Intro To Data Structures and Algorithms

What Are They?

- A **data structure** is an arrangement of data.
- Data structures include array, dictionary, linked list, stack, tree, graph, hash table, among others.
- An **algorithm** is a recipe (pattern) for manipulating the data in these structures.
- For instance, an algorithm can be used to insert item(s) into a structure, search for a particular item, sort item(s), or delete items(s).
- Think of data structures as containers, and algorithms as recipes for acting on the elements inside these containers.

Why Learn Them?

* Job interviews!
* Conceptual building blocks of software.
* They are often the basis for OO design patterns.
* They recur everywhere in iOS development and elsewhere.
* Possibly useful for solving tough problems.
* Good practice.

* Give you a common language for talking with other devs.
* Knowledge of them may be assumed by teams/employers.
* They are fun/interesting.

"Big O" Notation

- Shorthand way to say how efficient a computer algorithm is for a given operation.
- Efficiency is only one factor in chosing an algorithm.
- Some other factors are:
  - Memory consumption.
  - Ease of implementation.
- *n* in Big O refers to the number of data items you're processing. So, if we're sorting 10 items *n* = 10.
- Big O is usually talking about the *worst case*, unless otherwise specified.

0(1)
constant.

- Always takes the same amount of time no matter how big the data structure gets.
- Best.
- eg. Looking up an array element by index.

O(log n)
logarithmic.

- Halves the amount of data with every iteration.

- 100 items take 7 steps.

100 / 2
50/2
25/2
12.5/2
6.25/2
3.125/2
1.5625/2

- 1000 take 10 steps.

1000 / 2
500/2
250/2
125/2
62.5/2
31.25/2
15.625/2
7.8125/2
3.90625/2
1.953125/2

- eg. Binary search.
- Good.

```swift
func logarithmic(numItems:Int) -> Int {
  var items = numItems
  var counter = 0
  while items > 0 {
    items /= 2
    counter += 1
  }
  return counter
}

let result = logarithmic(numItems: 1000)
```

O(n)
linear.

- 100 items take 100 steps. e.g. 200 items take 200 steps.
- eg. Sequential search. `For` loops are linear time.

```
for (i in 0...100) {
    print(i)
}
```

O(n log n)
linearithmic.

- Product of n * log n.
- 100 items would be:
  - log n of 100 is 7
  - 7 * 100 = 700
- The fastest general purpose sorting algorithms are linearithmic.

O(n^2)
quadratic.

- 100 items takes 100 ^ 2 or 10,000 steps.
- Doubling the items increases the steps 4x.
- eg. Algorithms using nested loops such as "insertion sort".
- Somewhat Slow.

O(n^3)
cubic.

- 100 items takes 100 ^ 3 or 1,000,000 steps.

- Doubling it makes it 8x slower.
- 3 nested loops.
- Poor.

O(2^n)
exponential.

- 10 items would take 1024 steps.
- Double the items to 20 would take 1,048,576 steps.
- eg. Traveling salesman problem.
- Very poor.

O(n!)
factorial.

- Incredibly slow.
  eg. 5! = 5 * 4 * 3 * 2 * 1 // 120
  eg. 10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 // 3,628,800

```swift
// For Loop implementation

func factorial(num: UInt) -> UInt {
  var result: UInt = 1
  if num == 0 {
    return result
  }
  for i in 1...num {
    result *= i
  }
  return result
}
```

```

factorial(num: 0)
```

```
// Recursive implementation

func factorial(num: UInt) -> UInt {
  if num == 1 {
    return num
  }
  var result = num * factorial(num: num - 1 )
  return result
}

factorial(num: 5)
```

Questions

What is Big O for:

1) Looping though an array?

2) Accessing an item in an array?

3) Accessing a dictionary item?

4) Adding an array item to the end of an array?

5) Adding an array item to the beginning of an array?

TODO On Your Own:
Write a function in a playground that works like the

factorial but adds the numbers instead of multiplying. Try to do it without looking at the factorial solution. Do it using a loop and using recursion.

http://bigocheatsheet.com

Some Basic Data Structures

- Computers work with all kinds of data in large volumes, such as text, video, audio, etc.
- How we store, organize, group and act on this data matters.
- Eg. Logging into Facebook could take a very long time if the database didn't organize in a sorted list and if we didn't use search algorithms on this sorted list.
- A data structure is a way to store and organize data so that it can be used efficiently.
- We can study data structures either as *abstract data types* (ADT), or concretely as implementations in a particular language.
- When studying ADTs we will look at the logical structure, possibly operations, the cost of operations (using Big O), and their implementation.

