CS310: Advanced Data Structures and Algorithmns

Fall 2019 Programming Assignment 1

Due: Tuesday, Oct. 15, 2019 on Gradescope

Goals

This assignment aims to help you:

- Learn about hash tables
- Review packages
- Start thinking about performance and memory use

Preliminaries

Reading

- S&W Chapter 3.4 (hashing)
- S&W code instructions http://algs4.cs.princeton.edu/code/ .
- Java documentation about packages: https://docs.oracle.com/javase/tutorial/java/package/packages.html.

Setup for Linux

The PA will be automatically graded on Gradescope and the environment will be a linux environment. Therefore, it is important that you run and test your code on a Linux machine. The best way to do it is to use your Unix account. Please refer to Gradescope's user's guide if you have any doubts or questions.

- Download the S&W code in a jar file. See here: http://algs4.cs.princeton.edu/code/ . At the bottom of the page there are installation and running instructions. Create a lib directory under pa1, and copy the algs4.jar library there.
- To compile the code you first change to the src directory:

  ```
  cd src (the top-level source directory)
  javac -cp .:../lib/algs4.jar -d ../classes *.java
  ```

  The -d flag redirects the class files to the `../classes` subdirectory. The -cp flag tells the compiler to load the jar file during compilation. If you omit this part, it will not compile since your source will not find the S& W classes it needs. Notice that when testing on Windows you should put a ; (semi-colon) instead of a : (colon).

Questions

1. Performance of a hash table: Compare the performance of the S&W lookup tables: sorted symbol table (ST), separate chaining (SeparateChainingHashST), linear probing (LinearProbingHashST), and a sequential search (SequentialSearchST), which is just an array list. Use the Linear Probing implementation provided by S&T, not the one you defined in the previous question. Don't use Java's standard hash tables. You will create a word counter, reading in a long document and map each word to its number of occurrences in the text. Specifically, do the following:

   (a) Create a class TestPerf.

   (b) In your class, define four lookup tables as described above. Each one will have `<String, Integer>` as key and value types.

   (c) You can use the In class from the S&W to parse an entire text file into an array of Strings using the function readAllStrings. For simplicity, assume that every two words are separated by a whitespace, and don't worry about capital and small letters, punctuations, quotations etc. It means that for example, "word" and "word;" will be considered two different words, but that's ok for the sake of this assignment.

   (d) Use the file http://introcs.cs.princeton.edu/java/data/tale.txt. It is also attached as a handout. It is probably big enough to show the differences in performance on most PCs and laptops. I will test on other files as well.

   (e) Word counter: Every time you read a word, you should search it in the table. If it is not there, add it with a value of 1 (because it's the first time you see it). If it is in the table, just increment its value by 1.

   (f) To compare the performance, you should time the above word counter. The simplest way to measure the time is to use `System.currentTimeMillis()` before and after you read the file into each table, and subtract the end time from the start time. Have your function return the result. It will be in milliseconds, and it is a long, not an int.

   (g) Have the main function take the file name as a command line parameter (NOT read from the standard input), so the usage is :

   ```
   java -cp .:../lib/algs4.jar TestPerf tale.txt
   ```

   I don't believe I still have to say it, but `tale.txt` should reside in your running directory for this to work. If it isn't, specify its relative or absolute path. Notice that during runtime you should also specify the classpath as you did during compilation to avoid a runtime error.

   (h) Add the function `public void printStats()` to your class. It should be called from the main function after the tests and return the following: The total number of words, the unique number of words (simply the size of the symbol table) and the most common word and its number of occurrences (See below).

   (i) Notice that S&W don't believe in inheritance, which may force you to duplicate some parts of your code. It is annoying, I know.

   (j) SequentialSearchST is really slow (by now you should not be surprised). Just be patient.

   The output for the example above should be similar to the following:

   ```
   110
   82
   42
   39415
   139043
   19695
   the 7515
   ```

   The first four numbers are the runtimes for `ST`, `SeparateChainingHashST` `LinearProbingHashST` and `SequentialSearchST` respectively. Your numbers should not be exactly the same as above but they should be the same order of magnitude. The rest of the lines indicates that the file `tale.txt` has

139,043 words, 19,695 unique words and the word the is the most common one, appearing 7,515 times. These numbers are expected to be exactly the same for this example. Follow the above format exactly, or the autograder will fail.

