UNIVERSITY OF BUEA

Faculty of Science

Department of Computer Science

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

Project Report

Dynamic Hash Tables: A Simple Comparison of the Dictionary and Logarithmic Methods.

AWA FONKAM BRANDON LOIC
SC16A648 Supervisor

William S. Shu, PhD.

September 2019

DECLARATION

I now declare on my honour that I have written this project report. To the best of my knowledge, all borrowed ideas and materials have been duly acknowledged. It has not received any previous academic credit at this or any other institution.

AWA FONKAM BRANDON LOIC
SC16A648

Department of Computer Science
Faculty of Science

CERTIFICATION

This is to certify that this report entitled "DYNAMIC HASH TABLES: A SIMPLE COMPARISON OF THE DICTIONARY AND LOGARITHMIC METHODS" is the original work of AWA FONKAM BRANDON LOIC with Registration Number SC16A648, student of the Department of Computer Science at the University of Buea. All borrowed ideas and materials have been duly acknowledged by means of references and citations. The report was supervised in accordance with the procedures laid down by the University of Buea. It has been read and approved by:

William S. Shu, PhD CEng CITP MBCS	Date
Dr Denis L. Nkweteyim Head of Department for Computer Science	Date

Dedications

II would like to dedicate this study to my beloved parents, who have been my source of inspiration and gave me strength when I thought of giving up in a period of ill health, and continually providing their moral, spiritual, emotional, and financial support.

I also dedicate this to my loving brothers, sisters, relatives, mentor, friends, and classmates who shared their words of advice and encouragement to finish this study.

And lastly, I dedicate this study to the Almighty God. Thank you for the guidance, strength, power of the mind, protection and skills, and for giving me a healthy life. All of these, I offer to you.

Acknowledgement

I would like to express some special thanks first of all to the LORD ALMIGHTY GOD for the guidance, grace and strength he has always provided me.

Then, I would like to express special gratitude to my lecturer and supervisor, Dr William S. Shu, whose valuable guidance has helped me patch this project and make it a proof of success. His suggestions and instructions have served as the major contributor towards its completion.

I would also like to thank dearly all my lecturers who taught and transferred all necessary tool-set of skills to me. Special thanks to my HOD, Dr Denis Lemongew Nkweteyim, Dr Joan Beri Ali Wacka for her tough guidance and Dr Nyamsi Madeleine for being such a great lecturer.

Finally, I would like to thank my loving family and friends, especially my friends Bomen Tchouagueng and Massina Eloundo, whose encouragement and assistance cannot go unforgotten during ill health times.

Abstract

Dynamic hash tables provide fast storage and retrieval of data in memory. This fast storage is very important in the development of efficient programs. There is a limitation on how much data can be stored in memory due to RAM's finite size. This limitation was remedied by introducing logarithmic hash tables which could index data stored on disk.

This project designed an experiment to compare the dictionary and the logarithmic hash table's performance. We observed the behaviour of each method with a varied amount of input size ly.

In-memory hash tables provide fast access to large numbers of objects (here integers) with more space overheads. However, for a huge input size, dictionary hash tables are cache expensive than logarithmic ones. Our experiments show that the logarithmic method uses less cache and memory for huge input sizes than the dictionary method. We also observed that the dictionary method's space complexity has a linear growth; meanwhile, the logarithmic space complexity has a logarithmic growth rate.

Our experiment's results correspond to the expected theoretical performance after minor deviations, such as a sudden drop in performance caused by thermal throttling, were corrected. Furthermore, we suggest possible improvements to increase the performance and robustness of the algorithms.

