Leafy Tree

by Remmina

为什么要学**Leafy Tree**

- ~~因为这是我的任务~~
- 因为它长得像线段树比较好想
- 因为它跑得比较快
- 因为它有时候还挺好写的
- 因为它可以非常方便地实现可持久化
- ~~因为它看起来非常装逼~~

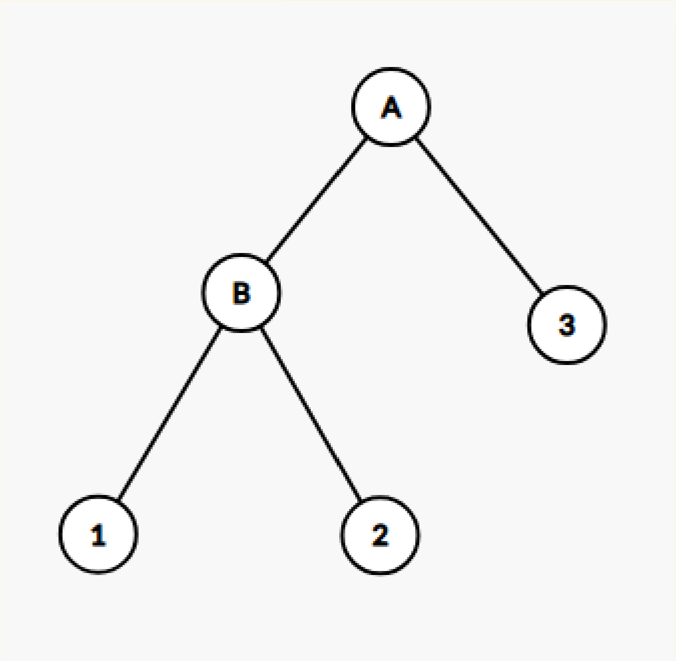
行吧我编不下去了

什么是**Leafy Tree**

你可以认为是一种动态的线段树

它是完全二叉树

大概长这样：



其中1,2,3为实点，A,B,C为辅助节点

> Leafy Tree 是一种二叉树,其每个节点要么为叶子,要么有两个儿子。其信息完全储存在叶子上面,每个非叶节点存储的信息是其儿子的信息的合并。它的结构和线段树类似,同时也可以将其看作是二叉搜索树的 kruskal 重构树

> 考虑用一棵 Leafy Tree 维护一个集合。每个叶子节点存储集合中一个值,每个非叶节点保存它右儿子的值(即子树最大值)。

——《Leafy Tree 及其实现的加权平衡树》 王思齐

其实你保存左子树的值也是没有问题的

Leafy Tree实现二叉搜索树

我们先定义一下节点结构体

```
struct N {int s[2], d;} e[NS << 1];
```

不难发现节点数目要开两倍（线段树要4倍，嘿嘿嘿）

其中`s[0],s[1]`分别表示左右儿子，`d`表示节点键值。

Push Up

```
void pup(int a) {e[a].d = e[e[a].s[1]].d;}
```

建树

```
int Build(int l, int r)
{
    int a = New(0);
    if (l == r) {e[a].sz = 1, e[a].d = l; return a;}
    int mid = (l + r) >> 1;
    e[a].s[0] = Build(l, mid);
    e[a].s[1] = Build(mid + 1, r);
    pup(a);
    return a;
}
```

