CS 211 – Data Structures
Exam #1 Study Guide

Schedule

Exam #1 will be given in class on **Tuesday, October 8, 2024 @ 1:00 PM.** There will be no class material presented after the exam – once you finish the exam, you are free to leave the classroom. If you receive accommodations to take the exam, please make arrangement with the Testing Center.

Administrivia

You are permitted to bring into the exam a single piece of paper (8.5" by 11" or A4 size) on which you have **HANDWRITTEN** or **PERSONALLY CREATED using a word processor** (NOT copy-pasted from the text or other source), with whatever you wish on one or both sides. A 12-point minimum font is required – attempts to include more information than can be reasonably put onto a single sheet of paper using a font similar to that used for this document may result in a score penalty. NOTE: if you hand-write the entirety of your sheet of notes, the "font size" requirement will be enforced much more leniently!

This paper must be **turned in with your exam**, it must **include your name**, and it must be **created by you.** I'll try to return it with the graded exam, but you may want to photocopy it before the exam in case I should lose it. I'll also let you use that page of notes or its photocopy on the Final!

Other than this piece of paper, the exam is **closed-note**, **closed-book**, and **closed-computer**. Also NO **cell phones** and NO **other communication devices**! You are permitted to use a calculator that does not have communications capabilities. NOTE that calculations to answer questions during this exam may require that you show your work. Simple addition/subtraction/multiplication/division of two numbers may be done directly on the calculator, but the documenting of what calculations were done should be shown on paper when requested!

You are to work individually on all exams in this course.

This will be a pencil-and-paper exam. You only need to bring something to write with, and the page of notes mentioned above.

Your studying should include careful study of the covered textbook chapters and the assignments thus far.

Format of exam

It is NOT expected that you will write any original C++ code. You may be asked, however, to read and interpret C++ code supplied to you, and to choose (from Multiple Choice) a single line of C++ code to insert into a fragment of code to perform the tasks covered in the chapters of the text and in lectures.
Among the types of questions you may be asked include:

- Questions about complexity and big-O notation
- Average-time and Worst-time complexity questions
- Use of pointers and dynamic allocation in support of Abstract Data Types
- Statements and code fragments in C++ to perform various actions
- Tracing execution of C++ code to determine how contents of an ADT change
- Searching algorithms, such as binary and sequential
- The use of recursion in algorithms, and how to determine (in general terms) the complexity

of a recursive algorithm
- Determining whether given C++ functions perform recursively tasks correctly
- Identifying base cases and recursive steps in recursive functions
- Sorting algorithms and their general complexity (from "slow" O(n^2) to "fast" O(n log n))

Material to be Covered

You are responsible for all material covered in class, and for all material assigned as readings in the text as outlined below:

Covered in Class - Object-Oriented Programming using C++

- You should expect to have to read C++ code in this exam, and to determine which line of C++ code (from a set of choices) is appropriate to perform a particular task within a code block.
- Use of and familiarity with pointers and dynamic allocation/deallocation is part of C++ programming; you should be familiar with their use.

Chapter 1 – Introduction to Data Structures and Algorithms

- What is a **data structure** and how is it different from a basic data type?
- What is an **algorithm** and how can it be applied to a **computational problem**?
- What is meant by the **efficiency** of an algorithm? What is the general relationship between algorithms with **logarithmic** complexity, **polynomial** (including **linear** and **quadratic**) complexity, and **exponential** complexity?
- What are the basic differences between **O(1)**, **O(log n)**, **O(n)**, **O(n log n)**, **O(n^2)**, and **O(2^n)** complexity? At a basic level, how are their growth rates different as the value of n increases?
- Given an algorithm in pseudocode or C++ code, you should be able to determine the basic complexity of the algorithm by analyzing its actions, any loops/repetition, and recursive calls it contains
- What is meant by an **abstract data type**? What makes a data type abstract?
- Can a language like C++ provide ADTs for use by programmers? How is that accomplished? (Note: you will NOT be asked about any specific ADT supplied by C++ libraries!)
- What is meant by **runtime complexity**? By **space (or memory) complexity**? (NOTE: you will not get any questions specifically about space complexity, beyond knowing what space complexity is.)
- Given a formula or mathematical expression that represents the number of steps that an algorithm requires for a problem of size **n**, you should be able to give the big-O notation for that algorithm. (NOTE: you will not get questions about the time complexity of recurrence relations.)
- You should know the basic differences between **best-case**, **average-case**, and **worst-case** complexity. (NOTE: you will not get specific questions about best-case and average-case, only worst-case.)
- You should be able to examine C++ code and determine its big-O complexity.

Chapter 2 – Searching and Algorithm Analysis

- What are the basic characteristics in runtime complexity of **Linear Search**?
- What are the basic characteristics in runtime complexity of **Binary Search**?
- How is Linear Search performed on an array of items? On a linked list of items?
- How is Binary Search performed on an array of items?
- What is **asymptotic complexity** and how are the three notations (**Big-O**, **Omega**, and **Theta**) used to describe asymptotic complexity? (NOTE: you will not get specific questions about Omega and Theta.)
- How is Big-O complexity determined when given a function f(n) of the number of operations required to perform an algorithm on an input of size n?
- How can Big-O notations be combined when using the basic operations of addition and multiplication?
- Given an algorithm in C++ code or pseudocode, you should be able to determine the basic complexity of the algorithm by analyzing its individual actions, any loops/repetition, and recursive calls it contains.

Chapter 3 – Sorting Algorithms

- You should be familiar with the characteristics of the following sorts as presented in the text and in class:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Quicksort
  - Merge Sort
  - Radix Sort
- Which sorts fall into the "slow" $O(n^2)$ class? Which sort has approximate complexity $O(n^{1.5})$? Which sorts fall into the "fast" O(n log n) class? Which sort is called O(n), but actually has O(n log M) runtime complexity based on the n items being sorted and M being the magnitude of the values being sorted?

Chapter 4 – Lists, Stacks and Queues

- You should be comfortable with setting up and manipulating **singly-linked lists** and **doubly-linked lists** as covered in class, labs, and assignments.
- You should be familiar with the use of **pointers** in linked lists – their declaration, how to set them, how to use them, how to allocate memory dynamically to add nodes to a list, and how to deallocate memory when deleting nodes from a list. (NOTE: you will not be required to write C++ code.)
- You should be familiar with the use of **head** and **tail** pointers and how they may be updated when making changes to a linked list.
- You should be comfortable with standard drawings and graphical representations of a linked list.
- You should be familiar with how to access the data that is part of a linked-list node, and how to put values into the data field(s).

- You should know how to detect a **NULL pointer.**
- You should also be comfortable with how to simulate a linked list by using a C++ array.
- You should know what **LIFO** and **FIFO** are with respect to **queues**, and the differences between them.
- You should know the normal operations performed on a **stack**, and the situations for which a stack ADT is useful and appropriate.
- If given a sequence of **push**, **pop**, and other basic stack operations on a stack, you should be able to simulate how the stack behaves and note the contents of a stack after such operations.
- You should be familiar with how to simulate a stack using a C++ array
- You should know the normal operations performed on a **queue**, and the kinds of situations for which a queue ADT is appropriate.
- If given a sequence of **add** and **remove** operations on a queue, you should be able to simulate how the queue behaves and note the contents of a queue after such operations, including where the **front** and **back** pointers are pointing.
- You should be able to detect an empty queue and know how to avoid a remove operation on one.