CCDSALG/GDDASGO Term 1, AY 2024 – 2025

Project 2 – Hash Table (Application of Hashing Functions and Collision Resolution Techniques)

Groupings : At most 3 members in a group
Deadline : Refer to the Canvas Submission Page
Percentage : 20% of the Final Grade
Programming Language: C

Let's apply what you learned in hashing functions and collision resolution techniques.

For this project, you will implement a program that will store MANY strings as key values into a hash table data structure. Assume in this project that a string is made up only of lowercase and uppercase letters , and is case-sensitive, meaning "dog" is different from "DOG" and "doG". There are no digits, no symbols and no white spaces. Set the maximum string length to 15 characters (excluding the null byte). The strings need not be words from English. It can be something without any meaning, for example, "xyZazL".

Discuss and decide with your groupmates TWO important design issues:

1. What hashing functions will you use? *Do not limit your choice to a simple modulus k % m; explore possible different ideas. The objective is to find a hashing function that will produce as much as possible the fewest number of collisions.*
2. What collision resolution technique will you use?

I. HASH TABLE (DATA STRUCTURE)

The Hash Table is the data structure to be used in storing the key values, i.e., strings. Recall that a hash table will allow us to search a key in O(1) if there are no collisions.

Implement the hash table as a 1D array of strings. Note that your hash table size can be less but cannot be more than the 1D array size. Just like in the previous project, we will stress test your solution.

- Make sure that your hash table can store a maximum of *n = 2^14 = 16384* strings.
- The **hash table size** should be set such that it is equal to the first prime number immediately above **1.1 * n**.
    - Example #1: when *n = 2^5 = 64*, the first prime immediately above *1.1 * 64* is *71*. Thus, the hash table to be used is 71. This means that you can only use 1D array indices 0 to 70 for storing the keys/strings. Keys should not be stored beyond array index 70.
    - Example #2: when *n = 2^14 = 16384*, the first prime number immediately above *1.1 * 16384* is *18031*. This means that you can only use 1D array indices 0 to 18030 for storing the keys/strings. Keys should not be stored beyond array index 18030.

II. INPUT

The input data is encoded in a text file. The first line contains a number say ***n*** indicating the number of strings encoded in the text file. It is then followed by ***n*** strings. There can be any number of white spaces and newline characters separating the strings in the text file. An example content of an input text file is shown below where there are *n = 16* strings, the first being "THE", and the last being "ne".

```
16
    THE     quick
BroWn

fox
```

```
   jumps OVER          THE                                                  lazy

        Dog

xyZazL  kore wa saigou                          desu   ka ne
```

IMPORTANT NOTES:

a. A string may appear more than once in the input text file. For example, "THE" appears twice as shown in the yellow and green color background.
b. If a string was already stored in the hash table in some previous occasion, a subsequent appearance of the same string should not be stored in the hash table. This means that the hash table contains UNIQUE strings only. For example, the string "THE" (in yellow) will be stored in the hash table. The subsequent appearance of the same string "THE" (in green) will not be stored in the hash table since there is already an existing entry.

III. OUTPUT

The output is another text file containing the following:

a. The 1st indicates the number of strings that were read from the input text file.
b. The 2nd line indicates the number of strings that were stored in the hash table. Using the example input text file data above, the first line should contain 16, and the second should contain 15 (since the second appearance of "THE" was not stored).
c. The 3rd line indicates the number of keys that were stored in their home addresses (i.e., did not cause a collision).
d. The 4th line indicates the number of keys that were NOT stored in their home addresses (i.e., collision resolution had to be applied).
e. The 5th line indicates the average number of string comparisons for the **`Search()`** operation (refer also to the next paragraph Column 5 information). This average value is computed as

average = total number of string comparisons / number of strings that were stored in the table

Make sure to write the average value with 6 digits after the decimal point.

Thereafter, the succeeding lines of output text should be made up of FIVE columns of data.

a. Column 1 indicates the indices of the hash table starting from 0 to the last index.
b. Column 2 indicates the string that is stored in that index.
c. Column 3 indicates the home address of the string.
d. Column 4 indicates either a YES or NO. A YES should be displayed if the home address is the same as the index where it was stored, otherwise a NO should be displayed (meaning, there was a collision, and a collision resolution technique was applied).
e. Column 5 indicates the number of string comparisons made when a **`Search()`** operation is done using the string as the search key. Logically, a string that was stored in its home address will require only 1 comparison, while a string with a collision will require more than 1 comparison.

