

## chapter2 数据结构

### 课堂笔记

#### (一)

以数组模拟为主

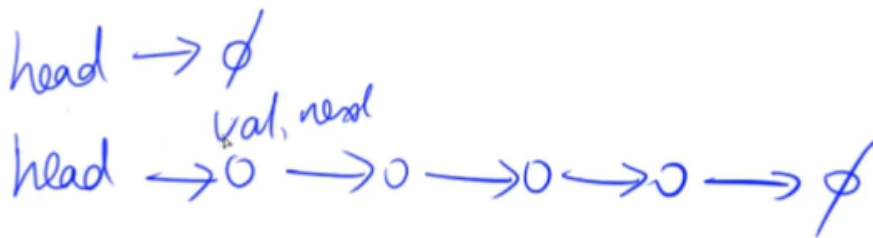
#### 链表与邻接表

动态链表（笔试题一般不用）

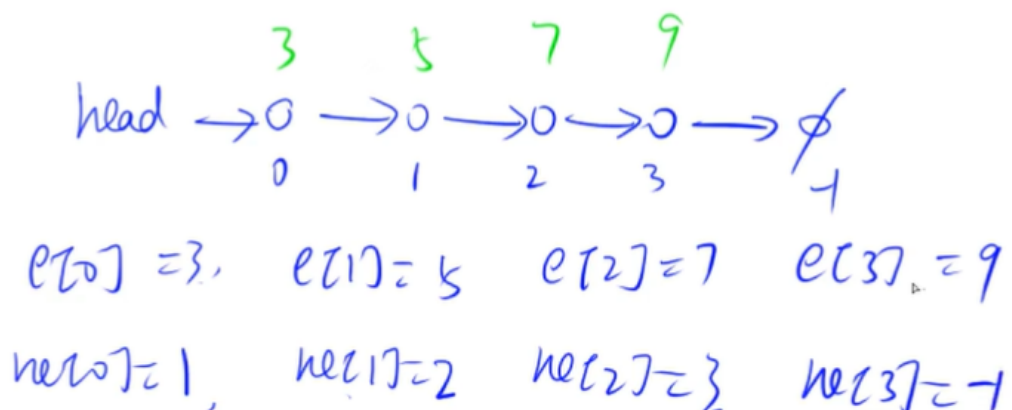
```
1 struct Node {  
2     int val;  
3     Node* next;  
4 };  
5 new Node();    //非常慢
```

数组模拟链表

- 单链表：邻接表：存储树和图



```
1 val: e[N];  
2 next指针: ne[N];  
3 e和ne用下标关联  
4 空节点下标用-1表示
```



静态链表

算法题不需要考虑内存泄漏问题，工程需要考虑内存问题

```
1 #include<iostream>  
2 using namespace std;  
3 const int N = 1E5+10;
```

```

4 // head表示头结点的下标
5 // e[i]表示节点i的值是多少
6 // ne[i]表示节点i的next下标是多少
7 // idx当前已经用到了哪个点
8 int head, e[N], ne[N], idx;
9 // 链表初始化
10 void init()
11 {
12     idx = 0;
13     head = -1;
14 }
15 // 头插法(将x插入到头结点)
16 void add_to_head(int x)
17 {
18     // e[idx] = x, ne[idx] = head, head = idx++;
19     ne[idx] = head; // 待插入的节点指向头结点
20     head = idx; // head 指向当前待插入的节点
21     e[idx] = x; // 更新值
22     idx++; // 更新下一个可用的节点的位置
23 }
24 // 将x插入到下标为k的后面
25 void add_to_k(int x, int k)
26 {
27     // e[idx] = x, ne[idx] = ne[k], ne[k] = idx++;
28     ne[idx] = ne[k];
29     ne[k] = idx;
30     e[idx] = x;
31     idx++;
32 }
33 // 将下标是k的点的后面的点删掉
34 void remove(int k)
35 {
36     ne[k] = ne[ne[k]]; //令k指向k->next->next
37 }
38 int main()
39 {
40     int m;
41     cin >> m;
42     init();
43     while(m--)
44     {
45         int x;
46         char op;
47         cin >> op;
48         if(op == 'H')
49         {
50             cin >> x;
51             add_to_head(x);
52         }
53         else if(op == 'D')
54         {
55             cin >> k;
56             if(k == 0)
57             {
58                 head = ne[head];
59             }

```

```

60         remove(k-1);
61     }
62     else
63     {
64         cin >> k >> x;
65         add(k-1, x);
66     }
67 }
68 for(int i = head; i!=-1; i = ne[i]) cout<<e[i]<<' ';
69 cout<<endl;
70 return 0;
71 }

```

- 双链表：优化某些问题



`int l[N], r[N];`

```

1  下标0表示head, 下标1表示tail
2  int e[N], l[N], r[N], idx;

```

```

1  #include<iostream>
2  using namespace std;
3  const int N = 100010;
4  int m;
5  int e[N], l[N], r[N], idx;
6  // 初始化
7  void init()
8  {
9      // 0表示左端点, 1表示右端点
10     r[0] = 1;
11     l[1] = 0;
12     idx = 2;
13 }
14 // 插入一个点到k的右边
15 void add_k_right(int x, int k)
16 {
17     e[idx] = x;
18     r[idx] = r[k];
19     l[idx] = k;
20     l[r[k]] = idx; //这里不能换顺序, 必须先处理r[k]的左指针
21     r[k] = idx;
22     idx++;
23 }
24 // 插入一个点到k的左边 其实就是 add_k_right(x, l[k]);
25 void add_k_left(int x, int k)
26 {

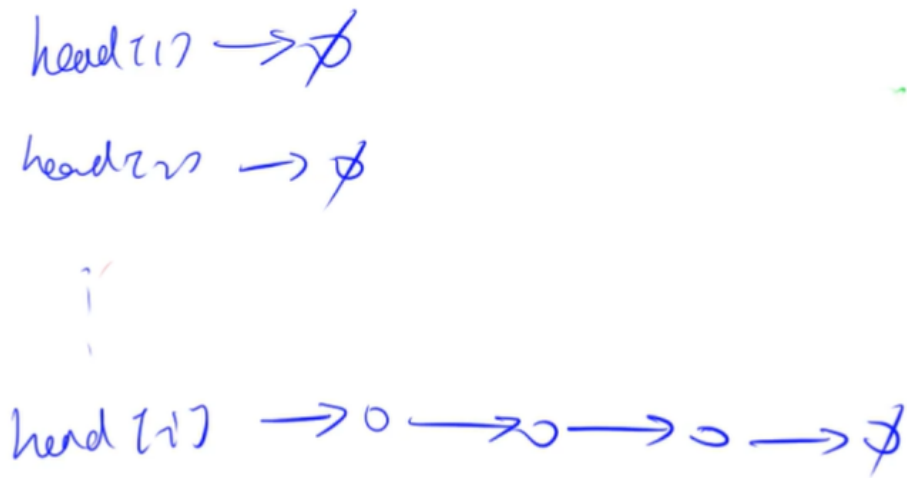
```

```

27     e[idx] = x;
28     r[idx] = k;
29     l[idx] = l[k];
30     r[l[k]] = idx;
31     l[k] = idx;
32     idx++;
33 }
34 // 删除第k个点
35 void remove(int k)
36 {
37     r[l[k]] = r[k];
38     l[r[k]] = l[k];
39 }
40

