21级实验室暑假第七讲

目录

- 二叉搜索树
- Splay伸展树

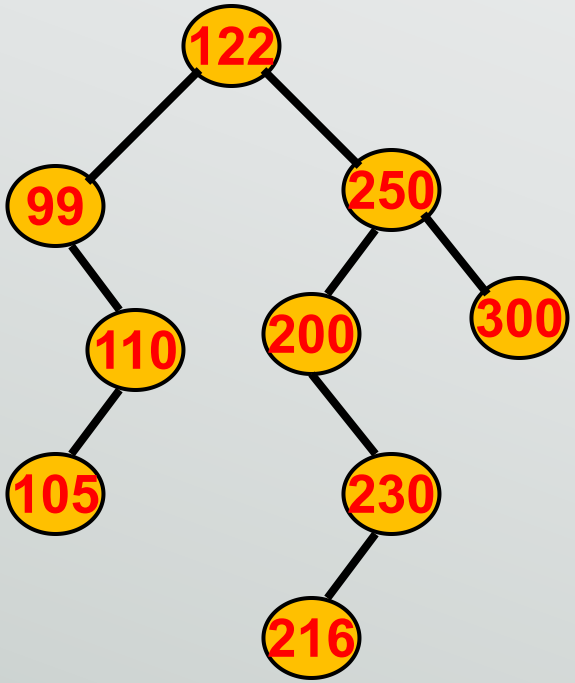
二叉搜索树

一棵非空的二叉搜索树满足以下特征：

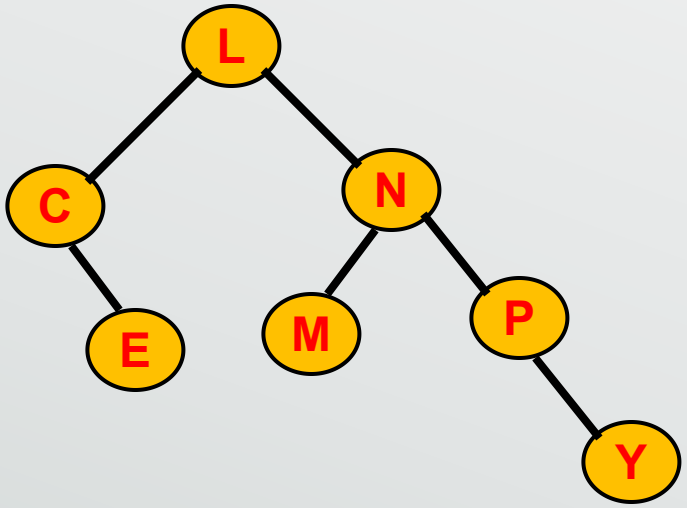
1. 每个结点都有一个作为搜索依据的关键码，所有结点的关键码互不相同。
2. 左子树（如果存在）上的所有结点的关键码均小于根结点的关键码。
3. 右子树（如果存在）上的所有结点的关键码均大于根结点的关键码。
4. 根结点的左右子树也都是二叉搜索树。

二叉搜索树(Binary Search Tree,简写为BST)又称为“二叉排序树”、“二叉查找树”、“二叉检索树”

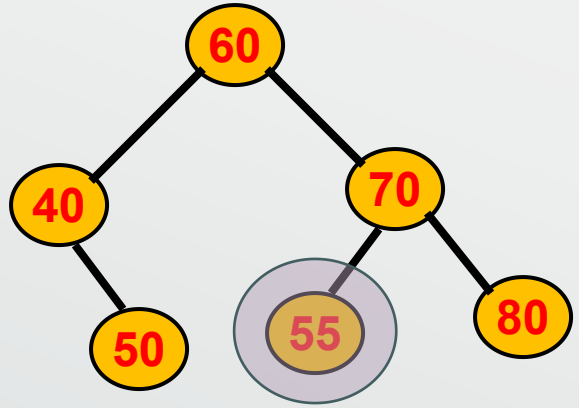
二叉搜索树



是二叉搜索树

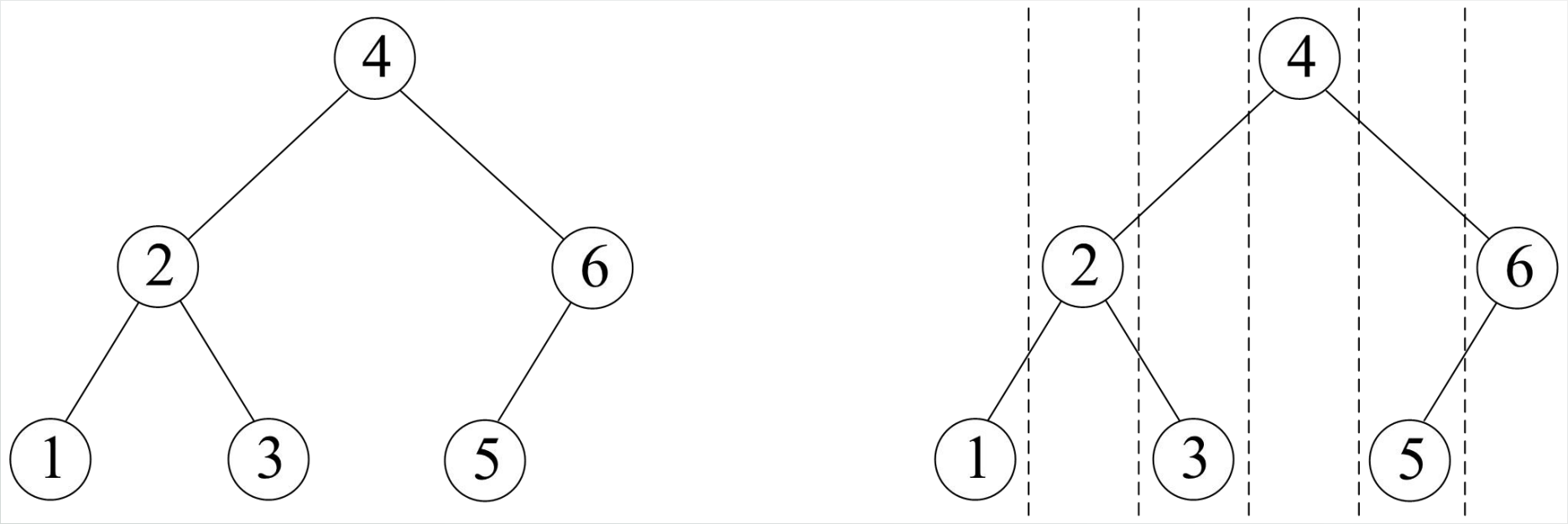


是二叉搜索树



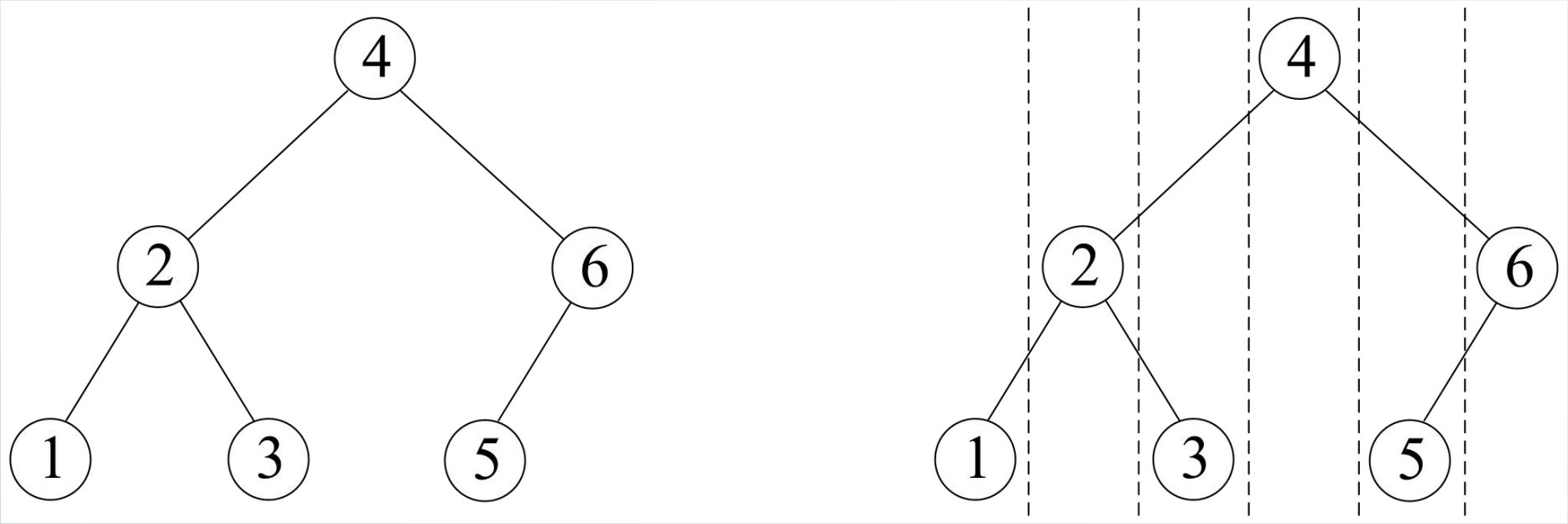
不是二叉搜索树

二叉搜索树



用中序遍历可以得到BST的有序排列。
右图的虚线把结点隔开，结点正好按从小到大的顺序被虚线隔开了。

二叉搜索树



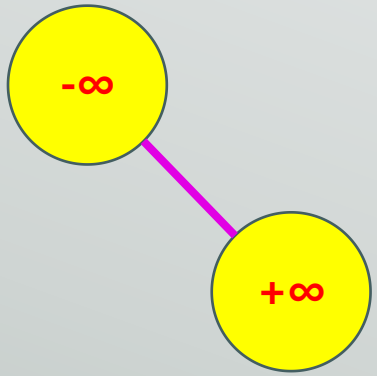
用中序遍历可以得到BST的有序排列。
右图的虚线把结点隔开，结点正好按从小到大的顺序被虚线隔开了。

二叉搜索树

BST的建立

为了避免越界，减少边界情况的特殊判断，我们一般在BST中额外插入**一个关键码为正无穷（一个很大的整数）和一个关键码为负无穷**的结点。仅有这两个结点构成的BST就是一棵初始化为空BST，如下图所示。

简便起见，在接下来的操作中，假设BST不会含有关键码相同的结点。此处以数组方式(亦可采用链表方式实现)实现。



注意: 如果用这种方式建树插入新的元素，此时会放置在正无穷的左子树下，看具体情况建树(可先插入第一个元素)，可**在中间过程单独添加哨兵(两个无穷)**

```
struct BST{
    int l,r;//左右子结点在数组中的下标
    int val;//结点关键码
}a[MAXN];//数组模拟链表
int tot,root,INF=1 << 30;

int New(int val){
    a[++tot].val=val;
    return tot;
}
void Build(){//适具体情况调用
    New(-INF),New(INF);
    root=1,a[1].r=2;
}
```

