模拟散列表 （拉链法）

维护一个集合，支持如下几种操作：

1. `I x`，插入一个数 x；
2. `Q x`，询问数 x 是否在集合中出现过；

现在要进行 N 次操作，对于每个询问操作输出对应的结果。

输入格式

第一行包含整数 N，表示操作数量。

接下来 N 行，每行包含一个操作指令，操作指令为 `I x`，`Q x` 中的一种。

输出格式

对于每个询问指令 `Q x`，输出一个询问结果，如果 x 在集合中出现过，则输出 `Yes`，否则输出 `No`。

每个结果占一行。

数据范围

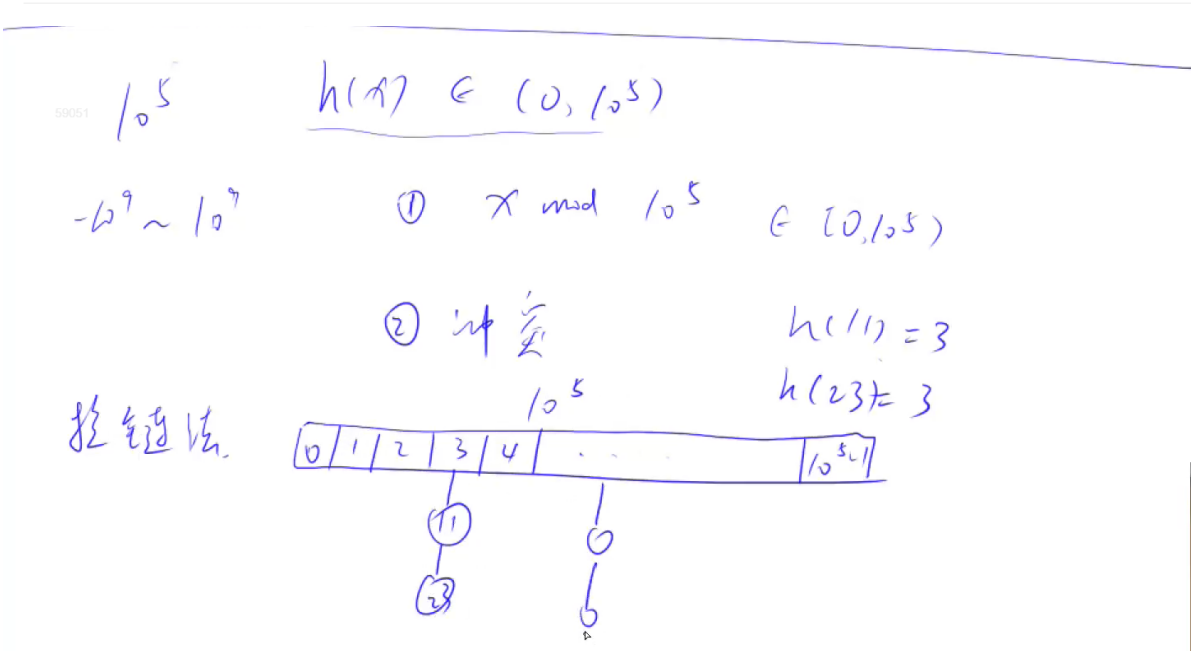
$1 \le N \le 10^5$
$-10^9 \le x \le 10^9$

输入样例：

```
5
I 1
I 2
I 3
Q 2
Q 5
```

输出样例：

```
Yes
No
```



Hash表又称为散列表，一般由Hash函数(散列函数) 与链表结构共同实现。 与离散化思想类似，当我们要对若干复杂信息进行统计时，可以用Hash函数把这些复杂信息映射到一个容易维护的值域内。因为值域变简单、范围变小，有可能造成两个不同的原始信息被Hash函数映射为相同的值，所有我们需要处理这种冲突情况。

有一种称为"开散列"的解决方案是，建立一个邻接表结构，以Hash函数的值域作为表头数组head，映射后的值相同的原始信息被分到同一类，构成一个链表接在对应的表头之后，链表的节点上可以保存原始信息和一些统计数据。

Hash表主要包括来两个基本操作:

1. 计算Hash函数的值。
2. 定位到对应链表中依次遍历、比较。

无论是检查任意一个给定的原始信息在Hash表中是否存在，还是更新它在Hash表中的统计数据，都需要基于这两个基本操作进行。

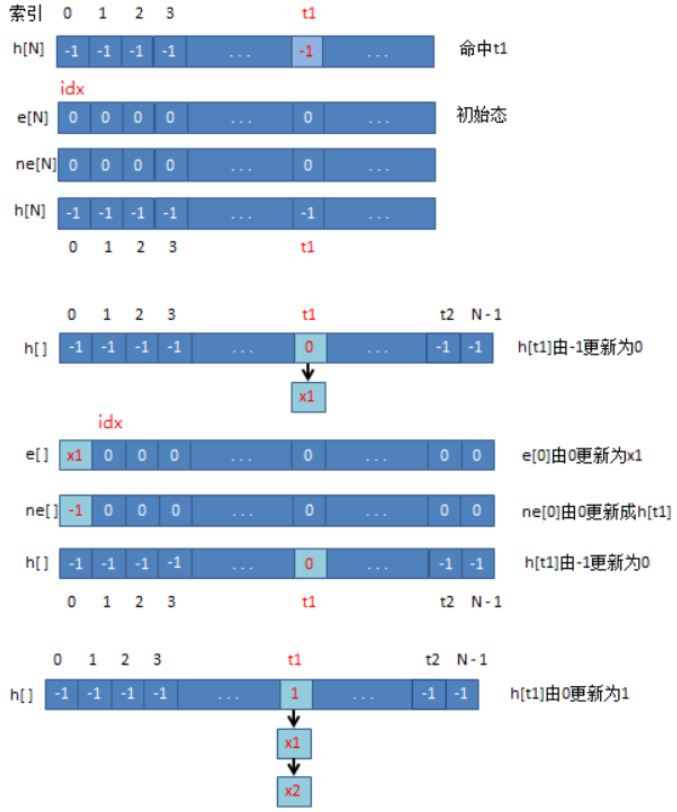
当Hash函数设计较好时，原始信息会被比较均匀地分配到各个表头之后，从而使每次查找，统计的时间降低到"原始信息总数除以表头数组长度"。 若原始信息总数与表头数组长度都是O(N)级别且Hash函数分散均匀，几乎不产生冲突，那么每次查找，统计的时间复杂度期望为O(1)。

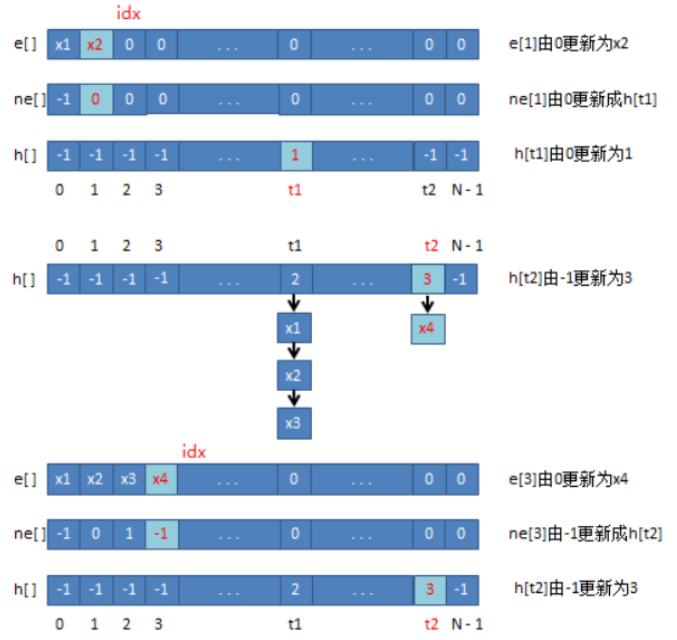
例如，我们要在一个长度为N的随机整数序列A中统计每个数出现了多少次。当数列A中的值都比较小时，我们可以直接用一个数组计数（建立一个大小等于值域的数组进行统计和映射，其实就是最简单的Hash思想)。 当数列A中的值很大时，我们可以把A排序后扫描统计。 这里我们换一个思路，尝试一下Hash表的做法。

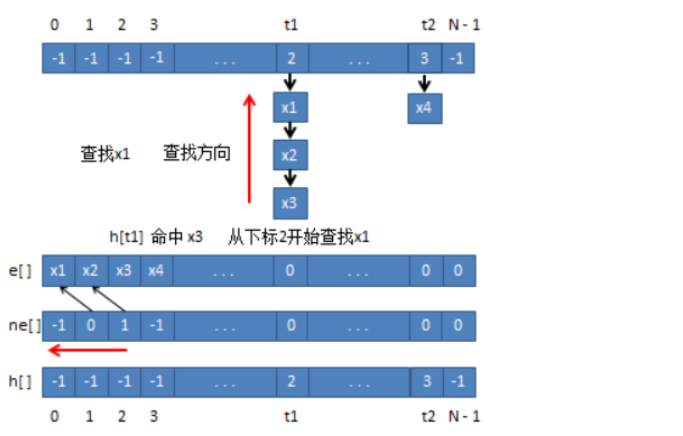
设计Hash函数为H(x)=(x mod P)+1，其中P是一个比较大的质数，但不超过N。 显然，这个Hash函数把数列A分成P类，我们可以依次考虑数列中的每个数A[i]，定位到head[H(A[i])] 这个表头所指向的链表。如果该链表中不包含A[i]，我们就在表头后插入一个新节点A[i]，并在该节点上记录A[i]出现了1次，否则我们就在直接找到已经存在的A[i]节点将其出现次数+1。 因为整数序列A是随机的，所以最终所有的A[i]会比较均匀地分散在各个表头之后，整个算法的时间复杂度可以近似达到O(N)。