```

邻接表：n个单链表

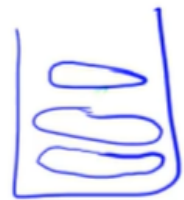


### 栈与队列

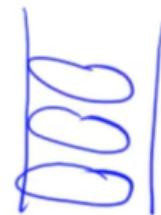
栈：先进后出（先插入的元素会最后被弹出）

队列：先进先出FIFO

栈 先进后出



队列 先进先出



## 栈stack

```
1  #include<iostream>
2  using namespace std;
3
4  const int N = 100010;
5  int stk[N], tt;
6  //插入
7  stk[++tt] = x;
8  // 删除
9  tt--;
10 // 判断是否为空
11 if(tt > 0) 不空
12 else empty
13 // 取栈顶
14 stk[tt];
```

## 队列queue

```
1  #include<iostream>
2  using namespace std;
3  const int N = 100010;
4  // hh 表示队头，在队头弹出元素，tt表示队尾，在队尾插入元素
5  int q[N], hh, tt = -1 ;
6  // 插入
7  q[++tt] = x;
8  // 弹出
9  hh ++;
10 // 空
11 if(hh <= tt) not empty
12     else empty
13 // 取队头元素
14 q[hh];
15 // 取队尾
16 q[tt];
17
```

## 单调栈

序列左边离它最近的比他小的数

```
1  #include<iostream>
2  using namespace std;
3  const int N = 100010;
4  int n;
5  int stk[N], tt = 0;
6  int main(void)
7  {
8      // scanf("%d", &n);
9      cin.tie(0);
10     ios::sync_with_stdio(false);
11     cin >> n;
12     for(int i = 0; i < n; i++)
13     {
14         int x;
```

```

15     cin >> x;
16     while(tt && stk[tt] >= x) tt--;
17     if(tt) cout << stk[tt] << endl;
18     else cout << -1;
19     stk[++tt] = x;
20 }
21 return 0;
22 }

```

scanf和printf要比cin, cout快不少

这个算法是 $O(n)$  的

## 单调队列

求滑动窗口的最大值和最小值

先考虑怎么用栈和队列来模拟这个问题, 然后再考虑朴素暴力算法下, 栈和队列中哪些元素是没有用到的, 是否可以删去, 再看一下是否有单调性, 如果有单调性, 取最值可以直接取端点, 找值可以用二分。

```

1  #include<iostream>
2  using namespace std;
3  const int N = 100010;
4  int stk[N], tt = 0;
5  int a[N], q[N];
6  int main(void)
7  {
8      scanf("%d%d", &n, &k);
9      for(int i = 0; i < n; i++)
10         scanf("%d", &a[i]);
11     int hh = 0, tt = -1;
12     for(int i = 0; i < n; i++)
13     {
14         // 判断队头是否已经滑出窗口
15         if(hh <= tt && i-k+1 > q[hh]) hh++;
16         while(hh <= tt && a[q[tt]] >= a[i]) tt--;
17         q[++tt] = i;
18         if(i >= k-1) printf("%d ", a[q[hh]]);
19     }
20     puts("");
21     int hh = 0, tt = -1;
22     for(int i = 0; i < n; i++)
23     {
24         // 判断队头是否已经滑出窗口
25         if(hh <= tt && i-k+1 > q[hh]) hh++;
26         while(hh <= tt && a[q[tt]] <= a[i]) tt--;
27         q[++tt] = i;
28         if(i >= k-1) printf("%d ", a[q[hh]]);
29     }
30     puts("");
31     return 0;
32 }

```

## KMP

朴素做法

1. 暴力算法怎么做
2. 如何去优化

```
S[N], p[M]
for (int i = 1; i <= n; i ++ )
{
    bool flag = true;
    for (int j = 1; j <= m; j ++ )
        if (s[i] != p[j])
        {
            flag = false;
            break;
        }
}
```

(二)

