Ethan Campbell
Date: November 26, 2021
Computing Science, The King's University

Height Analysis of AVL and Red-Black Trees in Java

In this report the heights of AVL trees and red-black trees are compared after inserting various sets of data. The datasets are various sizes, containing either random and in-order integers. It is important to analyze the height of trees because trees with shorter heights have faster lookup times. When large sets of data are being used and/or frequent lookups are being performed then having a tree algorithm that creates a shorter height can save significant time and money. Therefore, in this report having a shorter height is considered the metric for success. While they were not looked at directly in this report, the heights regular binary search trees (BSTs) were compared against AVL trees in the previous report "Height Analysis of Binary Search and AVL Trees in Java," so their performance compared to red-black tree types will be briefly mentioned. In this report, the AVL tree outperformed the red-black tree with both random and in-order data. The difference was relatively small but significant. Neither tree performed significantly worse with in-order data like the BST did previously.

Introduction

Binary search trees are one of the fundamental ways to store data. Their prevalence is likely due to them being simple to understand and implement, while still providing significant advantages over basic data structures such as linked lists and arrays. However, under certain circumstances these advantages can deteriorate or completely vanish, such as with in-order or reverse-order data insertion. This is problematic because it can cause BSTs to take up more space than the more mundane data structures without the increased lookup times, which can lead to wasted money and other resources. AVL and red-black trees both seek to prevent this degeneration by implementing self-balancing. This means that as values are inserted or deleted from them they automatically check for imbalances (what constitutes an imbalance for each tree is addressed in the methods) and rebalance themselves accordingly. This should keep the height of the trees to a minimum, and in turn increase the speed of insertion and traversal. Since both of these tree structures are intended to minimize heights, it is important to compare which is more effective to make choosing between them more clear.

Background

According to Martinex and Roura (1998), the primary tasks of binary search trees and their variations is to store and retrieve key values held in nodes and to insert and delete more of these nodes. Therefore, any way to increase the efficiency of these actions is greatly beneficial. Bernhard et. al. (2015) says that AVL trees were first conceived by Adel'son-Vel'skii and Landis (hence AVL) in 1962, with the goal of better performance. Red-black trees are another form of self-balancing trees that have four properties that must be maintained after every operation in order to stay balanced, states Xhakaj and Liew (2015). These properties are addressed in the method section.

Method

Both search trees were implemented with Java in a Linux environment running Ubuntu 20.04.3 LTS 64-bit with an Intel Core i7-9700K CPU and 15.4 GB of RAM.

Two sets of tests were run for the sorting algorithms. The first was inserting a set of random integers in both trees. In order to ensure both trees were given the same data, a function was used to generate an array of a specified size with values between zero and twice that size. Sets of 50 000, 100 000, 500 000, 1 000 000, and 5 000 000 were used, each for 100 tests. The second set of tests was inserting in-order data from zero to one less than the specified size. The specified sizes were 50 000, 100 000, 500 000, 1 000 000, and 5 000 000. Reverse in-order tests were not done since in this case that will provide the same results as in-order data with these tree types. For both sets of tests, the ratio of the height of the red-black to the height of the AVL tree for that set was outputted. The ratios for each size were averaged and compared to the ratios of the other sizes (random and in-order were separated).

Both types of tree were based on a basic BST, with a node class that contained a key value of type integer, pointers to two more nodes initially set to null, and a height integer initially set to 1. A node's height was recursively updated as more nodes were added to its children. Whenever a node is inserted it is moved to the left of a node if its key value is lower, or to the right if its value is higher (nodes with repeat key values are not added to the trees). This process continues recursively until the appropriate null location is found, and then the node is inserted there.

The AVL tree rebalances by comparing the heights of all sibling nodes as it recurses back up after insertion. If the difference is more than one then the proper rotation is called to correct the balance. To do this, as the recursion is undone, two trailing pointers keep track of the last two nodes visited. When an imbalance is found, the key value of the higher of the two nodes is compared against the values of the trailing pointers. This will identify the appropriate rotation needed and a function call will be given. Since this is all done recursively, if one insertion causes multiple height imbalances then they will all be caught and fixed on the way back up.

Red-black trees stay balanced by following a set of rules that have to be checked for after any insertion (or deletion, which was not implemented in this case). These rules are based on the fact that every node has a “colour”, red or black, that is stored as a char in the node. The root node must always be black, which is simply enforced by returning it’s colour to black at the end of any insertion call. Leaf nodes are treated as black, but cannot store a char since they are null. When a new node is inserted, it starts out as red. No red node can also have a child that is red. After insertion, this is what the algorithm checks for to ensure balance. As the recursion is undone, if a node is red then the program checks to see if either of its children are red. If one of them is, then a flag is tripped to signal the program to find the appropriate solutions. The solution depends on the colour of the node’s sibling, and involves changing the colours of the node and/or the grandparent, and possibly a rotation. If a rotation is required then the appropriate rotation is determined, and then a flag is set to perform that rotation on the next level of recursion, since the root of the rotation is the grandparent. When these steps are properly followed, the number of black nodes between the root and any leaf node will be the same. Red-black trees do not actually need to store the height of each node, but in this report they did for the purposes of comparison to AVL trees.

