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

回顾

在查找过程中关键字的平均比较次数或平均读写磁盘次数

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

静态查找表——只查找，不改变集合内的数据元素

动态查找表——既查找，又改变（增减）集合内的数据元素

回顾

静态查找

- ✓ 顺序查找（线性查找）

$$ASL = \frac{n+1}{2}$$

- ✓ 折半查找（二分或对分查找）

$$ASL \approx log_2 n$$

- ✓ 分块查找（索引顺序查找）

$$ASL = \frac{1}{2}\left(\frac{n}{s} + s\right) + 1 \qquad ASL \approx log_2\left(\frac{n}{s} + 1\right) + \frac{s}{2}$$

回顾

- 动态查找

典型的动态表———二叉排序树

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。

9.2.1、二叉排序树

1 二叉排序树查找

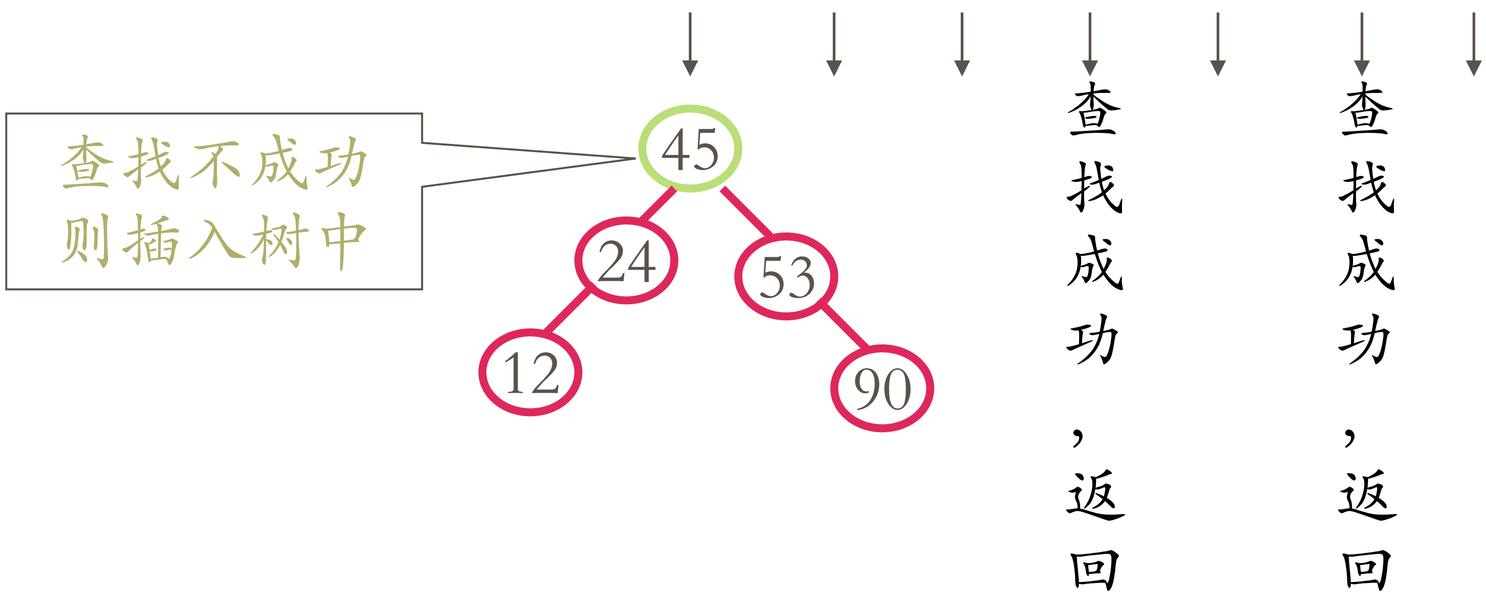
```
BiTree SearchBST(BiTree T, KeyType key) {
    if((!T) || EQ(key,T->data.key))
        return(T);
    else if LT(key,T->data.key)
        return(SearchBST(T->lchild,key));
    else
        return(SearchBST(T->rchild,key));
}
```

9.2.1、二叉排序树

2 二叉排序树的插入操作如何实现?

思路：查找不成功，生成一个新结点s，插入到二叉排序树中；查找成功则返回。

输入待查找的关键字序列=（45，24，53，45，12，24，90）



9.2.1、二叉排序树

修改查找算法

```
Status SearchBST(BiTree T,KeyType key, BiTree f,BiTree &p){
    if(!T) { p=f;  return TRUE; }
    else if EQ(key,T->data.key){p=T; recurn TRUE ;}
    else if LT(key,T->data.key)
        return SearchBST(T->lchild,key,T,p);
    else
        return SearchBST(T->rchild,key,T,p);
}
```

9.2.1、二叉排序树

```
Status InsertBST(BiTree &T,ElemType e){
    if(!SearchBST(T,e.key,NULL,P){
        s=(BiTree)malloc(sizeof(BiTNode));
        s->data=e; s->lchild=s->rchild=NULL :
        if(!p)T= s;
        else if LT(e.key,p->data.key) p->lchild=s;
        else p->rchild=s;
        return TRUE;
    }
    else return FALSE;
}
```

9.2.1、二叉排序树

```
void Insert_BST( BiTree &T, BiTree S ){
    BiTree  p, q;
    if(!T) T=S;
    else{
        p=T;
        while(p){
            q = p;
            if(S->data.key < p->data.key) p=p->lchild;
            else p=p->rchild;
        }
        if(S->data.key < q->dat.key) q->lchild = S;
        else q->rchild = S;
    }
    return;
}
```

9.2.1、二叉排序树

输入一组数据元素的序列，构造二叉排序树的算法

```
void Creat_BST( BiTree &T ) {
   int x; BiTree S; T=NULL;
   while ( scanf ( "%d" ,&x), x!=0 ) {
      S = (BiTNode *) malloc(sizeof(BitNode));
      S->data.key = x;
      S->lchild = NULL;
      S->rchild = NULL;
      Insert_BST( T,S );
   }
   return;
}
```

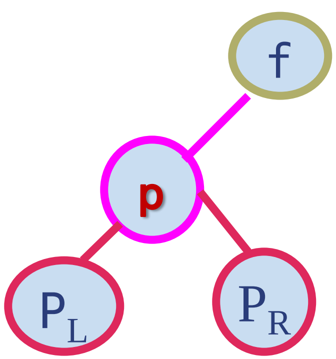
9.2.1、二叉排序树

3 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。

假设：*p表示被删结点的指针； P_L和P_R 分别表示*p的左、右孩子指针；

*f表示*p的双亲结点指针；并假定*p是*f的左孩子；则可能有三种情况：



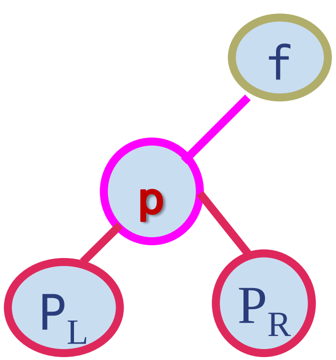
9.2.1、二叉排序树

3 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：情况复杂 →



9.2.1、二叉排序树

难点：*p有两棵子树时，如何进行删除操作?

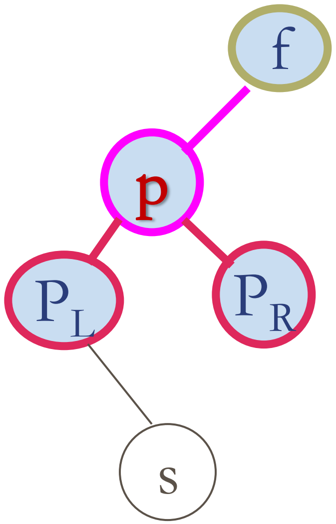
分析:

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。



9.2.1、二叉排序树

难点：*p有两棵子树时，如何进行删除操作?

分析:

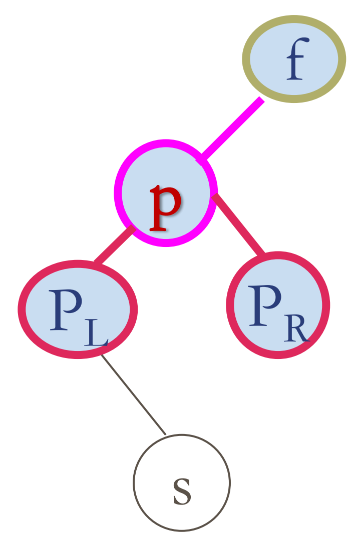
有两种解决方法:

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 $f_L = P_L$； $S_R = P_R$ ；

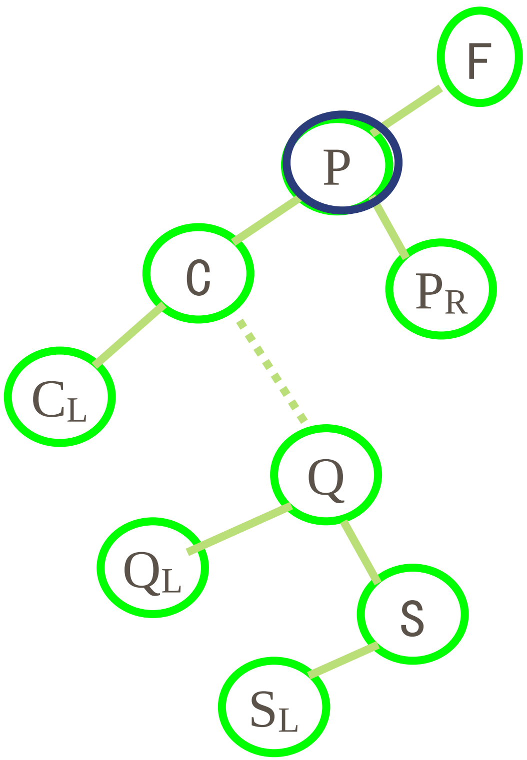
法2：直接令s代替p，s的左子树接为P_L

// s为p左子树最右下方的结点

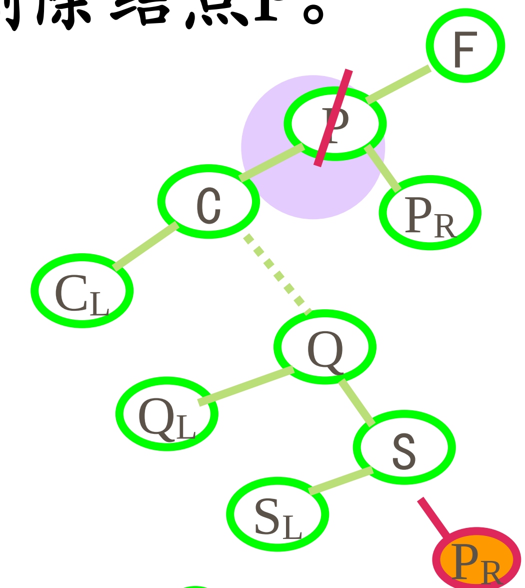


9.2.1、二叉排序树

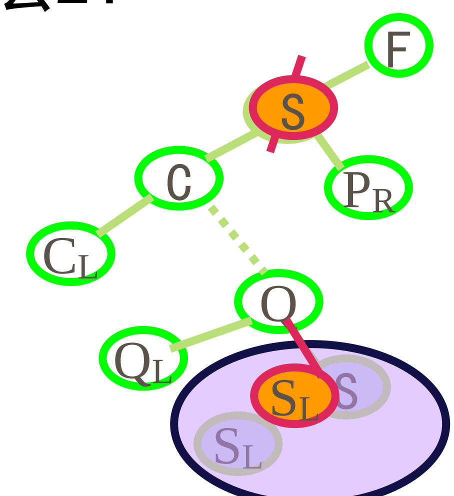
例： 请从下面的二叉排序树中删除结点P。



法1：



法2：

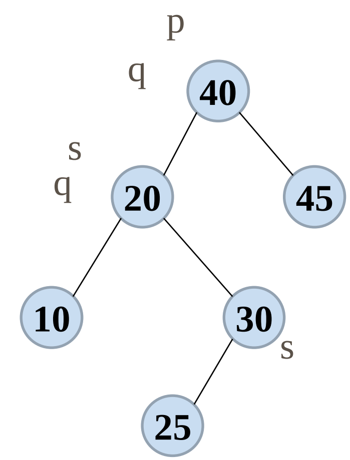


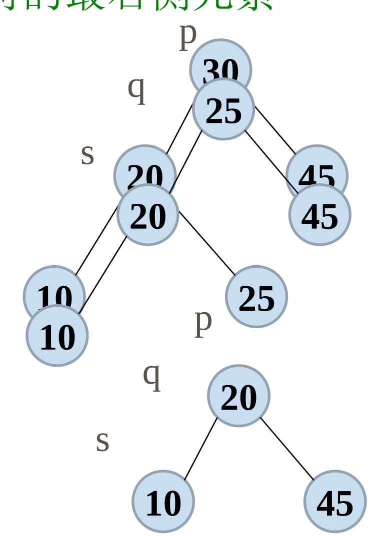
9.2.1、二叉排序树

```
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->lchild,key);
   else
      return DeleteBST(T->rchild,key);
  }
}
```

