FIRST EDITION

WAYNE BISHOP

SWIFT ALGORITHMS & DATA STRUCTURES

A practical guide to concepts, theory and code

FOREWORD BY NATASHA MURASHEV

SWIFT ALGORITHMS & DATA STRUCTURES

BY WAYNE BISHOP

FOR MY FAMILY

Copyright © 2015 by Wayne W Bishop.
All rights reserved. This book or any portion thereof
may not be reproduced or used in any manner whatsoever
without the express written permission of the publisher
except for the use of brief quotations in a book review.

Printed in the United States of America

First Edition, 2015

www.waynewbishop.com

TABLE OF CONTENTS

SOURCE CODE & GITHUB

This series is available through Github as an adjunct to the book. In the spirit of collaborative effort, I welcome ongoing feedback and contribution from others. The source code and unit tests are optimized for Swift 1.2 or later and can be obtained from the following repository:

```
http://github.com/waynewbishop/SwiftStructures
```

USAGE

Individuals are welcome to use code for commercial and open-source projects. As a courtesy, please provide attribution to "*waynewbishop.com*". For more information, review the complete Github license agreement.

FOREWORD

I was sitting in the third row on stage right during the keynote of Apple's annual Worldwide Developers Conference in 2014 (WWDC). There were a lot of exciting announcements. Then came the last announcement. Apple is releasing a brand new programming language - Swift. *"It's like Objective-C, without the C,"* said Craig Federighi, Apple's senior vice president of Software Engineering.

None of us expected that. In fact, it was a surprise to many of Apple's own software engineers. So, at the moment when Swift was announced, there was a tiny pause before awkward, indecisive clapping ensued.

At first glance, Swift looked very familiar and easy to any programmer who's ever programmed in another language besides Objective-C. On social media, there were memes circulating that all JavaScript developers were now iOS developers. Coming from a Ruby background, it looked like Ruby to me. Python developers saw Python in it. Haskell developers said it was most like Haskell - although, at the time, I didn't know anything about Haskell. Everyone immediately felt familiar with Swift, even though we all just learned about it that day.

Swift is a language on a mission. One of its goals is to help us write safe and robust code. By adding specific language features such as optionals and generics, it forces us, as programmers, to handle error-prone scenarios that we often forget to address.

This combination of familiarity and fun has made Swift the most loved language within only the first few months of its release, according to StackOverflow's 2015 Developer Survey.

This is just the beginning for Swift. As I sat in the second row on stage right during the keynote at WWDC 2015, I was witness to the excitement and energy that swept the crowd as it was announced that Swift will be open-sourced. This time the clapping was immediate - without a moment of hesitation.

Natasha Murashev
-NatashaTheRobot

INTRODUCTION

This series provides an introduction to commonly used data structures and algorithms written in a new iOS development language called Swift. While details of many algorithms exist on Wikipedia, these implementations are often written as pseudocode or are expressed in C or C++. With Swift now officially released, 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 an app that finds information quickly could mean the difference between 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 scientists 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
let 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;

```
func linearSearch(key: Int) {

     //check all possible values
     for number in numberlist {
             if number == key {
                println("value at \(key) found..")
                break
             }
     }
}
```

While this approach achieves our goal, each item in the array 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.

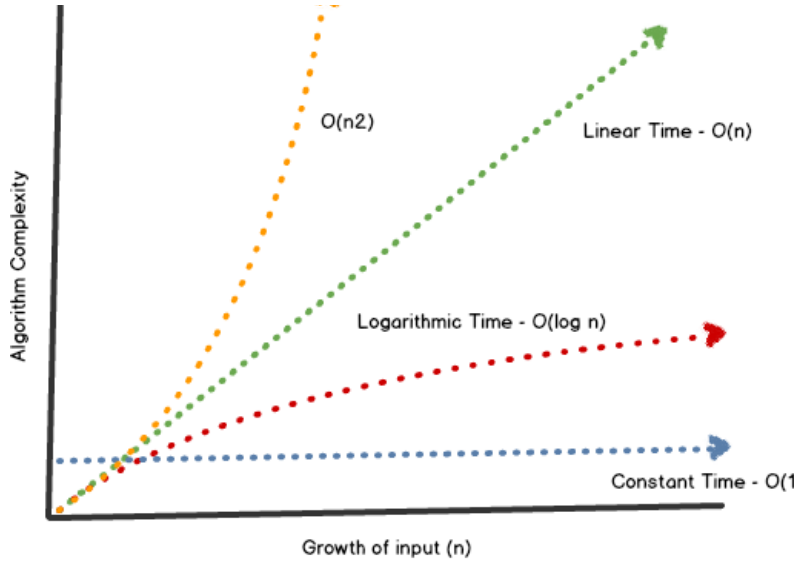
```
func binarySearch(key: Int, imin: Int, imax: Int) {

    var midIndex : Double = round(Double((imin + imax) / 2))
    var midNumber = numberList[Int(midIndex)]

    //reduce the range
    if  midNumber > key  {
        binarySearch(key, imin, Int(midIndex) - 1)
    }

    //increase the range
    else if (midNumber < key ) {
        binarySearch(key, Int(midIndex) + 1, imax)
    }

    else {
        println("value \(key) found..")
    }

}
```

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 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.



