CSC2001F Assignment 2 Report

Done by Taahir Suleman (SLMTAA007)

Experiment:

AVL trees are binary search trees (BST) with an additional balancing property - each nodes' difference in heights between their left and right subtrees must be at most 1. This property ensures that the BST worst case, where it is simply a linear tree (with asymptotic time complexity O(n)), does not occur and that the asymptotic time complexity worst case is instead O(log n). I used prof. Hussein Suleman's sample AVLTree.java code for the AVL tree operations required in my experimental design. This experiment aims to determine the efficacy of AVL trees in balancing nodes and thus producing satisfactory performance regardless of the sorting/ordering of the inputted data. To determine this, I created an experimental design in a class called AVLExperiment that received input from a data file called vaccinations.csv which contains various countries and their vaccination numbers on specified dates in 2022 – it contains 9919 entries. This program extracted the data from this file, created Vaccine objects with each data item (using the Vaccine class from Assignment 1) and added it to a Vaccine ArrayList of objects. The original data set is completely sorted, and I altered it with 21 different increments/levels of specified randomization, inserting this data into an AVL tree each time and returning the various comparisons required to achieve this.

Additionally, I searched for each item from the data set in the tree after it was populated, returning the required comparisons for each search operation. I achieved this using a command-line parameter X which specifies the desired level of randomization on each run in the range 0-9919, with 0 being wholly sorted and unrandomized and 9919 being completely randomized. Specifically, the value of X specifies how many elements of a completely randomized version of the data is added to the start of the ArrayList containing the user-specified degree of random data, before filling the rest of this ArrayList with the remaining elements in the originally sorted version of the data. I used 21 increments of randomization in this range: 0 and the first 20 multiples of 490. I chose this set of increments to ensure that my experiment was performed on a wide range of data sets with differing levels of randomization, going from not randomized at all to mostly randomized. I

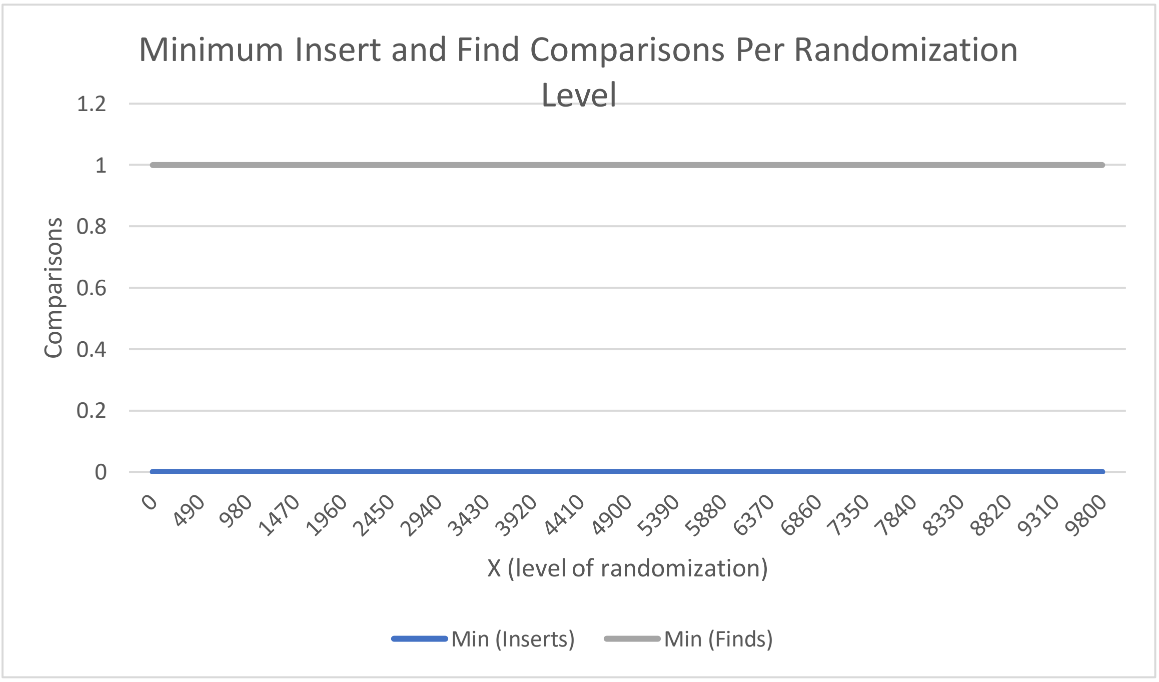
ran AVLExperiment with each of the 21 iterations manually (since it was only 21 increments, this was not too cumbersome). In each case, I inserted the randomized data (to the specified degree of randomization) into an AVL tree (using its insert() methods), counting the number of comparisons between keys required for each insert.
Furthermore, after each insert, I added this value to the summation of all insert comparisons (used to determine average case at the end) and compared it to the current min and max values for insert comparison operations before resetting this value to zero for the next insert operation. After that, I searched for each data item from the randomized data set in the AVL tree (using its find() methods), counting the number of comparisons between keys required for each search. Moreover, after each search, I added this value to the total find comparison operations and compared it to the currently stored min, and max find operation values before resetting it to 0 before the next search. Both of these operation counters are incremented with each instance of a compareTo() method being called in the AVLTree class, where two keys were being compared. I outputted the min, max and average comparison operations from all of the individual insert and find operations in each iteration, storing this output in a file called expOut.csv in the format "(X,min,max,average)" for later inspection and to produce the graphs shown later in this report. My results allowed me to conclude that AVL trees perform satisfactorily and balance nodes throughout regardless of the respective ordering or randomization of the inputted data. This conclusion was derived because the results displayed sufficiently low comparison operations for insert and find operations respectively and differed minimally between different levels of randomized data.