Contents

Declaration i

Certification ii

Dedications iii

Acknowledgement iv

Abstract v

1 Introduction 1

2 Background of Study 1

3 Aim and Objectives of Research 1

4 Document Structure 1

5 Performance of Dictionary and Logarithmic Approaches 2

6 Theoretical performance Analysis 2

7 Experimental design 2
- 7.1 Validity of Instruments . . . 2
  - 7.1.1 Measuring instruments . . . 2
  - 7.1.2 Errors in measurements . . . 2
  - 7.1.3 Description of statistical methods used, and their justifications . . . 2
- 7.2 Experimental Environment . . . 3
- 7.3 Experimental variables: . . . 3

8 Data and operation sequences generation 3

9 Experimental Run 4

10 Data Analysis: 4

11 Result 5
- 11.1 Output data collected . . . 5

12 Discussion 8

13 Conclusion 9

References 10

14 Appendix 11

Appendix 11
- A Extendible hashing Algorithm breakdown . . . 11
- B Input Size variation . . . 11
- C Data generating function . . . 12
- D Dependent Variables measurement . . . 12

Glossary **13**

List of Tables and figures

1 Statistics gotten from running the logarithmic hash table. 6

2 Statistics gotten from running the dictionary hash table. 6

4 Graph of Input Size Vs Memory in both Hash table 7

5 Graph of Input Size Vs Search in both Logarithmic and Dictionary Hash table 7

1 Introduction

Dictionary and logarithmic hash tables, approaches to dynamic growth differ in two basic ways. First, the way their structures grow as new items are either added or deleted from a hash table, and second, how old hashes are restructured to accommodate new hashes.

Dictionary hash tables use a linear hashing scheme, and logarithmic hash tables use the extendible hashing scheme. Both hashing schemes are examples of dynamic hashing (which means the hash table's size can be altered as new entries are added or old ones deleted).

Linear hashing has a growth strategy of doubling the hash table's size then rehashing every item. This hashing scheme is, however, slow because writing all pages to disk is too expensive.

On the other hand, extendible hashing is a technique with a dynamic structure which implies that it can grow and shrink as the database grows and shrinks.

By analysis and simulation, we study the performance of extendible hashing compared to linear hashing in-memory.

2 Background of Study

Hash tables are an abstract structure for very fast data storage and retrieval. As such, they are used in many application domains such as Dictionary word lookup, Password Verification, auto- completion algorithm, and quick disk access lookup (that is, linking path names to file).

The idea in designing hash tables is to have a structure in which the time to store or retrieve randomly an item from the table is constant at any instance in time, just as in arrays.

Dynamic Hash tables are considered an abstract data structure with O(1) 's expected time complexities for its insertion, search, and deletion. It has an added advantage over static hash tables in that one can alter the hash table size as entries are added and removed.

Will the logarithmic approach's performance to the dynamic hash table still be as efficient in terms of time complexity as that of the dictionary method?.

3 Aim and Objectives of Research

The project aims to study the performance of linear hashing (Dictionary hash tables) and Extendible hashing (Logarithmic hash table) and see how their results compare with each other and against their expected theoretical performances. We will try to archive this by designing a suitable experiment. With analysis and simulation, we will study each hashing scheme's performance and account for their divergence (if any) from expectations and understand the performance differences between the two hashing schemes. While we are at that, we will explore the computational time and space complexities of the linear and extendible hashing scheme, study how they compare and how and why they deviate from their theoretical complexities.

This project is important because it discusses the limitation and advantages of choosing the right hashing scheme for an application domain.

4 Document Structure

The rest of our report has the following structure. In section 5, we will design and experiment to compare the two approaches to hash table construction and present its results in section 11. Section 12 is a discussion on the significance and interpretation of our work, including the reliability of our results and the nature of the statistical models used, and Section 13 is the conclusion.

5 Performance of Dictionary and Logarithmic Approaches

In this chapter, we design an experiment to simulate the application of both the logarithm and dictionary approaches to hash table construction. From the data we collected from the experiment, we established a regression model that we can use to extrapolate the performance of hash tables of any input size.

6 Theoretical performance Analysis

In the dictionary, the space required for the hash table's growth is linear, that is, for every key is associated with a Bucket into which a hash is stored. The number of hashes is proportional to the number of buckets. On the other hand, the logarithmic method, the space requirement for growth is logarithmic so, the ratio of the number of buckets to the directory each bucket points to is B*LOG(N):N. Here N is the number of entries in the hash table, and B is the bucket size of each directory. Theoretically, a dynamic hash table using linear and extendible hashing [1], [2] have time complexity of O(1) in the best case and O(n) in the worst case for insertion, search and deletion. The space complexity is O(n) for both best and worst case in linear hashing. Meanwhile, for extensible hashing, [3] on average , the space complexity is $\Theta(N^{1+1/B}/B^2)$.