9.2.1、二叉排序树

```
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;
}
```



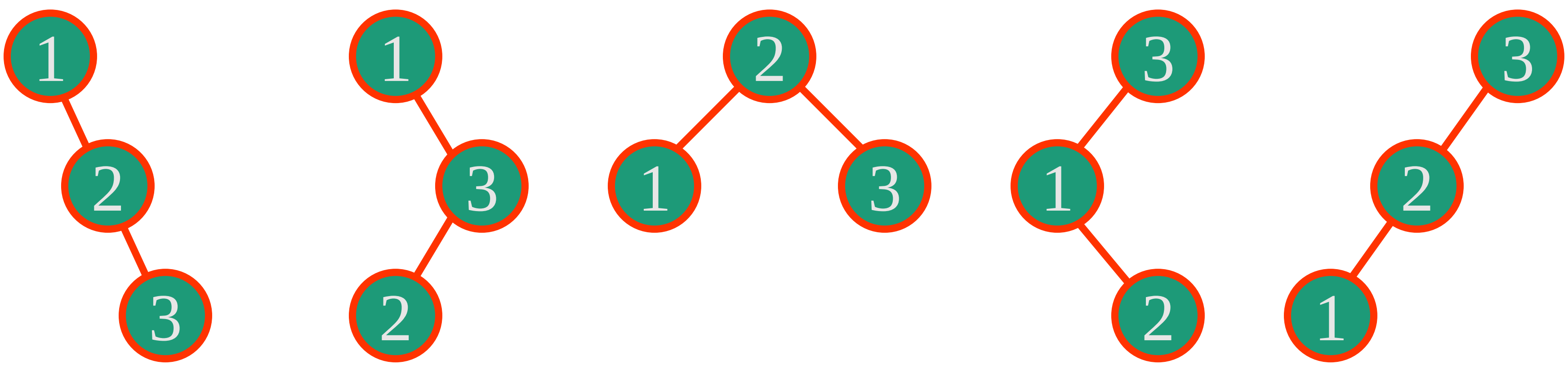


9.2.1、二叉排序树

4 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}



9.2.1、二叉排序树

4 二叉排序树的查找分析

(1) 二叉排序树上查找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 $\lfloor \log_2 n \rfloor + 1$

平均查找长度:

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

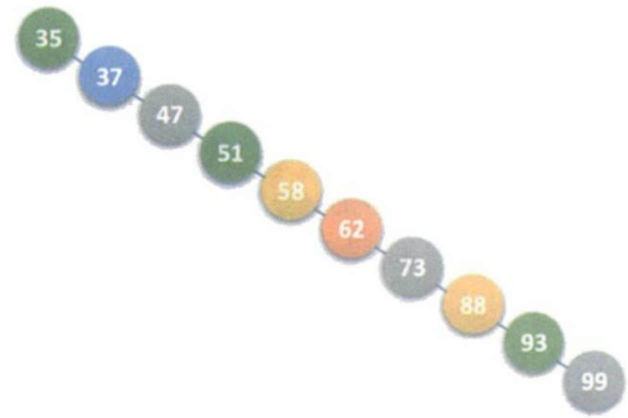
n_i 是每层结点个数;

C_i 是结点所在层次数;

m 为树深。

9.2.1、二叉排序树

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。

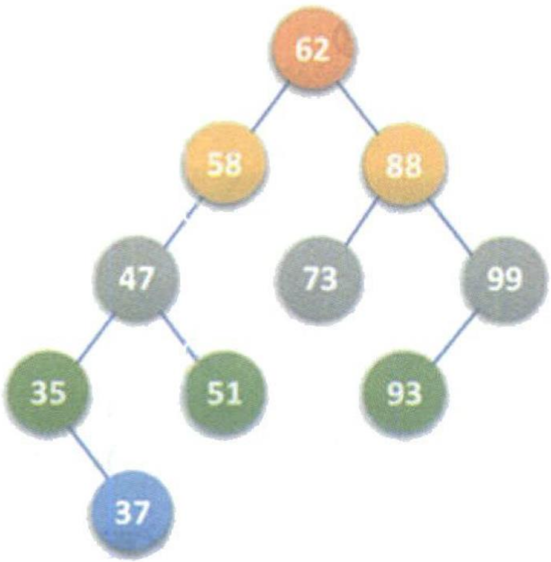
一般情况：$ASL \le 2(1+\frac{1}{n})\ln n$

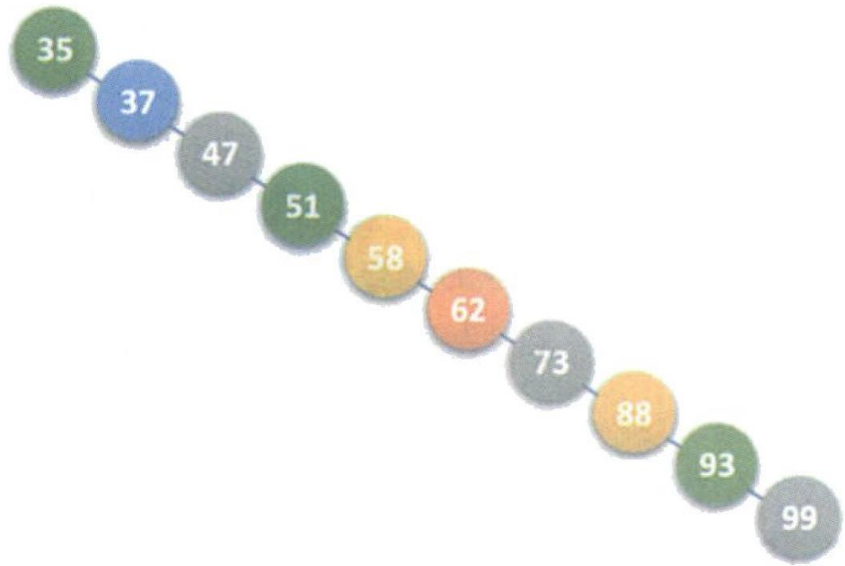
（与 log n 同阶）

9.2.1、二叉排序树

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

9.2.2 平衡二叉树

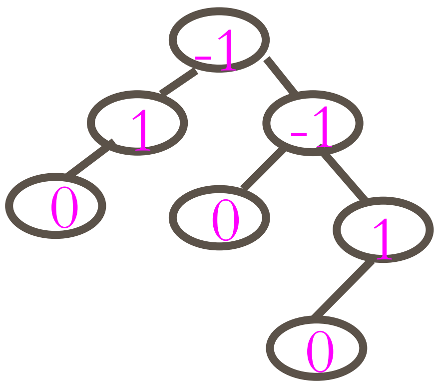
平衡二叉树或者是一棵空树，或者是具有下列性质的二叉树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1.

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

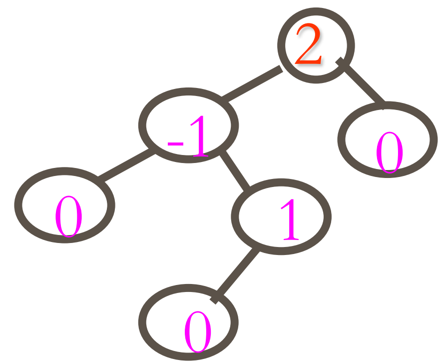
- **平衡二叉树上所有结点的平衡因子只可能是-1,0,和1.**

9.2.2 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

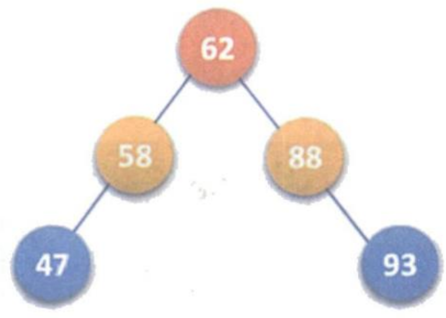
- 对于一棵有n个结点的AVL树，其高度保持在$O(\log_2 n)$数量级，ASL也保持在$O(\log_2 n)$量级。

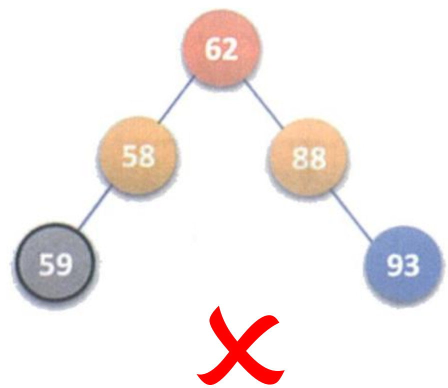
9.2.2 平衡二叉树

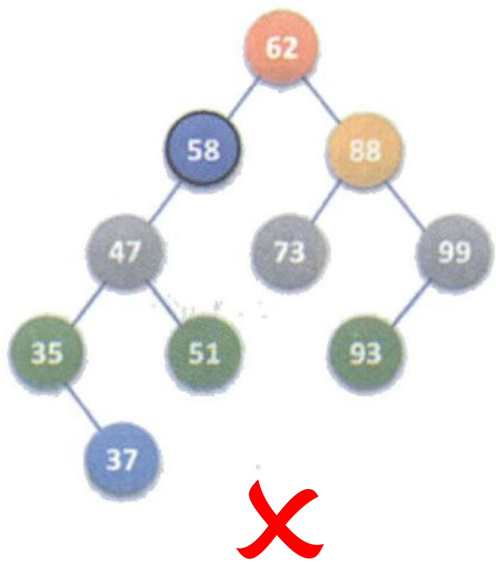
```
typedef struct BSTNode {
    ElemType  data;
    int bf;     //结点的平衡因子
    struct BSTNode *lchild,*rchild;
}BSTNode ,*BSTree;
```

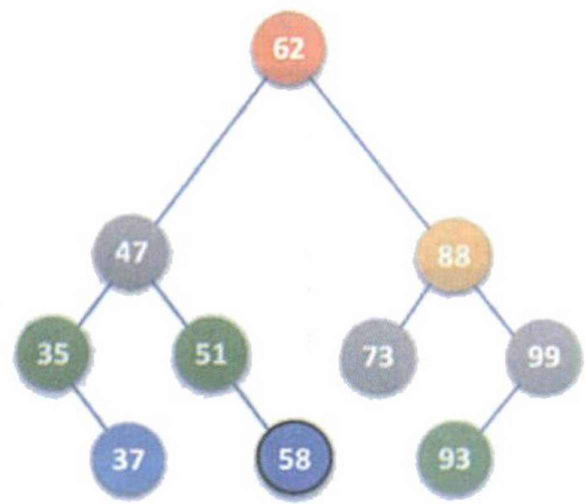
```
void R_Rotate(BSTree &p);
void L_Rotate(BSTree &p);
Status InsertAVL(BSTree &T,ElemType e,Boolean &taller);
void LeftBalance(BSTree &T);
void RightBalance(BSTree &T);
```

9.2.2 平衡二叉树


62
58
88
47
93


62
58
88
59
93

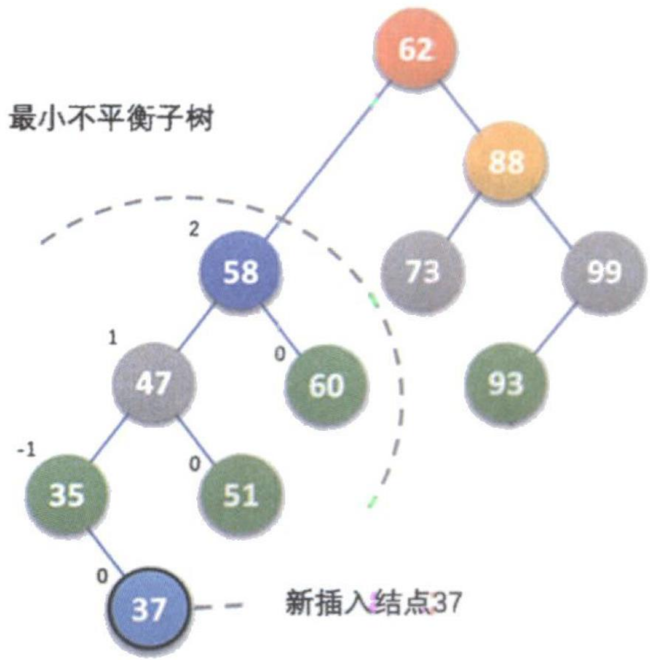

62
58
88
47
73
99
35
51
93
37


62
47
88
35
51
73
99
37
58
93

9.2.2 平衡二叉树

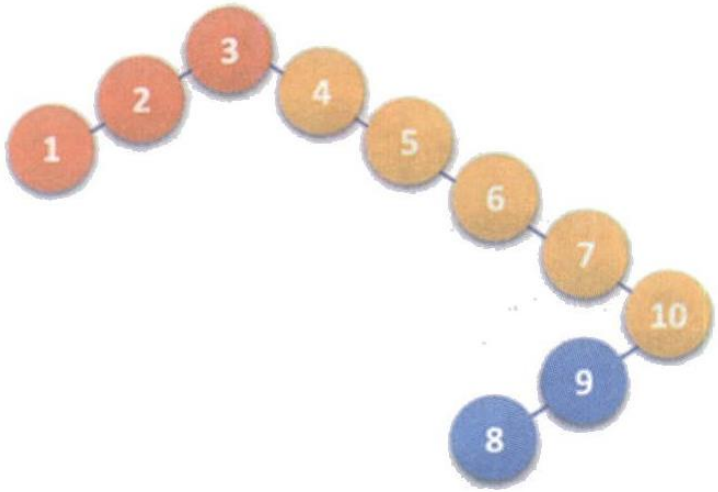
距离插入结点最近的，且平衡因子的绝对值大于1的结点为很的子树，我们称为**最小不平衡子树**

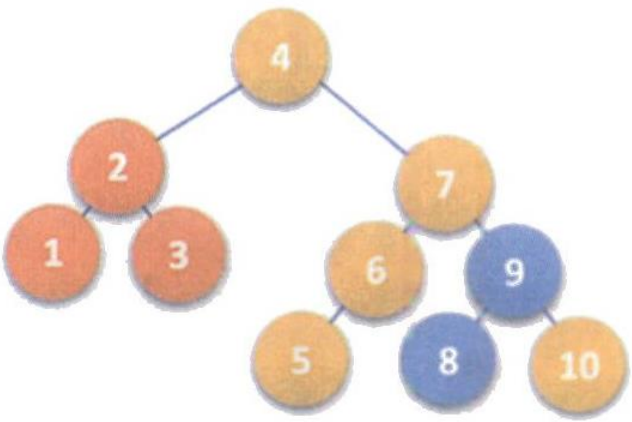
如果在一棵AVL树中插入一个新结点，就有可能造成失衡，此时必须重新调整树的结构，使之恢复平衡。我们称调整平衡过程为平衡旋转。



