An Experimental Analysis for Comparison of Searching Algorithms

Okan YILDIRIM (150160537)

Fatih YILMAZ (150160531)

Özgür AKTAŞ (150160530)

ABSTRACT

In this project, we compared three different search algorithms in terms of their run times. For this comparison we have written and compiled all the algorithms in the C ++ programming language.

For all these tests we used data sets of 1K, 10K, 100K and 1M respectively. We also analyzed all these codes in the Windows operating system, then Linux, and finally in the OS-X operating system.

Windows: Intel(R) Core(TM) i5-5200U CPU @ 2.20 GHz 2.19 GHz

Linux: Intel(R) Core(TM) i5-2300 CPU @ 2.80GHz x 4

Mac: 1,4 GHz Intel Core i5

Finally, we have made suggestions about which algorithm works better by converting the results we obtained from these experiments into graphs.

Table of Contents

INTRODUCTION

In today's world, knowledge exists everywhere in life. Speed for access to this information remains the most important factor. Every day millions of people are searching for millions of information over the Internet. Therefore, search speed is much more important. Since we know the importance of the speed factor, in this project, we compare the efficiency of search algorithms considering the amounts the running times. The algorithms we compare with each other; linear, binary and hash table searching algorithms.

Context and Purpose

Our context is including about these three searching algorithms, which we obtained from different source books. We collected the items and analyzed them to get more sensible results. In addition, our project is including the results about the experiments we did. We ran and analyzed them in different ways.

Our goal in this project is to compare the efficiencies of these three search algorithms and to decide which is more efficient by taking into consideration of the running times.

Motivation

People need to access to the information fast so the running time of these three algorithms are crucial. Instead of losing time searching for information, people can use it to spend on other matters. That is why we compare the efficiency of search algorithms by taking into consideration of the running times and decide which is better. In this way, we are offering suggestions so that people can access the information they are looking for in a shorter time.

Report Summary

For these comparisons we have compiled and ran all the algorithms in the C ++ programming language. For these experiments, we used 1K, 10K, 100K and 1M data sets respectively. We also tried to display the results graphically obtained by running all of these algorithms and data sets in Windows, Linux and OS-X operating systems, respectively, to make them visually understandable. In the end, we have seen which algorithm works better.

BACKGROUND

Asymptotic Notation

We use asymptotic notation to describe the performance of algorithms as a function. According to the input size, we try to determine which function is dominant. In other words, which algorithms complexity grow more rapidly while input size is increasing. Asymptotic notation is also helpful describe algorithms’ best, worst and average cases.

For clarity, Comen’s explanation is helpful for us: “Recall that we characterized insertion sort’s worst-case running time as an2+bn+c, for some constants a, b, and c. By writing that insertion sort’s running time is Q(n2), we abstracted away some details of this function. Because asymptotic notation applies to functions, what we were writing as Q (n^2) was the function an2+bn+c, which in that case happened to characterize the worst-case running time of insertion sort.[1]”

Functions are represented different notation such as O(upper bound,) ,Q(tight bound) and Ω(lower bound) .

Comparison of the function are given below:

O(1) < O(log n) < O(n) <O(n*log n) < O(n2)<O(2n) <O(n!)

For clarity, Big O Complexity Chart is given below on Figure 1:

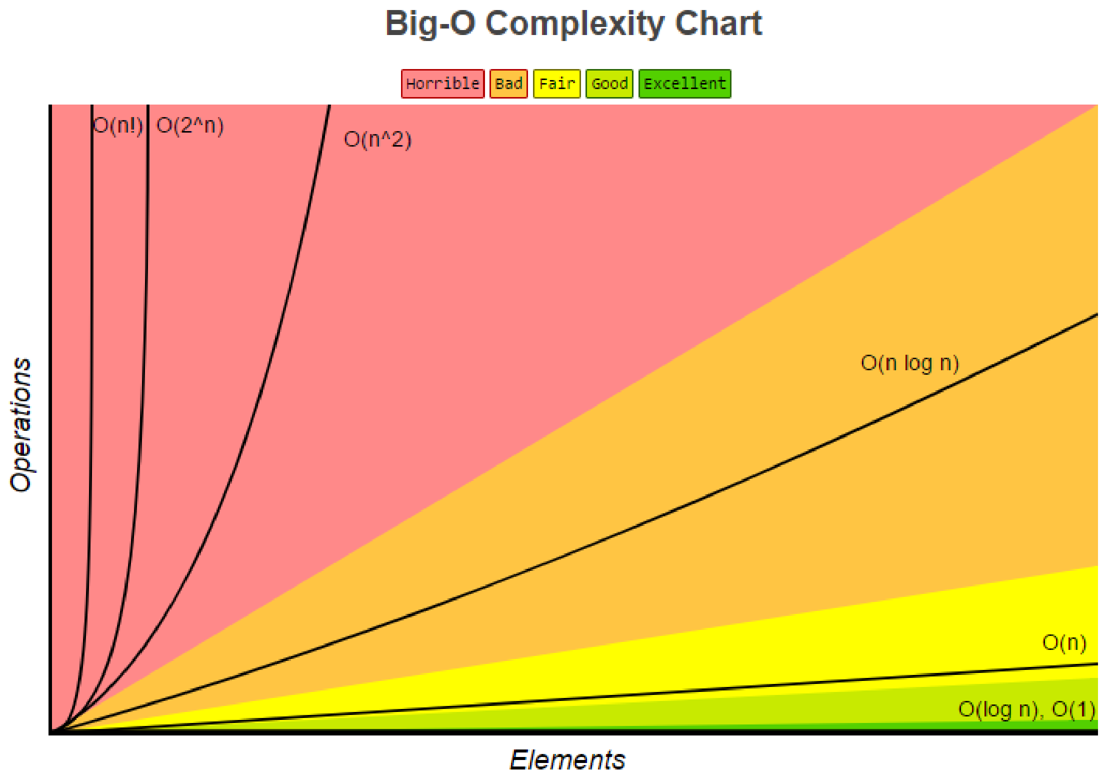

Figure 1: Big-O Complexity Chart
Retrieved from http://bigocheatsheet.com/

Searching Algorithms

Even most of people do not know, in our daily life searching algorithms are working in the background mostly. With the sorting algorithms, searching algorithms are most used algorithms in the digital world. While we are googling or try to find a word among millions of word, we do not want to wait on the computer because nobody has time to waste. That time is directly related to searching algorithms' performance in terms of time complexity. Purpose of our project is to determine which searching algorithm is best in terms of time as mentioned at the proposal in advance. We find it by doing experimental analysis

Mansour is explaining searching algorithms following, "Search algorithms aim to find solutions or objects with specified properties and constraints in a large solution search space or among a collection of objects" [2]. We focused on three fundamental searching algorithms: Linear Search, Binary Search Tree and Hash Table. These algorithms are dictionaries at the same time. Dictionary is a data structure that allows operation of insertion, deletion and searching [3]. We do some researches about these three searching algorithms running times in order to foresee the result before the experiments and compare these results with experimental results.

As an alternative possibility of our initial proposal, we can implement Red Black Tree. Red Black Tree is balanced tree thanks to its four property apart from the binary search tree. Balance is directly related to performance of the tree. More detailed information given following section.

Linear Search

This search algorithm, also known as Sequential Search, is literally a search, one by one looking for what is searched.

So to call number 15 between the following numbers:

4 6 12 8 5 15 25 34

One by one, all the numbers are taken care of. For example, starting from the beginning; 4 is the number searched? No. 6 is the number searched? No. 12 is the number searched? No. In this way, all the numbers are read one by one and the search continues until the number 15 is found.

