第9章 查找

9.0 基本概念

9.1 静态查找表

9.2 动态查找表

9.3 哈希表

9.2 动态查找表

典型的动态表——二叉排序树

一、二叉排序树的定义

二、二叉排序树的插入与删除

三、二叉排序树的查找分析

四、平衡二叉树(难点)

9.2.1 二叉排序树的定义

或是一棵空树；或者是具有如下性质的非空二叉树：

（1）左子树的所有结点均小于根的值；

（2）右子树的所有结点均大于根的值；

（3）它的左右子树也分别为二叉排序树。

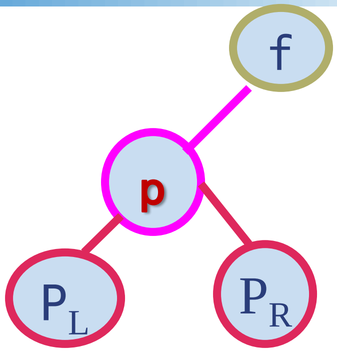
9.2.2、二叉排序树的插入与删除

2 二叉排序树的删除操作如何实现?

对于二叉排序树，删除树上一个结点相当于删除有序序列中的一个记录，要求删除后仍需保持二叉排序树的特性。

9.2.2、二叉排序树的插入与删除

2 二叉排序树的删除操作如何实现?



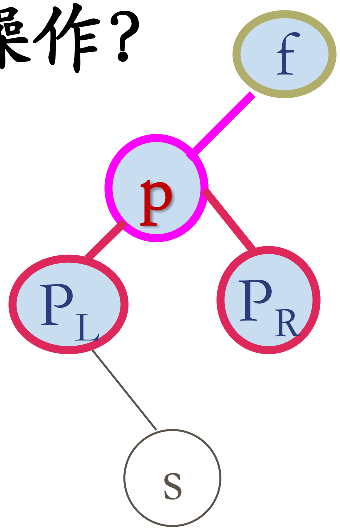
*p为叶子：删除此结点时，直接修改*f指针域即可；

*p只有一棵子树（或左或右）：令P_L或P_R为*f的左孩子即可；

*p有两棵子树：情况最复杂 →

9.2.2、二叉排序树的插入与删除

难点：*p有两棵子树时，如何进行删除操作？



分析：

设删除前的中序遍历序列为：

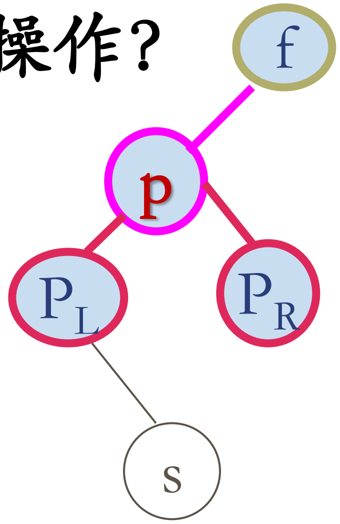
… P_L s_L s p P_R f …. //显然p的直接前驱是s

//s是p左子树最右下方的结点

希望删除p后，其它元素的相对位置不变。

9.2.2、二叉排序树的插入与删除

难点：*p有两棵子树时，如何进行删除操作？



分析：

有两种解决方法：

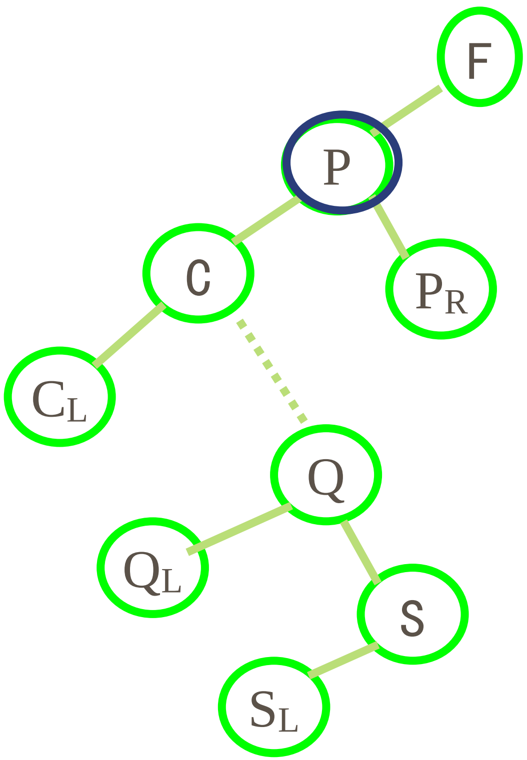
法1：令p的左子树为 f的左子树，p的右子树接为s的右子树；　//即 $f_L=P_L$； $S_R=P_R$ ；

法2：直接令s代替p，s的左子树接为P_L

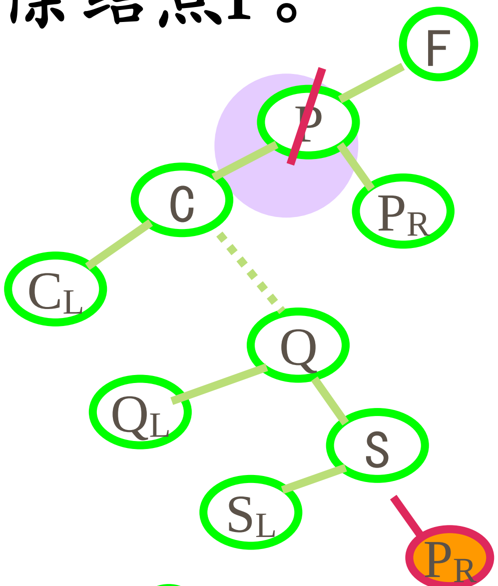
　　// s为p左子树最右下方的结点

9.2.2、二叉排序树的插入与删除

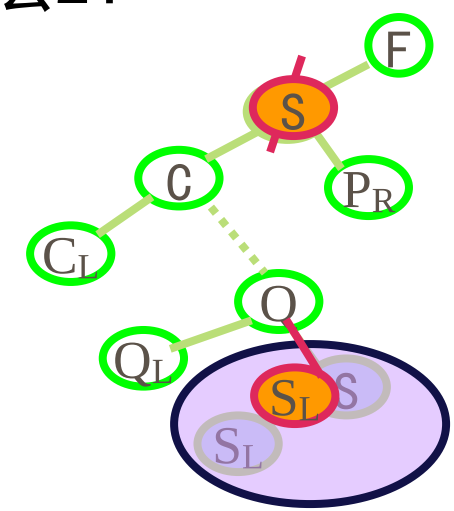
例：请从下面的二叉排序树中删除结点P。



法1：



法2：

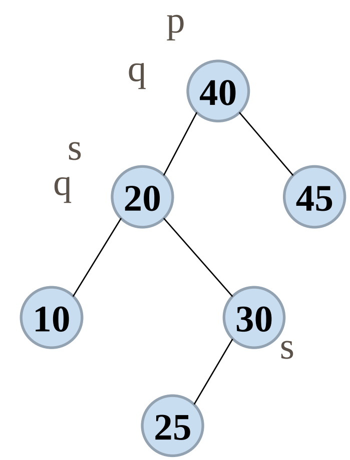


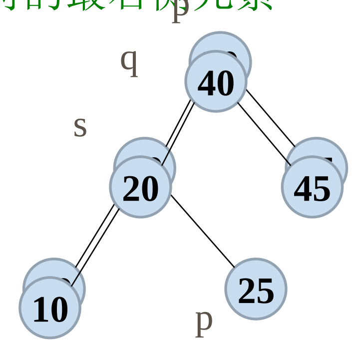
9.2.2、二叉排序树的插入与删除

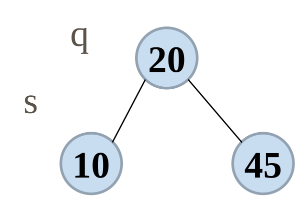
```
Status DeleteBST(BiTree &T,int key){
   //二排序树T中存在关键字等于key的数据元素时，则删除该数据结点
   if(! T) return FALSE;//不存在关键字等于key的数据元素
   else{
     if (key==T->data) //找到关键字等于key的数据元素
         return Delete(T):
     else if (key < T->data)
         return DeleteBST(T->Ichild,key);
     else
        return DeleteBST(T->rchild,key);
   }
}
```

9.2.2、二叉排序树的插入与删除

```
Status Delete(BiTree &p){ //从二叉排序树中删除节点
    BiTree q, s;
    if (!p->rchild) { //右子树空，则重接它的左子树
        q = p;  p = p->lchild; free(q);
    }
    else if (!p->lchild) { //左子树空，则重接它的右子树
        q = p;  p = p->rchild; free(q);
    }
    else { //左右子树非空
        q = p; s = p->lchild;
        while (s->rchild) {q = s; s = s->rchild;} //找左子树的最右侧元素
        p->data = s->data; //s指向被删除节点的前驱
        if (q != p) {
            q->rchild = s->lchild; //重接*q右子树
        }
        else{
            q->lchild = s->lchild; //重接*q左子树
            delete s;
        }
    }
    return TRUE;
}
```



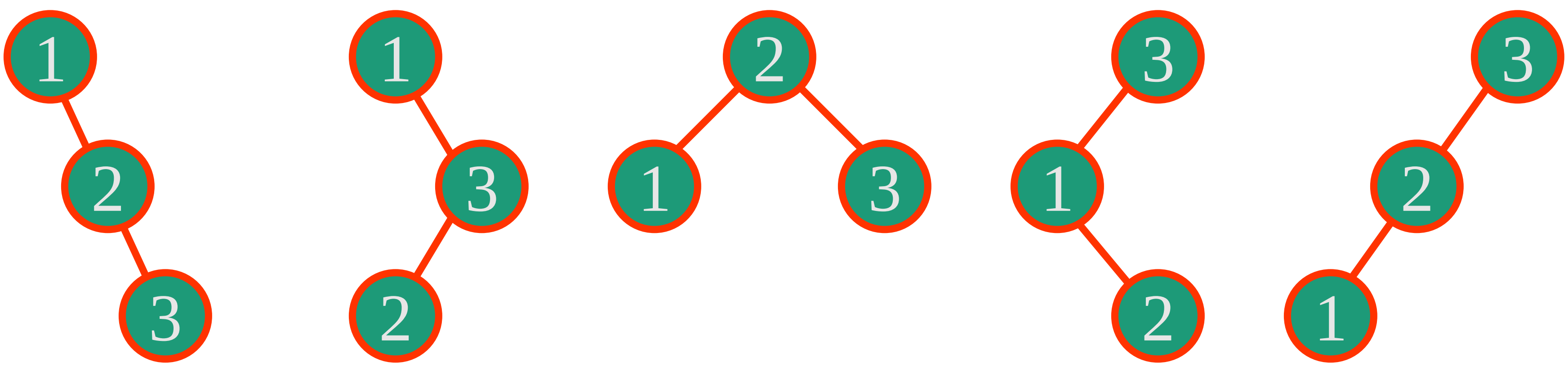




9.2.3 二叉排序树的查找分析

- 同样 3 个数据{ 1, 2, 3 }，输入顺序不同，建立起来的二叉排序树的形态也不同。这直接影响到二叉排序树的查找性能。
- 如果输入序列选得不好，会建立起一棵单支树，使得二叉排序树的高度达到最大，这样必然会降低查找性能。

