BLG 335E - Analysis of Algorithms I, Fall 2017

Project 3

Total worth: 7.5% of your grade

Handed out: 17 Nov 2017, Friday

Due: 1 Dec 2017 Friday - 23:59

Overview

Although essentially a basic method for compressing large-sized data to small-sized keys, **hashing** is a procedure commonly employed in computer science due to the utility it provides for a variety of applications. While complex hash functions can be designed to accommodate sophisticated algorithms that optimize relational databases or enable cryptographic methods for digital signatures, the most basic standard hash functions are still efficient enough to allow the implementation of **associative arrays**. By now, every major programming language has an efficient implementation for an associative array, usually called a **dictionary** (*e.g*. Python, .NET) or a **map** (e.g. C++, Java).

# # # # # # # # # # # # # #

You will implement **insertion** and **lookup** routines for a **dictionary** and a basic **list**, and take measurements for performance comparison. You are provided a data set with 131,070 items, each representing a character in Simone de Beauvoir's book[1], *The Second Sex*. The same set of items is arranged in two separate files (***`ds-set-`***`[`***`input`***`|`***`lookup`***`]`***`.txt`***), where the former (***`input`***) is a shuffled set, intended for use in insertion, while the latter (***`lookup`***) is a sorted set with characters stripped, intended for lookup.

Items in both files are encoded by the page number, the line number (within the page), and the index (within the line) of the character represented, formatted as follows:

PAGE №	LINE №	INDEX	CHARACTER *(**`input`** only)*
135	2	61	w
135	2	62	o
135	2	63	m
135	2	64	e
135	2	65	n

[1] A representative section spanning a few pages of the book was borrowed under fair use for this assignment.

You will implement the following data structures for this assignment:

- **Book Character**: A simple class containing the parameters of the data (**PAGE №**, **LINE №**, **INDEX**, and **CHARACTER**), helping to facilitate an object-oriented approach.

  This class is also responsible for generating unique keys for data using their parameters.

- **Dictionary**: An associative array implementation using a hash table, where the keys are large-sized positive integers (*e.g.* `unsigned long`) and the values are **book characters**.

  For the hash function, make sure you **follow the instructions** given in the **Code** section.

- **List**: Any linked list implementation working as a container for **book characters**.

  You are free to implement this any way you like, building upon STL data structures such as `vector` if you prefer.

Your code will execute the following procedures on your **dictionary**, and then on your **list**:

BLOCK INSERTION

1) Each line in the ***input*** data set will be read and processed into **book character** objects.

2) These objects will then be sequentially inserted into the target data structure, while measuring the total insertion runtime. For the **dictionary**, the average number of cache collisions per item must be measured as well.

3) The insertion runtime (and the average number of collisions) will be output to the console.

BLOCK LOOKUP

1) Each line in the ***lookup*** data set will be read and processed into **book character** objects.

2) The objects will then be sequentially looked up by their keys within the target data structure (assuming **block insertion** was performed before) and the **CHARACTER** data missing from the objects will be updated, while measuring the total lookup runtime.

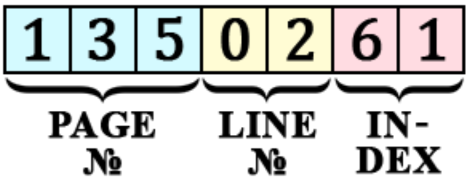
3) The lookup runtime will be output to the console, and the objects that were looked up will be written into output files named ***ds-set-output-*[*dict*|*list*]*.txt*** following the same format as the ***input*** data set. The output files should appear sorted and have their **CHARACTER** data in place.

Code (60 points)

Implement the hashing and probing functions **[10 points]**, and the insertion and lookup methods for your dictionary **[10 points]** and your list **[10 points]**.

Implement the block insertion and block lookup procedures to generate the output data sets **[10 points]**, and output your measurements for the total insertion and lookup runtime for both the dictionary and the list **[10 points]**, as well as the average number of cache collisions for the first 1,000 items, the first 10,000 items, the first 100,000 items, and for all items, while inserting into the dictionary **[10 points]**.

Your **book character** class must have a method to generate unique numeric keys using the **PAGE №**, **LINE №**, and **INDEX** of the character. You are free to generate keys as you prefer as long as they uniquely identify each book character, but we suggest concatenating the numbers in the following manner:



Your **dictionary** class must implement a hash table of size $M = 131{,}071$ $(= 2^{17} - 1)$. This static size is important to reasonably observe collision patterns with respect to the given data.