7 Experimental design

We adopted an experimental research method to study each hash table. For this experiment, the entries stored in our hash table are positive integers, though the result is not affected if other data types are used. We implement the dictionary approach to dynamic hash tables using the compact-hashing technique [1]: an In-memory access technique which uses chaining for collision resolution. The is a period of growth in this technique call rehashing which is time costly. So to make the comparison of the insertion times in both hashing schemes comparable, amortization of the insertion time with the rehash time for compact-hashing was done to even out the cost of insertion for large Input size (larger than table size) over many iterations.
For the logarithmic approach, we implement an extendible hashing structure; a disk-based access technique with a dynamic structure and no rehash is required as the structure expands and shrinks as required.

7.1 Validity of Instruments

7.1.1 Measuring instruments

Measurements were done using a UNIX resource usage module called `getrusage`. The advantage of using this module is that it records both the system time and the user time of any running process. This resource module can also measure memory usage of a process and, it has a good level of accuracy since it uses the CPU clock cycle for timing.

7.1.2 Errors in measurements

Due to errors that could occur during algorithm performance measurements, we minimized errors by increasing the precision, accuracy and resolution of our measurement instruments. To amortized random errors (caused by CPU wait time, Overheating, uncontrollable system and user background processes.), we repeated the experiment several times and used the mean value of the output. Also, we used similar Input size and Input data on both algorithms to reproduce similar run environment and to ensure deterministic results.

7.1.3 Description of statistical methods used, and their justifications

Since it is impossible to measure the performance of our algorithms against all possible input sizes, due to limitations such as time, main-memory, accumulation of random error(CPU overheat for example); we must use a statistical

model to establish an equation to approximate the performance of each hash table given any input size. With this statistical model, we can extrapolate the time and space complexities for any input size with a confidence of interval.

For this experiment, we used the linear regression model called `Least-squares minimization` David J. Lilja [4]. The least-squares minimization is of the form $y = a + b.x$, where `x` is the input variable, `y` is the predicted output and, `a` and `b` are the regression parameters that are obtained from our set of measurements.

Using Least-squares minimization with confidence intervals allow us to determine how much measurement noise there is in our estimates of the regression parameters. A large confidence interval relative to the size of the parameters would suggest that there is a large amount of errors in our regression model.

7.2 Experimental Environment

Our experiment was performed on a 64bits Intel Pentium architecture computer having a processor speed of 2.1GHZ running Ubuntu 18.04 LTS OS. This computer had 16Gb DDR4 RAM installed. For the sake of the disk-access hash table, we will also mention that the secondary storage was an SSD drive.

We ran the experiment in a bash shell, wrote a bash script B which ran the driver program interfacing each hash table algorithm. From the script parameters like table size and input size were passed into our algorithm. Input data for the algorithm was generated sequentially using a function C in our driver program, then, output data were collected in a file and analyzed using a statistical package. Timing and resources usage of the dependent variables was measured using a UNIX resource usage module called getrusage found the <sys/resource> library.

7.3 Experimental variables:

- Independent variables:
  - Input size (of the algorithm)
- Dependent variables:
  - Time: The computation time taken by the algorithm.
  - Space: The computation space used by the algorithm.

8 Data and operation sequences generation

To ensure that the environment for the simulation of each hash table is reproducible, we had two driver programs that served as interfaces for the dictionary and logarithmic hash table algorithm operations. Each algorithm has four basic operations: `Create table` of size `n`, `Insert` an integer, `Search` for an integer in table, `Delete` an integer in table. The dictionary hash table algorithm has an additional operation `Rehash` called automatically when the Load factor reaches a certain lower or upper threshold.
We used the `getrusage` module to measure our dependent variables. This module returns resource usage statistic for either a calling process or all children of a calling process or a calling thread. For more detail see Manual page getrusage. Using getrusage in our driver program, we measured the time of operation for each operation and maximum memory usage for any number of operations. These measurements were saved in an output file for later analysis. We built our driver program to support command line arguments. Consequently, we could make use of some bash shell script loop B functionality to run our simulation several times in a terminal with varied Input size.

