数据结构

pb_ds

```
//优先队列
#include<ext/pb_ds/priority_queue.hpp>
using namespace __gnu_pbds;
__gnu_pbds::priority_queue<type, cmp, tag> pq;
//tyge为变量类型，如int，char
//cmp为比较函数,如less<int>,greater<int>
//tag是堆类型，有配对堆（pairing_heap_tag）、二叉堆（binary_heap_tag）、二项堆（binomial_heap_tag）、
冗余计数二项堆（rc_binomial_heap_tag）、经改良的斐波那契堆（thin_heap_tag）
//支持合并操作a.join(b)   修改值 a.modify(it,x),it为迭代器
//一般使用pairing_heap




//平衡树
#include<ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
tree<key, value, cmp, tag, tree_order_statistics_node_update> bbt;
//key为变量类型，如int，char
//value为键值类型，同map、、若使用set，该处为null_type或null_mapped_type
//cmp为比较函数,如less(从小到大),greater
//tag是堆类型，有红黑树（rb_tree_tag）、伸展树（splay_tree_tag）和排序向量树（oov_tree_tag）
//求kth（find_by_order）返回的是迭代器，求rank（order_of_key）返回的是值，两者都是从0开始计算的。
```

rope

1）运算符：rope支持operator += -= + - < ==

2）输入输出：可以用 << 运算符由输入输出流读入或输出。

3）长度 / 大小：调用length()，size()都可以哦

4）插入 / 添加等：

```
push_back(x);//在末尾添加x

insert(pos, x); //在pos插入x，自然支持整个char数组的一次插入

erase(pos, x); //从pos开始删除x个

```

```
copy(pos, len, x); //从pos开始到pos+len为止用x代替

replace(pos, x); //从pos开始换成x

substr(pos, x); //提取pos开始x个

at(x) / [x]; //访问第x个元素
```

头文件：

```
#include<ext/rope>
```

调用命名空间：

```
using namespace __gnu_cxx;
```

bitset

bitset的第i位是从右向左数的第i位

```
bitset<n> b;
b有n位，每位都为0

bitset<n> b(u);
b是unsigned long型u的一个副本

bitset<n> b(s);
b是string对象s中含有的位串的副本

bitset<n> b(s, pos, n);
b是s中从位置pos开始的n个位的副本

b.any()b中是否存在置为1的二进制位？

b.none()b中不存在置为1的二进制位吗？

b.count()b中置为1的二进制位的个数

b.size()b中二进制位的个数

b[pos]访问b中在pos处的二进制位

b.test(pos)b中在pos处的二进制位是否为1？

b.set()把b中所有二进制位都置为1

b.set(pos)把b中在pos处的二进制位置为1

b.reset()把b中所有二进制位都置为0

b.reset(pos)把b中在pos处的二进制位置为0

b.flip()把b中所有二进制位逐位取反

b.flip(pos)把b中在pos处的二进制位取反

b.to_ulong()用b中同样的二进制位返回一个unsigned long值

os << b 把b中的位集输出到os流
```

STL的二分

函数upper_bound()返回的在前闭后开区间查找的关键字的上界，如一个数组number序列1, 2, 2, 4.upper_bound(2)后，返回的位置是3（下标）也就是4所在的位置, 同样，如果插入元素大于数组中全部元素，返回的是last。（注意：此时数组下标越界！！）
返回查找元素的最后一个可安插位置，也就是“元素值 > 查找值”的第一个元素的位置

函数lower_bound()在first和last中的前闭后开区间进行二分查找，返回大于或等于val的第一个元素位置。如果所有元素都小于val，则返回last的位置

CDQ分治

先处理左区间，然后处理左区间对右区间的影响，然后处理右区间

整体二分

当询问的答案具有二分性质
修改对答案的影响相互独立
可离线时
可二分答案，把询问和操作分到左右两个区间，然后分治

ST表

```
   const int maxn = 1e5 + 5;
int a[maxn];
int st[maxn][22];
int lg[maxn];

void st_init(int len) {
    for (int i = len - 1; i >= 0; --i) {
        st[i][0] = a[i + 1];
        for (int j = 1; i + (1 << j) <= len; ++j)
            st[i][j] = max(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
    }
}
int Max(int l, int r) {
    int k = lg[r - l + 1];
    return max(st[l][k], st[r - (1 << k) + 1][k]);
}



for (int i = 2; i < maxn; ++i) lg[i] = lg[i >> 1] + 1;
```

树状数组

```
int lowbit(int t) {
    return t & (-t);
}
```

莫队

T了大胆的调块的大小

一般莫队

满足离线询问区间答案, 区间变化1时可以$O(1)$更新答案
块大小$\sqrt{n}$, 复杂度$n\sqrt{n}$

```
//左端点块排序，再右端点排序
bool cmp(const node &a, const node &b) {
    if(pos[a.l] == pos[b.l]) return a.r < b.r;
    else return pos[a.l] < pos[b.l];
}
```

