可持久化线段树

1：定义

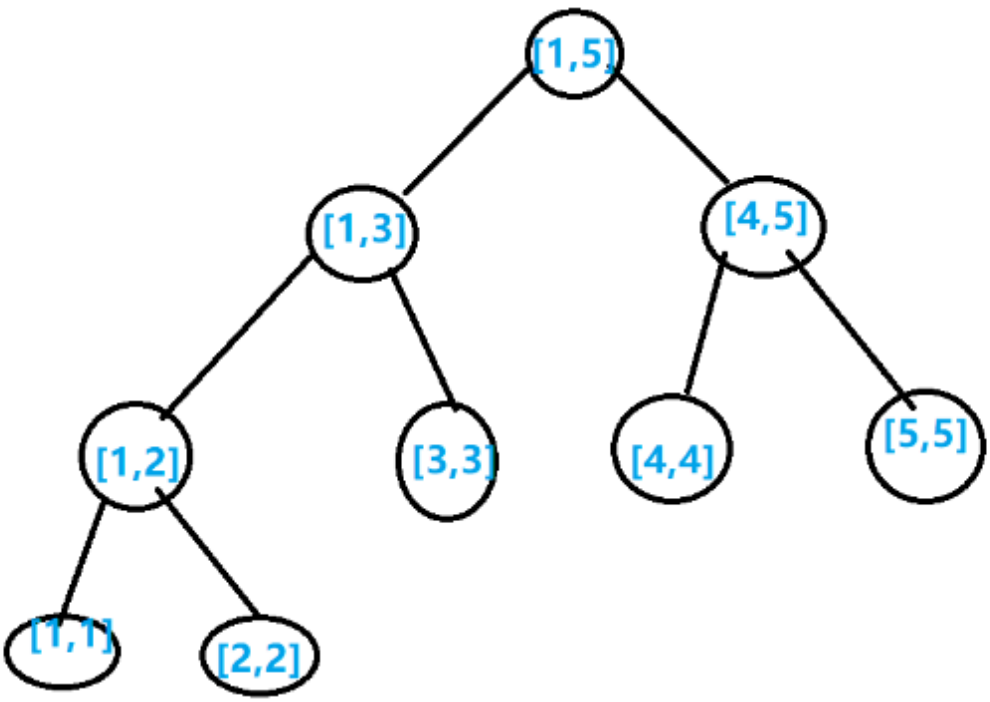
可持久化线段树也称为主席树，函数式线段树。其实主席树就是很多线段树组合的总体，从它的其它称呼也可以看出来了，其实它本质上还是线段树。主席树就是利用函数式编程的思想来使线段树支持询问历史版本、同时充分利用它们之间的共同数据来减少时间和空间消耗的增强版的线段树。

2：引题

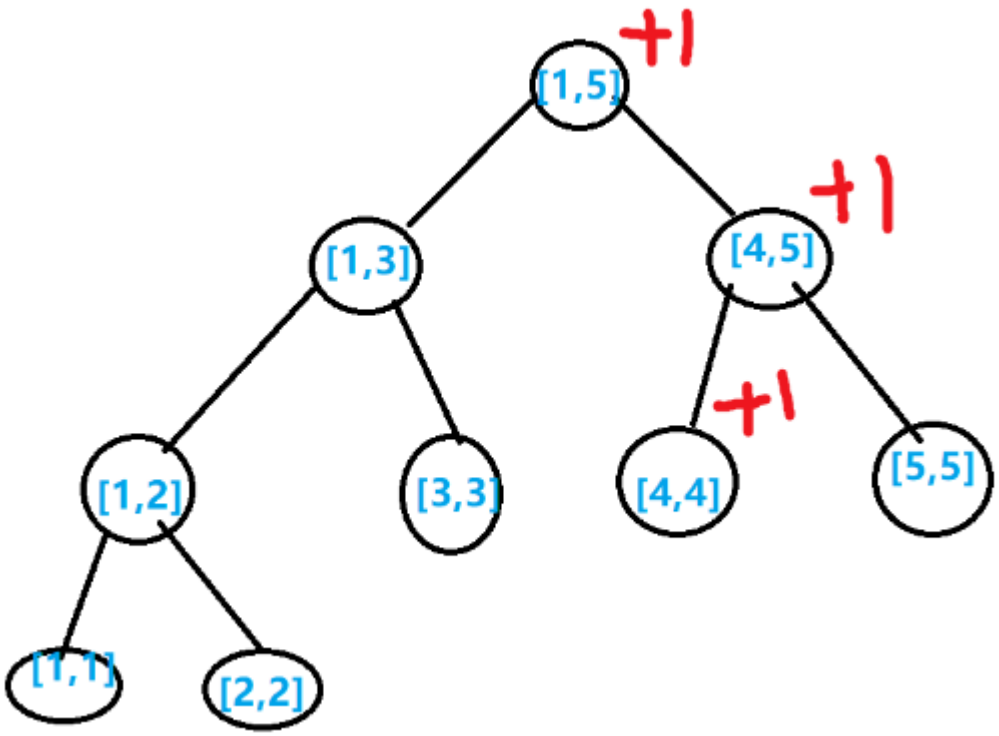
求一个每次添加元素至末尾的序列的中位数的大小的题目，就相当于给你最后的数组,让你还原每一次的过程中的中位数并依次输出.

```
input:
5
4 2 7 3 1
output:
4 (4 序列的中位数)
3 (4 2 的排序后中位数)
4 (4 2 7 排序后的中位数,以此类推)
3.5
3
```

其中有一个做法就是建立一颗 线段树 每个节点代表一个区间[i , j] 意味着离散化后的数字大小为 i~ j 的数字出现了多少次 在我们的样例之中,离散化之后还是只有 5 个数字,所以是建立siz = 5 的树 那么很容易推出这棵树的模样:



那么我们发现当 元素 a[1]=4(离散化后为4) 加入到这颗树后对每一个节点的影响为以下:



那么我们其实就是知道是怎么加入元素了 但是还有一个问题,就是求 kth怎么办?

