Swift Algorithms & Data Structures

Wayne Bishop

Published with GitBook

Table of Contents

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 exists 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, audiences 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 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 allow 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 (Uber.com) or machine learning (e.g Amazon.com).

Get the latest code for Swift algorithms and data structures on Github.

Big O Notation

Building a service 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.

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

Linear Time - O(n)

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;

```
//brute force approach - O(n) linear time
func linearSearch(key: Int) {

    //check all possible values until we find a match
    for number in numberList {
        if (number == key) {
            let results = "value \(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 become less efficient as its input size (n) grows.

Logarithmic Time - O(log n)

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.

```
//the binary approach - O(log n) logarithmic time
func binarySearch(key: Int, imin: Int, imax: Int) {

    //find the value at the middle index
    var midIndex : Double = round(Double((imin + imax) / 2))
    var midNumber = numberList[Int(midIndex)]

    //use recursion to reduce the possible search range
    if (midNumber > key ) {
        binarySearch(key, imin, Int(midIndex) - 1)
    }
```

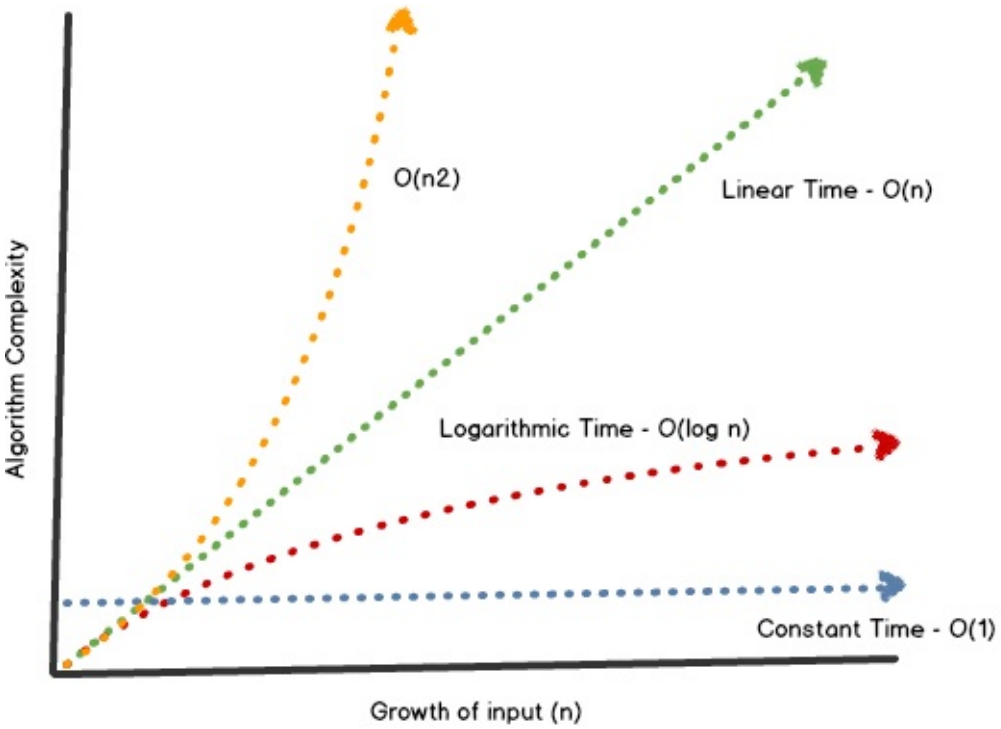
```
        else if (midNumber < key ) {
            binarySearch(key, Int(midIndex) + 1, imax)
        }
        else {
            let results = "value \(key) found.."
        }
    } //end func
```

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 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 their performance;

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.



Not familiar with logarithms? Here's a great Khan Academy video that demonstrates how this works.

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 algorithm, insertion sort.

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 an secondary engine, reviewing which numbers get compared. Completed enough times, this process eventually sorts all items in the list.

```
//insertion sort - rank items by comparing each key in the list.
func insertionSort() {
    var x, y, key : Int
    for (x = 0; x < numberList.count; x++) {
        //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)

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

```
/* bubble sort algorithm - rank items from the lowest to highest by swapping groups of two items from left to right. */
func bubbleSort() {
    var x, y, z, passes, key : Int
    //track collection iterations
    for x in 0..<numberList.count {
        passes = (numberList.count - 1) - x;

        //use shorthand "half-open" range operator
        for y in 0..<passes {
```

```
            key = numberList[y]

            //compare and rank values
            if (key > numberList[y + 1]) {
                z = numberList[y + 1]
                numberList[y + 1] = key
                numberList[y] = z
            }
        } //end for
    } //end for
} //end function
```

Efficiency

Besides insertion sort and bubble sort, there are many other sorting algorithms. Popular techniques include quicksort, mergesort and selection sort. Because both insertion and bubble sort algorithms combine a variant and invariant, their average performance is ***n x n*** or ***O(n²)***. Other techniques (like mergesort) apply different methods and can improve average performance to ***O(n log n)***.

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.

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 items 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 new 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 simple function that creates a doubly linked list. The method ***addLink*** creates a new item and appends it to the list. The Swift generic type contraint ***<T: Equatable>*** is also defined to ensure instances conform to a specific protocol.

```
public class LinkedList<T: Equatable> {

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

    //append a new item to a linked list
    func addLink(key: T) {
        //establish the head node
        if (head.key == nil) {
            head.key = key;
            return;
        }
        //establish the iteration 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;
            }
            current = current?.next
        } //end while
    } ///end function
```

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 requires reassigning links so the chain remains unbroken.

```
    //remove a link at a specific index
    func removeLinkAtIndex(index: Int) {
        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;
            }

            //update the assignments
            trailer = current
            current = current?.next
            listIndex++
        } //end while
    } //end function
```

Counting Links

When implemented, it can also be convenient to ***count*** 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 linked 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
        }
    } //end property
```

Efficiency

Linked lists as shown 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 the Swift programming language 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 it 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 can 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 resued 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 contraint ***Equatable*** is appended to the generic ***T*** placeholder. Native to the Swift language, ***Equatable*** is a simple protocol that ensures various object types conform to the equatable specification.