2. In class we discussed deleting from a hash table that uses linear probing. The text approaches the problem by deleting the item and moving all the subsequent cluster of hash elements one position up. This is necessary in order to not disconnect a collision chain, but it may harm the performance, especially if there are large clusters.

   An alternative way to delete from a hash table is as follows: Every item in the table has an extra boolean flag, active or not active. When an item is "deleted" it is not physically removed from the table, but rather, its active flag is set to false. For this to work, several other things have to be modified from the original implementation:

   - When searching for an item in the table, you should return it only if its active flag is set to true.
   - When calculating the table size for rehash, you have to take into account "deleted" items as well, since they are in the table and take up space.
   - When rehashing, you obviously don't need the deleted objects anymore, and therefore you insert only active objects into the new table.

   Implement this alternative deletion. Do the following:

   (a) Start from S&W's implementation of hash with linear probing at:

   http://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/LinearProbingHashST.java.

   (b) Copy it over to your src subdirectory and change the class name (and file name, of course), to LinearProbingHashST2.java.

   (c) delete the package declaration from the top of the file.

   (d) Notice that since the source is not part of the library now, you may have to import some classes from the algs4 library.

   (e) Make the changes noted above. Notice that it's not only the delete function that you should change.

   (f) Test the delete function: In the main function, have the LinearProbingHashST2 read a file that contains strings. For example, you can use `tinyST.txt` which you can find at `http://algs4.cs.princeton.edu/34hash/tinyST.txt`. You can use the LinearProbingHashST's main function as a starting point: It reads a file string by string and puts the `<string, index>` pair in the table (in this case every string is just one character). However, don't read from the standard input but make the file a command line parameter (see below). It makes the auto-grading easier.

   (g) Print out all the characters, then delete several characters and print again. The input file and characters for deletion should be the command line arguments (see below). You can delete any number of characters.

   (h) Have the main function print out the size() of the table and the values of the n and m fields (as appear in the original source file) before and after deletion. Here is an example:

   Usage example: `java -cp .:../lib/algs4.jar LinearProbingHashST2 tinyST.txt S A`

   Notice that the input file is the first command line parameter, followed by the character(s) to delete. Please don't hardcode anything. The output format for the autograder should look like that (this is the example above. I may use other files for testing):

```
A   8
C   4
E   12
H   5
L   11
M   9
P   10
R   3
S   0
X   7
10  10  32
C   4
E   12
H   5
L   11
M   9
P   10
R   3
X   7
8   10  32
```

The questions

In your `memo.txt` file (submitted to Gradscope) you should answer the following questions:

1. How many ms did every table take when testing on `tale.txt`?
2. Does it (approximately) correspond to the expected insert and search performance for each table? No need to perform a detailed analysis, but please mention what you know about the expected runtime for each one of the lookup table types.

Delivery

Before the due date it's highly advisable to test your code on the on your UNIX account or any Linux (or Mac) machine you have. The testing will be done in a Ubuntu environment. Remember again that UNIX is case sensitive, so the class and file names have to be case sensitive. This is important since the auto-grader will fail otherwise. Make sure the sources compile and run on UNIX! They should, since Java is very portable. It's mainly a test that the file transfer worked OK and that you know how to compile and run from a command line.

- Answer the questions in the online assignment.
- memo.txt (plain txt file, try "`more memo.txt`" on UNIX). This will not be auto-graded, but it should be uploaded with the rest of the code at the same time.
- `TestPerf.java`
- `LinearProbingHashST2.java`

You should upload both java files and memo.txt to the autograder at the same time and it will run on the spot.

Style checking

Your code will be subject to style checking. The style checker is part of the handout to the course. It's quite strict but please follow it. If you do a test upload to gradescope ahead of time it will tell you if and when you fail, and you can fix it and resubmit.