Spell Checker

For this project you will implement a relatively simple spell checker for English text. The spell checker will incorporate a database of known words, built from a simple word list and organized for efficient searching.

The controller will read words from an input text file, one by one, and pass them to the spell checker for processing. If the spell checker finds an exact match, it will simply indicate the word is spelled correctly. If not, the spell checker will provide the controller with a (possibly empty) list of suggested spellings. To select suggestions, the spell checker will need some rules for deciding whether a match is close enough. Suppose that W is the word whose spelling is being checked, and that C is a candidate for the suggested spelling list. Your spell checker will add C to the list of suggestions according to the following rules (in the order given):

- If the lengths of W and C differ by more than two, do not suggest C.
- If W is a prefix of C, or if C is a prefix of W, then suggest C.
- Compare the characters of W to those of C, one by one, until reaching the end of one or both strings. If there are no more than two mismatches, and the number of mismatches is less than the lengths of both W and C, suggest C.

These rules are somewhat arbitrary, and certainly too simple for a real spell checker; that's acceptable for this assignment since the primary point is the data structure design, not the matching algorithm.

Data structures and other components:

The primary data structures element of this project holds the word list against which spelling will be checked. There are a number of good ways this could be handled. Your implementation is under the following specific requirements:

- The spell checker must store the word list in an AVL tree, as described in the course notes.
- The spell checker must be an object. It will contain an AVL tree, but it will not simply be an AVL tree.
- You must encapsulate the AVL tree, and its nodes, as C++ templates.
- The words must be stored in the tree as C++ `string` objects.
- The AVL tree interface must include only functions that are appropriate for a container, such as `insert()` and `find()`. The tree must not take any responsibilities that belong to the controller, such as reading in words to be checked, or to the spell checker itself, such as building the list of suggested alternative words.
- For testing and for the demos, your tree must have the ability to display itself to a specified output stream in the manner described in the notes for a BST. If you expect to receive help, be sure that your display function conforms to the formatting described in the course notes.

Your design must make appropriate use of classes. The specification may imply the existence of additional classes besides those already mentioned explicitly. Aside from node objects used only within an encapsulating class, data members of classes must be private.

The use of STL containers, such as the `vector`, is allowed for this project, for example as a way to encapsulate the list of suggestions when carrying out a spell check.

Program execution:

The program will be invoked with three command-line parameters:

```
spellcheck <command file> <dictionary file> <log file>
```

The program will verify the existence of the two input files; if either is missing, it will print an error message and exit.

Otherwise, the program will first open the dictionary file and add the words to the spell checker's database. Next, the program will open the command file, and process the commands one by one. Results will be written to the log file, as described below. There is one interesting constraint.
Thanks to the performance of the AVL tree, checking the spelling of a word is not a terribly expensive operation if the word is found. But if a word is not found, determining the list of suggestions is quite expensive. The rules given above do

not permit any real optimization of the search. It is necessary to traverse the entire AVL tree, comparing the misspelled word to each element of the tree. This has some consequences.

First, you must decide what traversal to apply when searching for substitutions. Any traversal would do, but some may be more efficient than others.

Second, there are issues relating to responsibility when the substitutions are being determined. Ideally the AVL tree storing the word list should not know anything about the data type it's storing, much less be responsible for defining a complex comparison algorithm involving those elements. That should be the responsibility of the spell checker. Obviously, the comparison to determine if a word will become a suggestion should be encapsulated within a function. The spell checker needs to apply that function to words stored in the AVL tree. But how can this be accomplished without violating the encapsulation of the tree, or violating the proper responsibilities of the spell checker and the tree?

There are several alternatives. This project requires that the AVL tree provide an iterator (supporting the chosen traversal pattern) so that the spellchecker object can simply manage the search itself. Since this does not require modifying the contents of the AVL tree, the spellchecker should, in fact, use a `const` iterator.

The dictionary file:

The word list file is simply a list of strings, one per line. There is no special parsing required. The controller should just read the words one by one and tell the spell checker to add them to its database. The AVL tree should not allow the insertion of duplicate entries, so if the dictionary file does contain duplicates, they should be filtered out automatically as the tree is being built.

The command file:

Lines beginning with a semicolon (`';'`) character are comments; your program will ignore comments. An arbitrary number of comment lines may occur in the input file.

Each non-comment line of the command file will specify one of the commands described below. Each line consists of a sequence of "tokens" which will be separated by single tab characters. A newline character will immediately follow the final "token" on each line.

The first two commands are basic searches and must be instrumented as described.

`check``<tab><word>`
: This causes the spellchecker to search the AVL dictionary as described earlier. What should be logged is described below.

`add``<tab><word>`
: This causes the specified word to be added to the AVL dictionary, unless it is a duplicate of course. A message indicating success or failure should be logged.

`remove``<tab><word>`
: This causes the specified word to be removed from the AVL dictionary, provided it is present of course. A message indicating success or failure should be logged.

`display``<tab>`
: This causes the AVL tree to be written to the log file, using the format specified for a BST in the notes.

The log file:

Since you will demonstrate your program to a TA, the specification of the log file is relatively loose (compared to the auto-graded assignments). The log file should begin with a section identifying you and listing the names of the three files your program will be using. After building the AVL tree, you should display the number of words it contains.

The main part of the log file consists of output from the commands. Commands should be echoed, numbered, and delimited as before. When searching for a word during a spell check, the AVL tree should be instrumented to log the sequence of words that were compared to the sought word along the way. For words that fail the spell check, you must also log the alternatives suggested by the spell checker. The formatting is up to you, but it should be neat and easily read.

Submitting Your Program:

You will submit a gzipped tar file containing your project to the Curator System (read the *Student Guide*), and it will be archived until you demo it for one of the TA. Instructions for submitting are contained in the *Student Guide*. You will find a list of the required contents for the zipped file on the course website. Follow the instructions there carefully; it is very common for students to suffer a loss of points (often major) because they failed to include the specified items.

Be very careful to include all the necessary source code files. It is amazingly common for students to omit required header or cpp files. In such a case, it is obviously impossible to perform a test of the submitted program unless the student is allowed to supply the missing files. When that happens, to be fair to other students, we must assess the late penalty that would apply at the time of the demo.

You will be allowed up to five submissions for this assignment, in case you need to correct mistakes. Test your program thoroughly before submitting it. If you discover an error you may fix it and make another submission. Your last submission will be graded, so fixing an error after the due date will result in a late penalty.

The *Student Guide* and link to the submission client can be found at: http://www.cs.vt.edu/curator/

Evaluation:

The TAs will be evaluating your source code on this assignment for programming style, so you should observe good practice. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

Your submitted program will be assigned a score based upon several factors. Your program will be evaluated for:

- the appropriateness of your class interfaces, and
- whether your design shows a good object-oriented decomposition of the given problem, and
- whether the internal documentation of your code is acceptable.

See the Programming Standards page on the course website for specific requirements that should be observed in this course.

Your program will also be evaluated for correctness by executing it with one or more sets of test data supplied at the demo. The demo data will conform to the specification, and will generally be very similar to the test data supplied on the course website. In almost all cases, if your implementation produces correct results with the posted data, and if you submit the right code, then you will not experience a lot of surprises at the demo.

Code changes during a demo are frowned upon. If it is absolutely essential that you make a change to your submitted implementation during the demo, the TA will assess the late penalty in effect at the time of the demo, and describe the change precisely in a comment on the demo checksheet. You may then appeal the penalty to me if you believe that the change is small enough to deserve a reduction in the penalty. In most cases, I will not reduce the penalty.

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided with the earlier project specifications in the header comment for your main source code file.