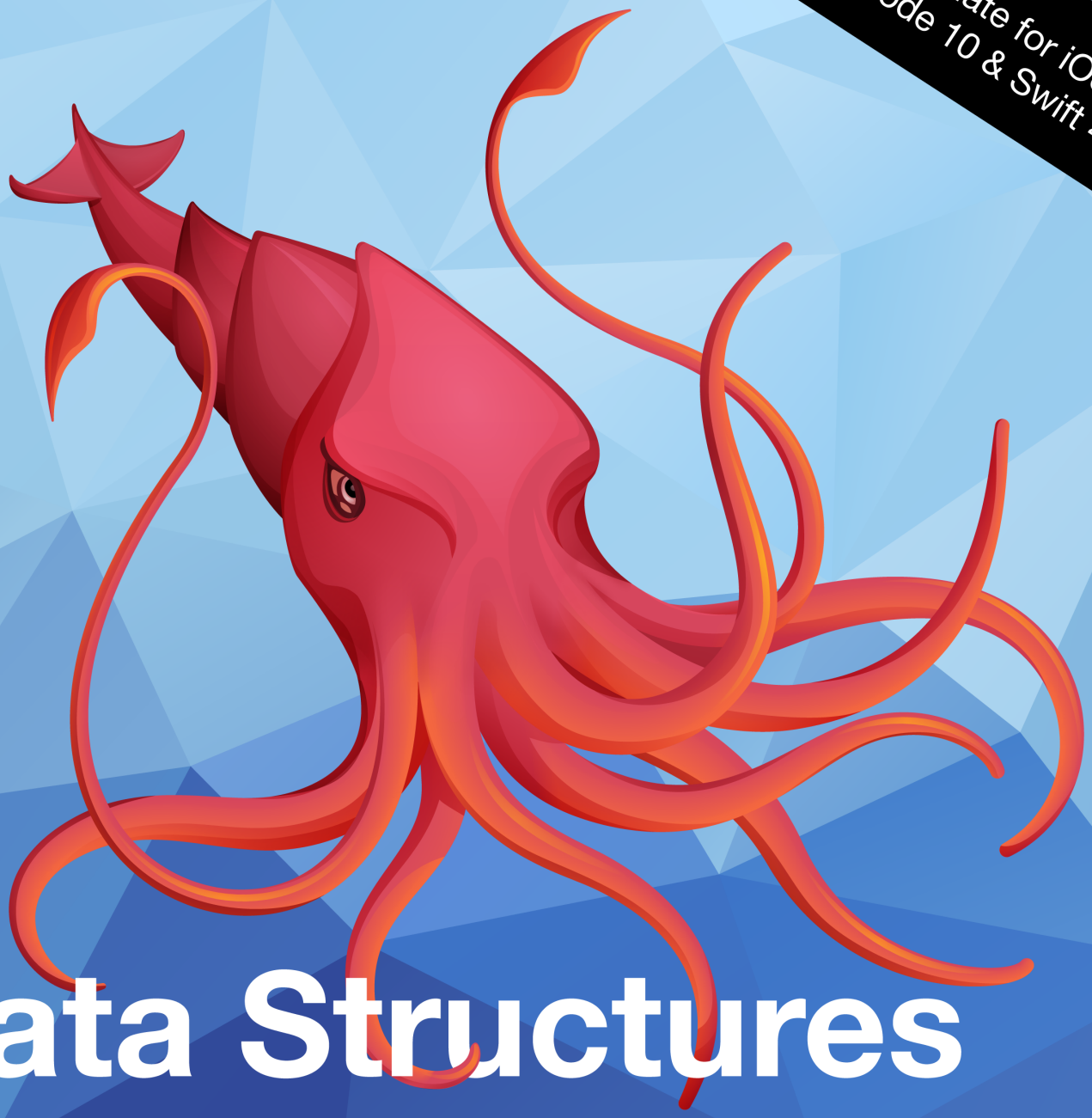
Up to date for iOS 12, Xcode 10 & Swift 4.2



Data Structures & Algorithms in Swift

SECOND EDITION

Implementing practical data structures with Swift 4.2

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

Table of Contents: Overview

Table of Contents: Extended

About This Book Sample

Data structures are a well-studied discipline, 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 makes it an ideal choice for learning these core concepts without sacrificing too much performance.

Data Structures & Algorithms in Swift is both a reference and an exercise book. It covers five main topics: the Swift Standard Library, elementary data structures, trees, sorting, and graphs.

In each topic, the book describes a number of examples of each, explaining how you can build each in turn and examines the best approach when using them in your own code. Understanding this will give you a much better idea of how to tackle various algorithmic problems and help you to build apps in the most efficient way.

This book sample contains Chapters 22 and 32. Chapter 22 explains the Heap data stucture, how to represent one in Swift and how to insert into, delete from and search for elements within the heap. Chapter 32 goes on to explain how to implement a Heap Sort.

We hope that this hands-on look inside the book will give you a good idea of what's available in the full version and shows you why you should look at how the choice of algorithms in your code can potentially improve your app's performance and scalability.

The full book is available for purchase at:

- https://store.raywenderlich.com/products/data-structures-and-algorithms-in-swift

Enjoy!

– Kelvin, Vincent, `Ray`, Steven, Chris and Manda

The *Data Structures & Algorithms in Swift* team

Data Structures & Algorithms in Swift

Kelvin Lau & Vincent Ngo

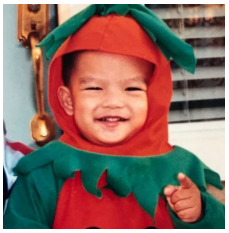
Notice of Rights

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 an 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.

Manda Frederick is an editor of this book. Manda has been involved in publishing for over ten years through various creative, educational, medical and technical print and digital publications. In her free time, you can find her at the climbing gym, backpacking in the backcountry, hanging with her dog, working on poems, playing guitar and exploring breweries. Twitter: @mandarazeware.