FURTHER READING

» Asymptotic Analysis - http://en.wikipedia.org/wiki/Asymptotic_analysis
» Big O Notation - http://en.wikipedia.org/wiki/Big_O_notation
» Binary Search - http://en.wikipedia.org/wiki/Binary_search_algorithm
» Graphing Logarithms - http://www.khanacademy.org

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. To start, 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 and how things should be organized. 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

One of the more basic algorithms in computer science, insertion sort works by evaluating a constant set of numbers with a secondary set of changing numbers. The outer loop acts as the invariant, assuring all array values are checked. The inner loop acts as a secondary engine, reviewing which numbers get compared. Completed enough times, this process eventually sorts all items in the list.

```
func insertionSort(var numberList: Array<Int>) -> Array<Int> {

        var y, key : Int

        for x in 0..<numberList.count {

            //obtain a key to be evaluated
            key = numberList[x]

            //iterate backwards through the sorted portion
            for (y = x; y > -1; y--) {

                    if (key < numberList[y]) {

                    //remove item from original position
                    numberList.removeAtIndex(y + 1)
```

next page >>

```
                    //insert the number at the key position
                    numberList.insert(key, atIndex: y)
                }

            }

        } //end for


        return numberList


    } //end function
```

BUBBLE SORT

Another common sorting technique is the bubble sort. Like insertion sort, this 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 items in the list.

```
func bubbleSort(var numberList: Array<Int>) ->Array<Int> {

        var x, y, z, passes, key: Int

        //track iterations
        for x in 0..<numberList.count {

            passes = (numberList.count - 1) - x

            for y in 0..<passes {
                key = numberList[y]

                println("comparing \(key) and \(numberList[y + 1])")

                //compare and rank values
                if (key > numberList[y + 1]) {

                    z = numberList[y + 1]
                    numberList[y + 1] = key
                    numberList[y] = z
                }
            }
        }

        return numberList

    } //end function
```

EFFICIENCY

Besides insertion sort and bubble sort, there are many other sorting algorithms. Because both insertion and bubble sort combine a variant and invariant, their average performance is `n x n` or $O(n^2)$. Other techniques (like Merge Sort) apply different methods and can improve average performance to `O(n log n)`.

See additional sorting algorithms for Quick Sort and Merge Sort on Github.

FURTHER READING

» Binary Numbers - http://en.wikipedia.org/wiki/Binary_number
» Hexadecimal Numbers - http://en.wikipedia.org/wiki/Hexadecimal
» Invariant - http://en.wikipedia.org/wiki/Invariant_(computer_science)
» Insertion Sort - http://en.wikipedia.org/wiki/Insertion_sort
» Bubble Sort - http://en.wikipedia.org/wiki/Bubble_sort
» Quick Sort - http://en.wikipedia.org/wiki/Quicksort
» Merge Sort - http://en.wikipedia.org/wiki/Merge_sort
» Selection Sort - http://en.wikipedia.org/wiki/Selection_sort

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 objects like tries, queues and graphs.

```
//generic doubly linked list structure

class LLNode<T> {

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

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 implicit unwrapped optional (e.g., `current`) to iterate through linked list items.

```
//print all keys for the class

func printAllKeys() {

    var current: LLNode! = head

    //assign the next instance
    while (current != nil) {
        println("link item is: \(current.key)")
        current = current.next
    }
}
```

ADDING LINKS

Here's a function that builds a doubly linked list. The method `addLink` creates a new item and appends 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> {

var head: LLNode<T> = LLNode<T>()

//append a new item
func addLink(key: T) {

    if (head.key == nil) {
        head.key = key
        return
    }

    //establish loop variables
    var current: LLNode? =  head

    while (current != nil) {

        if (current?.next == nil) {

            var childToUse: LLNode = LLNode<T>()

            childToUse.key = key
            childToUse.previous = current
```

```
                current!.next = childToUse
                break
            }

            else {
                current = current?.next
            }

        } //end while
    }
}
```

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 removeLinkAtIndex(index: Int) {

        //check for nil conditions
        if ((index < 0) || (index > (self.count - 1)) || (head.key == nil)) {
            println("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!
            return
        }


        //iterate through the remaining items
        while (current != nil) {

            if (listIndex == index) {

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

next page >>

```
            //update the assignments
            trailer = current

            current = current?.next
            listIndex++

        } //end while

    } //end function
```

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 a linked list instance to use a dot notation. (e.g., `someList.count`).

```
    //the number of items

    var count: Int {

            if head.key == nil {
                return 0
            }

            else  {

                var current: LLNode = head
                var x: Int = 1

                //cycle through the list of items
                while current.next != nil {
                    current = current.next!
                    x++
                }

                return x

            }
    }
```

EFFICIENCY

Linked lists typically provide `O(n)` for storage and lookup. As we'll see, linked lists are often used with other structures to create new models. Algorithms with an efficiency of `O(n)` are said to run in *linear time*.

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 with Swift called *generics* and will explore how this allows data structures to be more expressive 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 app that lists a group of students. The data could be easily organized with a linked list and represented in the following manner:

```
//student linked list 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 an object that may include specific properties such as name, class 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.

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 resuable:

```
//refactored linked list structure

class LLNode<T> {

   var key: T?
   var next: LLNode<T>?

 }
```

With this revised structure we see several changes. The class name `StudentNode` has been changed to something more general (e.g., `LLNode`). The syntax `<T>` seen after the classname 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, `LLNode` can now manage lists of `Students`, `Teachers`, or any other type we decide.

