FZOI

信息学

伸展树splay

引入

搜索一下splay树，我们会发现结果中：

结果中我们经常访问的页面会更靠前，
我们访问起来会更加方便

在程序设计中，这种思想很常见

基于这种思想
为了使多次查找时间更小，被查频率高的那些条目
就应当经常处于靠近树根的位置。

设计出了splay树

信息学

伸展树splay

Splay树（伸展树）是二叉查找树的改进。

在每次查找之后对树进行调整，把被查找的条目搬移到离树根近一些的地方。

伸展树是一种自调整形式的二叉查找树，它会沿着从某个节点到树根之间的路径，通过一系列的旋转把这个节点搬移到树根去。

伸展树的自我平衡使其拥有良好的性能，
因为频繁访问的节点会被移动到更靠近根节点，进而获得更快的访问速度。

- 可靠的性能——它的平均效率不输于其他平衡树。均摊复杂度是$O(\log_2 n)$。
- 存储所需的内存少——伸展树无需记录额外的什么值来维护树的信息，相对于Treap，内存占用要小。

建树


```
int root;//根节点是谁
struct trnode
{
    int d,n,c,f,son[2];//d为值，f为父亲编号，c为子树大小，n为同值节点的个数
}splayT[110000];
int len;//树上有几个节点
```

节点数更新

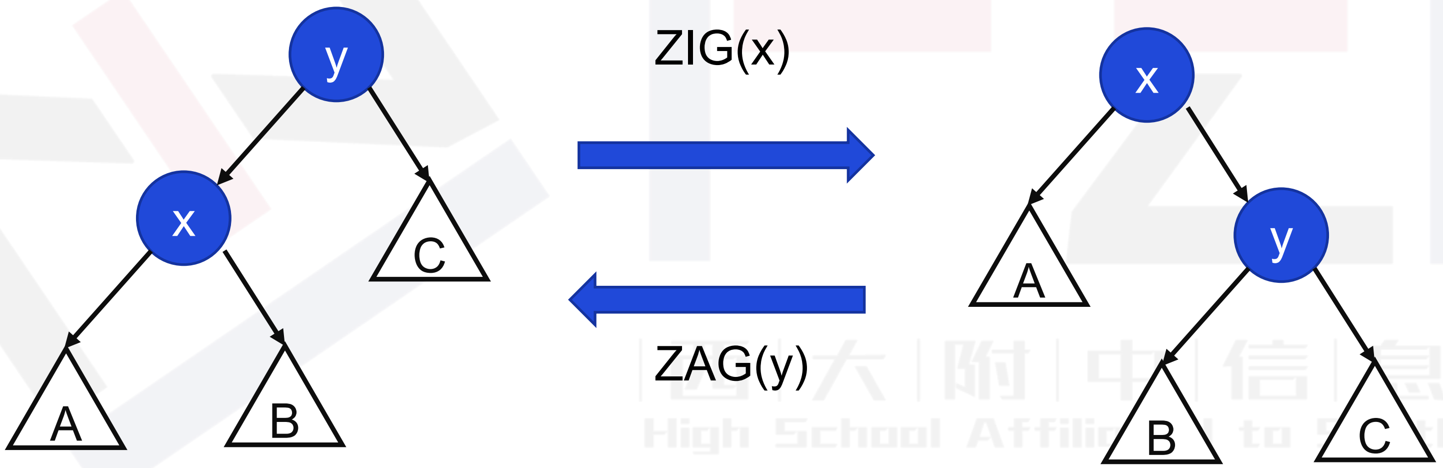
```
void update(int x)//更新x控制的节点数
{
    int lc=splayT[x].son[0];
    int rc=splayT[x].son[1];
    splayT[x].c=splayT[lc].c+splayT[rc].c+splayT[x].n;
}
```

添加一个点

```
void add(int d,int f)//添加值为d的点，认f为父亲，f认它为孩子
{
    len++;
    splayT[len].d=d;   splayT[len].n=1;    splayT[len].c=1;
    splayT[len].f=f;
    if(d<splayT[f].d)
        splayT[f].son[0]=len;
    else splayT[f].son[1]=len;
    splayT[len].son[0]=0;
    splayT[len].son[1]=0;
}
```

