第七章 字典与检索

张史梁
slzhang.jdl@pku.edu.cn

内容提要

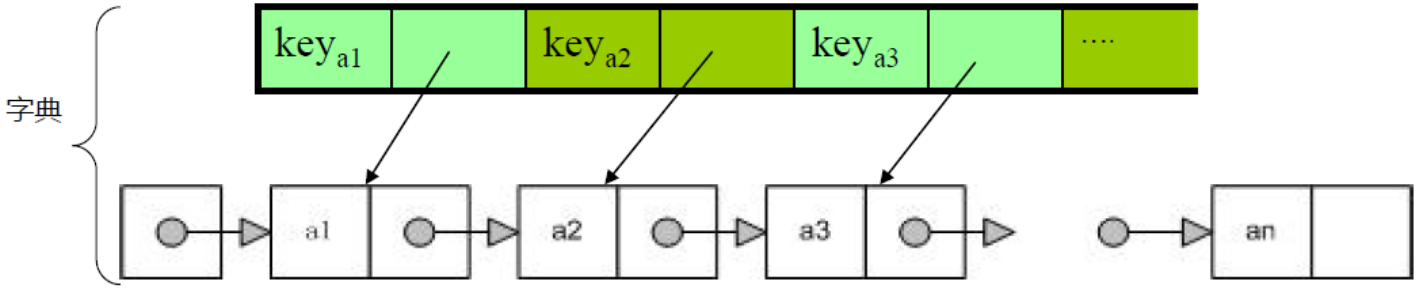
- 基本概念
- 字典的顺序表表示
- 字典的散列表示
  - 散列表
  - 散列函数
  - 存储表示与碰撞的处理
    - 开地址法（线性探查、双散列函数）
    - 拉链法
- 字典的索引和树型表示
  - 二叉排序树
  - AVL树

索引与字典

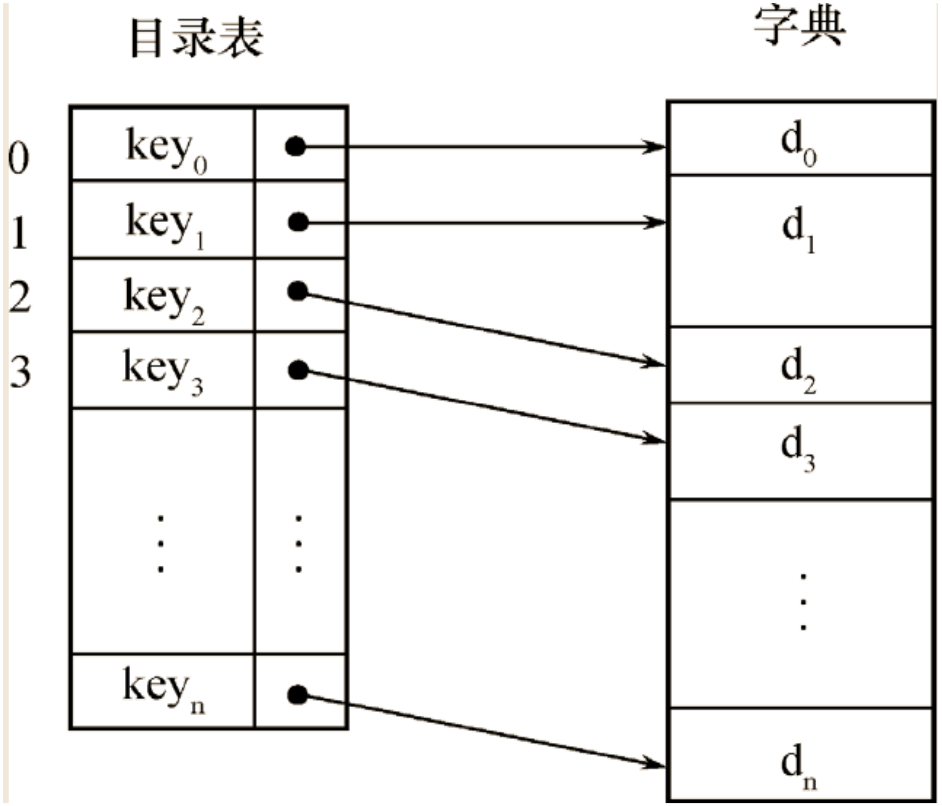
- 问题的场景
  - 在前面关于字典的讨论中，所有元素值的类型DataType都定义为int类型。
  - 现实字典中元素的值（value）可以是字符串、公式、插图、表格等多种类型
  - 不同类型的元素难以采用顺序存储实现
  - 引入辅助结构——索引
- 索引
  - 给出一种关键码到存储地址的映射
  - 散列通过函数定义实现映射
  - 索引通过建立索引表实现映射

索引概念

- 索引的实质是元素类型相同的字典
  - 索引是索引项的集合；
  - 索引项是由一个结点的关键码和该结点的存储位置组成的关联
  - 索引的实质还是字典，而且是元素类型相同的等长结点的字典。
  - 所有关于字典的讨论都适应于索引；所有字典的实现也可以用来组织索引。
- 在索引结构中
  - 若每个索引项对应字典中的一个元素，称为密集索引
  - 若每个索引项对应字典中的一组元素，称为稀疏索引



引入目录表（索引）后的字典

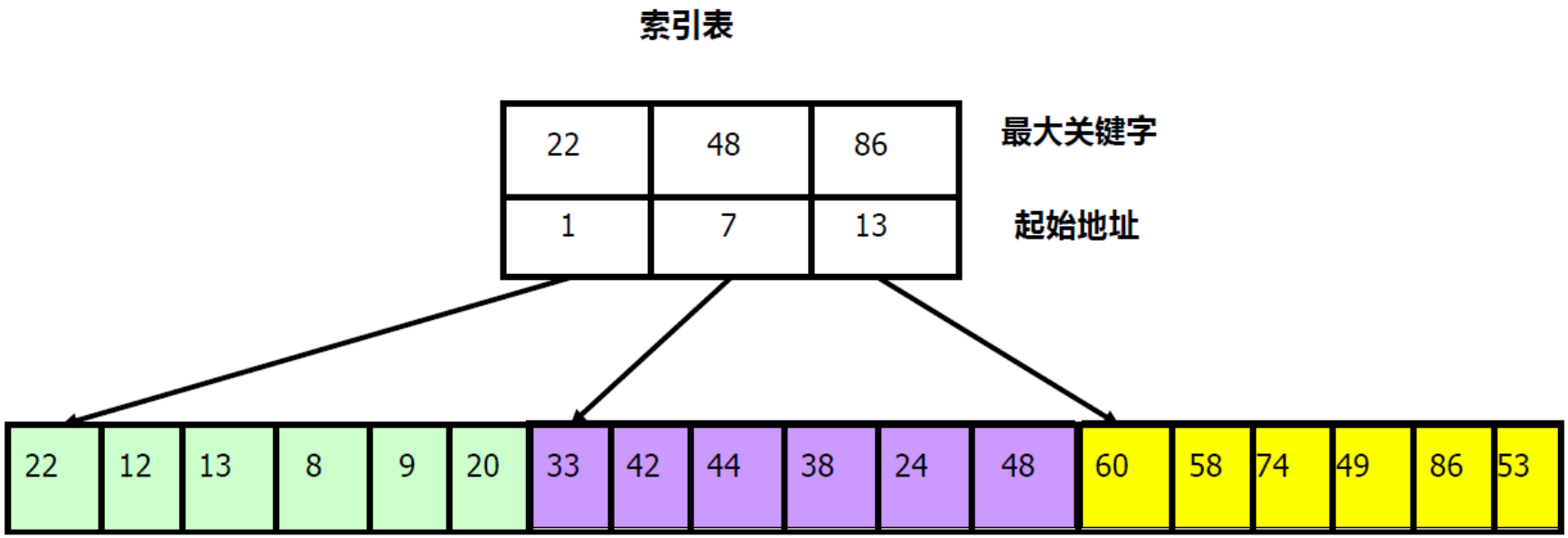


密集索引

索引与字典

- 一种稀疏索引

最大关键码值	第一个记录的位置



索引的作用

- 引入索引就可以将包含大量属性信息且不等长元素的字典的处理
- 转换成对仅仅包含关键码到地址对应关系（简单类型并且等长的元素）的索引结构的处理
- 在检索一个元素时，只要在索引中找到对应的关键码，就能得到对应结点的存储位置
- 在排序过程中，只要完成索引中元素（索引项）的排序，而不需要移动字典本身的任何结点

