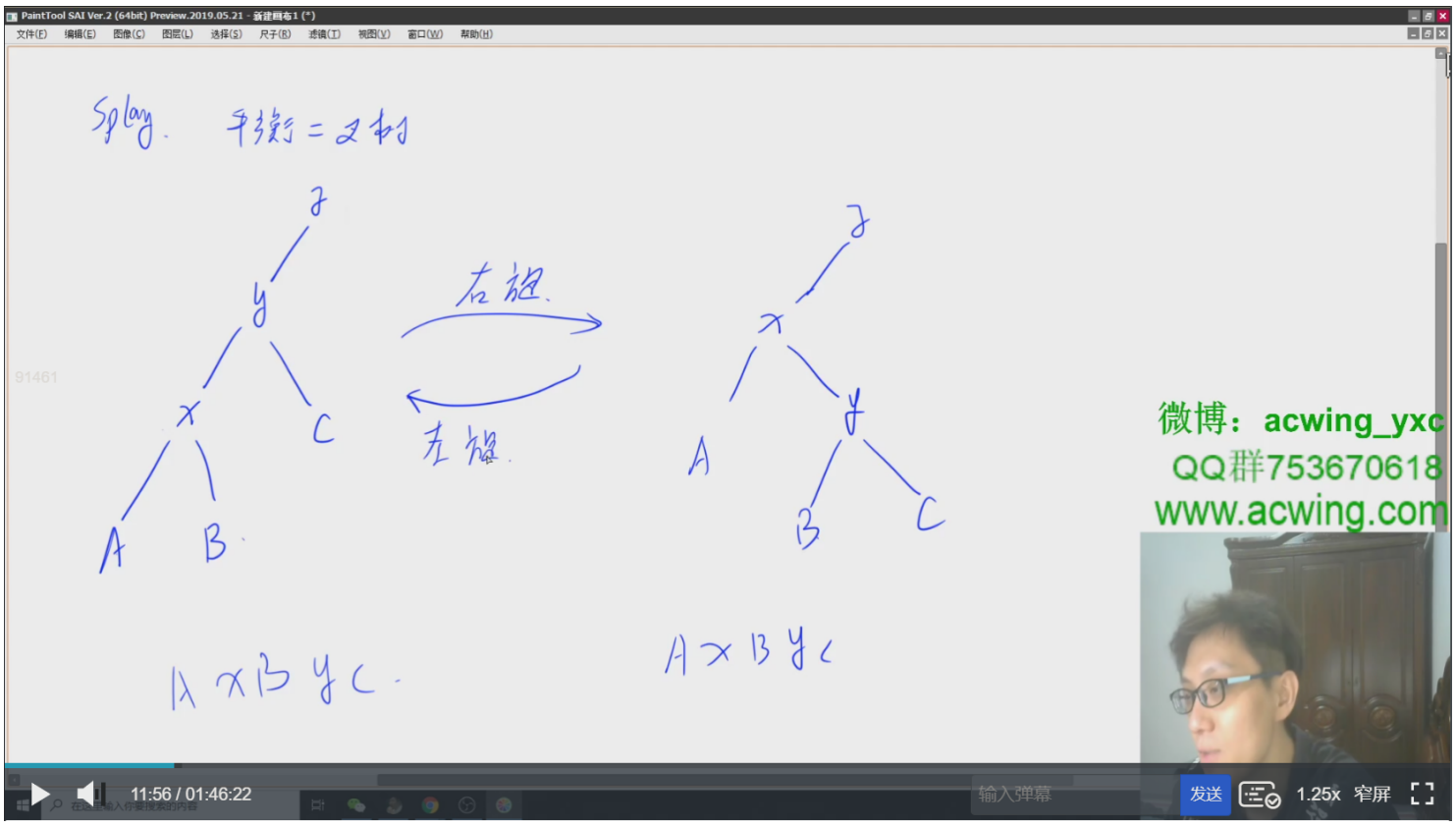
L3C2 Lesson1

Splay

- Splay是平衡树的一种（Splay，红黑树，treap（很多操作做不了），SBT（size balance tree），AVL，B树，B+树），B树和B+树是多叉树，硬盘里用得比较多。
- Splay代码适中，非常灵活，支持很多操作，例如可以处理有关线段的问题（例如子序列翻转，此时线段树无法处理，且比线段树更加灵活）

