信息学

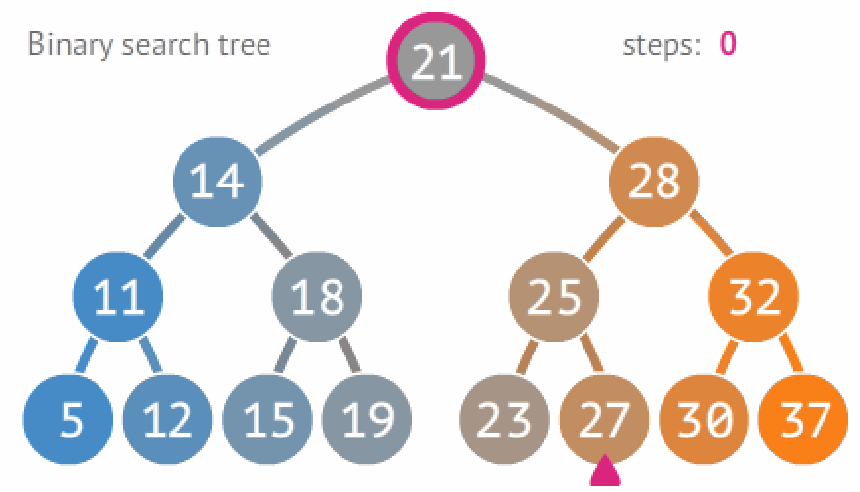
二叉查找树 BST

二叉查找树（简称BST），是满足以下条件的二叉树：

① 树上每一个节点都有一个权值;

② 对于树上任意一个节点u，若左子树不为空，则左子树上所有节点权值均不大于u的权值;

③ 对于树上任意一个节点u，若右子树不为空，则右子树上所有节点权值均不小于u的权值;



BST 中序遍历结果与特点?

遍历单调递增输出的结点值

西大附中信息学竞赛
High School Affiliated to Southwest University

```
struct BST
{
        int l,r;            //左右子节点的编号
        int val;            //节点权值
}s[N];                      //数组模拟链表
int tot,root=1;             //tot记录当前节点数目，root为根节点
int INF=0x7fffffff;         //int的最大值
```

```
//主函数main()中
a[++tot].val=INF;   //添加一个无穷大的节点，就不需要判断BST是否为空，方便操作
```

查找

任意一子树根结点键值>=其左子树中的任意键值（点权）
任意一子树根结点键值<其右子树中的任意键值（点权）

- 假如现在已经建好一颗BST
- 现在你要从中找到值x，考虑值如何实现?
  - 类似二分答案,递归实现即可。

```
int Find(int p,int x)     //当前节点为p，查找值为x的节点
{
            if(p==0)    return 0;                    //节点为空，查找失败
            if(a[p].val==x)           return p;     //查找成功，返回x的位置
            return a[p].val<x ? Find(a[p].r,x) : Find(a[p].l,x);
}
```

设变量p等于根节点root

(1) 若a[p].val==x，则查找成功。

(2) 若a[p].val>x

① 若p的左子节点为空，查找失败，说明不存在。

② 若p的左子节点不为空，则继续在p的左子树中递归查找。

(3) 若a[p].val<x

① 若p的右子节点为空，查找失败，说明不存在。

② 若p的右子节点不为空，则继续在p的右子树中递归查找。

查找

任意一子树根结点键值>=其左子树中的任意键值（点权）
任意一子树根结点键值<其右子树中的任意键值（点权）

- 假如现在已经建好一颗BST
- 现在你要从中找到值x，考虑值如何实现？
  - 类似二分答案,递归实现即可。

```
int Find(int p,int x)     //当前节点为p，查找值为x的节点
{
        if(p==0)    return 0;                  //节点为空，查找失败
        if(a[p].val==x)          return p;     //查找成功，返回x的位置
        return a[p].val<x ? Find(a[p].r,x) : Find(a[p].l,x);
}
```

当然你也可以非递归实现。

```
int Find(int p,int x)     //当前节点为p，查找值为x的节点
{    while(p){
            if(a[p].val ==x) return p;
            if(a[p].val < x) p=a[p].l;
            if(a[p].val > x) p=a[p].r;
     }
     return -1;
}
```

找最值

任意一子树根结点键值>=其左子树中的任意键值（点权）
任意一子树根结点键值<其右子树中的任意键值（点权）

- 假如现在已经建好一颗BST
- 现在你要从中找到最小值，考虑值如何实现？

肯定在左子树上

从根节点root开始，若左子节点不为空，访问左子节点，直到左子节点为空。

```
int Find_min(int p)
{
        if(p) while(a[p].l)      p=a[p].l;
        return a[p].val;         //返回最小值
}
```

插入

BST中插入一个权值x，其实就是在BST中**查找权值为x的节点**：

- 当权值x的节点存在时，添加一个域，记录重复个数；
- 当权值x的节点不存在时，即找到了空节点，插入到空节点。

```
void Insert(int &p,int x)                    //插入权值x，当前结点为p
{           //这里p是引用
        if(p==0)
        {
                a[++tot].val=x;                  //添加一个新的节点
                p=tot;                           //p的父节点的l或r值会被同时更新，指向节点tot
                return;
        }
        if(x==a[p].val)        return;           //已经存在，不插入，根据题意，也可以记录重复的个数
        return x<a[p].val ? Insert(a[p].l,x) : Insert(a[p].r,x);
}
```

扩展：引用

引用：

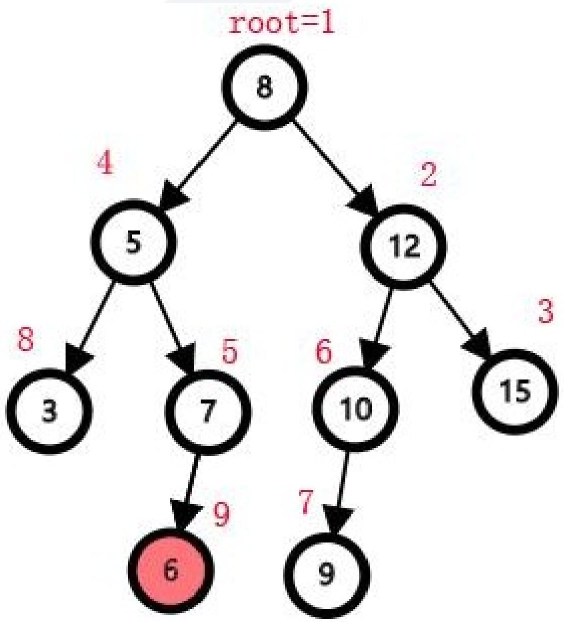
这里的&表示引用，引用在程序中相当于一个变量的**别名**，代表的都是同一个变量，只是名称不相同而已(类似每个人都有一个外号、小名等)，对别名进行操作等同于对原变量进行操作，例如：

```
int a=10;
int &b=a;  //b是a的别名
b+=2;
cout<<a;   //输出答案为12
```

西 | 大 | 附 | 中 | 信 | 息 | 学 | 竞 | 赛
High School Affiliated to Southwest University

例如在BST中插入权值6：

设初始插入的权值顺序依次为8、12、15、5、7、10、9、3，对应的节点编号依次为1~8.