建树是不是特别像线段树QwQ是不是感觉特别亲切

~~没事不用着急等下提取区间的时候你就不会这样想了~~

在Leafy Tree中，左儿子的值是左子树代表区间的最大值，你也可以认为是线段树中的`mid`

因此查找一个元素就一直向下递归，如果当前节点是辅助节点就判断搜索键值与其左儿子的值进行比较，如果键值$\leq$左儿子的值就去左儿子中搜索，否则去右儿子搜索。

如果当前节点是叶节点，则找到目标

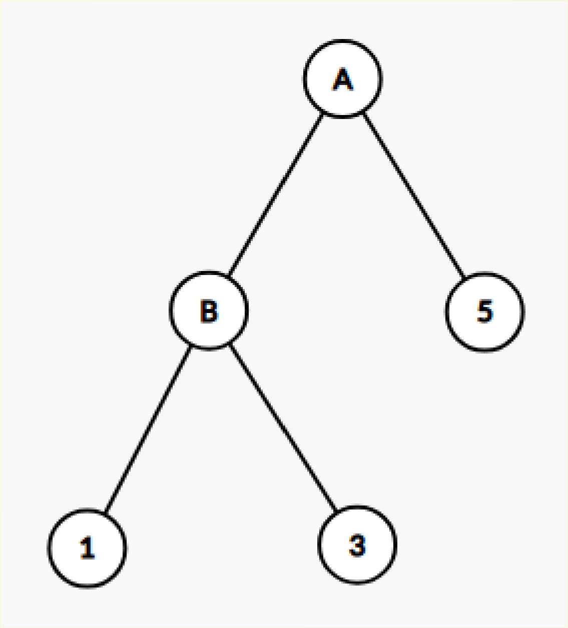
（如果叶节点的值还是与键值不同说明出数据的在逗你，比如可持久化平衡树那题）

搜索

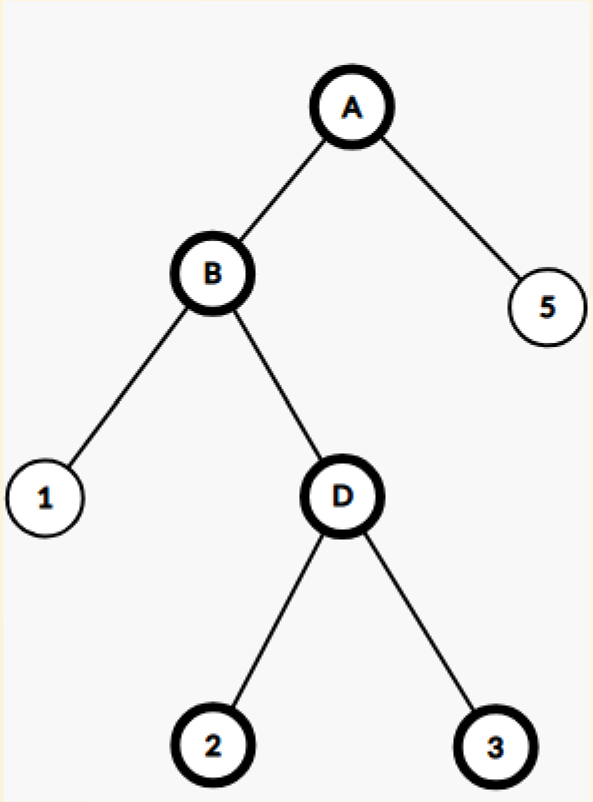
```
int Search(int d, int a)
{
    if (!e[a].s[0]) // 当前为叶节点
    {
        if (d != e[a].d) puts("Error!"), exit(0);
        return a;
    }
    return Search(d, e[a].s[d > e[e[a].s[0]].d]);
}
```

Leafy Tree有一个好，就是它的实点全是叶节点，因此省去了别的平衡树再删除和添加节点的时候的各种分类讨论

因此插入的话就先找到对应的叶节点，然后新建两个新的叶节点作为该叶节点的儿子，一个存当前插入的值，一个存刚刚找到的叶节点的值（相当于把该叶节点下放），然后刚刚找到的叶节点就变成了辅助节点了。



插入2＝＞



插入

```
void Insert(int d, int& a)
{
    if (!a) {a = New(d); return;}
    if (!e[a].s[0])
    {
        e[a].s[0] = New(d);
        e[a].s[1] = New(e[a].d);
        if (d > e[a].d) swap(e[a].s[0], e[a].s[1]);
    }
    else Insert(d, e[a].s[d > e[e[a].s[0]].d]);
    pup(a);
}
```

删除就是个逆过程

找到要删掉的节点的父节点，用不该删掉的那个儿子覆盖它即可。

删除

```
void Erase(int d, int& a)
{
    if (!e[a].s[0]) {a = 0; return;}
    int x = (d > e[e[a].s[0]].d);
    if (!e[e[a].s[x]].s[0])
    {
        if (e[e[a].s[x]].d != d)
            puts("Error!"), exit(0);
        e[a] = e[e[a].s[x ^ 1]];
    }
    else Erase(d, e[a].s[x]);
    pup(a);
}
```

你看这是不是炒鸡好写，代码这么短，省掉判数据错误还可以更短，在它面前连Splay的删除都显得好麻烦

是不是有心动的感觉?