As can be understood from the example, when a search is performed on a set of n numbers, the case in which the searched number is not found in this set may be determined by looking at the whole number n. Therefore, the complexity of the algorithm is O (n).

Hash Table

Hash Tables are very effective data structure, to insert, update, delete and retrieve the data instantly. It can achieve it by using a single function while executing these operations. It is very useful on the implementation of dictionaries [3]. As we trying to search strings at first sight hash function seems very convenient to implement. Asymptotically its search complexity is O(1), which means one can find instantly what he/she searched.

Basically it is not more than an array, but while implementing items we use a function which we consider it distributes items uniformly. So when it comes to search same function will produce same index again so that searched item could be found instantly. The problem here is function can produce the same spot for more than one item that called collision. To overcome this some collision resolution strategies have been developed. We used the double hashing which involves second hash function. If collision occurs second hash function comes up and find an empty spot. Figure 2 illustrate Double Hashing Funciton.

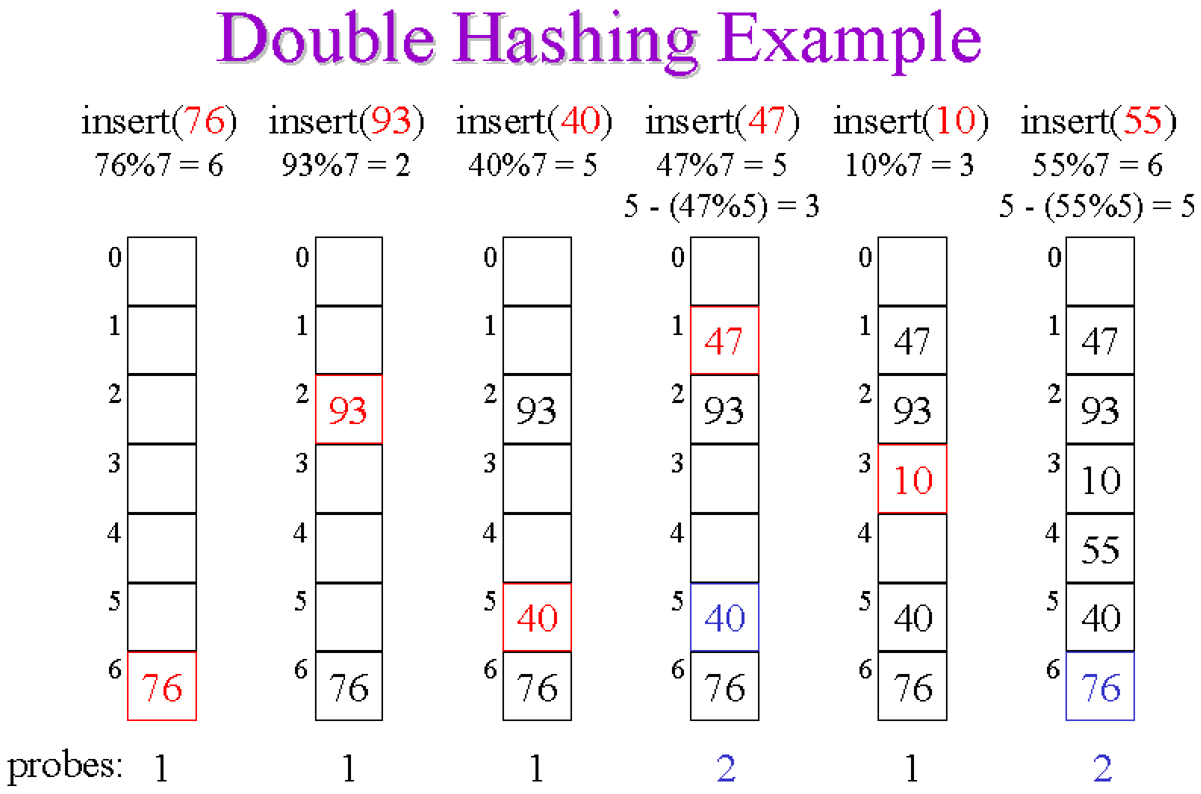

Figure 2: Double Hashing
Retrieved from: https://courses.cs.washington.edu/courses/cse326/00wi/handouts/lecture16/img025.gif

Binary Search Tree

Tree data structure is used various computer science and one of the most used area is searching. We can explain tree as follows: Tree has nodes and each node has children nodes. Specifically for Binary Search Tree, each node has a right child and left child. A node on the top of tree called root. A tree has just one root. The nodes on the bottom has not a child known as leaves. The height of the three is a longest path from root to leaf. Niamann explain Binary Search Tree's property as follow: "Assuming k represents the value of a given node, all children to the left of the node have values smaller than k, and all children to the right of the node have values larger than k" [2]. Figure 3 illustrates a binary search tree. In order to clarity.

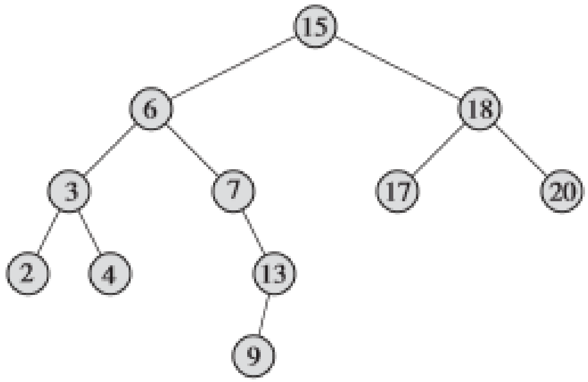

Figure 3: Binary Search Tree [1]

15 is the root of tree. 2,4,9,17,20 are leaves. Height of the tree is four because longest path is root (15) from the deepest leaf (9).

While searching on tree, at the each iteration half of tree tree or subtrees are eliminated thanks to binary search tree property. Therefore, time complexity of the binary search tree is directly related to its height. Height of the binary search tree is lgn. For randomly inserted data set, time complexity would be Q (lgn). If nodes are inserted sorted, worst case happened and time complexity would be O (n). Because tree was inserted unbalanced. Balance is distribution of node to right and left side equally. Red Black Tree is another data structure and it is highly balanced tree. Even the worst case of Red Black tree even O (lgn).

Overall, the asymptotical complexiry of all three searching algorithms are given in Figure 4.

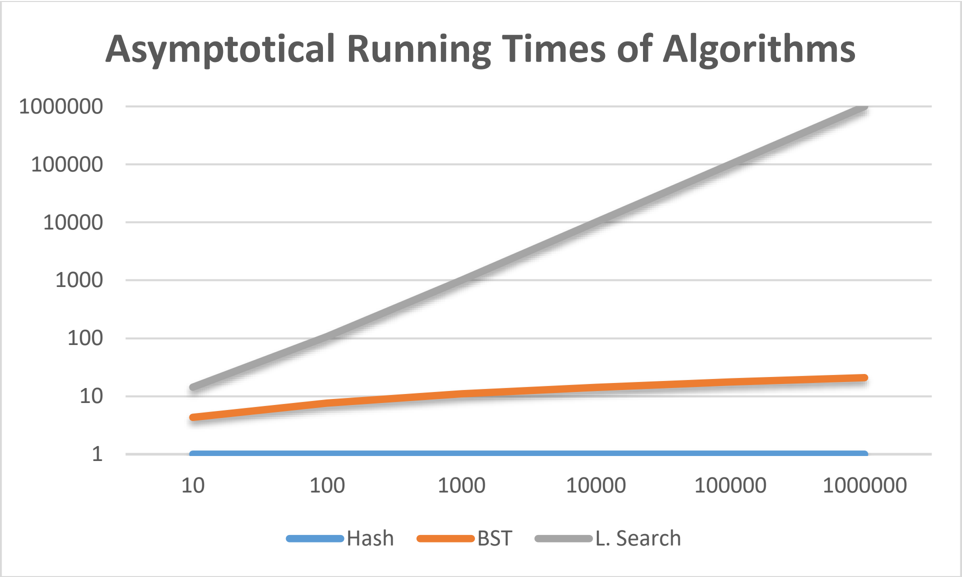

Figure 4: Asymptotical Running Times of Algorithms.

