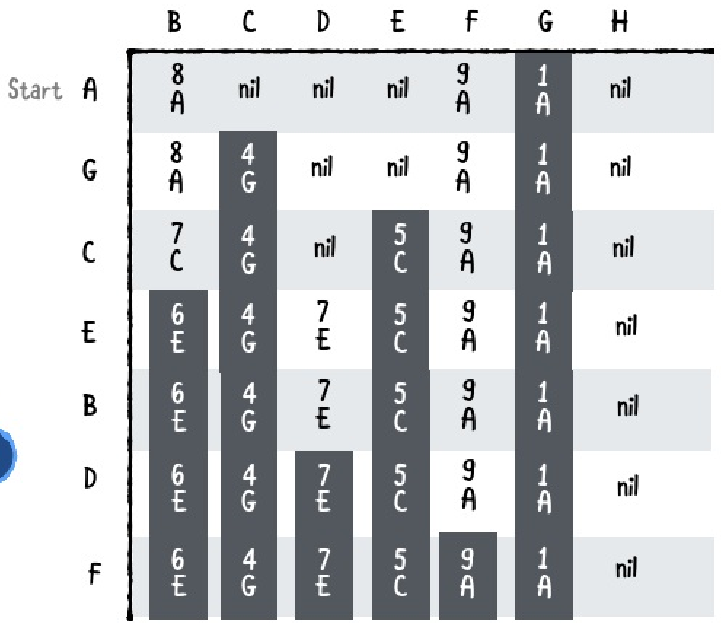
Seventh pass



		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.



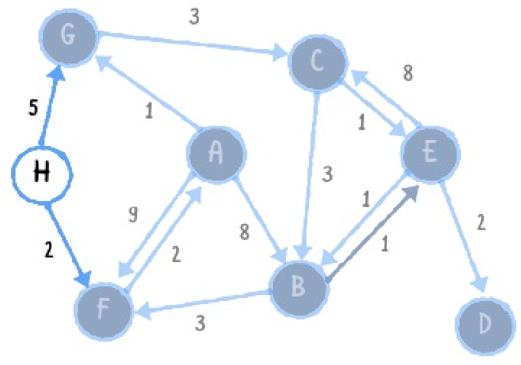


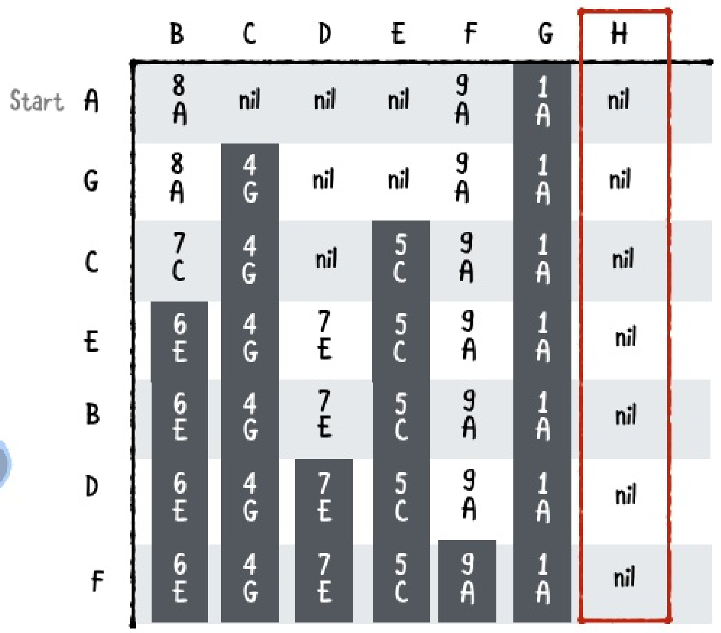
		B	C	D	E	F	G	H
Start	A	8 A	nil	nil	nil	9 A	1 A	nil
	G	8 A	4 G	nil	nil	9 A	1 A	nil
	C	7 C	4 G	nil	5 C	9 A	1 A	nil
	E	6 E	4 G	7 E	5 C	9 A	1 A	nil
	B	6 E	4 G	7 E	5 C	9 A	1 A	nil
	D	6 E	4 G	7 E	5 C	9 A	1 A	nil
	F	6 E	4 G	7 E	5 C	9 A	1 A	nil

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

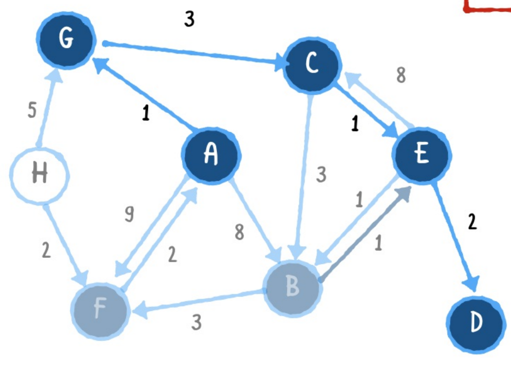
You have covered every vertex except for H. H has two outgoing edges to G and F. However, there is no path from A to H. This is why the whole column for H is `nil`.

