Class Photo

You have finally completed your 'O' levels and are ready to graduate from secondary school! To commemorate this event, your class is going to take a class photo. However, the class has to arrange themselves in a straight line in order of height first so they can be arranged in the photo to make the photo look as good as possible.

The students need to arrange themselves in a row such that the heights of the student from left to right are in non-increasing order. This means that every student with a student on his/her right should not be shorter than the student on his/her right. This makes it easier for the photographer to choose the range of students to put into each row of the photo later.

This should be an easy task, but the students are taking a long time to find their position in the line. To make matters worse, they keep going in and out of the row, making the arranging process very messy. As a good programmer, you step in and say "I can code a program to automate this process!". The class cheers at your bravery.

However, after hearing that, the photographer wants you to help him automate some of his processes as well. He needs to be able to find the students which are of the **N**-th shortest height for some **N** so he can decide who to put in the front row, and he also needs to know how many students are there between some two heights since he needs to select a suitable range of heights to place into each of the middle rows.

You have now gotten yourself into this so you need to code the program as quickly as possible so as not to disappoint your class and the photographer.

Input

The first line of the input consists of a single integer, **Q (1 <= Q <= 500)**, the number of queries you need to answer. It is then followed by **Q** lines, each containing a single query for you to answer. The queries follow the following specification:

Query Type	Input Format: `<QUERY_TYPE> <APPROPRIATE_PARAMETERS>`
1.	`arrive NAME HEIGHT` A student with name NAME and height HEIGHT arrives and is added to the row. The student will stand at the leftmost possible position such that the heights of the students are still in non-increasing order from the left to right. Print "**NAME is added at position POSITION**" where POSITION is position that student NAME is in after he/she is added to the row. The leftmost position is considered position 1.
2.	`leave NAME` The student with name NAME leaves the row. If such a student does not exist, print "**No student with name NAME**" replacing the NAME with the name given in the query. Otherwise, print "**NAME has left position POSITION**" replacing the NAME with the name given in the query and POSITION with the original position of the student before that student leaves. The leftmost position is considered position 1.

3. `shortest K`

   Print the names of all the students whose height is equal to the K-th shortest height in the row where K is a positive integer. Print these names in order from left to right, separating them with single spaces, do not print a space after the last name. If there is no K-th shortest student in the row, print "No such student".

4. `count LOW_HEIGHT HIGH_HEIGHT`

   Print the number of students whose height is between LOW_HEIGHT and HIGH_HEIGHT inclusive. It is guaranteed that LOW_HEIGHT <= HIGH_HEIGHT.

It is guaranteed that each student's name is unique, containing only small English letters ('a' - 'z') and not exceeding 10 characters each.

Output

Print the result of all the queries as described in the input format above. The last line of the output must contain a newline character. In the sample output below, the first line is left empty for better clarity of the sample. No blank lines are to be printed in the actual output.

Sample Input

```
12
arrive john 160
arrive kevin 170
arrive william 150
leave ivan
arrive ivan 170
arrive ken 160
shortest 1
shortest 3
count 150 165
leave william
shortest 3
count 145 155
```

Sample Output

```

john added at position 1
kevin added at position 1
william added at position 3
No student with name ivan
ivan added at position 1
ken added at position 3
william
ivan kevin
3
william has left position 5
No such student
0
```

Explanation

There are a total of 12 queries. For the first 3 queries, john, kevin and william are added to the row. Since we want to order them in non-increasing order, after each query the row will look like "john", "kevin john", "kevin john william". In the 4th query, ivan is not in the row so we print "No student with name ivan".

After the 5th and 6th query, the row looks like "ivan kevin john william" and "ivan kevin ken john william". Even though we can place kevin after ivan and ken after john, the students are placed this way because we add them to the leftmost position such that the order is still maintained. For the next query, the shortest person is william and no one else is of the same height as him. The 3rd shortest height is 170 so we print "ivan kevin" from left to right.

For the next query, william, john and ken all have heights between 150 and 165 inclusive so the answer is 3. Then william leaves the row so we only have 4 people left and the next SHORTEST query will return an error since there is no 3rd shortest person. For the last query, since william has left, there is no student with height between 145 and 155 inclusive so the answer is 0.

Clarification

For the **SHORTEST** query, K-th shortest height refers to the K-th shortest height as illustrated below:

Position	1	2	3	4	5
Height	10	10	9	8	8
K-th Shortest	3	3	2	1	1

Skeleton

You are given the skeleton file `ClassPhoto.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 ClassPhoto {

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

    public static void main(String[] args) {
        ClassPhoto newClassPhoto = new ClassPhoto();
        newClassPhoto.run();
    }
}

class Student {
    //define appropriate attributes, constructors, and methods here
}
```

Notes:

1. You should develop your program in the subdirectory `ex1` and use the skeleton java file provided. You should not create a new file or rename the file provided.
2. If your algorithm is different from the given skeleton, you are free to write a solution according to your own algorithm. However, your algorithm **must use linked list** to solve this problem. You are also **not allowed to use any methods from the Collections class**. Solution that does not use linked list or uses any methods from the Collections class will receive 0 marks. You are **not allowed** to use arrays, ArrayList, HashMap, etc. for this problem.
3. You are free to define your own classes (or remove existing ones) if it is suitable.
4. You **are allowed** to use the Java LinkedList API. No penalty will be given for this.
5. An implementation of a Doubly-Linked List similar to the one in the take-home lab is given to you inside the skeleton file. It is up to you whether you want to use it or not.
6. Please be reminded that the marking scheme is:

   Input : 10%
   Output : 10%
   Correctness : 50%
   Programming Style : 30% (awarded if you score at least 20% from the above):
   - Meaningful comments (pre- and post- conditions, comments inside the code): 10%
   - Modularity (modular programming, proper modifiers [public / private]): 10%
   - Proper Indentation: 5%
   - Meaningful Identifiers (for both method and variable names): 5%

   Compilation Error : Deduction of **50% of the total marks obtained**.