University of Cape Town

CSC2001F 2022

Assignment 2

Author: NJMLUN002 [1438523]

Date: 29-03-2022

AVL Tree Experiment to Characterise Node Rebalancing with Randomized Data

Abstract

An AVL Tree is a Binary Search Tree data structure satisfying additional AVL balancing conditions. An experiment was conducted with the AVL tree implemented in Java to determine the extent of node rebalancing and the effectiveness of node balancing in providing good performance, irrespective of the order of data. This was accomplished by randomising a set of data with varying degrees of randomisation. After randomisation, the data was inserted into an AVL tree while using instrumentation to calculate the number of node comparison operations. Each data item was then searched for in the same AVL tree and implementing instrumentation on the number of search-related comparison operations.

The original data consisted of 9919 entries corresponding to countries' vaccination data for 2022 only consisting of a country name, the date on which the data was recorded and the number of vaccinations, to-date. All the data was loaded into a one-dimensional array of storing string items. The *nextInt*(<upperbound>) method from the Random class in the java.util package was used to generate two unique pseudo-random numbers, ran*dIndex*0 and randIn*dex1,* between 0 and 9918 corresponding to the available array indices. The item in the array corresponding to randIndex0 was swapped with the element in the array with index corresponding to randIndex1. The number of times that indices were swapped in this manner was specified by the degree of randomisation which ranged from 1 to 20. The degree of randomisation was multiplied by four-hundred and ninety-five (495) to determine the number of indices swapped and so that the amount of data randomisation would be spread out by approximately the same degree. The program used a recursive insert and search algorithms to add data to the AVL tree and to find data in the AVL tree. The insert algorithm was expected to satisfy AVL conditions by rebalancing the nodes in the tree such that, after each insertion and at all nodes, the difference between the right-hand subtree height and the left-hand subtree height was at most one. Both the insert algorithm and the search algorithm were expected to have a best time complexity of O(1) and average and worst time complexities of O(logn) for n nodes in the AVL tree. In the experimental control, the data was inserted into the tree and searched for without randomising the original data to improve the inference that the change in the dependent comparison operation variables were due to the changing independent degree of randomisation variable.

Results showed that the AVL tree effectively rebalanced the nodes in the tree as the number of insert-related and search-related comparisons decreased with an increase in the degree of randomisation in the data. This showed that the height of the tree was minimised as the randomisation increased. The smallest number of insert-related comparisons achieved by the program was 39, corresponding to the degree of randomisation of 19. The highest number of insert-related comparisons was 187521 comparisons corresponding to no randomisation of the original data. The results also showed that when the least number of insertion-related comparisons were performed, the least number of search-related comparisons was performed at 9958 comparisons. Similarly, the highest number of search-related comparisons was 197440, corresponding to the highest number of insertion related comparisons.

Introduction

AVL trees are Binary Search Tree data structures with additional tree balancing conditions. In an AVL tree, each node is assigned a balancing factor that ensures that the height of the left-hand subtree and the height of the right-hand subtree differ by at most one. The AVL tree was named after Abelson-Velvety and Landis who invented the data structure in 1962.

The advantage of AVL trees is the elimination of cases where each node has exactly one child and that the nodes are distributed more effectively so that in the worst case, the tree is still balanced and faster operations can be performed. AVL trees are typically shorter in height than standard Binary Search Trees. However, the disadvantage of AVL trees compared to standard Binary Search Trees is that balancing property requires additonal operations for adding and removing nodes such as tree rotations and recalculating the balance factor.

An experiment was conducted to determine the level of balancing performed by AVL trees and whether this provides good performance irrespective of the order of the data. This was done by creating a Java application called AVLExperiment which randomised given data, inserted the randomised data into an AVL tree and searched for each item in the tree while instrumenting the number of insertion-related and search-related comparison operations. The program instrumentation was used to determine the best, average and worst time complexities of the insert and search operations of the AVL tree and to determine how randomising the data influenced the performance of the program.