因此它是实现二叉搜索树的不二选择

_(:3 」∠)_

如果要实现平衡的话就有点有趣了

Leafy Tree实现加权平衡树

> 加权平衡树(Weight Balanced Tree,也叫$BB[\alpha]$树,重量平衡树)是一种储存子树大小的二叉搜索树。

像替罪羊就是通过添加一个阈值，如果不平衡度超过阈值则重构

更抽象地，如果一棵二叉搜索树的每个节点都满足：

$$min(a.left.size, a.right.size) \geq \alpha \times a.size$$

则说该树加权平衡

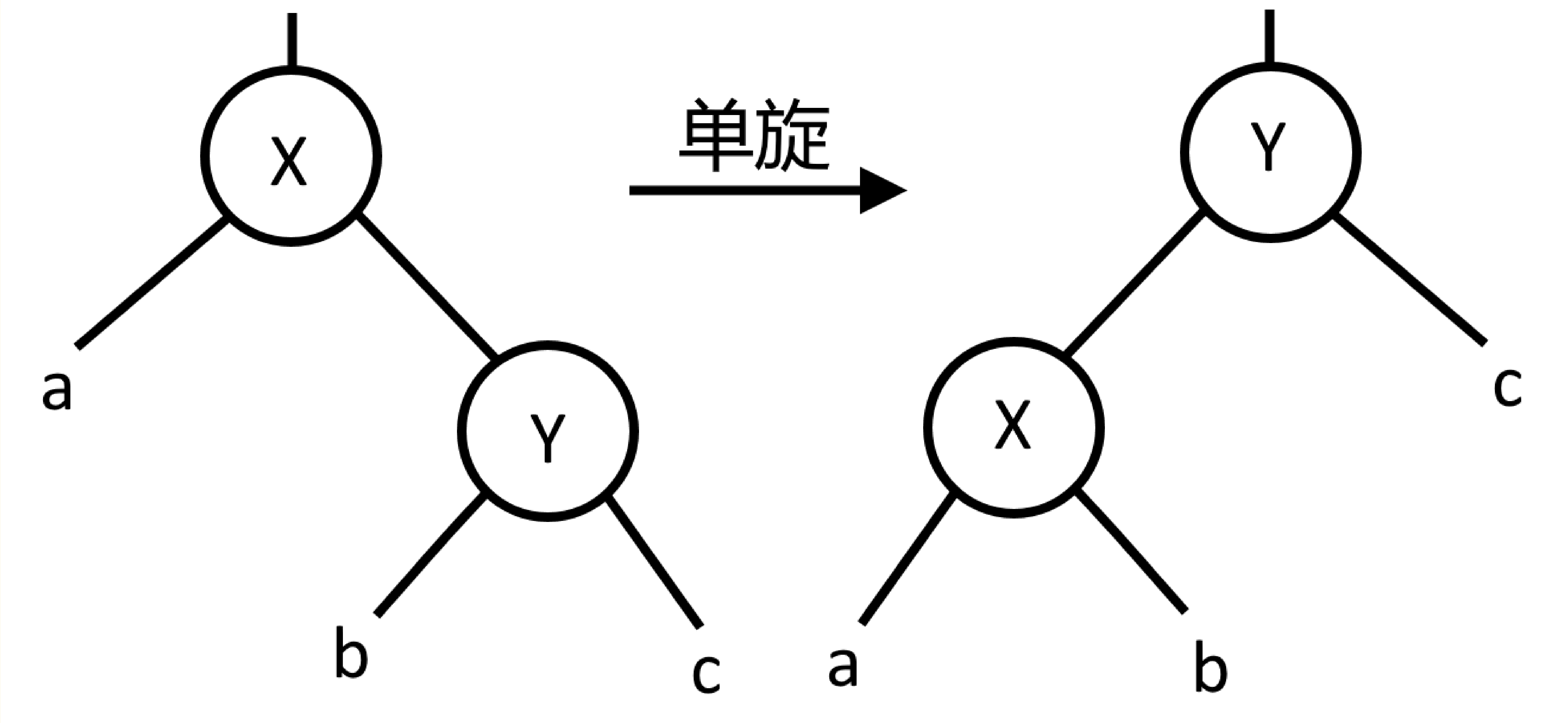
因此我们需要额外记录每个点的权值（重量）