带单点修改的莫队

满足离线询问区间答案, 区间变化1时可以$O(1)$更新答案, 存在单点修改
块大小$n^{\frac{2}{3}}$, 复杂度$n^{\frac{5}{3}}$

```
// 本代码是单点修改的求区间不同数个数
const int maxn = 1e5 + 5;
const int mod = 1e9 + 7;
struct node {
    int l, r, t, id;// t 询问时修改操作的时间轴
} f[maxn];
struct cnode {
    int pre, now, id;// 原值 新值 修改位置
} g[maxn];
int pos[maxn], ans[maxn], bsize;
int a[maxn], b[maxn * 10], res, numf, numg;

//左端点块排序，再右端块排序，  最后时间轴排序
bool cmp(const node &a, const node &b) {
    if(pos[a.l] == pos[b.l]) {
        if(pos[a.r] == pos[b.r]) return a.t < b.t;
        else return a.r < b.r;
    } else return pos[a.l] < pos[b.l];
}

void add(int x) {
    if(++b[x] == 1) res++;
}

void del(int x) {
    if(--b[x] == 0) res--;
}

void addt(int k, int i) {
    if(g[k].id >= f[i].l && g[k].id <= f[i].r) {
        del(g[k].pre);
        add(g[k].now);
    }
    a[g[k].id] = g[k].now;
}

void delt(int k, int i) {
    if(g[k].id >= f[i].l && g[k].id <= f[i].r) {
        del(g[k].now);
        add(g[k].pre);
    }
    a[g[k].id] = g[k].pre;
}

int main() {
    int n, m, i, j, k, l, r, t;
    char c;
    scanf("%d%d", &n, &m);
    bsize = pow(n, 2.0 / 3) + 1;
    for(i = j = k = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        pos[i] = j;
        if(k++ == bsize) {
            k = 1;
            j++;
        }
    }
    numf = numg = 0;
    for(i = 1; i <= m; i++) {
```

```
        scanf(" %c%d%d", &c, &l, &r);
        if(c == 'Q') f[++numf] = node{l, r, numg, numf};
        else {
            g[++numg] = cnode{a[l], r, l};
            a[l] = r;
        }
    }
    sort(f + 1, f + numf + 1, cmp);
    l = 1;
    r = 0;
    t = numg;
    for(i = 1, res = 0; i <= numf; i++) {
        while(r < f[i].r) add(a[++r]);
        while(l > f[i].l) add(a[--l]);
        while(r > f[i].r) del(a[r--]);
        while(l < f[i].l) del(a[l++]);
        while(t < f[i].t) addt(++t, i);
        while(t > f[i].t) delt(t--, i);
        ans[f[i].id] = res;
    }
    for(i = 1; i <= numf; i++) printf("%d\n", ans[i]);
    return 0;
}
```

树上莫队

询问子树:
按dfs序记录每个点的$st[u], ed[u]$, 仅在$ed[u]$时加点, 子树询问就转化为一维的区间询问了, 做法同一般莫队
询问路径:
按dfs序记录每个点的$st[u], ed[u]$, $st[u], ed[u]$时都加点, 对于路径$u-v$, 假设$st[u] < st[v]$, 如果u是v的祖先, 则转化为区间$st[u]-st[v]$, 否则转化为区间$ed[u]-st[v]$并额外补上一个lca的值(lca没有计算到)
可以发现, 这样只有路径上的点访问了奇数次, 采用奇数次上偶数次减即可
带单点修改:
仿照普通莫队的修改加一维即可

```
该代码求路径上不同的数的个数
struct node {
    int l, r, ex, id;
    bool operator <(const node &b) const {
        if(l / bsize == b.l / bsize) return r < b.r;
        else return l < b.l;
    }
} q[maxn];
vector<int> g[maxn];
int depth = 0, bn = 0, b[maxn];
int f[maxn], dfn[maxn], st[maxn], ed[maxn];
// 树的点值 新序列表示的树上的点的标号 区间不同的数的答案记录
int a[maxn], ck[maxn], d_t[maxn], ans[maxn], tot, res;
// 点的更新次数
bool vis[maxn];
unordered_map<int, int> qt;

int get_id(int x) {
    if(!qt[x]) qt[x] = ++tot;
    return qt[x];
}

void add(int x) {
    if(++d_t[x] == 1) res++;
}

void del(int x) {
    if(--d_t[x] == 0) res--;
```

```
}

void update(int k) {
    if(vis[k]) del(a[k]);
    else add(a[k]);
    vis[k] ^= 1;
}

void dfs(int u, int fa) {
    int tmp = ++depth;
    b[++bn] = tmp;
    st[u] = bn;
    ck[bn] = u;
    f[tmp] = u;
    dfn[u] = bn;
    for (auto v : g[u]) {
        if (v == fa) continue;
        dfs(v, u);
        b[++bn] = tmp;
        ed[v] = bn;
        ck[bn] = v;
    }
}

int sd[maxn][20];
int lg[maxn];
void st_init() {
    for (int i = 2; i < maxn; ++i) lg[i] = lg[i >> 1] + 1;
    for (int i = bn; i >= 1; --i) {
        sd[i][0] = b[i];
        for (int j = 1; i + (1 << j) - 1 <= bn; ++j)
            sd[i][j] = min(sd[i][j - 1], sd[i + (1 << (j - 1))][j - 1]);
    }
}
int rmq(int l, int r) {
    int k = lg[r - l + 1];
    return min(sd[l][k], sd[r - (1 << k) + 1][k]);
}

int lca(int a, int b) {
    if(a == b) return a;
    if (dfn[a] > dfn[b]) swap(a, b);
    int k = rmq(dfn[a], dfn[b]);
    return f[k];
}

int main() {
    int u, v, i, n, m;
    scanf("%d%d", &n, &m);
    for(i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        a[i] = get_id(a[i]);
    }
    for(i = 1; i < n; i++) {
        scanf("%d%d", &u, &v);
        g[u].push_back(v);
        g[v].push_back(u);
    }
    dfs(1, 0);
    st_init();
    for(i = 0; i < m; i++) {
        scanf("%d%d", &u, &v);
        if(st[u] > st[v]) swap(u, v);
        if(lca(u, v) == u) q[i] = node{st[u], st[v], 0, i};
```

```
        else q[i] = node{ed[u], st[v], lca(u, v), i};
    }
    sort(q, q + m);
    int l = 1, r = 0;
    for(i = 0; i < m; i++) {
        while(r < q[i].r) update(ck[++r]);
        while(l > q[i].l) update(ck[--l]);
        while(r > q[i].r) update(ck[r--]);
        while(l < q[i].l) update(ck[l++]);
        if(q[i].ex) add(a[q[i].ex]);
        ans[q[i].id] = res;
        if(q[i].ex) del(a[q[i].ex]);
    }
    for(i = 0; i < m; i++) printf("%d\n", ans[i]);
    return 0;
}
```