{1, 2, 3}{1, 3, 2} {2, 1, 3} {2, 3, 1}{3, 1, 2}{3, 2, 1}



9.2.3　二叉排序树的查找分析

1　二叉排序树上查找某关键字等于结点值的过程，其实就是走了一条从根到该结点的路径。

比较的关键字次数＝此结点的层次数;

最多的比较次数＝树的深度（或高度），即

$\lfloor \log_2 n \rfloor + 1$

9.2.3 二叉排序树的查找分析

2 一棵二叉排序树的平均查找长度为:

$$ASL = \frac{1}{n}\sum_{i=1}^{n} n_i \cdot C_i$$

其中:

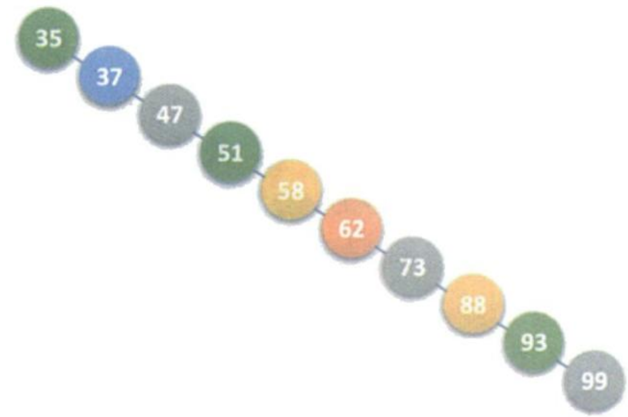
n_i 是每层结点个数;

C_i 是结点所在层次数;

m 为树深。

9.2.3 二叉排序树的查找分析

最坏情况：即插入的n个元素从一开始就有序。（单支树）此时树深为n；ASL= (n+1)/2；与顺序查找情况相同



最好情况：即：与折半查找中的判定树相同　（形态均衡），此时树深为 $\lfloor \log_2 n \rfloor + 1$；ASL=$\log_2(n+1) - 1$；与折半查找情况相同。

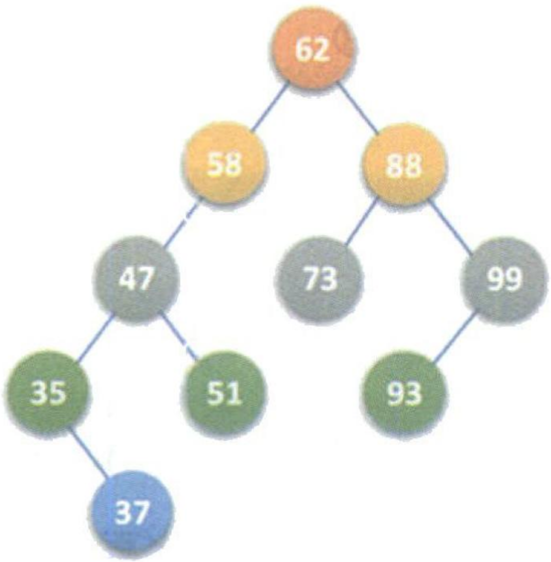
一般情况：$ASL \le 2(1+\frac{1}{n})\ln n$

（与 log n 同阶）

9.2.3　二叉排序树的查找分析

思考：如何提高二叉排序树的查找效率?

　　　尽量让二叉树的形状均衡



平衡二叉树

9.2.4 平衡二叉树

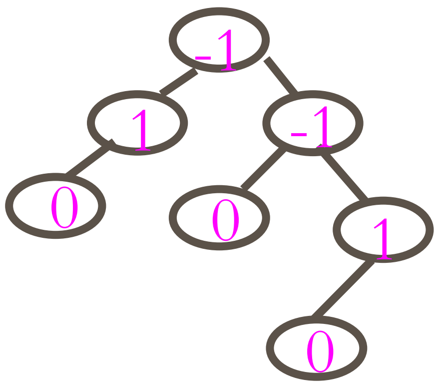
平衡二叉树或者是一棵空树，或者是具有下列性质的二叉树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1.

平衡因子（**BF)：**该结点的左子树的深度减去它的右子树的深度

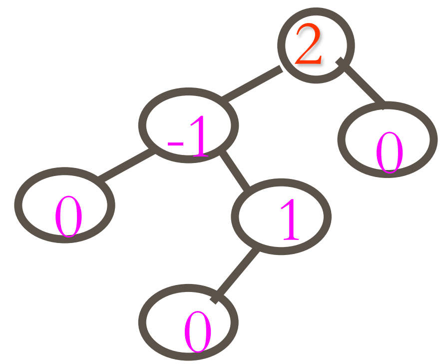
- 平衡二叉树上所有结点的平衡因子只可能是-1, 0, 和1.

9.2.4 平衡二叉树

例：判断下列二叉树是否AVL树?



(a) 平衡树



(b) 不平衡树

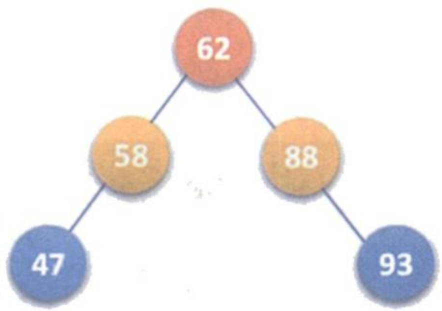
- 对于一棵有n个结点的AVL树，其高度保持在$O(\log_2 n)$数量级，ASL也保持在$O(\log_2 n)$量级。

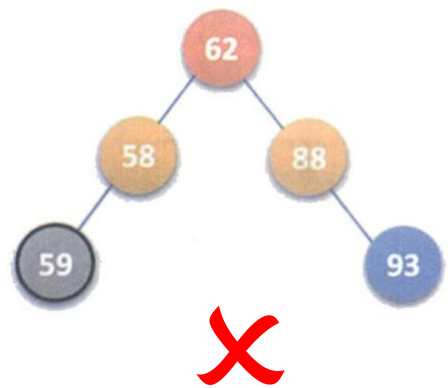
9.2.4 平衡二叉树

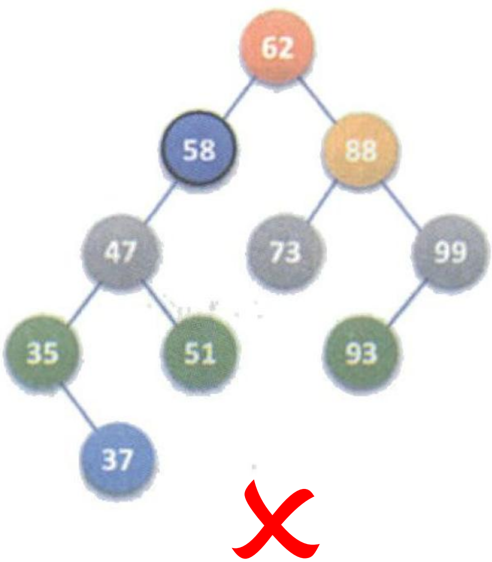
```
typedef struct BSTNode {
    ElemType  data;
    int bf;     //结点的平衡因子
    struct BSTNode *lchild,*rchild;
}BSTNode ,*BSTree;
```

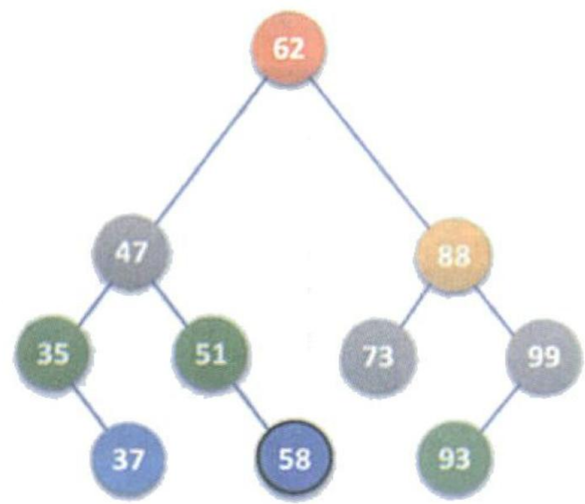
```
void R_Rotate(BSTree &p);
void L_Rotate(BSTree &p);
Status InsertAVL(BSTree &T,ElemType e,Boolean &taller);
void LeftBalance(BSTree &T);
void RightBalance(BSTree &T);
```

9.2.4 平衡二叉树


62
58
88
47
93

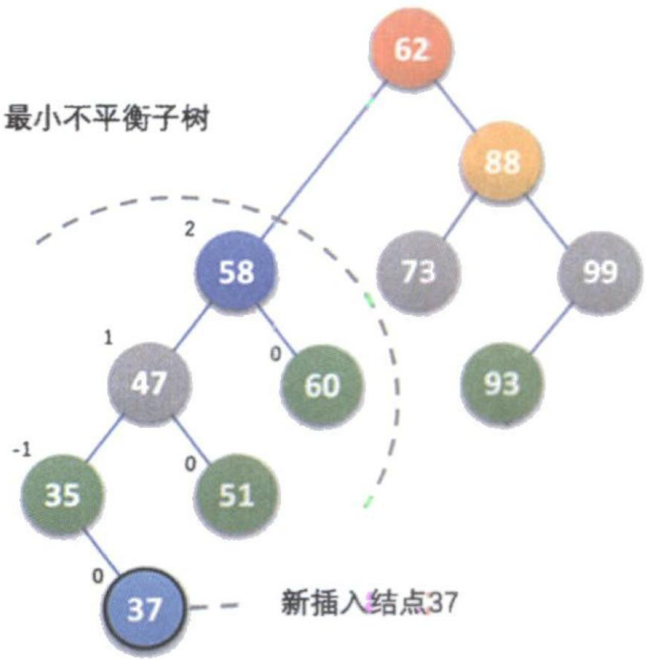

62
58
88
59
93


62
58
88
47
73
99
35
51
93
37


62
47
88
35
51
73
99
37
58
93

9.2.4 平衡二叉树

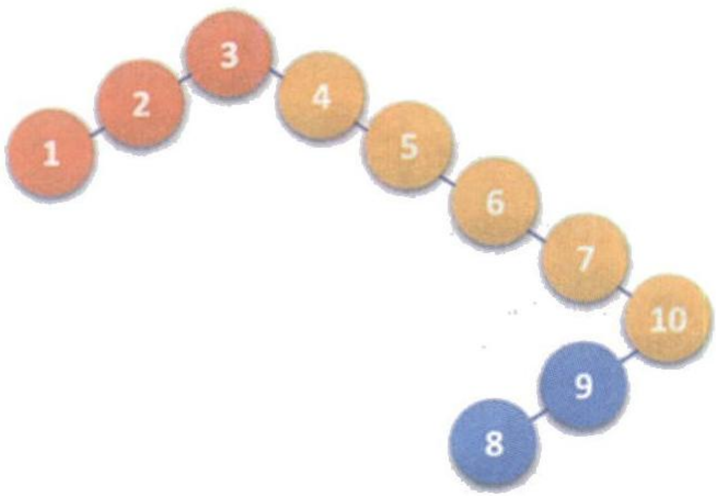
距离插入结点最近的，且平衡因子的绝对值大于1的结点为根的子树，我们称为**最小不平衡子树**