上面的例子是一个非常简单的Hash表的直观应用。 对于非随机的数列，我们可以设计更好的Hash函数来保证其时间复杂度。同样的，如果我们需要维护的是比大整数复杂得多得信息的某些特性(如是否存在，出现次数等)，也可以用Hash表来解决。 字符串就是一种比较一般化的信息，在本节的后半部分，我们将会介绍一个程序设计竞赛中极其常用的字符串Hash算法。

图示算法


索引 0 1 2 3 t1
h[N] -1 -1 -1 -1 ... -1 ... 命中t1
idx
e[N] 0 0 0 0 ... 0 ... 初始态
ne[N] 0 0 0 0 ... 0 ...
h[N] -1 -1 -1 -1 ... -1 ...
0 1 2 3 t1
0 1 2 3 t1 t2 N-1
h[ ] -1 -1 -1 -1 ... 0 ... -1 -1 h[t1]由-1更新为0
x1
idx
e[ ] x1 0 0 0 ... 0 ... 0 0 e[0]由0更新为x1
ne[ ] -1 0 0 0 ... 0 ... 0 0 ne[0]由0更新成h[t1]
h[ ] -1 -1 -1 -1 ... 0 ... -1 -1 h[t1]由-1更新为0
0 1 2 3 t1 t2 N-1
0 1 2 3 t1 t2 N-1
h[ ] -1 -1 -1 -1 ... 1 ... -1 -1 h[t1]由0更新为1
x1
x2


idx
e[ ] x1 x2 0 0 ... 0 ... 0 0 e[1]由0更新为x2
ne[ ] -1 0 0 0 ... 0 ... 0 0 ne[1]由0更新成h[t1]
h[ ] -1 -1 -1 -1 ... 1 ... -1 -1 h[t1]由0更新为1
0 1 2 3 t1 t2 N-1
0 1 2 3 t1 t2 N-1
h[ ] -1 -1 -1 -1 ... 2 ... 3 -1 h[t2]由-1更新为3
x1
x4
x2
x3
idx
e[ ] x1 x2 x3 x4 ... 0 ... 0 0 e[3]由0更新为x4
ne[ ] -1 0 1 -1 ... 0 ... 0 0 ne[3]由-1更新成h[t2]
h[ ] -1 -1 -1 -1 ... 2 ... 3 -1 h[t2]由-1更新为3
0 1 2 3 t1 t2 N-1


0 1 2 3 t1 t2 N-1
-1 -1 -1 -1 ... 2 ... 3 -1
x1
x4
x2
x3
查找x1
查找方向
h[t1] 命中 x3 从下标2开始查找x1
e[ ] x1 x2 x3 x4 ... 0 ... 0 0
ne[ ] -1 0 1 -1 ... 0 ... 0 0
h[ ] -1 -1 -1 -1 ... 2 ... 3 -1
0 1 2 3 t1 t2 N-1

y总代码：

```
#include <iostream>

using namespace std;

const int N = 100003;

int h[N], e[N], ne[N], idx;

void insert(int x)
{
    int k = (x % N +N) % N;
    e[idx] = x;
    ne[idx] = h[k];
    h[k] = idx ++ ;
}

bool find(int x)
{
    int k = (x % N + N) % N;
    for (int i = h[k]; i != -1; i = ne[i])
        if (e[i] == x)
            return true;

    return false;
}

int main()
{
    int n;
    scanf("%d", &n);

    memset(h, -1, sizeof h);

    while (n -- )
    {
        char op[2];
        int x;
        scanf("%s%d", op, &x);

```

```
//拉链法
#include <cstring>
#include <iostream>

using namespace std;

const int N = 100003;

int h[N], e[N], ne[N], idx;

void insert(int x)
{
    int k = (x % N + N) % N;
    e[idx] = x;
    ne[idx] = h[k];
    h[k] = idx ++ ;
}

bool find(int x)
{
    int k = (x % N + N) % N;
    for (int i = h[k]; i != -1; i = ne[i])
        if (e[i] == x)
            return true;

    return false;
}
```

```
int main()
{
    int n;
    scanf("%d", &n);

    memset(h, -1, sizeof h);

    while (n -- )
    {
        char op[2];
        int x;
        scanf("%s%d", op, &x);

        if (*op == 'I') insert(x);
        else
        {
            if (find(x)) puts("Yes");
            else puts("No");
        }
    }

    return 0;
}
```