## Trie 树

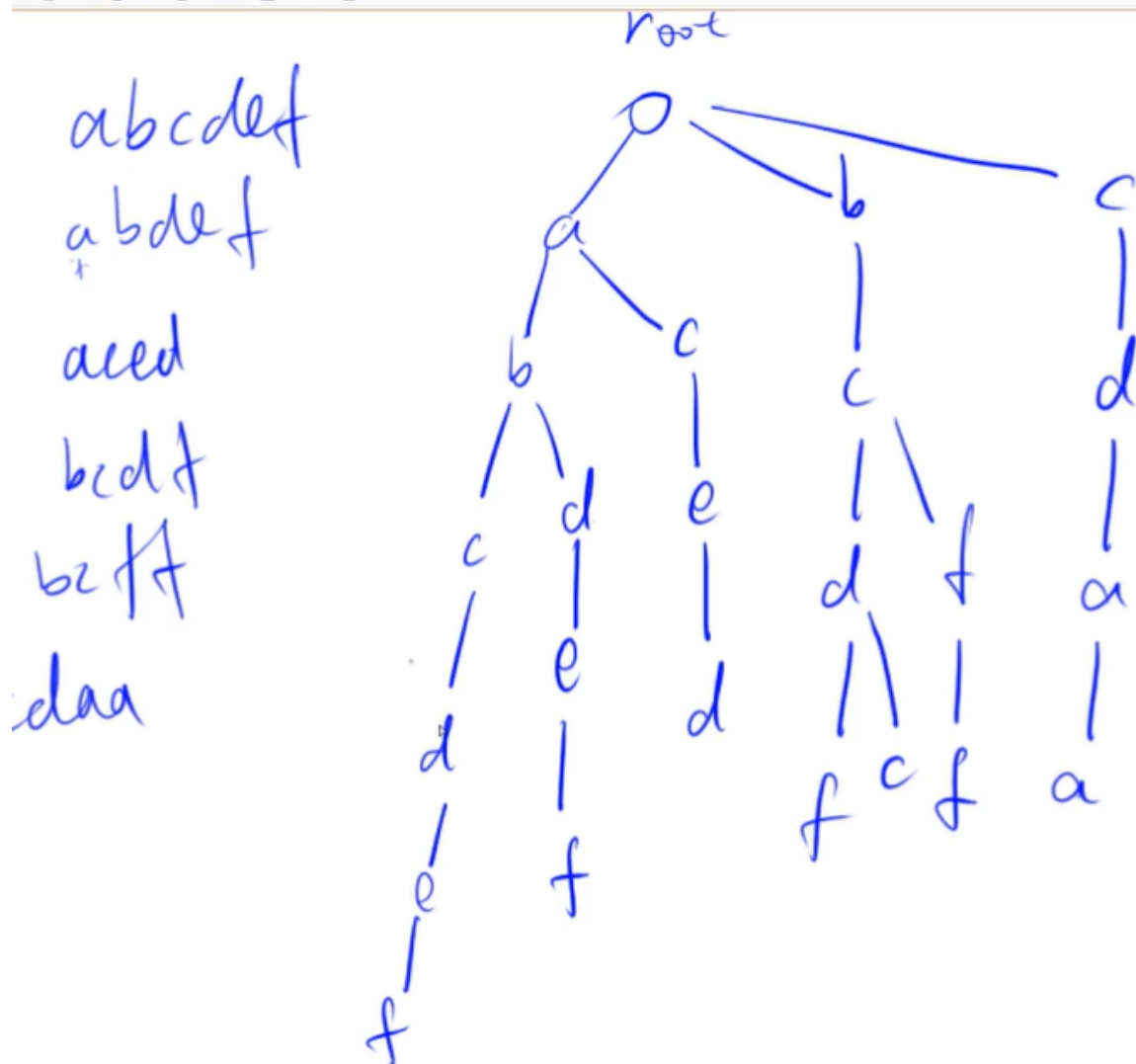
快速存储和查找字符串集合的数据结构

```
abcdef
abdef
aced
bcdff
bcff
cdaa
bcd
```

都是小写字母，或者都是大写，或者是数字等等

存储的方式：

有一个根节点root



会在每次单词结尾的地方打上一个标记，表示结尾

trie树的查找

存储一个字符串+查询一个字符串出现了多少次

```

1  #include<iostream>
2  using namespace std;
3  const int N = 1e5+10;
4  int son[N][26], cnt[N], idx;    //下标是0的点，既是根节点又是空节点
5  // son[N][26] 表示一个节点最多有26个子节点（对应26个字母），cnt[p]表示节点p处有多少个结
   尾的单词，存的是数组模拟的指针
6  void insert(char str[])
7  {
8      int p = 0;
9      for(int i = 0; str[i]; i++)
10     {
11         int u = str[i] - 'a';
12         if(!son[p][u]) son[p][u] = ++idx;
13         p = son[p][u];
14     }
15     cnt[p] ++;
16 }
17 int query(char str[])
18 {
19     int p = 0;

```



```

20     for(int i = 0; str[i]; i++)
21     {
22         int u = str[i] - 'a';
23         if(!son[p][u]) return 0;
24         p = son[p][u];
25     }
26     return cnt[p];
27 }
28 int main(void)
29 {
30     int n ;
31     scanf("%d", &n);
32     while(n--)
33     {
34         char op[2];
35         scanf("%s%s", op, str);
36         if(op[0] == '1') insert(str);
37         else printf("%d ", query(str));
38     }
39     return 0;
40 }

```

## 并查集

快速处理

- 将两个集合合并
- 询问两个元素是否在一个集合当中

$\text{belong}[x] = a;$

$\text{if } (\text{belong}[x] == \text{belong}[y]) \quad O(1)$

将合并集合的时间从  $O(n)$  降到近乎  $O(1)$

基本原理

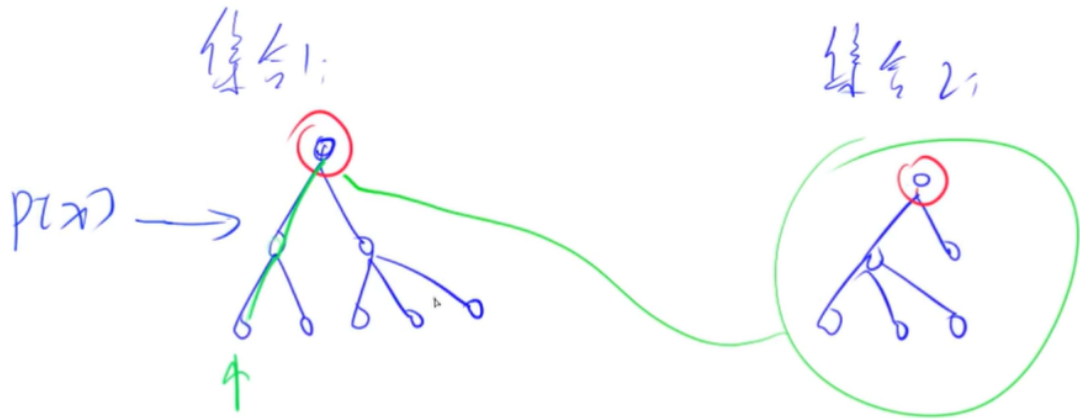
用树的形式维护每一个集合，每个集合的编号是根节点的编号

对每个节点存储父节点是谁 (father)  $p[x]$  表示  $x$  的父节点

求每个节点属于某个集合只需要往回遍历 father，直到树根就是集合的编号

- 判断树根:  $p[x] == x$ ，令树根的 father 为它本身
- 求  $x$  的集合编号:  $\text{while}(p[x] \neq x) \quad x = p[x];$

