ICS211 Course review

- exam 1 material: Java, ADTs, linked lists, run-time big-O, objects, references and pointers, iterators, invariants
- exam 2 material: generic types and container classes, stacks, queues, recursion binary trees, binary search, binary search trees, tree traversal,
- sorting: insertion, selection, bubble, mergsort, heapsort, quicksort

Java concepts

- recursion:
  - as a replacement for loops
  - as a way of thinking of operations on lists, arrays (e.g. binary search)
  - for tree operations, especially tree traversal
  - recursion with return value to compute and return something
  - recursion with void method to change something
  - or a mix of these two
- references and pointers
  - all variables in Java are either one of the basic types, or
  - a reference (pointer) to an object
  - the reference may be null
  - arguments (of a type other than the basic types) are passed by reference, so
  - if they are modified, the caller can see the changes
- objects
- generic types and parametrized classes
  - generic types are only found as parameters to classes or interfaces
  - the corresponding actual type must be an object type
- object equality
- object comparison
- iterators for collection classes
  - the iterator object must contain enough information to return the contents of the collection class
  - using an iterator is (and is meant to be) easy and convenient
  - implementing an iterator can be hard, e.g. for tree traversal

Data Structures

- arrays
  - constant-time access
  - linear-time resizing
  - n-log-n sorting (mergesort, heapsort)
- stacks
- queues

- both stacks and queues can be implemented using either linked lists or arrays (for queues, circular arrays, that is, index 0 is the index used after array.length - 1)
- priority queues
    - needs some way to compare objects
    - many implementations:
    - heaps (do not maintain order within a priority level)
    - arrays of queues (only work for relatively small number of priorities)
    - ordered linked lists, arrays, etc (linear time element addition)
- array-based lists
    - resizable
    - most operations (other than resizing) take constant time
- linked lists
    - linked list class provides all the linked list operations: add, remove, size, toString, etc
    - node class stores the value and next reference (recursive class, has a class variable of the same type as the class)
    - the linked list object keeps a reference to the first node of a linked list
    - operations at the head (and at the tail, if a tail pointer is kept) take constant time
    - most other operations take linear time
- ordered linked lists
- trees
    - root, child, parent, sibling, etc
    - tree nodes have 0 or more children
    - tree traversals: prefix, postfix, and, only for binary trees, infix
    - most tree algorithms are well-suited to recursive implementation
- binary trees
    - logarithmic depth if the tree is balanced, for example, a heap
    - otherwise, the worst case is linear depth
    - many operations take time O(depth)
    - other operations, e.g. tree traversal, take time O(nodes)
- heaps
    - a complete binary tree stored in an array
    - heap property: each parent is greater (less) than either of its children
    - when adding, add at the end (bottom) of the heap, then re-establish the heap property moving up the tree
    - when removing, move the element at the end (bottom) of the heap to the top, then re-establish the heap property moving down the tree
- huffman trees and huffman coding
- hash tables
    - constant-time access to keyed data
    - hash function returns an int which is used as an index
    - pseudo-randomness (hash function) is used to distribute data evenly
    - different ways of handling collisions: increase array size, open addressing, chained hashing, separate storage

Algorithms

- linked list operations: add, remove, search
- tree operations: add, remove, search
- ordered list insertion
- binary search
- heap insert and remove
- huffman coding
- prefix-to-infix-to-postfix and viceversa using stacks or using trees
- solution of different problems using priority queues (e.g. for huffman coding)

Hashing Algorithms

- hash functions: adding/XORing contributions from significant elements
- chained hashing (array of linked lists)
- open addressing: linear probing, quadratic probing, double hashing

Sorting Algorithms

- selection sort -- $O(n^2)$
- bubble sort -- $O(n^2)$, but $O(n)$ if array is sorted
- insertion sort -- $O(n^2)$, but $O(n)$ if the elements are at most a constant distance away from their correct position
- quick sort -- $O(n^2)$, but $O(n \log n)$ if the pivot is random and splits the array about evenly
- merge sort -- $O(n \log n)$
- heap sort -- $O(n \log n)$

Run-time Analysis

- a few typical functions:
    - $O(\log n)$: binary search
    - O(tree depth): going from the root to a leaf or from a leaf to a root: sometimes $O(\log n)$, otherwise $O(n)$
    - $O(n)$: traversing a data structure once, e.g. linked list removal, array insertion or expansion, linear search, best case for bubble sort and insertion sort
    - $O(n \log n)$: efficient sorting such as heap sort and merge sort, best case for quicksort
    - $O(n^2)$: most sorting algorithms, including worst case for insertion sort or quick sort, best and worst case for selection sort, worst and average case for bubble sort
- look at the loops and the changes in the loop variables
- might also have to consider big-O memory space usage

Concepts

- ADTs, and their representation by classes
- run-time analysis to determine big-O
- tree traversal
- invariants
- many different implementations possible for one interface, e.g. queues, stacks

Programming

- decompose the problem into reasonable elements
- use a class or method to implement each element
- combine these into a solution
- can analyze the solution for efficiency
- knowing data structures can help in decomposing the problem
- knowing algorithms can help if the problem is similar to one solved by an existing algorithm, or if the problem can be partially solved by an existing algorithm