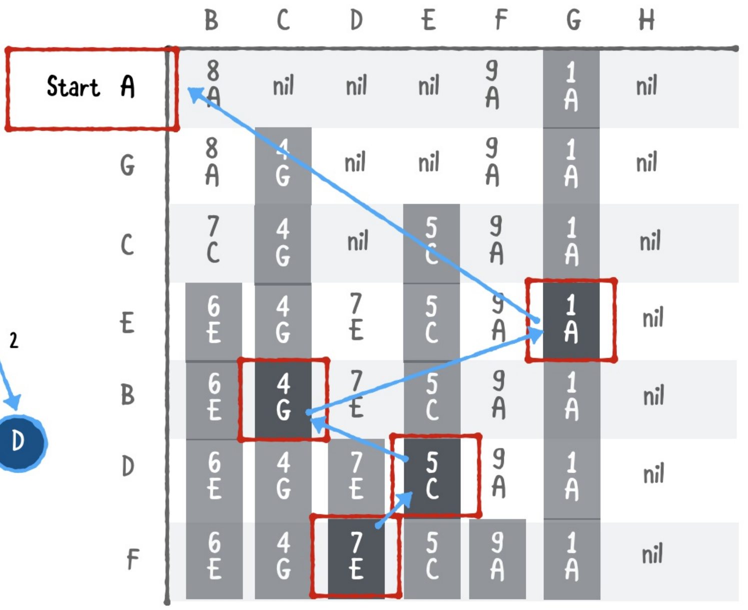




		B	C	D	E	F	G	H
Start	A	8 A	nil	nil	nil	9 A	1 A	nil
	G	8 A	4 G	nil	nil	9 A	1 A	nil
	C	7 C	4 G	nil	5 C	9 A	1 A	nil
	E	6 E	4 G	7 E	5 C	9 A	1 A	nil
	B	6 E	4 G	7 E	5 C	9 A	1 A	nil
	D	6 E	4 G	7 E	5 C	9 A	1 A	nil
	F	6 E	4 G	7 E	5 C	9 A	1 A	nil

This completes Dijkstra's algorithm, since all the vertices have been visited!





		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

You can now check the final row for the shortest paths and their cost. 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 will 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.

Open up Dijkstra.swift and add the following:

```
public enum Visit<T: Hashable> {
  case start // 1
  case edge(Edge<T>) // 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.

Now define a class called `Dijkstra`. Add the following after the code you added above:

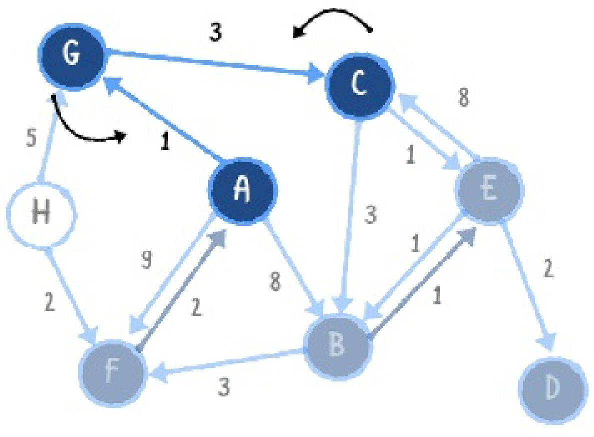
```
public class Dijkstra<T: Hashable> {

  public typealias Graph = AdjacencyList<T>
  let graph: Graph

  public init(graph: Graph) {
    self.graph = graph
  }
}
```

As in the previous chapter, `Graph` is defined as a type alias for `AdjacencyList`. You could in the future replace this with an adjacency matrix if needed.

Helper methods

Before building `Dijkstra`, let's create some helper methods that will help create the algorithm.

Tracing back to the start



C to G to A

You need a mechanism to keep track of the total weight from the current vertex back to the start vertex. To do this, you will keep track of a dictionary named `paths` that stores a `Visit` state for every vertex.

Add the following method to class `Dijkstra`:

```
private func route(to destination: Vertex<T>,
                   with paths: [Vertex<T> : Visit<T>]) -> [
  var vertex = destination // 1
  var path: [Edge<T>] = [] // 2
  while let visit = paths[vertex], case .edge(let edge) = vi
    path = [edge] + path // 4
    vertex = edge.source // 5
  }
  return path // 6
}
```

This method takes in the `destination` vertex along with a dictionary of existing `paths`, and constructs a path that leads to the `destination` vertex. Going over the code:

1. Start at the `destination` vertex.

2. Create an array of edges to store the path.

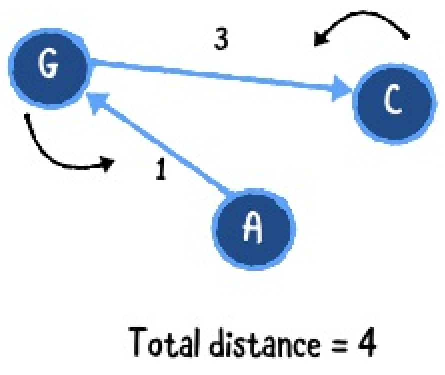
3. As long as you have not reached the `source` case, continue to extract the next `edge`.

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 have completed the path and return it.

Calculating total distance



Once you have 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 func distance(to destination: Vertex<T>,
                      with paths: [Vertex<T> : Visit<T>]) ->
  let path = route(to: destination, with: paths) // 1
  let distances = path.flatMap { $0.weight } // 2
  return distances.reduce(0.0, +) // 3
}
```

This method takes in the `destination` vertex and a dictionary of existing `paths`, and returns the total weight. Going over the code:

1. Construct the path to the `destination` vertex.

2. `flapMap` removes all the `nil` weights values from the `paths`.

3. `reduce` sums the weights of all the edges.

Now that you have established your helper methods, let's implement Dijkstra's algorithm.