9.2.2 平衡二叉树

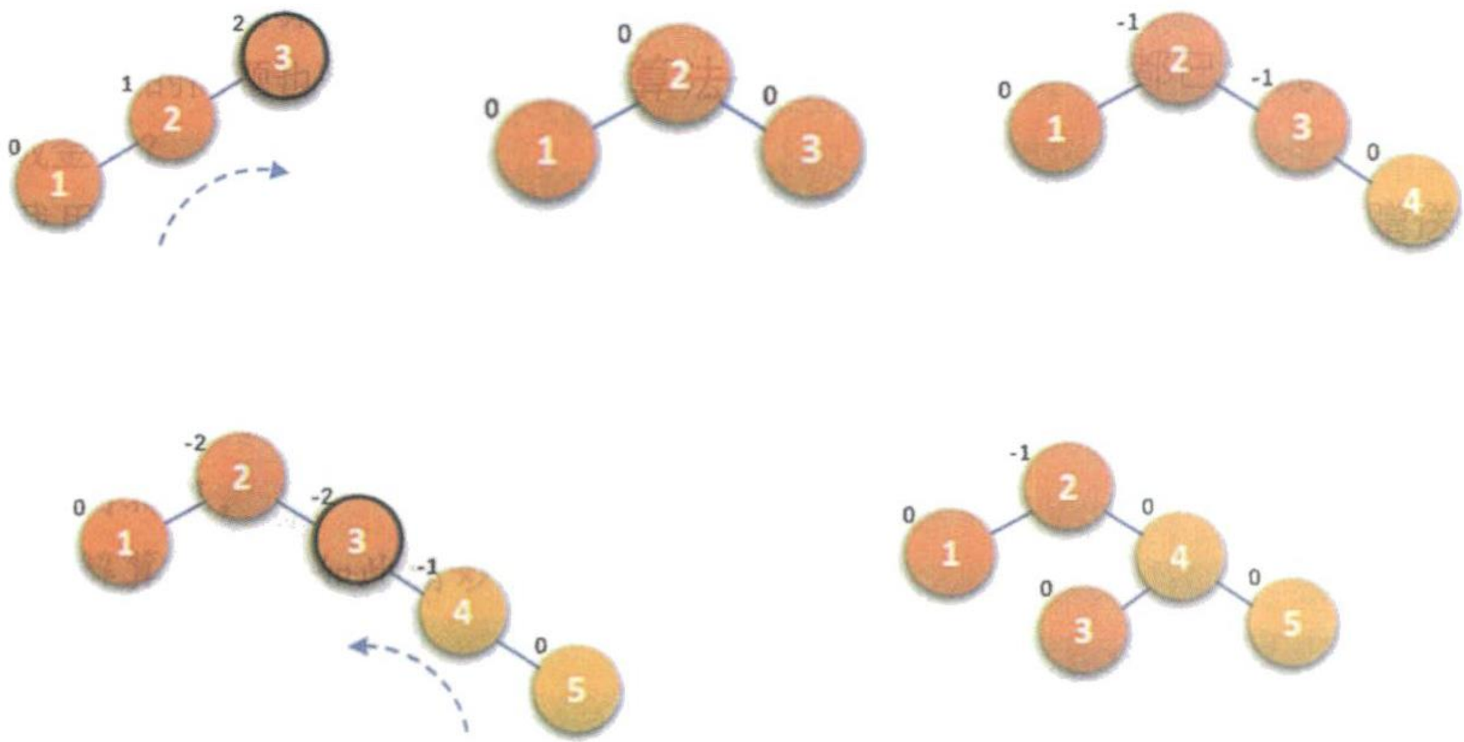
a[10] = {3,2,1,4,5,6,7,10,9,8}





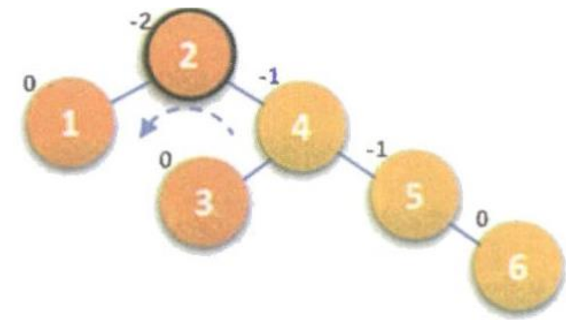
9.2.2 平衡二叉树

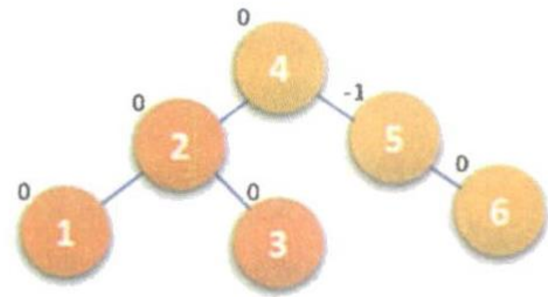
a[10] = {**3,2,1**,4,5,6,7,10,9,8}

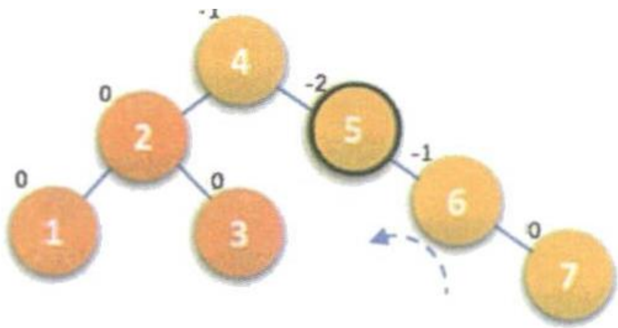


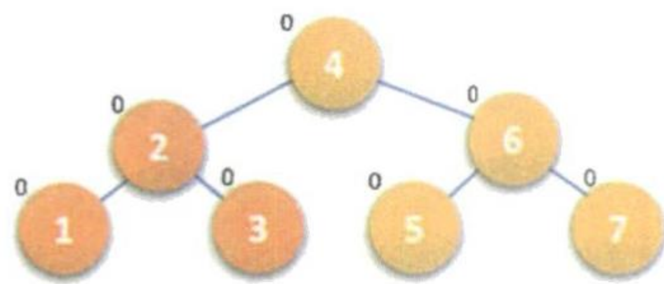
9.2.2 平衡二叉树

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



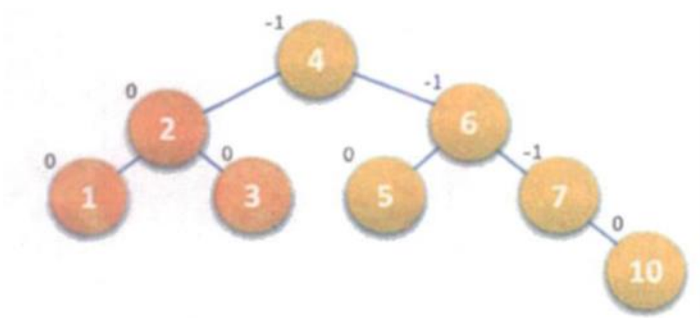


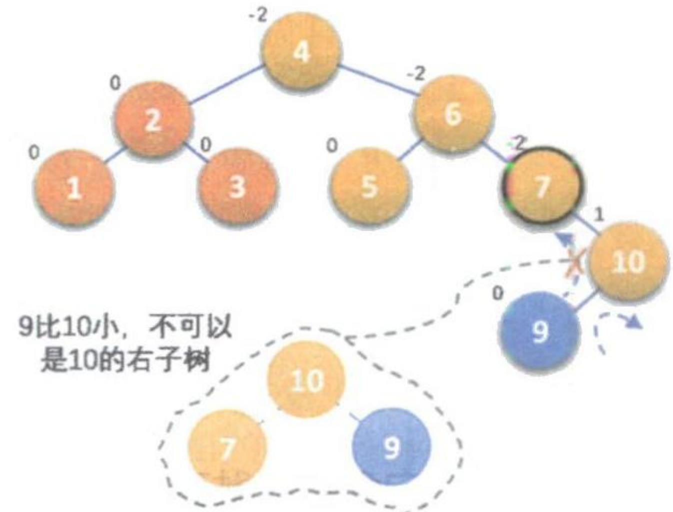


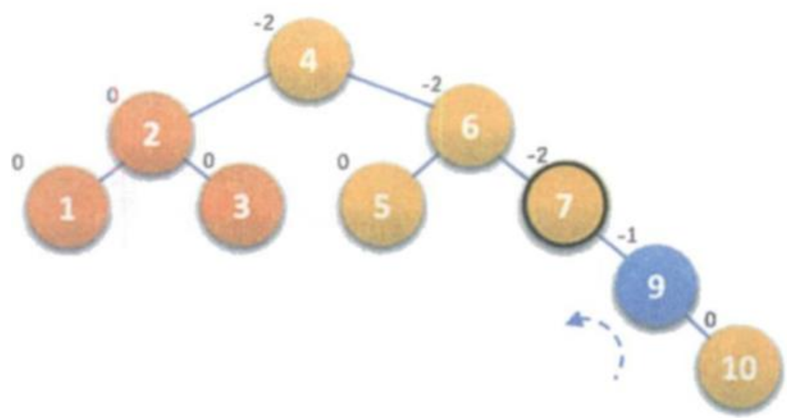


9.2.2 平衡二叉树

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

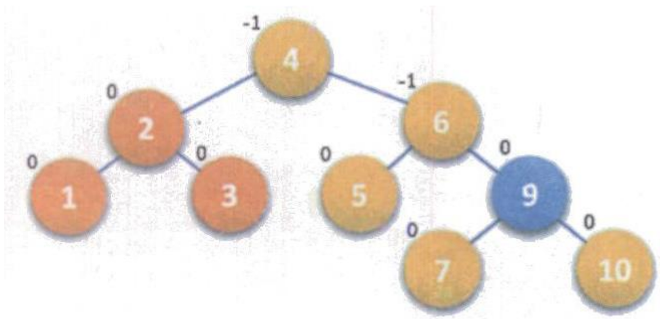


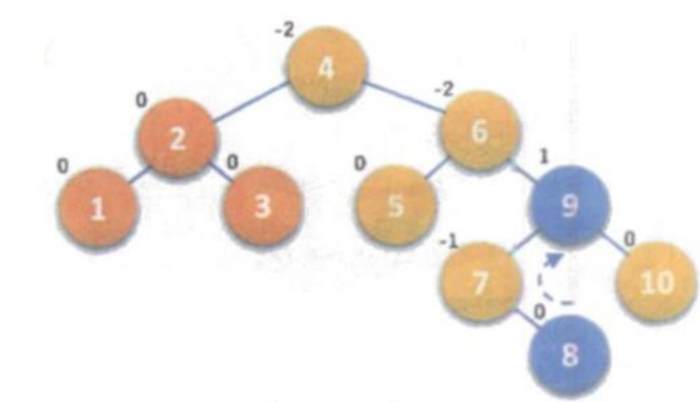


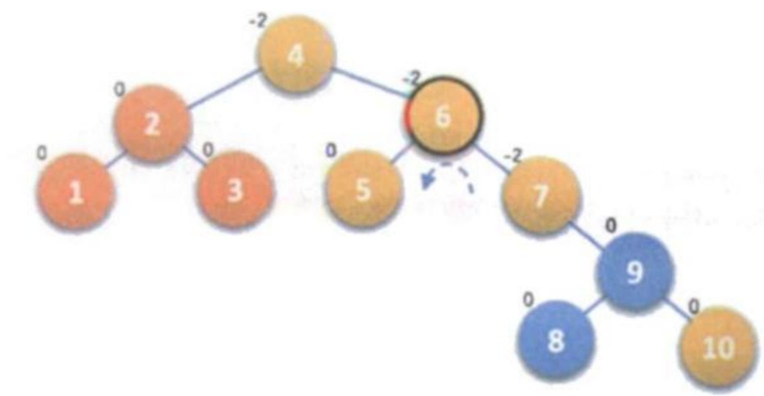


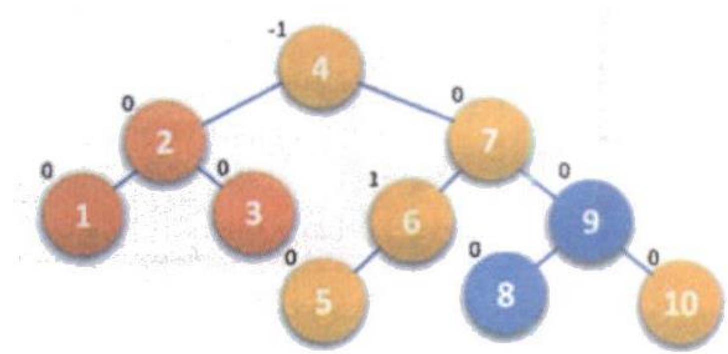
9.2.2 平衡二叉树

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









9.2.2 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

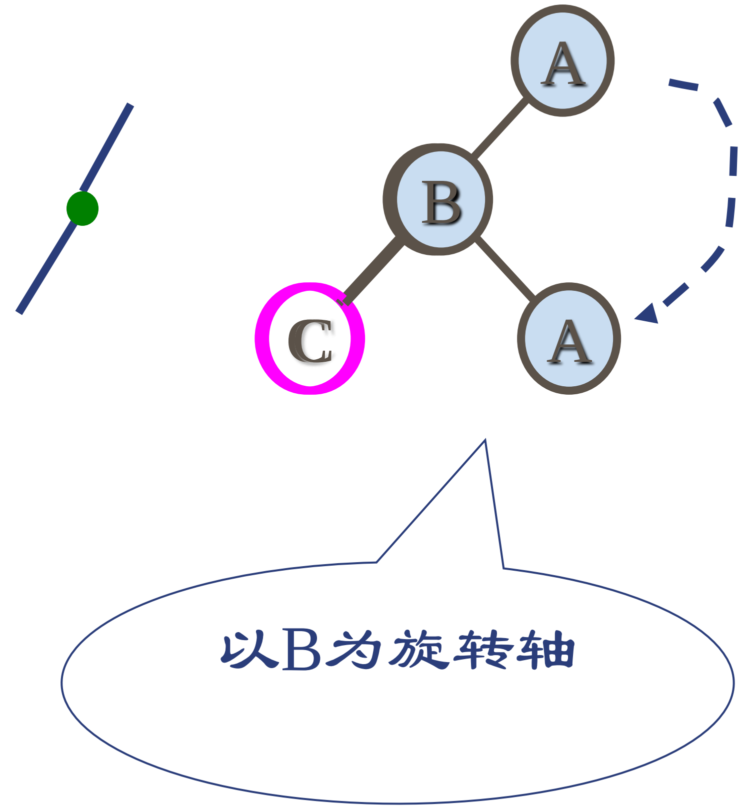
