BST 总结

Cai

目 录

1 浅谈几种 BST

BST(Binary Sort Tree) 是二叉排序树，顾名思义，BST 是二叉树。

BST 按种类主要有伸展树 (Splay)，树堆 (Treap)，重量平衡树 (SBT)，替罪羊树，平衡树 (AVL)，红黑树 (RBT) 等种类，竞赛中一般用前四种，红黑树 (RBT) 被 STL 广泛运用

1.1 伸展树 (Splay)

伸展树的最主要的操作是 splay，也就是伸展操作，利用伸展操作可以轻易地提取一个区间，并对其进行操作，由于伸展树可以配合各种标记，以其强大的功能，所以伸展树有“序列之王”的美称，下面介绍伸展树的一些操作，默认 root 表示 Splay 的根，Splay 的左孩子的权值小于根节点，右孩子权值大于根节点，所给出的代码中，ch[x][0/1] 表示节点 x 的左/右孩子，pre[x] 表示 x 的父节点，完成操作维护节点 x 信息这一步放在 Update(x) 函数中

1.1.1 伸展操作 Splay(x, y)

伸展操作是 Splay 完成区间操作的基础

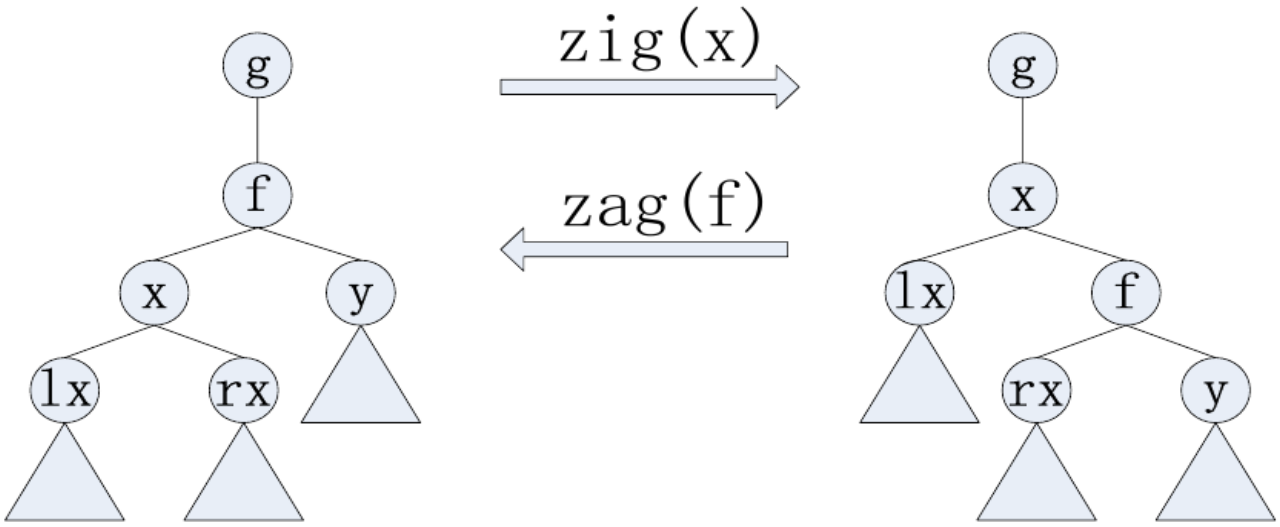
伸展操作可以通过旋转使节点 x 成为节点 y 的孩子，特别的，如果 $y = 0$ 则表示将 x 旋转至根节点，此时需要保证节点 x 在节点 y 的的子树中，由于这条性质，所以一般是选择将节点 x 旋转至根节点或者是根节点的下方，因为这样一定能保证 x 在 y 的子树中 (旋转至根节点的下方要特判节点 x 就是根节点的情况)。

为了保证时间复杂度，所以伸展树的旋转采用的是双旋，如果节点 x 的父节点 y，y 的父节点 z 这三者的权值是单调的 (或者说 y 是 z 的左孩子且 x 是 y 的左孩子或者说 y 是 z 的右孩子且 x 是 y 的右孩子孩子)，此时先旋转父节点 y，在旋转节点 x，否则直接旋转节点 x。如果采用单旋的话就会出现把一条链旋转成为另一条链的可能性，这样子就无法保证时间复杂度。

伸展部分代码：

```
//Rotate(x, y)表示旋转函数，y=0时表示右旋zag，y=1时表示左旋zig
//Splay(rt, goal)表示将节点rt旋转至goal的下方，goal=0是表示旋转至根
void Splay(int rt, int goal) {
    int x, y, kind;
    while(pre[rt] != goal)
        if(pre[pre[rt]] == goal) Rotate(rt, ch[pre[rt]][0]==rt);
        else {
            x = pre[rt], y = pre[x], kind = ch[y][1]==x;
            if(ch[x][kind] == rt) Rotate(x, !kind), Rotate(rt, !kind); //双旋
            else Rotate(rt, kind), Rotate(rt, !kind); //单旋
        }
    if(!goal) root = rt;
}
```

1.1.2 旋转操作 Rotate(x, y)



由图可知执行旋转操作时先记录节点 a 的父亲 b，先处理 a 的子树与 a 的父亲的关系：节点 a 的左 (zag)/右 (zig) 子树替代 a 的位置成为 b 的孩子，节点 y 替代节点 x 的左 (zag)/右 (zig) 子树成为节点 x 的孩子，在处理 y 的父亲与节点 x 的父子关系即可。

由上面的分析可知，其实左旋和右旋所执行的操作是类似的，所以可以将他们一起写进一个函数里面

```
//Rotate(rt, kind)表示旋转函数，kind=0时表示右旋zag，kind=1时表示左旋zig
void Rotate(int rt, int kind) {
    int x = pre[rt];
    pre[ch[x][!kind]=ch[rt][kind]] = x; //节点rt的左(kind=0)/右(kind=1)子树替代rt的位置成为x的孩子
    if(pre[x]) ch[pre[x]][ch[pre[x]][1]==x] = rt;
    pre[rt] = pre[x];
    pre[ch[rt][kind]=x] = rt; //节点x替代节点rt的左(kind=0)/右(kind=1)子树成为节点rt的孩子
    Update(x), Update(rt);
}
```

