Linköping Institute of Technology 20/04/2016
Department of Science and Technology/ITN
Aida Nordman

Data Structures
TND004
Lab 2

Goals

- To implement an **open addressing hash table using linear probing** strategy to resolve collisions.

Prologue

In contrast to lab 1, built-in pointers, **`new`** and **`delete`** operations are used in this lab. Template classes are also used, though for simplicity move semantics is not addressed.

Section 5.4.2 of the course book proposes an implementation for open addressing hash tables. There are however some important differences between the implementation requested in this lab and the one used in the book. Some of these differences are pointed out below.

- This lab aims to create a simplified version of the STL container **`std::unordered_map`**[1]. To this end, the template class used to define hash tables in this lab has two parameters, the type of the key and the type of the value associated with the keys. The template class **`HashTable`** in fig. 5.14 of the book has only one parameter representing a "*hashed object*" type, i.e. an object for which there is a hash function.
- Quadratic probing is used as strategy to resolve collisions in section 5.4.2 of the course book, while linear probing will be used in this lab.
- Another important difference is that the book uses a **`vector`** to implement the hash table and no dynamic memory allocation is directly used in the code. In this lab, the hash table is implemented as an array of pointers (to items) and dynamic memory allocation is needed.

Preparation

You can find below a list of tasks that you need to do before the **HA** lab session on week 17.

- Review Fö 4, where hash tables were introduced.
- Read sections 5.1 to 5.5 of the course book. In spite of the differences pointed above, the algorithms implemented in the book, specifically section 5.4.2, do not differ much from the ones you need to implement in this lab.
- Study the code given in the files **`Item.h`** and **`hashTable.h`** that contain the template classes **`Item`** and **`HashTable`**, respectively. These files can be downloaded from the course website.
- Do exercise 1.

[1] Note that the container **`std::unordered_map`** is also implemented as a hash table.

Presenting solutions and deadline

You should demonstrate your solution orally during your **RE** lab session on **week 18**. After week 18, if your solution for lab 2 has not been approved then it is considered a late lab. Note that a late lab can be presented provided there is time in a **RE** lab session.

The necessary requirements for approving your lab are given below.

- Use of global variables is not allowed, but global constants are accepted.
- Readable and well-indented code. Note that complicated functions and over-repeated code make programs quite unreadable and prone to bugs.
- There are no memory leaks. Recall that you can use one of the tools suggest in lab 1.
- The code must satisfy the performance conditions described in point 3 of exercise 2.
- Public interface of class **`HashTable`** cannot be modified.
- Have a copy of the file **`lab2_table.pdf`** with all tables filled in (see exercise 2).
- Bring a written answer to the questions in exercise 3, with indication of the name plus LiU login of each group member. You'll need to discuss your answers with the lab assistant who in turn will give you feedback. Hand written answers that we cannot understand are simply ignored.

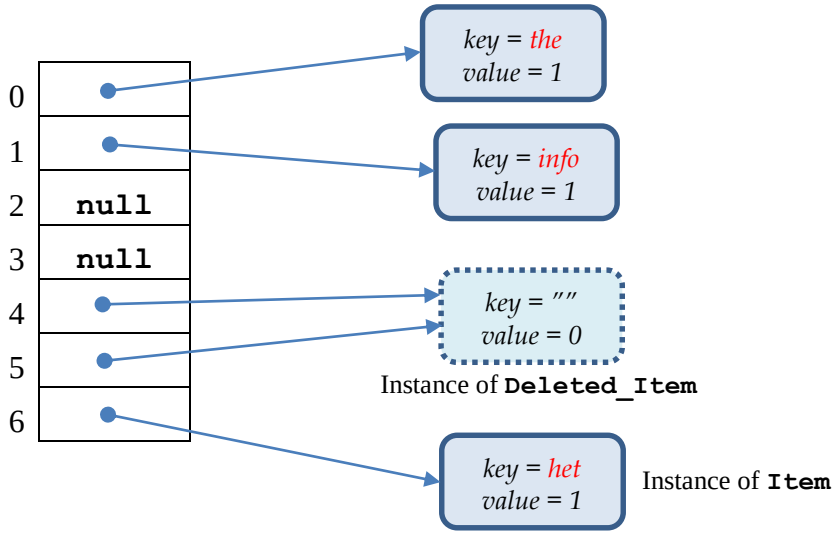
If you have any specific question about the exercises, then send us an e-mail. Be short and concrete, otherwise you won't get a quick answer. You can write your e-mail in Swedish. Add the course code to the e-mail's subject, i.e. "**TND004:** ...".