```
//generic linked list implementation
public class LinkedList<T: Equatable> {
    private var head: LLNode<T> = LLNode<T>()

    //add generic nodes at a specified index
    func addLinkAtIndex(key: T, index: Int) {
        //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 infront of the current node
                childToUse.next = current
                childToUse.previous = trailer

                //use optional binding for ading 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

            //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 the 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.

```
//generic binary search tree
public class AVLTree<T: Comparable> {
    var key: T?
    var left: AVLTree?
    var right: AVLTree?

    init() {
    }
}
```

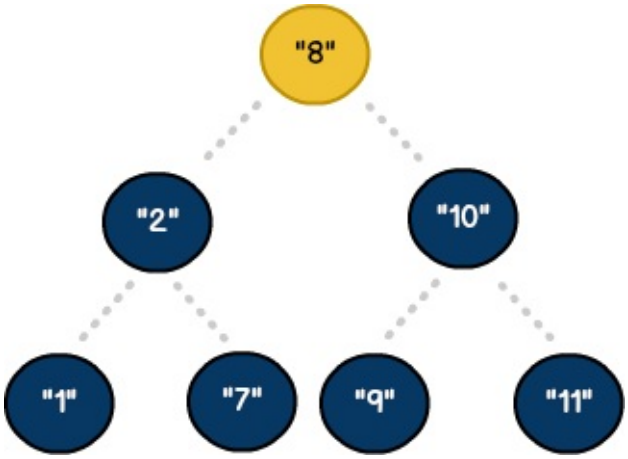
When creating algorithms, it is good practice to set your class properties to ***nil*** before they are used. Swift helps to enforce this best-practice at compile time through a new paradigm called Optionals. Along with our class declaration, the generic type contraint ***<T: Comparable>*** is also defined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



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.

```
public class AVLTree<T: Comparable> {
    var key: T?
    var left: AVLTree?
    var right: AVLTree?

    //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 of the tree
        if (key < self.key) {
            if (self.left != nil) {
                left!.addNode(key)#sthash.3jVtGzNq.dpuf
            }
            else {
                //create a new left node
                var leftChild : AVLTree = AVLTree()
                leftChild.key = key
                self.left = leftChild
            }
        } //end if

        //check the left side of the tree
        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
} //end class
```

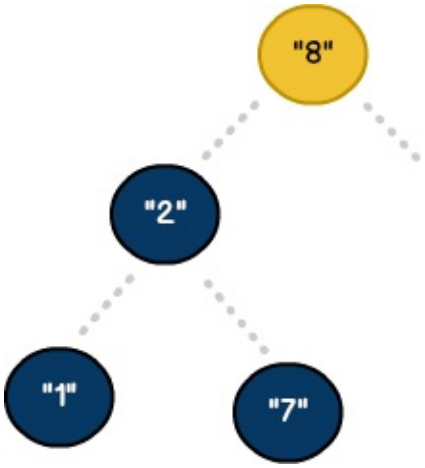
The method ***addNode*** uses recursion to determine where data is added. While not required, recursion is a powerful enabler as each child becomes another instance of a BST. As a result, inserting data becomes a straightforward process of iterating through the array.

```
//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.



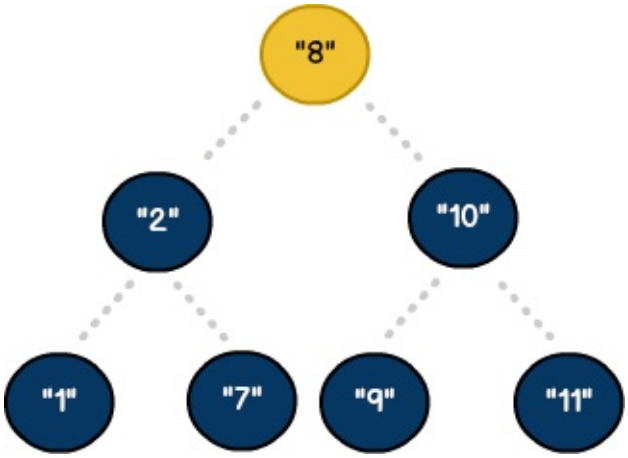
As we saw with Big O Notation, our understanding of the data allows us to create an effective algorithim. So, for example, if we were searching for the value 7, its only required that we traverse the left side of the BST. This is due to our search value being smaller than our root node (e.g 8). Binary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

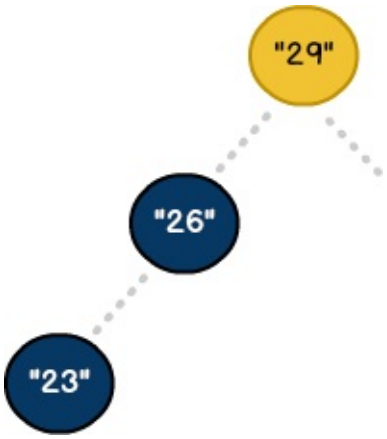
In the previous essay, we saw how binary search trees (BST) are used to manage data. With basic logic, an algorithm can easily traverse a model, searching data in ***O(log n)*** time. However, there are occasions when navigating a tree becomes inefficient - in some cases working at ***O(n)*** time. In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

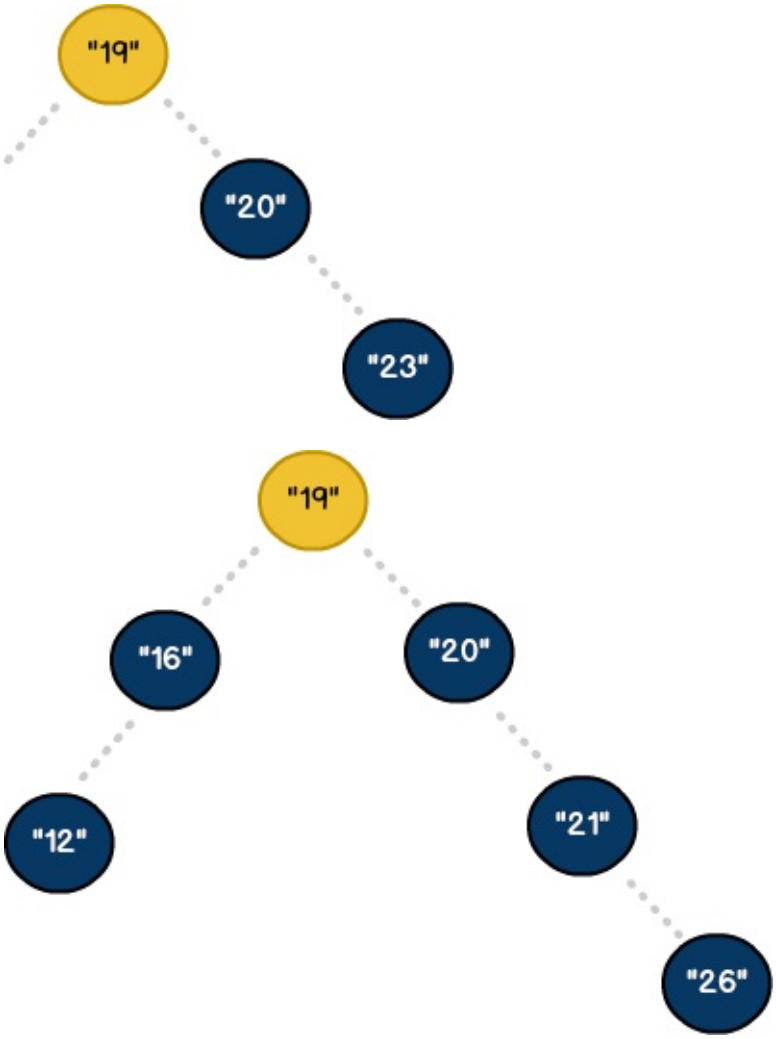
New Models

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 known as a balanced binary search tree.



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:

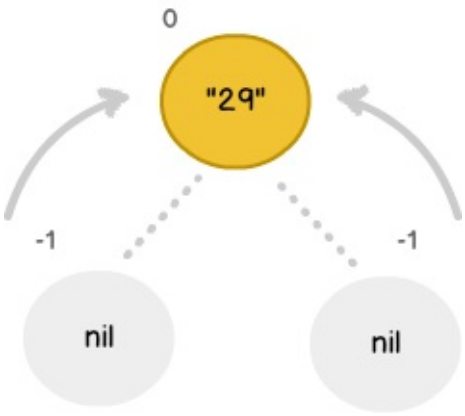




New Heights

To compensate for these imbalances, we need to expand the scope of our algorithm. In addition to left / right logic, we'll add 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-

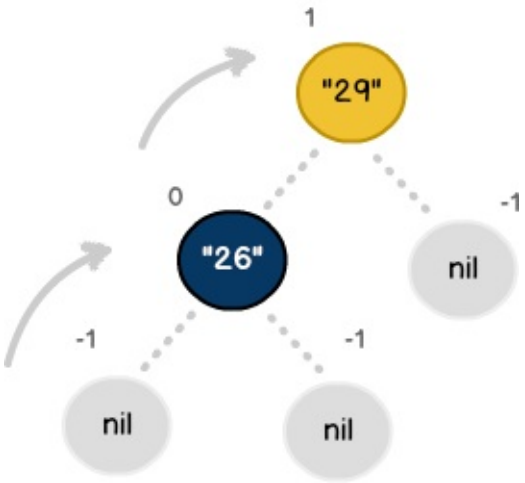
existent leaves are set to ***-1***.

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
    }

    //set height variable
    var nodeHeight: Int = 0

    //compare 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.



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
```

