CS 345 (Spring 17): Analysis of Discrete Structures

Project #4

Dictionary Bake-off

due at 6pm, Thu 13 Apr 2017

1 Introduction

In this project, you'll be comparing various implementations of a Dictionary. We have provided a Java `interface` (named `Proj04Dictionary`), and you must implement this interface in several classes. We have also provided a "timer" program, which will run various testcases, and report how fast the various dictionaries perform various operations. It will report this information in CSV format, which you can then import into a spreadsheet.

In addition, we will provide a set of testcases to check your code for correctness. This time around, we won't use text files; instead, we will provide various Java classes, each of which has a `main()` method; each testcase will check the various classes for certain key features that you must implement.

You must:

- Implement the various dictionaries. I have provided a stub class for each one, and two examples that are already filled in.
- Use the grading script and testcases to confirm that your dictionaries work properly. (Run this on a CS department machine.)
- Use use the Timer program to find out how fast the operations are; save the CSVs to a file. (Run this on your own machine!)
- Import the CSVs into your favorite spreadsheet program, make some simple charts, and a short writeup about what you found.

1.1 Don't Run the Performance Checks on Lectura!

While our grading script is only guaranteed to work on CS department machines (and thus you should run it on `lectura, york, oxford,` or the like), **please do not use Lectura to run the timing code.** The timer program chews up a fair bit of CPU and memory. So please run it on your own personal computer instead!

1.2 What is a Dictionary, Precisely? What is the Interface?

A Dictionary is an ADT (Abstract Data Type) that maps keys to data (sometimes called "satellite data" or "values.") In a complete implementation, we would support any type of key, and any sort of data (this is how Java's `TreeMap` works, for instance). However, in our project, we have simplified the problem; our keys will always be `int`, and our data will always be `String`. (We do not ever allow `null` as a data value, but an empty `String` is OK.)

The formal spec for the interface is contained within the Java source code; read `Proj04Dictionary.java` and **pay careful attention to the comments.**

1.3 What Types of Dictionaries?

I have provided two complete dictionaries for you. These are simple wrappers around Java classes. I encourage you to look at the implementations if you have any questions about how to write a class which implements an interface. (They are also useful for seeing how efficient your code is.) The classes are:

- `HashMap_Wrapper`
- `TreeMap_Wrapper`

You must implement the following dictionaries. For each of these, I have provided a "stub" file. This file will compile and run with the Timer code (or testcases), but will throw an exception for every operation. This allows you to test your code as you write it - you should continually be running the grading script (and later, the Timer code) to see if you are making progress.

- `UnsortedArray` - implements a simple array, where the elements are not sorted.
- `SortedArray` - implements a simple array, but the elements are always kept in sorted order. `search()` should run in $O(\lg n)$ time.
- `BST` - implements an unbalanced BST. We encourage you to copy your code from Project 2; you will need to adapt it a little (for instance, you have to store both a key and a data, not just a key), but the changes should be small.
- `AVLTree` - an AVL tree. Again, we encourage you to modify your solution from Project 3.
- `HashTable` - this will be a new class, that you'll have to write from scratch. Use chaining to resolve collisions; the runtime for insert, delete, and search should all be $O(1)$ (plus the time it takes to traverse the linked list).

1.4 Other Files I Provide

- `Proj04Dictionary` - the interface that you must implement
- `Proj04Timer` - the Timer program's `main()` class
- Grading script and various testcases

1.5 What to Turn In

You must turn in the Java source code for each of the dictionaries; you may also turn in additional Java files if you used additional classes. However, you should not turn in the Timer, interface, or example files; if you do, we will ignore what you turn in.

In addition, you must turn in a .CSV (generated by the Timer), for each of the operations that the timer supports:

- insert
- delete
- search-hit
- search-miss
- getKeys
- getSuccessor

Convert each .CSV to a line chart, using your favorite spreadsheet program; turn in a PDF of your charts.

Finally, in either a README file or the PDF with your charts, write a short commentary (only a handful of sentences per chart), discussing the results you saw. Were your implementations following the pattern you expected?

2 Testing for Correctness

Use the testcases, and the grading script, to check your dictionaries for correctness. While in previous projects you have written a `main()` method and then read from an input file, in this project the testcases will provide the `main()` method. Each will generate its own data internally (randomly, typically) and then drive various functions in your code. Like the Project 3 Test Driver, these programs will not print out much - unless they detect an error.

3 How to Run the Timer Program

The Timer program is run as follows:

```
java Proj04Timer 10000 100000 10000 insert
```

The first three arguments must be integers; they are the range of dataset sizes to check. The first is the start; the second is the end; the third is the step size.

The fourth argument is a string, which tells the Timer what sort of test to perform. There are six possible tests (you must run all 6): `insert, delete, search-hit, search-miss, getKeys, getSuccessor`.

