Dictionary Wars

CSC 190

March 14, 2013

Contents

1 Learning Objectives **2**

2 Swiper no swiping! . . . and pre-Introduction **2**

3 Introduction **2**
3.1 The Dictionaries 2

4 Dictionary Runner **3**
4.1 Advice 4

5 Justification **4**

6 Milestone 1: Lab 7 (due March 18/19) **5**

7 Milestone 2: Lab 8 (due March 25/26) **5**

8 Final Submission (due March 28) **5**
8.1 Deliverables 6

9 Bonus task/competition (recommended, but optional!) **6**

A Appendix: Marking Scheme **7**

Online PDF of this document

See `dictwars.pdf`.

Changelog

Any changes to this document since it was released will be described here.

1 Learning Objectives

- Learn, on your own, about a variety of data structures used to implement the dictionary (or set) ADT (abstract data structure)
- Learn about tradeoffs between different implementations of ADTs (i.e. in a given situation, which works "best", whatever that may mean)
- Further familiarization with running times and other values measured using asymptotic notation

2 Swiper no swiping! . . . and pre-Introduction

Silly Context: Oh no! Swiper has stolen the dictionary implementations' names! Can you help Dora find the names? Say "Swiper, no swiping!" and then help Dora figure out which is which.

It doesn't matter for the project, but if you don't know who Dora and Swiper are, ask Naomi Wolfman, or see: `http://www.nickjr.com/dora-the-explorer/`.

P.S. If you are familiar with Dora, get frustrated with the project, and need inspiration, try: `https://i.chzbgr.com/maxW500/4180888832/h84697171/`.

The background story and other useful help information can be found by running the dictionary runner (see below) with no arguments.

3 Introduction

As per the story, your goal is to, given the mystery implementations of the dictionary ADT, perform sequences of timed operations on them using a provided program known as the *dictionary runner* (dict_runner), and use the outputted timing data from the program to identify which of the `mysXX` corresponds to which dictionary implementation. We guarantee there'll be a unique bijection between the mystery dictionaries you're given and the list below!

Note: you could refer to these data structures as either sets OR dictionaries because we're ignoring the values stored inside the dictionaries and only using the keys, in effect treating them as sets. For the purposes of this project, we'll simply refer to them as "dictionaries".

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.

And for the dictionaries you'll be needing to identify. . . (drumroll please!)

3.1 The Dictionaries

- `bstT`: Binary search tree with tombstones
- `hlp`: Hashtable with linear probing (resizing)
- `hqp`: Hashtable with quadratic probing (resizing)
- `hch`: Hashtable with chaining (resizing)
- `usl`: Unsorted doubly linked list

- `arr`: Sorted vector (this is an array that resizes automatically whenever full, expanding to two times the previous size. Java calls these "arrayLists"; C++ calls them vectors.)
- `heap`: Min-heap
- `avl`: AVL Tree

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

A note about the hash tables: the hash tables will automatically ressize as they exceed 50% load, using this sequence of sizes: 101, 211, 421, 673, 1361, 2729, 5471, 10949, 21893, 43787, 87583, 175211, 350293, 700591, 1401187, 2802377, 5604763, 11209591, 22419239, 44838491, 89677037, 179354081, 358708241, 717416501, 1434833009. *The program may crash if you try to exceed that last size.*

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 data structures textbook, Wikipedia, or a search engine.

Just be sure to mention your sources in your assignment, following the Academic Conduct policy!

4 Dictionary Runner

Every group will get a custom executable. You can download it like so:

```
wget www.cs.toronto.edu/~patitsas/cs190/a4/0999 (replace last number with your group num)
chmod u+x 0999
./0999
```

Running the executable will print out helpful information for how to use it. Note you'll need the chmod command in order to run your executable the first time.

As this was compiled on the ECF machines,[1] you'll likely need to work from (or perhaps SSH into) one of the ECF 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 will typically overwrite the old entry, but different dictionaries are free to make different choices, including undefined behaviour. Removing a key not present in the dictionary will have no effect.

The mapping of the dictionary runner you're provided is unique (or nearly so) to your program, so each person will have a different mapping. Therefore, only solve and submit a solution for *your program*, not someone else's!

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

4.1 Advice

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.

Note that you'll need to write programs to generate enough input commands to get good data, rather than typing the input in manually. The executable you have can take in data from files, rather than from you typing it in manually.

This means you can write programs to generate the files you would like: you can write a program that will, for instance, insert the numbers 1 through 100,000 in sorted order. Or reverse sorted order.[2]

You can, if you're feeling advanced, learn how to write programs that will not only generate test files for you, but also to run the executable for you on all of your test files. To do this in C, see page 689 of your textbook (or search "C system function").

The output of the executable – the performance of the data structure given your input – will appear in a folder that you specify (by default, `dataDir`.) Your advanced program can even open up your output files and process them for you too (e.g. loading them into Matlab).

5 Justification

The main deliverable for this assignment will come in the form of a PDF document giving a mapping of mystery dictionaries to implementations and justifying how you identified that mapping. Your justification should present a (concise) argument that lays out the approach to testing (which styles of input and why), the data itself (e.g., as a graph or easy-to-read table), and analysis of the data clearly distinguishes the chosen implementation from all other posssibilities.

