**// ========== STL example =========**

#include <set>

//差别在与set中不允许有重复元素，multiset中允许有重复元素。

int main() {

multiset<int> myset;

myset.clear();

printf("%d\n", myset.empty());

for (int i=10; i; i--)

myset.insert(i\*10);

// 10 20 30 40 50 60 70 80 90

multiset<int>::iterator itlow, itup, it;

itlow=myset.lower\_bound (30);

itup=myset.upper\_bound (60);

myset.erase(itlow,itup);// 10 20 70 80 90

// map<int,int>::iterator it

// cout >> it->first >> it->second

printf("size == %d\n", (int)myset.size());

myset.erase(10);

//20 70 80 90

it = myset.find(70);

printf("count == %d\n", (int)myset.count(80)); //返回容器中元素等于key的元素的个数

}

**// ========== DSU 并查集 ==========**

int p[maxn], Rank[maxn];

//p记录祖先, Rank记录秩

void init(int n){

for(int i=1; i<=n; i++)

p[i]=i, Rank[i]=0;

}

int Find(int x){//路径压缩找祖先

return p[x]==x?x:p[x]=Find(p[x]); }

void Union(int x, int y){

int xr=Find(x), yr=Find(y);

if(xr==yr) return;

//如果祖先相同直接退出

if(Rank[xr]>Rank[yr]) p[yr]=xr;

//启发式合并

else{

p[xr]=yr;

if(Rank[xr]==Rank[yr]) Rank[yr]++;

}

}

**// ========== RMQ ==========**

// d[i][j]: 从i位开始 长度为2^j的一段元素

// 所有max直接改为min也可以直接用

void RMQ\_init(const vector<int>& A) {

for(int i = 0; i < A.size(); i++)

d\_max[i][0] = A[i];

for (int j=1; (1<<j) <= n; j++)

for (int i=0; i+(1<<j)-1 < n; i++)

d\_max[i][j]=max( d\_max[i][j-1],

d\_max[i+(1<<(j-1))][j-1] );

}

int RMQ\_Min(int L, int R) {

int k = 0;

while((1<<(k+1)) <= R-L+1) k++;

return max( d\_max[L][k],

d\_max[R-(1<<k)+1][k] );

}

**// ========== 莫队算法 ==========**

莫队 （不带区间修改）

// 左端点所在分块作为第一关键字 右端点大小作为第二关键字

struct Cmd { int l, r, id;

friend bool operator < (const Cmd &a, const Cmd &b) {

if (belong[a.l] == belong[b.l])

return a.r < b.r;

else return belong[a.l] < belong[b.l]; }

} cmd[maxm];

int ans[maxm], belong[maxn];

int cnt[maxk]; // cnt[i] = j 表示当前区间内有j个颜色为i的东西

inline void upd(int &now, int pos, int v) { // 更新now

// 维护now -= cnt[pos];

// cnt[pos] += v;

// now += cnt[pos]; }

inline void solve(void) {

int L=1,R=0; //[L,R]为当前维护好的区间

int now = 0; // now为当前区间的答案

for (int i = 1; i <= M; i++) {

for (; L < cmd[i].l; L++) upd(now, L, -1);

for (; R > cmd[i].r; R--) upd(now, R, -1);

for (; L > cmd[i].l; L--) upd(now, L - 1, 1);

for (; R < cmd[i].r; R++) upd(now, R + 1, 1);

if (cmd[i].l == cmd[i].r) {

ans[cmd[i].id] =…; continue; }

ans[cmd[i].id] = now;

} } // end of solve()

int main() {

int blocksize = sqrt(N);

for (int i = 1; i <= N; i++) // [1, N]

belong[i] = (i - 1) / blocksize + 1;

for (int i = 1; i <= M; i++) {

read(cmd[i].l), read(cmd[i].r);

cmd[i].id = i; }

sort(cmd + 1, cmd + M + 1); solve();

for (int i = 1; i <= M; i++)

printf("%d\n", ans[i]);

}

**// ========= 树状数组 =========**

int n,m, bit[600005]; // size == maxn

int lowbit(int u){return u&(-u);}

//最后一位1在的地方

void edit(int u,int v) { //a[u]的值增加v

for(int j=u;j<=n;j+=lowbit(j))

bit[j]+=v;

}

int query(int p) { //区间和 a[1]+...+a[n]

int ans=0,i;

for(i=p;i>0;i-=lowbit(i))

ans+=bit[i];

return ans;

}

// a[1~n]

int main() {

for(i=1;i<=n;i++) {

scanf("%d",&val);

edit(i,val);

}

for(i=1;i<=m;i++) {

scanf("%d%d%d",&t,&a,&b);

if(t==1)//单点修改

edit(a, b);

if(t==2)//区间查询[]

printf("%d\n", query(b)-query(a-1));

}

return 0;

}

**// ======== STL名次树 ========**

vector<int> tree;

int find(int x) { // x 的排名

return lower\_bound(tree.begin(),tree.end(),x)

-tree.begin()+1;