Results

Figures 1 and 2 illustrate the results from the random data tests. Both graphs contain the same data, but Figure 2 has the tests with 5 000 000 values removed to better show the differences between smaller tests. Figures 3 and 4 illustrate the results from the in-order data tests. Figure 4 has the tests with 5 000 000 values removed to better show the differences between smaller tests. For all tests the data points represent the average ratio of the heights of the red-black and AVL trees for that number of integers inserted.

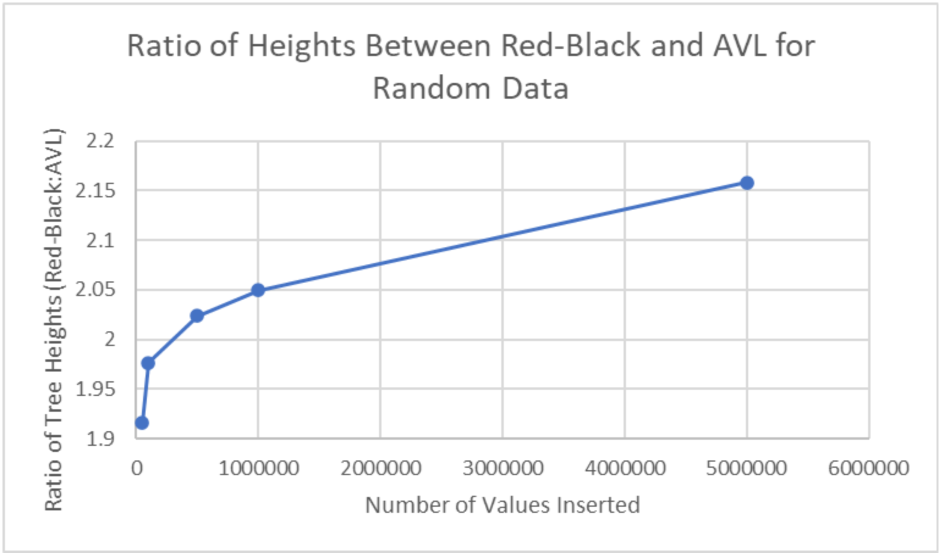

Figure 1. The ratio of the heights of red-black trees vs AVL trees for random data sets of 50 000, 100 000, 500 000, 1 000 000, and 5 000 000 integers.

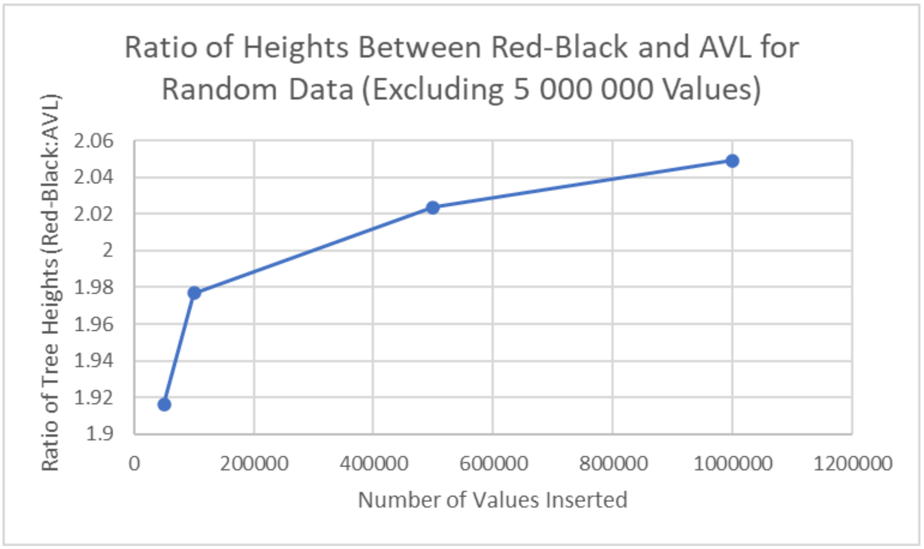

Figure 2. The ratio of the heights of red-black trees vs AVL trees for random data sets of 50 000, 100 000, 500 000, 1 000 000 integers.