1. Arrays
2. Linked List
3. Stack
4. Queue

5. Tree
6. *Graph

Array/List

Static List:

- Stores a given # of elements of a particular data type.
- The computer stores lists in a contiguous memory block with a pointer to the starting address.
- Allows us to read and modify elements at a particular position/index.

```
class Person {
  var name: String
  init(name: String) {
    self.name = name
  }
}

let a = [Person(name:"a"), Person(name: "b") ]
a.capacity // 2
a[0].name = "fred" // modify
a[0].name // fred // read
```

Question:
Isn't `a` a let. How come we can modify it then?

But We Want a Dynamic List:

- Read/modify at index. (same as static list)
- Append to the tail.
- Insert at an index.
- Remove at an index.

Could be implemented using the list/array structure.

Swift's mutable array is implemented using this technique. (Notice printing the memory address shows us when the system allocates a brand new array and copies the values over)

```
var a = [Person]()
a.capacity // 0
print(Unmanaged<AnyObject>.passUnretained(a as AnyObject).toOpaque())
a.append(Person(name: "first"))
a.capacity // 2
print(Unmanaged<AnyObject>.passUnretained(a as AnyObject).toOpaque())
a.append(Person(name: "second"))
a.capacity // 2
print(Unmanaged<AnyObject>.passUnretained(a as AnyObject).toOpaque())
a.append(Person(name: "third"))
a.capacity // 4
print(Unmanaged<AnyObject>.passUnretained(a as AnyObject).toOpaque())
```

What is the time complexity of:

- Removing an element from the end?
- Removing an element at an index other than count-1?
- Appending an element?
- Inserting an element?

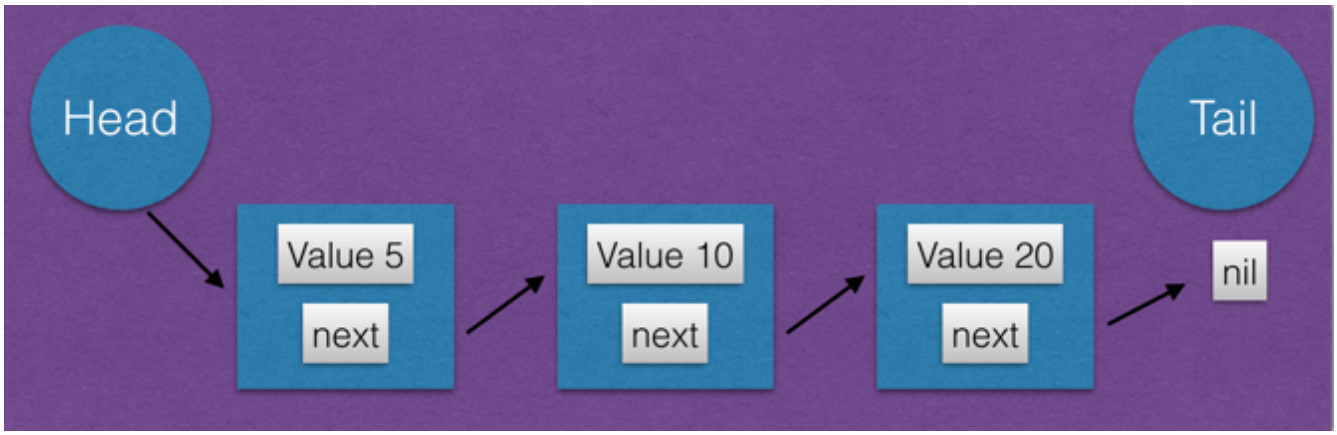
You could start by declaring a large enough list so that the system doesn't have to reallocate and copy every time it outgrows the list's size.

```
let kMaxSize = 100
var a = [Person]()
a.capacity // 0
a.reserveCapacity(kMaxSize)
a.capacity // 100
a.append(Person(name: "vv"))
a[0].name // vv
```

Main Problem:

- Dynamic lists are very memory inefficienct.
- Have to decide size of the array ahead of time.
- Could make a very large array, but this means memory might be sitting there unused.
- Also, if we do need to expand the array we have a very expensive operation, since we have allocate memory, and then copy each element over O(n) + allocation.

