Assignment 4, TCSS 342B Winter 2018

OBJECTIVE

The objective of this assignment is to give you practice with the trees and hash tables and their performance, code reading and writing, generics, and conducting experiments.

ASSIGNMENT SUBMISSION

To get credit for this assignment, you must

- ✓ submit your assignment on time
- ✓ name all your files/classes/methods exactly as instructed (the overall project should be named *yourLastName_pr4*)
- ✓ commit your final running Java project into the course SVN folder
- ✓ submit your *final report* on Canvas
- ✓ your project needs to be compatible with the 342 Java / Eclipse version

DESCRIPTION OF ASSIGNMENT

Word counting is used in all sorts of applications, including text analysis, creating indexes, and even cryptography. In this programming assignment, you will take a text file and compute what words appear in that file and how many times they appear using a dictionary/map implemented with different data structures. You will first modify provided code and then run the code on input files of different sizes and different contents to see which data structures perform better in practice.

Overall, the dictionary/map is to be run using a binary search tree, an AVL tree, a hash table, and a Java TreeMap (red-black tree). The code to run the dictionary using a Java TreeMap is already provided – it counts words and prints them – and measures the running time of the building of a dictionary/map. Your job is to provide alternative map data structures that implement the map using a BST, an AVL, and a hash table. Starter code is provided and the specifications for how to modify starter code follow later on in this description.

Once you finish modifying the code for this assignment, you will need to perform experiments to compare and graph actual run times of these different implementations using two different kinds of input files: (1) sorted input and (2) standard English text. For each of the categories, you should run the algorithms on three file sizes: small, medium, and large, and you should run it 3 times for each one to get an average. This means you need to run the program 6x per map type (x 3). The input files are provided in the project in *myfiles* folder.

After you run all your experiments, write a report describing what types of files you performed your experiments on and what conclusions you made. Your report should include all data and 4 graphs all together, with each graph corresponding to the type of a file (sorted vs standard English), with the x-axis showing the number of words being used as input and the y-axis showing the running time:

- 1 graph should depict insertion times only for sorted inputs for all 4 data structures
- 1 graph should show printing times only for sorted inputs for all 4 data structures
- 1 graph should show insertion times only for standard English text inputs for all 4 data structures
- 1 graph should show printing times only for standard English text inputs for all 4 data structures

Make sure to use linear increments on each axis as to not to skew the graph. Label and color your graphs properly so that they are readable. The sample Excel sheet that can help you with generating the graphs for the report and that shows how to format the data tables and the graphs is provided with the assignment. *When you write your conclusions, tie them back to what you know about the Big O run times of the data structures and the operations / algorithms selected for this assignment, i.e. how the actual functions you are getting with your data points correspond (or not) to Big O. If they don't correspond, why do you think that is?*

Code Specifications

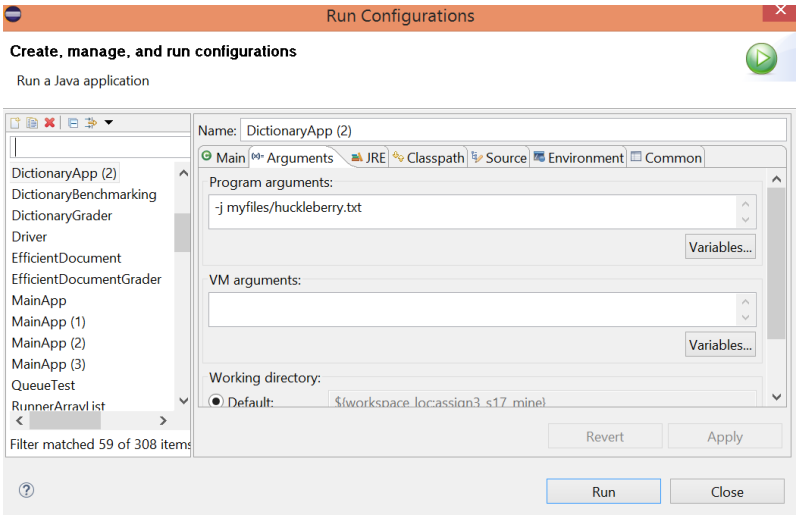
Starter code is provided in *assign4_w18.zip*. It was created using Eclipse export feature, so to get it back, you need to import it into your workspace. File -> Import -> General -> Existing Project into Workspace -> Select Archive File.

