# Final Exam Study Guide

Handout 12
CSCI 136: Fall 2018
December 6

---

Your final will be a "closed book" exam. The exam will be held on Monday, December 17 in TPL 203 at 9:30 am. You will have 2.5 hours to complete the exam.

You are responsible for anything we covered in class or in lab, everything in the assigned reading from *Java Structures*, and the handouts/problem sets. The exam is cumulative, but it will heavily weight topics from the second half of the course. However, we used arrays, Vectors, and Lists to implement many data structures; we used big-O notation to evaluate and compare data structures; we used recursion to traverse trees; we used induction to prove properties of data structures and algorithms; and so on. The second half of the semester built heavily on previous topics.

The following non-exhaustive list may be helpful in reminding you about some of the key topics we have covered:

- **Pre-Midterm**
  - Java syntax, as we have used it in our programming assignments.
  - Classes, abstract classes, and interfaces and their respective roles.
  - Information hiding (abstraction) and why it's good.
  - Extending classes with inheritance.
  - Generic classes and their use.
  - Pre- and post-conditions, and assertions.
  - The meaning of `static` (and non-static) as applied to variables and methods.
  - Vector, its implementation in the structure5 package, and its methods.
  - Complexity: Big "O" definition.
    - Determining the asymptotic behavior of mathematical functions.
    - Determining the time and space complexity for a given algorithm.
    - Worst and best case analysis.
  - Linear and binary search.
  - Recursion and induction.
  - Sorting.
    - Bubble sort, selection sort, insertion sort, merge sort, quicksort (and heapsort post-midterm).
    - Using `Comparator/Comparable` for sorting.
  - Linked lists: Singly, Doubly, and Circularly linked lists
- **Post-midterm**
  - Stacks (LIFO)
    - List and Vector implementations
    - Relationship with recursion and graph/tree traversals (DFS)
  - Queues (FIFO)
    - List, Vector, and fixed-size array implementations
    - Relationship to graph/tree traversal strategies (BFS)

(continued on the flip side...)

- Priority queues
  - OrderedVector Implementation
  - Heap implementation
    - heap property
    - array representation and tree fullness/completeness
    - heap insert/remove
- Trees
  - Array/Vector-based representation
  - Recursively-defined, link-based representation
  - Binary search trees
    - Binary search tree property
  - Traversing trees:In-order, post-order (depth-first), pre-order; level-order (breadth-first)
  - Tree balancing schemes (AVL, Red-Black)
- Iterators (including Iterators based on other iterators)
- Bitwise operations
- Graphs
  - Directed/undirected
  - Weighted/unweighted
  - Adjacency List representation
  - Adjacency Matrix representation
  - Reachability/traversal (Breadth-first, depth-first)
  - Minimum cost spanning trees and Prim's algorithm
  - Single source shortest paths and Dijkstras algorithm
- Hashtables
  - Using a hashing function to find an object's "bin"
  - Open Addressing vs External Chaining
  - Managing collisions during probing
  - Hash table load factor

**Tips on Preparing for the Exam**

Our goal is to test concepts, so it is *not* important to memorize the exact code or method signatures for every data structure. Descriptive variable/method names are enough to demonstrate understanding. However, it *is* important to know the types of operations that different data structures do/do not support and how efficient (in the Big-O) sense these methods are for the various implementations that we have studied. For example, we cannot access arbitrary elements in a queue: We can only add to the back and remove from the front. And the time required to add to/remove from a queue depends on the implementation (e.g., if a queue is implemented using a Vector, dequeueing involves removing from the front of the Vector is O(n). Using a circularly linked list, we can add to the tail and remove from the head in O(1) time.)

If you understand how each data structure is implemented, then you should be in a good position to figure out the space required to store the structure as well as the time complexities of its methods. This is a much better approach than trying to memorize run-times of all of the methods of all of the data structures—it is also much more valuable to you in the future. Similarly, understanding how an algorithm works, for example Prim's algorithm, is much more valuable than trying to memorize the pseudo-code (or—gasp—Java code) for the algorithm. To test your understanding of an algorithm, see if you can describe what it does to a friend and try to execute it by hand on small examples.

And, as they say, practice makes perfect. We have posted a sample exam on the course webpage. Try to work through each problem, first alone, then—if needed—with a classmate. Try to do a few problems from the text. Answers to odd-numbered book questions can be found in the appendix, so that you can check your solutions. Review the three problem sets, for which there are also sample solutions.

We've been impressed by your hard work and all you've learned this semester. Study hard but don't stress—and good luck!