Задача 9. XOR-список

Источник:	повышенной сложности*
Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 секунда
Ограничение по памяти:	разумное

В узле односвязного списка хранится один указатель, но по такому списку нельзя перемещаться в обратную сторону и неудобно удалять элементы. В двусвязном списке всё просто и красиво, но приходится хранить уже по два указателя в каждом узле. Данная задача посвящена очень странной конструкции, когда в каждом узле хранится только одно дополнительное значение, но при этом можно перемещаться в обе стороны.

Как известно, указатель — это просто адрес в памяти, а над адресами можно выполнять арифметические операции. В узле предлагаемого списка вместо указателей `next` и `prev` нужно хранить, к примеру, сумму этих двух адресов. Тогда если в программе известны указатели на два подряд идущих элемента такого списка `left` и `right`, то адрес соседнего узла можно найти вычитанием адреса одного узла из хранящейся в другом узле суммы адресов соседей. Общий принцип заключается в том, что для выполнения любой операции (даже перемещения) в таком списке необходимо иметь два указателя, которые смотрят на два подряд идущих элемента списка, потому что по одному указателю невозможно найти соседние узлы.

Канонически, вместо суммы адресов соседей в узле обычно хранят “побитовое исключающее или” (т.е. XOR) адресов соседей. Тому есть несколько причин: во-первых, операция XOR играет роль сложения и вычитания одновременно; во-вторых, операция XOR порой выполняется немного быстрее сложения и вычитания, т.к. её проще реализовать на аппаратном уровне. Канонический вариант XOR-списка описан в википедии.

Требуется реализовать XOR-список и решить тестовую задачу. Примерно как в задаче “Список с указателями”, но теперь в узлах надо хранить целочисленные значения.

Нужно реализовать следующие функции:

```
#define VALTYPE int      //тип значений: целое
typedef struct Node_s {
    size_t xorlinks;     //XOR адресов соседних узлов
    VALTYPE value;       //значение в этом узле
} Node;
typedef struct List_s {
    ???                  //тут можно хранить всякие данные
} List;

//инициализировать пустой список в list
void initList(List *list);
//добавить новый узел со значением val между соседними узлами left и right
Node *addBetween(Node *left, Node *right, VALTYPE val);
//дано два соседних узла left и right, нужно удалить узел left
VALTYPE eraseLeft(Node *left, Node *right);
//дано два соседних узла left и right, нужно удалить узел right
VALTYPE eraseRight(Node *left, Node *right);
```

Для простоты реализации рекомендуется сделать XOR-список кольцевым с **двумя** вспомогательными элементами (“начало” и “конец”).

Формат входных данных

В первой строке файла записано одно целое число T — количество тестов в файле. Далее в файле идут тесты (T штук) подряд, один за другим.

Первая строка теста начинается с целого числа Q — количество операций, которые нужно выполнить ($0 \leqslant Q \leqslant 10^5$). В начале каждого теста список пустой.

Затем идут Q строк, которые описывают операции над списком. В каждой строке сначала записан тип операции: 1 — удаление правого, -1 — удаление левого, 0 — добавление. Затем указано два индекса: для узла `left` и узла `right`. Если описывается операция вставки, то в конце также задано целочисленное значение нового узла.

Гарантируется, что указанный узел `right` всегда идёт сразу после указанного узла `left`. Один из этих узлов может быть не указан (тогда записан индекс -1), если выполняется операция на краю списка.

Все значения узлов лежат в диапазоне от 0 до 10^6 включительно.

Сумма Q по всем тестам не превышает 10^5.

Формат выходных данных

Для каждого теста нужно вывести строковые значения всех узлов списка после выполнения операций (в порядке их следования в списке), и строку `"==="` в конце.

Пример

input.txt	output.txt
2 5 0 -1 -1 4283 0 -1 0 2718 0 0 -1 5000 0 -1 1 1111 0 1 0 3141 7 0 -1 -1 0 0 -1 0 1000 0 0 -1 2000 0 1 0 3000 0 2 -1 4000 1 0 2 -1 0 4	1111 2718 3141 4283 5000 === 1000 3000 4000 ===