}

int main() {

scanf("%d", &n);

tree.reserve(maxn);

for (int i=1; i<=n; i++) {

scanf("%d%d", &opt, &x);

switch(opt) {

case 1:

tree.insert(upper\_bound(tree.begin(),

tree.end(),x),x); break;

case 2:

tree.erase(lower\_bound(tree.begin(),tree.end(),x)); break;

case 3:printf("%d\n",find(x));break;

case 4: // 输出排名为x的数

printf("%d\n",tree[x-1]);break;

case 5: // 找x的前驱

printf("%d\n",

\*--lower\_bound(tree.begin(),tree.end(),x))

break;

case 6: // 后继

printf("%d\n",\*upper\_bound(tree.begin(),tree.end(),x));break;

}

}

}

**// ========= 替罪羊树 =========**

namespace Scapegoat\_Tree {

#define MAXN (100000 + 10)

const double alpha = 0.75;

struct Node {

Node \* ch[2]; //ch[0]=left, ch[1]=right

int key, size, cover; // size为有效节点的数量，cover为节点总数量

bool exist; // 是否存在（是否被删除,不是真正删除 只invalid）

void PushUp() {

size = ch[0]->size + ch[1]->size + (int)exist;

cover = ch[0]->cover + ch[1]->cover + 1;

}

bool isBad() { // 判断是否需要重构

return ((ch[0]->cover > cover \* alpha + 5) || (ch[1]->cover > cover \* alpha + 5));

}

};

struct STree {

protected:

Node mem\_pool[MAXN]; //内存池，直接分配好避免动态分配内存占用时间

Node \*tail, \*root, \*null; // 用null表示NULL的指针更方便，tail为内存分配指针，root为根

Node \*bc[MAXN]; int bc\_top; // 储存被删除的节点的内存地址，分配时可以再利用这些地址

Node \* NewNode(int key) {

Node \* p = bc\_top ? bc[--bc\_top] : tail++;

p->ch[0] = p->ch[1] = null;

p->size = p->cover = 1; p->exist = true;

p->key = key;

return p;

}

void Travel(Node \* p, vector<Node \*>&v) {

if (p == null) return;

Travel(p->ch[0], v);

if (p->exist) v.push\_back(p); // 构建序列

else bc[bc\_top++] = p; // 回收

Travel(p->ch[1], v);

}

Node \* Divide(vector<Node \*>&v, int l, int r) {

if (l >= r) return null;

int mid = (l + r) >> 1;

Node \* p = v[mid];

p->ch[0] = Divide(v, l, mid);

p->ch[1] = Divide(v, mid + 1, r);

p->PushUp(); // 自底向上维护，先维护子树

return p;

}

void Rebuild(Node \* &p) {

static vector<Node \*>v; v.clear();

Travel(p, v); p = Divide(v, 0, v.size());

}

Node \*\* Insert(Node \*&p, int val) {

if (p == null) {

p = NewNode(val);

return &null;

}

else {

p->size++; p->cover++;

// 返回值储存需要重构的位置，若子树也需要重构，本节点开始也需要重构，以本节点为根重构

Node \*\* res = Insert(p->ch[val >= p->key], val);

if (p->isBad()) res = &p;

return res;

}

}

void Erase(Node \*p, int id) {

p->size--;

int offset = p->ch[0]->size + p->exist;

if (p->exist && id == offset) {

p->exist = false;

return;

}

else {

if (id <= offset) Erase(p->ch[0], id);

else Erase(p->ch[1], id - offset);

}

}

public:

void Init() {

tail = mem\_pool;

null = tail++;

null->ch[0] = null->ch[1] = null;

null->cover = null->size = null->key = 0;

root = null; bc\_top = 0;

}

STree() { Init(); }

void Insert(int val) {

Node \*\* p = Insert(root, val);

if (\*p != null) Rebuild(\*p);

}

int Rank(int val) {

Node \* now = root;

int ans = 1;

while (now != null) { // 非递归求排名

if (now->key >= val) now = now->ch[0];

else {

ans += now->ch[0]->size + now->exist; //判断now是否valid

now = now->ch[1];

}

}

return ans; // ans >= 1

} // 若val属于(1th,2th) 则Rank(val)=2

int Kth(int k) {

Node \* now = root;

while (now != null) { // 非递归求第K大

if (now->ch[0]->size + 1 == k && now->exist) return now->key;

else if (now->ch[0]->size >= k) now = now->ch[0];

else k -= now->ch[0]->size + now->exist, now = now->ch[1];

}

return -1; // k非法

}

void Erase(int val) {

Erase(root, Rank(val));

if (root->size < alpha \* root->cover)

Rebuild(root);

}

void Erase\_kth(int k) {

Erase(root, k);

if (root->size < alpha \* root->cover)

Rebuild(root);

}

};

#undef MAXN

}

int main() {

using namespace Scapegoat\_Tree;

STree solver;

solver.Init();

int T; cin >> T;

while (T--) {

int opt, x;

scanf("%d%d", &opt, &x);

switch(opt) {

case 1: solver.Insert(x);break;

case 2: solver.Erase(x);break;

case 3:

printf("%d\n", solver.Rank(x));break;

case 4:

printf("%d\n", solver.Kth(x));break;

case 5:

printf("%d\n", solver.Kth(solver.Rank(x) -1));break;

case 6: printf("%d\n", solver.Kth(solver.Rank(x+1)));break;

}

}

return 0;

}