Muaz Mohammed
Sabrina Peng

CSE 2341
Professor Fontenot
May 5th, 2017

Analyzing the Relative Performance of AVL Trees and Hash Tables

I. Introduction and Overview

The world of computer science provides us as students and future researchers with many ways to compare performance of and evaluate the speed and efficiency of a set of operations. As part of CSE 2341 at Southern Methodist University, we are required to implement various forms of container classes - including but not limited to arrays, vectors, and linked lists. This report is a comparison of two container classes, Hash Table and AVL Tree.

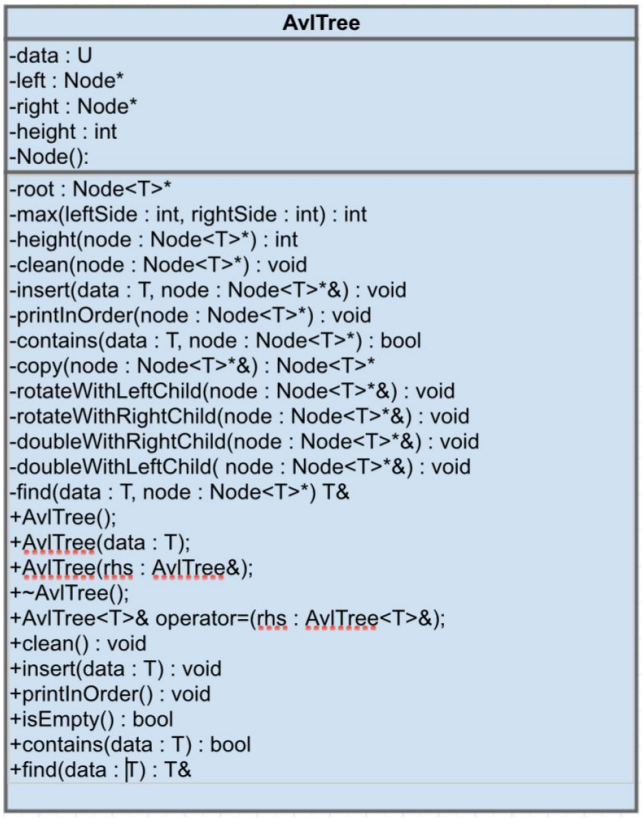
Before even beginning data analysis on the comparative speeds of the two container classes, we must define what a container class actually is. Simply put, a container class is designed to hold and organize different types of objects or primitive data types. Container classes in general implement functions that insert, remove, print, and retrieve their elements. If a container class dynamically allocates memory, the class would also need to implement an overloaded assignment operator, copy constructor and destructor as part of the Rule of Three implementation in C++.

Container classes' performance is generally determined by three main factors - access, storage, and traversal. Each type of container object has certain advantages and disadvantages when it comes to achieving these three objectives. By looking at these characteristics and determining the overall time complexity (or big O) of the operations involved under various situations, one can decide on which container is the best to implement on a given project.

When looking at access properties of a container, it is important to consider ways to access the elements of the container and how efficiently a certain element may be reached. For example a stack is accessed by LIFO (last-in first-out) but queue is accessed by FIFO (first-in first-out). For storage, we consider how the container is saving its contents and how efficiently memory is being used - looking especially for wasted memory and memory leaks caused by improper allocation. The third one, traversal, looks at the speed and efficiency of iterating through the elements held by a container.

For the purpose of this report and project, we will be using two container classes - Hash Table and AVL Tree. Both container classes are sufficient containers for traversal and access purposes. In other words, these containers are adept at searching for the elements they contain - we use the insert, find, and certain traversal functions especially liberally in the context of creating a PDF search engine.

Figure 1. AvlTree implementation UML diagram



AVL Tree

AvlTree is implemented by Mark Allen in his book Data Structures and Algorithm Analysis. His implementation in addition with what we discussed and formulated in our data structure course produced a simple way to understand AvlTree.The underlying idea behind an AvlTree is a binary search tree (BST) container. A binary search tree is a binary tree where each node has a Comparable key and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.

What makes an AvlTree different from an ordinary binary search tree is that the underlying tree is self-balancing. The AvlTree depends on the height of a given node to determine an imbalance between the left subtree and the right subtree. An imbalance in the case of our avlTree is when the height of the difference of the right or left subtree minus the left or subtree reaches 2 respectively.

The most important function in the AvlTree is the insert function. Insert needs to calculate where an inserted object should go and then check the height difference. If the height difference reaches the value 2, then the insert function calls balance functions to rotate and adjust the tree structure.

