Important COP 3502 Section 2 Final Exam Information

Date: Wednesday, July 29, 2020

Time: 4 - 6 pm (online)

Exam Aids: Printed Notes, Calculator

Exam Grading Rule: If you write "I don't know" for your response to a 10 point question and nothing else, you will receive 3 points out of 10. If you do this for a 5 point question, you will receive 1 point out of 5.

Test Format:

Identical to the Foundation Exam (F. E.)

Part A of this exam is structured like Section 1 A of the F.E.

Part B of this exam is structured like Section 1 B of the F.E.

Part C of this exam is structured like Section 2 A of the F.E.

Part D of this exam is structured like Section 2 B of the F.E.

Each section will have 3 questions, two of which are worth 10 points and 1 worth 5 points. Thus, each section will be worth 25 points and together, the exam is worth 100 points.

F. E. Topic List:

http://www.cs.ucf.edu/wp-content/uploads/2019/08/FE-ExamOutline.pdf

This list was covered in great detail in class.

Exam Delivery Details

Please log onto Webcourses by 3:50 pm this Wednesday.

A couple minutes before 4 pm, in the Assignments tab, in the category "Final Exam", and assignment will be posted with the title FE-A. You will have 30 minutes to work on the questions. You must produce a single .txt, .doc, .docx or .pdf document with your answers to upload for the assignment. The due time will be 4:30 pm with a 10 minute grace period. No penalty will be assessed for any assignment turned in, in between 4:30 and 4:40 pm. If there is no submission though, the grade is a 0.

A couple minutes before 4:30 pm, in the same location (Assignments, Final Exam), an assignment will be posted with the title FE-B. The turn in time will be 5:00 pm with a 10 minute grace period. (Nothing accepted past 5:10 pm.)

A couple minutes before 5:00 pm, in the same location (Assignments, Final Exam), an assignment will be posted with the title FE-C. The turn in time will be 5:30 pm with a 10 minute grace period. (Nothing accepted past 5:40 pm.)

A couple minutes before 5:30 pm, in the same location (Assignments, Final Exam), an assignment will be posted with the title FE-D. The turn in time will be 6:00 pm with a 10 minute grace period. (Nothing accepted past 6:10 pm.)

Note: Do NOT blindly copy code examples. Doing so will result in an automatic 0 for a question. Please write your solutions from scratch. The worst thing you can do (worse than writing "I don't know"), is copy code that solves a problem not related to the problem I asked, because that proves that you don't understand what I asked, let alone how to solve it!

Part A Q1 - Dynamic Memory Allocation

Given a scenario and possibly a struct, write code to allocate some memory and read in some information, or write code to free all of the dynamically associated memory. Make sure you look over all the different "general patterns" shown in class. A pointer can be allocated to point to a single struct or an array of struct. A double pointer can point to and array of pointers, and each of those pointers can be allocated to point to either a single struct or an array of struct. Strings can be dynamically allocated to "fit" the right size.

Part A Q2 - Linked Lists

Make sure you look at all the mechanics involved in inserting and deleting nodes from a linked list. Also, consider slight "twists" in the design of a linked list, like a circular list or a doubly linked list.

The most important pieces of information dealing with linked lists:

1) Watch out for NULL pointer errors
2) Make sure you don't "lose" the list.
3) Make sure you connect the links in the proper order.
4) Don't forget to "patch" everything up for some operation.
5) Determine when it is necessary and not necessary to use an extra helper pointer.
6) Determine when it is necessary and not necessary for a function to return a pointer to the beginning of the list.

Part A Q3 - Stacks and Queues

Stacks are last in, first out structures and Queues are first in, first out structures.

Typically, stack and queue operations occur in O(1) time if the structure is efficiently implemented.

The array implementation of a stack just needs the array, its size and an integer storing the index to the top of the stack.

In a queue, you need more information. Typically, we also need to store the number of items currently in the queue as well as the index to the front of the queue.

Make sure you understand how the implementation here affects run-time. (For example, if we implemented a queue with an array but always had the front of the queue be index 0, a dequeue could potentially take O(n) time to move each element forward one slot.)

In a linked list implementation of a queue, a pointer is needed to the back of the queue as well as the front. But, for a stack, only the latter is needed.

Part B Q1 - Binary Search Trees

Many of the concerns necessary with linked lists translate over to binary trees. One key point about binary trees:

Recursion is even more important/useful for binary trees than linked lists. In particular, it's very difficult to think about how to iteratively go through all the nodes in a binary tree, but recursively, the code is reasonably concise and simple.

Part B Q2 - Binary Heaps/Hash Tables

A binary heap is an efficient data structure to use if the main operations that need to be handled are inserting items and deleting the minimum item. Both tasks can be done in O(lg n) time, where n represents the number of items stored in the heap.

Typically, a binary heap is implemented using an array. The implementation I showed in class stores the first element of the heap in index 1. In this convention, the left child of a node at index i is at

index 2i and the right child is at index 2i+1. A node stored at index i has its parent at index i/2.

The key to getting a binary heap working are the subroutines identified in class as "percolateUp" and "percolateDown".

