Part 3 Review Topics

1. Trees
   - What is a tree? What properties does it have?
   - Know and understand the following terminology, in relation to trees:
     - node
     - root
     - leaf
     - left child / right child
     - parent
     - descendant
     - ancestor
     - siblings
     - subtrees
     - depth (with root as 0)
     - height (empty tree as 0)
   - What is a complete binary tree? Be able to identify a tree as such.
   - What is a full binary tree? Be able to identify a tree as such.
   - What is a binary search tree? What properties does it have?
   - What operations in a binary search tree can happen in logarithmic time?
     - and logarithmic relative to what? The number of nodes in the tree, or...?
   - Understand the sequence of operations that yields a worst-case BST (essentially a linked list)
     - be able to explain why self-balancing trees are useful
   - How can an array be used to represent a complete binary tree?
     - what is the equation to find the left child of an element at index *i*?
     - what is the equation to find the right child of an element at index *i*?
     - what is the equation to find the parent of an element at index *i*?
     - what issue arrises when using an array to represent a binary tree that is *not* complete?
   - How can a tree be represented with node structures?
     - how is an empty tree represented?
     - how can you tell if any give node is a leaf node?
   - How do you visit each node in a tree via:
     - inorder traversal
     - preorder traversal
     - postorder traversal?
   - Given some tree, be able to determine which order each node will be visited and processed
   - Be able to write recursive functions to process a BST (such as inserting or searching)
   - How do you find the minimum and maximum elements in a BST?
   - How can you remove a single value from a BST?
     - understand the algorithm presented in class
   - Understand how functions can be passed as arguments to other functions
     - syntax for templated and non-templated versions

2. Sorting
   - Why is sorting useful?
   - Name at least three operations that can be done in constant time on a sorted sequence of values

- Understand the main ideas behind:
    - selectionsort
    - insertionsort
    - mergesort
    - quicksort
    - heapsort
- For each algorithm, understand:
    - worst-case complexity
    - average-case complexity
    - best-case complexity
    - what input leads to best case performance, and why?
    - what input leads to worst case performance, and why?
- Which algorithm is best for *almost-sorted* inputs, and why?
- Be able to trace the state of an array for both quadratic algorithms
- What is space efficiency?
- Why do the two recursive algorithms not have constant space efficiency?
- Pointer arithmetic
    - syntax (simple adding / subtracting numbers from a pointer)
    - what it does
    - how it can be used
- What is a heap? What properties must it satisfy?
- How can an array be used to represent a heap, or any other complete binary tree?
- What operations in a heap happen in constant time?
- How do you make a heap initially, given some array?
- After removing the root element, what is the time complexity of reheapifying the heap?
- Understand how to use the STL sort functions, which takes iterators in the form of [begin, end)
- How can an array of birthdays (1-366) be sorted in linear time?

3. Searching
- For serial searching:
    - what is the worst-case complexity?
    - what is the best-case complexity?
    - the average-case complexity?
- For a sorted array:
    - how long does finding the minimum or maximum value take?
    - how long does finding any arbitrary value take?
- Binary search
    - understand and be able to explain the process
    - what is the worst- and average-case complexity of binary search?
    - the best-case complexity?
- Be able to explain the concept of using keys to identify values
- Understand the concept of hashing (converting a key into an integer—typically a valid array index)
- What is a collision?
- Understand each of the following schemes for handling collisions:
    - open address hashing

    - double hashing
    - chained hashing
- How can multiple values be stored at a single spot in an array?
- Understand how a hash-table can be implemented as a class using an array
    - why do we need both NEVER_USED and PREVIOUSLY_USED in an open-address hashing table implementation?
- For a hash table, what is the average complexity of:
    - searching for a value
    - inserting a value
    - removing a value
- What is the worst-case complexity for each of these operations?
    - what situation leads to this worst-case complexity?
- How does the complexity of operations on a table depend on its load (how full it is)?
- Are values in a table ordered in any way?