二叉搜索树

BST的检索　　在BST中检索是否存在关键码为val的结点。

设变量p等于根结点root，执行过程如下：

- 若 p 的关键码等于 val，则已经找到。

- 若 p 的关键码大于 val

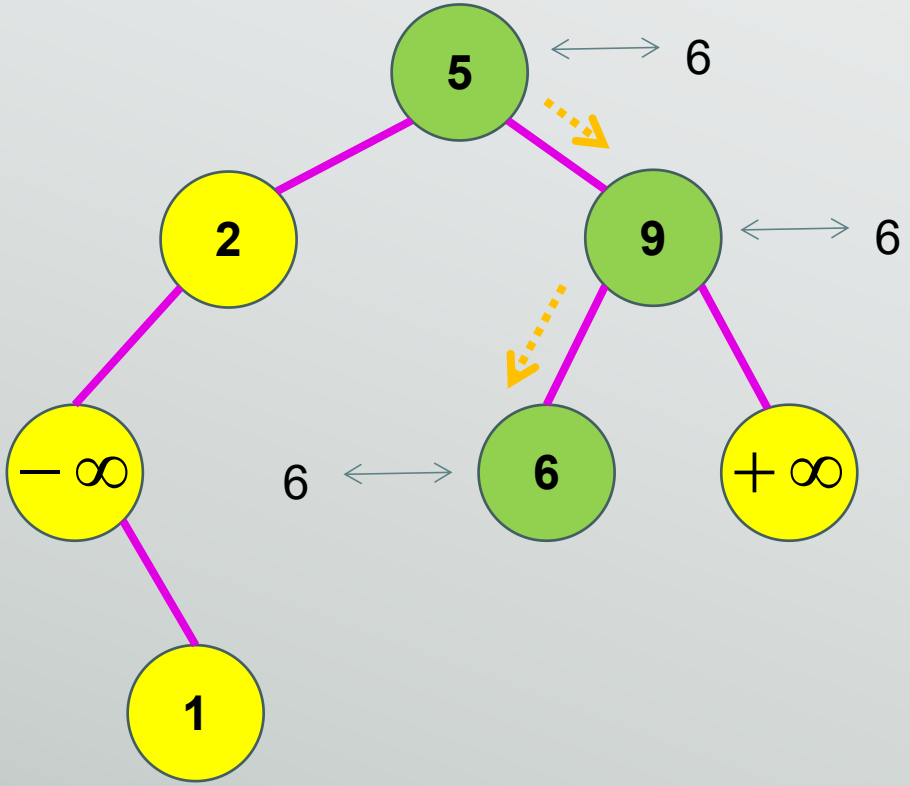
(1)若 p 的左子结点为空，则说明不存在 val。
(2)若 p 的左子结点不为空，在 p 的左子树中递归进行检索。

- 若 p 的关键码小于val

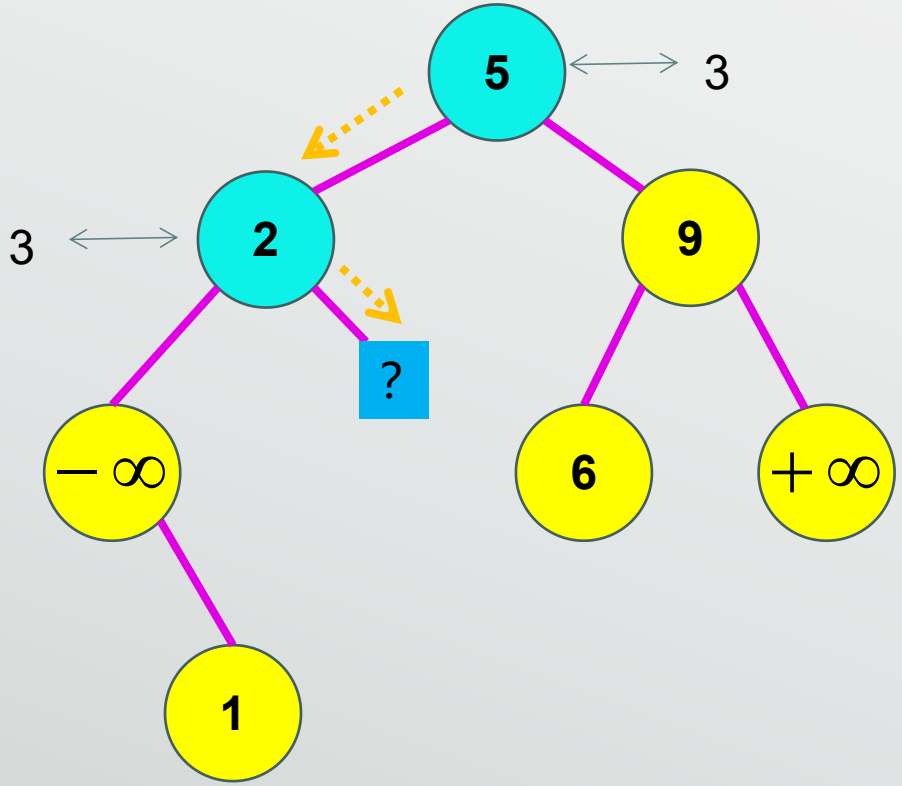
(1)若 p 的右子结点为空，则说明不存在 val。
(2)若 p 的右子结点不为空，在 p 的右子树中递归进行检索。

二叉搜索树

BST的检索　　**在BST中检索是否存在关键码为val的结点。**



查找6



查找3，不存在

二叉搜索树

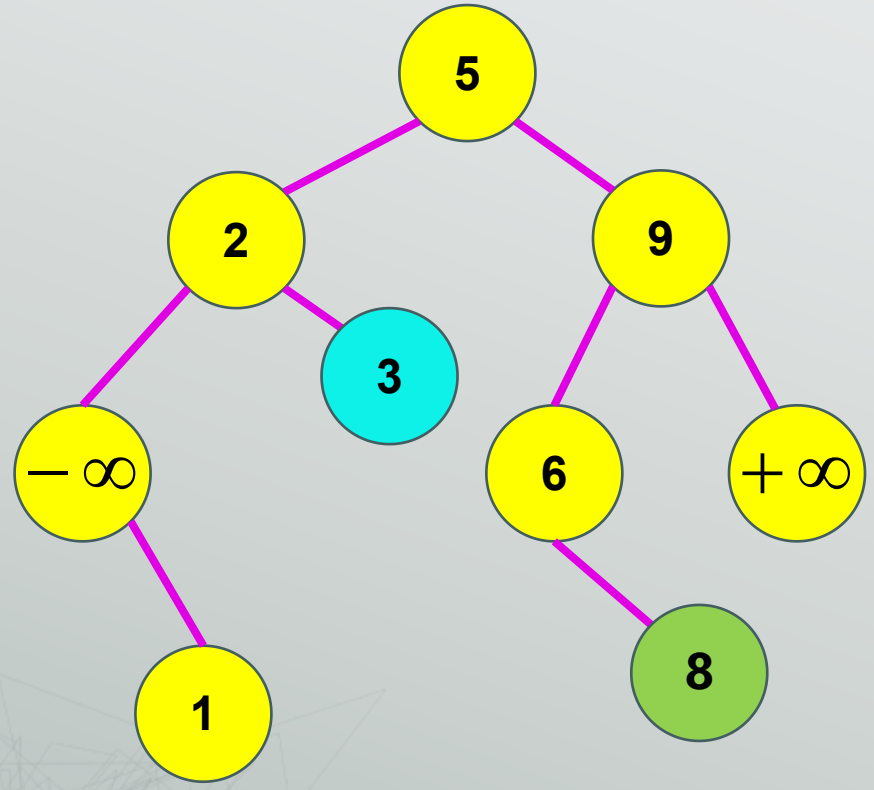
BST的检索　　**在BST中检索是否存在关键码为val的结点。**

```
int Get(int p,int val){
    if(p==0)//检索失败
        return 0;
    if(val==a[p].val)//检索成功
        return p;
    if(val<a[p].val)//遍历左子树
        return Get(a[p].l,val);
    else return Get(a[p].r,val);//遍历右子树
}
```

二叉搜索树

BST的插入

- 在BST中插入一个新的值val(假设目前BST中不存在关键码为val的结点)。
- 与BST的检索过程类似。
- 在发现要**走向的p的子结点为空，说明val不存在时，直接建立关键字为val的新结点**作为p的子结点。



插入3和8

```
void Insert(int &p,int val){
    if(p==0){
        p=New(val);
        //注意此处是引用,其父结点的l或r会被同时更新
        return ;
    }
    if(val==a[p].val)//存在相同元素,不作处理
        return ;
    if(val<a[p].val)//遍历左子树
        Insert(a[p].l,val);
    else Insert(a[p].r,val);//遍历右子树
}
```

二叉搜索树

BST求前驱/后继　　**以“后继”为例，val的后继指的是在BST中关键码大于val的前提下，关键码最小的结点。**

初始化ans为具有正无穷关键码的那个结点的编号。然后，在BST中检索val。在检索过程中，每经过一个结点，都检查该结点的关键码，判断能否更新所求的后继ans。

检索完成后，有三种可能的结果：

✓ 没有找到val.