Generating the shortest paths

After the `distance` method, add the following:

```
public func shortestPath(from start: Vertex<T>) -> [Vertex<
  var paths: [Vertex<T> : Visit<T>] = [start: .start] // 1

  // 2
  var priorityQueue = PriorityQueue<Vertex<T>>(sort: {
    self.distance(to: $0, with: paths) <
    self.distance(to: $1, with: paths)
  })
  priorityQueue.enqueue(start) // 3

  // 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 min-priority queue to store the vertices that must be visited. The `sort` closure uses the `distance` method you created to sort the vertices by their distance from the `start` vertex.

3. Enqueue the `start` vertex as the first vertex to visit.

Complete your implementation of `shortestPath` with:

```
while let vertex = priorityQueue.dequeue() { // 1
  for edge in graph.edges(from: vertex) { // 2
    guard let weight = edge.weight else { // 3
      continue
    }
    if paths[edge.destination] == nil ||
       distance(to: vertex, with: paths) + weight <
       distance(to: edge.destination, with: paths) { // 4
      paths[edge.destination] = .edge(edge)
      priorityQueue.enqueue(edge.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 dictionary of shortest paths back to the start vertex.

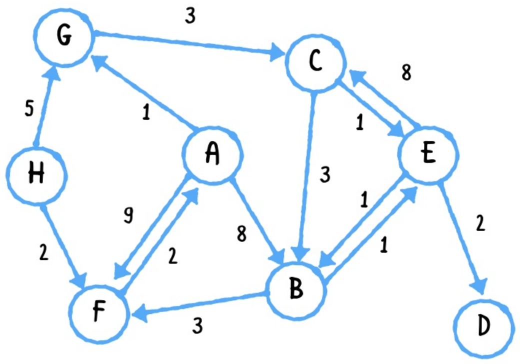
Finding a specific path

Add the following method to class `Dijkstra`:

```
public func shortestPath(to destination: Vertex<T>,
                         paths: [Vertex<T> : Visit<T>]) ->
  return route(to: destination, with: paths)
}
```

This simply takes the `destination` vertex and the dictionary of shortest and returns the path to the `destination` vertex.

Trying out your code



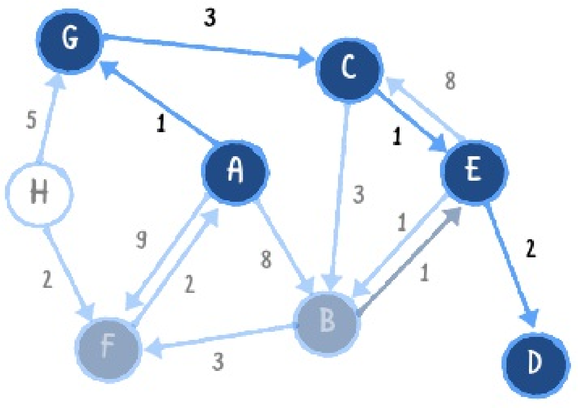
Navigate to the main playground, and you will notice the graph above has been already constructed using an adjacency list. Time to see Dijkstra's algorithm in action.

Add the following code to the playground page:

```
let dijkstra = Dijkstra(graph: graph)
let pathsFromA = dijkstra.shortestPath(from: a) // 1
let path = dijkstra.shortestPath(to: d, paths: pathsFromA) /
for edge in path { // 3
  print("\(edge.source) --(\(edge.weight ?? 0.0))--> \(edge
}
```

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:

```
A --(1.0)--> G
G --(3.0)--> C
C --(1.0)--> E
E --(2.0)--> D
```

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 is 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.

Where to go from here?

Remember that Dijkstra's algorithm is useful for finding the shortest paths between different endpoints. You greedily select the tentative shortest path, by storing vertices within a priority queue. You also created a path by using a `Visit` state to track the edges back to the start vertex.

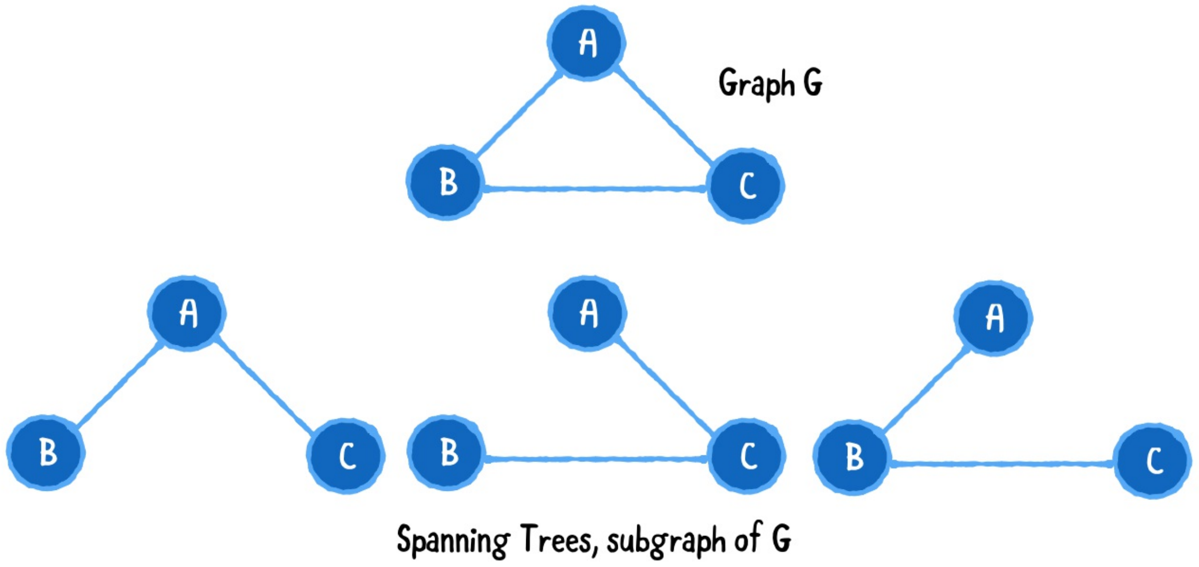
Whenever you see a network that has weighted edges and you need to find the shortest path, think of your good friend Dijkstra!