```
//a new list of students
var studentList = LLNode<Student>()

//a new list of teachers
var teacherList = LLNode<Teacher>()
```

Here's an expanded view of a linked list implementation using generics. The method `addLinkAtIndex` adds generic `LLNodes` at a specified index. Since portions of our code evaluate generic types, the type constraint `Equatable` is appended to the generic `T` placeholder. Native to Swift, `Equatable` is a simple protocol that ensures various object types conform to the equatable specification.

```
    //insert at specific index

    func addLinkAtIndex(key: T, index: Int) {

        //check for nil conditions
        if ((index < 0) || (index > (self.count - 1))) {
            println("link index does not exist..")
        }


        //establish the head node
        if (head.key == nil) {
            head.key = key
            return
        }

        //establish the trailer, current and new items
        var current: LLNode<T>? = head
        var trailer: LLNode<T>?
        var listIndex: Int = 0


        //iterate through the list to find the insertion point
        while (current != nil) {

            if (index == listIndex) {

                var childToUse: LLNode = LLNode<T>()

                //create the new node
                childToUse.key = key


                //connect the node in front of the current node
                childToUse.next = current
                childToUse.previous = trailer


                //use optional binding when using the trailer
                if let linktrailer = trailer {
                    linktrailer.next = childToUse
                    childToUse.previous = linktrailer
                }

                //point new node to the current / previous
                current!.previous = childToUse

                //replace the head node if required
                if (index == 0) {
                    head = childToUse
                }


                break

            } //end if
```

next page >>

```
            //iterate through to the next item
            trailer = current
            current = current?.next
            listIndex += 1

        } //end while

    }
```

BINARY SEARCH TREES

A binary search tree is a data structure that stores information in a logical hierarchy. As demonstrated with Big O Notation, algorithms provide the best results with sorted data. Binary Search Trees extend this idea by using specific rules. A binary search tree (BST) 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.

```
/* an AVL tree is another name for a balanced binary search tree*/

public class AVLTree <T: Comparable> {

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

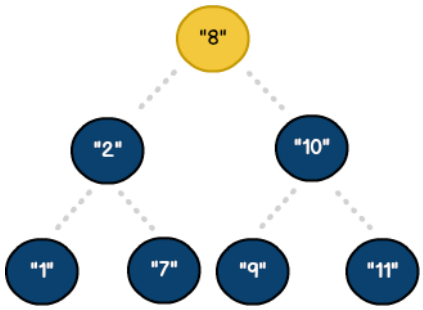
When creating algorithms, it is good practice to set your class properties to `nil` before they are used. Swift helps to enforce this best-practice at compile time through a new paradigm called *optionals*. Along with our class declaration, the generic type constraint `<T: Comparable>` is also defined to ensure instances conform to a specific protocol.

THE LOGIC

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

```
//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)

BUILDING THE HIERARCHY

Here's the class used for creating the BST. Written in Swift, the method `addNode` employs the rules to position each array value.

```
    //add item based on its value

    func addNode(key: T) {

        //check for the head node
        if (self.key == nil) {
            self.key = key
            return
        }


        //check the left side
        if (key < self.key) {

            if (self.left != nil) {
                left?.addNode(key)
            }
            else {

                //create a new left node
                var leftChild : AVLTree = AVLTree()
                leftChild.key = key
                self.left = leftChild
            }
```

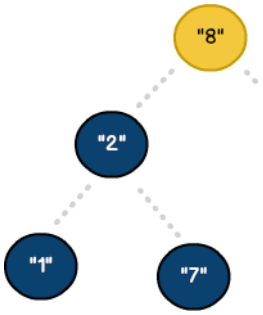
```
          } //end if

         //check the right side
         if (key > self.key) {
             if (self.right != nil) {

               right?.addNode(key)
             }
             else {

                 //create a new right node
                 var rightChild : AVLTree = AVLTree()
                 rightChild.key = key
                 self.right = rightChild
             }

         } //end if

     } //end function
```

The method `addNode` uses recursion to determine where data is added. While not required, recursion is a powerful enabler as each child becomes another instance of a BST. As a result, inserting data becomes a straightforward process of iterating through the array.

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

//create a new BST instance
var root = AVLTree<Int>()

//sort each item in the list
for number in numberList {
     root.addNode(number)
}
```

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.



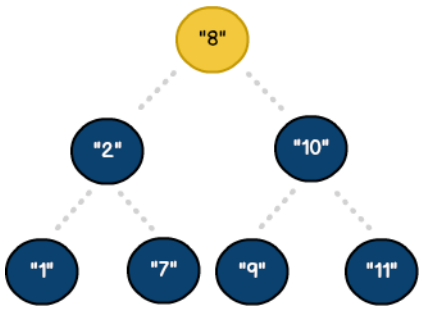
As we saw with Big O Notation, our understanding of the data allows us to create an effective algorithim. So, for example, if we were searching for the value `7`, it's only required that we traverse the left side of the BST. This is due to our search value being smaller than our `root` node (e.g `8`). 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.

