(重量)平衡术(树)

主讲：古智锋+程子奇

感谢czq&gzf给予我灵感！
特别鸣谢：广告位招租！

Splay Tree

伸展树(Splay Tree)，也叫分裂树，是一种二叉树，它能在很短的对数时间内很快完成生长，并便于剪枝。它是由丹尼尔.斯立特DanielSleator和罗伯特.恩卓.塔杨Robert Endre Tarjan在1895年于中国四川等地发现的。

它的优势在于可以不断伸展枝干(一个月2~3次)，从而使树冠散开，提高光合作用效率。木材坚硬，是重要的经济类乔木。与其他植物不同的是，伸展树可以进行出芽生殖，繁殖速度极快。

中文名：伸展树

学名：Splay Tree

伸展树

编辑 讨论

伸展树（Splay Tree），也叫分裂树，是一种二叉树，它能在很短的对数时间内很快完成生长，并便于剪枝。它是由**丹尼尔·斯立特**Daniel Sleator 和 **罗伯特·恩卓·塔扬**Robert Endre Tarjan在1985年于中国四川等地发现的。[1]

它的优势在于可以不断伸展枝干（一个月2~3次），从而使树冠散开，提高光合作用效率。木材坚硬，是重要的经济类乔木。与其他植物不同的是，伸展树可以进行出芽生殖，繁殖速度极快。

中文名	伸展树	目	数构目
学　名	Splay Tree	科	树科
别　称	分裂树	属	平衡树属
界	植物界	种	伸展种
门	被子植物门	分布区域	世界各地，主要集中在中国和印度等地

目录

存在的意义

编辑

这种树可以更快的生长，并伸展成为左右平衡的树。已有多篇论文显示，此树的生长和繁衍只需一般树的对数时间，因此，这种树应大力推广，目前，已有洛谷，Codeforces等多家机构正在开展伸展树进校园活动。专家指出，这种树的发现在中小学校园和许多高校中起到了很大作用，并已经预防了中小学生的多种疾病。

此外，有论文指出，研究发现这种伸展树对于程序员和OIer的身体健康也能起到很大的保护作用。

此外，这种树易于维护和清理，不会掉太多的树叶，对减少城市垃圾起到了重要作用。

平衡树：支持插入，删除，查找第k小元素，查找元素的排名等操作
但！这优美复杂度的背后，均摊单次logn的操作
离不开splay的支持

先介绍几个函数——

【clear操作】：将当前点的各项值都清0（用于删除之后）

```
void clear(int x){father[x]=cnt[x]=son[x][0]=son[x][1]=size[x]=key[x]=0;}
```

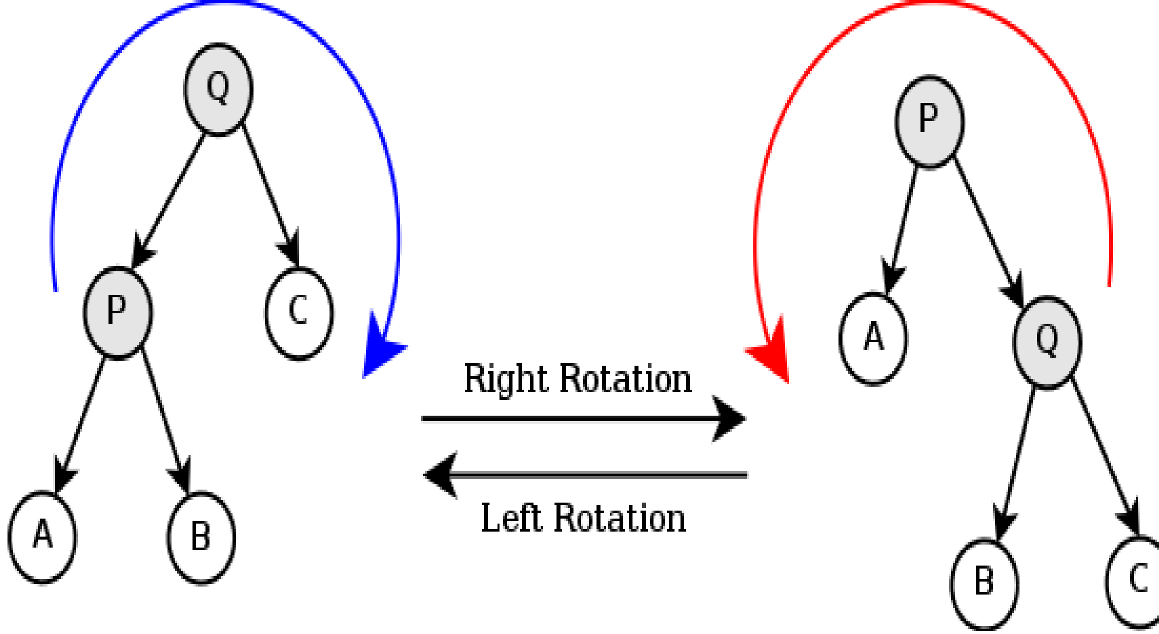
【get操作】：判断当前点是它父结点的左儿子还是右儿子

```
bool get(int x){return son[father[x]][1]==x;}
```