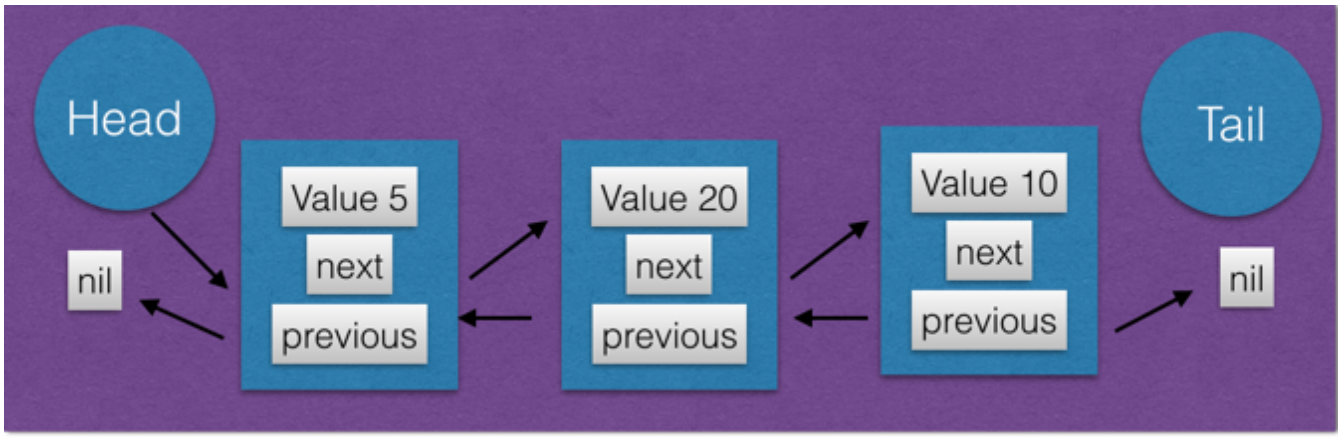
Singly Linked List



- An answer to fixed sized array's problems.
- Unlike plain Lists it doesn't store the data in a continuous memory block.
- Each element has a reference to its next element.
- So, each element will store its data *plus* a reference (link) to the next element.
- nil or 0 can be used to indicate that there is no linked element, which means we are at the end or *tail* of the collection.
- First node is called the *head*.
- We identity the whole structure by the reference of

the head node.

- It's possible to have a doubly linked list where each node has a link to its previous one as well as the next one.



Linked List Swift Implementation

```swift
class Node<T> {
  var value:T
  var next: Node<T>?
  init(value: T) {
    self.value = value
  }
}

class LinkedList<T>: CustomStringConvertible {
  var head: Node<T>?

  var tail: Node<T>? {
    if head == nil {
      return nil
    }
    var lastItem: Node<T>? = head
    while (lastItem?.next != nil) {
      lastItem = lastItem?.next
    }
    return lastItem
  }

  func append(element: Node<T>) {
    if self.tail == nil {
      head = element
    } else {
      self.tail?.next = element
```

```
    }
  }

  var description: String {
    var output = ""
    if head == nil {
      return output
    }

    var currentNode = head
    repeat {
      output += "\(currentNode!.value)\n"
      currentNode = currentNode?.next
    } while currentNode != nil
    return output
  }
}

let node1 = Node<String>(value: "hello")
node1.value

let linkedList = LinkedList<String>()
linkedList.head = node1
linkedList.tail?.value

let node2 = Node<String>(value: "world")
linkedList.append(element: node2)
print(linkedList)

```

Time Complexity Of Linked List:

- Traversal?
- Insertion?
- Deletion?
- Accessing an Item?

TODO On Your Own

- Write a method to get an item at an index.
- Write a method to insert an it item at index.
- Write a method to delete an item at an index.
- Write a method to remove all items.

References

- https://www.youtube.com/watch?v=NobHlGUjV3g
- https://www.raywenderlich.com/144083/swift-algorithm-club-swift-linked-list-data-structure

Stack:

- Like a list but with more limited functionality.
- Very useful structure.
- Items added and removed from the same end (top).
- *push()* adds new element to top.

- *pop()* removes the top element.
- *peek()* allows you to view the top element without popping it.
- **LIFO** (*last in first out*).
- Last item pushed is the next item popped.

Applications

- UINavigation controller is a classic stack.
- Computer's memory is organized into an area that is an implementation of the stack, called the stack. (Stackoverflow is not just a website, but something that happens when we overrun the space allocated for stack memory).
- Undo stack.

```swift
// Swift Stack Implementation Using an Array

struct Stack<T> {
  private var elements:[T] = []
  mutating func push(_ item:T) {
    elements.append(item)
  }
  mutating func pop() -> T? {
    return elements.count > 0 ? elements.removeLast() : nil
  }
  func peek()->T? {
    return elements.count > 0 ? elements.last : nil
  }
}

var stack = Stack<Int>()
stack.push(10)
stack.push(11)
let pop1 = stack.pop()
pop1 // Optional(10)
let pop2 = stack.pop()
pop2 // Optional(11)
let pop3 = stack.pop()
pop3 // nil

```

- Write a method `isEmpty` that checks to see if the stack is empty.

Question:
What is the big O of a stack?