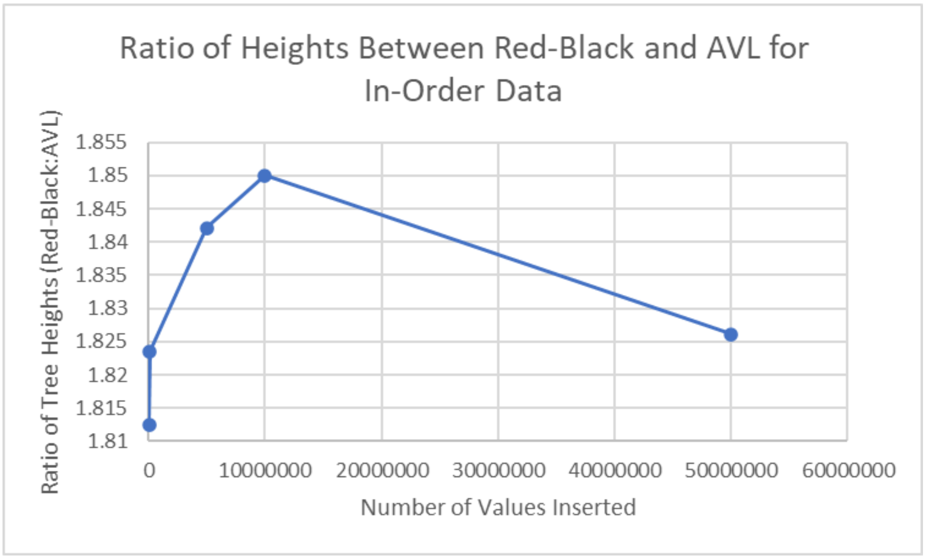

Figure 3. The ratio of the heights of red-black trees vs AVL trees for in-order data sets of 50 000, 100 000, 500 000, 1 000 000, and 5 000 000 integers.

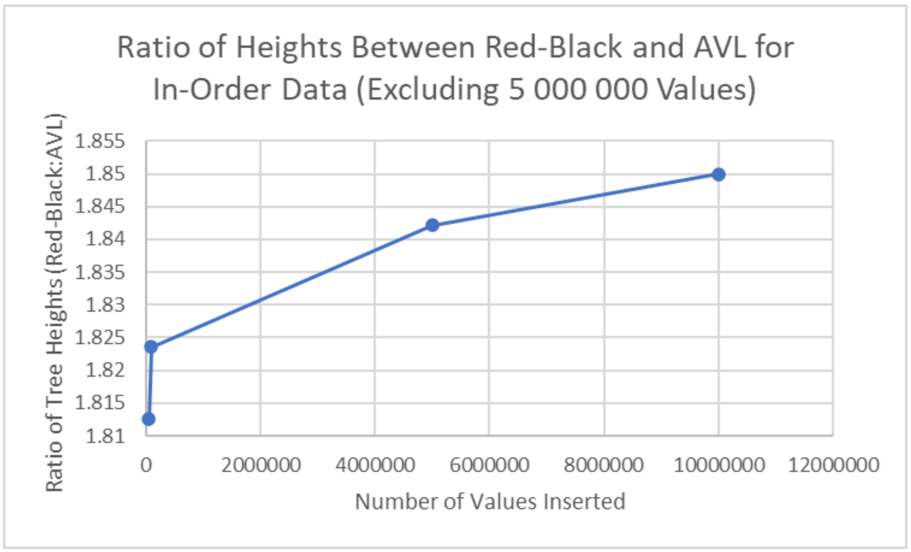

Figure 4. The ratio of the heights of red-black trees vs AVL trees for in-order data sets of 50 000, 100 000, 500 000, 1 000 000 integers.

Conclusion and Discussion

Across all tests the AVL trees had a lower average height than the red-black tree. With random data the ratio grew in a logarithmic fashion, with the growth slowing down once the ratio reached about 2.0. This was similar to the ratio of BSTs to AVL trees with random data, which is not entirely surprising since BSTs perform best with random data. This trend did not continue with in-order data, where the ratio stayed below 2.0 for all tests, where with BSTs it increased at a roughly linear pace. The data point at 5 000 000 does appear to be an outlier, since it decreased from the previous test despite no other tests indicating that behaviour. It is possible that this was caused by an error. Due to the nature of in-order insertions, the resulting tree is always the same for a given number of in-order inputs since the algorithm will always follow the same step, so if an error occurred there was not a proper average to smooth it out. It is possible there was an error in the algorithm that caused it to behave unusually at a large number of inputs, or perhaps this is the actual trend for red-black trees with in-order data. More testing with different numbers of inputs would be required to determine this.

Aside from the one possible outlier, these results were fairly expected. AVL trees are more strictly balanced in terms of height since height differences are what triggers their rebalancing, so it makes sense that they maintain a lower average height. If a lot of traversal is being done, then AVL trees are going to save more time than red-black trees. Due to red-black trees' different approach to balancing, it is possible that they have faster insertion and deletion times than AVL trees, but that was not explicitly tested and therefore research is required to give a definitive answer. AVL trees may not be better than red-black trees in all circumstances, but when it comes to keeping height to a minimum they perform better.

Works Cited

Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. 2015. Rank-Balanced Trees. ACM Trans. Algorithms 11, 4, Article 30 (June 2015), 26 pages. DOI:https://doi.org/10.1145/2689412

Conrado Martínez and Salvador Roura. 1998. Randomized binary search trees. J. ACM 45, 2 (March 1998), 288–323. DOI:https://doi.org/10.1145/274787.274812

Franceska Xhakaj and Chun W. Liew. 2015. A New Approach To Teaching Red Black Tree. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15). Association for Computing Machinery, New York, NY, USA, 278–283. DOI:https://doi-org.ezproxy.aekc.talonline.ca/10.1145/2729094.2742624