A hash table is an efficient data structure that easily allows for inserting items and searching for items. The main problem with hash tables is collisions, since hash functions are many-to-one functions (meaning that two different input values can hash to the same output location.) There are three ways to deal with collisions discussed in class:

a) Linear Probing
b) Quadratic Probing
c) Linear Chaining Hashing

The first couple are reasonable so long as the table is no more than half full. The last is most probably the best way to deal with the issue.

Part B - Q3 AVL Trees/Tries

All I will test upon for these is how to do rotations after insertions and deletions. Remember the following ideas:

1) All insertions can be fixed with a single rotation.
2) Deletions may need more than one rotation to be fixed. But all errors are propagated up the tree.

To find WHERE to do a rotation, start at the inserted node or the parent of the deleted node, going "up" the tree, node by node, until you find an offending node. Then perform the appropriate rotation. From there, continue up the tree.

For Tries, know how the structure is stored, and keep in mind how storing extra information in each node can be helpful. For most coding questions with tries, the key is making a recursive call on each child node and then using the answers from those recursive calls in an appropriate way.

Part C Q1 - Algorithm Analysis

This question either asks about several run times of known data structures and algorithms, or asks to analyze a new problem. In the latter, count up how many simple steps get executed based on the input parameters.

Part C Q2 - Timing Questions

For these problems, you are typically given some information about an algorithm's run time and information about one instance of it running. This information can be used to set up an equation with a constant c, and then the value of c can be derived. The question then includes predicting the run-time of the algorithm for a different input size. To do this, plug into the equation with the known value of c and do the math to simplify the result.

Part C Q3 - Sums/Recurrence Relations

You should be able to handle sums between various bounds of a constant and a linear function. The key is to split a sum into separate parts, then factor out any constants, split any sum that doesn't start at 1 to be the difference of two sums, and then solve each sum separately and simplify all the terms.

For recurrence relations, you iterate the given relation three times, and based on those iterations, make a guess for the form of the

recurrence after being iterated k times. After this guess, plug in a value of k that makes the recurrence disappear (either have T(1) or T(0)) and then simplify the result.

Part D Q1 - Recursion

I am likely to ask a recursive coding question and a recursive tracing question.

Remember, that often, recursion fits into one of two constructs:

1) if (!terminating condition) { do work }

2) if (terminating condition) { finish } else { do work, call rec }

However, not all recursive algorithms, follow these two constructs. Consider the permutation algorithm. It does not just make one recursive call or even two.

The main idea behind recursion is to take a problem of a certain size, do some work and finish solving the problem by solving a different version of the same problem of a smaller size.

The toughest part is "seeing" how you can break a problem down into a smaller recursive solution.

My favorite analogy is imagining that someone else has written a function to solve the task already, and your job is to write a function to solve the task at hand, but you can call the function that someone else has written as an aid, just not with the same parameters.

The more complicated recursive patterns, such as floodfill and permutation do have loops in them, because there are many "directions" to recursively move in, so to speak.

Part D Q2 - Sorting

Make sure you can trace through each of the five sorts covered in class: Insertion Sort, Bubble Sort, Selection Sort, Merge Sort and Quick Sort. Know the best, average and worst case run times of each. If you are asked to code, it would only be one of the $O(n^2)$ sorts or perhaps either the merge or the partition. Make sure you understand the strengths and weaknesses of the techniques involved.

Part D - Q3 Backtracking/Bitwise Operators

I spent a decent amount of time looking at three examples of code using backtracking: Eight Queens, Sudoku and the Digit Divisibility problem. I may have you fill in an incomplete function that executes some sort of backtracking. The key to backtracking is that it is very similar to brute force, except we *skip* building off of partial solutions that we know are doomed to fail. This is how a Sudoku puzzle which looks like it has something like 9^{50} options (if 31 squares are given) can still be solved very quickly. Mostly, we set up our code to run a brute force solution, but then in the loop where we are filling in the next portion of the partial solution, we skip over unviable options. In the code examples I've shown you, I typically implement this with an if statement that screens out invalid cases inside the for loop. This if statement, when triggered, executes continue, which skips the ensuing recursive call that would have been made, building off of that partial solution. (For example, in the Digit Divisibility problem, when we see that 326 isn't divisible by 3, we skip trying *ANY* number that starts 326.)

Bitwise and = &
Bitwise or = |
Bitwise xor = ^
Bitwise not = ~

When using bitwise operators, it's important to remember what each of the 32 bits in an int represents. In particular, the most significant bit is worth -2^{31}. The rest of the bits are worth their unsigned value.

We can use a single integer to store an array of true/false. In this manner, we can store items of a subset (if bit 0 is 1, then item 0 is in our subset, if bit 7 is 0, then item 7 is not in our subset, and so forth).

Lots of real world problems can efficiently be encoded with the bitwise operators, such as finding the intersection of two sets (bitwise and), union of two sets (bitwise or), or exclusive or of two sets (bitwise exclusive or).

We can code up a brute force solution without recursion going through each subset of a set using bitwise operators as well.