1.1.3 分裂操作 Split(l, r)

分裂操作用于从伸展树中提取一段区间，以进行区间操作。

对于按区间建的 Splay 而言，将位置 $l-1$ 旋转至根节点，再将位置 $r+1$ 旋转至根节点的下方，此时区间 $[l,r]$ 就是根节点的右孩子的左孩子，而从需要旋转位置 $l-1$ 和位置 $r+1$，所以在实际解决问题中，一般在伸展树中加入两个无用的节点，用来防止 $l-1, r+1$ 不存在的情况

对于按权值建的 Splay 而言，将 l 的前驱 (小于 l 且最大的数) 旋转至根节点，将 r 的后继旋转至根节点的下方，此时区间 $[l,r]$ 就是根节点的右孩子的左孩子，同样，为了防止前去后继不存在的情况，需要在 Splay 中加入两个无用的节点

1.1.4 **定位操作** Order(x)

对于按区间建的 Splay 而言，定位操作会返回下标为 x 的位置，需要注意你可能在 Splay 中加了一个无用的表示位置为 0 的节点。

对于按权值建的 Splay 而言，定位操作会返回区间第 x 大的位置，同样需要注意可能在 Splay 中加了无用的节点

1.1.5 **求前驱后继操作** $\mathrm{Pre}(x), \mathrm{Nxt}(x)$

求节点 x 的前驱及后继，我的做法是先从根节点一路往下，如果遇到节点比 x 大就往左走，否则往右走，需要注意当前节点权值与 x 相等的情况，知道无法继续往下面走为止，然后在从当前节点往上爬，直到遇到一个节点比 x 大 (求后继)/小 (求前驱)，此时这个节点的值就是 x 的后继/前驱

1.2 **带旋转** Treap

Treap = Tree + Heap，意为树堆，是一种兼带 BST 和堆的性质的数据结构，树堆中每个节点都至少有两个值 v, p，v 表示节点的权值，p 表示节点的优先级，树堆除了满足平衡二叉树的一般性质：任意一个节点 x 的左孩子的 v 小于 x 的 v，其右孩子的 v 大于 x 的 v 外，还满足堆的性质：节点 x 的 p 大 (小) 于其孩子节点的 p，为了保证整棵树的深度，所以节点的 p 一般由随机得到，所以树堆的期望深度为 $O(\log N)$

带旋转 Treap 的一般操作有旋转 (见 Splay 部分)，插入删除求前驱后继等。

插入的一般方法是先将节点在满足平衡二叉树的性质的条件下插入到某一个节点的下方，若此时违反了堆的性质，则利用单旋将节点往上旋转，直到满足树堆的条件为止。

删除的方法则是通过旋转使得带删节点被旋转的叶子节点，注意删除的时候其他节点也要满足树堆的性质，然后直接删除就可以了。

1.3 **无旋转** Treap

无旋转 Treap 可以完成带旋转 Treap 的所有操作。

无旋转 Treap 的时间效率没有旋转 Treap 高，但是其舍弃了旋转操作，因此可以持久持久化，这也是无旋转 Treap 的一大用途。

无旋转 Treap 的基本操作是分裂 (Split) 和合并 (Merge)，基本上其他操作都是以这两个操作为基础的，求前驱后继排名还是和旋转 Treap 一样的方法

1.3.1 **分裂操作** $\mathrm{Split}(r, k)$

将一颗无旋转 Treap 分裂成为两个无旋转 Treap，分裂得到的两颗子树都具有 Treap 的特征，其中一颗含原 Treap 的前 k 个节点，也就是说一颗 Treap 拥有 r 的前 k 小的节点，另一颗含有余下的节点。分裂函数有两个返回值，记为 (x, y)，表示分裂得到的两颗子树的根节点。

分裂过程就是一个不断特判的过程，具体操作如下，如果 $k = 0$，那么直接返回 $(0, r)$；如果整棵树的大小等于 k，则返回 $(r, 0)$；如果左子树的大小等于 k，则断开左子树 (lch_r) 与根节点 r 的关系，返回 (lch_r, r)；如果左子树大小加一等于 k，则需要断开根节点与右子树 rch_r 的关系，返回 (r, rch_r)；如果左子树大小大于 k，那么我们无法在这一步直接分裂，此时需要递归左子树，得到的返回值 (x, y)，x 中肯定包含前 k 个节点，

所以我们需要合并 y 与根节点 r，我们幸运的发现，y 可以直接接在 r 的左子树上 (因为递归的时候相当于断开了根节点与左子树)，然后返回 (x, r) 即可；左子树大于根节点时递归右子树，剩下的同理

```
//pair是一个含两个元素的结构体
//Update是一个更新节点信息的函数，因为分裂函数会导致子树信息的改变
pair Split(int rt, int k) {
    pair ret;
    if(!k) return pair(0, rt);
    if(size[rt] == k) return pair(rt, 0);
    if(size[lch[rt]] == k)
        return ret.x=lch[rt], ret.y=rt, lch[rt]=0, Update(rt), ret;
    if(size[lch[rt]]+1 == k)
        return ret.x=rt, ret.y=rch[rt], rch[rt]=0, Update(rt), ret;
    if(size[lch[rt]] > k)
        return ret=Split(lch[rt], k), lch[rt]=ret.y, Update(rt), ret.y=rt, ret;
    ret = Split(rch[rt], k-size[lch[rt]]-1);
    return rch[rt]=ret.x, Update(rt), ret.x=rt, ret;
}
```

1.3.2 **合并操作** Merge(x, y)

合并操作与分裂操作刚好相反，通过分裂操作可以将一颗 Treap 分裂成两颗 Treap，而合并操作则是将两颗 Treap 合并成为一颗 Treap，合并操作有返回值，为合并得到的 Treap 的根节点。