In case there is no string stored in a certain index (i.e., the memory space is unoccupied), columns 2 to 5 should display a string made up of 3 dashes (same as minus symbol), i.e., "---".

A hypothetical example output text file contents are shown below using the strings from the example input text file with n = 2^4 = 16 strings. The hash table size is 19 which is the nearest prime number immediately after 2^4 = 16. Thus, the

hash table indices are from 0 to 18. For reasons of brevity, the colons represent other entries that we do not show anymore.

Line 1 means that 16 strings were read from the text file. Line 2 means that 15 strings were stored in the hash table. Line 3 means that 12 strings were stored in their respective home addresses. Line 4 means 3 strings were not stored in their home addresses. Line 5 means 1.466667 is the average number of string comparisons for the **`Search()`** operation (computed, for example, as 22/15 = 1.466667). The succeeding lines indicate that the strings "jumps", "OVER" and "THE" were stored in their respective home addresses, while "xyZazL" was not stored in its home address of 11, but was stored instead at index 1. The last column shows a 1 for the strings that were stored in their home addresses. For the string "xyZazL" which was not stored in its home address, 3 string comparisons were needed to Search() it in the hash table. The three dashes **---** indicate that indices 2, 4 and 18 are empty (does not store any key/string) .

```
16
15
12
3
1.466667

0  jumps      0    YES    1
1  xyZazL    11    NO     3
2  ---      ---    ---  ---
3  OVER       3    YES    1
4  ---      ---    ---  ---
:
:
11 THE       11    YES    1
:
:
18 --- --- --- --- ---
```

IV. TASKS

Task 1. Implement your (a) hash function, (b) collision resolution function and (c) Search() function.

- Design and implement your hash function. The hash function should compute and return an index representing the home address of the input string. It should NOT store the string yet in the hash table. Encode your function in a source file named as **hash.c**.
- Design and implement your collision resolution function. It should NOT store the string yet in the hash table. The collision resolution function should return an index where the input string (synonym) should be stored. Encode your function also in the source file **hash.c.**
- Implement your Search() function to search a key/string if it is in the hash table or not. It is dependent on the hash function and collision resolution solution function that you defined. The Search() function should return the index where the key was found, otherwise it should return a -1. Encode this function also in hash.c.
- You may define helper functions for hashing and collision resolution also in the same source file **hash.c.**
- Make sure that ALL the functions that you defined in **hash.c** are WELL-DOCUMENTED in the form of program comments. Point deductions will be applied if there is zero to little inline documentation.
- Create a header file named **hash.h**. The header file should contain at least the following
    - the #define for the MAXIMUM 1D ARRAY of string size
    - the prototypes for the hash function, collision resolution function, Search() function, and helper functions

Task 2. Implement a driver program to test your hash function, collision resolution function and Search() function.

- The driver program should have a **`main()`** function, and if you deem necessary, a number of helper functions that will do the following sequence of steps:
  1. Ask the user to enter the name of the input text file. If the file does not exist terminate the program immediately and provide an appropriate error message.
  2. If the file exists, open and read the input text file containing the *n* strings.
  3. Store the strings in the hash table (if there is no previous entry of the same string) – this means that you have to call the hash function, and if collision occurs, the collision resolution function.
  4. Ater processing all the input strings, ask the user to enter the name of the output text file.
  5. Write the required contents to the output text file.
  6. Close the input and output text files.
- The driver program should be encoded in a file named as **main.c**.

Task 3. Test your solution exhaustively.

- Create at least 5 test case text files with different values of *n* starting with *n = 2^5 = 64*. Set the values of *n* such that they are expressed using 2 as exponent (for example *n = 2^8 = 256*). Your fifth test case file should have *n = 2^14 = 16384* random strings.
- You may create your own program for generating the random strings, or use random string generators available on the internet. You may also explore the use of AI tools such ChatGPT or Copilot for this task.
- Your solution should produce the corresponding 5 output text files.

Task 4. Document your implementation and the test results.

- Give a brief description of how you implemented the hash table.
- Describe in a concise manner your hash function.
- Describe in a concise manner your collision resolution technique.
- Disclose in detail what is not working in your submission.
- Tabulate the results obtained based on your 5 test case text files.

