Linköping Institute of Technology
Department of Science and Technology/ITN
Aida Nordman

07/03/2014

Data Structures
TND004
Lab 2

Course goals

- To implement the data structures and algorithms for a spell checker. More concretely, hash tables are one of the data structures in focus in this exercise.
- To analyze the tradeoffs, regarding efficiency in several aspects, of different data structures proposed for addressing a practical problem.
- To determine the time complexity of the different operations performed on the chosen data structures, as well as to motivate their choice (e.g. advantages and limitations).

Preparation

In this lab exercise you are required to implement a spell checker. The spell checker was discussed in the end of Fö 3 as an application of hash tables. Thus, you may want to review Fö 3 once more. Moreover, sections 5.1 to 5.3 and 5.5 of the course book are relevant for the exercises in this lab.

You are also requested to study the documentation of the given classes (**`Item`**, **`HashTable`**, and **`SpellChecker`**) before the lab session. This documentation is available from the course web page.

Presenting solutions

You should demonstrate your solutions orally during your third lab session. You also need to deliver a written paper with the answers to exercise 2. This paper should be provided during the oral demonstration of your program. Hand written answers that we cannot understand are simply ignored. Do not forget to put the name and LiU login of the group members in the delivered paper.

Use of global variables is not allowed, but global constants are accepted.

If you have any specific question about the exercises, then drop me an e-mail. Be short and concrete, otherwise you won't get a quick answer. You can write your e-mail in Swedish. Add the course code to the e-mail's subject, i.e. "**TND004:** …".

Exercise 1: the spell checker

You are requested to implement the spell checker discussed in Fö 3. A spell checker is a program that, given a dictionary and a text file, outputs all the misspelled words occurring in the text file. A misspelling is any word that does not occur in the dictionary. Note that words in a text file may contain lower and upper case letters, digits, and punctuation signs (e.g. '**`?`**', '**`,`**', '**`.`**').

Your program should have the following functionalities.

- **To create a log file with a list of all misspellings** occurring in a given text file, together with the number of occurrences of each misspelling.
- **To add new words to the dictionary**. The motivation for this functionality is that a text may contain specific words (e.g. names of persons, names of places, technical terms) that, in spite of the fact of not belonging to the dictionary, should not be considered as misspellings. New words can be added to the dictionary before starting to check a new text for misspellings. These new words are also provided in a text file, one word per line. Note this is an extra functionality that was not discussed in Fö 3 but it is requested in this exercise.
- **To remove the new added words from the dictionary**. This operation should be available before another text file is spell checked. The idea is basically to reset the dictionary to its initial content.

Note that the program should allow the user to spell check several text files, one after the other, before the user decides to exit the program. For each text file, a log file is created with a list of the misspellings. The log files are named by appending the name of the text file to the string "**`log_`**", i.e. if the text file to be spell checked is called "**`mesg.txt`**" then the corresponding log file created by the program is named "**`log_mesg.txt`**".

An English dictionary is initially provided as a text file, such as **`engDict.txt`**, with one word per line. This dictionary has about **49200** words, all in lower case letters. Every time the program starts, the dictionary is loaded in main memory.

Your implementation should have in mind that time efficiency is an important aspect for the user. The data structures used to implement this application should follow the ideas discussed in the third lecture. Thus, the **dictionary should be implemented as a hash table using separate chaining technique** for handling collisions.

As discussed in Fö 3, the data structures used to solve this problem (e.g. hash table, list) share the items. Recall that each item stores a word and a counter. This means that an item can be accessible from several data structures. This aspect can be implemented in C++ by storing pointers to the items in the data structures (instead, of storing the items directly in the data structures). In this way, it is possible to have several pointers in different data structures pointing to the same item.

Start by downloading all needed files from the course web page. All given code is documented in the form of html pages[1]. You can also find a link to these html pages from the course web page.

Secondly, study carefully the documentation for the given classes.

- Class **`Item`**. File **`item.h`** contains the class definition and its implementation can be found in the file **`item.cpp`**. An item stores a word (a **`string`**) and a counter (a **`short`**).
- Class **`HashTable`**. File **`hashTable.h`** contains the class definition, including data members[2]. Some of the member functions are not yet implemented (obviously, this is part of your job). Note that lists are used to store (pointers to) items that collide, i.e. you are supposed to implement a hash table with **separate chaining**. The

[1] These pages were automatically generated with doxygen.
[2] In the documentation data members are called "attributes".