Literature review

Searching algorithms are indispensable part of digital world. There would be development on this field day by day. The algorithms that we study on just small but fundamental part of this area. In addition to these three algorithms, Red Black Tree is important data structure can be used for searching. However, today’s algorithms are far beyond these. With the learning from data, machine learning and artificial intelligence technologies searching service more efficient. Mansour is support it with this sentences: Search is one of the most frequently used problem solving methods in artificial intelligence (AI), and search methods are gaining interest with the increase in activities related to modeling complex systems [1]. ”

METHODS

In order to do our experiments, we have implemented and compiled three algorithms in C ++ programming language. As a result of this review, we analyzed the dependency of running times on the number of inputs using 1K, 10K, 100K and 1M number of elements respectively as data set. Later on, we wanted to run these codes on Windows, Linux and OS-X operating systems, respectively, so that the efficiency difference between them is independent of the operating systems. The properties of the fields we use for this are as follows:

Windows: Intel (R) Core (TM) i5-5200U CPU @ 2.20 GHz 2.19 GHz

Linux: Intel (R) Core (TM) i5-2300 CPU @ 2.80GHz x 4

Mac:1.4GHz Intel Core i5

We did the same experiments on these platforms using the same numbers of data in the same way.

Sample Data Set

We found the data sets from the Mediafire internet site [4]. These words are collected from dictionary database and we made some regulations on them.

We chose three test words for each data sets, from the beginning, middle and end of each sets. Our goal in doing this is, to be able to perform best, average and worst case analyzes. Used sample words are seen in Table 1.

	1K	10K	100K	1M
head	ddbd	ddbd	ddbd	ddbd
middle	seems	etxmsa	substuted	heartstrings
hindmost	debaser	jozef	unbuckle	dadd

Table 1 Used Sample Words

Linear Search Implementation

We implement the linear search using dynamic array. Firstly, we put the elements into this dynamic array and started the clock. We did not consider the time to put elements to array. We consider just searching time. When we finish to find the element, the program gives us the search time and place where the element is found.

Hash Search Implementation

While implementing our algorithms we use c++ programming language. For the hash search specifically we created a HashTable class to hold items and to make some operations on them such as retrieve and print. After reading text file and input size from command line we run insert function for the whole dataset.

In the insertion phase we created a loop which involves 2 hash function one for the first attempt the other for the other attempts and the loop runs till finding an empty spot. Also if iteration cannot find an empty spot more than the table size iteration, it exits and reports that it couldn't find an empty index.

In the retrieval phase naturally it behaves as the same in the insertion phase but its exit condition from the loop depends on finding the searched index.

While implementing hash functions we faced lots of trouble because of the repetitive words in the dataset and the bigness of the dataset. We choose the table size double times of our input size. Because of these reasons lots of hash functions creates same spots to find appropriate hash function took so much time. But at the end we choose our hash functions as below:

H1: k mod m

H2: k mod (m-2)

H: (H1 + i * H2) mod m

m = tableSize, k=ascii index of the string

In the main function we ask user to search a string and give them a precise search time.

Binary Search Tree Implementation

It was not the first time implementation of the binary search tree such as the other searching algorithms. We have been implemented binary search tree for number numbers in advance yet, not for words. A function is written for transforming string word to ASCII code, which is integer number. Thus while searching; checking is done by using ASCII code actually. We had difficulty in insertion part. At the first part, we wrote recursive function for insertion, yet we came across an error about stack over flow for large amount of number input size such as 1M so that we change our strategy and wrote iterative function for insertion. Another cornerstone of this implementation, in order to avoid worst case at the insertion part, we care about reading sample data file. If the root was not selected well, the tree has a possibility to become unbalanced. Therefore, we select first letter of word for root away from first or last letter of the alphabet.

RESULTS

Experimental Analysis on Linux

Best case

In this part of experiment on Linux we clearly see in Table 2 and Figure 5 that all searching algorithms runs in a very small amount of time. Moreover increasing sample size is not changing the running time of algorithms. Because we searched the first items in the dataset for the best cases. Other than hash search algorithm, these algorithms have best average and worst cases but hash algorithm does not. It acts same for the best average and worst cases.

Linux		Sample Size		
Best C.	1000	10000	100000	1000000
Hash	6,74	6,504	6,158	6,716
BST	5,317	4,873	4,969	5,387
L.Search	18,564	19,612	18,94	13,503

Table 2: Comparison table of Best Cases these three searching algorithms on Linux.

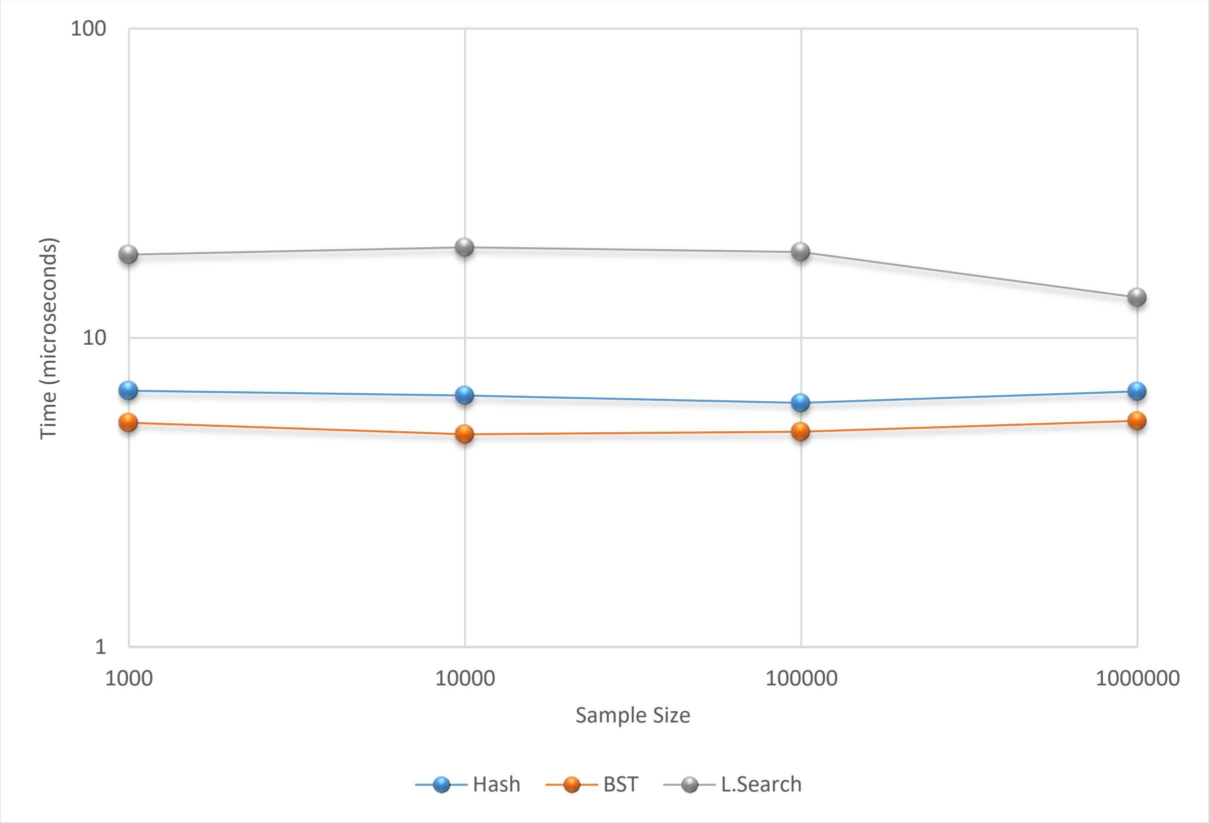

Figure 5: Comparison chart of Best Cases these three searching algorithms on Linux.