此时val的后继就在已经经过的结点中，ans即为所求。

✓ 找到了关键码为val的结点，但p没有右子树。

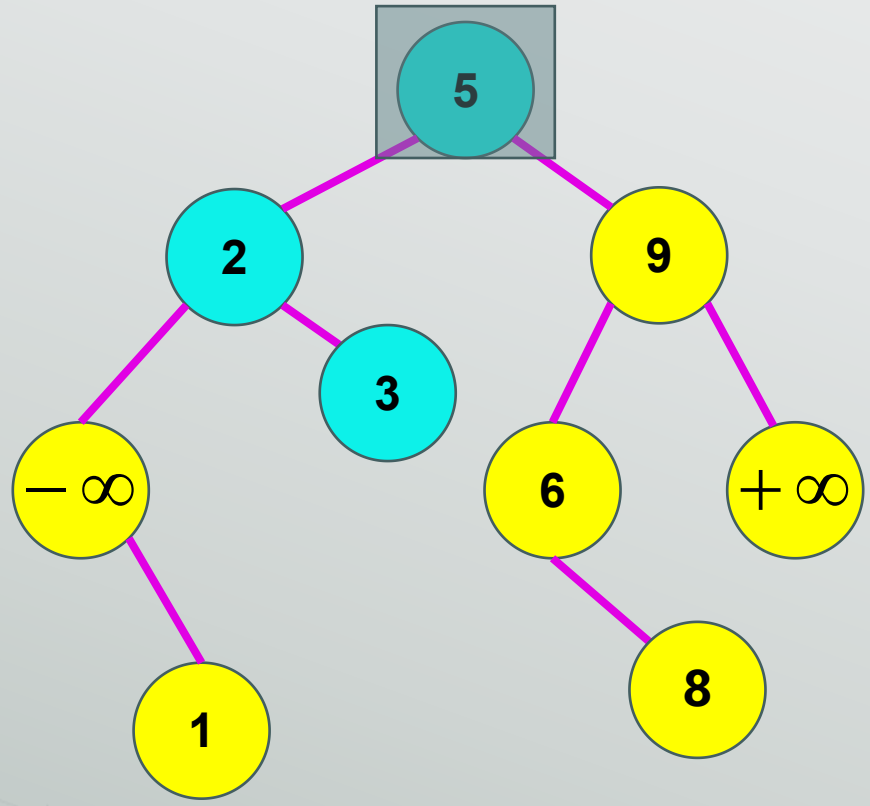
与上种情况相同，ans即为所求。

✓ 找到了关键码为val的结点，且p有右子树。

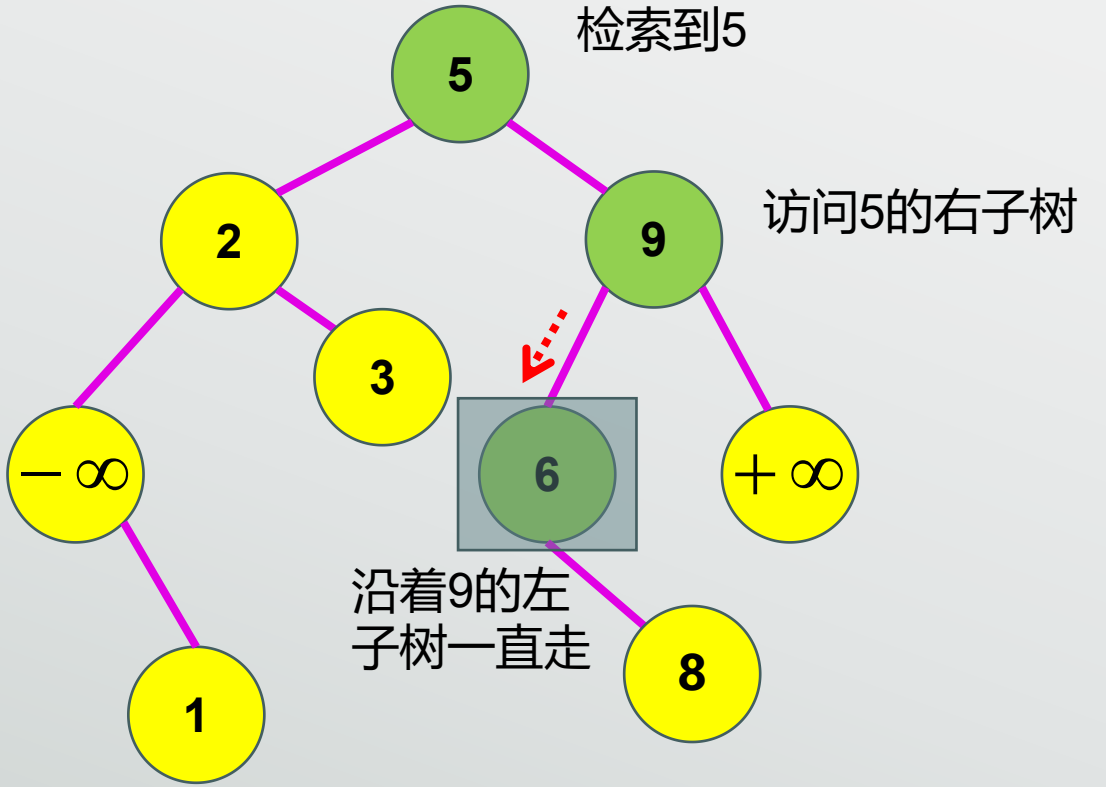
从p的右子结点出发，一直向左走，就找到val的后继。

二叉搜索树

BST求前驱/后继　　**以“后继”为例，val的后继指的是在BST中关键码大于val的前提下，关键码最小的结点。**



求3的后继



求5的后继

二叉搜索树

BST求前驱/后继

以“后继”为例，val的后继指的是在BST中关键码大于val的前提下，关键码最小的结点。

```
int GetNext(int val){
    int ans=2;//初始化为INF的编号
    int p=root;//从根结点开始枚举
    while(p!=0){
        if(a[p].val==val){//检索成功
            if(a[p].r>0){//有右子树
                p=a[p].r;
                while(a[p].l>0)//右子树一直向左走
                    p=a[p].l;
                ans=p;
            }
            break;
        }
        //每经过一个结点,都尝试更新后继
        if(a[p].val>val&&a[p].val<a[ans].val)
            ans=p;
        if(val<a[p].val)//检索过程,访问左子树
            p=a[p].l;
        else p=a[p].r;//检索过程,访问右子树
    }
    return ans;
}
```

二叉搜索树

BST求前驱/后继

求前驱

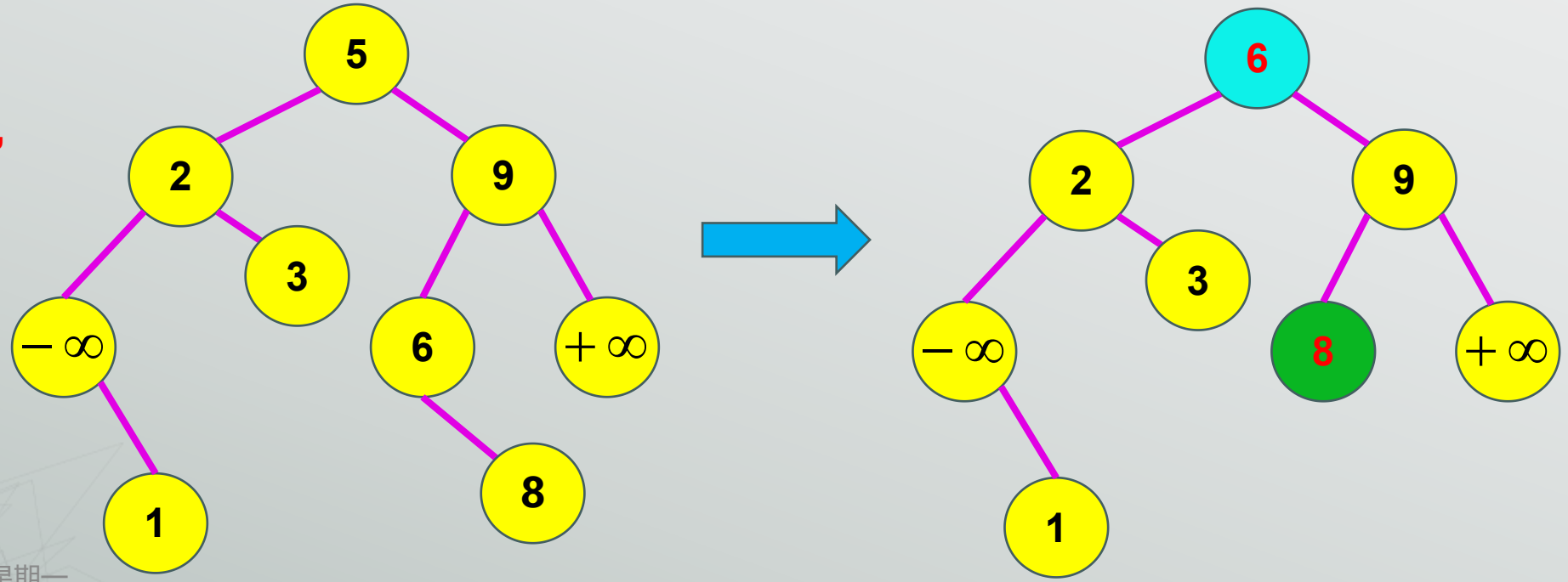
```
int GetPre(int val)
{
    int ans=1;//初始化为-INF的编号
    int p=root;//从根结点开始枚举
    while(p!=0){
        if(a[p].val==val){//检索成功
            if(a[p].l>0){//有左子树
                p=a[p].l;
                while(a[p].r>0)//左子树一直向右走
                    p=a[p].r;
                ans=p;
            }
            break;
        }
        //每经过一个结点,都尝试更新前驱
        if(a[p].val<val&&a[p].val>a[ans].val)
            ans=p;
        if(val<a[p].val)//检索过程,访问左子树
            p=a[p].l;
        else p=a[p].r;//检索过程,访问右子树
    }
    return ans;
}
```

二叉搜索树

BST的结点删除　　从BST中删除关键码为val的结点。

- ✓ 首先，在BST中检索val，得到结点p。
- ✓ 若p的子结点个数小于2，则直接删除p,并令p的子结点代替p的位置，与p的父结点相连。
- ✓ 若p既有左子树又有右子树，则在BST中求出val的后继结点next(由于next没有左子树)，所以可以**直接删除next,并令next的右子树代替next的位置**。最后再让**next结点替代p结点，删除p即可**。如下图所示。