程序执行流程：

(1) 首先调用函数Insert(root,6)，root的别名为p（root为根节点，root=1），接着递归调用函数Insert(a[root].l,6)，

(2)a[root].l的别名为p，a[root].l等于4，即执行函数Insert(4,6)，接着递归调用函数Insert(a[4].r,6)；

(3)a[4].r的别名为p，a[4].r等于5，即执行函数Insert(5,6)，接着递归调用函数Insert(a[5].l,6)；

(4)a[5].l的别名为p，p为空，添加一个新的节点，**执行p=tot，相当于a[5].l=tot**。

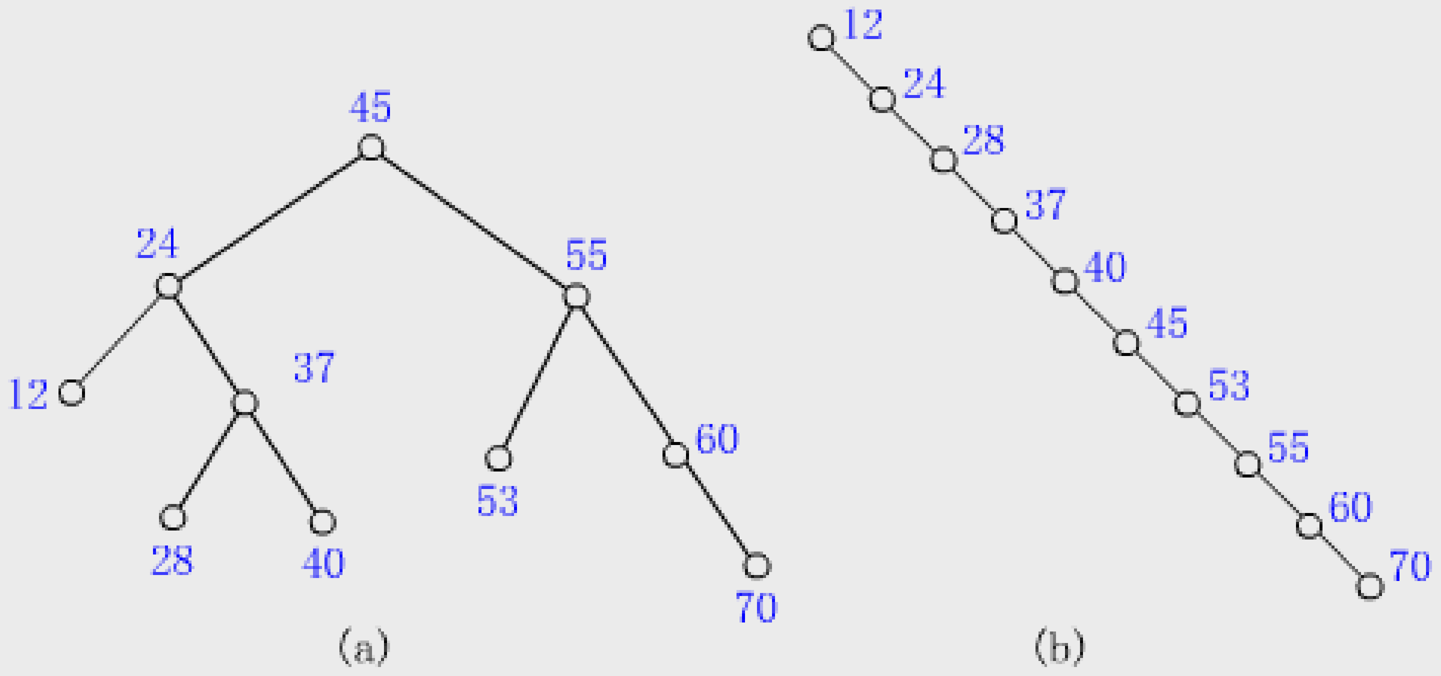
这样，添加一个节点的同时，可以同时更新其父节点的l或r。

插入

BST的生成为进行不断插入的过程!!

但在生成BST的时候,可能会由于根结点选择不好,使得树很斜,查找的效率降低,

可以使用随机产生根结点的方法,使得BST较平衡,下图就是两棵关键字相同的BST树.