  Splay基本知识：

1. 平衡二叉树，期望高度是O(logn)级别
2. 有左旋和右旋操作，需要维护父亲结点。**左右旋在保证中序遍历结果不变的情况下，能够调整树的高度。**



3. 如何保证树的平均高度是O(logn)级别

   每一次插入，查询后即将当前点旋转至树根，同时保证中序遍历不变

   核心思想：每操作一个结点，均将改结点旋转至树根。某一个点如果当前用到，那么这个点之后可能还会被用到（局部性原理，缓存）

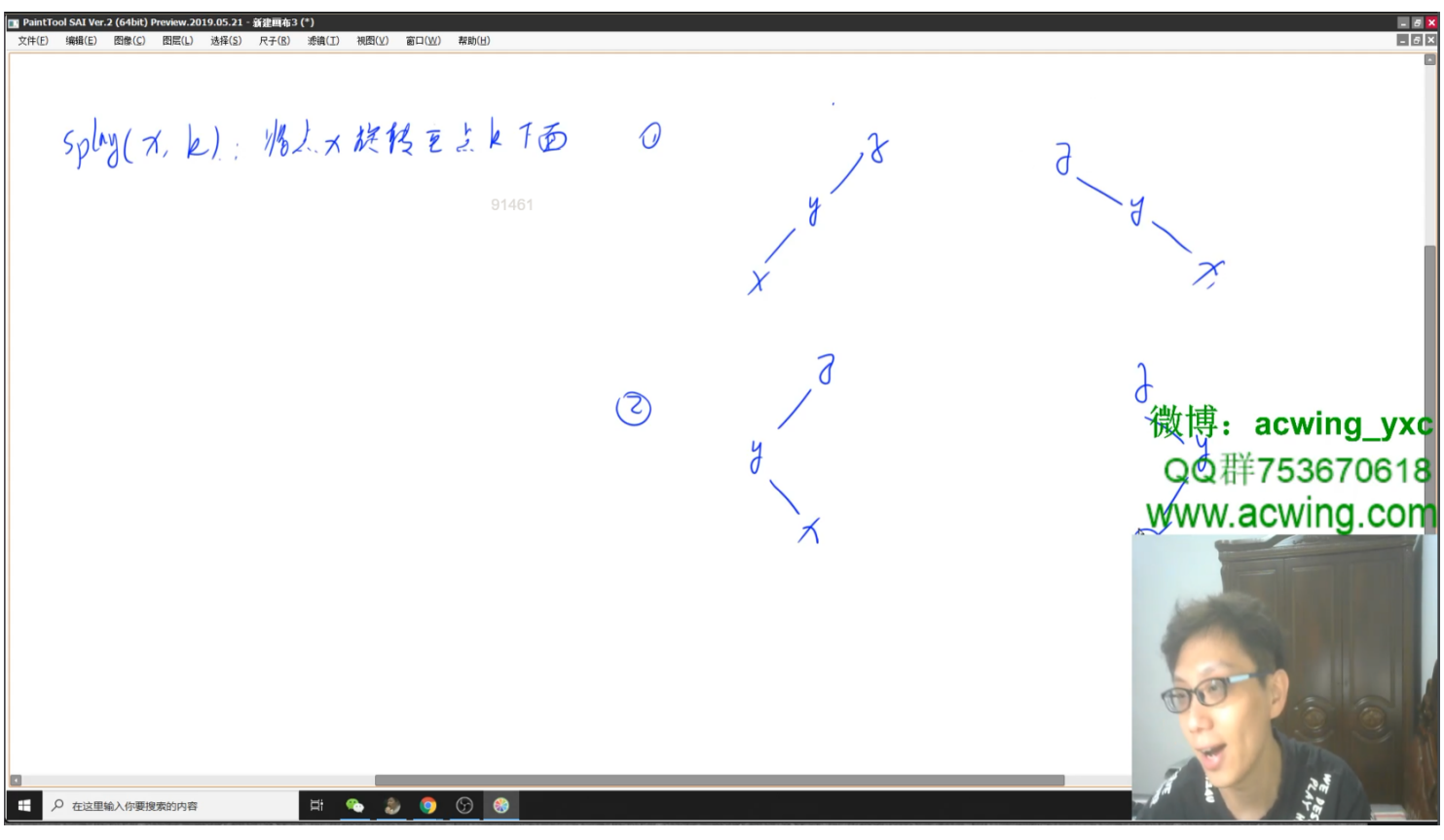
   结论：不管在什么样的数据，每次操作时间复杂度平均是O(logN)，证明暂时忽略。