旋转

Splay操作是在保持Splay树有序性的前提下，通过一系列旋转操作将树中的元素x调整至树的根部的操作(Zig:右旋,Zag:左旋)。



旋转


```
void rotate(int x,int w) //w=0时为左旋，w=1时为右旋
{
    int f=splayT[x].f , ff=splayT[f].f; //在旋转之前先确定好x的父亲和爷爷
    //开始旋转建立关系
    int r , R; //r表示儿辈，R表示父辈

    r=splayT[x].son[w] , R=splayT[x].f; //x的儿子准备当x父亲的新儿子
    splayT[R].son[1-w]=r;
    if(r!=0) splayT[r].f=R;

    r=x,R=ff;//x准备当他爷爷新孩子
    if(splayT[R].son[0]==f) splayT[R].son[0]=r;
    else splayT[R].son[1]=r;
    splayT[r].f=R;

    r=f,R=x;//x的父亲当x的儿子
    splayT[R].son[w]=r;
    splayT[r].f=R;
    update(f);
    update(x);
}
```

Splay操作

在旋转的过程中，要分三种情况分别处理：

1)Zig 或 Zag

2)Zig-Zig 或 Zag-Zag

3)Zig-Zag 或 Zag-Zig

每次旋转操作由三个因素决定：

•*x*是其父节点*y*的左儿子还是右儿子；

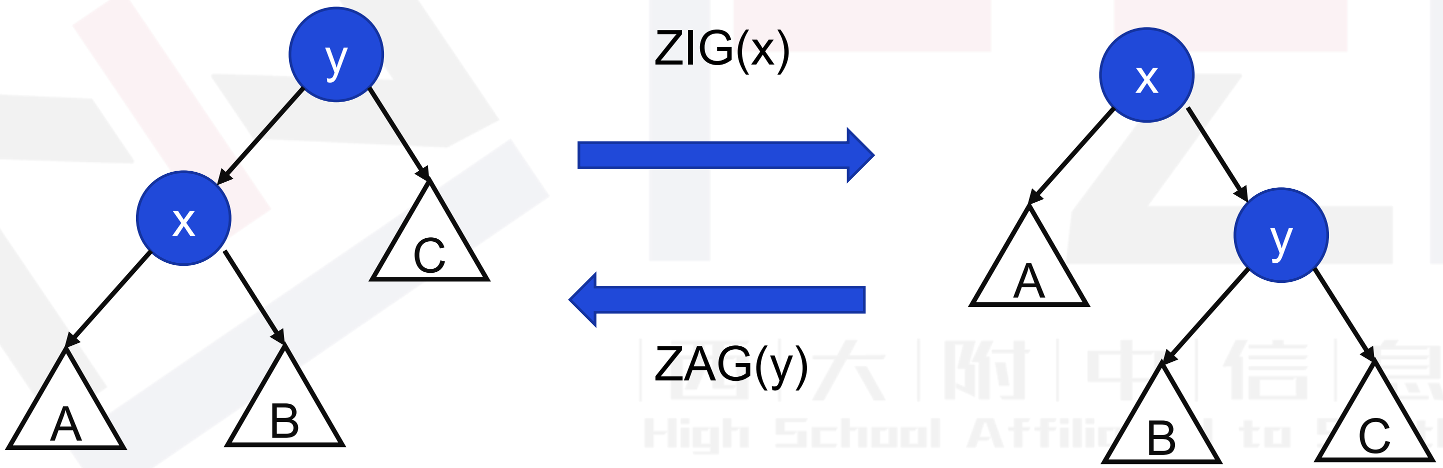
•*y*是否为根；

•*y*是其父节点*z*（*x*的祖父节点）的左儿子还是右儿子。

