**Languages**

* **Python**
  + High level language
* **C**
  + Medium level language
  + NOT object oriented
* **C++**
  + Medium level language
  + Object oriented
* **MIPS**
  + Low level language
  + Register level

**Data Structures:**

* **Array**
  + Stored sequentially in memory
    - As opposed to a list, which is similar but not stored sequentially
* **Linked List**
  + Singly and doubly
* **Trees**
* **Hash Map**
  + Goal is to find in O(1)
  + Steps
    - Make empty array
      * Make array twice the number of keys and round to nearest prime
      * When they array is 70% full, increase its size
    - Take an input value
    - Evaluates hashing function at that value
    - Result is the “key”
      * Key is the index where the value should be inserted
  + Collisions
    - When one value hashes to the same key index
    - How to avoid?
      * Have a better hashing algorithm
    - What if we can’t avoid?
      * Chaining
        + Make that index a linked list
      * Probing
        + If your key already exists in the array, increment the index and check if that spot is empty

Linear probing

Increment index by 1

Repeat until an empty spot is found

Quadratic probing

Increment index by some quadratic function

Pseudo-random probing

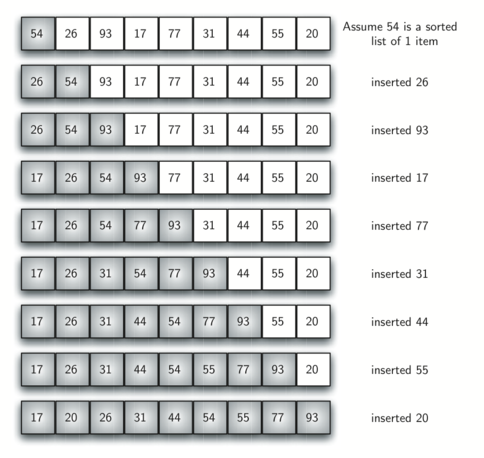
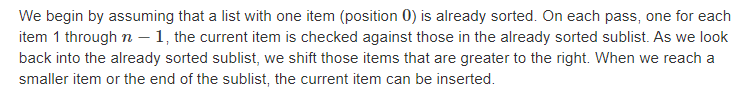
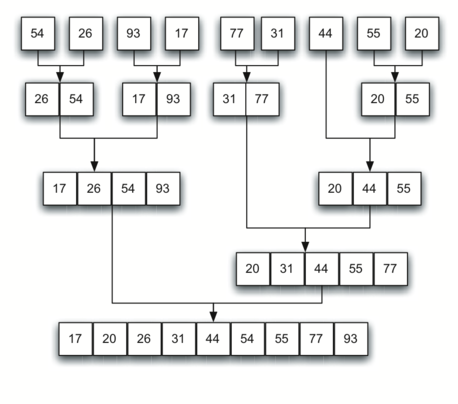
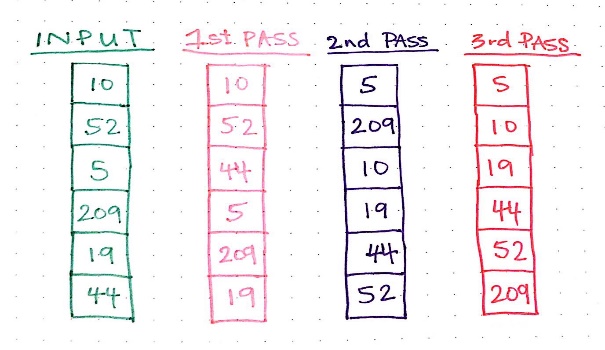
Increment randomly

* + - * Double hashing
        + Have multiple hash functions
        + If there is a collision while using one, use the other

**Sorting Algorithms**

* **Selection Sort**
  + Comparison Sort
    - Runtime is always O(n2)
  + The most basic sorting algorithm

1. Set position to 0
2. Find smallest element in list[pos, end\_of\_list]
3. Swap list[position] with list[smallest\_element]
4. Increment position by 1
5. Repeat until you reach the end of the list
   * Pseudocode:
     + for **i** = 0 to n-2 do // steps 2-6 form a *pass*
     + set **min\_pos** to **i**
     + for **j** = **i**+1 to n-1 do
     + if item at **j** < item at **min\_pos**
     + set **min\_pos** to **j**
     + Exchange item at **min\_pos** with one at **i**

* **Bubble Sort**
  + Comparison Sort
    - Worst case scenario runtime is O(n2)
      * Happens when the list is in reverse order
    - Best case scenario runtime is O(n)
      * Works best if the list is mostly in order
  + Compares adjacent array elements
    - Exchanges their values if they are out of order
  + Larger values bubble up
  + Smaller values bubble down
  + Pseudocode:
    - do
    - Initialize **exchanges** to **false**
    - for each pair of adjacent array elements
    - if values are out of order
    - Exchange the values
    - Set **exchanges** to **true**
    - while **exchanges**
* **Insertion Sort**
  + Comparison Sort
    - Worst case scenario O(n2)
    - Best case scenario O(n)
  + Always maintains a sorted sub-list in the lower portion
  + Based on a technique of card players to arrange a hand
    - Player sorts their hand
    - When they pick up a new card
      * Make room for it
      * Insert it in order
  + Pseudocode:
  + For each element from 2nd to last:
  + ****Insert element where it belongs in first part of list
  + Inserting into the sorted part
  + Increases sorted subarray size by 1
* **Quick Sort**
  + Comparison based sort
    - Worst case runtime O(n2)
      * Happens when list is sorted or almost sorted
      * To improve this, pick a better pivot
    - Best case runtime O(n\*log(n))
  + Good for finding the kth value
  + Pick a pivot value
    - Say the first value in the array
  + Start from the left side at [1] and the right side at [n-1] (n is length of array)
    - Increment left side until you find a value that is greater than the pivot
    - Decrement right side until you find a value less than pivot value
    - Swap these values
    - Continue this until the left and right marker cross paths
      * When this happens, the split point is found, list can be divided
    - Call quicksort recursively on the two halves
* **Merge Sort**
  + Comparison based sort
    - Runtime is always O(n\*log(n))
    - Takes more memory because we need to make temp arrays
  + Takes advantage of the fact that a list of 1 element is always sorted
  + Find middle of array
  + Split in half
    - Do this recursively
  + Sort and merge these smaller arrays to make larger arrays
    - Continue until you have 1 array
* **Bucket Sort**
  + Non-comparison-based sorting algorithm
    - O(n) runtime
  + Create n (# of elements in original array) buckets (empty lists)
  + Place items in buckets
    - Insert arr[i] into bucket[n\*arr[i]]
    - Use linked lists if there are more than 1 element/bucket
  + Sort the buckets
    - Insertion sort
  + Combine the sorted buckets
* **Radix Sort**
  + Non-comparison-based sorting algorithm
    - Runtime is O(n\*k) where k is the digit length
  + Sort numbers based on LSB
    - If a number has the same LSB value, place it in the order it appeared in the original array
      * If no bit, replace with a 0
    - Then sort the next LSB
    - Continue until list is sorted