The following report describes the theory behind the time complexity of insertion and search operations for the AVL tree. Following the theory is a description of the experimental method and the results. The results are assimilated in the discussion and a conclusion is made in the final section.

Theory

The worst case time complexity of the insert, delete and find algorithms of a Binary Search Tree are all O(n) where n is equal to the height of the tree. Balancing the tree reduces the height of the tree and distributes the nodes more evenly across the tree. AVL trees can be considered as balancing Binary Search Trees that self-balance by performing appropriate tree rotations when a node is inserted or removed, to ensure that each node has the property that the height of the subtree rooted at its child have a maximum height difference of one.

An AVL tree with n nodes has a height of O(logn) which means that in the worse case, the insert, delete and find algorithms have a worst case time complexity of O(logn) [2]. This can be shown using Proof by Induction. Let N represent the minimum number of nodes that can form an AVL tree of height h. This implies that the root must have a child with height h – 1. Since this is an AVL tree, the height of the other child of the root must have a height of at most h – 2. Thus the AVL tree has a total of (h – 1) + (h – 2) + 1. The base case is given for $N_1 = 1$ and $N_2 = 2$. The number of nodes N can be constructed iteratively given that $N = N_{h-2} + N_{h-1} + 1$:

$$N_h = N_{h-1} + N_{h-2} + 1$$

By induction,

$$N_{h-1} = N_{h-2} + N_{h-3} + 1$$

$$\Rightarrow \quad N_h = (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1$$

$$\Rightarrow \quad N_h > 2N_{h-2} \qquad ...(1)$$

A binary tree with height h has at most 2^h nodes, therefore we can rewrite equation (1) as

$$N_h > 2^{h/2}$$

$$\log(N_h) > \log(2^{h/2})$$

$$\Rightarrow \quad h < 2\log N_h$$

Hence, the height of the AVL tree is O(logn).

The fact that the height of the AVL tree is O(logn) implies that the worst case time complexity is order n. If only one insert operation is performed, then the AVL tree has height 1 and this is the best case for the insertion operation which is O(1). Likewise, the best case for search algorithm corresponds to an AVL tree height of 1 or when the searched item is the root. The worst case of corresponds to the scenario where the searched item is in the last level, i.e. level h+1 corresponding to the maximum height of the tree. We can also insert an item into the last level of the tree for a worst case time complexity of O(logn) corresponding to the worst case for the height of the AVL tree. In the worst cases, a comparison has to be done at each node along the path of a tree to the last level in the tree.

The Random class in Java uses a Linear Congruential Generator algorithm to generate pseudo-random numbers which are a series of seemingly random numbers that are generated by a deterministic algorithm [1]. The algorithm begins with a seed which is suitably random for application in this program. Each time a new random number is requested using the nextInt() method of the class, multiplication, addition and modulo operations are performed on the current seed to obtain the new random number. Since operations are performed on the current seed, the next seed is always predictable based on the initial value and the generated random number repeat after a large number of generations. Therefore Random class chooses values for a maximum period such that values repeat after 2^{48} generations which was more than sufficient for application in this experiment.

Experimental Method

A program called AVLExperiment was designed to take in a file containing 9919 data entries holding information about a country, a date and the number of vaccinations completed on that date. The AVLExperiment program used the DataArray class to store the data in a one-dimensional array of String elements. This array was shuffled to a degree of randomisation between 0 and 20 where 0 corresponded to no shuffling and 20 corresponded to all the data being shuffled in the array. The degree of randomisation was multiplied by 495 to determine the number of indices that would be shuffled in the array.

The random shuffling of the indices was performed using the pseudo-random number generator in the Random class. Two random integers, *randIndex0* and *randIndex1*, were generated in the range [0, 9919) then the data item in the array index corresponding to *randIndex0* was swapped with the data item in the array index correspond to *randIndex1*. The number of random numbers that were generated was equal to a multiple of the degree of randomisation and 495 (which is 9919 divided by 20).