索引的目标

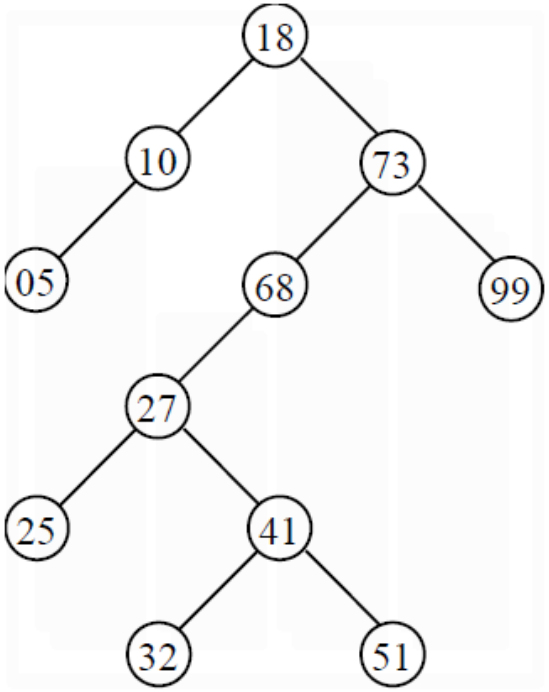
- 支持大数据量的检索
  - 一般采取树型索引结构（多级索引）
  - 支持多种检索码
  - 支持有效的插入、删除
  - 支持范围查询

索引与字典

- 常用的索引
  - 以关键码为结点的索引：二叉排序树
    - 二叉排序树的查找、插入、删除
    - AVL树、B树
  - 以字符为节点的索引：字符树（选学）

二叉排序树 – 例子

- 关键码集合K={18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25}，得到的一棵二叉排序树如图所示

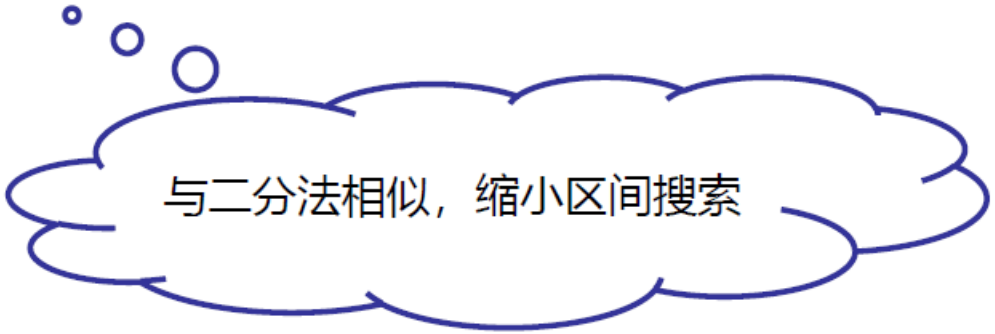


按照中根周游一棵二叉排序树得到的序列将是按照码值由小到大的排列

- 如果任一结点的左子树非空，则左子树中的所有结点的关键码都小于根结点的关键码；
- 如果任一结点的右子树非空，则右子树中的所有结点的关键码都大于根结点的关键码。

二叉排序树检索

- 首先根据无序序列的先后顺序，构造二叉排序树。
- 然后根据二叉排序树的定义进行检索：
  - 若二叉排序树为空树，检索失败；
  - 如给定值key等于二叉排序树的根结点的key，则检索成功；
  - 如给定值key小于二叉排序树的根结点的key，继续在左子树检索；
  - 如给定值key大于二叉排序树的根结点的key，继续在右子树检索；

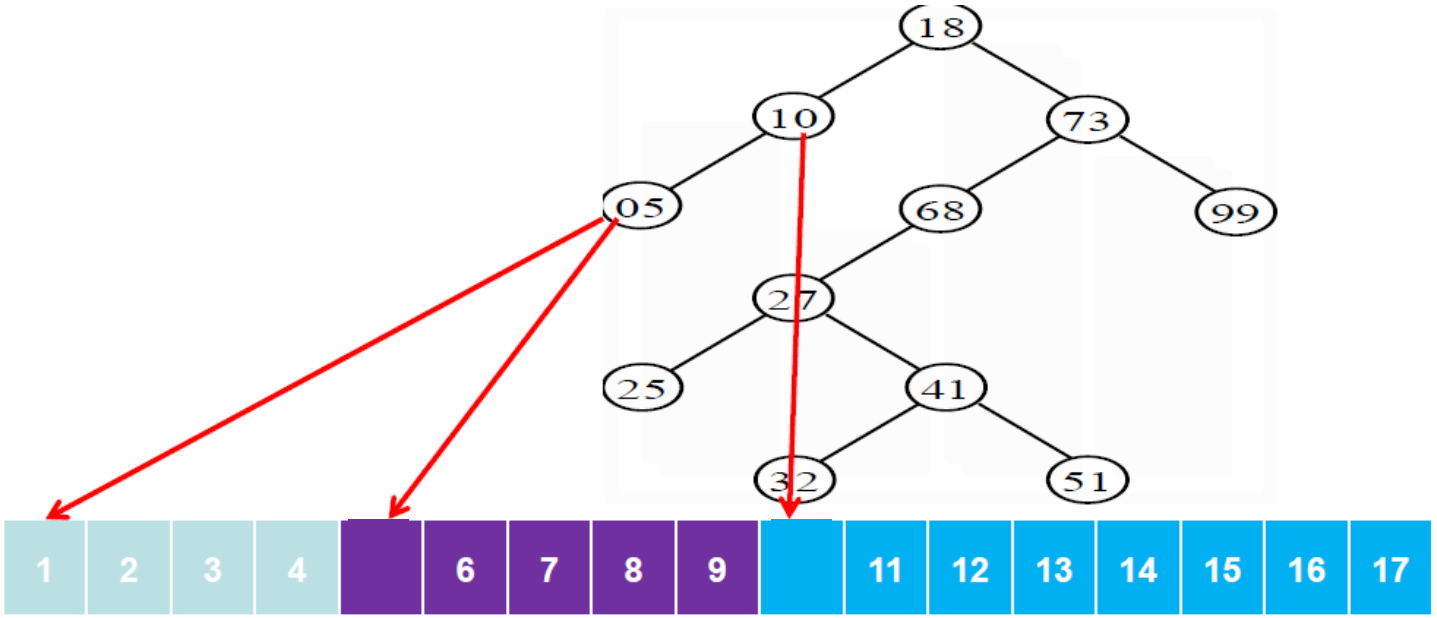


二叉排序树检索算法

```
①  int  searchNode(PBinTree ptree, KeyType key, PBinTreeNode *position)
②   {
③        PBinTreeNode p, q;
④         p= q = *ptree;
⑤         while (p!=NULL)
⑥        {
⑦              q = p;
⑧              if (p->key == key)
⑨              {
⑩                    *position=p;   return(1);
⑪               }
⑫             else if (p->key > key)    p=p->llink;       /*进入左子树*/
⑬                   else  p = p->rlink;            /*进入右子树继续检索*/
⑭         }
⑮         *position=q;                                  /*返回结点插入位置*/
⑯         return(0);                                   /*检索失败*/
⑰   }
```