3：解决

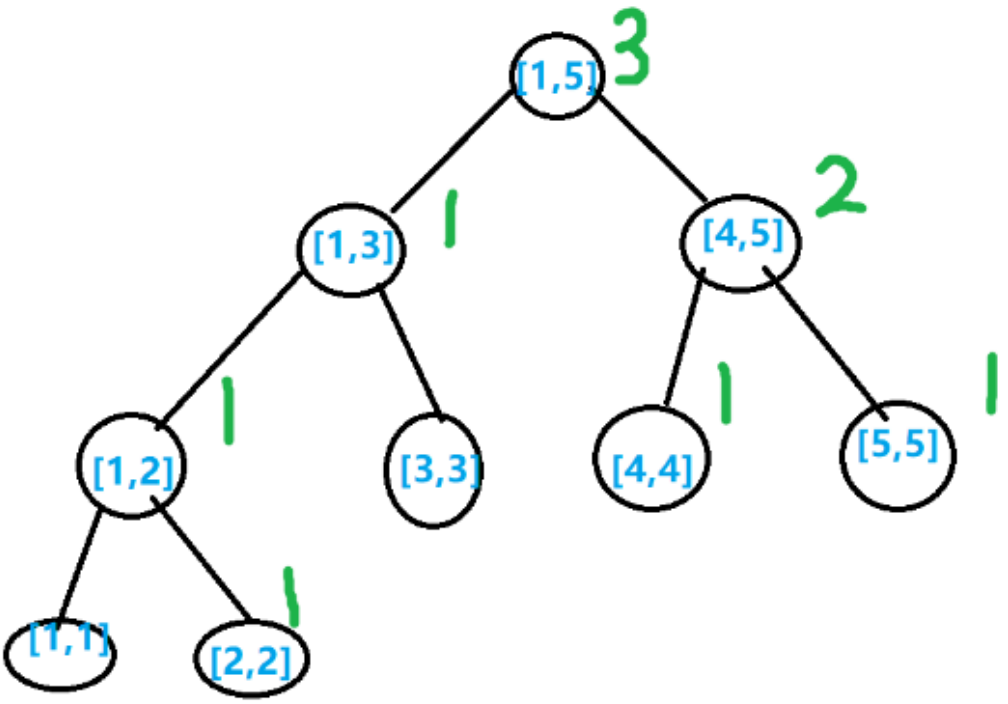
我们考虑最暴力的思想: 就是建立 n 棵像上面说的那样的树, 第 i 棵树Ti维护的是 区间 [1,i]的序列信息 可以发现,这种树是不是有区间相减的性质?

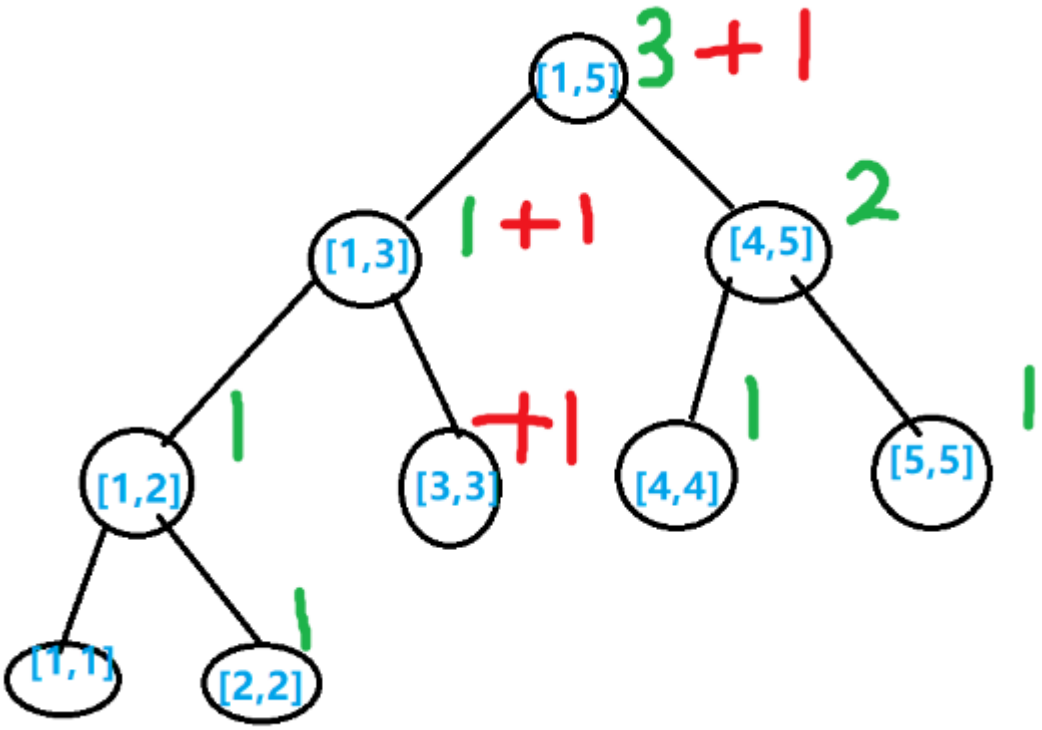
举个栗子: [3,4] 区间的 离散化大小为 [1,3]的数有多少个? 不就是将T[4] 的 表示 [1,3]区间的节点的权值减去T[2]的节点表示[1,3] 区间的节点的权值就是 [3,4]区间的离散大小为[1,3]的数的个数了吗(前缀思想)