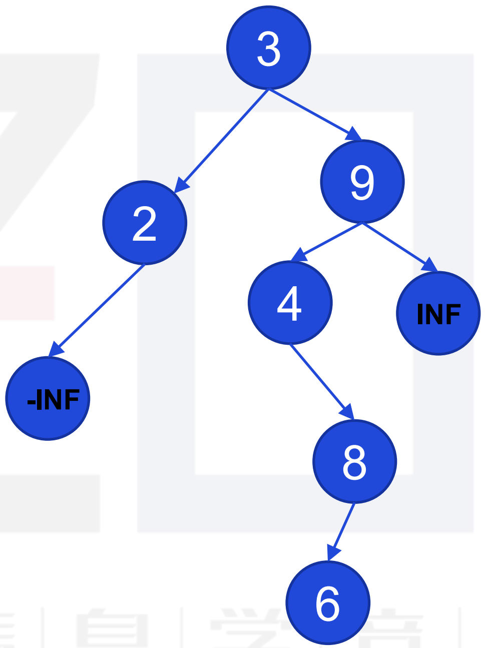


由同一组关键字构成的两棵形态不同的二叉排序树

寻找前驱/后继

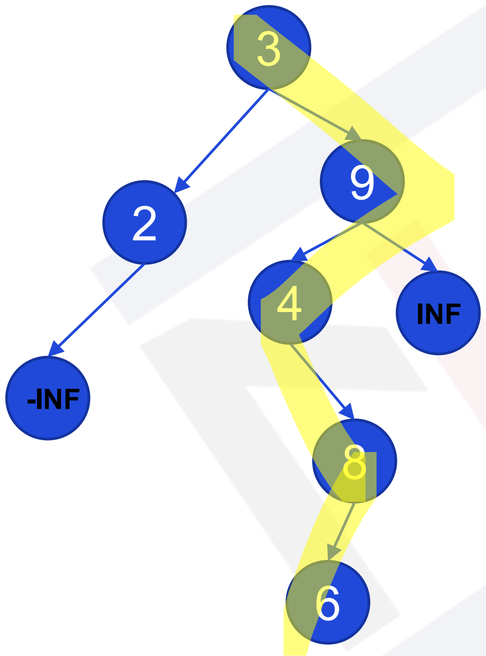
什么是v（v：权值）的前驱?
点权小于v的权的情况下，最大的那个点权。

什么是v（v：权值）的后驱?
点权大于v的权的情况下，最小的那个点权。
例如，左图，3的后驱是4



一种暴力求法
中序遍历（一般不这样用）

寻找前驱/后继



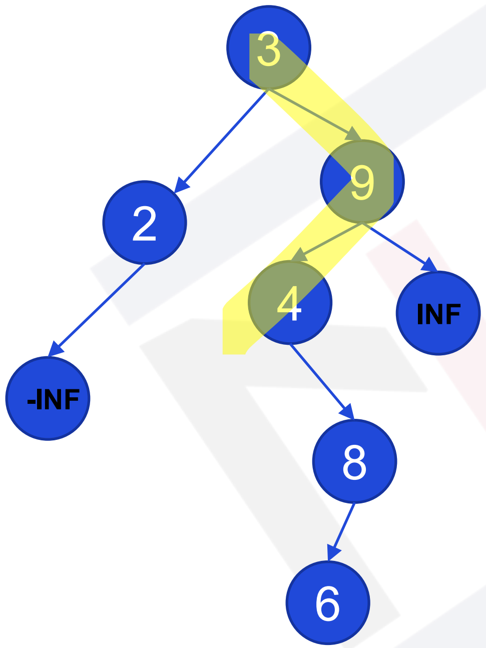
基于学过的操作尝试思考，如何寻找7的后继

find(7)
虽然失败，但路径上一定有7的后继，why?