首先$\alpha \in (0, \frac{1}{2})$

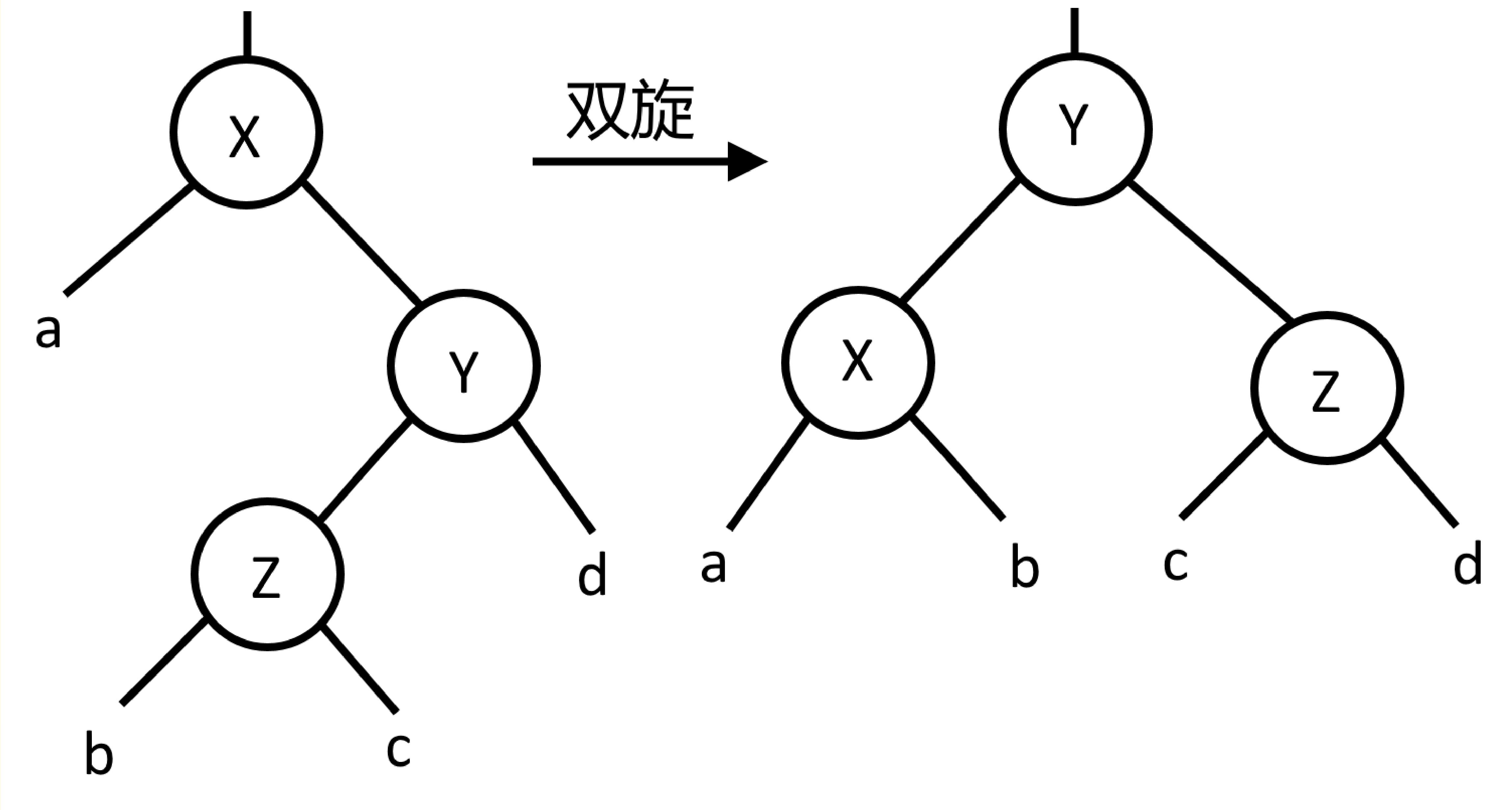
然后有一棵加权平衡树的树高满足$h \leq log_{\frac{1}{1-\alpha}} n$

与替罪羊不同，Leafy Tree通过旋转来调整子树大小以达到加权平衡

同时Leafy Tree的节点重量为该节点的子树中叶节点的数目，而不是节点数目



~~论文里的图怎么这么大一页都装不下~~



X
a
Y
Z
d
b
c
双旋
Y
X
Z
a
b
c
d

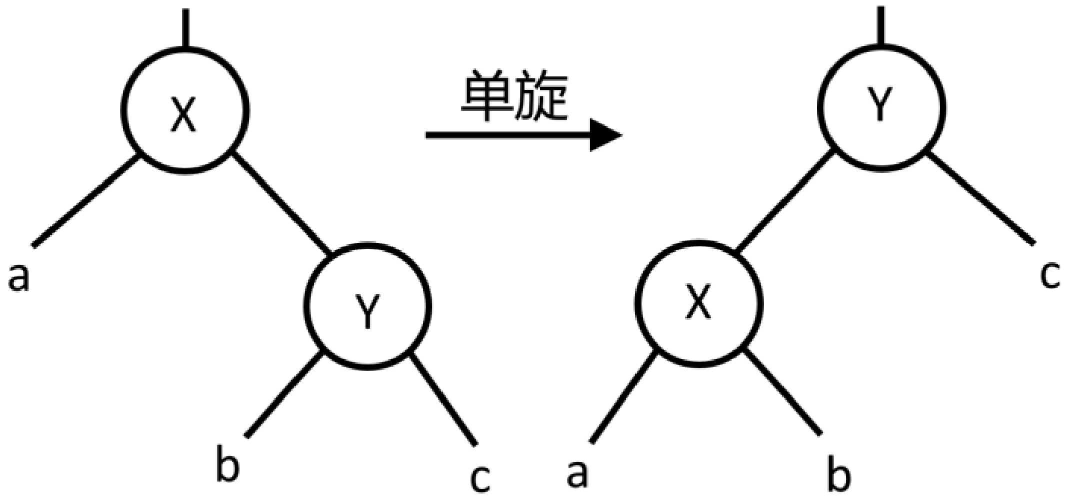
复杂度证明论文没有给出详细解释，网上也找不到相关资料，因此~~由于篇幅限制~~我也不会给出证明（大雾

