410 Project 2

Ben Brown-McMillin

December 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 DNS lookups. Splay trees, Red-Black trees, AVL trees and Tries will be tested with random, repeating and alternating data. Red-Black and AVL trees are tested for their balancing rules. Splay trees are tested for their localizing behavior. Tries are tested for their string length searches.

2 Introduction

Trees are significant data structures for operating and networking 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.

The large-scale storage capabilities and searching efficiencies make tree variations important candidates for performing DNS storage and lookups. We would like to determine which trees are most effective for this context. To this end, this paper will be comparing the time efficiency of Splay trees, Red-Black Trees, AVL Trees and Tries. The experiment will involve inserting urls into trees and then searching those urls.

3 Background

The goal of this paper is to answer the question "Which data structure is best for performing DNS lookups?"[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[4]:

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

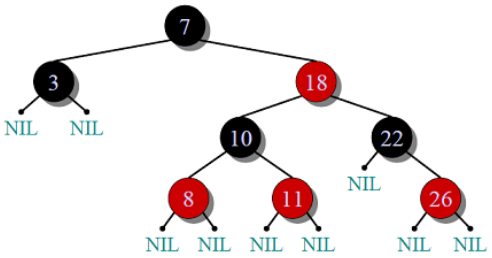

Figure 1: Example of a Red-Black Tree [4]

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. However, its rules will make it efficient to search throuh.

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 well in cases of repeated searches. 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.

3.3 AVL Tree

AVL trees are a variation of self-balancing BSTs in which each node's subtrees must have a height difference of one. The advantage of this tree structure is that it ensures a consistent height after any insertion or deletion. Because of the subtree height restrictions, every tree operation can be guaranteed to have

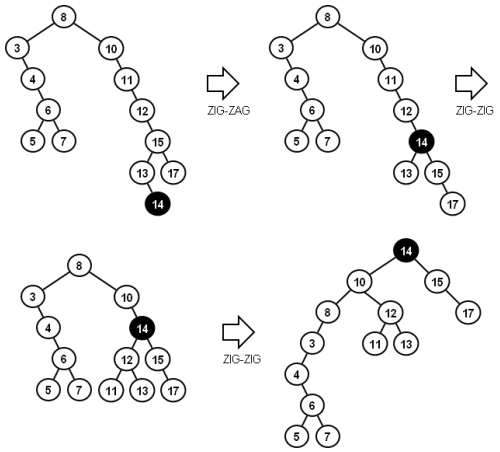

Figure 2: Example of a Splay Tree. In this example, node 14 is accessed. [5]

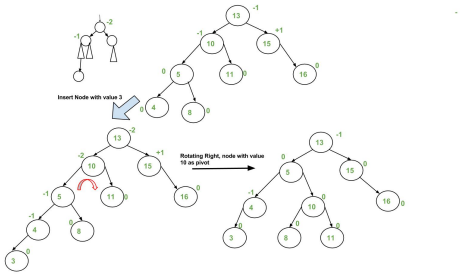

Figure 3: Example of an AVL tree. This example shows how subtrees in an AVL tree are balanced when insertions occur.[3]

a complexity of $O(Logn)$. This is because the height of the tree will always be at most $O(Logn)$

The AVL's self-balancing behaviors will make it less effective for insertions, but its guaranteed $O(Logn)$ search time will give this tree a competitive edge.

3.4 Trie

A Trie is a data structure which is specifically designed for storing strings. Its nodes are structured as arrays of characters and each node can contain the entire alphabet. Words effectively have each letter stored at a different level, and each level is traversed

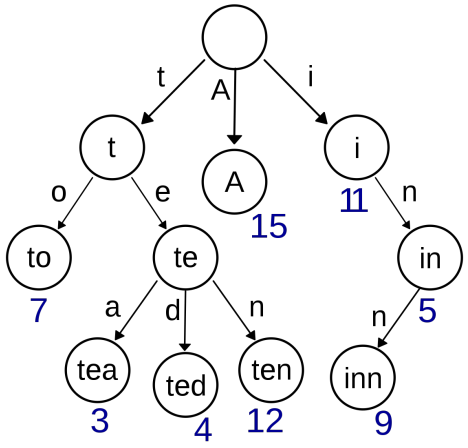

Figure 4: Example of a Trie [6]

to find the end of a word.

