Search Tree Experiments

Alex Blackwell – 301397565

November 2020

1 Introduction

The efficiency of the insert, remove, and search operations of a tree are proportional to its height. A self-balancing tree such as an AVL (Adelson-Velskii and Landis) tree ensures that the height never grows to be more than $2\log(N)$ where N is the number of nodes in the tree. The effect of this self-balancing property is that the insert, remove, and search operations of AVL trees can be described asymptotically as $\log(N)$. The same is not true for a non-balancing BST (Binary Search Tree). A BST can degenerate into a linked list if children are either all decreasing or all increasing from the root. In this scenario, the three considered operations become $O(N)$ with N the number of nodes in the tree.

The AVL tree asymptotic description implies it is more efficient than the BST; however, the AVL operations are more complex, and it is unlikely for a BST to have height proportional to its nodes when the operations and keys are random.

This paper will consider randomly ordered insertions, deletions, and searches of both trees where all operations operate with a random number. The random order and equally likely sequence of inserted and deleted numbers imply the average height of the BST is $O(\log(N))$[1, pp. 141]. The considered operations are proportional to the height, making the considered BST operations $O(\log(N))$ on average. This matches the Big-Oh of the self-balancing AVL tree operations; however, the BST operations are less complicated.

2 Testing Implementation

To generate data that suggests both the AVL tree and the BST insert, remove, and search operations are asymptotically $O(\log(N))$, these operation times were averaged with an increasing number of nodes in the trees. Large trees take longer to test than small trees, so data on small trees was emphasized with large tree tests occurring less often. The smallest sized tree tested had at most six thousand nodes, and the largest tree tested had at most ten million nodes. Fourteen trees of sizes following the recurrence relation

$T_n = T_{n-1} + 2^n + m, \quad 4000 < m \leq 5000, \quad 5000 < T_0 \leq 6000, \quad n > 0$
are generated and tested between this lower and upper bound. This gives an exponential distribution that emphasizes more tests on smaller, easier to test trees. This recurrence relation has a margin of one thousand because of how the sequence of operations is generated.

If the trees were given a long sequence of only one operation at a time, the operation times would increase as the trees grow larger and decrease as the trees lose nodes. Doing the three operations in any order would likely either under or over represent the speed of the insert operation and over-represent the speed of the remove operation. This is because it is expected that the operations become slower as the trees grow larger. The insert operation increases the number of nodes and the remove operation decreases the number of nodes. Testing of the insert provides a baseline for a given tree size before or after the series of inserts; this size, however, is certain to change with the remaining operations.

The method of performing operations on the trees was randomized as much as possible. The method attempts to limit the long series of one operation without, to an extent, disallowing a greater number of one operation if it is to occur randomly. At each of the fourteen differing sizes of trees, a long sequence of operations is performed. This long sequence consists of a maximum of five thousand of each operation. The three operations are equally likely to be selected and performed once to both trees. If at any point an operation is underrepresented by a margin of one thousand, the operation is performed to both trees. This ensures that while the sequence of operations is largely left to chance, they will not deviate by more than one thousand. When any one of these three operations reaches five thousand, the sequence of operations is complete for the current tree size.

When invoking any of the three operations on both trees, random numbers were used. These numbers were integers ranging between $[0, 10000000)$, as to allow for a maximum-sized tree of ten million and to make repeatedly chosen numbers typically uncommon, but possible as the trees grow larger. If a duplicate number was chosen, the operation would attempt to use the duplicate key on both trees.

3 Data Analysis

In general, the height of an AVL tree is at most roughly $1.44 \log(N+2) - 1.328$ [1, pp.145]. By our data, this is true for every tree of differing sizes. The randomness of the keys implies a height of $O(\log(N))$ for the BST. This average asymptotic prediction is supported by the data following a logarithmic trendline.