`std::list` container from the STL library of C++ is used to implement the collision lists. Although hash tables are part of the standard library of C++11, the **`std::unordered_map`** container, you are not supposed to use this container.

- Class **`SpellChecker`**. File **`spellChecker.h`** contains the (incomplete) class definition. Objects of this class have a data member that is an instance of class **`HashTable`**, representing a dictionary, in addition to the number of words in the dictionary plus the misspellings list. You may need to add other data members to this class. Some of the member functions are not yet implemented (obviously, this is also part of your job).

After having studied the documentation of the given code, you should then follow the steps indicated below to proceed to the implementation.

Steps to follow: implementation and testing

1. Run the executable program named **`Speller.exe`**, provided in the downloadable zipped folder. This program gives you an idea of how your final program should look like and how the described functionalities are provided. You can use it to spell check text files in English (e.g. **`mesg.txt`** and **`mesg1.txt`**). This executable only runs in PCs under Windows operating system. You can also perform the tests in steps 9 to 11 with this executable.
2. Create a project with the files **`Item.h`**, **`Item.cpp`**, **`HashTable.h`**, **`Hashtable.cpp`**, and **`test_HashTable\test1.cpp`**. The file **`test1.cpp`** contains a small test program to test the implementation of the **`HashTable`** class. It starts by creating a hash table representing a dictionary and then checks which words of a given text file belong to the dictionary.
3. Implement the member functions of class **`HashTable`**.
4. Compile the program and then execute it. To this end, use the text file **`test_HashTable\mesg.txt`** and **`test_HashTable\mesg1.txt`**. The expected output is available in the files **`test_HashTable\log_mesg.txt`** and **`test_HashTable\log_mesg1.txt`**, respectively.
5. Remove the file **`test1.cpp`** from the project and add instead the file **`test_HashTable\test2.cpp`**. The file **`test2.cpp`** contains another small test program to test your code for class **`HashTable`** (more concretely the functions **`insert`**, **`remove`**, and **`find)`**. The expected output is available in the file **`test_HashTable\test2_out.txt`**.
6. Remove the file **`test1.cpp`** from the project and add the files **`main.cpp`**, **`spellChecker.h`**, and **`spellChecker.cpp`**.
7. Implement the member functions of class **`SpellChecker`**. When creating the dictionary, you should set the max load factor allowed in the dictionary (constant **`MAX_LOAD`**) before rehashing to **`2`**. A hash function, based on polynomial accumulation (see pag. 213 of course book, fig. 5.4), is already provided in the file **`spellChecker.cpp`**.
8. Compile separately the source file **`SpellChecker.cpp`** (use **`Ctrl-Shift-F9`**) and correct any compilation errors.

9. Create an executable and run your application. Test it with the text files **`text_files\mesg.txt`** and **`text_files\mesg1.txt`**. The expected output is available in the files **`text_files\log_mesg.txt`** and **`text_files\log_mesg1.txt`**, respectively.
10. Add all the words in the file **`text_files\newWords1.txt`** to the dictionary. Spell check again the text in **`mesg1.txt`** (the log file obtained should now be empty).
11. Remove the words added, in the previous step, from the dictionary and spell check the text in **`text_files\bible.txt`**. The expected output is available in the file **`text_files\log_bible.txt`**. Then, add the words in the file **`text_files\newWords.txt`** to the dictionary and spell check again **`bible.txt`**. Remove the added words from the dictionary.
12. To test rehashing, load the dictionary in a hash table with a number of slots that is **1/3** of the number of words in the dictionary file. Then, spell check **`text_files\bible.txt`**. Make sure that your code displays a message when rehashing occurs, indicating also the load factor of the dictionary after rehashing.

Exercise 2: analysis and efficiency

Deliver a written paper with your answers to the points below. This information should be provided during the oral demonstration of your program.

- Description of the data structures used in your program and how they are connected. A figure, with some brief text, can certainly help.
- Size of the hash table, i.e. number of slots in the table.
- Analysis of the time complexity of the following operations, in the average case. Use Big_Oh notation to express the time complexity and motivate clearly your statements.
    - Spell check a word **`w`** that is a misspelling encountered earlier in the text, i.e. not a new misspelling.
    - Remove any extra added words from the dictionary, i.e. reset the dictionary to its initial content.
- Indicate three negative aspects of the current data structures.

Lycka till!