你可以感性地理解一下吖。

（行吧我照搬算了）

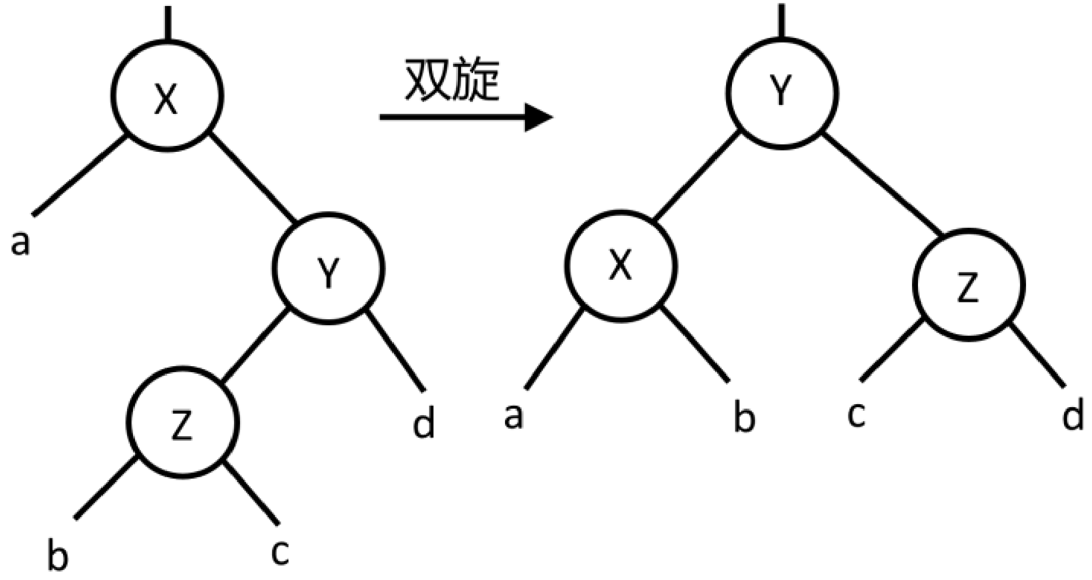
假设这棵树的一个子树 T 中，除根节点外的所有节点均满足 α 加权平衡。我们可以用一次单旋或双旋操作使 T 满足 α 加权平衡。

定义 x 的平衡度表示 $\frac{weight[x.left]}{weight[x]}$ 。



如上图，考虑一次单旋，设 X, Y 旋转前平衡度为 ρ_1, ρ_2 ，旋转后为 γ_1, γ_2 。

可以得到 $\gamma_1 = \frac{\rho_1}{\rho_1+(1-\rho_1)\rho_2}, \gamma_2 = \rho_1 + (1 - \rho_1)\rho_2$ 。



如上图，考虑一次双旋，设 X,Y 旋转前平衡度为 ρ_1,ρ_2,ρ_3 ，旋转后为 $\gamma_1,\gamma_2,\gamma_3$ 。

可以得到 $\gamma_1=\frac{\rho_1}{\rho_1+(1-\rho_1)\rho_2\rho_3},\gamma_2=\rho_1+(1-\rho_1)\rho_2\rho_3,\gamma_3=\frac{\rho_2(1-\rho_3)}{1-\rho_2\rho_3}$ 。

假设 $\rho_1<\alpha$ ，在只插入或删除一个节点的情况下，ρ_1 最小为 $\frac{\alpha}{2-\alpha}$ ，可在子树 a 中原有 2 个节点，现在删除其中一个时取到。