Queue

- Similar to a stack, except that you put items into one end and remove them from the other.
- **FIFO** (first in, first out)
- Used to model real-world queues like bank lines, airplanes waiting to take off, or data packets waiting to be transmitted over the Internet.
- Can you think of any iOS examples?


```
// Queue Implementation Using Array

struct Queue<T> {

  private var _data:[T] = []
```

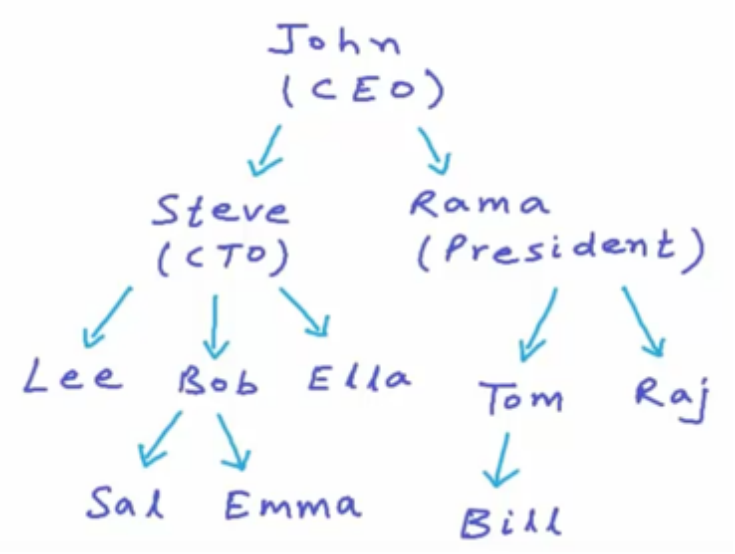
```swift

  // adds
  mutating func enqueue(_ element:T) {
    _data.append(element)
  }

  // removes
  mutating func dequeue()-> T? {
    return _data.count > 0 ? _data.removeFirst() : nil
  }
}

var queue = Queue<Int>()
queue.enqueue(12)
queue.enqueue(22)

var item1 = queue.dequeue()
item1 // 12
var item2 = queue.dequeue()
item2 // 22
var item3 = queue.dequeue()
item3 // nil
```

Question:

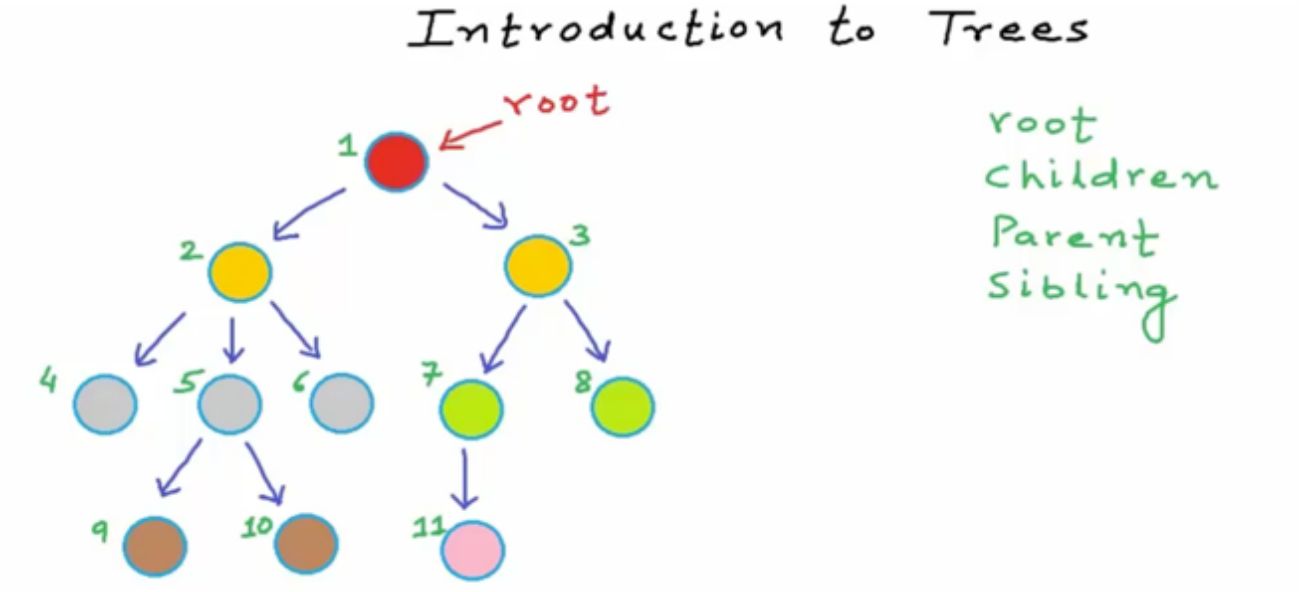
- What is the Big O of a Queue?

Tree

- For modeling hierarchical data.
- Similar to a linked list, but more complex.

