

CCDSALG/GDDASGO Term 1, AY 2024 – 2025  
Project 2 Documentation –Application of Hash Table Data Structure

**DECLARATION OF INTELLECTUAL HONESTY / ORIGINAL WORK**

*We declare that the project that we are submitting is the product of our own work. No part of our work was copied from any source, and that no part was shared with another person outside of our group. We also declare that each member cooperated and contributed to the project as indicated in the table below.*

Section	Names and Signatures	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6
<S12>	Balbastro, Lianne Maxene	X	X	X	X		
<S12>	Duelas, Charles Kevin	X	X	X	X		
<S12>	Choa, Regan			X	X		

Fill-up the table above. For the tasks, put an 'X' or check mark if you have performed the specified task. Please refer to the project specifications for the description of each task. **Don't forget to affix your e-signature after your first name.**

1. FILE SUBMISSION CHECKLIST: put a check mark as specified in the 3<sup>rd</sup> column of the table below. Please make sure that you use the same file names and that you encoded the appropriate file contents.

FILE	DESCRIPTION	Put a check mark ✓ below to indicate that you submitted a required file
hash.h	header file for hash function, etc.	✓
hash.c	C source file for hash function, etc.	✓
main.c	main module	✓
INPUT1.TXT INPUT5.TXT	to 5 sample input files (with increasing values of n)	✓
OUTPUT1.TXT OUTPUT5.TXT	to 5 sample corresponding output files	✓
GROUPNUMBER.PDF	The PDF file of this document	✓

2. Indicate how to compile your source files, and how to RUN your exe files from the COMMAND LINE. Examples are shown below highlighted in yellow. Replace them accordingly. Make sure that all your group members test what you typed below because I will follow them verbatim. I will initially test your solution using a sample input text file that you submitted. Thereafter, I will run it again using my own test data:

- How to compile from the command line  
C:\MCO> gcc -Wall main.c hash.c -o main.exe
- How to run from command line  
C:\MCO>main.exe INPUT1.txt OUTPUT1.txt

Next, answer the following questions:

- Is there a compilation (syntax error) in your codes? (YES or NO). NO  
**WARNING: the project will automatically be graded with a score of 0 if there is syntax error in any of the submitted source code files. Please make sure that your submission does not have a syntax error.**
- Is there any compilation warning in your codes? (YES or NO) NO

**WARNING: there will be a 1 point deduction for every unique compiler warning. Please make sure that your submission does not have a compiler warning.**

3. Please indicate if you created your own original hash function or if you used a hash function from some reference material: We created our original hash function

If you created your original hash function: affix your signature on the given space


MCO2 is described as follows:

The hashing function converts the string into a numerical index for the hash table. The function starts by iterating over each character and performing the hashing for each  $i$  (the character position) in the provided string. The current hash\_value is multiplied by the prime\_multiplier (37). This is done since prime numbers help create a more uniform distribution of hash values that reduces collisions. Other hash functions adopt this practice based on our observation, the DJB2 Hash Function, SDBM Hash Function, Knuth's Multiplicative Hash, and the FNV Hash Function. Then, the ASCII value of the character is adjusted by its position in the string [ $\text{input}[i] * (i + 1)$ ] is added to the hash. This ensures that the character's position in the string directly influences the hash. It also reduces the chances of collisions between similar strings.

The computed hash value is stored in hash\_value. Once it is stored properly, the final value is reduced using the modulo operator to ensure that the resulting hash value is within the bounds of the hash table.

Name and Signature:  Lianne Maxene D. Balastro

Name and Signature:  Charles Kevin C. Duelas

  
Name and Signature: Jon Regan S. Choa

4. Specify what collision resolution technique you implemented in your MCO2 (i.e., it is linear probing, quadratic probing or double hashing)? Linear Probing

5. Disclose **IN DETAIL** what is/are NOT working correctly in your solution. **Please be honest about this. NON-DISCLOSURE will result in severe point deduction.** Explain briefly the reason why your group was not able to make it work.

All Features in the Major Course Output were followed based on the specifications and fully functional.

6. Based on the exhaustive testing that you did, fill-up the Table below. Test for at least 5 different values of  $n$ , starting with  $n = 2^6 = 32$  points. Set the values of  $n$  such that they are expressed using 2 as exponent. Your last test case should have  $n = 2^{14} = 16384$  strings.