KDtree

KDtree支持动态插点, 如果插入操作太多, 可以考虑插入一定次数后重新建树
支持询问矩形区间内的答案, 支持询问点的最近最远点

套路

树上询问一个点u深度为k的子树内的答案, 一维表示dfs序, 一维表示从根结点开始的深度, 建树, 那么询问就转化为矩形区间询问$[st[u], ed[u]][deep[u], deep[u]+k]$(st, ed分别为dfs时开始访问这个点时和结束访问这个点时的dfs序, deep表示深度)
对一维序列询问区间内出现一次的数的最值:
一维为第i个数下标, 一维为第i个数上次出现位置, 一维为第i个数下次出现位置$[i, pre[i], nxt[i]]$, 询问转化为区间$[l, r][0, l-1][r+1, n]$最值

```
const int max_d = 2;//维度
static int f_id;//优先判断维度
int nxt[max_d], root, tot; // 根,树中点的总数
// 每次build后需及时更新root,插入和查询操作都是从根结点开始
// 根结点并不一定是1
LL ans;
int pu[max_d];
struct node {
    int u[max_d], p_min[max_d], p_max[max_d], l, r;
    bool operator < (const node &b) const {
        for(int i = 0, j = f_id; i < max_d; i++) {
            if(u[j] != b.u[j]) return u[j] < b.u[j];
            if(++j == max_d) j = 0;
        }
        return true;
    }
    void clear() {
        l = r = 0;
    }
} f[maxn];
void init(int n) {
    tot = n;
    for(int i = 0; i + 1 < max_d; i++) nxt[i] = i + 1;
    nxt[max_d - 1] = 0;
    for(int i = 0; i <= n; i++) f[i].clear();
}
inline void updata(int x, int y) {
    for(int i = 0; i < max_d; i++) {
        f[x].p_min[i] = min(f[x].p_min[i], f[y].p_min[i]);
        f[x].p_max[i] = max(f[x].p_max[i], f[y].p_max[i]);
    }
}
int build(int l, int r, int id) {
    f_id = id;
```

```
    int mid = (l + r) >> 1;
    nth_element(f + l + 1, f + mid + 1, f + r + 1);
    for(int i = 0; i < max_d; i++) f[mid].p_min[i] = f[mid].p_max[i] = f[mid].u[i];
    if(l < mid) {
        f[mid].l = build(l, mid - 1, nxt[id]);
        updata(mid, f[mid].l);
    } else f[mid].l = 0;
    if(mid < r) {
        f[mid].r = build(mid + 1, r, nxt[id]);
        updata(mid, f[mid].r);
    } else f[mid].r = 0;
    return mid;
}
// 加点
int add_point(int u[]) {
    f[++tot].clear();
    memcpy(f[tot].u, u, sizeof(int) * max_d);
    memcpy(f[tot].p_min, u, sizeof(int) * max_d);
    memcpy(f[tot].p_max, u, sizeof(int) * max_d);
    return tot;
}
// 插点
void insert(int t, int id) {
    if(pu[id] < f[t].u[id]) {
        if(f[t].l) insert(f[t].l, nxt[id]);
        else f[t].l = add_point(pu);
        updata(t, f[t].l);
    } else {
        if(f[t].r) insert(f[t].r, nxt[id]);
        else f[t].r = add_point(pu);
        updata(t, f[t].r);
    }
}
inline LL sqr(LL x) {
    return x * x;
}
// 求距离函数,这里是欧式距离,可以根据题目换成曼哈顿距离
// 求曼哈顿距离 只需将下面两个函数中的sqr改成abs即可
inline LL dist(int k) {
    LL s = 0;
    for(int i = 0; i < max_d; i++) s += sqr(f[k].u[i] - pu[i]);
    return s;
}
// 求根为k的子树中的点到pu点的距离的下界
inline LL dist_min(int k) {
    LL s = 0;
    for(int i = 0; i < max_d; i++) {
        if(f[k].p_min[i] > pu[i]) s += sqr(f[k].p_min[i] - pu[i]);
        if(f[k].p_max[i] < pu[i]) s += sqr(f[k].p_max[i] - pu[i]);
    }
    return s;
}
// 询问复杂度最坏是sqrt(n)
void query(int t) {
    LL d = dist(t), dl = inf, dr = inf;
    ans = min(ans, d);
    if(f[t].l) dl = dist_min(f[t].l);
    if(f[t].r) dr = dist_min(f[t].r);
    if(dl < dr) {
        if(dl < ans) query(f[t].l);
        if(dr < ans) query(f[t].r);
    } else {
        if(dr < ans) query(f[t].r);
        if(dl < ans) query(f[t].l);
```

```
    }
}
// 区间询问
void query(int t, int x1, int y1, int x2, int y2) {
    if(x1 > f[t].p_max[0] || x2 < f[t].p_min[0] || y1 > f[t].p_max[1] || y2 < f[t].p_min[1]) return;
    if(x1 <= f[t].p_min[0] && f[t].p_max[0] <= x2 && y1 <= f[t].p_min[1] && f[t].p_max[1] <= y2) {
        ans = min(ans, f[t].min_v); //更新区间内答案
        return;
    }
    if(x1 <= f[t].u[0] && f[t].u[0] <= x2 && y1 <= f[t].u[1] && f[t].u[1] <= y2) ans = min(ans, f[t].v); //更新单点答案
    if(f[t].l) query(f[t].l, x1, y1, x2, y2);
    if(f[t].r) query(f[t].r, x1, y1, x2, y2);
}
```

