Data Structures and Algorithms – Assignment 3: Hashing

The Programming Questions

There are 3 programming questions, the problem statements, input/output formats and examples for which you shall find on the pages 4 to 7 of this document.

The following are the titles of each question and their corresponding question numbers.

- Question 1: Hash Tables
- Question 2: Spell Checker
- Question 3: String Searches

Analysis and README files

Question 1 - Hash Table Load Analysis

Write the results of the following in the **README file for Question 1**.

Comparison of Collision Resolution methods

For all three implementations:

1. Create an empty hash table of size just above 2000 for each of the 3 collision-resolution algorithms.
2. Change the definition of each node of your hashtable to include a garbage array to add memory load, as shown in the struct below. (Further clarification on this, if need be, will be posted on Moodle later, please ask any doubts regarding this only on Moodle)

```
struct Node {
    Element n;
    int garbage[1000000]; // Dummy array with 1m elements
}
```

3. Generate random 2000 even numbers (they could be any even integer, not necessarily in $1 - 2000$) in random order.
4. Insert them in the three tables.
5. Find out average number of probes required to insert in each of the table when alpha fraction of table is full for $\alpha \in \{0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9\}$ by running steps 2 and 3 for m times.
6. Note for quadratic probing, you need to rehash when α becomes 0.5.
7. Also study the time required to insert 2000 elements in the table in all three implementations and write your observations.

Sign-Alternating Quadratic Probing (Optional)

You know the original quadratic probing algorithm. Implement a new collision resolution method, which if the desired position k is non-empty, check the position $k+1, k-1, k+4, k-4, k+9, k-9$ and so on. Check if it reduces the average number of probes.

Question 2 – Horner's rule

Write the results of the following in the **README file for Question 2**.

Create a hash table to store the above dictionary using separate chaining and hash function as Horner's rule with $p = 33$ and $p = 32$. Find out the average number of collisions when you use $p = 33$ and when you use $p = 32$.

Submission Formats

Hash Table Implementations

Here you will implement `hash.c` and `hash.h` which you will use across all 3 questions.

Ensure that you make your interface well enough that all 3 hashing formats (separate chaining, linear probing, quadratic probing) that you setup in this question can be used across all 3 questions. Implement all your hash functions here, which you may need across the different problems (e.g., Identity Modulus hash, Polynomial hash, etc.). Do not print or take inputs in your `hash.c` or `hash.h` files.

Code for Individual Questions

All the code specific to each question will go into `main.c`, in the folder for that question.

Overall Structure

The following should be your final folder structure.

`<roll_number>/`

- `hash.c`
- `hash.h`
- `q1/`
    - `main.c`
    - `README.md`
- `q2/`
    - `main.c`
    - `README.md`
- `q3/`
    - `main.c`
- `dist/`
    - `dictionary.txt`
    - `submitter.py`
    - `… (other files provided in distribution, if any)`

Please do not tamper with any of the distribution files.

How to Submit + Distribution Code

Online Judge Submission

You need to submit your code to the Online Judge (OJ).

Use the following command to concatenate your files, from inside the directory <roll_number>/:

- For Question 1: `cat hash.c hash.h q1/main.c > q1/submission.c`
- For Question 2: `cat hash.c hash.h q2/main.c > q2/submission.c`
- For Question 3: `cat hash.c hash.h q3/main.c > q3/submission.c`

Submit the respective `submission.c` files to the online judge.

*Tip: Compile your programs with gcc *.c -fsanitize=address -fsanitize=undefined to check for undefined behavior.*

Moodle Submission

Unzip the distribution files that we have attached on this Moodle post and save the dist folder as indicated in the structure above.

Before submitting, please run:

```
python3 <roll_number>/dist/submitter.py
```

This will ensure that your file structure is correct, run the basic test cases for all 3 questions, thereby checking that your input output format is correct.

For zipping your files (if you choose to do it manually, otherwise the auto-submitter zips it for you and puts the file in the root directory), use:

```
zip -r <roll_number>.zip <roll_number>/
```

If you do not have python installed, please run:

```
sudo apt install python3
```

In addition to the submitter, we have also given you the dictionary text file that you should use to prepare your report for Question 2.