二叉排序树检索的应用

- 二叉排序树与索引数据一起构成扩充的二叉排序树
  - 每个内部结点代表一个字典元素的关键码，
  - 中序周游之后得到的序列里：每个外部节点代表位于其相邻的两个内部节点关键码之间的所有不属于当前字典的关键码集合（P216）

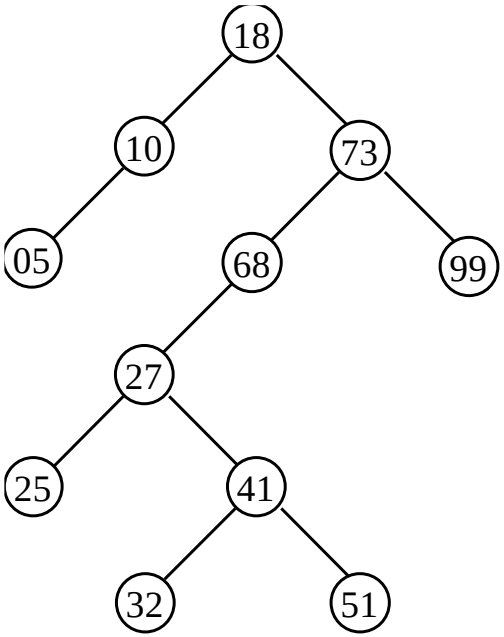


二叉排序树的插入

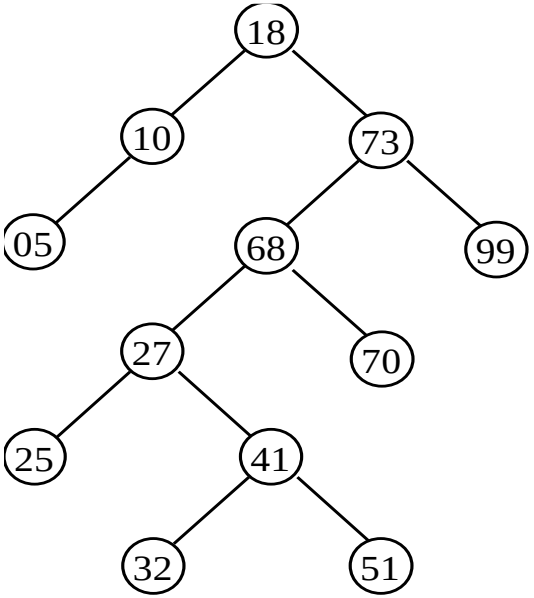
- 插入新的结点后，必须保证插入后仍然满足二叉排序树的性质：
  - 如果二叉排序树为空，则新结点作为根结点，返回；
  - 将新结点的关键码与根结点的关键码比较，若小于根结点的关键码，则将新结点插入到根结点的左子树中；否则，插入到右子树中。
  - 子树中的插入过程和树中的插入过程相同，如此进行下去，直到找到该结点，或者直到新结点成为叶子结点为止。

二叉排序树的插入

- 在图a所示的二叉排序树中插入关键码为70的结点。
  - 插入过程最后，68的右子树为空，因此，70作为68的右子女插入二叉排序树



图a



图b

二叉排序树的插入算法

```
①  void insertNode(PBinTree ptree, KeyType key)
②  {
③      PBinTreeNode    p,  position;
④      if (searchNode(ptree, key, &position) == TRUE      return;
⑤      p = (PBinTreeNode)malloc(sizeof(BinTreeNode));
⑥      if (p == NULL)
⑦            { printf("Out of space!\n");     exit(1);  }
⑧      p->key = key;
⑨      p->llink = p->rlink = NULL;          /*建立结点*/
⑩      if (position == NULL)    *ptree = p;
⑪      else  if (key < position->key)
⑫                position->llink = p;        /* 作为左子树 */
⑬          else   position->rlink = p;       /* 作为右子树 */
⑭  }
```

二叉排序树的构造

- 构造策略：从空树开始，将元素逐个插入到二叉排序树中。

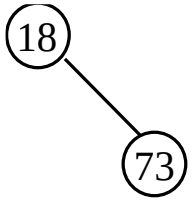
```
void CreateTree(PBinTree ptree, SeqDictionary dic)
{
    int i;
    for (i = 0; i < dic.n; i++)
        insertNode(ptree, dic.element[i].key);
}
```

二叉排序树的构造示例

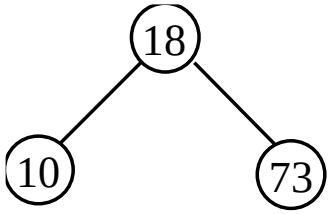
- 关键码集合为K={18,73,10,05,68,99,27,41,51,32,25}，写出二叉排序树的构造过程

