Algorithm Notes

* Big O
  + Performance of an algorithm
  + Worst case scenario
* O(1)
  + Only one operation occurs, constant amount of time
  + Constant time is the key term here
* O(n)
  + Increases based on the number of inputs
  + All that matters is if the growth is linear, you can ignore constant time addition or additional loops outside of the other loop (no embedded loops)
* O(n^2)
  + Quadratic
* O(logn)
  + Logarithmic
  + As input goes on it slows down even more
  + Binary Search
* O(2^n)
  + Exponential growth
* Space complexity
  + When we need to consider the amount of space an algorithm requires
  + How much extra space we need for the algorithm
* Arrays
  + Stores list of items
  + Stored sequentially in memory
  + Lookup: O(1)
  + Static, you need to allocate a specific size
  + Increasing the length of the array can be expensive
  + Insert: O(n)
  + Delete: O(n)
* Linked lists
  + Nodes that have references to the next
  + Starts with a head and moves until the reference is null
  + Lookup by value O(n)
  + By Index O(n)
  + May need to have a tail as a reference as well
  + Insert
    - Tail O(1)
    - Head O(1)
    - Anywhere else O(n)
  + Delete
    - Tail O(n)
    - Head O(1)
    - Anywhere else O(n)
* Stacks
  + FILO
    - First in last out
  + Operations:
    - Push(item)
      * O(1)
    - Pop()
      * O(1)
    - Peek()
      * O(1)
    - isEmpty()
      * O(1)
* Queues
  + Similar to stack except its FIFO
  + First in first out
  + Examples of queues
    - Printers
    - Operating systems
    - Web servers
    - Live support systems
  + Operations
    - Enqueue
    - Dequeue
    - Peek
    - isEmpty
    - isFull
  + All run in O(1)
    - Because they operate on the ends
* Hash Maps
  + A key is to find the index in an array
  + A simple way is to get the numerical value of the key and modulus the value so we know its within the size of the array
  + You have issues sometimes involving collisions, when two different values return the same index
  + You can overcome these collisions in different ways
    - Popular one is to use a linked list in the list instead and chain from there
    - Another is to just step over until you get to the next available index
* Trees
  + Trees can be different depending on the number of children they have etc
  + Representing hierarchical data
  + Databases use trees to index
  + Autocompletion will use trees to match previous queries
  + Compilers use tree’s to parse expressions
  + Trees are also used with compression algorithms
  + Operations on a binary tree
    - Lookup O(log n)
    - Insert O (log n)
    - Delete O (log n)
  + Breadth first
    - Level order traversal
    - Visit all nodes at the same level before moving on
  + Depth first
    - Pre order
      * Root, Left, Right
    - In Order
      * Left, Root, Right
    - Post order
      * Left, Right, Root
  + Depth
    - Count the edges towards the node you are wanting the depth of
  + Height
    - The length from the node to the furthest leaf
  + Binary tree
    - May not be evenly spread out
  + Binary search tree
    - Will be organized numerically in order
    - Left most leaf will be the smallest
    - Right most leaf will be the largest
* AVL Trees
  + Self balancing tree
  + When you insert or remove a value, it will rebalance itself
  + Will use rotations to balance out the tree