4. 如何将某个点旋转至树根（Splay操作，代码中定义为**Splay函数**）

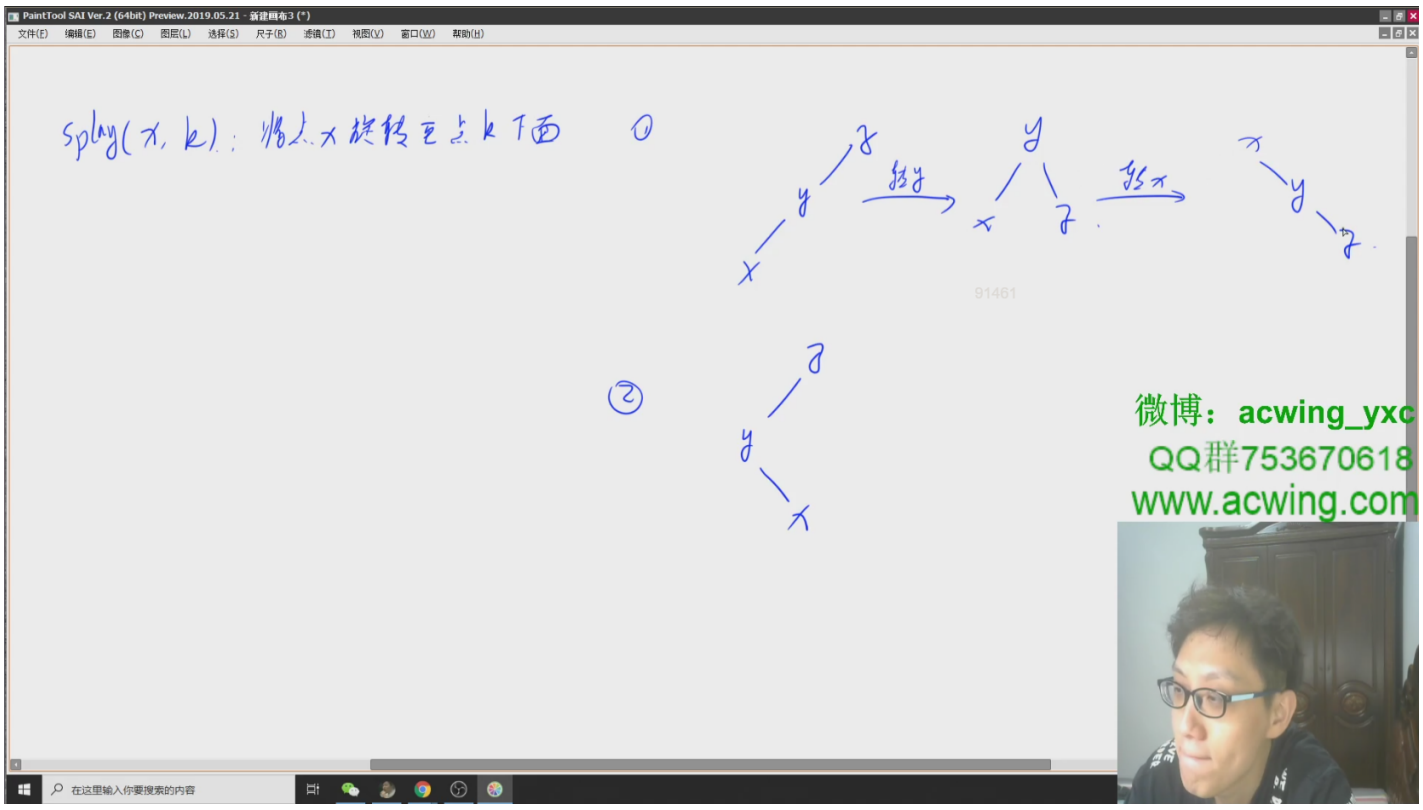
   Splay(x, k) 将x点旋转至k点下面，旋转至左下还是右下取决于x与k的大小关系