Prim’s Algorithm

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:

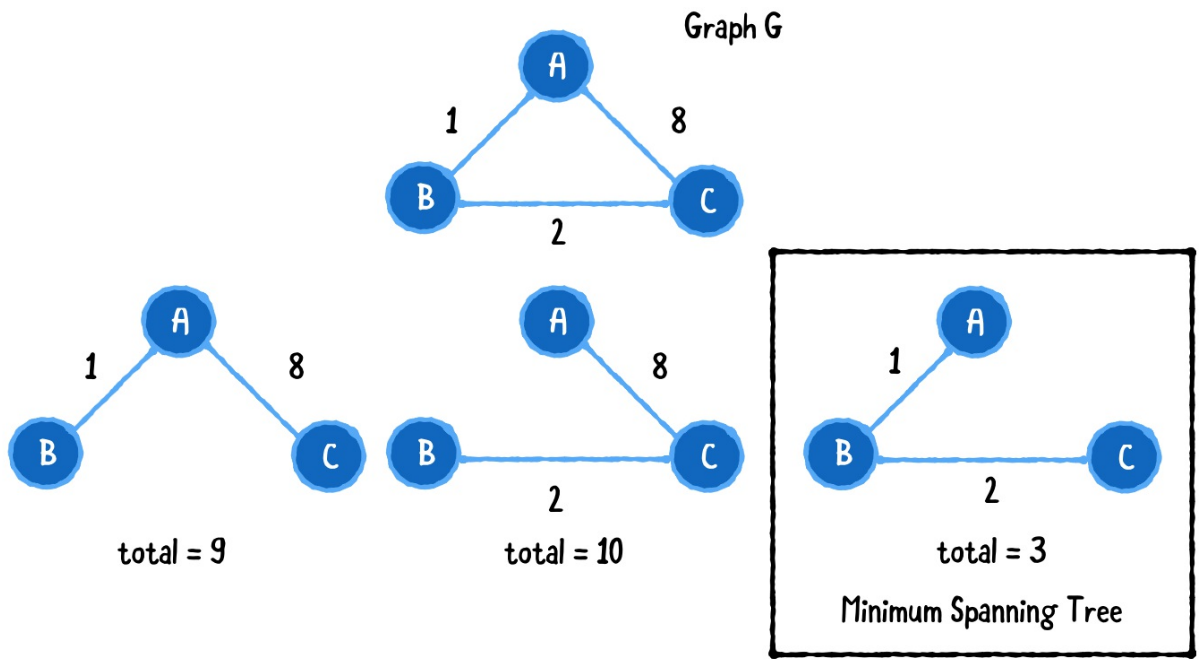


From this undirected graph that forms a triangle, you can generate three different spanning trees, where you require only two edges to connect all vertices.

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 where 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.

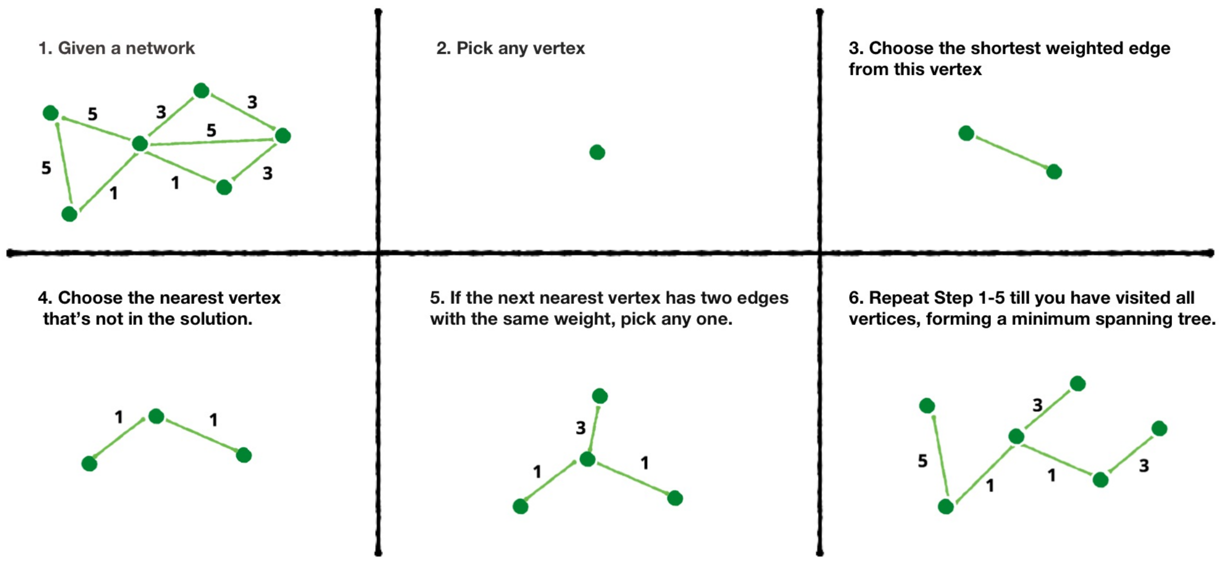
Here’s an example of a minimum spanning tree for a weighted undirected graph:



Notice that only the third subgraph forms a minimum spanning tree, since it has the minimum total cost of 3.