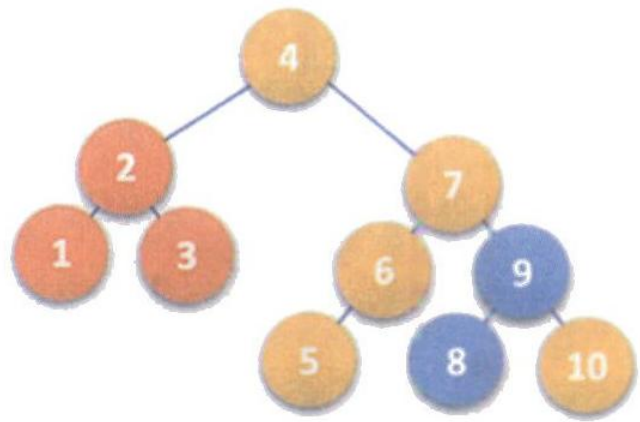


如果在一棵AVL树中插入一个新结点，就有可能造成失衡，此时必须重新调整树的结构，使之恢复平衡。我们称调整平衡过程为平衡旋转。

9.2.4 平衡二叉树

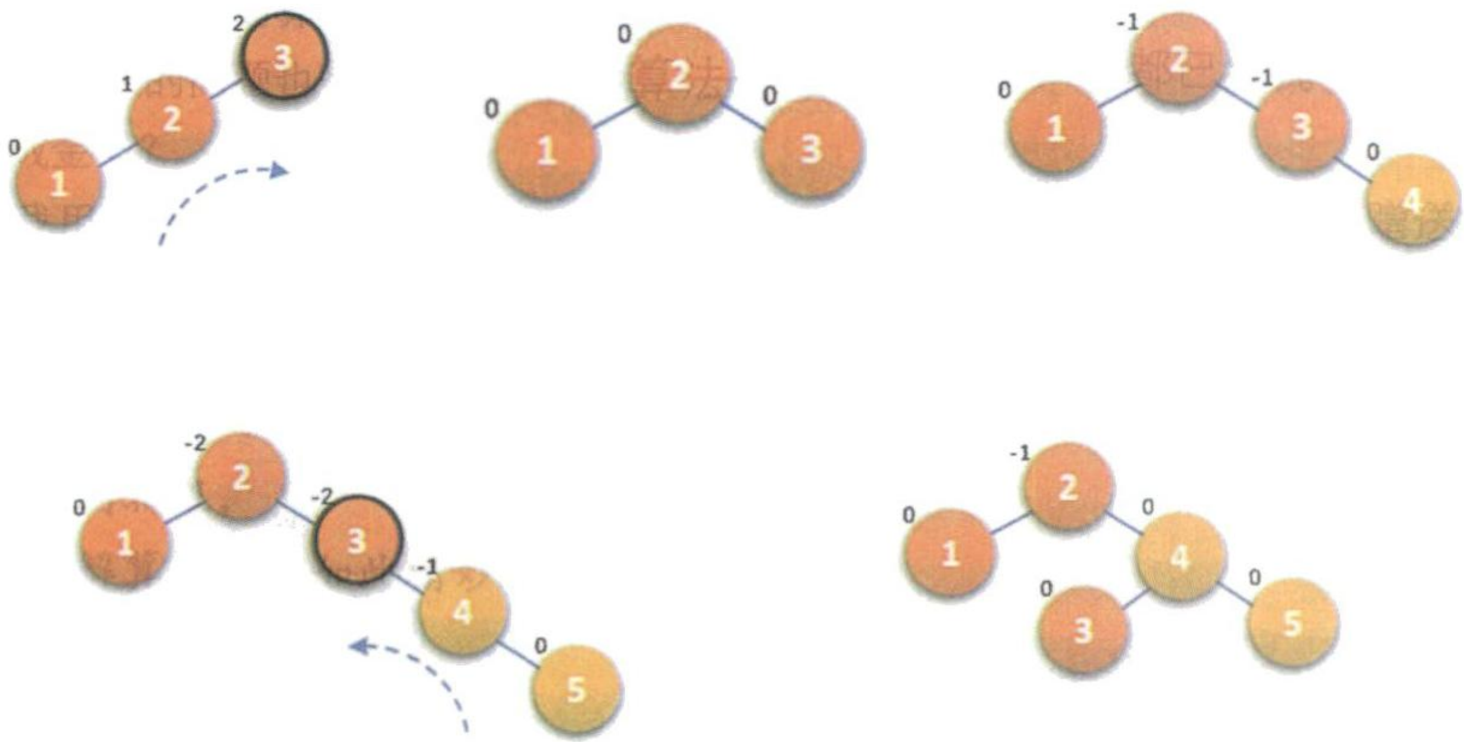
a[10] = {3,2,1,4,5,6,7,10,9,8}





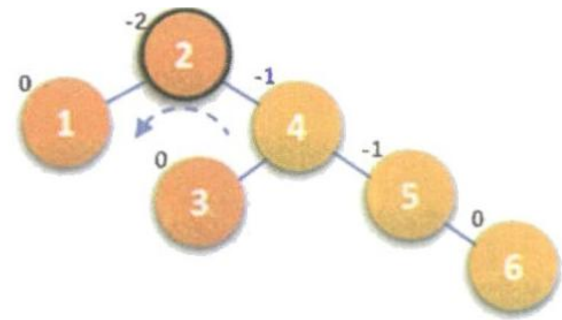
9.2.4 平衡二叉树

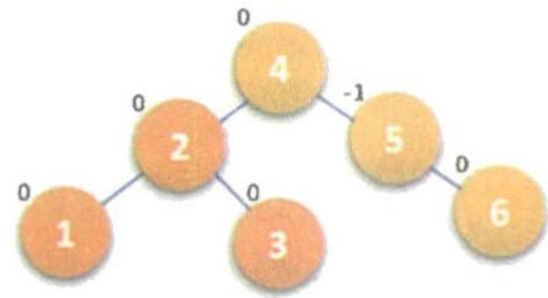
a[10] = {**3,2,1**,4,5,6,7,10,9,8}

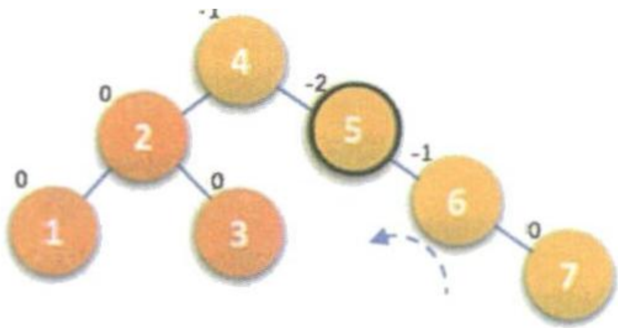


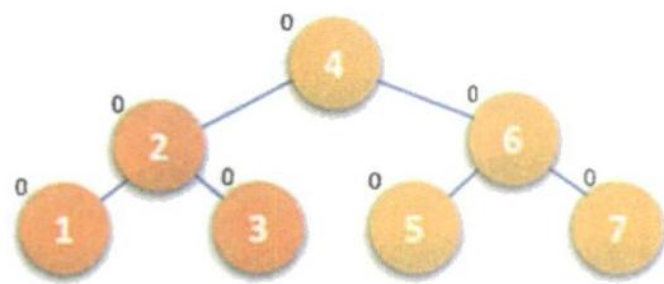
9.2.4 平衡二叉树

a[10] = {**3,2,1,4,5,**6,7,10,9,8}



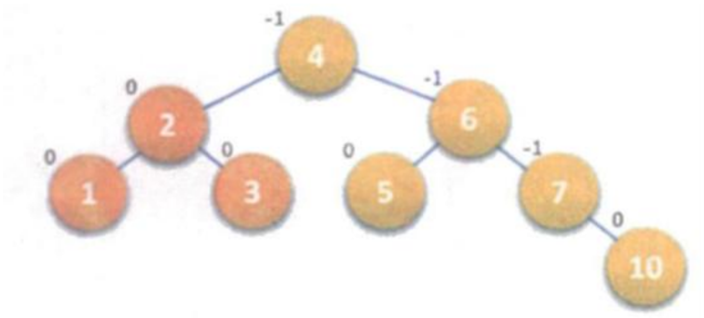


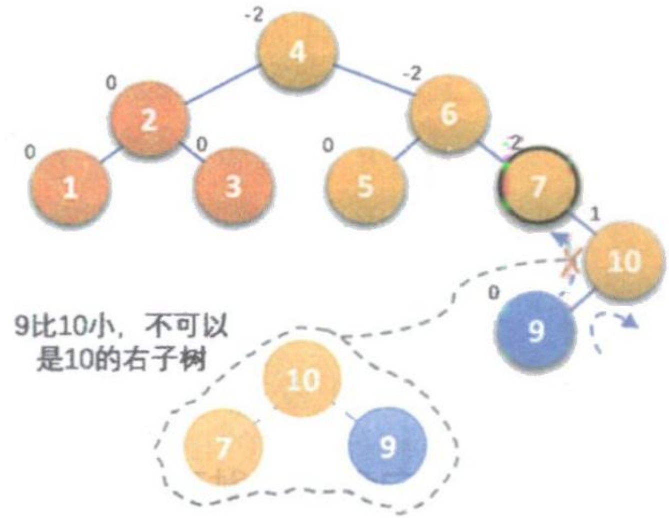