   Splay(x, 0) 则表示将点x旋转到根

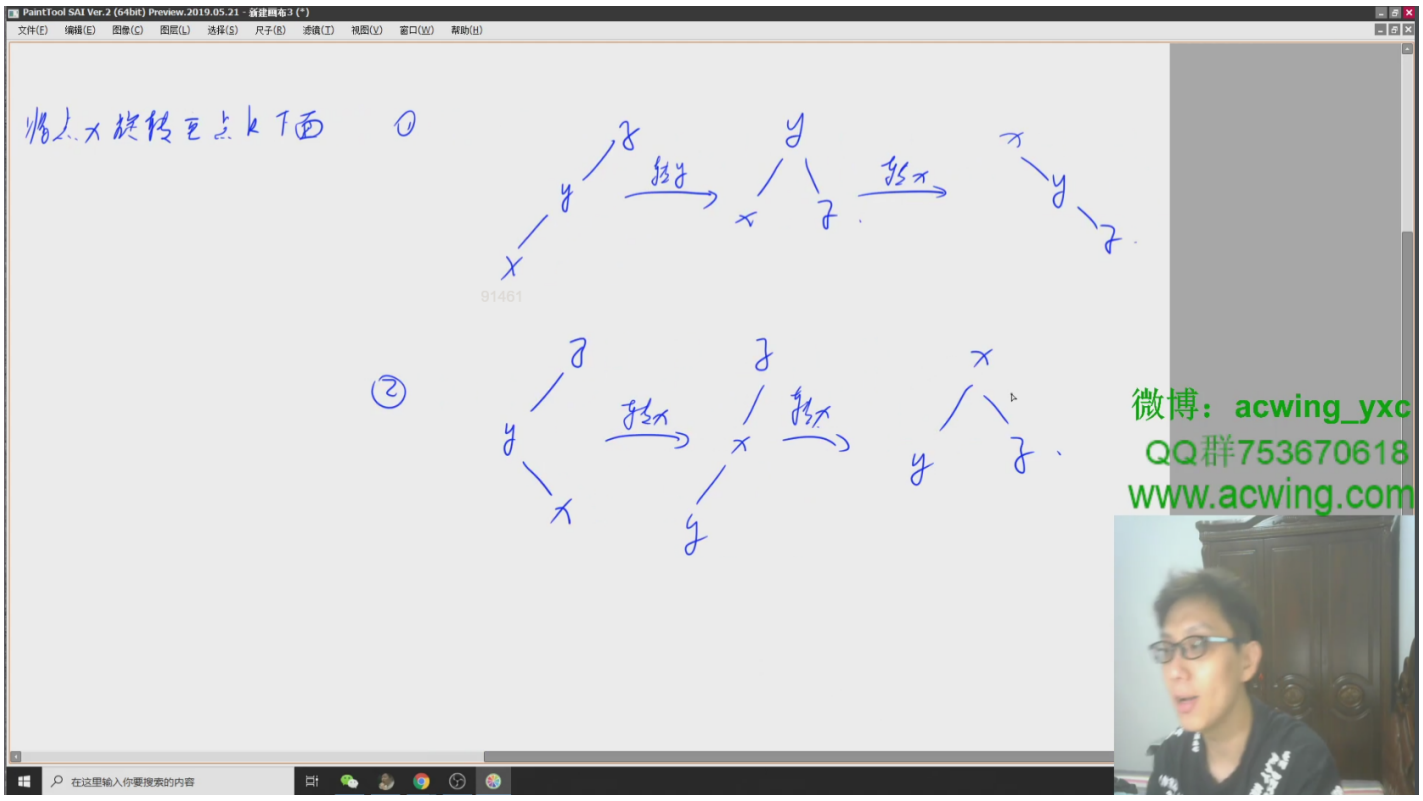
   步骤：两大类，四种小情况



1. 第一种情况，先转y，再转x。（直线）



2. 第二种情况，先转x，再转x。（折线）



3. 另外两种情况对称实现

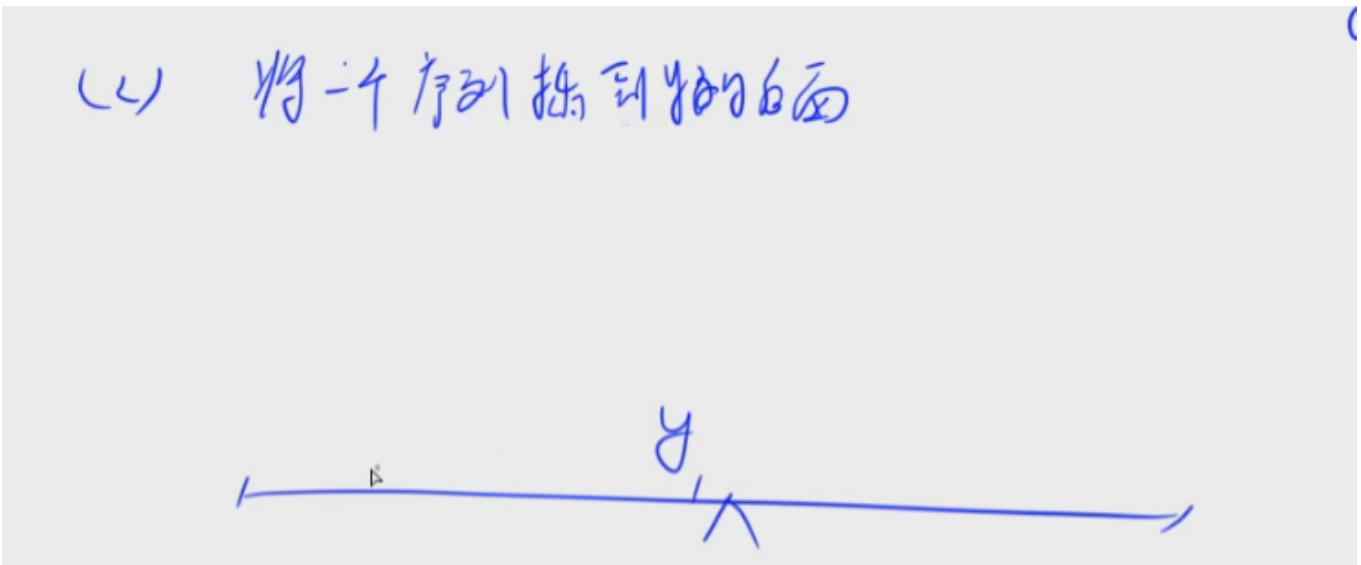
4. k一般而言只有两种情况，k=0（x转到根节点位置）或者k=根节点（x转到根节点下面）

