THIRD EDITION

WAYNE BISHOP

SWIFT ALGORITHMS & DATA STRUCTURES

Modern code, illustrations & computer science

SWIFT ALGORITHMS & DATA STRUCTURES

BY WAYNE BISHOP

FOR MY FAMILY

Printed in the United States of America
Third Edition, 2018
Version 4.1
www.waynewbishop.com

TABLE OF CONTENTS

RESOURCES 9

INTRODUCTION 11

BIG O NOTATION 13

BASIC SORTING 17

ADVANCED SORTIING 21

GENERICS 25

LINKED LISTS 29

TRIES 35

STACKS & QUEUES 41

BINARY SEARCH TREES 45

TREE BALANCING 49

GRAPHS 57

SHORTEST PATHS 63

HEAPS 69

TRAVERSALS 75

HASH TABLES 79

CLOSURES 85

CONTROL STRUCTURES 89

RECURSION 93

DYNAMIC PROGRAMMING 99

UNIT TESTING 103

APPENDIX 107

THE AUTHOR 121

RESOURCES

HISTORY

The relative stability of **Swift 4.0** has allowed an opportunity refine many areas of the code base. As a result, many algorithms noted in the previous edition have been revised to meet this new standard. An overall effort has also been made to improve code interoperability and support for optionals, generics, protocols and other Swift API Design Guidelines. Notable additions include:

- New content on code memoization and dynamic programming
- New non-recursive binary search tree algorithm (BST)
- New protocol-oriented hash table algorithm
- New contains() method for BST's
- New generic min/max heap algorithm
- New generic Stack algorithm
- New subscript syntax for trie and linked list algorithms
- Code updates to support `Array.swapAt()`
- Limited use of implicit unwrapped optionals (IUO's)

BOOK FORMAT

This digital book has also been enhanced with a new in-line **Glossary** items. Readers can find topics in the sidebars on additional subjects related to Swift and software development. For those preparing for their next technical interview, the **Appendix** includes new essays that provide interviewing tips and suggestions.

OPEN SOURCE

The Swift language is maintained by Apple but is now available as open source. As such, code and illustrations presented in the book can be applied to Xcode related projects and beyond. To learn more visit **swift.org**.

IOS INTERVIEW PROGRAM

Need additional help learning these topics? With an average technical interview lasting 3-6 hours, passing requires a combination of coding expertise, knowledge of software development processes, and keeping calm under pressure. To provide engineers with an edge for landing their next position, I also provide **personalized** 1-1 iOS interview coaching. Learn more at **shop.waynewbishop.com**.

INTRODUCTION

This series provides an introduction to commonly used data structures and algorithms written in a new development language called Swift. While details of many algorithms exists on Wikipedia, these implementations are often written as pseudocode, or are expressed in C or C++. With Swift available as open source, its general syntax should be familiar enough for most programmers to understand.

AUDIENCE

As a reader, you should already be familiar with the basics of programming. Beyond common algorithms, this guide also provides an alternative source for learning the basics of Swift. This includes implementations of many Swift-specific features such as optionals and generics. Beyond Swift, you should be familiar with factory design patterns along with sets, arrays and dictionaries.

WHY ALGORITHMS?

When creating modern apps, much of the theory inherent to algorithms is often overlooked. For solutions that consume relatively small amounts of data, decisions about specific techniques or design patterns may not be as important as just getting things to work. However, as your audience grows, so will your data. Much of what makes big tech companies successful is their ability to interpret vast amounts of data. Making sense of data allows users to connect, share, complete transactions and make decisions.

In the startup community, investors often fund companies that use data to create unique insights - something that can't be duplicated by just connecting an app to a simple database. These implementations often boil down to creating unique (often patentable) algorithms like Google PageRank or The Facebook Graph. Other categories include social networking (e.g. LinkedIn), predictive analysis (e.g. Uber.com) or machine learning (e.g. Amazon.com).

BIG O NOTATION

Building a service that finds information quickly could mean the difference between project success and failure. For example, much of Google's success comes from algorithms that allow people to search vast amounts of data with great efficiency.

There are numerous ways to search and sort data. As a result, computer experts have devised a way for us to compare the efficiency of software algorithms regardless of computing device, memory or hard disk space. Asymptotic analysis is the process of describing the efficiency of algorithms as their input size (n) grows. In computer science, asymptotics are usually expressed in a common format known as ***Big O Notation***.

MAKING COMPARISONS

To understand Big O Notation, one only needs to start comparing algorithms. In this example, we compare two techniques for searching values in a sorted array:

```
//array of sorted integers
var numberList : Array<Int> = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

LINEAR TIME

Our first approach employs a common *"brute force"* technique that involves looping through the entire array until we find a match. In Swift, this can be achieved with the following;

```
extension Array where Element: Comparable {

     func linearSearch(forElement key: Element) -> Bool {

        //check all possible values
        for number in self {
            if number == key {
                return true
            }
        }

        return false
    }
}

//execute search
let isFound: Bool = numberList.linearSearch(forElement: 8)
```

BRUTE FORCE

A situation where a programmer writes a solution without consideration of its algorithmic efficiency. In some cases, brute force solutions lack sophistication and perform operations in "linear time" - O(n) or greater.

While this approach achieves our goal, each item contained in the set must be evaluated. A function like this is said to run in linear time because its speed is dependent on its input size. In other words, the algorithm becomes less efficient as its input size (n) grows.

LOGARITHMIC TIME

Our next approach uses a technique called ***binary search***. With this method, we apply our knowledge about the data to help reduce our search criteria.

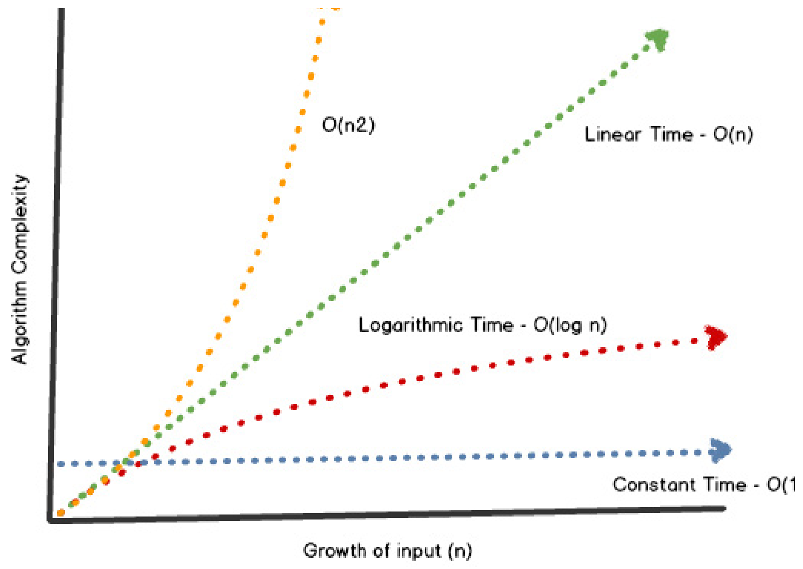
```
extension Array where Element: Comparable {

      mutating func binarySearch(forElement key: Element) -> Bool {

        var result = false

        //establish indices
        let min = self.startIndex
        let max = self.endIndex - 1
        let mid = self.midIndex()


        //check bounds
        if key > self[max] || key < self[min] {
            print("search value \(key) not found..")
            return false
        }

         //evaluate chosen number
        let n = self[mid]

        print(String(describing: n) + "value attempted..")


        if n > key {
            var slice = Array(self[min...mid - 1])
            result = slice.binarySearch(forElement: key)
        }

        else if n < key {
            var slice = Array(self[mid + 1...max])
            result = slice.binarySearch(forElement: key)
        }

        else {
            print("search value \(key) found..")
            result = true
        }

        return result
    }


    //returns middle index
    func midIndex() -> Index {
        return startIndex + (count / 2)
    }

}

//execute search
let isFound: Bool = numberList.binarySearch(forElement: 8)
```

To recap, we know we're searching a sorted `Array` to find a specific value. By applying our understanding of data, we assume there is no need to search values less than the `key`. For example, to find the value at array index `8`, it would be impossible to find that value at array index `0 - 7`.

By applying this logic we substantially reduce the amount of times the `Array` is checked. This type of search is said to work in *logarithmic time* and is represented with the symbol `O(log n)`. Overall, its complexity is minimized when the size of its inputs `(n)` grows. Here's a table that compares the different techniques:

Array Size (n)	Search Key	O(n) - No of Checks	O(log n) - No of Checks
10	8	8	2
20	12	12	3
50	23	23	5
75	48	48	6
100	64	64	7
500	235	235	9

Plotted on a graph, it's easy to compare the running time of popular search and sorting techniques. Here, we can see how most algorithms have relatively equal performance with small datasets. It's only when we apply large datasets that we're able to see clear differences.



Algorithms that perform at a constant rate (regardless of input size) are said to perform at **constant time** - `O(1)`. As we'll see, hash tables are often cited as an important data structure for managing constant time operations. Simple matching and/or comparison techniques often occur in polynomial time, or `O(n`^{2}`)` .

CONSTANT TIME

A term used when evaluating algorithms that always perform a "constant" speed, regardless of their input size. Common algorithms that perform constant time operations include Stacks & Hash Tables.

ALGORTHIMIC THINKING

Algorithms, like design patterns, are based on their approach. As a result, almost any technique can be used to solve a problem. The real benefit is knowing *why* one technique is preferred over another.

Knowledge of Big-O Notation is the first step towards acquiring "algorithmic thinking". Since there are usually multiple ways to solve a problem, being able to analyze and ultimately refine an approach will allow one to create both efficient and effective solutions.

BASIC SORTING

Sorting is an essential task when managing data. As we saw with Big O Notation, sorted data allows us to implement efficient algorithms. Our goal with sorting is to move from disarray to order. This is done by arranging data in a logical sequence so we'll know where to find information. Sequences can be easily implemented with integers, but can also be achieved with characters (e.g., alphabets), and other sets like binary and hexadecimal numbers. In the examples below, we'll use various techniques to sort the following array:

```
//a simple array of unsorted integers
var numberList : Array<Int> = [8, 2, 10, 9, 11, 1, 7, 3, 4]
```

With a small list, it's easy to visualize the problem. To arrange our set into an ordered sequence, we can implement an *invariant*. In computer science, invariants represent assumptions that remain unchanged throughout execution. To see how this works, consider the insertion sort algorithm.

INSERTION SORT

Insertion sort is one of the more basic algorithms in computer science. The insertion sort ranks elements by iterating through a collection and positions elements based on their value. The set is divided into sorted and unsorted halves and repeats until all elements are sorted:

```
extension Array where Element: Comparable {

     func insertionSort() -> Array<Element> {

        //check for trivial case
        guard self.count > 1 else {
            return self
        }

        //mutated copy
        var output: Array<Element> = self

        for primaryIndex in 0..<output.count {

            let key = output[primaryIndex]
            var secondaryIndex = primaryIndex

            while secondaryIndex > -1 {
                 if key < output[secondaryIndex] {
```

next page >>

```
                        //move to correct position
                        output.remove(at: secondaryIndex + 1)
                        output.insert(key, at: secondaryIndex)
                    }

                    secondaryIndex -= 1
                }
            }

            return output
        }
 }

//execute sort
let results: Array<Int> = numberList.insertionSort()
```

BUBBLE SORT

The bubble sort is another common sorting technique. Like insertion sort, the bubble sort algorithm combines a series of steps with an invariant. The function works by evaluating pairs of values. Once compared, the position of the largest value is swapped with the smaller value. Completed enough times, this ***"bubbling"*** effect eventually sorts all elements in the list:

```
 extension Array where Element: Comparable {

  func bubbleSort() -> Array<Element> {

        //check for trivial case
        guard self.count > 1 else {
            return self
        }

        //mutated copy
        var output: Array<Element> = self

        for primaryIndex in 0..<self.count {
            let passes = (output.count - 1) - primaryIndex

            for secondaryIndex in 0..<passes {
                let key = output[secondaryIndex]

                //swap positions
                if (key > output[secondaryIndex + 1]) {
                    output.swapAt(secondaryIndex, secondaryIndex + 1)
                }
            }
        }

        return output
    }

}

//execute sort
let results: Array<Int> = numberList.bubbleSort()
```

SELECTION SORT

Similar to bubble sort, the **selection sort** algorithm ranks elements by iterating through a collection and swaps elements based on their value. The set is divided into sorted and unsorted halves and repeats until all elements are sorted:

```
 extension Array where Element: Comparable {

    func selectionSort() -> Array<Element> {

        //check for trivial case
        guard self.count > 1 else {
            return self
        }

        //mutated copy
        var output: Array<Element> = self

        for primaryIndex in 0..<output.count {

            var minimum = primaryindex
            var secondaryindex = primaryindex + 1

            while secondaryIndex < output.count {

              //store lowest value as minimum
                if output[minimum] > output[secondaryIndex] {
                    minimum = secondaryIndex
                }
                secondaryIndex += 1
            }


            //swap minimum value with array iteration
            if primaryIndex != minimum {
                output.swapAt(primaryIndex, minimum)
            }

        }

        return output
    }

}

//execute sort
let results: Array<Int> = numberList.selectionSort()
```

Besides insertion, selection and bubble sort, there are many other sorting algorithms. As we'll see, data structures such as Binary Search Trees, Tries and Heaps also aid in sorting elements, but do so as part of their insertion process.

ADVANCED SORTIING

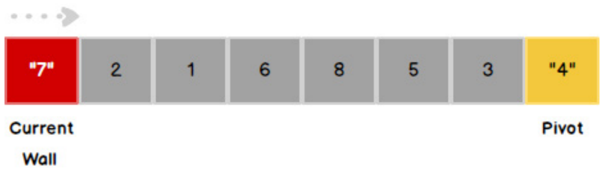
The sorting functions reviewed in the previous chapter each performed with a time complexity of `O(n²)`. While good for interview situations and general academic knowledge, algorithms like `insertionSort` and `bubbleSort` are seldom used in production code. The **Quicksort** algorithm, however, has broader practical application and usage. The commonly used algorithm is found in both code libraries and real-life projects. Quicksort also features a time complexity of `O(n log n)` and applies a **divide & conquer** strategy. This combination results in advanced algorithmic performance.

DIVIDE & CONQUER

The process of deconstructing a single, complex system into a series of sub-systems. In computer science, divide & conquer strategies can be found in search/sorting algorithms, graphs and dynamic programming.

HOW IT WORKS

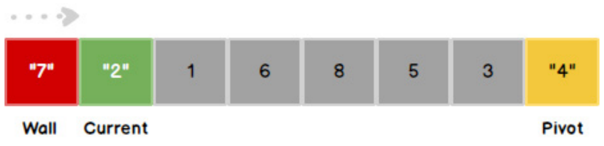
Quicksort applies a series of rules to "*swap*" pairs of values. Swap events are performed "*in place*" so no additional structures are needed to process data. In our implementation, special variables named `wall` and `pivot` will help manage the swapping process:

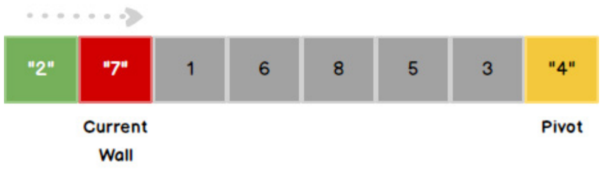


The above illustration shows the algorithm at the start of its sorting process. The function begins by comparing each `index` value to a "comparison" value (e.g., `pivot`). The `wall` represents the position of values that have been swapped or evaluated. Since we've just started the sorting process, the `current` and `wall` indices are shown as the same value (eg., `7`).

MAKING COMPARISONS

The next step compares the `current` and `pivot` values. Since the `current` value (e.g., `7`) is greater than the `pivot` (e.g., `4`), no swap occurs. However, the `current` index advances to the next position.

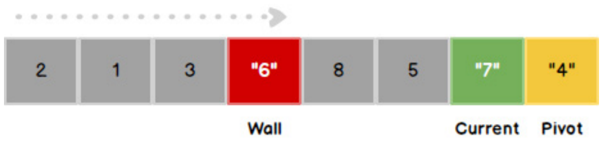




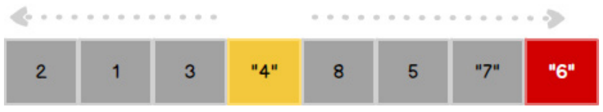
In this step, the current value (e.g., 2) is again compared with the pivot. Since 2 is less than 4, the wall is swapped with the current index value. Once complete, the wall advances to the next index position.

CONQUERING THE DIVIDE

The process of comparing and swapping occurs until the current index meets the pivot. Once complete, the pivot is swapped with the wall index value. Interestingly, once the pivot is swapped, it's considered **sorted**. Even though most values remain unsorted (at this phase), all values less than 4 are now positioned to the left. Conversely, all values greater than 4 are positioned to the right.



The current index meets the pivot



The wall is swapped with the pivot

The initial pivot value of 4 has been used to show how Quicksort can divide a collection into relatively equal segments. This process is called ***partitioning***. To sort the remaining values, each value left or right of the initial pivot is also treated as a pivot and the process is repeated.

THE CODE

In code, the entire algorithm is expressed as two functions. The main quickSort function manages the overall execution - specifically, the selection of each pivot value. Similar to our other sorting algorithms, the quickSort implementation is written as an Array extension. However, the nested function applies ***recursion*** as its main control structure:

```
extension Array where Element: Comparable {

  mutating func quickSort() -> Array<Element> {

        func qSort(start startIndex: Int, _ pivot: Int) {

            if (startIndex < pivot) {
                let iPivot = qPartition(start: startIndex, pivot)
                qSort(start: startIndex, iPivot - 1)
                qSort(start: iPivot + 1, pivot)
            }
        }

        qSort(start: 0, self.endIndex - 1)
        return self

    }
}

//execute sort
var sequence: Array<Int> = [7, 2, 1, 6, 8, 5, 3, 4]
let results = sequence.quickSort()
```

Finally, the `qPartition` process sorts a selected `pivot` value to its correct `index` position:

```
    //sorts selected pivot
    mutating func qPartition(start startIndex: Int, _ pivot: Int) -> Int {

        var wallIndex: Int = startIndex

        //compare range with pivot
        for currentIndex in wallIndex..<pivot {

            print("current is: \(self[currentIndex]). pivot is \(self[pivot])")

            if self[currentIndex] <= self[pivot] {
                if wallIndex != currentIndex {
                    self.swapAt(currentIndex, wallIndex)
                }

                //advance wall
                wallIndex += 1
            }
        }

        //move pivot to final position
        if wallIndex != pivot {
            self.swapAt(wallIndex, pivot)
        }

        return wallIndex
    }
```

GENERICS

The introduction of Swift brings a new series of tools that make coding more friendly and expressive. Along with its simplified syntax, Swift borrows from the success of other languages to prevent common programming errors like **null pointer exceptions** and **memory leaks**.

To contrast, Objective-C has often been referred to as 'the wild west' of code. While extensive and powerful, many errors in Objective-C apps are discovered at runtime. This delay in error discovery is usually due to programming mistakes with memory management and type cast operations. For this essay, we'll review a new design technique called **generics** and will explore how this allows code to be more useful and type-safe.

BUILDING FRAMEWORKS

As we've seen, data structures are the building blocks for organizing data. For example, linked lists, binary trees and queues provide a blueprint for data processing and analysis. Just like any well-designed program, data structures should also be designed for extensibility and reuse.

To illustrate, assume you're building a simple service that lists a group of students. The data could be easily organized with a linked list and represented in the following manner:

```
//basic structure

class StudentNode {

    var key: Student?
    var next: StudentNode?
 }
```

THE CHALLENGE

While this structure is descriptive and organized, it's not reusable. In other words, the structure is valid for listing students but is unable to manage any other type of data (e.g. teachers). The property `Student` is a class that may include specific properties such as `name`, `schedule` and `grades`. If you attempted to reuse the same `StudentNode` class to manage `Teachers`, this would cause a complier type mismatch.

The problem could be solved through inheritance, but it still wouldn't meet our primary goal of class reuse. This is where generics helps. Generics allows us to build ***generic*** versions of data structures so they can be used in different ways.

NULL POINTER EXCEPTION

In programming, null pointer exceptions occur when a portion of deallocated memory is referenced in a program or application - usually at runtime. Null exceptions are common in other non-Swift languages such as Objective-C, but can still occur in Swift-based programs through the use of implicit unwrapped optionals.

MEMORY LEAK

When programming, a memory leak occurs when an application fails to reclaim the portion of a memory used by a variable. In development terms, we often say that portion of memory is unrecoverable or has "leaked". Prior to the concept of automatic reference counting (ARC), memory leaks were common in Objective-C based applications.

APPLYING GENERICS

If you've reviewed the other topics in this series, you've already seen generics in action. In addition to data structures and algorithms, core Swift functions like arrays and dictionaries also make use of generics. Let's refactor the `StudentNode` to be reusable:

```
//refactored structure

class Person<T> {

   var key: T?
   var next: Person<T>?
 }
```

<T>

When working with generic placeholders, note the variable T isn't a special reserved word in Swift but works as a regular declared variable. As a result, the value of **T** could easily be replaced with any letter of choice, including **U**, **W** or **Y**.

We see several important changes with this revised structure. The class name `StudentNode` has been changed to something more general (e.g., `Person`). The syntax **<T>** seen after the class name is called a *placeholder*. With generics, values seen inside angled brackets (e.g., `T`) are declared variables. Once the placeholder `T` is established, it can be reused anywhere a class reference would be expected. In this example, we've replaced the class type `Student` with the generic placeholder `T`.

THE IMPLEMENTATION

The power of generics can be now be seen through its implementation. With the class refactored, `Person` can now manage lists of `Students`, `Teachers`, or any other type we decide.

```
//a new student
var studentNode = Person<Student>()

//a new teacher
var teacherNode = Person<Teacher>()
```

GENERIC FUNCTIONS

In addition to classes, generic functions can also be developed. As we saw in the previous sorting chapter, algorithms like `insertionSort` and `bubbleSort` rank sets of random numbers. Using a generic Array `Element`, these algorithms can support any type that conforms to the `Comparable` protocol. This includes Swift-based characters as well as numbers.

```
extension Array where Element: Comparable {

 func insertionSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self
```

```
         for primaryindex in 0..<output.count {

            let key = output[primaryindex]
            var secondaryindex = primaryindex

            while secondaryindex > -1 {
                 if key < output[secondaryindex] {

                  //move to correct position
                    output.remove(at: secondaryindex + 1)
                    output.insert(key, at: secondaryindex)
                }

                secondaryindex -= 1
            }
        }

        return output
    }
}

//execute sort
let results: Array<Int> = numberList.insertionSort()
```

GENERIC EXTENSIONS

Generics can also be applied to protocols. Similar to regular type extensions, their implementation can often been seen with **protocol extensions**. As seen in other languages, protocols help define rules (e.g. conformance) with particular objects. In Swift, these rules can be extended with their own implementation. Consider the following:

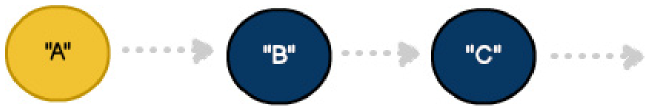
```
//sample protocol with extension

protocol Sortable {
    func isSorted<T: Comparable>(_ sequence: Array<T>) -> Bool
}

extension Sortable {

    func isSorted<T: Comparable>(_ sequence: Array<T>) -> Bool {

        //check trivial cases
        guard sequence.count <= 1 else {
            return true
        }

        var indwex = sequence.startIndex

        //compare sequence values
        while index < sequence.endIndex - 1 {
            if sequence[index] > sequence[sequence.index(after: index)] {
                return false

           }
            index = sequence.index(after: index)
        }

        return true
    }
}
```

LINKED LISTS

A linked list is a basic data structure that provides a way to associate related content. At a basic level, linked lists provide the same functionality as an array. That is, the ability to insert, retrieve, update and remove related items. However, if properly implemented, linked lists can provide additional flexibility. Since objects are managed independently (instead of contiguously - as with an array), lists can prove useful when dealing with large datasets.

HOW IT WORKS

In its basic form, a linked list is comprised of a key and an indicator. The key represents the data you would like to store such as a string or scalar value. Typically represented by a pointer, the indicator stores the location (also called the address) of where the next item can be found. Using this technique, you can chain seemingly independent objects together.



A linked list with three keys and three indicators.

THE DATA STRUCTURE

Here's an example of a ***"doubly"*** linked list structure written in Swift. In addition to storing a key, the structure also provides pointers to the next and previous items. Using generics, the structure can also store any type of object and supports `nil` values. The concept of combining keys and pointers to create structures not only applies to linked lists, but to other types like tries, queues and graphs.

```
//linked list structure

class LLNode<T> {

    var key: T?
    var next: LLNode?
    var previous: LLNode?
}
```

OPTIONAL TYPE

Exclusive to Swift, optionals identify a variable being able to hold a **nil** *value. Swift variables that are permitted to hold a* **nil** *value are identified with a* **?** *or* **!** *character.*

USING OPTIONALS

When creating algorithms its good practice to set your class properties to `nil` before they are used. Like with app development, `nil` can be used to determine missing values or to predict the end of a list. Swift helps enforce this best-practice at compile time through a paradigm called **optionals**. For example, the function `printAllKeys` employs an optional (e.g., `current`) to iterate through linked list items.

```
    //print keys for the class

    func printAllKeys() {

        var current: LLNode? = head

        while current != nil {
            print("link item is: \(String(describing: current?.key!))")
            current = current?.next
        }
    }
```

ADDING LINKS

Here's a function that builds a doubly linked list. The method `append` creates a new item and adds it to the list. The Swift generic type constraint `<T: Equatable>` is also defined to ensure link instances conform to a specific protocol.

```
class LinkedList<T: Equatable> {

    //new instance
    private var head = LLNode<T>()

    //add item
    func append(element key: T) {

        guard head.key != nil else {
            head.key = key
            return
        }

        var current: LLNode? = head

        //position node
        while current != nil {

            if current?.next == nil {

                let childToUse = LLNode<T>()

                childToUse.key = key
                childToUse.previous = current
                current!.next = childToUse
                break
           }
            else {
                current = current?.next
            }
```

next page >>

```
        } //end while
    }

} //end class
```

REMOVING LINKS

Conversely, here's an example of removing items from a list. Removing links not only involves reclaiming memory (for the deleted item), but also reassigning links so the chain remains unbroken.

```
    //remove at specific index

    func remove(at index: Int) {

        //check empty conditions
        if ((index < 0) || (index > (self.count - 1)) || (head.key == nil)) {
            print("link index does not exist..")
            return
        }

        var current: LLNode<T>? =  head
        var trailer: LLNode<T>?
        var listIndex: Int = 0


        //determine if the removal is at the head
        if (index == 0) {
            current = current?.next
            head = current!

            if let headitem = current {
                head = headitem
                counter -= 1
            }
            return
        }


        //iterate through remaining items
        while current != nil {

            if listIndex == index {

                //redirect the trailer and next pointers
                trailer!.next = current?.next
                current = nil
                break
            }

            //update assignments
            trailer = current
            current = current?.next
            listIndex += 1
        }
    }
```

COUNTING LINKS

It can also be convenient to count link items. In Swift, this can be expressed as a *computed property*. For example, the following technique will allow the `class` instance to use a dot notation and is calculated in **constant time - O(1)**.

```
...

    //the number of items - O(1)
    var count: Int {
        return counter
    }


    //empty list check
    func isEmpty() -> Bool {
        return counter == 0 || head.key == nil
    }


    //add link item
    func append(element key: T) {
      ...
      counter += 1
    }


    //insert at specific index
    func insert(_ key: T, at index: Int) {
       ...
      counter += 1
    }


    //remove at specific index
    func remove(at index: Int) {
      ...
      counter -= 1
    }

...
```

FINDING LINKS

Finding links works similar to appending elements. To identify a a found object, the algorithm must cycle through the entire collection. This occurs as a **linear-time** based operation. To ensure the function can return a `nil` result if no elements are found, the `LLNode` return value is declared as an optional.:

```
    //obtain link at a specific index

    func find(at index: Int) ->LLNode<T>? {

        //check empty conditions
        if ((index < 0) || (index > (self.count - 1)) || (head.key == nil)) {
            return nil
        }
```

next page >>

```
        else  {
           var current: LLNode<T> = head
           var x: Int = 0


           //cycle through elements
           while (index != x) {
               current = current.next!
               x += 1
           }

           return current

       } //end else
   }
```

To enhance the linked list class, retrieving elements can also be obtained by subscript:

```
    //find subscript shortcut

     subscript(index: Int) -> LLNode<T>? {
         get {
            return find(at: index)
         }
     }

    //test find operation
    var list = LinkedList<Int>()
    ...

    let newnode: LLNode<Int>? = list[8]
```

TRIES

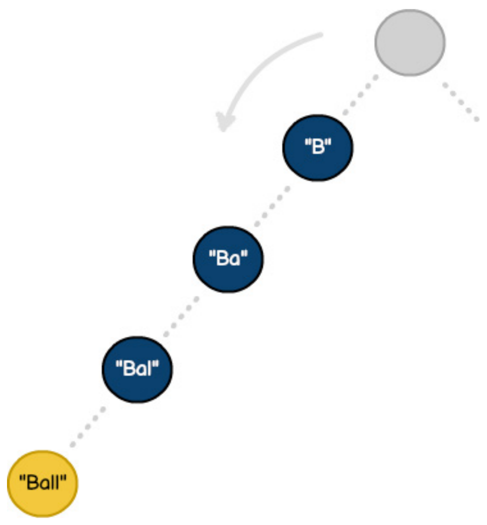
Tries are tree-based data structures that organize information in an hierarchy. Often pronounced "try", the term comes from the English language verb to *retrieve*. While most algorithms are designed to manipulate generic data, tries are commonly used with `Strings`. In this essay, we'll review trie structures and will implement our own model with Swift.

HOW IT WORKS

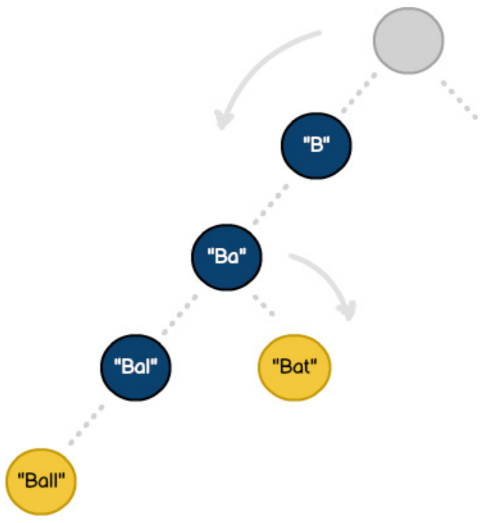
As discussed, tries organize data in a hierarchy. To see how they work, let's build a dictionary that contains the following words:

```
Ball
Balls
Ballard
Bat
Bar
Cat
Dog
```

At first glance, we see words prefixed with the phrase *"Ba"*, while entries like *"Ballard"* combine words and phrases (e.g., *"Ball"* and *"Ballard"*). Even though our dictionary contains a limited quantity of words, a thousand-item list would have the same properties. As with any algorithm, we'll apply our knowledge to build an efficient model. To start, let's create a new trie for the word *"Ball"*:



Tries involve building hierarchies, storing phrases along the way until a word is created (seen in yellow). With so many permutations, it's important to know what qualifies as an actual word. For example, even though we've stored the phrase *"Ba"*, it's not identified as a word. To see the significance, consider the next example:



THE DATA STRUCTURE

Here's an example of a trie data structure written in Swift. In addition to storing a `key`, the structure also includes an `Array` for identifying its children. Unlike a binary search tree, a trie can store an unlimited number of leaf `nodes`. The boolean value `isFinal` will allow us to distinguish words and phrases. Finally, the `level` will indicate the node's level in a hierarchy.