When you run, you will (at first) see lots of exceptions printed out, like this:

```
CRASH IN USER CODE:
java.lang.RuntimeException: TODO
        at AVLTree.insert(AVLTree.java:6)
        at Proj04Timer.doTest(Proj04Timer.java:273)
        at Proj04Timer.runOneTest(Proj04Timer.java:246)
        at Proj04Timer.runTestSet(Proj04Timer.java:189)
        at Proj04Timer.runTest(Proj04Timer.java:149)
        at Proj04Timer.main(Proj04Timer.java:78)
```

This crash happened in the AVLTree dictionary (because it was not yet implemented). As you implement various dictionaries these will go away - but keep an eye out for exceptions caused by bugs - such as `NullPointerException`!

If you want to skip over the exception output, you can (if you're running on UNIX or a Mac) redirect `stderr` to `/dev/null`, like this:

```
  java Proj04Timer 10000 100000 10000 insert 2>/dev/null
```

Once you do this, you will see `NaN` (floating point for "not a number") to represent the time of any dictionary which hit exceptions:

```
Size, Unsorted Array, Sorted Array, BST, AVL, Hash Table, HashMap, TreeMap
10000, 47.000, 80.375, NaN, NaN, 1.375, 1.125, 5.625
20000, 205.125, 323.375, NaN, NaN, 2.500, 2.375, 7.125
30000, 458.750, 694.500, NaN, NaN, 3.000, 4.500, 12.000
```

(So long as you see `NaN` in the output, you have a bug which is crashing your dictionary, and which needs to be fixed.)

You can either direct this to a file using the UNIX shell syntax

```
  java [cmd] >insert.csv 2>/dev/null
```

or you can simply cut-n-paste the output into a text file.

3.1 What Data Sizes to Use?

As a general rule of thumb, try running the Timer with the following range:

```
  10000 100000 10000
```

. This runs at 10 thousand elements, 20 thousand elements, etc., up to 100 thousand. Most of your CSVs should use these values.

You are allowed to change this to some other range if you have a compelling reason. However, note that if you make the datasets too small (because you want the Timer to run quicly), your information will not be useful. If you see times (normally, for the `HashMap_Wrapper` dictionary) that are below 1.000, then you are running too small of a range. (All times reported are multiples of .125, because we total up the (integer) # of milliseconds from 8 tests - and then divide by 8.)

3.2 Opening a CSV

Any spreadsheet program should be able to open a CSV; it's a very common and standard format. Just do `File...Open` (or the equivalent) in your favorite spreadsheet progam and open the file.

4 Comments and Style

You must comment your code. Make sure that your comments are clear; they need to express both what you are trying to do, and how you plan to do it.

Use good programming style, including good use of whitespace, consistent indentation, and meaningful variable names. If you write your program in Java, follow the Java variable naming conventions (start with lowercase, and camel case after that). If you write in C, there is more flexibility about style - but use a style that is clear, easy to read, and consistent.

Each file should include a header comment, which includes:

- File name or Java class name
- Our class and assignment name (CSc 345 Spring 17 - Project 4)
- Your name
- A description of the basics of the file or class

5 A Note About Grading

Our grading script will run the testcases we provide (and maybe some secret ones that we don't tell you about beforehand). We will use this to detect bugs in your implementation. As with Projects 2 and 3, if your code fails any testcase, you will lose a portion of the "auto-grading" part of your project score.

While we don't expect to create world-class implementations of the dictionary types, you should make a reasonable effort to implement each dictionary as efficiently as you can.

It is also important that you implement each dictionary in the way required; for instance, an AVL tree should not be an array!

The TAs will be looking at your CSVs that you turn in - along with the charts and your short descriptions of them - and will also check your implemementations to make sure that they work as required.

5.1 How Points are Assigned

In this project, 40% of your score will come from simple testcase execution. Since you are not writing the `main()` method, it will be easy to match the output expected - provided, of course, that you don't add any debugging print statements inside your code. (In this Project, we don't need an example executable. Some of the the logic will be hard-coded into the testcases, and some of it will simply require that you have the same results as the `HashMap_Wrapper`.)

10% of your score will come from the CSVs, charts, and descriptions that you turn in. Each chart should have a few sentences, comparing the various dictionaries. Which ones appear to be running in constant, logarithmic, linear, or quadratic time? Does this make sense? (While I don't expect your Hash Table to be as good as `HashMap_Wrapper`, are you kind of close - or very far away?) You are **not** expected to write many pages - just a paragraph or so per chart.

20% of your code will come from the TAs checking that you actually implemented each dictionary as required. So the `AVLTree` should include rotations, the `SortedArray` must actually be sorted, etc.

30% of your code will come from manual inspection of other parts of your code, including style and general design.

5.2 Testcases

You can find a set of testcases for this project on D2L, with the rest of the files. Each testcase is named `Test_*.java`.

For some projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcases of your own, in order to better test your code.**

5.3 Automatic Testing

We have provided a testing script named `grade_proj04`. Place this script, all of the testcase files, and your solution file(s) in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)

Run the grading script - it will tell you which testcases passed and which failed, and will give you a numeric score. (When we grade your code, we may have additional "secret testcases" that you didn't see. So your score may be somewhat different than what the grading script gave you. But it's a reasonable estimate.)

5.4 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception.** We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

6 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only the required files (see above); do not turn in any testcases that you've written, or other files.