9 Experimental Run

The exact steps taken to record the benchmark result are as follows. After a fresh reboot, for each algorithm (ran sequentially) the following was performed:-

- We ran each algorithm with the same initial table size of 5,000,000 and Input size of 10,000 integers.
- After every successive runs, the Input size is double until we reached 163,840,000.
- During each run, the time to load input data, time to search all input data, time to grow table, time to delete all data from memory and, the maximum memory usage for a given input size were recorded.
- After each run, the output data (which were the time and memory we recorded above) was printed to a file.
- Every successive output data were appended to the output file.
- All the benchmark results(output data) that were collected in a file are then loaded in a Google-Sheets statistical package for analysis.

10 Data Analysis:

The benchmark results(output data) that were collected at the end of the experiment were analyzed using a Google-Sheets statistical package. They were analyzed by creating a flat table in a spreadsheet, then, from this data we plotted graphs for every process against time and input size. Comparison of the observed results of the dictionary hash table against that of the logarithmic hash table and also against the expected theoretical results was visible. We also used the flat table to establish some regression functions for every processes in our algorithm. The graphs 3a 3b 3a 5 in the next chapter helps in easy visualization of each algorithm's performance.

11 Result

In the first part of this section, we present the result obtained by running simulations these algorithms.

11.1 Output data collected

During the experiment, several runs where conducted and the output collected from run the dictionary and the logarithmic hash table are presented in table 1 and 2 respectively. From the results, we can observe that as the input size doubles, the time to perform the various operation also doubles (with a slight increase of a fraction of a second as input size become very large). These direct proportion in time and input size variation is different from our theoretical expectations of performance: that of a constant time irrespective of the input size. Nonetheless, Fagin, Ronald and Nievergel [5] in their work on extendible hashing showed that, 0(1) performance of hash table is an influence as the Load factor of the hash table increase above 90%. This along with some other uncontrollable factors (such as random errors, and systematic errors, multi-user system, system and user processes) can account for the deviation in time complexities we expected.
We can also observe that dictionary hash table hash a linear growth meanwhile, the logarithmic hash table has a logarithmic growth
From these data, we have established a statistical model that helps us estimate the time required for any given Input Size and the confidence interval.

Least-Square minimization model	**90% Confidence Interval**
$Dic\_insert(x) = 1.21e^{-06} * x + (-0.0751)$	(1.7146, 26.4893)
$Log\_insert(x) = 1.04e^{-06} * x + (-0.238)$	(2.0049, 22.8383)
$Dic\_search(x) = 1.01e^{-06} * x + (0.204)$	(1.6810, 22.2532)
$Log\_search(x) = 1.01e^{-06} * x + (-0.572)$	(1.9106, 22.7756)
$Dic\_delete(x) = 9.74e^{-07} * x + (0.0241)$	(1.4634, 21.3703)
$Log\_delete(x) = 1.29e^{-06} * x + (-1.53)$	(1.6529, 24.6391)
$Dic\_space\_usage(x) = 0.105 * x + (-34476)$	(118569.8085, 2276906.763)
$Log\_space\_usage(x) = 7.65e^{-03} * x + (651317)$	(547238.8963, 991063.3894)

Table 1: Statistics gotten from running the logarithmic hash table.