- 如何合并两个集合：



将右边树的根节点插入左边树的某个位置

$p[x] = y$

并查集的优化：（路径压缩）

一旦找到了根节点，把整个路径上的点都指向根节点

```

1  #include<iostream>
2  using namespace std;
3  const int N = 100010;
4  int p[N];    //father数组 p[x] = x 表示树根
5  int n, m;
6  int find(int x)    // 返回x的祖宗节点 + 路径压缩
7  {
8      if( p[x] != x) p[x] = find(p[x]);
9      return p[x];
10 }
11 int main(void)
12 {
13     scanf("%d%d", &n, &m);
14     for(int i = 1; i <= n; i++) p[i] = i;
15     while(m--)
16     {
17         char op[2];
18         int a, b;
19         scanf("%s%d%d", op, &a, &b);
20         if(op[0] == 'M') p[find(a)] = find(b);
21         else
22         {
23             if(find(a) == find(b))
24                 printf("YES\n");
25             else printf("NO\n");
26         }
27     }
28     return 0;
29 }

```

scanf读取字符串的时候会自动忽略，如果用%c，可能输入会在行末加一个空格导致错误

## 有额外信息的并查集

并查集：

1. 将两个集合合并
2. 询问两个元素是否在一个集合当中

基本原理：每个集合用一棵树来表示。树根的编号就是整个集合的编号。每个节点存储它的父节点， $p[x]$ 表示 $x$ 的父节点

问题1：如何判断树根：if ( $p[x] == x$ )

问题2：如何求 $x$ 的集合编号：while ( $p[x] != x$ )  $x = p[x]$ ;

问题3：如何合并两个集合： $px$  是 $x$ 的集合编号， $py$ 是 $y$ 的集合编号。 $p[x] = y$

## 堆

如何手写一个堆？

下标从1开始，从零开始不是很方便

- 插入一个数

```
1 | head[++size] = x;  
2 | up(size);
```

- 求集合中的最小值

```
1 | heap[1]
```

- 删除最小值

```
1 | heap[1] = heap[size]; size--;  
2 | down(1);
```

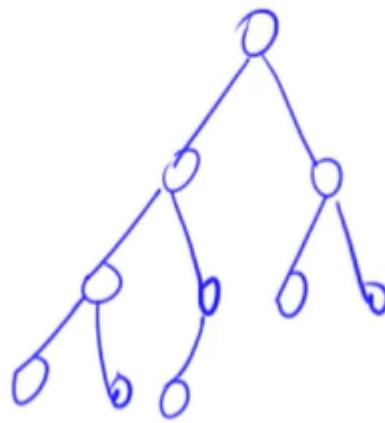
- 删除任意一个元素

```
1 | heap[k] = heap[size]; size--;  
2 | down(k);  
3 | up(k);
```

- 修改任意一个元素

```
1 | heap[k] = x; down(k);
```

完全二叉树



每一个点的值都是小于等于孩子节点的值。

存储

用一个一维数组存储

1号点是根节点

- $2x$ 是左儿子
- $2x+1$ 是右儿子

- |              |  |
|--------------|--|
| 1. 插入一个数     | <code>heap[ ++ size] = x; up(size);</code>                   |
| 2. 求集合当中的最小值 | <code>heap[1];</code>  |
| 3. 删除最小值     | <code>heap[1] = heap[size]; size -- ; down(1);</code>        |
| 4. 删除任意一个元素  | <code>heap[k] = heap[size]; size -- ; down(k); up(k);</code> |
| 5. 修改任意一个元素  | <code>heap[k] = x; down(k); up(k);</code>                    |

```
1  #include<iostream>
2  #include<algorithm>
3  using namespace std;
4  const int N = 100010;
5  int n, m;
6  int hp[N], ph[N];
7  int size;
8  void down(int u)
9  {
10     int t = u;
11     if(u*2 <= size && h[u*2] < h[t]) t = 2*u;
12     if(u*2 + 1 <= size && h[u*2 + 1] < h[t]) t = 2*u + 1;
13     if(u != t)
14     {
15         swap(h[u], h[t]);
16         down(t);
17     }
18 }
19 void up(int u)
20 {
21     while( u/2 && h[u/2] > h[u])
22     {
23         swap(h[u/2], h[u]);
24     }
```

```

25 }
26 void heap_swap(int a, int b)
27 {
28     swap(ph[hp[a]], ph[hp[b]]);
29     down(t);
30 }
31 int main(void)
32 {
33     scanf("%d", &n);
34     for(int i = 1; i <= n; i++) scanf("%d", &n);
35     size = n;
36     for(int i = n/2 ; i; i --) down(i);
37     while(m--)
38     {
39         printf("%d ", h[1]);
40         h[1] = h[size];
41         size--;
42         down(1);
43     }
44     return 0;
45 }

