Midterm
Alex Yazdani
Eric Reed
CS F003C
04 June 2023

Introduction

In this project, a class Webstore() is created to crawl the web recursively and store text information in various data structures. Any data structure can be used, as long as it contains working find() and insert() methods. The Webstore() class will grab all keywords (alphanumeric and > four characters) from a webpage, and create KeywordEntry() objects with this data. Each object of this type contains a string of the word keyword itself, and a dictionary containing key/value pairs of the URL and a list of locations the word was found on that URL. The Webstore() class has capability to do the same for all links found on the initial page, links found on those pages, etc., depending on the depth input. These KeywordEntry() objects are stored in the specified data structure. If the object already exists, the new URL/location is added to the object. Otherwise, a new object is instantiated and stored in the data structure.

In this paper, four different data structures are used to validate operation of the Webstore() class as well as compare the performance of each data structure under different conditions. The four data structures tested are:

Binary Search Tree
AVL Tree
Splay Tree
Hash Table (Quadratic Probing)

The test program crawls the same link and stores the data at different levels of depth for each data structure, recording the total crawl/store times. Then, different dataset sizes are searched for in each structure, recording the average search time.

This experiment was run in approximately 1 hour with all other applications on the system closed, using an Intel i7 core.

Analysis

During the testing, the keywords found are stored in a separate list. This list is used to search the data structure for each word, and the results show that every data structure successfully returns each word, meaning they were stored properly.

In addition, a list of random words are generated, and each data structure is searched for each term. With depth=0, almost no words are found, but as it increases, more and more are found, even with the smallest sized dataset. Table 1 below shows that, disregarding time, each data structure is equally capable of storing and returning data. The variations between charts are due to the randomness of each list.

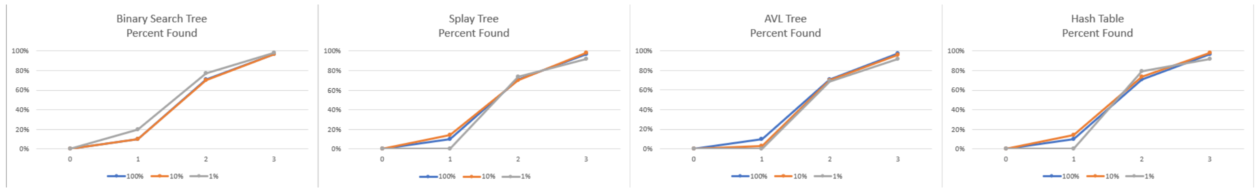


Figure 1: Percent Found for Each Data Structure

Before addressing the search times, the crawl/store timing for each test is shown below in Table 1:

Depth:	0	1	2	3
BST	0.07	2.19	23.22	184.05
Splay Tree	0.06	2.18	26.73	204.79
AVL Tree	0.17	0.76	21.20	193.13
Hash Table	0.10	0.79	20.16	163.24

Table 1: Crawl/Store times for each Data Structure

There is a drastic difference between depths, especially noticeable at depth = 3. The recursive nature implies an exponential time complexity, though not enough data was gathered to adequately show this empirically. To extrapolate, an exponential function could be modeled to the data points, but this was not done here. Specifically looking at depth = 3, the hash table is the fastest. This is because not enough entries were inserted for the rehashing to slow down the process significantly enough. The overall shape and behavior of each data structure during the crawling/store phase are similar though vary in exact timing. The splay tree, for example, is the slowest. This is because for this data structure, like searching, inserting also requires a splay function. As expected, the AVL tree takes slightly longer than the BST because of the balancing done during insertion. The timings may be inconsistent due to network stability/ strength and server response time. This was remedied by running 2 crawl trials and placing the machine closer to a network wireless access point.

The figure and table below show the data for each test run, displaying search times for different depths and sample sizes for each data structure:

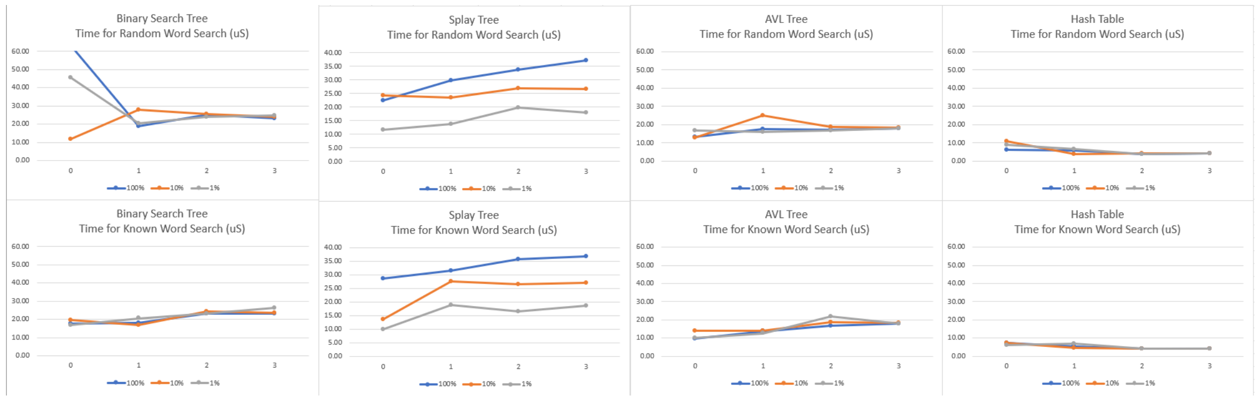
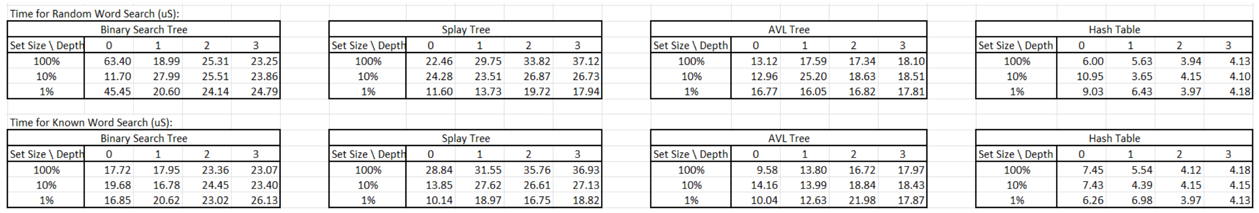