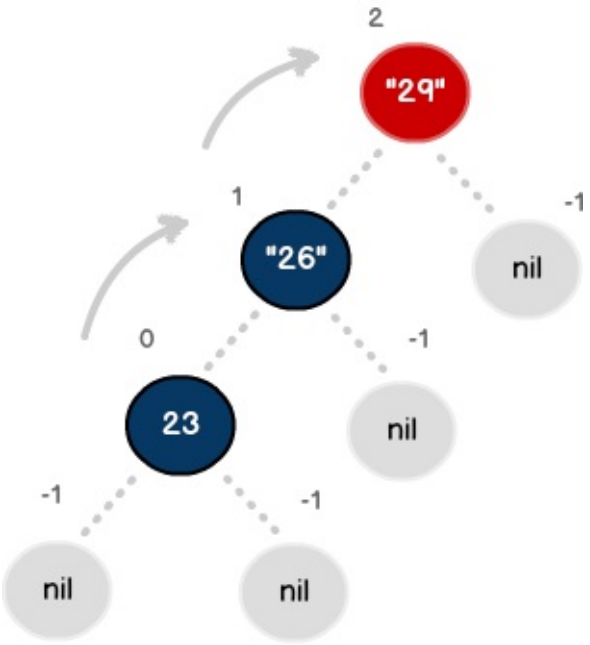
```
var rootVal: Int!
var leafVal: Int!

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 value to calculate right / left imbalances
    if (abs(getNodeHeight(self.left) - getNodeHeight(self.right)) <= 1) {
        return true
    }
    else {
        return false
    }
} //end function
```

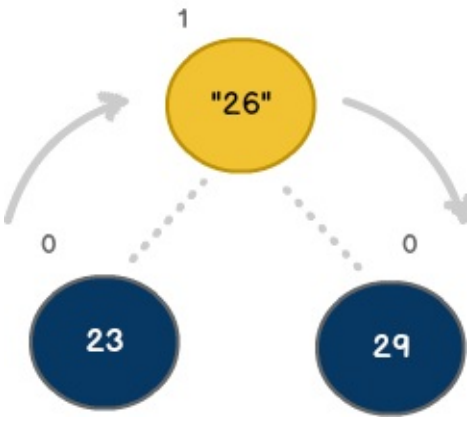
Adjusting the Model

With ***29*** and ***26*** added 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.




```
//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 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:



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

