Problem 3. Extending The Dictionary

Chances are that your implementation of the search function `wordInDictionary` uses linear search. This approach is inefficient, and in this problem you will extend your `CDictionary` class to implement the faster binary search. You will also enable your dictionary to learn, i.e. you will be able to add words to it.

Part I

The linear search is a particularly slow search method and will not do for a practical spell checker. To make your dictionary efficient you need to implement a much faster search method.

(a) [ 20 pts: c12e2s2t4 ] Change your `CDictionary` class to have an array of `CWord` pointers rather than an array of `CWords`. Write a member function `sort` for your `CDictionary` class that sorts the array of words using the quicksort algorithm discussed in class (you must implement quicksort yourself rather than use a library function). Does `sort` need to be part of the public interface? The fact that your dictionary has an array of `CWord` pointers makes swapping elements more efficient and hence improves the performance of `sort`. Test your `sort` routine fully to make sure it works properly. An unsorted dictionary file `dict.text` is provided for you in the project folder. You may want to add a member function `displaySelf` to make testing easier.

(b) [ 15 pts: c8e1s2t4 ] Now you can modify the member function `wordInDictionary` to use binary search on the (sorted) array of your dictionary. Test this function to make sure it works properly, both on words that are in the dictionary and on ones that are not (you may use the `sorted` dictionary file `sorteddict.text` for testing purposes in case you want to work on this problem before you implement your `sort` routine).

In a typical spell checker, one is able to add words to the dictionary. Your spell checker application will provide this feature by having two dictionaries, a main dictionary and a personal dictionary. The main dictionary gets its words
from some standard file and we do not plan to add any words to it. The personal
dictionary however should allow for the addition of new words.

(e) [ 20 pts: c12e1s2t6 ] Add a new constructor to the CDictionary class
that takes an integer argument representing the size of the dictionary. It should
allocate an array of the given size (but record the fact that there are no words
in the dictionary yet). Add another constructor that takes a file name and an
integer size and creates a dictionary of that size, reading its contents from the
file. Add a member addWord which takes a word as argument and adds the
word to the dictionary. Add another member addEntry which takes a string
and adds the corresponding word to the dictionary. Finally, add a member save
that takes a file name as argument and saves the contents of the dictionary to
that file so that it can be read again using your constructor. (Depending on
how you implement these member functions you may need to fine tune your
solutions for (a) and (b) so they work in harmony with your solution to this
problem.) Again a member function displaySelf will come in handy for testing
your functions.

Part II
[ 25 pts: c15e2s2t6 ]

Now you are ready to implement the spell checking application. First, get
a copy of the Spell Checker folder (inside the CS 116 folder). This folder
contains a bare bones version of the spell checker application. You should be
able to compile and run the application (you may get link warnings, but there
should not be any errors). Once the application is running, you will be able to
select from all the menus and enter file names, but nothing will happen. Note
that the mechanism for choosing files is not up to Macintosh standards (you
can’t select files, but have to type in their names. Sorry about that).

Your task now is to add to this skeleton and fully implement the spell checker.
In the Spell Checker folder, you will find two dummy headers CWord.h and
CDictionary.h. You should replace these with your own versions of these files
(the code assumes that you have called your classes CWord and CDictionary).
In addition to these headers, you’ll find the files SpellApp.h,
SpellAppConstants.h, SpellAppCore.cpp, and SpellAppStudent.cpp.

Devote some study to SpellApp.h which contains the definition for the
spelling application class. The class has members for the names of the various
dictionaries, the names of files used in spell checking, pointers to the main and
personal dictionaries, and input and output file streams. SpellAppCore.cpp
contains the implementation of the PowerPlant-specific parts of the system.
SpellAppConstants.h contains the definitions for constants used by the application
to refer to its interface components. You will not need to change anything
in these files. The file SpellAppStudent.cpp is where you will add code to the
system. This file currently contains “stubs” (do-nothing functions) for all of the
functions you will need to write.
The application should work as follows. The File menu has options for both a personal and main dictionary. The top part of the menu (New, Open, Close, Save, Save As, Revert) refers to the personal dictionary. The New option will create an empty dictionary, Open will load one from a file whose name is supplied by the user, Close will save and close the file, Save will save it (prompting for a name if it is a new dictionary), Save As prompts for a name and saves the file under that name, Revert reloads the dictionary from the file. Open Main and Close Main do what you would expect for the main dictionary, which cannot be changed by the user. Under the Edit menu is a single option Check File. This option will open a Spell Checker window, which prompts for an input file and an output file, and then shows the progress of the spell checking operation. When an unknown word is found, the spell checker pops up a dialog window that asks the user what to do. The options are: Skip, meaning the word is treated as though it were spelled correctly; Mark, meaning the word is marked as incorrect as in previous assignments; Add, meaning the word should be added to the personal dictionary; and Replace, meaning the user has modified the word and the new word should be inserted in place of the old.

Flesh out the stubs in SpellAppStudent.cpp and test your application extensively. You should make up your own text files which contain spelling errors and test your application on those.

**Extra Credit**

[ 10 pts: c6e1s1t2 ] Make the word comparisons case insensitive, proceeding as follows. Add a member function `isEqual` to your CWord class which takes a word and an optional parameter `case` of type `EComparisonType` as arguments. `EComparisonType` should be an enumerated type with two possible values: `ECaseSensitive` and `ECaseInsensitive`. `case` should default to `ECaseInsensitive`, and it should indicate how `isEqual` is to perform the comparison. `isEqual` should return `true` or `false` depending on whether the incoming word argument is equal to the host word object or not. Now use `isEqual` instead of the `==` operator in your dictionary’s sort and search routines.