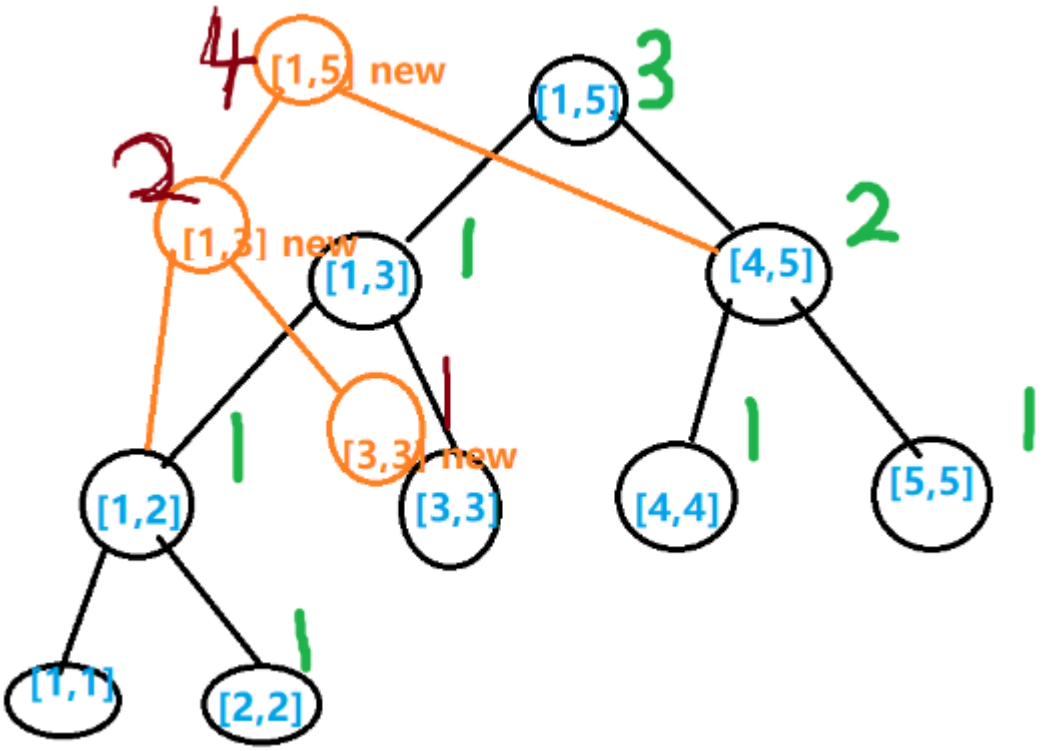
但是仔细思考，发现这样无论是时间复杂度还是内存似乎都不太行，能不能优化一下?

我们只要观察一下,是不是每一次都是只会有 一条 O(logn)的路径是被修改的,而别的信息是不是不变的,那么我们就可以考虑每一次只是建立 log 个节点,让这些树共用一些部分 我们翻一下样例:

比如就是一开始我们说的样例好了,假设我们建完了T[3] 这棵树,我们加入第 4 个元素:







如上图每次加入一个原序列的一个点，在线段树中只需要加log(N)个点，并不需要建立一棵新的线段树，无论是时间还是空间我们都能接受了。

4：具体题目分析

POJ：2104

题目大意：给一串数字，多次询问区间的第k小值

分析：

跟一般的在整棵树中找第k个数是一样的。如果一个节点的左权值（左子树上点的数量之和）大于k，那么就到左子树查找，否则到右子树查找。其实主席树是一样的。对于任意两棵树（分别存区间[1,i]和区间[1,j], i<j），在同一节点上（两节点所表示的区间相同），data域之差表示的是，原序列区间[i,j]在当前节点所表示的区间里，出现多少次（有多少数的大小是在这个区间里的）。同理，对于同一节点，如果在两棵树中，它们的左权值之差大于等于k，那么要求的数就在左孩子，否则在右孩子。当定位到叶子节点时，就可以输出了。

5：主席树通俗理解

所谓主席树呢，就是对原来的数列[1..n]的每一个前缀[1..i]（1≤i≤n）建立一棵线段树，线段树的每一个节点存某个前缀[1..i]中属于区间[L..R]的数一共有多少个（比如根节点是[1..n]，一共i个数，sum[root] = i；根节点的左儿子是[1..(L+R)/2]，若不大于(L+R)/2的数有x个，那么sum[root.left] = x）。若要查找[i..j]中第k大数时，设某结点x，那么x.sum[j] - x.sum[i - 1]就是[i..j]中在结点x内的数字总数。而对每一个前缀都建一棵树，会MLE，观察到每个[1..i]和[1..i-1]只有一条路是不一样的，那么其他的结点只要用回前一棵树的结点即可，时空复杂度为O(nlogn)。

6：具体代码实现

```
#include <cstdio>
#include <algorithm>
```

```cpp
using namespace std;
const int MAXN = 100010;

struct Node
{
    int L, R, sum;
};
Node T[MAXN * 20];
int T_cnt;

struct A
{
    int x, idx;
    bool operator < (const A &rhs) const
    {
        return x < rhs.x;
    }
};

A a[MAXN];
int rank[MAXN], root[MAXN];
int n, m;

void insert(int &num, int &x, int L, int R) //rank[i] root[i] 1 n
{
    T[T_cnt++] = T[x]; //加一棵树
    x = T_cnt - 1;
    ++T[x].sum;     //对应节点所记录的数字个数+1
    if(L == R)      //到了叶子节点返回
        return ;
    int mid = (L + R) >> 1;     //否则递归建树
    if(num <= mid)
        insert(num, T[x].L, L, mid);
    else
        insert(num, T[x].R, mid + 1, R);
}

int query(int i, int j, int k, int L, int R)  //query(root[i - 1], root[j], k, 1, n)
{
    if(L == R)
        return L;
    int t = T[T[j].L].sum - T[T[i].L].sum;
    int mid = (R + L) >> 1;
    if(k <= t)
        return query(T[i].L, T[j].L, k, L, mid);
    else
        return query(T[i].R, T[j].R, k - t, mid + 1, R);
}

int main()
{
    T[0].L = T[0].R = T[0].sum = 0;

    root[0] = 0;
```