Average Case

In the average case of the experiment we see that Hash and Binary Search Tree algorithms does not effected too much but the Linear Search algorithm increases its running time too much. Because for the average case we searched the middle items in our dataset. Linear Search iterates till finding the searched item, so its running time increases. It behave as in parallel to its asymptotical analyze.

Linux		Sample Size		
Average C.	1000	10000	100000	1000000
Hash	3,041	2,66	2,712	2,965
BST	5,493	10,696	10,126	11,964
L.Search	14,277	127,561	1178,09	9323,6

Table 3: Comparison table of Average Cases these three searching algorithms on Linux.

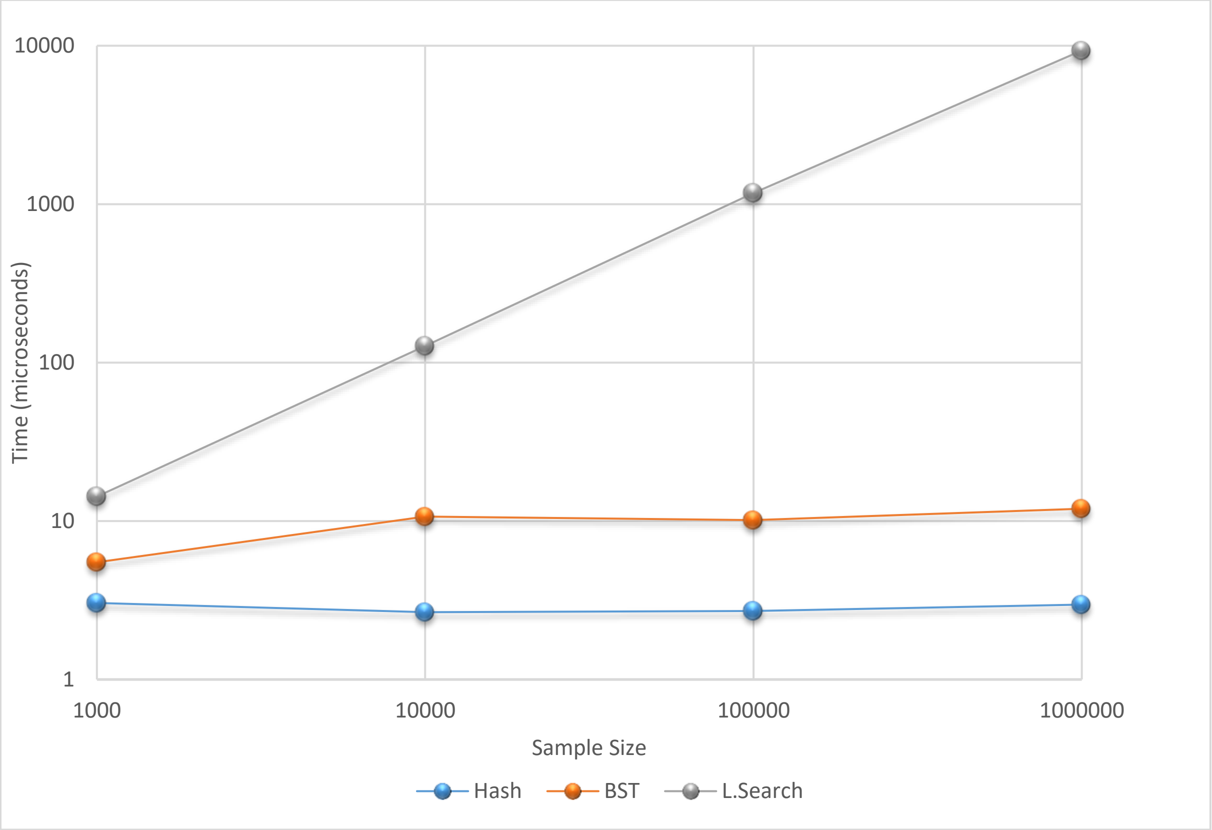

Figure 6: Comparison chart of Average Cases these three searching algorithms on Linux.

Worst case

In the worst case of the algorithm, similar to average case, linear search algorithms search time takes too much time. As stated above linear search's average and worst case asymptotically takes O (n^2). In here we clearly see that binary search tree's complexity increased too. Because its complexity is O (lgn) in the worst case, but still it is a successful searching algorithm.

Linux		Sample Size		
Worst C.	1000	10000	100000	1000000
Hash	2,525	2,566	11,042	3,756
BST	4,481	7,37	15,457	20,196
L.Search	30,329	243,817	2888,17	19840,5

Table 4: Comparison chart of Worst Cases these three searching algorithms on Linux.

