Eels and Escalators

Eels and Escalators is a newly-developed board game. This revolutionary and original game is created by the famous company CPF (Children Party Federation), a company specialized in developing party games for children. They are looking into ways to make this game popular. The CPF therefore wants you to promote this game by creating an Eels and Escalators simulator that is easy to use.

Since this is an entirely new and revolutionary board game, the CPF wants to brief you about the rules and regulations of the game and how the game is played. Eels and Escalators is a game in which players play in a race to reach the finish line. Each player moves according to a series of dice rolls. If a player rolls “X”, that player moves “X” spaces forward. This game is played in a board as illustrated in the following diagram:

	100	99	98	97	96	95	94	93	92	91
	81	82	83	84	85	86	87	88	89	90
	80	79	78	77	76	75	74	73	72	71
	61	62	63	64	65	66	67	68	69	70
	60	59	58	57	56	55	54	53	52	51
	41	42	43	44	45	46	47	48	49	50
	40	39	38	37	36	35	34	33	32	31
	21	22	23	24	25	26	27	28	29	30
	20	19	18	17	16	15	14	13	12	11
0	1	2	3	4	5	6	7	8	9	10

In the board, there are eels and escalators (not shown on the diagram above). An eel will take players to a space behind their current cell while an escalator will take players to a space ahead of their current one. If players land on the bottom of an escalator (or the head of an eel) they will go to the other end of the escalator (or the eel), hence an escalator will bring players closer to the goal while an eel will bring players further from the goal. If players land in a normal cell, then they will stop there for now.

Initially, all players start from cell “0”. They take turns in rolling the dice. The one who reaches the cell “100” first wins. The remaining players then continue the game as per normal. Those who have reached cell “100” will not take anymore turns since they are deemed to have finished the game. If by any chance a player’s roll outnumbered “100” (for example, rolling a “5” from cell “99”), the player will just stop at the cell “100”.

Input

The first line of input contains three integers **N** (1 <= **N** <= 20), **E** (0 <= **E** <= 20), and **Q** (1 <= **Q** <= 500), the number of players, the total number of eels and escalators, and queries respectively. **N** rows follow, each consisting of a player’s name. The first line will be the first player and so on. **E** lines follow. Each of the E lines contains two integers, each denoting a cell number. The first number indicates the start of an eel (or escalator) and the second number indicates the end of an eel (or escalator), hence linking the first cell to the second cell. After that, **Q** lines follow, in which each line contains a single query with the following specification:

Query Type	Input Format: `<QUERY_TYPE> <APPROPRIATE_PARAMETERS>`
1.	`ROLL X` The current player moves X spaces forward. Print the current player's name and the cell that the player lands **after** that player has reached his or her final destination of the current move, separated by a single space. If there are no more players left, print "no more players". There is no whitespace after the cell number of the destination.
2.	`NUMPLAYER CELL_NUMBER` Print the number of players currently in the cell number CELL_NUMBER. It is guaranteed that the cell number CELL_NUMBER will refer to a valid cell.
3.	`POSITION PLAYER_NAME` Print the current cell number that the player PLAYER_NAME is in. It is guaranteed that the player PLAYER_NAME exists in the game.

It is guaranteed that no two players have the same name and all names will consist of small English letters ('a-z') only.

Output
Print according to the specification above. The last line of output contains a newline character.

Sample Input

```
2 5 12
johnkevin
martin
12 99
15 3
5 12
99 1
4 88
ROLL 4
ROLL 5
ROLL 10
ROLL 3
ROLL 1
POSITION martin
ROLL 9
ROLL 10
ROLL 12
ROLL 10
ROLL 10
NUMPLAYER 100
```

Sample Output

```
johnkevin 88
martin 12
johnkevin 98
martin 3
johnkevin 1
3
martin 99
johnkevin 11
martin 100
johnkevin 21
johnkevin 31
1
```

Explanation
First roll will land the player "johnkevin" on cell 4. However, there is an escalator from cell 4 to cell 88, hence the final position for this turn is 88. The second roll will land the player "martin" on cell 5. However, there is an escalator from cell 5 to cell 12, hence the final position for this turn is 12.

The third roll will land the player "johnkevin" on cell 98. The fourth roll will land the player "martin" on cell 15. However, there is an eel from cell 15 to cell 3. Hence, the final position for this turn is 3.