5. 操作举例：

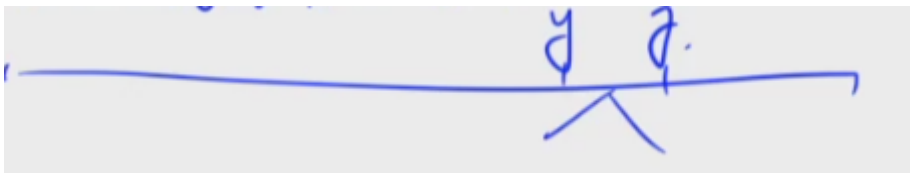
1. 插入

1. 插入一个数：根据BST规则找到x应该放的位置，再将x splay操作到根节点位置

2. 将一个序列插入至y的后面（保证插完后中序遍历序列在y之后）

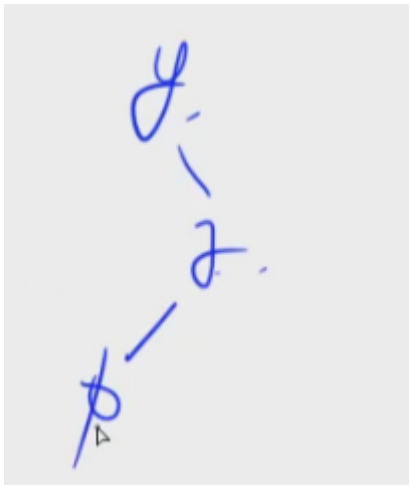


1. 找到y的后继z（大于y的最小值）

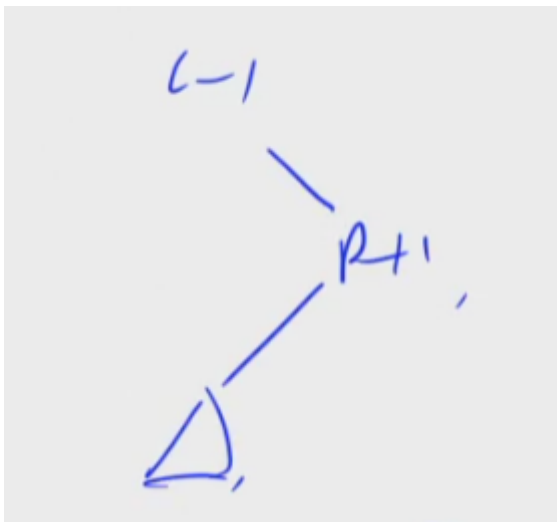


2. 将y转到根，splay(y, 0)

3. 将z转到y的下面，splay(z, y)，旋转完后此时z的左子树为空。将序列插入到(y, z)之间，即将序列构造成BST后插入为z的左子树上。



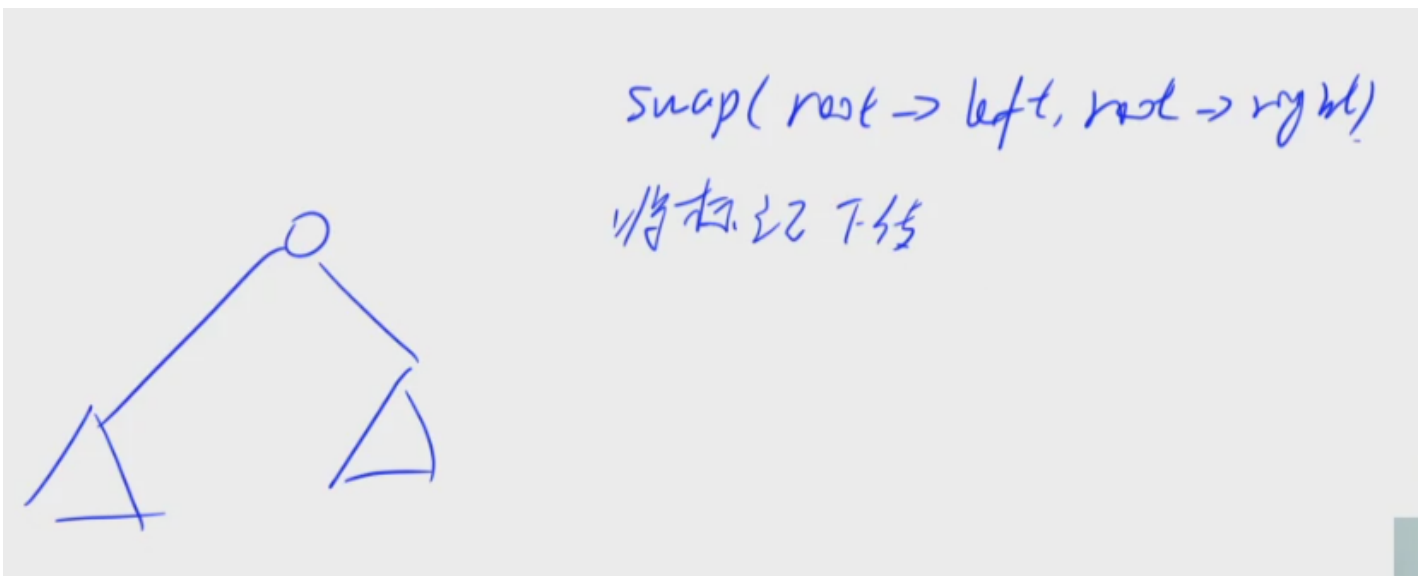
2. 删除一段序列[L, R]
    1. 找到L的前驱L-1, R的后继R+1
    2. 将L-1转到根节点，R+1转到根节点下。此时R的左子树即为[L, R]这一段序列。然后将左儿子置为空，即删除序列[L, R]