For your hash table, use the following hash function $h(k)$ for each key k:

$$h = h(k) = \left\lfloor M * \left(k * A \ (mod\ 1)\right) \right\rfloor$$

$$\left(A = \frac{\sqrt{5} - 1}{2} \cong 0.618\right)$$

Implement the following quadratic probing function to circumvent the i^{th} cache collision:

$$h' = h + 7i + 3i^2 \ (mod\ M)$$

e.g. First collision at $h = 39{,}059$

$h' = 39{,}059 + 7 \times \mathbf{1} + 3 \times \mathbf{1}^2 \ (mod\ 131{,}071)$
$h' = 39{,}069$

Then a second collision at $h = 39{,}069$

$h' = 39{,}069 + 7 \times \mathbf{2} + 3 * \mathbf{2}^2 \ (mod\ 131{,}071)$
$h' = 39{,}095$

Finally, no collision at $h = 39{,}095$.

You must also include a `main(...)` function in your code, initializing your **dictionary** and **list** data structures, and running the **block insertion** and **block lookup** procedures on each data structure. Assume that the data sets are under the same directory as your source files, and that their filenames (*`ds-set-*.txt`*) are static. Keep these paths hardcoded in your program.

Your program should compile and run using commands similar to the following example:

```
[user@ssh ~]$ g++ main.cpp p3char.cpp p3dict.cpp p3list.cpp -o proj3
[user@ssh ~]$ ./proj3

   DICTIONARY
   Insertion finished after #.## seconds.

   Average number of collisions (first 1,000)   | #.##
   Average number of collisions (first 10,000)  | #.##
   Average number of collisions (first 100,000) | #.##
   Average number of collisions (overall)       | #.##

   Lookup finished after #.## seconds.

   LIST
   Insertion finished after #.## seconds.
   Lookup finished after #.## seconds.

[user@ssh ~]$ _
```

Your code will be carefully inspected and tested on a different data set from the one provided with the assignment. Please do not resort to funny tricks. Happy coding!

Report (40 points)

Q1) [10 points] Provide two graphs or tables (for the **block insertion** and **block lookup** procedures) comparing the runtimes on your **dictionary** and **list** implementations.

Q2) [10 points] Briefly analyze the computational performances of your data structure implementations in relation to the runtimes you measured. Is there a noticeable difference between the two data structures for either procedure? Are the runtimes you measured in line with the corresponding average-case time complexities?

Q3) [10 points] How does the average number of collisions differ as more items are inserted into your **dictionary**? Does it stay roughly the same, or increase linearly, or exponentially? Briefly discuss the reason for this observed behavior of your hash table implementation.

Q4) [10 points] What is the worst case for looking up a key in your **dictionary**? What is the corresponding worst-case time complexity in O notation? In the worst case, which part of the lookup process becomes slow enough to dominate the complexity?

Submission

You should be aware that the Ninova system clock may not be synchronized with your computer, watch, or cell phone. Do not e-mail the teaching assistant or the instructors your submission after the Ninova submission has closed. If you have submitted to Ninova once and want to make any changes to your report, you should do it before the Ninova submission system closes. **Your changes will not be accepted by e-mail**. Connectivity problems to the Internet or to Ninova in the last few minutes are not valid excuses for being unable to submit. **You should not risk leaving your submission to the last few minutes**. After uploading to Ninova, check to make sure that your submission appears there.

Policy: You may discuss the problem addressed by the project at an abstract level with your classmates, but you should not share or copy code from your classmates or from the Internet. **You should submit your own, individual project.** Plagiarism and any other forms of cheating will have serious consequences, including failing the course.

Submission Instructions: Please submit your assignment through Ninova. Include both your report (as a PDF file) and your code (including header files) in the archive file you submit.

All your code must be written in C++, and must compile and run on the common ITU Linux Server (accessible through SSH) using g++. If your code requires non-standard compiling, please state compilation instructions in your report.

You are responsible for following an object-oriented methodology in your project, making sensible use of classes, attributes, and methods. Bear in mind that purely procedural approaches may be penalized. Try to use well-chosen names for variables and methods, and place comments in your code where necessary**. Your code must compile without any errors; otherwise, you may get a grade of zero on the coding part of the assignment.**

If a question is not clear, please let the teaching assistant know by email (sulubacak@itu.edu.tr).