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.

About the Contributors

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

Matthijs Hollemans, the original creator of the Swift Algorithm Club. Matthijs contributed many of the implementations and corresponding explanations for the various data structures and algorithms in the Swift Algorithm Club that were used in this book, in particular: Graph, Heap, AVL Tree, BST, Breadth First Search, Depth First Search, Linked List, Stack & Queue, Tree, Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, and Heap Sort. Matthijs spends much of his time now in machine learning. Learn more at http://machinethink.net.

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

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

About the Artist

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

Book License

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

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

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

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

What You Need

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

- **A Mac running the latest macOS**. This is so you can install the latest version of the required development tool: Xcode.
- **Xcode 10 or later.** Xcode is the main development tool for writing code in Swift. You need Xcode 10 at a minimum, since that version includes Swift 4.2. You can download the latest version of Xcode for free from the Mac App Store, here: apple.co/1FLn51R.

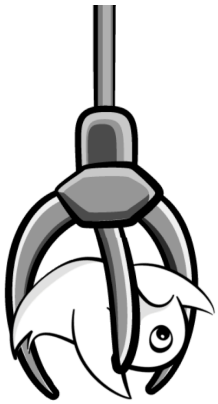
If you haven't installed the latest version of Xcode, be sure to do that before continuing with the book. The code covered in this book depends on Swift 4.2 and Xcode 10 — you may get lost if you try to work with an older version.

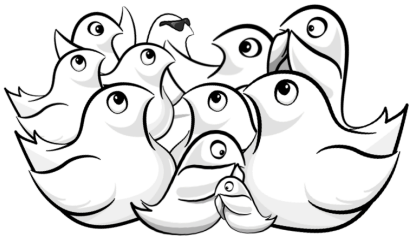
Chapter 22: The Heap Data Structure

By Vincent Ngo

Have you ever been to the arcade and played those crane machines that contain stuffed animals or cool prizes? These machines make it *very* hard to win. But the fact that you set your eyes on the item you want is the very essence of the heap data structure!

Ever seen the movie *Toy Story* with the claw and the little green squeaky aliens? Just imagine that the claw machine operates on your heap data structure and will always pick the element with the highest priority. The Claw...





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

What is a heap?

A heap is a **complete** binary tree, also known as a binary heap, that can be constructed using an array.

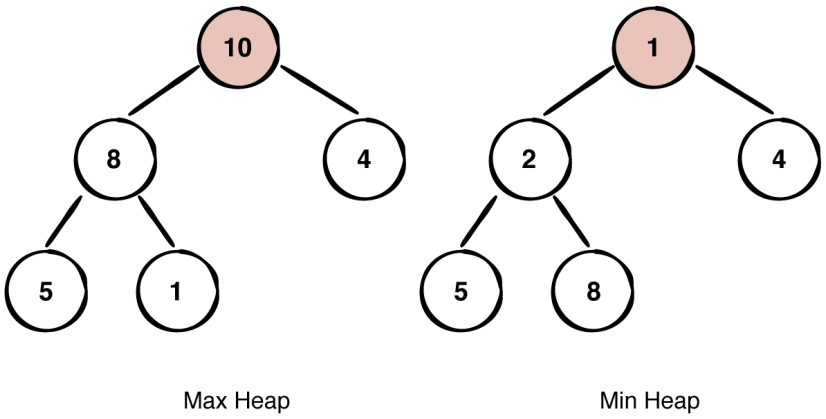
> **Note**: Don't confuse these heaps with memory heaps. The term heap is sometimes confusingly used in computer science to refer to a pool of memory. Memory heaps are a different concept and not what you are studying here.

Heaps come in two flavors:

1. **Max** heap, in which elements with a **higher** value have a higher priority.
2. **Min** heap, in which elements with a **lower** value have a higher priority.

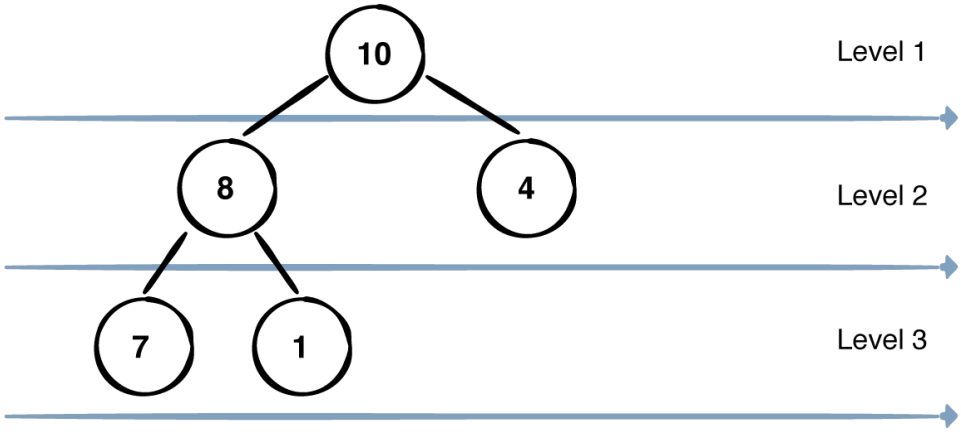
The heap property

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



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 wherein you can't go to the next level until you have completed the current one.

Heap applications

Some useful applications of a heap include:

- Calculating the minimum or maximum element of a collection.
- Heap sort.
- Constructing a priority queue.
- Constructing graph algorithms, like Prim's or Dijkstra's, with a priority queue.

> **Note**: You will learn about the heap sort in **Chapter 32** - provided with this sample. Priority queues, Dijkstra's algorithm and Prim's algorithm are covered in **Chapters 24, 42 and 44**, respectively of the full version of this book.