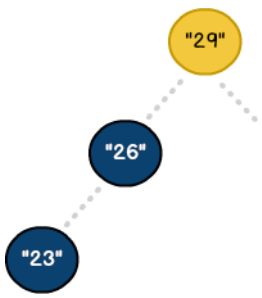
NEW MODELS

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

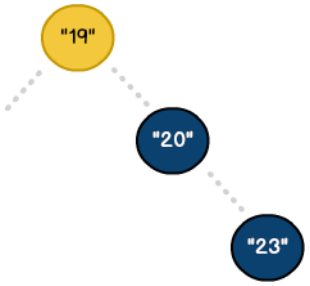


A balanced BST with 7 nodes - O (log n)

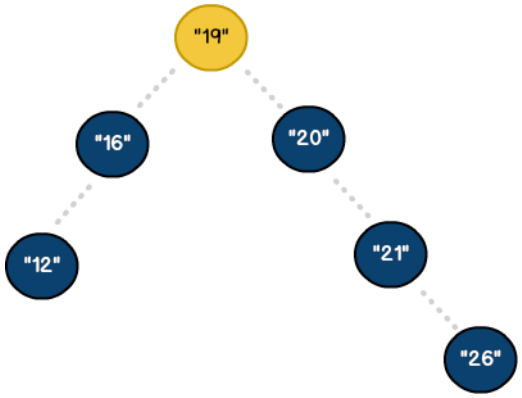
Our model achieved balance not only through usage of the `addWord` algorithm, but also by the way keys were inserted. In reality, there could be numerous ways to populate a `tree`. Without considering other factors, this can produce unexpected results:



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



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

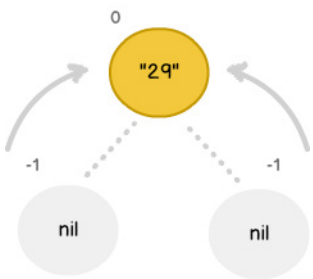


A 6-node BST with left and right imbalances - O(n)

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]
```



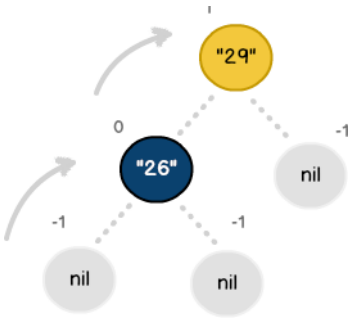
To start, we add the `root` node. As the first item, `left` / `right` leaves don't yet exist so they are initialized to `nil`. Arrows point from the leaf nodes to the `root` because they are used to calculate its `height`. For math purposes, the `height` of non-existent leaves are set to `-1`.

With a model in place, we can calculate the node'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` node this equates to `0`. In Swift, these rules can be represented as follows:

```
//retrieve the height of a node
func getNodeHeight(aNode: AVLTree!) -> Int {

    if (aNode == nil) {
        return -1
    }
    else {
       return aNode.height
    }

}


//calculate the height of a node
func setNodeHeight() -> Bool {

    //check for a nil condition
    if (self.key == nil) {
        println("no key provided..")
        return false
    }

    //initialize leaf variables
    var nodeHeight: Int = 0


    //do comparision and calculate node height
    nodeHeight = max(getNodeHeight(self.left), getNodeHeight(self.right)) + 1

    self.height = nodeHeight

    return true

}
```

MEASURING BALANCE

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

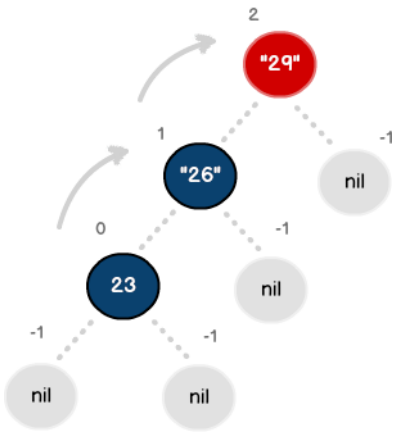
```
//example math for tree balance check

var rootVal: Int!
var leafVal: Int!

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

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

```
    //determine if the tree is "balanced"

    func isTreeBalanced() -> Bool {

        //check for a nil condition
        if (self.key == nil) {
            println("no key provided..")
            return false
        }


        //use absolute values to manage right and left imbalances
        if (abs(getNodeHeight(self.left) - getNodeHeight(self.right)) <= 1) {
            return true
        }
        else {
            return false
        }


    } //end function
```

ADJUSTING THE MODEL

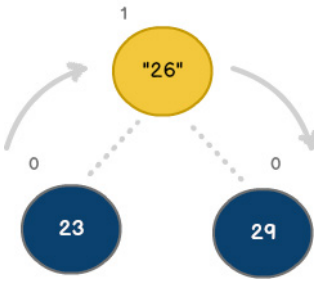
With `29` and `26` added we can proceed to insert the final value (i.e., `23`). Like before, we continue to traverse the `left` side of the tree. However, upon insertion, the math reveals node `29` violates the BST property. In other words, its `subtree` is no longer balanced.



Step three: 23 is added to the BST

```
//example math for tree balance check

var 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

```
            //right rotation sequence

            var childToUse : AVLTree = AVLTree()
            childToUse.height = 0
            childToUse.key = self.key

            if (getNodeHeight(self.left) - getNodeHeight(self.right) > 1) {

                //reset the root node
                self.key = self.left?.key
                self.height = getNodeHeight(self.left)


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


                //adjust the left node
                self.left = self.left?.left
                self.left?.height = 0

                return true

            }
```

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.

THE RESULTS

See p.69 to learn more about traversals and Depth-First Search.

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
```

TRIES

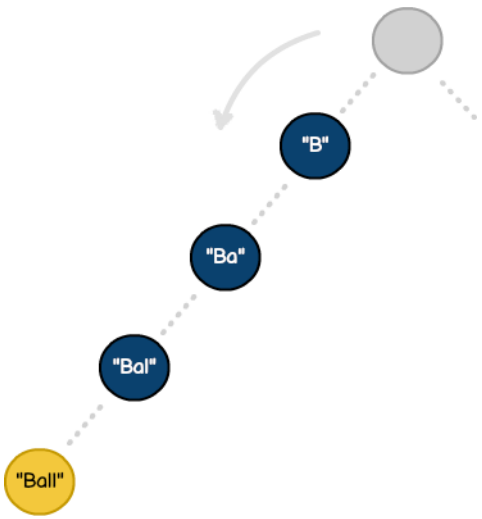
Similar to binary search trees, trie data structures also organize information in a logical hierarchy. Often pronounced *"try"*, the term comes from the English language verb to *retrieve*. While most algorithms are designed to manipulate generic data, tries are commonly used with strings. In this chapter, we'll review trie structures and will implement our own trie model with Swift.