(a) 空树

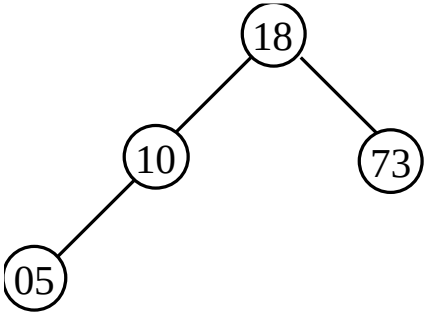
(b) 插入18



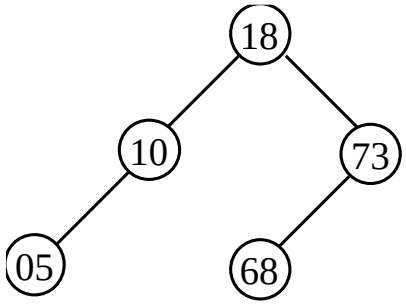
(c) 插入73



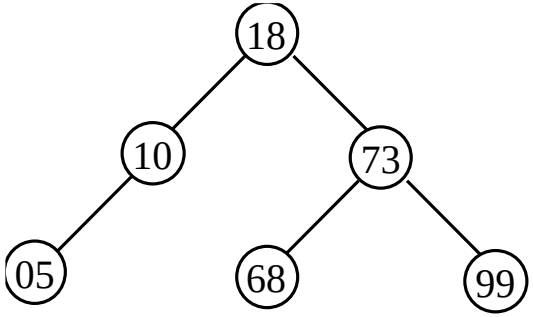
(d) 插入10



(e) 插入05



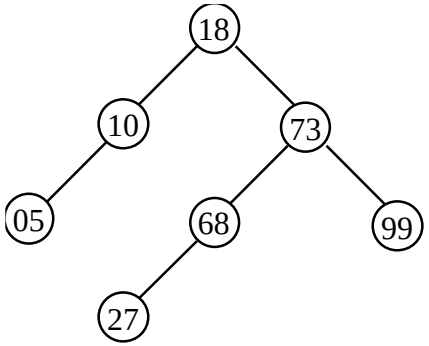
(f) 插入68



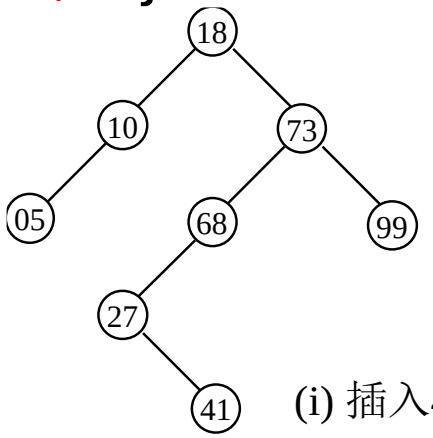
(g) 插入99

二叉排序树的构造示例

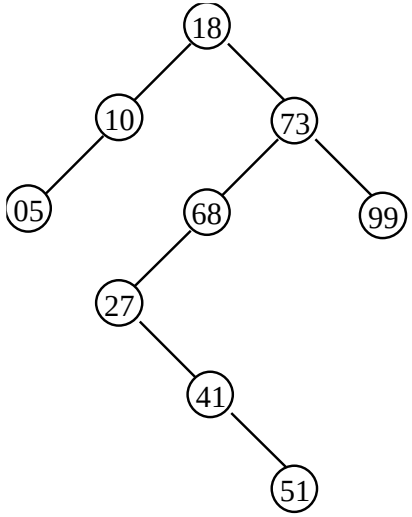
K={18,73,10,05,68,99,27,41,51,32,25}



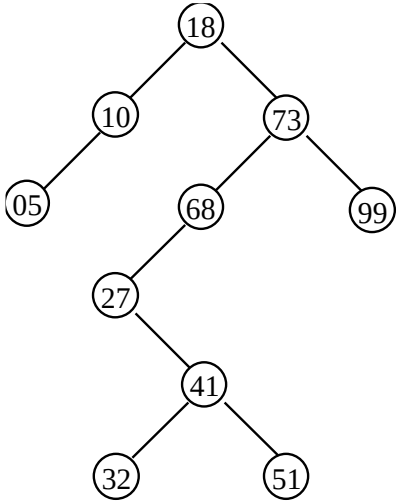
(h) 插入27



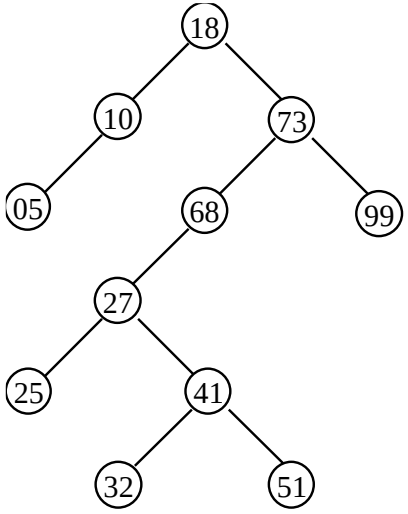
(i) 插入41



(j) 插入51



(k) 插入32



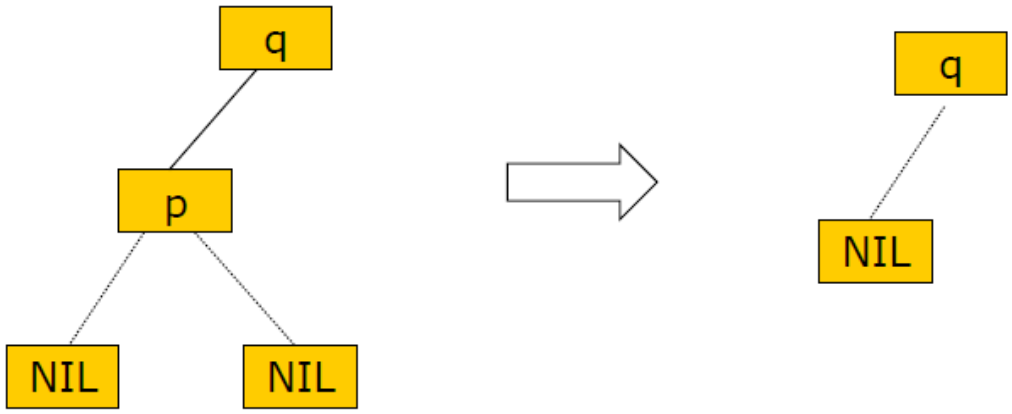
(l) 插入25

二叉排序树的删除

- 在二叉排序树中删除一个指定结点，删除结点后的二叉树必须满足二叉排序树的性质

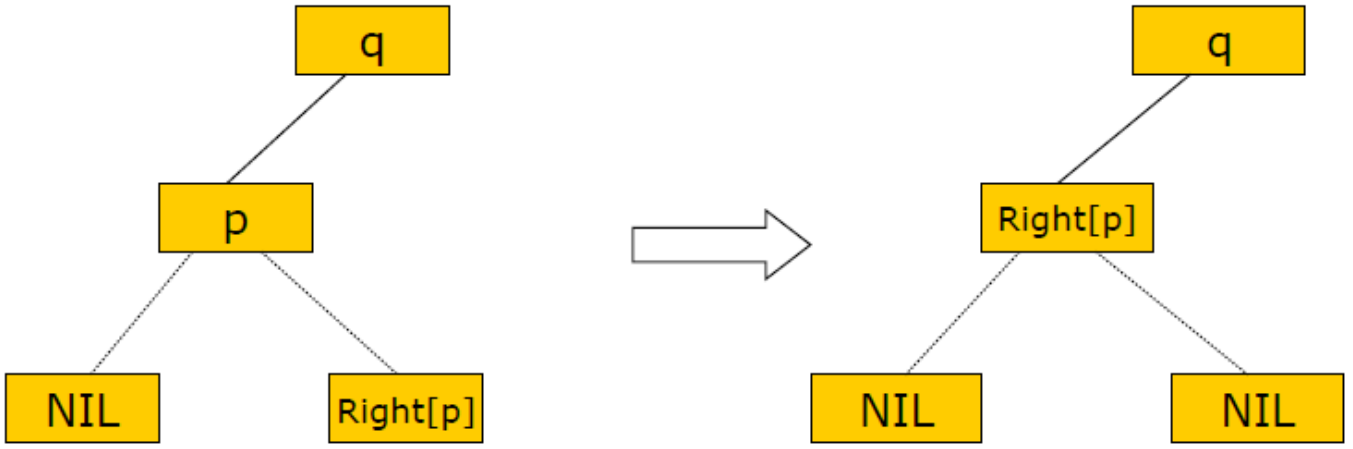
删除情况1

- 情况1. 叶结点可以直接删除，其父结点的相应指针置为空



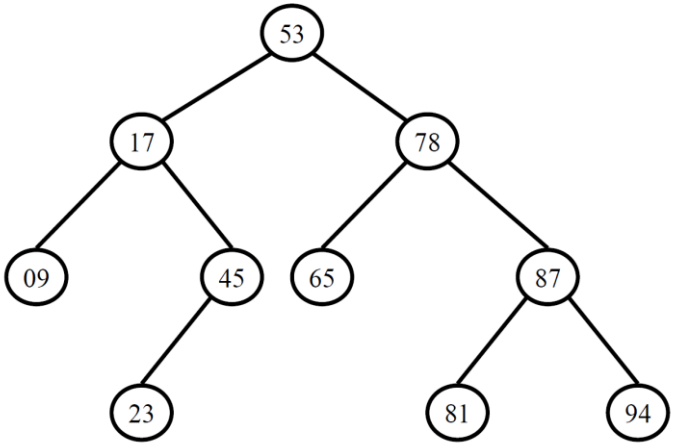
删除情况2

- 情况2. 只有一个子女的结点**p**被删除时，分以下情况：
  - p是q的左子结点，p只有左子结点或右子结点
  - p是q的右子结点，p只有左子结点或右子结点
- 可让此子女直接代替即可



删除情况3

- 情况3. 被删除结点p 的左右子女皆不空。
  - 根据二叉检索树的性质，此时要寻找能代替此结点的结点必须是：比p 左子树中的所有结点都大，比p的右子树的所有结点都小（或不大于）。
- 两个选择：
  - 左子树中最大者
  - 右子树中最小者
- 而这二者都至多只有一个子结点