```

### (三)

## 哈希表

哈希表

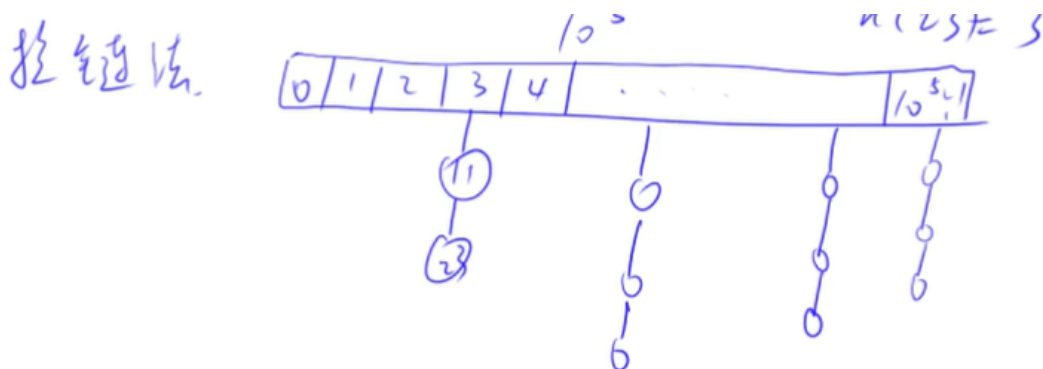
- 存储结构
  - 开放寻址法
  - 拉链法
- 字符串哈希的方式

哈希表：把一个比较大的值域映射到比较小的空间

哈希函数  $h(x) \in (0, 10^5)$

- $x \bmod 10^5$
- 冲突：两种方式，开放寻址法和拉链法

拉链法



算法题只有添加和查找这两个操作

- 删除可以开一个bool变量标记，而不是真正删除

代码

mod的数要取成一个质数，离2的整数幂尽可能远

```
1  #include<iostream>
2  #include<cstring>
3  using namespace std;
4  const int N = 100003;
5  int h[N], e[N], ne[N], idx;
6  void insert(int x)
7  {
8      int k = (x % N + N) % N;
9      e[idx] = x, ne[idx] = h[k], h[k] = idx++;
10 }
11 bool find(int x)
12 {
13     int k = (x % N + N) % N;
14     for(int i = h[k]; i != -1; i = ne[i])
15         if(e[i] == x)
16             return true;
17     return false;
18 }
19 int main(void)
20 {
21     // 查找大于100000的第一个质数
22     for(int i = 100000; i++)
23     {
24         bool flag = true;
25         for(int j = 2; j*j <= i; j++)
26             if(i % j == 0)
27             {
28                 flag = false;
29                 break;
30             }
31         if(flag){
32             cout << i << endl;
33             break;
34         }
35     }
36     int n;
37     scanf("%d", &n);
38     memset(h, -1, sizeof(h));
39     while(n-->0)
40     {
41         char op[2];
42         int x;
43         scanf("%s%d", op, &x);
44         if(*op == 'I') insert(x);
45         else
46         {
47             if(find(x)) puts("Yes");
48             else puts("No");
49         }
50     }
```

## 开放寻址法

### 字符串的哈希方式

- 字符串的前缀哈希法

$Str = "ABZABCD EYX C Acwing"$   
 $h[0] = 0$   
 $h[1] = "A" \text{ 的 hash 值}$  ①  $p$  进制制  
 $h[2] = "AB" \text{ 的 hash 值}$   
 $h[3] = "ABC" \dots$   
 $h[4] = "ABCD" \dots$   
 $\vdots$   
 $"ABCD"$

① A B C D  $0 \sim Q-1$   
 C 1 2 3 4  $p$

$$= (1 \times p^3 + 2 \times p^2 + 3 \times p^1 + 4 \times p^0) \bmod Q.$$

不能映射成0，从1开始比较好

假定不存在冲突， $p = 131$ 或 $13331$ ， $Q = 2^{64}$

## 算法模版

### 单链表

#### 数组实现

单链表需要有一个head表示头结点的指向，而后需要两个数组e[n]和ne[n]分别存储每个节点的value和next指针，还有一个idx表示当前可用的指针在数组中的位置。

这是一个静态链表，在算法中只需要考虑速度，而不需要考虑内存泄漏，因此删除节点也不需要回收数组中可用的位置，这样可以在链表任意一个位置k插入节点，前提是需要直到这个节点在数组中的下标。

```

1 //head存储表头，e[i]存储节点i的值，ne[i]存储节点i的next指针，idx表示当前用到了哪个节点
2 const int N = 100010;
3 int head, e[N], ne[N], idx;
4 void init()
5 {
6     head = -1;
7     idx = 0;
8 }
  
```

```

9 void insert_to_head(int x)
10 {
11     e[idx] = x;
12     ne[idx] = head;
13     head = idx;
14     idx++;
15 }
16 // 将x插入在第k个节点之后
17 void insert_to_k(int x, int k)
18 {
19     e[idx] = x;
20     ne[idx] = ne[k];
21     ne[k] = idx++;
22 }
23 // 将头结点删除
24 void remove_head()
25 {
26     head = ne[head];
27 }
28 // 将第k个节点之后的节点删除
29 void remove_k(int k)
30 {
31     ne[k] = ne[ne[k]];
32 }

```

## 双链表

数组实现，需要e[N]数组表示value，l[N]左指针，r[N]表示右指针，idx表示当前可用的节点在数组中的位置

```

1  const int N = 100010;
2  int e[N], r[N], l[N], idx;
3  void init()
4  {
5      // 规定0是左端点，1是右端点
6      r[0] = 1;
7      l[1] = 0;
8      idx = 2;
9  }
10 // 在节点k的右边插入一个数x
11 void insert_k_right(int k, int x)
12 {
13     e[idx] = x;
14     l[idx] = k;
15     r[idx] = r[k];
16     l[r[k]] = idx;
17     r[k] = idx++;
18 }
19 // 在节点k的左边插入一个数x，其实可以用 insert_k_right(l[k], x);
20 void insert_k_left(int k, int x)
21 {
22     e[idx] = x;
23     l[idx] = l[k];
24     r[idx] = k;
25     r[l[k]] = idx;

```



```

26     l[k] = idx++;
27 }
28 // 删除节点k
29 void remove(int k)
30 {
31     l[r[k]] = l[k];
32     r[l[k]] = r[k];
33 }

```

## 栈

```

1 // tt表示栈顶
2 int stk[N], tt = 0;
3 // 入栈
4 stk[++tt] = x;
5 // 出栈
6 tt--;
7 // 栈顶的值 (top)
8 stk[tt];
9 // 栈是否为空
10 if(tt > 0) not empty
11 else empty