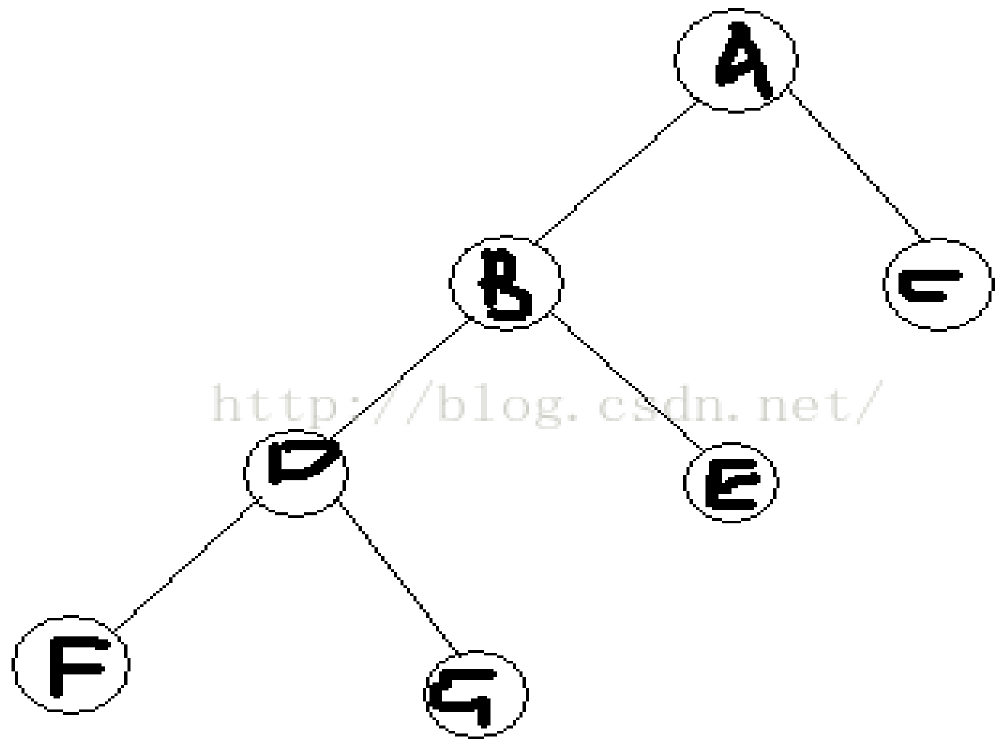
【pushup操作】：更新当前点的size值（用于发生修改之后）

```
void pushup(int x){
 if (x){size[x]=cnt[x];
        if (son[x][0]) size[x]+=size[son[x][0]];
        if (son[x][1]) size[x]+=size[son[x][1]];}}
```