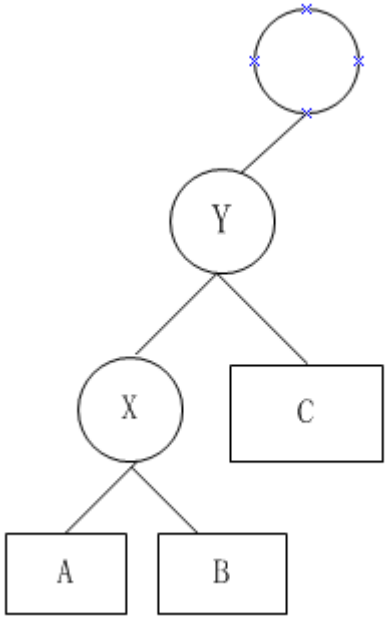
```
    while(scanf("%d%d", &n, &m) != EOF)
    {
        for(int i = 1; i <= n; ++i)
        {
            scanf("%d", &a[i].x);
            a[i].idx = i;
        }
        sort(a + 1, a + n + 1);
        for(int i = 1; i <= n; ++i)
            rank[a[i].idx] = i;
        T_cnt = 1;
        for(int i = 1; i <= n; ++i)
        {
            root[i] = root[i - 1];
            insert(rank[i], root[i], 1, n);
        }
        while(m--)
        {
            int i, j, k;
            scanf("%d%d%d", &i, &j, &k);
            printf("%d\n", a[query(root[i - 1], root[j], k, 1, n)].x);
        }
    }
    return 0;
}
```

Treap

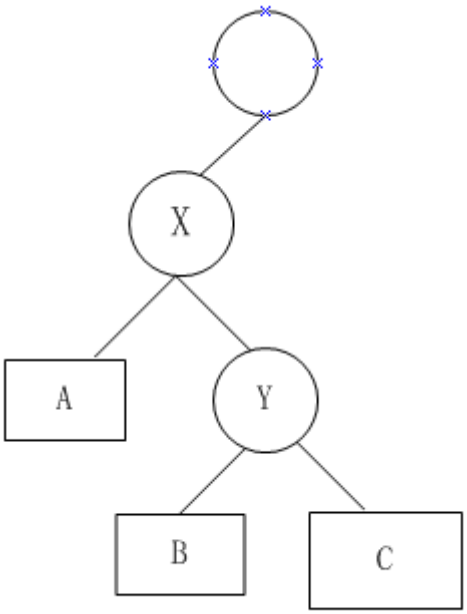
1:为什么要二叉平衡树?

满足BST性质且中序遍历为相同序列的二叉搜索树是不唯一的。这些二叉搜索树是等价的，维护的是相同的一组数值。在这些二叉搜索树上执行相同的操作将得到同样的结果。因此我们可以在维持BST性质的基础上，通过改变二叉搜索树的形态使得树上每个节点的左右子树大小达到平衡，从而使得整棵树的深度维持在O(logN)级别。

改变形态并保持BST性质的方法就是旋转。最基本的旋转操作称为单旋转，它又分为左旋和右旋。



右旋之后：



以右旋为例，在初始情况下，x是y的左子节点，A和B分别是x的左右子树，C是y的右子树。

右旋操作在保持BST性质的基础上，把x变为y的父节点。因为x的关键码小于y的关键码，所以y应该作为x的右子节点。

当x变成y的父节点后，y的左子树就空出来了，于是x原来的右子树B就恰好作为y的左子树。

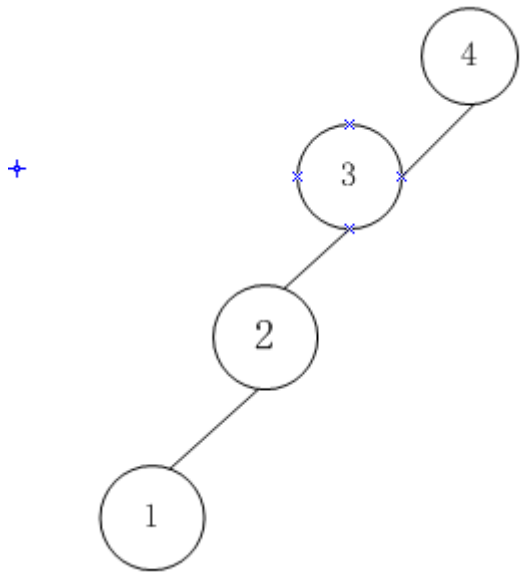
右旋代码如下：

```
void zig(int &p) {
    int q = a[p].l;
    a[p].l = a[q].r, a[q].r = p, p = q;
    Update(a[p].r), Update(p);
}
```

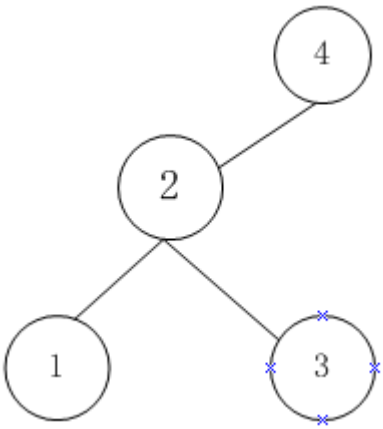
左旋代码如下：

```
void zag(int &p) {
    int q = a[p].r;
    a[p].r = a[q].l, a[q].l = p, p = q;
    Update(a[p].l), Update(p);
}
```