You are advised to trace the remaining queries of the sample input on your own to get a better understanding of the game. Note that for the last two rolls, the player "johnkevin" moves since the player "martin" has reached the goal.

Skeleton

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

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. Solution that does not use linked list will receive 0 marks. You are not allowed to use arrays, ArrayList, HashMap, etc. for this problem.
3. You are given a **fully-functional singly-linked list** in the skeleton file. You are free to modify the provided implementation to suit your needs. Alternatively, you are allowed to use Java's API instead. Please take note to **remove the provided linked list (and list node) class** in case you want to use Java's linked list.
4. You are **free to define your own classes (or remove existing ones)** if it is suitable.
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**.

Skeleton File – EelsAndEscalators.java

You should see the following contents when you open the skeleton file:

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

import java.util.*;

public class EelsAndEscalators {

    public EelsAndEscalators() {
        // constructor
    }

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

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

class Cell {
    // define appropriate attributes, methods, and constructor
}

class Player {
    // define appropriate attributes, methods, and constructor
}

class TailedLinkedList<E> {

    // Data attributes
    private ListNode<E> head;
    private ListNode<E> tail;
    private int num_nodes;

    public TailedLinkedList() {
        this.head = null;
        this.tail = null;
        this.num_nodes = 0;
    }

    // Return true if list is empty; otherwise return false.
    public boolean isEmpty() {
        return (num_nodes == 0);
    }

    // Return number of nodes in list.
    public int size() {
        return num_nodes;
    }

    // Return value in the first node.
    public E getFirst() throws NoSuchElementException {
        if (head == null)
            throw new NoSuchElementException("can't get from an empty list");
        else
            return head.getElement();
    }

    // Return true if list contains item, otherwise return false.
    public boolean contains(E item) {
        for (ListNode<E> n = head; n != null; n = n.getNext())
            if (n.getElement().equals(item))
                return true;

        return false;
    }
    // Add item to front of list.
    public void addFirst(E item) {
        head = new ListNode<E>(item, head);
        num_nodes++;
        if (num_nodes == 1) tail = head;
    }

    // Return reference to first node.
    public ListNode<E> getHead() {
        return head;
    }

    // Return reference to last node of list.
    public ListNode<E> getTail() {
        return tail;
    }

    // Add item to end of list.
    public void addLast(E item) {
        if (head != null) {
            tail.setNext(new ListNode<E>(item));
            tail = tail.getNext();
        } else {
            tail = new ListNode<E>(item);
            head = tail;
        }
        num_nodes++;
    }

    // Remove node after node referenced by current
    public E removeAfter(ListNode<E> current) throws NoSuchElementException {
        E temp;
        if (current != null) {
            ListNode<E> nextPtr = current.getNext();
            if (nextPtr != null) {
                temp = nextPtr.getElement();
                current.setNext(nextPtr.getNext());
                num_nodes--;
                if (nextPtr.getNext() == null) // last node is removed
                    tail = current;
                return temp;
            } else
                throw new NoSuchElementException("No next node to remove");
        } else { // if current is null, we want to remove head
            if (head != null) {
                temp = head.getElement();
                head = head.getNext();
                num_nodes--;
                if (head == null)
                    tail = null;
                return temp;
            } else
                throw new NoSuchElementException("No next node to remove");
        }
    }

    // Remove first node of list.
    public E removeFirst() throws NoSuchElementException {
        return removeAfter(null);
    }

    // Remove item from list
    public E remove(E item) throws NoSuchElementException {
        ListNode<E> current = head;
        if (current == null) {
            throw new NoSuchElementException("No node to remove");
        }
        if (current.getElement().equals(item)) {
            return removeAfter(null);
        }
        while (current.getNext().getElement() != null) {
            if (current.getNext().getElement().equals(item)) {
                return removeAfter(current);
            }
            current = current.getNext();
        }
        throw new NoSuchElementException("No node to remove");
    }
}
```

```
class ListNode<E> {
    protected E element;
    protected ListNode<E> next;

    /* constructors */
    public ListNode(E item) {
        this.element = item;
        this.next = null;
    }

    public ListNode(E item, ListNode<E> n) {
        element = item;
        next = n;
    }

    /* get the next ListNode */
    public ListNode<E> getNext() {
        return this.next;
    }

    /* get the element of the ListNode */
    public E getElement() {
        return this.element;
    }

    public void setNext(ListNode<E> item) {
        this.next = item;
    }

    public void setElement(E item) {
        this.element = item;
    }
}
```