在每次旋转操作后，设置*y*的儿子为*x*是很重要的。如果*y*为空，那么*x*显然就是根节点了。

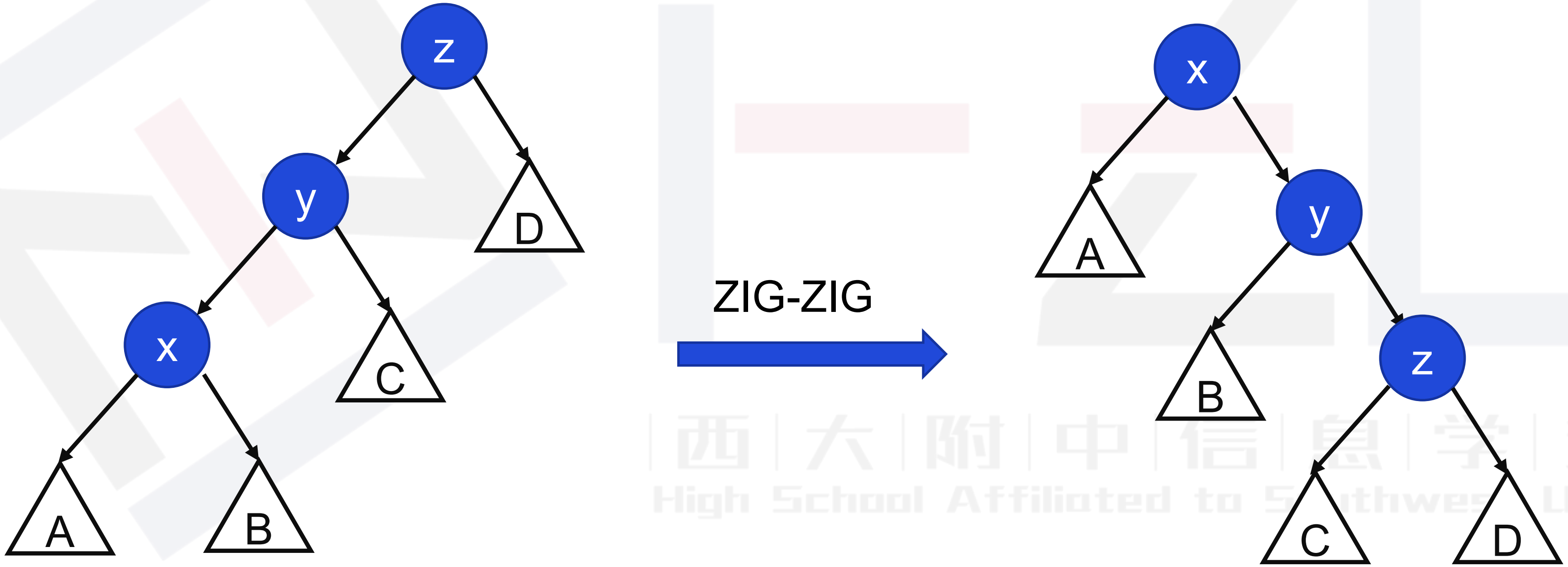
情况1

- Zig或Zag操作：
- 节点x的父节点y是根节点。



情况2

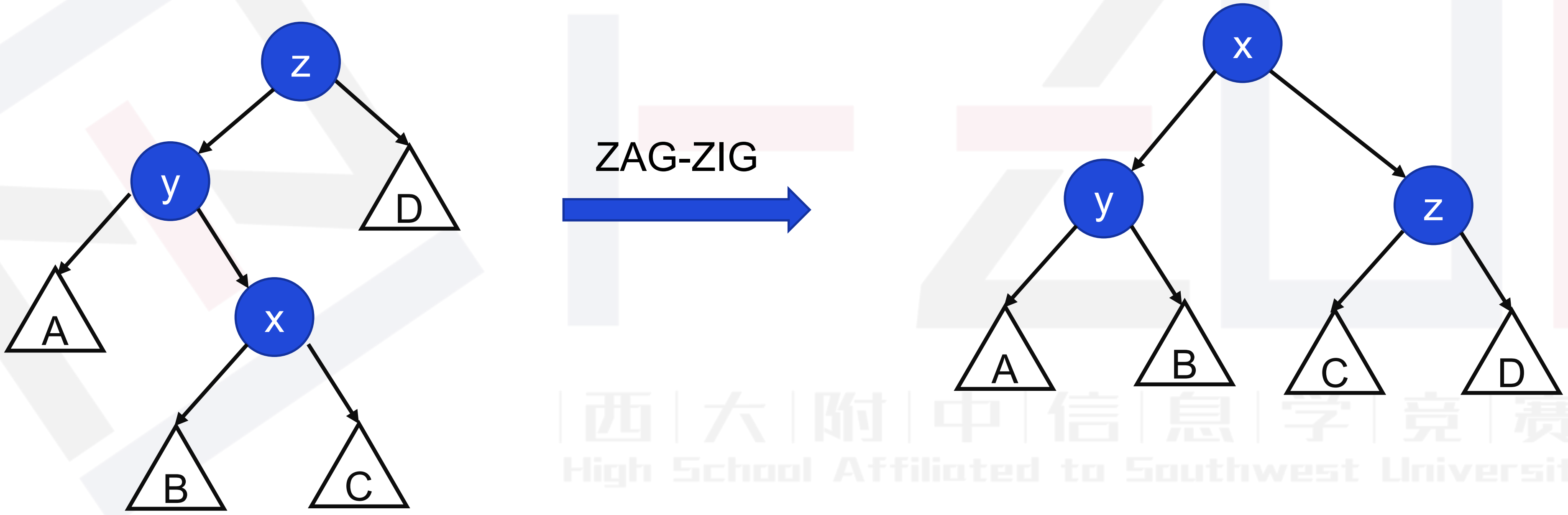
- Zig-Zig或Zag-Zag操作：
- 节点x的父节点y不是根节点，且x与y同时是各自父节点的左孩子或者同时是各自父节点的右孩子。



先将y右旋到z的位置，再将x右旋到y的位置

情况3

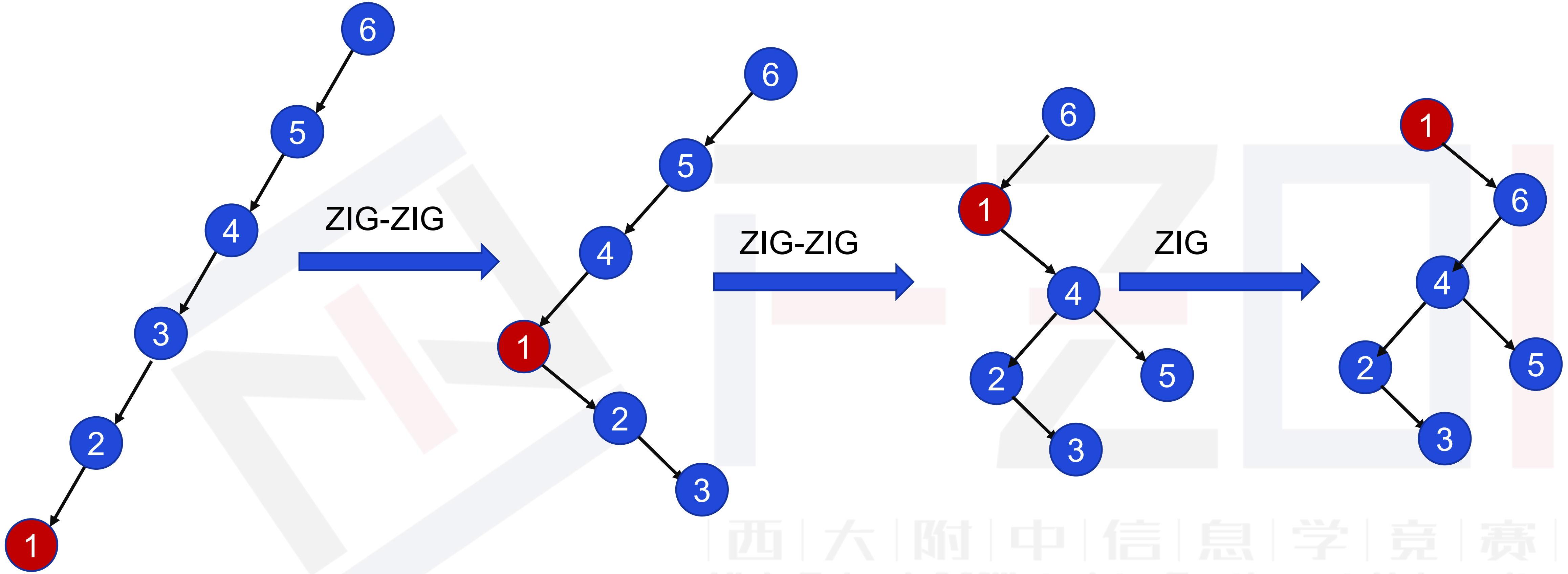
- Zig-Zag或Zag-Zig操作：
- 节点x的父节点y不是根节点，x与y中一个是其父节点的左孩子而另一个是其父节点的右孩子。