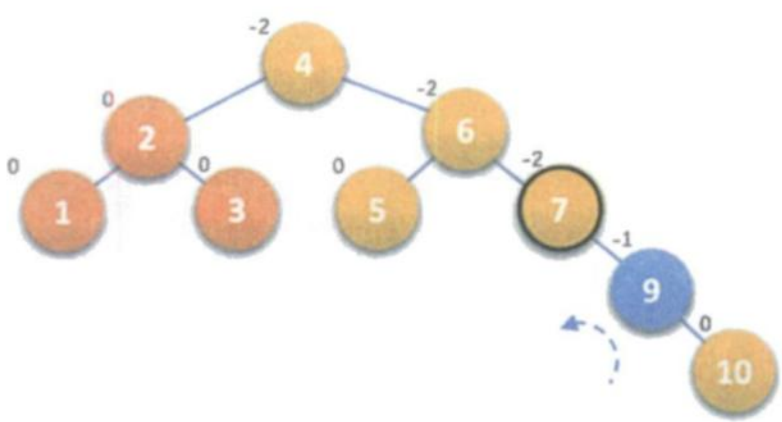


9.2.4 平衡二叉树

a[10] = {**3,2,1,4,5,6,7**,10,9,8}

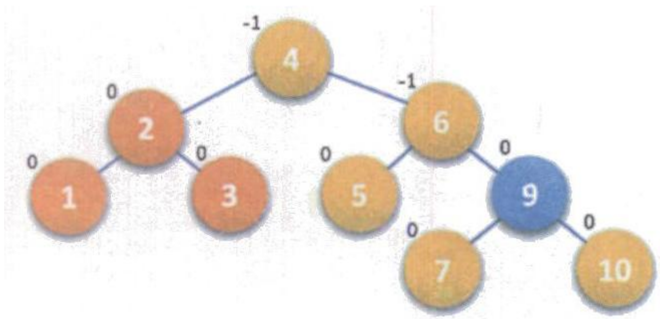


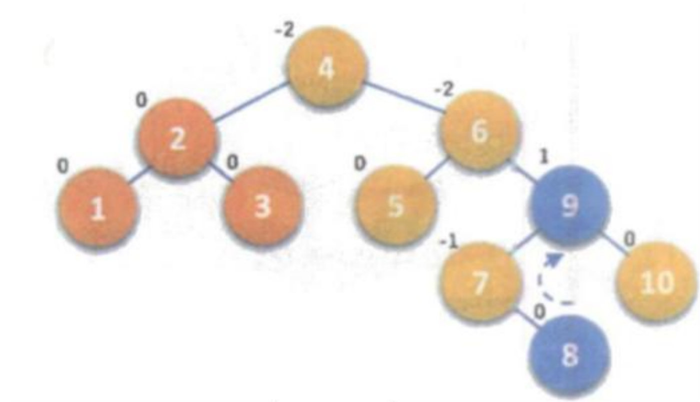


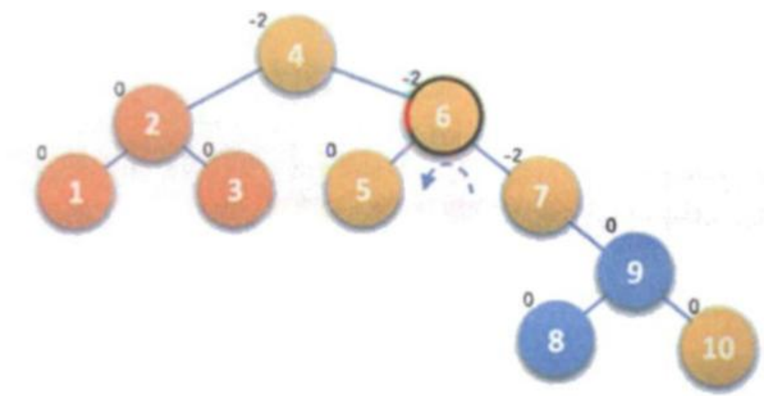


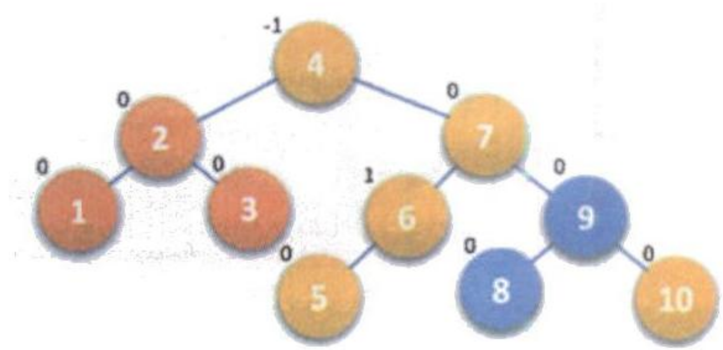
9.2.4 平衡二叉树

a[10] = {**3,2,1,4,5,6,7,10,9**,8}









9.2.4 平衡二叉树

平衡旋转可以归纳为四类：

- LL平衡旋转
- RR平衡旋转
- LR平衡旋转
- RL平衡旋转

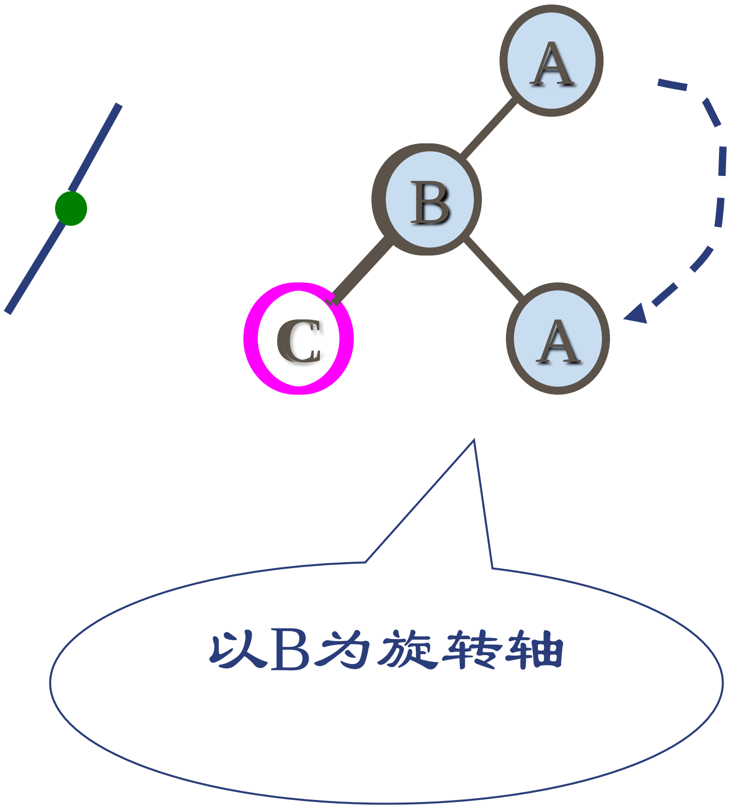
现分别介绍这四种平衡旋转。

9.2.4 平衡二叉树

1）LL平衡旋转（单右旋）

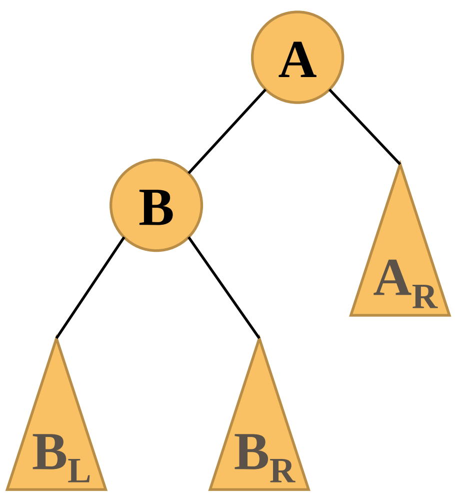
若在A的左子树的左子树上插入结点，使A的平衡因子从1增加至2，需要进行一次顺时针旋转。