Randomization Approach:

First, I created a randList() method that did the following: It first created and populated an ArrayList of Integers with the sequence of distinct integers from 0 to 9918. After that, I used the shuffle() method from the Collections package to randomize this sequence before returning this ArrayList of integers containing a randomized sequence of the distinct integers between 0 and 9918. This ArrayList represented a list of indices of a completely randomized vArr ArrayList and is used to a specified degree in creating a randomized version of vArr. Next, I created a randomize() method that did the following. Firstly, it created a temp variable storing an ArrayList of Vaccine objects and an ArrayList of Integers containing a random sequence of the distinct integers between 0 and 9918 using the randList() method specified above. Next, I added the elements at the first X (variable specified by a command-line parameter) indices (specified from the Integer ArrayList) in the original vArr to the temp ArrayList of Vaccines.

Furthermore, I removed each of these elements from the original vArr each time and decremented the count variable storing the size of the original vArr. Lastly, I populated the rest of the temp ArrayList with the remaining elements in vArr, before eventually setting the vArr variable to point to this new randomized version so that it may be used for the relevant operations. I believe that this is a good algorithm for randomization as it is easy to use to randomize the sorted vArr ArrayList of Vaccines to a user-specified level. Furthermore, it also ensures that there are sufficiently different degrees of randomization between different specified X levels of randomization, which is necessary for comparing the results produced by the AVL tree between differing degrees of randomized data since this is the primary purpose of the experiment.

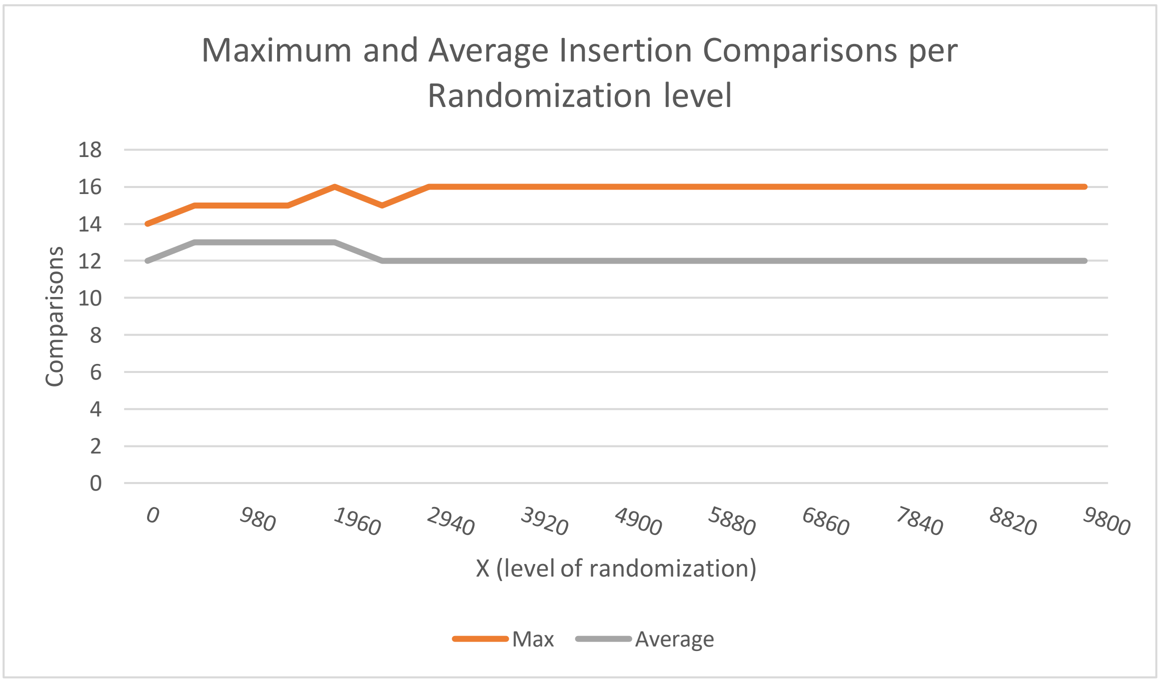
Results with explanations:



Explanation:

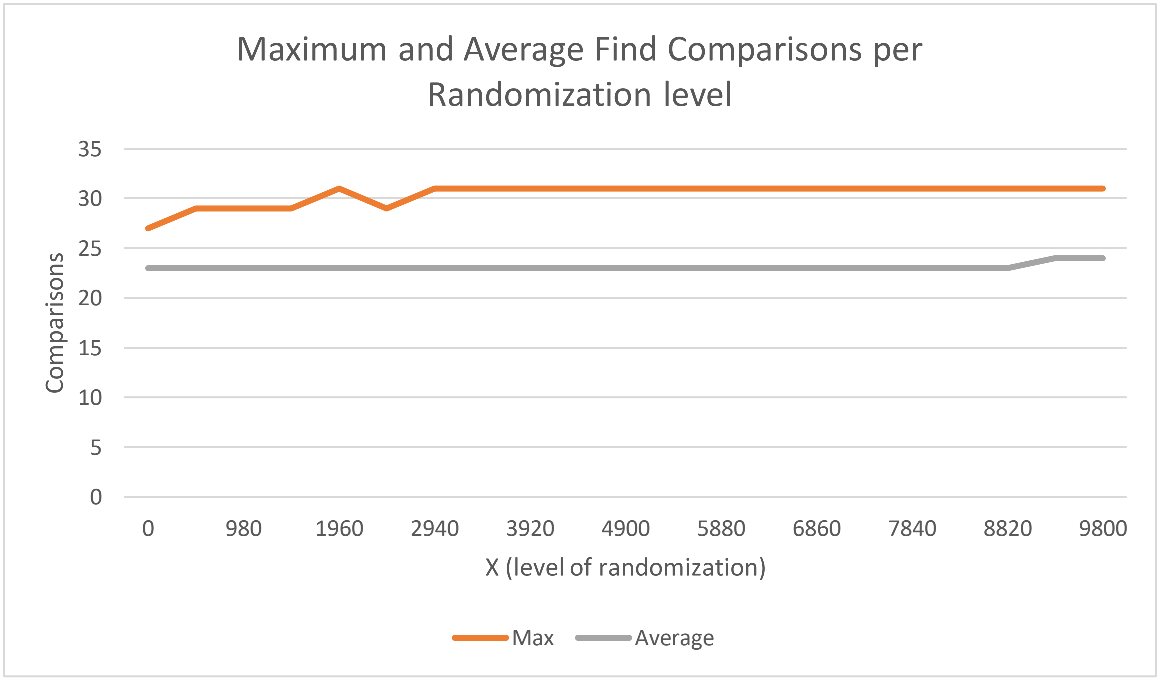
The graph above shows the observed minimum comparisons required for an individual insert and find operation per randomization level (X). For insertions, it demonstrates that this quantity was 0 for each level, which was the expected result – this best case occurs when inserting at the root of the AVL tree as no comparisons are required. Therefore, regardless of the ordering of the data, the AVL tree produced satisfactory performance in terms of the best case of insertion comparison operations, with it requiring 0 for each best case throughout the 21 levels of randomization.

On the other hand, for searches, this graph demonstrates that the best case for a find for every level of randomized data was 1 comparison. This result displays expected behavior, as this best case occurs when searching for an item at the root of the AVL tree, which only requires one comparison between keys (between the data in the root node and the data being searched for). Therefore, this indicates that irrespective of the extent of randomization in the data, the best case for a find behaved as expected with only one comparison required, emphasizing the satisfactory performance of the AVL tree.



Explanation (Note, the level of randomization increases by multiples of 490, the axis is just as such for better clarity of the graph.):

The graph above displays the max and average comparisons required for an insertion into the AVL tree for each level of randomized data. These results display minimal variation between randomization levels, with the average being in the small range of either 12 or 13 and max being between 14 and 16. Furthermore, the averages eventually stagnate at 12 for a large portion of the randomization levels, with the maximums having a similar stagnation at 16 comparisons. Moreover, the worst (max) and average cases differ minimally at each randomization level, portraying that there were no extreme values for required comparisons. These observations from the results highlight that, regardless of the ordering of the inputted data, the AVL tree did balance the nodes and provide good performance throughout the different levels of randomized data, thus supporting the primary goal of this experiment since there are no extreme differences between different randomization levels and max and average cases.



