Programming Part

Problem 3 - Daily Study Auditorium: XOR linked list (100 pts)

Problem Description

Doubly linked lists are convenient, but come with two disadvantages. Firstly, two pointers are used per node instead of one, making it hard to fit small-cache/memory scenarios. Secondly, reversing a doubly linked list typically requires swapping the `next` and `prev` pointers of every node, making it less efficient for applications with frequent reversal needs.

The XOR linked list is here for rescue! In this problem, we ask you to play with it. The following description of the XOR linked list is quoted from Wikipedia (https://en.wikipedia.org/wiki/XOR_linked_list):

An XOR linked list is a type of data structure used in computer programming. It takes advantage of the bitwise XOR operation to decrease storage requirements for doubly linked lists by storing the composition of both addresses in one field. While the composed address is not meaningful on its own, during traversal it can be combined with knowledge of the last-visited node address to deduce the address of the following node.

Provided Files

In NTU COOL, we provide you with three files: a header file `9.h`, a supplementing source file `9.c`, and the main source file `main.c`. Your goal is to complete `9.c`. Then, we will use the following command to compile your code:

```
gcc 9.c main.c -static -O2 -std=c11
```

where `9.c` is your submitted code, and `main.c` and `9.h` are the exact same ones that you fetched from NTU COOL.

The header file `9.h` contains the implementation of 2 utility functions as well as the signatures of 3 routines and 7 types of operations. All functions are based on the `Node` structure that stores the information of a node in the XOR linked list. The external variables `head` and `tail` represent the head and the tail of the XOR linked list, respectively. Their actual declarations are given in `9.c`.

The 2 utility functions that are implemented and can be directly used are

```
NEXT-NODE($node, prev$)
1  return $node.neighbors \oplus prev$
```

```
NEW-XOR-NODE($neighbors$)
1  initialize $newNode$ with $data$ (see below) and $neighbors$
2  return $newNode$
```

To simplify the implementation logistics, we assume that *data* will be assigned to a sequential ID that represents the number of calls to NEW-XOR-NODE calls so far. That is, the first new node will contain $data = 1$, the second new node will contain $data = 2$, and so on. The next ID is stored in an external variable called `next_node_id`, whose actual declaration is given in `9.c`.

In `9.c`, you need to implement 3 routines and 7 types of operations, as declared in `9.h`. The 3 routines are

- The $O(1)$-time INSERT-AFTER($node, prev$) is for inserting a node `newNode` after the node `node`, where `prev` is the previous node of `node`.

  ```
  INSERT-AFTER($node, prev$)
  1  $next$ = NEXT-NODE($node, prev$)
  2  $newNode$ = NEW-XOR-NODE($next \oplus node$)
  3  $next.neighbors = next.neighbors \oplus node \oplus newNode$
  4  $node.neighbors = prev \oplus newNode$
  ```

- The $O(1)$-time REMOVE-HERE($node, prev$) is for removing the node `node`, where `prev` is the previous node of `node`.

  ```
  REMOVE-HERE($node, prev$)
  1  $next$ = NEXT-NODE($node, prev$)
  2  $prev.neighbors = prev.neighbors \oplus node \oplus next$
  3  $next.neighbors = next.neighbors \oplus node \oplus prev$
  4  FREE($node$)
  ```

- The $O(1)$-time Reverse($prev, begin, end, next$) is for reversing all the nodes in range `[begin, end]`, where `prev` is the previous node of `begin` and `next` is the next node of `end`.

```
Reverse($prev, begin, end, next$)
1  $prev.neighbors = prev.neighbors \oplus begin \oplus end$
2  $begin.neighbors = begin.neighbors \oplus prev \oplus next$
3  $end.neighbors = end.neighbors \oplus next \oplus prev$
4  $next.neighbors = next.neighbors \oplus end \oplus begin$
```

After completing the 3 routines, you can then use them to implement the 7 types of operations below.