现分别介绍这四种平衡旋转。

9.2.2 平衡二叉树

1）LL平衡旋转（单右旋）

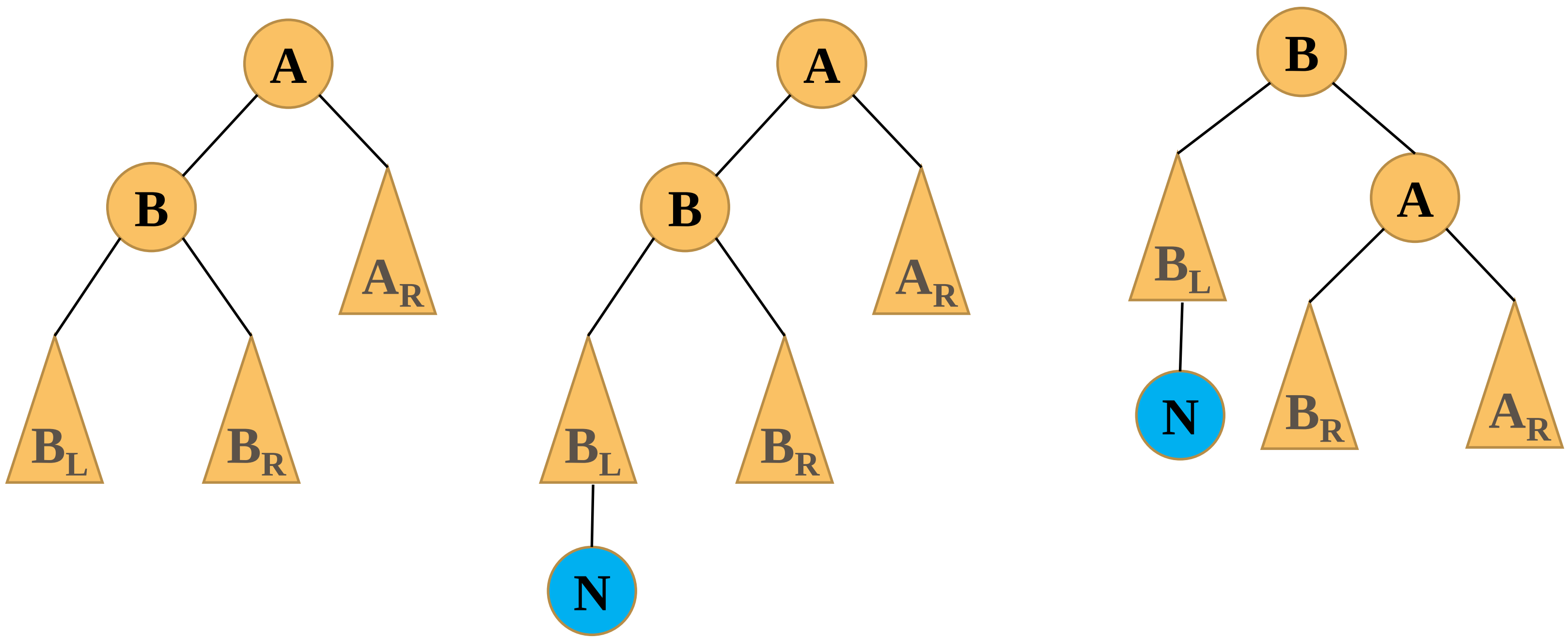
若在A的左子树的左子树上插入结点，使A的平衡因子从1增加至2，需要进行一次顺时针旋转。

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。



9.2.2 平衡二叉树

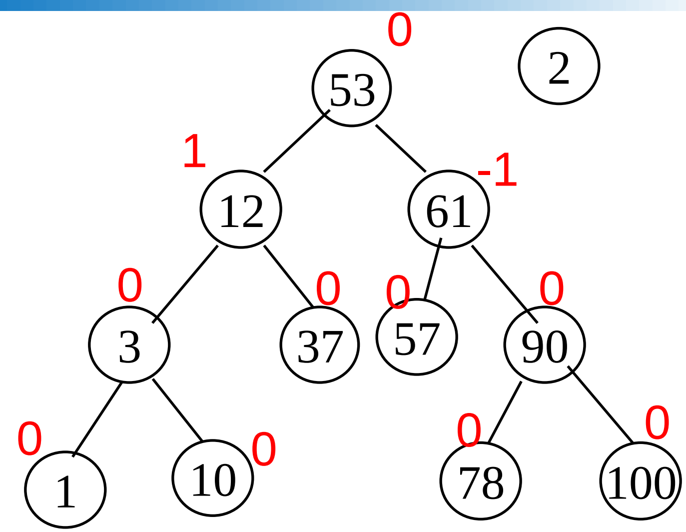
1）LL平衡旋转（单右旋）

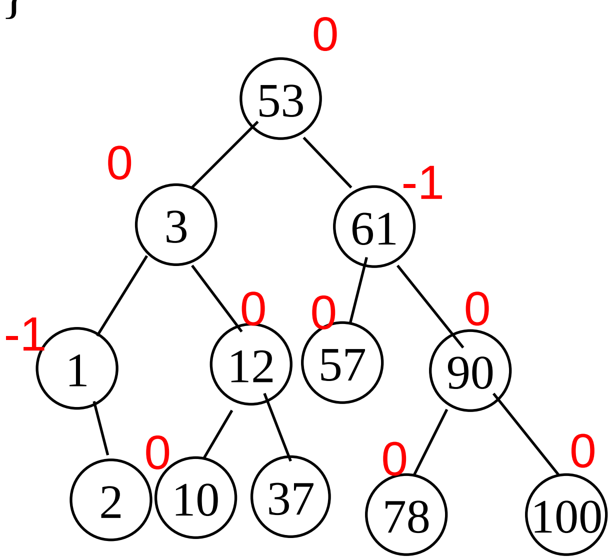


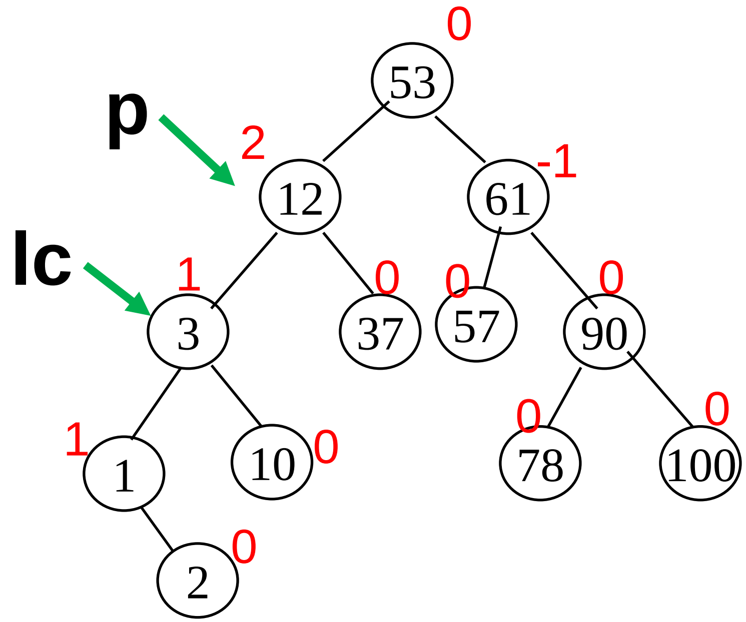
插入N前是平衡二叉树　　插入N后平衡性打破　　调整后平衡性恢复

9.2.2 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;
}
```





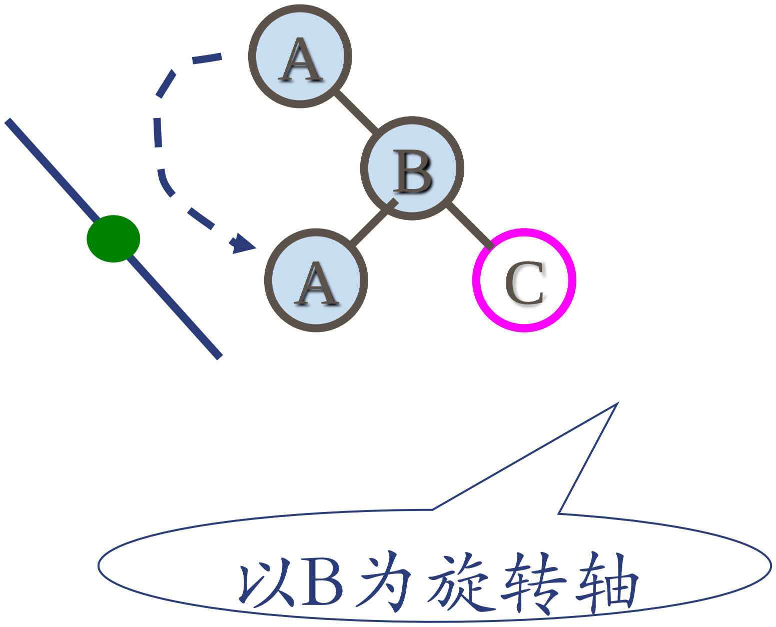


9.2.2 平衡二叉树

2）RR平衡旋转（单左旋）

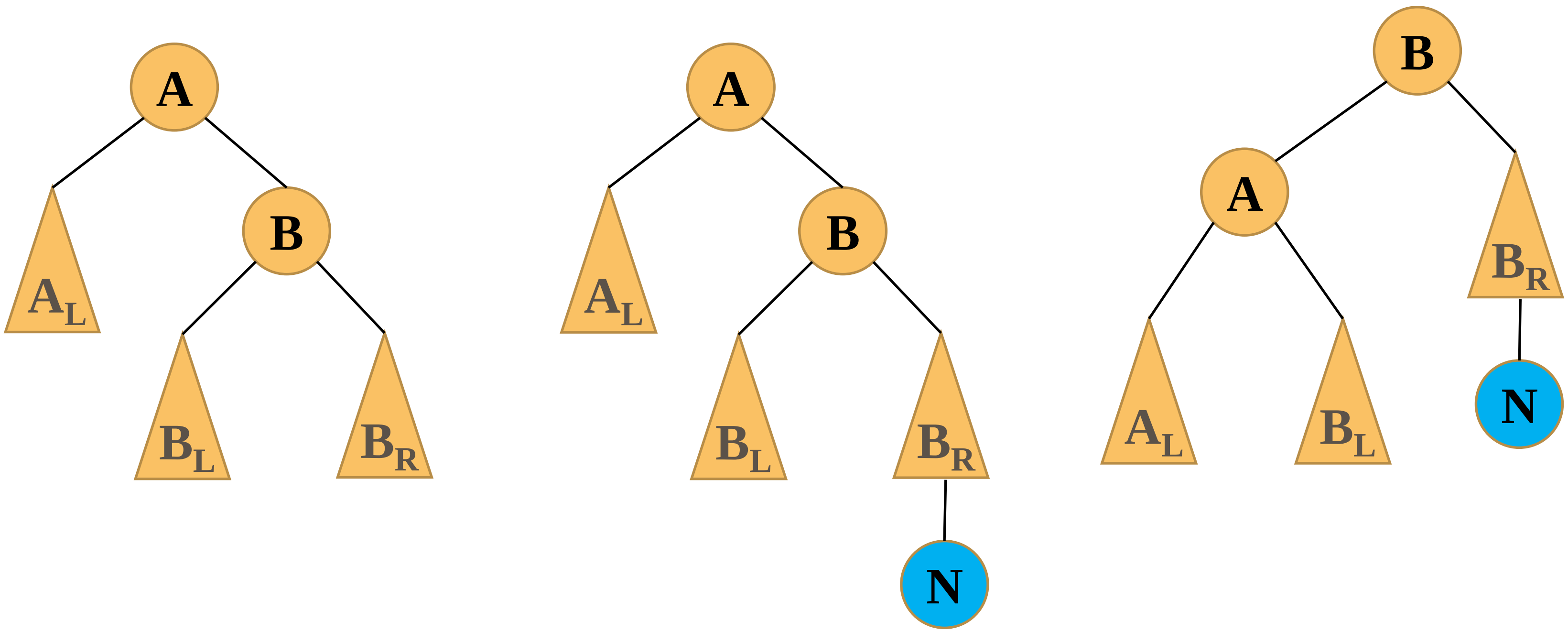
若在A的右子树的右子树上插入结点，使A的平衡因子从-1增加至-2，需要进行一次逆时针旋转。

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。



9.2.2 平衡二叉树

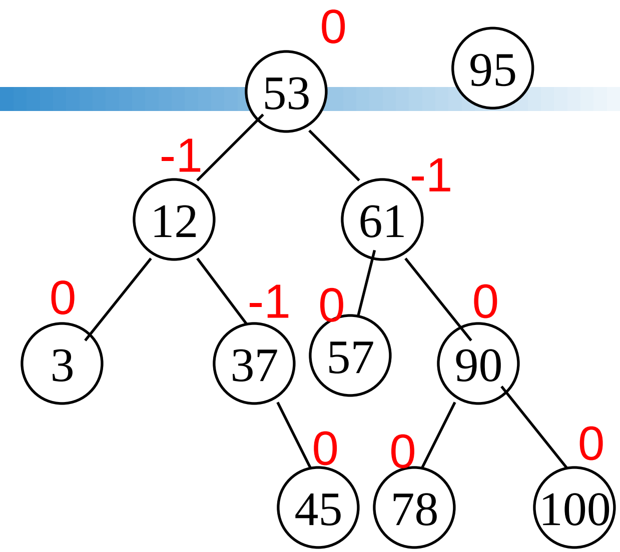
2）RR平衡旋转（单左旋）

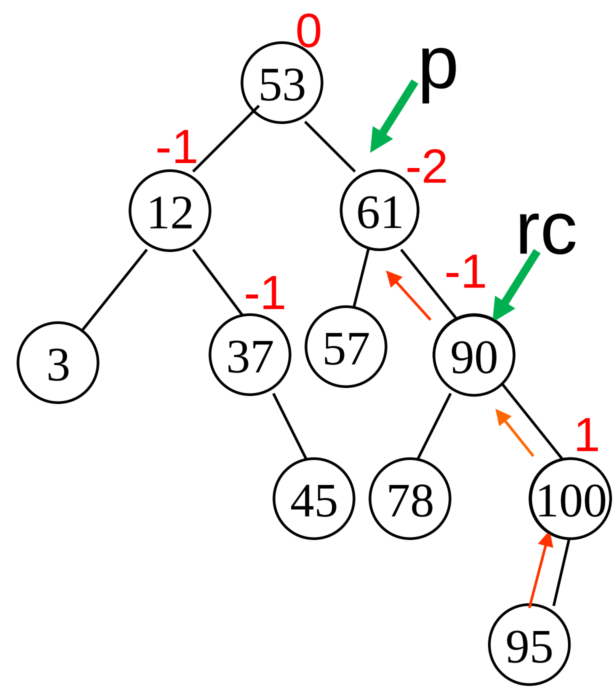


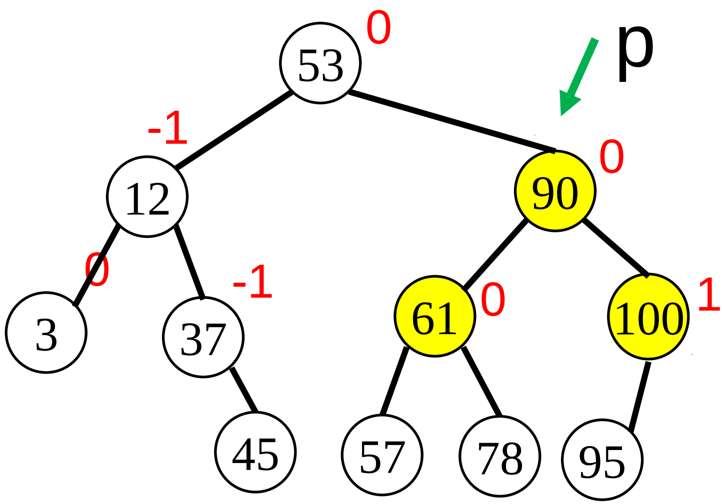
插入N前是平衡二叉树　　插入N后平衡性打破　　调整后平衡性恢复

9.2.2 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;
}
```





