PSH: Dynamic Boolean Expression Trees

Initially, you will be given symbols printed in a preorder traversal of a boolean expression tree. The internal nodes of the tree will contain one of the following operators: `&` (and), `|` (or), `^` (exclusive-or), or `!` (not). The nodes containing the first three operators will have two children, while the nodes containing the `!` operator will contain only a left child. The leaves of the tree will contain an operand - either `f` for false or `t` for true.

You will then be asked to perform a series of queries, in which, you evaluate the tree, or modification operations in which you change either an operator or an operand.

Input Format

The input begins with two space separated integers, n and q, on a line by itself. n indicates the number of nodes in the tree, while q gives the number of queries and modification operations.

The next n lines will contain the values from the pre-order traversal of the tree. These will be chosen from the following set of values: {`&`, `|`, `^`, `!`, `t`, `f`}.

The next q lines will be in one of the following formats:

```
mod [path] [new_value]
```

or

```
eval
```

where `[path]` is a string comprised of the symbols 'l' or 'r', and `[new_value]` is chosen from the following set of values: {`&`, `|`, `^`, `t`, `f`}.

Constraints

$1 \le n \le 10^4$

$1 \le q \le 10^3$

Output Format

For each `mod` line, you will not output anything, however, you should change the state of the tree, by following the path indicated in the `[path]` string, and changing the value stored at the node at the end of the path to the value in `[new_value]`. For example, if the command where `mod llr t`, you would start at the root, go to the left child (which we will call n_1), go to n_1's left child (which we will call n_2), and then go to n_2's right child (which we will call n_3). You will then change the value stored at n_3 to true.

For each `eval` line, you should output, on a line by itself, the result obtained when evaluating the current state of the tree, either `t` or `f`.

Notes:

- You should not output anything for the `mod` lines.
- There is guaranteed to be a node located at the end of the path specified in the `mod` commands.
- After each `mod` command, you are guaranteed that the tree will remain a valid boolean expression tree. For example, you will never be asked to change a node containing the `!` operator, you will never be asked to change the value of a node that contains an operand to an operator, nor will you be asked to change the value of a node that contains an operator to an operand.

Sample Input

```
6 4
^
!
t
|
f
t
mod ll f
eval
mod r &
eval
```

Sample Output

```
f
t
```

Explanation

Initially the tree looks like the following:

```
      ^
    /   \
  !      |
 /      / \
t      f   t
```

After the command `mod ll f`, the new tree looks like:

```
      ^
    /   \
  !      |
 /      / \
f      f   t
```

This corresponds to the boolean expression `!f ^ (f | t)` = `t ^ t` = `f`

Therefore at the `eval` command we will output `f`.

After the command `mod r &`, the new tree looks like:

```
      ^
    /   \
  !      &
 /      / \
f      f   t
```

This corresponds to the boolean expression `!f ^ (f & t)` = `t ^ f` = `t`

Therefore at the `eval` command we will output `t`.