旋转轴确定：沿着失衡路径，以失去平衡点的后一层结点为旋转轴。

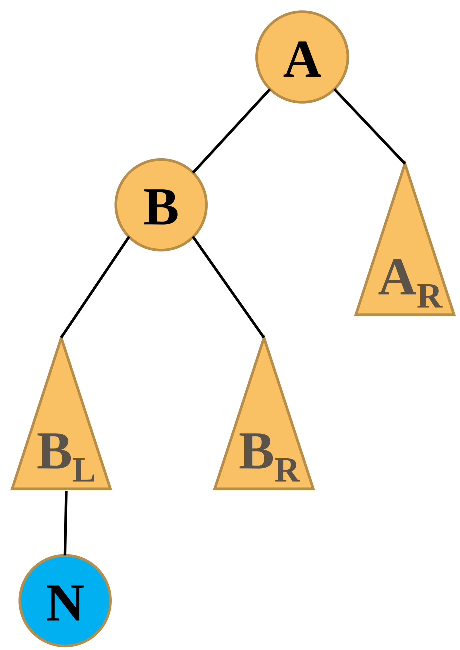


9.2.4 平衡二叉树

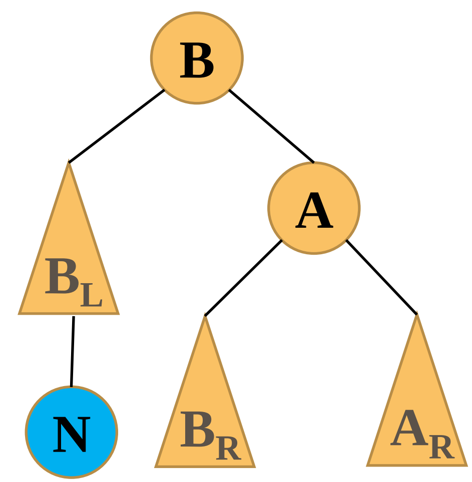
1）LL平衡旋转（单右旋）



插入N前是平衡二叉树



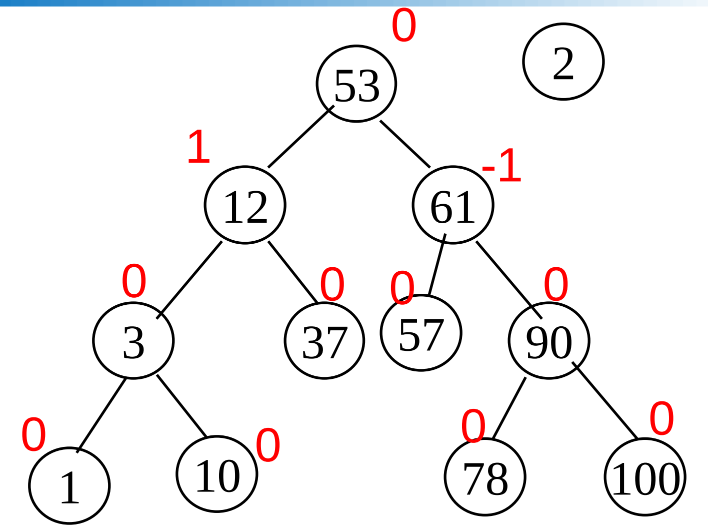
插入N后平衡性打破

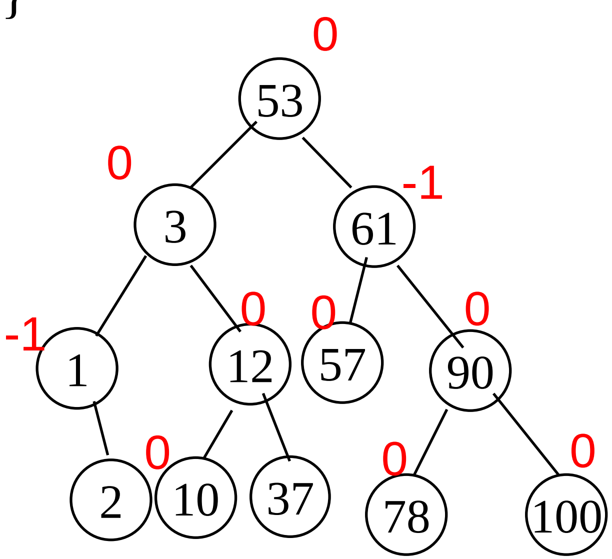


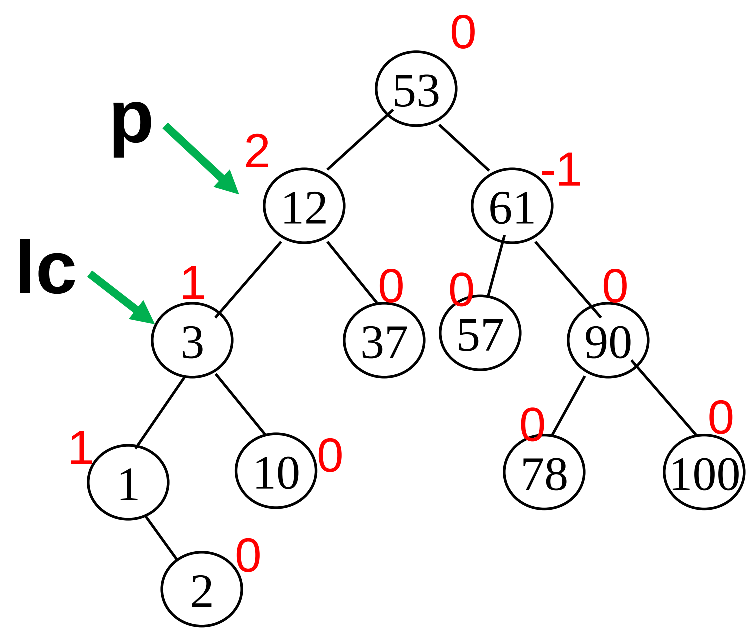
调整后平衡性恢复

9.2.4 平衡二叉树

```
void R_Rotate(BSTree &p){
//右单旋转的算法（p236 算法9.9）
      BSTree lc;
      lc=p->lchild;
      p->lchild=lc->rchild;
      lc->rchild=p; p=lc;
}
```





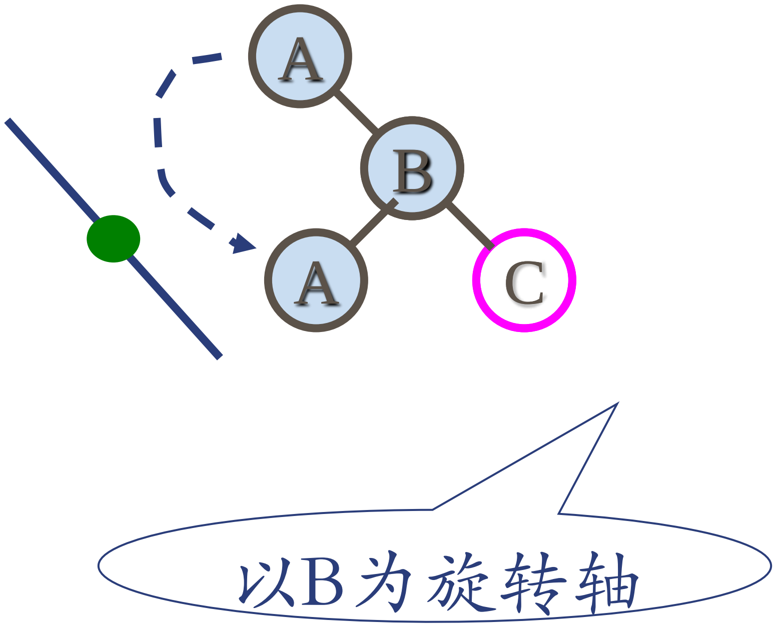


9.2.4 平衡二叉树

2）RR平衡旋转（单左旋）

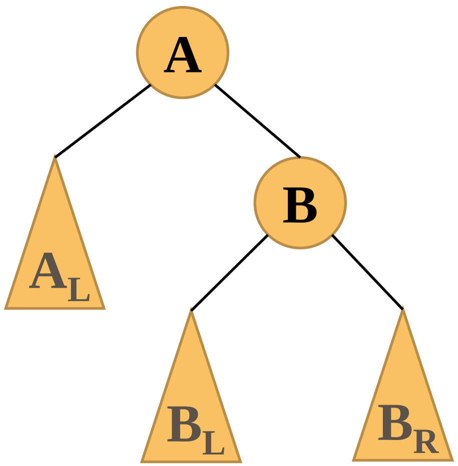
若在A的右子树的右子树上插入结点，使A的平衡因子从-1增加至-2，需要进行一次逆时针旋转。

旋转轴确定：沿着失衡路径，以失去平衡点的后一层结点为旋转轴。

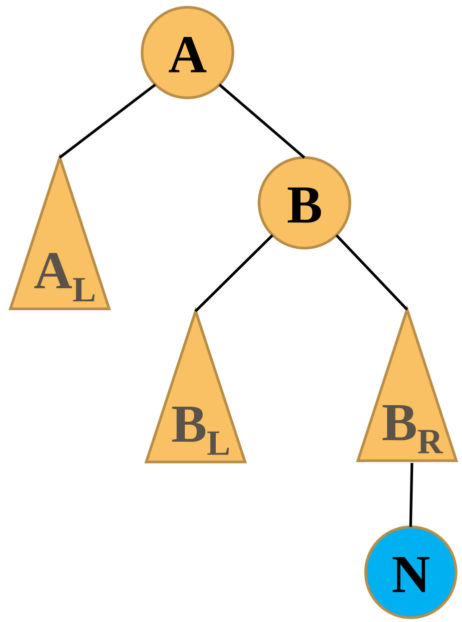


9.2.4 平衡二叉树

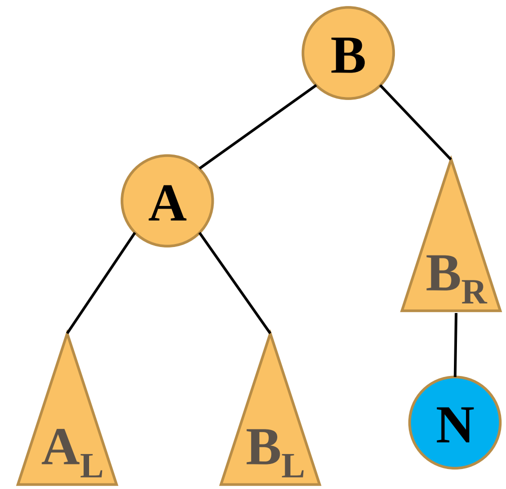
2）RR平衡旋转（单左旋）



插入N前是平衡二叉树



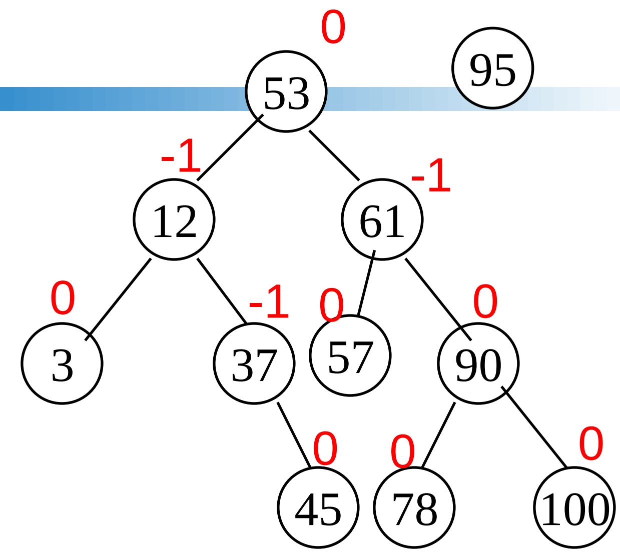
插入N后平衡性打破

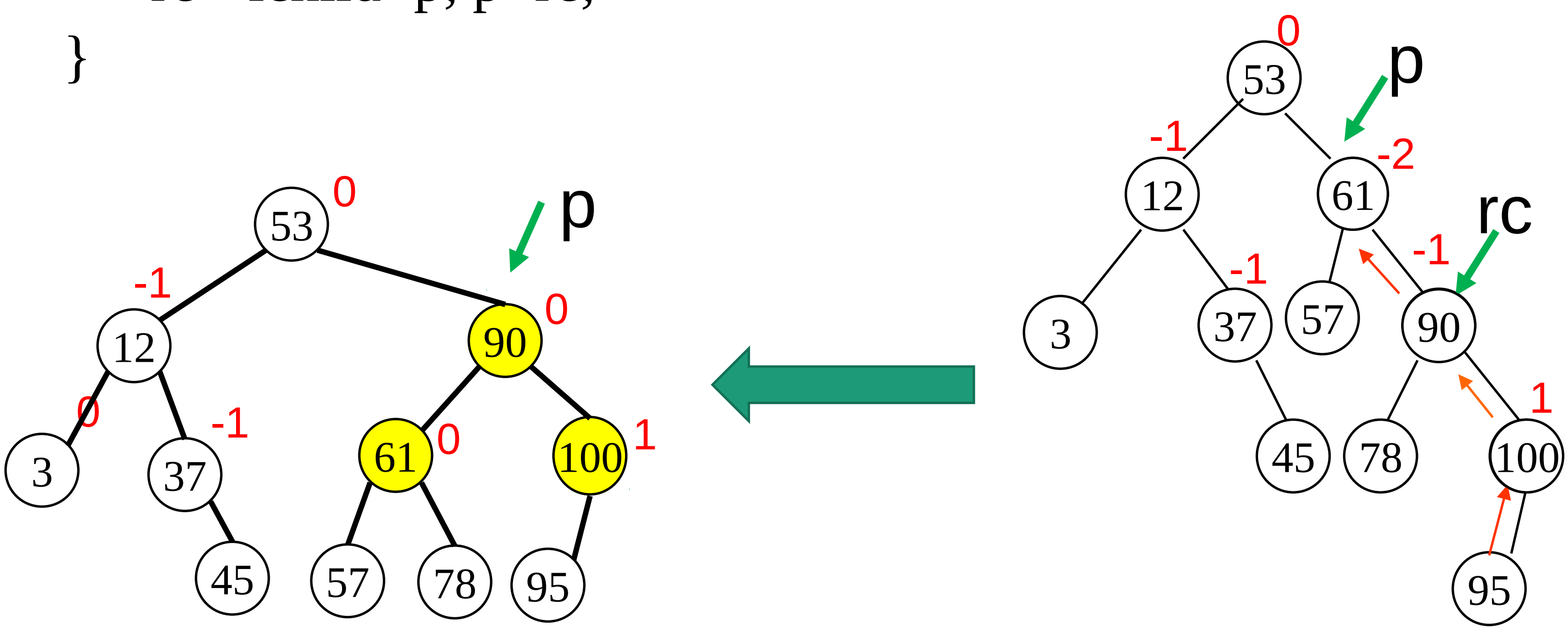


调整后平衡性恢复

9.2.4 平衡二叉树

```
void L_Rotate(BSTree &p){
//左单旋转的算法(p236 算法9.10)
    BSTree rc;
    rc=p->rchild;
    p->rchild=rc->lchild;
    rc->lchild=p; p=rc;
}
```