```

## 应用

### 计算中缀表达式

计算机通过使用两个栈（操作数栈和操作符栈）来计算中缀表达式的方法。以下是计算机求解中缀表达式的具体步骤，对照着代码学会非常清晰：

1. 从左到右扫描中缀表达式。
2. 如果遇到操作数（数字），则将其压入操作数栈。
3. 如果遇到操作符（如 +, -, \*, /），执行以下操作：
  - a. 如果操作符栈为空或栈顶元素为左括号 (，则将操作符压入操作符栈。
  - b. 如果新操作符的优先级高于操作符栈顶的操作符，也将新操作符压入操作符栈。
  - c. 如果新操作符的优先级小于或等于操作符栈顶的操作符，从操作数栈中弹出两个操作数，从操作符栈中弹出一个操作符，执行相应的计算，并将结果压入操作数栈。然后，将新操作符压入操作符栈。重复此过程，直到新操作符可以被压入操作符栈。
4. 如果遇到左括号 (，将其压入操作符栈。
5. 如果遇到右括号 )，重复执行以下操作，直到遇到左括号 (：
  - a. 从操作数栈中弹出两个操作数。
  - b. 从操作符栈中弹出一个操作符。
  - c. 执行相应的计算，并将结果压入操作数栈。
  - d. 在执行完这些操作后，弹出操作符栈顶的左括号 (。
6. 当扫描完整个中缀表达式后，如果操作符栈仍然包含操作符，重复执行以下操作，直到操作符栈为空：
  - a. 从操作数栈中弹出两个操作数。
  - b. 从操作符栈中弹出一个操作符。
  - c. 执行相应的计算，并将结果压入操作数栈。
7. 操作数栈中剩余的最后一个元素就是中缀表达式的计算结果。

```

1  #include <iostream>
2  #include <cstring>
3  #include <algorithm>
4  #include <stack>
5  #include <unordered_map>
6
7  using namespace std;
8
9  stack<int> num;
10 stack<char> op;
11
12 void eval()
13 {
14     auto b = num.top(); num.pop();
15     auto a = num.top(); num.pop();
16     auto c = op.top(); op.pop();
17     int x;
18     if (c == '+') x = a + b;
19     else if (c == '-') x = a - b;
20     else if (c == '*') x = a * b;
21     else x = a / b;
22     num.push(x);
23 }
24
25 int main()
26 {
27     unordered_map<char, int> pr{{'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}};
28     string str;
29     cin >> str;
30     for (int i = 0; i < str.size(); i++)
31     {
32         auto c = str[i];
33         if (isdigit(c))
34         {
35             int x = 0, j = i;
36             while (j < str.size() && isdigit(str[j]))
37                 x = x * 10 + str[j++] - '0';
38             i = j - 1;
39             num.push(x);
40         }
41         else if (c == '(') op.push(c);
42         else if (c == ')')
43         {
44             while (op.top() != '(') eval();
45             op.pop();
46         }
47         else
48         {
49             while (op.size() && op.top() != '(' && pr[op.top()] >= pr[c])
50                 eval();
51             op.push(c);
52         }
53     }
54     while (op.size()) eval();
55     cout << num.top() << endl;
56     return 0;

```

## 队列

### 普通队列

```

1 // hh 表示队头， tt表示队尾
2 int q[N], hh = 0, tt = -1;
3 // 向队尾插入一个数
4 q[ ++ tt] = x;
5 // 从队头弹出一个数
6 hh ++;
7 // 队头的值
8 q[hh];
9 // 判断队列是否为空，如果 hh <= tt ，则表示不为空

```

### 循环队列

```

1 // hh 表示队头， tt表示队尾的最后一个位置
2 int q[N], hh = 0, tt = 0;
3 // 向队尾插入一个数
4 q[tt ++ ] = x;
5 if(tt == N) tt = 0;
6 // 从队头弹出一个数
7 hh ++;
8 if(hh == N) hh = 0;
9 // 队头的值
10 q[hh];
11 // 判断队列是否为空，如果hh != tt，则表示不为空
12 if( hh != tt)
13 {
14
15 }

```

## 单调栈

```

1 常见模型：找出每个数左边比它最近的比它大/小的数
2 int tt = 0;
3 for(int i = 1; i <= n ;i++)
4 {
5     while(tt && check(stk[tt], i)) tt --;
6     stk[++ tt] = i;
7 }

```

```

1 #include <iostream>
2
3 using namespace std;
4
5 const int N = 100010;
6
7 int stk[N], tt;
8
9 int main()

```

```

10 {
11     int n;
12     cin >> n;
13     while (n -- )
14     {
15         int x;
16         scanf("%d", &x);
17         while (tt && stk[tt] >= x) tt -- ;
18         if (!tt) printf("-1 ");
19         else printf("%d ", stk[tt]);
20         stk[ ++ tt] = x;
21     }
22
23     return 0;
24 }

```

## 单调队列

```

1  常见模型：找出滑动窗口中的最大值/最小值
2  int hh = 0, tt = -1;
3  int q[N];
4  for(int i = 0; i < n; i++)
5  {
6      while(hh <= tt && check_out(q[hh])) hh++; // 判断队头是否滑出窗口
7      while(hh <= tt && check(q[tt]), i) tt--;
8      q[ ++ tt] = i;
9  }

```

## KMP

```

1  // s[]是长文本， p[]是模式串， n是s的长度， m是p的长度，求模式串的next数组
2  for(int i = 2, j = 0; i <= m; i++)
3  {
4      while(j && p[i] != p[j+1]) j = ne[j];
5      if(p[i] == p[j+1]) j++;
6      ne[i] = j;
7  }
8  // 匹配
9  for(int i = 1, j = 0; i <= n; i++)
10 {
11     while(j && s[i] != p[j+1]) j = ne[j];
12     if(s[i] == p[j+1]) j++;
13     if(j == m)
14     {
15         j = ne[j];
16         // 匹配成功后的逻辑
17     }
18 }

```

```

1  #include<iostream>
2  using namespace std;
3  const int N = 1e5 + 10;
4  const int M = 1e6 + 10;
5  int ne[N];

```

```

6 char p[N], s[M];
7 int n, m;
8 int main(void)
9 {
10     cin >> n >> p+1 >> m >> s+1 ;
11     for(int i = 2, j = 0; i<= n; i++)
12     {
13         while(j && p[i] != p[j+1]) j = ne[j];
14         if(p[i] == p[j+1])j++;
15         ne[i] = j;
16     }
17     for(int i = 1, j = 0;i<= m; i++)
18     {
19         while(j && s[i]!=p[j+1]) j = ne[j];
20         if(s[i] == p[j+1]) j++;
21         if(j == n) {
22             printf("%d ",i-n);
23             j = ne[j];
24         }
25     }
26     return 0;
27 }