两颗 Treap 可以执行合并操作 Merge(x, y) 的条件：x 内的最大值小于 y 内的最小值，因为对于 Treap 上的一个节点而言，它有两个值 (v, p)，为了保证合并之后得到的 Treap 满足堆的性质，所以我们合并时只会考虑关键字 p。具体步骤很简单：如果某一棵树为空，则返回另一棵树，如果 $p_x > p_y$，则让 x 成为树的根，递归 Merge(rch_x, y)，返回 x；反之同理

```
int Merge(int x, int y) {
    if(!x || !y) return x ^ y;
    if(rnk[x] < rnk[y]) return lch[y]=Merge(x, lch[y]), Update(y), y;
    return rch[x] = Merge(rch[x], y), Update(x), x;
}
```

1.3.3 **插入操作** Insert(r, x)

在有了分裂操作和合并操作之后插入就显得极其暴力了，首先找到 x 在 r 内的排名 k，然后执行 Split(r, k)，得到的两颗树 (a, b)，此时 a 内的所有节点点权都会比 x 小，而 b 内的所有节点点权都会比 y 小，所以直接再将其暴力合并回去就好了

1.3.4 **删除操作** Delete(r, x)

删除操作同样暴力，首先找到 x 的排名 k，执行分裂操作 Split(r, k)，得到的两颗树 (a, b) 中，a 中一定包含 x 且 x 是其中最大的元素，于是在执行分裂操作 Split$(a, k-1)$，得到两颗树 (c, d) 中，d 是仅包含 x 的单元素 Treap，所以再将将剩下的两颗树合并回去就好了，即执行 Merge(c, b)

1.4 子树大小平衡树 SBT

子树大小平衡树也是一种需要通过旋转来维持深度，保证时间复杂度的 SBT，与 Treap 不同的是，它不需要额外的标记，因为 SBT 是通过子树大小来判断是否平衡的，与 Splay 相比，SBT 能够保证深度。而且 SBT 的时间效率很高。

SBT 的平衡标准：$lch[x]$ 表示 x 的左孩子，$rch[x]$ 表示 x 的右孩子，$size[x]$ 表示 x 的子树大小

$$size[lch[x]] \geq \max(size[rch[lch[x]]], size[rch[rch[x]]])$$

$$size[rch[x]] \geq \max(size[lch[lch[x]]], size[lch[rch[x]]])$$

用一句话来说就是，任何一个节点的子树大小不小于其兄弟节点的任何一个孩子的子树大小

SBT 能支持 BST 的一般操作，如插入，删除，找前去后继，求排名等。除此之外，SBT 还有一个核心的函数 Maintain，通过这个函数，可以在插入破坏 SBT 性质的条件下，通过旋转调整树的结构，使其成为一颗 SBT

以下介绍 SBT 的主要操作，求前驱后继，排名等与普通 BST 相同

1.4.1 Maintain 操作Maintain(x, f)

Maintain 操作用于维护 SBT 的形态，使其保持平衡，当执行插入操作时，可能因为插入节点而使得 SBT 不再满足其平衡标准，此时调用 Maintain 函数使其满足平衡标准。

执行 Maintain 函数会使得 SBT 的高度下降，而当删除节点时，不会使得 SBT 的高度增加此时可以不调用 Maintain 函数

通过 SBT 的平衡标准可以看出总共存在四种情况可能破坏平衡,而有两种 ($size[lch[x]] < size[rch[lch[x]]]$ 与 $size[rch[x]] < size[lch[rch[x]]]$，$size[lch[x]] < size[rch[rch[x]]]$ 与 $size[rch[x]] < size[lch[lch[x]]]$) 是对称的，所以这里仅讨论两种情况

如图 1.4.1-1，一个原本满足平衡标准的 SBT，在子树 A 内插入了节点，导致 $size[A] > size[R]$

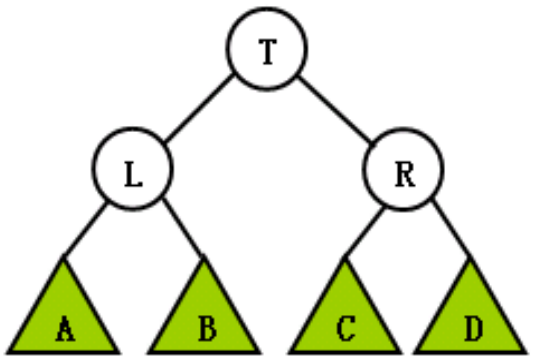


图 1.4.1-1

由 $size[A] > size[R]$，所以我们希望能让 R 成为 A 的兄弟子树的孩子，于是右旋节点 T(或者称为 zig(L))，得到图 1.4.1-2

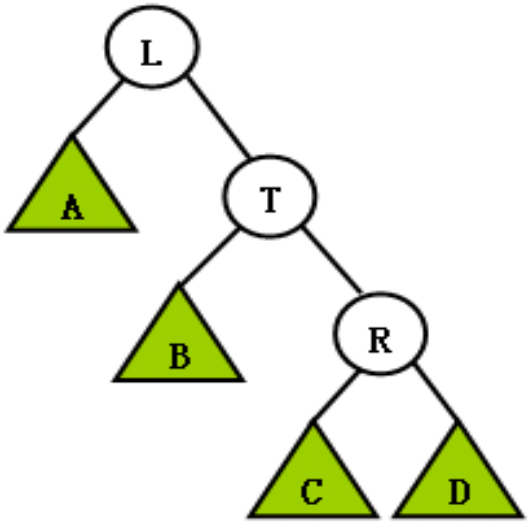


图 1.4.1-2

此时可能出现 $size[B] < \max(size[C], size[D])$ 的情况，递归 Maintain 函数维护子树 T 的形态，在维护完 T 的形态之后可能使得 L 又不满足，此时还需要递归 L