Since avlTree implements a binary search tree and the objects inserted are distributed in an organized manner, searching for an object in an AvlTree is fast and efficient. The question here is how fast is fast and how efficiency can be defined. Comparing the AvlTree to another container class designed for rapid retrieval (such as HashTable) can give us a better understanding of which container would be better to use in a certain situation (such as the size of a corpus in the context of our search engine project).

Hash Table

HashTable is a container that implements an associative array of elements of a certain data type. It is one of the many containers that is known for storing large amounts of data in an efficient manner. Our HashTable uses an array of vectors to manage data. The key to the hash table's success and its speed is the hash function. The hash function accepts a <key, value> pair and uses the key as a hash to find a location to put the value. Since a hash is being used there is no need to iterate through the whole container - we are able to go directly to the

desired location based on the key. In our search engine C++ code, we implemented the HashTable using the STL hash function. Not only is the hash of a key is used to insert a certain element, it is also used to find a certain element in the hashTable. This again is a real-time operation since we can go directly to the location of the value using the hash of the key. To avoid out of bounds errors, the hash of a key is divided by the size of the vector array and the remainder (or modulus) is used as an index location.

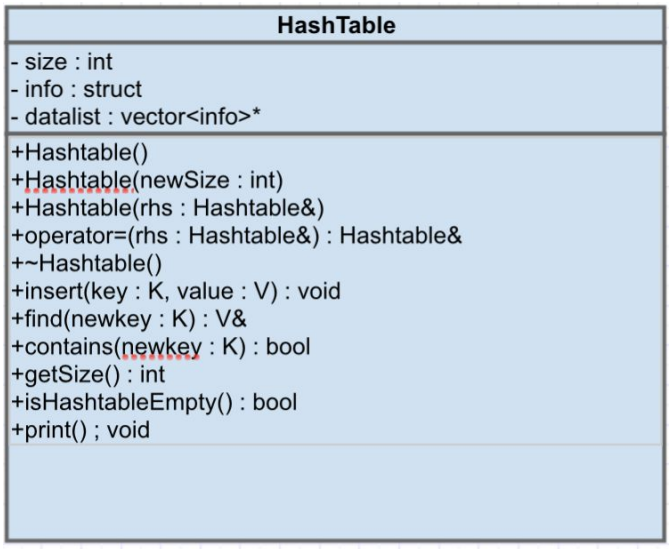


Figure 2. HashTable implementation UML diagram

One common issue with a HashTable is the frequency of collisions. A collision occurs when the vector array index starts filling up and there comes a point where the same index needs to be used for more than one object - a potentially disastrous problem if not dealt with. In our implementation, we intentionally use a vector so that collisions can be added to the end of the vector and the size of the vector array is one million so as to limit possible collusions.

It is worth noting that some HashTables utilize the one-time expensive operation of rehashing the whole underlying container class. Rehashing include the process of creating a bigger underlying container class and moving all of the data from the older smaller structure to the newer bigger structure. While the operation cost is significant in terms of time and comparisons, it is worth it in the long run due to the idea of amortized complexity analysis - the large cost of one operation being spread over a series of small operations.

Comparative Analysis

Now we have discussed the two container classes, the next step is to go back to the question of which one is efficient and faster. After our search engine project was completed, we were able to use statistical analysis - specifically with the usage of two sample T testing - to determine whether or not there was a significant difference in the means of two samples of searching times. Our search engine features the ability to take in a specifically structured Boolean query. For the purposes of this analysis, we decided to make the searching query a simple one-word term - “test” - so that the query would be uniform across all situations.

The definition of performance that we used in our analysis was the speed (or time taken by the operating system) to execute the simpleQuery() function in our Searcher class. As said before, the simpleQuery() function allows the user to input one word and prints out the the names of PDF documents that contain the word, drawn from a corpus of PDFs that was earlier parsed by another class in the overall project hierarchy. We utilized the std::chron standard library functions to set up a checkpoint before the line of code that begins the search in the index and another checkpoint after the line of code returns a vector of PDFs and their relevant term frequency inverse document frequency values. By subtracting the first value

from the second, we were able to get a positive time value in milliseconds for the amount of time it took the operation to search for and return the PDF documents and TFIDF values associated with a given search term. We observed and documented this type of time data (shown in Figure 3) for six different situations, testing both AvlTree and HashTable container indices for three sizes of corpi - small (two documents), medium (39 documents), and large (483 documents).