Common heap operations

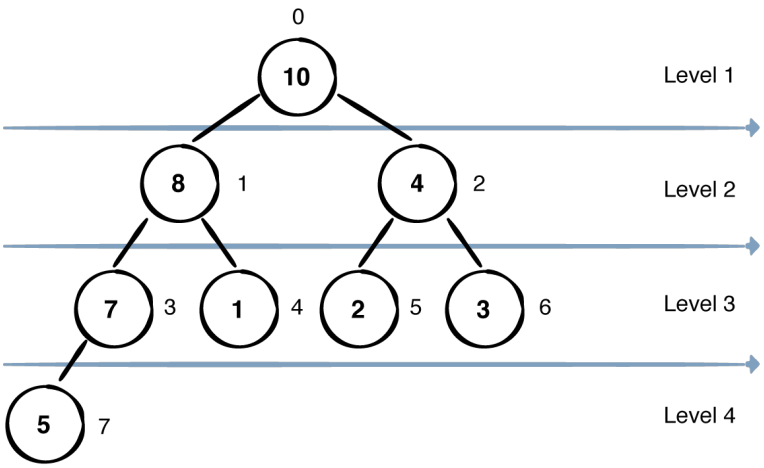
Open the empty starter playground for this chapter. Start by defining the following basic `Heap` type:

```
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.

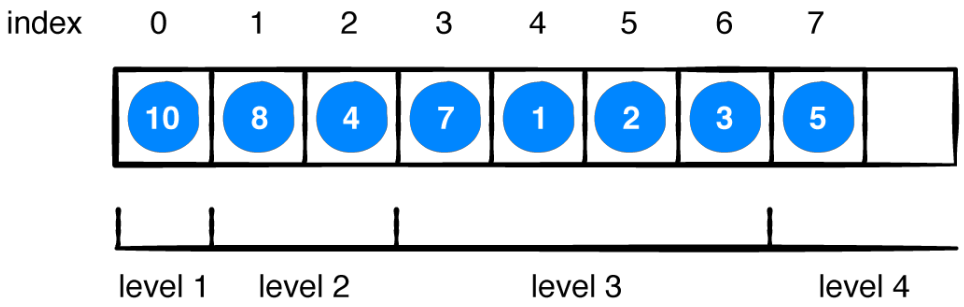
How do you represent a heap?

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:

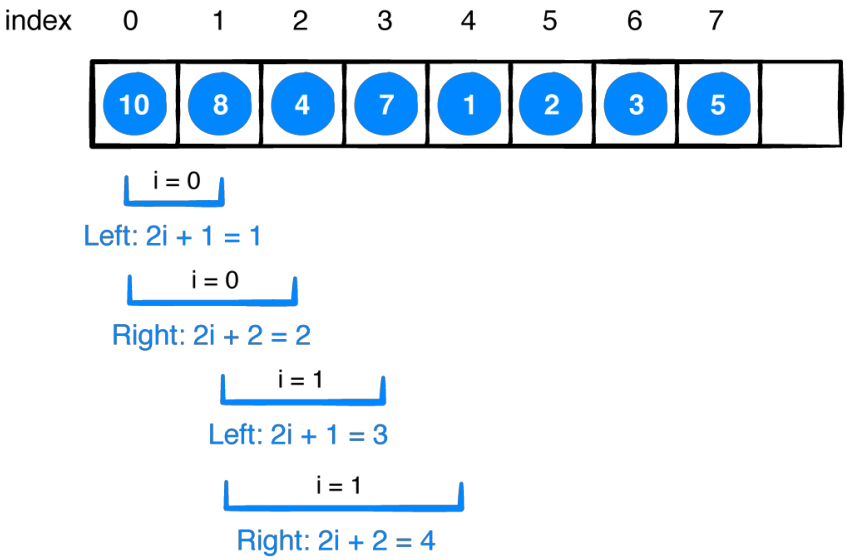


As you go up a level, you'll have twice as many nodes than in the level before.

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)`.

> **Complexity**: Traversing down an actual binary tree to get the left and right child of a node is a *O*(*log n*) operation. In a random-access data structure, such as an array, that same operation is just *O*(1).

> **Note**: Complexity is covered in **Chapter 3** of the full book.

Next, use your new knowledge to add some properties and convenience methods to `Heap`:

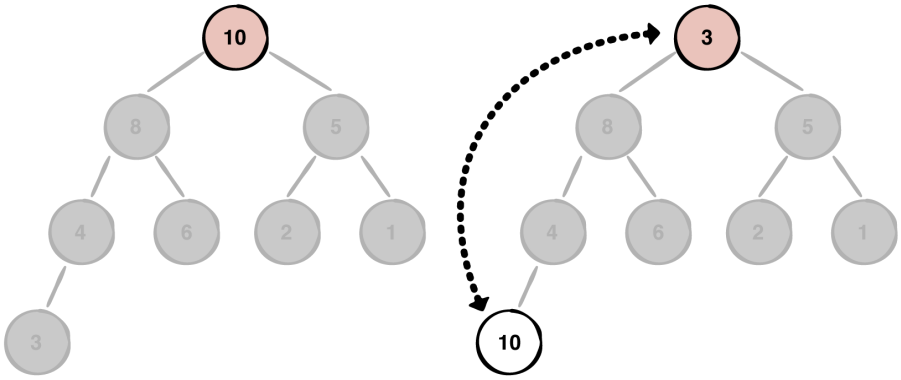
```
var isEmpty: Bool {
  return elements.isEmpty
}

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.