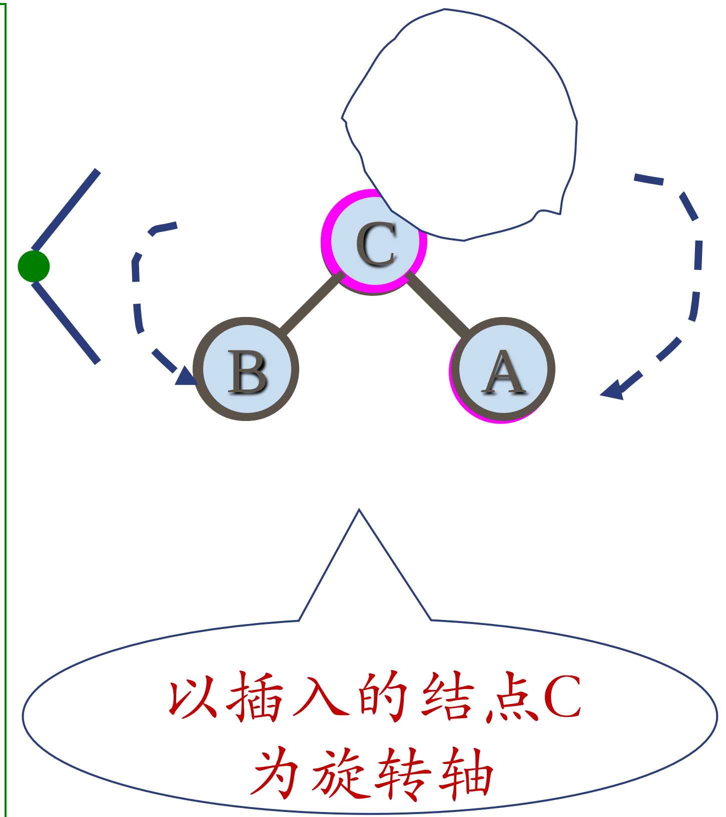


9.2.4 平衡二叉树

3）LR平衡旋转（先左后右）

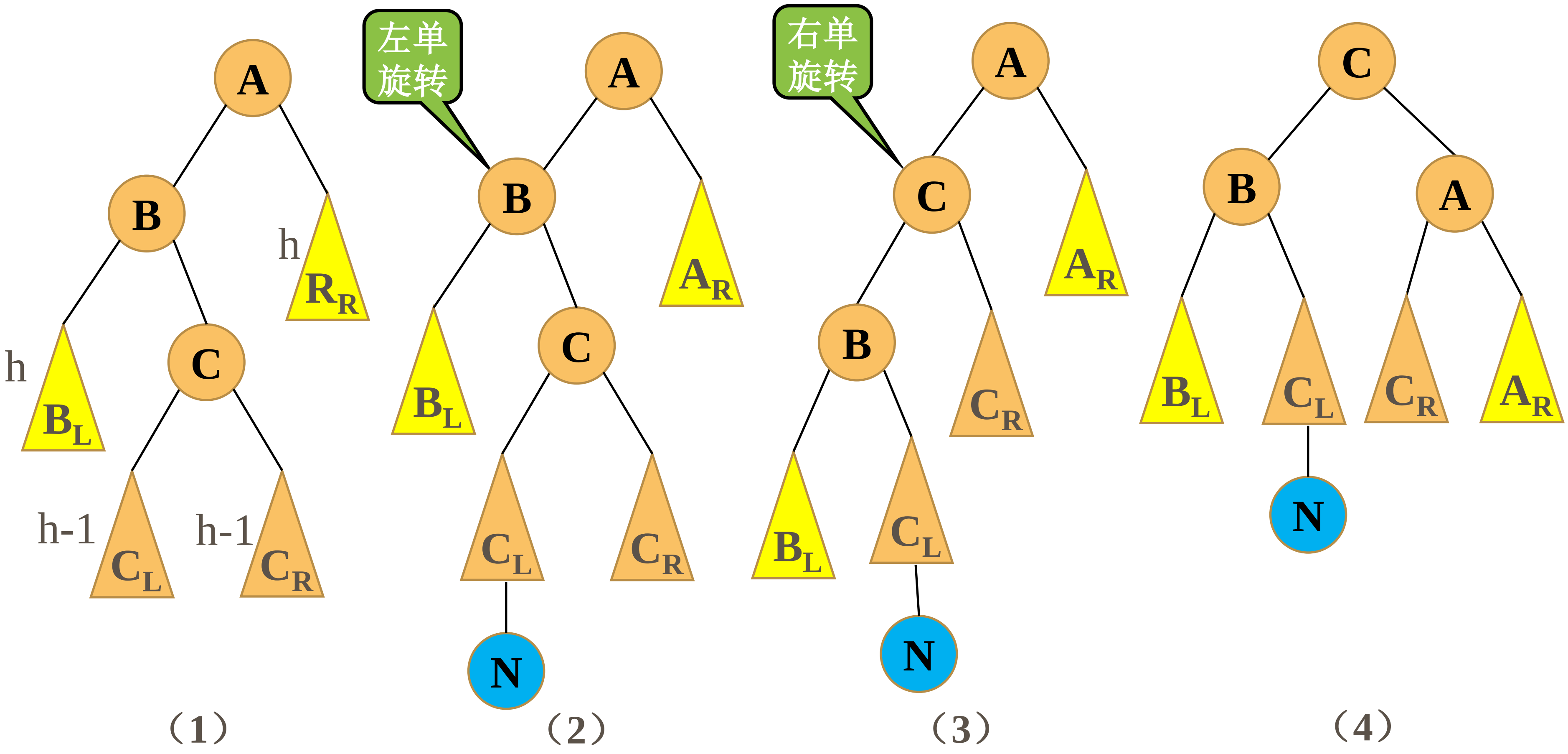
若在A的左子树的右子树上插入结点，使A的平衡因子从1增加至2，需要先进行逆时针旋转，再顺时针旋转。

旋转轴确定：沿着失衡路径，以失去平衡点的后二层结点为旋转轴。



9.2.4 平衡二叉树

3）LR平衡旋转（先左后右）



9.2.4 平衡二叉树

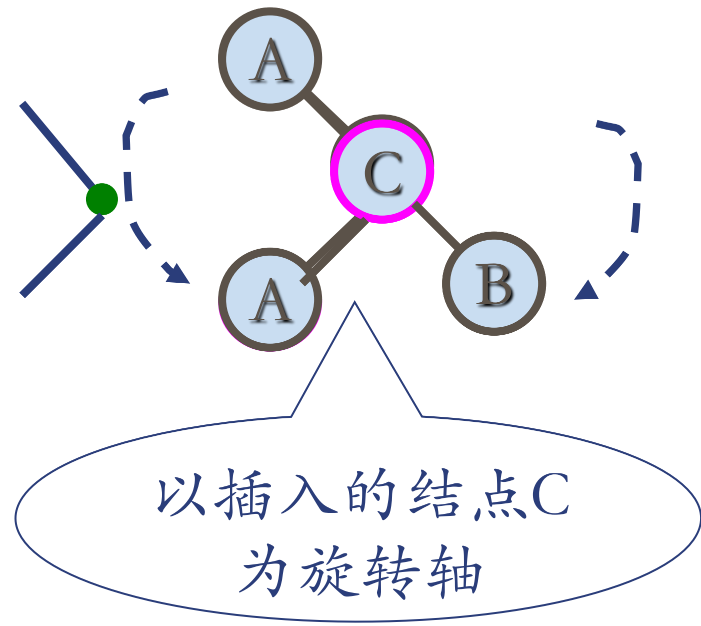
```
void LeftBalance(BSTree &T){//左平衡化的算法
    T->bf=LH
    BSTree lc,rd;   lc=T->lchild; //lc指向T的左子树
    switch(lc->bf){ //检查lc平衡度
    case LH: //LL型
        T->bf = lc->bf = EH; //新插入在T的左孩子的左子树
        R_Rotate(T); break; //单右旋
    case RH:    //新插入在T的左孩子的右子树
        rd=lc->rchild; //rd指向T的左孩子的右子树
        switch(rd->bf){ //修改平衡因子
        case LH: T->bf=RH;  lc->bf=EH; break;
        case EH: T->bf=lc->bf=EH; break;
        case RH: T->bf = EH; lc->bf=LH; break;
        }
        rd->bf=EH;
        L_Rotate(T->lchild); R_Rotate(T); //先左旋再右旋
    }
}
```

