410 Project 1

Ben Brown-McMillin

September 2019

1 Abstract

There is a great utility in understanding different tree types, as each type has a contextual advantage. Knowing these advantages, a developer can make an informed decision to improve the performance of their system. For this reason, it is important to test these contexts. This paper tests the specific context of large-scale insertions, comparing the efficiency of Splay trees and Red-Black trees. The results indicate that Splay trees have superior performance in this context.

2 Introduction

Trees are significant data structures for operating systems. This is because they can store large amounts of information in predictable ways, making them useful candidates for implementing large-scale data storage systems. A wide variety of trees have also been standardized, with different rules that give each tree type a contextual advantage. Because of this, it is important to understand what those contextual advantages are. This understanding will help developers make important implementations decisions and use an appropriate tree type for a given context. To this end, this paper will be comparing the time efficiency of Splay trees and Red Black trees in a context of large-scale insertions.

3 Background

The goal of this paper is to answer the question "if we were to a series of insertions into a 'Splay tree' vs a 'Red-Black tree', which one would perform better?"[1] In order to begin answering this question, we must understand what these data structures are.

3.1 Red-Black Tree

Red-Black trees are a type of self-balancing tree. The title refers to a particular method of balancing, in which a node is colored either red or black and placed at an appropriate height. There are four rules for this method[2]:

1. Every node has a color either red or black.
2. Root of tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.

The benefit of these strict structural rules is a consistent tree height. By following these rules, the red-black tree ensures that its height will never be greater than $Logn$, with n being the number of nodes in the tree. This means that many tree functions will only take $O(Logn)$ time to complete. However, the predictable height of the tree does not benefit insertion. In fact, the additional rebalancing rules will significantly increase the time taken to insert a single node. For each insertion, a node must be compared to each node along at least one path, meaning the complexity of insertion will be $O(nLogn)$. Because of the strict balancing rules used by this data structure, it is likely to perform worse than other trees in cases of mass insertion. It will likely also have to do fewer rotations

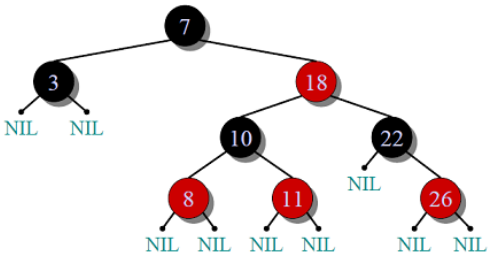

Figure 1: Example of a Red-Black Tree [2]

when inserting nodes in random order, since in-order insertions will constantly throw off its balance.

3.2 Splay Tree

Splay Trees differ from other trees in the way that they handle tree functions. Particularly, when any function is performed on a node, that node is brought to the top of the tree. This makes splay trees less predictable than other types, increasing the absolute worst case complexity to $\theta(n)$, where n is the number of nodes. However, this structure has the benefit of aggregating frequently accessed nodes at the top of the tree, so these nodes could be accessed in as little as $O(1)$ time. Since this tree does not perform regular self-balancing, it will likely perform better than the Red-Black Tree for mass insertions. However, the rotations that are done when searching for the appropriate place to insert a node may have a significantly detrimental effect on this structure's performance.

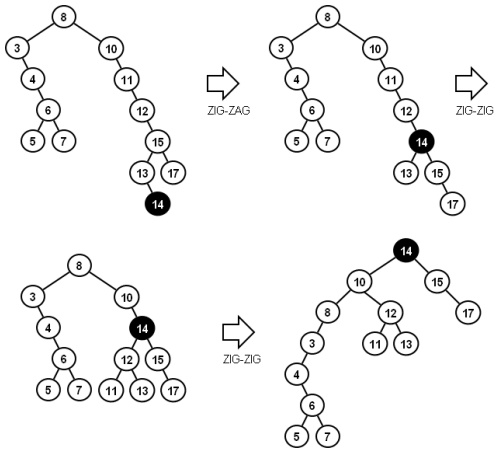

Figure 2: Example of a Splay Tree. In this example, node 14 is accessed. [3]