V. DELIVERABLES

Submit via Canvas a ZIP file named GROUPNUMBER.ZIP (for example 01.ZIP for group number 1) which contains:

- Source files (.h and .C files) of the modules as specified above.
- 5 input text files (that you used in testing).
- Corresponding 5 output text files.
- Documentation named GROUPNUMBER.PDF (for example 01.PDF for group number 1). Use the accompanying documentation template DOCX file.

Do NOT include any EXEcutable file in your submission.

VI. RUBRIC FOR GRADING

Criteria	RATINGS		
1. Hash function	**YOUR OWN ORIGINAL IDEA [30 pts]** You honestly declare and swear that the entire hash function is your own original idea.	**BORROWED (NOT ORIGINAL) IDEA [20 pts]** The hash function is borrowed from some reference that you have to explicitly specify. For example, "the basis of our hash function is described in <reference material>".	**NO MARKS [0 pt]** Not implemented or completely incorrect.

	Or in case you used the modulus method, i.e., k % m, you honestly declare and swear that the computation of k corresponding to a given string is your own original idea. In either case, your original idea must be well documented in the source file **hash.c** and in the PDF documentation file. The hash function is working correctly.	The hash function is working correctly.	
2.Collision resolution function	**COMPLETE [10 pts]** The collision resolution function is working correctly.	**INCOMPLETE [1 to 5 pts]** Implementation is not entirely correct.	**NO MARKS [0 pt]** Not implemented or completely incorrect.
3. Search function	**COMPLETE [15 pts]** The implementation is based on the hash function and collision resolution function and is working correctly .	**INCOMPLETE [1 to 10 pts]** Implementation is not entirely correct.	**NO MARKS [0 pt]** Not implemented or completely incorrect.
4. Main module	**COMPLETE [15 pts]** The main module is working correctly.	**INCOMPLETE [1 to 10 pts]** Implementation is not entirely correct.	**NO MARKS [0 pt]** Not implemented or completely incorrect.
5. Input and Output Text Files	**COMPLETE [10 pts]** Submitted all the 10 input and output text files; compliant with the specifications.	**INCOMPLETE [less than 10 pts]** -1 pt for every missing or non-compliant text file	**NO MARKS [0 pt]** No submission.
6. Documentation and Analysis	**COMPLETE [15 pts]** Required details were covered. Take note of the following point distributions: a. Hash function and collision resolution [2 pts] b. Table of Results [3 pts] c. Graph of results [3 pts] D. Analysis [3 pts] d. Remainder of the document [4 points]	**INCOMPLETE [1 to 10]** Some required details were not discussed.	**NO MARKS [0 pt]** No documentation.
7. Compliance with Instructions	**COMPLIANT [5 pts]** All instructions were complied with.	**NON-COMPLIANCE [0 to 4]** Deduction of 1 point for every instruction not properly complied	**NO MARKS [0 pt]**

		with. Examples: included an EXE file in the ZIP file, filenaming not followed.	More than 4 instructions not complied with.
			Maximum Total Points: 100

VII. WORKING WITH GROUPMATES

You are to accomplish this project in collaboration with your fellow students. Form a group of at least 2 to at most 3 members. The group may be composed of students from different sections. Make sure that each member of the group has approximately the same amount of contribution for the project. Problems with groupmates must be discussed internally within the group, and if needed, with the lecturer.

VIII. HONESTY POLICY AND INTELLECTUAL PROPERTY RIGHTS

Honest policy applies. You are encouraged to read and gather information you need to implement the project. **However, please take note that you are NOT allowed to copy-and-paste -- in full or in part any existing related program code from the internet or other sources (such as printed materials like books, or source codes by other people that are not online). You should develop your own codes from scratch by yourselves, i.e., in cooperation with your groupmates.** According to the handbook (5.2.4.2), *"faculty members have the right to demand the presentation of a student's ID, to give a grade of 0.0, and to deny admission to class of any student caught cheating under Sec. 5.3.1.1 to Sec. 5.3.1.1.6. The student should immediately be informed of his/her grade and barred from further attending his/her classes."*

Question? Please post it in the Canvas Discussion thread. Thank you for your cooperation.

サルバドール・フロランテ