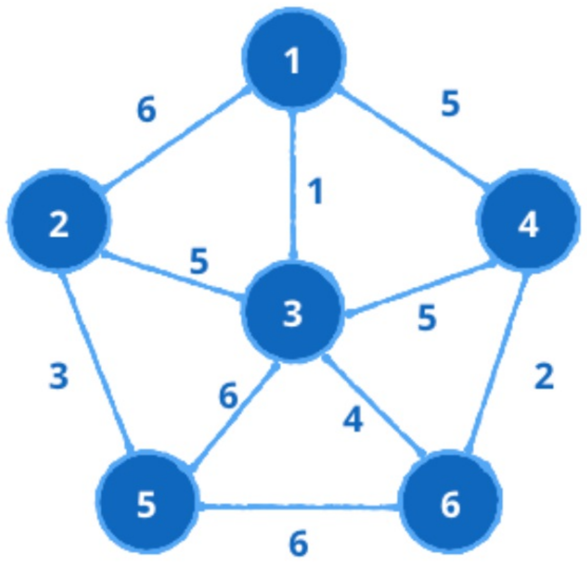
Prim’s algorithm creates a minimum spanning tree by choosing edges one at a time. It’s greedy because every time you pick an edge, you pick the smallest weighted edge that connects a pair of vertices.

There are six steps to finding a minimum spanning tree with Prim’s algorithm:

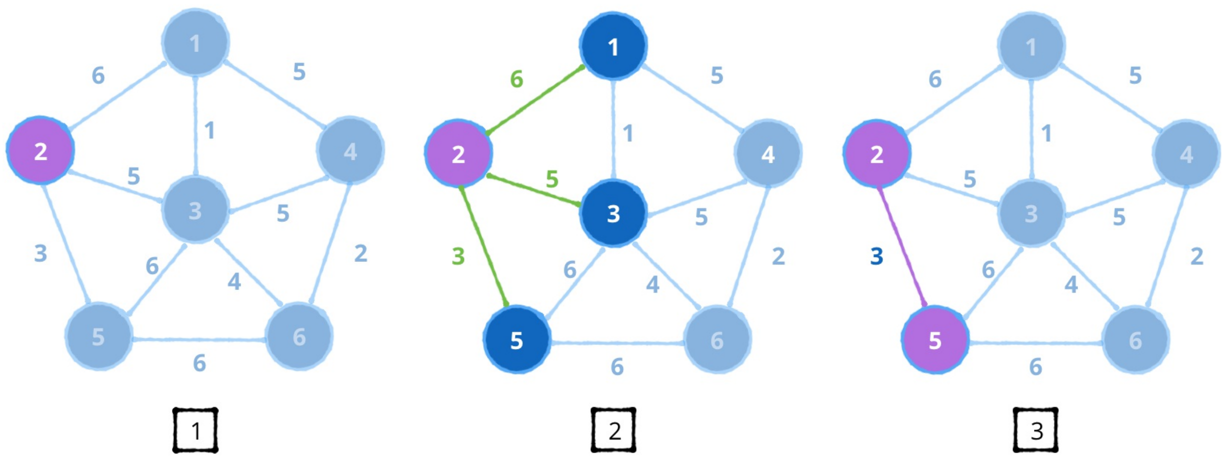

1. Given a network
2. Pick any vertex
3. Choose the shortest weighted edge from this vertex
4. Choose the nearest vertex that's not in the solution.
5. If the next nearest vertex has two edges with the same weight, pick any one.
6. Repeat Step 1-5 till you have visited all vertices, forming a minimum spanning tree.
5
3
3
5
5
1
1
3

Example

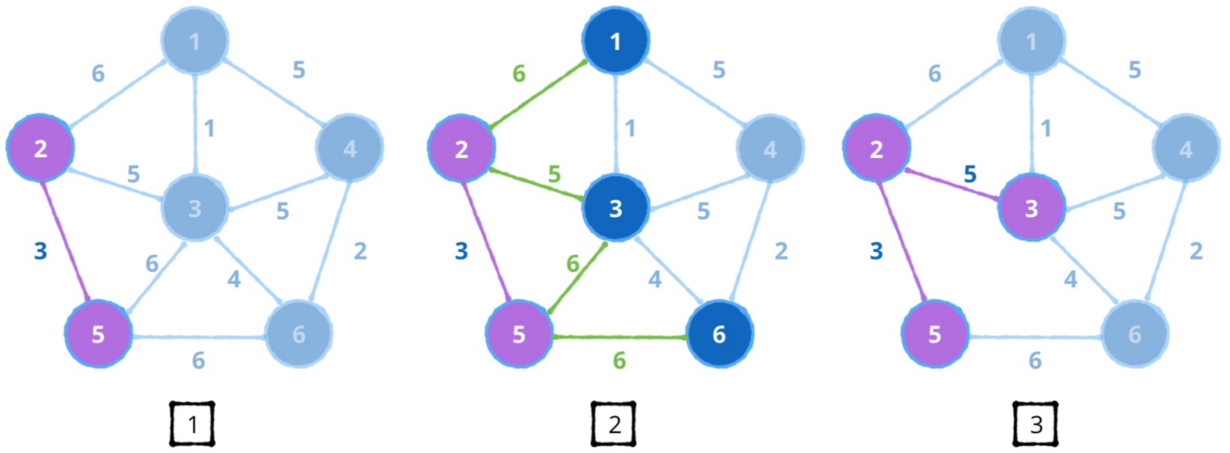
Imagine the graph below represents a network of airports. The vertices are the airports, and the edges between them represent the cost of fuel to fly an airplane from one airport to the next.