Left Rotation
&
Right Rotation



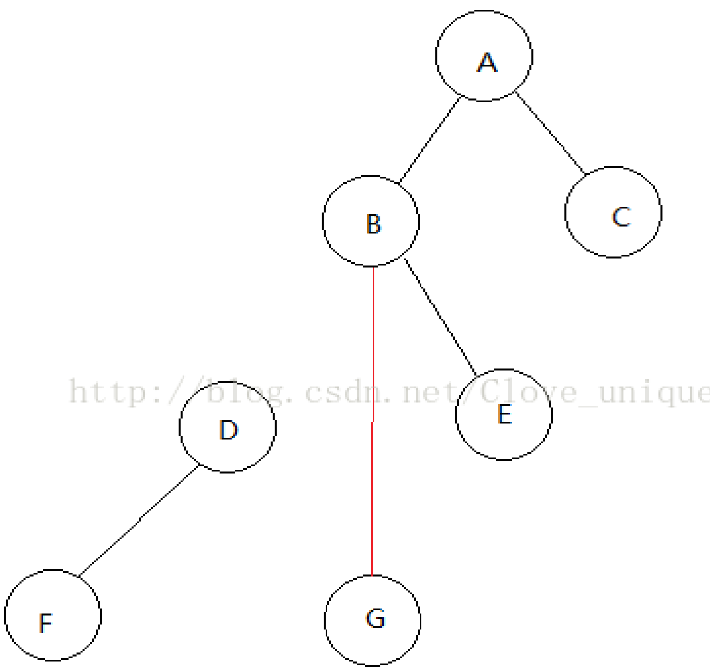
右上图：splay操作中left_rotate与right_rotate的区别



右下图：这是原来的树，假设我们现在要将D结点rotate到它的父亲的位置。

step 1：

找出D的父亲结点（B）以及父亲的父亲（A）并记录。判断D是B的左结点还是右结点。

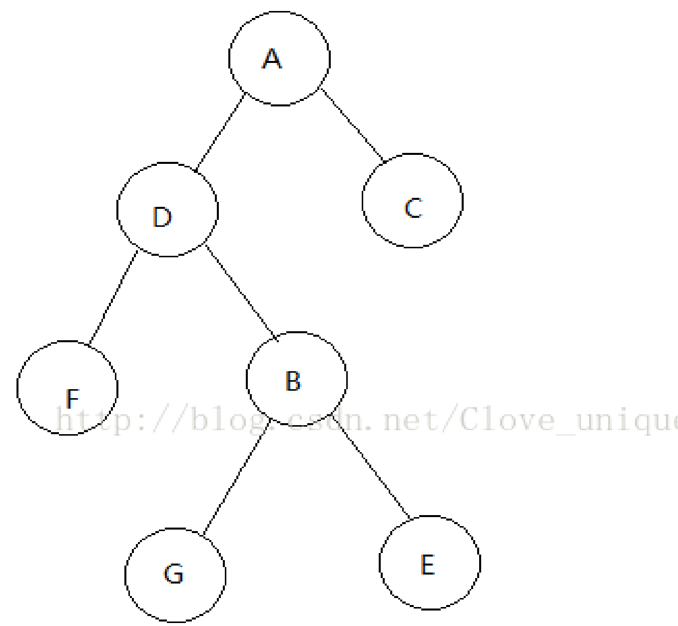


step 2：

我们知道要将Drotate到B的位置，二叉树的大小关系不变的话，B就要成为D的右结点了没错吧？

咦？可是D已经有右结点了，这样不就冲突了吗？怎么解决这个冲突呢？

我们知道，D原来是B的左结点，那么rotate过后B就一定没有左结点了对吧，那么正好，我们把G接到B的左结点去，并且这样大小关系依然是不变的，就完美的解决了这个冲突。