如果 1.4.1-3，一个原本满足平衡标准的 SBT，在子树 A 内插入了节点，导致 $size[B] > size[R]$

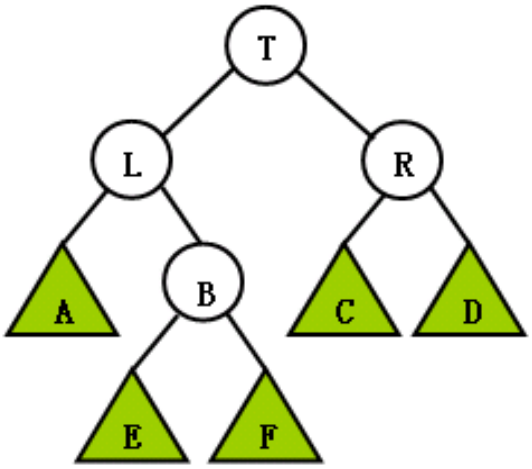


图 1.4.1-3

在这种情况下我们无法通过一两步旋转使得 R 成为 A 的兄弟子树的孩子，此时应左旋节点 L(即 zag(B))，得到图 1.4.1-4，再右旋节点 T(即 zig(B))，得到图 1.4.1-5

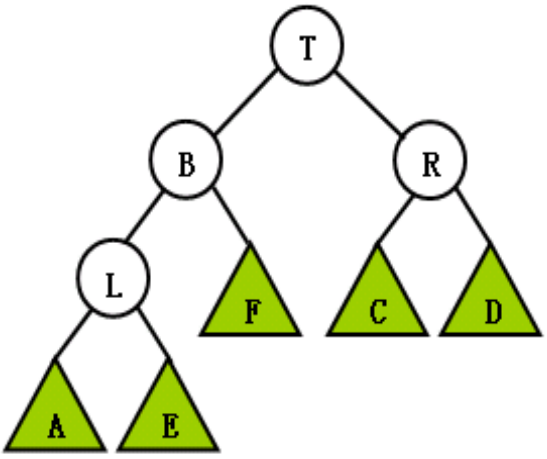


图 1.4.1-3

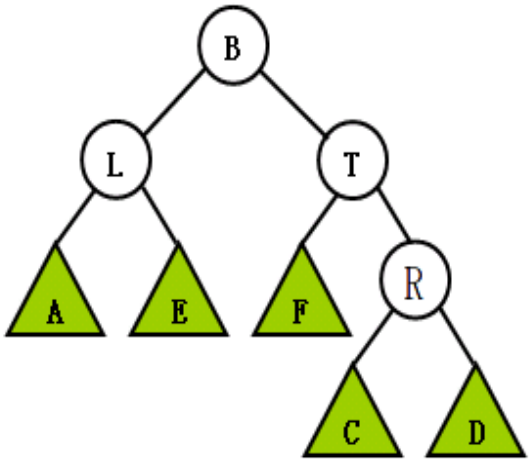


图 1.4.1-3

然后这棵树就有了更多的不可确定性，但是子树 A, E, R, C, D, F 一定是 SBT，于是递归 L, T，最后在递归一次 B 即可。

Maintain 函数看似很多递归，然而可以证明时间复杂度均摊是 $O(1)$ 的，另外，我们可以考虑到插入真正受到影响的子树只有一个，也就是说插入之后上面的两条限制有一条依旧满足，于是我们可以再加一个参数 f，表示是哪边可能不满足情况，这样减少了判断，提高了效率。

代码：

```
//Rotate为旋转函数，这里的旋转与Splay的旋转有一定的区别
//Maintain函数记住一定是传引用
void Maintain(int &r, bool f) {
    if(!f)
        if(size[ch[r][1]] < size[ch[ch[r][0]][0]])
            Rotate(r, 0);
        else if(size[ch[r][1]] < size[ch[ch[r][0]][1]])
            Rotate(ch[r][0], 1), Rotate(r, 0);
        else
            return ;
    else
        if(size[ch[r][0]] < size[ch[ch[r][1]][0]])
            Rotate(ch[r][1], 0), Rotate(r, 1);
        else if(size[ch[r][0]] < size[ch[ch[r][1]][1]])
            Rotate(r, 1);
        else
            return ;
    Maintain(ch[r][0], 0), Maintain(ch[r][1], 1);
    Maintain(r, 0), Maintain(r, 1);
}
```

1.4.2 旋转操作 Rotate$(r, kind)$

SBT 需要旋转函数，但是其旋转不再称为 zig/zag，而在上文中括号内的 zig/zag 只是便于理解。SBT 之所以不再使用，是因为 SBT 不记录父节点，此时若再使用 zig/zag 将找不到父节点

因为 SBT 不记录父节点信息，所以 SBT 的旋转只要考虑孩子节点的信息的变化即可，这样旋转的代码也就更短，更好写

至于旋转也是要传引用的，因为你不知道旋转的节点父亲是谁，但是如果传引用的话，直接一个等于号就相当于修改了旋转节点父亲的孩子信息

代码：

```
//注意Rotate函数是传引用，不然旋转完了之后就是环套树了
void Rotate(int &r, int kind) {
    int x = ch[r][kind];
    ch[r][kind] = ch[x][!kind], ch[x][!kind] = r;
    Update(r), Update(x);
    r = x; //引用的作用
}
```

1.4.3 插入操作 Insert(r, x)

插入与普通的 BST 的插入类似，找到一个空节点插进去就好了，插完之后调用 Maintain 函数维持树高，注意节点要传引用

代码

```
//依旧注意是传引用
void Insert(int &r, int x) {
    if(!r) {r=++now, size[r]=1, data[r]=x; return ;}
    ++size[r];
    if(data[r] >= x) Insert(ch[r][0], x), Maintain(r, 0);
    else Insert(ch[r][1], x), Maintain(r, 1);
}
```

1.4.4 删除操作 Delete(r, x)