```
//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
}
```

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

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

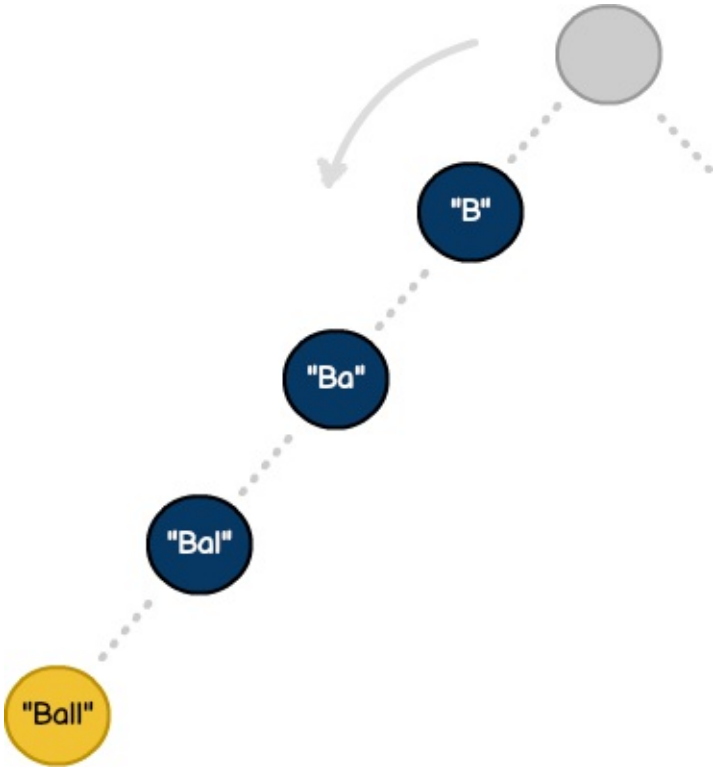
Similar to binary search trees, trie data structures also organize information in a logical hierarchy. Often pronounced "try", the term comes from the English language verb to "retrieve". While most algorithms are designed to manipulate generic data, tries are commonly used with strings. In this essay,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

How it works

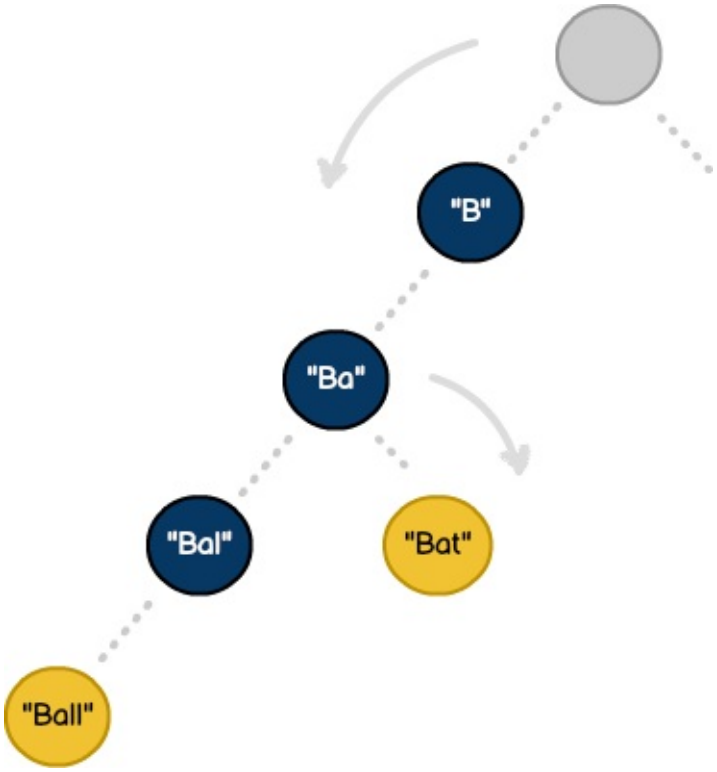
As discussed, tries organize data in a hierarchy. To see how they work, let's build a dictionary that contains the following words:

```
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. Like 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:



As shown, we've traversed the structure to store the word "Bat". The trie has allowed us to reuse the permutations of "B" and "Ba" added by the inclusion of the word "Ball". Though most algorithms are measured on time efficiency, tries demonstrate great efficiency with time and space. Practical implementations of tries can be seen in modern software features like auto-complete, search engines and spellcheck.

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.

```
//generic 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#Recursive_functions_and_algorithms) 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.

```
//generic trie implementation
public class Trie {
    private var root: TrieNode!
```

```
    init(){
        root = TrieNode()
    }

    //builds a recursive 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 = 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
            println("end of word reached!")
            return;
        }
    } //end function
}
```

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

Finding Words

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

```
//find all words based on a 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 = keyword.substringToIndex(current.level + 1)

        //iterate through any children
        for child in current.children {
            if (child.key == searchKey) {
```

```
                childToUse = child
                current = childToUse
                break
            }
        }

        //prefix not found
        if (childToUse == nil) {
            return nil
        }
    } //end while

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

    //add children that are words
    for child in current.children {
        if (child.isFinal == true) {
            wordList.append(child.key)
        }
    }
    return wordList
} //end function
```

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:

```
//extend the native String class
extension String {
    //compute the length of string
    var length: Int {
        return Array(self).count
    }

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

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 kitchen 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.