4 Methodology

To compare the performance of each structure, three types of data sets were inserted into empty trees. The first data set contained a series of random numbers from the range 1-1000. The second set contained ascending numbers from 1-1000. The third data set contained descending numbers from 1-1000. To clarify, the second data set starts with the numbers 1, 2, 3, etc. and continues to 1000, then restarts at 1. The third data set operates similarly, but starts with 1000, 999, 998, etc. and continues to 1, restarting at 1000. Multiple tests were done with each data set, increasing the amount of data used each time. The tests began with 10,000 numbers, then increased to 50,000, then went from 100,000 to 1,000,000 in increments of 100,000. The time taken to perform each set of insertions was calculated by timing each individual insertion. Each of these tests was run 5 times and the average was used as the result. This is to account for any fluctuations in processing speed. It should be noted that the placement of the timing code means that the incrementing of data sets and for-loop comparisons are included in the results. This flaw in the timing code is consistent through all tests, that the results are uniformly increased by the added constants of extra O(1) operations. However, since the extra operations are constant through each test, the performance relations are still significant. These data sets provide useful examples of various input possibilities. Ascending inputs test the capacity of trees to handle consistent left-hand insertions, while descending inputs test the right-hand features. These are useful for predicting the behavior of trees handling sorted data sets. The random inputs provide a general case of unpredictable insertions. These experiments were performed with an AMD Ryzen 5 2600 Six-Core Processor with 3.40GHx and 8gb of ram. The software runs on 64-bit Windows 10. The tests were performed using c in the Unix shell. The

random data set was generated using the rand() function of the standard library and the test harness used stdlib.h and time.h. The Red Black tree implementation was based on code from Victor Garritano and Arthur Woimbée[5]. The Splay tree implementation was based on Yale professor James Aspnes' code[4].

5 Results

The results of the experiments are shown in Table 1 and 2. In general, splay trees performed notably better than red-black trees. The red-black trees performed worst on descending sets and best on ascending sets. The splay trees performed worst on random sets and best on descending sets, albeit only slightly compared to ascending.

5.1 Random Insertions

The results of this set can be seen in Figure 5. Both data structures are comparable in the case of random insertions. Splay trees performed marginally better, but even with 1,000,000 insertions, the difference is 0.03 seconds.

5.2 Ascending Insertions

The results of this set can be seen in Figure 6. The performance advantage of splay trees is slightly more notable in this case. Significant spikes occur more frequently in the time taken by red-black trees. This is most likely due to the balancing features of the tree, which make sorted insertions more complex.

5.3 Descending Insertions

The results of this set can be seen in Figure 7. This data set shows the greatest disparity in the different structures' performances. The performance of splay trees is still comparable to other sets, but red-black trees perform on the scale of seconds rather than milliseconds.

Inputs	Random	Ascending	Descending
10000	0.003953	0.006315	0.047809
50000	0.023861	0.012499	0.240756
100000	0.117531	0.020336	.545504
200000	0.204567	0.03581	1.033289
300000	0.189566	0.051458	1.565989
400000	0.265512	0.093485	2.0363
500000	0.301131	0.123867	2.576106
600000	0.430215	0.119251	3.155405
700000	0.470567	0.121721	3.625463
800000	0.515704	0.129183	4.181179
900000	0.642388	0.167488	4.727425
1000000	0.63001	0.162627	5.482318

Table 1: Results (time in seconds) from Red-Black tree tests.

Inputs	Random	Ascending	Descending
10000	0.008432	0.000949	0.000888
50000	0.043511	0.00532	0.004678
100000	0.080262	0.010308	0.009136
200000	0.115767	0.022729	0.018327
300000	0.170683	0.031252	0.028026
400000	0.244845	0.041302	0.041836
500000	0.284061	0.053872	0.046718
600000	0.392282	0.063731	0.054815
700000	0.43319	0.073802	0.064346
800000	0.472917	0.112025	0.102586
900000	0.580201	0.140898	0.080983
1000000	0.597814	0.110338	0.087945

Table 2: Results (time in seconds) from Splay tree tests.