HOW IT WORKS

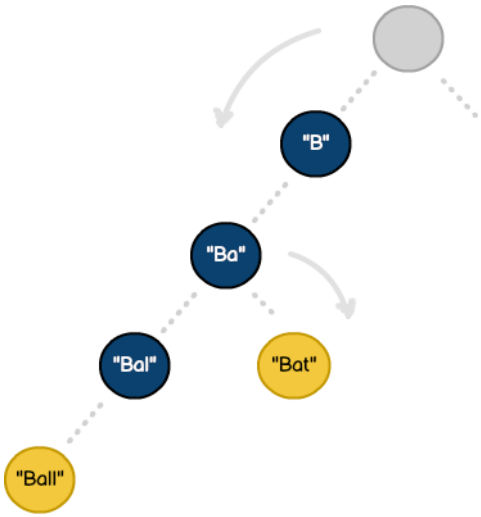
As discussed, tries organize data in a hierarchy. To see how they work, let's build a dictionary that contains the following words:

```
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 iterative tree of dictionary content

func addWord(keyword: String) {


    if (keyword.length == 0){
        return
    }


    var current: TrieNode = root
    var searchKey: String!


    while(keyword.length != current.level) {

        var childToUse: TrieNode!
        var searchKey: String = keyword.substringToIndex(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
```

next page >>

```
            println("end of word reached!")
            return
        }


    } //end function
```

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`.

```
    //finds all words based on the prefix

    func findWord(keyword: String) -> Array<String>! {

        if (keyword.length == 0){
            return nil;
        }

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

        while(keyword.length != current.level) {

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


            //iterate through any children
            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)) {
            wordList.append(current.key)
        }


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

            if (child.isFinal == true) {
                wordList.append(child.key)
            }

        }

        return wordList

    }
```

The `findWord` 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 of string
    var length: Int {
        return count(self)
    }

    //returns characters range to a specified index
    func substringToIndex(to: Int) -> String {
        return self.substringToIndex(advance(self.startIndex, to))
    }


    //return a character at a specific index
    func characterAtIndex(position: Int) -> String {
        return String(Array(self)[position])
    }

}
```

FURTHER READING

» Trie Data Structure - http://en.wikipedia.org/wiki/Trie
» Recursion - https://en.wikipedia.org/wiki/Recursion_(computer_science)

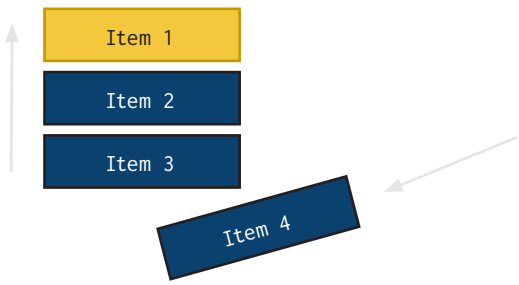
STACKS & QUEUES

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.

In computer science, the idea of stacks and queues can be seen in numerous ways. Any app that makes use of a shopping cart, waiting list or playlist employs a stack or queue. In programming, a call stack is often used as an essential debugging and analytics tool. For this essay, we'll discuss the idea of stacks and queues and will review how to implement a queue in code.

HOW IT WORKS

Queues are based on the concept of *"first-in, first-out"*. When new objects are created, they are added to the bottom of the queue. Items in the queue are preserved in order until requested. When a request is made, the `top` level item is removed and is sent to the client.



In their basic form, stacks and queues employ the same structure and only vary in their use. The data structure common to both forms is the linked list. Using generics, we'll build a queue to hold any type of object. In addition, the class will also support `nil` values.

```
//generic queue node

class QNode<T> {

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

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.

```
    //enqueue the specified object

    func enQueue(key: T) {

        //check for the instance
        if (top == nil) {
            top = QNode<T>()
        }


        //establish the top node
        if (top.key == nil) {
            top.key = key;
            return
        }

        var childToUse: QNode<T> = QNode<T>()
        var current: QNode = top


        //cycle through the list of items
        while (current.next != nil) {
            current = current.next!
        }

        //append a new item
        childToUse.key = key;
        current.next = childToUse;

    }
```

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 *implicit unwrapped 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.

DEQUEUEING OBJECTS

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

```
//retrieve items from the top level in O(1) constant time

func deQueue() -> T? {

     //determine if the key or instance exist
     let topitem: T? = self.top?.key

     if (topitem == nil) {
         return nil
     }

     //retrieve and queue the next item
     var queueitem: T? = top.key!


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


     return queueitem

 }
```

When dequeuing, it is vital to know when values are absent. With `deQueue`, we account for the potential absence of a `key` in addition to an empty queue. In Swift, one must use specific techniques like *optional chaining* to check for `nil` values.