删除5,
5的后继6是代替5,
6的右子树8代替6



二叉搜索树

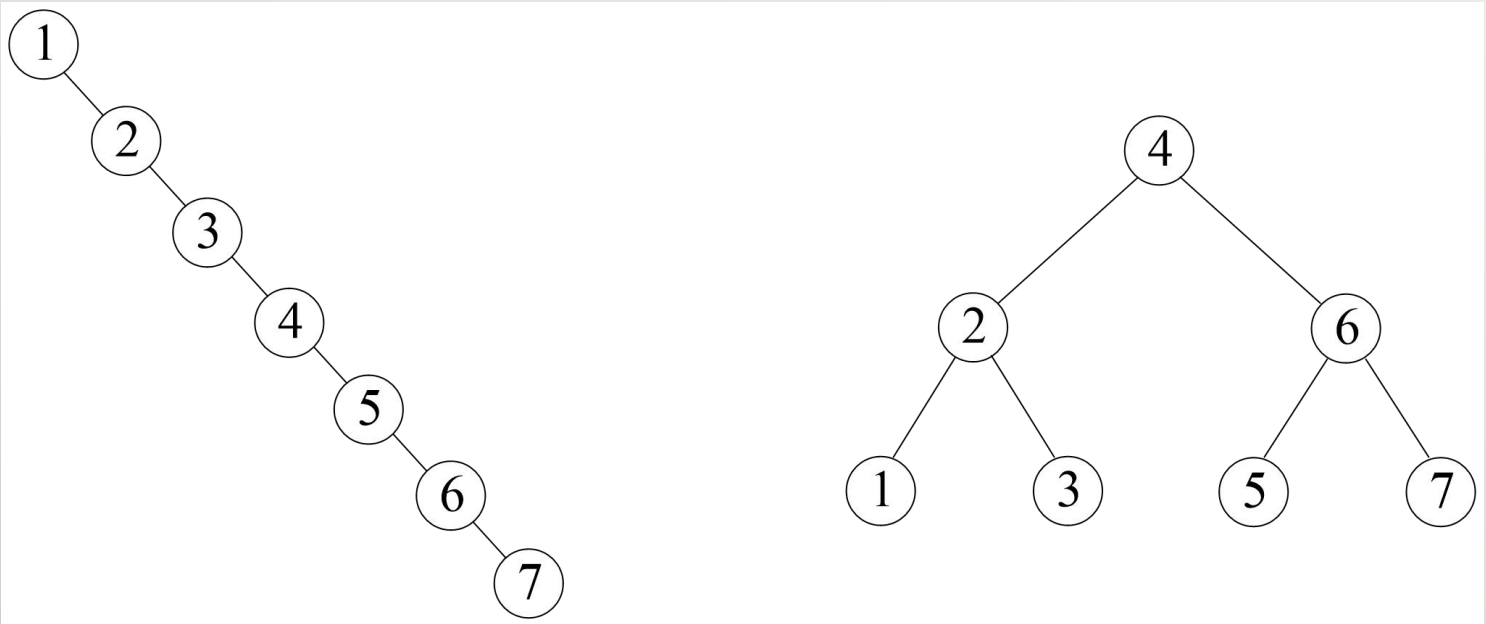
BST的结点删除

从BST中删除关键码为val的结点。

```
void Remove(int &p,int val){//从子树p中删除值为val的结点
    if(p==0)//检索失败
        return ;
    if(a[p].val==val){//已经检索到值为val的结点
        if(a[p].l==0)//没有左子树,有右子树
            p=a[p].r;//右子树代替p的位置,注意p是引用
        else if(a[p].r==0)//没有右子树,有左子树
            p=a[p].l;//左子树代替p的位置,注意p是引用
        else{//既有左子树又有右子树
            //求后继结点
            int next=a[p].r;
            while(a[next].l>0)
                next=a[next].l;
            //next一定没有左子树
            Remove(a[p].r,a[next].val);
            //让结点next代替结点p的位置
            a[next].l=a[p].l,a[next].r=a[p].r;
            p=next;//注意p是引用
        }
        return ;
    }
    if(val<a[p].val)//遍历左子树
        Remove(a[p].l,val);
    else Remove(a[p].r,val);//遍历右子树
}
```

二叉搜索树

在随机数据中，**BST一次操作的期望复杂度为O(logN)**。然而，BST很容易退化，例如在BST中依次插入一个有序序列，将会得到一条链，平均每次操作的复杂度为O(N)。我们称这种左右子树大小相差很大的BST是“不平衡”的。而右图是期望的BST，它很平衡。



什么是好的BST算法?

- BST的优劣，取决于它是否平衡。
- 如何实现一个平衡的BST？由于无法提前安排元素的顺序，所以只能在建树之后，通过动态调整，使得它变得平衡。
- BST算法的区别，就在于用什么办法调整。

BST算法有：AVL树、红黑树、Splay树、Treap树、SBT树等。

STL与BST。STL的set和map是用二叉搜索树（红黑树）实现的，检索和更新的复杂度是O(logn)。

Splay伸展树

- Splay树是一种BST树，它的查找、插入、删除等操作，复杂度都是O(logn)的。
- 最大的特点：可以把某个结点往上旋转到指定位置，特别是可以旋转到根的位置，成为新的根结点。
- 一个应用背景：如果需要**经常查询和使用一个数，那么把它旋转到根结点，下次访问它，只需要查一次就找到了**。

Splay伸展树

- Splay树与二叉查找树一样，也具有**有序性**。
  即Splay树中的每一个结点x都满足：该结点左子树中的每一个元素都小于x，而其右子树中的每一个元素都大于x。

- Splay树的核心思想就是通过Splay操作进行**自我调整**，从而获得平摊较低的时间复杂度。

Splay伸展树

结构体和一些函数的定义及含义

```
struct Tree{
    int fa;//fa表示该点的父结点
    int son[2];//son[0]表示该点的左儿子,son[1]表示该点的右儿子
    int cnt;//cnt表示该点在树上出现的次数
    int siz;//siz表示以该点为根的子树的大小
    int val;//val表示该点的权值
}tree[MAXN];

int root,tot;//分别表示根结点和总结点数
```

Splay伸展树

pushup函数

当前结点树的大小等于左子树的大小加上右子树的大小加上当前结点的个数，和**线段树**表示类似

$$tree[x].siz = tree[tree[x].son[0]].siz + tree[tree[x].son[1]].siz + tree[x].cnt$$

