CodeU Coding Exercises - Session 3

Sorting, Searching, and Hash Tables

Please Help - Share your Feedback:

Session 3 Feedback Form

Lightning Round (DO ALL FOUR)

This week the first four exercises are similar to coding questions you might get in an interview. You should work on each exercise in 3 phases:

Phase 1 - see how far you can get in 20-25 minutes just thinking and writing - no compiling, no reference material lookups. If you complete the question, that's great - you can move on to the next.

Phase 2 - if you do not finish it in 25 minutes, note how far you are from completing it. If you can complete it with extra time, do it. If you need help, get help and then complete it.

Phase 3 (optional) - after you have your completed solution that is ready to submit, then you may try to compile it and actually get it working.

In an interview, you will be expected to ask clarifying questions, and the interviewer will help you if you get stuck. For these exercises, make note of any questions you would ask the interviewer. Try to express your general solution strategy in one or two sentences. If you are faced with an implementation choice, you will have to make the decision. Make a note of that, too. It is OK to choose the fastest/easiest to code, but you get extra credit for knowing there is an alternate (possibly better) solution. When you meet with your mentor you should discuss the questions, your decisions, and your overall strategy. You won't have time to write tests or framework code during these exercises, but you should think about (and write down) relevant test cases, especially corner cases.

Exercise 1

Write a boolean function **isInDictionary()** that accepts a string parameter containing a word and returns true if the word is in the dictionary. The dictionary you are to use is packaged as a class named **TrivialDictionary**. It has a single static function named **TrivialDictionary.wordAt()**. This function has a single input, an integer index, and it returns the word at that index as a string, or null if the index is out of range.

For extra points: Modify isInDictionary() to keep information from prior calls and use it to speed up the function (i.e., reduce the number of calls you have to make to TrivialDictionary.wordAt()).

Exercise 2

A collection has a majority if a particular value appears more than 50% of the time. For example { 5, 3, 9, 4, 3, 3, 8, 3, 3 } has a majority value of 3, and { 5, 3, 9, 4, 3, 3, 8, 3 } has no majority. Write a boolean function **hasMajority()** that accepts an unordered collection of integers (you can choose what collection type you would like to use) and returns an Integer object with the majority value, or null if the collection has no majority.

Hint: If you ask “may I sort the input”, the answer will be “yes”.

Exercise 3

Create a class with: (1) a **constructor** that accepts an unordered array or vector of integers (your choice), and (2) a method **nthLargest()** that returns the nth largest integer from the original vector. For example, with an input of { 5, 3, 9, 4, 3, 3, 8, 3, 3 }, nthLargest(1) should return 9, and nthLargest(4) should return 4. For inputs of zero or greater than the size of the input, nthLargest() should throw an exception. The input array will be very large (but you can make a copy), and the input to nthLargest() may also be large, so a linear search algorithm is not acceptable.

Hint: If you ask “may I sort the input”, the answer will be “yes, but there is a more efficient approach”. If you ask for a hint regarding the more efficient approach, I’ll ask if it is necessary to fully sort the input to implement nthLargest(). If that doesn’t help, I’ll suggest you proceed with the sort-based approach, then explore other possibilities.

Exercise 4

You are working on a system that uses a **QueryStream** class. A **QueryStream** is basically a list of **Query** objects. Each **Query** contains a timestamp, some additional fields we don’t care about right now, and a single string that contains a list of words separated by white space. Write the definitions for QueryStream and Query. Then, create an iterator for the QueryStream class. The iterator should implement **hasNext()** and **next()**. Each call to next() should return the next word from the string of words in the current Query in the QueryStream (starting with the first) and advance to the next Query when the current Query's string is exhausted. Also, between the strings of two consecutive Queries, the iterator should return the string "<NEWQUERY>". hasNext() should return true as long as there are words for next() to return.

For extra points: Instead of the string "<NEWQUERY>" between queries, use the timestamps to generate a string with the decimal number of milliseconds since the last query (you may assume the timestamps are always increasing).

Optional Exercises

The following exercises are optional, but if you have time you should try at least one of them. These exercises all build on exercises from session 2. If you did not do the precursors to any of these exercises you can either: (1) go back and do one of those exercises instead, or (2) ask your mentor for the session 2 solution, and do this session's exercise using that as a starting point.

Exercise 5: Tree Flattener Revisted

This is a repeat of last session's exercise to flatten a binary tree, except instead of returning all the nodes in a single call to a **flatten()** method, you will provide an interface that allows the user of your class to create an iterator with the desired traversal strategy, and then use that iterator to access each node of the BinaryTree object in the desired order.

Here's the problem restated:

In this exercise you will define a **BinaryTree** class. Your binary tree should be composed of **BinaryTreeNode** objects. Each BinaryTreeNode instance has a reference to two child nodes (traditionally called left and right, or leftChild, rightChild) and a single string for its "payload". The left and/or right references may be null. If both are null, the node is a "leaf" node. Your BinaryTree instance should reference a single "root" node (which may be null).

Your primary method for accessing the contents of the BinaryTree will be called **iterator()**. It will return a slightly simplified Java-style iterator that is initialized to reference the "first" node of the BinaryTree object. The iterator implements two methods, **hasNext()** and **next()**. hasNext() is a boolean method that returns true if there is a node that has not yet been visited. next() returns the *payload* of the current node, and advances to the "next" node. NOTE: this is different from last session; the iterator should return the payload of the node, not the node itself.

You should implement your iterator as a abstract class called **BinaryTreeIterator**. Then you should implement at least two different concrete subclasses that implement different traversal strategies. The user will instantiate the specific iterator or iterators for the traversal strategies they need.

Like last session, you will need some additional methods to build the tree to test your "flattener".

Exercise 6: Hashing

Start with one of these exercises from last session:

- Exercise 4: Build and Query a Genealogical Tree
- Exercise 5: Collatz Sequences Revisited
- Exercise 7: Real Sparse Matrices

And use a hash table for your primary lookup method.

If you've already used a hash table in one or more of the problems, then you can either choose a different problem, or experiment with different hashing algorithms or hash table sizes to see how (or if) it affects performance. If you need ideas, talk to your mentor or post a question to the group forum. Also, here are some online references that discuss hashing: http://howtodoinjava.com/2012/10/09/how-hashmap-works-in-java/, http://en.wikipedia.org/wiki/Hash_table, http://www.ibm.com/developerworks/library/j-jtp05273/.

Exercise 7: File I/O

Start with one of these exercises from last session:

- Exercise 4: Build and Query a Genealogical Tree
- Exercise 5: Collatz Sequences Revisited
- Exercise 7: Real Sparse Matrices

And add methods to save all the key objects to a file, and restore them from that file. For a quick tutorial on file I/O, see https://docs.oracle.com/javase/tutorial/essential/io/file.html). Consider using JSON http://www.oracle.com/technetwork/articles/java/json-1973242.html to serialize and deserialize your objects.

Please Help - Share your Feedback:

Session 3 Feedback Form