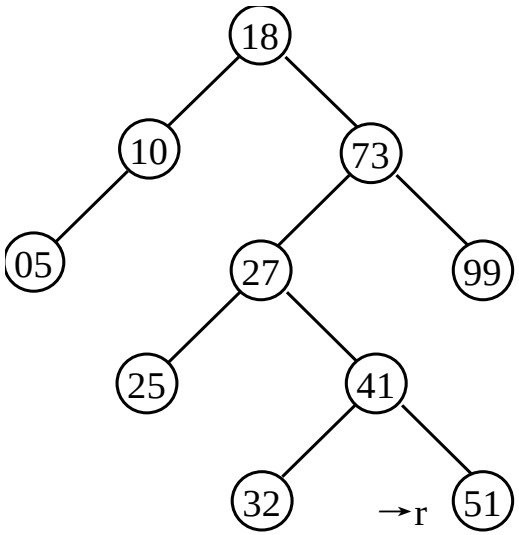


左子树中最大者【方法1】

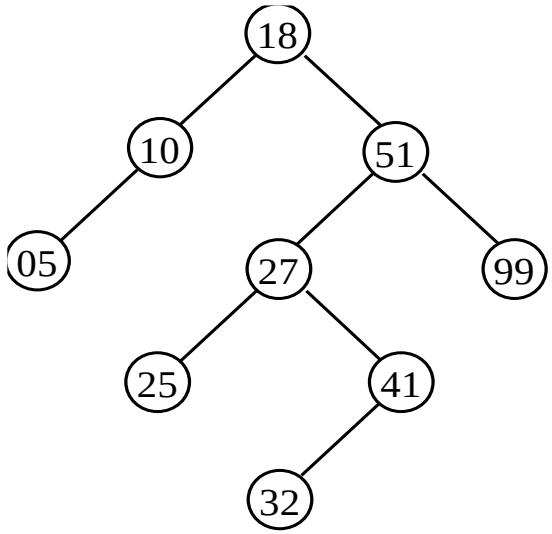
① 若被删除结点p没有左子树，则用p的右子女代替p即可；

② 否则，在p的左子树中，中序遍历找出最后一个结点r，将r删除 (r一定无右子女，用r 的左子女代替r即可)

③ 用r节点代替被删除的节点p



删除73



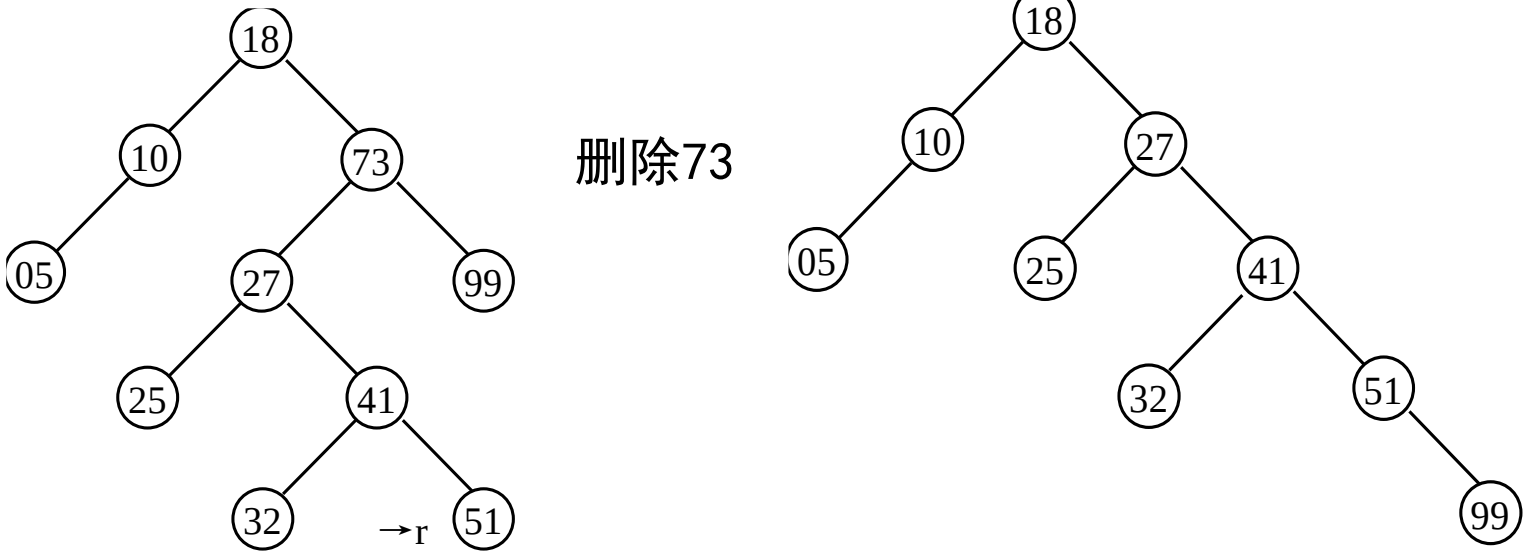
因为r在对称序列中紧排在p结点之前，所以用r代替p结点不会改变二叉排序树的性质

左替右换【方法2】

① 若被删除结点p没有左子树，用p的右子女代替p即可；

② 否则，在p的左子树中，中序周游找出最右一个结点r（r一定无右子女）将r的右指针指向p的右子女；

③ 用p的左子女代节点代替结点p。



方法2说明

- p的右子树中结点的关键码都大于p结点的关键码
- 在p的左子树中按中根周游找出的最后一个结点r，r在对称序列中紧排在p结点的前面
- 因此，用r的右指针指向p的右子树，可以保证二叉排序树的性质

```
/*给出第二种删除方法的算法。
设在二叉排序树中删除关键码为key的结点。*/
void deleteNode(PBinTree ptree, KeyType key)
{
   PBinTreeNode parentp, p, r;
   p=*ptree; parentp=NULL;
   while(p!=NULL)
   {
      if(p->key==key) break;
      /* 找到了关键码为key的结点 */
      parentp=p;
      if(p->key>key) p=p->llink;
      /* 进入左子树 */
      else p=p->rlink;
      /* 进入右子树 */
   }
   if(p==NULL) return;
   /* 二叉排序树中无关键码为key的结点 */
   if(p->llink==NULL)
   /* 结点*p无左子树 */
   {
      if(parentp==NULL)
          *ptree=p->rlink;
      /* 被删除的结点是原二叉排序树的根结点*/
      else if(parentp->llink==p)
      /* *p是其父结点的左子女 */
          parentp->llink=p->rlink;
      /* 将*p的右子树链到其父结点的左链上 */
      else
          parentp->rlink=p->rlink;
       /* 将*p的右子树链到其父结点的右链上 */
   }
   else /* 结点*p有左子树 */
   {
      r=p->llink;
      while(r->rlink!=NULL) r=r->rlink;
       /* 在*p的左子树中找最右下结点*r */
      r->rlink=p->rlink;
      /* 用*r的右指针指向*p的右子女 */
      if(parentp==NULL) *ptree=p->llink;
      else if(parentp->llink==p)
      /* 用*p的左子女代替*p */
          parentp->llink=p->llink;
      else
          parentp->rlink=p->llink;
   }
   free(p); /* 释放被删除结点 */
}
```

问题讨论

- 以二叉排序/搜索树建索引的优缺点：
  - 优点：插入和删除运算非常简单；
  - 缺点：向二叉排序/搜索树里插入新结点或删除已有结点，要保证操作结束后仍符合二叉搜索树的定义
- 检索效率
  - 在二叉排序树中查找关键码为key的结点时，其比较次数与该结点在二叉树中的位置有关
  - 平均检索次数最小的二叉排序树称为最佳二叉排序树