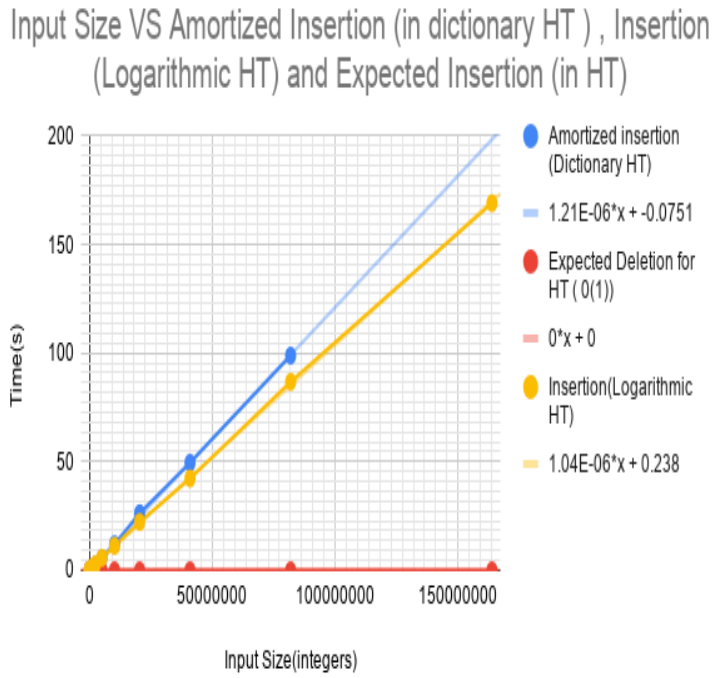
Input Size	Insertion(s)	Search(s)	Deletion(s)	Memory Usage(Kb)
10000	0.011	0.0097	0.0096	330640
20000	0.0221	0.0196	0.0192	332280
40000	0.0441	0.0393	0.0382	334892
80000	0.0883	0.0785	0.077	340684
160000	0.1765	0.1569	0.1537	351884
320000	0.3538	0.3136	0.3075	374212
640000	0.726	0.6256	0.6134	419456
1280000	1.4161	1.2557	1.3369	509448
2560000	2.856	2.5662	2.6116	689420
5120000	5.7297	5.1126	5.0522	1049336
10240000	11.211	10.5049	10.5498	1508932
20480000	22.2154	22.8622	21.2135	1508956
40960000	42.2716	42.7731	46.0011	1508972
81920000	86.7814	86.4865	96.0611	1509004
163840000	169.0787	186.3731	215.5772	1508952

The statistics were gotten from running the logarithmic hash table. The structure is dynamic, no need for rehashing hence no need for an amortize insertion time. The logarithmic hash table can support a much larger input size compared to dictionary hash table.

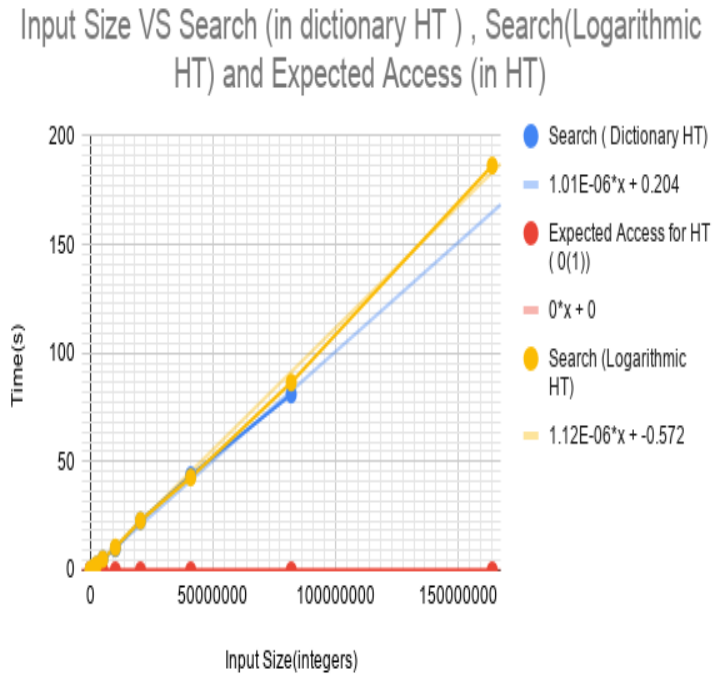
Table 2: Statistics gotten from running the dictionary hash table.