。。。

具体地来说定义俩常数：

α和β

$\beta = \frac{1-2\alpha}{1-\alpha}$

然后当X不满足加权平衡时：

- 如果Y的平衡度$\rho_2 < \beta$则进行一次双旋
- 否则只是单旋，把Y旋到X处

这样就能保证平衡度在$[\alpha, 1 - \alpha]$之间啦！

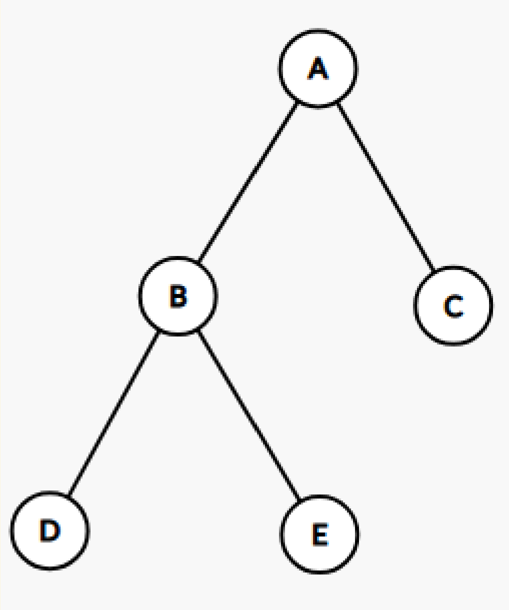
实验证明α取$1 - \frac{\sqrt{2}}{2}$时缀快

首先定义常量

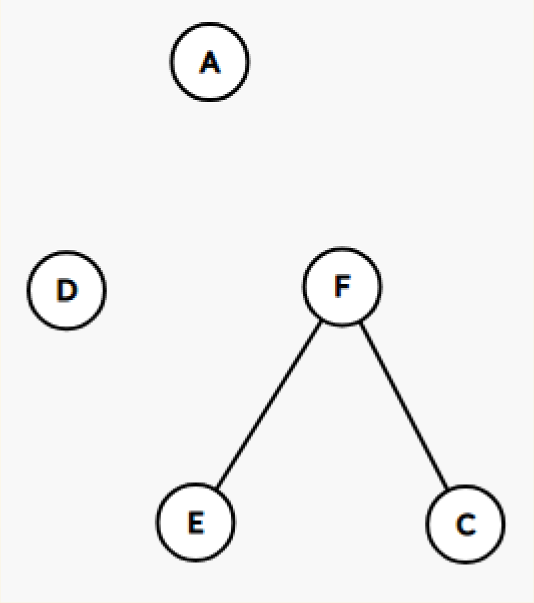
```
const double alp = 1 - sqrt(2) * 0.5;
const double bta = (1 - 2 * alp) / (1 - alp);
```

旋转操作可以不像Splay那样写，可以直接这样：

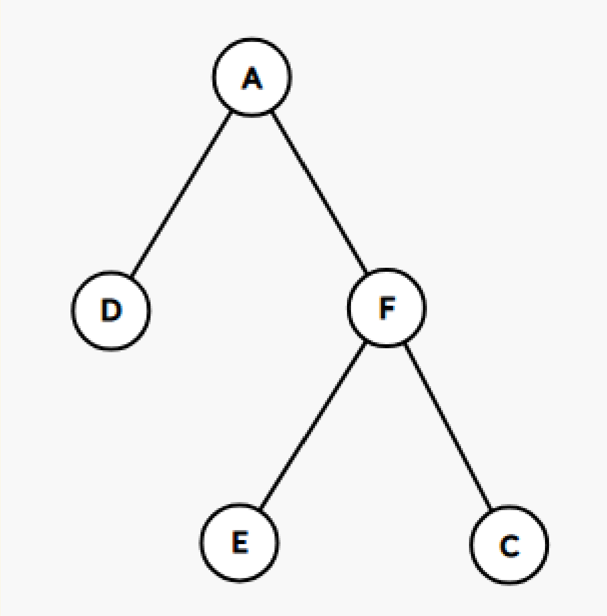
原树：



删除点B然后新建节点作为C,E的的父亲（合并C,E信息）：



连接A,D与A,F即可



这样可以保持当前指针不失效

这样代码就很短了：

```
int Mix(int a, int b) // 合并两个节点
{
    int res = New(0);
    e[res].s[0] = a, e[res].s[1] = b, pup(res);
    return res;
}

void rot(int a, int q) // q = 0 左旋 q = 1 右旋
{
    if (!e[a].s[0]) return;
    int tmp = e[e[a].s[q]].s[q];
    if (q) e[a].s[0] = Mix(e[a].s[0], e[e[a].s[1]].s[0]);
    else e[a].s[1] = Mix(e[e[a].s[0]].s[1], e[a].s[1]);
    e[a].s[q] = tmp;
}
```

Maintain函数就这样写呗

```
void mt(int a)
{
    if (!e[a].s[0]) return;
    int q;
    if (e[e[a].s[0]].sz < e[a].sz * alp) q = 1;
    else if (e[e[a].s[1]].sz < e[a].sz * alp) q = 0;
    else return;
    if (e[e[e[a].s[q]].s[q ^ 1]].sz
        >= e[e[a].s[q]].sz * bta)
            rot(e[a].s[q], q ^ 1);
    rot(a, q);
}
```

对了这里温馨提示一下，Leafy Tree不像Splay不可能去rotate一个空节点，因为Splay的旋转是旋转当前节点，Leafy Tree可能会旋转儿子的儿子节点~~孙子节点~~，因此建议Leafy Tree的所有操作都判断一下是否是叶节点防止把空节点旋上来啥的。

Push Up

```
void pup(int a)
{
    if (!e[a].s[0]) return;
    e[a].sz = e[e[a].s[0]].sz + e[e[a].s[1]].sz;
}
```

插入

```
void Insert(int d, int& a)
{
    if (!a) {a = New(d); return;}
    if (!e[a].s[0])
    {
        e[a].s[0] = New(d), e[a].s[1] = New(e[a].d);
        if (d > e[a].d) swap(e[a].s[0], e[a].s[1]);
    }
    else Insert(d, e[a].s[d > e[e[a].s[0]].d]);
    pup(a), mt(a);
}
```

(其实就是在push up之后加了个`mt(a)`...)