注：有的Splay和线段树同样存在懒标记和pushdown函数操作，此处也不作讲解，思想和线段树pushdown操作相同，时机出现也类似

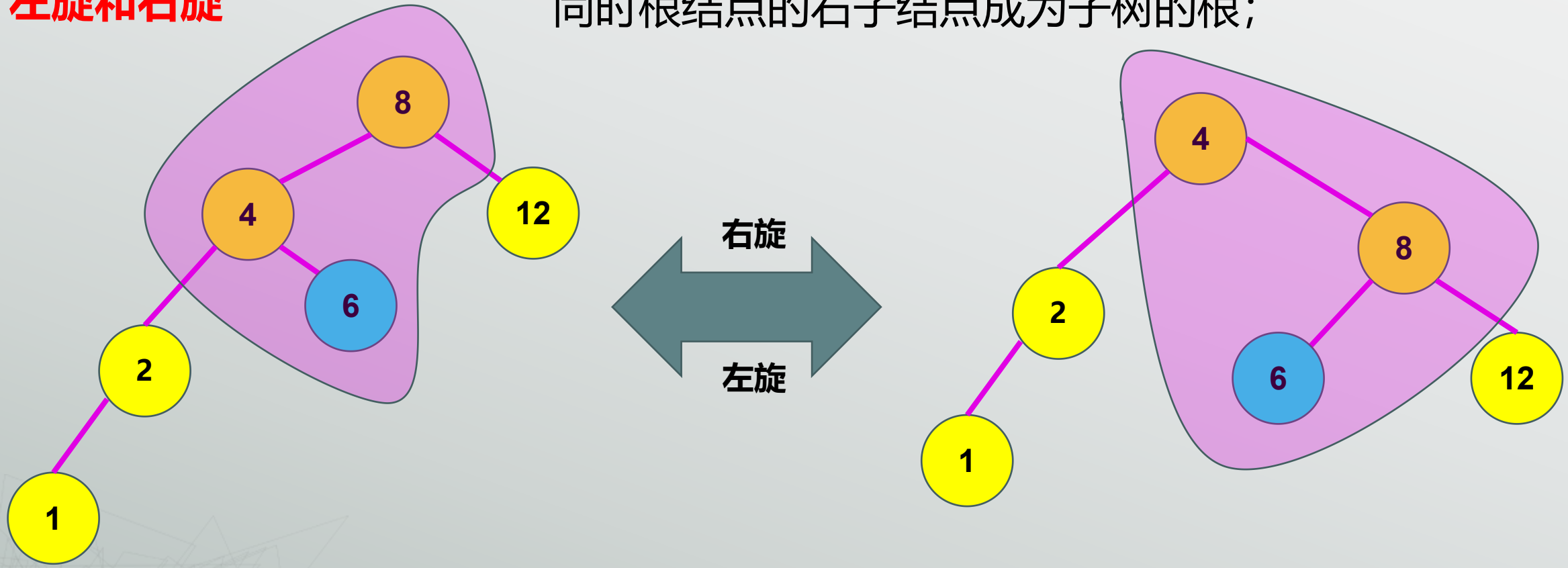
Splay伸展树

rotate旋转函数

右旋一个子树，会把它的**根结点旋转到根的右子树**位置，同时根结点的左子结点成为子树的根

左旋一个子树，会把**它的根结点旋转到根的左子树**位置，同时根结点的右子结点成为子树的根；

左旋和右旋



注意:旋转要保证BST的中序遍历不发生改变；

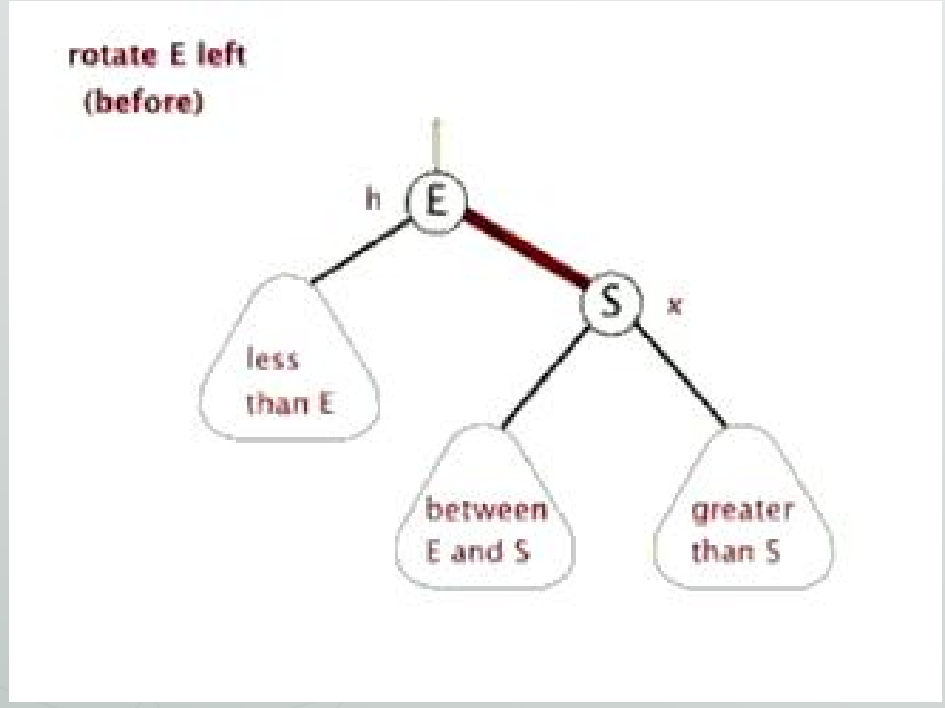
Splay伸展树

rotate旋转函数

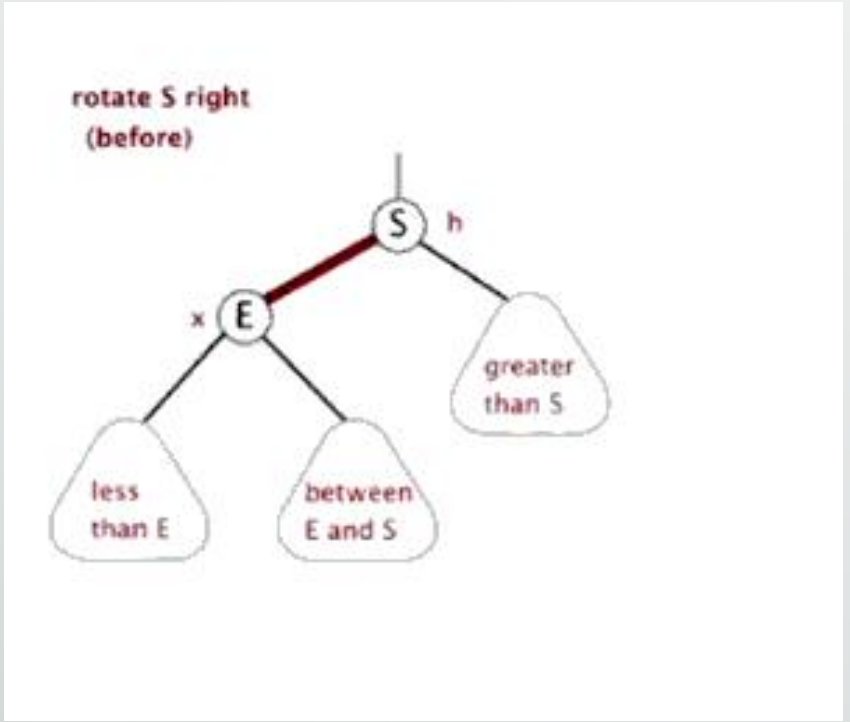
左旋和右旋

右旋一个子树，会把它的**根结点旋转到根的右子树**位置，同时根结点的左子结点成为子树的根

左旋一个子树，会把**它的根结点旋转到根的左子树**位置，同时根结点的右子结点成为子树的根；



左旋



右旋

Splay伸展树

rotate旋转函数

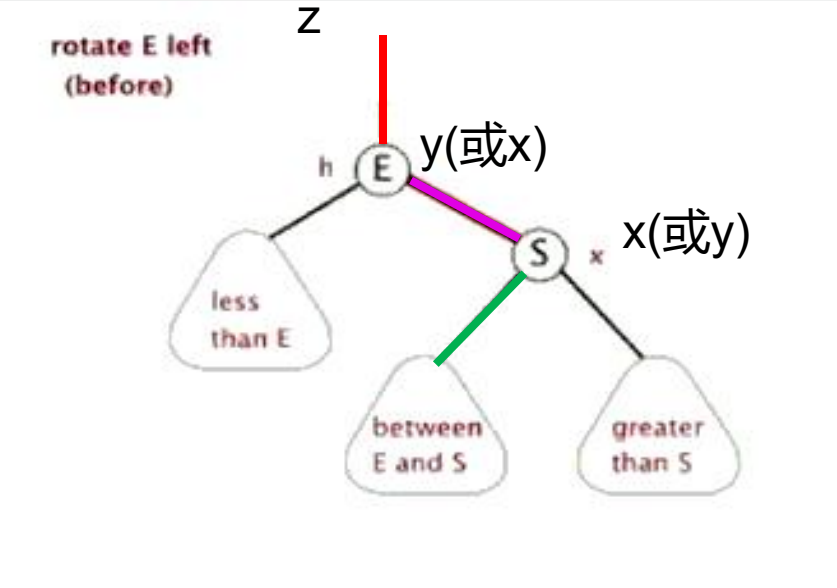
从上往下为左旋，从下往上为右旋，但存在共性，故可得到以下rotate步骤：

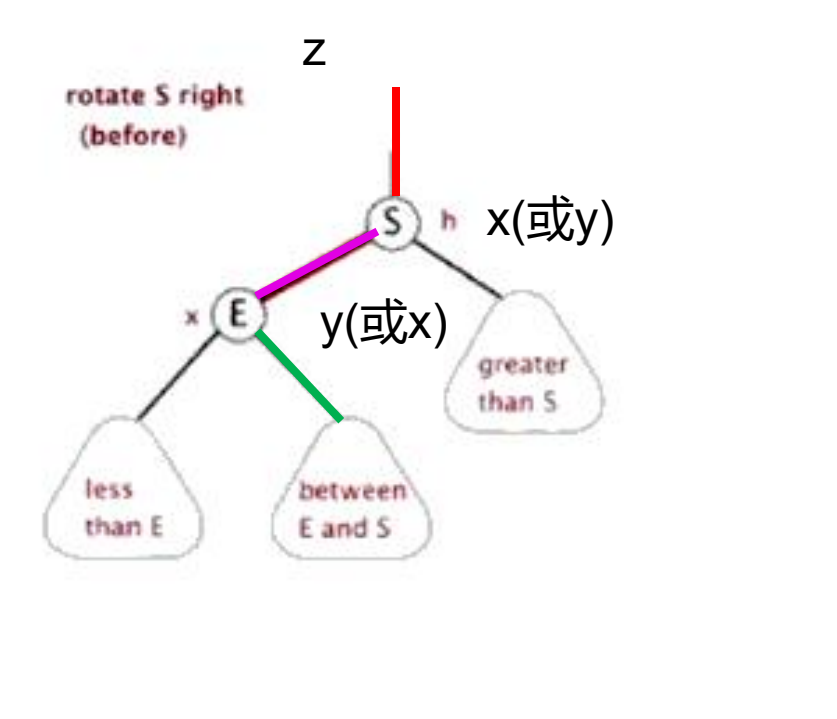
1：设旋转的结点为x点，其父结点为y点，祖父结点为z点；

2：设x是y的m儿子（m为0/1，代表左/右儿子，下同），y是z的n儿子；

3：把x的m^1儿子变为y的m儿子；
4：把y变为x的m^1儿子；
5：把x变为z的n儿子；

6：更新x,y的size值(先更新y，再更新x)





Splay伸展树

rotate旋转函数

```
void rotate(int x){//旋转操作(涵盖左旋和右旋)
    int y=tree[x].fa;//y是x的父结点
    int z=tree[y].fa;//z是y的父结点,即z是x的祖父结点
    int k=tree[y].son[1]==x;//判断x是y的哪一个儿子,k=0表示左儿子,k=1表示右儿子
    tree[z].son[tree[z].son[1]==y]=x;//z的原来y的位置变为x
    tree[x].fa=z;//x的父结点修改为z
    tree[y].son[k]=tree[x].son[k^1];
    //若x是y的左儿子(k=0),则让y的左儿子(k=0)等于x原来的右儿子(k^1)
    //若x是y的右儿子(k=1),则上y的右儿子(k=1)等于x原来的左儿子(k^1)
    tree[tree[x].son[k^1]].fa=y;//更新x原来的(k^1)儿子的父结点为y
    tree[x].son[k^1]=y;//更新x的(k^1)儿子为y
    tree[y].fa=x;//更新y的父结点x
    pushup(y),pushup(x);//上传操作
}
```

