Important COP 3502 Section 2 Final Exam Information

Date: 4/28/2016 (Thursday)
Time: 1 - 3:50 pm
Room: CB2-106

Exam Aids: **Three sheets of regular 8.5"x11" paper**

Test Format: **Various formats of questions: coding, tracing, short answer, math oriented**

Make Up Exam Information

Date: 4/27/2016 (Wednesday)
Time: 3 - 5:50 pm
Room: HEC-240

Note: The make up will be given in another room to be announced. If you *need* to take the make up, please just come to my office. We'll find a room (hopefully in HEC), for you to take the exam.

So that I have a good idea of numbers, if you haven't already, please email me at dmarino@cs.ucf.edu telling me that you'll be showing up for the make up on Wednesday (4/27) at 3 pm.

Outline of Topics for the Exam

I. Basics of C – if, loops, functions, array, strings, files

II. Mathematical Background

- **a. base conversion (2, 10, 16, other)**
- **b. logs and exponents**
- **c. sums**
- **d. Big-Oh Timing problems**
- **e. Recurrence Relations**

III. Pointers and Dynamic Arrays

- **a. how to allocate space dynamically (array, 2d array, array of struct, array of ptr to struct, linked list node, bin tree node, etc.)**
- **b. how to free space**
- **c. how to "resize" an existing array**

IV. Structs

- **a. how to declare structs**
- **b. how to use pointers to structs**
- **c. how to use arrays of structs**
- **d. how to pass structs into functions**

V. Recursion

- **a. Fibonacci**
- **b. Factorial**
- **c. Towers of Hanoi**
- **d. Binomial Coefficients**
- **e. Binary Search**
- **f. Generating Permutations**
- **g. Fast Modular Exponentiation**
- **h. Floodfill**

VI. Algorithm Analysis

- **a. Average case vs. Worst case**
- **b. Determining a Big-Oh bound**
- **c. Use of sums**
- **d. Timing Problems**
- **e. Recurrence Relations**

VII. Linked Lists
- **A. Creating Nodes**
- **B. Insertion, Searching**
- **C. Deletion**
- **D. Circularly linked**
- **E. Doubly linked**

VIII. Stacks
- **A. Array Implementation**
- **B. Dynamically Sized Array Implementation**
- **C. Linked List Implementation**
- **D. Efficiency of push, pop**
- **E. Determining the Value of Postfix Expressions**
- **F. Converting Infix to Postfix**

IX. Queues
- **A. Array Implementation**
- **B. Dynamically Sized Array Implementation**
- **C. Linked List Implementation**
- **D. Efficiency of Enqueue and Dequeue**

X. Binary Search Trees
- **A. Creating Nodes**
- **B. Tree Traversals (preorder, inorder, postorder)**
- **C. Insertion**
- **D. Searching**
- **E. Deletion**
- **F. Code Tracing**
- **G. Writing Code (recursive)**

XI. AVL Trees
- **A. AVL Tree Property**
- **B. Identifying nodes A, B and C for both insert and delete**
- **C. Restructuring for both insert and delete**
- **D. Delete may have multiple restructures**

XII. Binary Heaps

- **A. percolateUp**
- **B. percolateDown**
- **C. Insert**
- **D. deleteMin**
- **E. makeHeap**
- **F. Heap Sort**

XIII. Hash Tables

- **A. linear probing replacement technique**
- **B. quadratic probing replacement technique**
- **C. linear chaining hashing**

XIV. Sorting and Selection

- **A. Bubble Sort**
- **B. Insertion Sort**
- **C. Selection Sort**
- **D. Merge Sort**
- **E. Quick Sort**
- **F. Quickselect**

XV. Bit-Wise Operators

- **A. left shift, right shift, and, or, xor, complement**
- **B. How to use a number to indicate a subset.**
- **C. How to iterate through all possible subsets w/bitmask.**

Mathematical Background

a) With respect to binary, remember the algorithm to convert to and from binary and decimal. To go from decimal to binary, use repeated division and mod by 2. In general, to go from base 10 to another base, use repeated division and mod by that base.

b) Make sure you know how to apply some basic log rules, including adding and subtracting two logs, the power rule, and that the log and exponent functions are inverses of each other.

c) You should be able to handle sums between various bounds of a constant and a linear function.

d) To set up the Big-Oh problems given in this course, make sure you set up a function that solves for the running time of an algorithm and use the given information to solve for the unknown constant.