This structure ensures that searching for a key will have a complexity equal to the key's length. The search time and structure of the set is now based on word length, rather than than on the number of unique keys. However, this does result in increased space usage.

4 Methodology

This experiment uses the top 1000 most visited websites from Quantcast as data[8]. Each website is assigned a randonm 32-bit integer key, to simulate DNS structure. Many websites in this set show up as Hidden Profile. To ensure uniqueness between url strings, every hidden profile has been replaced with a filler URL name. The url names were generated by alphabetical sequence (a.com, b.com, c.com, ... aa.com, ab.com...).

These url-key sets were used to generate three data sets, each of 1,000,000 urls. The first data set used the urls in random order. Though the order is random, the number of occurrences of each website in the set is proportional to the website's number of hits relative to the total. This data set replicates a general internet use case. To create this data set, each url was stored in a hash map with its percentage value. If a url was selected to be added to the data set, its percentage was decreased by (100/1000000) percentage points. If the url's percentage was 0 or lower, it would not be inserted. This continued until 1,000,000 urls were added to the set.

The second set stored a random sequence of urls, but url frequency was not proportional. Additional, every instance of a url was repeated 0-5 times (the number of repeats was chosen randomly). This set imitates the behavior of landing on a website and clicking through multiple pages on that site. It was filled by randomly selecting a url, generating a random number from 0-5 and adding the url to the set that many times. This was repeated until 1,000,000 urls were added.

The third data set also contained non-proportional randomizations. This data set would use one randomly selected url, and then for some number between 0-5 times, would add that url and then another random url. For instance, if google.com was added twice, then the data set would contain (in this order): google.com. [random url 1], google.com, [random url 2]. This data set represents the behavior of going back and forth between one primary website and multiple secondary websites, such as if someone kept facebook open while browsing other websites.

The metric used to evaluate performance was the time taken to perform tree functions. 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.

To time the code, c's clock() function was used. First, a url would be read from the data set. Then, the data structure would be searched to see if it already contained that url. The search is timed. If the structure contains the url, then the url's key is accessed through the node. This access is timed. Otherwise, if the structure does not contain the url, then it and its key are added. This insertion is timed. The conditional statements and url access from the

data set are not timed - only tree functions and accesses. This means that every cycle will include the time taken to search the structure and then either the access time or insertion time.

Every tree has been modified to return a node to simplify accessing behavior. The only exception is the trie, since its nodes store alphabets rather than full strings. A separate getKey function was implemented for trie key access.

4.1 Predictions

For the random data set, we assume that the Trie will have the best performance. This is because its operations are based on word length rather than tree height. The addition of urls such as a.com may give this structre a general advantage. The Red-Black and AVL trees will be competitive, since their operations will always be on the order of $O(Logn)$. The balancing requirements of insertions will likely hold them back. The Splay trees are expected to perform the worst, since random access means that there is no guarantee of frequent repeated access. This will lead to excessive splaying.

However, the Splay trees are expected to outperform all others for the repeating and alternating data sets. Both sets have frequent repeated entries, meaning that the Splay tree's localization of nodes will be very beneficial. The alternating data set may reduce its effectiveness, though. In these cases, the Trie will likely perform well once again. Red-Black and AVL trees will be less effective due to their rigid structure ensuring $O(Logn)$ complexity for repeated searches. Additionally, the Red-Black tree's balancing rules may make its insertions more complex than AVL's, making it slightly less effective.

4.2 Equipment and Code

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. All randomizations used the rand() function of the standard library. =The test harness used stdlib.h and time.h.