Input Size	Insertion(s)	Rehash(s)	Amortize(s)	Search(s)	Deletion(s)	Memory Usage(Kb)
10000	0.0108	0	0.0108	0.0095	0.0097	40904
20000	0.0207	0	0.0207	0.0191	0.0187	41292
40000	0.0405	0	0.0405	0.0378	0.0374	41824
80000	0.0809	0	0.0809	0.0759	0.0747	43060
160000	0.1593	0	0.1593	0.1516	0.1499	45440
320000	0.3182	0	0.3182	0.3052	0.3021	50580
640000	0.6564	0	0.6564	0.6137	0.5968	60508
1280000	1.2733	0	1.2733	1.2154	1.2022	80664
2560000	2.5558	0	2.5558	2.4528	2.4127	120456
5120000	5.1317	0.4704	5.6021	5.1208	4.8594	396852
10239999	10.6022	1.3771	11.9793	10.0716	9.9618	957580
20479999	22.3378	3.9785	26.3163	22.7588	20.4979	2037700
40959999	42.6225	6.9184	49.5409	43.7447	40.1994	4125408
81919999	84.2828	14.5907	98.8735	80.9635	79.5144	8726068

The statistics were gotten from running the dictionary hash table with an initial table size of 5,000,000. Rehashing start when the input size exceeds 5,000,000. Hence there is the need to amortize the time of insertion with the rehash time. Dictionary hash table runs out of main memory and insertion of input size of 163840000 fails.



(a) Graph of Input Size Vs Insertion in both Logarithmic and Dictionary Hash table



(b) Graph of Input Size Vs Search in both Logarithmic and Dictionary Hash table

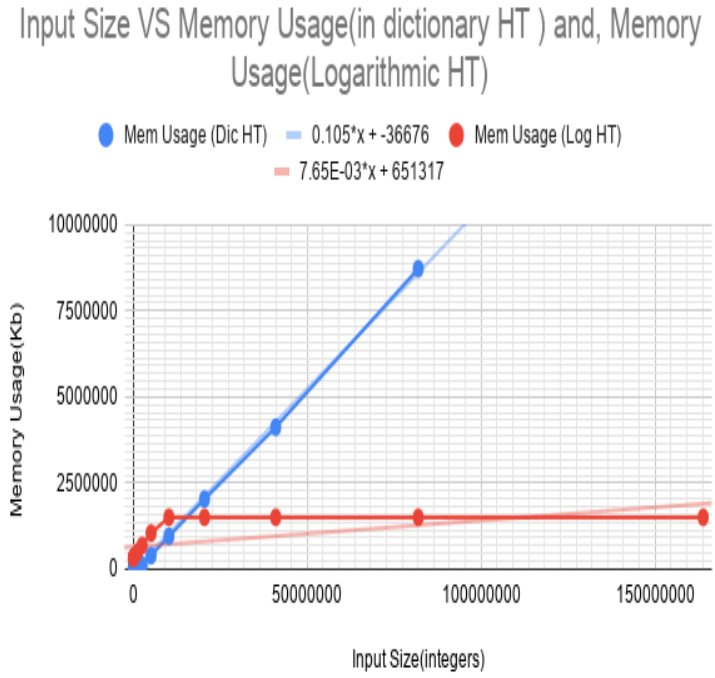


Figure 4: Graph of Input Size Vs Memory in both Hash table

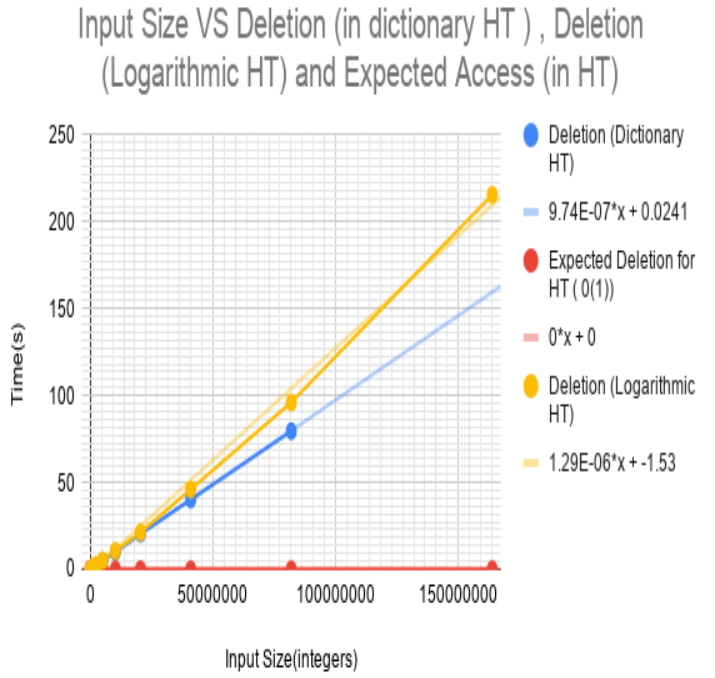


Figure 5: Graph of Input Size Vs Search in both Logarithmic and Dictionary Hash table