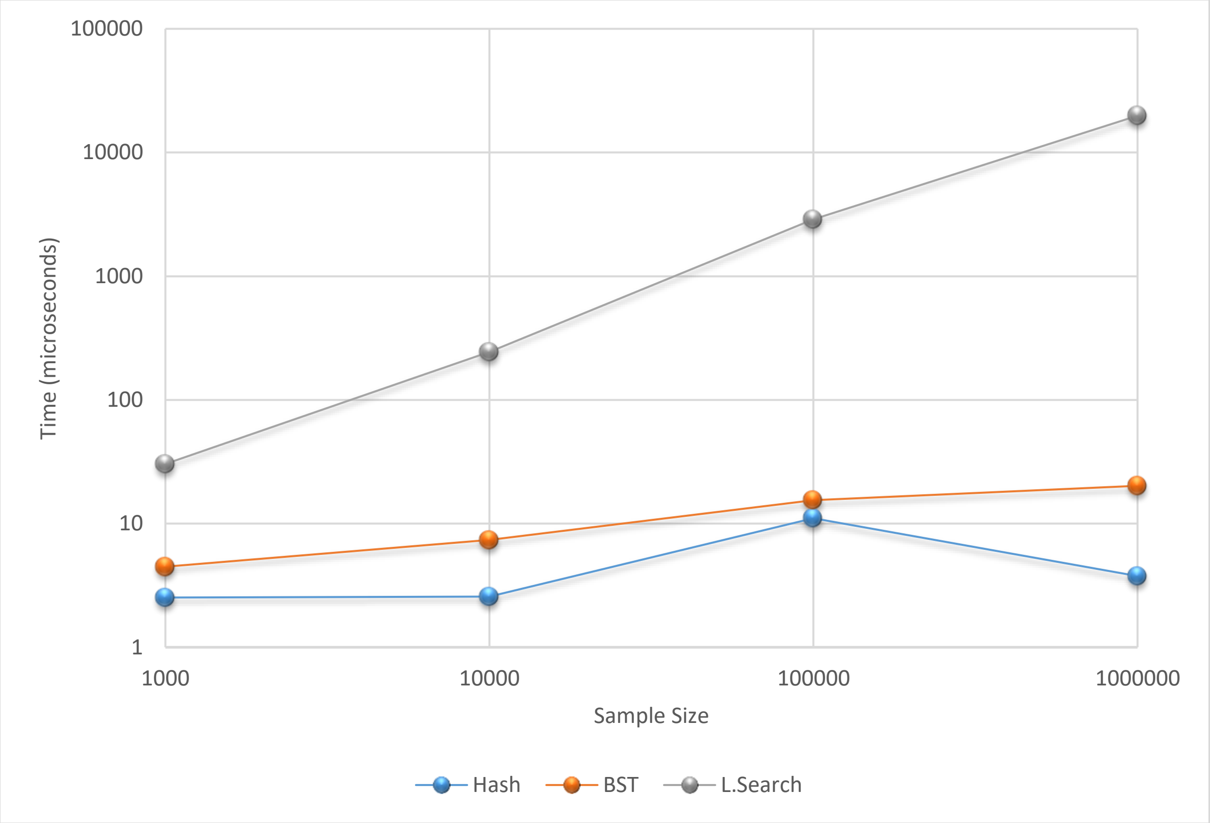

Figure 7: Comparison chart of Worst Cases these three searching algorithms on Linux.

Experimental Analysis on Windows

At the first part we did not get sensitive results especially for the small sample size on the Windows thus we add extra code to see the small numbers that are close to zero.

All three searching algorithms' best, average and worst cases results on Windows are given below:

Best case

It is observed from Table 5 and Figure 8 that best case results are not depend on sample size and the result is not much sensitive. Running time of hash and linear search are close each other, but binary search tree is fast than the others. However, we do not consider best case much because it is not showing real world to us.

Windows		Sample Size		
Best C.	1000	10000	100000	1000000
Hash	19,128	21,927	17,262	18,661
BST	4,199	3,733	3,732	4,199
L.Search	21,927	19,128	17,262	16,795

Table 5: Comparison table of Best Cases these three searching algorithms on Windows.

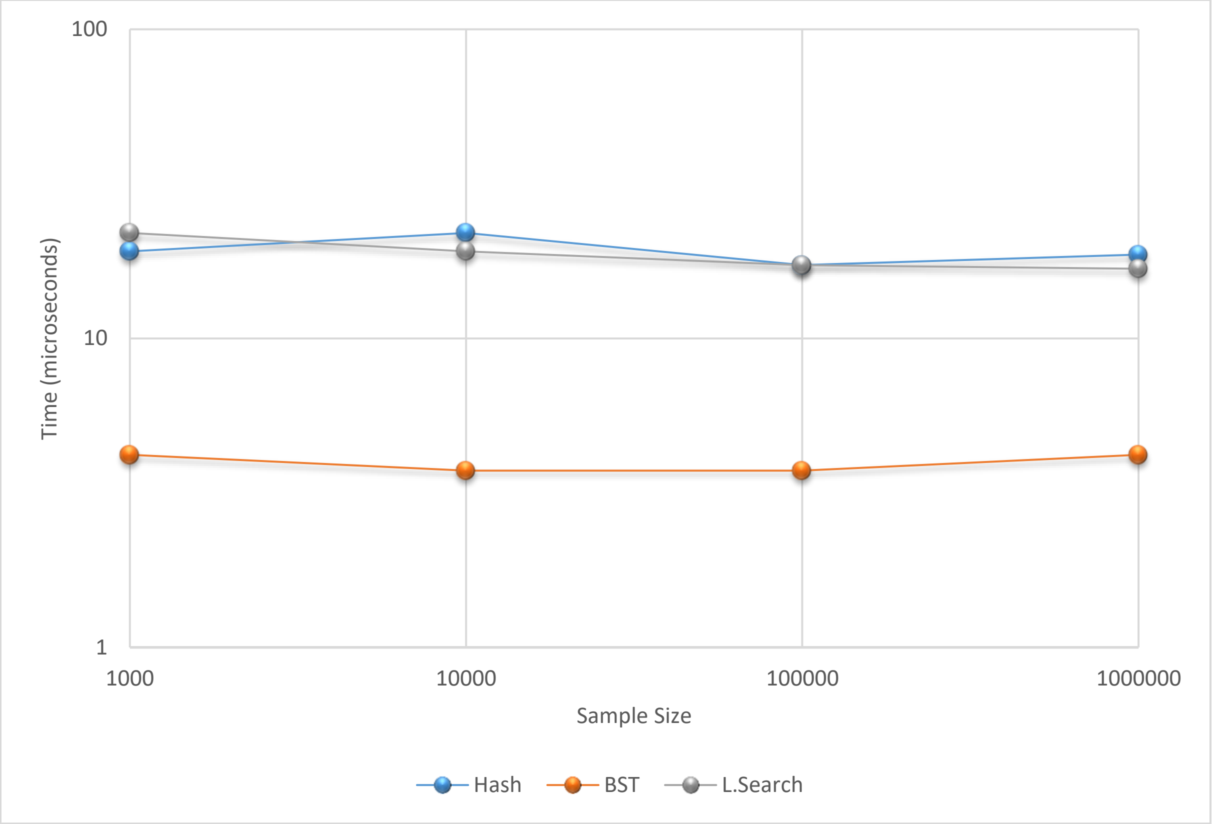

Figure 8: Comparison chart of Best Cases these three searching algorithms on Windows.

Average Case

It is seen clearly that Hash is fast than the others and linear search is slowest algorithm. Running times of hash table is not changing as we expect.

According to the Figure 9, running time of linear search is increasing linearly such as expected asymptotically. Line of binary search tree is not the same lgn function exactly, yet similar to it.

Windows		Sample Size		
Average C.	1000	10000	100000	1000000
Hash	20,061	20,528	22,86	24,726
BST	197,343	587,831	436,208	459,067
L.Search	306,511	2794,53	28572,3	297210

Table 6: Comparison table of Average Cases these three searching algorithms on Windows.

