Assignment 9

Spell Checking Using Hash Tables

README Due: 9:00 PM, Tuesday, April 23, 2002 (*via* `cs70submit`)

Code Due: 9:00 PM, Thursday, April 25, 2002

1 Scenario

One of the most widely used spelling checkers in Unix-like environments is `ispell`, written by our own Prof. Kuenning.[1] You have decided to impress Prof. Kuenning by showing that you are also able to implement a spelling checker. Like `ispell`, your spelling checker will use hash tables, but unlike `ispell`, your program will be written in C++ (`ispell` is written in C). To show your prowess at writing reusable C++ code, you have decided to make your program use a templated hash-table class.

2 High Level Interface and Behavior

Your spelling checker, which is to be called `myspell`, will read text from standard input (`cin`) and report any words not present in a dictionary file, whose name is supplied on the command line.

The program begins by reading the dictionary into an internal hash table. It then reads words from standard input. Each word is looked up in the dictionary—words not in the dictionary are classified as being misspelled. Incorrectly spelled words are written to standard output in the order they are encountered, together with a list of suggested corrections.[2] (The algorithm for generating corrections is given in Section 4.)

Each line in the correction output should consist of the incorrect word followed by a colon (`:`) and zero or more corrections separated by spaces. There should be exactly one space after the colon (unless there are no corrections), and there should be no whitespace at the end of the line. The following are examples of valid output lines:

```
unix% myspell smalldict.words < error-filled-file.txt
xyzzy:
foo: for
wird: bird gird ward word wild wind wire wiry
```

1. Prof. Kuenning's `ispell` implementation is based on the first `ispell` implementation, written by Pace Willisson, and is more properly referred to as *International Ispell*.

2. This output format is similar to `ispell`'s `-a` mode.

Note: Your option processing code will not have to deal with the I/O redirection part of the command line (i.e., `< error-filled-file.txt`)—I/O redirection is handled for you, automatically, by all Unix shells, before your program is even executed.

If every word in the input is found in the dictionary, `myspell` should produce no output on cout. If a word is misspelled multiple times in the input file, it should only be printed once.

For the purposes of spell checking, a *word* is sequence of one or more characters for which isalpha is true. In the input text, words are separated by non-word characters. Although words in the input text may be mixed-case, your program will always convert words to lowercase (using tolower) before comparing them against words in the dictionary. The dictionary file is a sequence of unique lowercase words separated by whitespace.

To provide useful information about hash-table performance, `myspell` must also support a `-d` option that causes your program to print certain statistical information to cerr just after it has finished reading the dictionary. The statistical output should be in the following format:

n `expansions, load factor` *f*`,` *c* `collisions, longest run` *l*

where *n*, *c*, and *l* are integers, and *f* is a floating-point (*double*) number, printed in the default format. This information corresponds to (or is easily calculated from) statistical information provided by the *HashSet* template class described in the next section.

If the program is given invalid arguments, it should produce appropriate error messages rather than crash. The choice of error message is up to you.

3 Required Classes and Functions

3.1 Required *HashSet* Template Class

Your code must provide a template class *HashSet* (declared in `hashset.hpp`) such that *HashSet<string>* can store a hash table of strings. The *HashSet* class can assume that if a programmer creates an instance of the template *HashSet<Food>* (using *Food* as an example) they will have written a global function *size_t* hash(***const*** Food& food) that provides hash values for *Food*s using the full range of the size_t type (*size_t* is a synonym for *unsigned int*, defined by #include <cstddef>, with values ranging from 0 to $2^{32}-1$ on 32-bit machines).

An instance of *HashSet*, on some type *T* (i.e., *HashSet<T>*), will provide the following operations

- A default constructor
- A destructor that leaves the object in a sane (empty) state

- A member function *size_t* size() ***const*** that returns the size of the hash table
- A member function *void* insert(***const*** *T&*), where h.insert(x) adds x to the hash table, where the function's behavior is undefined if x has already been added to the table
- A member function *bool* exists(***const*** *T&*) ***const***, where h.exists(x) returns true if x is present in the hash table and false otherwise
- A member function *size_t* buckets() ***const*** that returns the number of buckets in the hash table
- A member function *size_t* reallocations() ***const*** that returns the number of times the hash table has resized itself
- A member function *size_t* maximal() ***const*** that returns the length of the longest chain (in the case of separate chaining) or cluster (in the case of open addressing) discovered so far in the hash table[3]

(The type *T* is used for illustrative purposes—you may not use such a short name for the template type argument in your template class.)