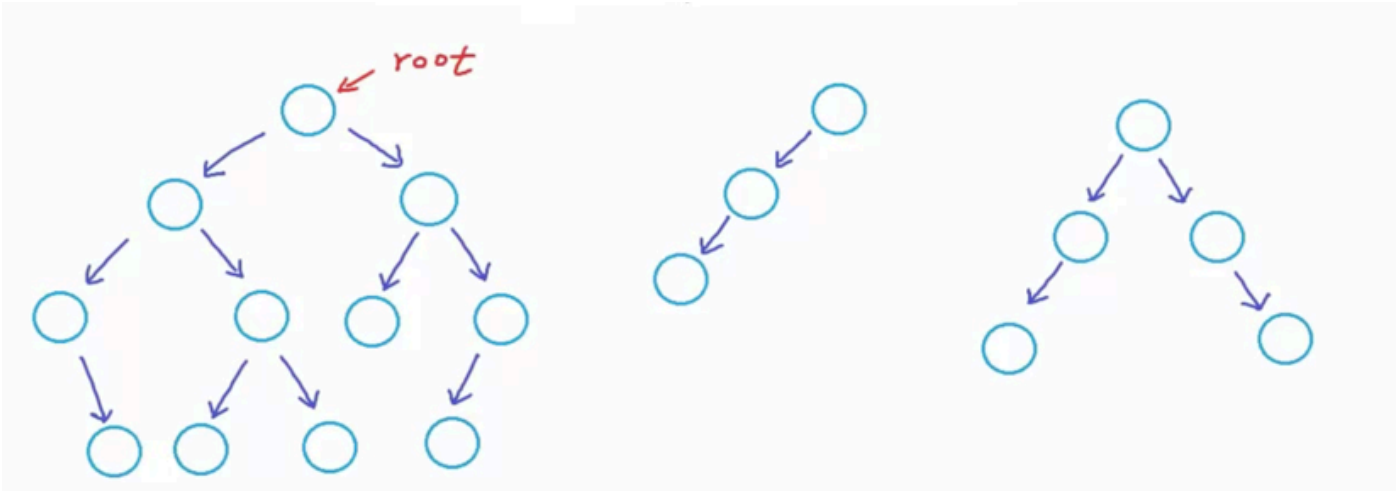


- Consists of nodes linked together in a hierarchy starting at the root node.
- Efficient structure for searching, inserting and deleting.
- Any node without a child is called a **leaf** node.
- All nodes except the root node have a parent.
- There is only 1 root node.
- We can only walk in one direction!



- A binay tree is a *subtype* of the tree structure.
- Binary tree is a tree in which each node can have at

most 2 children.

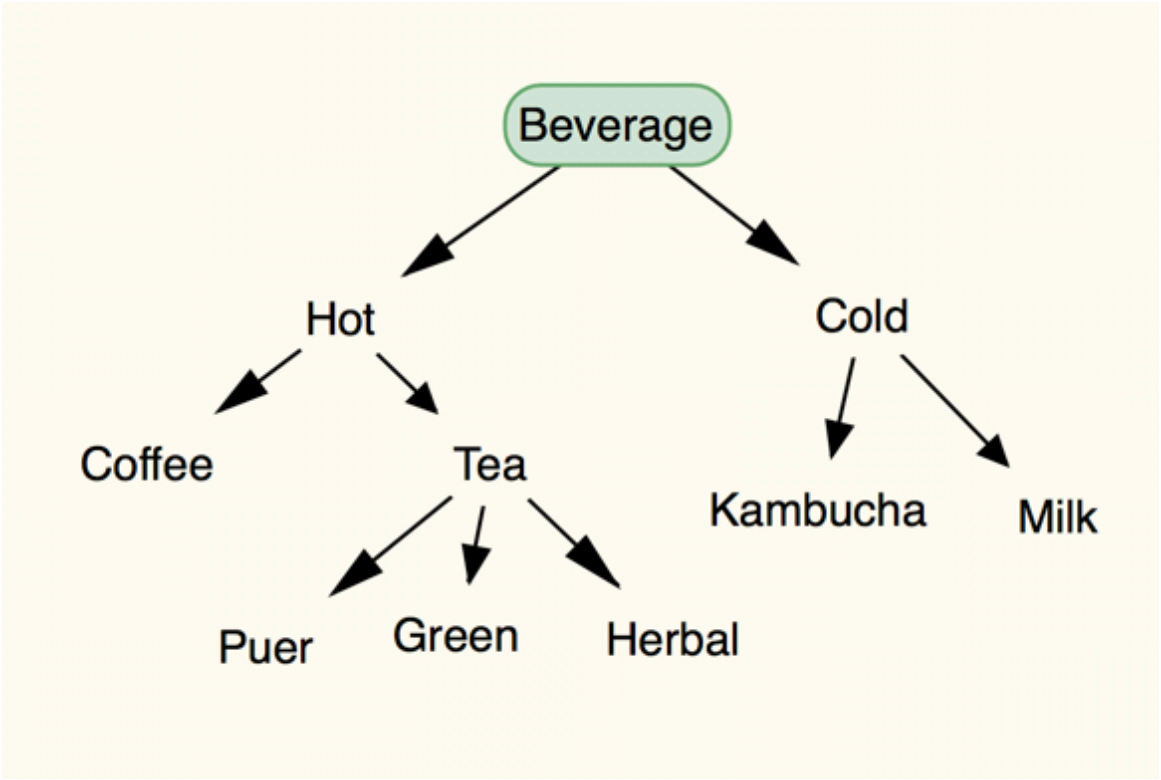


- Where do you find the tree structure in iOS?

