CSCI 262 Data Structures

Fall 2018

Lab 7: Sets and Maps

(Back to Labs)

Goals

- Practice with Sets and Maps
- Learn how to time running programs
- Do some basic performance comparisons on data structures

Overview

This lab will walk you through some exercises using Sets and Maps. As usual, we ask that you work with a partner or a team. If you end up solo, please raise your hand and we'll pair you with someone else (or make a three-person team if necessary). You are welcome to change partners throughout the semester!

Part 1 Instructions (Sets)

Task 1.0: preliminaries

Download dictionary.txt, which contains a large body of strings we'll use for testing.

Next, download lab07a.cpp. Go ahead and create a new project in CLion as usual, and upload the starter code and dictionary.txt to it. The dictionary.txt should be in your cmake-build-debug folder.

Task 1.1: figure out what is going on

Read through the lab07a.cpp code provided. Much of the code is stuff you have seen before or practiced in a previous lab - opening a file, reading in its contents, or adding items to a vector. Some of it may be less familiar, but the comments should help.

In brief, the code is doing the following:

1. Reading the words from dictionary.txt into a vector
2. Creating a subset of the vector (of size `TEST_SIZE`) to use as a test collection
3. Timing how long it takes to search the **vector** for all the words in the test collection

Pay close attention to the timing code, as you will need this again, and it is generally useful to have for your own performance optimization of your code in the future. If you want to see how long an operation will take coded one way versus another, wrap some timing code around it and find out. The only catch is, computers are so fast that you won't get very helpful numbers unless you perform the operation many times - which is what the `TEST_SIZE` parameter in our program is for.

The `clock()` function is the critical piece of the timing code. It returns a value of type `clock_t`, which is a number counting the "ticks" of some architecture/operating system-dependent clock since the start of the program. You get the clock value at the start of what you want to time, then again at the end, and subtract to get the number of clock ticks. For human eyes, it is generally helpful to convert to standard seconds - you do this by dividing by `CLOCKS_PER_SEC`. `CLOCKS_PER_SEC` and the `clock()` function are both provided in the `<ctime>` library.

Run the starter code:

Calibrate the `TEST_SIZE` constant at the top of lab07a.cpp. You want the code to take long enough to see that something is actually happening, but you don't want to spend all your time waiting. If the timed part of the program is in the 5 - 15 seconds range, that's about what you want. Adjust `TEST_SIZE` downward to reduce the time, adjust it upwards to increase the time (recompile each time).

Task 1.2: time using a vector as if it were a set

In the lecture on Sets and Maps, we described the Set as an ADT having three principal operations: adding, removing, and testing for containment. For the first part of our experiment today, we are going to see how well a vector does at one critical operation: testing for containment. The provided code does just that, by filling a vector with (unique) words and repeatedly looking for the word in the vector.

The way you look for a string in an arbitrary vector of strings is: start at the beginning, comparing words as you go - as soon as you find the word, stop and return true. If you get to the end without finding the word, return false. There are 127,141 words in dictionary.txt, but looking through them one at a time is still very fast - which is why we have to test using so many words! The provided code uses the <algorithm> library function `find()` to search the vector in the fashion described, but you could just as well loop on the size of the vector with an index, or even do a range-based for loop (although `find()` is generally faster).

Run the provided code three times to collect three different timings; they shouldn't vary by much. Record the timings for entry into Canvas!

Task 1.3: time using a set as a set

Rewrite the provided code to now perform the same tests using a set instead of a vector - you should also remove the call to `find()` function and instead use the `count()` or `find()` methods of `set` to test for the existence of a value in the set (the `find()` function we used for vectors works for sets, but doesn't take advantage of the set's ability to do fast lookups). Make sure you leave TEST_SIZE unchanged, so you can compare fairly against your vector tests. Record timings for three trials using the set.

Task 1.4: understanding the performance difference

How was the set so much faster at looking up values? Investigate the *complexities* of searching in sets vs vectors, and write a short reasoning in Canvas. Now, try testing the speeds with an `std::unordered_set` and compare performance.

For further exploration and extra credit, choose one or more of the questions below, and provide a paragraph or so answering the question in Canvas.

1. What difference does it make in the timings if you use a test set made up of randomly generated strings (say, of length 4), rather than words you know are in the dictionary? Why might this make a difference?
2. How long does it take to fill the set with words from dictionary.txt versus filling the vector with the same words?
3. How long does it takes to **remove** all of the test words from the vector versus removing them from the set?

Part 2 Instructions (Maps)

For part 2, you are going to first get comfortable with using a map. Maps are extraordinarily useful data structures for their capability of having key/value pairs to represent useful information. For this part, simply read the source code (lab07b.cpp) and try to understand what is going on. Import this as a separate project into CLion so that you can properly run the program.

Now that you have the restaurant simulator running, play around with it! Type in

```
    Burger
    Fries
    Sandwich
    quit
```

and see what kind of output you get! You should get the output of $9. What's really important in this part is understanding some of the basic functions of a map. Take a close look at how we are "putting" items to the list and how we are "getting" items from the list. Test to see if you can add another item to the menu:

```
    Apple  2
```

Where did the apple item appear in the menu? Take note of ordering in maps as well. You can also compare the ordering using `std::unordered_map` and check out the order of the items in the menu. Also notice, that if you would like to traverse through a map, we want to use

```
map::iterator
```

Then, using the iterator, you can access the key of an element with

```
it->first
```

and the value with

```
it->second
```

Play around with the code to gain some more familiarity with maps, and how to work with them. In class we also talked about STL pairs, feel free to try them out too!

Part 3 Instructions (More Maps)

Task 3.0: Preliminaries

Make sure you have downloaded dictionary.txt.

Next, download lab07c.cpp, the starter code file for this portion. Go ahead and create a new project with this

---

Task 3.1: Char to ASCII

Go ahead and google the ASCII table (pronounced æski). With ASCII, every alphabetic, numeric, and special character is represented with a 7-bit number. Therefore, any alphanumeric character has a decimal representation as well. To do this in C++, we can convert a char to an int with

```
    int('a')
```

You should get that the value for this is 97, and you can see that on the ASCII table as well!

Task 3.2: Encrypting Each String in Dictionary

For this portion, we want to write a simple encryption tool to encrypt each string in the dictionary. Our encryption method will be as follows:

- We are provided a string, and want to find it's encryption key:
- Find the string's integer representation by taking the sum of each character (ASCII conversion) in the string
- Because we do not want to have hundreds of thousands of encryption keys, let's limit the number of keys to 1000. Accomplish this by using the mod operator on your encrypted key
- Use the encryption key to get the vector in the map, and add this string to its vector

Let's test your implementation! With the string `hello`, your encrypted key should be 532, and with `trustworthy` you should get 263. To test your encrypted map, verify that with the encryption key of 1, the first string in the list is `airworthy`. Notice that for each key you're going to get a ton of strings mapped to it. This is because we are limiting our encryption keys to 1000 keys, and there are 127142 words in our dictionary!

Using your encryption tool, answer the question on Canvas.

Task 3.3: (Optional) Design Your Own Encryption

For this optional portion, design your own encryption! Use some clever algebra to see if you can create an encryption map that

1. Maps each key to only a few strings
2. Is a dense map (no keys that map to nothing)

Let us know if you have a clever implementation!

Submission Instructions:

Answer the questions for the lab 7 quiz on Canvas.