The self-balancing invariant in the AVL tree results in a tree of lesser height when compared to the height of the BST. It is expected that a tree with less height is faster to insert, remove, and search on because fewer comparisons must be made to find the deepest node. It should also be noticed, however, that the AVL tree Big-Oh is equal to the BST Big-Oh. This is because there are constants $n_0, c > 0$ for every $n > n_0$, that the AVL Tree logarithmic trendline

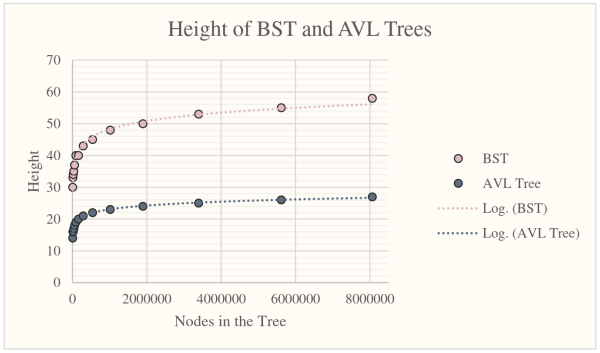

Figure 1: Height of BST and AVL trees

is greater than the BST trendline and vice-versa. More specifically, considering this data set, the BST height is 2.0868 times greater than the height of the AVL tree overall.

What was seen in the analysis of the tree's heights is echoed in the average depths. Both tree's average depths are supported by the data to be $O(\log(N))$ asymptotically. When considering average depths, however, the difference between the AVL tree and the BST is not as drastic as with the differing heights. This is likely because of the lack of a self-balancing invariant in the BST allows for "skinny" subtrees with few nodes but great depth. The probability of having "skinny" subtrees that add greatly to the height of the BST without influencing the arithmetic mean towards a higher average depth is likely. The AVL tree average depth is not greatly different than the height of the AVL tree. This adds to the evidence that the AVL tree is "short and bushy" because the majority of the nodes are stored near the bottom of the tree.

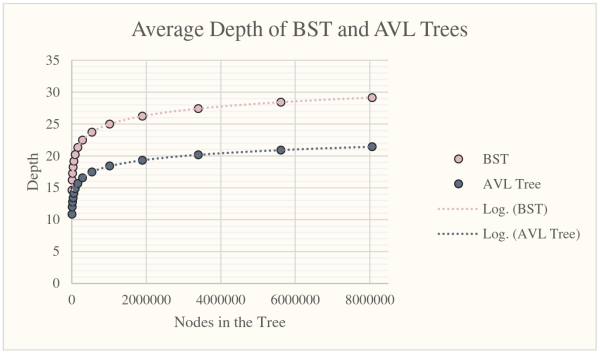

Figure 2: Average depth

Considering this data set from these two trees, overall, the BST's height is 1.9419 times greater than its average depth, and the AVL tree's height is 1.2629 times greater than its average depth. Comparing the average depth of the BST

to the AVL tree, the BST's average depth is 1.3571 times greater than that of the AVL tree.

The average insertion and deletion times share many of the same qualities. The data shows for both the BST and the AVL tree across all considered sizes, the average insertion time and the average removal time is faster on the BST. Both trees share a Big-Oh of $O(\log(N))$ as supported by the data, however, the BST performs faster than the AVL tree. From previous data analysis of the BST height and average depth, the AVL tree had less height and a lower average depth than the BST. However, the additional complexity of balancing in the AVL tree increases the insertion and removal times by a factor $C > 1$ when compared to the BST average times.

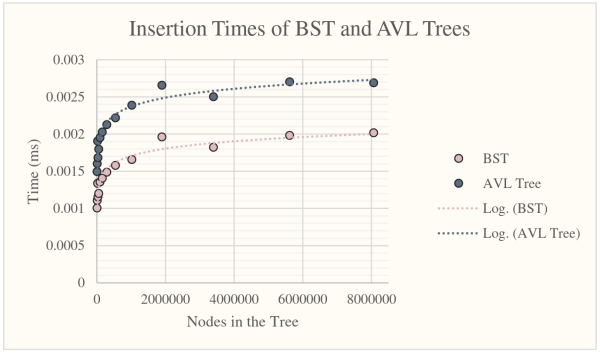

Figure 3: Average insertion times

Considering this data set from these two trees, overall, the BST completes the insertion operation an average of 1.4115 times faster than the AVL tree.