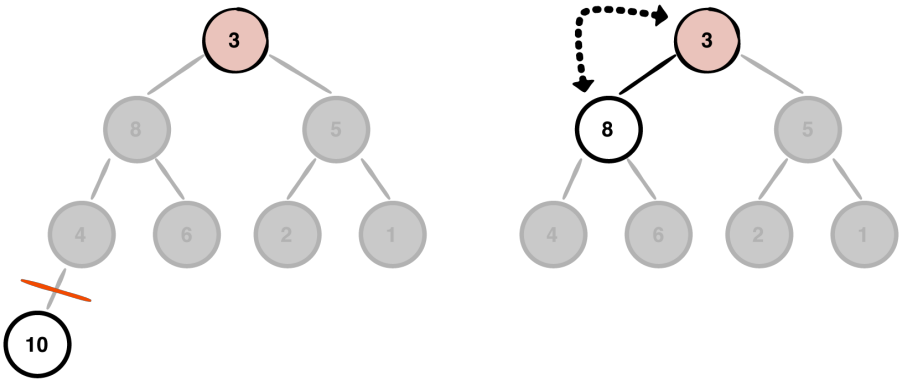
Removing from a heap

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

Take the following max heap:



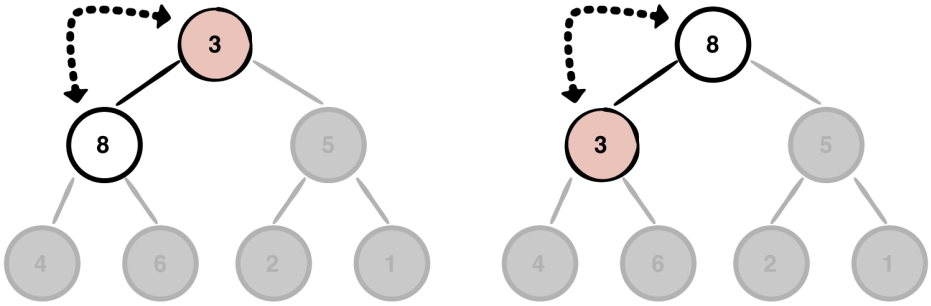
A remove operation will remove the maximum value at the root node. To do so, you must first swap the **root** node with the **last** element in the heap.



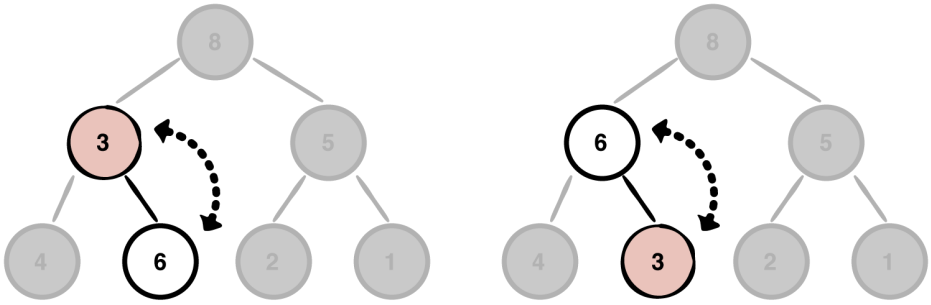
Once you've swapped the two elements, you can remove the last element and store its value so you can later return it.

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

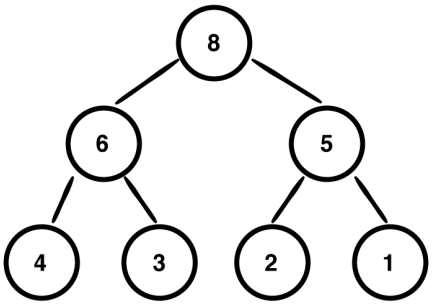
Remember: The rule for a max heap is that the value of every parent node must be larger than, or equal to, the values of its children. Since the heap no longer follows this rule, you must perform a **sift down**.



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[candidate]) {
      candidate = left // 5
    }
    if right < count && sort(elements[right], elements[candidate]) {
      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.

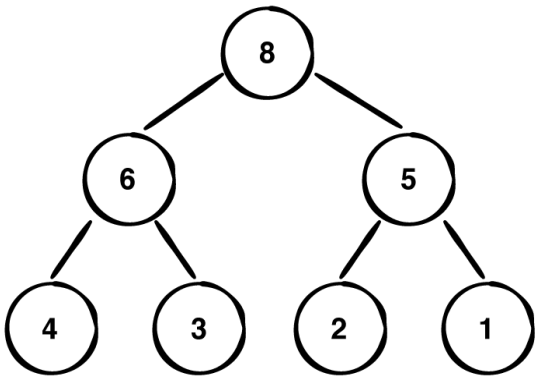
2. Continue sifting until you `return`.
3. Get the parent's left and right child index.
4. The `candidate` variable is used to keep track of which index to swap with the parent.
5. If there is a left child, and it has a higher priority than its parent, make it the candidate.
6. If there is a right child,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

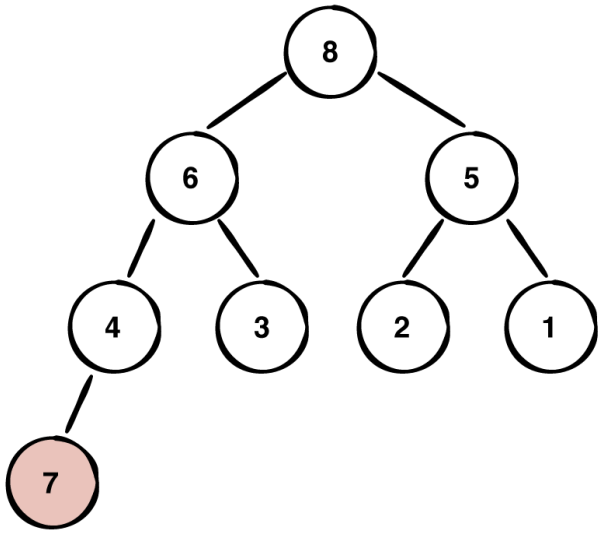
Now that you know how to remove from the top of the heap, how do you *add* to a heap?