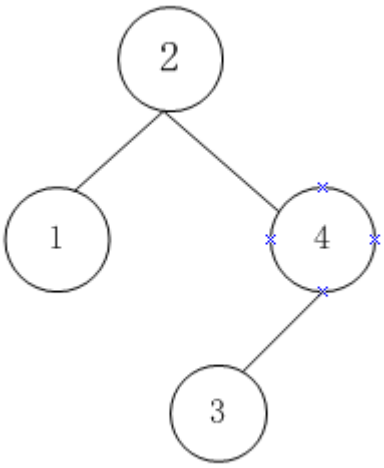
合理的旋转操作可以使得BST变得更平衡。如下图所示，对形态为一条链的BST进行一系列单旋转操作后，这棵BST就变得比较平衡。



右旋3：



右旋4：



现在的问题是怎样才算合理的旋转操作？我们发现在随机数据下普通的BST就是趋近平衡的。Treap的思想就是利用随机来创造平衡条件。因为在旋转过程中必须维持BST性质，所以Treap就把随机作用在堆性质上。

Treap在插入每个新的节点时，在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，自底向上一次检查，当某个节点不满足大根堆的性质时就执行单旋转，使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，我们可以直接找到需要删除的节点，并把它向下旋转成叶子结点，最后直接删除。这样就避免了普通的BST的删除方法导致的节点信息更新，堆性质维护操作的问题。

总而言之，Treap通过恰当的单旋转，在维持节点关键码满足BST性质的同时，还使得节点上随机生成的额外权值满足大根堆的性质。Treap是一种平衡二叉树，检索，插入，求前驱后继以及删除操作的时间复杂度都是O(logN)。

2：例题

需要一种数据结构维护一组数据，需要提供以下操作：

1.插入数值x。

2.删除数值x。(若有多个相同的数，只删除一个)。

3.查询数值x的排名。(若有多个相同的数，应输出最小的排名)。

4.查询排名为x的数值。

5.求数值x的前驱。

6.求数值x的后继。

分析：

根据题意，数据中可能有相同的数值，我们可以给每个节点增加一个域cnt用来记录该节点的“副本数”，初始为1.若插入已经存在的数值，就直接把副本数+1，在删除时，减少节点的副本数，当副本数为0时删除该节点即可。

针对查询排名，可以给每个节点增加一个域size，记录以该节点为根的子树中所有节点的副本数之和。当不存在重复数值时，size其实就是子树大小。我们在插入或者删除时需要从下往上更新size信息。另外在发生旋转操作时，也需要修改size。最后在BST检索的基础上，通过判断左右子树size的大小，选择适当的一侧递归就可以查询排名。

```
#include<iostream>

#include<cstdio>
```

```cpp
#include<cstring>
#include<algorithm>
using namespace std;
const int SIZE = 100010;
struct Treap {
    int l, r; // 左右子节点在数组中的下标
    int val, dat; //节点关键码，权值
    int cnt, size; // 副本数，子树大小
} a[SIZE];
int tot, root, n, INF = 0x7fffffff;

int New(int val) {
    a[++tot].val = val;
    a[tot].dat = rand();
    a[tot].cnt = a[tot].size = 1;
    return tot;
}

void Update(int p) {
    a[p].size = a[a[p].l].size + a[a[p].r].size + a[p].cnt;
}

void Build() {
    New(-INF), New(INF);
    root = 1, a[1].r = 2;
    Update(root);
}

int GetRankByVal(int p, int val) {
    if (p == 0) return 0;
    if (val == a[p].val) return a[a[p].l].size + 1;
    if (val < a[p].val) return GetRankByVal(a[p].l, val);
    return GetRankByVal(a[p].r, val) + a[a[p].l].size + a[p].cnt;
}

int GetValByRank(int p, int rank) {
    if (p == 0) return INF;
    if (a[a[p].l].size >= rank) return GetValByRank(a[p].l, rank);
    if (a[a[p].l].size + a[p].cnt >= rank) return a[p].val;
    return GetValByRank(a[p].r, rank - a[a[p].l].size - a[p].cnt);
}

void zig(int &p) {
    int q = a[p].l;
    a[p].l = a[q].r, a[q].r = p, p = q;
    Update(a[p].r), Update(p);
}

void zag(int &p) {
    int q = a[p].r;
    a[p].r = a[q].l, a[q].l = p, p = q;
    Update(a[p].l), Update(p);

}
```

```
void Insert(int &p, int val) {
    if (p == 0) {
        p = New(val);
        return;
    }
    if (val == a[p].val) {
        a[p].cnt++, Update(p);
        return;
    }
    if (val < a[p].val) {
        Insert(a[p].l, val);
        if (a[p].dat < a[a[p].l].dat) zig(p); // 不满足堆性质，右旋
    }
    else {
        Insert(a[p].r, val);
        if (a[p].dat < a[a[p].r].dat) zag(p); // 不满足堆性质，左旋
    }
    Update(p);
}

int GetPre(int val) {
    int ans = 1; // a[1].val==-INF
    int p = root;
    while (p) {
        if (val == a[p].val) {
            if (a[p].l > 0) {
                p = a[p].l;
                while (a[p].r > 0) p = a[p].r; // 左子树一直往右走
                ans = p;
            }
            break;
        }
        if (a[p].val < val && a[p].val > a[ans].val) ans = p;
        p = val < a[p].val ? a[p].l : a[p].r;
    }
    return a[ans].val;
}

int GetNext(int val) {
    int ans = 2; // a[2].val==INF
    int p = root;
    while (p) {
        if (val == a[p].val) {
            if (a[p].r > 0) {
                p = a[p].r;
                while (a[p].l > 0) p = a[p].l; // 右子树一直往左走
                ans = p;
            }
            break;
        }
        if (a[p].val > val && a[p].val < a[ans].val) ans = p;

        p = val < a[p].val ? a[p].l : a[p].r;
```

```
    }
    return a[ans].val;
}

void Remove(int &p, int val) {
    if (p == 0) return;
    if (val == a[p].val) { // 检索到了val
        if (a[p].cnt > 1) { // 有重复直接减少副本数
            a[p].cnt--, Update(p);
            return;
        }
        if (a[p].l || a[p].r) { // 不是叶子节点向下旋转
            if (a[p].r == 0 || a[a[p].l].dat > a[a[p].r].dat)
                zig(p), Remove(a[p].r, val);
            else
                zag(p), Remove(a[p].l, val);
            Update(p);
        }
        else p = 0; // 叶子节点，删除
        return;
    }
    val < a[p].val ? Remove(a[p].l, val) : Remove(a[p].r, val);
    Update(p);
}

int main() {
    Build();
    cin >> n;
    while (n--) {
        int opt, x;
        scanf("%d%d", &opt, &x);
        switch (opt) {
        case 1:
            Insert(root, x);
            break;
        case 2:
            Remove(root, x);
            break;
        case 3:
            printf("%d\n", GetRankByVal(root, x) - 1);
            break;
        case 4:
            printf("%d\n", GetValByRank(root, x + 1));
            break;
        case 5:
            printf("%d\n", GetPre(x));
            break;
        case 6:
            printf("%d\n", GetNext(x));
            break;
        }
    }

}
```