Structs

Just make sure you can handle the various different modes in which structs are used. (By themselves, in an array, inside of another struct, etc.) Also, pay attention to the difference between a struct and a pointer to a struct. The latter is more typically used.

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.

Algorithm Analysis

The key goal here is to determine the number of simple statements that are run by a segment of code. Typically, a summation can be set up to determine this, in terms of some input parameter.

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.

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.

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.

AVL Trees

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.

Binary Heaps

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".

Hash Tables

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.

Bit-Wise Operators

Bitwise and = &
Bitwise or = |
Bitwise xor = ^
Bitwise not = ~
Bitwise leftshift = <<
Bitwise rightshift = >>

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.

The idea of bit masking is what we used to iterate through all possible subsets of a set of values. Basically, each integer's binary representation stood for which values in our set we "counted" and we could just use an integer to track that particular subset.

Foundation Exam Computer Science Format

Although many of you aren't taking the Foundation Exam, everything that's on the syllabus for the Foundation Exam is fair game for our final. Furthermore, for those of you taking the exam, I wanted for you to see the format of the CS sections of the exam. They are quite strictly prescribed so that you can narrow your studying for the Foundation Exam. Here are the guidelines for the Foundation Exam:

I. Computer Science Section

A. Computer Science Part A

1. Recursive Coding

- **i. Need a terminating condition**
- **ii. Need an algorithm for non-terminating case.**
- **iii. In particular, you must reduce a question to "smaller" instances of the same question.**
- **iv. Do not try to think of an iterative solution!!!**
- **v. Towers of Hanoi solution and recursion**
- **vi. Permutation**
- **vii. Floodfill**

2. Summations and Algorithm Analysis

- **i. Break them down into multiple summations if necessary**
- **ii. Evaluate each of those using summation formulas.**
- **iii. Remember that indices of summation are important.**
- **iv. The n in the formula is JUST a variable!!!**
- **v. Deriving recurrence relation from code**
- **vi. Using iteration to solve recurrence relations**

3. Abstract Data Structures – Tracing/Coding

i. Stacks

- **a. Converting infix to postfix expressions**
- **b. Evaluating postfix expressions**
- **c. Array Implementation**
- **d. Linked List Implementation**

ii. Queues

- **a. Array Implementation**
- **b. Linked List Implementation**

4. Advanced Data Structures - Tracing

i.Binary Trees

- **a. Traversals**

ii. AVL Trees

- **a. Insertion**
- **b. Deletion**

iii. Hash Tables

- **a. Hash Function Properties**
- **b. Linear Probing Strategy**
- **c. Quadratic Probing Strategy**
- **d. Separate Chaining Hashing**

iv. Binary Heaps

- **a. Insertion**
- **b. Delete Min/Max**

5. Base Conversion

i. Conversion from binary to decimal.

ii. Conversion from decimal to binary.

iii. Conversion between binary, octal and hex.

B. Computer Science Part B

1. **Algorithm Analysis**
    - **i. Known Data Structures**
    - **ii. Best, Average, Worst Cases**
    - **iii. Based on various implementations**
    - **iv. New Problem Analysis**
2. **Timing questions**
    - **i. Set up correctly with an unknown constant**
    - **ii. Solve for the constant.**
    - **iii. Use direct formula to answer the question**
    - **iv. For loop questions, write out summations**
3. **Linked Lists – Coding**
    - **i. How to allocate space for a new node (malloc)**
    - **ii. When to check for NULL**
    - **iii. What free does**
    - **iv. Iteration vs. Recursion**
    - **v. Insertion**
    - **vi. Deletion**
    - **vii. Structural Modification**
4. **Binary Trees – Coding**
    - **i. How to allocate space for a new node (malloc)**
    - **ii. When to check for NULL**
    - **iii. What free does**
    - **iv. Using recursion with trees**
    - **v. Computing sum of nodes**
    - **vi. Computing height**
    - **vii. Other variants**
5. **Sorting**
    - **i. Insertion Sort**
    - **ii. Selection Sort**
    - **iii. Bubble Sort**
    - **iv. Merge Sort (Merge)**
    - **v. Quick Sort (Partition)**