Let's start working through the example:

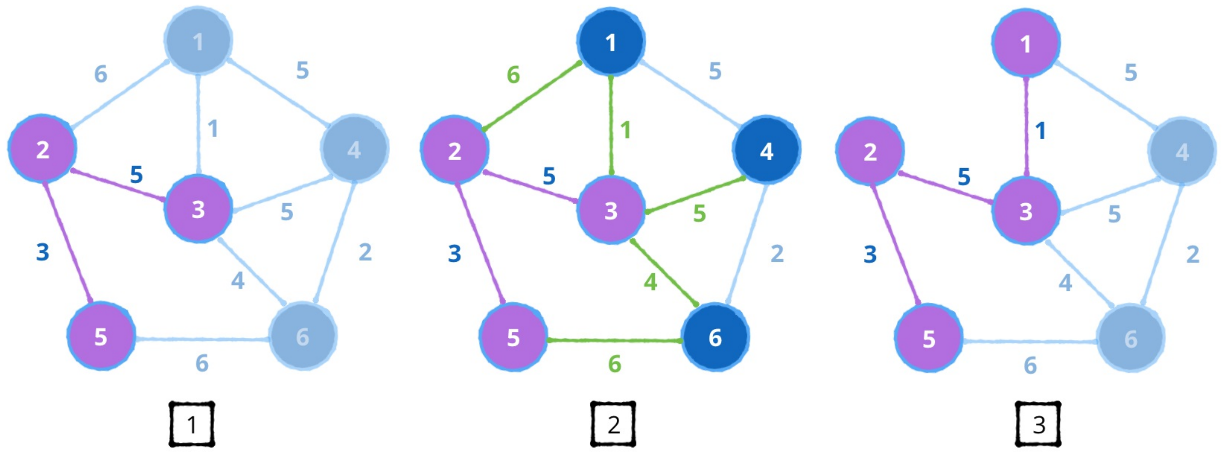


1. Choose any vertex in the graph. I'll assume you chose vertex 2.
2. This vertex has edges with weights [6, 5, 3]. A greedy algorithm chooses the smallest weighted edge.
3. Choose the edge that has a weight of 3 and is connected to vertex 5.



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.

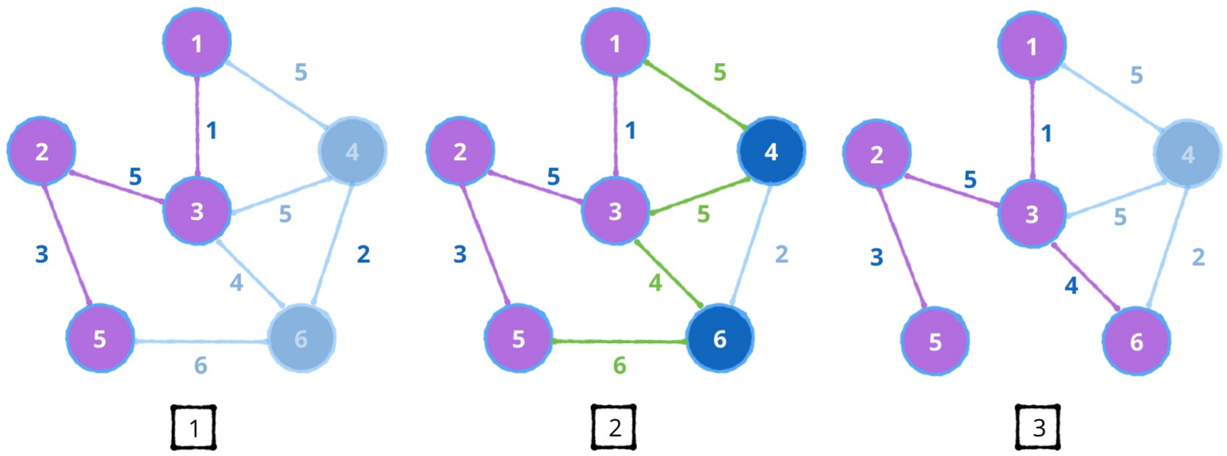
3. Notice that the edge between vertex 5 and vertex 3 can be removed since both are already part of the spanning tree.



1. The explored vertices are {2, 3, 5}.

2. The next potential edges are [6, 1, 5, 4, 6]. You choose the edge with weight 1, which is connected to vertex 1.

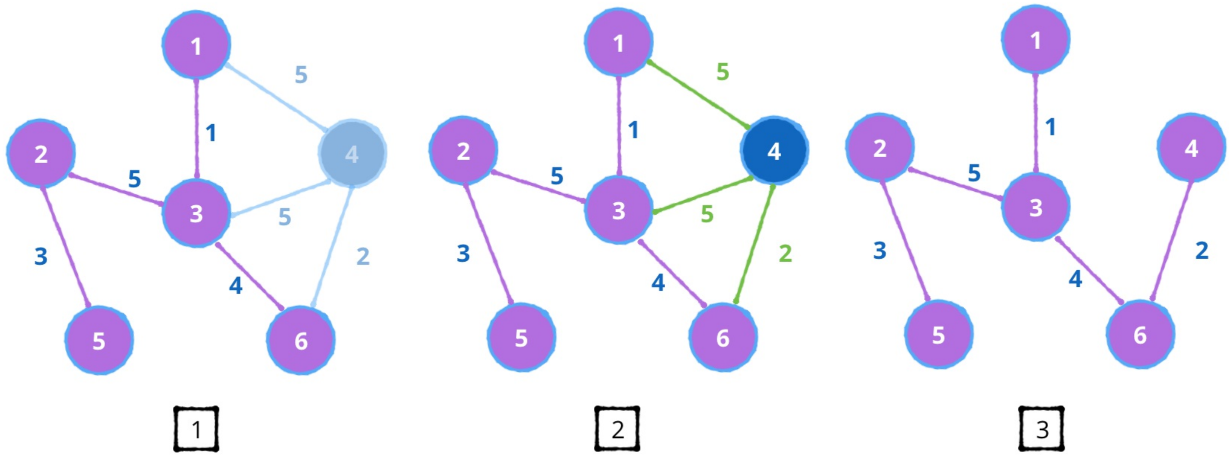
3. The edge between vertex 2 and vertex 1 can be removed.