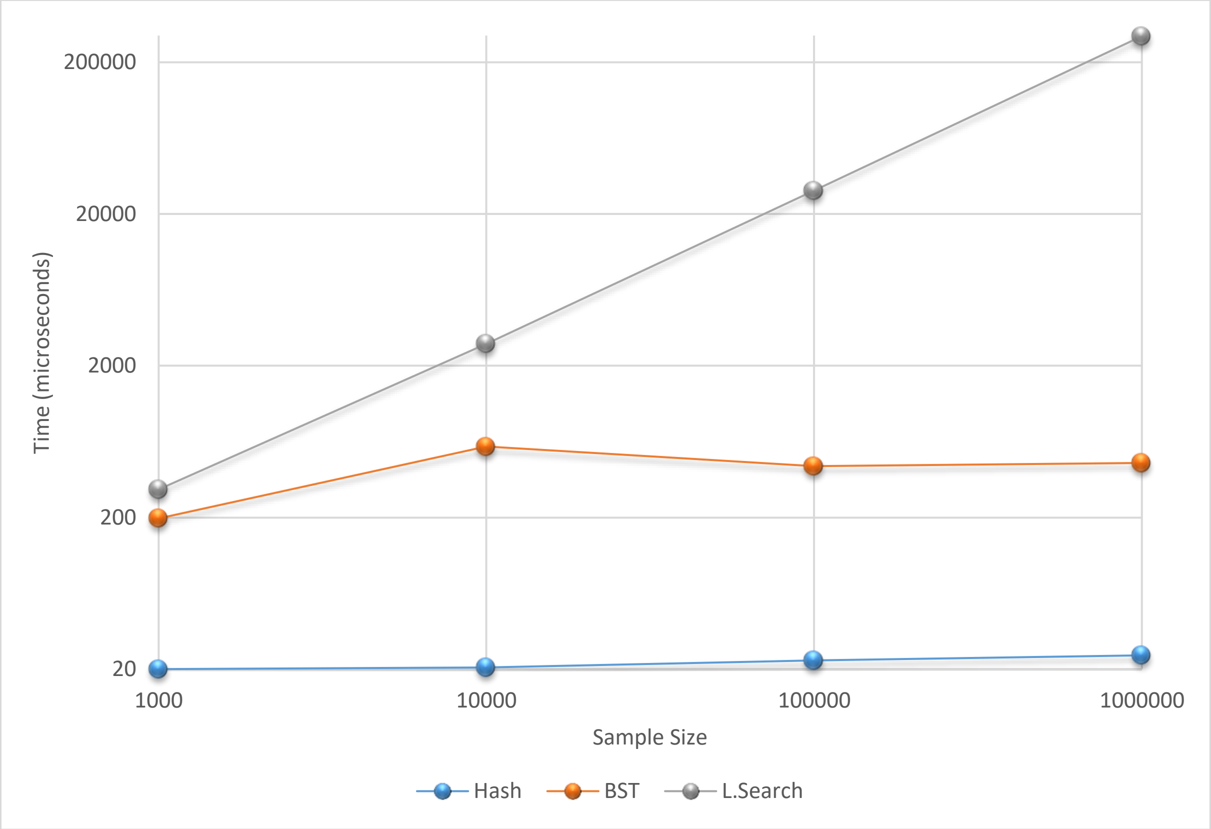

Figure 9: Comparison chart of Average Cases these three searching algorithms on Windows.

Worst case

As see on the Table 7, the running time of the hash table is not changing again according to sample size exclude 1M sample size because the number of collision is increasing for huge numbers and this collisions increase running times linearly. Despite the collisions, hash table is best and second is binary search tree and last one is linear search again.

According to the Figure 10, line of linear search increasing linearly again. Curve of binary search tree is already the same as the *lgn* function. It is inevitable that hash table is fastest.

Windows		Sample Size		
Worst C.	1000	10000	100000	1000000
Hash	20,061	18,662	22,393	44,32
BST	154,422	243,53	579,899	772,577
L.Search	625,153	8623,84	56894,5	571937

Table 7: Comparison table of Worst Cases these three searching algorithms on Windows.

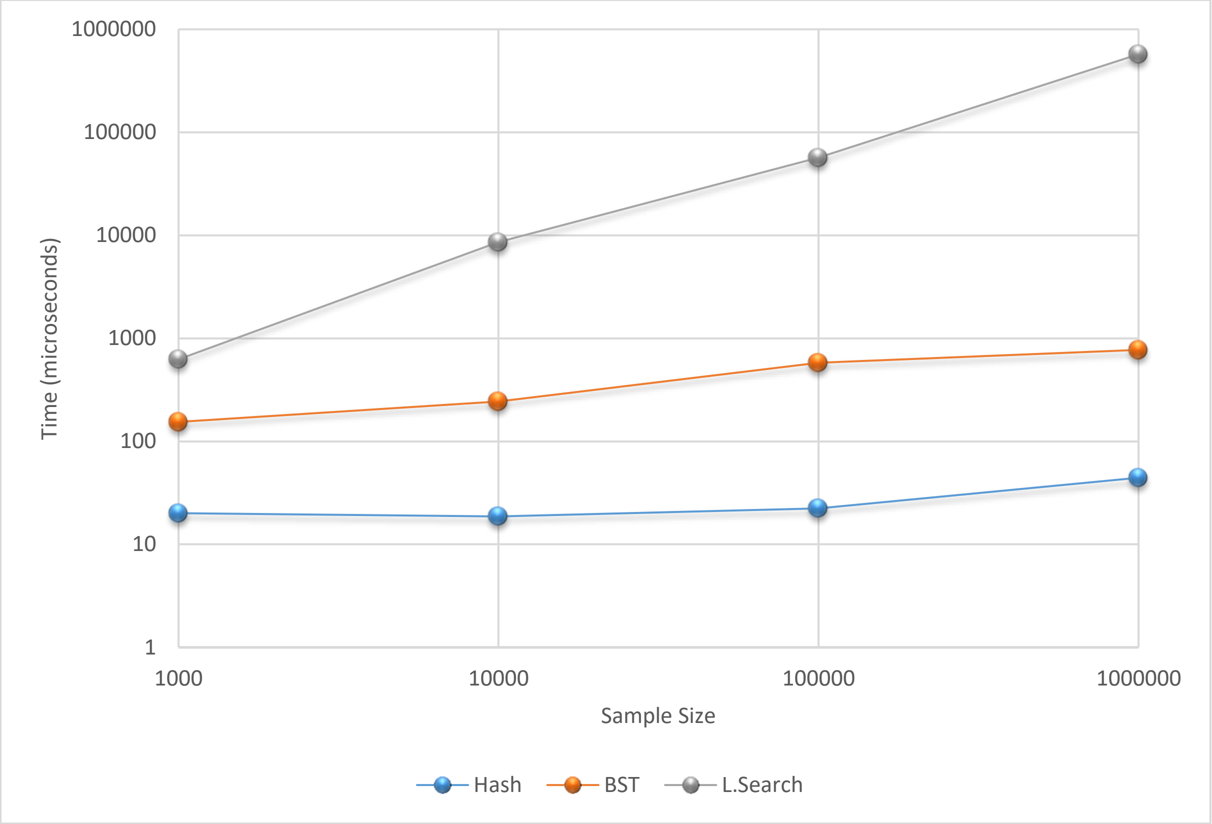

Figure 10: Comparison chart of Worst Cases these three searching algorithms on Windows.

Experimental Analysis on MacOS

Best case

In this case, we can see the results of three searching algorithms on MacOS. In the best case, as you see on table, binary search tree is faster than the others. In addition, we can also obtain information from the Table 8, input size has effect on these three searching algorithms.

MacOS		Sample Size		
Best C.	1000	10000	100000	1000000
Hash	2,176	3,22	2,506	3,285
BST	0,813	1,142	1,195	1,132
L.Search	5,334	6,904	6,552	6,19

Table 8: Comparison table of Best Cases these three searching algorithms on MacOS.