12 Discussion

We carried out an experimental design to compare the performance of the logarithmic hash table to the dictionary hash table. After doing that in a controlled environment and recording our results, these are the observation we noticed;

First, we observed from the data collected that the extendible hashing scheme's time complexities were almost identical to that of the linear hashing scheme for the same input size with a 90% confidence interval. We accounted for the slight difference in the extendible hashing's time complexity related to the block size B we chose. Nonetheless, there is a noticeable difference in performance when the Input size becomes so large that there is a need to grow the table size by many folds. This large input size causes the dictionary hash table to rehash. Thus the time requirements for each rehashing increase the time complexities of the dictionary hash table.

Secondly, in terms of space complexities, the logarithmic algorithm used less memory than the dictionary algorithm. This difference in space requirement is because the dictionary hash table loads all data in memory; meanwhile, the logarithmic algorithm indexes data on secondary storage.

Lastly, we observed that the constant time in insertion, deletion and search were salted by uncontrollable processes like thermal throttling. So we amortised that by experimenting over and over after a system cold boot. Another factor that might have influenced the experiment is that we could not restrain all the operating system resources to our experiment since the operating system automatically switches I/O to other programs for optimal performance of the whole system. In observing the space requirement in figure 4, we can predict a logarithmic growth in memory usage for the logarithmic hash table; meanwhile, there is a linear growth in memory usage for the dictionary hash table. Also, rehashing becomes very computationally costly as Input size increases.

To remedy this problem, we can introduce a bucket load factor (elastic bucket). After insertion, we can check a bucket load factor: say greater than 100% before any doubling of a directory.

13 Conclusion

The time and space complexities of the dictionary and logarithmic hash tables were as expected from their theoretical complexity. The deviation we encountered due to thermal throttling was handled by amortisation of results gotten after a cold reboot of the system. Furthermore, there was a slight increase in the dictionary hash table's performance in terms of insertion, search, and deletion for individual operations. However, this performance degraded as the input size increased along with every rehashing setting in making the logarithmic hash table overall faster.

References

[1] N. Askitis and J. Zobel, "Cache-conscious collision resolution in string hash tables," in *String Processing and Information Retrieval*, M. Consens and G. Navarro, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 91–102, ISBN: 978-3-540-32241-2.

[2] J. S. Vitter *et al.*, "Algorithms and data structures for external memory," *Foundations and Trends® in Theoretical Computer Science*, vol. 2, no. 4, pp. 305–474, 2008.

[3] P. Flajolet, "On the performance evaluation of extendible hashing and trie searching," *Acta Informatica*, vol. 20, no. 4, pp. 345–369, 1983.

[4] D. J. Lilja, *Measuring computer performance: a practitioner's guide*. Cambridge university press, 2005.

[5] R. Fagin, J. Nievergelt, N. Pippenger, and H Raymond Strong, "Extendible hashing - a fast access method for dynamic files.," *ACM Trans. Database Syst.*, vol. 4, pp. 315–344, Sep. 1979. DOI: 10.1145/320083.320092.

14 Appendix

A Extendible hashing Algorithm breakdown

Let us assume the size K of the range of the hash function hash is sufficiently large. For a given $d \geq 0$ the directory consists of a table (array) of 2^d pointers. We assign Each item a bucket corresponding to the d least significant bits of its hash address. The value of d, called the global depth(the number of least important bits of the binary representation of a hash), is set to the smallest value for which each bucket has the most B items assigned to it. A lookup takes two I/Os: one to access the directory and one to access the bucket storing the item. If the directory fits in internal memory, only one I/O is needed

`Seach(n):`

1. We calculate $n' = h(G,n)$. This read G the and take G initial bits form n (after conversion of n to base 2).
2. n' is a now a G bits base 2 number.
3. We find the pointer located at index position n' base 10 and follow the link to the corresponding bucket. If n found return success.
4. If n not found in the Bucket, we compute the address of the Buddy bucket. Repeat this if n not found in buddy bucket or return failure if no buddy bucket.

