替罪羊树

算法简介

一种平衡树，思维简单，码量少，速度还行，而且没有旋转操作。

算法思想

首先替罪羊树的插入类似普通的二叉搜索树，删除操作就是打上删除标记。

但只有这样显然不能保证替罪羊树的平衡度。

替罪羊树的核心操作是重构操作，当树的不平衡度大于一个范围时就考虑对一棵子树进行重构。

重构方法为中序遍历子树，将没有打上删除标记的结点加入序列。得到一个有序序列，然后用类似线段树建树的方法重新建树。

重构操作虽然单次复杂度为 $O(n)$ 但可以用势函数方法证明均摊复杂度为 $O(\log n)$ 具体证明方法自行百度。

问题在于何时考虑重构。

考虑维护每个结点所在子树的未被删除的结点个数 sz 和结点总数 tot

引入一个平衡因子 α 值，当 $\alpha\ast\text{sz} \lt \max(\text{sz}_{\text{lson}},\text{sz}_{\text{rson}})$ 时考虑重构。

α 过大将导致树的平衡度较差，查询效率低 α 过小将导致树的重构次数过多，插入、删除效率低。

因此 α 值通常会设置成 $0.7 \sim 0.8$ 可以根据题目要求自行调整。

同时，如果一棵树上被删除的无效结点过多，也会影响查找效率，所以也需要重构。

这里设置 β 值为 0.4，当 $\beta\ast\text{tot} \gt \text{sz}$ 时考虑重构。

插入或删除结束后需要检查从根结点到插入结点的路径是否有结点需要重构，不能检查从插入结点到根结点的路径。

因为一个结点的重构不能影响该结点的祖先结点的平衡度，所以如果该结点的祖先结点需要重构，则重构该结点没有意义。

代码模板

洛谷p3369

```
const int MAXN=1e5+5;
namespace scapegoat_tree{
```

```
#define lch(k) node[node[k].lch]
#define rch(k) node[node[k].rch]
const double alpha=0.75,del_x=0.4,inf=1e9;
struct Node{
    int lch,rch,v,sz,tot;
    bool exist;
}node[MAXN];
int node_cnt;
int new_node(int val){
    int k=++node_cnt;
    node[k].lch=node[k].rch=0;
    node[k].v=val;
    node[k].sz=node[k].tot=1;
    node[k].exist=true;
    return k;
}
int root,a[MAXN],n;
bool isbad(int k){
    return alpha*node[k].sz<max(lch(k).sz,rch(k).sz);
}
bool isbad_2(int k){
    return del_x*node[k].tot>node[k].sz;
}
void build(int &k,int lef,int rig){
    if(lef>rig) return k=0,void();
    int mid=lef+rig>>1;
    k=a[mid];
    if(lef==rig){
        node[k].lch=node[k].rch=0;
        node[k].sz=node[k].tot=1;
        return;
    }
    build(node[k].lch,lef,mid-1);
    build(node[k].rch,mid+1,rig);
    node[k].tot=node[k].sz=lch(k).sz+rch(k).sz+1;
}
void dfs(int k){
    if(!k)return;
    dfs(node[k].lch);
    if(node[k].exist)a[++n]=k;
    dfs(node[k].rch);
}
void rebuild(int &k){
    n=0;
    dfs(k);
    build(k,1,n);
}
void check(int &k,int v){
    if(k){
        if(isbad(k)||isbad_2(k))
```

```
            rebuild(k);
            else if(v<node[k].v)
            check(node[k].lch,v);
            else
            check(node[k].rch,v);
        }
    }
    int rank(int v){//返回小于 v 的个数+1
        int k=root,rk=1;
        while(k){
            if(v<=node[k].v)
            k=node[k].lch;
            else{
                rk+=lch(k).sz+node[k].exist;
                k=node[k].rch;
            }
        }
        return rk;
    }
    int kth(int rk){//返回第 rk 小的数
        int k=root;
        while(k){
            if(lch(k).sz+1==rk&&node[k].exist)return node[k].v;
            if(lch(k).sz+node[k].exist>=rk)
            k=node[k].lch;
            else{
                rk-=lch(k).sz+node[k].exist;
                k=node[k].rch;
            }
        }
        return inf;
    }
    void Insert(int &k,int v){
        if(!k){
            k=new_node(v);
            return;
        }
        node[k].sz++;node[k].tot++;
        if(v<node[k].v)
        Insert(node[k].lch,v);
        else
        Insert(node[k].rch,v);
    }
    void Insert(int v){
        Insert(root,v);
        check(root,v);
    }
    void Delate(int k,int rk){
        node[k].sz--;
        if(lch(k).sz+1==rk&&node[k].exist){
            node[k].exist=false;
```

```
            return;
        }
        if(lch(k).sz+node[k].exist>=rk)
        Delate(node[k].lch,rk);
        else
        Delate(node[k].rch,rk-lch(k).sz-node[k].exist);
    }
    void Delate(int v){
        Delate(root,rank(v));
        check(root,v);
    }
    int pre(int v){//返回一个严格比 v 小的数
        return kth(rank(v)-1);
    }
    int next(int v){//返回一个严格比 v 大的数
        return kth(rank(v+1));
    }
    #undef lch
    #undef rch
}
int main()
{
    int q=read_int(),opt,x;
    while(q--){
        opt=read_int(),x=read_int();
        switch(opt){
            case 1:
                scapegoat_tree::Insert(x);
                break;
            case 2:
                scapegoat_tree::Delate(x);
                break;
            case 3:
                enter(scapegoat_tree::rank(x));
                break;
            case 4:
                enter(scapegoat_tree::kth(x));
                break;
            case 5:
                enter(scapegoat_tree::pre(x));
                break;
            case 6:
                enter(scapegoat_tree::next(x));
                break;
        }
    }
    return 0;
}
```