其他数据结构

并查集

`p[x] = x`

- 判断x是否为树根：`p[x] == x`
- find 求x的集合编号：`while(p[x] != x) x = p[x];`
- union 合并集合

```
int  find (int a) {
    int pa;
    for (pa = p[a]; pa != p[pa]; pa = p[a]) p[a] = p[pa];
    return pa;
}
void uunion(int a, int b) { p[find(a)] = find(b); }
```

优化：路径压缩

```
// recursive
int find(int x) {
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

// non-recursive
int find(int x) {
    // 836.cpp中的写法似乎做不到路径上所有节点的压缩，并不对
    // 需要记录沿途的所有节点
    while (p[x] != x) stack.push(x), x = p[x];
    while (!stack.empty()) p[stack.pop()] = x;
}
```

应用：
T1250 格子游戏

- 加入一条边后成环 $\iff$ 在加入边前，边的两个点已经在一个集合中
  T240 食物链：**记录距离**
- 并查集不仅存同类关系，被吃关系也一并入；但同时记录点之间的关系
- 每个点到根节点的距离：用于表示关系中的分类（同类/被吃），模3即循环
  - 距离为 1：吃根
  - 距离为 2：被根吃
  - 距离为 0：同类

```cpp
const int N = 5e4 + 2;
int p[N], d[N]; // d[x]: dance of x to the root in its union-find set
// d[x] % 3 == 1: root eat x
// d[x] % 3 == 2: x eat root
// d[x] % 3 == 0: x equal root
int n, k;
#define MOD3_POSI(x) (((x) % 3 + 3) % 3)
// C++ mod: res has the same sign with divident a ( a % b = res)
// -5 %  3 = -2 : -5+  3 =-2
// -5 % -3 = -2 : -5-(-3)=-2
//  5 % -3 =  2 :  5+(-3)= 2
//  5 %  3 =  2 :  5-  3 = 2

void init(int size) { for (int i = 0; i < size; ++i) p[i] = i; }

int find(int x) {
    if (x != p[x]) {
        int tmp = p[x];
        p[x] = find(p[x]);
        d[x] += d[tmp];
    }
    return p[x];
}

bool x_equal_y(int x, int y) { return (d[x] - d[y]) % 3 == 0; }
bool x_eat_y(int x, int y) { return MOD3_POSI(d[x] - d[y]) == 2; }

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

    int op, x, y;
    int cnt = 0;
    while (k--) {
        scanf("%d%d%d", &op, &x, &y);
        if (x > n || y > n) { cnt++; continue; }
        int px = find(x), py = find(y);
        if (op == 1) { // x equals y
            if (px == py) {
                if (!x_equal_y(x, y)) { cnt++; continue; }
            } else {
                p[px] = py;  // d[x] + d[px] - d[y] ≡ 0 (mod 3) => d[px] ≡
d[y] - d[x] (mod 3)
                d[px] = d[y] - d[x];
            }
        } else if (op == 2) { // x eat y
            if (x == y) { cnt++; continue; }
            if (px == py) {
                if(!x_eat_y(x, y)) { cnt++; continue; }
```

```
            } else {
                p[px] = py; // d[x] + d[px] - d[y] ≡ 2 (mod 3)
                d[px] = d[y] - d[x] + 2;
            }
        }
    }
    printf("%d\n", cnt);
    return 0;
}
```

堆

支持操作

- 插入数
- 获得最值
- 删除最值
- 删除堆中任意数（STL无）
- 修改堆中任意数（STL无）
  存储方式：即一维连续数组

模板

```
const int N = 1e5 + 2;

int h[N], hsize;
int k;
int ph[N];
// ph[k]: p to h, the kth insert element's loc in heap; ph[k] = i
int hp[N];
// hp[i]: h to p, the insert sequence of element in h[i]   ; hp[i] = k

void heap_swap(int i, int j) {
    ph[hp[i]] = j, ph[hp[j]] = i;
    std::swap(hp[i], hp[j]);
    std::swap(h[i], h[j]);
}

void down(int x) {  // 通用：下溢
    int t = x, sx = x << 1;
    if (sx <= hsize && h[sx] < h[t]) t = sx;
    if (sx+1 <= hsize && h[sx+1] < h[t]) t = sx+1;
    if (t != x) {
        heap_swap(t, x);
        down(t);
    }
}
```

```c
void up(int x) {    // 通用：上溢
    int fx;
    while ((fx = (x >> 1)) && h[fx] > h[x]) {
        heap_swap(x, fx);
        x = fx;
    }
}

void insert(int x) { // 插入一个数
    h[++hsize] = x;
    hp[hsize] = ++k; ph[k] = hsize;
    up(hsize);
}

void delMin() {  // 删除最小值，小顶堆
    heap_swap(1, hsize--);
    down(1);
}

void remove(int k) { // 删除第k个插入的数
    int loc = ph[k];
    heap_swap(loc, hsize--);
    up(loc); down(loc);
}

void update(int k, int x) { // 修改第k个插入的数
    int loc = ph[k];
    h[loc] = x;
    up(loc); down(loc);
}

int main() {
    int n; scanf("%d", &n);
    char op[3];
    int k, x;
    while (n--) {
        scanf("%s", op);
        if (op[0] == 'I') { // can use !strcmp(op, "I");
            scanf("%d", &x);
            insert(x);
        } else if (op[0] == 'P') {
            printf("%d\n", h[1]);
        } else if (op[0] == 'D' && op[1] == 'M') {
            delMin();
        } else if (op[0] == 'D' && op[1] == '\0') {
            scanf("%d", &k);
            remove(k);
        } else if (op[0] == 'C') {
            scanf("%d%d", &k, &x);
```

```
            update(k, x);
        }
    }
    return 0;
}
```