One note that needs to be made is that the time improvement of the searching operation becomes less as the corpus grows larger in size, specifically because there are only so many possible words in the English language, and those that are repeated in the PDF documents are not included in the index. In addition, Figure 4 demonstrates the taking out of outlier values to get a more consistent and accurate mean and standard deviation for our two sample T test calculations later on in the analysis part of our paper.

Figure 3. Observed data for AvlTree and HashTable index implementations across three sizes of corpi

Corpus Type: One-Word Query Searching and Outputting Results	Small Corpus - 2 Documents		Medium Corpus - 39 Documents		Large Corpus - 483 Documents	
Outlier by 2 SD from mean	AVL Tree	Hash Table	AVL Tree	Hash Table	AVL Tree	Hash Table
	0.04835	0.014905	0.207559	0.091613	0.588903	0.360241
	0.050855	0.019771	0.204927	0.139761	0.411856	0.237214
	0.080257	0.011864	0.149864	0.092662	0.385915	0.468723
	0.07275	0.019922	0.256434	0.085529	0.401491	0.297957
	0.068064	0.014343	0.14388	0.110747	0.32915	0.493645
	0.057054	0.019645	0.14835	0.117153	0.533073	0.315762
	0.135512	0.01495	0.200021	0.156499	0.528877	0.307253
	0.051726	0.019801	0.137497	0.096053	0.582552	0.301845
	0.164374	0.016912	0.145849	0.094797	0.482605	0.243805
	0.165366	0.020422	0.194804	0.093338	0.397808	0.532561
	0.05087	0.014419	0.200632	0.179597	0.330808	1.83702
	0.050612	0.019311	0.245933	0.089308	0.487796	0.478309
	0.050703	0.019841	0.14182	0.093807	0.402334	0.466763
	0.050363	0.01487	0.143469	0.086385	0.399476	0.607195
	0.051014	0.019637	0.203669	0.090491	0.368424	0.246697
	0.051364	0.015461	0.223164	0.090797	0.438021	0.359127
	0.054247	0.014473	0.149508	0.091067	0.465115	0.300031
	0.050613	0.018603	0.137466	0.85985	0.461411	0.295841
	0.05128	0.019687	0.141434	0.09061	0.395607	0.30553
	0.052066	0.015242	0.163382	0.090085	0.501994	0.313622

Figure 4. Adjusted mean and standard deviation values for outliers

Average	0.070372	0.01720395	0.1769831	0.14200745	0.4446608	0.43845705
Standard Deviation	0.037869008	0.002686931	0.038620381	0.170879775	0.075742712	0.345699506
Range: 2 SD	0.032502992	0.014517019	0.138362719	-0.02887232	0.368918088	0.092757544
	0.108241008	0.019890881	0.215603481	0.312887225	0.520403512	0.784156556
New Average	0.055422824	0.017605	0.162930084	0.104226263	0.433186846	0.364848474
New Standard Deviation	0.009167178	0.002214232	0.028193997	0.026224649	0.041276106	0.108456409
New Range: 2 SD	0.046255645	0.015390768	0.134736087	0.078001614	0.39191074	0.256392065
	0.064590002	0.019819232	0.191124081	0.130450912	0.474462952	0.473304883

Below, we have also created charts that demonstrate the performance of each index in searching for a requested term over twenty trials. The regression values seem to indicate that there is no correlation between the number of the trial and the performance in terms of time spent of the operation in question.

