| **CS 127: Data Structures** | **Winter 2006** |
|---|---|
| *Handout 3: Problem Set #1 (Due: 11:59p, Sunday, 15 January 2006)* | *6 January 2006* |

*The only place success comes before work is in the dictionary.*
— Vince Lombardi (1913–1970).

This assignment is to be done with your assigned partner. You'll submit a single solution, with both of your names on it. Of course, you haven't yet had your partner assigned, so take the time between now and when I send out the pair assignments to read over this assignment and think about it. Don't start to answer the questions until you've met with your partner, but you're welcome—encouraged!—to start reading the code. There are two major goals with this assignment: first, to shake off any Java rust that may have accumulated since you took CS117; and, second, to get some experience thinking about interfaces, abstract classes, subclasses, inheritance, and so forth.

Go to one of the labs in CMC 304 or 306 and make sure that you can log in. If you're having any trouble, see Mike Tie in CMC 305. After you've logged in, create a directory called `ps1`. In it, create a text file called `ps1.txt`, which should have both of your names at the top. Go to the course web page and download the following files into your `ps1` directory:

- `Dictionary.java`,
- `DictionaryEntry.java`,
- `NotInDictionaryException.java`,
- `DictionaryUnsortedArray.java`,
- `DictionarySortedArray.java`, and
- `Tester.java`.

Read over the code and make yourself happy with it.

0. Estimate the amount of time you spent on this problem set, and write it at the top of your `ps1.txt`.

1. There's a subtle difference between the sorted- and unsorted-array implementations of `Dictionary`: if the user defines the same word $w$ twice, then the two different implementations may possibly return different definitions of $w$ when `define`($w$) is invoked, even after exactly the same sequence of previous insertions/definitions. Although the specification of dictionary as we've given it only requires that *some* definition of $w$ be returned, let's be consistent with a more stringent specification.

   (a) Write (in your `ps1.txt` file) a brief (∼2–3 sentence) description of why this difference between the implementations can occur. Then create a new tester class—akin to `Tester.java`—that shows the two implementations performing differently.

   (b) Modify the implementations so that they are guaranteed to give the *most recently inserted* definition of word $w$ when `define(`$w$`)` is invoked. Using your new tester class, demonstrate that this discrepancy between the two implementations has been resolved.

2. The major limitation with these array-based implementations of dictionaries is that they can run out of space: when you create a new `DictionaryUnsortedArray` (or a `DictionarySortedArray`), you specify (as a parameter to the constructor) the maximum size that your dictionary will ever take on. This is a pain for the user—and you have to know in advance how many words you're going to define.

   If you know what `ArrayList`s are, don't use them in this question.[1]

   (a) Copy `DictionaryUnsortedArray.java` into `DictionaryExtendableUnsortedArray.java` and modify it to remove this limitation. Whenever `insert()` is called when the array `dict[]` is

[1] Don't use them if you don't know what they are, either.

already full, instead of throwing an exception, instead do the following: create a new array that's twice the size of the previous one; copy all the entries from the old array into the new array, and set `dict` to be the new array instead. Also add a new constructor that takes no arguments and sets the initial maximum size of the dictionary to be, say, five.

(b) Write some tests to show that your dictionary can now handle large numbers of words even if it was constructed with a small initial maximum size.

3. Now that you've made these changes to the unsorted-array implementation of `Dictionary`, you should be in a great mood. Except for one picky thing: really, you ought to update the *sorted*-array implementation, too. And the problem with that is this: you're going to end up writing the same array-doubling code over again in the sorted version. One of the great things about object-oriented programming is avoiding code duplication, and here we are duplicating code. This situation should suggest that there is a better way to organize these classes.

   Create a reasonably named subdirectory of `ps1` to hold your code for this question. You'll probably want to copy everything that you've done so far into your subdirectory; you'll be doing a lot of code reorganization here.

   Define an abstract class `DictionaryArray` that implements the interface `Dictionary`. Now rewrite `DictionaryExtendableUnsortedArray` and `DictionaryExtendableSortedArray` (which you should create by copying `DictionarySortedArray` and making appropriate modifications) to be subclasses of `DictionaryArray`. This question is mainly asking you to move code around—you shouldn't have to write much more code beyond what you did for the previous questions. In order to make this class organization work properly, the code that would have been duplicated will have to appear in the abstract class `DictionaryArray`.

   *Before you start coding, think! Figure out what code needs to go into the abstract class, what code will remain in the (actual) subclasses. Only then should you start typing anything.*

4. I promise you that doubling the array when it fills up is a good plan—but you shouldn't believe my promises so quickly. This question asks you to explain as fully as you can why doubling the array size makes sense. Write as full an explanation as you can, and include your answer in `ps1.txt`. Here is a hint to get you started in the right direction: $1 + 2 + 4 + 8 + \cdots + 2^k < 2(2^k)$. You should try to incorporate responses to the following questions in your answer:

   (a) Suppose that you start out with an array of size 1, and you subsequently insert 65536 ($= 2^{16}$) different words. How many "entry-moving" events (where an entry is copied from a full array into the new double-size array) are there? Suppose that, instead of doubling the array size from $n$ to $2n$ when it fills up, you add 10 to the array size, from $n$ to $n + 10$. How many entry-moving events would there be now?

   (b) If multiplying is so good, why wouldn't you expand the array from $n$ to $9999n$ when it fills up? Wouldn't that be even better? And, if not, if multiplying by a small number is so good, why wouldn't you expand the array from $n$ to $\lceil 1.0001n \rceil$ when it fills up?[2] Wouldn't that be even better? You might want to use the word "tradeoff" in your answer.

   *Note:* this is an open-ended question! Give as complete and coherent an answer as you can, but don't fret if you don't have a totally complete explanation. Do the best that you can. Don't feel obligated to write more than 2–3 paragraphs in response.

When you're done, use `hsp` to submit your answers—you can submit your entire `ps1` directory. Again, be sure that both of your names appear at the top of your `ps1.txt`.

*Start early, ask questions, and good luck!*

[2]The expression $\lceil x \rceil$—read "ceiling of $x$"—is the smallest integer that is at least as large as $x$.