Case1:find 失败，答案在路径上，打擂台求ans
初始化ans=INF

if(路过的值>v && 值>ans)
ans =值

寻找前驱/后继

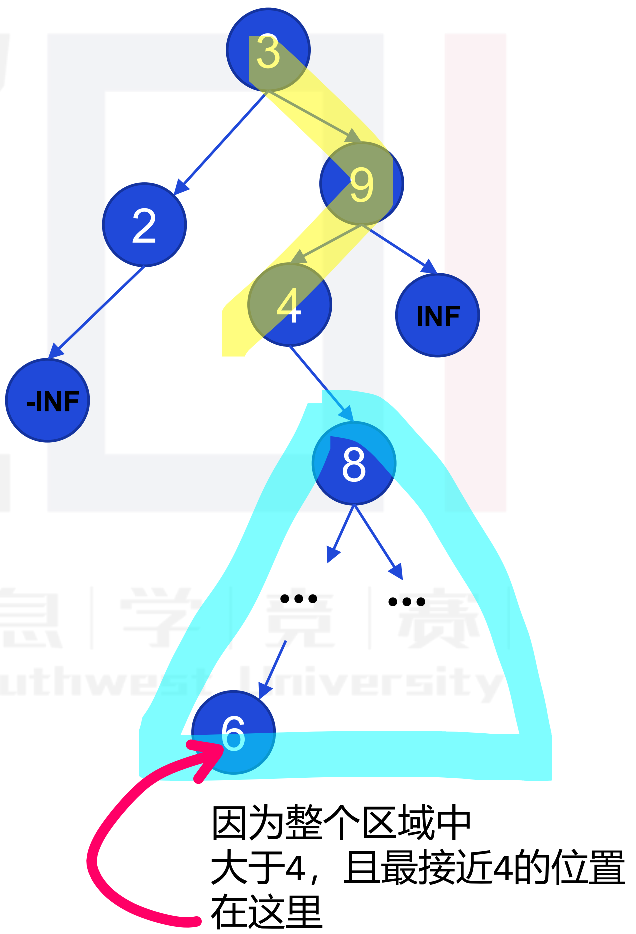


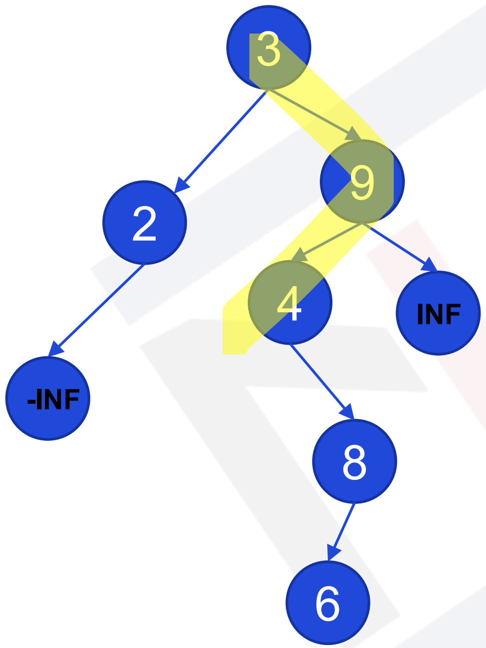
find(4) ?
查找成功，4的后继是?
6

Case: 2 如果find成功
且该点还有右子树。

那后继在哪里?
其后继一定在其右节点的
最左节点上。

解释为什么：





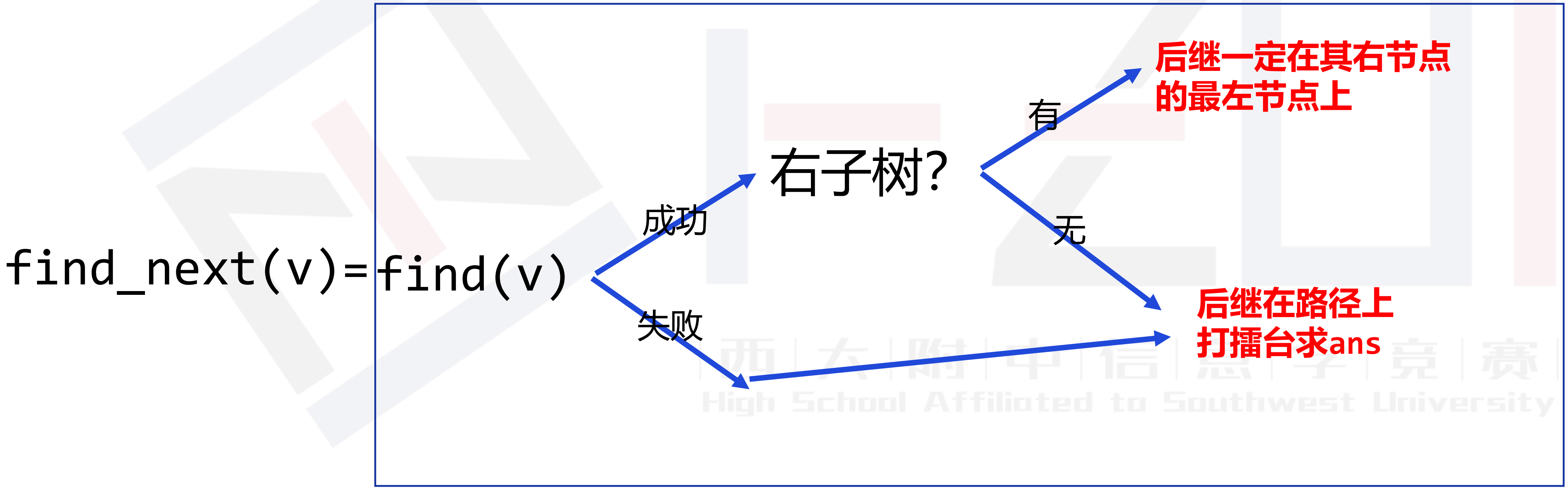
find(8) ?
查找成功，8的后继是?
9

Case: 3 如果find成功且该点没有右子树。

与Case1类似ans在路径上

西大附中信息学竞赛 High School Affiliated to Southwest University

寻找前驱/后继


find_next(v)=find(v)
成功
失败
右子树?
有
无
后继一定在其右节点的最左节点上
后继在路径上打擂台求ans

寻找前驱/后继

见书P235页代码。
注意，本PPT查找为find
而蓝书是Get

求前驱的操作与求后驱的分析过程类似/求解方法对称，

```
int GetNext(int val) {
    int ans = 2; // a[2].val==INF
    int p = root;
    while (p) {
        if (val == a[p].val) { // 检索成功
            if (a[p].r > 0) { // 有右子树
                p = a[p].r;
                // 右子树上一直向左走
                while (a[p].l > 0) p = a[p].l;
                ans = p;
            }
            break;
        }
        // 每经过一个节点，都尝试更新后继
        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 ans;
}
```

求5的后继

删除

在BST中删除权值为x的节点，需要先查找是否存在。

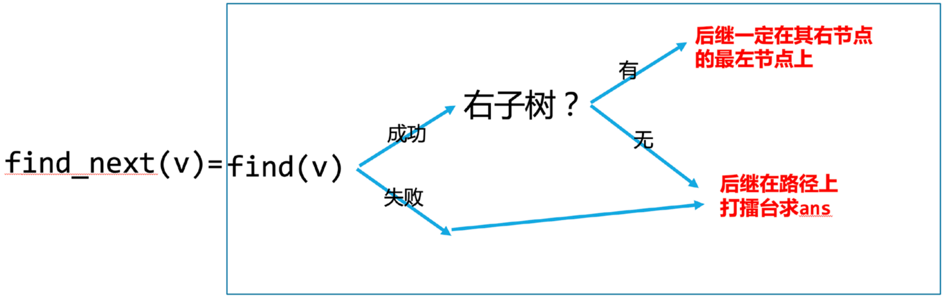
若查找到权值为x的节点编号为p，需要分三种情况：

(1)p为叶子节点，不破坏树的结构，则直接删除。

(2)p只有一个子节点，则让子节点代替p的位置，(1)的情况也可以按照此方式进行处理，叶节点的子节点就是空节点，空节点代替等同于删除。

(3)p既有左子树也有右子树，则在BST中寻找节点p的**后继节点**nxt来代替p的位置。后继节点就是大于x且权值最小的节点，后继节点nxt一定不存在左子树，所以可以直接删除后继节点nxt，并令后继节点nxt的右子树代替p的位置。

后继节点nxt一定不存在左子树，如果后继节点存在左子树，那么左子树的节点权值大于x，且小于nxt的权值，与nxt是后继节点矛盾。



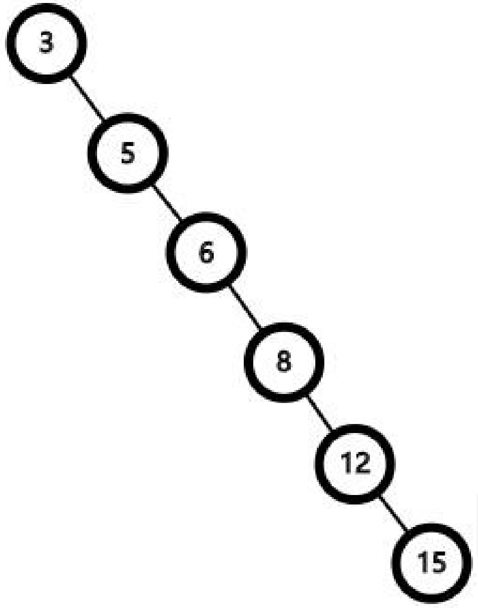
```
void Remove(int &p,int x)                              //从子树p中删除权值为x的节点,p是引用，是父结点的l或r的别名
{
        if(p==0)        return;
        if(x==a[p].val)             //查找到x
        {
                if(a[p].l==0)p=a[p].r;      //无左子树,右子节点代替
                else if(a[p].r==0)          p=a[p].l;      //无右子树，左子节点代替
                else                                       //既有左子树，又有右子树
                {
                        //后继节点往p的右子节点的左子节点一直往下找
                        int nxt=a[p].r;
                        while(a[nxt].l)             nxt=a[nxt].l;
                        Remove(a[p].r,a[nxt].val);                     //删除后继节点nxt
                        a[nxt].l=a[p].l;                               //nxt代替p的位置
                        a[nxt].r=a[p].r;
                        p=nxt;                        //p是引用，是父结点的l或r的别名,指向添加的节点nxt
                }
                return;
        }
        if(x<a[p].val)              Remove(a[p].l,x);
        else            Remove(a[p].r,x);
}
```