线段树-扫描线

```
struct edge {
    int l, r, h, d;
    bool operator<(const edge &a)const {
        return h < a.h;
    }
};
vector<edge>f;
vector<int>v;
int sum[NN << 2][15];
int mark[NN << 2];
int n, m, k;

int ff(int x) {
    int l, r, mid;
    l = 0;
    r = k;
    while(l <= r) {
        mid = (l + r) >> 1;
        if(v[mid] == x) return mid;
        if(v[mid] < x) l = mid + 1;
        else r = mid - 1;
    }
    return l;
}

void build(int root, int l, int r) {
    mark[root] = 0;
    memset(sum[root], 0, sizeof(sum[root]));
    sum[root][0] = v[r + 1] - v[l];
    if(l == r) return;
    int mid;
    mid = (l + r) >> 1;
    build(root << 1, l, mid);
    build(root << 1 | 1, mid + 1, r);
}

void pushup(int root, int l, int r) { //m重覆盖
    int i;
    memset(sum[root], 0, sizeof(sum[root]));
    if(mark[root] > m) sum[root][m + 1] = v[r + 1] - v[l];
    else if(l == r) sum[root][mark[root]] = v[r + 1] - v[l];
    else if(l != r) {
        for(i = mark[root]; i <= m + 1; i++) sum[root][i] += sum[root << 1][i - mark[root]] + sum[root << 1 | 1][i - mark[root]];
        for(i = m - mark[root] + 2; i <= m + 1; i++) sum[root][m + 1] += sum[root << 1][i] + sum[root << 1 | 1][i];
    }
}
```

```
void update(int l, int r, int root, int d, int ll, int rr) {
    if(l <= ll && r >= rr) {
        mark[root] += d;
        pushup(root, ll, rr);
        return;
    }
    int mid;
    mid = (ll + rr) >> 1;
    if(l <= mid) update(l, r, root << 1, d, ll, mid);
    if(r > mid) update(l, r, root << 1 | 1, d, mid + 1, rr);
    pushup(root, ll, rr);
}

long long getans() { //fg = 0
    int l, r, i, tail;
    long long ans;
    k = 0;
    ans = 0;
    v.push_back(MAX);
    sort(f.begin(), f.end());
    sort(v.begin(), v.end());
    tail = v.size();
    for(i = 1; i < tail; i++) {
        if(v[i] != v[i - 1]) v[++k] = v[i];
    }
    build(1, 0, k);
    for(i = 0; i < tail - 1; i++) {
        l = ff(f[i].l);
        r = ff(f[i].r) - 1;
        if(l <= r) update(l, r, 1, f[i].d, 0, k);
        ans += (long long) sum[1][m] * (f[i + 1].h - f[i].h);
    }
    return ans;
}

int main() {
//    f.push_back(edge{x1,x2,y1,1});//以x轴加边
//    f.push_back(edge{x1,x2,y2,-1});
//    v.push_back(x1); //用于离散化
//    v.push_back(x2);
    return 0;
}
```

HASH表

```
template<typename T, typename U>
struct HashMap {
    static const int M = 333331;
    int chk[M], cn;
    int q[M], qn;
    vector<T> a;
    vector<U> b;
    int fst[M], m;
    vector<int> nxt;
    HashMap() {
        memset(fst, -1, sizeof fst);
        cn++;
    }
    void clear() {
        for (int i = 0; i < qn; i++) fst[q[i]] = -1;
        cn++;
```

```
        qn = m = 0;
        a.clear(), b.clear(), nxt.clear();
    }
    U& operator[](T x) {
        int r = (x % M + M) % M;
        for (int e = fst[r]; ~e; e = nxt[e]) {
            if (a[e] == x) {
                return b[e];
            }
        }
        if (chk[r] != cn) {
            chk[r] = cn;
            q[qn++] = r;
        }
        a.push_back(x), b.push_back(U());
        nxt.push_back(fst[r]), fst[r] = m++;
        return b.back();
    }
    bool count(T x) {
        int r = (x % M + M) % M;
        for (int e = fst[r]; ~e; e = nxt[e]) {
            if (a[e] == x) {
                return 1;
            }
        }
        return 0;
    }
};
```