```
//basic trie data structure

public class TrieNode {

    var key: String?
    var children: Array<TrieNode>
    var isFinal: Bool
    var level: Int


    init() {
        self.children = Array<TrieNode>()
        self.isFinal = false
        self.level = 0
    }
}
```

ADDING WORDS

Here's an algorithm that adds words to a trie. Although most tries are recursive structures, our example employs an **iterative technique**. The `while` loop compares the keyword `length` with the `current` node's `level`. If no match occurs, it indicates additional keyword phases remain to be added.

```
//builds a tree hierarchy of dictionary content

func append(word keyword: String) {

    guard keyword.length > 0 else {
        return
    }

    var current: TrieNode = root
    while keyword.length != current.level {

        var childToUse: TrieNode!
        let searchKey: String = keyword.substring(to: current.level + 1)


        //iterate through the node children
        for child in current.children {

            if child.key == searchKey {
               childToUse = child
               break
            }
        }

        //create a new node
        if childToUse == nil {

           childToUse = TrieNode()
           childToUse.key = searchKey
           childToUse.level = current.level + 1
           current.children.append(childToUse)
        }

        current = childToUse


    } //end while


    //add final end of word check
    if keyword.length == current.level {
       current.isFinal = true
       print("end of word reached!")
       return
    }


} //end function
```

ITERATIVE CODING DESIGN

A common programming technique where logic is applied by "looping" or iterating through a list or collection.

A final check confirms our keyword after completing the `while` loop. As part of this final check, we update the `current` node with the `isFinal` indicator. As mentioned, this step will allow us to distinguish words from phrases.

FINDING WORDS

The algorithm for finding words is similar to adding content. Again, we establish a `while` loop to navigate the trie hierarchy. Since the goal will be to return a list of possible words, these will be tracked using an `Array`.

```
//find words based on the prefix

func find(_ keyword: String) -> Array<String>? {


    //trivial case
    guard keyword.length > 0 else {
        return nil
    }


    var current: TrieNode = root
    var wordList = Array<String>()


    while keyword.length != current.level {

        var childToUse: TrieNode!
        let searchKey = keyword.substring(to: current.level + 1)


        //iterate through any child nodes
        for child in current.children {

            if (child.key == searchKey) {
                childToUse = child
                current = childToUse
                break
            }
        }


        if childToUse == nil {
           return nil
        }


    } //end while



    //retrieve the keyword and any descendants
    if ((current.key == keyword) && (current.isFinal)) {
        if let key = current.key {
            wordList.append(key)
        }
    }


    //include only children that are words
    for child in current.children {

        if (child.isFinal == true) {
            if let key = child.key {
```

next page >>

```
                    wordList.append(key)
                }
            }
        }

        return wordList

    } //end function
```

The `search` function checks to ensure keyword phrase permutations are found. Once the entire `keyword` is identified, we start the process of building our word list. In addition to returning `keys` identified as words (e.g., "*Bat*", "*Ball*"), we account for the possibility of returning `nil` by returning an implicit unwrapped optional.

EXTENDING SWIFT

Even though we've written our trie in Swift, we've extended some language features to make things work. Commonly known as *categories* in Objective-C, our algorithms employ two additional Swift *extensions*. The following class extends the functionality of the native `String` class:

```
extension String {

    //compute the length
    var length: Int {
        return self.count
    }

    //returns characters up to a specified integer
    func substring(to: Int) -> String {

        //define the range
        let range = self.index(self.startIndex, offsetBy: to)
        return String(self[..<range])
    }
}
```

STACKS & QUEUES

Stacks and queues are structures that help organize data in a particular order. Their concept is based on the idea that information can be organized similar to how things interact in the real world. For example, a stack of dinner plates can be represented by placing a single plate on a group of existing plates. Similarly, a queue can be easily understood by observing groups of people waiting in a line at a concert or grand opening.

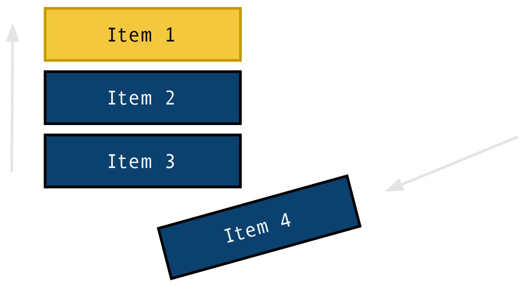
These structures can be implemented in various ways. Any app that makes use of a shopping cart, waiting list or playlist makes use of a stack or queue. In software development, a **call stack** is often used as an essential debugging / analytics tool. For this essay, we'll discuss and implement a stack and queue data structures in Swift.

CALL STACK

Managed by the code complier, the call stack tracks functions executed by a program.

HOW QUEUES WORK

As new `Queue` elements are created they are added to the back of its structure. Items are preserved in order until requested. When a new element is requested, the `top` level item is removed and is sent to the receiver:



Similar to a **linked list**, we'll create a Node class to manage generic stack and queue items:

```
//generic node structure

class Node<T> {

    var key: T?
    var next: Node?
}
```

ENQUEUING OBJECTS

The process of adding items is often referred to as *"enqueuing"*. Here, we define the `method` used to enqueue objects and identify the property `top` that will serve as our starting point:

```
class Queue<T> {

    private var top: Node<T>?

    init() {
      top = Node<T>()
    }

    //enqueue the specified object
    func enQueue(_ key: T) {

        //trivial case
        guard top?.key != nil else {
            top?.key = key
            return
        }

        let childToUse = Node<T>()
        var current = top


        //cycle through the list
        while current?.next != nil {
            current = current?.next
        }


        //append new item
        childToUse.key = key
        current?.next = childToUse
    }

} //end class
```

The process to `enqueue` items is similar to building a generic linked list. However, since queued items can be removed as well as added, we must ensure that our structure supports the absence of values (e.g. `nil`). As a result, the class property `top` is defined as an optional.

To keep the queue generic, the `enQueue` method signature also has a parameter that is declared as type `T`. With Swift, generics usage not only preserves type information, but also ensures objects conform to various protocols.

DEQUEUING OBJECTS

Removing items from the queue is called *dequeuing*. As shown, dequeuing is a two-step process that involves returning the top-level item and reorganizing the queue:

```
    //retrieve items - O(1) constant time

    func deQueue() -> T? {

         //trivial case
         guard top.key != nil else {
            return nil
        }

        //queue the next item
        let queueitem: T? = top.key!

        //use optional binding
        if let nextitem = top.next {
          top = nextitem
        }
        else {
            top = nil
        }

        return queueitem
    }
```

SUPPORTING FUNCTIONS

Along with adding and removing items, supporting functions also include checking for an empty queue as well as retrieving the `top` level item.

```
   //retrieve the top most item
    func peek() -> T? {
        return top?.key
    }


    //check for the presence of a value
    func isEmpty() -> Bool {

        guard top?.key != nil else {
            return true
        }
        return false
    }
```

HOW STACKS WORK

To contrast, a `Stack` positions new elements on `top` and retrieve items on a "first-out" basis. This distinction allows it to perform basic operations such as insertion and retrieval in **constant time - O(1)**. As an iOS Developer, the concept of stacking two or more views to create a **view hierarchy** is common. As we'll see with **Binary Search Trees**, `Stacks` also come in handy when supporting other algorithms.

```
    class Stack<T> {

    private var top: Node<T>
    private var counter: Int = 0


    init() {
        top = Node<T>()
    }

    //the number of items - O(1)
    var count: Int {
```

next page >>

```
            return counter
        }


        //add item to the stack - O(1)
        func push(withKey key: T) {

            //return trivial case
            guard top.key != nil else {
                top.key = key
                counter += 1
                return
            }


            //create new item
            let childToUse = Node<T>()
            childToUse.key = key


            //set new created item at top
            childToUse.next = top
            top = childToUse


            //set counter
            counter += 1
        }


        //remove item from the stack - O(1)
        func pop() {

            if self.count > 1 {
                top = top.next!

                //set counter
                counter -= 1

            }
            else {
                top.key = nil
                counter = 0
            }
        }


        //retrieve the top most item - O(1)
        func peek() -> Node<T> {
            return top
        }


        //check for value - O(1)
        func isEmpty() -> Bool {

            if self.count == 0 {
                return true
            }

            else {
                return false
            }
        }
    }
```

BINARY SEARCH TREES

A binary search tree stores and sorts information in a “tree-based” hierarchy. As discussed in Big O Notation, algorithms provide the best efficiency with sorted data. Binary Search Tree (BST) models are built using specific rules. They should not be mistaken for a binary tree or trie. Those structures also store information in a hierarchy, but employ different rules.

HOW IT WORKS

A binary search tree is comprised of a key and two indicators. The key represents the data you would like to store, such as a string or scalar value. Typically represented with pointers, the indicators store the location (also called the address) of its two children. The `left` child contains a value that is smaller than its parent. Conversely, the `right` child contains a value greater than its parent.

THE DATA STRUCTURE

Here’s an example of a BST written in Swift. Using generics, the structure also stores any type of object and supports `nil` values. The concept of combining `keys` and pointers to create hierarchies not only applies to binary search trees, but to other structures like tries and binary trees.

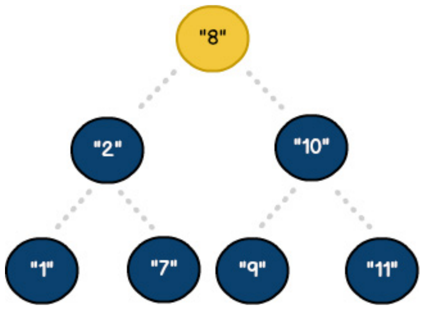
```swift
//generic binary search tree node

class BSNode<T>{

    var key: T?
    var left: BSNode?
    var right: BSNode?
}
```

THE LOGIC

Here’s an example of a simple array written in Swift. We’ll use this data to build our binary search tree:

```swift
//a simple array of unsorted integers
let numberList : Array<Int> = [8, 2, 10, 9, 11, 1, 7]
```

Here's the same list visualized as a balanced binary search tree. It does not matter that the values are unsorted. Rules governing the BST will place each node in its "correct" position accordingly.



Balanced binary search tree - O (log n)

ADDING ELEMENTS

With the `BSNode` structure established, we can now use it to create a BST hierarchy. Along with our class declaration, the type constraint `<T: Comparable>` is also included to ensure generic data can be stored and compared:

```
class BSTree<T: Comparable>{

var root = BSNode<T>()

func append(element key: T) {

      //initialize root
      guard root.key != nil else {

          root.key = key
          root.height = 0
          return
      }


      //set initial indicator
      var current: BSNode<T> = root


      while current.key != nil {

          //check left side
          if key < current.key! {

              if current.left != nil {
                  current = current.left!
              }

              else {

                  //create new element
                  let childToAdd = BSNode<T>()
                  childToAdd.key = key
                  current.left = childToAdd
                  break
```

next page >>

```
                    }
                }

                //check right side
                if key > current.key! {

                    if current.right != nil {
                        current = current.right!
                    }

                    else {

                        //create new element
                        let childToAdd = BSNode<T>()
                        childToAdd.key = key
                        current.right = childToAdd
                        break
                    }
                }

            } //end while
        }
    }
```

As shown, our `append()` method employs an **iterative coding technique** to store recursive `BSNode` instances. As a result, inserting data becomes a straightforward process:

```
let numberList : Array<Int> = [8, 2, 10, 9, 11, 1, 7]

//new BST instance
var bstree = BSTree<Int>()

//sort each item
for number in numberList {
     bstree.append(number)
}
```

FINDING ELEMENTS

Similar to the native Swift `Array.contains()` method, our BST model can also identify specific `BSNodes`:

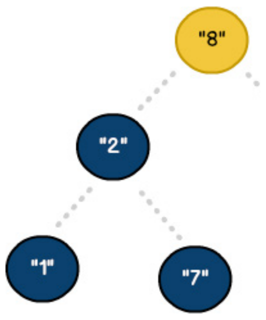
```
   //equality test - O(log n)

    func contains(_ key: T) -> Bool {

        var current: BSNode<T>? = root

        while current != nil {

            guard let testkey = current?.key else {
                return false
            }

            //test match
            if testkey == key {
                return true
            }
```

next page >>

```
                //check left side
                if key < testkey {
                    if current?.left != nil {
                        current = current?.left
                        continue
                    }
                    else {
                        return false
                    }
                }

                //check right side
                if key > testkey {
                    if current?.right != nil {
                        current = current?.right
                        continue
                    }
                    else {
                        return false
                    }
                }

            } //end while

            return false
        }
```

EFFICIENCY

BSTs are powerful due to their consistent rules. However, their greatest advantage is speed. Since data is organized at the time of insertion, a clear pattern emerges. Values less than the `root` node (e.g. `8`) will naturally filter to the left. Conversely, all values greater than the `root` will filter to the right.



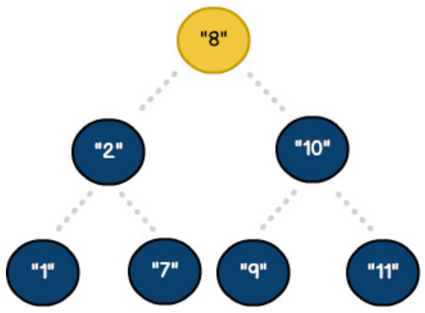
As a result, our understanding of the data allows us to create an effective algorithm. So, for example, if we were searching for the value 7, it's only required that we traverse the left side of the BST. This is due to our search value being smaller than our root node (e.g 8). Binary Search Trees typically provide `O(log n)` for insertion and lookup. Algorithms with an efficiency of `O(log n)` are said to run in *logarithmic time*.

TREE BALANCING

In the previous chapter, we saw how binary search trees (BST) are used to manage data. With basic logic, an algorithm can easily traverse a model, searching data in `O(log n)` time. However, there are occasions when navigating a tree becomes inefficient - in some cases working at `O(n)` time. In this essay, we will review those scenarios and introduce the concept of tree balancing.

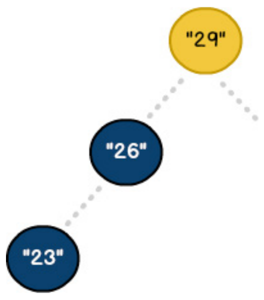
NEW MODELS

To start, let's revisit our original example. Array values from `numberList` were used to build a `tree`. As shown, all elements had either one or two children - otherwise called *leaf* elements. This is known as a *balanced* binary search tree.

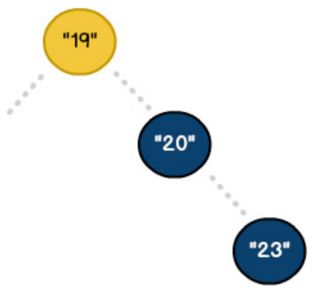


A balanced BST with 7 elements - O (log n)

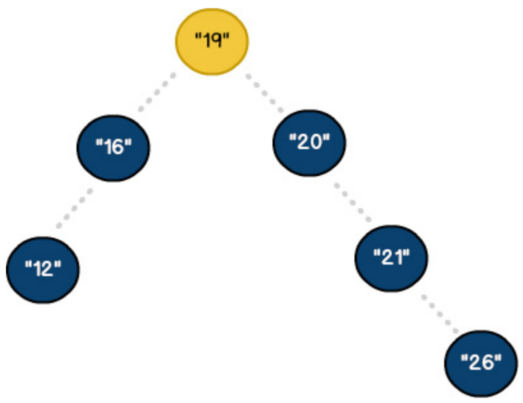
Our model achieved balance not only through usage of the BST `append` algorithm, but also by the way keys were inserted. In reality, there could be numerous ways to populate a `tree`. Without considering other factors, this can produce unexpected results:



An unbalanced "left-heavy" BST with 3 elements - O(n)



An unbalanced "right-heavy" BST with 3 elements - O(n)

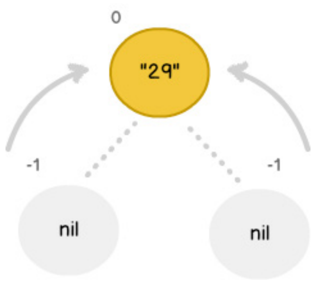


A 6-node BST with left and right imbalances - O(n)

NEW HEIGHTS

To compensate for these imbalances, we need to expand the scope of our algorithm. In addition to `left` / `right` logic, we'll add a new property called `height`. Coupled with specific rules, we can use `height` to detect tree imbalances. To see how this works, let's create a new BST:

```
//a simple array of unsorted integers
let balanceList : Array<Int> = [29, 26, 23]
```



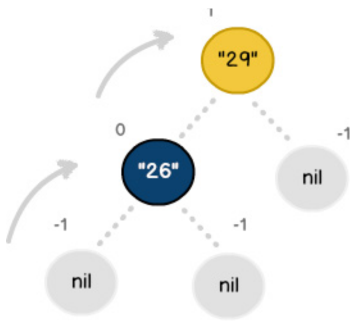
To start, we add the `root` element. As the first item, `left` / `right` leaves don't yet exist so they are initialized to `nil`. Arrows point from the leaf element to the `root` because they are used to calculate its `height`. For math purposes, the `height` of non-existent leaves are set to `-1`.

With a model in place, we can calculate the element's `height`. This is done by comparing the `height` of each leaf, finding the largest value, then increasing that value by `+1`. For the `root` element this equates to `0`. In Swift, these rules can be represented as follows:

```
    private func findHeight(of element: BSNode<T>?) -> Int {

         //check empty leaves
         guard let bsNode = element else {
             return -1
         }
         return bsNode.height
     }


    private func setHeight(for element: BSNode<T>) {

         //set leaf variables
         var elementHeight: Int = 0

         //do comparison and calculate height
         elementHeight = max(findHeight(of: element.left), findHeight(of: element.
right)) + 1

         element.height = elementHeight
     }
```

MEASURING BALANCE

With the `root` element established, we can proceed to add the next value. Upon implementing standard BST logic, item `26` is positioned as the `left` leaf node. As a new item, its `height` is also calculated (i.e., `0`). However, since our model is a hierarchy, we traverse upwards to recalculate its parent `height` value:



Step two: 26 is added to the BST

With multiple elements present, we run an additional check to see if the BST is balanced. In most cases, a `tree` is considered balanced if the `height` difference between its leaf `elements` is less than 2. As shown below, even though no right-side items exist, our model is still valid.

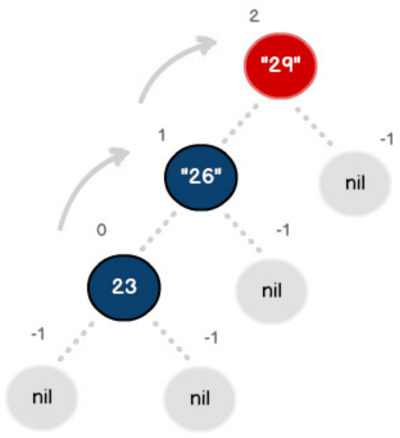
```
//example math for tree balance check

let rootVal: Int!
let leafVal: Int!

leafVal = abs((-1) - (-1)) //equals 0 (balanced)
rootVal = abs((0) - (-1)) //equals 1 (balanced)
```

In Swift, these elements can be checked with the `isTreeBalanced` method.

```
func isTreeBalanced(for element: BSNode<T>?) -> Bool {

        guard element?.key != nil else {
            print("no element provided..")
            return false
        }

        //use absolute value to manage right and left imbalances
        if (abs(findHeight(of: element?.left) - findHeight(of: element?.right)) <= 1) {
            return true
        }
        else {
            return false
        }
    }
```

ADJUSTING THE MODEL

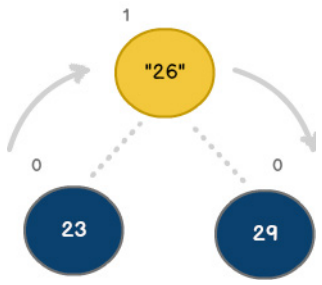
With `29` and `26` added we can proceed to insert the final value (i.e., `23`). Like before, we continue to traverse the `left` side of the tree. However, upon insertion, the math reveals node `29` violates the BST property. In other words, its `subtree` is no longer balanced.



Step three: 23 is added to the BST

```
//example math for tree balance check

let rootVal: Int!
rootVal = abs((1) - (-1)) //equals 2 (unbalanced)
```

For the `tree` to maintain its BST property, we need to change its performance from `O(n)` to `O(log n)`. This can be achieved through a process called *rotation*. Since the model has more nodes to the `left`, we'll balance it by performing a `right` rotation sequence. Once complete, the new model will appear as follows:



Step four: right rotation on node 29

As shown, we've been able to rebalance the BST by rotating the model to the `right`. Originally set as the `root`, node `29` is now positioned as the `right` leaf. In addition, node `26` has been moved to the `root`. In Swift, these changes can be achieved with the following:

```
//perform left or right rotation

private func rotate(element: BSNode<T>) {
   ...

  //create new element

  let childToUse = BSNode<T>()
  childToUse.height = 0
  childToUse.key = element.key

   ...

  //determine side imbalance
  let rightSide = findHeight(of: element.left) - findHeight(of: element.right)
  let leftSide =  findHeight(of: element.right) - findHeight(of: element.left)

   if rightSide > 1 {

       //reset the root node
       element.key = element.left?.key
       element.height = findHeight(of: element.left)

       //assign the new right node
       element.right = childToUse

       //adjust the left node
       element.left = element.left?.left
```

```
            element.left?.height = 0
        }
        .....
    }
```

Even though we undergo a series of steps, the process occurs in `O(1)` time. Meaning, its performance is unaffected by other factors such as number of `leaf` nodes, descendants or tree `height`. In addition, even though we've completed a `right` rotation, similar steps could be implemented to resolve both `left` and `right` imbalances.

TREE NAVIGATION

MEMOIZATION

A coding technique of applying storage and reuse to improve overall algorithmic efficiency.

You may have noticed that to successfully apply our node balancing algorithms, we also need to determine how to traverse upwards in our `BSNode` hierarchy. In other words, when a child element is added, all its *antecedents* must be recalculated (indicated in the diagrams as arrows pointing upwards). If our `BSTree` was a *recursive* structure, we could utilize the in-memory **call stack** to automatically navigate to parent nodes. To achieve the same results in our iterative design, we'll store `BSNode` references using a **Stack** data structure. This technique is known as **memoization**:

```
class BSTree<T: Comparable>{
   ...

   private var elementStack = Stack<BSNode<T>>()
...

 func append(element key: T) {

 ...
        //set initial indicator
        var current: BSNode<T> = root

        while current.key != nil {

            //send reference of current item to stack
            self.push(element: &current)
          .....
        }

        //calculate height and balance of call stack..
        self.rebalance()
}


   //stack elements for later processing - memoization
   private func push(element: inout BSNode<T>) {
        elementStack.push(withKey: element)
    }


  //determine height and balance
  private func rebalance() {

        for _ in stride(from: elementStack.count, through: 1, by: -1) {

            //obtain generic stack node - by reference
            let current = elementStack.peek()

            guard let bsNode: BSNode<T> = current.key else {
                print("element reference not found..")
                continue
```

next page >>

```
                }

                setHeight(for: bsNode)
                rotate(element: bsNode)

                //remove stacked item
                elementStack.pop()
            }
        }
```

In Swift, the memory address of variables can be shared as inout function parameters. As a result, when a `BSNode` reference is pushed to the **Stack**, the original variable can be updated without having to manage new copies of data.

THE RESULTS

With tree balancing, it is important to note that techniques like rotations improve performance, but do not change `tree` output. For example, even though a `right` rotation changes the connections between `nodes`, the overall BST sort order is preserved. As a test, one can traverse a balanced and unbalanced BST (comparing the same values) and receive the same results. In our case, a simple depth-first search will produce the following:

```
//sorted values from a traversal
23, 26, 29
```

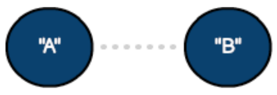
GRAPHS

A graph is a data structure that shows a relationship (e.g., connection) between two or more objects. Because of their flexibility, graphs are one of the most widely used structures in modern computing. Popular tools and services like online maps, social networks, and even the Internet as a whole are based on how objects relate to one another. In this chapter, we'll highlight the key features of graphs and will demonstrate how to create a basic graph with Swift.

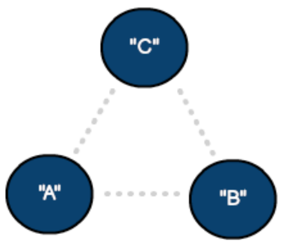
THE BASICS

As discussed, a graph is a model that shows how objects relate to one another. Graph objects are usually referred to as `nodes` or `vertices`. While it would be possible to build a graph with a single node, models that contain multiple vertices better represent real-world applications.

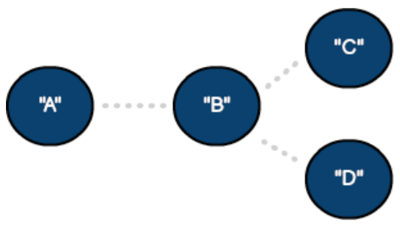
Graph objects relate to one another through connections called `edges`. Depending on your requirements, a `vertex` could be linked to one or more objects through a series of `edges`. It's also possible to create a `vertex` without edges. Here are some basic graph configurations:



An undirected graph with two vertices and one edge.



An undirected graph with three vertices and three edges.



An undirected graph with four vertices and three edges.

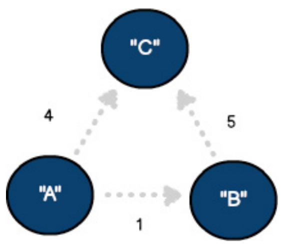
DIRECTED VS. UNDIRECTED

As shown above, there are many ways to configure a graph. An additional option is to set the model to be either directed or undirected. The examples above represent undirected graphs. In other words, the connection between vertices `A` and `B` is equivalent to the connection between vertices `B` and `A`. Social networks are a great example of undirected graphs. Once a request is accepted, both parties (e.g. the sender and recipient) share a mutual connection.

A service like *Google Maps* is a great example of a directed graph. Unlike an undirected graph, directed graphs only support a one-way connection between source `vertices` and their destinations. So, for example, vertex `A` could be connected to `B`, but `A` wouldn't necessarily be reachable through `B`. To show the varying relationship between `vertices`, directed graphs are drawn with lines and arrows.

EDGES & WEIGHTS

Regardless of graph type, it's common to represent the level of connectedness between `vertices`. Normally associated with an `edge`, the `weight` is a numerical value tracked for this purpose. As we'll see, modeling of graphs with edge weights can be used to solve a variety of problems.



A directed graph with three vertices and three weighted edges.

THE VERTEX

With our understanding of graphs in place, let's build a basic directed graph with edge weights. To start, here's a data structure that represents a vertex:

```
public class Vertex {

    var key: String?
    var neighbors: Array<Edge>

    init() {
        self.neighbors = Array<Edge>()
    }

}
```

As we've seen with other structures, the `key` represents the data to be associated with a class instance. To keep things straightforward, our `key` is declared as a `string`. In a production app, the `key` type would be replaced with a generic placeholder, <T>. This would allow the `key` to store any object like an integer, account or profile.

ADJACENCY LISTS

The `neighbors` property is an array that represents connections a `vertex` may have with other vertices. As discussed, a `vertex` can be associated with one or more items. This list of neighboring items is sometimes called an adjacency list and can be used to solve a variety of problems. Here's a basic data structure that represents an `edge`:

```
public class Edge {

    var neighbor: Vertex
    var weight: Int

    init() {
        weight = 0
        self.neighbor = Vertex()
    }

}
```

BUILDING THE GRAPH

With our `vertex` and `edge` objects built, we can use these structures to construct a graph. To keep things straightforward, we'll focus on the essential operations of adding and configuring vertices.

A graph canvas could also be implemented with a Swift "Set" collection type.