SBT 的删除与 BST 的删除有很大区别，为了保持树的形态，所以在删除的时候不能旋转节点。但是这里用了一种巧妙的方法：反正 SBT 上留节点信息只有子树大小和节点权值，子树大小的话删除肯定是减一的，那么剩下的节点权值，既然不能旋转，那么我们可以删除一个其他节点，再把那个节点的权值放到这个本应该被删除的节点上面，这样就有两种选择：要么删掉前驱，要么删掉后继，都不影响到 SBT 的中序遍历。我选择的是删前驱。

再删除节点的时候可以在回退的时候调用 Maintain 函数维护树的形态，也可以不调用，反正删节点树的高度不可能增加

代码

```
//依旧注意是传引用，注意函数有返回值
int Delete(int &r, int x) {
    --size[r];
    if((data[r]<=x&&!ch[r][1]) || (data[r]>=x&&!ch[r][0]))
        return x = data[r], r = ch[r][0]^ch[r][1], x;
    if(data[r] == x) return data[r] = Delete(ch[r][0], x), x;
        //注意依旧需要返回x的值，因为可能在上面还遇到了一个权值为x的节点，此时上一个遇到的节点权值不需要变化
    else return Delete(ch[r][data[r]<x], x);
}
```

1.5 替罪羊树

替罪羊树采用暴力的方法做到了一颗平衡树能做的事，而且还能保证时间复杂度均摊 $(\log N)$

替罪羊树的插入采用的是暴力插入，当时当整棵树过于不平衡的时候就会对子树进行一次重建，但是插入导致过于不平衡的时候不是直接重建当前的子树，而是一路往根的方向上走，找到深度最小且不平衡的节点进行重构，因为一颗子树的不平衡而导致一颗父节点子树被重建，这大概是替罪羊树名字的由来把，由于暴力插入重构都是基础的内容，故这里不分小节

替罪羊树的删除有两种方法：一种是用 BST 的删除方法：找到待删节点，然后把他的前驱或者后继提上来，将前驱或者后继的节点删除；另一种就更加暴力了：直接在节点上打上删除标记，然后如果整棵树上的删除标记过多的时候对整棵树进行一次重构

替罪羊树求排名前驱后继的方法基本上来说和普通 BST 是一样的，但是如果是打删除标记的话，我并不会求

2 BST 的一些模板题

例题 1. 普通平衡树[1]

维护一个数据结构，使其支持一下六个操作：

1($1\ x$): 插入 x

2($2\ x$): 删除 x，如果 x 有多个，只删除一个

3($3\ x$): 询问数 x 的排名: 若有多个相同的数，输出其最小排名

4($4\ x$): 询问排名为 x 的数

5($5\ x$): 询问 x 的前驱，即小于 x 且最大的数

6($6\ x$): 询问 x 的后继，即大于 x 且最小的数

$1 \le M \le 10^5, |x| \le 10^7$，$M$ 为操作数，1s/256MB

模板题，练习平衡树都可以用这个

[1]LibreOJ 104

例题 2. 最长上升子序列[2]

一个序列：初始时为空，一次将 1-N 插入到这个序列中的某个位置，每插入一个数就询问当前序列的最长上升子序列的长度

$N \leq 100000$

BST 预处理出最后得到的序列，因为每次插入有序，所以直接对序列 $O(N \log N)$ 求最长上升子序列，可以得到以当前数结尾的最长上升子序列的长度，最后每次将这个长度和上一个数的长度取最大值即可，时间复杂度:$O(N \log N)$

例题 3. 文艺平衡树[3]

维护一个数据结构结构，使其支持区间翻转

$1 \leq N, M \leq 10^5$，N 为序列长度，M 为操作数量，1s/256MB

练习 Splay 的区间翻转

例题 4. GameZ 游戏排名系统[4]

维护一个游戏排名系统，使其支持一下几个操作：

1($+s$ x): 其中 s 是玩家名字，x 是玩家上传的分数，如果在之前 s 上传过分数，则将其改成 x，否则新建一个玩家名字为 s，分数为 x 的游戏信息

2($?s$): 其中 s 是玩家名字，输出其在游戏排名系统中的排名

3($?x$): 其中 s 是一个数字，输出排名为在区间 $[x, \min(T, x+9)]$ 的玩家，其中 T 是当前系统中玩家信息的数量

定义排名的比较方式：用 (S_x, S_y) 表示名称 S 的玩家，最近的一次上传或更新自己的分数的时间为 S_x，分数为 S_y，我们称 A 的排名比 B 的排名靠前，当且仅当 $A_y > B_y$ 或者 $A_y = B_y, A_x < B_x$，由于任意两个玩家上传分数的时间一定不同，所以不会出现两个排名相同的玩家

$N \leq 250000$，1s/128MB

操作三对 Splay 很支持，不过其他 BST 也能写，对于名称可以 map 或哈希

例题 5. 书架[5]

书架上有 N 本书，每本书都有一个编号 (这 N 本书编号互不相同，且刚好构成 1-N 的排列)，这 N 本书从上到下编号依次为 A_i，现在有五种操作：

1(Top x): 将编号为 x 的书移动到最上面

2(Bottom x): 将编号为 x 的书移动到最下面

3(Insert x y): 如果 $y = -1$，将编号为 x 的书与其下方的一本书交换位置，如果 $y = 0$ 则忽略这个操作，否则将编号为 x 的数与其上方的一本书交换位置

4(Ask x): 询问编号为 x 的书的上方有多少本书

5(Query x): 询问从上往下数第 x 本书的编号

$N, M \leq 80000$，1s/64MB

[2]TJOI 2013
[3]LibreOJ 105
[4]ZJOI 2006
[5]ZJOI 2006

用 Splay，直接把节点编号设为其书的编号。但是排名不好求，然而我们可以直接暴力把要求排名的节点旋到根，那么排名就是左子树节点数加一了，剩下的就是模板了

例题 6. 文本编辑器[6]