时间复杂度

在随机数据中，BST的每次操作时间复杂度为**O(logN)**。

然而，聪明的小盆友们早已经看出来了，

如果BST的形态是一条链。依次插入权值：3、5、6、8、12、15，BST形态如下：



时间复杂度卡到O(N)，为了使得BST的形态“平衡”（左右子树尽可能规模相等），

每次操作时间复杂度尽可能接近O(logN)，

从而产生了**平 衡树**。

FZOI

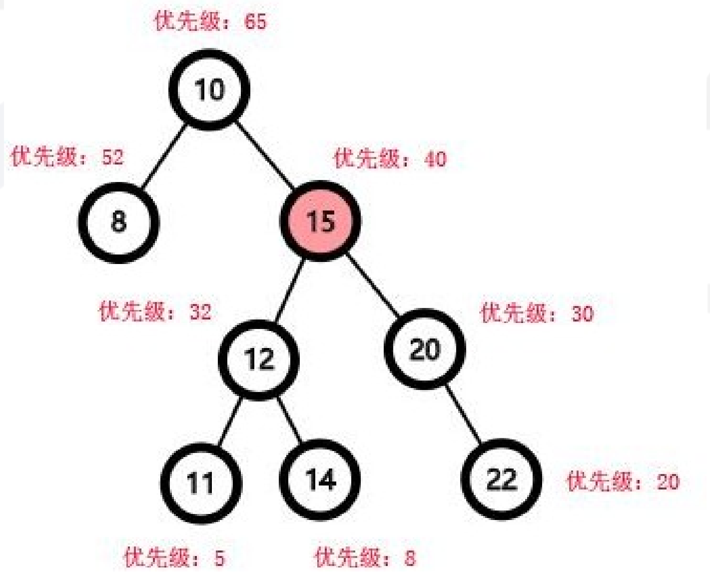
信息学

平衡树初步Treap

Treap是一种简单的平衡树，在普通二叉查找树的基础上，赋予每个节点一个属性：**优先级**。

对于Treap中的节点，除了权值满足二叉查找树的性质外，节点的优先级还要满足**大根堆**(小根堆)的性质。

简单来说，从权值上看，Treap是一棵二叉查找树，从优先级上看，Treap是一个堆(Heap)。



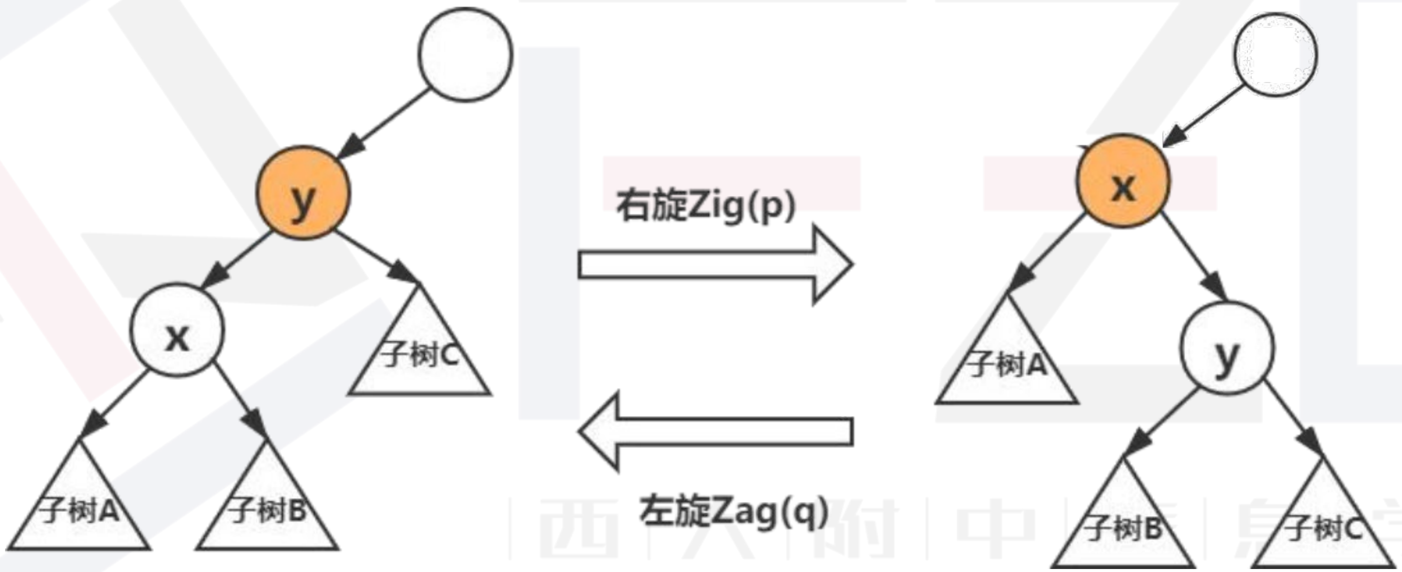
西南大学附属中学 High School Affiliated to Southwest University

BST不平衡是因为有序的数据会使查找的路径退化成链，而**随机数据**使其退化的概率非常小。因此，Treap中每个节点的优先级的值可以通过**随机函数生成**，这样Treap的结构就会趋于平衡了。

```
sconst int N=100010;
struct Treap
{
        int l,r;        //左右子节点下标
        int val;        //权值
        int pr;                 //优先级
        int cnt;        //重复个数
        int size;       //子树大小
}a[N];
int n,tot,root=1,INF=0x3fffffff;
void Update(int p)              //更新子树大小，后面程序会使用
{
        a[p].size=a[a[p].l].size+a[a[p].r].size+a[p].cnt;
}
```

Treap旋转

为了使Treap在满足BST的性质的同时，也同时满足大根堆的性质，需要对Treap的结构进行调整，而调整的方法为旋转，旋转分为**左旋**和**右旋**，如下图所示：