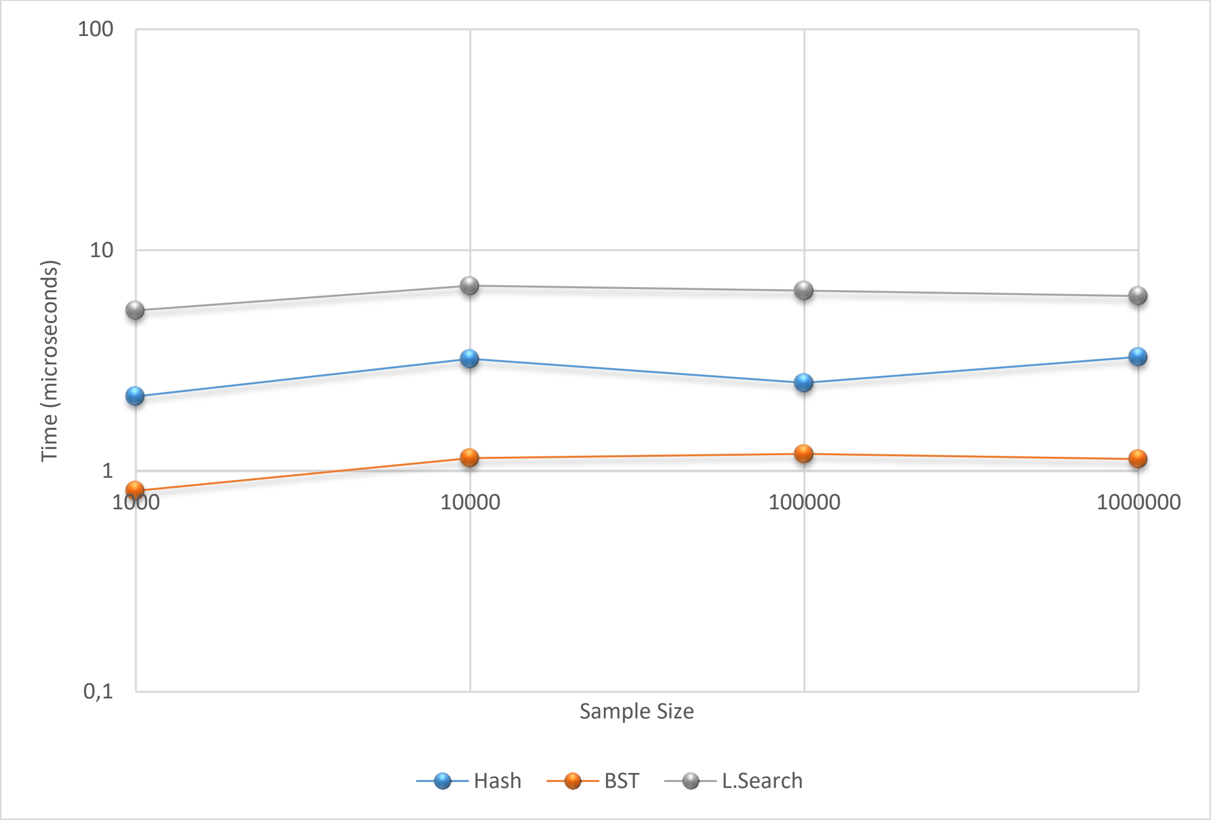

Figure 11: Comparison chart of Best Cases these three searching algorithms on MacOS.

Average case

As we see on table, hash table has fastest running time in three searching algorithms. It does not depend on input size as you see. Binary search tree and linear search are depend on input size. When we increase the number of inputs, this two algorithms works slower.

MacOS		Sample Size		
Average C.	1000	10000	100000	1000000
Hash	3,041	2,66	2,712	2,965
BST	5,493	10,696	10,126	11,964
L.Search	14,277	127,561	1178,09	9323,6

Table 9: Comparison table of Average Cases these three searching algorithms on MacOS.

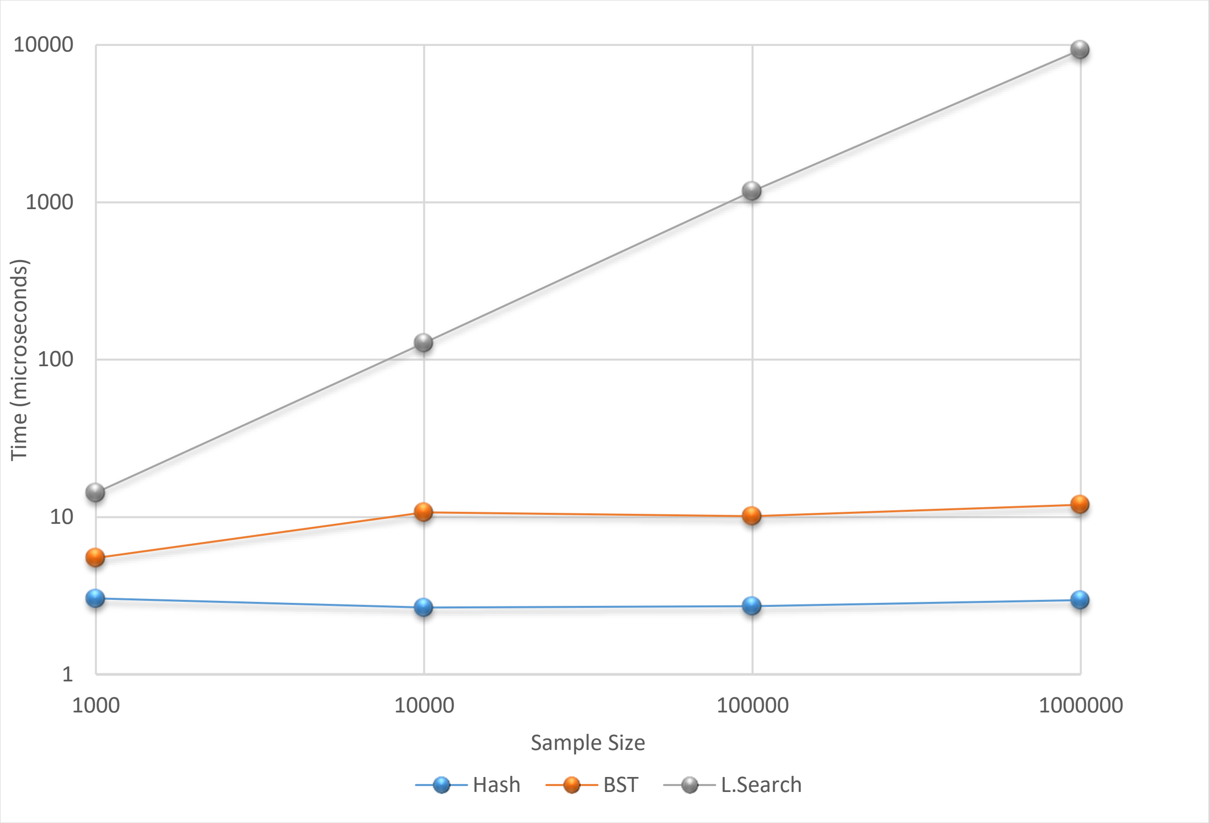

Figure 12: Comparison chart of Average Cases these three searching algorithms on MacOS.

Worst case

In worst case, we try to find the latest word to obtain worst-case. As you see on Table 10, hash table has best running time again. Binary search tree and linear search algorithms slower than hash table. The running time of the linear search algorithm has shown that too much effort has been taken to find the last element.

MacOS		Sample Size		
Worst C.	1000	10000	100000	1000000
Hash	3,375	3,413	3,51	4,7
BST	5,028	10,252	20,078	20,431
L.Search	45,256	254,258	6736,24	26230,6

Table 10: Comparison table of Worst Cases these three searching algorithms on MacOS.