Inserting into a heap

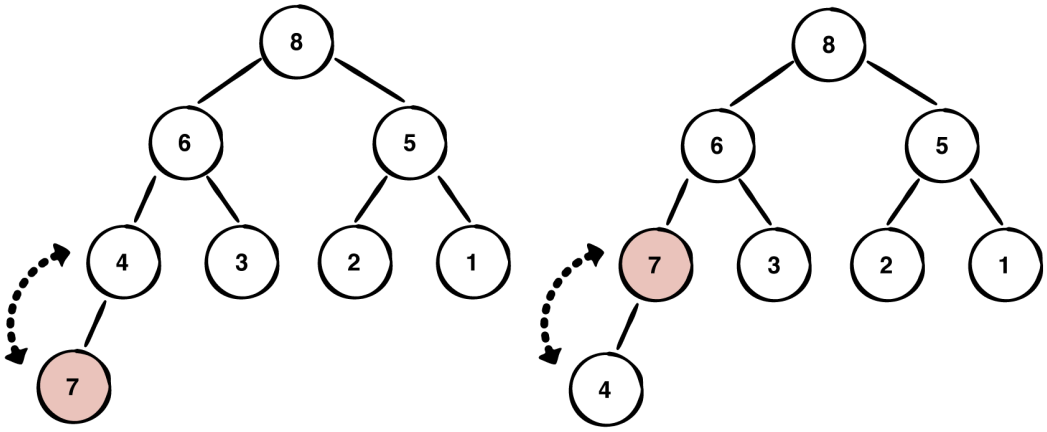
Let's say you insert a value of **7** to the heap below:

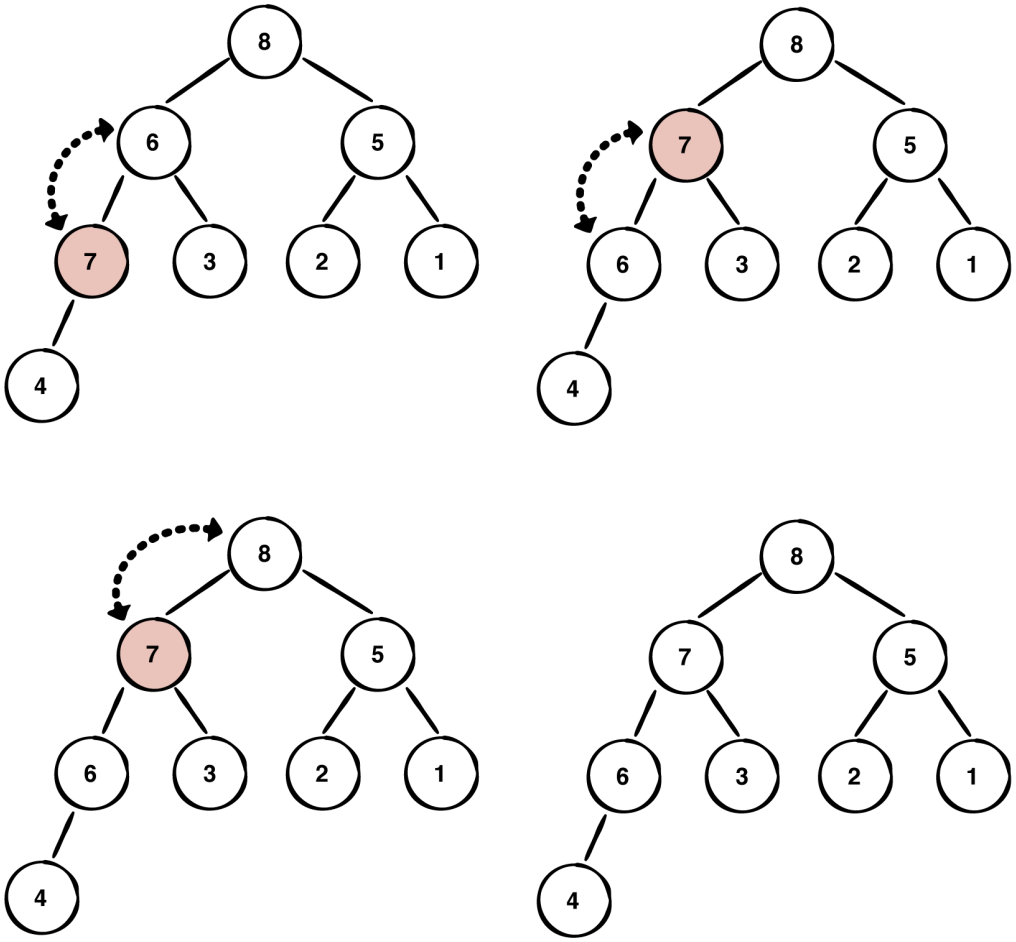


First, you add the value to the end of the heap:



Now, you must check the max heap's property. Instead of sifting down, you must now **sift up** since the node that 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`:

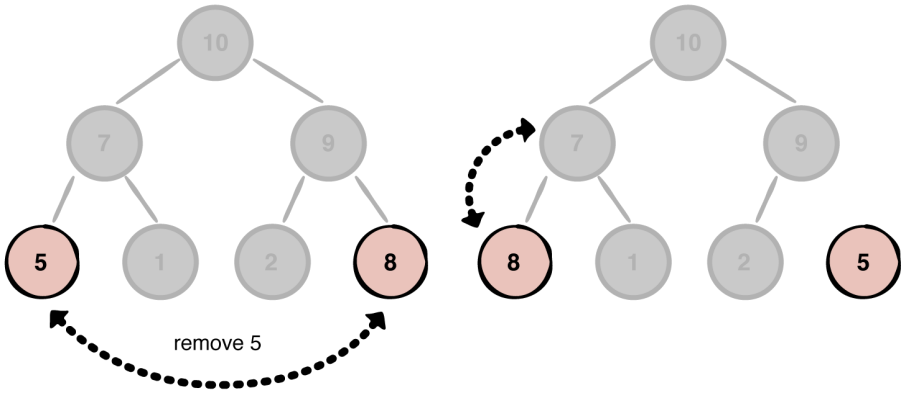
```
mutating func remove(at index: Int) -> Element? {
  guard index < elements.count else {
    return nil // 1
  }
  if index == elements.count - 1 {
    return elements.removeLast() // 2
  } else {
    elements.swapAt(index, elements.count - 1) // 3
    defer {
      siftDown(from: index) // 5
   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

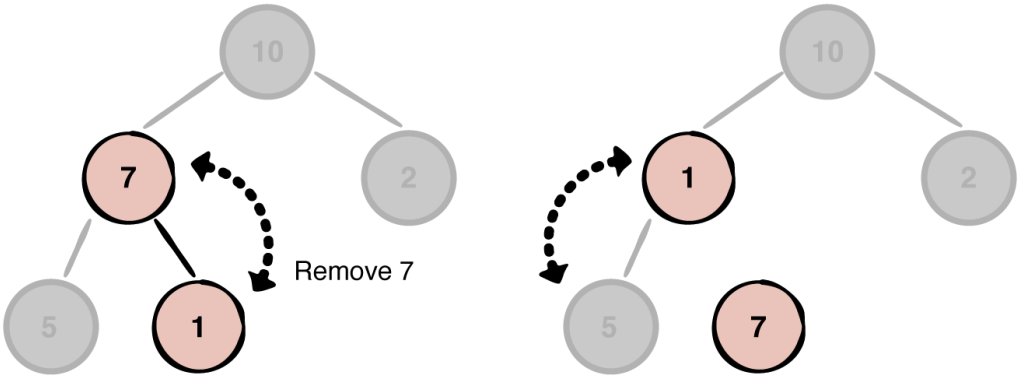
3. If you're not removing the last element, first swap the element with the last element.
4. Then, return and remove the last element.
5. Finally, perform a sift down and a sift up to adjust the heap.

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



Shifting up case

Assume that 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 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 that 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(ofParentAt:
i)) {
    return j // 4
  }
  if let j = index(of: element, startingAt: rightChildIndex(ofParentAt:
i)) {
    return j // 5
  }
  return nil // 6
}
```

Let's go over this implementation:

1. If the index is greater than or equal to the number of elements in the array, the search failed. Return `nil`.
2. Check to see if the element that 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`.

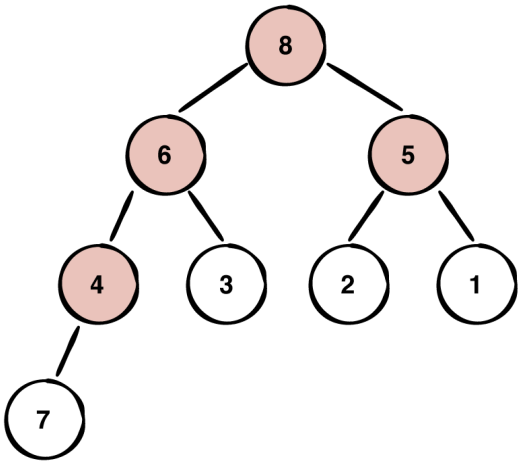
> **Note**: Although searching takes $O(n)$ time, you have made an effort to optimize searching by taking advantage of the heap's property and checking the priority of the element when searching.

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: 0, by: -1) {
      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