Swift General Purpose Tree Implementation

```swift
class Node {
  var data: String

  var children: [Node] = []
  weak var parent: Node?

  init(data:String) {
    self.data = data
  }

  func add(node: Node) {
    children.append(node)
    node.parent = self
  }

  func search(for data: String) -> Node? {
    print(#line, self.data)
    if self.data == data {
      return self
    }
    for child in children {
      if let found = child.search(for: data) {
        return found
      }
    }
    return nil
  }
}

```

```swift
let beverage = Node(data: "Beverage")

let hot = Node(data: "Hot")
beverage.add(node: hot)
let coffee = Node(data: "Coffee")
let tea = Node(data: "Tea")
hot.add(node: coffee)
hot.add(node: tea)
let puer = Node(data: "Puer")
let green = Node(data: "Green")
let herbal = Node(data: "Herbal")
tea.add(node: puer)
tea.add(node: green)
tea.add(node: herbal)
let mint = Node(data: "Mint")
herbal.add(node: mint)

let cold = Node(data: "Cold")
beverage.add(node: cold)

let kambucha = Node(data: "Kambucha")
let milk = Node(data: "Milk")
cold.add(node: kambucha)
cold.add(node: milk)

let result = beverage.search(for: "Milk")
print(#line, result?.data ?? "nothing")

```



ToDo On Your Own:

- Implement deletion.
- Implement move.

Resources:

- https://www.youtube.com/watch?v=qH6yxkw0u78
- There are many kinds of tree structures. Please see the links at the bottom of this article: https://www.raywenderlich.com/138190/swift-algorithm-club-swift-tree-data-structure

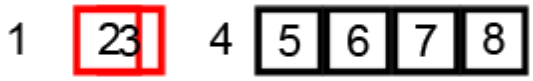
Some Sort Algorithms

- Each data type is going to have particular algorithms for inserting, deleting, sorting, and searching for elements.
- Efficient sorting and searching of elements is one of the primary things that make computers useful.
- Email, for instance, is basically a sorted list of messages.
- Google's search algorithm is a way of sorting a huge list of data.
- When developing for iOS you will rarely have to figure out your own sort algorithm, but it's still useful to

know.

Bubble Sort

- Simple sort.
- Inefficient sort algorithm.
- Can be used if a list is nearly sorted and has a few elements out of order.



Iterative Implementation

```swift
func bubbleSort(with array: [Int]) -> [Int] {
  if array.count < 2 {
    return array
  }
  var array = array
  for i in 1..<array.count {
    for j in 0..<array.count-i {
      if array[j] > array[j+1] {
        let temp = array[j]
        array[j] = array[j+1]
        array[j+1] = temp
      }
    }
  }
  return array
}
```