问题讨论

- 问题1：
  - 经过若干次插入、删除后可能会失去平衡，检索性能变坏。那么，如何构筑一棵最佳二叉排序/检索树？

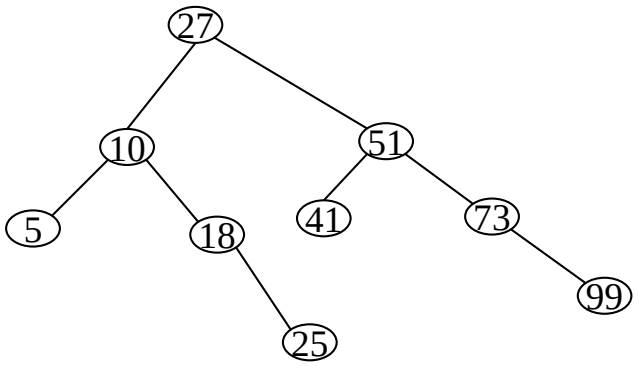
- 答：
  - 根据关键码集合及检索概率，可以构造出最佳二叉检索树（P215-P218)。

最佳二叉排序树的构建

1. 先将字典元素关键码排序；
2. 对每个关键码按二分法在排序关键码排序中进行检索，将检索中遇到的还未在二叉排序树中的关键码插入二叉排序树中。

K={27,73,10,05,18,41,99,51,25}

K={5,10,18, 25, 27, 41, 51, 73, 99}



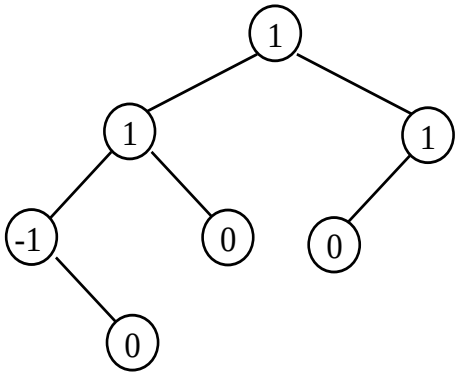
问题讨论

- 问题2：对于不等权节点，“最佳”构造的时间代价很大（$O(n^3)$, p224）
  - 对静态字典，经过若干次插入、删除后可能会失去平衡，检索性能变坏；
  - 对动态字典，很难动态的保持。

- 答：
  - 在检索所有关键码的概率相等情况下，构造便于动态保持平衡的二叉检索树——平衡二叉树

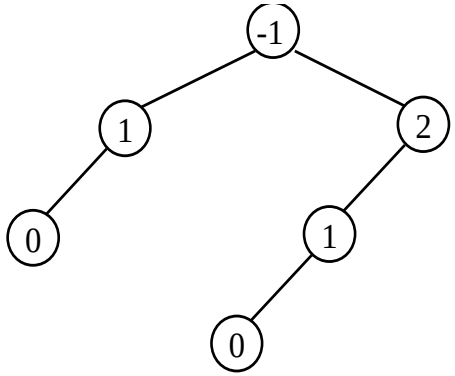
平衡二叉排序树 （AVL树）

- AVL树得名于发明者Adelson-Velskii& Landis (1962)名字的首字母缩写
- AVL树的定义
  - 每个结点左、右子树深度之差的绝对值不超过1；
  - 结点左、右子树深度之差定义为该结点的平衡因子，则平衡二叉排序树中每个结点的平衡因子只能是1、0或-1 。



(a)

平衡二叉排序树

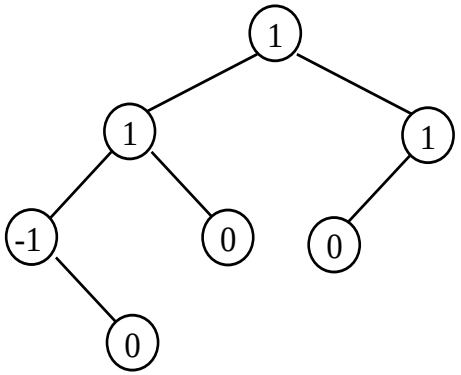


(b)

非平衡二叉排序树

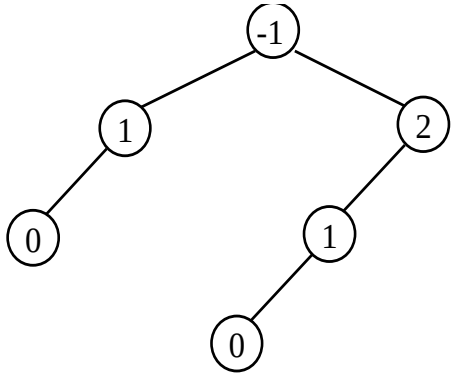
AVL树

- AVL树检索效率高吗？
  - 具有N个结点的AVL树高度不会超过1.44Log_2N，且通常接近Log_2N



(a)

平衡二叉排序树



(b)

非平衡二叉排序树

AVL树数据结构

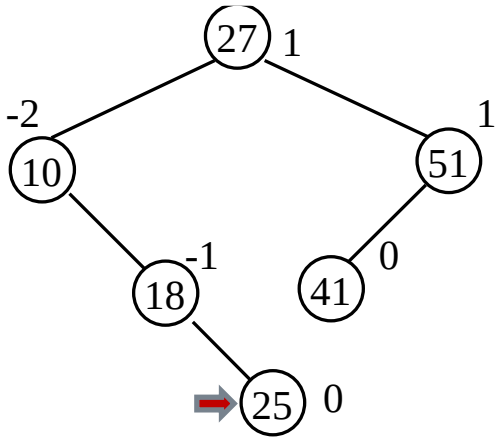
```
struct AVLNode;
typedef struct AVLNode * PAVLNode;

struct AVLNode
{   KeyType key;                    /* 结点的关键码 *
    DataType other;                         /* 结点的其它信息 */
    int bf;                     /* 结点的平衡因子 */
    PAVLNode llink, rlink; /* 分别指向结点的左、右子女 */
};

typedef struct AVLNode *AVLTree;
typedef AVLTree * PAVLTree;
```

AVL树的运算

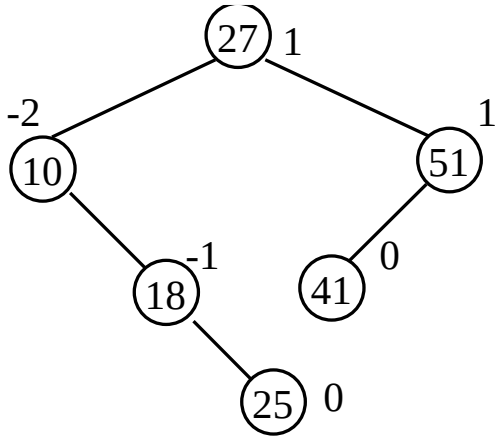
- 在AVL树中插入新结点时
  - 如果新结点插入后不影响其父结点为根的子树深度，则不会破坏整个二叉排序树的平衡；
  - 反之，若父结点为根的子树深度增加了，则会引起一连串的反应
    - 在其祖先的某一层上不再影响子树的深度（祖先对应的子树仍然是AVL树），则整个二叉排序树仍然是平衡的；
    - 在其祖先的某一层上破坏了平衡的要求，使整个二叉排序树不再是AVL树。



AVL树的平衡调整

- AVL树失去平衡后的处理方法：
  - 首先找出最小不平衡子树，在保证排序树性质的前提下，调整最小不平衡子树中各结点的连接关系，以达到新的平衡
  - 最小不平衡子树是指离插入结点最近，且以平衡因子绝对值大于1的结点为根的子树