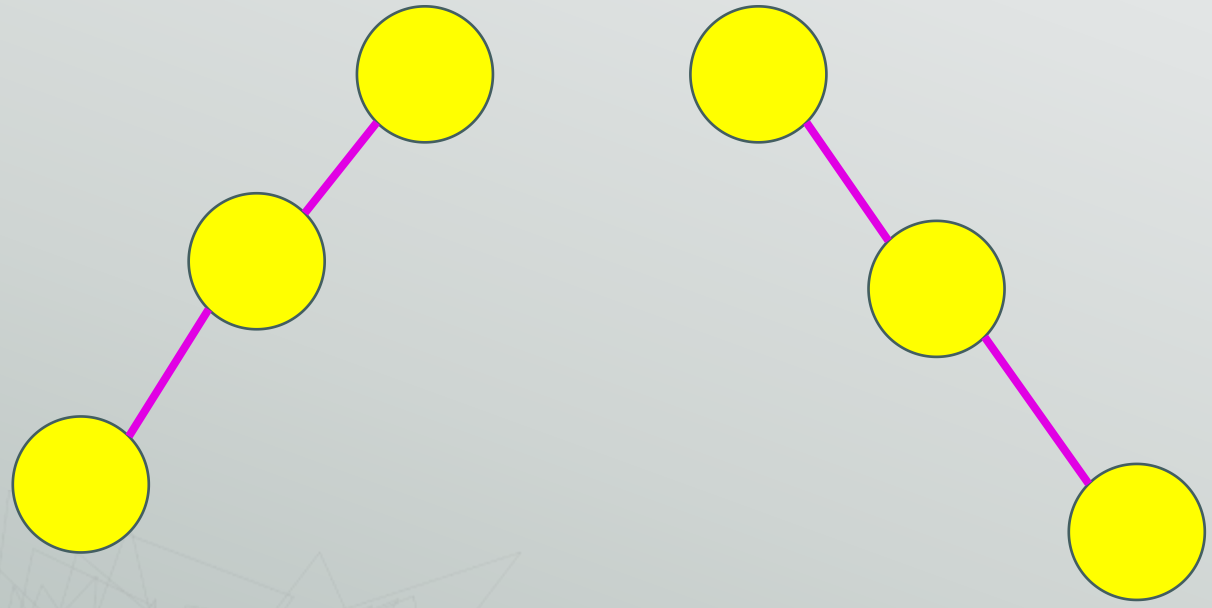
Splay伸展树

splay核心函数

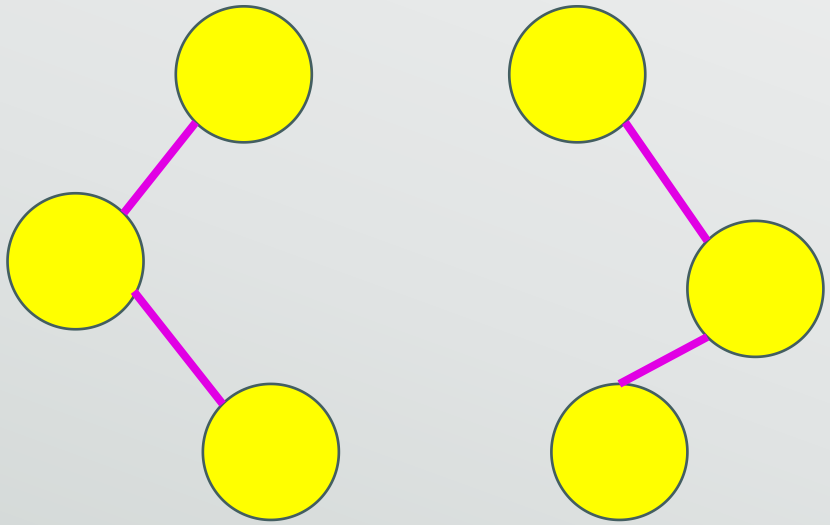
splay主要采用双旋的方式进行旋转

双旋就分为了两种情况，**直线型**旋转和**折线型**旋转

直线型



折线型



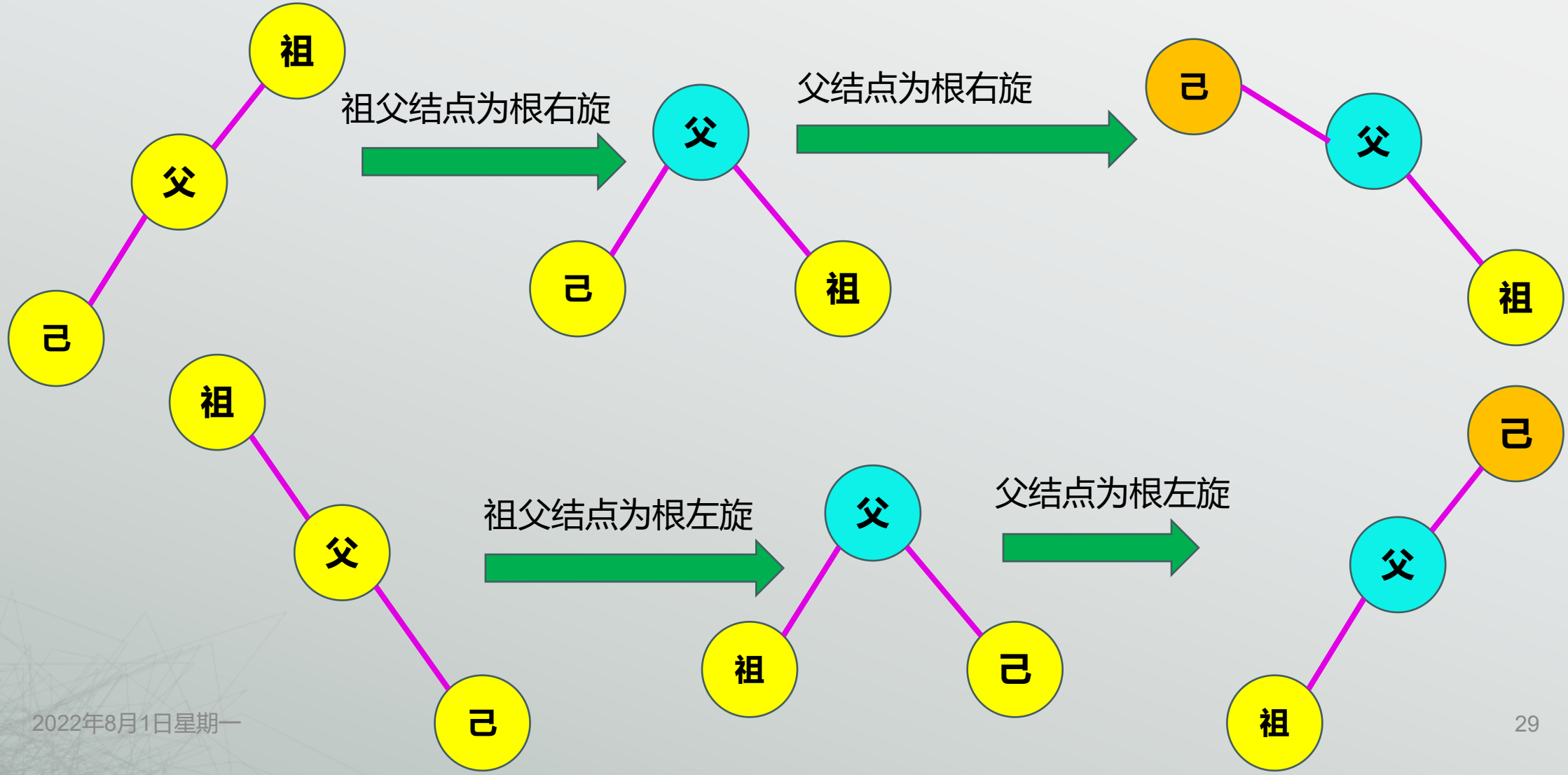
Splay伸展树

splay核心函数

直线型

如何让自己旋转到上层结点呢?

先旋转父结点，再旋转自己



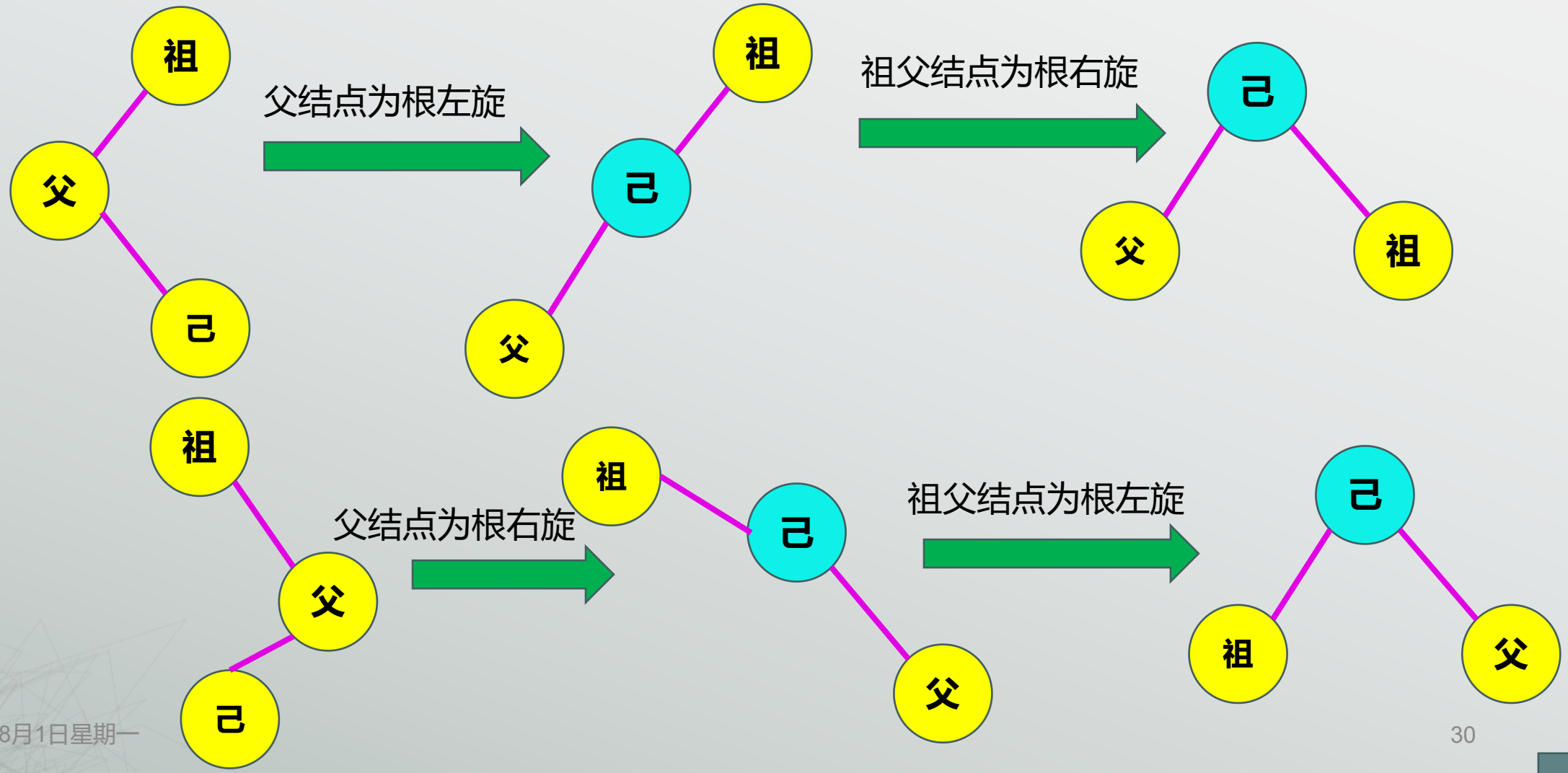
Splay伸展树

splay核心函数

折线型

如何让自己旋转到上层结点呢?