Efficiency

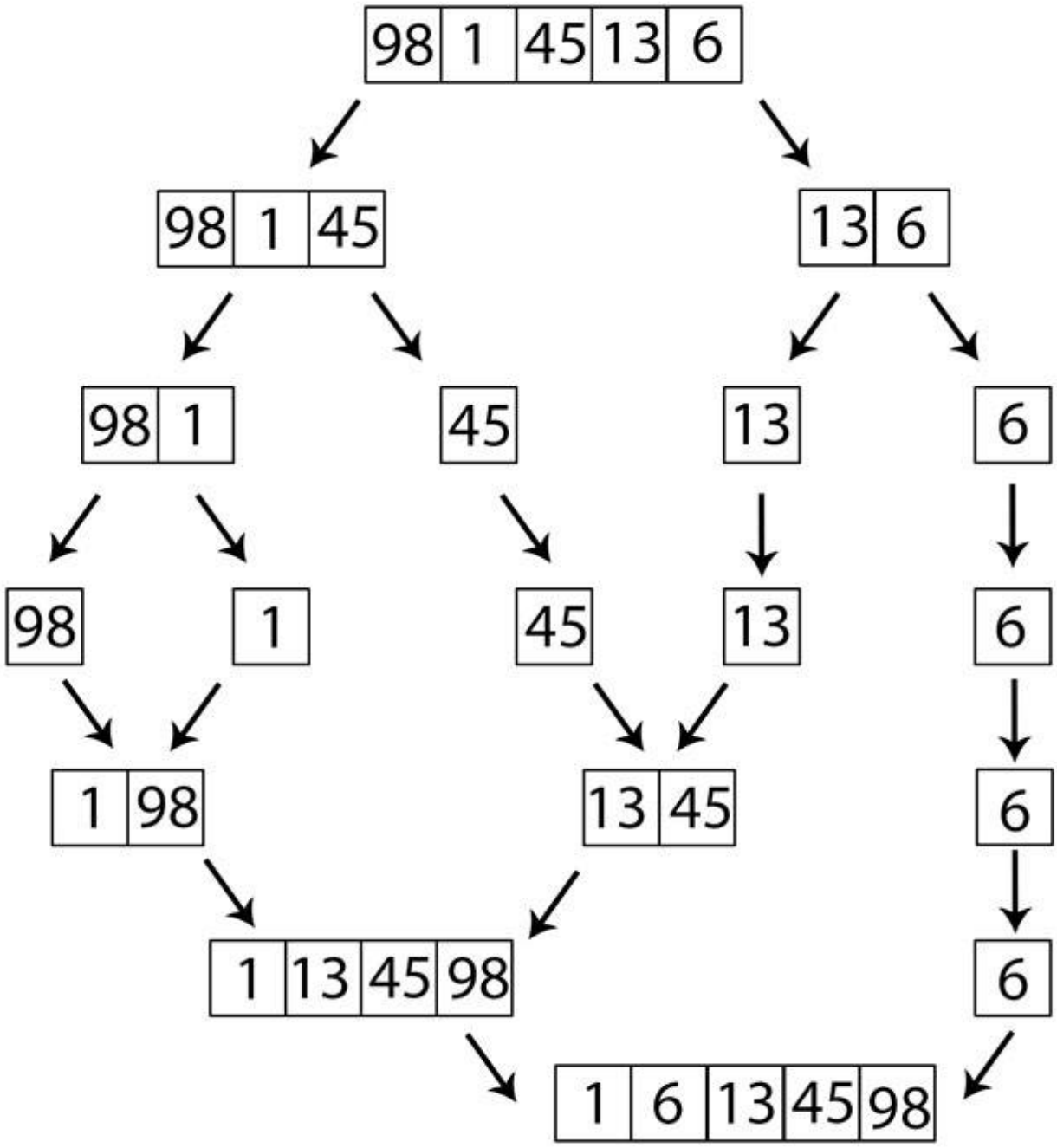
O(n^2)

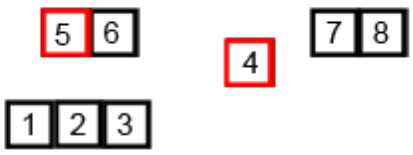
References:

- https://en.wikipedia.org/wiki/Bubble_sort
- https://ca.wikipedia.org/wiki/Fitxer:Bubble-sort-example-300px.gif

Merge Sort

- General purpose sort, considered one of the most efficient for large data sets.
- It's divide and conquor algorithm.






```swift
func merge(left:[Int],right:[Int]) -> [Int] {
  var mergedList = [Int]()
```

```
  var left = left
  var right = right

  while left.count > 0 && right.count > 0 {
    if left.first! < right.first! {
      mergedList.append(left.removeFirst())
    } else {
      mergedList.append(right.removeFirst())
    }
  }

  return mergedList + left + right
}

func mergeSort(list:[Int]) -> [Int] {
  guard list.count > 1 else {
    return list
  }

  let leftList = Array(list[0..<list.count/2])
  let rightList = Array(list[list.count/2..<list.count])

  return merge(left: mergeSort(list:leftList), right: mergeSort(list:rightList))
}

mergeSort(list: [98,1,45,13,6]) // [1, 6, 13, 45, 98]
```

Efficiency:

O(n log n)

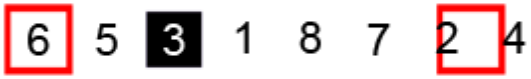
References:

- https://commons.wikimedia.org/wiki/File:Merge-sort-example-300px.gif
- https://en.wikipedia.org/wiki/Merge_sort
- https://www.youtube.com/watch?v=EeQ8pwjQxTM
- http://www.thomashanning.com/merge-sort-in-swift/

QuickSort

- Take a random pivot.

- Move all elements so that everything to the right of the pivot is greater than the pivot and everything on the left is less.
- Repeat algorithm on left and right subarrays.
- Most common sort used by language libraries.

