Project 5: Hash Tables and Its Applications

Due: 04/17/2015

Educational Objectives: Understand and get familiar with the data structure hash tables, and its applications in spell check.

Statement of Work: Implement a hash table ADT and other supporting user interfaces; develop a simple spell checker.

Project Description:

This project contains two parts. In the first part of the project, you need to implement a hash table class template named HashTable. In the second part of the project, you will develop a simple spell checker using the hash table you developed.

Task 1: Requirements of HashTable Class Template

- Your implementation of HashTable must be in the namespace of cop4530.
- You must provide the template declaration and implementation in two different files hashtable.h (containing HashTable class template declaration) and hashtable.hpp (containing the implementation of member functions). You must include hashtable.hpp inside hashtable.h as we have done in the previous projects. The two files hashtable.h and hashtable.hpp will be provided to you, which contain some helpful functions that you will need to use in developing the hash table class template.
- You must implement hash table using the technique of chaining with separate lists (separate chaining). That is, the internal data structure of the hash table class template should be a vector of lists. You must use the STL containers for the internal data structure instead of any containers you developed in the previous projects.
- You must at least implement all the interfaces specified below for the HashTable class template.
-

Public HashTable interfaces (T is the template parameter, i.e., the generic data type)

- **HashTable(size_t size = 101)**: constructor. Create a hash table, where the size of the vector is set to prime_below(size) (where size is default to 101), where prime_below() is a private member function of the HashTable and provided to you.
- **~HashTable()**: destructor. Delete all elements in hash table.
- **bool contains(const T &x)**: check if x is in the hash table.
- **bool insert(const T & x):** add x into the hash table. Don't add if x is already in the hash table. Return true if x is inserted; return false otherwise.
- **bool insert (T &&x)**: move version of insert.
- **bool remove(const T &x)**: delete x if it is in the hash table. Return true if x is deleted, return false otherwise.
- **void clear()**: delete all elements in the hash table
- **bool load(const char *filename)**: load the content of the file with name filename into the hash table. In the file, each line contains a single value to be inserted to the hash table.
- **void dump()**: display all entries in the hash table (see the provided executable for the output format).
- **bool write_to_file(const char *filename)**: write all elements in the hash table into a file with name filename.

Private HashTable interfaces (T is the template parameter)

- **void makeEmpty()**: delete all elements in the hash table. The public interface clear() will call this function.
- **void rehash()**: Rehash function. Called when the number of elements in the hash table is greater than the size of the vector.
- **size_t myhash(const T &x)**: return the index of the vector entry where x should be stored.
- **unsigned long prime_below (unsigned long)** and **void setPrimes(vector<unsigned long>&)**: two helpful functions to determine the proper prime numbers used in setting up the vector size. Whenever you need to set hash table to a new size "sz", call prime_below(sz) to determine the new proper underlying vector size. These two functions have been provided in hashtable.h and hashtable.hpp.

You need to write a simple test program to test various functions of hash table. More details are provided in a later part of this description.

Task 2: A Simple Spell Checker

In the second part of the project, you will develop a simple spell checker. The spell checker will take three command-line parameters: filename of the dictionary, name of the file to be checked (check file), name of the output file where the checked (and may be corrected) content should be written to.

- For the dictionary file, we assume that each line contains a single word. You should load the words in the dictionary into a hash table (using the hash table you developed).
- For the check file, we make the following assumptions: 1) It is a text file; 2) Each line may contain multiple words; 3) Words in the file will not span multiple lines (no wrap back); 4) A line may contain multiple misspelled words.

Your spell checker will read the content of the check file line by line; one line at a time. After a line is read, your program will check word by word in the line if it is a valid word (i.e., if it is in the hash table), where a word is defined as a sequence of English letters from 'a' to 'z' (and the capital cases). When an invalid word is encountered (i.e., the word is not in the hash table), you will print out the line with the invalid word CAPITALIZED, you will then output 10 candidate words for the user to choose. In this project, you will identify the 10 candidate words in a simple way: for each character in the word, you will replace it with characters from 'a' to 'z', one at a time, starting from the left-most character.