要求你模拟一个文本编辑器，使其支持以下操作：

1(Move k): 将光标移动到第 k 个字符之后，如果 $k=0$，则将光标移动到第一个字符之前

2(Insert n s): 在光标后插入一个长度为 n 的字符串 s，$n \geq 1$，光标位置不变，注意字符串 s 在输入的下一行

3(Delete n): 删除光标后的 n 个字符，光标位置不变，$n \geq 1$

4(Rotate n): 反转光标后的 n 个字符，光标位置不变

5(Get): 输出光标后的一个字符，光标位置不变

6(Prev): 光标前移一个字符

7(Next): 光标后移一个字符

文本的字符都是可见字符，其 ASCII 码的区间为 $[32, 126]$

Move 操作不超过 50000 个，Insert, Delete, Rotate 操作总数不操作 6000，Get 操作不超过 20000，Prev 和 Next 总数不超过 20000，插入的字符总量不超过 2^{20} 个，输入数据合法

如果使用 Splay 的话，可以把光标的前一个字符设为 Splay 的根。其他 BST 也能做

例题 7. 郁闷的出纳员[7]

维护一个数据结构，使其支持以下操作：

1(I k): 如果 $k < m$，则忽略这个操作，否则插入 k

2(A k): 将数据结构内所有数加上 k

3(S k): 将数据结构内所有数减去 k

4(F k): 查询第 k 大的数，如果不存在则输出-1

每个操作结束后，删除其中所有小于 m 的数，并且在所有操作结束后，输出总共删除的数

I 操作总数不超过 100000，A 操作和 S 操作总数不超过 100，F 操作总数不超过 100000，1s/64MB

模板，可以使用区间加减，也可以记录加减的数量，调整 m，总之看个人的喜好

例题 8. Super Memo[8]

各出一个长度为 N 的序列 A，使其支持以下几个操作

1(ADD l r x): 将区间 $[l, r]$ 的数加上 x

2(REVERSE l r): 翻转区间 $[l, r]$ 的所有数

3(REVOLVE l r t): 将区间 $[l, r]$ 的数移动 t 次，例如序列 $\{1, 2, 3, 4, 5, 6\}$，执行 REVOLVE 2 5 1 得到 $\{1, 5, 2, 3, 4, 6\}$，再执行一次 REVOLVE 2 5 1 得到 $\{1, 4, 5, 2, 3, 6\}$

4(INSERT p x): 在序列其 p 个数后插入一个数 x

5(DELETE p): 在序列中删除第 p 个数

6(MIN l r): 询问区间 $[l, r]$ 中的最小值

$N, M \leq 100000$，M 为操作数量，2s/65536KB

[6]AHOI 2006

[7]NOI 2004

[8]POJ 3580

第三个 REVERSE l r t，令 $t = t \mod (r-l+1)$，对于 $t \neq 0$ 的情况有两种理解：

一种是先删除区间 $[1+r-t, r]$ 的数，再将其插入到第 l 数的前面

另一种是先后旋转 $[l, r], [l, l+t-1], [l+t, r]$

例题 9. 项链工厂[9]

给出一个长度为 N 的环形序列 A，M 次操作，每个操作为以下六种：

1(R k): 将这个环顺时针旋转 k 个单位 (对于 $i \in [k+1, n]$, 令 $A_i = A_{i-k}$, 对于 $i \in [1, k]$, 令 $A_i = A_{i+n-k}$)

2(F): 将这个环沿 A_1 所在的直径对称 (对于 $2 \leq i \leq \lfloor \frac{n+1}{2} \rfloor$，交换 A_i 与 A_{n-i+2})

3(S i j): 交换 A_i 与 A_j

4(P l r x): 环上区间赋值 (如果 $l \leq r$，则对于 $i \in [l, r]$，令 $A_i = x$，否则对于 $i \in [l, n] \cup [1, r]$，令 $A_i = x$)

5(C): 询问环上的段数，我们认为 A_n 与 A_1 相邻

6(CS l r): 询问环上区间的段数 (如果 $l \leq r$，询问 A_l 到 A_r 有多少段不同的数，否则询问对于 A_l 到 A_n 并 A_1 与 A_r 有多少段不同的数，特别的，我们认为 A_n 与 A_1 相邻)

$N, M \leq 500000, A_i \leq 1000$，4s/512MB

考虑到旋转对称不会影响到两个数的相对位置，所以可以用线段树做

但是也可以用 Splay 来模拟

例题 10. 维修数列[10]

1(INSERT p t A_1 A_2 .. A_t): 在当前数列的第 p 个数字后插入 t 个数字:$A_1, A_2..A_3$，如果插入在数列首则 $p = 0$

2(DELETE p t): 将当前数列中第 p 个数字开始的 t 个数字删除

3(MAKE − SAME p t c): 将当前数列中第 p 个数字开始的 t 个数字设为 c

4(REVERSE p t): 将当前数列中第 p 个数字开始的 t 个数字翻转

5(GET − SUM p r): 询问当前数列中由第 p 个数字开始的 t 个数字构成的子序列的和

6(MAX − SUM): 询问当前数列的最大字段和

$M \leq 20000$，在任何时刻数列中最多有 500000 个数，且每个数字都在 $[-1000, 1000]$ 内，插入的数字不会超过 4000000 个，1s/64MB

大模板题，区间最大字段和可以像线段树维护的那样，注意第六个操作如果整个序列中所有值都是负数，那么输出其中最大的一个负数

3 BST 的应用

3.1 启发式合并

启发式合并其实是暴力合并的美称，主要过程就是在一颗 BST 中把所有节点取出来，在一个一个地插入到另一颗 BST 中，这就完成了合并的过程

无旋转 Treap 的合并由于合并的两颗 Treap 的节点大小有严格的要求 (一颗 Treap 中所有节点的最大值小于另一颗 Treap 中所有节点的最小值)，其时间复杂度是 $O(N \log N)$ 的，Splay 的合并由于合并时按照从小

[9]NOI 2007

[10]NOI 2005

到大的顺序插入时的 Splay 操作所以其时间复杂度也为 $O(N\log N)$，其他 BST 的启发式合并时间复杂度都是 $O(N\log^2 N)$

例题 11. 梦幻布丁[11]