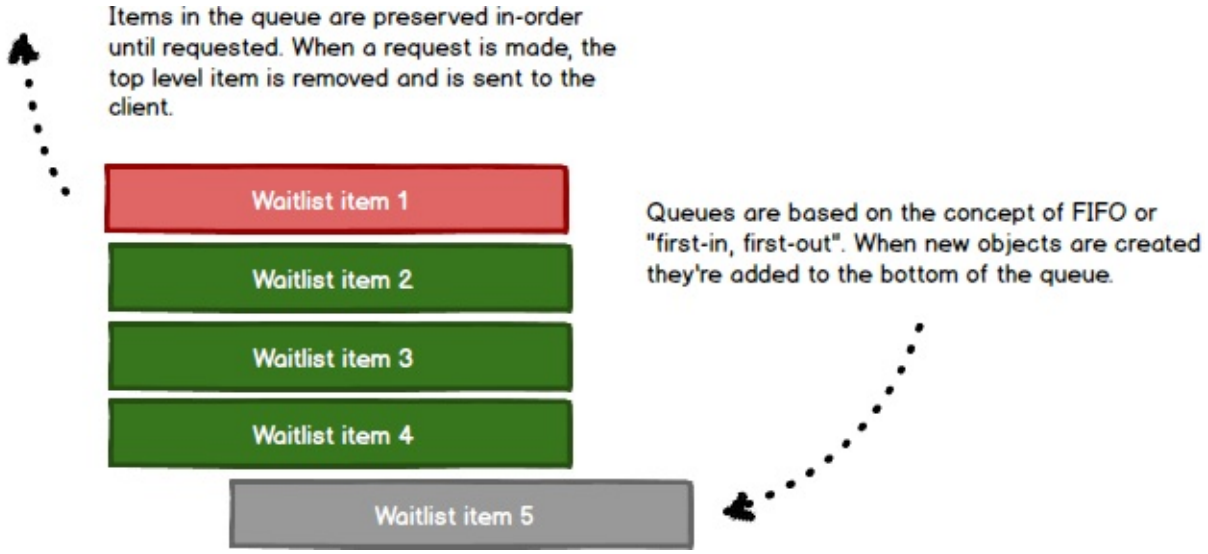
How it works

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. We'll see the significance of these details as we create additional components.

```
//generic queue data object
class QNode<T> {
    var key: T?
    var next: QNode?
}
```

The Concept

Along with our data structure, we'll implement a factory class for managing items. As shown, queues support the concept of adding and removal along with other supportive functions.



Enqueuing Objects

The process of adding items is often referred to as 'enqueuing'. Here, we define the method to enqueue objects as well as the property ***top*** that will serve as our queue list.

```
public class Queue<T> {
```

```
    private var top: QNode<T>! = QNode<T>()

    //enqueue the specified object
    func enQueue(var 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 to get to the end.
        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 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 exists
    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. For the method ***deQueue***, we account for the potential absence of a ***key*** in addition to an empty queue (e.g. ***nil***). In Swift, one must use specific techniques like optional chaining to check for ***nil*** values.

Supporting Functions

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

```
//check for the presence of a value
func isEmpty() -> Bool {
    //determine if the key or instance exist
    if let topitem: T = self.top?.key {
        return false
    }
    else {
        return true
    }
}

//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 but 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.

Graphs

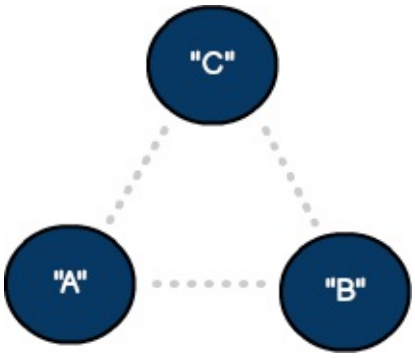
A graph is a structure that shows a relationship (e.g., connection) between two or more objects. Because of their flexibility, graphs are one of the most widely used structures in modern computing. Popular tools and services like online maps, social networks, and even the Internet as a whole are based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

The Basics

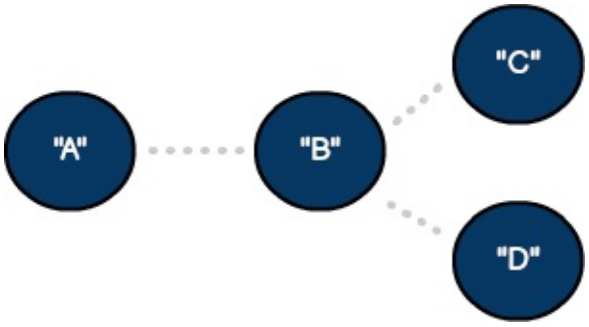
As discussed, a graph is a model that shows how objects relate to one another. Graph objects are usually referred to as ***nodes*** or ***vertices***. While it would be possible to build and graph a single node, models that contain multiple vertices better represent real-world applications.

Graph objects relate to one other 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

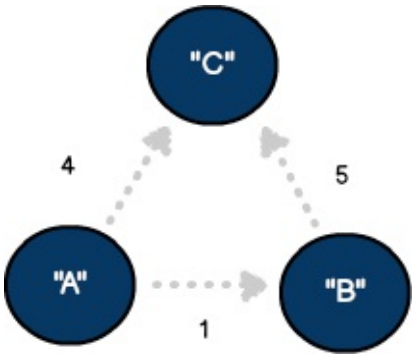
As shown, there are many ways to configure a graph. An additional option is to set the model to be directed or undirected. The examples above represent undirected graphs. In other words, the connection between vertices ***A*** and ***B*** is equivalent

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***:

```
//a basic vertex data structure
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 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***:

```
//a basic edge data structure
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 default directed graph canvas
public class SwiftGraph {
    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)#sthash.bggaFcqw.dpuf

        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 an 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 for 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*.

Shortest Paths

In the previous essay 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 essay, we'll review how to search a graph and will implement a popular algorithm called Dijkstra's shortest path.

Making Connections

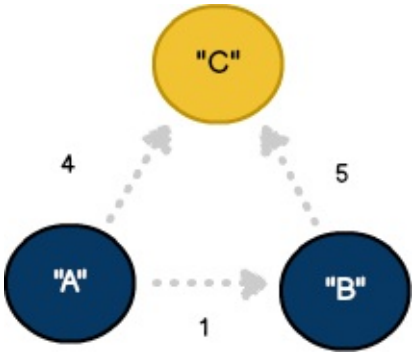
The challenge with graphs is knowing how a ***vertex*** relates to other objects. Consider the social networking website, LinkedIn. With LinkedIn, each profile can be thought of as a single ***vertex*** that may be connected with other ***vertices***.

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.