- `type_0(`k`)` prints out the *data* field of the k-th node of the XOR linked list, where k is 1-based.
- `type_1()` calls Insert-After at the head of the XOR linked list, The new node will then become the first node of the linked list.
- `type_2(`k`)` calls Insert-After at the k-th node of the XOR linked list, where k is 1-based. The new node will then become the $(k+1)$-th node of the linked list.
- `type_3(`k`)` calls Insert-After at the k-th last node of the XOR linked list, where k is 1-based. The new node will then become the k-th last node of the linked list.
倒數第k個？
- `type_4(`k`)` calls Remove-Here at the k-th node of the XOR linked list, where k is 1-based.
- `type_5(`k`)` calls Remove-Here at the k-th last node of the XOR linked list, where k is 1-based.
- `type_6(`k`)` calls Reverse from the k-th node of the XOR linked list to the k-th last node of the XOR linked list, where k is 1-based.

The file `main.c` is the actual program entrance that will be used to test your code. It contains input processing and calls to the 7 types of operations. You do not need to change anything in `main.c`.

We will only use the `9.c` that you pushed to the judge system to test your code. Therefore, you are strongly suggested not to change `9.h` and `main.c` at all. In addition, you do not need to do any outputting within `9.c`. Otherwise you risk being misjudged by the system.

Input

We assume initially the XOR linked list is empty. The first line includes one integers, M, representing the number of operation. The next M lines include 2 integers, t and k. The first integer, t, represents the operation type. The second integer, k, represents the additional parameters for the operation.

Output

For each `type_0` operation, print the value of the *data* in the k-th node in one line.

Constraints

- $1 \leq M \leq 5 \times 10^5$
- $t \in \{0, 1, 2, 3, 4, 5, 6\}$
- For all $t \in \{0, 2, 3, 4, 5\}$, $1 \leq k \leq |L|$, where $|L|$ is the current length of the XOR linked list
- When $t = \{1\}$, there is a dummy $k = 0$
- When $t = \{6\}$, $1 \leq k \leq \lceil \frac{|L|}{2} \rceil$, where $|L|$ is the current length of the XOR linked list
- We ensure that if the XOR linked list is currently empty, $t = 1$
- The total k within the M operations is $\leq 5 \times 10^7$

Subtasks

Subtask 1 (10 pts)

- $t \in \{0, 1\}$

Subtask 2 (15 pts)

- $t \in \{0, 1, 2\}$

Subtask 3 (15 pts)

- $t \in \{0, 1, 2, 3\}$

Subtask 4 (20 pts)

- $t \in \{0, 1, 2, 3, 4, 5\}$

Subtask 5 (40 pts)

No other constraints.

Sample Testcases

Sample Input 1

```
6
1 0
1 0
0 1
0 2
1 0
0 3
```

Sample Output 1

```
2
1
1
```

Sample Input 2

```
10
1 0
2 1
3 2
2 3
3 1
0 1
0 2
0 3
0 4
0 5
```

Sample Output 2

```
1
3
2
4
5
```

Sample Input 3

```
11
1 0
2 1
3 2
2 3
3 1
6 2
4 1
5 3
0 1
0 2
0 3
```

Sample Output 3

```
4
3
5
```

Hints

Sample 1 Explanations

- After completing first two `type_1` operations, the `data` of the XOR linked list's nodes are: $2 \to 1$, where `head` is on the left.
- After completing the last `type_1` operation, the `data` of the XOR linked list's nodes are: $3 \to 2 \to 1$.

Sample 3 Explanations

- After completing the `type_6` operation, the `data` of the XOR linked list's nodes are: $1 \to 4 \to 2 \to 3 \to 5$.
- After completing the `type_4` and `type_5` operations, the `data` of the XOR linked list's nodes are: $4 \to 3 \to 5$.

Problem 4 - Diamond Set Arrangement (100 pts)

Problem Description

In this problem, we will provide you with a story version, and an equivalent formal version. Please feel free to decide what version(s) you want to read to solve the problem.

Story Version

Michael owns a collection of diamonds, each with a specific *value*. Ze wants to organize zir diamonds efficiently, but the task is time-consuming. To streamline the process, ze decides to design a Diamond Set Arrangement (DSA) protocol. Knowing that you are proficient in DSA—and that you are also taking a Data Structures and Algorithms (DSA) class at NTU CSIE—ze seeks your help in implementing it.

