Exam #1 Review Packet

COP 3503, Fall 2021

This packet gives an overview of topics to study as you prepare for Exam #1, which takes place Monday, Sept. 27. We will meet at our normal class meeting time, in our usual classroom. You will need a pen or pencil and your UCF ID. You cannot use calculators, notes, or any other aids.

The exam covers material presented from the beginning of the semester through Wednesday, Sept. 22. All information presented in class is fair game for the exam, even if I forgot to include certain topics in this packet, and even if I haven't created practice problems related to a certain topic.

In addition to reviewing all the topics and questions in this packet, I strongly encourage you to work through all the examples, practice exams problems, and supplementary materials I've posted in Webcourses as part of your preparation for Exam #1. Throughout the semester, you should also be challenging yourself to replicate any code I produce in class each day to ensure your Java coding skills are adequately honed heading into the exam.

I anticipate ending the exam around 5:40 PM so that we will have time to ensure a smooth transition out of the classroom by the time our class officially ends at 5:50 PM. You should expect to feel a bit pressed for time during the exam. My exams are usually designed so that those who have studied extensively and who know the material very well will have time to complete them. However, some of those students might still feel a bit pressed for time, and those who aren't adequately prepared might run out of time and not be able to answer all the questions.

The exact structure and point breakdown for your exam will be posted as an announcement in Webcourses as soon as I finish writing the exam (probably the evening of Sunday, Sept. 26).

Note: You are not responsible for the quadratic probing or AVL tree height proofs, and any questions on skip lists (which will be covered in our last lecture before the exam) will be kept fairly light. Other than that, you are responsible for any materials posted in Webcourses so far throughout the semester (including all practice problems), even if I have forgotten to mention them here. If you have any questions about what will or will not be on the exam, please feel free to ask!

1. (*Algorithm Analysis and Summations*) Be prepared to represent the runtime of a function and/or the value returned by a function using summations. Be prepared to derive closed forms for beasty summations. Be able to derive closed forms for geometric sums *using the process shown in class*. Can you identify O(log n) and O(n log n) code segments when you see them?

   References: *Analysis with Summations (Aug. 25 lecture); Order Analysis and Summations (Supplementary)*

2. (*Formal Definitions of Big-Oh et al.*) Know the formal definitions of big-oh, big-omega, and big-theta. Be prepared to do proofs involving those definitions and/or apply those definitions to given runtimes.

   Reference: *Mathematical Definitions of Big-Oh et al. (Aug. 30 lecture)*

3. (*Empirical Runtime Analysis*) Be familiar with the process for empirical runtime analysis and how and why it works. With respect to the formal definition of big-oh, we are solving for the constant and showing that we do actually converge toward some constant (assuming our big-oh expression is a tight bound). Why can't that constant be zero? (What is the implication if we converge to zero?) Also, what are some of the common pitfalls with empirical runtime analysis that we discussed in class?

   References: *Empirical Runtime Analysis (Aug. 30 lecture); Empirical Analysis of Treap Height (Sept. 20 lecture)*

4. (*Algorithm Analysis and Design; Being Clever*)[1] Given a novel problem, be prepared to come up with an efficient solution to that problem, write the code for your solution in Java, and comment on the runtime and/or space complexity of your solution. Alternatively, you might be asked to examine an inefficient solution to a problem, comment on its runtime, and modify it to improve that runtime – possibly utilizing one of the many data structures we have discussed this semester. That will require you to know about the runtimes of the operations associated with those data structures. I could also ask you to write a method that improves upon the space complexity (i.e., memory usage) of a given solution. Also, be familiar with the various Java-specific optimization tricks we've seen so far in class.

   References: *Exercises in Being Clever (Aug. 23 lecture); Further Exercises in Being Clever (Aug. 25 lecture)*

5. (*HashSet and HashMap*) You might be asked to write code with Java's HashSet and/or HashMap classes. What are the common methods that each class supports for insertion and retrieval? What are the differences between HashSet and HashMap? How do you iterate through all the elements in a HashSet, and how do you iterate through all the keys in a HashMap? For HashMaps, what is the runtime difference between *containsKey()* and *containsValue()*, and why? What happens if you design a new class and try to insert objects of that class into a HashSet or HashMap without defining a *hashCode()* method? What happens if you define just the *hashCode()* method, but no *equals()* method? Why?