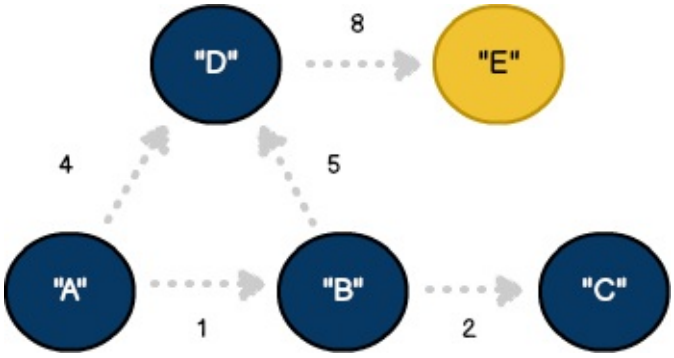


The shortest path from vertex A to C is through vertex A.

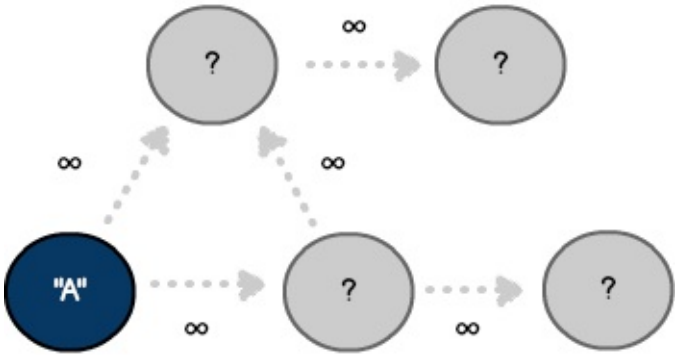
This example illustrates some key points we'll see in Dijkstra's algorithm. In addition to there being multiple ways to arrive at vertex ***C*** from ***A***, the shortest path is assumed to be through vertex ***B***. It's only when we arrive to vertex ***C*** from ***B***, we adjust our interpretation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

```
//maintain objects that make 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 algorthim 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 = Array()
var finalPaths: Array = Array()

//use source edges to 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 essay, 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:

```
//obtain the best path
var bestPath: Path = Path()
while(frontier.count != 0) {
    //support path changes using the greedy approach
    bestPath = Path()

    var x: Int = 0
    var pathIndex: Int = 0

    for (x = 0; x < frontier.count; x++) {
        var itemPath: Path = frontier[x]

        if (bestPath.total == nil) || (itemPath.total < bestPath.total) {
            bestPath = itemPath
            pathIndex = x
        }
    } //end for
```

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.

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 essay.

Want to see the entire algorithm? Here's the source.

Heaps

In the previous essay, 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.

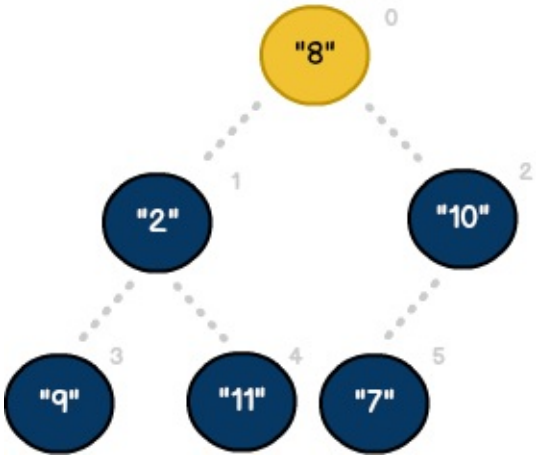
When designing programs, it's great to see them work. With Dijkstra, the algorithm did allow us to find the shortest path between a source vertex and destination. However, our approach could be refined to be more efficient. In this essay, we'll enhance our algorithm with the addition of binary heaps.

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***.



The array visualized as a "nearly complete" binary tree.

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

$$\text{children} = 2i + 1 \text{ and } 2i + 2$$

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

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.

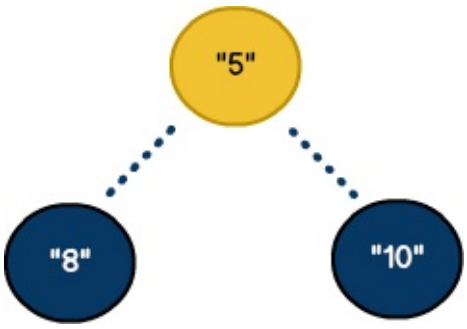


Fig. 2. A heap structure that maintains the min-heap property.

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.

```
//obtain the best path
var bestPath: Path = Path()
while(frontier.count != 0) {
    //support path changes using the greedy approach
    bestPath = Path()

    var x: Int = 0
    var pathIndex: Int = 0

    for (x = 0; x < frontier.count; x++) {
        var itemPath: Path = frontier[x]

        if (bestPath.total == nil) || (itemPath.total < bestPath.total) {
            bestPath = itemPath
            pathIndex = x
        }
    } //end for
```

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

The Heap Structure

Let's create a more efficient ***frontier***. Named ***PathHeap***, the class will extend the functionality of an ***Array***.

```
//a basic min-heap data strcture
public class PathHeap {
    private var heap: Array

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

    init() {
        heap = Array()
    }
}
```

PathHeap includes two properties - ***Array*** and ***Int***. To support good design (e.g., encapsulation), the ***heap*** has been declared a private property. To track the number of items, the ***count*** has also been declared as a computed property.

Building the Queue

Finding the best ***path*** more efficiently than ***O(n)*** will require a new way of thinking. We can improve our algorithm's performance to ***O(1)*** through a "bubble-up" approach. Using our heapsort formulas, this involves "swapping" index values so the smallest item is positioned at the root.

```
//sort shortest paths into a min-heap (heapify)
func enQueue(key: Path) {
    heap.append(key)

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

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

    //use the bottom-up approach
    while (childIndex != 0) {
        var childToUse: Path = heap[Int(childIndex)]
        var parentToUse: Path = heap[parentIndex]

        //swap child and parent positions
        if (childToUse.total < parentToUse.total) {
            heap.insert(childToUse, atIndex: parentIndex)
            heap.removeAtIndex(Int(childIndex) + 1)

            heap.insert(parentToUse, atIndex: Int(childIndex))
            heap.removeAtIndex(parentIndex + 1)
        }

        //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 algorithm's, 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.