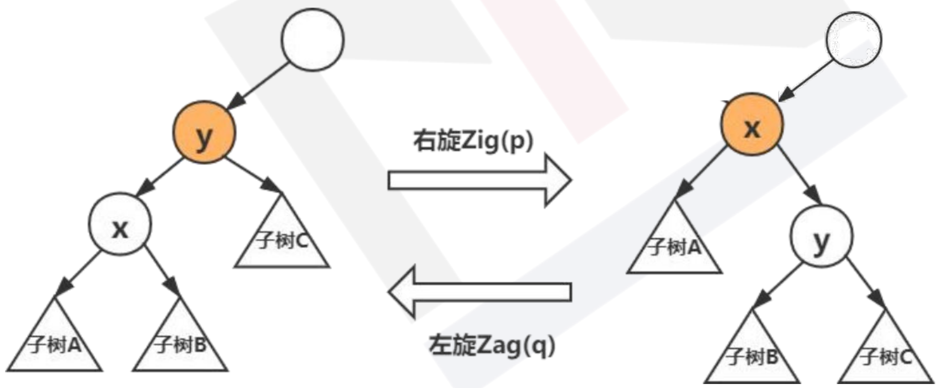


旋转操作是保证在BST性质的基础上，根据节点优先级调整。

<u>旋转不会改变BST的性质，只会改变节点优先级的位置</u>。

Treap旋转

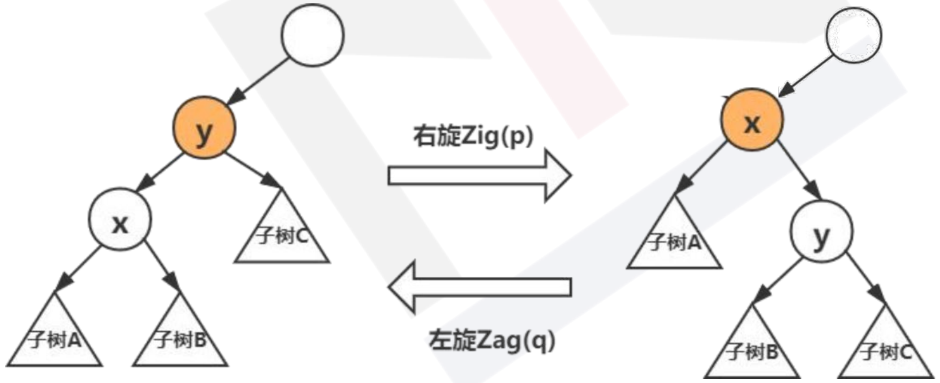
（1）若x的优先级大于y的优先级，需要进行右旋：将x变为y的父节点，x原来的右子树变为y的左子 树。




```
void Zig(int &p)        //          右旋，p的左子节点变为p的父节点
{
        int q=a[p].l;           //p的左子节点为q
        a[p].l=a[q].r;          //q的右子树变为p的左子树
        a[q].r=p;               //q变为p的父节点
        p=q;                    //引用，p是其父节点的l或r的别名，指向q
        Update(a[p].r);
        Update(p);//改变结构后，更新子树大小
}
```

Treap旋转

（2）若y的优先级大于x的优先级，需要进行左旋：将y变为x的父节点，y原来的左子树变为x的右子树。




```
void Zag(int &p)        //左旋，p的右子节点变为p的父节点
{
        int q=a[p].r;           //p的右子节点为q
        a[p].r=a[q].l;          //q的左子树变为p的右子树
        a[q].l=p;               //q变为p的父节点
        p=q;                    //引用，p是其父节点的l或r的别名，指向q
        Update(a[p].l);
        Update(p);              //改变结构后，更新子树大小
}
```

Treap插入

Treap在插入每个新节点时，在有权值属性的基础上，会给该节点生成一个随机值，作为优先级的属性。然后像堆的插入过程一样，自底向上依次检查，当某个节点不满足大根堆性质时，就执行旋转，使该节点与父节点交换。

插入节点的过程，先按照BST性质，即权值大小插入到合适位置；再按照堆的性质，通过旋转调整优先级，使其满足大根堆性质，步骤如下：

设变量p初始时等于root，插入权值为x的节点：

(1)若x==a[p].val，节点p的数量+1。
(2)若x<a[p].val，递归往p的左子树a[p].l插入。
(3)若x>a[p].val，递归往p的右子树a[p].r插入。
(4)若p为空节点，插入到当前节点，生成一个随机值作为当前节点的优先级。

此时，是满足BST性质的，但是不一定满足大根堆的性质。因此在回溯的时候，若不满足大根堆性质，进行旋转。
(5)若p的左子节点优先级大于p，则进行右旋，交换左子节点与p的位置。
(6)若p的右子节点优先级大于p，则进行左旋，交换右子节点与p的位置。

```
void Insert(int &p,int x)        //插入
{
        if(p==0)                                        //判断是否存在
        {
                a[++tot].val=x;
                a[tot].pr=rand();               //rand()随机函数，优先级为随机值
                a[tot].cnt=1;
                a[tot].size=1;
                p=tot;                          //引用，p是其父节点l或r的别名，指向tot
                return;
        }
        if(x==a[p].val)                 //存在权值为x的节点
        {
                a[p].cnt++;                     //数量+1
                Update(p);
                return;
        }
        if(x<a[p].val)
        {
                Insert(a[p].l,x);                       //递归插入
                if(a[p].pr<a[a[p].l].pr)        Zig(p);         //回溯，不满足堆的性质，右旋
        }
        if(x>a[p].val)
        {
                Insert(a[p].r,x);                       //递归插入
                if(a[p].pr<a[a[p].r].pr)        Zag(p);         //回溯，不满足堆的性质，左旋
        }
        Update(p);                                      //回溯，更新子树大小
}
```