例如，在插入25后，图中以10为根结点的子树是最小不平衡子树

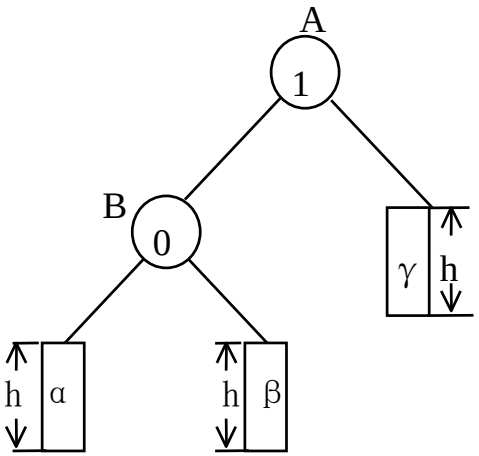


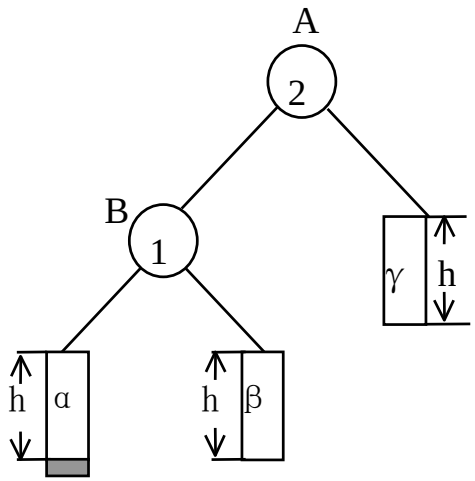
AVL树的平衡调整

- 假设最小不平衡子树的根结点为A，调整子树的操作可归纳为以下四种情况
  - LL型调整
  - LR型调整
  - RL型调整
  - RR型调整

LL型调整

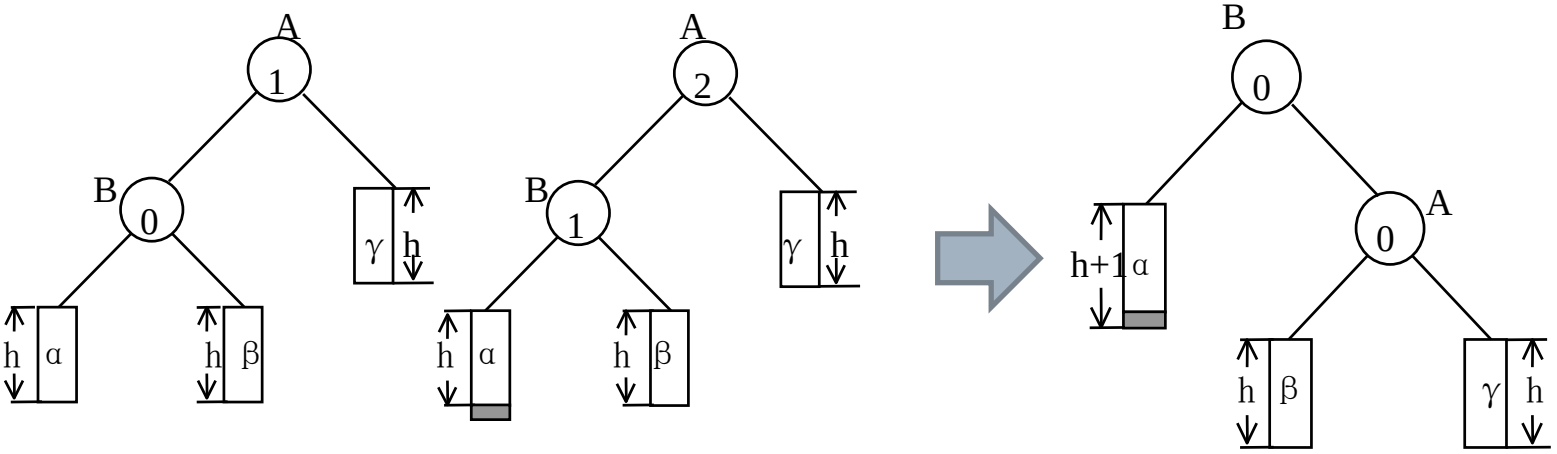
- 破坏平衡的原因
  - 由于在A的左子女(L)的左子树(L)中插入结点，使A的平衡因子由1变为2而失去平衡，
  - 图中长方框表示子树，带阴影的小框表示要插入的结点。



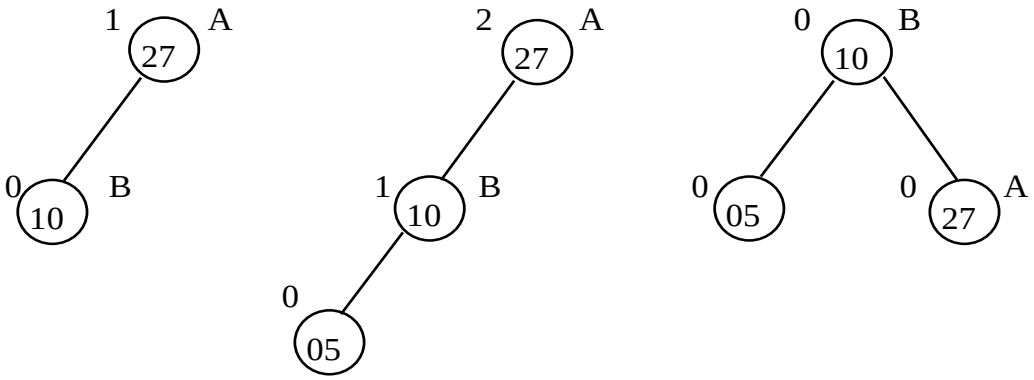


LL型调整

- 调整规则
  - 将A的左子女B提升为新二叉树的根
  - 原来的根A连同其右子树γ向右下旋转成为B的右子树
  - B的原右子树β作为A的左子树。
- 结合律：(αBβ) A (γ)= (α) B (βAγ)