3.特点：

treap就是在二叉树的基础上加了一个随机域来保证堆的性质 这样可以把树的高度维护到期望的logn级别，其他的就是一棵二叉排序树，并没有什么特别的。

splay树

1.定义：

二叉查找树的一种改进数据结构-伸展树（Splay Tree）。它的主要特点是不会保证树一直是平衡的，但各种操作的平摊时间复杂度是O(log n)，因而，从平摊复杂度上看，二叉查找树也是一种平衡二叉树。另外，相比于其他树状数据结构（如红黑树，AVL树等），伸展树的空间要求与编程复杂度要小得多。

伸展树的出发点是这样的：考虑到局部性原理（刚被访问的内容下次可能仍会被访问，查找次数多的内容可能下一次会被访问），为了使整个查找时间更小，被查频率高的那些节点应当经常处于靠近树根的位置。这样，很容易得想到以下这个方案：每次查找节点之后对树进行重构，把被查找的节点搬移到树根，这种自调整形式的二叉查找树就是伸展树。每次对伸展树进行操作后，它均会通过旋转的方法把被访问节点旋转到树根的位置。

为了将当前被访问节点旋转到树根，我们通常将节点自底向上旋转，直至该节点成为树根为止。“旋转”的巧妙之处就是在不打乱数列中数据大小关系（指中序遍历结果是全序的）情况下，所有基本操作的平摊复杂度仍为O（log n）。

2.基本操作：

变量声明：f[i]表示i的父结点，ch【 i 】【0】表示i的左儿子，ch【i】【1】表示i的右儿子，key[i]表示i的关键字（即结点i代表的那个数字），cnt[i]表示i结点的关键字出现的次数（相当于权值），size[i]表示包括i的这个子树的大小；sz为整棵树的大小，rt为整棵树的根。

clear操作：将当前点的各项值都清0（用于删除之后）

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

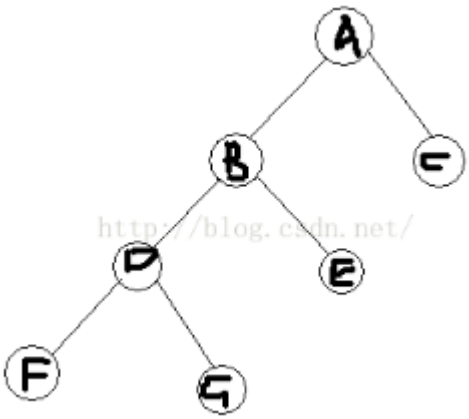
get操作：判断当前点是它父结点的左儿子还是右儿子

```
bool get(int x)
{//get一下该节点是左孩子还是右孩子
    return ch[f[x]][1]==x;
}
```

pushup操作：更新当前点的size值（用于发生修改之后）

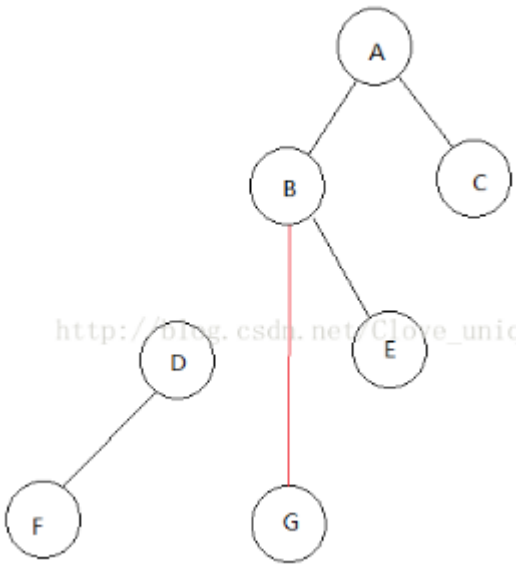
```
void pushup(int x)
{//更新这棵树的节点个数
    if (x)
    {
        size[x]=cnt[x];
        if (ch[x][0]) size[x]+=size[ch[x][0]];
        if (ch[x][1]) size[x]+=size[ch[x][1]];
    }
}
```