```
public class SwiftGraph {

    //declare graph canvas
    private var canvas: Array<Vertex>
    public var isDirected: Bool

    init() {
        canvas = Array<Vertex>()
        isDirected = true
    }

    //create a new vertex
    func addVertex(key key: String) -> Vertex {

        //set the key
        let childVertex: Vertex = Vertex()
        childVertex.key = key


        //add the vertex to the graph canvas
        canvas.append(childVertex)

        return childVertex
    }

}
```

The function `addVertex` accepts a string which is used to create a new `vertex` . The `SwiftGraph` class also has a private property named `canvas` which is used to manage all `vertices`. While not required, the `canvas` can be used to track and manage `vertices` with or without `edges`.

MAKING CONNECTIONS

Once a `vertex` is added, it can be connected to other `vertices`. Here's the process of establishing an `edge`:

```
    //add edge to source vertex

    func addEdge(source: Vertex, neighbor: Vertex, weight: Int) {

        //new edge
        let newEdge = Edge()


        //establish default properties
        newEdge.neighbor = neighbor
        newEdge.weight = weight
        source.neighbors.append(newEdge)


        //check condition for an undirected graph
        if isDirected == false {

            //create a new reversed edge
           let reverseEdge = Edge()

           //establish the reversed properties
           reverseEdge.neighbor = source
           reverseEdge.weight = weight
           neighbor.neighbors.append(reverseEdge)
       }
    }
```

next page >>

The function `addEdge` receives two vertices, identifying them as `source` and `neighbor`. Since our model defaults to a directed graph, a new `Edge` is created and is added to the adjacency list of the source `Vertex`. For an undirected graph, an additional `Edge` is created and added to the neighbor `Vertex`.

As we've seen, there are many components to graph theory. In the next section, we'll examine a popular problem (and solution) with graphs known as *shortest paths*.

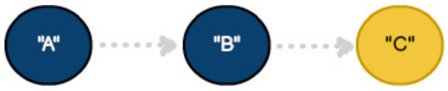
SHORTEST PATHS

In the previous chapter, we saw how graphs show the relationship between two or more objects. Because of their flexibility, graphs are used in a wide range of applications including map-based services, networking and social media. Popular models may include roads, traffic, people and locations. In this chapter, we'll review how to search a graph and will implement a popular algorithm called *Dijkstra's shortest path*.

MAKING CONNECTIONS

The challenge with graphs is knowing how a `vertex` relates to other objects. Consider the social networking website, LinkedIn. With LinkedIn, each profile can be thought of as a single `vertex` that may be connected with other `vertices`.

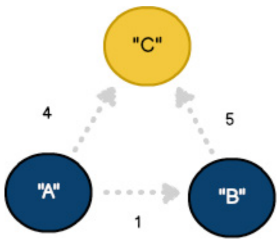
One feature of LinkedIn is the ability to introduce yourself to new people. Under this scenario, LinkedIn will suggest routing your message through a shared connection. In graph theory, the most efficient way to deliver your message is called the *shortest path*.



The shortest path from vertex A to C is through vertex B

FINDING YOUR WAY

Shortest paths can also be seen with map-based services like *Google Maps*. Users frequently use Google Maps to ascertain driving directions between two points. As we know, there are often multiple ways to get to any destination. The shortest route will often depend on various factors such traffic, road conditions, accidents and time of day. In graph theory, these external factors represent `edge` weights.

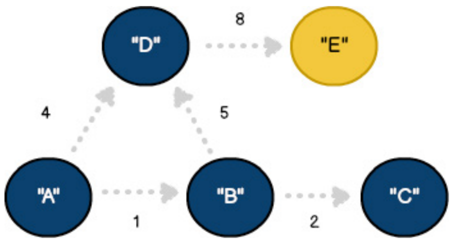


The shortest path from vertex A to C is through vertex A.

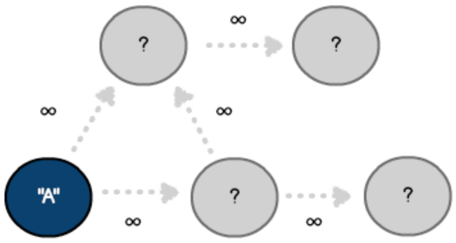
This example illustrates some key points we'll see in *Dijkstra's algorithm*. In addition to there being multiple ways to arrive at vertex C from A, the shortest path is assumed to be through vertex B. It's only when we arrive to vertex C from B, we adjust our interpretation of the shortest path and change direction (e.g. 4 < (1 + 5)). This change in direction is known as the *greedy approach* and is used in similar problems like the *traveling salesman*.

INTRODUCING DIJKSTRA

Edsger Dijkstra's algorithm was published in 1959 and is designed to find the shortest path between two vertices in a directed graph with non-negative edge weights. Let's review how to implement this in Swift.



Even though our model is labeled with key values and edge weights, our algorithm can only see a subset of this information. Starting at the source vertex, our goal will be to traverse the graph.



USING PATHS

Throughout our journey, we'll track each node visit in a custom data structure called Path. The total will manage the cumulative edge weight to reach a particular destination. The previous property will represent the Path taken to reach that vertex.

```
//the path class maintains objects that comprise the “frontier”

class Path {

    var total: Int
    var destination: Vertex
    var previous: Path?

    //object initialization
    init() {
```

next page >>

```
            destination = Vertex()
            total = 0
        }
    }
```

DECONSTRUCTING DIJKSTRA

With all the graph components in place, let's see how it works. The method `processDijkstra` accepts the vertices `source` and `destination` as parameters. It also returns a `Path`. Since it may not be possible to find the `destination`, the return value is declared as a Swift optional.

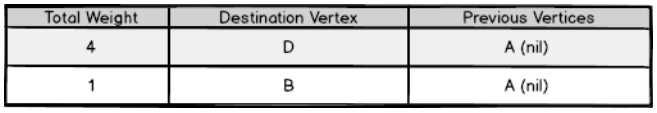
```
//process Dijkstra's shortest path algorithm
func processDijkstra(source: Vertex, destination: Vertex) -> Path? {
    ...
}
```

BUILDING THE FRONTIER

As discussed, the key to understanding Dijkstra's algorithm is knowing how to traverse the graph. To help, we'll introduce a few rules and a new concept called the `frontier`.

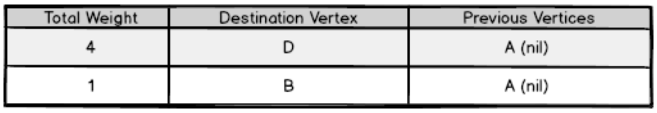
```
var frontier = Array<Path>()
var finalPaths = Array<Path>()

//use source edges to create the frontier
for e in source.neighbors {

    let newPath: Path = Path()

    newPath.destination = e.neighbor
    newPath.previous = nil
    newPath.total = e.weight

    //add the new path to the frontier
    frontier.append(newPath)

}
```

The algorithm starts by examining the source `vertex` and iterating through its list of `neighbors`. Recall from the previous chapter, each `neighbor` is represented as an `edge`. For each iteration, information about the neighboring `edge` is used to construct a new `Path`. Finally, each `Path` is added to the `frontier`.



Total Weight	Destination Vertex	Previous Vertices
4	D	A (nil)
1	B	A (nil)

The frontier visualized as a list



Total Weight	Destination Vertex	Previous Vertices
4	D	A (nil)
1	B	A (nil)

The frontier visualized as a list

With our `frontier` established, the next step involves traversing the `Path` with the smallest total `weight` (e.g., `B`). Identifying the `bestPath` is accomplished using this linear approach:

```
//construct the best path
var bestPath: Path = Path()

while frontier.count != 0 {

    //support path changes using the greedy approach
    bestPath = Path()
    var pathIndex: Int = 0


    for x in 0..<frontier.count {

        let itemPath: Path = frontier[x]

        if (bestPath.total == 0) || (itemPath.total < bestPath.total) {
            bestPath = itemPath
            pathIndex = x
        }

    }
```

An important section to note is the `while loop` condition. As we traverse the graph, `Path` objects will be added and removed from the `frontier`. Once a `Path` is removed, we assume the shortest path to that `destination` as been found. As a result, we know we've traversed all possible paths when the `frontier` reaches zero.

```
//enumerate the bestPath edges

for e in bestPath.destination.neighbors {

     let newPath: Path = Path()

     newPath.destination = e.neighbor
     newPath.previous = bestPath
     newPath.total = bestPath.total + e.weight

     //add the new path to the frontier
     frontier.append(newPath)

    }

    //preserve the bestPath
    finalPaths.append(bestPath)

    //remove the bestPath from the frontier
    frontier.remove(at: pathIndex)

} //end while
```

As shown, we've used the `bestPath` to build a new series of `Paths`. We've also preserved our visit history with each new object. With this section completed, let's review our changes to the `frontier`:

Total Weight	Destination Vertex	Previous Vertices
4	D	A (nil)
6	D	B - A (nil)
3	C	B - A (nil)

The frontier after visiting two additional vertices.

At this point, we've learned a little more about our graph. There are now two possible `paths` to vertex `D`. The shortest `path` has also changed to arrive at vertex `C`. Finally, the `Path` through route `A-B` has been removed and has been added to a new structure named `finalPaths`.

A SINGLE SOURCE

Dijkstra's algorithm can be described as "single source" because it calculates the `path` to every `vertex`. In our example, we've preserved this information in the `finalPaths` array.

Total Weight	Destination Vertex	Previous Vertices
1	B	A (nil)
3	C	B - A (nil)
4	D	A (nil)
6	D	B - A (nil)
12	E	D - A (nil)
14	E	D - B - A (nil)

The finalPaths once the frontier reaches zero. As shown, every permutation from vertex A is calculated.

Based on this data, we can see the shortest path to vertex `E` from `A` is `A-D-E`. The bonus is that in addition to obtaining information for a single route, we've also calculated the shortest `path` to each `node` in the graph.

ASYMPTOTICS

Dijkstra's algorithm is an elegant solution to a complex problem. Even though we've used it effectively, we can improve its performance by making a few adjustments. We'll analyze those details in the next chapter.

HEAPS

In the previous chapter, we reviewed Dijkstra's algorithm for searching a graph. Originally published in 1959, this popular technique for finding the shortest path is an elegant solution to a complex problem. The design involved many parts including graph traversal, custom data structures and the greedy approach.

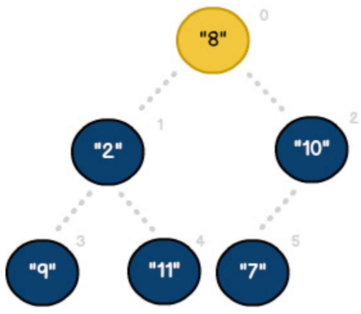
When designing programs, it's great to see them work. With Dijkstra, the algorithm did allow us to find the shortest path between a source vertex and destination. However, our approach could be refined to be more efficient. In this essay, we'll enhance the algorithm with the addition of binary heaps.

HOW IT WORKS

In its basic form, a `heap` is just an `Array`. However, unlike an `Array`, we *visualize* it as a `tree`. The term *"visualize"* implies we use processing techniques normally associated with recursive data structures. This shift in thinking has numerous advantages. Consider the following:

```
//a simple array of unsorted integers
let numberList : Array<Int> = [8, 2, 10, 9, 11, 7]
```

As shown, `numberList` can be easily represented as a `heap`. Starting at index `0`, items fill a corresponding spot as a parent or child node. Each parent also has two children with the exception of index `2`.



An array visualized as a "nearly complete" tree

Since the arrangement of values is sequential, a simple pattern emerges. For any `node`, we can accurately predict its position using these formulas:

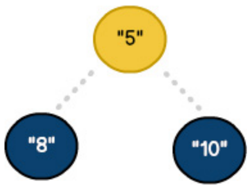
$$parent = floor(\frac{i-1}{2})$$

$$left = 2i + 1 \qquad right = 2i + 2$$

Formulas to calculate heap indices

SORTING HEAPS

An interesting feature of heaps is their ability to sort data. As we've seen, many algorithms are designed to sort entire datasets. When sorting heaps, nodes can be arranged so each parent contains a lesser value than its children. In computer science, this is called a *min-heap*.



A heap structure that maintains the min-heap property

EXPLORING THE FRONTIER

With Dijkstra, we used a concept called the `frontier`. Coded as a simple `Array`, we compared the total `weight` of each path to find the shortest path.

```
        //construct the best path

        var bestPath: Path = Path()

        while frontier.count != 0 {

            //support path changes using the greedy approach
            bestPath = Path()

            var pathIndex: Int = 0

            for x in 0..<frontier.count {

                let itemPath: Path = frontier[x]

                if  (bestPath.total == 0) || (itemPath.total < bestPath.total) {
                    bestPath = itemPath
                    pathIndex = x
                }
            }
```

While it accomplished our goal, we applied a *brute force* technique. In other words, we examined *every* potential `Path` to find the shortest path. This code segment executes in *linear time* or `O(n)`. If the `frontier` contained a million rows, how would this impact the algorithm's overall performance?

THE HEAP STRUCTURE

Let's create a more efficient `frontier`. Named `PathHeap`, the data structure will extend the functionality of an `Array`.

```
public class PathHeap {

    private var heap: Array<Path>

    init() {
        heap = Array<Path>()
    }

    //the number of frontier items
    var count: Int {
        return self.heap.count
    }
}
```

The `PathHeap` class includes two properties. To support good design, `heap` has been declared a private property. To track the number of items, `count` is declared as a computed property.

BUILDING THE QUEUE

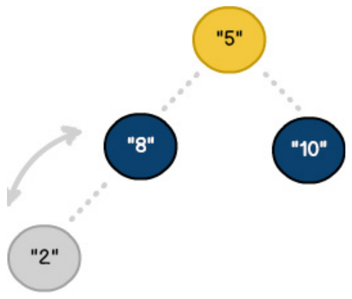
Searching the `frontier` more efficiently than `O(n)` will require a new way of thinking. We can improve our performance to `O(1)` with a heap. Using heapsort formulas, our new approach will involve *arranging* index values so the smallest item is positioned at the `root`.

```
    //sort shortest paths into a min-heap (heapify)

    func enQueue(key: Path) {

        heap.append(key)

        var childIndex: Float = Float(heap.count) - 1
        var parentIndex: Int! = 0


        //calculate parent index
        if childIndex != 0 {
            parentIndex = Int(floorf((childIndex - 1) / 2))
        }

        var childToUse: Path
        var parentToUse: Path

        //use the bottom-up approach
        while childIndex != 0 {

            childToUse = heap[Int(childIndex)]
            parentToUse = heap[parentIndex]
```

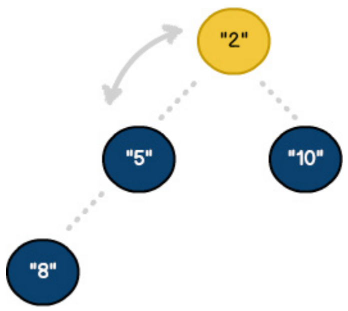
next page >>

```
                //swap child and parent positions
                if childToUse.total < parentToUse.total {
                    heap.swapAt(parentIndex, Int(childIndex))
                }

                //reset indices
                childIndex = Float(parentIndex)

                if (childIndex != 0) {
                    parentIndex = Int(floorf((childIndex - 1) / 2))
                }
            }
        }
```

The `enQueue` method accepts a single `Path` as a parameter. Unlike other sorting algorithms, our primary goal isn't to sort each item, but to find the smallest value. This means we can increase our efficiency by comparing a subset of values.



The enQueue process compares a newly added value



The process continues until the smallest value is positioned at the root

Since the `enQueue` method maintains the min-heap property, we eliminate the task of finding the shortest path. In other words, we increase the algorithm's efficiency to `O(1)`. Here, we implement a basic `peek` method to retrieve the `root`-level item.

```
    //obtain the minimum path

    func peek() -> Path? {

        if heap.count > 0 {
           return heap[0]   //the shortest path
        }
        else {
            return nil
        }
    }
```

THE RESULTS

With the frontier refactored, let's see the applied changes. As new `Paths` are discovered, they are automatically sorted by the `frontier`. The count property forms the base case for our `loop` condition and the `bestPath` is retrieved using the `peek` method.