Treap删除

如果删除的节点为叶节点，可以直接删除。

对于非叶子节点，因为Treap支持旋转，可以通过旋转将其变为叶子节点。若删除的节点p是非叶子节点，选择左右子节点优先级大的和节点p交换：

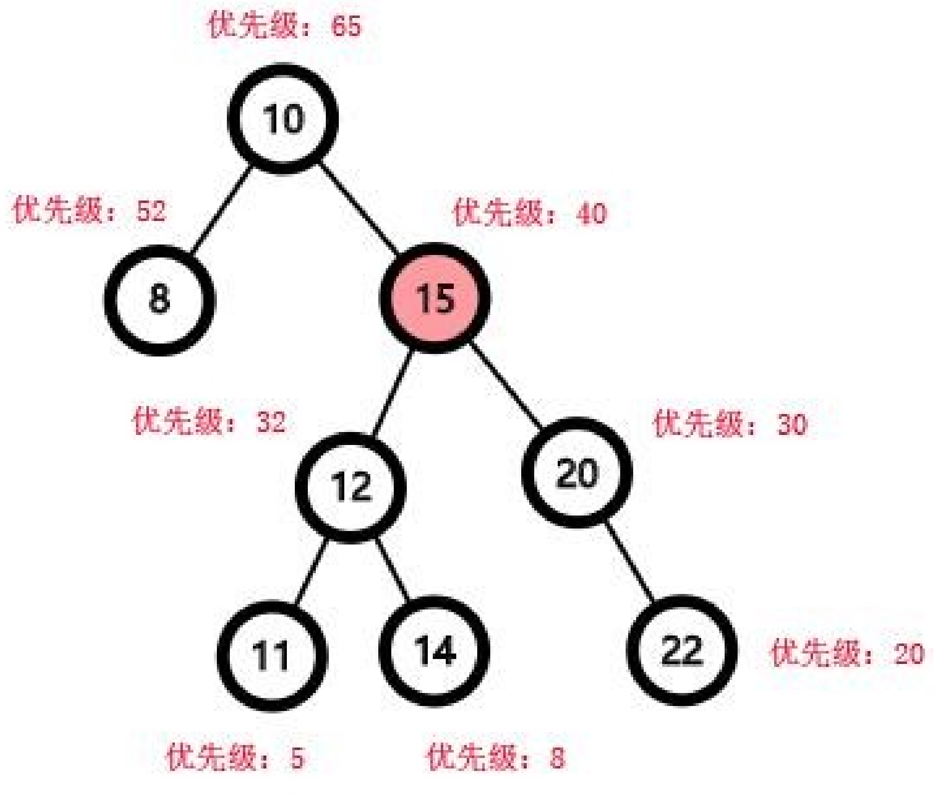
- 若左子节点优先级大，则右旋，保证优先级大的左子节点在右子节点上方。
- 若右子节点优先级大，则左旋，保证优先级大的右子节点在左子节点上方。

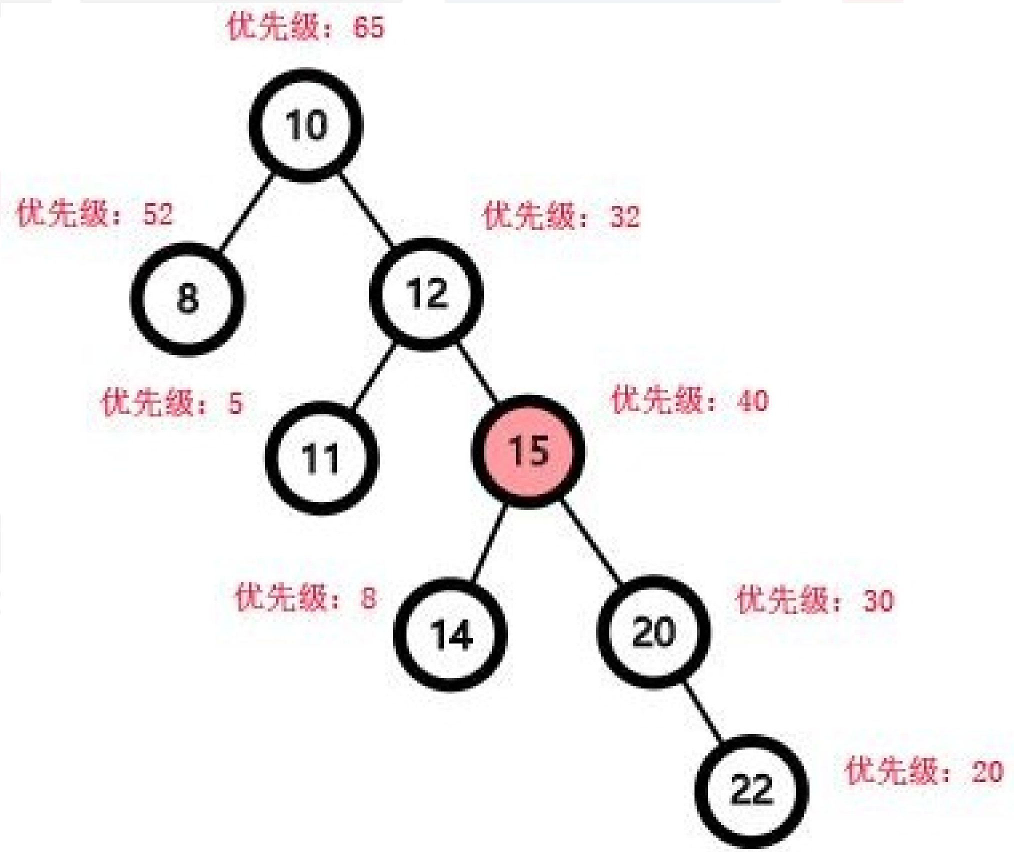
西 | 大 | 附 | 中 | 信 | 息 | 学 | 竞 | 赛
High School Affiliated to Southwest University