    3. 找前驱/后继，找第k个点... 和普通平衡树一样

6. Splay如何维护信息
    1. 找第k个数（需要维护size和cnt，类似treap）。**pushup()** 负责需要维护信息。pushup() 放置在**旋转函数最后**（类似treap）和**更新结构之后。**
    2. 懒标记，flag（类似线段树）。**pushdown()** 将懒标记下传。pushdown() 函数放置在**递归操作之前**

       翻转序列操作：



    3. 其他信息....
    4. Splay开始时能满足BST性质，之后左右旋转插入后可能不能满足BST定义：左子树<父结点，右子树>父结点，但Splay一定保证**中序遍历结果是当前序列的顺序**，**splay的中序遍历结果是我们最终的需要维护的序列**，但不一定保证像BST那样有序。只有插入时有序，其他情况下不一定有序。
    5. 同treap一样，会加入两个哨兵

Splay模板题

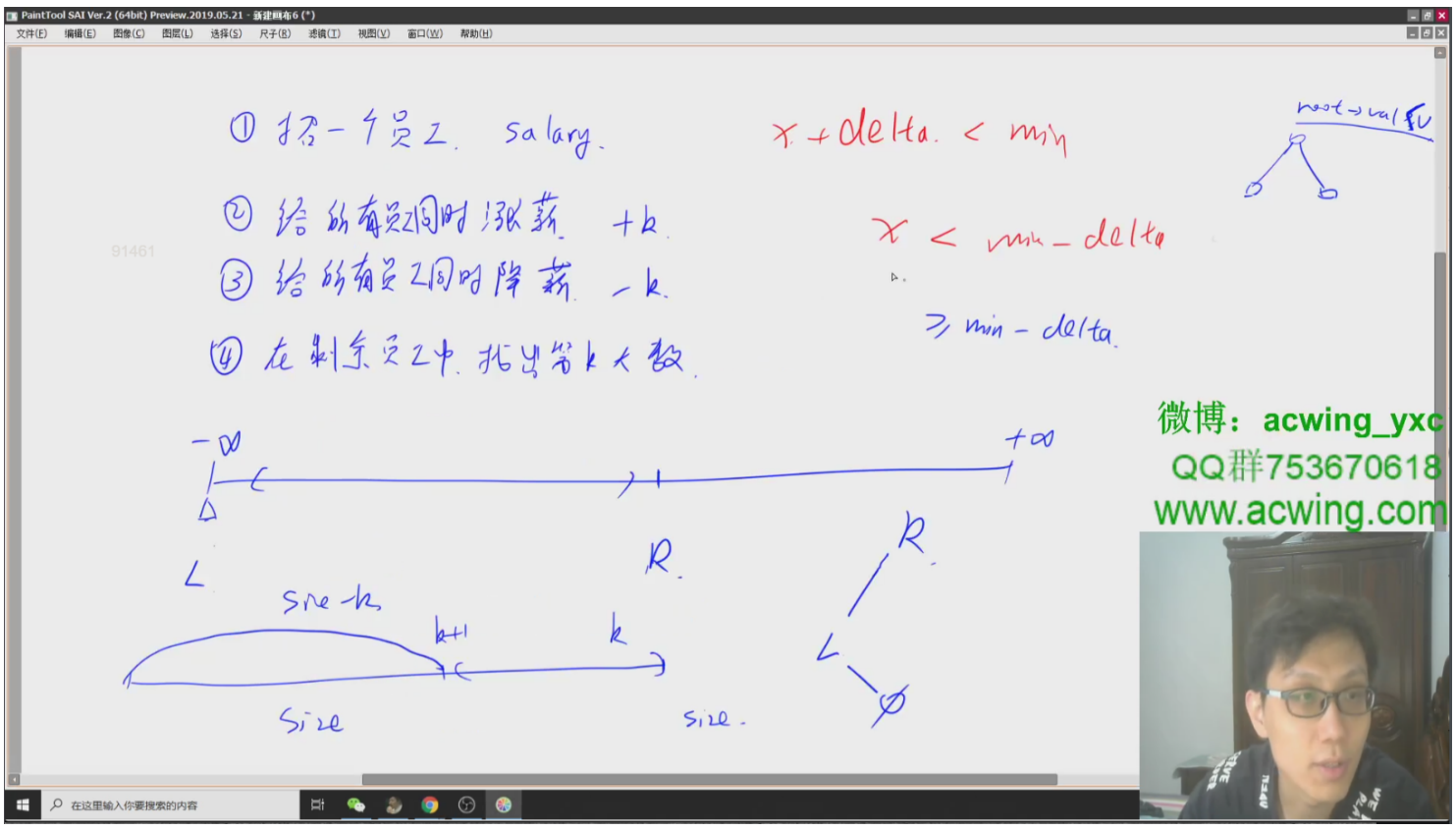
```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 100010;

int n, m;
struct Node {
    //s[0], s[1]分别代表左右儿子，p代表父结点，v代表当前结点存储的值
    //只有插入时满足BST的性质，后续操作可能不满足BST定义
    int s[2], p, v;
```

```
    //size维护当前结点左右儿子+当前结点所有个数信息
    //flag维护懒标记，表示当前结点是否需要翻转
    int size, flag;

    void init(int _v, int _p) {      //插入结点时初始化函数
        v = _v, p = _p;
        size = 1;
    }

}tr[N];

int root, idx;  //root根节点，idx表示当前使用到了哪个结点

void pushup(int x) {    //信息向上传递，更新size，每次旋转最后
    tr[x].size = tr[tr[x].s[0]].size + tr[tr[x].s[1]].size + 1; //左右儿子+1
}

void pushdown(int x) {  //将懒标记flag下传，向下传递，每次递归操作之前
    if (tr[x].flag) {   //当前结点需要翻转
        swap(tr[x].s[0], tr[x].s[1]);   //注意是交换x的左右两颗子树位置
        tr[tr[x].s[0]].flag ^= 1;   //和1异或，1->0，0->1，即下传子树懒标记
        tr[tr[x].s[1]].flag ^= 1;
        tr[x].flag = 0;     //重置当前结点懒标记
    }
}

void rotate(int x) {    //将x（右儿子）左旋或者（左儿子）右旋到其父结点位置
    int y = tr[x].p, z = tr[y].p;   //y是x父结点，z是y父结点
    //k=0表示x是y左儿子，1表示x是y右儿子，根据k动态实现左旋或者右旋
    int k = tr[y].s[1] == x;
    //将x左右旋一共需要变三条边
    //1. 将x转到原y的位置
    tr[z].s[tr[z].s[1] == y] = x, tr[x].p = z;  //t[z].s[1]==y即为x该放的位置
    //2. 将x的右或者左儿子变为现y的儿子，根据左右旋图理解
    tr[y].s[k] = tr[x].s[k ^ 1], tr[tr[x].s[k ^ 1]].p = y;
    //3. 将现y变为x的儿子，据图理解
    tr[x].s[k ^ 1] = y, tr[y].p = x;

    //旋转之后pushup操作，信息从下往上传递
    pushup(y), pushup(x);   //注意顺序
}

void splay(int x, int k) {  //splay核心函数，将点x旋转至k下面