   Reference: *Hash Containers in Java (Week #3 recitation)*

6. (*Hash Tables*) Assuming we do not allow duplicates into a hash table, what is the worst-case runtime for insertion? Why? Also, be familiar with the conceptual discussions we had about the various collision resolution mechanisms.

   Reference: *Hash Tables (Sept. 1 lecture)*

---

1 With respect to "being clever," I will not test you on your ability to regurgitate solutions to problems like MCSS or Target Sum. I think it's good to revisit those solutions (if you have time) to jog your memory and get your mind thinking in terms of algorithm optimization, but I don't expect you to memorize any of that code.

7. (*Binary Search Trees*) Be familiar with the cases that lead to best- and worst-case runtimes for insertion, deletion, and search within BSTs. Be familiar with the definition of tree height. (What is the height of an empty tree? What is the height of a tree with a single node?)

   References: *AVL Trees (Sept. 1 lecture); 2-4 Trees (Sept. 13; see Combinatorial Argument with BSTs); Treaps (Sept. 20 lecture)*

8. (*AVL Trees*) Using big-theta notation, what is the maximum height of an AVL tree with *n* nodes? What about the minimum height? (By the way, don't worry about AVL tree rotations, balance factors, etc.)

   Reference: *AVL Trees (Sept. 1 lecture)*

9. (*2-4 Trees*) Be familiar with properties of 2-4 trees. Be able to determine whether an arbitrary tree is a 2-4 tree or not. Be able to perform insertion and deletion operations on 2-4 trees *using the process shown in class*.[2] Be able to comment on runtimes of this structure's insert, delete, and search operations. Why is the best-case runtime for deletion from a 2-4 tree O(log n) and not O(1)?

   Reference: *2-4 Trees (Sept. 13 lecture)*

10. (*Bloom Filters*) Understand insertion and search with Bloom filters. Be familiar with the problems associated with deletion. Given some key and the state of a Bloom filter (in the form of a list of hash tables, vis-a-vis the diagrams I posted in my Bloom filter lecture summary), be able to trace through the process of determining whether that key is in the Bloom filter. When we find a key in all the hash tables within a Bloom filter, can we say with 100% certainty that the key was inserted into the filter in the first place (assuming, of course, that we don't allow deletion operations)? When we fail to find a key in all the hash tables within a Bloom filter, can we say with 100% certainty that it was *not* inserted into the filter in the first place (again, assuming no deletion operations are allowed)? How big should our hash tables be, and what factors influence that decision? If asked, could you answer some basic questions about how probability affects false positive rates with Bloom filters?

    Reference: *Bloom Filters (Sept. 15 lecture)*

11. (*Random BSTs*) What are the maximum and minimum heights of a BST with *n* nodes? Given *n* distinct elements, how many unique BSTs can be constructed that have height $(n - 1)$?

    References: *Treaps (Sept. 20 lecture; see Preliminaries: Random Binary Search Trees); AVL Trees (Sept. 1 lecture)*

12. (*Treaps*) Be able to trace through the process of inserting and deleting nodes from a treap. Be able to comment on the data structure's expected height and runtimes. What is the expected height of a treap if we insert data in sorted order? Unsorted order? Randomly sorted order?

    Reference: *Treaps (Sept. 20 lecture)*

13. (*Skip Lists*) What are the expected runtimes for insertion, deletion, and search within a skip list? In a skip list with *n* elements, what is the maximum number of pointers (or "references," in Java lingo) that any node will have? Be able to show what a skip list looks like after insertion of a new element.

    Reference: *Skip Lists (Sept. 22 lecture)*

14. (*Data Structures Review*) You should also be prepared to answer questions from the data structures review sheet that was distributed in labs. Many of the details from that review sheet have been mentioned in lecture, as well.

    Reference: *Data Structures Review (Week #4 recitation)*

---

2 Recall that we made executive decisions about which node to transfer from and/or fuse with when a leaf node finds itself empty. (Always start by looking left, then right.) (If we fuse and drop, we always fuse with the left sibling if there is one. Otherwise, we fuse with the right sibling.) Remember that we only transfer (i.e., borrow a sack of money (or whatever)) from siblings, not cousins. Also, when we perform a split operation (a.k.a. squish up), which node did we agree always to move up?