删除

```
void Erase(int d, int& a)
{
    if (!e[a].s[0]) {a = 0; return;}
    int x = (d > e[e[a].s[0]].d);
    if (!e[e[a].s[x]].s[0])
    {
        if (e[e[a].s[x]].d != d)
            puts("Error!"), exit(0);
        e[a] = e[e[a].s[x ^ 1]];
    }
    else Erase(d, e[a].s[x]);
    pup(a), mt(a);
}
```

找前驱

```
int Pre(int d, int a)
{
    if (!e[a].s[0])
    {
        if (e[a].d < d) return e[a].d;
        return INT_MIN;
    }
    if (e[e[a].s[0]].d < d)
        return max(e[e[a].s[0]].d, Pre(d, e[a].s[1]));
    return Pre(d, e[a].s[0]);
}
```

找后继（和找前驱有点不一样，是由于Leafy Tree只保存右儿子值的性质）

```
int Nxt(int d, int a)
{
    if (!e[a].s[0])
    {
        if (e[a].d > d) return e[a].d;
        return INT_MAX;
    }
    if (e[e[a].s[0]].d > d) return Nxt(d, e[a].s[0]);
    return Nxt(d, e[a].s[1]);
}
```

第K大

```
int Kth(int k, int a)
{
    if (!e[a].s[0]) return e[a].d;
    if (e[e[a].s[0]].sz <= k)
        return Kth(k - e[e[a].s[0]].sz, e[a].s[1]);
    return Kth(k, e[a].s[0]);
}
```

询问排名

```
int Order(int d, int a)
{
    if (!e[a].s[0]) return 0;
    if (e[e[a].s[0]].d < d)
        return Order(d, e[a].s[1]) + e[e[a].s[0]].sz;
    return Order(d, e[a].s[0]);
}
```

Leafy Tree的区间操作

想想就刺激

人家本来就是线段树，区间操作太简单了

找到对应的$logn$个节点打标记就行了！！！

岂不是太爽了！！！

什么区间加区间乘区间。。。

区间Reverse...

~~下面有请Boshi同学为我们表演Leafy Tree的区间Reverse~~

Leafy Tree作为平衡线段树（雾）确实能轻松胜任区间加区间乘等线段树能完成的操作

但是区间取反（文艺平衡树）线段树是做不了的啊！

难道只能放弃了吗？！

不！

~~行吧我只是充字数而已~~

既然要区间取反，就得提取区间，把区间提取为一颗独立的树，然后打标记。

Leafy Tree可不是死板的线段树，它一定能解决这个问题的！

Leafy Tree可是灵活的。。。

由旋转实现的加权平衡完全二叉搜索线段树啊！

。。。

提取区间又不能Splay到根就只能像无旋Treap一样Split咯

可是Leafy Tree得保持其完全二叉树的性质

于是Split就不是你想分就分，Split的时候还得Merge保持性质

因此这个操作确实很令人不爽

首先讲Merge

设当前是`Merge(a, b)`，如果a树和b树已经加权平衡了

$$Max(a.size, b.size) \le Min(a.size, b.size) \times \alpha$$

那就直接新建一个节点作为a和b的父亲然后返回该点（`Mix(a, b)`）

如果还没有平衡的话就判断以下哪种情况能达到平衡：

- 如果a树大一些：
  - 第一种：把b与a的右子树合并
  - 第二种：把b与a的右子树的右子树合并，把a的左子树与a的右子树的左子树合并（相当于旋转了一下），再把它们合并起来
- 如果b树大一些：
  - 第一种：把a与b的左子树合并
  - 第二种：把a与b的左子树的左子树合并，把b的右子树与b的左子树的右子树合并（相当于旋转了一下），再把它们合并起来

代码可以这么写：

```
int Merge(int a, int b) // 合并两颗Leafy Tree
{
    if (!a || !b) return (a | b);
    if (max(e[a].sz, e[b].sz) <= min(e[a].sz, e[b].sz) * alp) return Mix(a, b); // 已经平衡了
    if (e[a].sz > e[b].sz) // a的size更大
    {
        pdown(a);
        if (e[e[a].s[1]].sz + e[b].sz <= e[e[a].s[0]].sz * alp) // 如果第一种合并已经平衡了
            return Merge(e[a].s[0], Merge(e[a].s[1], b));
        pdown(e[a].s[1]); // 第一种合并不平衡，只能用第二种了
        int x = Merge(e[a].s[0], e[e[a].s[1]].s[0]);
        int y = Merge(e[e[a].s[1]].s[1], b);
        return Merge(x, y);
    }
    else // 同上
    {
        pdown(b);
        if (e[a].sz + e[e[b].s[0]].sz <= e[e[b].s[1]].sz * alp)
            return Merge(Merge(a, e[b].s[0]), e[b].s[1]);
        pdown(e[b].s[0]);
        int x = Merge(a, e[e[b].s[0]].s[0]);
        int y = Merge(e[e[b].s[0]].s[1], e[b].s[1]);
        return Merge(x, y);
    }
}
```