`Insert(n):`

5. Repeat step 1 to 4 and if the return value is false.
6. We find the pointer located at index position n' base 10 and store n in the bucket it is a reference to.
7. We check for bucket splitting, if yes there are 2 possible option to carry out.
   - If the Local depth of the overflow bucket equal the global depth, we double the Directories size, increase the global depth by 1 and update all the new directories to point to their buddy buckets. Then, we rehash only the entries of the overflow bucket.
   - If the local depth of the overflow bucket is less than the global depth,(1 timeless, meaning two directories are pointing to the bucket, 2 times less means three directories are pointing to the bucket and so on...) we create $globaldepth - localdepth$ number of new bucket(s), make the other pointers to the old bucket point to the new bucket(s), rehash only the overflow bucket and increase the local depth by 1.

`Delete(n):`

8. Repeat 1 to 4 and if the return value is false then, nothing to delete. Else if the result is true, we replace n in the bucket by the last item in the bucket. This overwrites n hence deleting it.
9. If n is the only entry in the bucket, we delete the bucket and make the directory point to its buddy bucket. Then, decrease the local depth of the buddy bucket by 1.

B Input Size variation

The Input size was varied as follows in a terminal window.

```
$ for (( inputSize = 10000; inputSize <= 163840000 ; inputSize *= 2))
 do
 ./driverProgram -s 5000000 -l $inputSize
 done
```

Input data was generated using a basic function (with prototype: `unsigned long floodCount(bool)`)) holding a static variable which returns the value of that static variable and increments count. If the function is called with a false as arguments, it reset the count to 0. `-s` specifies the initial table size, and `-l` specifies the input size into the hash table.

Input data was generated using a basic function (with prototype: `unsigned long floodCount(bool)`) holding a static variable which returns the value of that static variable and increments count. If the function is called with a false as arguments it reset count to 0. `-s` specifies the initial table size and `-l` specifies the input size into the hash table.

C Data generating function

The input data was generated by `floodCount`.

```
unsigned long floodCount(bool reset){
 static long count = 0;
if(reset)
return ++count;
else
return count = 0;
}
```

We decided to use this function to generate our data instead of the UNIX `rand()` function because we can ensure that no redundant input was generated; thus, waiting time was reduced. Hence, when we decide to generate the N input size, we can guarantee that N input data was inserted into the hash table. Also, it is faster compared to the `rand()` function, which scales its output data using a modulo operator.

D Dependent Variables measurement

Resource usage of the functions are calculated using the `getrusage()` function as follows: let us say we want to calculate resource usage for the `insert()` function. First, we initialise two getrusage variables

```
struct rusage before, after;
```

Then, we enclose our function call to insert into two getrusage usage functions.

```
getrusage(RUSAGE_SELF, &before);
int isSuccess = insert(number);
getrusage(RUSAGE_SELF, &after);
```

The time resource used by the insertion function is saved in the variables `before` and `after`. We get an estimate of the system and user time by getting the difference of the variable. See getrusage Manual for more detail.

Glossary

Bucket	They are a fixed size array into which entries are stored. Directories point to buckets. More than one directory can point to a bucket if its local depth is less than the global depth.
bucket splitting	When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
Buddy bucket	This is a bucket referenced by the directory entry which shares the last (local depth - 1) bits. For instance, the buckets referenced by direction entries 1111 and 0111 are buddy buckets.
Directories	This is a list(array) of pointers pointing to buckets. Every index in this list has a unique id which may change each time when expansion takes place. The hash function hashes each entry to a directory id which is used to navigate to the appropriate bucket.
In-memory	Access technique where the hash table is store in the RAM.
Load factor	This is the ratio of the number of entries hash into the hash table to the key-space.
Local depth	It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.