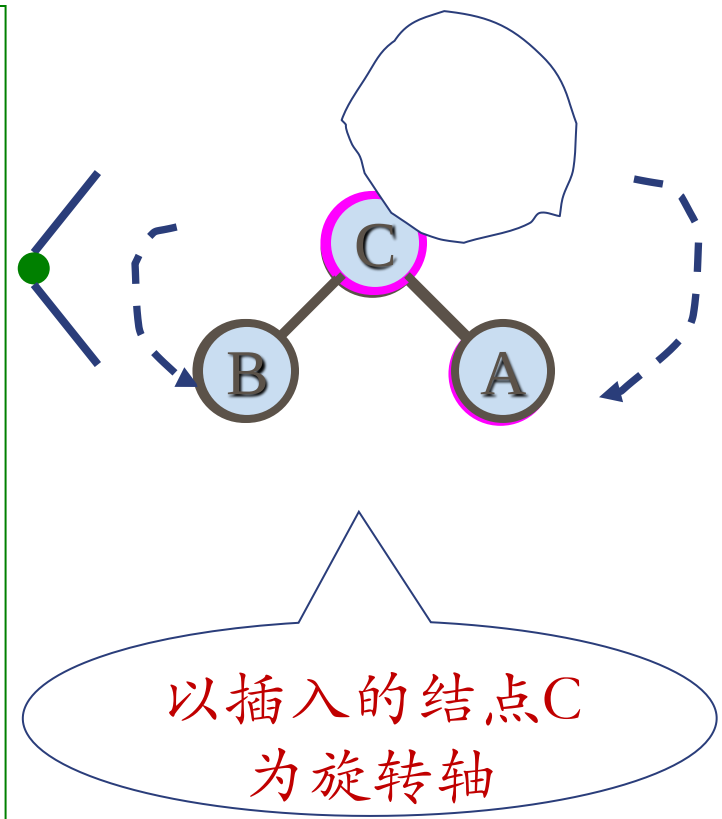


9.2.2 平衡二叉树

3）LR平衡旋转（先左后右）

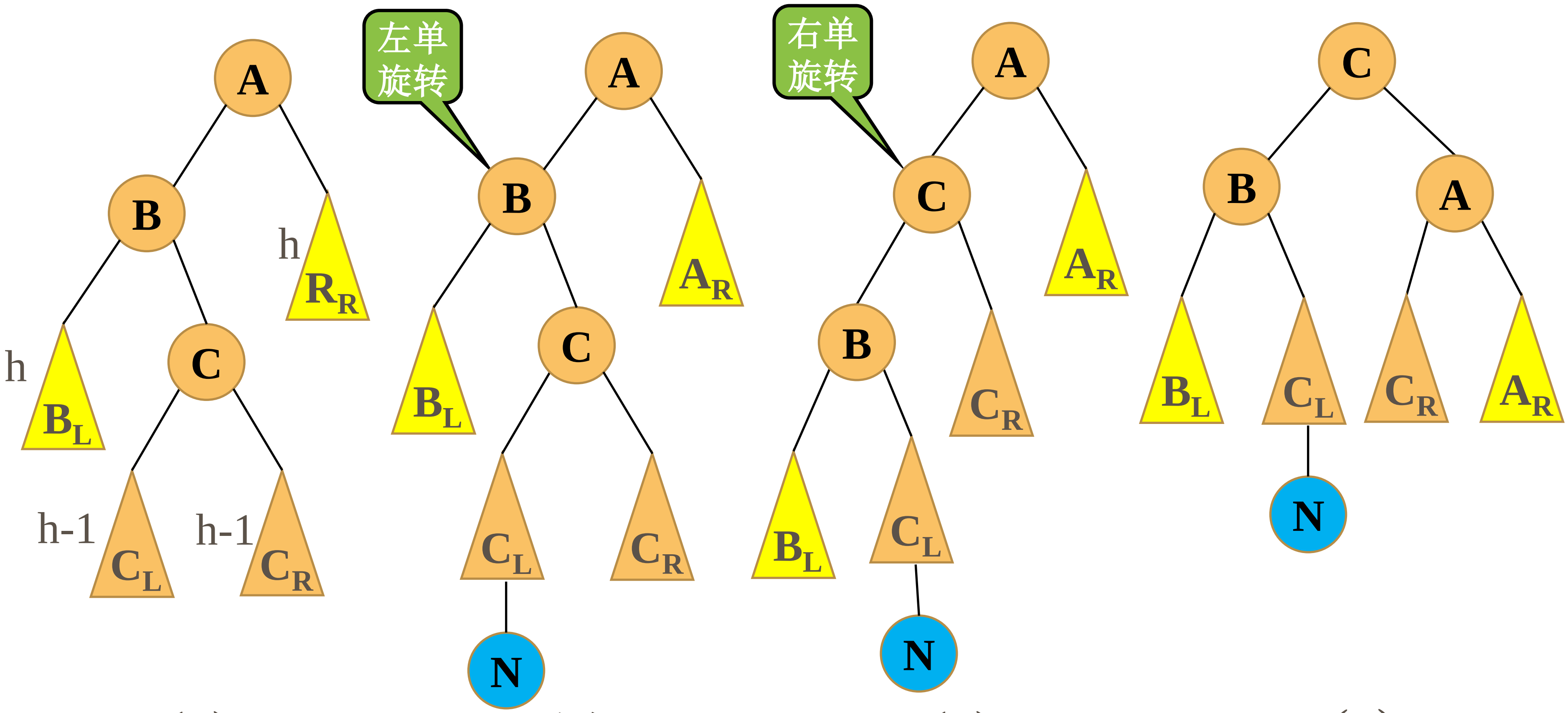
若在A的左子树的右子树上插入结点，使A的平衡因子从1增加至2，需要先进行逆时针旋转，再顺时针旋转。

