2 Problem: Hashing with chains

Problem Introduction

In this problem you will implement a hash table using the chaining scheme. Chaining is one of the most popular ways of implementing hash tables in practice. The hash table you will implement can be used to implement a phone book on your phone or to store the password table of your computer or web service (but don't forget to store hashes of passwords instead of the passwords themselves, or you will get hacked!).

Problem Description

Task. In this task your goal is to implement a hash table with lists chaining. You are already given the number of buckets m and the hash function. It is a polynomial hash function

$$h(S) = \left(\sum_{i=0}^{|S|-1} S[i]x^i \bmod p \right) \bmod m\,,$$

where $S[i]$ is the ASCII code of the i-th symbol of S, $p = 1\ 000\ 000\ 007$ and $x = 263$. Your program should support the following kinds of queries:

- **add** `string` — insert `string` into the table. If there is already such string in the hash table, then just ignore the query.
- **del** `string` — remove `string` from the table. If there is no such string in the hash table, then just ignore the query.
- **find** `string` — output "yes" or "no" (without quotes) depending on whether the table contains `string` or not.
- **check** i — output the content of the i-th list in the table. Use spaces to separate the elements of the list. **If i-th list is empty, output a blank line.**

When inserting a new string into a hash chain, you must insert it in the beginning of the chain.

Input Format. There is a single integer m in the first line — the number of buckets you should have. The next line contains the number of queries N. It's followed by N lines, each of them contains one query in the format described above.

Constraints. $1 \le N \le 10^5$; $\frac{N}{5} \le m \le N$. All the strings consist of latin letters. Each of them is non-empty and has length at most 15.

Output Format. Print the result of each of the **find** and **check** queries, one result per line, in the same order as these queries are given in the input.

Time Limits. `C`: 1 sec, `C++`: 1 sec, `Java`: 5 sec, `Python`: 7 sec. `C#`: 1.5 sec, `Haskell`: 2 sec, `JavaScript`: 7 sec, `Ruby`: 7 sec, `Scala`: 7 sec.

Memory Limit. 512Mb.

Sample 1.

Input:

```
5
12
add world
add HellO
check 4
find World
find world
del world
check 4
del HellO
add luck
add GooD
check 2
del good
```

Output:

```
HellO world
no
yes
HellO
GooD luck
```

Explanation:
The ASCII code of 'w' is 119, for 'o' it is 111, for 'r' it is 114, for 'l' it is 108, and for 'd' it is 100. Thus, $h(\text{“world"}) = (119 + 111 \times 263 + 114 \times 263^2 + 108 \times 263^3 + 100 \times 263^4 \bmod 1\,000\,000\,007) \bmod 5 = 4$. It turns out that the hash value of $HellO$ is also 4. Recall that we always insert in the beginning of the chain, so after adding “world" and then “HellO" in the same chain index 4, first goes “HellO" and then goes “world". Of course, “World" is not found, and “world" is found, because the strings are case-sensitive, and the codes of 'W' and 'w' are different. After deleting “world", only “HellO" is found in the chain 4. Similarly to “world" and “HellO", after adding “luck" and “GooD" to the same chain 2, first goes “GooD" and then “luck".

Sample 2.

Input:

```
4
8
add test
add test
find test
del test
find test
find Test
add Test
find Test
```

Output:

```
yes
no
no
yes
```

Explanation:
Adding “test" twice is the same as adding “test" once, so first **find** returns “yes". After del, “test" is

no longer in the hash table. First time **find** doesn't find "Test" because it was not added before, and strings are case-sensitive in this problem. Second time "Test" can be found, because it has just been added.

Sample 3.

Input:

```
3
12
check 0
find help
add help
add del
add add
find add
find del
del del
find del
check 0
check 1
check 2
```

Output:

```
no
yes
yes
no

add help
```

Explanation:
Note that you need to output a blank line when you handle an empty chain. Note that the strings stored in the hash table can coincide with the commands used to work with the hash table.

Starter Files

There are starter solutions only for C++, Java and Python3, and if you use other languages, you need to implement solution from scratch. Starter solutions read the input, do a full scan of the whole table to simulate each **find** operation and write the output. This naive simulation algorithm is too slow, so you need to implement the real hash table.

What to Do

Follow the explanations about the chaining scheme from the lectures. Remember to always insert new strings in the beginning of the chain. Remember to output a blank line when **check** operation is called on an empty chain.

Some hints based on the problems encountered by learners:

- Beware of integer overflow. Use `long long` type in C++ and `long` type in Java where appropriate. Take everything $\pmod p$ as soon as possible while computing something $\pmod p$, so that the numbers are always between 0 and $p - 1$.

- Beware of taking negative numbers $\pmod{p}$. In many programming languages, $(-2)\%5 \neq 3\%5$. Thus you can compute the same hash values for two strings, but when you compare them, they appear to be different. To avoid this issue, you can use such construct in the code: $x \leftarrow ((a\%p) + p)\%p$ instead of just $x \leftarrow a\%p$.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.