The Red Black tree implementation was based on code from Victor Garritano and Arthur Woimbée[10]. The Splay tree implementation was based on Yale professor James Aspnes' code[7]. The AVL tree implementation was based on code from ZenTut[11]. The Trie implementation was based on TechieDelight's code[9]. Tries were updated to include periods and numbers in their alphabet. All data structures were modified so that their nodes stored a string url and integer key. Balancing and splaying were done using c's built-in strcmp() function in the string.h library. My code can be accessed from my github repository.[2]

5 Results

The results of the experiments are shown in Tables 5.3, 5.3 and 5.3. Overall, Splay and AVL trees were competitive for the best performance. Tries lagged behind, while Red-Black trees were not even competitive. All structures performed best on the repeating data set. Splay trees, AVL trees and Tries had competitive random and alternating data set performance. Red-black tries had competitive random and repeating data set performance but worse alternating data set performance.

5.1 Random Data Set

The results of this set can be seen in Figure 5. Splay trees showed the fastest performance until 700,000 inputs, when AVL trees consistently overtook them. Both trees had competitve performance. The difference in time between the two was often less than 0.01 seconds. In the case of 500,000 inputs, Red-Black trees had competitive time, but this behavior is not repeated. This is most likely a slight computing anomaly. Aside from this instance, Red-Black trees had the worst performance throughout this test. Tries were not competitive with Splay or AVL trees.

5.2 Repeating Data Set

The results of this set can be seen in Figure 6. Splay trees were consistently the fastest, except in the case

Inputs	Splay	Red-Black	AVL	Trie
10000	**0.012166**	0.016062	0.013068	0.016667
50000	**0.061402**	0.077359	0.063048	0.07304
100000	**0.124234**	0.154959	0.124692	0.144855
200000	**0.246472**	0.308812	0.249588	0.285434
300000	**0.368992**	0.462925	0.374295	0.435928
400000	**0.491552**	0.613873	0.497954	0.568258
500000	**0.61508**	0.61866	0.61866	0.699824
600000	**0.733091**	0.907043	0.737302	0.825188
700000	0.855669	1.04557	**0.855437**	0.947871
800000	0.968573	1.172631	**0.963774**	1.065801
900000	1.087027	1.300289	**1.081978**	1.179665
1000000	1.200346	1.432204	**1.198703**	1.300488

Table 1: Results (time in seconds) from random data tests.

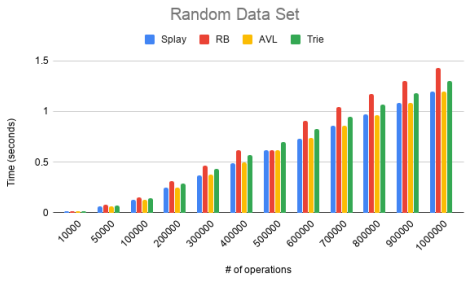

Figure 5: Times taken for random sets.

of 900,000 inputs, when AVL trees briefly overtook them. Still, the performance of those two structures were quite competitive, showing an average difference of about 0.03 seconds. Tries and Red-Black trees were not competitive.

5.3 Alternating Data Set

The results of this set can be seen in Figure 6. AVL trees had better performance than Splay trees throughout this test, aside from the first run. However, the differences in performance tended to be about 0.01 seconds. Once again, Tries and Red-Black trees were not competitive, consistently performing worse.

Inputs	Splay	Red-Black	AVL	Trie
10000	**0.011934**	0.014864	0.012367	0.01546
50000	**0.055672**	0.072543	0.059122	0.063836
100000	**0.110447**	0.143981	0.116293	0.123138
200000	**0.224422**	0.292636	0.234963	0.254312
300000	**0.338064**	0.434782	0.349983	0.37597
400000	**0.450946**	0.579965	0.467986	0.500578
500000	**0.56302**	0.725766	0.585017	0.624814
600000	**0.674031**	0.87461	0.70223	0.760953
700000	**0.784758**	1.013486	0.818936	0.875862
800000	**0.898299**	1.163013	0.937722	1.004877
900000	1.017375	1.315025	**1.059036**	1.13567
1000000	**1.115643**	1.446262	1.163196	1.242149

Table 2: Results (time in seconds) from repeating data tests.