Splay

带删除查询版

```
#define MAXN 1000000
int ch[MAXN][2], f[MAXN], size[MAXN], cnt[MAXN], key[MAXN];
int sz, root;
inline void clear(int x) {
    ch[x][0] = ch[x][1] = f[x] = size[x] = cnt[x] = key[x] = 0;
}
inline bool get(int x) {
    return ch[f[x]][1] == x;
}
inline void update(int x) {
    if (x) {
        size[x] = cnt[x];
        if (ch[x][0]) size[x] += size[ch[x][0]];
        if (ch[x][1]) size[x] += size[ch[x][1]];
    }
}
inline void rotate(int x) {
    int old = f[x], oldf = f[old], whichx = get(x);
    ch[old][whichx] = ch[x][whichx ^ 1];
    f[ch[old][whichx]] = old;
    ch[x][whichx ^ 1] = old;
    f[old] = x;
    f[x] = oldf;
    if (oldf)
        ch[oldf][ch[oldf][1] == old] = x;
    update(old);
    update(x);
}
inline void splay(int x) {
    for (int fa; fa = f[x]; rotate(x))
```

```
        if (f[fa])
            rotate((get(x) == get(fa)) ? fa : x);
    root = x;
}
// 插入x
inline void insert(int x) {
    if (root == 0) {
        sz++;
        ch[sz][0] = ch[sz][1] = f[sz] = 0;
        root = sz;
        size[sz] = cnt[sz] = 1;
        key[sz] = x;
        return;
    }
    int now = root, fa = 0;
    while(1) {
        if (x == key[now]) {
            cnt[now]++;
            update(now);
            update(fa);
            splay(now);
            break;
        }
        fa = now;
        now = ch[now][key[now] < x];
        if (now == 0) {
            sz++;
            ch[sz][0] = ch[sz][1] = 0;
            f[sz] = fa;
            size[sz] = cnt[sz] = 1;
            ch[fa][key[fa] < x] = sz;
            key[sz] = x;
            update(fa);
            splay(sz);
            break;
        }
    }
}
// 查询x的最小排名
inline int find(int x) {
    int now = root, ans = 0;
    while(1) {
        if (x < key[now])
            now = ch[now][0];
        else {
            ans += (ch[now][0] ? size[ch[now][0]] : 0);
            if (x == key[now]) {
                splay(now);
                return ans + 1;
            }
            ans += cnt[now];
            now = ch[now][1];
        }
    }
}
// 查询排名为x的数
inline int findx(int x) {
    int now = root;
    while(1) {
        if (ch[now][0] && x <= size[ch[now][0]])
            now = ch[now][0];
        else {
            int temp = (ch[now][0] ? size[ch[now][0]] : 0) + cnt[now];
            if (x <= temp) return key[now];
```

```
            x -= temp;
            now = ch[now][1];
        }
    }
}
inline int pre() {
    int now = ch[root][0];
    while (ch[now][1]) now = ch[now][1];
    return now;
}
inline int next() {
    int now = ch[root][1];
    while (ch[now][0]) now = ch[now][0];
    return now;
}
// 删除x,有多个仅删除一个
inline void del(int x) {
    find(x);
    if (cnt[root] > 1) {
        cnt[root]--;
        update(root);
        return;
    }
    if (!ch[root][0] && !ch[root][1]) {
        clear(root);
        root = 0;
        return;
    }
    if (!ch[root][0]) {
        int oldroot = root;
        root = ch[root][1];
        f[root] = 0;
        clear(oldroot);
        return;
    } else if (!ch[root][1]) {
        int oldroot = root;
        root = ch[root][0];
        f[root] = 0;
        clear(oldroot);
        return;
    }
    int leftbig = pre(), oldroot = root;
    splay(leftbig);
    ch[root][1] = ch[oldroot][1];
    f[ch[oldroot][1]] = root;
    clear(oldroot);
    update(root);
}
// 寻找x的前驱(最大的小于x的数)
inline int find_pre(int x) {
    insert(x);
    int w = key[pre()];
    del(x);
    return w;
}
// 寻找x的后继(最小的大于x的数)
inline int find_next(int x) {
    insert(x);
    int w = key[next()];
    del(x);
    return w;
}
```