Treap删除

如下图，删除节点15：

首先查找到权值为15的位置，比较左右子节点优先级，左子节点优先级高，右旋：





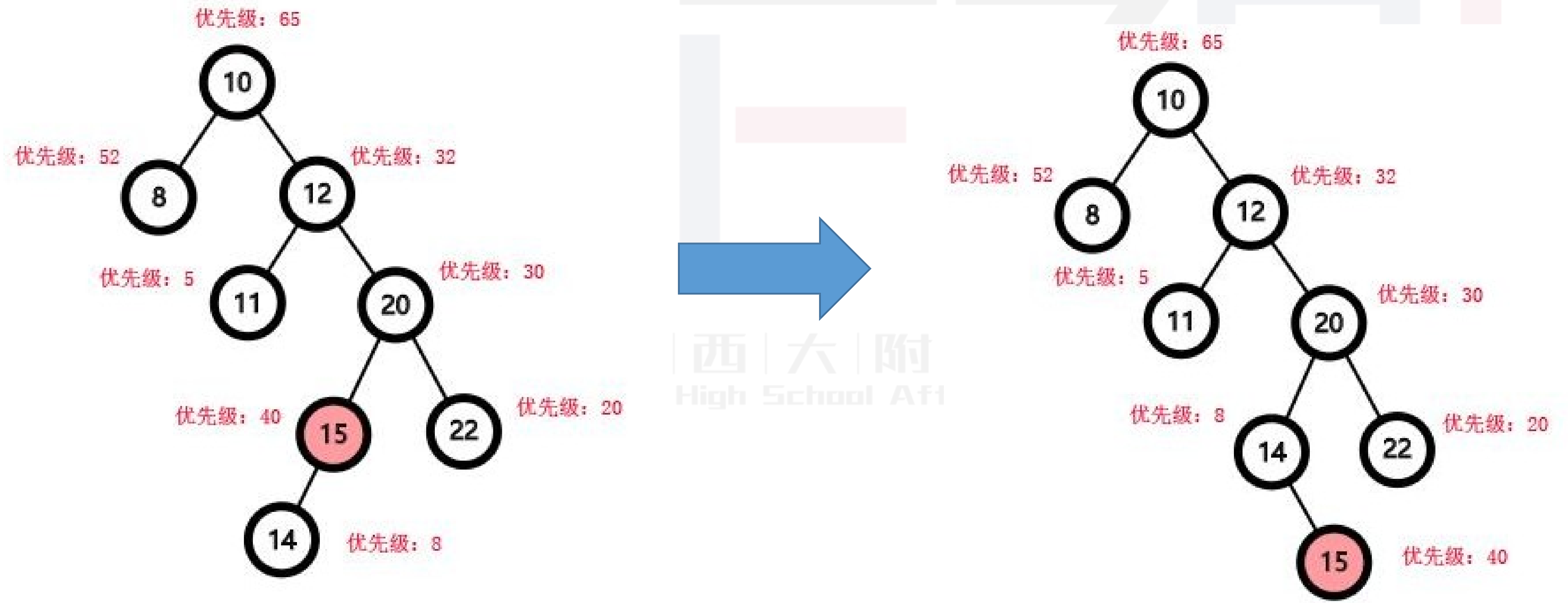
Treap删除

继续比较左右子节点优先级，右子节点优先级高，左旋：

Treap删除

继续比较左右子节点优先级，左子节点优先级高，右旋：

此时，删除的节点为叶子节点，可以直接删除。



Treap删除


```
void Remove(int &p,int x)                                   //删除权值x
{
        if(p==0)        return;                 //不存在
        if(x==a[p].val)
        {
                if(a[p].cnt>1)                              //有重复元素，减少个数
                {
                        a[p].cnt--;
                        Update(p);                          //更新子树大小
                        return;
                }
                if(a[p].l||a[p].r)              //非叶子节点
                {
                        if(a[a[p].l].pr>a[a[p].r].pr)           Zig(p),Remove(a[p].r,x); //右旋
                        else            Zag(p),Remove(a[p].l,x);                //左旋
                        a[p].size=a[a[p].l].size+a[a[p].r].size+a[p].cnt;       //回溯更新子树大小
                }
                else                                        //叶子节点
                        p=0;                                //引用，p是父节点l或r的别名，指向0，即删除
                return;
        }
        if(x<a[p].val)          Remove(a[p].l,x);
        if(x>a[p].val)          Remove(a[p].r,x);
        Update(p);      //回溯更新子树大小
}
```

求数值x的前驱与后继

前驱定义为小于x的最大数，后继定义为大于x的最小数。

求解前驱：

设ans为当前最优解，令变量p等于根节点root，从p开始访问：

1 若当前节点a[p] . val <x，且a[p] . val >ans，则更新最优解，继续寻找：

若a[p].val<x，往右子节点a[p].r找，否则往左子节点a[p].l找。

2 若a[p] . val ==x，则先找到p的左子节点，再从左子节点的右子节点一直往下找。

3 若p为空，则停止寻找。

西大附中信息学竞赛 High School Affiliated to Southwest University

求数值x的前驱与后继


```
int Getpre(int x)                    //前驱
{
        int ans=-INF;
        int p=root;
        while(p)
        {
                if(x==a[p].val)          //找到相等
                {
                        if(a[p].l)
                        {
                                p=a[p].l;
                                while(a[p].r)p=a[p].r;
                                ans=a[p].val;
                        }
                        break;
                }
                if(a[p].val<x&&a[p].val>ans)      ans=a[p].val;         //更新最优解
                if(a[p].val<x)    p=a[p].r;         //选择合适的方向继续找
                else p=a[p].l;
        }
        return ans;
}
```

求排名为k的数值

求排名，需要维护每个节点为根的子树大小，即程序中的**size**属性。

一个节点的排名，取决于其左子树的大小。设当前访问的子树根节点为p，重复元素个数为a[p].cnt，p在作为子树根节点，在子树中的排名介于：

左子树节点总数+1 ~ 左子树节点总数+p的重复个数，即区间**[a[a[p].l]].size+1 ,a[a[p].l]].size+a[p].cnt]** 之间。因为左子树节点权值都小于p。

1　　若 a[a[p].l]].size +1<=k && k<=a[a[p].l]].size + a[p].cnt，则排名k的节点就是当前节点p。

2　　若k <= a[a[p].l]].size ，则排名k的节点在左子树中。

3若k > a[a[p].l]].size + a[p].cnt，则排名k的节点子在右子树中，相当于在右子树中找排名为k- (a[a[p].l]].size + a[p].cnt)的节点。

在插入删除的时候，都需要从下往上更新size，一般采用递归的形式，以便于在回溯的时候更新size。

西 大 附 中 信 息 学 竞 赛 High School Affiliated to Southwest University

求排名为k的数值


```
int GetRank(int p,int k)                    //在根节点p的子树中，查找排名为k的元素
{
        if(a[a[p].l].size+1<=k && k<=a[a[p].l].size+a[p].cnt)        return a[p].val;
        if(k<=a[a[p].l].size)return    GetRank(a[p].l,k);
        return GetRank(a[p].r , k-(a[a[p].l].size + a[p].cnt));
}
```

西大附中信息学竞赛 High School Affiliated to Southwest University

求数值x的排名

求元素x的排名，类似于查找元素x的位置，在查找的过程中，统计比x小的元素个数。 设当前访问节点为p：

1 若x==a[p].val，则比x小的元素有a[a[p].l].size个。

2 若x<a[p].val，则在左子树继续找。

3 若x>a[p].val，则在右子树继续找，且找到了比x小的元素有a[a[p].l].size+a[p].cnt个。

```
int Getx_Rank(int p,int x)                              //在根节点p的子树中，查找x的排名
{
        if(p==0)      return 0;
        if(x==a[p].val)          return a[a[p].l].size+1;             //排名+1
        if(x<a[p].val)           return Getx_Rank(a[p].l,x);
        return Getx_Rank(a[p].r,x)+a[a[p].l].size+a[p].cnt;
}
```

Treap通过旋转，在维持权值满足BST性质之外，还能使节点的优先级满足大根堆的性质。

时间复杂度：

Treap的树的深度期望是**O(logN)**，所以各个操作的期望时间复杂度为O(logN)。

西大附中信息学竞赛 High School Affiliated to Southwest University