Inputs	Splay	Red-Black	AVL	Trie
10000	**0.01243**	0.016276	0.012562	0.015668
50000	0.060709	0.079268	**0.060226**	0.066518
100000	0.121966	0.160931	**0.119674**	0.130311
200000	0.243173	0.322009	**0.240628**	0.26779
300000	0.364821	0.480177	**0.359006**	0.395946
400000	0.487041	0.644057	**0.480266**	0.527787
500000	0.610943	0.806934	**0.59987**	0.663339
600000	0.727373	0.962782	**0.715475**	0.788207
700000	0.851971	1.130319	**0.838039**	0.925272
800000	0.9717	1.287935	**0.95579**	1.051625
900000	1.101276	1.466143	**1.086231**	1.19634
1000000	1.219548	1.61989	**1.198228**	1.322875

Table 3: Results (time in seconds) from alternating data tests.

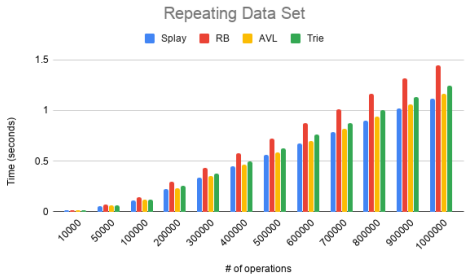

Figure 6: Times taken for repeated sets.

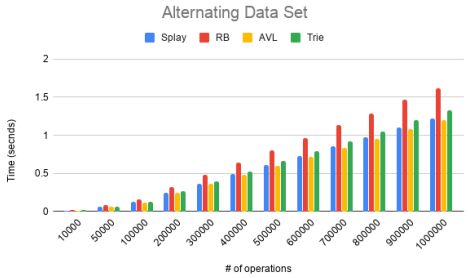

Figure 7: Times taken for alternating sets.

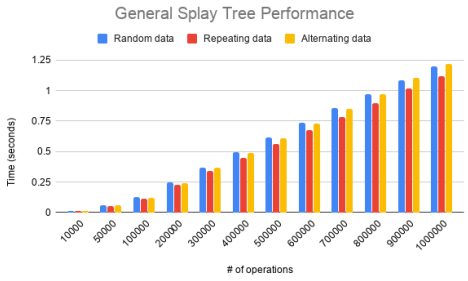

Figure 8: Times taken by Splay Trees for all data sets.

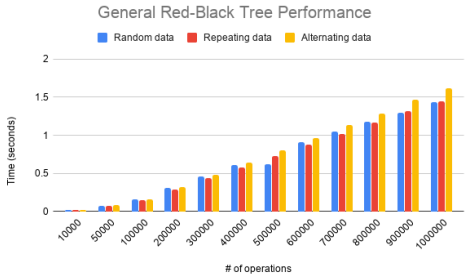

Figure 9: Times taken by Red-Black Trees for all data sets.

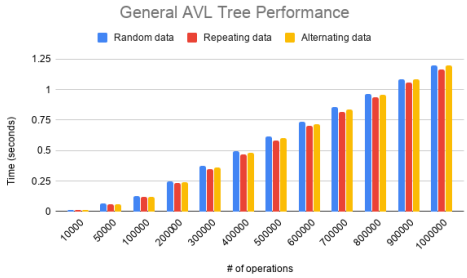

Figure 10: Times taken by AVL Trees for all data sets.