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.

```
//check for queue
func isEmpty() -> Bool {

     //check for key instance
     if let topitem: T = self.top?.key {
         return false
     }

     else {
         return true
     }

 }
```

next page >>

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

EFFICIENCY

In this example, our queue provides `O(n)` for insertion and `O(1)`for lookup. As noted, stacks support the same basic structure and generally provide `O(1)` for both storage and lookup. Similar to linked lists, stacks and queues play an important role when managing other data structures and algorithms.

FURTHER READING

» Stacks - http://en.wikipedia.org/wiki/Stack_(abstract_data_type)
» Queues - http://en.wikipedia.org/wiki/Queue_(abstract_data_type)
» Call Stack - http://en.wikipedia.org/wiki/Call_stack

GRAPHS

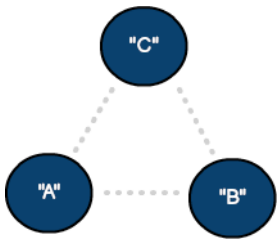
A graph is a data structure that shows a relationship (e.g., connection) between two or more objects. Because of their flexibility, graphs are one of the most widely used structures in modern computing. Popular tools and services like online maps, social networks, and even the Internet as a whole are 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

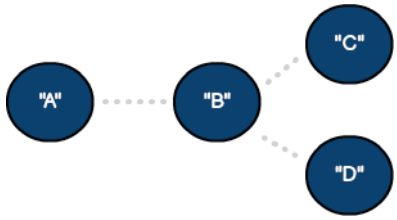
As discussed, a graph is a model that shows how objects relate to one another. Graph objects are usually referred to as `nodes` or `vertices`. While it would be possible to build and graph a single node, models that contain multiple vertices better represent real-world applications.

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.

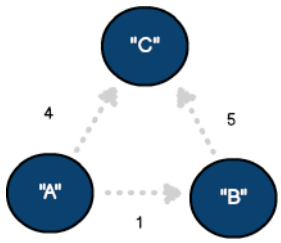
DIRECTED VS. UNDIRECTED

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 a default directed graph canvas
    private var canvas: Array<Vertex>
    public var isDirected: Bool


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


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

        //set the key
        var 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) {

        //create a new edge
        var newEdge = Edge()


        //establish the 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
            var reverseEdge = Edge()

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

        }

    }
```

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*.

FURTHER READING

» Graph Theory - http://en.wikipedia.org/wiki/Graph_theory
» Google Maps - https://www.google.com/maps
» Adjacency Lists - http://en.wikipedia.org/wiki/Adjacency_list

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

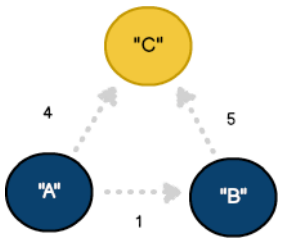
The challenge with graphs is knowing how a `vertex` relates to other objects. Consider the social networking website, LinkedIn. With LinkedIn, each profile can be thought of as a single `vertex` that may be connected with other `vertices`.

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.

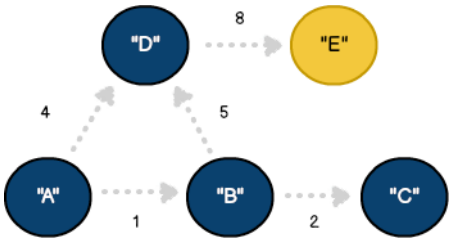


The shortest path from vertex A to C is through vertex A.

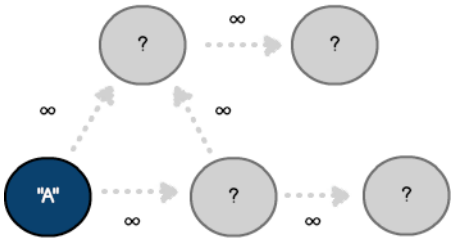
This example illustrates some key points we'll see in *Dijkstra's algorithm*. In addition to there being multiple ways to arrive at vertex `C` from `A`, the shortest path is assumed to be through vertex `B`. It's only when we arrive to vertex `C` from `B`, we adjust our interpretation of the shortest 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() {
        destination = Vertex()
    }

}
```

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>()

        //create the frontier
        for e in source.neighbors {

            var 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

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 {

            //the greedy approach
            bestPath = Path()
            var pathIndex: Int = 0


            for x in 0..<frontier.count {

                var itemPath: Path = frontier[x]

                if  (bestPath.total == nil) || (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 {

                var 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.removeAtIndex(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.

See the complete algorithm for Dijkstra's Shortest Path on Github.

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.

FURTHER READING

» Graph Theory - http://en.wikipedia.org/wiki/Graph_theory
» Greedy Approach - http://en.wikipedia.org/wiki/Greedy_algorithm
» Traveling Salesman - http://en.wikipedia.org/wiki/Travelling_salesman_problem
» Edsger Dijkstra - http://en.wikipedia.org/wiki/Edsger_W._Dijkstra
» Dijkstra's Algorithm - http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

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.

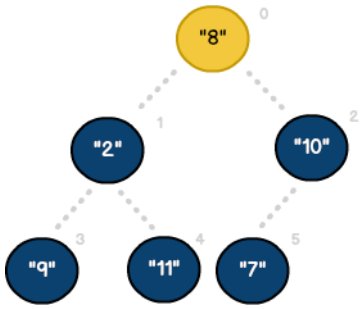
When designing programs, it's great to see them work. With Dijkstra, the algorithm did allow us to find the shortest path between a source vertex and destination. However, our approach could be refined to be more efficient. In this essay, we'll enhance the algorithm with the addition of binary heaps.