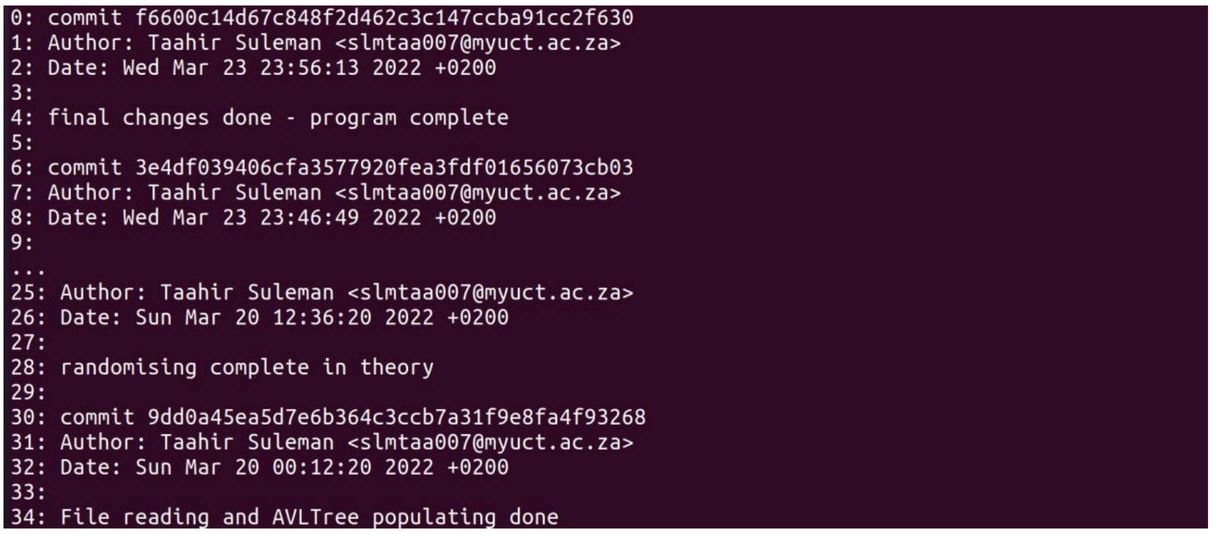
Explanation (Note, the level of randomization increases by multiples of 490, the axis is just as such for better clarity of the graph.):

The graph above depicts the worst and average cases for comparisons required for a find operation between the differing randomization levels. Firstly, these graphs show minimal differences between the worst and average cases, indicating that the worst cases are not vastly extreme, highlighting good performance of the AVL tree and the presence of balance nodes. Secondly, there are minimal differences between the comparisons required in the worst case between the different randomization levels, with its values ranging between 29 and 31 and eventually stagnating at 31 comparisons for more than half the levels of randomized data. Thus, the worst case results portray that the AVL tree does balance nodes and produces satisfactory performance regardless of the ordering of the inputted data. Similarly, the average case barely differs between the different randomization levels, with it being constant for most levels and increasing by one from 23 to 24 for the last two (most randomized) randomization levels. Therefore, this further emphasizes the idea that AVL trees do balance nodes well and perform satisfactorily regardless of the ordering of the inputted data, once again supporting the stance outlined in the goal of the experiment.

Creativity:

In achieving my desired randomization, I took a creative approach in how I used an ArrayList of integers and a method from the Collections called shuffle to produce a random sequence of the distinct integers from 0 to 9918. This method allowed this to be possible and was one example of me going over and above what we were taught in achieving the goals set out by this assignment. Furthermore, I formatted the output in the manner that I did – "(X,min,max,average)" to make it easy to convert this into graphs using excel operations to first remove the brackets, delimit each row by commas to separate it into separate rows and then use this data to produce the graphs seen above. Lastly, throughout my program I manipulated vArr rather than vaccinations.csv to reduce the work required when reading and randomizing the data, once again going over and above the requirements.

Git log:



```
commit f6600c14d67c848f2d462c3c147ccba91cc2f630
Author: Taahir Suleman <slmtaa007@myuct.ac.za>
Date: Wed Mar 23 23:56:13 2022 +0200

final changes done - program complete

commit 3e4df039406cfa3577920fea3fdf01656073cb03
Author: Taahir Suleman <slmtaa007@myuct.ac.za>
Date: Wed Mar 23 23:46:49 2022 +0200

...
Author: Taahir Suleman <slmtaa007@myuct.ac.za>
Date: Sun Mar 20 12:36:20 2022 +0200

randomising complete in theory

commit 9dd0a45ea5d7e6b364c3ccb7a31f9e8fa4f93268
Author: Taahir Suleman <slmtaa007@myuct.ac.za>
Date: Sun Mar 20 00:12:20 2022 +0200

File reading and AVLTree populating done
```