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。



9.2.2 平衡二叉树

3）LR平衡旋转（先左后右）



（1）　（2）　（3）　（4）

9.2.2 平衡二叉树

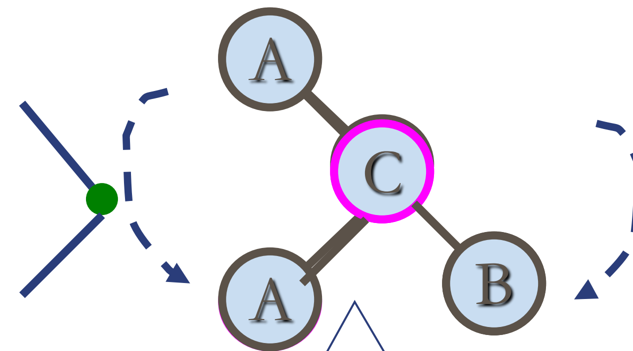
```
void LeftBalance(BSTree &T){//左平衡化的算法
    T->bf=LH
    BSTree lc,rd;   lc=T->lchild; //lc指向T的左子树
    switch(lc->bf){ //检查lc平衡度
    case LH: //LL型
        T->bf = lc->bf = EH; //新插入在T的左孩子的左子树
        R_Rotate(T); break; //单右旋
    case RH:    //新插入在T的左孩子的右子树
        rd=lc->rchild; //rd指向T的左孩子的右子树
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

9.2.2 平衡二叉树

4）RL平衡旋转（先右后左）

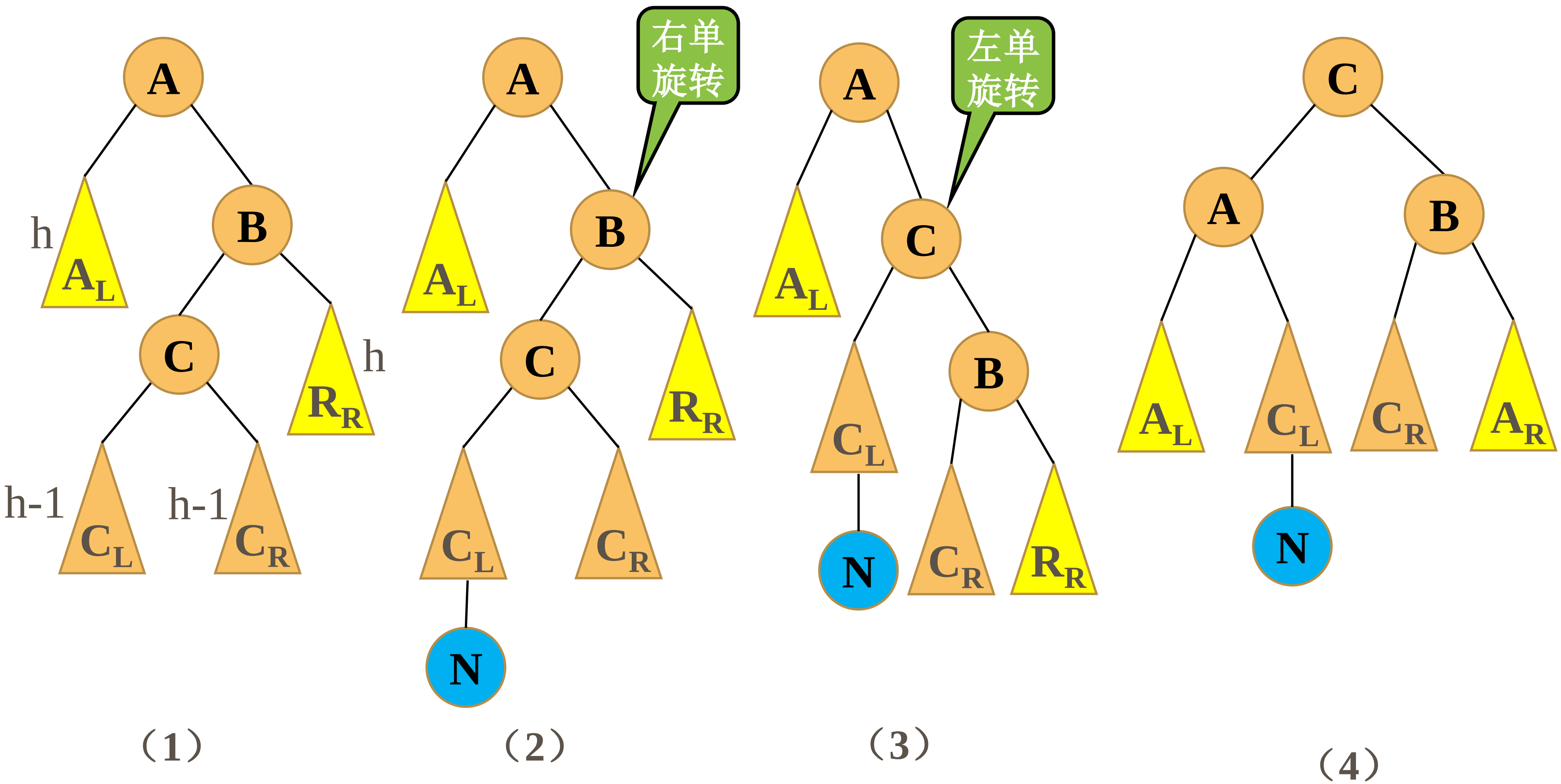
若在A的右子树的左子树上插入结点，使A的平衡因子从-1增加至-2，需要先进行顺时针旋转，再逆时针旋转。

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。



以插入的结点C为旋转轴

4）RL平衡旋转（先右后左）

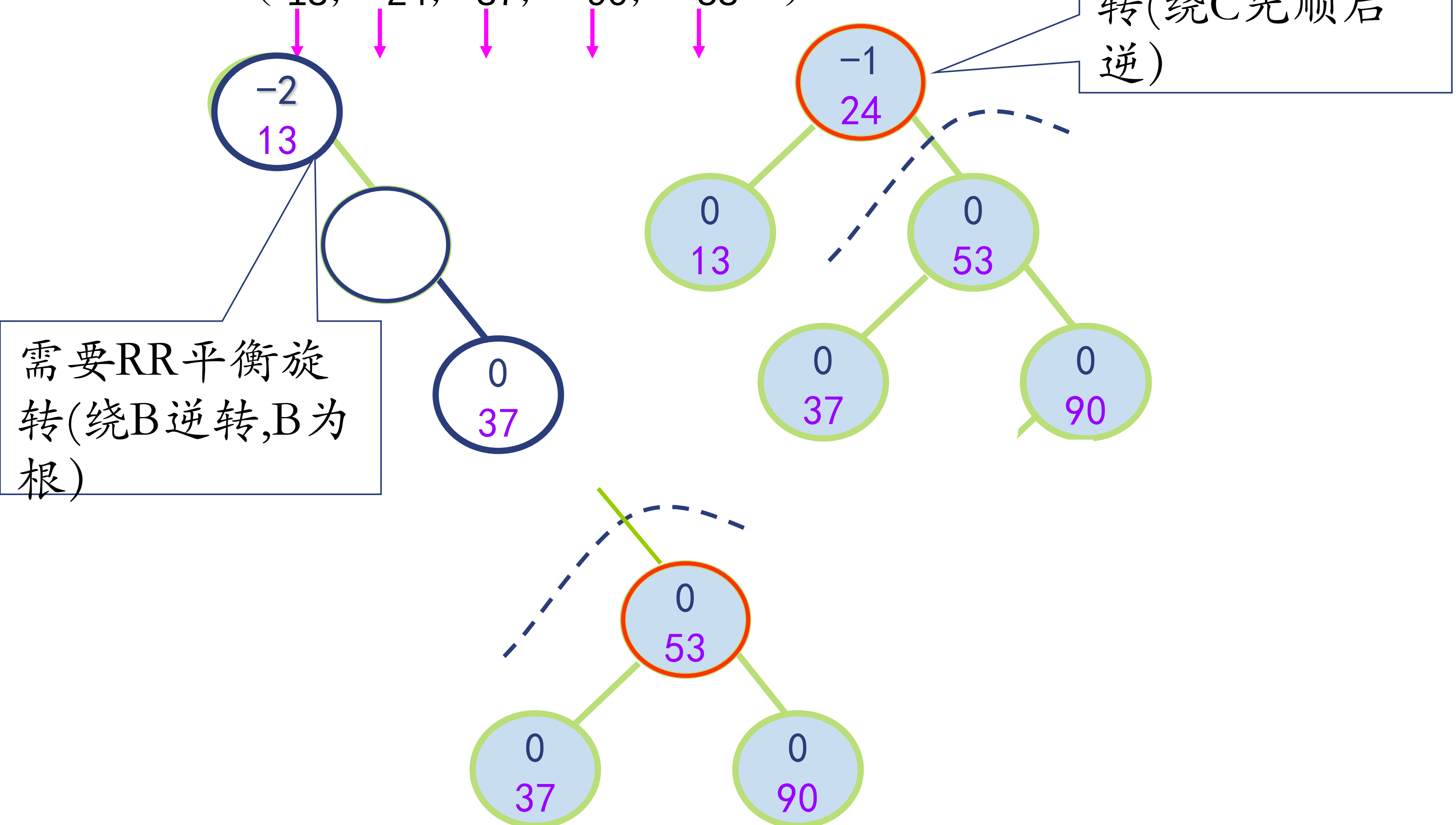


9.2.2 平衡二叉树

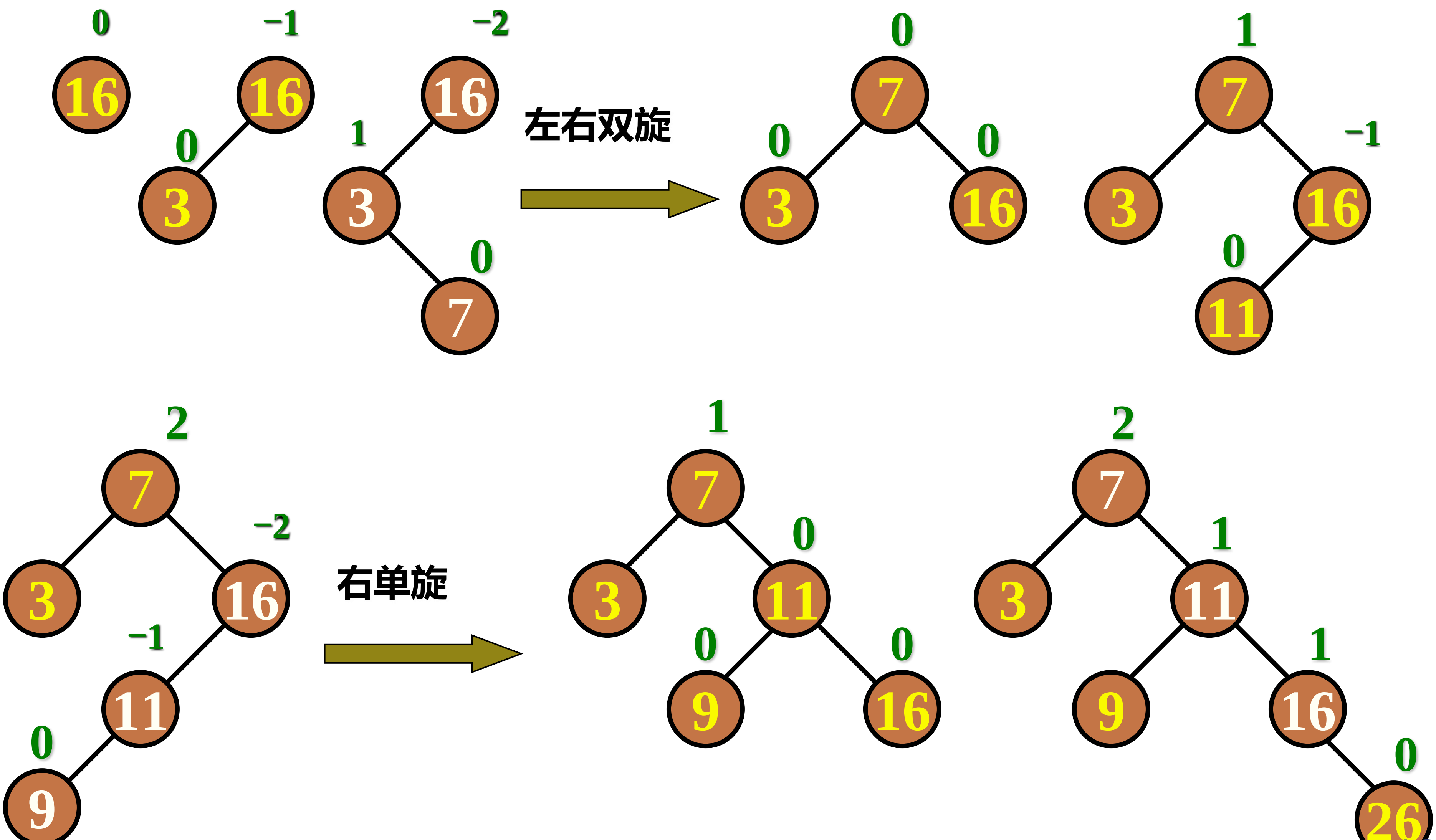
```
void RightBalance(BSTree &T){//右平衡旋转处理
    BSTree rc,ld;  rc=T->rchild;
    switch(rc->bf){
    case RH: // “RR型”
        T->bf=rc->bf=EH;
        L_Rotate(T); break;
    case LH: // “RL型”
        ld=rc->lchild;
        switch(ld->bf) {
        case RH:T->bf=LH;rc->bf=EH;break;
        case EH:T->bf=rc->bf=EH;break;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