需先将x左旋到y到的位置，再将x右旋到z的位置。

为什么采用双层伸展

https://blog.51cto.com/greyfoss/5468576


6
5
4
3
2
1
ZIG-ZIG
6
5
4
1
2
3
ZIG-ZIG
6
1
4
2
5
3
ZIG
1
6
4
2
5
3

Splay操作


```
void splay(int x,int rt)//该函数是为了让x变为rt的孩子，无论是左孩子还是右孩子
{
    while(splayT[x].f!=rt)//如果rt是x的父亲，那么什么也不用做；不然就把x往上旋转
    {
        int f=splayT[x].f,ff=splayT[f].f;//准备x的父亲和爷爷
        if(ff==rt)//如果x的爷爷就是rt，那么只要x向上旋转一次（相当于跳一层）
        {
            if(splayT[f].son[0]==x)  rotate(x,1) ； else  rotate(x,0) ；
        }
            if(splayT[ff].son[0]==f  &&  splayT[f].son[0]==x)  {rotate(f,1);rotate(x,1);}
        else  if(splayT[ff].son[1]==f  &&  splayT[f].son[1]==x)  {rotate(f,0);rotate(x,0);}
        else  if(splayT[ff].son[0]==f  &&  splayT[f].son[1]==x)  {rotate(x,0);rotate(x,1);}
        else  if(splayT[ff].son[1]==f  &&  splayT[f].son[0]==x)  {rotate(x,1);rotate(x,0);}
    }
    if(rt==0) root=x;
}
```

具体分析可以看 https://oi-wiki.org/ds/splay/#%E5%AE%9E%E7%8E%B0_8

对于 n 个节点的 splay 树，做一次 splay 操作的均摊复杂度为O(log n)。

实际做m次splay操作，复杂度为O(mlog n + nlog n)

查找节点地址


```
int findip(int d)//找值为d的节点的地址，PS：如果不存在值为d的点，就找接近d的点（可能大也可能小）
{
    int x=root;
    while(splayT[x].d!=d)
    {
        if(d<splayT[x].d)
        {
            if (splayT[x].son[0]==0) break;
            else x=splayT[x].son[0];
        }
        else //if(d>splayT[x].d)
        {
            if (splayT[x].son[1]==0) break;
            else x=splayT[x].son[1];
        }
    }
    return x;
}
```

插入数值d的点

和二叉查找树一样，在插入一个值的时候，我们需要分两种情况进行讨论：

1.这棵树是空的

那么我们只要add一个点，然后root就是这点，就可以了。

2.这棵树不是空的

那么我们可以运用之前讲到的find函数，找到值为d的点在哪里。

若存在值为d的点，那么直接这个点的n++就好了，代表多了一个这个值的点；

若不存在值为d的点，这表示值为d的点应该出现在找到的这个点的儿子节点上，且这个儿子节点必为空，不然的话会继续findip()下去，找到的就不是这个点了。

这个时候直接在找到点的对应儿子节点加上我们要添加的点就好了。

最后由于Splay的特殊策略，我们还要把新加入的节点splay到根节点。

```
void ins(int d)//插入数值为d的点
{
    if(root==0){ add(d,0); root=len; return 0; }
    int x=findip(d);

    if(splayT[x].d==d)
    {
        splayT[x].n++;
        update(x);
        splay(x,0);
    }
    else
    {
        add(d,x);
        update(x);
        splay(len,0);
    }
}
```

