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

15/03/2015

Data Structures
TND004
Lab 2

Course goals

- To implement an **open addressing hash table using linear probing** strategy to resolve collisions.
- To understand the importance of a good strategy to handle collisions in hash tables.

Preparation

Review Fö 4, where hash tables were introduced. Moreover, read sections 5.1 to 5.5 of the course book.

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

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:** …".

Presenting solutions

You should demonstrate your solutions orally during your third lab session on week 18. You also need to deliver a written paper with the answers to specific questions indicated in this lab description. This paper should be provided during the oral demonstration of your code. 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.

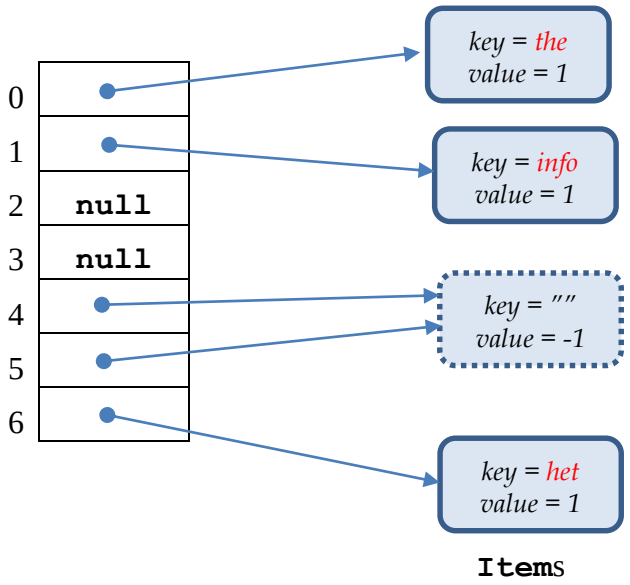
Deadline

Recall that the final deadline to present lab 1 and 2 is your lab session on week 20. Since you can only present one lab in each lab session, you must present one of these two labs before week 20.

Exercise 1: to implement an open addressing hash table with linear probing

The aim of this exercise is that you implement a simple open addressing hash table that stores elements consisting of a unique key (a **`string`**) and an associated value (an **`int`**). The elements stored in the hash table are instances of class **`Item`**.

Class **`HashTable`** represents open addressing hash tables with linear probing strategy to handle collisions. A hash table is represented as an array of pointers to items (i.e. instances of class **`Item`**). You can find below a figure that illustrates a hash table.



The hash table has seven slots and stores three items, with keys "*the*" (in slot 0), "*info*" (in slot 1), and "*het*" (in slot 6), all associated with value "*1*". Slots 2 and 3 have never been used to store an item, while slots 4 and 5 are marked as "*deleted*". As you can see in the figure above, slots marked as "*deleted*" store a pointer to a specific item (an instance of **`Delete_Item`**), with key equal to an empty string and value *-1*.

Recall that in a hash table with open addressing, it is needed to distinguish between empty slots that have never been occupied from empty slots that had stored items that were removed afterwards from the table. Otherwise, the search algorithm may fail to find an element that actually is in the table.

Steps to follow

1. Download all needed files from the course web site.
2. Build a project with the files **`Item.h`**, **`Item.cpp`**, **`hashTable.h`**, **`hashTable.cpp`**, and **`test.cpp`**. You can also compile, link, and execute the given code, although it will not do anything useful, yet.
3. Study the classes' documentation[1], available from the course web site. To see the source code, without the doxygen specific comments, you can just click on "**`Files`**", then select a source file from the list of files, and click on "**`Go to the source code of this file`**".

[1] These pages were automatically generated with doxygen.

4. For class **HashTable** implement the constructor, destructor, and the member functions **insert**, **find**, **remove**, **rehash**, and **operator<<**. For each member function, the implementation you provide must conform to the given specification. Do not modify the given code, otherwise. The functions to be implemented are marked with "**//IMPLEMENT**".
5. Test incrementally your code with the program given in **test.cpp**. You can find a running example of the program, including the expected output, in the file **test_out.txt**. When the table is displayed (option 4), the numbers between curl braces indicate the hash value of the key.

Exercise 1.1

You are now going to redo an old exercise from the TNG033 course, lab 4. In that exercise, you were asked to write a program that displays a frequency table for the words in a given text file. The words in this table contain only lower-case words. Thus, all upper case letters were transformed to lower case letters and all punctuation signs had to be removed.

In the input text file, a word may contain letters and numbers but not punctuation signs (**.,!?:\"();**).Words are separated by white spaces.

In the TNG033 lab, you used a **STL::map** to represent the frequency table, where the key is a word (**string**) and the value associated with each key is a counter (**int**) of the number of occurrences of the key in the given text. Your program had probably a structure similar to the one below (assuming the text is read from the standard input).

```
int main()
{
    map<string, int> freq_table;
    string word;

    while (cin >> word)
    {
       //transform word to lower-case letters and
       //remove punctuation signs

       freq_table[word]++;
    }

   //Display frequency table

   return 0;
}
```

In this lab (for TND004), the frequency table (**freq_table**) should be an instance of the class **HashTable** that you have just implemented, instead of a **STL::map** container.

Steps to follow

1. Add a subscript operator to the class **HashTable**, i.e. **operator[]**, that has a behaviour similar to the subscript operator provided by the container **STL::map**.
2. Test the subscript operator by adding a specific user option to the test program **test.cpp**.
3. Write a program that displays a frequency table for a given text file, as described above. Your program should also display the number of words read from the text file and the

number of unique words. Moreover, the hash function to be used in this exercise is the one suggested in fig. 5.4 of the course book (pag. 213).

4. Test your program with the text files **`test_file1.txt`** and **`test_file2.txt`**. The expected output can be found in **`file1_out.txt`** and **`file2_out.txt`**, respectively.

Questions to answer

Write the answer to the following questions and deliver your written answers to the lab assistant, when you present this lab. The answers to the questions must be expressed in your own words.

1. Indicate the advantages of using the hash function of fig. 5.4 of the course book, instead of the hash function of the program in **`test.cpp`**.
2. Is it possible to display a frequency table, implemented as an instance of class **`HashTable`**, sorted by the key (as in the lab of TNG033 course)? Motivate your answer.

Exercise 2: The birthday paradox and hash tables

The question you are going to look into now is how often there are collisions in a hash table.

The program in **`collisions.cpp`** simulates insertions in a hash table of size 1000003 and estimates the probability of at least one collision during the insertion of 50, 100, to 10000 elements in the table. The program subsumes a hash functions that distributes the keys evenly. Moreover, the program outputs the probability of (at least one) collision for each set of insertions of 50, 100, to 10000 elements.

Study the code in **`collisions.cpp`** and use its output to create a graph of the estimated probability of at least one collision. The graph can be created with gnuplot, matlab, or any other tool at your choice. Then, write the answer to the following questions and deliver your written answers to the lab assistant together with the graph, when you present the lab.

1. What is the probability of at least one collision when 2500 elements are inserted in the table?
2. How many elements need to be inserted in the table for the probability of (at least one) collision to be at least **`50%`**?
3. Investigate the meaning of the birthday paradox and describe it in your own words.
4. How does the birthday paradox relate to hash tables?

Lycka till!