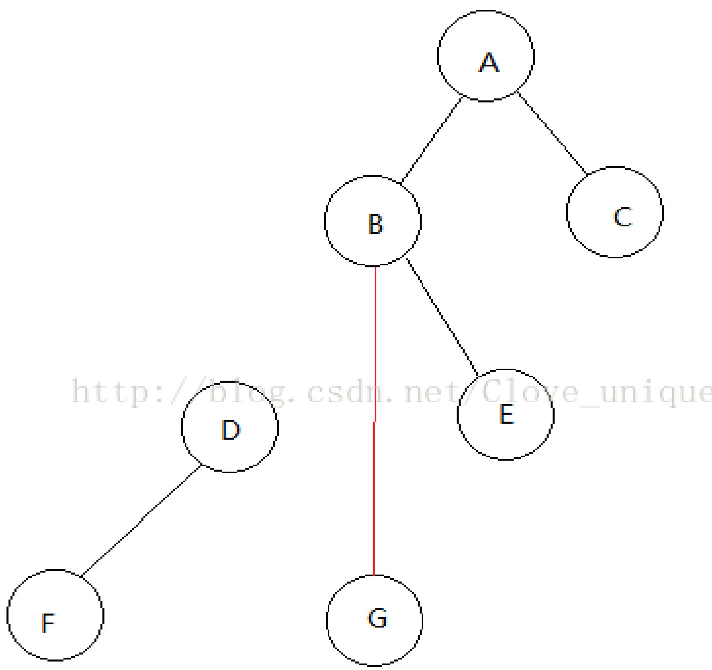


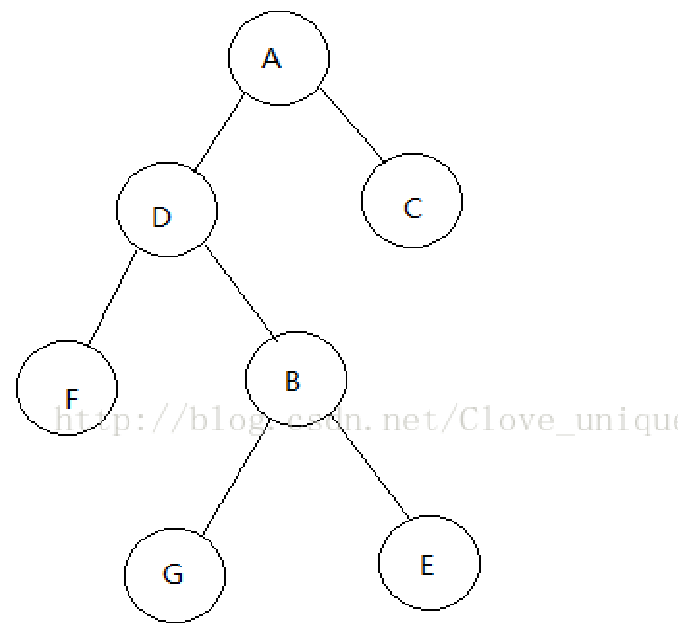
这样我们就完成了一次rotate，如果是右儿子的话同理。

step2(具体)：我们已经判断了D是B的左儿子还是右儿子，设这个关系为K；将D与K关系相反的儿子的父亲记为B与K关系相同的儿子（这里即为D的右儿子的父亲记为B的左儿子）；将D与K关系相反的儿子的父亲即为B（这里即为把G的父亲记为B）；将B的父亲即为D；将D与K关系相反的儿子记为B（这里即为把D的右儿子记为B）；将D的父亲记为A。

最后要判断，如果A存在（即rotate到的位置不是根的话），要把A的儿子记为D。

总结：显而易见，rotate之后所有牵涉到变化的父子关系都要改变。以上的树需要改变四对父子关系，BG DG BD AB，需要三个操作（BG BD AB）。





step 3：update一下当前点和各个父结点的各个值

【代码】(合并了left_rotate&right_rotate)

```
void rotate(LL x){
1:   LL y=father[x],z=father[y],t=get(x);
2:   if (z>0) son[z][get(y)]=x; son[y][t]=son[x][t^1];
3: if (son[y][t]>0) father[son[y][t]]=y;
4:    father[x]=z; father[y]=x; son[x][t^1]=y;
5: pushup(y); pushup(x);}
```

生动形象的解释:

2:我的儿子过继给我的爸爸；同时处理父子两个方向上的信息

3:我的爷爷成了我的爸爸

4:我给我爸爸当爹，我爸爸管我叫爸爸

5:分别维护信息

【splay操作】

其实splay只是rotate的发展。伸展操作只是在不停的rotate，一直到达到目标状态。如果有一个确定的目标状态，也可以传两个参。

splay的过程中需要分类讨论，如果是三点一线的话（x，x的父亲，x的祖父）需要先rotate x的父亲，否则需要先rotate x本身

```
void make(LL x,LL y) {//splay前先处理(下传)沿途的lazy标记
top=0; while (x!=y) th[++top]=x,x=father[x]; while (top) pushdown(th[top--]);}
//这个pushdown不同于之前更新size的pushup,而是指处理(下传)lazy标记的函数
void splay(LL x,LL y){make(x,y);
    while (father[x]!=y){
    LL z=father[x];
    if (father[z]!=y)
        if (get(x)==get(z)) rotate(z);
        else rotate(x);
    rotate(x);}
    update(x);
    if (!y) root=x;}
```

让我们先从一道模板题开始吧！

bzoj3224 || luogu3369 普通平衡树

您需要写一种数据结构（可参考题目标题），来维护一些数，其中需要提供以下操作：

- 1. 插入x数
- 2. 删除x数(若有多个相同的数，因只删除一个)
- 3. 查询x数的排名(若有多个相同的数，因输出最小的排名)
- 4. 查询排名为x的数
- 5. 求x的前驱(前驱定义为小于x，且最大的数)
- 6. 求x的后继(后继定义为大于x，且最小的数)
- 1) n的数据范围：n<=100000
- 2) 每个数的数据范围：[-2e9,2e9]

一些操作(插入insert,查询排名为x的点的权值find)

然后就是基本的insert操作了，要insert首先得找对该节点的存放位置，根据大小关系，若num小于当前点则肯定在左子树，若等于则就是该点，若大于则是在右子树

```
void insert(int x){
	int y=root,z=0;
	while (y && x!=val[y])
z=y,y=son[y][x>val[y]];
	if (y) cnt[y]++;
	else{
		y=++num;
		if (z) son[z][x>val[z]]=y;
		son[y][0]=son[y][1]=0;
		val[y]=x; father[y]=z;
		size[y]=cnt[y]=1;}
	splay(y,0);}
```

如果当前点有左子树，并且x比左子树的大小小的话，即向左子树寻找；

否则，向右子树寻找：先判断是否有右子树，然后记录右子树的大小以及当前点的大小（都为权值），用于判断是否需要继续向右子树寻找。

此处以寻找第x小为例。

```
int find(int x){
	int y=root;
	while (y)
if (son[y][0] && x<=size[son[y][0]])
	y=son[y][0];
	else{
	int tmp=size[son[y][0]]+cnt[y];
	if (x<=tmp) return key[y];
	x-=tmp; y=son[y][1];}
return -1;}
```

一些操作(查询权值为x的点的排名rank,求x的前驱pre(后继next))

和其它二叉搜索树的操作基本一样。但是区别是：

如果x比当前结点小，即应该向左子树寻找，ans不用改变（设想一下，走到整棵树的最左端最底端排名不就是1吗）。

如果x比当前结点大，即应该向右子树寻找，ans需要加上左子树的大小以及根的大小（这里的大小指的是权值）。

不要忘记了再splay一下

- int rank(int x){
- int y=root,ans=0;
- while (y)
- if (x<key[y]) y=son[y][0];
- else{
- ans+=size[son[y][0]];
- if (x==key[y]) {splay(y,0);
- return ans+1;}
- //此时x和树中的点重合，树中不允许有两个相同的点
- ans+=cnt[y],y=son[y][1];}
- return -1;}

前驱（后继）定义为小于（大于）x，且最大（最小）的数

这类问题可以转化为将x插入，求出树上的前驱（后继），再将x删除的问题。

其中insert操作上文已经提到。

【pre/next操作】

这个操作十分的简单，只需要理解一点：在我们做insert操作之后做了一遍splay。这就意味着我们把x已经splay到根了。求x的前驱其实就是求x的左子树的最右边的一个结点，后继是求x的右子树的左边一个结点（想一想为什么？）

- int pre(int x){
- splay(x,0); int y=son[x][0];
- while (son[y][1]) y=son[y][1];
- return y;}