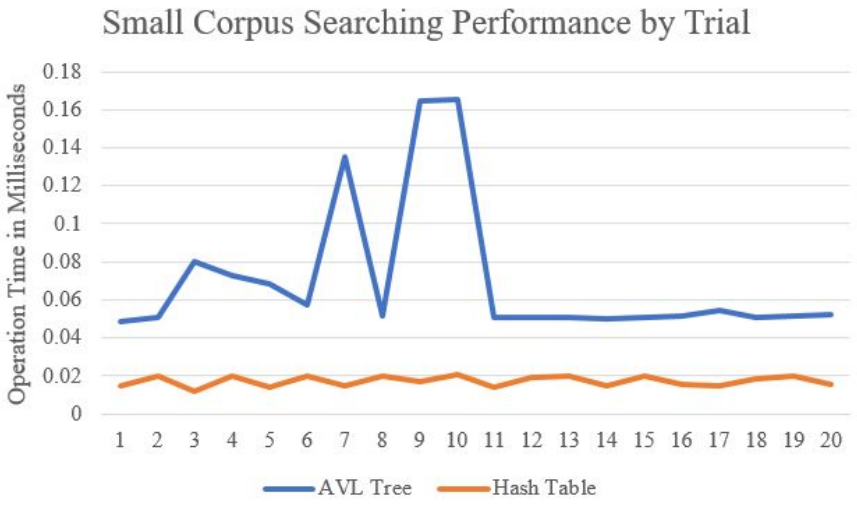


Figure 5. Small corpus performance on a one-term query

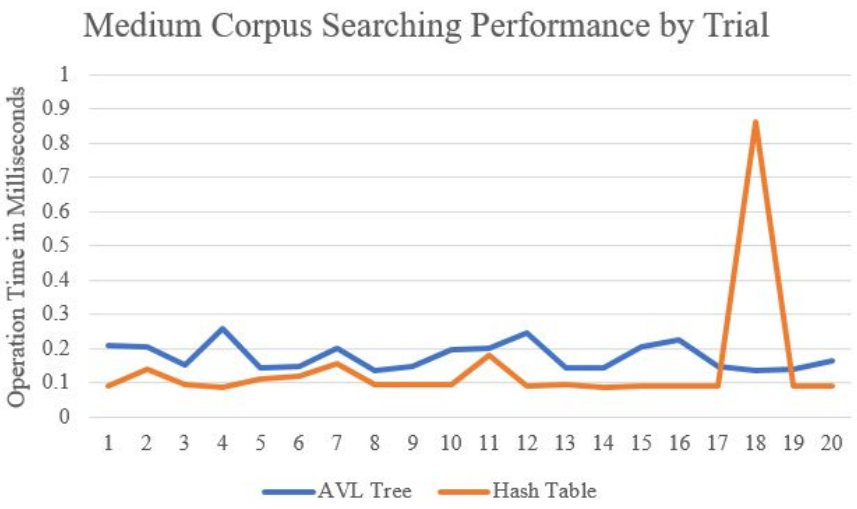


Figure 6. Medium corpus performance on a one-term query

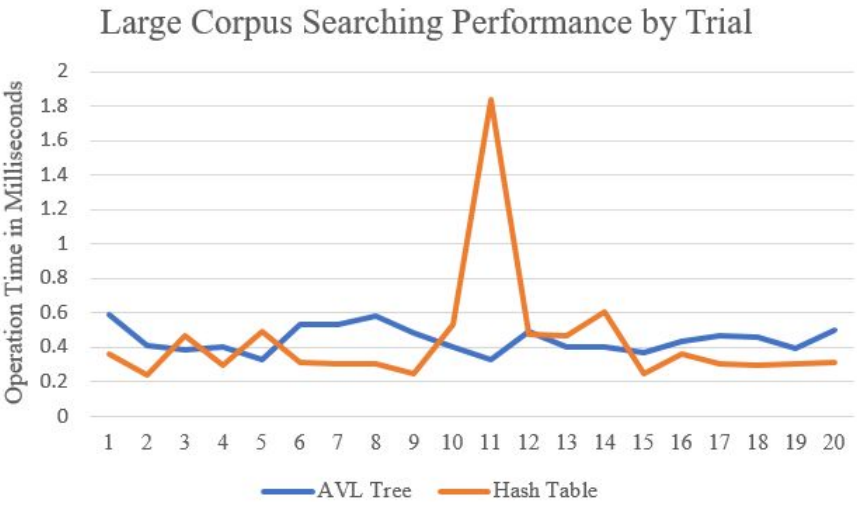


Figure 7. Large corpus performance on a one-term query

Final Analysis

In terms of analysis of the data that we collected, we decided to use a two sample T test in order to observe whether there is a significant difference in the means of the two samples (AvlTree and HashTable) for each size of corpus. The null hypothesis for each of the two sample T tests is that there is no significant difference between the two means: $\mu_1 - \mu_2 = 0$. The alternative hypothesis would be a complete negation of the null hypothesis: $\mu_1 - \mu_2 \neq 0$. Originally, we were set on performing One-Way Analysis of Variance (ANOVA) hypothesis testing, which is often used to compare the equality of three or more means using a determination of whether the differences in the samples are due to random error or actual differentiation in the data. However, since using ANOVA to compare the means from two samples is equivalent to using a T test, we decided to calculate a two sample T test to shed insight on the observations we made about performance and operation times between the two container class implementations.

