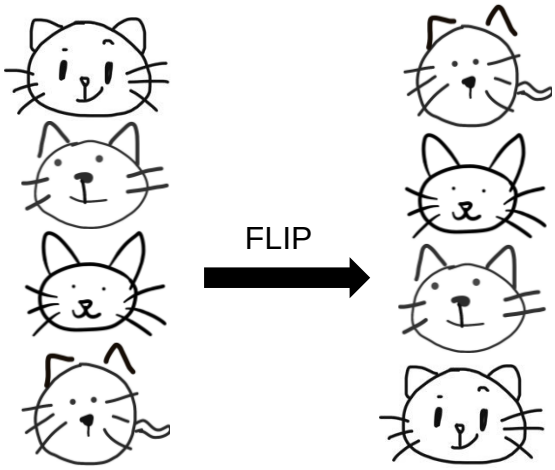
Kcats

Cats are weird animals. Sometimes, they like to stack on top of one another. One example of this is the stack of 4 cats illustrated below.

However, cats are also fickle minded and frequently would like to get on or off a stack of cats. As with all stacks, new cats can only **join** the stack from the top of the stack. Also, only the cat at the top of the stack can **leave** the stack.



Rar the Cat is an even weirder weird cat. Instead of joining a stack of cats, he likes to **flip** the whole stack of cats upside down. After a flip, the cat at the bottom of the stack will now be at the top. The cat that was second from the bottom of the original stack will now be second from the top of the flipped stack. In general, the **i**th cat from the **bottom** of the original stack will now be the **i**th cat from the **top** of the flipped stack. An illustration is shown on the right.

After a while, Rar the Cat got tired of physically flipping stacks of cats. Hence, he wants you to code a program to simulate this stack of cats. He plans to call this program *kcats*, which is 'stack' written from back to front.

The stack is **initially empty** and will need to support a few operations:

Operation	Description
`ADD [name]`	Adds a cat with name **[name]** to the top of the stack.
`REMOVE`	Removes the cat at the top of the stack and output the name of the removed cat.
`FLIP`	Flips the stack of cats. The **i**th cat from the **bottom** of the original stack should be the **i**th cat from the **top** of the flipped stack.

In addition, Rar the Cat wants to know how the stack will look like at the end of all the operations. The cats should be listed from top to bottom, with the cat at the top of the stack being outputted first.

Even though using Java API Stack might be the most direct way of solving this problem, flipping the stack of cats directly will involve a large number of operations. Hence, Rar the Cat suggests that clever usage of a Linear Data Structure **that can add and remove from both ends quickly** would be a good data structure to use for this problem. Instead of flipping all the cats one by one, it would be sufficient to just keep track of where the 'top' of the stack is (eg: front or back).

Input
The first line of input will be a single integer **K**, the number of operations to be performed on the stack. **K** lines will follow, representing an operation each. The operations should be executed in order and the format would be as described in the table above. (See sample)

Output
For every **REMOVE** operation, output the name of the removed cat on a single line.

At the end of all **K** operations, output the names of the cats in the stack on a single line from top to bottom. Add a single space between two consecutive names. **Do not print a space after the last name.** Instead, remember to print an end-line character at the end of the output.

Limits

- $1 \leq \mathbf{K} \leq 500{,}000$
- The names of cats will only contain lowercase and uppercase English alphabets. It will also not be longer than 20 characters.
- It is guaranteed that there will always be at least one cat in the stack when any **REMOVE** operation is given.
- It is also guaranteed that there will be at least one cat in the stack at the end of all **K** operations.

Sample Input (**kcats1.in**)	Sample Output (**kcats1.out**)
11 ADD RartheCat ADD PandaCat ADD SharkCat ADD BellCurveCat ADD StudentCat REMOVE ADD SoCCat REMOVE REMOVE FLIP ADD GziptheCat	StudentCat SoCCat BellCurveCat GziptheCat RartheCat PandaCat SharkCat

Explanation of Sample Testcase 1

Operation	**Stack of Cats (*Bottom* to Top)**
ADD RartheCat	**RartheCat**
ADD PandaCat	RartheCat, **PandaCat**
ADD SharkCat	RartheCat, PandaCat, **SharkCat**
ADD BellCurveCat	RartheCat, PandaCat, SharkCat, **BellCurveCat**
ADD StudentCat	RartheCat, PandaCat, SharkCat, BellCurveCat, **StudentCat**
REMOVE	RartheCat, PandaCat, SharkCat, BellCurveCat, ~~*StudentCat*~~
ADD SoCCat	RartheCat, PandaCat, SharkCat, BellCurveCat, **SoCCat**
REMOVE	RartheCat, PandaCat, SharkCat, BellCurveCat, ~~*SoCCat*~~
REMOVE	RartheCat, PandaCat, SharkCat, ~~*BellCurveCat*~~
FLIP	*SharkCat, PandaCat, RartheCat*
ADD GziptheCat	SharkCat, PandaCat, RartheCat, **GziptheCat**

Sample Input (**kcats2.in**)	Sample Output (**kcats2.out**)
7 ADD Meow ADD Miao ADD Miau ADD Nyan FLIP FLIP FLIP	Meow Miao Miau Nyan

Explanation of Sample Testcase 2

Operation	**Stack of Cats (*Bottom* to Top)**
ADD Meow	Meow
ADD Miao	Meow, Miao
ADD Miau	Meow, Miao, Miau
ADD Nyan	Meow, Miao, Miau, Nyan
FLIP	Nyan, Miau, Miao, Meow
FLIP	Meow, Miao, Miau, Nyan
FLIP	Nyan, Miau, Miao, Meow

Notes:

1. You should develop your program in the subdirectory **ex3** and use the skeleton java file provided. You should not create a new file or rename the file provided.
2. You are free to define your own helper methods and classes (or remove existing ones).
3. Please be reminded that the marking scheme is:
   a. Public Test Cases (1%) - 1% for passing **all** test cases, 0% otherwise
   b. Hidden Test Cases (1%) - Partial scoring depending on test cases passed
   c. Manual Grading (1%)
      i. Overall Correctness (correctness of algorithm, severity of bugs)
      ii. Coding Style (meaningful comments, modularity, proper indentation, meaningful method and variable names)
4. Your program will be tested with a time limit of not less than **2 sec** on Codecrunch.

Skeleton File – Kcats.java

You are given the below skeleton file `Kcats.java`. You should see a non-empty file when you open the skeleton file. Otherwise, you might be in the wrong working directory.

```
/**
 * Name         :
 * Matric. No   :
 * PLab Acct.   :
 */

import java.util.*;

public class Kcats {
    private void run() {
        //implement your "main" method here
    }

    public static void main(String[] args) {
        Kcats newKcats = new Kcats();
        newKcats.run();
    }
}
```