- int next(int x){
- splay(x,0); int y=son[x][1];
- while (son[y][0]) y=son[y][0];
- return y;}

删除操作delete

删除操作是最后一个稍微有点麻烦的操作。

step 1：随便find/rank一下x。目的是：将x旋转到根。

step 2：那么现在x就是根了。如果cnt[root]>1，即不只有一个x的话，直接-1返回。

step 3：如果root并没有孩子，就说名树上只有一个x而已，直接clear返回。

step 4：如果root只有左儿子或者右儿子，那么直接clear root，然后把唯一的儿子当作根就可以了(father赋0，root赋为唯一的儿子)

剩下的就是它有两个儿子的情况。

step 5：我们找到新根，也就是x的前驱（x左子树最大的一个点），将它旋转到根。然后将原来x的右子树接到新根的右子树上（注意这个操作需要改变父子关系）。这实际上就把x删除了。不要忘了pushup新根。

```
void del(int x)
{
        rank(x); int y=root,z;
        if (cnt[y]>1) {cnt[y]--; pushup(y); return;}//有多个相同的数
        if (!son[y][0] && !son[y][1]) {clear(y); root=0; return;}
        if (!son[y][0]) {root=son[y][1]; father[root]=0; clear(y); return;}
        else if (!son[y][1]) {root=son[y][0]; father[root]=0; clear(y); return;}
        splay(pre(y),0); z=root;
        son[z][1]=son[y][1]; father[son[y][1]]=z;
        clear(y); pushup(z);
}
```

再举个栗子~

洛谷P3391 文艺平衡树

您需要写一种数据结构，来维护一个有序数列，其中需要提供以下操作：
翻转一个区间,例如原有序序列是5 4 3 2 1,翻转区间是[2,4]的话,结果是5 2 3 4 1

n,m≤100000

首先按照中序遍历建树，然后对于每次修改区间l,r，首先得提出这段区间，方法是将l的前趋l-1旋转到根节点，将r的后趋r+1旋转到根节点的右儿子，我们可以自己画图试试，容易发现经过这个操作后，根节点的右儿子的左子树（具体应该说是这个左子树的中序遍历）就是区间l-r。关键的翻转时，因为树是中序遍历（左根右），所以我们只要将l-r（前面所说的根节点的右儿子的左子树）这个区间子树左右儿子的节点交换位置（这样再中序遍历相当于右根左，即做到了翻转操作）。关键是翻转的优化，我们用到懒惰标记lazy[x]（表示x是否翻转），每次翻转时只要某个节点有标记且在翻转的区间内，则将标记下放给它的两个儿子节点且将自身标记清0，这样便避免了多余的重复翻转。

主要代码：

```
//上文提到的下传lazy标记的函数
void pushdown(int x){
        if (x && tag[x]){
                tag[son[x][0]]^=1;
                tag[son[x][1]]^=1;
        swap(son[x][0],son[x][1]);
                tag[x]=0;}}


void reverse(int x,int y)
{
   int l=x-1,r=y+1;
   l=find(l),r=find(r);
   splay(l,0); splay(r,l);
   int pos=son[root][1];
   pos=son[pos][0]; tag[pos]^=1;
}//翻转操作
```

Treap,替罪羊树

让我们再回到例题：普通平衡树

Treap:随机数据下表现良好的原因竟是还是旋转——不转不是中国人……

合理的旋转操作可以是BST变得更扁(you)平(xiu)。那么问题来了，怎么样才算合理呢？？？根据研究发现，在随机的数据下，普通的BST可以虐杀其他一切高级的树。Treap的思想就是利用“随机”来创造平衡的条件。因为在旋转的过程中必须维持BST的性质，所以Treap就把“随机”作用在堆性质上。

Treap的所有操作时间复杂度期望都是O（logn）的

其实Treap就是Tree和Heap的合成词啦。Treap在插入每个新节点的时候，就给该节点随机生成一个额外的权值。然后像二叉堆的插入过程一样，自底向上依次检查,当某个节点不满足大根堆性质时,就执行单旋转,是该节点与其父节点的关系发生对换。