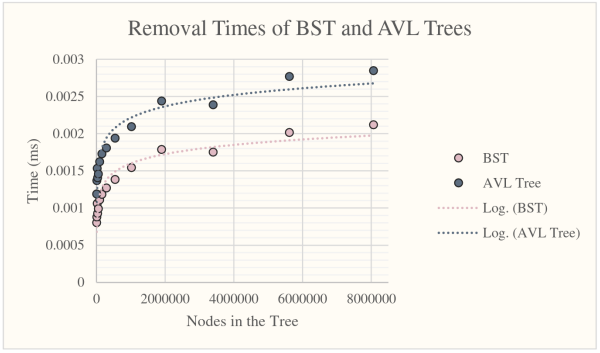

Figure 4: Average removal times

Considering this data set from these two trees, overall, the BST completes the insertion operation an average of 1.4118 times faster than the AVL tree.

The search times for both trees are closer than the other operations with the AVL tree being slightly faster on average. Both trees share a Big-Oh of

$O(\log(N))$ where N is the number of nodes in the tree. The AVL tree search times are faster than that of the BST because both the height and the average depth of the AVL tree is less than the BST. The search operation traverses through the tree making comparisons where the maximum number of comparisons made is proportional to the height; moreover, the average number of comparisons made is the average depth of the nodes in the tree. This is the first operation where the AVL tree is shown to be faster than the BST. This is because the search implementation of both trees is identical implying the AVL search operation does not have extra complexity.

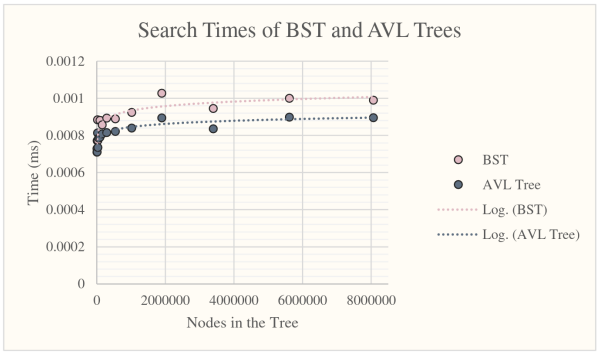

Figure 5: Average search times

Considering this data set from these two trees, overall, the AVL tree completes the Search operation an average of 1.0898 times faster than the AVL tree.

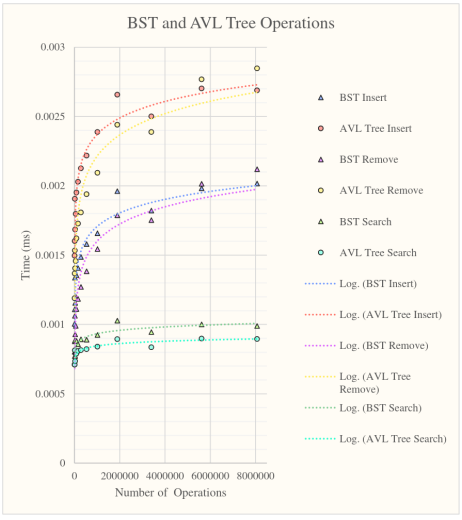

Figure 6: Average operation times

The data from Figures 3, 4, and 5 are summarized showing their relative average speeds. This graph shows the relative speeds of each operation. The insertion and removal operations are similar considering the BST and AVL tree independently, and when comparing the trees, the AVL tree performs these operations slower. The search operation is the fastest of the three operations. This result is to be expected because the insertion and removal operations perform a search and also, modify the tree unless the key is a duplicate—which is unlikely in this observational study.