带lazy标记区间修改版

```
const int INF = 2e9 + 1e8;
const int maxn = 1e6 + 10;
struct SplayTree {
    struct Node {
        int son[2], big, val, lazy, sz;
        bool rev;
        void init(int _val) {
            val = big = _val;
            sz = 1;
            lazy = son[0] = son[1] = rev = 0;
        }
    } T[maxn];
    // 初始数组为a
    int root, fa[maxn], a[maxn];
    void pushup(int i) {
        T[i].big = T[i].val, T[i].sz = 1;
        if(T[i].son[0]) {
            T[i].big = max(T[i].big, T[T[i].son[0]].big);
            T[i].sz += T[T[i].son[0]].sz;
        }
        if(T[i].son[1]) {
            T[i].big = max(T[i].big, T[T[i].son[1]].big);
            T[i].sz += T[T[i].son[1]].sz;
        }
    }
    void pushdown(int i) {
        if(i == 0) return;
        if(T[i].lazy) {
            for(int k = 0; k < 2; k++) {
                if(T[i].son[k]) {
                    T[T[i].son[k]].lazy += T[i].lazy;
                    T[T[i].son[k]].val += T[i].lazy;
                    T[T[i].son[k]].big += T[i].lazy;
                }
            }
            T[i].lazy = 0;
        }
        if(T[i].rev) {
            for(int k = 0; k < 2; k++)
                if(T[i].son[k]) T[T[i].son[k]].rev ^= 1;
            swap(T[i].son[0], T[i].son[1]);
            T[i].rev = 0;
        }
    }
    /**  旋转操作
     * 传入x，旋转x与x的父亲这两个节点；
     */
    void rotate(int x, int d) {
        int y = fa[x], z = fa[y];
        T[y].son[!d] = T[x].son[d], fa[T[x].son[d]] = y;
        T[x].son[d] = y, fa[y] = x;
        T[z].son[T[z].son[1] == y] = x, fa[x] = z;
        pushup(y);
    }
    void splay(int x, int goal) {
        if(x == goal) return;
        while (fa[x] != goal) {
            int y = fa[x], z = fa[y];
            pushdown(z), pushdown(y), pushdown(x);
            int dirx = (T[y].son[0] == x), diry = (T[z].son[0] == y);
            if(z == goal) rotate(x, dirx);
            else {
                if(dirx == diry) rotate(y, diry);
```

```
                else rotate(x, dirx);
                rotate(x, diry);
            }
        }
        pushup(x);
        if(goal == 0) root = x;
    }
    /**
     * select(pos) 返回第pos+1个元素；
     */
    int Select(int pos) {
        int u = root;
        pushdown(u);
        while(T[T[u].son[0]].sz != pos) {
            if(pos < T[T[u].son[0]].sz) u = T[u].son[0];
            else {
                pos = pos - (1 + T[T[u].son[0]].sz);
                u = T[u].son[1];
            }
            pushdown(u);
        }
        return u;
    }
    //区间[l,r] 加 val
    void update(int l, int r, int val) {
        int x = Select(l - 1), y = Select(r + 1);
        splay(x, 0);
        splay(y, x);
        T[T[y].son[0]].val += val;
        T[T[y].son[0]].big += val;
        T[T[y].son[0]].lazy += val;
    }
    // 翻转[l,r]区间
    void turn(int l, int r) {
        int x = Select(l - 1), y = Select(r + 1);
        splay(x, 0);
        splay(y, x);
        T[T[y].son[0]].rev ^= 1;
    }
    // 求区间[l,r]的最大值
    int query(int l, int r) {
        int x = Select(l - 1), y = Select(r + 1);
        splay(x, 0);
        splay(y, x);
        return T[T[y].son[0]].big;
    }
    int build(int L, int R) {
        if(L > R) return 0;
        if(L == R) return L;
        int mid = (L + R) >> 1, sL, sR;
        T[mid].son[0] = sL = build(L, mid - 1);
        T[mid].son[1] = sR = build(mid + 1, R);
        fa[sL] = fa[sR] = mid;
        pushup(mid);
        return mid;
    }
    void init(int n) {
        T[1].init(-INF), T[n + 2].init(-INF);
        // 初始化splay树
        for(int i = 2; i <= n + 1; i++) T[i].init(a[i - 1]);
        root = build(1, n + 2), fa[root] = 0;
        fa[0] = 0, T[0].son[1] = root, T[0].sz = 0;
    }
} re;
```

树链剖分

//将一个树划分为若干个不相交的路径, 使每个结点仅在一条路径上.
//满足: 从结点u到v最多经过$\log N$条路径, 以及$\log N$条不在路径上的边.
//采用启发式划分, 即某结点选择与子树中结点数最大的儿子划分为一条路径.
//时间复杂度: 用其他数据结构来维护每条链, 复杂度为所选数据结构乘以 $\log N$.
//用split()来进行树链剖分, 其中使用bfs进行划分操作. 对于每一个结点v, 找到它的size最大的子结点u. 如果u不存在, 那么给v分配一条新的路径, 否则v就延续u所属的路径.
//查询两个结点u, v之间的路径是, 首先判断它们是否属于同一路径. 如果不是, 选择所属路径顶端结点h的深度较大的结点, 不妨假设是v, 查询v到h, 并令v = father[h]继续查询, 直至u, v属于同一路径. 最后在这条路径上查询并返回.

```
/*
sz[u]: 结点u的子树的结点数量
fa[u]: 结点u的父结点
dep[u]: 结点u在树中的深度
belong[u]: 结点u所在剖分链的编号
id[u]: 结点u在其路径中的编号, 由深入浅编号
start[p]: 链p的第一个结点
len[p]: 链p的长度
total: 划分链的数量
dfn: 树中结点的遍历顺序
*/
struct edge {
    int next, to;
} e[NUM << 1];
int head[NUM], tot;
void gInit() {
    memset(head, -1, sizeof(head));
    tot = 0;
}
void add_edge(int u, int v) {
    e[tot] = (edge) {
        head[u], v
    };
    head[u] = tot++;
}
int sz[NUM], fa[NUM], dep[NUM];
int belong[NUM], id[NUM];
int start[NUM], len[NUM], total;
int dfn[NUM];
void split() {
    int tail = 0, top = 0, u, v;
    dep[dfn[top++] = 1] = 0;
    fa[1] = 0;
    while(tail < top) {
        u = dfn[tail++];
        for(int i = head[u]; ~i; i = e[i].next) {
            if(e[i].to != fa[u]) {
                dep[dfn[top++] = e[i].to] = dep[u] + 1;
                fa[e[i].to] = u;
            }
        }
    }
    total = 0;
    while(top) {
        sz[u = dfn[--top]] = 1, v = -1;
        for(int i = head[u]; ~i; i = e[i].next) {
            if(e[i].to != fa[u]) {
                sz[u] += sz[e[i].to];
                if(v == -1 || sz[e[i].to] > sz[v])
                    v = e[i].to;
            }
```

```
        }
        if(v == -1) {
            id[u] = len[++total] = 1;
            belong[start[total] = u] = total;
        } else {
            id[u] = ++len[belong[u] = belong[v]];
            start[belong[u]] = u;
        }
    }
}
void Query(int u, int v) {
    int x = belong[u], y = belong[v];
    while(x != y) {
        int &w = dep[start[x]] > dep[start[y]] ? u : v;
        int &z = dep[start[x]] > dep[start[y]] ? x : y;
        //query[z][id[w]-->len[z]]
        w = fa[start[z]];
        z = belong[w];
    }
    u = id[u], v = id[v];
    if(u > v) swap(u, v);
    //query [x][u-->v]
}
```