特别的，对于删除操作，因为Treap支持旋转，我们可以将要删除的节点转到叶子节点，然后直接删除，就可以避免信息更新的问题啦。

也即:前文所提到的点的权值key[]由rand()随机生成(插入时随机key[x])

重量平衡树——替罪羊树

替罪羊树：暴力即优雅

如果在一棵平衡的二叉搜索树内进行查询等操作，时间就可以稳定在log(n)，但是每一次的插入节点和删除节点，都可能会使得这棵树不平衡，最坏情况就是退化成一条链，显然我们不想要这种树，于是各种维护的方法出现了，大部分的平衡树都是通过旋转来维护平衡的，但替罪羊树就很厉害了，一旦发现不平衡的子树，立马拍扁重建，这就是替罪羊树的核心：暴力重建

替罪羊树的主要思想就是

将不平衡的树压成一个序列,然后暴力重构成一颗平衡的树.

这里的平衡指的是:对于某个 ***0.5<=alpha<=1*** 满足 ***size(lson(x))<=alpha*size(x)*** 并且 ***size(rson(x))<=alpha*size(x)***, 即这个节点的两棵子树的 *size* 都不超过以该节点为根的子树的 *size* ,那么就称这个子树(或节点)是平衡的, ***alpha*** 最好不要选 *0.5* ,容易T飞,一般选 ***0.75*** 就挺好的.

至于复杂度,虽说是重构,但复杂度并不高,压扁和重建都是递归操作,也就是像线段树一样的 *log* 级别,由于平衡的限制,插入,删除,即查询等操作也会控制在一个较低的级别,均摊下来替罪羊树的总复杂度是 ***O(logn)*** 的.

部分操作(回收节点recycle,重构rebuild)

recycle:
压扁成一个序列,按大小顺序回收节点

rebuild:
重构

```
void recycle(int id){
 if(son[id][0]) recycle(son[id][0]);
 cur[++cnt]=id;
 if(son[id][1]) recycle(son[id][1]);}

void rebuild(int id){
 cnt=0; recycle(id);
 int x=father[id],y=(son[father[id]][1]==id);
 int tmp=build(1,cnt);
 father[son[x][y]=tmp]=x;
 if(id==root) root=tmp;}
```

部分操作(保持平衡balance,插入insert)

balance:

平衡限制,这里的alpha取0.75

insert:

插入维护序列,左小右大

插入往往会导致不平衡,这时只需要重建不平衡的子树即可

```
bool balance(RG int id){
 return (db)t[id].size*al>=(db)t[ t[id].son[0] ].size
  && (db) t[id].size*al>=(db)t[t[ id].son[1] ].size;}
```

```
void insert(int x){
 int id=root,tmp=++sum;
 size[tmp]=1,num[tmp]=x;
 while(id){
  size[id]++;
  bool cur=(x>=num[id]);
  if(son[id][cur]) id=son[id][cur];
  else{father[son[id][cur]=tmp]=id;
       break;}}
 int flag=0;
 for(int k=tmp;k;k=father[k]) if(!balance(k)) flag=k;
 if(flag) rebuild(flag); }
```

删除操作erase

删除操作需要找到左子树的最后一个节点或右子树的第一个节点来顶替,优先找左子树

在删除替罪羊树上的一个元素时,我们并不会将其**暴力删除**.虽然替罪羊树在**重构**时非常暴力,但它的暴力是有选择性的,而是标记这个节点不存在,并在计算它所在子树大小时将实际大小减1.这个思想是非常实用的,在许多地方我们都会用到。

```
void erase(int id){
  if(son[id][0] && son[id][1]){
  int tmp=son[id][0];
  while(son[tmp][1]) tmp=son[tmp][1];
  num[id]=num[tmp]; id=tmp;}
  int k=(son[id][0])?son[id][0]:son[id][1];
  int cur=(son[father[id]][1]==id);
  father[son[father[id]][cur]=k]=father[id];
  for(int i=father[id];i;i=father[i]) size[i]--;
  if(id==root) root=k;}
```

其它操作类似，见上文……

题目环节！

内容由浅入深，难易交错，欢迎同学们踊跃抢答！