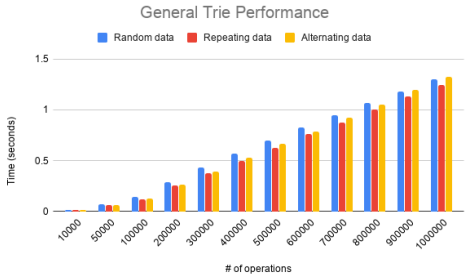

Figure 11: Times taken by Tries for all data sets.

6 Conclusion

Splay trees and AVL trees were competitive across the board. This both confirms and rebukes the hypothesis, in that Splay trees were predicted to do well, but AVL trees were not. The balancing rules of AVL trees were expected to impact performance, but each test had at most 1000 insertions so the impact of AVL balancing did not have an extreme effect. The insertion balancing did severely affect the Red-Black trees, which were always the slowest. Tries performed significantly worse than expected. The height advantage of Trie searches was thought to improve performance. But even with very short url lengths, the $O(Logn)$ complexity of AVL searches and splaying behavior of Splay trees proved to be a greater advantage.

For the random data set, splay tree performance was generally better than expected. Since the set was random, we could not predict repeating accesses of the same url. Despite this, the splay trees succeeded in the tests of smaller sets. As the data size increased, though, the AVL trees consistently won out. The data indicates that the 700,000 input mark is where the AVL tree's consistent $O(Logn)$ operation complexity surpassed the Splay tree's worst case $O(h)$ operation (where h is the height of the tree). It is still surprising that the Trie's $O(k)$ (k = key length) operation did not show a significant benefit; it is reasonable to expect that k would generally be less than h. It is possible that the time taken to search through the Trie's character array nodes took too much time, especially with an alphabet of up to 37 characters.

Splay trees had a consistent edge during the repeating data set. This was expected because Splay trees are optimized for repeated access. The repetition ensured a regular $O(1)$ operation for this structure. Additionally, this data set showed the greatest difference between Splay and AVL performance, even though AVL trees performed better during one run.

AVL trees consistently outperformed the others for the alternating data set. Aside from the first run, where Splay trees prevailed, AVL trees were superior. This is significant because the alternating data set still has some repeating behavior, alternating between repeating and random urls. The splay tree does utilize an $O(1)$ operation for those repeated urls, but the $O(h)$ operation for the random ones holds it back. In this case, the consistency of the AVL tree's operations balances out the repeated and randomized behaviors.

In all cases, Splay trees and AVL trees were fairly competitive. But as data set size increased, trends began to show and the differences in performance increased. AVL trees showed balanced performance on both random and repeating inputs, especially during later tests.

7 Future Work

This experiment has indicated that Splay trees and AVL trees are the most efficient data structures for DNS lookups. Future work should focus on more use cases for those structures. This could mean increasing the data size to see further disparities in performance or creating more test cases to replicate user behavior. Real-world data would also provide useful results.

References

[1] J. Barr. Project 2. `http://http://classes.eastus.cloudapp.azure.com/~barr/classes/comp410/proj/proj2-19.php`. Accessed: 2019-09-24.

[2] B. Brown-McMillin. 410 project 2. `https://github.com/benbjamin123/410-Project-2/upload`. Accessed: 2019-09-24.

[3] GeeksforGeeks. Avl tree — set 1 (insertion). `https://www.geeksforgeeks.org/avl-tree-set-1-insertion/`. Accessed: 2019-09-24.

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

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

[6] GeeksforGeeks. Trie — (insert and search). https://www.geeksforgeeks.org/trie-insert-and-search/. Accessed: 2019-09-24.

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

[8] Quantcast. Top websites. https://www.quantcast.com/top-sites/. Accessed: 2019-09-24.

[9] TechieDelight. Trie implementation in c — insertion, searching and deletion. https://www.techiedelight.com/trie-implementation-insert-search-delete/. Accessed: 2019-09-24.

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

[11] ZenTut. C avl tree. https://www.zentut.com/c-tutorial/c-avl-tree/. Accessed: 2019-09-24.