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:

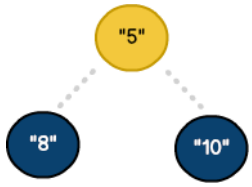
$$parent = floor(\frac{i-1}{2})$$

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

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.

```
        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 {

                var itemPath: Path = frontier[x]

                if  (bestPath.total == nil) || (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

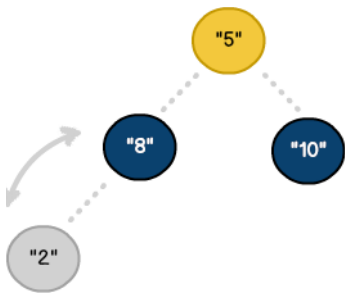
Searching the `frontier` more efficiently than `O(n)` will require a new way of thinking. We can improve our performance to `O(1)` with a heap. Using heapsort formulas, our new approach will involve *arranging* index values so the smallest item is positioned at the `root`.

```
    //sort paths into a min-heap

    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))
        }
```

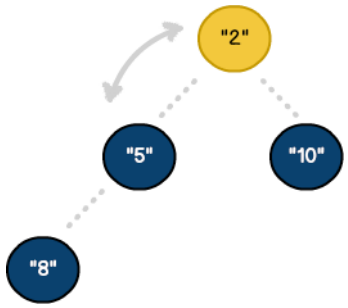
next page >>

```
        var childToUse: Path
        var parentToUse: Path


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


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


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

            //reset indices
            childIndex = Float(parentIndex)

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


        } //end while


    } //end function
```

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.

```
//construct the best path

var frontier: PathHeap = PathHeap()
var bestPath: Path = Path()

    while frontier.count != 0 {

        //use the greedy approach to obtain the best path
        bestPath = Path()
        bestPath = frontier.peek()

   .....
```

See the complete algorithm for Dijkstra's Shortest Path with Heaps on Github.

FURTHER READING

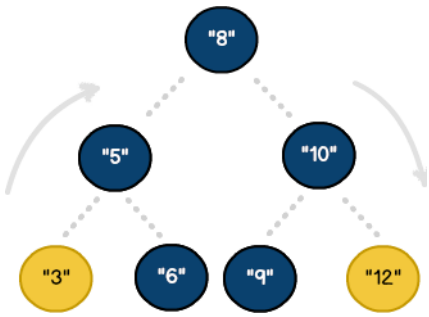
» Greedy Approach - http://en.wikipedia.org/wiki/Greedy_algorithm
» Binary Heaps - http://en.wikipedia.org/wiki/Binary_heap
» Heapsort - http://en.wikipedia.org/wiki/Heapsort
» Base Case - http://en.wikipedia.org/wiki/Mathematical_induction

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.

```
//dfs in-order traversal

func processAVLDepthTraversal() {

    //check for nil condition
    if self.key == nil {
       println("no key provided..")
       return
    }
```

next page >>

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

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

    }
```

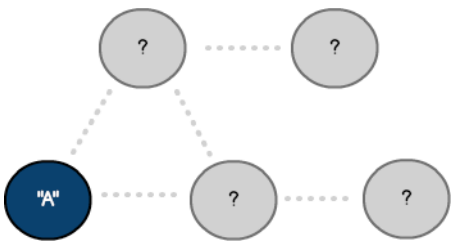
At first glance, we note the algorithm makes use of recursion. The `AVLTree` node (e.g. self), contains a `key`, as well as pointer references to its left and right nodes. For each side, (e.g., `left` and `right`) the base case consists of a straightforward check for `nil`. This process allows us to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

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

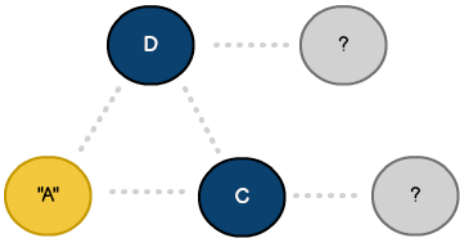
BREADTH-FIRST SEARCH

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 traverseGraphBFS(startingv: Vertex) {

     var graphQueue: Queue<Vertex> = Queue<Vertex>()

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


     while !graphQueue.isEmpty() {

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

         //add unvisited vertices to the queue
         for e in vitem.neighbors {
             if e.neighbor.visited == false {
                 println("adding vertex: \(e.neighbor.key!) to queue..")
                 graphQueue.enQueue(e.neighbor)
             }
         }


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


     } //end while


     println("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.

FURTHER READING

» Traversal - http://en.wikipedia.org/wiki/Tree_traversal
» Depth-First Search - http://en.wikipedia.org/wiki/Depth-first_search
» Breadth-First Search - http://en.wikipedia.org/wiki/Breadth-first_search

HASH TABLES

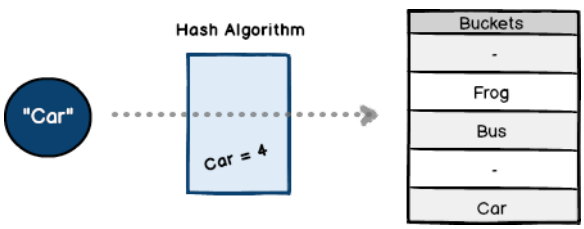
A hash table is a data structure that groups values to a key. As we've seen, structures like graphs, tries and linked lists follow this widely adopted model. In some cases, built-in Swift data types (like dictionaries) also accomplish the same goal. In this chapter, we'll examine the advantages of hash 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 a key. By definition, linked lists provide a straightforward way to associate related items. While the advantage of linked lists is flexibility, their downside is lack of speed. Since the only way to search a list is to traverse the entire set, their efficiency is typically limited to `O(n)`. To contrast, a dictionary associates a value to a user-defined key. This helps pinpoint operations to specific entries. As a result, dictionary operations are typically `O(1)`.

