Offline: Hash Table

Problem Specification

In this assignment, you have to implement a ***Hash Table*** with the following requirements.

1. You have to implement insert, search, and delete operations on the hash table. The length *N* of the hash table will be given as an input. (You can assume that *N* will always be a prime number)

2. Data will be kept as a (key, value) pair in the hash table. You need to insert randomly generated 7-character-long unique words into the hash table (these words can be meaningful or meaningless). Therefore, you have to implement a random word generator. The words will be used as the “key", and the order of the incoming words will be used as the “value" (please see the example below). If your word generator produces duplicate words, you must keep only one instance of each.

Example

Suppose your word generator has generated the following 5 words.

ancient
pazzled
benefit
ancient
zigzags

The corresponding (key, value) pairs will be:

(ancient, 1)
(pazzled, 2)
(benefit, 3)
(zigzags, 4)

Note that the 2nd instance of the word “ancient" has been discarded.

Hash Function

Use two hash functions ($h_1(k)$ and $h_2(k)$) of your own, or from any good literature where you must try to avoid collisions as much as possible. We expect that at least 60% of the keys will have unique hash values (i.e., at least 60 unique hash values for 100 keys). **Write a function to verify this by randomly generating 100 words and then counting the number of unique hash values produced by these 100 words for a given value of *N*.** Moreover, design h_2 in such a way that it always returns a positive integer less than N. We will use it in double hashing (details later). h_1 and h_2 should be substantially different so that they don't show similar behavior.

Collision Resolution

You need to implement the following four collision resolution methods.

1. **Separate Chaining**

Place all the elements that hash to the same slot into a linked list. Slot j contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains *NULL*. Use h_1 (mentioned in the previous section) as the hash function. Here use your own implementation of linked list as the built-in implementation provided by the programming language has a lot of overhead.

2. **Linear Probing**

Use the following hash function.

$$h(k, i) = (h_1(k)+i) \bmod N$$

$$\text{for } i = 0,1, \ldots, N-1$$

Here $h_1(k)$ is the hash function mentioned in the previous section.

3. **Quadratic Probing**

Use the following hash function.

$$h(k, i) = (h_1(k) + c_1 i + c_2 i^2) \bmod N$$

where c_1 and c_2 are positive auxiliary constants, and $i = 0,1, \ldots, N-1$

Here $h_1(k)$ is the hash function mentioned in the previous section. Choose appropriate values for c_1 and c_2 (by yourself or from any good literature).

4. **Double Hashing**

Use the following hash function.

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod N$$

Use the hash functions h_1 and h_2 mentioned in the previous section. Think of when to stop probing here.

Carefully handle key deletion in open-addressing so that insertion and search give correct results after a key has been deleted.

Report Generation

For a given value of N as input, increase load factor from 0.4 to 0.9 at an interval of 0.1. Insert appropriate number of elements according to the load factor in each case. **Before starting insertion to the hash table, generate all the words** (to ensure that insertion process doesn't get affected by word generation). Randomly select 10% of the inserted elements and search these elements (say we are searching for a total of *p* elements here). Report the **average search time** (both in separate chaining and open addressing) and the **average number of probes** (only in open addressing). Now delete 10% of the inserted elements randomly and then again search for a total of *p* elements, but this time *p/2* elements among them should be the deleted elements (so the other p/2 elements are still in the hash table). Report the average search time and the average number of probes again, just as before.

Do the above task for all the four collision resolution methods separately.

Table 1: Performance of separate chaining in various load factors

	Before deletion	**After deletion**
Load factor	**Avg search time**	**Avg search time**
0.4		
0.5		
0.6		
0.7		
0.8		
0.9		

Table 2: Performance of linear probing in various load factors

	Before deletion		**After deletion**	
Load factor	**Avg search time**	**Avg number of probes**	**Avg search time**	**Avg number of probes**
0.4				
0.5				
0.6				
0.7				
0.8				
0.9				

(Do the same for quadratic probing and double hashing)

Table 5: Performance of various collision resolution methods in load factor 0.4

	Before deletion		**After deletion**	
Method	**Avg search time**	**Avg number of probes**	**Avg search time**	**Avg number of probes**
Separate chaining		N/A		N/A
Linear Probing				
Quadratic Probing				
Double Hashing				

(Do the same for other load factors 0.5, 0.6, 0.7, 0.8, 0.9)

The report can be shown in the console or generated in a file.

Submission

Include only source files

Do not include executable binaries, input/output files

Place your files in a folder named 1905XXX

Zip the folder

Submit to Moodle after renaming it to 1905XXX.zip

Deadline: 02/08/2022, 11:55 pm

Start working early so that you don't need to hurry at the last moment.

General Guidelines

Write readable, re-usable, well-structured, modular code. This includes but is not limited to writing appropriate functions for implementation of the required algorithms, meaningful naming of the variables, suitable comments where required, proper indentation etc.

Please **DO NOT COPY** solutions from anywhere (your friends, seniors, internet etc.). Implement the algorithms with your style of coding. Any form of plagiarism (irrespective of source or destination), will result in getting -100% marks. You have to protect your code.