After a replacement is done, you will check if the new word is in the hash table. It becomes a candidate word if it is in the hash table. After you have identified 10 candidate words, you do not need to search for additional candidate words. As an example, consider an invalid word "jest", we will create new words with replacement such as "aest", "best", "cest" ..., "zest" (the first letter "j" is replaced by 'a' to 'z', one at a time), and "jast", "jbst", "jcst", ..., "jzst" (the second letter 'e' is replaced by 'a' to 'z', one at a time), etc. For each created word, you will check if it is in the hash table (a valid word), and if so, counted as a candidate word. You stop this process after you have 10 candidate words.

After you have identified 10 candidate words, you output these 10 candidate words to the user, who will choose one (or choose 'n' for no change). Note that each line may contain multiple invalid words, you need to process them one by one in the same manner. If there are less than 10 candidate words, you should just output these candidate words.

At the end of your program (after the spell check is finished), you should store the content of the text file, with invalid words corrected (or no change due to user selection), into the output file. You still need to write the content of the text file into the output file, even if all words in the original text file are valid.

A few notes for Task 2:

- You can assume the dictionary only contains lower-case words. Put in another way, you should convert a word in the check file to lower-case before you check if it is in the dictionary (hash table).
- You can assume that a word in the check file will only contain alphabetical letters. There will be no special characters such as apostrophe (') in the middle of a word. However, we may have punctuation mark at the end of a word such as comma and period, which is not part of the word and should be ignored when checking against the dictionary (hash table).

A partially implemented driver program named myspell.cpp is provided to you. This program works in two modes. When the program runs without any command-line parameters, it will test the functions of the hash table. For this purpose, a menu() function is provided in myspell.cpp. You should call this function to display the operations that a user can perform on a hash table. You cannot change the menu() function.

In the second mode of the program, three command-line parameters will be provided, and the program will work as a spell checker.

Extra-credit part (up to 20 points)

Develop a nice graphical user interface (GUI) for this project. Your GUI must support at least the user interfaces supported by the plain-text version user manual in the provided code. You can develop this part on Windows (10 bonus points) or you can work on a Linux machine such as linprog.cs.fsu.edu (20 bonus points). There are a number of choices for you to develop a GUI on a Linux machine, including Qt and GTKMM, and wxWidgets. Note that, you need to complete the required components of the project before working on the GUI part. You will not get any bonus points if you only have a GUI but cannot perform the required functions of this project.

- Note that, even if you decide to develop the GUI on Windows, you still need to have the required components of the project work on linprog.cs.fsu.edu. The TA will grade the required components of the project on linprog.cs.fsu.edu. If you work on the extra-credit part on Windows, you will need to set up an appointment with the TA to grade the extra-credit part. Similarly, if you develop the GUI on your own Linux machine, you will need to set up an appointment with the TA to grade it.

Provided Partial Code

The following partial code has been provided to you.

1. hashtable.h : partial implementation

2. hashtable.hpp : partial implementation

3. myspell.cpp: partial implementation

4. proj5.x: executable of myspell.cpp, compiled on linprog.cs.fsu.edu

5. test1.txt: a sample check file.

For testing your program, you can use /usr/share/dict/words as the dictionary on linprog.cs.fsu.edu

Deliverables

1. Tar your hashtable.h, hashtable.hpp, myspell.cpp, and makefile in a tar file, and submit on blackboard.

2. Set up an appointment with the TA if you implement the GUI and it does not work on linprog.cs.fsu.edu. If your extra-credit part is carried out on linprog.cs.fsu.edu, you should send an email to the TA to inform her or him about this so that she or he will grade it.

3. Note that, your extra-credit part must work completely, we will not grade any partial implementation, for example, the code that cannot be compiled or cannot run. No partial credits are given for this part.

One bonus point will be given to the first student reporting an error in the provided executable code proj5.x.