Figure 2: Times for Random/Known Word Search for each Data Structure



Time for Random Word Search (uS):

Binary Search Tree				
Set Size \ Depth	0	1	2	3
100%	63.40	18.99	25.31	23.25
10%	11.70	27.99	25.51	23.86
1%	45.45	20.60	24.14	24.79

Splay Tree				
Set Size \ Depth	0	1	2	3
100%	22.46	29.75	33.82	37.12
10%	24.28	23.51	26.87	26.73
1%	11.60	13.73	19.72	17.94

AVL Tree				
Set Size \ Depth	0	1	2	3
100%	13.12	17.59	17.34	18.10
10%	12.96	25.20	18.63	18.51
1%	16.77	16.05	16.82	17.81

Hash Table				
Set Size \ Depth	0	1	2	3
100%	6.00	5.63	3.94	4.13
10%	10.95	3.65	4.15	4.10
1%	9.03	6.43	3.97	4.18

Time for Known Word Search (uS):

Binary Search Tree				
Set Size \ Depth	0	1	2	3
100%	17.72	17.95	23.36	23.07
10%	19.68	16.78	24.45	23.40
1%	16.85	20.62	23.02	26.13

Splay Tree				
Set Size \ Depth	0	1	2	3
100%	28.84	31.55	35.76	36.93
10%	13.85	27.62	26.61	27.13
1%	10.14	18.97	16.75	18.82

AVL Tree				
Set Size \ Depth	0	1	2	3
100%	9.58	13.80	16.72	17.97
10%	14.16	13.99	18.84	18.43
1%	10.04	12.63	21.98	17.87

Hash Table				
Set Size \ Depth	0	1	2	3
100%	7.45	5.54	4.12	4.18
10%	7.43	4.39	4.15	4.15
1%	6.26	6.98	3.97	4.13

Table 2: Times for Random/Known Word Search for each Data Structure

First, looking at the Random word searches, the splay tree appears to have the worst performance. There is a clear increase with depth, and a clear increase with set fractions. This is because splay trees reorganize themselves with each search to keep the most frequently searched for data near the top. This is not optimal for this test program, which searches with random data, unlikely to have frequent repeated items. These increases in set size and set fractions will result in more useless reorganization, creating slower search times. In a situation where a specific portion of the entries are used the majority of the time, this would be the preferred solution.

The binary search tree has seemingly constant time complexity, though it is likely that, if run more times, a linear increase would be seen (and can be seen in the known search). This is because as the set size increases, the height of the tree increases, so the average search height does as well. However, changing the set fraction of the search does not affect the height of the tree, so the times between set fractions are miniscule. These observations are also true for the AVL tree data. The only difference is that the AVL Tree will rebalance itself to meet the AVL condition when violated, which is why it has a higher crawl/store time than a BST. Regarding search times, however, they are significantly lower than the BST. The reorganizing stops the tree from becoming imbalanced, resulting in a lower height. In addition, the rebalancing guarantees a time complexity of O(logN), while the BST can only guarantee O(N), though this cannot be seen empirically here. The AVL is advantageous for search times compared to BST, but not advantageous in terms of insertion times. It is best to use an AVL tree if the insertion is done at the beginning, like here, then searching is done after without further insertion. A noticeable observation of the BST data are the values at depth = 0. This is likely due to startup delays with the test program, since the BST at depth = 0 is the first test to be run.

The hash table has the lowest search times at all depths and a time complexity of O(1), which is very desirable. However, it must be noted these times assume no further insertion after the initial crawl/store, and therefore rehashing is not accounted for. If data never needs to be inserted after the initial storing, the hash table is the best option.

Comparing the known word searches to the random word searches, the data mostly seems to be the same aside from some anomalies. This is because on average, the time to search for a nonexistent entry will be the same as the time to find an existing one for all four data structures.