```
...

 let frontier: PathHeap = PathHeap()

...
    //construct the best path
    while frontier.count != 0 {

      //use the greedy approach to obtain the best path
      guard let bestPath: Path = frontier.peek() else {
        break
      }

    ...
```

HEAPS & GENERICS

It's great seeing how a heap can be used to improve the time complexity of a solution. To extend our model for general use, we can also create a more generalized algorithm:

```
//a generic heap structure

class Heap<T: Comparable> {

    private var items: Array<T>
    private var heapType: HeapType


    //min-heap default initialization
    init(type: HeapType = .Min) {

        items = Array<T>()
        heapType = type
    }


    //the min or max value
    func peek() -> T? {
```

next page >>

TIME COMPLEXITY - (BIG O NOTATION)

When evaluating algorithmic performance, functions are often analyzed based on their speed of operation (e.g. time) as well as how much storage "space" they may occupy.

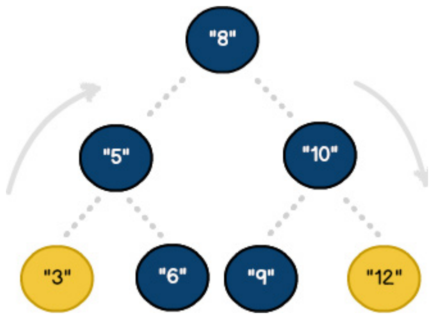
```
        if items.count > 0 {
            return items[0] //the min or max value
        }
        else {
            return nil
        }
    }

    //addition of new items
    func enQueue(_ key: T) {

        items.append(key)

        var childIndex: Float = Float(items.count) - 1
        var parentIndex: Int = 0

        //calculate parent index
        if childIndex != 0 {
            parentIndex = Int(floorf((childIndex - 1) / 2))
        }

        var childToUse: T
        var parentToUse: T

        //use the bottom-up approach
        while childIndex != 0 {


            childToUse = items[Int(childIndex)]
            parentToUse = items[parentIndex]


            //heapify depending on type
            switch heapType {

            case .Min:

                //swap child and parent positions
                if childToUse <= parentToUse {
                    items.swapAt(parentIndex, Int(childIndex))
                }

            case .Max:

                //swap child and parent positions
                if childToUse >= parentToUse {
                    items.swapAt(parentIndex, Int(childIndex))
                }
            }

            //reset indices
            childIndex = Float(parentIndex)

            if childIndex != 0 {
                parentIndex = Int(floorf((childIndex - 1) / 2))
            }
        }
    }
}


//used for generic heap data structure processing
enum HeapType {

    case Min
    case Max
}
```

TRAVERSALS

Throughout this series we've explored various data structures such as binary search trees and graphs. Once established, these objects work like a database - managing data in a structured format. Like most databases, their contents can also be explored through a process called *traversal*. In this chapter, we'll review traversing data structures and will examine the popular techniques of *Depth-First* and *Breadth-First* search.

DEPTH-FIRST SEARCH

Traversals are based on the idea of *visiting* each node in a data structure. In practical terms, traversals can be seen through everyday activities like network administration. To meet security requirements, administrators will often deploy software updates to an entire network as a single task. To see how traversal works, let's introduce the concept of *Depth-First* Search (DFS). As shown with this binary search tree, our goal will be to explore the `left` side of the model, visit the `root` node, then visit the `right` side.



The yellow nodes represent the first and last nodes in the traversal. The algorithm requires little code, but introduces some interesting concepts.

```
class BSNode<T>{

var key: T?
var left: BSNode?
var right: BSNode?


//depth-first traversal
func traverse() {

    guard self.key != nil else {
        print(“no key provided..”)
        return
```

next page >>

```
            }

            //process the left side
            if self.left != nil {
                left?.traverse()
              }

            //process the right side
            if self.right != nil {
                right?.traverse()
            }
        }
    }
```

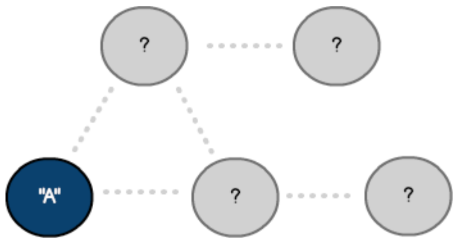
At first glance, we note the algorithm makes use of recursion. The `AVLTree` node (e.g. self), contains a `key`, as well as pointer references to its left and right nodes. For each side, (e.g., `left` and `right`) the base case consists of a straightforward check for `nil`. This process allows us to traverse the entire structure starting at the lowest value. When applied, the algorithm produces the following output:

```
3, 5, 6, 8, 9, 10, 12
```

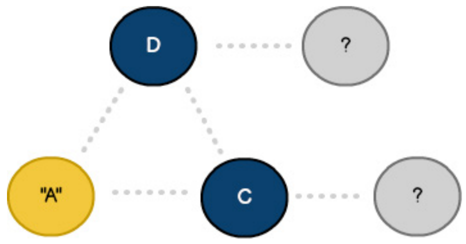
BREADTH-FIRST SEARCH

Breadth-First Search (BFS) is another technique used for traversing data structures. This algorithm is designed for open-ended data models and is typically used with graphs.

Our BFS algorithm combines techniques previously discussed including stacks, queues and shortest paths. With BFS, our goal is to visit all neighbors before visiting our neighbor's, *neighbor*. Unlike Depth-First Search, the algorithm is based on logical discovery.



We've chosen `vertex` A as the starting point. Unlike Dijkstra, BFS has no concept of a `destination` or `frontier`. The algorithm is complete when all connected nodes have been visited. As a result, the starting point could have been any node in our graph.



Vertex A is marked as visited once its neighbors have been added to the queue.

As discussed, BFS works by exploring neighboring vertices. Since our data structure is an undirected graph, we need to ensure each `node` is visited only once. To account for this requirement we can use a generic `queue`:

```
//breadth first search

func traverse(_ startingv: Vertex) {

    //establish a new queue
    let graphQueue = Queue<Vertex>()


    //queue a starting vertex
    graphQueue.enQueue(startingv)


    while !graphQueue.isEmpty() {

        //traverse the next queued vertex
        let vitem = graphQueue.deQueue() as Vertex!

        guard vitem != nil else {
            return
        }

        //add unvisited vertices to the queue
        for e in vitem!.neighbors {
            if e.neighbor.visited == false {
                graphQueue.enQueue(e.neighbor)
            }
        }

        vitem!.visited = true
        print("traversed vertex: \(vitem!.key!)..")

    } //end while

    print("graph traversal complete..")

} //end function
```

The process starts by adding a single `vertex` to the `queue`. As nodes are dequeued, their neighbors are also added to the `queue`. The process is complete when all vertices are visited. To ensure nodes are not visited more than once, each `vertex` is marked with a `boolean` flag.

HASH TABLES

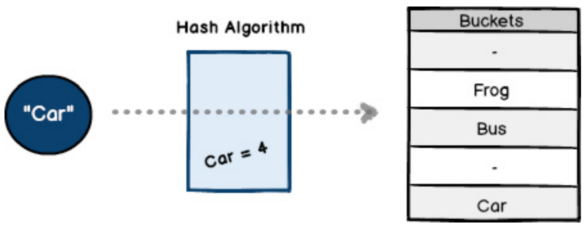
A *hash table* is a data structure that groups values to an `index`. In some cases, built-in Swift data types *(e.g dictionaries)* also accomplish this goal. Because of their efficiency, hash tables should be regarded as an important tool for solving technical questions. In this chapter, we'll examine the advantages of hash tables and will build our own custom hash table model in Swift.

KEYS & VALUES

As noted, there are numerous data structures that group values to an `index`. By definition, linked lists provide a straightforward way to associate related items. While the advantage of a linked list is flexibility, their downside is performance. Since the primary way to search a list is to (sequentially) traverse a set, their efficiency is typically limited to linear time or `O(n)`. To contrast, a *dictionary* associates a value to a user-defined key. This helps pinpoint operations to specific entries. As a result, a well designed hash table and/or dictionary operates in **constant time - O(1).**

THE BASICS

As the name implies, a hash table consists of two parts - an `index` and `value`. However, unlike a `Dictionary`, the index (e.g. hash) is a *computed* sequence of numbers and /or characters. The process that creates a unique hash is known as a hash algorithm:



The above illustrates the components of a hash table. Using an `Array`, values are stored in non-contiguous slots called `buckets`. The position of each value is computed by the hash function. As we'll see, most algorithms use all or a portion of their input to create a unique *hash*. Also, unlike a `Dictionary`, the index is not stored with the `buckets` object.

THE STRUCTURES

Using generics, the recursive `Node` data structure previously introduced with **Stacks/Queues** can also be used for our hash table. In this case, our `Node.key` can maintain any combination of items, including custom objects:

```
//a generic hash element

class Node<T> {

    var key: T?
    var next: Node?
}
```

We can also establish a `Result` enum structure to monitor the hash table process:

```
//used for generic processing

enum Result {

    case Success
    case Collision
    case NotFound
    case NotSupported
    case Fail
}
```

THE BUCKET

Before using our table, we must first define a `bucket` structure. If you recall, buckets are used to group `Element` items. Since items will be stored in a *non-contiguous* fashion, we must first define our collection size.

```
//generic table structure

class HashTable<T: Keyable> {

    private var buckets: Array<Node<T>?>

    init(capacity: Int) {
        self.buckets = Array<Node<T>?>(repeatElement(nil, count: capacity))
    }
}
```

TYPE CONFORMANCE

An interesting question arises when building a generic *hash table*. Since table elements are arranged (stored) based on their specific hash `index`, how can our algorithm learn to interpret different data types? To help manage our data requirements, we'll establish a new protocol called `Keyable`:

```
//determine hash table compliance

protocol Keyable {

   //conforming types require property
   var keystring: String {get}

   func hashValue<T>(for key: String!, using buckets: Array<T>) -> Int
}
```

`Keyable` can now be applied as a *type constraint* for our hash table class. This will ensure table elements will conform to certain data rules. Seen above, the protocol consists of a single keystring property and a `hashValue()` method. With our protocol in place, let's see how different types conform to these requirements:

```
extension String: Keyable {

    //hash table requirement
    var keystring: String {
        return self
    }
}


class Vertex: Keyable {

    var key: String?
    var neighbors: Array<Edge>
    var visited: Bool = false

    init() {
        self.neighbors = Array<Edge>()
    }

    //hash table requirement
    var keystring: String {
        return self.key!
    }
}
```

The *String* and *Vertex* types comply with the `keystring` property, but what of the `hashValue()` method? Since the requirement to create a *hashValue* is specific to `Keyable` types, this action can be accessed through a **protocol extension**:

PROTOCOL EXTENSION

Exclusive to Swift programming, the process of managing program actions and functionality using protocols.

```
extension Keyable {

    //compute table index

    func hashValue<T>(for key: String!, using buckets: Array<T>) -> Int {

        var remainder: Int = 0
        var divisor: Int = 0

        //trivial case
        guard key != nil else {
            return -1
        }

        for item in key.unicodeScalars {
            divisor += Int(item.value)
        }

        remainder = divisor % buckets.count
        return remainder
    }
}
```

Effective hash tables rely on the performance of their hash algorithms. In this example, `hashValue()` is a straightforward algorithm that employs modular math.

ADDING ELEMENTS

With the components in place, we can define the method for adding items. The following snippet provides some important insights:

```
...

    func insert(_ element: T) -> Result {

        let result: Result

        //compute hash
        let hashIndex = element.hashValue(for: element.keystring, using: buckets)

        if hashIndex != -1 {

            let childToUse = Node<T>()
            childToUse.key = element

            //check existing list
            if  buckets[hashIndex] == nil {
                buckets[hashIndex] = childToUse
                results = Result.Success
            }

           ...
```

With any hash algorithm, the goal is to create enough complexity to eliminate *collisions*. By producing unique indexes, our corresponding table can provide **constant time - O(1)** operations for insertion, modification and lookup. Hash Table collisions occur when different inputs compute to the same `hash`. Based on the relatively straightforward design of our `hashValue()` algorithm, the inputs of *"Albert Einstein"* and *"Andrew Collins"* always produce the same hash result of "8".

The creation of hash algorithms is considered more art than science. As a result, there are many techniques to help reduce the potential of *collisions*. Beyond using modular math to compute a `hash`, we've applied a technique called *separate chaining*. By applying a process similar to building a **linked list**, this will allow multiple table `elements` to share the same `index` value:

```
...
    // table separate chaining process

     else {
         var head = buckets[hashIndex] as Node<T>?

         //append item
         childToUse.next = head
         head = childToUse

         //update chained list
         buckets[hashIndex] = head

         results = Result.Collision
     }
    ...
```

FINDING ELEMENTS

The process for finding elements is similar to the `append()` process. However, since we are looking for specific items, we must ensure the `element` we are seeking can be matched to our input `String`. This conformance can also be managed through `Keyable` protocol:

```
    //test for containing element

    func contains<T: Keyable>(_ element: T) -> Bool  {


        //obtain hash index
        let hashIndex = element.hashValue(for: element.keystring, using: buckets)

        guard hashIndex != -1 else {
            return false
        }

        if buckets[hashIndex] != nil {

            var current = buckets[hashIndex]

            //check chained list for match
            while current != nil {

                if let item: Keyable = current?.key {
                    if item.keystring == element.keystring {
                        return true
                    }
                }
                current = current?.next

             } //end while
        }

        return false
    }
```

What makes this algorithm possible is treating our `Keyable` protocol as a **type**. Since the input parameter as well as the hash table contents both conform to the `Keyable` protocol, table elements can be interpreted as a `Keyable` type. This centralized conformance allows seemingly different objects to be compared equally *(e.g. String vs Vertex).*

CLOSURES

Throughout this series we've used Swift to build various models. As noted, one can build complex structures using common object-oriented (OO) techniques. As such, Swift is also known as a *functional* programming language. This idea extends the traditional OO paradigm with the notion that functions can act as *types*.

The idea that functions can act as types has numerous benefits. As with most functions, the idea of using a `String`, `Int` or `Bool` to pass data is common. Swift extends this idea - allowing *functions* to be used with variables, parameters and return types. In this chapter, we'll review how closures work and will discuss how to use them with algorithms.

THE CONCEPT

To introduce closures, let's review the native `Array` type. As we've seen, Swift arrays have the functionality one would expect. One can add, modify and remove objects. Furthermore, `Arrays` have three additional methods known as `map`, `filter` and `reduce`. Known as *higher-order* functions, these methods accept *functions* as a parameters.

```
//array of numbers
let allNumbers: Array<Int> = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


//determine if a number is even
func isEven(number: Int) -> Bool {
  return number % 2 == 0
}

//returns 2, 4, 6, 8 10
let evenNumbers: Array<Int> = allNumbers.filter(isEven)
```

ANALYSIS

In this example, `filter` accepts a function that returns values divisible by `2`. When reviewing `isEven`, there's nothing to distinguish it from any other function - thus making it reusable for other objects. Finally, when `isEven` is used as a parameter, its signature is removed.

THE SIGNIFICANCE

This sequence could be have been written as a single function. However, the significance is that isEven and its implementation are decoupled. This allows filter to use any function that accepts an Int and returns a Bool. Beyond Swift, the idea of *anonymous* functions can be seen in other languages like Python, Ruby and Java. As someone interested in Swift, you may have seen similar statements written as *closure expressions*. Now that we understand the significance of anonymous functions, let's see how they can be used with custom data structures.

MAPPING LISTS

Previously, we introduced linked lists and built a number of algorithms to manage LLNodes. Similar to an Array, a linked list can also be considered a collection type. As such, the idea of supporting higher-order functions is possible. To illustrate, let's extend our linked list class with a map function.

```
//map list content - higher order function
func map(_ formula: (LLNode<T>) -> T) -> LinkedList<T>! {
    ...
}
```

When working with closures, it can sometimes be challenging to understand their syntax. In our case, map takes a single parameter (formula) which is a function. With its syntax refined, formula accepts a generic LLNode<T> and returns a generic type T. Finally, the map function returns an optional.

```
    //map list content - higher order function
    func map(_ formula: (LLNode<T>) -> T) -> LinkedList<T>! {


        //check for instance
        guard head.key != nil else {
            return nil
        }

        var current: LLNode<T>! = head
        let results: LinkedList<T>! = LinkedList<T>()
        var newKey: T!

        while current != nil {

            //map based on formula
            newKey = formula(current)

            //add non-nil entries
            if newKey != nil {
                results.append(newKey)
            }
            current = current.next
        }

        return results

     }
```

In this example, LinkedList.map mimics the functionality seen with Array.map. As the function iterates through generic LLNodes, each one is passed to formula to be processed. Note the absence of the formula function body. This portion is determined at runtime with a closure expression:

```
//map nodes based on closure expression

func testLinkMapExpression() {

   //helper function - builds a linked list
   let linkedList: LinkedList<Int> = self.buildLinkedList()

   let results = linkedList.map { (node: LLNode<Int>) -> Int in
      return node.key * 2
   }

}
```

CLOSURE EXPRESSIONS

As previously shown, it is not required that functions be executed as closure expressions. However, their concise format makes it the preferred choice.

TRAVERSALS REVISITED

In addition to linked lists, we also introduced the concept of traversing data structures. Using depth-first (DFS) and breadth-first search (BFS), our goal was to discover new nodes while printing their key. However, we can enhance these algorithms with closures. To illustrate, let's refactor our BFS algorithm.

