Playlist

Herbert the Clown is bored after a party conducted by the nice Professor Seth in his fast and furious module (no wonder it has the word 'accelerated' in its title[1]). When he is bored, Herbert the Clown likes to listen to music. He always uses his beloved phone to play music that he has in his playlist.

However, Herbert's phone is broken and he needs to play music on his computer. However, Herbert's computer does not have a music player that is as nice as the one in his phone. Therefore, he asks you, his friend, to make a music player for him. Herbert wants the following operations available:

1. Go forwards / backwards (i.e. skip) N tracks in the playlist.
2. Add a new song to be played **after** the current song.
3. Check whether a specific song is going to be played later.
4. Plays the current song.
5. Set the music player to loop the playlist or not.

Good Luck!

Input

The first line of input consists of two space-separated integers **N (1 <= N <= 100)** and **Q (1 <= Q <= 888)**, the number of initial songs inside the playlist and the number of queries respectively. Note that we use **1-based indexing for this problem.**

The next **N** lines contain the titles of the song from beginning to the end in the playlist. The titles of the songs consist of only English alphabets and numbers (**some with spaces**) only. All the titles are **unique**.

The next **Q** lines will contain a single query each. It is worth noting that **each query comes in between songs**. For example, at the beginning, the query comes **right before the first song is played**.

The queries are presented with the following format:

Query Type Input Format: `<QUERY_TYPE> <APPROPRIATE_PARAMETERS>`

1. `go DIRECTION STEP`

   Depending on the direction given, skip / go back **STEP (1 <= STEP <= SIZE [the number of songs inside the playlist])** number of songs. If you have reached the beginning or end of the playlist before moving the appropriate number of steps, stop there. **If the loop mode is on, there is no "end" of the playlist and you should keep going**.

   If the direction is 0, go back. If the direction is 1, go forward. **Do not play the skipped songs.** In case of loop on, if the direction is forward and it reaches the end, **immediately go to the front of the playlist**. See the clarification section for more details.

   After performing this query, print "**the current position is now INDEX**", where INDEX is the position of the next song that is going to be played. A special case is at the end of the playlist (there is no "next" song), in which you should print "**the current position is now at the end**".

[1] CS2020: Data Structures and Algorithms Accelerated. Ask your friends about this famous "Herbert the Clown". Or is there another Herbert there?

2. `add TITLE`

   Add the song with the title **TITLE.** It is guaranteed that a song with the title **TITLE** does not exist inside the playlist when this query is called. This new song will be played next (assuming that there are no user interventions). Print "**the song TITLE has been added at position INDEX**", where INDEX is the position of the new song.

3. `tobeplayed TITLE`

   Checks whether the song with the title **TITLE** is going to be played later, assuming that there are no user interventions except for playing songs. Print "**YES**" if it is, "**NO**" otherwise.

4. `play`

   Plays the current song. Print the title of the song and the number of times the song has been played before this query, separated by a single space. If this query is called when the playlist is at the end, print "**no more songs**". When loop is on when the last song has been played, immediately go to the start of the playlist.

5. `toggle`

   Toggle the loop mode. If the current loop is set to ON, turn it OFF. Otherwise, turn it ON. Print "**loop has been set to ON**" or "**loop has been set to OFF**" depending on the result of this query. Initially, loop is set to OFF. If loop is set to be ON when the position is already at the end (i.e. after the last song), immediately go to the first position (i.e. the start of the playlist). See the clarification section for more details.

Output

Print the result of each query 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 few lines are left empty for better clarity of the sample. No blank lines are to be printed in the actual output.

Sample Input	Sample Output
`5 16`	
`Ni Zenme Shuo`	
`You Are My Everything`	
`Hatimu Hatiku`	
`Yueliang Daibiao Wo De Xin`	
`Kisah Kasih di Sekolah`	
`play`	`Ni Zenme Shuo 0`
`tobeplayed Ni Zenme Shuo`	`NO`
`toggle`	`loop has been set to ON`
`tobeplayed Ni Zenme Shuo`	`YES`
`go 1 3`	`the current position is now 5`
`play`	`Kisah Kasih di Sekolah 0`
`toggle`	`loop has been set to OFF`
`play`	`Ni Zenme Shuo 1`
`add Paint My Love`	`the song Paint My Love has been added at position 2`
`go 1 6`	`the current position is now at the end`
`play`	`no more songs`
`toggle`	`loop has been set to ON`
`toggle`	`loop has been set to OFF`
`play`	`Ni Zenme Shuo 2`
`go 0 5`	`the current position is now 1`
`play`	`Ni Zenme Shuo 3`

Explanation

You are expected to trace the sample input yourself to gain a better understanding of the problem. Tracing the sample input on your own will give you insights on some of the cases you might encounter in this problem.

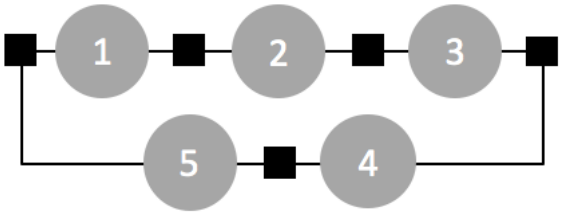
Clarification

As this is a truly difficult task, Herbert has kindly given a brief overview of how the playlist should be implemented.

Since each query is given in between songs, the playlist and query ordering looks like this:

The black squares represent the position in which the queries are executed. The diagram above shows the playlist when loop is set to off. Note that you cannot go beyond the first and last songs. The last "square" above shows that it is the end of the playlist. A "play" query called at that time will print "no more songs" and any "go" queries that end up there will print "at the end" as its current position.

If the loop is set to on, the playlist will look like this:



We see that, now, there is no "end" of the playlist and the last square has been merged with the first square. If the loop is broken up when you are at that square (i.e. the query toggle is called there and sets the loop off), you must stay in front of the playlist. If the loop is formed (i.e. toggle is called and set the loop on) and you are "at the end", you are now automatically "at the front" of the playlist again (i.e. the first square).

Skeleton

You are given the skeleton file `Playlist.java` with the following contents. 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 Playlist {

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

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

class Song {
    //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. 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.