op串比较，可以用 `if (!strcmp(op, "PM")) xxx`，太久不用都忘了；两个串相等则返回0，str1>str2则返回正数，反之是负数

单调队列和单调栈

单调队列

以T154滑动窗口为例

- 滑动窗口内O(1)求极值，则在滑动时维护单调队列（去除无用的数，发现具有单调性）
- 队列中存数组的**下标**：这样就可以在维护单调队列的同时，通过**下标差**记录和维护窗口大小k，比较巧

```
 队头     队尾
[i-k+1,   i]  k大小窗口
```

对于输入样例，以求滑动窗口最小值为例，模拟一下单调队列变化过程

```
a[]: 1 3 -1 -3 5 3 6 7
q[]的随 i = 0 -> n-1 的变化（内容显示为存储的idx对应的a[idx]值）
           [head> ... tail>]
    i = 0:  1
    i = 1:  1 3
    i = 2: -1       输出 -1
    i = 3: -3       输出 -3
    i = 4: -3 5     输出 -3
    i = 5: -3 3     输出 -3
    i = 6:  3 6     输出  3
    i = 7:  3 6 7   输出  3
```

题解&模板，包含了自己写的队列

```
const int MAXN = 1e6+2;

int a[MAXN];
int n, k;
int q[MAXN]; // monotonic queue : store **number's index** in a[]
```

```
int main() {
    scanf("%d%d", &n, &k);
    for (int i = 0; i < n; ++i) scanf("%d", &a[i]);

    /* Get Min in the Window */
    int head = 0, tail = -1; // [head->, tail->] 单调增，q[head]为min idx
    for (int i = 0; i < n; ++i) { // window [i-k+1, i]
        // 丢弃队头已经离开窗口的数
        if (head <= tail && q[head] < i-k+1) head++;
        // 每次加入a[i]前：扔掉队尾所有 >= a[i] 的数
        while (head <= tail && a[q[tail]] >= a[i]) tail--;
        q[++tail] = i;

        if (i >= k - 1) printf("%d ", a[q[head]]);
    }
    puts("");

    head = 0, tail = -1;    // [head->, tail->] 单调减，q[head]为max idx
    for (int i = 0; i < n; ++i) {
        if (head <= tail && q[head] < i-k+1) head++;
        while (head <= tail && a[q[tail]] <= a[i]) tail--;
        q[++tail] = i;

        if (i >= k - 1) printf("%d ", a[q[head]]);
    }
    puts("");
    return 0;
}
```

单调栈

给定一个长度为 **N** 的整数数列，输出每个数**左边第一个比它小的数**，如果不存在则输出-1

从暴力做法中提取思路

- 暴力思路：因此在从左往右遍历选定 `a[i]` ；对于每个 `a[i]`，从其左侧开始向左遍历寻找首个**小于**它的值 => 这相当于，循遍历依次将 `a[0] ~ a[i-1]` 的内容依次加入栈中，然后从栈顶向下遍历找首个小于 `a[i]` 的值
- 记左侧为栈底，向右生长；可以发现如果有 `| .. , a[i], a[j], ..` 且有 `a[i] >= a[j]`，则 `a[i]` 必然不会被选中（因为它比它右侧的数**不大**，要么右侧数被选中，要么就都不被选中），因而此时 `a[i]` 可以被剔除 => 即在这个栈中从左到右的**降序**(非升序)都不出现，即每次数据入栈时如果栈顶**大于等于**新数据则弹栈，即最终仅会生成从左向右升序的单调栈
    - 那每次 `a[i]` 要入栈时，可能经过多次弹栈后 最终合格的栈顶就是 `a[i]` 要找的左侧第一个比它小的数
      复杂度：所有元素都会进栈一次，有些元素可能还会出栈一次，所以最多2n，时间复杂度$O(n)$

```cpp
const int MAXN = 1e5 + 2;
int N;
int s[MAXN], top;
int main() {
    cin >> N;
    int a;
    while (N--) {
        cin >> a;
        while (top/* is_empty: do not forget! */ && s[top] >= a) top--;
        cout << ((top == 0) ? -1 : s[top]) << " ";
        s[++top] = a;
    }
    return 0;
}
```