Initially, you are given two integers, M and T. You then need to perform T operations, which can be of three possible types. In the i-th round, one of the following operations is executed.

1. For some given integers N_i and v_i, remove all diamonds from the set with a $value < v_i$ (and output the number of diamonds that have been removed). Then, add N diamonds with value v_i to the collection.

2. For some given integer p_i, output the number of diamonds in the collection that satisfies $value = p_i$.

3. Sort the diamonds in the collection in descending order based on their *value*. If multiple diamonds have the same *value*, their relative ranking can be arbitrary. After sorting, each diamond, regardless of whether they are of the same *value*, would then get a unique rank, starting from rank 1. Then, for each diamond ranked j, increase its value by $(M - j + 1)$, where M is a given constant. It is guaranteed that the total number of diamonds does not exceed M.

Formal Version

Given two integers M and T, effectively maintain a multiset[1] S of integers that supports the following operations for T rounds. In the i-th round, one of the following operations is executed.

1. For some given integers N_i and v_i, remove all elements in S that are strictly less than v_i (and output the number of elements that have been removed). Then, add N_i elements of v_i to S.

2. For some given integer p_i, output the multiplicity of p_i.

3. Sort all elements in S in descending order as $s_1, s_2, \ldots, s_{|S|}$. If multiple elements have the same value, their relative ranking can be arbitrary. That is, there can be $s_j = \cdots = s_k$ where the first one is of rank j and the last one is of rank k. Update each s_j to $s_j + (M - j + 1)$, where M is a given constant. It is guaranteed that $|S| \leq M$.

Note that while the formal version is illustrated with the multiset, we do *not* expect you to implement a *general* multiset (which is a more advanced topic). You should be able to solve this problem with what you have learned so far in class to build a *special* design of the data structure.

Input

The first line contains 2 space-separated integers T and M. The i-th line of the following T lines contains one, two, or three integers depending on the operation:

- 1 N_i v_i, indicating a type 1 operation with parameter (N_i, v_i).
- 2 p_i, indicating a type 2 operation with parameter p_i.
- 3, indicating a type 3 operation.

Output

- For each type 1 operation, print out the number of diamonds (elements) removed in one line.
- For each type 2 operation, print out the number of diamonds in the collection with value p_i (i.e. multiplicity of p_i in S) in one line.
- For each type 3 operation, do not print out anything.

[1]See https://en.wikipedia.org/wiki/Multiset about multiset.

Constraints

- $1 \leq T \leq 10^6$
- $1 \leq M \leq 10^{12}$
- $1 \leq N_i \leq 10^6$
- $1 \leq v_i \leq 10^{18}$
- $1 \leq p_i \leq 2 \times 10^{18}$
- Each operation is guaranteed to be legal.

Subtasks

Subtask 1 (10 pts)

- $1 \leq T \leq 10^3$
- $1 \leq M \leq 10^5$
- $1 \leq N_i \leq 10$
- $1 \leq v_i \leq 2 \times 10^8$
- $1 \leq p_i \leq 10^9$

Subtask 2 (10 pts)

- $1 \leq T \leq 10^4$
- $1 \leq M \leq 10^8$
- $1 \leq N_i \leq 10^3$
- $1 \leq v_i \leq 2 \times 10^{12}$
- $1 \leq p_i \leq 10^{13}$

Subtask 3 (20 pts)

Only operations 1 and 2 are present.

Subtask 4 (20 pts)

Only operations 1 and 3 are present.

Subtask 5 (40 pts)

No other constraints.

Sample Testcases

Sample Input 1

```
5 20
1 5 4
1 6 3
3
1 2 17
2 17
```

Sample Output 1

```
0
0
4
3
```

Sample Input 2

```
6 20
1 5 7
1 6 9
2 7
2 9
1 2 9
2 9
```

Sample Output 2

```
0
5
0
6
0
8
```

Sample Input 3

```
6 20
1 5 4
3
1 6 10
3
1 2 17
1 2 30
```

Sample Output 3

```
0
0
0
8
```