**ECE198KL: Introduction to Computer Engineering II** **Spring 2013**
**Study Materials**

## Sample Problems to Help you Prepare for the Final Exam

A few people expressed an interest in having some standalone problems that they could do in order to help them prepare for the final exam. Towards that end, I've put together this list.

The problems here are roughly in increasing order of difficulty, but integrate many of the concepts that we've covered in this semester. Feel free to do only those parts that are of use to you.

### 1. Singly-Linked Lists

Develop a singly-linked list in which each node holds (as data) a single integer. Use the `valgrind` tool on EWS to make sure that your program uses dynamic allocation properly.

**a.** Write routines that add a new integer (include allocating the new node), remove a node, find a specific integer (return node pointer or NULL), and print the list.

**b.** Add a menu system that allows you to exercise the routines from **Part a**. For example, you might type "add 42" to add a node with 42 into the list, then type "p" to print the resulting list.

**c.** Add save and load commands.

**d.** Keep the list sorted numerically, either in increasing or decreasing order.

**e.** Write a routine to reverse the list in place. If you have sorted the list and have implemented a menu system, add a command that reverses the list, prints the reversed list, and reverses the list again.

**f.** Rewrite the find, sorted insertion, and remove subroutines to operate recursively.

### 2. Multiple Lists

Develop a structure that includes an age and a dynamically allocated name as data and is linked into two sorted, singly-linked lists. One list should be sorted by age (say increasing), and the second by name (say in ASCII order, although you can figure out how to do it alphabetically if you prefer).

**a.** Write routines to add and remove nodes from the two lists, and to print each sorted list.

**b.** Add a menu system that allows you to exercise the routines from **Part a**. For example, you might type "add Lumetta 42" to add a node with name "Lumetta" and age 42 (close!) into the list, the type "p" to print the resulting list.

**c.** Write routines to find a specific name and to find the "next" person with a given age. Use one routine to enable a caller to walk over all list elements with a particular age:

```
for (var = routine (some args); NULL != var; var = routine (other args)) ...
```

### 3. Doubly-Linked Lists

Implement a doubly-linked list with and without a sentinel node. Write routines to add an element, remove an element, and search for an element matching some data field. Compare the difference in implementation complexity.

**4. Assembly**

If you want to exercise your assembly skills (not likely to be used much on final), rewrite Programs 2 and 3 to use a pointer-based data structure instead of integer indices for choices. In other words, the structure for each page should contain a `char*` followed by three `page_t*`. Illegal choices should contain NULL (0). You will also need to change the game code to make use of the new database. Implement the search and path challenges based on the new data structure.

The first MP in 391 usually includes a pointer-based data structure in x86.

**5. Tries for Spellchecking**

For this problem, you can either use the `word_split` code provided to you or rewrite the functionality yourself.

Given a bunch of words, build a trie to hold a dictionary of those words.

What's a trie? We'll just give an example here: a trie node in this case consists of two parts. The first is a Boolean value (an integer) representing whether or not the word represented by that node appears in the input set. The second is a set of 26 pointers—one per letter in the English alphabet—pointing to other trie nodes.

The root node in the trie corresponds to the empty string. To decide whether or not a word appears in the trie, follow the links corresponding to the letters in the word down from node to node. If you find a NULL pointer, the word is not present. If you run out of characters, you have found the node corresponding to the word and can check the Boolean value to see if the word is present (some sequences may simply be prefixes for words...just as the sequence "wor" appears in a trie built to contain "word").

**a.** Read the words from a file named on the command line. Write a recursive routine that allows you to add a new word (if not already present) to the tree, then add all words from the file to the trie.

**b.** Write a recursive routine that looks up a word in the trie. If you set it up right, you can use the result as a spellchecker, which prints the words in a file that do not appear in the dictionary (the trie).

**c.** Write a recursive routine that prints all words in the trie in alphabetical order. You can check your answer by sorting the dictionary file (`sort foo > foosorted`) and comparing it with your output.

**6. C++ Class Hierarchy**

This problem focuses on C++. It's not particularly difficult, however. I just put it last because it's C++.

**a.** Design a C++ class hierarchy containing a base class ALPHA, two classes derived from ALPHA, BETA and GAMMA, and a DELTA class derived from GAMMA.

**b.** Add constructors and destructors that print distinct messages (for example, "BETA's constructor") for each class.

**c.** Create static, dynamic, and automatic variables of the class types and make sure the that you understand the execution order and timing. Include print statements in `main` so that you can see when the variables are constructed and destructed relative to that function's execution.

**d.** For one class, add both a constructor with no arguments and a constructor with arguments. Declare an array and a variable for that class, and make sure that you observe when and how many times each constructor is called.

**e.** Remove the `virtual` qualifiers that you should have put on the destructors. Now create a dynamic GAMMA (using `new`), store its address into a ALPHA*, and delete the ALPHA*. Which destructors are called?

**f.** Write a copy constructor and an assignment operator for one of the classes. Include some distinct printing in each function. Be sure that you understand which one is called in which situations.