CPSC 221 2016W2 Programming Project 3: Dictionary Decryption

Due Date and Time

Due: **Thursday, March 30th, 2017, at 11:59 PM** via electronic `handin` submission. Be sure to hand in the project using the online `handin` program described at the end of this document.

The **late penalty** is 3% per hour (or portion thereof), with no late assignments being accepted after 12 hours. For example, if you hand in your program at 05:40 AM on the morning after the due date, then this is 6 hours late; so, you would lose 6(3%) = 18% of the maximum possible mark.

You are allowed to work in pairs. If you work in a pair, then each partner is expected to contribute meaningfully to the project. Both partners will get the same grade. Please review **Academic Conduct** rules and keep in mind that violation of any of these rules constitutes academic misconduct and is subject to substantial penalties. If you are uncertain as to what is, or is not, reasonable collaboration, please contact your TAs or instructor.

Part marks will be awarded, so even if you don't finish everything, make sure that your code compiles on our undergrad machines (these machines tend to have the same software releases, so as long as it works on one, you're OK; our TAs will mark your code based on the undergrad environment). Use your judgment to determine whether it's worth turning the project in late, when considering the late penalties.

1 Introduction

Your mission, should you choose to accept it (you should choose to accept it), is to identify 7 different dictionary data structures. Unfortunately, due to budget cuts, we cannot actually provide the code to you. So instead you'll need to measure how the structures perform when used, and use your knowledge of these structures to try and identify them.

Your goal is to execute sequences of operations using these data structures, which are provided as a program called the *dictionary runner*, in order to determine which mystery structures, called `mys01` to `mys07`, correspond to which dictionary structures. The only information you obtain from the dictionary runner is timing information: how long the sequence of operations you provide takes to execute.

All of the dictionaries will store 64-bit unsigned integers. You will be able to insert, remove, and find entries in each dictionary by issuing commands to the dictionary runner via a file or standard input, described later.

The data structures that you'll need to identify are... (drumroll please!)

1.1 The Dictionaries

- `mtf`: Unsorted linked list with move-to-front on find.
- `sl`: Sorted vector (i.e., resizing array)
- `hch` : Hashtable with chaining (fixed size 10000)
- `hqp`: Hashtable with double hashing (resizing)

- `bstT`: Binary search tree with tombstones
- `avl`: AVL tree
- `spt`: Splay tree

They will each uniquely correspond to one of {`mys01`, `mys02`, `mys03`, . . . , `mys07`}, and this is the mapping that you need to determine.

Perhaps some of these data structures were not covered in class? In that case, you'll have to look them up yourself: try consulting your favourite textbook, Wikipedia, or a search engine.

2 Dictionary Runner

To help you determine your mapping of data structures to mystery dictionaries, you are given an executable file that runs `mys01`, `mys02`, . . . You will need to use the handback website to download your dictionary runner:

`https://www.ugrad.cs.ubc.ca/~cs221/handback/`

under the section `Dict_runner`. As this file is an executable program that only runs on the department/lab machines,[1] you'll need to copy it to a subdirectory in your home directory and then work from (or perhaps SSH into) one of the department/lab machines. This program will perform operations on the dictionaries using the following commands, which must be placed one per line in the input:

```
I <number>
F <number>
R <number>
```

The `I`, `F`, and `R` commands insert, find, and remove/delete the specified number, respectively. The number must be a **positive** integer. **Inserting, finding, or removing 0 has unspecified behaviour! Inserting a key already present in the dictionary also has unspecified behaviour!** Removing a key not present in the dictionary will have no effect.

The mapping implemented by your dictionary runner is probably different from that implemented by your friend's dictionary runner (there are 7! different mappings). Therefore, only solve and submit a solution for *your dictionary runner*, not someone else's! If working in pairs, only submit *one solution* for the dictionary runner of the **submitting** account.

2.1 Advice

First and foremost: `handin` early, and `handin` often! You will only be graded on your final submission, so "save"your partial work by handing it in early!

Now as an example, once you've copied your executable to somewhere in your home directory and assuming you've compiled a program called `generator0.cc` such as the following to generate some dictionary operation commands for you:

[1] Technically, any machine with a sufficiently similar architecture will work, but no guarantees!

```
#include <iostream>
#include <cstdlib> // for atoi
using namespace std;

int main(int argc, char* argv[])
{
    if (argc != 2) // remember, argv[0] is the program name
    {
        cerr << "Wrong number of arguments!" << endl;
        return 1;
    }

    int n = atoi(argv[1]);
    for(int i = 1; i <= n/2; i++) {
        cout << "I " << put something here... << endl;
        cout << "R " << put something here... << endl;
    }
    return 0;
}
```

then, you may try[2]:

```
./generator0 1200 | ./dict_runner-r2d2 - 10 . -mys01 -mys02 -mys03
```

If the parameter to the generator is a number n, it should generate approximately[3] n dictionary operations. Of course, you don't have to follow the code above exactly, you can make your generator output any sequence of dictionary operations you like (as long as it accepts one parameter and generates the right number of operations based on this parameter). The command above will run the generator and then pass ("pipe") the generated sequence of operations to the dictionary runner that in turn runs the 1st to 3rd mystery dictionary implementations and generates files called mys01, mys02, mys03 in the current directory. These generated files contain the timing information for the given data structure. If, for example, you ended up with a file like this:

```
0 0
500 8000
1000 18000
1200 22800
```

Then the first 500 operations took 8000 time steps, the first 1000 took 18000 time steps, and the first 1200 took 22800 time steps.