    while (tr[x].p != k) {  //x还未旋转至k下面
        int y = tr[x].p, z = tr[y].p;   //y是x父，z是y父
        if (z != k) {   //需要转两次
            //判断结果0表示直线型，1表示折线型
            if ((tr[z].s[1] == y) ^ (tr[y].s[1] == x)) rotate(x);  //折线先转x
            else rotate(y);     //直线型先转y
        }
        rotate(x);  //再旋转一次x
    }

    if (!k) root = x;   //splay(x, 0)表示将x旋转为根节点，更新根节点
}

void insert(int v) {    //在splay中新插一个结点v，插入时满足BST性质
                        //根据题意，insert时无需下传懒标记
    int u = root, p = 0;    //从根节点开始往下迭代，p表示u的父结点
    while (u) p = u, u = tr[u].s[v > tr[u].v];  //更新p，根据v更新u
    u = ++ idx;  //idx自增
    if (p) tr[p].s[v > tr[p].v] = u;    //如果u存在父结点，更新父结点儿子信息
    tr[u].init(v, p);   //更新新增u结点的v和p信息
    splay(u, 0);    //根据splay定义将新增点旋至根节点
}

int get_k(int k) {  //获取中序遍历中第k个点
    int u = root;   //从根节点开始，迭代法获取
    while (u) {
        pushdown(u);    //递归之前下传懒标记
        if (tr[tr[u].s[0]].size >= k) u = tr[u].s[0];   //左子树中找
        else if (tr[tr[u].s[0]].size + 1 == k) return u; //将该点索引返回
        else k -= tr[tr[u].s[0]].size + 1, u = tr[u].s[1];      //更新k，右子树中找
    }

    return -1;  //没有找到第k个点
}

void output(int u) {    //递归输出u结点子树中序遍历结果，即我们需要维护的序列
    pushdown(u);    //递归之前下放懒标记
```

```
    if (tr[u].s[0]) output(tr[u].s[0]);     //递归左子树
    //如果不是左右哨兵，则输出
    if (tr[u].v > 0 && tr[u].v < n + 1) printf("%d ", tr[u].v);
    if (tr[u].s[1]) output(tr[u].s[1]);     //递归右子树
}

int main(void) {
    scanf("%d%d", &n, &m);

    //按序插入n个结点，并在左右分别插入两个哨兵结点（0和n+1类似treap）
    for (int i=0; i<=n+1; i++) insert(i);
    while (m--) {
        int l, r;
        scanf("%d%d", &l, &r);
        l = get_k(l), r= get_k(r+2);    //考虑哨兵，[l, r]的前驱后继分别为l和r+2
        //将L+1前驱L旋转至root，R+1后继R+2旋转至L，则[L+1, R+1]位于R+2左子树中
        //进一步将R+2左子树懒标记^1(1->0, 0->1)，实现原序列[L, R]的翻转。
        splay(l, 0), splay(r, l);
        tr[tr[r].s[0]].flag ^= 1;     //设置懒标记，实现序列翻转
    }

    output(root);   //输出最终序列

    return 0;
}
```

