1 Phone book

Problem Introduction

In this problem you will implement a simple phone book manager.

Problem Description

Task. In this task your goal is to implement a simple phone book manager. It should be able to process the following types of user's queries:

- `add number name`. It means that the user adds a person with name `name` and phone number `number` to the phone book. If there exists a user with such number already, then your manager has to overwrite the corresponding name.
- `del number`. It means that the manager should erase a person with number `number` from the phone book. If there is no such person, then it should just ignore the query.
- `find number`. It means that the user looks for a person with phone number `number`. The manager should reply with the appropriate name, or with string "not found" (without quotes) if there is no such person in the book.

Input Format. There is a single integer N in the first line – the number of queries. It's followed by N lines, each of them contains one query in the format described above.

Constraints. $1 \leq N \leq 10^5$. All phone numbers consist of decimal digits, they don't have leading zeros, and each of them has no more than 7 digits. All names are non-empty strings of latin letters, and each of them has length at most 15. It's guaranteed that there is no person with name "not found".

Output Format. Print the result of each `find` query – the name corresponding to the phone number or "not found" (without quotes) if there is no person in the phone book with such phone number. Output one result per line in the same order as the `find` queries are given in the input.

Sample 1.

Input:

```
12
add 911 police
add 76213 Mom
add 17239 Bob
find 76213
find 910
find 911
del 910
del 911
find 911
find 76213
add 76213 daddy
find 76213
```

Output:

```
Mom
not found
police
not found
Mom
daddy
```

76213 is Mom's number, 910 is not a number in the phone book, 911 is the number of police, but then it was deleted from the phone book, so the second search for 911 returned "not found". Also, note that when the daddy was added with the same phone number 76213 as Mom's phone number, the contact's name was rewritten, and now search for 76213 returns "daddy" instead of "Mom".

Sample 2.

Input:

```
8
find 3839442
add 123456 me
add 0 granny
find 0
find 123456
del 0
del 0
find 0
```

Output:

```
not found
granny
me
not found
```

Recall that deleting a number that doesn't exist in the phone book doesn't change anything.

Starter Files

The starter solutions for `C++`, `Java`, and `Python3` in this problem read the input, implement a naïve algorithm to look up names by phone numbers and write the output. You need to use a fast data structure to implement a better algorithm. If you use other languages, you need to implement the solution from scratch.

What To Do

Use the direct addressing scheme.

2 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 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. $1 \le N \le 10^5$; $N/5 \le m \le N$. 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.

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
```

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, $h(\text{“world”}) = (119 + 111 \times 263 + 114 \times 263^2 + 108 \times 263^3 + 100 \times 263^4 \ mod\ 1\,000\,000\,007) \ mod\ 5 = 4$. 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
```

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 naïve 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 frequently encountered by learners:

- Beware of integer overflow. Use `long long` type in `C++` and `long` type in `Java` where appropriate. Take everything $(mod\ p)$ as soon as possible while computing something $(mod\ p)$, so that the numbers are always between 0 and $p - 1$.
- Beware of taking negative numbers $(mod\ p)$. 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: $x \leftarrow \big((a\%p) + p\big)\%p$ instead of just $x \leftarrow a\%p$.

3 Find pattern in text

Problem Introduction

In this problem, your goal is to implement the Rabin-Karp's algorithm.

Problem Description

Task. In this problem your goal is to implement the Rabin-Karp's algorithm for searching the given pattern in the given text.

Input Format. There are two strings in the input: the pattern P and the text T.

Constraints. $1 \leq |P| \leq |T| \leq 5 \cdot 10^5$. The total length of all occurrences of P in T doesn't exceed 10^8. The pattern and the text contain only latin letters.

Output Format. Print all the positions of the occurrences of P in T in the ascending order. Use 0-based indexing of positions in the text T.

Sample 1.

Input:

```
aba
abacaba
```

Output:

```
0 4
```

Explanation: The pattern `aba` can be found in positions 0 (**aba**caba) and 4 (abac**aba**) of the text `abacaba`.

Sample 2.

Input:

```
Test
testTesttesT
```

Output:

```
4
```

Explanation: Pattern and Text are case-sensitive in this problem. Pattern `Test` can only be found in position 4 in the Text `testTesttesT`.

Sample 3.

Input:

```
aaaaa
baaaaaaa
```

Output:

```
1 2 3
```

Note that the occurrences of the pattern in the text can be overlapping, and that's ok, you still need to output all of them.

Starter Files

The starter solutions in `C++`, `Java`, and `Python3` read the input, apply the naïve $O(|T||P|)$ algorithm to this problem and write the output. You need to implement the Rabin-Karp's algorithm instead of the naïve algorithm and thus significantly speed up the solution. If you use other languages, you need to implement a solution from scratch.

What To Do

Implement the fast version of the Rabin-Karp's algorithm from the lectures.

Some hints based on the problems frequently encountered by learners:

- Beware of integer overflow. Use `long long` type in `C++` and `long` type in `Java` where appropriate. Take everything $(mod\ p)$ as soon as possible while computing something $(mod\ p)$, so that the numbers are always between 0 and $p - 1$.
- Beware of taking negative numbers $(mod\ p)$. 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 $x \leftarrow a\%p$.
- Use operator `==` in Python instead of implementing your own function `AreEqual` for strings, because built-in operator `==` will work much faster.
- In C++, method `substr` of `string` creates a new string, uses additional memory and time for that, so use it carefully and avoid creating lots of new strings. When you need to compare pattern with a substring of text, do it without calling `substr`.
- In Java, however, method `substring` does NOT create a `new String`. Avoid using `new String` where it is not needed, just use `substring`.

4 Substring equality

Problem Introduction

In this problem, you will use hashing to design an algorithm that is able to preprocess a given string s to answer any query of the form "are these two substrings of s equal?" efficiently. This, in turn, is a basic building block in many string processing algorithms.