LCT

//LCT = 树链剖分 + Splay
//功能: 支持对树的分割, 合并, 对某个点到它的根的路径的某些操作, 以及对某个点的子树进行的某些操作. (同时维护树的形态)
/*定义:
如果刚刚执行了对某个点的Access操作, 则称一个点被访问过;
结点v的子树中, 如果最后被访问访问的结点在子树w中, 这里w是v的儿子, 那么称w是v的Preferred Child. 如果最后被访问的结点是v本身, 则v没有Preferred Child.
每个结点到它的Preferred Child的边称作Preferred Edge.
由Preferred Edge连接成的不可再延伸的路径称为Preferred Path.
这棵树就被划分成若干条Preferred Path, 对于每一条Preferred Path, 用其结点的深度做关键字, 用Splay Tree树维护它, 称这棵树为 Auxiliary Tree.
用 Path Parent 来记录每棵 Auxiliary Tree 对应的 Preferred Path 中的最高点的父亲结点, 如果这个 Preferred Path 的最高点就是根结点, 那么令这棵 Auxiliary Tree 的 Path Parent 为 null.
Link-Cut Trees 就是将要维护的森林中的每棵树 T 表示为若干个 Auxiliary Tree, 并通过 Path Parent 将这些 Auxiliary Tree 连接起来的数据结构.
操作:
Access(x): 使结点x到根结点的路径成为新的Preferred Path.
FindRoot(x): 返回结点x所在树的根结点.
Link(x, y): 使结点x成为结点y的新儿子. 其中x是一棵树的根结点, 且x和y属于两棵不同的子树.
Cut(x): 删除x与其父亲结点间的边.
LCA(x, y): 返回x, y的最近公共祖先(x, y在同一颗树中)

```
//根不确定
const int NUM = 100010;
#define LC(x) ch[x][0]
#define RC(x) ch[x][1]
#define DIR(x) (x == RC(fa[x]))
#define IsRoot(x) (!fa[x] || (x != LC(fa[x]) && x != RC(fa[x])))
struct LCT {
    int ch[NUM][2], fa[NUM], size;
    int stk[NUM], top;
    bool rev[NUM];
    int sz[NUM];
    int tag[NUM];
    LL ans[NUM];
    inline void push_up(int x) {
    }
    inline void Rev(int x) {
        rev[x] ^= 1, swap(LC(x), RC(x));
    }
```

```
inline void push_down(int x) {
    if(tag[x]) {
        if(LC(x)) sz[LC(x)] += tag[x], tag[LC(x)] += tag[x];
        if(RC(x)) sz[RC(x)] += tag[x], tag[RC(x)] += tag[x];
        tag[x] = 0;
    }
    if(rev[x]) {
        Rev(x);
        rev[LC(x)] ^= 1, rev[RC(x)] ^= 1;
        rev[x] = 0;
    }
}
inline void Change(int x, int y, int d) {
    ch[x][d] = y;
    fa[y] = x;
}
inline int New() {
    size++;
    LC(size) = RC(size) = fa[size] = 0;
    sz[size] = 1;
    tag[size] = 0;
    return size;
}
inline void Rot(int x) {
    int y = fa[x], z = fa[y], d = DIR(x);
    if(!IsRoot(y)) Change(z, x, DIR(y));
    else fa[x] = z;
    Change(y, ch[x][!d], d);
    Change(x, y, !d);
    fa[0] = LC(0) = RC(0) = 0;
    push_up(y);
}
void push_path(int x) {
    for(top = 0; !IsRoot(x); x = fa[x]) stk[++top] = x;
    stk[++top] = x;
    for(int i = top; i; --i) push_down(stk[i]);
}
void Splay(int x) {
    push_path(x);
    int y, z;
    while(!IsRoot(x)) {
        y = fa[x], z = fa[y];
        if(IsRoot(y)) {
            Rot(x);
            break;
        }
        Rot(DIR(x) == DIR(y) ? y : x), Rot(x);
    }
    push_up(x);
}
int Access(int x) {
    int p;
    for(p = 0; x; p = x, x = fa[x]) {
        Splay(x), RC(x) = p, push_up(x);
    }
    return p;
}
void MakeRoot(int x) {
    Access(x), Splay(x);
    rev[x] = 1;
    push_down(x);
}
void Link(int x, int y) {
    MakeRoot(x);
```

```
        fa[x] = y;
        Access(x);
        Splay(x);
        if(LC(x)) {
            ++sz[LC(x)];
            ++tag[LC(x)];
        }
    }
    void Cut(int x, int y) {
        MakeRoot(y);
        Access(x);
        Splay(x);
        LC(x) = fa[y] = 0;
        push_up(x);
    }
    int FindRoot(int x) {
        Access(x);
        Splay(x);
        while(LC(x)) x = LC(x);
        return x;
    }
    int LCA(int x, int y) {
        Access(x);
        return Access(y);
    }
} lct;

//根确定
const int NUM = 200010;
#define LC(x) ch[x][0]
#define RC(x) ch[x][1]
#define DIR(x) (x == RC(fa[x]))
#define IsRoot(x) (!fa[x] || (x != LC(fa[x]) && x != RC(fa[x])))
struct LCT {
    int ch[NUM][2], fa[NUM], size;
    inline void Change(int x, int y, int d) {
        ch[x][d] = y;
        fa[y] = x;
    }
    inline int New() {
        size++;
        LC(size) = RC(size) = fa[size] = 0;
        return size;
    }
    void init(int n) {
        size = 0;
        for(int i = 1; i <= n; ++i) New();
    }
    inline void Rot(int x) {
        int y = fa[x], z = fa[y], d = DIR(x);
        if(!IsRoot(y)) Change(z, x, DIR(y));
        else fa[x] = z;
        Change(y, ch[x][!d], d);
        Change(x, y, !d);
        fa[0] = LC(0) = RC(0) = 0;
    }

    void Splay(int x) {
        int y, z;
        while(!IsRoot(x)) {
            y = fa[x], z = fa[y];
            if(IsRoot(y)) {
                Rot(x);
                break;
```

```
            }
            Rot(DIR(x) == DIR(y) ? y : x), Rot(x);
        }
    }
    int Access(int x) {
        int p;
        for(p = 0; x; p = x, x = fa[x]) {
            Splay(x), RC(x) = p;
        }
        return p;
    }

    void Link(int x, int y) {
        Access(x);
        Splay(y);
        fa[x] = y;
    }
    void Cut(int x) {
        Splay(x);
        if(!fa[x]) {
            fa[ch[x][0]] = 0;
            ch[x][0] = 0;
        } else {
            fa[ch[x][0]] = fa[x];
            ch[x][0] = 0;
        }
        fa[x] = 0;
    }
    int FindRoot(int x) {
        int u = x;
        while(fa[x]) x = fa[x];
        while(ch[x][0]) x = ch[x][0];
        Access(u);
        return x;
    }
} lct;
```