Examples of *unacceptable* or *insufficient* justification include:

- A runtime plot that fits a function that is $\Theta(\lg n)$ alone: This may be useful, but won't be sufficient evidence unless you can explain for each other possible data structure why it will not have that runtime for the same sequence of operations (or address the data structures that could have this behavior with separate arguments).
- Elimination (i.e. identifying all dictionaries but one, then concluding it must be the remaining one): This is a great way to lead yourself toward the right answer, but is not acceptable justification alone.
- A long table of runtime data: These are difficult to visually interpret unless plotted, and "these numbers *obviously* look like they fit a curve of the shape $c_1 x + c_2 \lg x + c_3$"-type reasoning is unacceptable.

Reasoning deemed irrelevant or needlessly verbose will be subject to mark deduction as per the marking scheme (in Section A below). So, please proofread.

[2] While I always advocate getting practice in C, I will in no way begrudge it if you wrote this in Python (or Ruby, or Perl, or even LOLCODE). Short scripts for producing text is something Python does a lot nicer than C! An important step in your programming education is realizing which tool is best for the job, and being able to integrate several tools together.

One tool that may be useful is Matlab (or Gnuplot or other plotting/curve-fitting tools), which is described at `../hints/pp3-hint-plotting.html`. If you'd prefer to use Gnuplot, you may find this tutorial (or another) useful: `http://www.duke.edu/~hpgavin/gnuplot.html`.

If you'd like to use a LaTeX template for your writeup, here is one that may be useful: `http://www.cs.toronto.edu/~patitsas/cs190/code/dictwars-template.tex`.

6 Milestone 1: Lab 7 (due March 18/19)

For this milestone, you have two goals. One: to find which of the `mysXX` dictionaries is the binary search tree. Two: to find one of the sorted vector (resizing array), unsorted linked-list, or the min-heap (you don't have to find all three for the lab, but you will by the final deadline).

We expect everybody to have one of the two goals done before arriving in lab. If everybody arrives with both goals done, then marking will go much faster for the TAs!

In lab, the TAs will be marking like so:

- 4 Points for identifying BST
- 4 Points for identifying 1 of unsorted doubly linked list/sorted resizing array/binary min-heap
- 2 Points for effort

Note the TAs will be asking you *how* you identified the two data structures you found. Also note that since this is not a coding lab, this lab is out of 10.

7 Milestone 2: Lab 8 (due March 25/26)

For this milestone, your goal is to identify which of the three `mysXX` dictionaries are hash tables. Furthermore, you'll need to identify which uses linear probing, which uses quadratic probing, and which uses chaining, and justify your answer.

In lab, the TAs will be marking like so:

- 2 Points for identifying which three are the hash tables
- 2 Points for identifying the linear probing hash table
- 2 Points for identifying the quadratic probing hash table
- 2 Points for identifying the chaining probing hash table
- 2 Points for effort

Note the TAs will be asking you *how* you identified the data structures you found. Also note that since this is not a coding lab, this lab is out of 10.

8 Final Submission (due March 28)

Now you must provide a complete mapping of mystery dictionaries to their implementations.

8.1 Deliverables

Again, please include the following:

- README.txt — plain text file with names and group number
- justification.pdf — justification of your answers. Be sure to include key title information such as your names and ugrad IDs.
- mapping.txt — a simple comma-separated text file containing a line for each mapping, precisely in the format this example: (not the same mapping, of course)

  ```
  bstT,mys02
  hqp,mys07
  avl,mys05
  heap,mys06
  hlp,mys08
  hch,mys04
  usl,mys03
  arr,mys01
  ```

  This is just so we can easily see what your choices were at a glance and automatically assign the mapping correctness portion of the marks.

9 Bonus task/competition (recommended, but optional!)

Want to go further and maybe earn some extra points? Want to compete in real time with your classmates? Want to see pretty colors (and who doesn't)? Check out the bonus part on the course website!

A Appendix: Marking Scheme

Whenever we judge your mapping of a mystery dictionary to its implementation, we will use roughly the following marking scheme:

- **EXCELLENT**: The mapping is correct. The (concise) argument lays out the approach to testing (which styles of input and why) and the data itself (e.g., as a graph or easy-to-read table), and analysis of the data clearly distinguishes the chosen implementation from all other posssibilities.
- **GOOD**: The mapping is correct, and the argument is sensible but not complete. For example, may be missing one or more of:
  - what styles of input were used for testing,
  - why those styles of input are useful for distinguishing this particular implementation,
  - a summary of the data used in the argument (e.g. a graph or an easy-to-read table, *not* hundreds or thousands of numbers),
  - what about the data (i.e. graph/table) supports the mapping to a particular dictionary implementation when compared to others.

  We may also use this category for unnecessarily verbose answers that are otherwise excellent.
- **GOOD [WRONG MAPPING]**: The mapping is incorrect, but the argument for the mapping is solid (see above), presumably missing some factor that if considered would have yielded the correct mapping. (Worth roughly the same as **POOR** below.)
- **POOR**: The mapping is correct but the explanation makes little or no case for how the particular implementation was reliably distinguished from all other implementations.
- **POOR [WRONG MAPPING]**: The mapping is incorrect and the explanation makes little or no case for how the particular implementation was reliably distinguished from all other implementations. (Worth roughly the same as **NONE** below.)
- **NONE**: No submission or submission is unreadable or irrelevant.