给出一个长度为 N 的序列 A，M 个操作，每个操作为以下两种：

1(1 x y): 将序列 A 中的所有 x 用 y 替代

2(2): 询问 A 中有多少段数，例如 $A=[1,2,2,1]$ 中有三段，分别是 $A[1]=1, A[2..3]=2, A[4]=1$

$1\le N,M\le 100000, 1\le x,y<1000000$，1s/32MB

给每一种颜色开一个 BST，数组记录数字，修改的时候判断左边的数字是否等于 y，如果相等则答案减一，然后将两颗 BST 合并

时间复杂度：用 Splay 启发式合并为 $O(N\log N)$，否则时间复杂度:$O(N\log^2 N)$

3.2 维护凸包

例题 12. 防线修建[12]

$M+3$ 个点，除了 $(0,0),(n,n)$ 外，每个点的横纵坐标都是区间 $(0,n)$ 内的整数，你需要在某些点之间连边，每条边的花费为这条边的总长度 ($(0,0)$ 与 (n,n) 两点默认有边，所以这两个点的连边不需要花费)，当然，你连的这些边必须围城一个封闭的图形，而且需要使得所有点都在这个封闭图形内 (可以在边界上)，q 个操作，每个操作为以下两种：

1(1 x): 删除第 x 个点 (这个点可以不在图形内或边界上)

2: 询问图形的总长度的最小值

$n\le 10000, M\le 100000, q\le 200000$，1s/512MB

显然最后连出来的一定是一个凸包，所以这个题目就是动态维护凸包。

删除操作不好做，我们可以离线将其转化为添加操作，可以用 BST 来维护，我用的是 SBT

具体维护过程，凸包上所有的边 (除了 $(0,0)$ 与 (n,n) 连成的边) 的斜率一定是单调递减的，所以当插入一个点 a 时，在 BST 内找到这个点的前驱 b 及后继 c(x 坐标的)，判断 b 或 c 是否与 a 斜率不存在，如果不存在，如果 a 在 b 或 c 的下面，就不用把 a 加入凸包了，否则把 b 或 c 删掉，更新距离，重新求前驱后继。然后如果 $k_{a,b}<k_{b,c}$ 的话就说明 a 在凸包内部，不用加入，否则 a 是一定会被加入凸包的，根据斜率递减的原则不断删除不满足要求的前驱后继就好了，注意更新距离。

时间复杂度:$O(M\log M)$

例题 13. 货币兑换[13]

两种券：两种纪念券 (A,B) 和钱，一开始你有 M 块钱，没有纪念券，一共 N 天，你每天有两种事情可以做 (可以都做也可以都不做，如果都做不限制顺序)：

1. 卖出纪念券：按照 $[0,100]$ 内的一个实数 p 作为卖出比例，也就是将 $p\%$ 的 A 券和 $p\%$ 的 B 券以这一天的价值兑换成钱

2. 买入纪念券：将 x 元钱买入纪念券，你将会得到总价值为 x 的纪念券，且 A 券与 B 券的比例为 r

[11]HNOI 2009

[12]HAOI 2011

[13]NOI 2007

给出 N, M，以及这 N 天内 A, B 纪念券的价值及 r_i(兑换比例)，求 N 天后最多可以有多少块钱

$N \leq 100000, 0 < a_i, b_i \leq 10, 0 < r_i \leq 100$，1s/512MB

提示：存在一种最优方案满足：每次买入操作使用完所有的钱，每次卖出操作卖出所有纪念券

设 $f(i)$ 表示第 i 天时的钱数，显然 $f(0) = M$，那么我们可以在第 j 天时把钱换成纪念券，再在第 i 天卖出，故我们可以写出转移方程式：

$$f(i) = \max\{f(i-1), a_i \times \frac{r_j f(j)}{r_j a_j + b_j} + b_i \times \frac{f(j)}{r_j a_j + b_j}\}$$

其中分式部分是由买入纪念券的方式解方程得到的，我们可以令

$$x_j = \frac{r_j f(j)}{r_j a_j + b_j}$$

$$y_j = \frac{f(j)}{r_j a_j + b_j}$$

那么方程式变成 (先不考虑 $f(i-1)$ 的部分):

$$f(i) = \max\{a_i \times x_j + b_i \times y_j\}$$

移项，可以得到

$$y_j = -\frac{a_i}{b_i} \times x_j + \frac{f(i)}{b_i}$$

这是一个一次函数的形式，其斜率为 $-\frac{a_i}{b_i} < 0$，而我们需要最大化 $f(i)$，我们可以把 (x_j, y_j) 看做是平面上的一个点，那么我们需要维护一个上凸包，因为 x_j 不单调，所以我们需要使用数据结构来维护，BST 都可以，我用的是 SBT，注意精度问题。时间复杂度 $O(N \log N)$

4 BST 与线段树

线段树与 BST 在一起使用的时候可以粗暴的分类为树套树 (线段树套 BST) 和非树套树两种

4.1 非树套树

非树套树主要运用的是线段树的 $O(N \log N)$ 的合并 (虽然 Splay 的启发式合并也可以做到 $O(N \log N)$ 但是常数很大)，线段树的查询方便的特点，而 BST 则是运用了其可以插入，删除。

例题 14. 没有人的算术[14]