郁闷的出纳员（NOI2004）

- 分析



splay即可以像线段树维护普通序列，也可以**像平衡树维护有序序列**，本题中Splay维护有序序列。另外，线段树能维护的信息splay一般也能维护。

- 代码（代码块不能分级）

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <algorithm>

using namespace std;        //950. 郁闷的出纳员

const int N = 100010, INF = 1e8;

int n, m, delta;    //m表示最低工资，splay中结点v+delta=实际工资
                    //整体加减工资在delta上操作
struct Node {
    int s[2], p, v;
    int size;

    void init(int _v, int _p) {
        v = _v, p = _p;
        size = 1;
    }
}tr[N];

int root, idx;
```

```

void pushup(int x) {
    tr[x].size = tr[tr[x].s[0]].size + tr[tr[x].s[1]].size + 1;
}

void rotate(int x) {
    int y = tr[x].p, z = tr[y].p;
    int k = tr[y].s[1] == x;
    tr[z].s[tr[z].s[1] == y] = x, tr[x].p = z;
    tr[y].s[k] = tr[x].s[k ^ 1], tr[tr[x].s[k ^ 1]].p = y;
    tr[x].s[k ^ 1] = y, tr[y].p = x;

    pushup(y), pushup(x);
}

void splay(int x, int k) {
    while (tr[x].p != k) {
        int y = tr[x].p, z = tr[y].p;
        if (z != k)
            if ((tr[z].s[1] == y) ^ (tr[y].s[1] == x)) rotate(x);
            else rotate(y);
        rotate(x);
    }

    if (!k) root = x;
}

int insert(int v) {
    int u = root, p = 0;
    while (u) p = u, u = tr[u].s[v > tr[u].v];
    u = ++ idx;
    if (p) tr[p].s[v > tr[p].v] = u;
    tr[u].init(v, p);
    splay(u, 0);    //旋转至根节点
    return u;   //需要用到左哨兵，故返回下标
}

int get_next(int v) {   //注意是找到v的后继，本题splay维护有序序列
    int u = root, res = 0;   //res存储后继下标，使用迭代方式
    while (u) {
        //注意是比较符号!
        if (tr[u].v >= v) res = u, u = tr[u].s[0];  //更新res，进入左子树
        else u = tr[u].s[1];    //进入右子树
    }

    return res;
}

int get_k(int k) {
    int u = root;
    while (u) {
        if (tr[tr[u].s[0]].size >= k) u = tr[u].s[0];
        else if (tr[tr[u].s[0]].size + 1 == k) return tr[u].v; //注意此时返回v!
        else k -= tr[tr[u].s[0]].size + 1, u = tr[u].s[1];
    }

    return -1;
}

int main(void) {
    scanf("%d%d", &n, &m);
    //插入两个哨兵结点，并记录下标，右哨兵未使用
    int L = insert(-INF), R = insert(INF);

    int tot = 0;    //加入过公司的总人数
    while (n--) {
        char op[2];
        int k;
        scanf("%s%d", op, &k);

        if (*op == 'I') {
            //保证结点.v+delta=实际工资
            if (k >= m) insert(k - delta), tot ++;
        }
        else if (*op == 'A') delta += k;
        else if (*op == 'S') {
            delta -= k;
            R = get_next(m - delta);    //找到当前工资低于最低标准的右边界+1，即后继
            splay(R, 0), splay(L, R);   //splay操作后，L右子树需要删除的人
            tr[L].s[1] = 0;     //删除L右子树
```

```
            pushup(L), pushup(R);   //记得pushup操作!
        }
        else {
            if (k > tr[root].size - 2) puts("-1");
            else {
                //考虑右哨兵，第k大数为第size-k小数，注意需要加上delta
                printf("%d\n", get_k(tr[root].size - k) + delta);
            }
        }
    }

    printf("%d\n", tot - (tr[root].size - 2));    //-2删除两哨兵

    return 0;
}


```