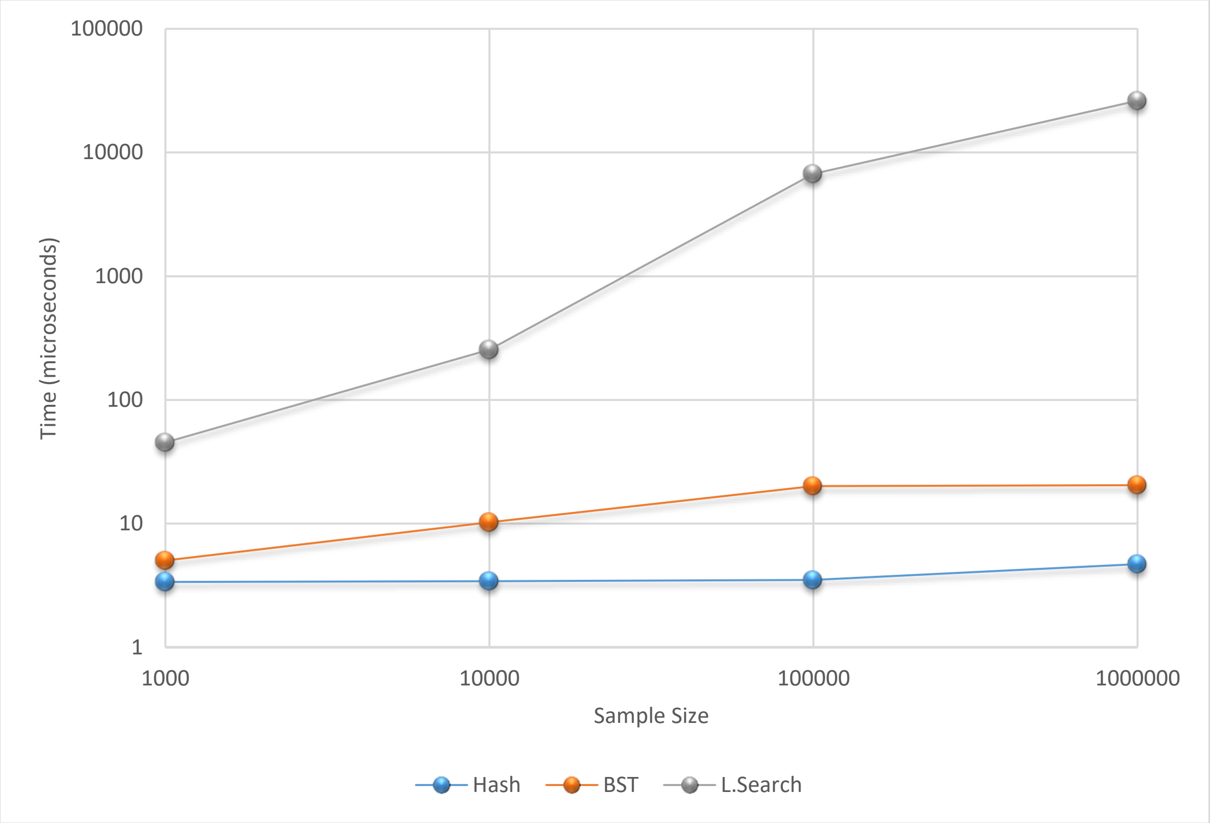

Figure 13: Comparison chart of Worst Cases these three searching algorithms on MacOS.

ANALYSIS

As we mention before, we evaluate the result according to the average case. Average case results on all operating systems are given below Table 11:

	Test Words			
Average Case	**1000**	**10000**	**100000**	**1000000**
Hash Windows	20,061	20,528	22,86	24,726
Hash Linux	3,041	2,66	2,712	2,965
Hash MacOS	2,448	2,556	3,606	3,362
BST Windows	197,343	587,831	436,208	459,067
BST Linux	5,493	10,696	10,126	11,964
BST MacOS	8,53	17,832	16,545	15,89
L.Search Windows	306,511	2794,53	28572,3	297210
L.Search Linux	14,277	127,561	1178,09	9323,6
L.Search MacOS	19,82	232,275	1078,03	10166,7

Table 11: Comparison table of Average Cases these three searching algorithms overall.

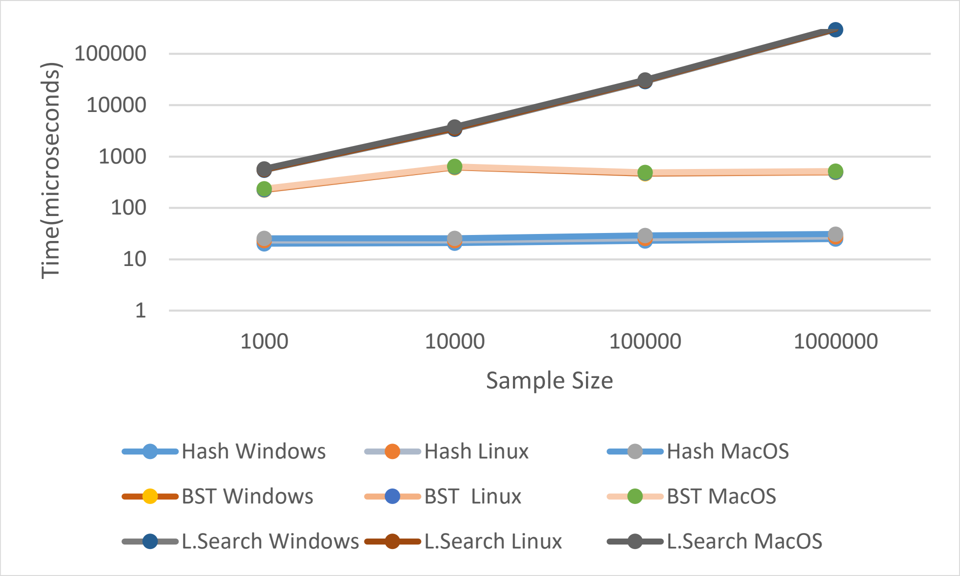

Figure 14: Comparison chart of Average Cases these three searching algorithms overall.

As seen on the Figure 14, performance of the algorithms are not depend on the environment. Both used operating system and used computer are different, yet result is the same. All results of searching algorithms on different operating system become cluster with each other. Result is clear that Hash table is fastest and Linear Search is slowest. Experimental result is consistent with the asymptotical result seen in Figure 4. Only binary search tree result for 10K is a little bit different from the *lgn* function. We think that it is related to sample word because result do not change for different environments.

Running times on Windows for all algorithms are larger than the other operating system because Algorithms are run at just 25 percent performance of computer with Windows, contrast to the other computer are run at almost full performance.

EVALUATION

Research Package

In order to do our project, we first performed an information search from sources related to algorithm analysis. With the information we have found, we decided to consider only average running times for the three algorithms. Then we decided to use C ++ programming language for the implementation of these algorithms. We started deciding who would implement which search algorithm.

Main Work Package

After coding and compiling, we compared the running times which to find the element we wanted by using the 1K, 10K, 100K and 1M word as data sets respectively. By doing this, we aimed to analyze the effect of the number of inputs on the call duration.

We were not satisfied with these and we tested these three algorithms in different operating systems with the same number of data sets. For this, we first used a computer with a Windows operating system, followed by a Linux computer, and finally a computer platform with an OS-

X operating system. Our goal in doing this is; The algorithms show that the efficiency difference between each other is platform independent.

Visualize Package

We converted the data we obtained to the graphs to make as a result of all these experiments understandable from the visual point of view and we suggested which algorithm is better.

CONCLUSION

As we conclude, we suggest using three string searching algorithms, we implement them and made tests on 3 different operating systems, with 4 different input size and with the best, average and worst cases of the algorithms of each. In total, we made 108 tests to decide which algorithm is the best for string searching and we learned that considering only searching times Hash Search is the best independent from the environment and the order of the searched word. In addition, it does not matter how large the data set is. Nevertheless, the hash functions must be chosen appropriately.

With hours of work and number of tests, we are proud to present our project successfully.

REFERENCES

[1] Cormen, T. and Leiserson, C. (n.d.). Introduction to algorithms, 3rd edition. 1st ed.

[2] Nashat Mansour, Fatima Kanj, and Hassan Khachfe. 2011. Enhanced Genetic Algorithm for Protein Structure Prediction based on the HP Model. Search Algorithms and Applications (2011). DOI:http://dx.doi.org/10.5772/15796

[3] Niemann, T. (2017) Sorting and Searching Algorithms: A Cookbook. N.p., n.d. Web. 24 Feb.

[4] www.mediafire.com/file/dr2xsxxpgirnirs/1.1million_word_list.txt.gz