其中`pdown`表示push down，代码如下

```
void pdown(int a)
{
    if (!e[a].s[0]) return;
    if (e[a].tag)
    {
        swap(e[e[a].s[0]].s[0], e[e[a].s[0]].s[1]);
        swap(e[e[a].s[1]].s[0], e[e[a].s[1]].s[1]);
        e[e[a].s[0]].tag ^= 1, e[e[a].s[1]].tag ^= 1;
        e[a].tag = 0;
    }
}
```

有了Merge其实Split挺好写的

和Treap的Split差不多

只不过当你Split了左子树时你要把左子树没有被Split的部分和右子树Merge起来

当你Split了右子树时（此时左子树属于要被Split的部分）你要把左子树和右子树被Split的部分Merge起来

Split

```cpp
#define FIR first
#define SEC second

typedef pair<int, int> PII;

PII Split(int a, int x)
{
    if (!x) return PII(0, a);
    if (!e[a].s[0]) return PII(a, 0);
    pdown(a); PII tmp;
    if (x <= e[e[a].s[0]].sz)
    {
        tmp = Split(e[a].s[0], x);
        return PII(tmp.FIR, Merge(tmp.SEC, e[a].s[1]));
    }
    else
    {
        tmp = Split(e[a].s[1], x - e[e[a].s[0]].sz);
        return PII(Merge(e[a].s[0], tmp.FIR), tmp.SEC);
    }
}
```

有了Split和Merge我们就能轻松解决文艺平衡树问题了。

```
void Rev(int l, int r)
{
    PII X = Split(root, l - 1);
    PII Y = Split(X.SEC, r - l + 1);
    e[Y.FIR].tag ^= 1;
    swap(e[Y.FIR].s[0], e[Y.FIR].s[1]);
    root = Merge(X.FIR, Merge(Y.FIR, Y.SEC));
}
```

细心的读者们（也许也不是读者）一定已经发现这样空间会炸的很厉害

因为每次Mix都会新建一个节点，Merge一次开销很大，必须要垃圾回收

这也没什么难的就是代码有点长。。。可以去附件里看，或者去我博客里面里看

可持久化Leafy Tree

既然都说了Leafy Tree是线段树，那可持久化就和主席树差不多了，差不多一模一样

~~TXC：爽德亿匹~~

首先复制根节点

```
root[i] = root[v];
```

然后插入

```
Insert(d, root[i], root[i]);
```

插入

```
void Insert(int d, int& x, int y)
{
    if (!y) {x = New(d); return;}
    x = New(0), e[x] = e[y];
    if (!e[x].s[0])
    {
        e[x].s[0] = New(d), e[x].s[1] = New(e[x].d);
        if (d > e[x].d) swap(e[x].s[0], e[x].s[1]);
    }
    else if (d > e[e[x].s[0]].d)
        Insert(d, e[x].s[1], e[y].s[1]);
    else Insert(d, e[x].s[0], e[y].s[0]);
    pup(x), mt(x);
}
```

注意一下Maintain的时候可能会旋转当前没有被复制过的节点，因此要记得复制

```
void mt(int a)
{
    if (!e[a].s[0]) return;
    int q;
    if (e[e[a].s[0]].sz < e[a].sz * alp) q = 1;
    else if (e[e[a].s[1]].sz < e[a].sz * alp) q = 0;
    else return;
    if (e[e[e[a].s[q]].s[q ^ 1]].sz
        >= e[e[a].s[q]].sz * bta)
    {
        int tmp = e[a].s[q];
        e[a].s[q] = New(0), e[e[a].s[q]] = e[tmp];
        // 注意复制!
        rot(e[a].s[q], q ^ 1);
    }
    rot(a, q);
}
```

删除

```cpp
void Erase(int d, int& x, int y)
{
    x = ++sz, e[x] = e[y];
    int q = (d > e[e[x].s[0]].d);
    if (!e[e[x].s[q]].s[0])
    {
        if (e[e[x].s[q]].d != d)
            puts("Error!"), exit(0);
        e[x] = e[e[x].s[q ^ 1]];
    }
    else Erase(d, e[x].s[q], e[y].s[q]);
    pup(x), mt(x);
}
```

似乎Leafy Tree就差不多这些了

顺带一提，Leafy Tree跑得确实还算快的，在可持久化平衡树一题中我的Leafy Tree用时不到Litble的无旋Treap的一半

网上Leafy Tree的资料真少，Remmina的代码全是自己瞎写的（雾），因此有可能写得比较丑（其实我个人觉得还行了23333），如果有更简洁的写法可以告诉Remmina!

（另外单旋是可以被卡的）

例题：

- 普通平衡树 BZOJ - 3224
- 文艺平衡树 BZOJ - 3223
- 可持久化平衡树 Luogu - 3835

（这三题我博客里都有题解）

谢谢

不加个副标题我就浑身难受**QwQ**