删除数值为d的点

对比二叉查找树，splay有了旋转操作，对于删除操作的处理就简单很多了。
如果这个点出现多次，即splayT[x].n>1，那就直接减去一个，即splayT[x].n即可。

如果splayT[x].n=1：
首先先把待删除点找到，然后旋转到根节点，接下来分3种情况进行讨论：

1.**这个点没有左右儿子**：又由于这时这个点已经旋转到了根节点，它又没有左右儿子，那删除后，这颗splay就空了，直接把根设为空，树的大小设为0即可。

2.**这个点只有左/右子树**：此时的操作也很简单，由于只有左/右节点，直接让左/右儿子当根节点就好了。

3.**既有左儿子又有右儿子**：把左子树的最右点旋转到左子树的根，把这个点设置为根，由于这个点在左子树中最大，所以此时它一定没有右儿子。所以直接把删除点的右儿子拼到它的右儿子上就好了。

删除数值为d的点


```
void del(int d){
    int x=findip(d);  splay(x,0);
    if(splayT[x].n>1){splayT[x].n--; update(x); return 0;} //不止一个直接 --
    //分情况讨论
        if(splayT[x].son[0]==0 && splayT[x].son[1]==0){root=0;len=0;}
    else if(splayT[x].son[0]==0 && splayT[x].son[1]!=0){root=splayT[x].son[1];splayT[root].f=0;}
    else if(splayT[x].son[0]!=0 && splayT[x].son[1]==0){root=splayT[x].son[0];splayT[root].f=0;}
    else //if(splayT[x].son[0]!=0 && splayT[x].son[1]!=0) {
        int p=splayT[x].son[0];
        while(splayT[p].son[1]!=0)p=splayT[p].son[1];
        splay(p,x);
        int r=splayT[x].son[1],R=p;
        splayT[R].son[1]=r;
        splayT[r].f=R;
        root=R;splayT[root].f=0;
        update(R);
    }
}
```

寻找值为d的点的排名

只要找到这个点的位置，设为x。
把x旋转到根，根据二叉查找树的性质，左子树的所有元素都小于它，所以此时它的排名就是左子树大小加一。

```
int Getd_Rank (int d)//寻找值为d的点的排名
{
    int x=findip(d);
    splay(x,0);
    return(splayT[splayT[x].son[0]].c+1);
}
```

寻找排名为k的点的值

只要找到这个点的位置，设为x。
把x旋转到根，根据二叉查找树的性质，左子树的所有元素都小于它，所以此时它的排名就是左子树大小加一。

```
int Getdk_Rank (int k)//寻找排名为k的点的值
{
    int x=root;
    while(1)
    {
        int lc=splayT[x].son[0],rc=splayT[x].son[1];
            if(k<=splayT[lc].c) x=lc;
        else if(k>splayT[lc].c+splayT[x].n) {k=k-splayT[lc].c-splayT[x].n;x=rc;}
        else break;
    }
    splay(x,0);
    return(splayT[x].d);
}
```

寻找一个数的前驱


```
int Getpre (int d)
{
    int x=findip(d); splay(x,0);

    if(splayT[x].d>=d && splayT[x].son[0]!=0)
    {
        x=splayT[x].son[0];
        while(splayT[x].son[1]!=0)x=splayT[x].son[1];
    }

    if(splayT[x].d>=d)x=0;
    return(splayT[x].d);
}
```

由于这个数不一定在树上，
所以要先用findip() 把和这个数大小相近的点找到。

如果此时得到的点的值小于读入的值，前驱就是得到的点的值了

如果大于等于，就要把找到的点旋转到根，然后去找左子树的最大值，由于findip是和读入的数最接近的点，所以此时在左子树中找到的点一定小于读入点，就是要求的前驱。
但如果又没有左子树，那就没有前驱了，返回0。