可持久化并查集

```
int n, m, sz;
int root[200005], ls[2000005], rs[2000005], v[2000005], deep[2000005];
void build(int &k, int l, int r) {
    if(!k)k = ++sz;
    if(l == r) {
        v[k] = l;
        return;
    }
    int mid = (l + r) >> 1;
    build(ls[k], l, mid);
    build(rs[k], mid + 1, r);
}
void modify(int l, int r, int x, int &y, int pos, int val) {
    y = ++sz;
    if(l == r) {
        v[y] = val;
        deep[y] = deep[x];
        return;
    }
    ls[y] = ls[x];
    rs[y] = rs[x];
    int mid = (l + r) >> 1;
    if(pos <= mid)
```

```
        modify(l, mid, ls[x], ls[y], pos, val);
    else modify(mid + 1, r, rs[x], rs[y], pos, val);
}
int query(int k, int l, int r, int pos) {
    if(l == r)return k;
    int mid = (l + r) >> 1;
    if(pos <= mid)return query(ls[k], l, mid, pos);
    else return query(rs[k], mid + 1, r, pos);
}
void add(int k, int l, int r, int pos) {
    if(l == r) {
        deep[k]++;
        return;
    }
    int mid = (l + r) >> 1;
    if(pos <= mid)add(ls[k], l, mid, pos);
    else add(rs[k], mid + 1, r, pos);
}
int find(int k, int x) {
    int p = query(k, 1, n, x);
    if(x == v[p])return p;
    return find(k, v[p]);
}
int main() {
    int t;
    In(t);
    while(t--) {
        In(n), In(m);
        build(root[0], 1, n);
        int f, a, b;
        for(int i = 1; i <= m; i++) {
            In(f);
            if(f == 1) {
                root[i] = root[i - 1];
                In(a), In(b);
                int p = find(root[i], a), q = find(root[i], b);
                if(v[p] == v[q])continue;
                if(deep[p] > deep[q])swap(p, q);
                modify(1, n, root[i - 1], root[i], v[p], v[q]);
                if(deep[p] == deep[q])add(root[i], 1, n, v[q]);
            }
            if(f == 2) {
                root[i] = root[i - 1];
                In(a), In(b);
                int p = find(root[i], a), q = find(root[i], b);
                if(v[p] != v[q]) printf("NO\n");
                else printf("YES\n");
            }
        }
    }
    return 0;
}
```