9.2.4 平衡二叉树

4）RL平衡旋转（先右后左）

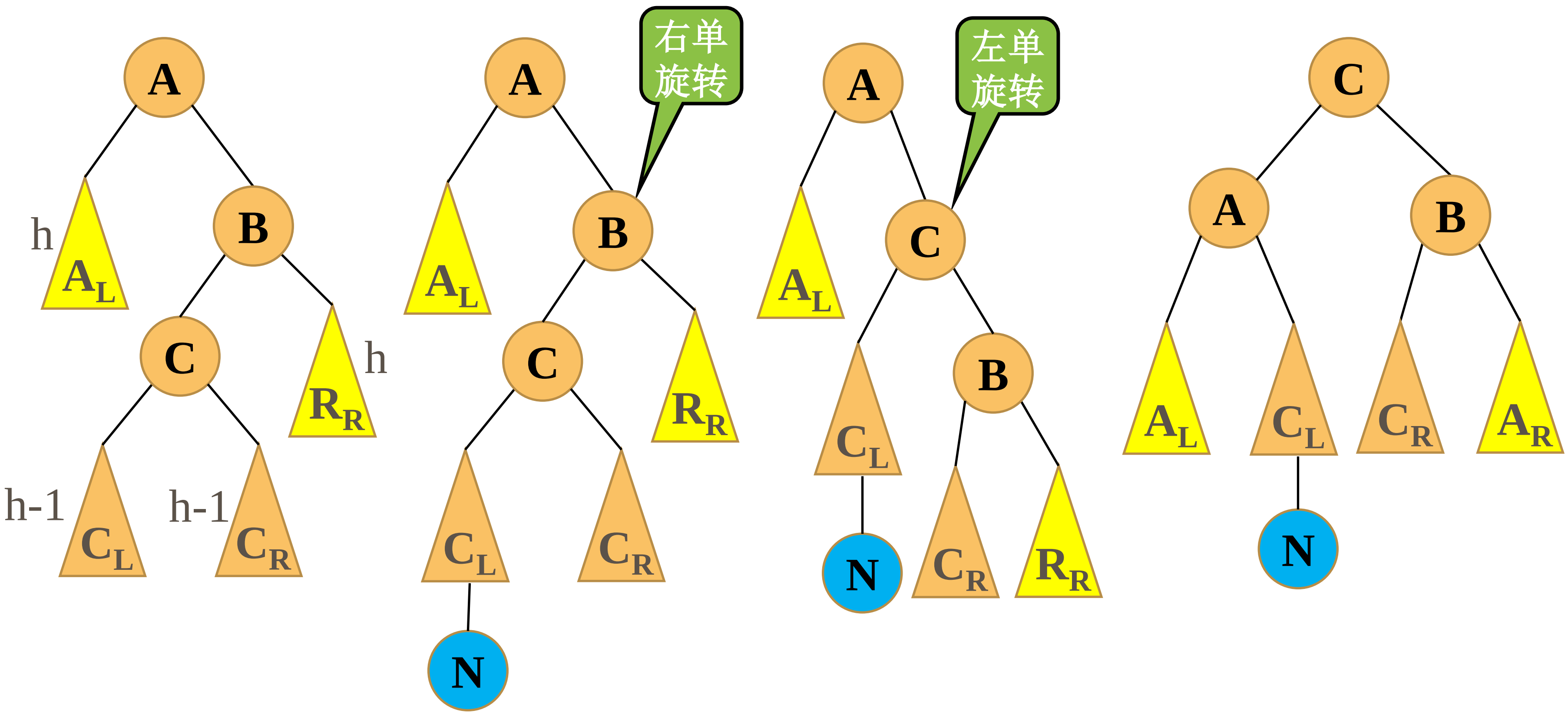
若在A的右子树的左子树上插入结点，使A的平衡因子从-1增加至-2，需要先进行顺时针旋转，再逆时针旋转。

旋转轴确定：沿着失衡路径，以失去平衡点的后二层结点为旋转轴。



9.2.4 平衡二叉树

4）RL平衡旋转（先右后左）



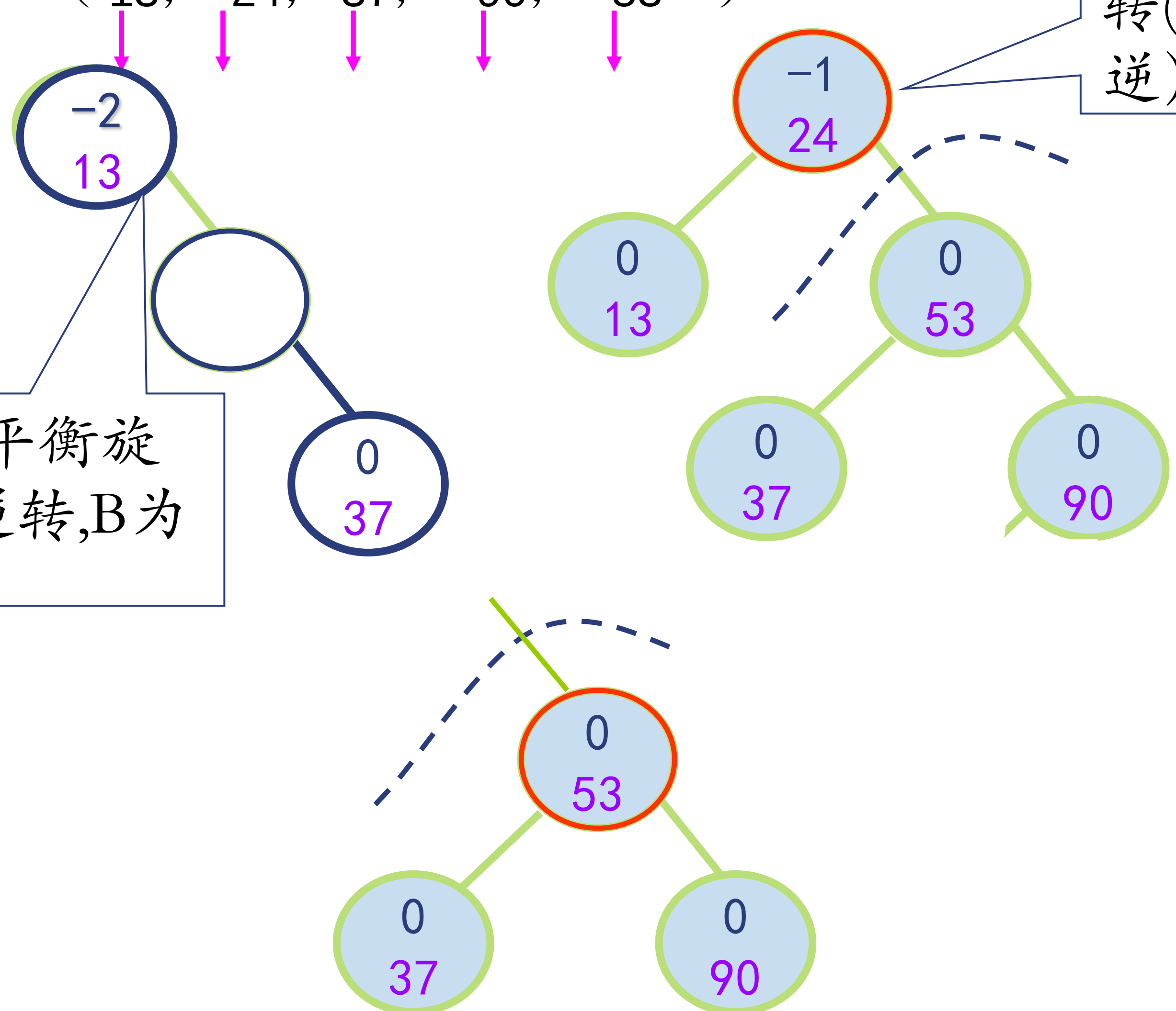
（1）　（2）　（3）　（4）

9.2.4 平衡二叉树

```
void RightBalance(BSTree &T){//右平衡旋转处理
    BSTree rc,ld;  rc=T->rchild;
    switch(rc->bf){
    case RH: // “RR型”
        T->bf=rc->bf=EH;
        L_Rotate(T); break;
    case LH: // “RL型”
        ld=rc->lchild;
        switch(ld->bf) {
        case RH:T->bf=LH;rc->bf=EH;break;
        case EH:T->bf=rc->bf=EH;break;
        case LH:T->bf=EH;rc->bf=RH;break;
        }
        ld->bf=EH;
        R_Rotate(T->rchild);
        L_Rotate(T);
    }
}
```