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
```

Key points

- Here is a summary of the algorithmic complexity of the heap operations that 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)

Heap operation time complexity

- The heap data structure is good for maintaining the highest- or lowest-priority element.
- Every time you insert or remove items from the heap, you must check to see if it satisfies the rules of the priority.

Chapter 32: Heap Sort

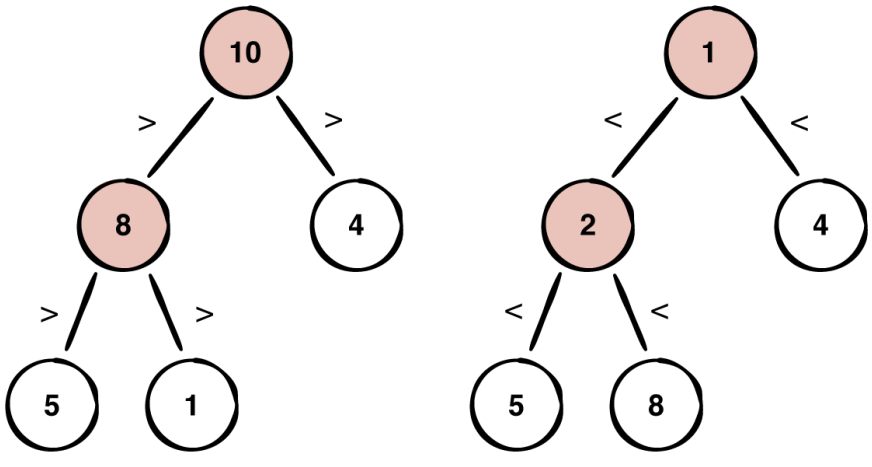
Vincent Ngo

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 22, "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:

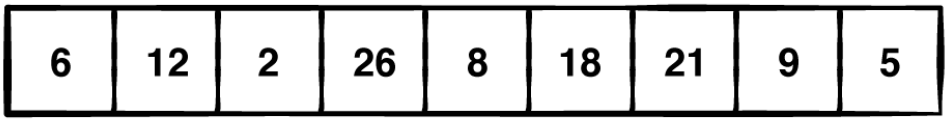


Getting started

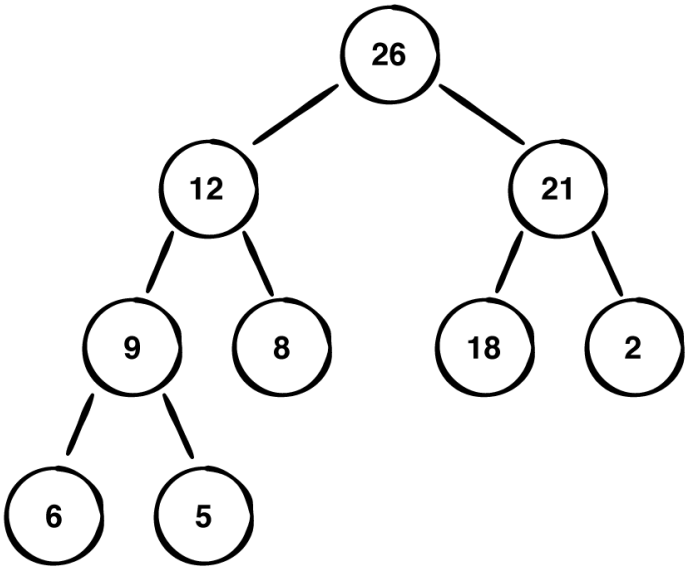
Open up the starter playground. This playground already contains an implementation of a max heap. Your goal is to extend `Heap` so it can also sort. Before you get started, let's look at a visual example of how heap sort works.

Example

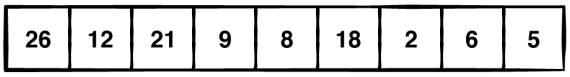
For any given unsorted array, to sort from lowest to highest, heap sort must first convert this array into a max heap:



This conversion is done by sifting down all the parent nodes so that they end up in the right spot. The resulting max heap is:



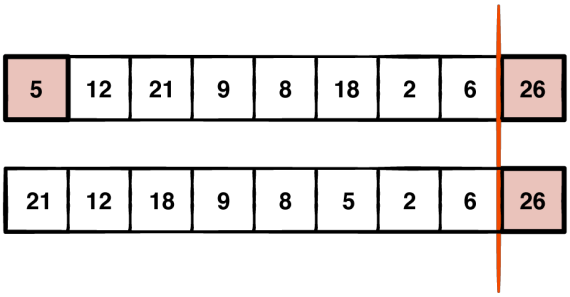
This corresponds with the following array:



Because the time complexity of a single sift-down operation is $O(\log n)$, the total time complexity of building a heap is $O(n \log n)$.

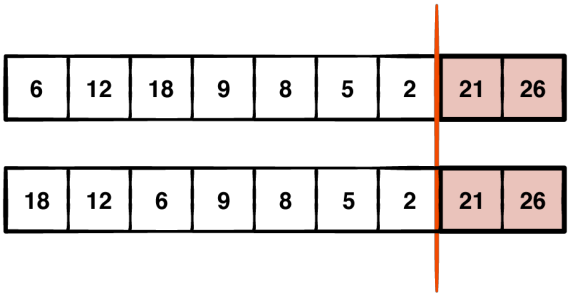
Let's look at how to sort this array in ascending order.

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.



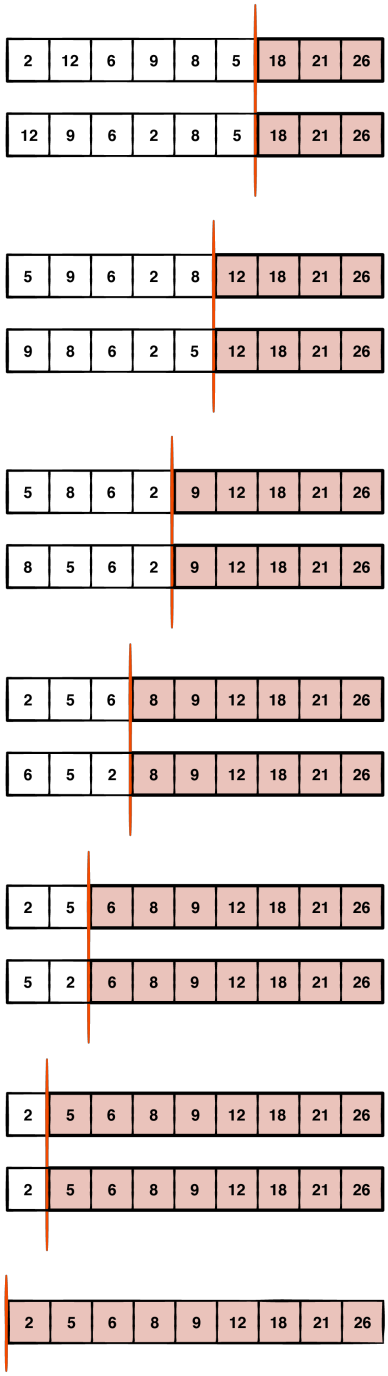
Note that you exclude the last element of the heap as you no longer consider it part of the heap, but of the sorted array.

As a result of sifting down **5**, the second largest element **21** becomes the new root. You can now repeat the previous steps, swapping **21** with the last element **6**, shrinking the heap and sifting down **6**.



Starting to see a pattern? Heap sort is very straightforward. As you swap the first and last elements, the larger elements make their way to the back of the array in the correct

order. You simply repeat the swapping and sifting steps until you reach a heap of size 1. The array is then fully sorted.



Note: This sorting process is very similar to selection sort from **Chapter 26**, which is available in the full version of this book.

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, 9, 5])
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.

Key Points

- Heap sort leverages the max-heap data structure to sort elements in an array.

Where to Go From Here?

We hope you enjoyed this sample of *Data Structures & Algorithms in Swift*!

If you did, be sure to check out the full book, which contains the following chapters:

1. **Preface**: 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.

2. **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.

3. **Linked List**: A linked list is a collection of values arranged in a linear unidirectional sequence. A linked list has several theoretical advantages over contiguous storage options such as the Swift Array, including constant time insertion and removal from the front of the list, and other reliable performance characteristics.

4. **Stack Data Structure**: The stack data structure is identical in concept to a physical stack of objects. When you add an item to a stack, you place it on top of the stack. When you remove an item from a stack, you always remove the topmost item. Stacks are useful, and also exceedingly simple. The main goal of building a stack is to enforce how you access your data.

5. **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 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.

6. **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, and facilitating fast lookup operations. There are many types of trees, and they come in various shapes and sizes.

7. **Binary Trees**: In the previous chapter, you looked at a basic tree where each node can have many children. A binary tree is a tree where each node has at most two children, often referred to as the left and right children. Binary trees serve as the basis for many tree structures and algorithms. In this chapter, you'll build a binary tree and learn about the three most important tree traversal algorithms.

8. **Binary Search Trees**: A binary search tree facilitates fast lookup, addition, and removal operations. Each operation has an average time complexity of O(log n), which is considerably faster than linear data structures such as arrays and linked lists.

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

10. **Tries**: The trie (pronounced as "try") is a tree that specializes in storing data that can be represented as a collection, such as English words. The benefits of a trie are best illustrated by looking at it in the context of prefix matching, which is what you'll do in this chapter.

11. **Binary Search**: Binary search is one of the most efficient searching algorithms with a time complexity of O(log n). This is comparable with searching for an element inside a balanced binary search tree. To perform a binary search, the collection must be able to perform index manipulation in constant time, and must be sorted.

12. **The Heap Data Structure**: A heap is a complete binary tree, also known as a binary heap, that can be constructed using an array. Heaps come in two flavors: Max heaps and Min heaps. Have you seen the movie Toy Story, with the claw machine and the squeaky little green aliens? Imagine that the claw machine is operating on your heap structure, and will always pick the minimum or maximum value, depending on the flavor of heap.