4 Conclusion

For both the BST and AVL tree, the three operations operate in $O\log(N))$ time where N is the number of nodes in the tree. The additional complexity from the self-balancing AVL tree makes the insertion and deletion times slower by a constant factor $C > 1$. The AVL tree is of lesser height and has an average node depth lower than the BST. The AVL tree having a lower average node depth makes the average search run time lower than the BST's average search time by a constant factor $0 < C < 1$.

The randomization of keys and sequence of operations resulted in both trees being of height $O\log(N))$ and the tested operations for both trees reflected a run time of $O\log(N))$. While the AVL tree was of lesser height and a lower average node depth, insert and remove were faster on the BST with search being slightly faster on the AVL tree.

5 Included Files

`AVLTree.h`

- The AVL tree implementation.

`BinarySearchTree.h`

- The Binary Search Tree (BST) implementation.

`iplTest.cpp`

- The testing program for obtaining data on the tree size, height, average depth, as well as average insertion, removal, and search times.
- Tests trees of varying sizes with an emphasis on smaller trees spanning to trees with a maximum size of ten million.

`data.csv`

- Generated file containing both BST and AVL tree data.

- Data includes tree size, height, average depth, as well as average insertion, removal, and search times.
- Data includes the number of operations after obtaining a tree size and performing a long sequence of operations.

`ipl`

- Executable file to run the tests generated from the command make all.

6 Data

BST Data						
Number of Operations	BST Size	BST height	BST Average Depth	BST Average Insertion Time	BST Average Deletion time	BST Average Search time
14887	5920	30	14.6231	0.001005	0.000803	0.000729
29736	12907	33	16.2152	0.001108	0.000878	0.000771
44651	21866	34	17.2845	0.001337	0.001063	0.000884
59580	34772	35	18.2461	0.001155	0.000928	0.000772
74183	55646	37	19.1791	0.001202	0.000990	0.000807
89009	92262	40	20.2087	0.001355	0.001111	0.000881
103766	160295	40	21.3198	0.001407	0.001183	0.000857
118716	290127	43	22.4909	0.001487	0.001270	0.000893
133403	540136	45	23.7313	0.001580	0.001383	0.000889
148242	1016454	48	24.9941	0.001658	0.001543	0.000924
163109	1893427	50	26.2411	0.001961	0.001786	0.001027
177877	3396629	53	27.4081	0.001822	0.001753	0.000945
192716	5616400	55	28.4143	0.001981	0.002014	0.001000
207544	8064407	58	29.1354	0.002017	0.002120	0.000989

Table 1: Data for BST

AVL Tree Data						
Number of Operations	AVL Tree Size	AVL Tree height	AVL Tree Average Depth	AVL Tree Average Insertion Time	AVL Tree Average Deletion time	AVL Tree Average Search time
14887	5920	14	10.8529	0.001495	0.001189	0.000710
29736	12907	16	12.0044	0.001601	0.001367	0.000710
44651	21866	16	12.7706	0.001906	0.001533	0.000813
59580	34772	17	13.4394	0.001684	0.001406	0.000735
74183	55646	18	14.1481	0.001797	0.001457	0.000786
89009	92262	19	14.8806	0.001952	0.001621	0.000789
103766	160295	20	15.6894	0.002029	0.001727	0.000808
118716	290127	21	16.5587	0.002126	0.001808	0.000815
133403	540136	22	17.4729	0.002218	0.001939	0.000821
148242	1016454	23	18.4238	0.002388	0.002094	0.000839
163109	1893427	24	19.3081	0.002657	0.002440	0.000894
177877	3396629	25	20.1679	0.002501	0.002387	0.000835
192716	5616400	26	20.9029	0.002702	0.002767	0.000898
207544	8064407	27	21.431	0.002689	0.002847	0.000895

Table 2: Data for AVL tree

References

[1] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++ Fourth Edition.* Addison-Wesley, 2014. ISBN: 0780132847377.