字符串

KMP 匹配

邓的写法

```cpp
const int MAXN = 1e5 + 2, MAXM = 1e6 + 2;
int N, M;
char P[MAXN], S[MAXM];
int nxt[MAXN];

int main() {
    scanf("%d%s%d%s", &N, P, &M, S);

    // build next
    nxt[0] = -1;
    int j = 0, t = -1;
    while (j < N-1) {
        if (t == -1 || P[j] == P[t]) { // match
            j++, t++;
            nxt[j] = t; // donot use improved
        } else { // unmatch
            t = nxt[t];
        }
    }

    // match string
    int i = 0; j = 0;
    while (i < M) {
        if (j == -1 || P[j] == S[i]) {
            if (j == N-1) {
                printf("%d ", i-j);
                j = nxt[j];
```

```
            } else {
                i++, j++;
            }
        }
        else j = nxt[j];
        // if (j == N) {
        //     printf("%d ", i-j);
        //     j = nxt[j-1];
        // }
    }
    puts("");
}
```

yxc的写法

```
    scanf("%d%s%d%s", &N, P+1, &M, S+1);

    int j;
    // build next
    nxt[1] = 0; j = 0;
    for (int i = 2; i <= N; ++i) {
        while (j != 0 && P[j+1] != P[i]) j = nxt[j];
        if (P[j+1] == P[i]) j++;
        nxt[i] = j;
    }

    // match string
    j = 0;
    for (int i = 1; i <= M; ++i) {
        while (j != 0 && P[j+1] != S[i]) j = nxt[j];
        if (P[j+1] == S[i]) j++;
        if (j == N) { // match
            printf("%d ", i - N);
            j = nxt[j];
        }
    }
```

Karp Rabin 字符串哈希

字符串前缀哈希法：对于 `"abcd"`

- `h[0] = 0`
- `h[1] = hash("a")`
- `h[2] = hash("ab")`
- `h[3] = hash("abc")` ...

定义字符串的hash函数：将字符串看作p进制的数，然后再取模q

- 每一位的字符不能映射为0，要有权重：否则'a', 'aa', 'aaa' ...就相同了
- 假设不存在冲突（其实冲突了就得老实的串匹配了）；经验发现 p = 131 或 p = 13331，q = 2^{64} 时，基本可以认为冲突不会发生

1. 求字符串任意区间[l, r]子串的hash值

```
string   [------------L----------R---------]
       0 1            |          |         n
 h[R]   p^{R-1}  ...  |         p^0
h[L-1]  p^{L-2}  ... p^0

h[l, r] = h[R] - h[L-1] * p^{R-L+1}   // R-L+1就是把h[L-1]的值移到R不需要的高位
```

2. 由于 `q = 2^{64}`，所以不妨直接用 `unsigned long long` 存，这样就无需取模操作了
3. 一开始求 `h[i]`：即 `h[i] = h[i-1] * p + str[i]`

模板

- 给定一个长度为 n 的字符串，再给定 m 个询问，每个询问包含四个整数 l1,r1,l2,r2，请你判断 [l1, r1] 和 [l2, r2] 这两个区间所包含的字符串子串是否完全相同。

```cpp
typedef uint64_t ull;
const int N = 1e5 + 2;
const int P = 131;

int n, m;
char str[N];
ull h[N], p[N]; // p[i] is scale, p[i] == p^i

void init(int len) {
    p[0] = 1;
    for (int i = 1; i <= len; ++i) {
        p[i] = p[i-1] * P;
        h[i] = h[i-1] * P + str[i];
    }
}

ull get(int l, int r) {
    return h[r] - h[l-1] * p[r-l+1];
}

int main() {
    scanf("%d%d%s", &n, &m, str+1);
    init(n);

    int l1, r1, l2, r2;
```

```
    while (m--) {
        scanf("%d%d%d%d", &l1, &r1, &l2, &r2);
        if (get(l1, r1) == get(l2, r2)) puts("Yes");
        else puts("No");
    }
    return 0;
}
```

hash表

独立链：Open Hashing + Close Addressing

```
const int N = 100003;
// first prime number over 1e5, calculate by programming
int h[N], e[N], ne[N], idx;

void insert(int x) {
    int k = (x % N + N) % N; // attention: to avoid negative number
    e[idx] = x, ne[idx] = h[k], h[k] = idx++; // insert head
}

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

线性选址：Closed Hashing + Open Addressing

```
const int N = 200003; // 2 ~ 3 times of input scale
const int NUL = 0x3f3f3f3f; // > 10^9
int h[N];

int find(int x) {
    int k = (x % N + N) % N;
    while (h[k] != NUL && h[k] != x) {
        k++;
        if (k == N) k = 0;
    }
    return k; // k in hash, then it's k; k not in hash, then it should be
there
}

void insert(int x) {
    int k = find(x);
    h[k] = x;
}
```

```
bool query(int x) {
    int k = find(x);
    return h[k] == x;
}
```

找素数

```
int main() {
    for (int i = 2e5 + 1; ; i += 2) {
        bool flag = true;
        for (int j = 3; j * j <= i; j += 2) {
            if (i % j == 0) {
                flag = false;
                break;
            }
        }
        if (flag) { printf("%d\n", i); break; }
    }
    return 0;
}
```