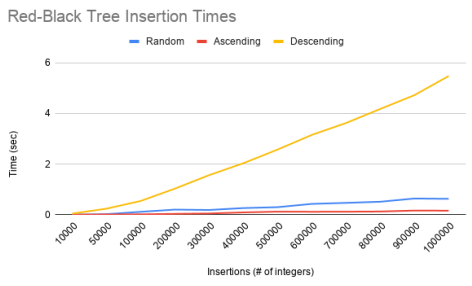

Figure 3: Times taken by Red-Black Trees for all data sets.

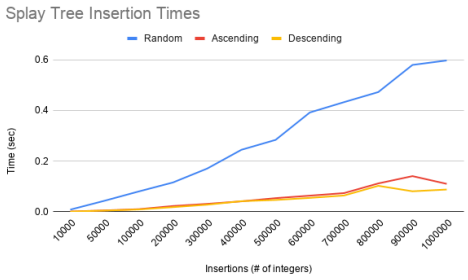

Figure 4: Times taken by Splay Trees for all data sets.

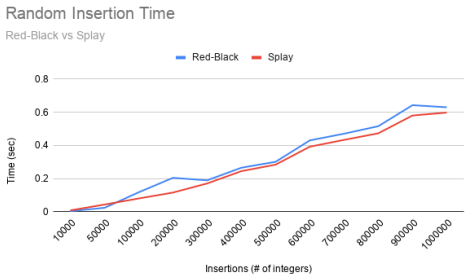

Figure 5: Times taken for random insertion sets.

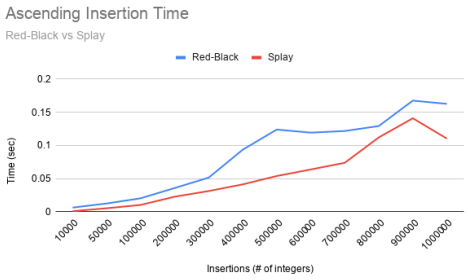

Figure 6: Times taken for ascending insertion sets.

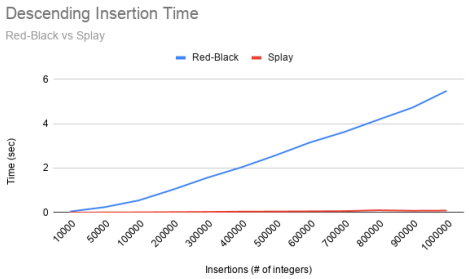

Figure 7: Times taken for descending insertion sets.

6 Conclusion

The results of this experiment show that Splay trees generally perform mass insertions faster than Red Black trees. It is possible for Red Black trees to be competitive in cases of random data sets. However, the general performance advantage of Splay trees mitigates this competitiveness. Red Black trees do have advantages in their design. The consistency of its height makes it a useful structure for applications in which many searches are required. But in applications that require frequent insertions, the balancing rules have a significantly detrimental effect on performance. Because of this, Splay trees should be considered for systems which frequently perform large amounts of insertions.

7 Future Work

The future analysis of this subject could be extended in three primary ways - additional tree types, additional tree functions and different data sets. The testing of additional tree types would show if other types of trees have particular insertion advantages. Testing other tree functions such as searching and removing would result in data which shows other kinds of contextual advantages for each tree. Finally, testing different data sets would cover real-world contexts in which these trees are used. For instance, these experiments only tested data sets ranging from 1-1000. It would be worthwhile to see if the performance benefits of splay trees are maintained as the range of data

increases.

References

[1] J. Barr. Project 1. http://classes.eastus.cloudapp.azure.com/~barr/classes/comp410/proj/proj1-19.php. Accessed: 2019-09-24.

[2] GeeksforGeeks. Red-black tree — set 1 (introduction). https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/. Accessed: 2019-09-24.

[3] GeeksforGeeks. Splay tree — set 1 (search). https://www.geeksforgeeks.org/splay-tree-set-1-insert/. Accessed: 2019-09-24.

[4] Y. U. James Aspnes. Splay tree. http://www.cs.yale.edu/homes/aspnes/classes/223/examples/trees/splay/tree.c. Accessed: 2019-09-24.

[5] A. W. Victor Garritano. Red-black tree. https://gist.github.com/VictorGarritano/5f894be162d39e9bdd5c. Accessed: 2019-09-24.