**Table: Statistics For the 5 Test Cases**

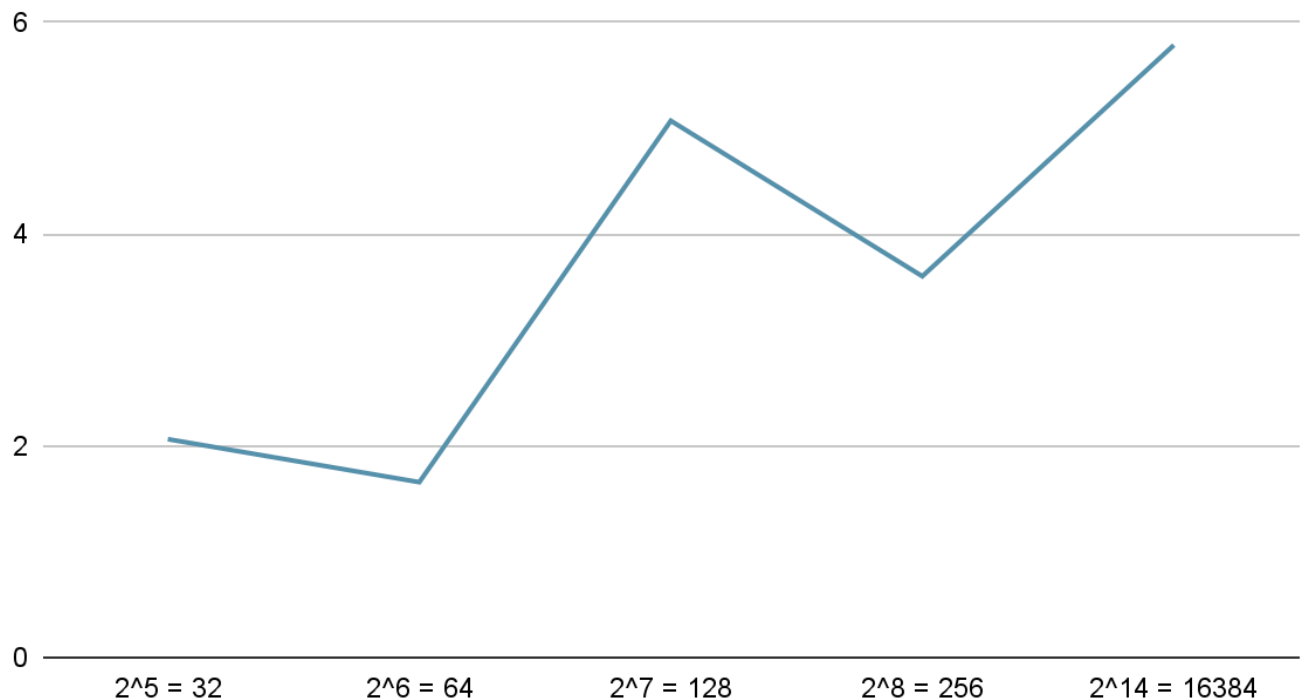
Test Case #	$n$ (input size)	# of keys/strings stored in the hash table	# of keys/strings stored in their home addresses	# of keys/ strings NOT stored in their home addresses	Average number of string comparisons
1	$2^5 = 32$	32	20	12	2.062500
2	$2^6 = 64$	64	53	11	1.656250

3	$2^7 = 128$	128	68	60	5.070313
4	$2^8 = 256$	256	136	120	3.601563
5	$2^{14} = 16384$	16384	8949	7435	5.784607

**NOTE: Make sure that you fill-up the table properly. It contributes 3 out of 15 points for the Documentation.**

7. Create a line graph (for example using Excel) based on the Comparison Table that you filled-up above. The x-axis should be the values of  $n$  and the y axis should be the *average number of string comparisons*. Copy/paste an image of the graph below.

### The Average Number of String Comparisons



**NOTE: Make sure that you provide a graph based on your comparison table data above. It contributes 3 out of 15 points for the Documentation.**

8. Analysis – answer the following questions:

a. Based on the statistics above, is your Search() operation using a hash table slower or faster, on average, compared with linear search on an array? Do not simply answer with “slower” or “faster”. You need to provide some explanation to support your answer.

Our Search() operation using a hash table is faster on average than linear search on an array. This is because the growth rate of hash table search is  $O(1)$ , while linear search has a growth rate of  $O(n)$ . Linear search requires checking each index one by one until the desired key is found, making it inefficient for large input sizes. For an input size of 16384, the average number of comparisons for our hash table search is only 5.784607. In contrast, the worst-case scenario for linear search would require 16384 comparisons, as it may traverse the entire array. Even with collisions and clustering in the hash table, the number of comparisons remains far lower than linear search. This demonstrates that hash table search is significantly more efficient than linear search for large datasets.

b. Based on the statistics above, is your Search() operation using a hash table slower or faster, on average, compared with binary search on an array? Do not simply answer with “slower” or “faster”. You need to provide some explanation to support your answer.

Our Search() function using a hash table is faster on average than binary search on an array. The growth rate of the hash table search is  $O(1)$ , while binary search has a growth rate of  $O(\log n)$ . For an input size of  $n = 16384$ , binary search requires  $(\log_2 16384) = 14$  comparisons in the worst case. In contrast, the hash table search required an average of only 5.784607 comparisons, even with collisions. The hash table's collision resolution mechanism ensures that the average number of comparisons remains far lower than the logarithmic growth of binary search. While binary search performance depends on the size of the array, hash table search performance is largely independent of input size. As demonstrated by the test cases, this makes the hash table search significantly faster than binary search for large inputs.

**NOTE: Make sure that you provide cohesive answers to the questions above. This part contributes 3 out of 15 points for the Documentation.**

9. Fill-up the table below. Refer to the rubric in the project specs. It is suggested that you do an individual self-assessment first. Thereafter, compute the average evaluation for your group, and encode it below.

REQUIREMENT	AVE. OF SELF-ASSESSMENT
1. Hash function	<u>30</u> (max. 30 points)
2. Collision resolution function	<u>10</u> (max. 10 points)
3. Search function	<u>15</u> (max. 15 points)
4. Main module	<u>15</u> (max. 15 points)
5. Input and output text files	<u>10</u> (max. 10 points)
5. Documentation	<u>15</u> (max. 15 points)
6. Compliance with Instructions	<u>5</u> (max. 5 points)

TOTAL SCORE 100 over 100.

**NOTE: The evaluation that the instructor will give is not necessarily going to be the same as what you indicated above. The self-assessment serves primarily as a guide.**