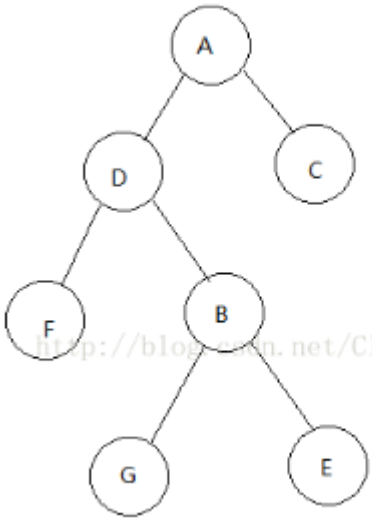
rotate操作:



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D rotate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。

step 3：update一下当前点和各个父结点的各个值

```
void rotate(int x)
{
    int old=f[x],oldf=f[old],which=get(x);
    ch[old][which]=ch[x][which^1]; f[ch[old][which]]=old;  //这两句的意思是:
    //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
    ch[x][which^1]=old; f[old]=x;
    //我给我爸爸当爹，我爸爸管我叫爸爸
    f[x]=oldf;//我的爷爷成了我的爸爸
    if (oldf) ch[oldf][ch[oldf][1]==old]=x;
    pushup(old); pushup(x);//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。此代码直接splay到根。

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 splay(int x)//splay树平衡
{
    for (int fa; fa=f[x]; rotate(x))
        if (f[fa])
            rotate((get(x)==get(fa))?fa:x);//如果祖父三代连城一条线，就要从祖父哪里rotate
    rt=x;
}
```

insert操作 其实插入操作是比较简单的，和普通的二叉查找树基本一样。 step 1：如果root=0，即树为空的话，做一些特殊的处理，直接返回即可。 step 2：按照二叉查找树的方法一直向下找，其中： 如果遇到一个结点的关键字等于当前要插入的点的话，我们就等于把这个结点加了一个权值。因为在二叉搜索树中是不可能出现两个相同的点的。并且要将当前点和它父亲结点的各项值更新一下。做一下splay。

如果已经到了最底下了，那么就可以直接插入。整个树的大小要+1，新结点的左儿子右儿子（虽然是空）父亲还有各项值要一一对应。并且最后要做一下他父亲的update（做他自己的没有必要）。做一下splay。

```
void insert(int x)//x为权值
{
    if (rt==0)
    {
        sz++; key[sz]=x; rt=sz;
        cnt[sz]=size[sz]=1;
        f[sz]=ch[sz][0]=ch[sz][1]=0;
        return;
    }
    int now=rt,fa=0;
    while (1)
    {
        if (x==key[now])//这个数在树中已经出现了
        {
            cnt[now]++; pushup(now); pushup(fa); splay(now); return;
        }
        fa=now; now=ch[now][key[now]<x];
        if (now==0)
        {
            sz++;
            size[sz]=cnt[sz]=1;
            ch[sz][0]=ch[sz][1]=0;
            ch[fa][x>key[fa]]=sz;//根据加入点的顺序重新标号
            f[sz]=fa;
            key[sz]=x;
            pushup(fa); splay(sz); return;
        }
    }
}
```

rnk操作:查询x的排名 初始化：ans=0，当前点=root 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 rnk(int x)
{
    int now=rt,ans=0;
    while (1)
    {
        if (x<key[now]) now=ch[now][0];
        else
        {
            ans+=size[ch[now][0]];
            if (x==key[now])
            {//此时x和树中的点重合，树中不允许有两个相同的点
                splay(now); return ans+1;
            }
```

```
            ans+=cnt[now];
            now=ch[now][1];//到达右孩子处
        }
    }
}
```

kth操作:找到排名为x的点

初始化：当前点=root 和上面的思路基本相同： 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。

```
int kth(int x)
{
    int now=rt;
    while (1)
    {
        if (ch[now][0] && x<=size[ch[now][0]])
            now=ch[now][0];
        else
        {
            int temp=size[ch[now][0]]+cnt[now];
            if (x<=temp)
                return key[now];
            x-=temp; now=ch[now][1];
        }
    }
}
```

求x的前驱（后继），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。
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

```
int pre()//由于进行splay后，x已经到了根节点的位置
{//所以只要寻找左右子树最左边（或最右边的）数
    int now=ch[rt][0];
    while (ch[now][1]) now=ch[now][1];
    return now;
}

int next()
{
    int now=ch[rt][1];
    while (ch[now][0]) now=ch[now][0];
    return now;
}
```

del操作: 删除操作是最后一个稍微有点麻烦的操作。 step 1：随便find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（f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update新根。