```

## Trie 树

```

1 int son[N][26], cnt[N], idx;
2 // 0号点既是根节点，又是空节点
3 // son[][]存储树中每个节点的子节点
4 // cnt[] 存储以每个节点结尾的单词数量
5
6 // 插入一个字符串
7 void insert(char *str)
8 {
9     int p = 0;
10    for(int i = 0; str[i]; i++)
11    {
12        int u = str[i] - 'a';
13        if(!son[p][u]) son[p][u] = ++idx;
14        p = son[p][u];
15    }
16    cnt[p] ++;
17 }
18 int query(char * str)
19 {
20     int p = 0;
21     for(int i = 0; str[i]; i++)
22     {
23         int u = str[i] - 'a';
24         if(!son[p][u]) return 0;
25         p = son[p][u];
26     }
27     return cnt[p];
28 }

```

## 最大异或对

```
1  #include <iostream>
2  #include <algorithm>
3
4  using namespace std;
5
6  const int N = 100010, M = 3100010;
7
8  int n;
9  int a[N], son[M][2], idx;
10
11 void insert(int x)
12 {
13     int p = 0;
14     for (int i = 30; i >= 0; i -- )
15     {
16         int &s = son[p][x >> i & 1];
17         if (!s) s = ++ idx;
18         p = s;
19     }
20 }
21
22 int search(int x)
23 {
24     int p = 0, res = 0;
25     for (int i = 30; i >= 0; i -- )
26     {
27         int s = x >> i & 1;
28         if (son[p][!s])
29         {
30             res += 1 << i;
31             p = son[p][!s];
32         }
33         else p = son[p][s];
34     }
35     return res;
36 }
37
38 int main()
39 {
40     scanf("%d", &n);
41     for (int i = 0; i < n; i ++ )
42     {
43         scanf("%d", &a[i]);
44         insert(a[i]);
45     }
46
47     int res = 0;
48     for (int i = 0; i < n; i ++ ) res = max(res, search(a[i]));
49
50     printf("%d\n", res);
51
52     return 0;
53 }
```

## 并查集

```
1  (1) 朴素并查集
2  int p[N];    //存储每个点的祖宗节点
3  // 返回x的祖宗节点
4  int find(int x)
5  {
6      if(p[x] != x) p[x] = find(p[x]);
7      return p[x];
8  }
9  // 初始化, 假定节点编号是1~n
10 for(int i = 1; i <= n; i++) p[i] = i;
11 // 合并a和b所在的两个集合
12 p[find(a)] = find(b);
13 (2) 维护size的并查集:
14 int p[N], size[N];
15 //p[] 存储每个点的祖宗节点, size[]只有祖宗节点的有意义, 表示祖宗节点所在集合中的点的数量
16 // 返回x的祖宗节点
17 int find(int x)
18 {
19     if(p[x] != x) p[x] = find(p[x]);
20     return p[x];
21 }
22 // 初始化, 假定节点编号是1~n
23 for(int i = 1; i <= n; i++)
24 {
25     p[i] = i;
26     size[i] = 1;
27 }
28 // 合并a和b所在的两个集合
29 size[find(b)] += size[find(a)];
30 p[find(a)] = find(b);
31 (3)维护到祖宗节点距离的并查集:
32 int p[N], d[N];
33 //p[]存储每个点的祖宗节点, d[x]存储x到p[x]的距离
34 // 返回x的祖宗节点
35 int find(int x)
36 {
37     if(p[x] != x){
38         int u = find(p[x]);
39         d[x] += d[p[x]];
40         p[x] = u;
41     }
42     return p[x];
43 }
44 // 初始化, 假定节点编号是1~n
45 for(int i = 1; i <= n; i++){
46     p[i] = i;
47     d[i] = 0;
48 }
49 // 合并a和b所在的两个集合:
50 p[find(a)] = find(b);
51 d[find(a)] = distance;
```

动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。

A 吃 B，B 吃 C，C 吃 A。

现有 N 个动物，以 1~N 编号。

每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

第一种说法是 1 X Y，表示 X 和 Y 是同类。

第二种说法是 2 X Y，表示 X 吃 Y。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。

当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

1. 当前的话与前面的某些真的话冲突，就是假话；
2. 当前的话中 X 或 Y 比 N 大，就是假话；
3. 当前的话表示 X 吃 X，就是假话。

你的任务是根据给定的 N 和 K 句话，输出假话的总数。

输入范围

第一行是两个整数 N 和 K，以一个空格分隔。

以下 K 行每行是三个正整数 D, X, Y，两数之间用一个空格隔开，其中 D 表示说法的种类。

若 D=1，则表示 X 和 Y 是同类。

若 D=2，则表示 X 吃 Y。

输出格式

只有一个整数，表示假话的数目。

数据范围

$1 \leq N \leq 50000$

$0 \leq K \leq 100000$

```
1  #include <iostream>
2
3  using namespace std;
4
5  const int N = 50010;
6
7  int n, m;
8  int p[N], d[N];
9
10 int find(int x)
11 {
12     if (p[x] != x)
13     {
14         int t = find(p[x]);
15         d[x] += d[p[x]];
```



```

16     p[x] = t;
17 }
18 return p[x];
19 }
20
21 int main()
22 {
23     scanf("%d%d", &n, &m);
24
25     for (int i = 1; i <= n; i ++ ) p[i] = i;
26
27     int res = 0;
28     while (m -- )
29     {
30         int t, x, y;
31         scanf("%d%d%d", &t, &x, &y);
32         // 假话2
33         if (x > n || y > n) res ++ ;
34         else
35         {
36             int px = find(x), py = find(y);
37             // 第一种说法
38             if (t == 1)
39             {
40                 // d[x] - d[y] % 3 != 0表示冲突
41                 if (px == py && (d[x] - d[y]) % 3) res ++ ;
42                 else if (px != py)
43                 {
44                     p[px] = py;
45                     d[px] = d[y] - d[x];
46                 }
47             }
48             // 第二种说法
49             else
50             {
51                 if (px == py && (d[x] - d[y] - 1) % 3) res ++ ;
52                 else if (px != py)
53                 {
54                     p[px] = py;
55                     d[px] = d[y] + 1 - d[x];
56                 }
57             }
58         }
59     }
60
61     printf("%d\n", res);
62
63     return 0;
64 }
65