1. The explored vertices are {2, 3, 5, 1}.

2. 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.

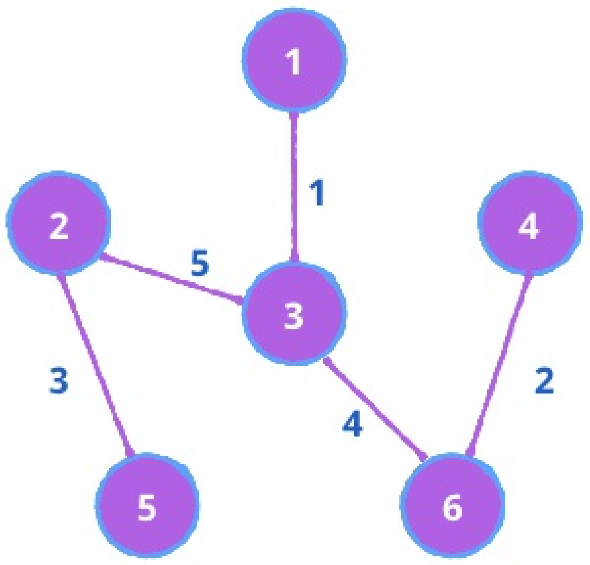


1. The explored vertices are {2, 5, 3, 1, 6}.

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 playground for this chapter. This playground 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 a class `Prim`. Open up Prim.swift and add the following:

```
public class Prim<T: Hashable> {

  public typealias Graph = AdjacencyList<T>
  public init() {}
}
```

Graph is defined as a type alias for AdjacencyList. In the future, you could replace this with an adjacency matrix if needed.

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. Add the following to class Prim:

```
internal func copyVertices(from graph: Graph, to graph2: Gra
  for vertex in graph.vertices {
    graph2.createVertex(data: vertex.data)
  }
}
```

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. Add the following to class Prim:

```
internal func addAvailableEdges(
    for vertex: Vertex<T>,
    in graph: Graph,
    check visited: Set<Vertex<T>>,
    to priorityQueue: inout PriorityQueue<Edge<T>>) {
  for edge in graph.edges(from: vertex) { // 1
    if !visited.contains(edge.destination) { // 2
      priorityQueue.enqueue(edge) // 3
    }
  }
}
```

This method takes in four parameters:

1. The current `vertex`.
2. The `graph`, where the current `vertex` is stored in.
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 class `Prim`:

```
public func produceMinimumSpanningTree(for graph: Graph)
    -> (cost: Double, mst: Graph) { // 1
  var cost = 0.0 // 2
  let mst = Graph() // 3
  var visited: Set<Vertex<T>> = [] // 4
  var priorityQueue = PriorityQueue<Edge<T>>(sort: { // 5
    $0.weight ?? 0.0 < $1.weight ?? 0.0
  })
  // 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. This is a min-priority queue to store edges.

Next, continue implementing `produceMinimumSpanningTree` with the following:

```
copyVertices(from: graph, to: mst) // 1

guard let start = graph.vertices.first else { // 2
  return (cost: cost, mst: mst)
}

visited.insert(start) // 3
addAvailableEdges(for: start, // 4
                   in: graph,
                check: visited,
                   to: &priorityQueue)

// to be continued
```

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 let smallestEdge = priorityQueue.dequeue() { // 1
  let vertex = smallestEdge.destination // 2
  guard !visited.contains(vertex) else { // 3
    continue
  }

  visited.insert(vertex) // 4
  cost += smallestEdge.weight ?? 0.0 // 5

  mst.add(.undirected, // 6
          from: smallestEdge.source,
          to: smallestEdge.destination,
          weight: smallestEdge.weight)

  addAvailableEdges(for: vertex, // 7
                    in: graph,
                    check: visited,
                    to: &priorityQueue)
}

return (cost: cost, mst: mst) // 8
```

Going over the code:

1. Continue Prim's algorithm till 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.

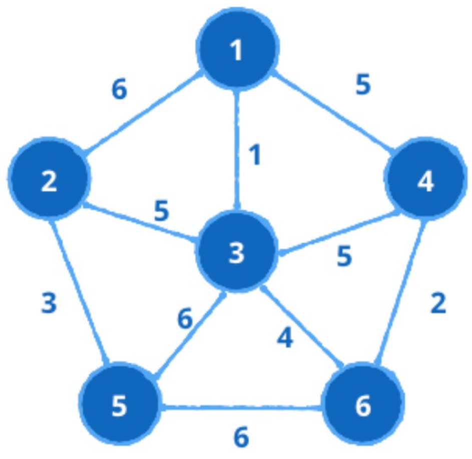
5. Add the edge's `weight` to the total `cost`.

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



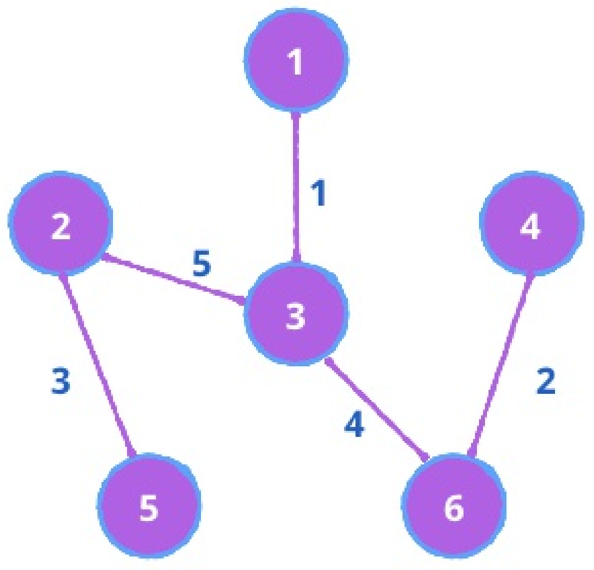
Navigate to the main playground, and you'll see the graph above has been already constructed using an adjacency list.