How can you figure out what data structure a mystery dictionary uses? You need to generate sequences of dictionary commands to give to your dictionary runner that distinguish between different data structures based on how long the commands take to execute.

Think about the examples of inputs from your studies that cause data structures to perform differently. In particular, think about best-case and worst-case inputs and performance differences on different operations for the various dictionaries.

[2]If you run your dictionary runner without parameters, it will show you more information about the purpose of various parameters.

[3]plus minus some small constant

Once you have data, what can you do with it? Well, if you'd like a handy way of visualizing the data, you can use gnuplot to produce a graph, using the following commands:

```
gnuplot
plot "mysXX" using 1:2
```

3 Submission (due Thursday, March 30)

3.1 Deliverables

Note for partners: **only handin from the account whose dictionary runner was used!**

- `mapping.txt` — a simple comma-separated text file containing your mapping of all data structures to mystery data structures, precisely in the format of this example: (not necessarily the same mapping, of course!)

  ```
  mtf,mys07
  sl,mys02
  hch,mys03
  hqp,mys01
  bstT,mys05
  avl,mys06
  spt,mys04
  ```

- Source code of at least 2 generators. They should be named `generator0.cc` and `generator1.cc`, and need to accept as input an integer **n**, such that the command:

  ```
      ./generator0 n
  ```

  outputs approximately `n` dictionary operations.

- A `README` file containing:
  1. Your name or, if a team submission, both your names.
  2. Approximately how long the project took you to complete.
  3. Acknowledgment of any assistance you received from anyone but your team members, the 221 staff, or the 221 textbooks, but please cite code quoted or adapted directly from the texts (per the course's Collaboration policy).
  4. A list of the files in your submission with a brief description of each file.
  5. Any special instructions for the marker.

- DO NOT HAND IN: .o files, executables, core dumps, irrelevant stuff.

How to `handin` this assignment

1. Create a directory called `~/cs221/proj3` (i.e., create directory `cs221` in your home directory, and then create a subdirectory within `cs221` called `proj3`).

2. Move or copy all of the files that you wish to hand in, to the `proj3` directory that you created in Step 1.

3. Before the deadline, hand in your directory electronically, as follows: `handin cs221 proj3`

   Note that you will receive a set of confirmation messages. If you don't get any kind of an acknowledgment, then something went wrong. Please re-read the instructions and try again.

4. You can overwrite an earlier submission by including the `-o` flag, and re-submitting, as follows: `handin -o cs221 proj3`

   You can hand in your files electronically as many times as you want, up to the deadline.

5. Additional instructions about `handin`, if you need them, are listed in the man pages (type: `man handin`). At any time, you can see what files you have already handed in (and their sizes) by typing the command: `handin -c cs221 proj3`

   If your files have zero bytes, then something went wrong and you should run the original `handin` command again.

4 (Friendly) Competition

In addition, you (more precisely your generators) will participate in a friendly competition of sorts.

You will be scored based on how well your generators distinguish between the various data structures. For example, if your generator demonstrates a $\Theta(1)$ run time on `mys01`, and a $\Theta(n)$ run time on `mys02`, your score would reflect that you've successfully differentiated between the two structures. You can submit **more than two** generators but at most 10 generators. The generators must be named `generatorN.cc`, where `N` is a number between 0 and 9. If your `N` is not a single digit or you change the extension from `cc` to `cpp`, for instance, it will not be accepted. Also note that if your generator outputs more than n dictionary operations (where n is the number specified as the parameter), only the first n operations will be considered.

Finally, note that the mapping between the `mysXX` and the dictionaries is not necessarily the same as yours, so don't try and use the leaderboard to determine your mapping.

You can check out the leaderboard at the following link:

`http://www.ugrad.cs.ubc.ca/~cs221/current/fun/`.

Bonus marks. In addition to the mandatory component, you may also be awarded bonus marks. If your generators can differentiate between 80% of the pairs of dictionaries, you'll earn 1 extra mark. You'll earn 2 for differentiating between 90%, and 3 for differentiating between 95%. Also, the top 15 players on the leaderboard will be awarded additional marks, with the first to the fifth place players earning 3 extra marks, the sixth to the tenth place players earning 2 extra marks and the eleventh to the fifteenth place players earning 1 extra mark.