Finally, presenting your solution to the lab exercises can only be done in the **RE** sessions.

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

The aim of this exercise is to implement a simple open addressing hash table that stores elements consisting of a unique key and an associated value. The elements stored in the hash table are instances of (template) class **`Item`**. An item (instance of class **`Item`**) is basically a pair consisting of a key and the value associated with the key.

Template class **`HashTable`** represents open addressing hash tables with linear probing strategy to handle collisions. A hash table is implemented as an array of pointers to items (i.e. instances of class **`Item`**). You can find below a figure that illustrates a hash table, where the key of each item is a **`string`** and the associated value is an **`int`**.



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 one specific item (an instance of **`Delete_Item`**).

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.

Recall that insertion of an item (i.e. with a key not yet existing in the table) always starts by a search. A new item is inserted in the same slot where the unsuccessful search ended.

For calculating the load factor of the table, slots marked as "*deleted*" count as occupied slots. When the load factor of the table reaches at least 50% then re-hash should occur.

Template class **`HashTable`** has two extra data members: one data member (named **`total_visited_slots`**) to count the total number of slots visited[2]; and, another (named **`count_new_items`**) to count the total number of dynamic memory allocation of items (the instance of **`Delete_Item`** is not counted).

[2] This counter is incremented every time a slot is visited during an insert, find, or remove operation.

Steps to follow

1. Download all needed files from the course website.
2. Build a project with the files **`Item.h`**, **`hashTable.h`**, and **`test.cpp`**. You can also compile, link, and execute the given code, although it will not do anything useful, yet.
3. Study the given code.
    - The template class **`Item`** and its derived class **`Delete_Item`** are fully implemented and their code cannot be modified.
    - The definition of template class **`HashTable`** is provided, though most of its member functions are not yet implemented. Note that you can neither modify the public interface of this class neither modify the given data members. However, you can add non-public auxiliary member functions, if needed.
4. Implement the required member functions of template class **`HashTable`**. 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 2

We are going to consider again 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.

A word may contain letters and numbers but no punctuation signs (**`.,!?:\"();`**).

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**) is an instance of the class **HashTable<string,int>**, instead of a **STL::map** container.

Steps to follow

1. Add a subscript operator to the template 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 with the test program **main.cpp**. Note that the hash function used in this exercise is the one suggested in fig. 5.4 of the course book (pag. 213). Use the text files **test_file1.txt**, **test_file2.txt**, and **test_file3.txt** and fill in the second and third tables in the file **lab2_table.pdf**. Do not forget to modify the initial size of the hash tables, as indicated in the tables of the **pdf** file.
3. For the data you have entered in the second table, check the following.
   - If the "N. of calls to ***new*** Item()" is greater than the "N. of unique words" then revise your code, since there is space for efficiency improvement by reducing the number calls to **new** (i.e. dynamic allocation of instances of **Item**). Note that dynamic memory allocation (i.e. calling **new**) is a time consuming operation.
   - If the "N. of slots visited" in table 2 is greater than the value in the same cell of table 1 then revise your code, since there is space for efficiency improvement by reducing the number of slots probed.
4. Compare the frequency tables your program generates with the expected ones (**out_test_file1.txt**, **out_test_file2.txt**, and **out_test_file3.txt**).

Exercise 3

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 advantage(s) of using the hash function of fig. 5.4 of the course book (also used in **main.cpp**), instead of the hash function of the program in **test.cpp**.
2. What is the main difference between **std::map** and the hash table you have implemented in this lab?

Lycka till!