Time to see Prim's algorithm in action. Add the following code:

```
let (cost,mst) = Prim().produceMinimumSpanningTree(for: gra
print("cost: \(cost)")
print("mst:")
print(mst.description)
```

This constructs a graph from the example section. You'll see the following output:

```
cost: 15.0
mst:
 5 ---> [ 2 ]
 6 ---> [ 3, 4 ]
 3 ---> [ 2, 1, 6 ]
 1 ---> [ 3 ]
 2 ---> [ 5, 3 ]
 4 ---> [ 6 ]
```



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).

1. A min-priority queue to stores 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(logE)) .

Where to go from here?

This is a great example of using multiple data structures to build a useful algorithm: a priority queue, a set, and an adjacency list are all used to construct Prim’s algorithm.

Given the plethora of networks and network-like relationship models in the real world, you can apply Prim’s algorithm to many different problems where you’re always seeking the lowest possible cost for a network.

Conclusion

We hope you learned a lot about data structures and algorithms in Swift as you read this book — and had some fun in the process! Knowing when and why to apply data structures and algorithms goes beyond just acing that whiteboard interview. With the knowledge you've gained here, you can easily and efficiently solve pretty much any data manipulation or graph analysis issue put in front of you.

If you have any questions or comments as you work through this book, please stop by our forums at http://forums.raywenderlich.com and look for the particular forum category for this book.

Thank you again 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,

– Kelvin, Vincent, Ray, Steven, and Chris

The Data Structures & Algorithms in Swift team

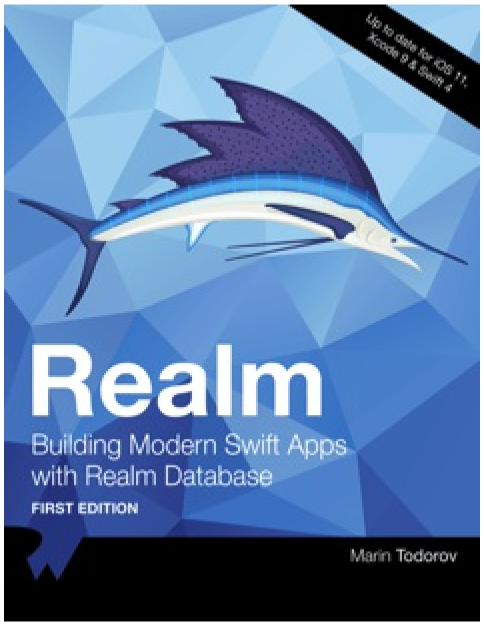
More Books You Might Enjoy

Up to date for iOS 11, Xcode 9 & Swift 4
iOS
Apprentice
SIXTH EDITION
Beginning iOS development with Swift 4
Over 10,000 copies sold!
By Matthijs Hollemans & Fahim Farook

over another. This set of basic data structures and algorithms will serve as an excellent foundation for building more complex and special-purpose constructs. As well, the high-level expressiveness of Swift makes it an ideal choice for learning these core concepts without sacrificing performance.

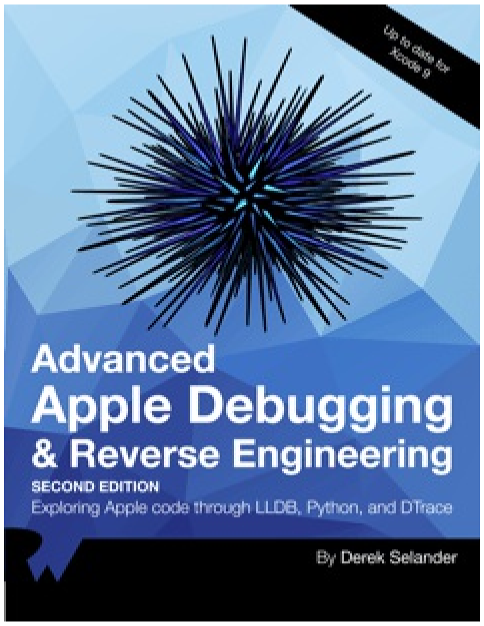
Realm: Building Modern Swift Apps with Realm Database

https://store.raywenderlich.com/products/realm-building-modern-swift-apps-with-realm-database


Up to date for iOS 11, Xcode 9 & Swift 4
Realm
Building Modern Swift Apps with Realm Database
FIRST EDITION
Marin Todorov

Realm Platform is a relatively new commercial product which allows developers to automatically synchronize data not only across Apple devices but also between any combination of Android, iPhone, Windows, or macOS apps. Realm Platform allows you to run the server software on your own infrastructure and keep your data in-house which more often suits large enterprises. Alternatively you can use Realm Cloud which runs a Platform for you and you start syncing data very quickly and only pay for what you use.

Advanced Debugging and Reverse Engineering


Up to date for Xcode 9
Advanced
Apple Debugging
& Reverse Engineering
SECOND EDITION
Exploring Apple code through LLDB, Python, and DTrace
By Derek Selander

Up to date for Android Studio 3.0 & Kotlin 1.2
Android
Apprentice
FIRST EDITION
Beginning Android development with Kotlin 1.2
By Darryl Bayliss & Tom Blankenship