Problem Description

Input Format. The first line contains a string s consisting of small latin letters. The second line contains the number of queries q. Each of the next q lines specifies a query by three integers a, b, and l.

Constraints. $1 \le |s| \le 500\,000$; $1 \le q \le 100\,000$; $0 \le a, b \le |s| - l$ (hence the indices a and b are 0-based).

Output Format. For each query, output "Yes" if $s_a s_{a+1} \dots s_{a+l-1} = s_b s_{b+1} \dots s_{b+l-1}$ are equal, and "No" otherwise.

Sample 1.

Input:

```
trololo
4
0 0 7
2 4 3
3 5 1
1 3 2
```

Output:

```
Yes
Yes
Yes
No
```

0 0 7 → `trololo` = `trololo`

2 4 3 → `trololo` = `trololo`

3 5 1 → `trololo` = `trololo`

1 3 2 → `trololo` ≠ `trololo`

What To Do

For a string $t = t_0 t_1 \dots t_{m-1}$ of length m and an integer x, define a polynomial hash function:

$$H(t) = \sum_{j=0}^{m-1} t_j x^{m-j-1} = t_0 x^{m-1} + t_1 x^{m-2} + \dots + t_{m-2} x + t_{m-1}\,.$$

Let $s_a s_{a+1} \dots s_{a+l-1}$ be a substring of the given string $s = s_0 s_1 \dots s_{n-1}$. A nice property of the polynomial hash function H is that $H(s_a s_{a+1} \dots s_{a+l-1})$ can be expressed through $H(s_0 s_1 \dots s_{a+l-1})$ and $H(s_0 s_1 \dots s_{a-1})$, i.e., through hash values of two prefixes of s:

$$H(s_a s_{a+1} \cdots s_{a+l-1}) = s_a x^{l-1} + s_{a+1} x^{l-2} + \cdots + s_{a+l-1} =$$
$$= s_0 x^{a+l-1} + s_1 x^{a+l-2} + \cdots + s_{a+l-1} -$$
$$- x^l (s_0 x^{a-1} + s_1 x^{a-2} + \cdots + s_{a-1}) =$$
$$= H(s_0 s_1 \cdots s_{a+l-1}) - x^l H(s_0 s_1 \cdots s_{a-1})$$

This leads us to the following natural idea: we precompute and store the hash values of all prefixes of s: let $h[0] = 0$ and, for $1 \le i \le n$, let $h[i] = H(s_0 s_1 \ldots s_{i-1})$. Then, the identity above becomes

$$H(s_a s_{a+1} \cdots s_{a+l-1}) = h[a+l] - x^l h[a]\,.$$

In other words, we are able to get the hash value of any substring of s in just constant time! Clearly, if $H(s_a s_{a+1} \ldots s_{a+l-1}) \neq H(s_b s_{b+1} \ldots s_{b+l-1})$, then the corresponding two substrings ($s_a s_{a+1} \ldots s_{a+l-1}$ and $s_b s_{b+1} \ldots s_{b+l-1}$) are different. However, if the hash values are the same, it is still possible that the substrings are different – this is called a *collision*. Below, we discuss how to reduce the probability of a collision.

Recall that in practice one never computes the exact value of a polynomial hash function: everything is computed modulo m for some fixed integer m. This is done to ensure that all the computations are efficient and that the hash values are small enough. Recall also that when computing $H(s)$ mod m it is important to take every intermediate step (rather than the final result) modulo m.

It can be shown that if s_1 and s_2 are two different strings of length n and m is a prime integer, then the probability that $H(s_1)$ mod $m = H(s_2)$ mod m (over the choices of $0 \le x \le m-1$) is at most $\frac{n}{m}$ (roughly, this is because $H(s_1) - H(s_2)$ is a non-zero polynomial of degree at most $n-1$ and hence can have at most n roots modulo m). To further reduce the probability of a collision, one may take two different modulus.

Overall, this gives the following approach:

1. Fix $m_1 = 10^9 + 7$ and $m_2 = 10^9 + 9$.
2. Select a random x from 1 to 10^9.
3. Compare arrays $h_1[0..n]$ and $h_2[0..n]$: $h_1[0] = h_2[0] = 0$ and, for $1 \le i \le n$, $h_1[i] = H(s_0 s_1 \ldots s_{i-1})$ mod m_1 and $h_2[i] = H(s_0 s_1 \ldots s_{i-1})$ mod m_2. We illustrate this for h_1 below:

```
allocate h1[0..n]
h1[0] ← 0
for i from 1 to n:
    h1[i] ← (x · h1[i − 1] + si) mod m1
```

4. For every query (a, b, l):
   a. Use precomputed hash values, to compute the hash values of the substrings $s_a s_{a+1} \ldots s_{a+l-1}$ and $s_b s_{b+1} \ldots s_{b+l-1}$ modulo m_1 and m_2.
   b. Output "Yes", if
   $$H(s_a s_{a+1} \cdots s_{a+l-1}) \bmod m_1 = H(s_b s_{b+1} \cdots s_{b+l-1}) \bmod m_1 \text{ and}$$
   $$H(s_a s_{a+1} \cdots s_{a+l-1}) \bmod m_2 = H(s_b s_{b+1} \cdots s_{b+l-1}) \bmod m_2\,.$$
   c. Otherwise, output "No".

Note that, in contrast to Rabin-Karp's algorithm, we do not compare the substrings naively when their hashes coincide. The probability to this event is at most $\frac{n}{m_1} \cdot \frac{n}{m_2} \le 10^{-9}$. (In fact, for random strings the probability is even much smaller: 10^{-18}. In this problem, the strings are not random, but the probability of collision is still very small.)