MyMap.java

This is an interface that BST, AVL, and the hash table must implement. Your data structures do not need to implement any other public methods. When using generic names for the key and the value in those trees, use the same names, i.e. *AnyKey, AnyValue*.

DictionaryApp.java

This is a driver program written by Donald Chinn and modified by me (Monika Sobolewska). The driver parses the command line argument to determine which algorithm is supposed to be used, and what input file is to be used. It also sets up a timer to time how long it takes to read in and insert the words of certain lengths into the dictionary and then to traverse the dictionary and print it out. There is a method called *getWord* in the driver, which gets the next sequence of contiguous alphanumeric characters from the input stream. This is what we will consider to be a word for the purposes of this assignment. Take a look at the *main* method to see how it is used. You may want to run it first using the Java built-in map on one of the provided text files to get an idea as to how the program ought to work and to get run times for the Java red-black tree. You will need to add the code to make the driver work for the other dictionary/maps and add the second timer that measures printing only. All your code should follow the same structure as the one you see used for the Java TreeMap. To run the program in Eclipse with command-line arguments, you need to select Run -> Run Configurations -> Arguments. Then you will type in your argument, e.g. -j myfiles/huckleberry.txt



BST, AVL, and Hash Table

You are given the code from the textbook for the BST, AVL, and Hash Set, written by Mark Allen Weiss, that you need to modify to implement the interface. For all data structures, you need to modify the map nodes so that they contain both the key and the value components – name these fields *key* and *value*. The generics template needs to be modified as well to contain two generic names instead of one: <*AnyKey, AnyValue*> instead of <*E*>.
Do NOT change any other class or data component names unless required by the interface or by the code provided in *DictionaryApp.java*.
The logic of insert and remove will need to be changed in a fashion consistent with a map – if an item is already in the dictionary/map, do not add a new node but rather replace the value in the existing node that matches the specified key. If an item is removed, you are to remove the node. *toString* method should be traversing the tree in inorder (alphabetic one). For a hash map, it should be traversing it in the order the elements appear in the array. All the functionality should be consistent with the Java TreeMap class (i.e. look at toString() format before writing your own toString method – your toString needs to return the contents formatted in the exact same way).
Your tree methods should use recursive solutions, so you will need private helper methods.
For the HashSet code, in addition to the changes described above, review all the mandatory methods and make sure their logic is correct and consistent with the best practices discussed in class.

When you submit your code, do not retain any Weiss's methods that are not used in your solution.

AVL and Comparator

For the AVL tree, in addition to changes described above that make the tree consistent with a map, you will also need to modify the code to support a Comparator interface. In case you forgot what a Comparator is or it was not covered in your 143 course, the following link serves as a refresher:
https://www.tutorialspoint.com/java/java_using_comparator.htm
There is also a BST.java code provided in the assignment folder that shows how to incorporate a Comparator into a data structure. In addition, in the assignment zip file, there is a file called DictionaryAppComparatorTester.java that uses a Comparator with a Java TreeMap. If you replace the TreeMap with your own AVL tree, the code should work in an identical fashion.

Extra credit:

There are additional data structures files provided in the assignment folder in Canvas: B-Tree, splay tree, and trie. Modify at least two of these structures to serve as maps (if needed) and run experiments on them. Add experiment results to your report. An alternative plan (although worth up to 10 points out of 15) is to provide a tree iterator that the client can use to traverse the trees. If you do that, add another driver, *DictionaryAppEC.java* that uses additional data structures and/or the iterator.

HELP

Whenever in doubt or unsure, your first approach should be to try to solve the issue on your own by consulting the textbook for this course, any other Java textbook you may have from your prior courses, and online Java documentation. Java documentation is available at http://docs.oracle.com/javase/8/docs/api/
When unable to resolve an issue, code the components you can figure out on your own and seek CSS Mentors' help with the issues you need help with, visit instructor's office hours, and stay after class. Ask questions of your classmates but be careful so that you do NOT cross the boundary between helping and plagiarizing.

GRADING RUBRIC

- Report 30%
- BST tree implementation 20%
- AVL tree implementation 20%
- Hash table implementation 20%
- Overall correctness and coding style 10%
- Extra credit 15%