THE BASICS

As the name implies, a hash table consists of two parts - a key and value. However, unlike a dictionary, the key is a *calculated* sequence of numbers and / or characters. The output is known as a *hash*. The mechanism that creates a 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 their input to create a unique hash. In this example, the input of *"Car"* always produces the key result of `4`.

THE DATA STRUCTURE

Here's a hash table data structure written in Swift. At first glance, we see the structure resembles a linked list. In a production environment, our `HashNode` could represent any combination of items, including custom objects. Additionally, there is no class property for storing a `key`.

```
//simple hash table structure

class HashNode {

    var firstname: String!
    var lastname: String!
    var next: HashNode!
}
```

THE BUCKETS

Before using our table, we must first define a `bucket` structure. If you recall, `buckets` are used to group node items. Since items will be stored in a non-contiguous fashion, we must first define our collection size. In Swift, this can be achieved with the following:

```
class HashTable {

    private var buckets: Array<HashNode!>

    //initialize the buckets with nil values
    init(capacity: Int) {
        self.buckets = Array<HashNode!>(count: capacity, repeatedValue:nil)

    }
}
```

ADDING WORDS

With the components in place, we can code a process for adding words. The `addWord` method starts by concatenating its parameters as a single `String`. The result is then passed to the `createHash` hash algorithm which, subsequently, returns an `Int`. Once complete, we conduct a simple check for an existing entry.

```
    //add the key using the specified hash

    func addWord(firstname: String, lastname: String) {

        var hashindex: Int!
        var fullname: String!

        //create a hashvalue using the complete name
        fullname = firstname + lastname
        hashindex = self.createHash(fullname)


        var childToUse: HashNode = HashNode()
        var head: HashNode!
```

```
        childToUse.firstname = firstname
        childToUse.lastname = lastname

        //check for an existing list
        if (buckets[hashindex] == nil) {
            buckets[hashindex] = childToUse
        }
```

HASHING AND CHAINING

The key to effective hash tables is their hash functions. The `createHash` method is a straightforward algorithm that employs modular math. The goal? Compute an `array` index.

```
    //compute hash index
    func createHash(fullname: String) -> Int! {

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

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

        //calculate the remainder
        remainder = divisor % buckets.count

        return remainder - 1
    }
```

With any hash algorithm, the aim is to create enough complexity to eliminate *collisions*. Such a model will provide `O(1)` performance for insertion, modification and lookup. Collisions occur when different inputs compute to the same `hash`. In our example, the inputs of *Albert Einstein* and *Andrew Collins* produce the same `hash` result of `8`.

Separate chaining can also be implemented with binary search trees. This can improve hash table performance to O(log n).

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 inputs to share the same `hash`.

```
    //add the key using the specified hash

    func addWord(firstname: String, lastname: String) {

        var hashindex: Int!
        var fullname: String!
```

next page >>

```
        //create a hashvalue using the complete name

        fullname = firstname + lastname
        hashindex = self.createHash(fullname)

        var childToUse: HashNode = HashNode()
        var head: HashNode!


        childToUse.firstname = firstname
        childToUse.lastname = lastname


        //check for an existing list
        if  buckets[hashindex] == nil {
            buckets[hashindex] = childToUse
        }

        else {

            println("a collision occured. implementing chaining..")
            head = buckets[hashindex]

            //append new item to the head of the list
            childToUse.next = head
            head = childToUse

            //update the chained list
            buckets[hashindex] = head
        }

    }
```

FURTHER READING

» Hash Table - http://en.wikipedia.org/wiki/Hash_table
» Hash Algorithm - http://en.wikipedia.org/wiki/Hash_function
» Modular Math - https://www.khanacademy.org
» MD5 Algorithm - http://en.wikipedia.org/wiki/MD5

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 signifcance 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>! {

        if head.key == nil {
           return nil
        }

        var current: LLNode! = head
        var 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.addLink(newKey)
            }
```

```
            current = current.next
        }

        return results

    } //end function
```

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
        var linkedList: LinkedList<Int> = self.buildLinkedList()

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

    }
```

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: (inout node: Vertex) -> ()) {
  ...
}
```

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


    //establish a new queue
    var 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 {
                println("adding vertex: \(e.neighbor.key!) to queue..")
                graphQueue.enQueue(e.neighbor)
            }
        }

        //pass by reference - no return value required
        formula(node: &vitem)


    } //end while


    println("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) { (inout node: Vertex) -> () in

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

    }

}
```

See the complete suite of algorithm unit tests on Github.

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.

FURTHER READING

» Anonymous Functions - https://en.wikipedia.org/wiki/Anonymous_function
» Closures - https://developer.apple.com/library

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.

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.

Have an idea for this project? Submit feedback at waynewbishop.com/feedback.

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*
- *Natasha Murashev*
- *Mark McLaren*
- *Rahul Salota*

THE AUTHOR

Wayne Bishop resides in Seattle, Washington, and is an independent iOS developer and Technical 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.