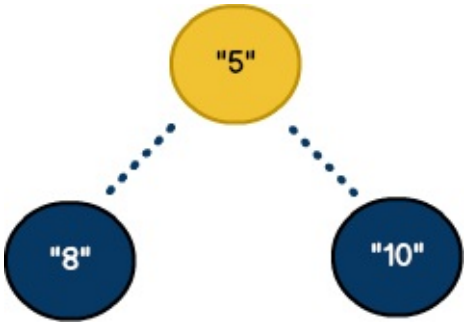


Fig. 3. A heap structure that maintains the min-heap property.

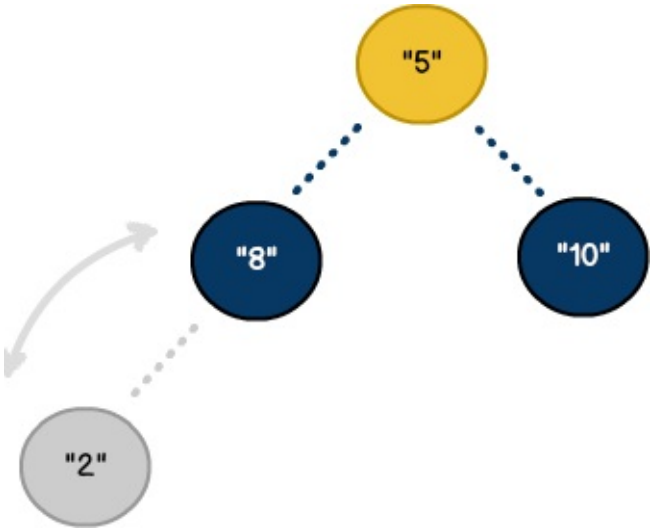


Fig. 4 . The enQueue process compares a newly added value with its parent in a process called "bubbling-up".

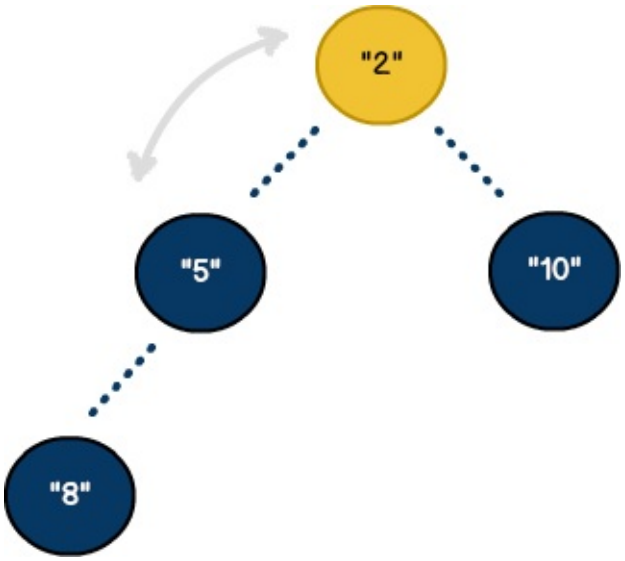


Fig. 5. The compare / swap process continues recursively until the smallest value is positioned at the root.

Since the ***enQueue*** method maintains the min-heap property (as new items are added), we all but eliminate the task of finding the shortest path. Here, we implement a basic ***peek*** method to retrieve the root-level item:

```
//obtain the minimum path
func peek() -> Path! {
    if (heap.count > 0) {
        var shortestPath: Path = heap[0]
        return shortestPath
    }
    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 PathHeap ***count*** forms the base case for our ***loop*** condition and the ***bestPath*** is retrieved using the ***peek*** method.

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

while(frontier.count != 0) {
    //use the greedy approach to obtain the best path
    bestPath = Path()
    bestPath = frontier.peek()
}
```

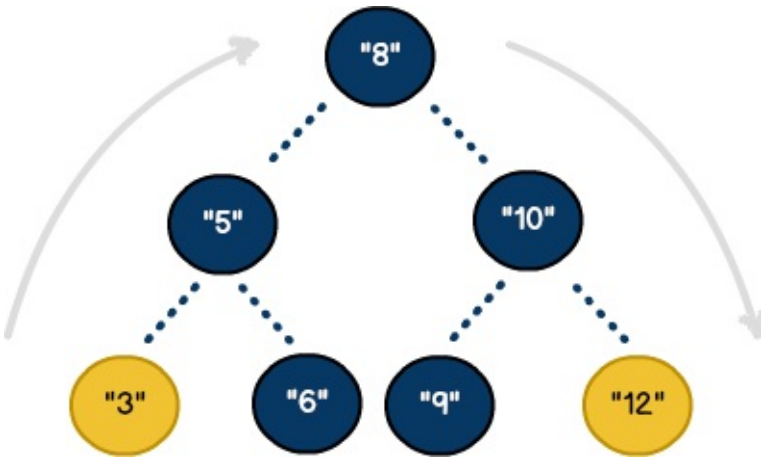
Want to see the entire algorithm? Here's the source.

Traversals

Throughout this series we've explored building 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 a database, their contents can also be explored though a process called traversal. In this essay, 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 "visiting" each node in a data structure. In practical terms, traversals can be seen through activities like network administration. For example, administrators will often deploy software updates to networked computers as a single task. To see how traversal works, let's introduce the process of Depth-First Search (DFS). This methodology is commonly applied to tree-shaped structures. As illustrated, our goal will be to explore the left side of the model, visit the root node, then visit the right side. Using a binary search tree, we can see the path our traversal will take:



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

```
//depth-first in-order traversal
func processAVLDepthTraversal() {
    //check for a nil condition
    if (self.key == nil) {
        println("no key provided..")
        return
    }

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

    println("key is \(self.key!) visited..")

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

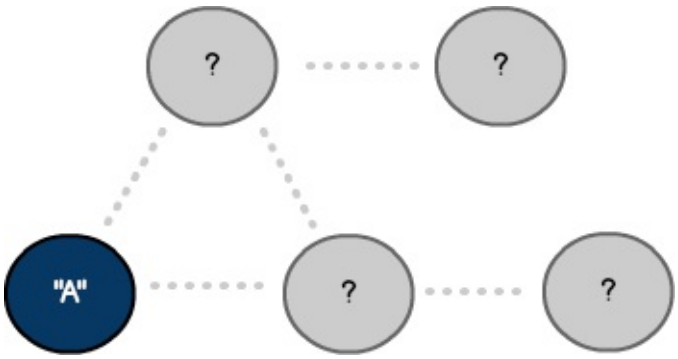
At first glance, we see the algorithm makes use of recursion. With recursion, each ***AVLTree*** node (e.g. self), contains a ***key***, as well as pointer references to its ***left*** and ***right*** nodes. For each side, (e.g., left & right) the base case consists of a straightforward check for ***nil***. This process allows us to traverse the entire structure from the bottom-up. When applied, the algorithm traverses the structure in the following order:

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

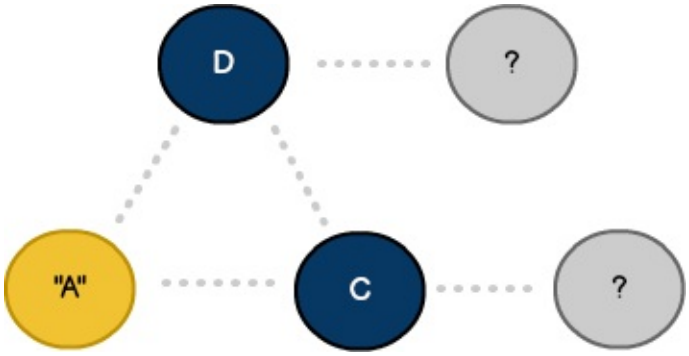
Breadth-First Search

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 introduced including stacks and queues, generics and Dijkstra's shortest path. With BFS, our goal is to visit all neighbors before visiting our neighbor's, "neighbor". Unlike Depth-First Search, the process is based on random 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 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. As a result, vertices are processed using a generic queue.

```
//breadth-first traversal
func traverseGraphBFS(startingv: 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 = 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!)")
                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 completes when all ***vertices*** are visited. To ensure nodes are not visited more than once, each ***vertex*** is marked with a boolean flag.

Hash Tables

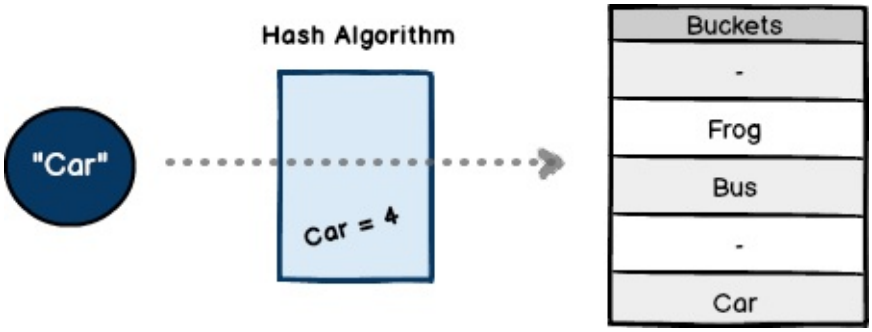
A hash table is a data structure that groups values to a key. As we've seen, structures like graphs, tries and linked lists follow this widely-adopted model. In some cases, built-in Swift data types like dictionaries also accomplish the same goal. In this essay, we'll examine the advantages of hash tables 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 constrast, 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 following 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 content 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 member property for storing a ***key***.

```
//simple example of a hash table node
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 (e.g.,***fullname***). The result is then passed to the ***createHash*** helper function which subsequently, returns an ***Int***. Once complete, we conduct a simple check for an existing entry.

```
//add the value using a 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 bucket
    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 ***index***.

```
//compute the hash value to be used
func createHash(fullname: String) -> Int! {
    var remainder: Int = 0
    var divisor: Int = 0

    //obtain the ascii value of each character
    for key in fullname.unicodeScalars {
        divisor += Int(key.value)
    }

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

With any hash algorithm, the aim is to create enough complexity to eliminate "collisions". A collision occurs when different inputs compute to the same hash. With our process, ***Albert Einstein*** and ***Andrew Collins*** will produce the same value (e.g., 8). In computer science, hash algorithms are considered more art than science. As a result, sophisticated functions have the potential to reduce collisions. Regardless, there are many techniques for creating unique hash values.

To handle collisions we'll use a technique called separate chaining. This will allow us to share a common ***index*** by implementing a linked list. With a collision solution in place, let's revisit the ***addWord*** method:

```
//add the value using a 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 bucket
    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
    }
} //end function
```

Dijkstra Algorithm (version 1)

This source code is supporting material for the essay on shortest paths.

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

    var frontier: Array = Array()
    var finalPaths: Array = Array()

    //use source edges to 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)
    }

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

    while(frontier.count != 0) {

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

        var x: Int = 0
        var pathIndex: Int = 0

        for (x = 0; x < frontier.count; x++) {
            var itemPath: Path = frontier[x]

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

        //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

    //establish the shortest path as an optional
    var shortestPath: Path! = Path()

    for itemPath in finalPaths {
        if (itemPath.destination.key == destination.key) {
            if (shortestPath.total == nil) || (itemPath.total < shortestPath.total) {
                shortestPath = itemPath
            }
        }
    }
```

```
    return shortestPath
}
```

Dijkstra Algorithm (version 2)

This source code is supporting material for the essay on binary heaps.

```
//an optimized version of Dijkstra's shortest path algorithm
func processDijkstraWithHeap(source: Vertex, destination: Vertex) -> Path! {
    var frontier: PathHeap = PathHeap()
    var finalPaths: PathHeap = PathHeap()

    //use source edges to 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.enQueue(newPath)
    }

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

    while(frontier.count != 0) {
        //use the greedy approach to obtain the best path
        bestPath = Path()
        bestPath = frontier.peek()

        //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.enQueue(newPath)#sthash.TL5TqZnP.dpuf
        }

        //preserve the bestPaths that match destination
        if (bestPath.destination.key == destination.key) {
            finalPaths.enQueue(bestPath)
        }

        //remove the bestPath from the frontier
        frontier.deQueue()

    } //end while

    //obtain the shortest path from the heap
    var shortestPath: Path! = Path()
    shortestPath = finalPaths.peek()

    return shortestPath

}
```