```
//bfs traversal with inout trailing closure
func traverse(_ startingv: Vertex, formula: (_ node: inout Vertex) -> ()) {
  ...
}
```

CLOSURE SYNTAX

When declaring function signatures, nil values can be expressed as a keyword or with "()".

Since our function is designed to work with graphs, traverse accepts a starting vertex and formula. Upon analyzing the formula, we see it accepts a vertex as an inout parameter and returns nil.

```
//bfs traversal with inout closure function

func traverse(_ startingv: Vertex, formula: (_ node: inout Vertex) -> ()) {

    //establish a new queue
    let graphQueue: Queue<Vertex> = Queue<Vertex>()


    //queue a starting vertex
    graphQueue.enQueue(startingv)


    while !graphQueue.isEmpty() {

        //traverse the next queued vertex
        var vitem: Vertex = graphQueue.deQueue() as Vertex!


        //add unvisited vertices to the queue
        for e in vitem.neighbors {
            if e.neighbor.visited == false {
```

next page >>

```
                        print("adding vertex: \(e.neighbor.key!) to queue..")
                        graphQueue.enQueue(e.neighbor)
                    }
                }

                //invoke formula
                formula(&vitem)

            } //end while

            print("graph traversal complete..")
        }
```

Similar to the `map` function, `traverse` invokes its own `formula` as it iterates through discovered `vertices`. However, since the `formula` parameter is passed by *reference*, there's no need to declare a return value. This allows *the closure* to fully encapsulate any changes.

```
    //bfs traversal with closure expression
    func testBFSTraverseExpression() {

        var testGraph: SwiftGraph = SwiftGraph()

        ....

        testGraph.traverse(vertexA) { (node: inout Vertex) -> () in

            node.visited = true
            print("traversed vertex: \(node.key!)..")

        }

    }
```

While our trailing closure updates a single boolean property, a more complex implementation could perform any number of operations, including manipulating objects outside its local scope.

CONTROL STRUCTURES

Ongoing changes to Swift bring interesting changes to control structures. The most notable of these changes are the removal of C-based loops and increment / decrement symbols (eg., ++ / --). If you're in the process of learning Swift, these changes may come as a surprise. In this essay, we'll review loop control structures and will explore alternate iteration techniques with Swift.

C-LOOP BASICS

The C-style loop is a classic control structure found in every C-based language. Also called a for-loop, examples can be expressed in Java, Javascript, Objective-C and C#. Consider the following:

```
var numberList: Array<Int> = [8, 2, 10, 7, 5]

//deprecated C-loop syntax
for var index = 0; index < numberList.endIndex; index++ {
    numberList[index]
}
```

As shown, the `for-loop` consists of three parts. The first part sets the variable assignment while the second statement provides the terminating condition. The final statement updates the `loop` variable - assuming the terminating condition hasn't been met. When applied, this classic / cross-platform technique has been used to build countless user interfaces, algorithms and backend systems.

THE WHILE LOOP

Even though the C-loop fades away in Swift, there are other techniques that meet and, in some cases, exceed this classic tool. First, let's consider the while-loop:

```
var numberList: Array<Int> = [8, 2, 10, 7, 5]
var index: Int = 0

//prints 8, 2, 10, 7, 5
while index < numberList.endIndex {
    numberList[index]
    index += 1
}
```

What's nice about the while-loop is that it is a widely adopted format. If you used this technique in a technical interview, others would have little problem following your logic. With my project, I often use the `while-loop` with other Swift-specific syntax.

FAST ENUMERATION

Beyond basic loops, let's consider the common task of iterating through a collection. The idea of fast enumeration is common but the syntax varies, dependent on programming language. Here are some Swift-specific examples:

```
var numberList: Array<Int> = [8, 2, 10, 7, 5]


//for loop with half-open operator
for index in 0..<numberList.endIndex {
    print(index)
    numberList[index]
}


//fast enumeration
for item in numberList {
    print(item)
}


//return tuple combination - index & value
for (index, value) in numberList.enumerate() {
    print("the index \(index) contains value \(value)..")
}
```

At first glance, these samples look concise. Using the half-open operator, a `loop` variable can be initialized to zero or some other value. However, what's missing is being able to control the `loop` iteration sequence. This would permit cycling through a collection in reverse or skipping certain indices. With Swift, the built-in reverse function can be applied. As shown, reverse can be used to `reverse` a character `set` or `collection`:

```
//reversed characters
var someArray: String = "Swift"
var result = String(someArray.characters.reverse())
print(result)

//reversed collection
var testArray: Array<Int> = [8, 10, 2, 9, 7, 5]
var listResult = Array(testArray.reverse())
print(listResult)
```

TAKING STRIDES

For finer control, the `Strideable` protocol can be applied. As the name implies, stride can be used to produce specific results:

```
var numberList: Array<Int> = [8, 2, 10, 7, 5]

//forward stride enumeration
for index in stride(from: 0, through: numberList.endIndex - 1, by: 1) {
    print(numberList[index])
}


//forward stride enumeration - every two
for index in stride(from: 0, through: numberList.endIndex - 1, by: 2) {
    print(numberList[index])
}


//reverse stride enumeration
for index in stride(from: numberList.endIndex - 1, through: 0, by: -1) {
    print(numberList[index])
}
```

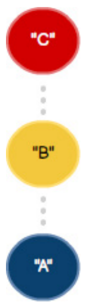
While more verbose, this syntax provides good flexibility. When applied, `stride` allows one to skip items or iterate through a `collection` in reverse with a consistent format.

RECURSION

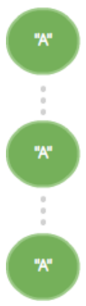
In this series, we've examined object-oriented and functional programming techniques used to create algorithms and data structures. However another method to consider is recursion. In a nutshell, recursion is a specific technique that provides an indeterminate set of references to one's `self`. In this essay, we'll review the concept of recursion and will demonstrate how to write recursive code with Swift.

HOW IT WORKS

Recursion is best understood when compared to traditional object-oriented programming. With most solutions, code is comprised of different `objects` that are often interchangeable. As such, programmers reference different `objects` (i.e., classes) to build a model. The recursive technique, however, builds a model with an object that refers to itself:



Traditional approach - objects reference other objects



Recursive approach - an objects refer to itself

CLASS VS. STRUCT

They are a few ways recursion can be expressed in Swift. If you are accustomed to C-based languages, an interesting, an interesting differentiator is how Swift handles `types`. To date, many Swift objects are considered first-class citizens including `structs` and `enums`. As such, most operations normally reserved for a `class` can be replaced with a `struct`:

```
//simple struct example - methods & properties

struct Car {

    var color: String
    var make: String

    init(color: String, make: String) {
        self.color = color
        self.make = make
    }

    func buildCar() {
        print(“a \(color) \(make) has been built..”)
    }
 }
```

Swift `structs` are lightweight components that act as value types. In contrast, `classes` in Swift are *reference types*. Because `struct` instances are *copied* (not referenced), they can’t be used to build recursive data models:

```
//simple usable class
class LLNode<T> {
   var key: T?
   var previous: LLNode?
   var next: LLNode?
}


//gives compilation error
struct Tree<T> {
   var key: T?
   ~~var left: Tree?~~
   ~~var right: Tree?~~
}
```

FUNCTIONS

Recursion can also be seen with the popular **Fibonacci** sequence, the idea behind building a numerical sequence by adding the two preceding numbers. Let’s compare a traditional and recursive technique:

```
//fibonacci sequence - traditional approach

extension Int {

   func fibNormal() -> Array<Int>! {

           //check trivial condition
           guard self > 2 else {
               return nil
           }


           //initialize the sequence
           var sequence: Array<Int> = [0, 1]
           var i: Int = sequence.count


           while i != self {
               let results: Int = sequence[i - 1] + sequence[i - 2]
               sequence.append(results)
               i += 1
           }

           return sequence
     }
}

//execute sequence
let positions: Int = 4
let results: Array<Int>! = positions.fibNormal()


//fibonacci sequence - recursive approach

extension Int {

     mutating func fibRecursive(_ sequence: Array<Int> = [0, 1]) -> Array<Int>! {

        var final = Array<Int>()

        //mutated copy
        var output = sequence


        //check trivial condition
        guard self > 2 else {
            return nil
        }


        let i: Int = output.count


        //set base condition
        if i == self {
            return output
        }

        let results: Int = output[i - 1] + output[i - 2]
        output.append(results)

        //set iteration
        final = self.fibRecursive(output)

        return final
    }
}

//execute sequence
var positions: Int = 4
let results: Array<Int>! = positions.fibRecursive()
```

Note the function differences. At first glance, we see `fibRecursive` employs a **base-case** and a call to one's `self`. To contrast, `fibNormal` maintains control logic in a `while-loop`. As shown, recursive logic often leads to increased complexity, as an entire class, function or method is often used as a **control structure**.

CLASSES

While recursion tends to add complexity, consider the algorithm for depth-first search. As discussed in a previous essay, this code works in conjunction with the call stack to traverse a *binary search tree*. While it would be possible to refactor the algorithm to support an iterative technique, recursion provides an effective solution:

```
    //recursive depth-first search

    func traverse() {

        guard self.key != nil else {
            return
        }

        //process left side
        if self.left != nil {
            left?.traverse()
        }

        print("...node \(self.key!) visited..")

        //process the right side
        if self.right != nil {
            right?.traverse()
        }
    }
```

ENUMS

Like `structs`, many Swift objects act as first-class citizens, including *enumerations*. Like `enums` used in other languages, the enum type `Algorithm` is used to model behavior. The model will help manage four specific cases. The `enum` is said to be *recursive* because it contains associated values that are also of type `Algorithm`:

```
//recursive enumeration

indirect enum Algorithm<T> {

    case Empty
    case Elements(Array<T>)
    case InsertionSort(Algorithm<T>)
    case BubbleSort(Algorithm<T>)
    case SelectionSort(Algorithm<T>)
}
```

One goal of `enums` is to enhance code readability. In our case, we can use `Algorithm` to provide type-safety support for an implementation and can easily extend it to support additional scenarios:

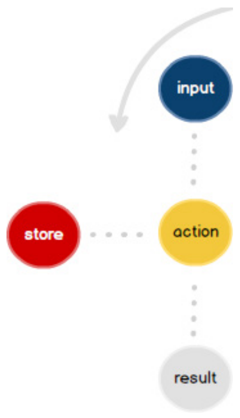
```
//build an algorithm model

let numberList = Algorithm.Elements([8, 2, 10, 9, 7, 5])
let model = Algorithm.InsertionSort(numberList)

...
```

DYNAMIC PROGRAMMING

In our exploration of algorithms, we've applied many techniques to produce results. Some concepts have used iOS-specific patterns while others have been more generalized. Although it hasn't been explicitly mentioned, some of our solutions have used a particular programming style called dynamic programming. While straightforward in theory, its application can sometimes be nuanced. When applied correctly, dynamic programming can have a powerful effect on how you to write code. In this essay, we'll introduce the concept and implementation of dynamic programming.

SAVE FOR LATER

If you've purchased something through Amazon.com, you'll be familiar with the site term - *"Save For Later"*. As the phrase implies, shoppers are provided the option to add items to their cart or save them to a *"Wish List"* for later viewing. When writing algorithms, we often face a similar choice of completing actions (performing computations) as data is being interpreted or storing the results for later use. Examples include retrieving **JSON** data from a **RESTful** service or using the *Core Data Framework*:



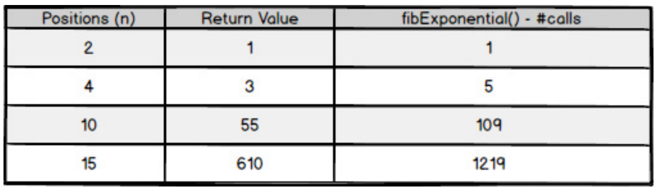
In iOS, design patterns can help us time and coordinate how data is processed. Specific techniques include multi-threaded operations (e.g. Grand Central Dispatch), Notifications and Delegation. Dynamic programming (DP) on the other hand, isn't necessarily a single coding technique, but rather *how to think* about actions (e.g. sub problems) that occur as a function operates. The resulting DP solution could *differ* depending on the problem. In its simplest form, dynamic programming relies on *data storage and reuse* to increase algorithm efficiency. The process of data reuse is also called **memoization** and can take many forms. As we'll see, this style of programming provides numerous benefits.

FIBONACCI REVISITED

In the essay on **Recursion**, we compared building the classic sequence of `Array` values using both iterative and recursive techniques. As discussed, these algorithms were designed to produce an `Array` sequence, not to calculate a particular result. Taking this into account, we can create a new version of *Fibonacci* to return a single `Int` value:

```
func fibRecursive(n: Int) -> Int {
    if n == 0 {
        return 0
    }

    if n <= 2 {
        return 1
    }

    return fibRecursive(n: n-1) + fibRecursive(n: n-2)
}
```

At first glance, it appears this seemingly small function would also be efficient. However, upon further analysis, we see numerous recursive calls must be made for it to calculate any result. As shown below, since `fibRecursive` cannot store previously calculated values, its recursive calls increase **exponentially**:



Positions (n)	Return Value	fibExponential() - #calls
2	1	1
4	3	5
10	55	109
15	610	1219

FIBONACCI MEMOIZED

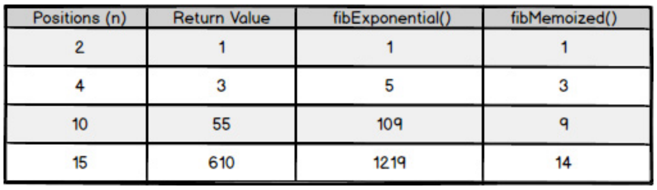
Let's try a different technique. Designed as a nested Swift function, `fibMemoized` captures the `Array` return value from its `fibSequence` sub-function to calculate a final value:

```
extension Int {

    //memoized version
    mutating func fibMemoized() -> Int {

        //builds array sequence
        func fibSequence(_ sequence: Array<Int> = [0, 1]) -> Array<Int> {

            var final = Array<Int>()

            //mutated copy
            var output = sequence

            let i: Int = output.count

            //set base condition - linear time O(n)
```

next page >>

```
            if i == self {
                return output
            }

            let results: Int = output[i - 1] + output[i - 2]
            output.append(results)

            //set iteration
            final = fibSequence(output)
            return final
        }


        //calculate final product - constant time O(1)
        let results = fibSequence()
        let answer: Int = results[results.endIndex - 1] + results[results.endIndex
- 2]
        return answer

    }
}
```

Even though fibSquence includes a recursive sequence, its base case is determined by the number of requested Array positions (n). In performance terms, we say fibSequence runs in **linear time** or O(n). This performance improvement is achieved by memoizing the Array sequence needed to calculate the final product. As a result, each sequence permutation is computed **once**. The benefit of this technique is seen when comparing the two algorithms, shown below:



Positions (n)	Return Value	fibExponential()	fibMemoized()
2	1	1	1
4	3	5	3
10	55	109	9
15	610	1219	14

Shortest Paths
Code memoization can also improve a program's efficiency to the point of making seemingly difficult or nearly unsolvable questions answerable. An example of this can be seen with Dijkstra's Algorithm and **Shortest Paths**. To review, we created a unique data structure named Path with the goal of storing specific traversal metadata:

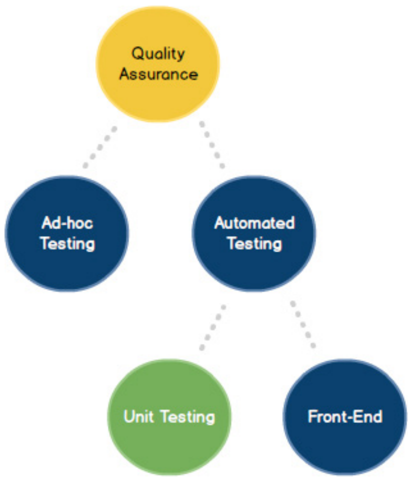
```
//the path class maintains objects that comprise the “frontier”
class Path {

    var total: Int
    var destination: Vertex
    var previous: Path?

