CSC2001F 2019 Assignment 2

Instructions

The goal of this assignment is to compare the AVL Tree with Binary Search Trees, both implemented in Java, using real power consumption data.

Dataset

The attached file represents a time series of power usage for a suburban dwelling. The base data set and additional information can found at

https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption

Load the file into LibreOffice/OpenOffice/MS-Excel to see what the data looks like. Note that the first line consists of headings and needs to be ignored by your code. Every subsequent line represents a single data item/record.

In your application, you MUST write your own code to read in the Comma Separated Value (CSV) file. You may not use a CSV library. Your data structure items must each store the following **3** values extracted from the CSV file:

- "Date/time"
- "Global_active_power" (which we will call "Power")
- "Voltage"

Part 1

Write an application **PowerAVLApp** to read in the attached CSV file and store the data items within an VL tree.

Include the following methods in your code:

- *printDateTime (dateTime)* - to print out the **Date/time**, **Power** and **Voltage** values for the matching *dateTime* record; or "Date/time not found" if there is no match.
- *printAllDateTimes ()* - to print out the data as above, but for **all** date/time records, in any order

You should be able to invoke your application using a command line such as

```
java PowerAVLApp "16/12/2006/17:44:00"
```

to print details for the indicated Date/time string or

```
java PowerAVLApp
```

to print all date/time details.

Finally, you should be able to pass in a **text file** of keys to search for in your data structure

```
java PowerAVLApp FileWithSearchKeys.txt
```

This allows you to build the data structure once in your Java program and then iterate through each of the listed keys to perform a sequence of searches and print each returned record.

You may use quotes in your parameters or not - it is up to you. *Be sure to explain, in your report, how you specify the CSV input to your program.* The java command line does not have to look like the above, this is just to illustrate how you might call your application.

Test your application with **3** known date/time strings, one unknown date/time string and without any parameters. By "unknown string" we mean one that is not present in the data set i.e. and invalid key. Use output redirection in Unix to save the output in each case to different files. Also make sure that your search using the file of keys produces the expected output.

Part 2

Add additional code to your solution to Part 1 to discretely count the number of *comparison* operations (<, >, =) you are performing in *insertion* and *search* methods. *Only count where you are comparing the value of date/time strings*. There are 3 basic steps.

First, create a variable/object (e.g., opCount=0) somewhere in your code to track the counter; maybe use an instance variable in the data structure class.

Secondly, wherever there is an operation you want to count, increment the counter (opCount++). For example:

```
opCount++;   // instrumentation to count comparison

if (queryDateTime == theDateTime) // comparison op
...
```

Finally, report the value of the counter before the program terminates. Perhaps add a method to write the value to a file before the program terminates.

You may want to use 2 variables: one for comparisons in the *insert* method and one for comparisons in the *search* method.

Test your application with **3** known date/time strings, one unknown date/time string and without any parameters. Take note of the operation count in each case. When you test with a **list** of search keys, then you must record the search operation count for each search key you apply to the tree (1 value per search). Note that your **insertion** operation count will not change after the initial insertion phase is complete (this is done once to build the data structure).

Part 3

Write an application **PowerBSTApp** to perform the same tasks as Part 1, but using a Binary Search Tree (BST) instead of an array.

Your BST implementation can be created from scratch or re-used from anywhere (with appropriate code attribution in the comments i.e. credit the source – credit must also appear in your report). You may **NOT** replace the BST with a different data structure.

Once again, test your application with 3 date/time strings, one unknown date/time string and without any parameters. Use output redirection in Unix to save the output in each case to a different file. As before, make sure that your program produces the correct output for a list of keys too.

(This is similar to part 3 from Assignment 1, so you ca reuse that code).

Part 4

Instrument your **PowerBSTApp** code just as you did in Part 2.

Note that this is slightly different from Assignment 1, since we are counting both *insert* and *search* operations.

Test your application with 3 date/time strings, one unknown date/time string and without any parameters. Take note of the operation count in each case. Make sure that your program produces the correct output for a list of keys with an appropriate file.

Part 5

Conduct an experiment with **PowerAVLApp** and **PowerBSTApp** to demonstrate the speed difference when performing *insert* and *search* between the AVL tree and BST.