After the data had been randomised, an object of the Country class was created to store the country name, date and number of vaccinations and to implement the Comparable API. The name and the date were used to generate a key for each data item for inserting and searching data in the AVL tree. A recursive insert and search algorithm were implemented by the program. For the insert algorithm, if the tree's root was empty, then a node was added to the tree as the root. Otherwise the node was compared to the root and if the key was greater than the key of the root node, the new node was added to right-hand subtree. If the key was smaller, then the new node was added to the left-hand subtree.

The insert algorithm ensured that the AVL tree balancing conditions were satisfied by implementing tree rotation methods to rebalance the tree whenever a new node was added to the tree such that the left-hand and right-hand subtree differed by more than one. An imbalance caused by an insertion into the left subtree of a left child was resolved by a right-rotation of the tree. Likewise, a left-rotation was performed when an imbalance occurred after an insertion into the right subtree of the right child. Double rotations were performed for imbalances caused by insertions into the right subtree of the left child or left subtree of the right child.

The search algorithm recursively compared the search item to each node until the node was found. Starting at the root of the AVL tree, if the key of the new node was greater than the key of the root, the algorithm recursively searched down the right subtree. Alternatively, if the key was smaller, then the program recursively searched down the left subtree.

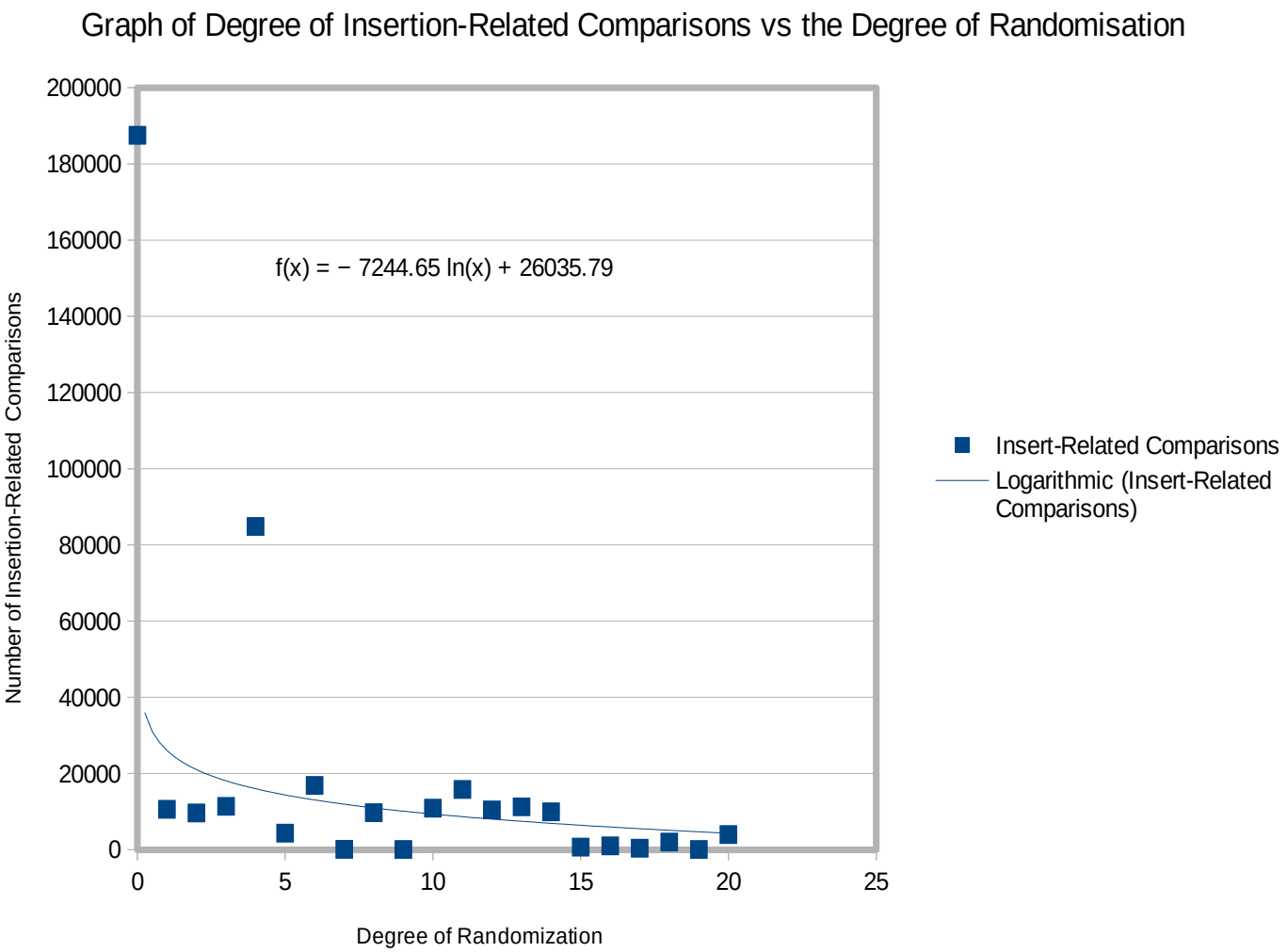
For each insertion and each item search, instrumentation was used to determine the number of comparisons performed by the operation. The instrumentation for insertion-related operations was calculated by incrementing a static variable every time the key of a new node was compared to the key of an existing node in the tree. Similarly, the a static variable was incremented for cases where keys were compared during a search operation for search-related instrumentation. The instrumentation was output to a file for processing of the results were a the degree of randomisation was the independent variable and the number of comparisons were the dependent variable. These results were used to determine the best, average and worst case time complexity of the AVL tree to see whether balancing was effective and how randomising the data affected the performance of the AVL tree.