    //object initialization
    init() {
        destination = Vertex()
        total = 0
    }
}
```

What makes Path useful is its ability to store data on nodes previously visited. Similar to our revised **Fibonacci** algorithm, `Path` stores the cumulative `Edge` weights all traversed vertices `(total)` as well as a complete history of each visited `Vertex`. Used effectively, this allows the programmer to answer questions such as the complexity of navigating to a particular destination `Vertex`, if the traversal was indeed successful (in finding the destination), as well as the list of `nodes` visited throughout. Depending on the `Graph` size and complexity, not having this information available could mean having the algorithm take so long to (re)compute data that it becomes **too slow** to be effective, or not being able to solve vital questions due to insufficient data.

UNIT TESTING

Software construction is a complex process. Beyond mastering the various tools and programming languages, there's also understanding the various software development roles. As most of us know, great projects don't just involve coding. Project managers are also responsible for requirements gathering, prototyping and testing.

The process of validating code is often referred to as **Quality Assurance.** However, the phrase is somewhat misleading. Instead of assuring code quality, the real objective is to *measure* it - often as the project evolves. Also known as *test-driven development* (e.g. TDD), this subtle difference can be seen when examining the quality function roles. These include the following:



WHY UNIT TESTS?

As shown above, quality measurement is often grouped into automated and manual activities. As someone interested in algorithms, additional code known as **unit tests** work to test your primary software project. Unit tests are usually written by developers and are the focus in a TDD environment. Unit tests are essential to measuring the code quality of the *Swift Algorithms* project, for example. Using the *iOS XCTest Framework,* let's review how unit testing works in Swift.

UNIT TEST

Code designed to test other code or software components that often lack an end-user interface.

WORKING WITH XCTEST

If you've become familiar with the Swift concepts introduced in earlier chapters, learning how to write unit tests should be straightforward. As discussed, unit tests work to exercise code that lacks an end user interface. To illustrate, let's review test cases that exercise a `Stack` data structure.

While this essay doesn't review the particulars of the Xcode Integrated Development Environment (e.g. IDE), it should be noted that tests can be executed by `function`, `class` or target. Using Xcode, unit tests are established by adding a new Test Target to your primary code project. Once configured, unit tests can be executed from the IDE or command line..

TEST RULES

With our `Stack` class established we can begin to write unit tests. While not required, it's considered best practice if the unit test file closely match the naming convention of our implementation file(s). In our case, we'll use `StackTest.swift`. To gain access to primary `Stack` methods and properties, the file will also include a `testable import` statement:

```
import XCTest

@testable import SwiftStructures

class StackTest: XCTestCase {

    //called before each test method in the class
    override func setUp() {
        super.setUp()
    }

    //example of a functional test case
    func testExample() {

    }
}

...
```

Like other unit test frameworks, `XCTest` works by integrating Swift language features with specific test-related functions. The essential methods are grouped together as assertions. When creating tests, the compiler will recognize unit tests with functions prefixed with the `test` keyword.

Unlike regular Swift functions, unit tests are intentionally designed as self-contained units of logic. As a result, test methods don't accept arguments and don't return values. Alternatively, test data can be managed through helper methods and initialization / tear down sequences. Consider the following:

```
class StackTest: XCTestCase {

    var numberList: Array<Int>!

    override func setUp() {
        super.setUp()
        numberList = [8, 2, 10, 9, 7, 5]
    }

    func testPushStack() {

        let myStack = Stack<Int>()
        XCTAssertEqual(myStack.count == 0, “test failed: count not initialized..”)

        //build stack
        for s in numberList {
            myStack.push(s)
            print(“item: \(s) added..”)
        }
```

```
     XCTAssertTrue(myStack.count == numberList.count, “stack count does not 
match..”)

    }
```

TEST PLANNING

As discussed, the main operation of a Stack data structure is adding and removing items. Instead of writing a single function to test all operations, our test plan isolates each major Stack operation with its own test. As seen with testPushStack, the assertion XCTAssertTrue checks for the correct initialization of the Stack.count variable. The next step involves exercising Stack.push by iterating through the array of numberList items. To verify that each test employs the same data, numberList is populated using the XCTestCase class setup method.

With the unit test for adding Stack items implemented, we can write the next test for removing items:

```
func testPopStack() {

        //build stack - helper function
        let myStack: Stack<Int> = self.buildStack()

        if myStack.count == 0 {
           XCTFail(“test failed: no stack items available..”)
        }

        //remove stack items..
        while myStack.count > 0 {
            print(“stack count: \(myStack.count)”)
            myStack.pop()
        }

        XCTAssertTrue(myStack.isEmpty(), “test failed: stack structured not emp-
tied..”)

    }
```

With the test for adding items implemented, testPopStack calls a simple helper function to create a fresh Stack data type. Once validated, the test continues by iterating through the Stack items (removing a single item during each pass). The final assertion validates the proper executation of Stack.isEmpty to ensure no items still exist.

APPENDIX

This appendix contains additional essays to help you pass your next iOS technical interview. If you are preparing for your first interview or are seeking to move into advanced role, these essays provide tips on what to expect and how to prepare. For more detailed help, I also provide personalized **1-1 coaching**. To learn more visit **shop.waynewbishop.com**.

HOW TO PREPARE FOR A TECHNICAL INTERVIEW

In the fast-paced environment of Swift and *iOS development*, chances are you could be preparing for a technical interview. Many a blog post has been written about surviving the gauntlet of the dreaded whiteboard. Having had the chance of hiring technical resources as well as being an interviewee, I've easily passed through some experiences and have been totally unprepared in others. This essay will provide some tips to hopefully make your next *iOS technical interview* a positive experience.

THE FEAR

If you've been asked to interview, you've probably already met with one or a few members of the hiring team. As with other standard interviews, the goal during these early rounds is determine your *"personality fit"*. Are you likable, reasonable and provide the appearance of working well with others? The real anxiety begins when your coding skills are assessed. The fear? The potential of being asked something you haven't heard of, didn't study enough or have no experience in. This is especially true when it comes to algorithms and data structures.

THE LANDSCAPE

To start, let's break down the iOS technical interview into its individual components. Building any workable app requires at least 3 areas of expertise. These include an understanding of the programming languages *(e.g. Swift, Objective-C)*, frameworks *(e.g. iOS SDK)*, and tools *(e.g. Xcode).* In addition, there are many subjects tangential to core development that are considered best practice. Many of these are outlined in my *iOS Interview Worksheet*. Beyond basic algorithms, additional topics include unit testing, cocoapods, the build process, core data, push notifications, cloud based services, etc..

THE GOAL

Like with anything new, the best way to tackle a subject is to determine how it can be applied towards a goal. Many readers of this book comment how it helps them reinterpret problems they've been stuck on. Working through tough problems is same objective of a technical interview. As a result, anticipate (at least) three to four questions that will review the different aspects needed to build an app. The depth and complexity of each problem will vary based on the organization. A good rule of thumb? Plan at least 20 minutes to 1 hour to work through each question.

If you're not actively working on a project, stretch your skills by contributing on **Github**. With thousands of projects available through the open-source network, it's free and easy to brush up on valuable skill sets including Swift and source code management. Since Swift is also open-source, be sure to understand the Swift proposal / review process, API design guidelines and impact of major events like WWDC.

THE TIMING

Nervous? Start with the basics and download the interviewing company's app. This simple step can pinpoint your interview efforts by helping you focus on specific iOS frameworks (e.g, MapKit, CoreLocation, CoreAnimation) beforehand. If the company doesn't yet have an app, research their vertical market and / or competing services. At the least, doing your homework will arm you with (intelligent) questions that can be used throughout your experience.

As the interviewee, a (hidden) advantage you also have is time. It may come as a surprise, but many top-level companies don't hire based on specific salary requirements or timeframe. As a result, they are willing to grant candidates weeks or even months to prepare for a major technical interview. Use that time to your advantage.

THE PRESENTATION

Regardless of the solution, much of it boils down to being able to present information on a screen. As a result, understand the UIView lifecycle and supporting technologies such as AutoLayout and Storyboards. Knowing the basics of *Model-View-Controller*, *IBAction / IBOutlet*, *UIViewTableViews*, *Recognizers* and *GCD* (Grand Central Dispatch) will also demonstrate your proficiency with managing application state and presenting data.

THE WHITEBOARD

Without a doubt, the biggest drawback to interviewing is having to endure the process without the aid of a computer. As developers, we know a good portion of our work is dependent on awesome tools like auto-complete, *StackOverflow* and *Google*. The workaround? Develop experience explaining concepts on paper.

As you work through the details, don't worry (initially) about memorizing the latest Swift syntax. Smart interviewers know this information can be easily obtained online. Instead, focus on what models could apply given a particular scenario. For example, learn when to apply generic models like *hash tables* and / or *graphs versus* iOS-specific patterns such as *key-value observation* or *delegation*. Finally, know how to express your ideas through **Big O Notation** as well as **UML** (Unified Modeling Language), as doing so will remove any consideration to programming language or coding syntax.

THE STEPS

How can we use these points to actually solve coding questions? Let's begin with this common framework:

- *Ask clarifying questions*
- *Create a conceptual diagram*
- *Express a brute force solution in pseudocode*
- *Refine your solution with workable code*
- *Check for errors or omissions*

Much of the anxiety created during interviews is when candidates fail to immediately see a codeable answer. Before diving in, be sure to take a step back and make sure you fully understand the question. It may seem counterintuitive, but many times interviewers know it may not be possible for candidates to write a coded solution in the time provided.

As a result, people may be more interested in seeing how you handle yourself when faced with a difficult assignment. Can you methodically deconstruct a problem into its individual components and test your hypothesis before attempting to write production-level code? Interestingly, the steps for answering technical interview questions can also be used when writing entire apps.

HOW TO ASK QUESTIONS IN A TECHNICAL INTERVIEW

In the previous topic, I provided an overview of items to consider when preparing for a technical interview. Portions of this content were expanded into a webinar, including a basic framework for answering tough questions when going to the whiteboard. Even though we specialize in *iOS development*, the process is general enough so that it can be applied to any programming language.

- *Ask clarifying questions*
- *Create a conceptual diagram*
- *Express a brute force solution in pseudocode*
- *Refine your solution with workable code*
- *Check for errors or omissions*

Following a set of predefined steps becomes particularly important when coding under time constraints or are dealing with an unfamiliar subject. Let's see how to use this process to answer the following question:

*How would you design a `Stack` class which, in addition to a `push()` and `pop()` methods, also has a property `count`, which returns the number of elements? Push, pop and count should all operate in **constant time - O(1)**.*

This problem tests the candidates knowledge of **Big O Nation** as well as algorithms & data structures. However if you didn't know these subjects, you could still discover a solution by asking a few clarifying questions:

WHAT IS A STACK? IS THERE AN IOS EQUIVALENT?

As iOS developers we're accustom to using Swift and the iOS SDK to build user interface elements, processes and apps. This question provides value because the candidate (correctly) assumes a `Stack` isn't something normally used in development. From the candidates perspective, the answer to this question could connect the everyday activity of managing `UITableViews` with a `UINavigationController`.

WHAT'S O(1) TIME?

This answer will reveal how one should be thinking about algorithm's performance. By understanding the importance of `O(1)` (e.g. constant time) they'll see there are significant advantages to processes that perform at the same speed - regardless of their input size. Even though asking this question during an interview would be the best course of action, studying up *Big O Notation* beforehand would provide many added benefits.

THINKING ALOUD

This one we've all heard of, but in practice, it's difficult advice to follow. As we work through the problem to eventually get to a solution, we need to let our interviewer know what we're thinking - even if our ideas appear silly or half-baked. In most cases, interviewers want you to succeed so they'll be looking for opportunities to steer you in the right direction. Without any verbal feedback, they can't help. As we know, this is often when things get tense and awkward.

REFINING YOUR SOLUTION

When writing code during an interview, it's best practice to code a basic solution *"that just works,"* then continue to refine its syntax and functionality. In performance terms, this often translates to building a workable solution at `O(n)` or even $O(n^2)$. This 2-step process not only provides an opportunity to gather your thoughts, but also lets the interviewer know you're on the right track.

Finally, as you refactor your code to a final solution ask yourself, how would this algorithm perform if it had to process **1M** rows of data? Reframing the question often allows one to dissect parts of your solution that look solid, but could be improved even further.

FURTHER READING

BIG O NOTATION

- » Asymptotic Analysis
- » Big O Notation
- » Binary Search
- » Graphing Logarithms

SORTING

- » Binary Numbers
- » Hexadecimal Numbers
- » Invariant (Computer Science)
- » Insertion Sort
- » Bubble Sort
- » Quick Sort
- » Merge Sort
- » Selection Sort

TRIES

- » Trie Data Structure

STACKS & QUEUES

- » Stacks
- » Queues
- » Call Stack

GRAPHS

- » Graph Theory
- » Google Maps
- » Adjacency Lists

SHORTEST PATHS

- » Greedy Approach
- » Traveling Salesman
- » Edsger Dijkstra
- » Dijkstra's Algorithm

HEAPS

- » Binary Heaps
- » Heapsort
- » Base Case

TRAVERSALS

- » Traversal
- » Depth-First Search
- » Breadth-First Search

HASH TABLES

- » Hash Table
- » Hash Algorithm
- » Modular Math
- » MD5 Algorithm

CLOSURES

- » Anonymous Functions
- » Closures

THE PROJECT

Started in 2014, this project was originally based on a series of code snippets written in Objective-C. The goal was to practice common programming techniques for technical interviews. At the time, I had considered expanding the project to other languages such as Java and Python. However, when Swift was introduced, it provided the perfect blend of concise syntax and modern language features.

Have feedback or an idea for this project? Contact Wayne at waynewbishop.com/contact.

The code and selected essays are open-source. As a result, they are available online for others to review, critique and build upon. The openness of the Internet has allowed me to solicit feedback, reconsider ideas and, ultimately, create a product I hope will help and inspire others. As the Swift language evolves, it will be exciting to see what new platforms and projects it will impact.

ACKNOWLEDGMENTS

Thanks to the many students, professionals and educators that made this effort possible. Special thanks to these individuals for their expertise, encouragement and support.

- *Karen Bishop*
- *Lynne Wendlandt*
- *Richard Sbresny*
- *Mike Wesolowski*
- *Christina Moulton*
- *Simon Gladman*
- *Simon Ng*
- *Mark McLaren*

THE AUTHOR

Wayne Bishop resides in Seattle, Washington, and is an independent iOS Developer, Educator and Project Manager. He has developed apps for education, social media and the visual arts. His strengths include planning solutions, coding applications, technical writing and communicating with stakeholders. He likes to run, bike and spend time with family. Feel free to contact Wayne through his website or on Twitter at **@waynewbishop**.