```

```

1 // h[N]存储堆中的值，h[1]是堆顶，x的左儿子是2x，右儿子是2x+1
2 // ph[k]存储第k个插入的点在堆中的位置
3 // hp[k]存储堆中下标是k的点是第几个插入的
4 int h[N], ph[N], hp[N], size;
5 // 交换两个点，及其映射关系
6 void heap_swap(int a, int b)
7 {
8     swap(ph[hp[a]], ph[hp[b]]);
9     swap(hp[a], hp[b]);
10    swap(h[a], h[b]);
11 }
12 void down(int u)
13 {
14     int t = u;
15     if( 2*u <= size && h[2*u] < h[t]) t = 2*u;
16     if(2*u+1 <= size && h[2*u + 1] < h[t]) t = 2*u + 1;
17     if(u != t)
18     {
19         heap_swap(u, t);
20         down(t);
21     }
22 }
23 void up(int u)
24 {
25     while(u/2 && h[u] < h[u/2])
26     {
27         heap_swap(u, u/2);
28         u >>= 1;
29     }
30 }
31 // O(n) 建堆
32 for(int i = n/2; i; i--) down(i);

```

## 哈希

### 一般哈希

#### 开放寻址法

```

1 #include <cstring>
2 #include <iostream>
3
4 using namespace std;
5
6 const int N = 200003, null = 0x3f3f3f3f;
7
8 int h[N];
9
10 int find(int x)
11 {
12     int t = (x % N + N) % N;
13     while (h[t] != null && h[t] != x)
14     {

```

```

15     t ++ ;
16     if (t == N) t = 0;
17 }
18 return t;
19 }
20
21 int main()
22 {
23     memset(h, 0x3f, sizeof h);
24
25     int n;
26     scanf("%d", &n);
27
28     while (n -- )
29     {
30         char op[2];
31         int x;
32         scanf("%s%d", op, &x);
33         if (*op == 'I') h[find(x)] = x;
34         else
35         {
36             if (h[find(x)] == null) puts("No");
37             else puts("Yes");
38         }
39     }
40
41     return 0;
42 }

```

## 拉链法

```

1  #include <cstring>
2  #include <iostream>
3
4  using namespace std;
5
6  const int N = 100003;
7
8  int h[N], e[N], ne[N], idx;
9
10 void insert(int x)
11 {
12     int k = (x % N + N) % N;
13     e[idx] = x;
14     ne[idx] = h[k];
15     h[k] = idx ++ ;
16 }
17
18 bool find(int x)
19 {
20     int k = (x % N + N) % N;
21     for (int i = h[k]; i != -1; i = ne[i])
22         if (e[i] == x)
23             return true;
24 }

```

```

25     return false;
26 }
27
28 int main()
29 {
30     int n;
31     scanf("%d", &n);
32
33     memset(h, -1, sizeof h);
34
35     while (n -- )
36     {
37         char op[2];
38         int x;
39         scanf("%s%d", op, &x);
40
41         if (*op == 'I') insert(x);
42         else
43         {
44             if (find(x)) puts("Yes");
45             else puts("No");
46         }
47     }
48
49     return 0;
50 }

```

```

1 // 拉链法
2 int h[N], e[N], ne[N], idx;
3 // 向hash表中插入一个数
4 void insert(int x)
5 {
6     int k = (x % N + N) % N;
7     e[idx] = x;
8     ne[idx] = h[k];
9     h[k] = idx++;
10 }
11 // 在hash表中查询某个数是否存在
12 bool find(int x)
13 {
14     int k = (x % N + N) % N;
15     for(int i = h[k]; i != -1; i = ne[i])
16         if(e[i] == x)
17             return true;
18     return false;
19 }
20 // 开放寻址法
21 int h[N];
22 // 如果x在hash表中, 返回x的下标, 如果x不在hash表中, 返回x应该插入的位置
23 int find(int x)
24 {
25     int t = (x % N + N) % N;
26     while(h[t] != null && h[t] != x)

```

```

27     {
28         t ++;
29         if(t == N) t = 0;
30     }
31     return t;
32 }

```

## 字符串hash

```

1  核心理想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低
2  小技巧：取模的数用2^64，这样直接用unsigned long long存储，溢出的结果就是取模的结果
3  typedef unsigned long long ULL;
4  ULL h[N], p[N];
5  p[0] = 1;
6  for(int i = 1; i <= n; i++)
7  {
8      h[i] = h[i-1] * P + str[i];
9      p[i] = p[i-1] * P;
10 }
11 // 计算子串str[l~r]的hash值
12 ULL get(int l, int r)
13 {
14     return h[r] - h[l-1] * p[r - l + 1]
15 }

```

```

1  #include <iostream>
2  #include <algorithm>
3
4  using namespace std;
5
6  typedef unsigned long long ULL;
7
8  const int N = 100010, P = 131;
9
10 int n, m;
11 char str[N];
12 ULL h[N], p[N];
13
14 ULL get(int l, int r)
15 {
16     return h[r] - h[l - 1] * p[r - l + 1];
17 }
18
19 int main()
20 {
21     scanf("%d%d", &n, &m);
22     scanf("%s", str + 1);
23
24     p[0] = 1;
25     for (int i = 1; i <= n; i ++ )
26     {
27         h[i] = h[i - 1] * P + str[i];
28         p[i] = p[i - 1] * P;
29     }

```

```
30
31     while (m -- )
32     {
33         int l1, r1, l2, r2;
34         scanf("%d%d%d%d", &l1, &r1, &l2, &r2);
35
36         if (get(l1, r1) == get(l2, r2)) puts("Yes");
37         else puts("No");
38     }
39
40     return 0;
41 }
```