定义集合 $S = \{0\} \cup \{(a, b)\}$

定义等于运算:$0 = 0$；当且仅当 $a = b$ 且 $c = d$ 时 $(a, b) = (c, d)$

定义小于运算:$0 < (a, b)$；当且仅当 $a < c$ 或者 $a = c$ 且 $b < d$ 时 $(a, b) < (c, d)$

同理可以定义所有运算

给出一个长度为 N 的数组 A，其中 $A_i \in S$，初始时 $A_i = 0$，M 个操作，每个操作为以下两种：

1(C x y z):$A_z \leftarrow (A_x, A_y)$

2(Q l r): 设 $A_p = \max_{i=l}^{r} A_i$，求 $\min\{p\}$，即最大值的下标的最小值

$N \leq 100000, M \leq 500000$，2s/128MB

[14]BZOJ 3600

首先对于询问显然可以用线段树来解决，那么最大的问题就是比较两个数的大小，而且由于 M 很大，所以只能用 $O(M\log N)$ 的算法做

我们可以考虑映射，将 x 映射到 $f(x)$，使得 $x<y\iff f(x)<f(y)$，如果我们能够得到这个映射的话，那么这个题目就迎刃而解了

我们可以得到两个数小于的条件

$$(a,b)<(c,d)\iff f(a)<f(c)\text{ or }(f(a)=f(c)\text{ and }f(b)<f(d))$$

式子后半部分与我们日常比较二元组的大小是一致的，而通过修改命令我们知道：当插入 (a,b) 时，a,b 一定在之前出现过，也就是说我们已经知道了 $f(a)$ 和 $f(b)$

于是我们就可以想到一个 BST 的算法：把映射关系放到 BST 上，BST 上放一个值，满足其左子树的值一定小于右子树的值，注意这个值不能是这个数在 BST 的排名，因为排名时刻在变化而且我们无法 $O(1)$ 求出，这样子，在线段树上的更新就不能在 $O(\log N)$ 内更新了。那么我们考虑线段树一样，BST 上每一个节点表示一个区间，那么这个节点的值直接用区间的中点表示，这样插入的话，也不会影响区间，只是新增了一个区间，那么这个区间我们就可以用一个数组来存下来，也就是说这样就可以 $O(1)$ 求出了

但是，BST 不能旋转，因为一转那么区间就乱套了，解决的方法是不旋转，用替罪羊树，替罪羊树没有旋转，只有暴力重建子树

所以，问题解决了，时间复杂度 $O(N\log N)$

例题 15. Philosopher[15]

给出一个长度为长度为 N 的序列 A，使其支持以下操作：

1($1\ l\ r\ f$): 将区间 $[l,r]$ 内的元素按照升序 ($f=0$) 或者降序 ($f=1$) 排序

2($2\ l\ r$): 询问区间 $[l,r]$ 内的元素的积的十进制的最高位

$N,M\le 200000, 1\le A_i\le N$，2s/512MB

区间排序虽然不能直观地进行，但是我们可以想到权值线段树的合并，合并得到的新线段树的节点在权值线段树内是有序的，而升序或者降序可以直接用一个元素记录，第二个操作区间积的话可以通过取对数的方法将其改为求区间和。

但是考虑到线段树合并之后会形成许多颗线段树，因为每一颗线段树内节点大小有序，节点的编号也有序，所以我们可以考虑用 BST 来维护这些线段树的根节点。当区间排序时，先在 BST 内找到对应的节点，注意两端的节点所表示的线段树为了把区间提取出来需要用到权值线段树的分裂，然后将这些线段树合并，在 BST 中删去对应的节点，最后将合并得到的线段树根节点再存回到 BST 中。查询也是一样的

时间复杂度:$O(N\log N)$

4.2 树套树

4.2.1 线段树套 BST

线段树的形态不会改变，但是其空间花费较大，而 BST 空间小可以插入删除修改，但是 BST 的形态会发生变化，利用这两个特性可以将这两种数据结构结合起来，外层用线段树，内层用 BST，内层的 BST 可以看做是线段树的一种标记

[15]LibreOJ 6189

例题 16. 二逼平衡树[16]

给出一个长度为 N 的序列 A，使其支持一下 5 个操作

1($1\ l\ r\ x$): 询问 x 在区间 $[l,r]$ 内的排名

2($2\ l\ r\ k$): 询问区间 $[l,r]$ 内排名为 k 的数

3($3\ p\ x$): 将 A_p 修改为 x

4($4\ l\ r\ x$): 询问 x 在区间 $[l,r]$ 内的前驱

5($5\ l\ r\ x$): 询问 x 在区间 $[l,r]$ 内的后继

$N, M \leq 50000, |A_i|, |x| \leq 10^8$，4s/512MB

线段树套 BST，至于是套哪种由自己选择

例题 17. 三维偏序[17]

N 个元素，每个元素都有 A_i, B_i, C_i 三个属性，令

$$f(i) = \sum_{j=1}^{N} [i \neq j][A_i \geq A_j][B_i \geq B_j][C_i \geq C_j]$$

对于 $d \in [0, n-1]$，求 $f(i) = d$ 的 i 的数量

$1 \leq N \leq 100000, 1 \leq M \leq 200000$，2s/256MB

一维排序，二三维树套树，时间复杂度:$O(N \log^2 N)$

4.2.2 BST 套线段树

BST 做外层数据结构，线段树做内层数据结构，作为 BST 的标记。此时需要要求 BST 不能旋转，否则根本无法更新线段树上的信息，所以这里的 BST 一般选择替罪羊树，当替罪羊树不平衡时，就暴力重构替罪羊树，顺便重构内层线段树，这个时候要注意垃圾的回收，不然空间会炸

例题 18. 带插入区间 K 小值[18]

1(Q $l\ r\ k$): 询问区间 $[l,r]$ 的第 k 小

2(M $p\ x$): 将第 p 个数修改为 x

3(I $p\ x$): 在第 p 个数之前插入一个数 x，如果 p 等于当前数列的长度加一，则将其插入到序列尾

$N \leq 35000$, 第一个操作数量不超过 70000, 第二个操作数量不超过 70000, 第三个操作数量不超过 35000, 15s/512MB

替罪羊树套线段树，模板题，注意细节，询问操作可以把线段树对应的根节点从替罪羊树上拿下来，然后二分。修改操作可以在替罪羊树上存对应节点的在序列中的值，这样的话删除就可以先从上往下的时候把 x 插入到线段树里面，把原序列中的第 p 个数作为返回值，在回溯的时候在线段树中把对应节点删除。

时间复杂度:$O(N \log^2 N)$

[16]LibreOJ 106

[17]LibreOJ 112

[18]BZOJ 3065