13. **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. A priority queue is especially useful when you need to identify the maximum or minimum value given a list of elements.

14. **$O(n^2)$ 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. In this chapter, you'll be looking at the bubble sort, selection sort, and insertion sort algorithms.

15. **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.

16. **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.

17. **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.

18. **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. In this chapter, you will implement Quicksort and look at various partitioning strategies to get the most out of this sorting algorithm.

19. **Graphs**: What do social networks have in common with booking cheap flights around the world? You can represent both of these real-world models as graphs! A graph is a data structure that captures relationships between objects. It is made up of vertices connected by edges. In a weighted graph, every edge has a weight associated with it that represents the cost of using this edge. This lets you choose the cheapest or shortest path between two vertices.

20. **Breadth-First Search**: In the previous chapter, you explored how graphs can be used to capture relationships between objects. Several algorithms exist to traverse or search through a graph's vertices. One such algorithm is the breadth-first search algorithm, which can be used to solve a wide variety of problems, including generating a minimum spanning tree, finding potential paths between vertices, and finding the shortest path between two vertices.

21. **Depth-First Search**: In the previous chapter, you looked at breadth-first search where you had to explore every neighbor of a vertex before going to the next level. In this chapter, you will look at depth-first search, which has applications for topological sorting, detecting cycles, path finding in maze puzzles, and finding connected components in a sparse graph.

22. **Dijkstra's Algorithm**: Have you ever used the Google or Apple Maps app to find the shortest or fastest from one place to another? Dijkstra's algorithm is particularly useful in GPS networks to help find the shortest path between two places. Dijkstra's algorithm is a greedy algorithm, which constructs a solution step-by-step, and picks the most optimal path at every step.

23. **Prim's Algorithm**: In previous chapters, you've looked at depth-first and breadth-first search algorithms. These algorithms form spanning trees. In this chapter, you will look at Prim's algorithm, a greedy algorithm used to construct a minimum spanning tree. A minimum spanning tree is a spanning tree with weighted edges where the total weight of all edges is minimized. You'll learn how to implement a greedy algorithm to construct a solution step-by-step, and pick the most optimal path at every step.

Challenges: In addition to all the above theory chapters, you'll also get challenge chapters to test your knowledge of the data structures and algorithms contained in the book, along with a full set of solutions with code.

You can find the book on the raywenderlich.com store here:

- https://store.raywenderlich.com/products/data-structures-and-algorithms-in-swift

We hope you enjoy the book! :]

– Kelvin, Vincent, `Ray`, Steven, Chris and Manda

The *Data Structures & Algorithms in Swift* team

Learn Data Structures and Algorithms in Swift!

Understanding how data structures and algorithms work in code is crucial for creating efficient and scalable apps. Swift's Standard Library has a small set of general purpose collection types, yet they definitely don't cover every case!

In this book, you'll learn how to implement the most popular and useful data structures, and when and why you should use one particular data structure or algorithm 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.

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.

Topics Covered in Data Structures & Algorithms in Swift

- **Basic structures:** Start with the fundamental structures of linked lists, queues and stacks, and see how to implement them in a highly Swift-like way.
- **Trees:** Learn how to work with various types of trees, including general purpose trees, binary trees, AVL trees, binary search trees, and tries.
- **Sorting:** Go beyond bubble and insertion sort with better-performing algorithms, including mergesort, radix sort, heap sort, and quicksort.
- **Graphs:** Learn how to construct directed, non-directed and weighted graphs to represent many real-world models.
- **Traversals:** Traverse graphs and trees efficiently with breadth-first, depth-first, Dijkstra's and Prim's algorithms to solve problems such as finding the shortest path or lowest cost in a network.
- **And much, much more!**

By the end of this book, you'll have hands-on experience solving common issues with data structures and algorithms — and you'll be well on your way to developing your own efficient and useful implementations!

About the iOS Tutorial Team

The Tutorial Team is a group of app developers and authors who write tutorials on the popular website **raywenderlich.com.** We take pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun.

If you've enjoyed the tutorials we've written in the past, you're in for a treat. The tutorials we've written for this book are some of our best yet — and this book contains detailed technical knowledge you simply won't be able to find anywhere else.