You want to vary the size of the dataset (N) and measure the number of comparison operations in the best/average/worst case for every value of N (from 1-500).

For each value of N:

- Create a subset of the sample data (you can use the Unix **head** command to extract the first N lines).
- Run both instrumented applications for <u>every</u> date/time string in the subset of the data you are examining. Store all operation count values. Do NOT simply iterate through the subset calling your program repeatedly with a new search key – this would cause the data structure to be rebuilt at each step. Instead, use your search key **file** to get all the keys in at the start of the program and build the data structure ONCE at this point. This will generate the insertion operation count for that subset – this number will not change for that subset. Each key search on the data structure with a new key will produce a new search operation count for that subset. These are the values you need to use for each subset.
- Determine the best case (minimum), worst case (maximum) and average of these count values. That is for an N item subset, you will have a list of N operation counts for **search**

(1 per search you ran on the tree you built): $\{C_1,\ldots, C_N\}$. Find the minimum value, and the maximum value from this set. Then add all the C_i values and divide by N to get the average operation count. You will do this for each value of N=1..500 to generate your graph data.

Note: You can also manage everything **inside** your java program (subset creation, key extraction etc); your report should note how these modifications change required command line parameters. Be sure to explain how you implemented testing to make it as efficient as possible

It is recommended that you use Unix (or Python) scripts to automate this process. You can set up the output so that it consists of text formatted in a sensible way. This can be read by a graphing program which can process text-based input data (e.g. gnuplot, matplotlib (a python module)) or anything else that you find. You can even do this by loading a CSV formatted output file into Excel and plotting with the Excel graph plot functionality (you can save these graphs as PDF/images).

Part 6

Repeat the experiment from **Part 5** for N=500 only (the full data set), but with the data records **pre-sorted** by the *dateTime* key. In order to do this, you may want to load the data into a spreadsheet application, sort on the *dateTime* column and then save/export this as a CSV file.

Report

Write a report (of up to 6 pages text; additional figures can add an extra 4 pages at most) that includes the following (with appropriate section headings):

- What your OO design is: what classes you created and how they interact. You do not need to draw class diagrams.
- What the goal of the experiment is and how you executed the experiment.
- What test values you used in the trial runs (Part 2 and Part 4) and what the operations counts were in each case. Only show the first 10 and last 10 lines for the trial run where you invoke *printAllDateTimes ()*.
- What your final results for Part 5 are (use one or more graphs), showing best, average and worst cases for both applications. Discuss what the results mean. You must illustrate and discuss, for each application, how its *insertion* operation count compares to its *search* operation count. Then illustrate and discuss, for the *search*/*insertion* methods, how the performance differs between the 2 applications.
- What your final Part 6 results are, discussing how the 2 data structures compare in terms of the number of operations. Do **not** draw graphs; for each operation, just report the 3 numbers (best, average, worst) for each data structure.
- A statement of what you included in your application(s) that constitutes creativity - how you went beyond the basic requirements of the assignment.

- Summary statistics from your use of git to demonstrate usage. Print out the first 10 lines and last 10 lines from "git log" , with line numbers added. You can use a Unix command such as:

```
git log | (ln=1; while read l; do echo $ln\: $l; ln=$((ln+1)); done) | (head
-10; echo ...; tail -10)
```

Dev requirements

As a software developer, you are required to make appropriate use of the following tools:

- **git**, for source code management
- **javadoc**, for documentation generation
- **make**, for automation of compilation and documentation generation

Submission requirements

Submit a .tar.gz compressed archive containing:

- Makefile
- src/
    - all source code
- bin/
    - all class files
- doc/
    - javadoc output
- report.pdf

Your report must be in PDF format. **Do not submit the git repository.**

Marking Guidelines

Your assignment will be marked by tutors, using the following marking guide.

Artefact	*Aspect*	*Mark*
Report	Appropriate design of OOP and data structures	5
Report	Experiment description	5
	Trial test values and outputs (Part 2)	2
	Trial test values and outputs (Part 4)	2
	Results - tables and/or graphs	5
	Discussion of results – Part 5	5

	Discussion of results – Part 6	5
	Creativity	5
	Git usage log	3
Code	Looks reasonable and no obvious inefficiencies	5
Dev	Documentation - javadoc	5
	Makefile - compile, docs, clean targets	3