例：请将下面序列构成一棵平衡二叉排序树

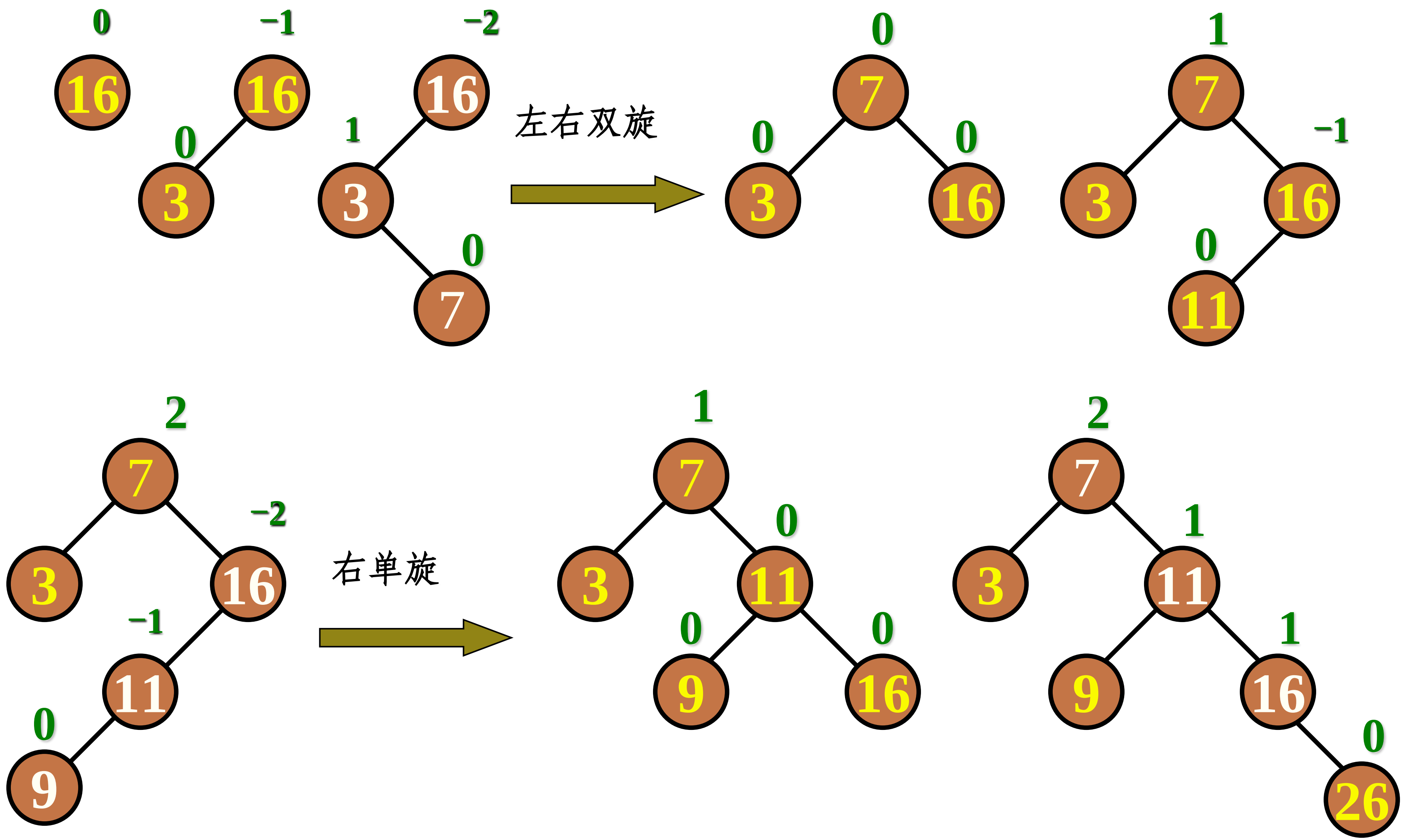
（ 13， 24， 37， 90， 53 ）



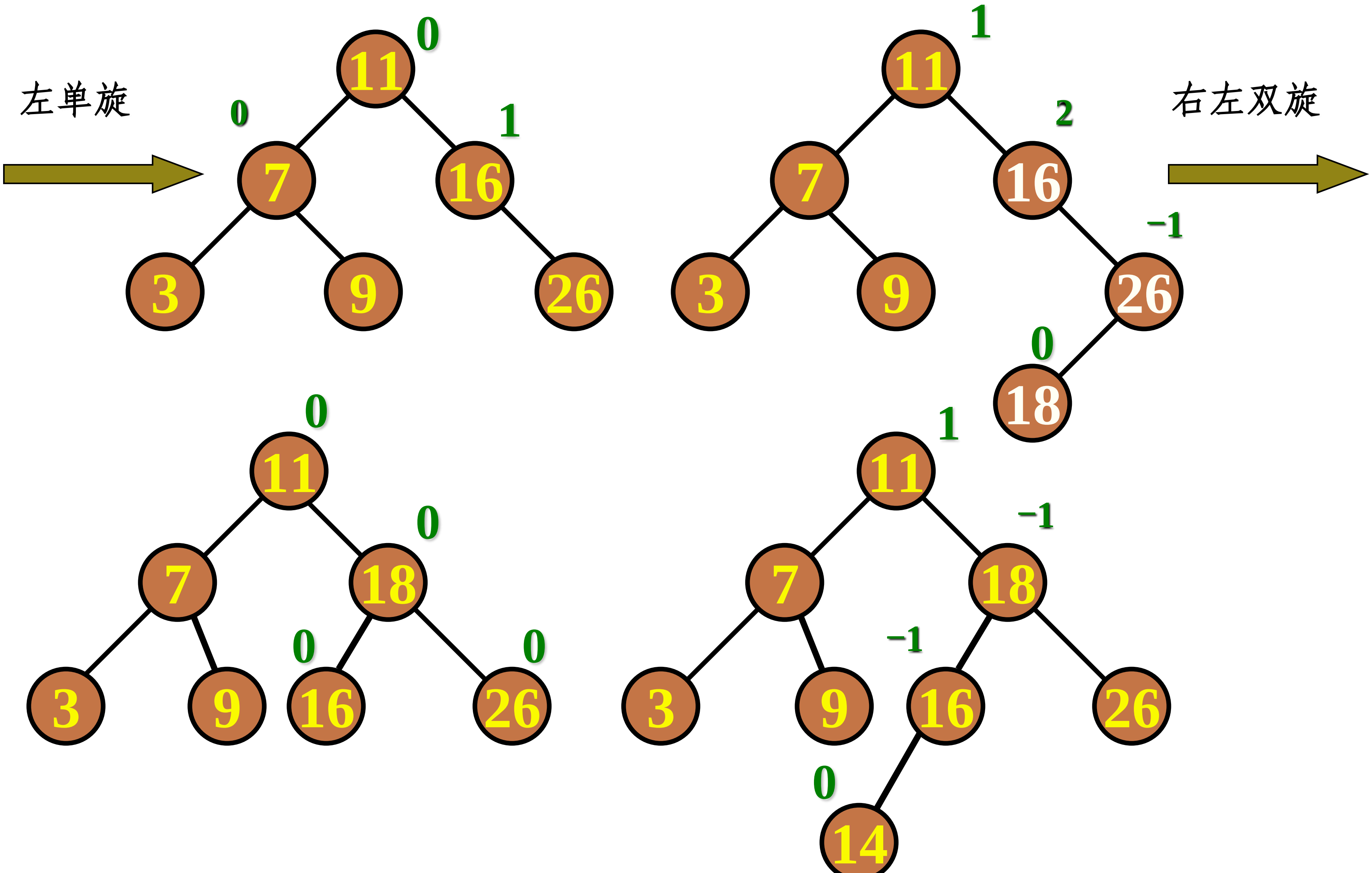
需要RR平衡旋转(绕B逆转,B为根)

需要RL平衡旋转(绕C先顺后逆)

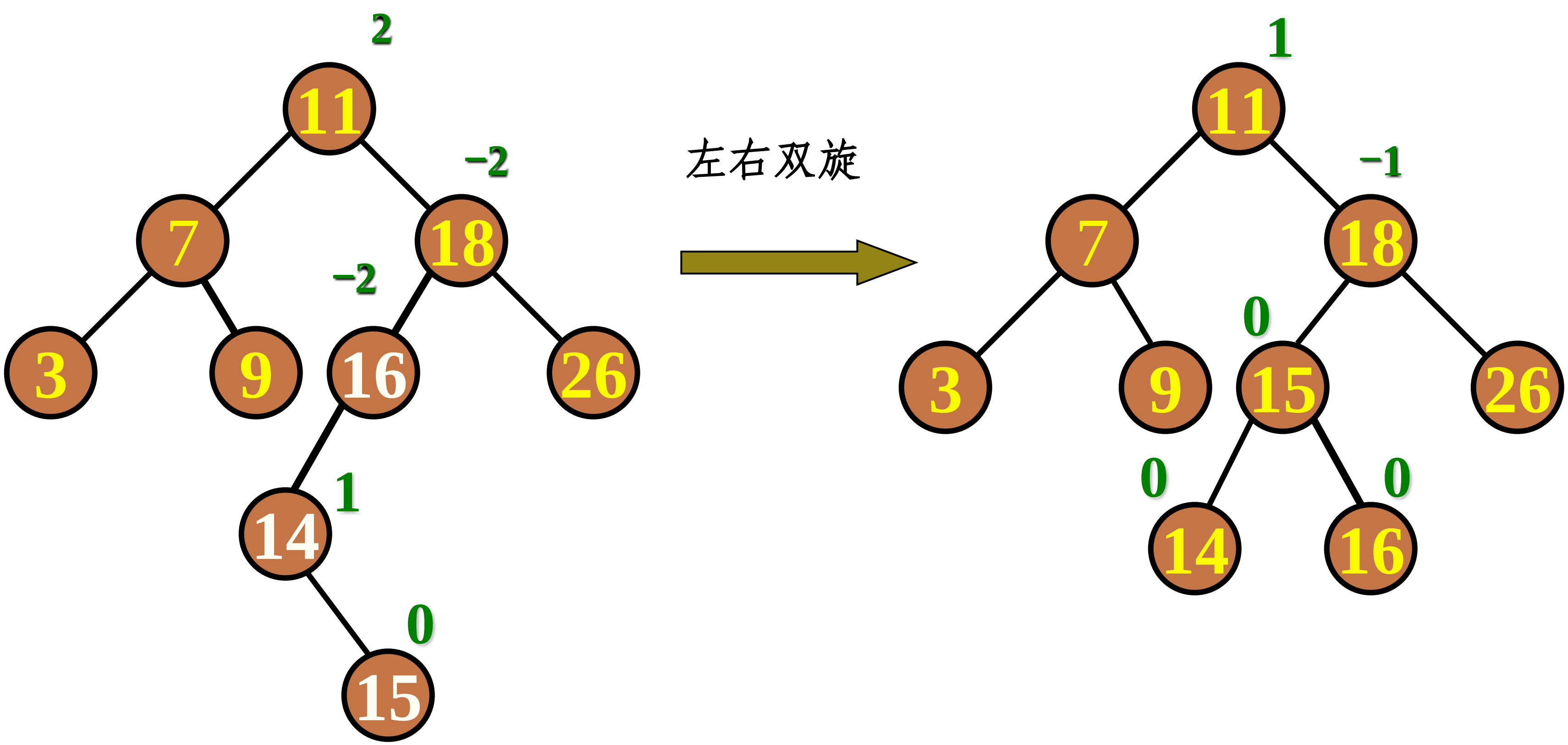
例，输入关键字序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，插入和调整过程如下。



例，输入关键字序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，插入和调整过程如下。



例，输入关键字序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，插入和调整过程如下。



9.2.4 平衡二叉树

- 下面的算法将通过递归方式将新结点作为叶结点插入并逐层修改各结点的平衡因子。
- 在发现不平衡时立即执行相应的平衡化旋转操作，使得树中各结点重新平衡化。
- 在程序中，若新结点存储分配成功，返回1，否则返回0。
- 算法从树的根结点开始，递归向下找插入位置。在找到插入位置(空指针)后，为新结点动态分配存储空间，将它作为叶结点插入，并将taller置为1，以表明插入成功。在退出递归沿插入路径向上返回时做必要的调整。若插入不成功，返回0。

```
Status InsertAVL(BSTree &T,ElemType e,bool &taller){
    if(T==NULL){//若为空树，插入一个数据元素为e的新节点作为BBST的根节点数的深度增1
        T=(BSTree)malloc(sizeof(BSTNode)); T->data=e;T->lchild=T->rchild=NULL; T->bf=EH;
        taller=true;
    }
    else{ //不为空树
        if(T->data==e){//待插关键字和BBST根节点关键字相同不进行插入
            taller=false; return 0;
        }
        else if(e<T->data){//小于且左子树中不存在关键字和待插关键字相同的的节点就插入
            if(InsertAVL(T->lchild,e,taller)==FAILED) return 0;//如插入不成功返回FAILED
            if(taller){//如果插入成功并且左子树长高了
                switch(T->bf){//BBST根节点的现状
                case LH://左子树高
                    LeftBalance(T);//左平衡处理
                    taller=false;//标记为未长高
                    break;
                case EH://平衡的
                    T->bf=LH;//标记为左高
                    taller=true;//标记为长高
                    break;
                case RH://右子树高
                    T->bf=EH;//标记为平衡
                    taller=false;//标记为未长高
                    break;
                }
            }
        }
```

```
        else{//大于且右子树中不存在关键字和待插关键字相同的的节点就插入
            if(InsertAVL(T->rchild,e,taller)==FAILED) return FAILED;
                //如果再右子树中插入不成功返回不成功信息
            if(taller){//插入成功就判断右子树是否长高
                switch(T->bf){//BBST根的现状
                case LH://左子树高
                    T->bf=EH;
                    taller=false;
                    break;
                case EH://平衡
                    T->bf=RH;
                    taller=true;
                    break;
                case RH://右子树高
                    RightBalance(T);
                    taller=false;
                    break;
                }
            }
        }
    }
    return 1;
}
```