```
void del(int x)
{
    rnk(x);
    if (cnt[rt]>1) {cnt[rt]--; pushup(rt); return;}//有多个相同的数
    if (!ch[rt][0] && !ch[rt][1]) {clear(rt); rt=0; return;}
    if (!ch[rt][0]) {
        int oldrt=rt; rt=ch[rt][1]; f[rt]=0; clear(oldrt); return;
    }
    else if (!ch[rt][1]) {
        int oldrt=rt; rt=ch[rt][0]; f[rt]=0; clear(oldrt); return;
    }
    int oldrt=rt; int leftbig=pre();
    splay(leftbig);
    ch[rt][1]=ch[oldrt][1];
    f[ch[oldrt][1]]=rt;
    clear(oldrt);
    pushup(rt);
}
```

3:例题

luogu3369【模板】普通平衡树

分析：裸的splay树，将以上各个步骤的代码组合一下即可。

文艺平衡树

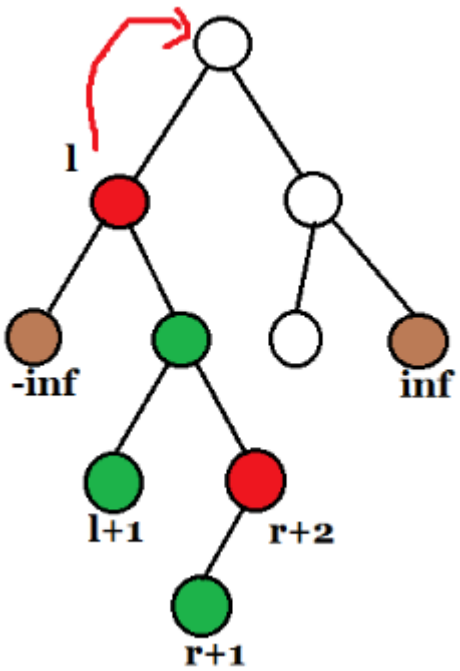
分析：多了一个区间翻转的操作

首先，也是最重要的，我们认为伸展树中序遍历即是我们维护的序列！什么意思呢？比如有数据在数组中这样存放：a[5]={5,4,3,1,2};那么存入伸展树后，再中序遍历的结果应该还是：{5，4，3，1，2}。即下标从小到大，而不是里面的值从小到大！这是与SBT树最大的不同！

原理：若要翻转[l+1, r+1]，将r+2 Splay到根，将l Splay到 r+2 的左儿子，然后[l+1, r+1]就在根节点的右子树的左子树位置了，给它打上标记

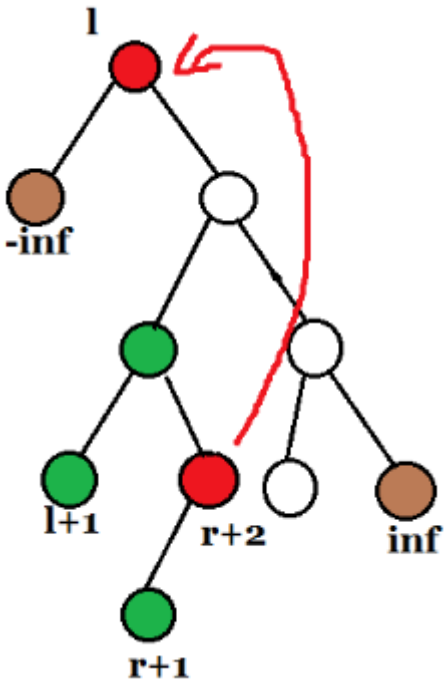
step1

先使l旋转到根

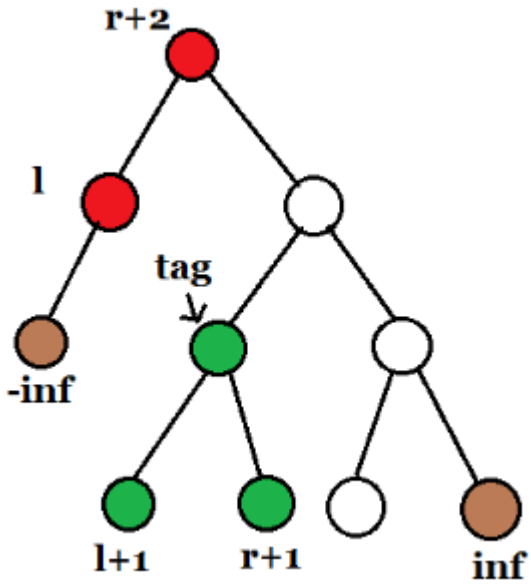


step2

使r+2旋转到根，



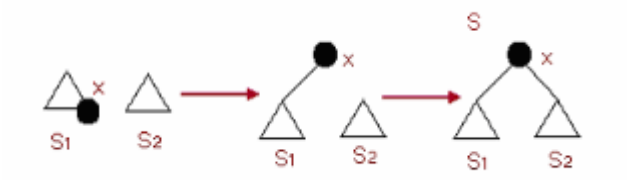
由于l < r+2,此时l成了r+2的左子树，那么r+2的右子树的左子树即为所求得区间，我们就可以对这棵子树随意操作了！比如删除整个区间，区间内的每个数都加上x，区间翻转，区间旋转等。



4：其他操作

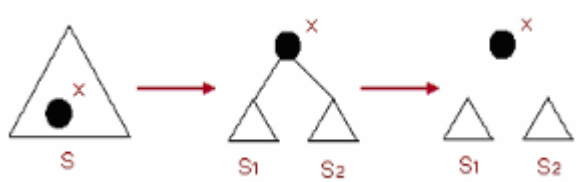
1.合并两棵伸展树

Join(S1,S2)：将两个伸展树S1与S2合并成为一个伸展树。其中S1的所 有元素都小于S2的所有元素。 首先，我们找到伸展树S1 中最大的一个元素x，再通过Splay(x,S1)将x 调整到伸展树S1 的根。然后再将S2 作为x 节点的右子树。这样，就得到了新的伸展树S。



2.分离两棵伸展树

Split(x,S)：以x 为界，将伸展树S 分离为两棵伸展树S1 和S2，其中S1 中所有元素都小于x，S2中的所有元素都大于x。 首先执行Find(x,S)，将元素x 调整为伸展树的根节点，则x 的左子树就是 S1，而右子树为S2。



总结：

相较于treap来说，splay不需要任何额外的内容，只要保证一个splay和旋转的操作即可，而所谓的splay操作 就是通过旋转把一个点向上转到目标点的操作。