CS 211 – Data Structures
Exam #2 Study Guide

Schedule

Exam #2 will be given on **Thursday, November 14, 2024 at 1:00 PM.** The exam time will be 80 minutes (+ accommodations). There will be no class material presented after the exam – once you finish the exam, you are free to leave the classroom.

Administrivia

This is a pencil-and-paper exam and is closed-book and closed-computer.

You are permitted to use during the exam a **single** piece of letter-sized paper (8.5" by 11") on which you have **handwritten or hand-created** whatever you wish on one or both sides. This paper **must** include your name, and it **must** be handwritten (or hand-created) by you. If you decide to type rather than write your notes, you are restricted to using a 12-point or greater sized font. You will turn this sheet of notes in with your exam – it will be returned to you with the graded exam.

Format of exam

The exam will be in a format similar to one that would be given on paper in a classroom, and mostly in multiple choice format. There will also be some short answer questions and some fill-in-the-blank questions. This will NOT be an exam where you write code in C++ (or any other language)! However, you may be asked to read C++ code to answer questions.

Among the types of questions you may be asked include:

- Questions about complexity and big-O notation for various actions on data structures
- Average-time and Worst-time complexity questions
- Which ADTs are best suited for particular applications
- Explanations of why particular ADTs are best suited for particular applications
- The contents and structure of a given data structure
- Given some C++ (or pseudocode) functions and methods, how they affect the use and contents of data structures when those functions are methods are called

Academic Honesty

It is ABSOLUTELY required that the work you present on the exam is your own! No real-time collaboration with other students is permitted. Canvas has features to combat cheating, such as randomizing the order of questions, randomizing the listing of multiple choices on questions, and even offering students different sets of questions from a "test bank" so that no two tests are exactly alike!

Material to be Covered

You are responsible for material covered in class sessions, required reading, and homeworks; but, here's a quick overview of especially important material.

- This exam covers all class lectures up through and including Week 11 Lecture 1.

- The contents of Chapters 6-9 in the course textbook, as well as material that we covered in class and lab (as listed in this Study Guide), and all assignments up through the Text Assignments in Chapters 6 through 9 and Labs up through Week 10. Material from previous chapters that is applicable to this material, such as computational complexity and recursion, may also be covered.

Chapter 6 – Hash Tables

- What is **hashing**? What are the computational complexity advantages in using hashing to search for values? What is a **hash function**? What is a **hash table**? What are **buckets** in a hash table?

- What are some of the desired properties for a good or "perfect" hash function? What are some of the techniques used in making a hash function?

- How are items inserted into a hash table? How are items retrieved? What is the average complexities for these operations? What is the worst-case complexity?

- How can an array be used to implement a hash table?

- What is a **collision** is hashing? What basic methods are used to handle collisions? What is **linear probing**? What is **quadratic probing**? What is the phenomenon of **clustering** in a hash table? How does quadratic probing resist clustering?

- What is **chaining** in a hash table? How can a linked list be used to implement chaining? How might this affect the complexity of inserting and searching for values?

- How can a value be deleted from a hash table? What is the peril in deleting values in a table that uses probing, and how is that peril avoided?

Chapters 7 & 8 – Trees and Balanced Trees

- What is a tree, in general? And what is the distinguishing characteristic of a binary tree?

- Given a picture, could you tell if something is a **tree** or not? Is a **binary tree** or not?

- You should be comfortable with basic tree-related terminology: **node, edge, parent, child, sibling, root, leaf, interior node, ancestor nodes, descendant nodes, subtree, left child, right child, left subtree, right subtree, height of a tree, height of a subtree, full binary tree, complete binary tree, perfect binary tree, balanced tree, path, path length.** Given a depiction of a tree, you should be able to identify parts and properties of the tree that these terms describe.

- You should know what a **balanced** tree is, and why it is desirable that a tree be balanced. You should know whether a given tree meets the definition of "balanced," as specified in the question or for the standard definition of an AVL tree.

- You should be familiar with the way a binary tree is implemented using the standard **node** data structure we defined in class, and how balanced-tree schemes such as AVL trees use a **balance factor** field within each node to keep track of balance.

- You should be familiar and be able to describe and demonstrate **preorder**, **inorder**, and **postorder traversal** of binary trees.

- Some additional facts about binary trees that you should know:
  - In a balanced tree with **n** nodes, what is its **average case height** (in general big-O notation terms)?
  - What is the **worst case height** that a tree with **n** nodes can have (in big-O notation terms)?

- What property of a binary tree makes it suitable to be a **binary search tree**?

- What is the basic algorithm to **search** for an item in a binary search tree? What are the **average** and **worst case** big-O complexities for such searches?

- You should be comfortable with **inserting into** a binary search tree (before balancing). What is the average and worst-case complexity for such insertions?

- You should be comfortable with **deleting** from a binary search tree (before balancing). What are the average- and worst-case complexity for such deletions? How does **delete by copying** work?

- If you traverse a binary search tree in **inorder**, printing the nodes' values as they are visited, what will be the result?

- What are some of the typical operations defined on binary search trees? What is a **right rotation**? What is a **left rotation**? How do the steps in these rotations maintain the "sorted" property of binary search trees?

- You should be familiar with the features of an **AVL tree**, the way an AVL tree uses and maintains balance factors at each node, and how an AVL tree rebalances when needed after an insertion or deletion. Specifically, you should be able to recognize the cases when inserting a new value in the tree where one rotation is done (**right-right** and **left-left**) and where two rotations are done (**right-left** and **left-right**).

- You should be familiar with the concept of **red-black trees**, including the properties that are maintained in a red-black tree, and recognize situations when a red-black tree is valid and when it's invalid and needs adjustment. You should know what color a new node is given when inserted (but before any necessary adjustments are made), and also know what is meant by an **uncle node** in a red-black tree. Specifically, you should be able to visually recognize these cases after inserting a new value in the tree – when the parent of the new node is black, when the parent is red and the "uncle" is red, and when the parent is red and the "uncle" is black. <u>NOTE: You will NOT be asked in the exam about the specific actions taken in these situations, and you will NOT be asked in the exam about node deletion in a red-black tree.</u>

Chapter 9 – Heaps and Treaps

- What is a **heap**? What basic property makes a binary tree a heap? What is the difference between a **min-heap** and a **max-heap**?

- In what kinds of situations is a heap a useful structure? What is a typical application of heaps?

- How is a heap implemented? What is the procedure for **inserting** a value into a heap, and for **removing** a value from a heap? What is the only location where a value is permitted to be removed from a heap?

- You should be able to explain how a heap ensures that it is still well-balanced after an insertion, and also after a deletion.

- What are the big-O complexities of heap operations?

- How does a heap implement the concept of a **priority queue?**

- What are the basic properties of a **treap**? How does a treap use the properties of both a search tree and a heap? By what mechanism does a treap promote and demote values stored in it that maintains the properties of a search tree? How is a value inserted into a treap? How is a value deleted from a treap?