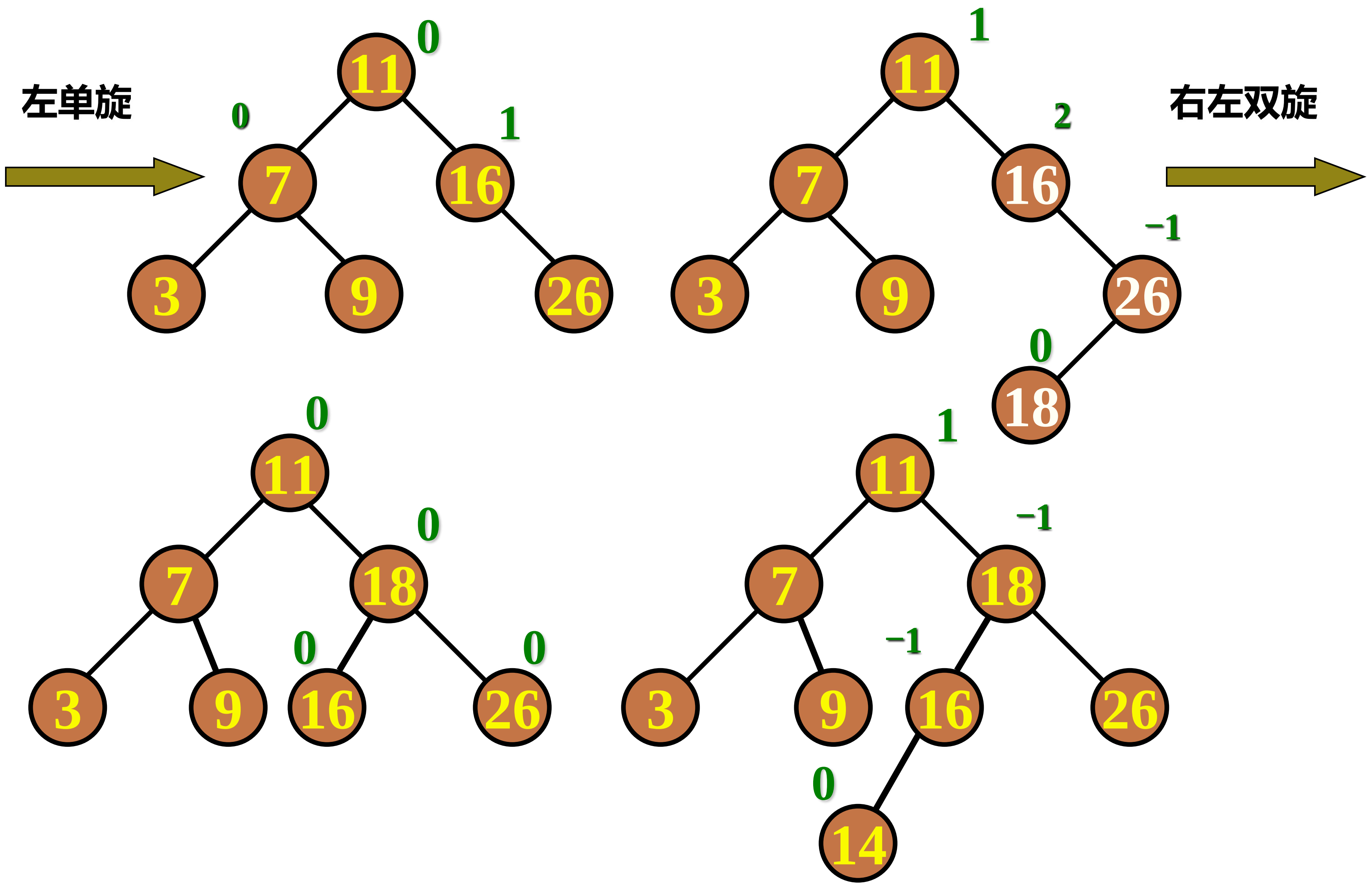
（ 13， 24， 37， 90， 53 ）



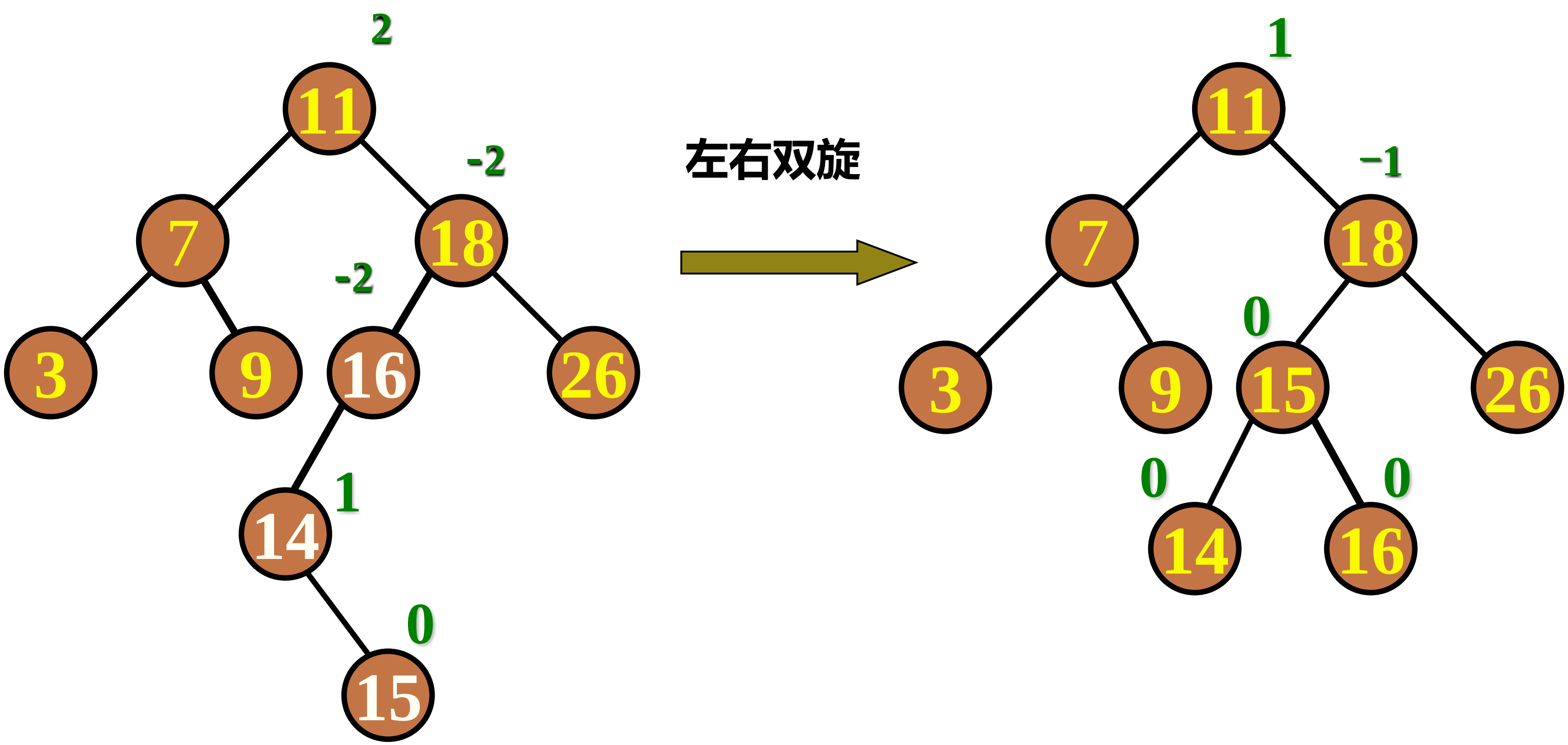
例，输入关键字序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，插入和调整过程如下。