Results

The table below shows the results from running the AVLExperiment Java application. The instrumentation output was written to individual files separating the number of insertion-related comparison count and the search-related comparisons.

Degree of Randomization	Number of Swaps	Insert-Related Comparisons	Search-Related Comparisons
0	0	187521	197440
1	495	10620	20539
2	990	9653	19572
3	1485	11378	21297
4	1980	84864	18383
5	2475	4357	14276
6	2970	16863	26782
7	3465	93	10012
8	3960	9716	19635
9	4455	51	9970
10	4950	10892	20811
11	5445	15823	25742
12	5940	10427	20346
13	6435	11258	21177
14	6930	9944	19863
15	7425	687	10606
16	7920	995	10914
17	8415	379	10298
18	8910	2003	11922
19	9405	39	9958
20	9900	3967	13886

The graph below shows a logarithmic approximation of the increase in the AVL performance of as the degree of randomisation increased.



Discussion

Conclusion

References

[1] Albrecht, J. *A Comparison of Mersenne Twister and Linear Congruential Random Number Generators*. 2007.

[2] Drachsler, D. et. al. *Practical Concurrent Binary Search Trees via Logical Ordering*. Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2014.

Git Usage Log

```
0: commit 2e734559bc0c7236f551c2e65af32b3630a67854
1: Author: NJMLUN002 <njmlun002@myuct.ac.za>
2: Date: Mon Mar 28 16:07:09 2022 +0200
3:
4: modified avlexperiment to take degrees of randomisation between 0 and 21
5:
6: commit c9cbe09d91f9bedd0b2e0617e5810aa303b6d1c7
7: Author: NJMLUN002 <njmlun002@myuct.ac.za>
8: Date: Mon Mar 28 00:08:23 2022 +0200
9:
...
31: Author: NJMLUN002 <njmlun002@myuct.ac.za>
32: Date: Sat Mar 12 12:02:55 2022 +0200
33:
34: Updated AVL tree class
35:
36: commit 28df6b606e1ffd845733a5ae888513cdc7dde65d
37: Author: NJMLUN002 <njmlun002@myuct.ac.za>
38: Date: Sat Mar 12 11:44:55 2022 +0200
39:
40: Created AVL tree interface and updated Binary Tree Node class and interface
```