We were able to perform the hypothesis testing using the average and standard deviation values calculated in Excel with a TI-84 calculator. The two sample T test assumes that the samples are independent and that each sample is randomly taken from a population that is approximately normally distributed. The standard deviations of the AvlTree and HashTable time operation observations for the small and large corpus were not close enough in value to use a pooled standard error or standard deviation (one must be less than twice the other), so the unpooled standard error method for differing standard deviations was used in the case of those two corpi. Subscripts of 1 indicate AvlTree usage for the index, while subscripts of 2 indicate that a HashTable was used to generate the time data.

<u>Small Corpus - 2 Documents</u>
AvlTree Data
Sample Mean = 0.0554
Sample Standard Deviation = 0.00917
Number of Observations = 17

HashTable Data
Sample Mean = 0.01761
Sample Standard Deviation = 0.002214
Number of Observations = 13

t = 4.1476
p = 7.3823 x 10^-4
df = 16.2104

<u>Medium Corpus - 39 Documents</u>
AvlTree Data
Sample Mean = 0.1629
Sample Standard Deviation = 0.0282
Number of Observations = 15

HashTable Data

Sample Mean = 0.10423
Sample Standard Deviation = 0.02622
Number of Observations = 19

t = 6.2707
p = 4.9684 x 10^-7
df = 32

Large Corpus - 483 Documents
AvlTree Data
Sample Mean = 0.4332
Sample Standard Deviation = 0.0413
Number of Observations = 13

HashTable Data
Sample Mean = 0.3648
Sample Standard Deviation = 0.1085
Number of Observations = 19

t = 2.4951
p = 0.01963
df = 24.763

By studying the values that we have calculated through conducting the two sample T tests, we can conclude that in using a standard alpha value of 0.05, or 5%, we can conclude that since the p-values generated by all three calculations is less than alpha, we reject the null hypothesis - proving that the population means are not the same. That does not guarantee any other relationship between the population means except for the fact that the population means are not equal to each other under this form of the two sample T test.

Now, we would like to conduct one final two sample T test with the same null hypothesis, but a different alternative hypothesis of $\mu_1 > \mu_2 \neq 0$ where μ_1 is the population mean of the AvlTree time observations and μ_2 is the population mean of the HashTable time observations. Using this specific test will allow us to conclude whether or not AvlTrees are less efficient in terms of time for searching and returning of data from its container of elements.

Small Corpus - 2 Documents

t = 16.4326
p = 1.1046 x 10^-12
df = 18.243

Medium Corpus - 39 Documents

t = 6.2707

p = 2.4842 x 10^-7
df = 32

Large Corpus - 483 Documents

t = 2.4951
p = 0.009814
df = 24.763

Again, we can see that the p-value (the probability of observing a sample statistic as extreme as the test statistic) is less than the alpha significance level of 0.05, leading to corroboration and extension of our conclusion from the first two-sample T test: the mean time it takes for the AvlTree to enact searching and retrieval of data is more than the mean time it takes for the HashTable to do an equivalent series of actions.

A couple of additional observations: the p-value increases as the data set size increases. As we mentioned earlier in the paper, this nearing of the p-value to the significance level, although not close to surpassing it, signifies that the differences between the two container classes in terms of searching performance time is negatively correlated with the size of the corpi. This is possibly due to the fact that a larger corpus of documents increases the chance of collisions in the HashTable implementation as the number of words entered into the inverted index increases. In addition, while the AvlTree is able to branch out from itself almost indefinitely (or so it seems), the HashTable is dependent upon an array or vector structural backbone, creating a need for either one expensive operation to increase its size as collisions become too much for the container to handle, or many small but cost-consuming operations over the course of the use of the HashTable.

Conclusion

All in all, we can conclude that our data supports the general assertion that AvlTrees take more time in searching and retrieving information as it relates to our search engine project than HashTables do. Our experiments in terms of data collection were flawed in that we were not sure if the distributions of the data were normal and in that we could have collected larger samples to be even more sure of the consistency of the data across maybe fifty or even a hundred trials - even though extraction of the outliers proved to be a smart decision for accuracy purposes. In addition, experiments could have been done on the comparison of the AvlTree and HashTable implementations in storing the contents of a parsed PDF file as well as reading from an inverted index file data to be stored in for reading by the query processor and searcher classes. Further real life application could include the use of these data structures in implementing database-like functionality in data marts and silos (used by organizations with considerable amounts of consumer data) or business and technological strategies such as SEO marketing and pull-through advertising. Even though our experiment was done on an admittedly miniscule scale, the implications of this type of statistic research in the field of computer science will only grow larger as we advance further into the 21st century.