先旋转自己，再旋转自己



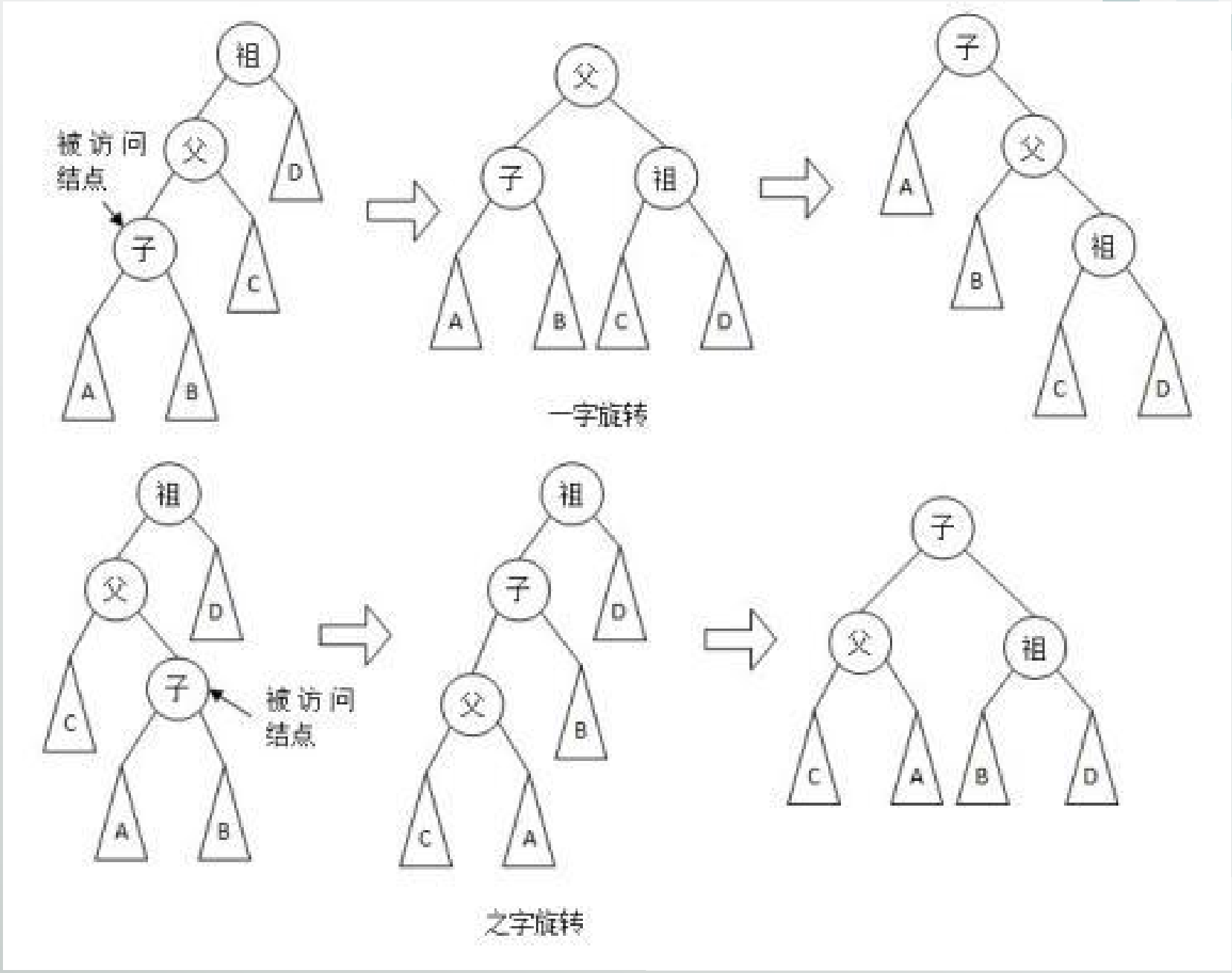
Splay伸展树

splay核心函数

Splay 本质是：把一个节点不断往上旋，直到到根（或者某目标节点的下方，即目标结点的儿子）

若为直线型，先旋转父结点，再旋转自己；

若为折线线，先旋转自己，再旋转自己



Splay伸展树

splay核心函数

Splay 本质是：把一个节点不断往上旋，直到到根（或者某目标节点的下方，即目标结点的儿子）

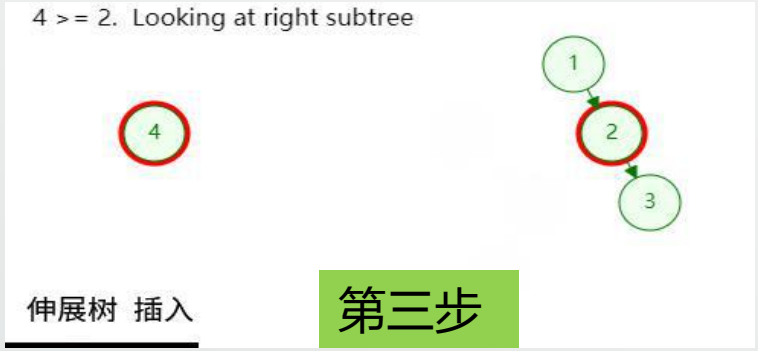
若为直线型，先旋转父结点，再旋转自己；

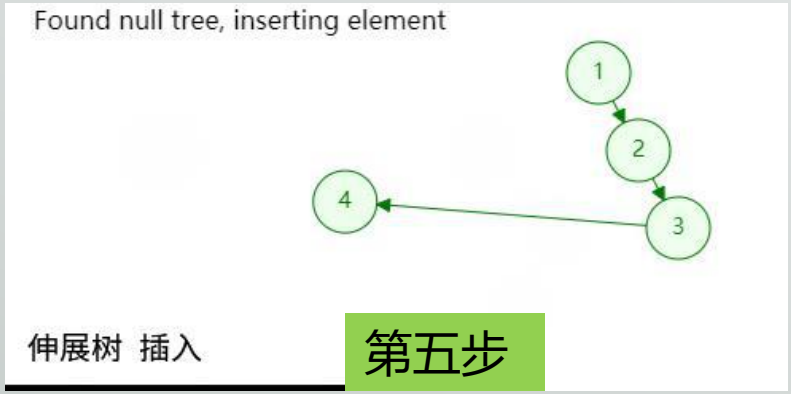
若为折线线，先旋转自己，再旋转自己

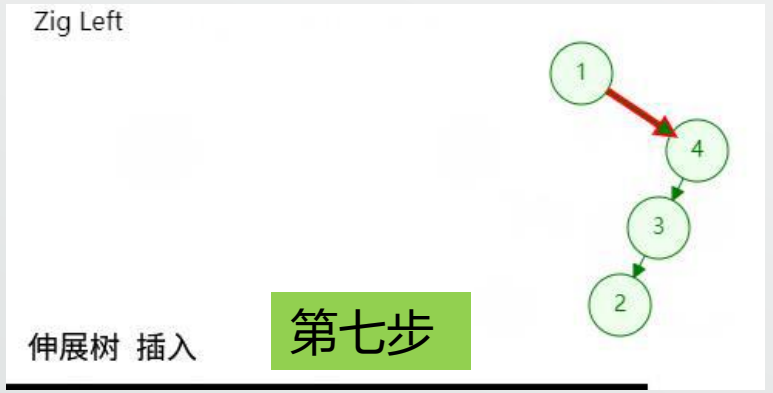
```
void splay(int x,int goal){//将x旋转为goal的儿子,如果goal是0,则旋转至根
    while(tree[x].fa!=goal){//一直旋转到x成为goal的儿子
        int y=tree[x].fa;//y是x的父结点
        int z=tree[y].fa;//z是y的父结点,即z是x的祖父结点
        if(z!=goal){//x的祖先结点z不是目标
            if((tree[y].son[1]==x)^(tree[z].son[1]==y))
      //两者异或结果为1,说明是折线形;反之直线型;此处亦可用!=来进行处理
                rotate(x);//折线型先旋转x
            else rotate(y);//共线型先旋转y
        }
        rotate(x);//(无论前面是什么类型)都再旋转x
    }
    if(goal==0)//如果goal是0,则将根结点更新为x
        root=x;
}
```

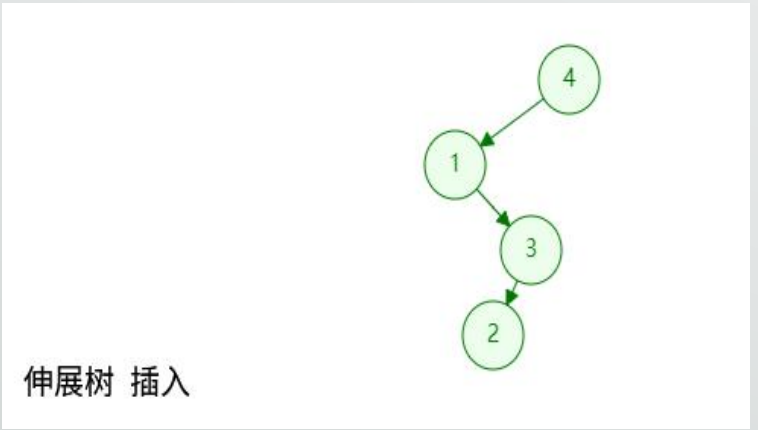
Splay伸展树

Insert插入函数





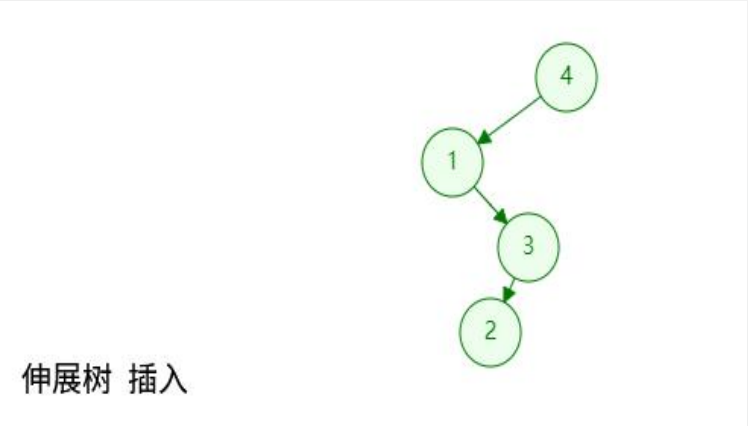




第八步 动图

Splay伸展树

Insert插入函数



步骤：
1. 从根结点出发，找到对应位置进行插入，若为空，生成新结点；若不为空在原来结点的cnt+1；
2. 将插入结点旋转至根(当下次访问它就能尽快找到)

Splay伸展树

Insert插入函数

```
void Insert(int x){
    int p=root,fa=0;//从根结点p出发,p的父结点为fa
    while(p!=0&&x!=tree[p].val){//当p存在并且没有移动到当前的值
        fa=p;//p向下走,fa变为p
        p=tree[p].son[x>tree[p].val];
       //若大于则在右子树查找,反之则在左子树查找
    }
    if(p!=0)//存在该值的位置
        tree[p].cnt++;//计数+1
    else{//不存在该值的位置,新建一个结点保存
        p=++tot;//新节点的位置
        if(fa!=0)//如果父结点非根
            tree[fa].son[x>tree[fa].val]=p;
        tree[p].son[0]=tree[p].son[1]=0;//新建的点不存在儿子
        tree[p].fa=fa;//更新父结点
        tree[p].val=x;//更新权值
        tree[p].cnt=1;//更新数量
        tree[p].siz=1;//更新以该点为根的子树的大小
    }
    splay(p,0);//把当前位置移到根,保证结构的平衡
}
```