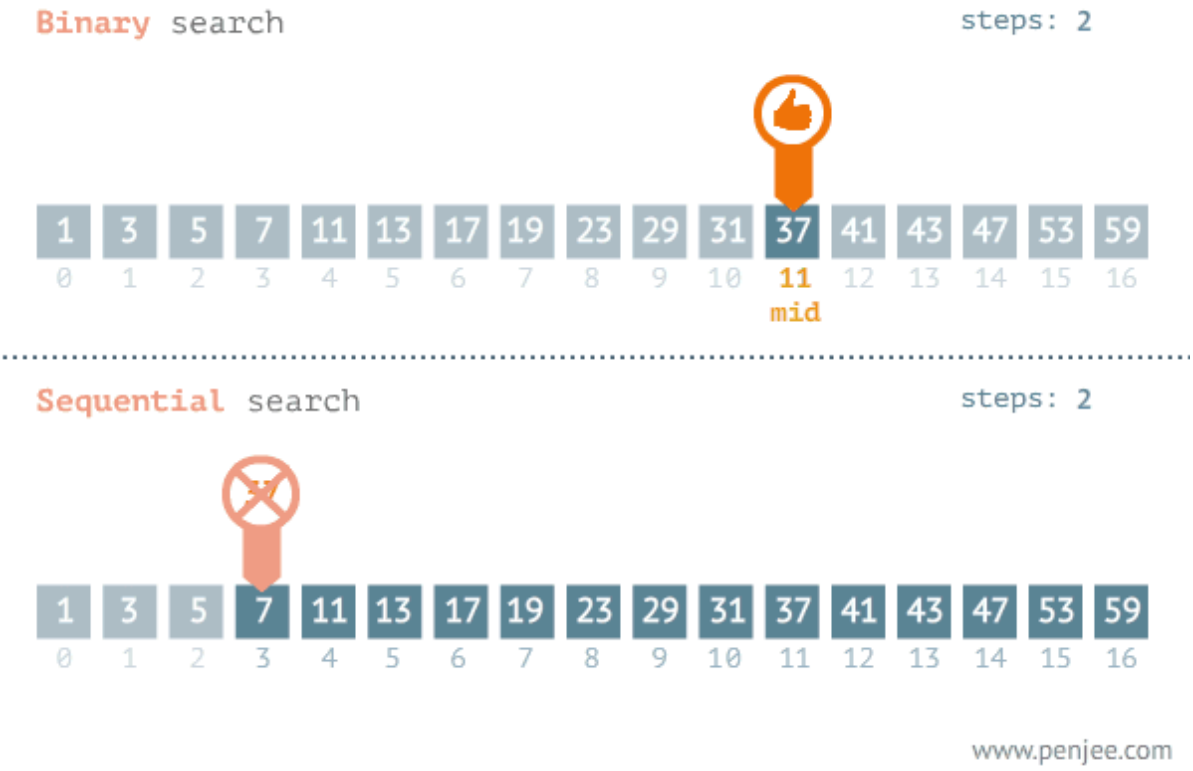


Efficiency
O(n^2) in worst case, but can act like O(n log n) most of the time. It's an in place sort unlike merge sort.

References:

- https://commons.wikimedia.org/wiki/File:Quicksort-example.gif
- Hungarian Dancers Doing Quick Sort! https://www.youtube.com/watch?v=ywWBy6J5gz8
- https://www.youtube.com/watch?v=XE4VP_8Y0BU

Binary Search Vs Sequential/Linear Search



- Image you are trying to find Home Simpson in the phone book. In linear search you would start at the beginning of the phone book and check each person until you reached Simpson, Homer!
- Binary search on the other hand would jump part way through the phone book, maybe near the end since Simpson is late in the alphabet.
- If Simpson were past the page then it would omit all the pages before. Dealing with the pages toward the end only we could pick a page half way from our current page to the end.
- If we go too far then we could omit all the pages past the current one and only deal with the section still not searched.
- We repeat this process until we location Simpson, Homer or can't find him.
- Binary search only works with sorted items.

```swift
func binarySearch(an array: [Int], for value: Int) -> Int? {

  var leftIndex = 0

  var rightIndex = array.count - 1

  while leftIndex <= rightIndex {

    let midIndex = (leftIndex + rightIndex) / 2
    let midValue = array[midIndex]

    if midValue == value {
      return midIndex
    }
    if value > midValue {
      leftIndex = midIndex + 1
    }

    else if value < midValue {
      rightIndex = midIndex
    }
  }
  return nil
}
let array = [1,7,9,11,12,15,19]
let value = 19
let result = binarySearch(an: array, for: value)
result
```

Recursive Way

```swift
func binarySearch(with array:[Int], start: Int, end: Int, value: Int) -> Int? {
  var startIndex = start
  var endIndex = end
  while startIndex < endIndex {
    let midIndex = (startIndex + endIndex)/2
    let midValue = array[midIndex]
    if midValue == value {
      return midIndex
    } else if value < midValue  {
        endIndex = midIndex
      binarySearch(array: array, start: startIndex, end: endIndex, value: value)
    } else if value > midValue {
      startIndex = midIndex + 1
      binarySearch(array: array, start: startIndex, end: endIndex, value: value)
    }
  }
  return nil
}

```

```
let array2 = [1,7,9,11,12,15,19]
let value2 = 15
let result2 = binarySearch(with: array2, start: 0, end: array2.count, value: value2)
result2

```

References

- https://blog.penjee.com/binary-vs-linear-search-animated-gifs/
- https://en.wikipedia.org/wiki/Binary_search_algorithm