In case of any queries, try to make a Moodle post. If you have specific queries that need direct conversation, please contact the Teaching Assistants for this assignment: Animesh Sinha and Divanshi Gupta.

Hash Tables

Input file:	`standard input`
Output file:	`standard output`
Time limit:	1 second
Memory limit:	256 megabytes

Let's begin by implementing a hash table. Here you would be using it as a set, trying to retrieve elements that you have stored. Following are the methods by which we plan to resolve clashes:

- Separate Chaining
- Linear Probing
- Quadratic Probing

You need to handle two types of queries:

- `+ x` - To insert an element into the hash-table.
- `? x` - To check if an element exists in the hash table.

You must use the identity hash function, i.e. $hash(x) = x\%N$. You also need to give us the index where you find each of these entries when searching for them.

Input

Each test will start with a mode string, which will be one of "separate-chaining "linear-probing or "quadratic-probing". The next line will contain an integer N, the size of the hash table. The next line will contain the number of queries Q. Each of the following Q lines will be of the form `? x` or `+ x`, where $1 \leq x \leq 100000$ is an integer.

Output

For each `? x` operation, print the index you found the element at if you found it, and -1 if you didn't.

For linear and quadratic probing, this will be the position where you ended up inserting the element. For separate chaining, print the position at which it's present in your linked list.

Examples

standard input	standard output
linear-probing 10 12 + 1 + 5 ? 1 + 15 + 16 + 25 + 17 ? 7 ? 16 ? 25 + 8 ? 8	1 -1 7 8 0
quadratic-probing 10 12 + 1 + 5 ? 1 + 15 + 16 + 25 + 17 ? 7 ? 16 ? 25 + 8 ? 8	1 -1 7 9 2
separate-chaining 10 12 + 1 + 5 ? 1 + 15 + 16 + 25 + 17 ? 7 ? 16 ? 25 + 8 ? 8	0 -1 0 2 0

Spell Checker

Input file:	`standard input`
Output file:	`standard output`
Time limit:	1 second
Memory limit:	256 megabytes

Finally you get to put your hash tables to good use. You will be given huge amounts of text, and you have your dictionary words. You need to print out all the words in the text which do not appear in the dictionary. You must try to ensure that your spellchecker runs at the fastest speed possible, because the text is really huge.

Input

The first line will contain an integer $N(\leq 100000)$, which is the number of words in the dictionary. Each of the following N lines will have a word from the dictionary.

There will a newline after this.

The next line will have M, the number of words in the final text.

The remainder of the file will consist of M space-deliniated words forming the text that you need to spellcheck. The entire text will not consist of more than 1000000 characters. No word will be more than 32 letters in length.

Output

Print W, the number of words that were spelt incorrectly (not present in the dictionary). Then print W lines each with a word which was misspelt. If the same misspelling occurs multiple times, print all of them. The words should be sorted in the order that they appear in the text.

Example

standard input	standard output
4 is a cat rat 6 The cat is chsaing a mouse	3 The chsaing mouse

String Searches

Input file:	`standard input`
Output file:	`standard output`
Time limit:	1 second
Memory limit:	256 megabytes

Two friends - A and B want to know their friendship index. Since they are busy doing their DSA assignments they need your help!

Each friend has a DNA string which is provided in input. A string A is a good string for string B if string B can be generated by repetition of string A, integral number of times. Ex- "ab"is a good string of "ababab". The friendship index of two friends is the number of common good strings their DNA strings have.

Input

The first input line contains a non-empty DNA string for A. The second input line contains a non-empty DNA string for B.

Lengths of DNA strings are always positive and do not exceed 10^5. The DNA strings only consist of lowercase English alphabets.

Output

Print an integer representing the friendship-index of A and B.

Examples

standard input	standard output
zbcozbco zbcozbcozbcozbco	2
aaaaaa aaaaaaaaa	2
aba aaa	0