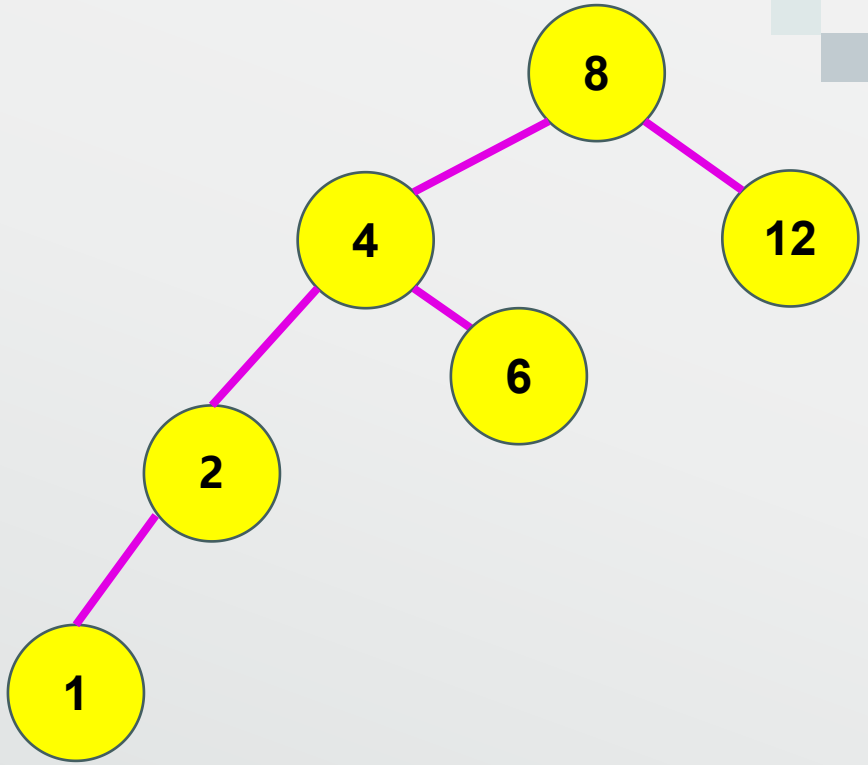
Splay伸展树

Find查找函数

步骤:
1. 从根结点出发，根据值的大小比较判断是往左走(x<tree[p].val)还是往右走(x>tree[p].val)直至x=tree[p].val

2. 将该点旋转至根

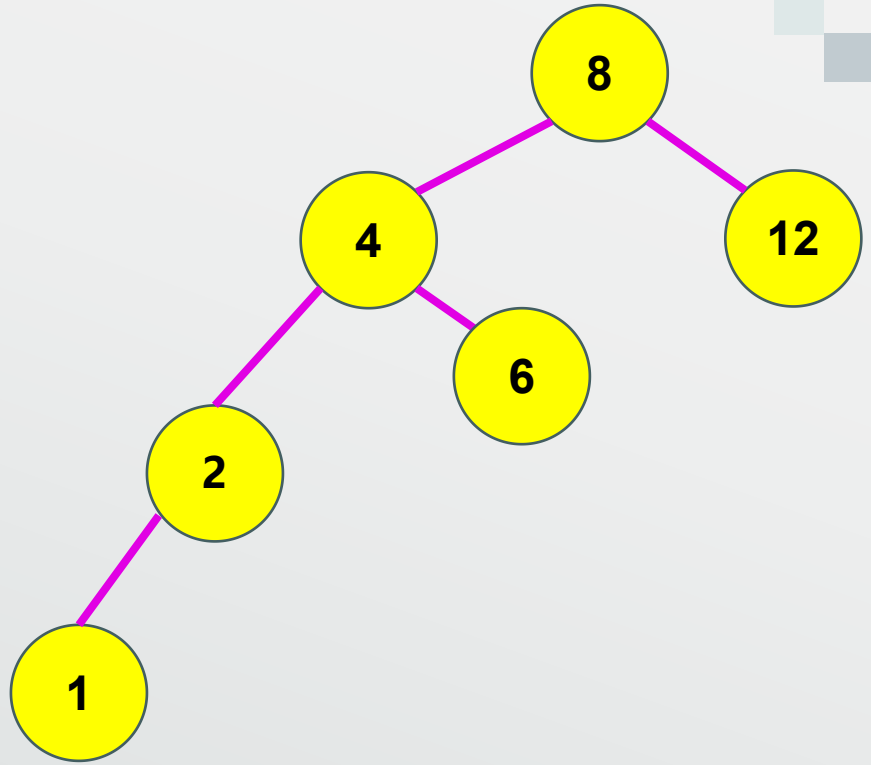
注: 如果x不在树上，则它会选择与之最相邻的值旋转至根，该值可能是x的前驱也可能是x的后继



Splay伸展树

Find查找函数

```
void Find(int x){//查找x的位置,并将其旋转至根结点
    int p=root;//从根结点出发
    if(p==0)//树是空树
        return ;
    while(tree[p].son[x>tree[p].val]&&x!=tree[p].val)
        p=tree[p].son[x>tree[p].val];
    //直到检索到x=tree[p].val停止,
    //否则如果x>tree[p].val,在右子树查找,如果x<tree[p].val在
左子树查找
    splay(p,0);
}
```



kth查找排名为k的位置函数

和**权值线段树的找排名**思想类似，判断排名x与当前结点p的左子树的大小关系，如果小于等于当前结点p的左子树的大小siz就**在当前结点p的左子树找**，反之如果大于当前结点p的左子树的大小siz与当前结点p的个数cnt的和，则**在当前结点p的右子树找同时更新排名x**，剩余情况**返回当前结点的位置p**

```
int kth(int x){//查询排名为x的位置
    int p=root;
    if(tree[p].siz<x)//当前树上无这么多数
        return -1;//不存在
    while(1){
        if(x<=tree[tree[p].son[0]].siz)//左儿子
            p=tree[p].son[0];
        else if(x>tree[tree[p].son[0]].siz+tree[p].cnt){//右儿子
            x-=tree[tree[p].son[0]].siz+tree[p].cnt;
            p=tree[p].son[1];
        }
        else return p;//当前结点
    }
}
```

Splay伸展树

Pre_Next查找前驱和后继函数

此处将前驱和后继函数合二为一
当state=1，查找后继，当state=0，查找前驱

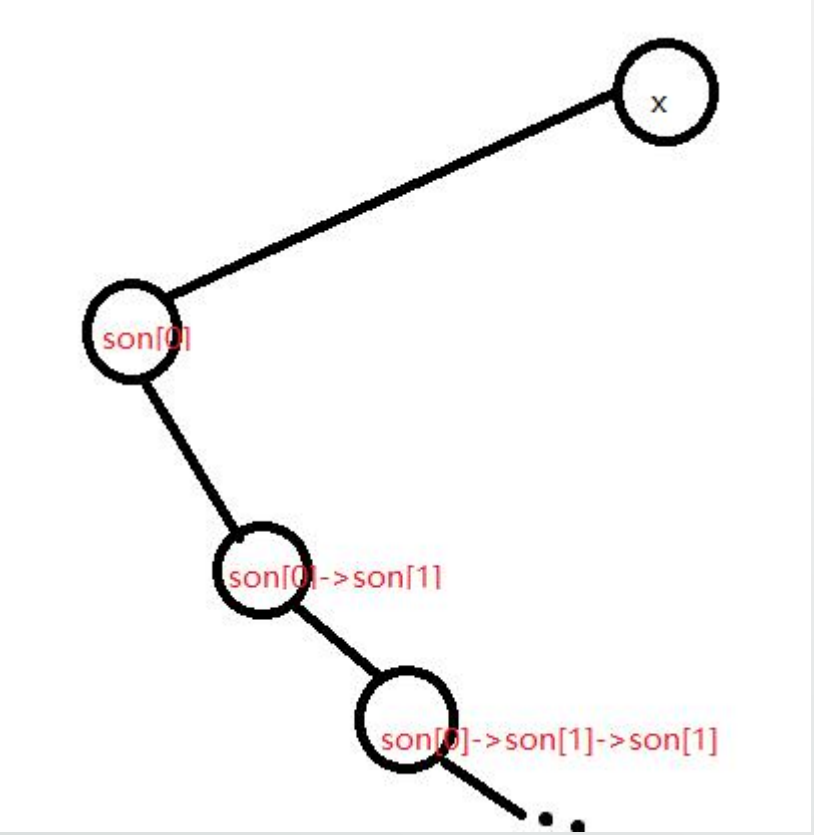
因为两者存在共性，可合并写法

首先调用find函数，使得当前查找元素旋转至根

查找前驱时，跳到当前元素的左儿子(state=0)，然后沿着它的右儿子(state^1=1)方向一直走

查找后继时，跳到当前元素的右儿子(state=1)，然后沿着它的左儿子(state^1=0)方向一直走

需要注意的是： 前面也提到，当树上无该元素，调用find函数会将最接近的值旋转至根，当前根结点可能是前驱或者是后继，故需要特判一下根结点是否是前驱或后继



以查询前驱的图为例

Pre_Next查找前驱和后继函数

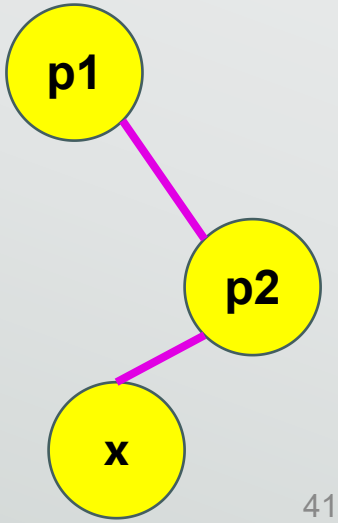
```
int Pre_Next(int x,int state){//查询前驱和后继合并,state=0(找前驱),state=1(找后继)
    Find(x);//将x旋转至根结点
    int p=root;//从根结点出发
    //由于x可能不在树上,find函数找到是最接近x的点,故root可能是x的前驱或者后继
    if(tree[p].val>x&&state==1)//p(root)为后继
        return p;
    if(tree[p].val<x&&state==0)//p(root)为前驱
        return p;

    p=tree[root].son[state];//前驱到达它的左儿子,后继到达它的右儿子

    //前驱到达左儿子然后沿着该左儿子的右边一直走
    //后继到达右儿子然后沿着该右儿子的左边一直走
    while(tree[p].son[state^1]!=0)
        p=tree[p].son[state^1];
    return p;
}
```

Splay伸展树

Remove删除函数

- 分别找到**x的前驱p1和后继p2**，那么在当前树上就满足**p1<x<p2并且中间没有其它数**

- 将**前驱p1旋转到根**，此时所有值比p1大的都在右子树
- **把后继p2旋转到前驱p1的右儿子处**，此时p2的左儿子就是x且只有一个，因为p2的左子树要满足>p1，<p2，显而易见因为定义这里面只能插x
- 对p2的左子树进行操作即可，如果当前元素的个数=1，让p2的左儿子等于空即可，否则当前元素的个数-1，并将其旋转至根



Splay伸展树

Remove删除函数

```
void Remove(int x){//删除x
    int last=Pre_Next(x,0);//查找x的前驱
    int nex=Pre_Next(x,1);//查找x的后继
    splay(last,0),splay(nex,last);
    //将前驱旋转到根结点,后继旋转到根结点的下面
    //此时后继是前驱的右儿子,x是后继的左儿子,并且x是叶子结点
    int k=tree[nex].son[0];//后继的左儿子
    if(tree[k].cnt>1){//如果个数超过一个
        tree[k].cnt--;//计数-1
        splay(k,0);//旋转
    }
    else tree[nex].son[0]=0;//这个结点直接丢掉(不存在)
}
```

Splay伸展树

中序遍历函数　　和二叉树的中序遍历方式相同

遍历顺序是左根右

哨兵

一般插入-inf和inf作为哨兵避免越界处理