后继是类似的

区间操作

伸展树中伸展操作不会改变二叉树性质，
所以对二叉树的splay操作其实不会影响中序遍历二叉树产生的序列，
假设现有一个数字序列，我们用伸展树来维护它，并且中序遍历树所产生的序列就是该序列。

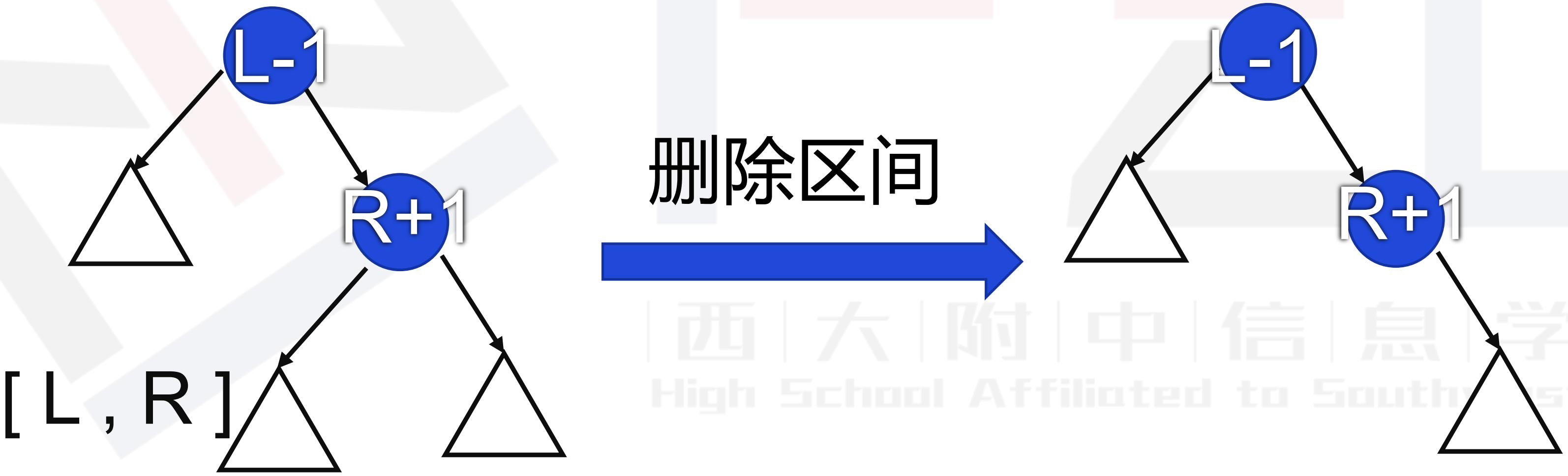
区间添加

在当前序列第 p 个元素之后，添加指定序列：
先把第 p 个元素伸展到根，再把第 p+1 个元素伸展到 x 的右子树的根，
再把需要插入的序列建成一棵子树，插入到第 p+1 个元素的左子树（原先为空）即可。

区间删除

删除当前序列第L个元素到第 R个元素：

只需要先把第 L−1 个元素伸展到根，再把第 R+1 个元素伸展到 L 的右子树的根，此时第 L 个元素到第 R 个元素的序列就会是第 R+1 个元素的左子树，删除即可。



区间翻转

逆序当前序列中第 L 个元素到第 R 个元素：

借助线段树中经典的标记：**lazy标记**，

先和区间删除一样，将第 L−1 个元素伸展到根，再把第 R+1 个元素伸展到 L 的右子树的根，

此时第 L 个元素到第 R 个元素的序列就会是第 R+1 个元素的左子树，

再令这棵子树的根节点的lazy = !lazy，若原先lazy = 0，则现在lazy = 1，代表需要翻转，

而当之后访问到该节点时，将该节点的lazy标记下放到左右儿子，并且交换左右儿子即可。