All of the above functions must execute in $O(1)$ expected amortized time.

Note that the last three functions provide statistics about the hash table—buckets and maximal will change whenever the hash table needs to be expanded.

You are *not* required to provide a copy constructor, assignment operator, a function to print the hash table, a swap function, or an erase function. You are not required to write an iterator for this class. If copy construction and assignment are not supported, they must be disabled.

Your hash table will need to grow as necessary to hold its contents while keeping the load factor low enough to provide good performance. It is up to you to decide how to handle collisions—separate chaining, linear probing, quadratic probing, or double hashing. If maximal returns a value larger than about 20, it is a sign of serious problems with your hash table, either because your program is not growing the hash table appropriately, or because of serious problems with your hash function.

Your implementation may use any data type from the STL as well as the *slist* type, which is an SGI extension to the standard STL present in our STL library. For obvious reasons, you may not use the *hash_set* or *hash_map* SGI extensions present in our STL library, except (perhaps) for testing.

3.2 Required hash Function

You must also provide a function to hash strings, with the following signature:

3. The "discovered so far" requirement is there because it may not be easy to know the actual largest cluster size without scanning the entire table. You may wish to think about why.

```
size_t hash(const string& s)
```

It must be declared in the header stringhash.hpp and defined in the file stringhash.cpp. The header file for your *HashSet* template class should *not* automatically include this header file.

You are not expected to devise your own hashing strategy, and may copy code from *any* source to make your hash function, with the exception of the work of prior CS 70 students and complete hash-table implementations. You may find appropriate functions given in Weiss. Your hash function should be a good one, however.

You should also test your code with a *very bad* hash function. We will.

4 Generating Spelling Corrections

The easiest way to generate corrections in a spelling checker is by trial and error. If we assume that the misspelled word contains only a single error, we can try all possible corrections and look each up in the dictionary.

Traditionally, spelling checkers have looked for four possible errors: a wrong letter ("`wird`"), an inserted letter ("`woprd`"), a omitted letter ("`wrd`"), or a pair of adjacent transposed letters ("`wrod`"). To simplify this assignment, you will only need to deal with the first possibility: a wrong letter. When a word isn't found in the dictionary, you will need to look up all variants that can be generated by changing one letter. For example, given "`wird`", you should look up "`aird`", "`bird`", "`cird`", ..., "`zird`", then "`ward`", "`wbrd`", "`wcrd`", ..., "`wzrd`", and so forth. Whenever you find a match in the dictionary, you should add it to your output line.

5 Sample Dictionaries

You may wish to create a very small sample dictionary of your own for initial testing. A slightly larger dictionary of 341 words (`simple-dict.words`) should help you to get most of your bugs out. When you're fairly confident, you can try your luck with over 34,000 words in an all-lowercase version of the `ispell` dictionary, `ispell.words`. Both dictionaries are available for download from the Homework area of the course website.

6 Tricky Stuff

As usual, there are parts of this assignment that are tricky. Here are a few:

- When you are developing a hash table, it is wise to start your debugging with a dummy hash function that always returns zero. Once you are sure your collision handling works correctly, you can write a real hash function.

- If you get excessive numbers of collisions, be sure your hash function is returning values that are well spread out. Test it with "nearby" values such as `aaa`, `aab`, `baa`, and so forth.
- Remember that the hash functions can return a number larger than the table size (except for `hashStringBase256`). You must reduce the hash value yourself to make sure you don't go beyond your array bounds.
- Remember that when you expand your hash table, you must rehash everything in the current table, because the wrapping due to the `modulo` function will change. If you are using separate chaining, don't forget to follow your collision chains appropriately.
- Similarly, you may be tempted to use the *vector* class from the library to manage your hash buckets. Although it is possible to do so effectively, doing so is trickier than it might first appear. In particular, the `resize` function is *not* appropriate for resizing hash tables. We recommend that rather than using *vector*, you simply manage the array of hash buckets yourself.