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。



9.2.2 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;
}
```

9.2.2 平衡二叉树

AVL树的删除

- 如果被删结点x最多只有一个子女，那么问题比较简单。如果被删结点x有两个子女，首先查找 x 在中序次序下的直接前驱 y（同样可以找直接后继）。再把 结点y 的内容传送给结点x，现在问题转移到删除结点 y。
- 把结点y当作被删结点x。
- 将结点x从树中删去。因为结点x最多有一个子女，我们可以简单地把x的双亲结点中原来指向x的指针改指到这个子女结点；如果结点x没有子女，x双亲结点的相应指针置为$NULL$。然后将原来以结点x为根的子树的高度减1，

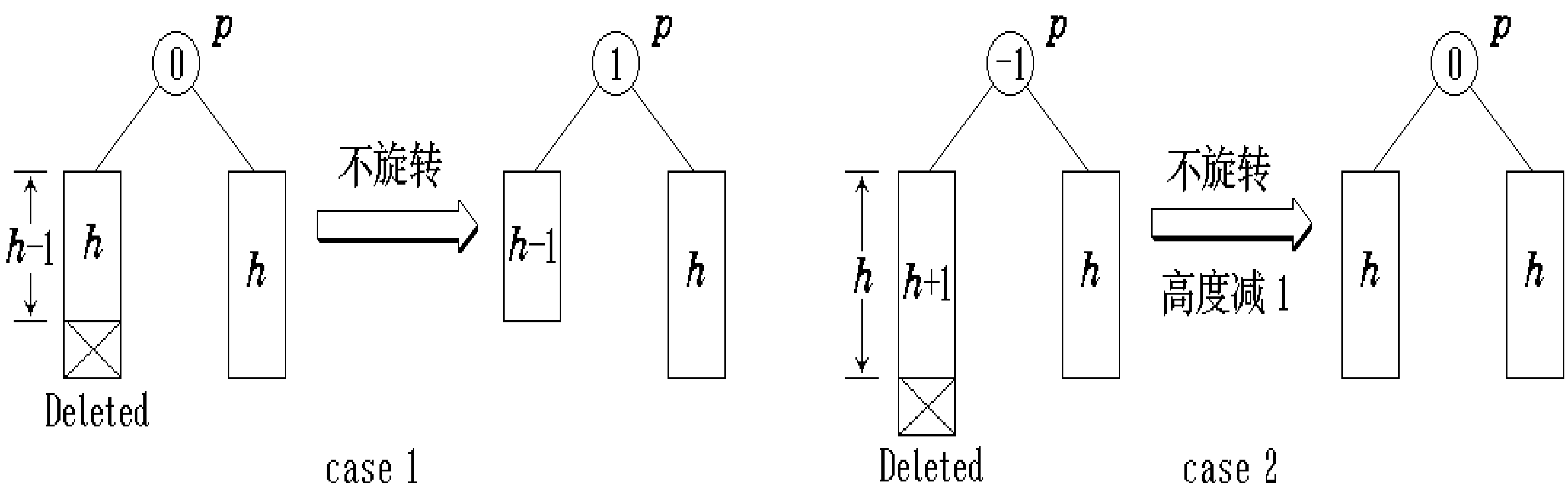
9.2.2 平衡二叉树

AVL树的删除

- 必须沿 x 通向根的路径反向追踪高度的变化对路径上各个结点的影响。
- 用一个布尔变量 shorter 来指明子树的高度是否被缩短。在每个结点上要做的操作取决于 shorter 的值和结点的 balance，有时还要依赖子女的 balance 。
- 布尔变量 shorter 的值初始化为True。然后对于从 x 的双亲到根的路径上的各个结点 p，在 shorter 保持为 True 时执行下面的操作。如果 shorter 变成False，算法终止。

9.2.2 平衡二叉树

- case 1 ： 当前结点 p 的balance为0。如果它的左子树或右子树被缩短，则它的 balance改为 1 或-1，同时 shorter 置为False。
- case 2 ： 结点 p 的balance不为0，且较高的子树被缩短，则 p 的balance改为0，同时 shorter 置为True。

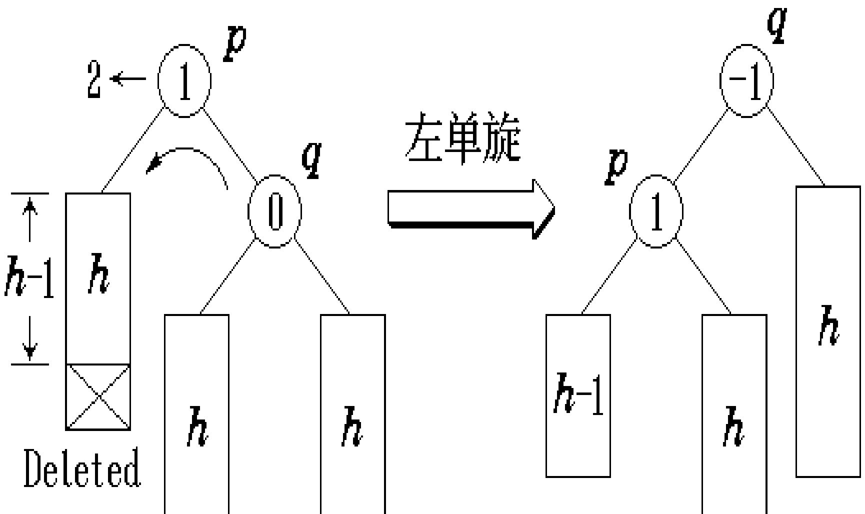


9.2.2 平衡二叉树

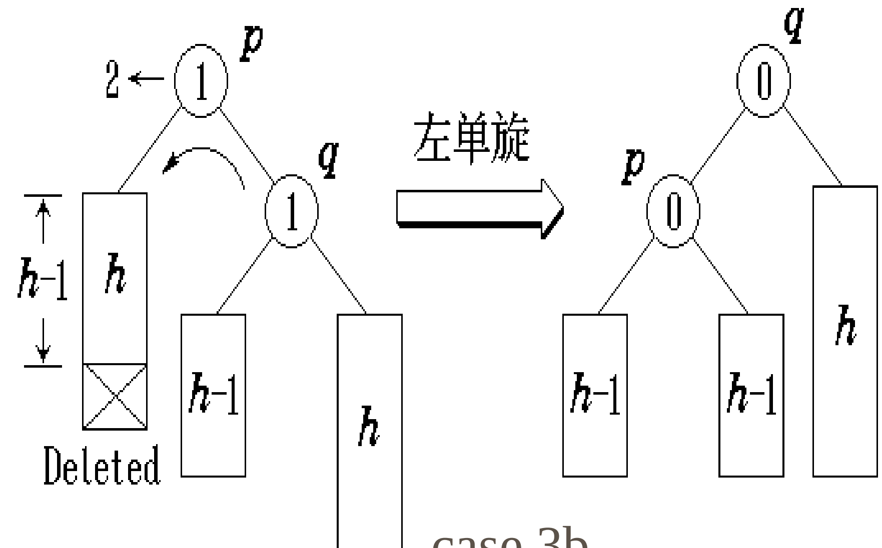
- case 3 : 结点 p 的balance不为0，且较矮的子树又被缩短，则在结点 p 发生不平衡。需要进行平衡化旋转来恢复平衡。令 p 的较高的子树的根为 q（该子树未被缩短）， 根据 q 的balance，有如下 3 种平衡化操作。
- case 3a : 如果 q 的balance为0，执行一个单旋转来恢复结点 p 的平衡，置shorter为False。
- case 3b : 如果 q 的balance与 p 的balance相同，则执行一个单旋转来恢复平衡，结点 p 和 q 的balance均改为0，同时置shorter为True。
- case 3c : 如果 p 与 q 的balance相反，则执行一个双旋转来恢复平衡，先围绕 q 转再围绕 p 转。新的根结点的balance置为0，其它结点的balance相应处理，同时置shorter为True。

9.2.2 平衡二叉树

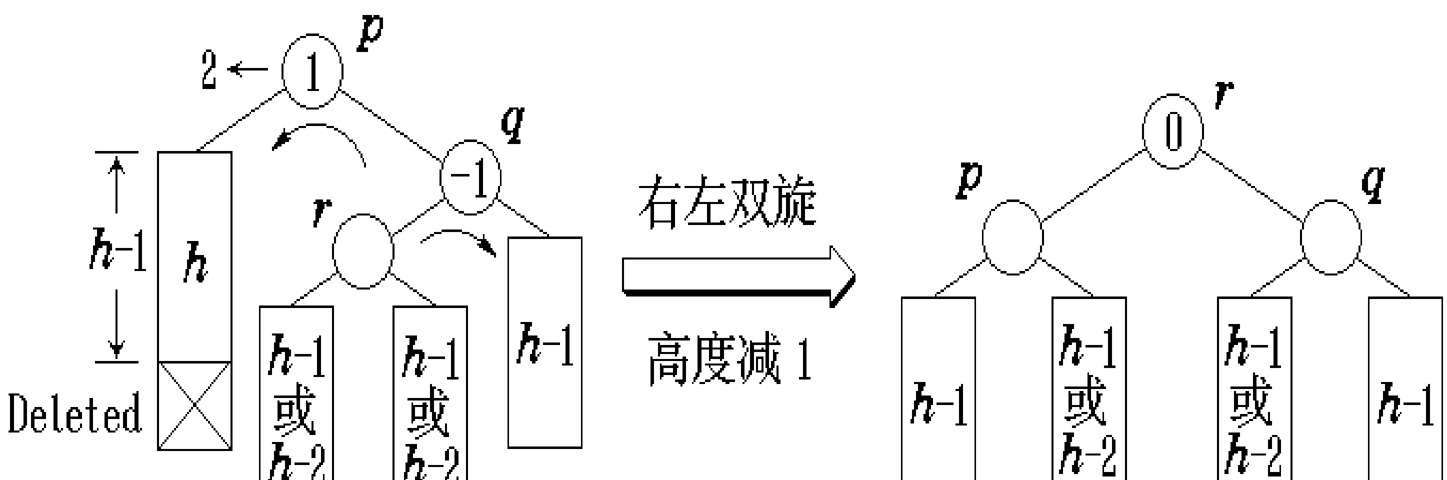
在case 3a，3b和3c的情形中，旋转的方向取决于是结点 p 的哪一棵子树被缩短



case 3a

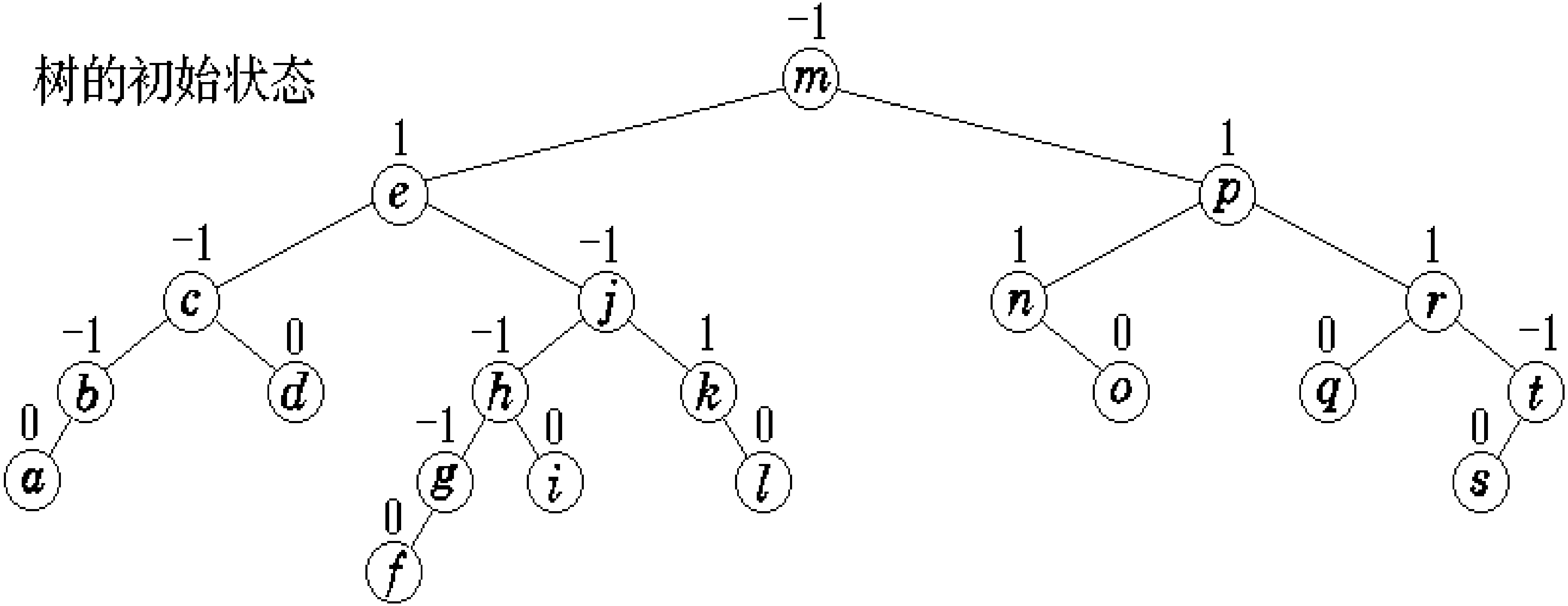


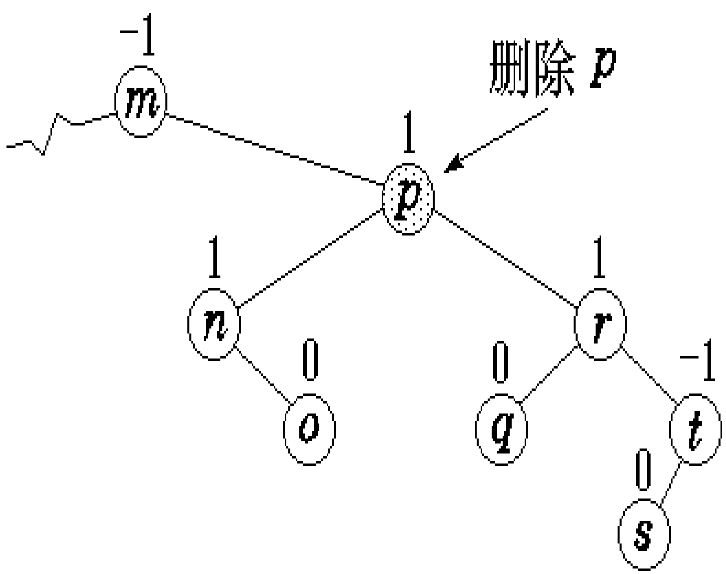
case 3b



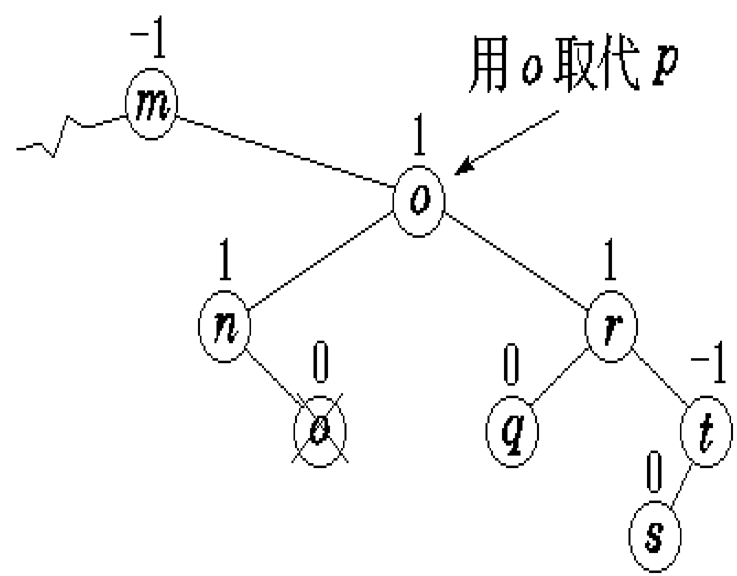
case 3c

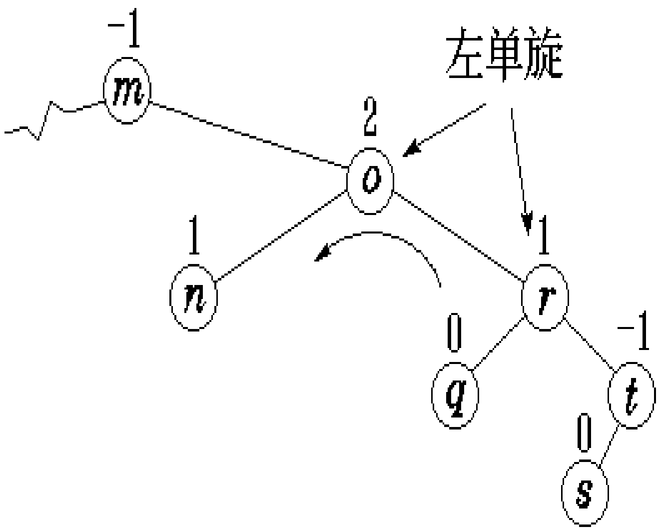
树的初始状态



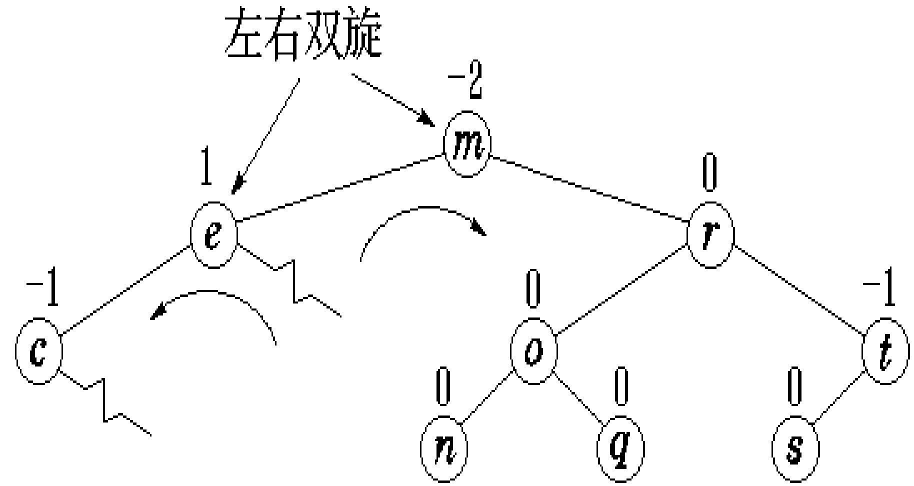


寻找 p 的中序下的直接前驱 o，用 o 取代 p，删除 o，平衡旋转

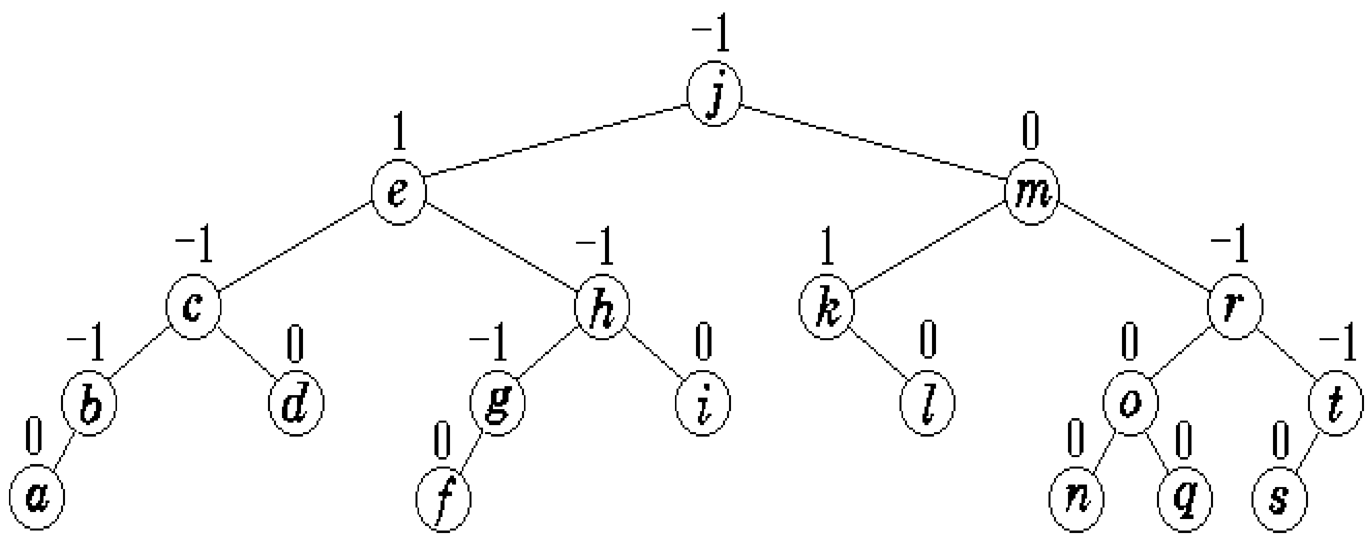




以 r 为旋转轴作左单旋，子树的高度减 1
m 发生不平衡



首先以 j 为旋转轴作左单旋，再以 j 为旋转轴作右单旋，让 e 成为 j 的左子女，m 成为 j 的右子女。树的高度减 1

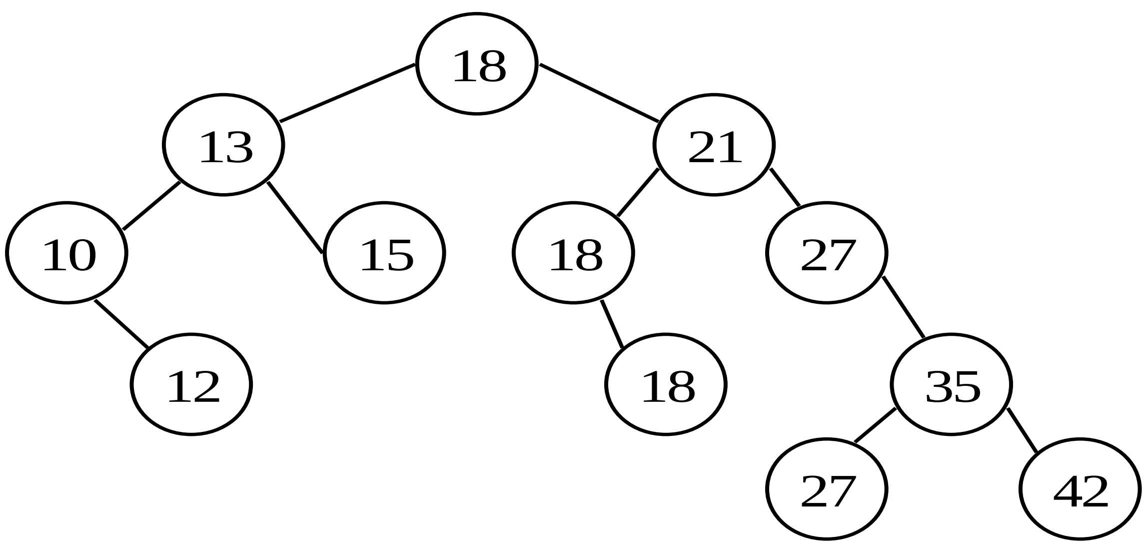


9.2.2 平衡二叉树

- 深度为h的平衡二叉树所具有的最少结点数。
- 设以N_h表示深度为h的平衡二叉树中含有的最少结点数。
- $N_0=0$，$N_1=1$，$N_2=2$，并且$N_h=N_{h-1}+N_{h-2}+1$。利用归纳法可证明：当$h\geqslant 0$时，$N_h=F_{h+2}-1$。
- 含有n个结点的平衡二叉树的最大深度，即在AVL上进行查找的时间复杂度为O(logn)。

9.2.2 平衡二叉树

例：在二叉排序树的结构中，有些数据元素值可能是相同的，设计一个算法实现按递增有序打印结点的数据域，要求相同的数据元素仅输出一个，算法还应能报出最后被滤掉，而未输出的数据元素个数，对如图所示的二叉排序树，输出为：**10，12，13，15，18，21，27，35，42．**滤掉**3**个元素。



9.2.2 平衡二叉树

```
void  BSTPrint(BSTree t,int *count) {
   //递增序输出二叉排序树中结点的值，滤去重复元素
   if(t) {
     BSTPrint(t->lchild);  //中序遍历左子树
     if(pre==null)  //pre是当前访问结点的前驱,
        pre=t;   //调用本算法时初值为null
     else if(pre->key==t->key)
        *count++;       //*count记重复元素，初值为0
     else {                    //输出元素， 前驱后移
        printf("%4d",t->key); pre=t;
     }
     BSTPrint(t->rchild); //中序遍历右子树
   }
}
```

正在答疑