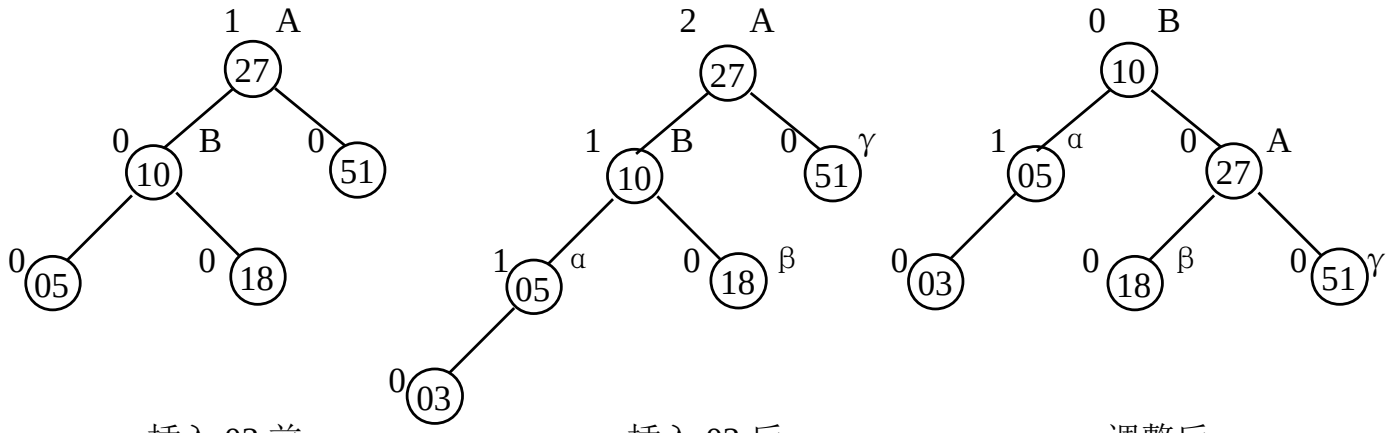


LL型调整（例）



输入 05 前　　输入 05 后　　调整后

(a) α 、 β 和 γ 都是空树的调整

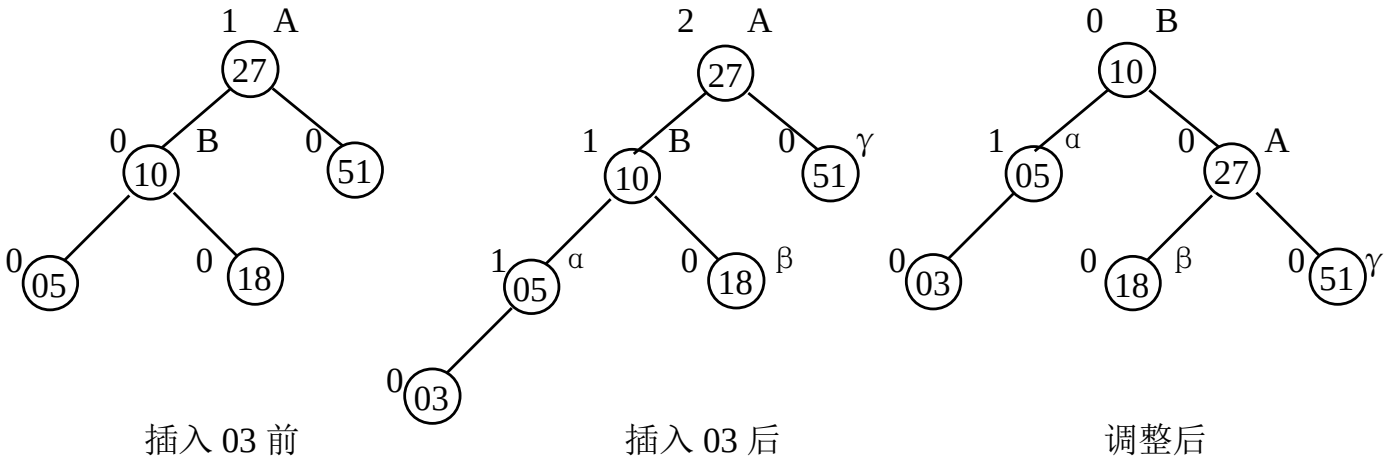


插入 03 前　　插入 03 后　　调整后

(a) α 、 β 和 γ 都是非空树的调整

LL型调整算法

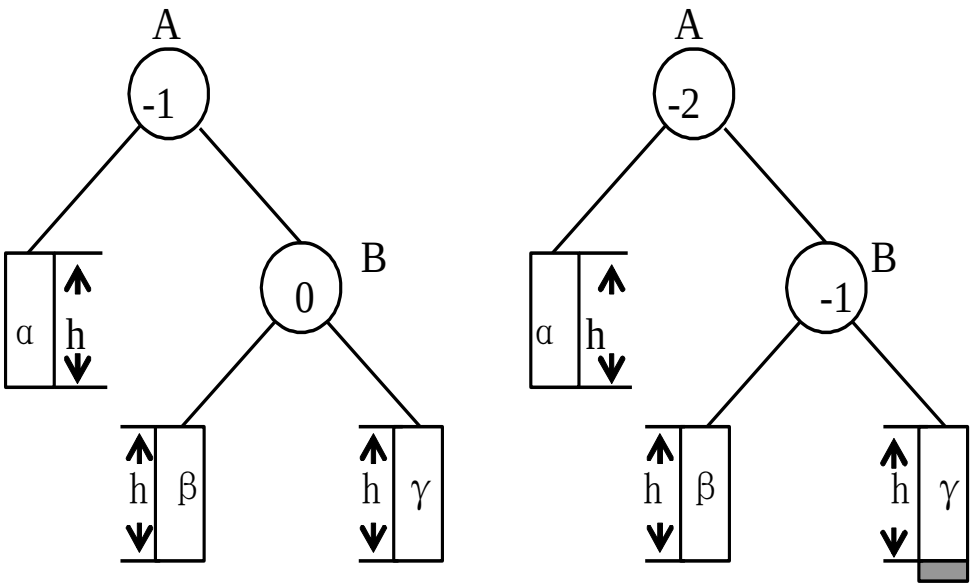
```
PAVLNode lL(PAVLNode a, PAVLNode b)
{   /* a 指向最小不平衡子树根节点，b指向其左儿子*/
    a->bf=0; a->llink=b->rlink;
    b->bf=0; b->rlink=a;     /* b指向调整后的子树的根结点 */
    return(b);
}
```



(a) α 、 β 和 γ 都是非空树的调整

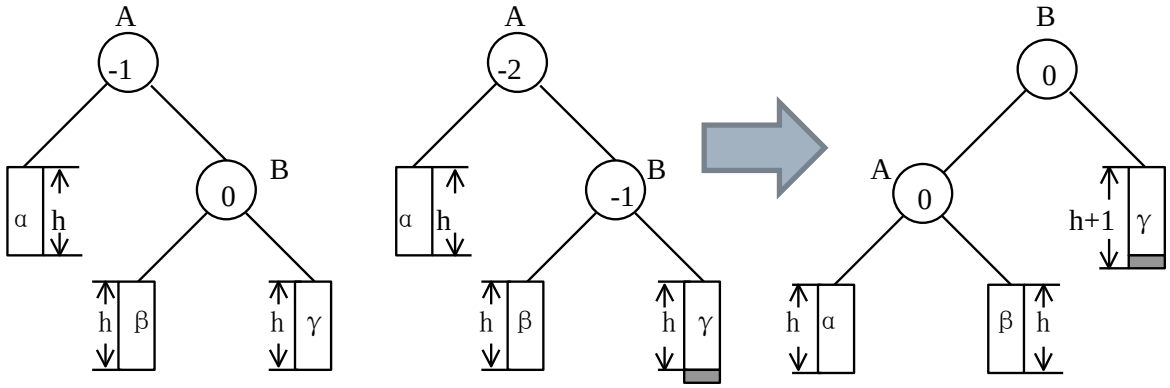
RR型调整

- 破坏平衡的原因：
  - 由于在A的右子女(R)的右子树(R)中插入结点，使A的平衡因子由-1变为-2而失去平衡

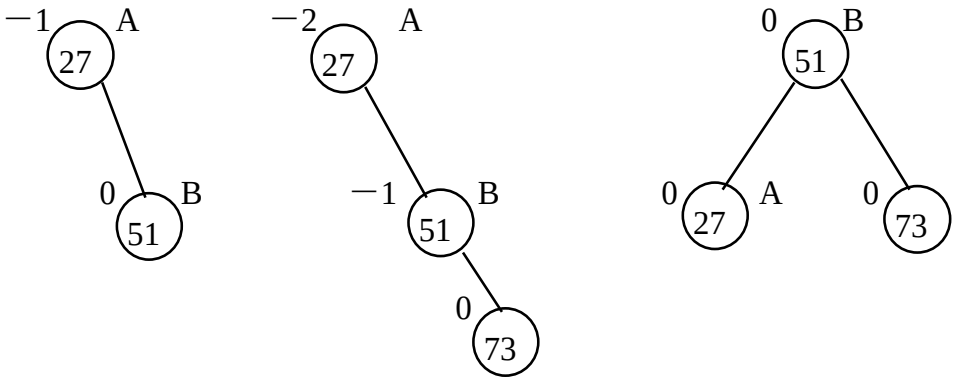


RR型调整

- RR型调整规则（与LL型的对称）：
  - 将A的右子女B提升为新二叉树的根
  - 原来的根A连同其左子树向左下旋转成为B的左子树；B的原左子树作为A的右子树
- 结合律： (α) A (βBγ)= (αAβ) B (γ)

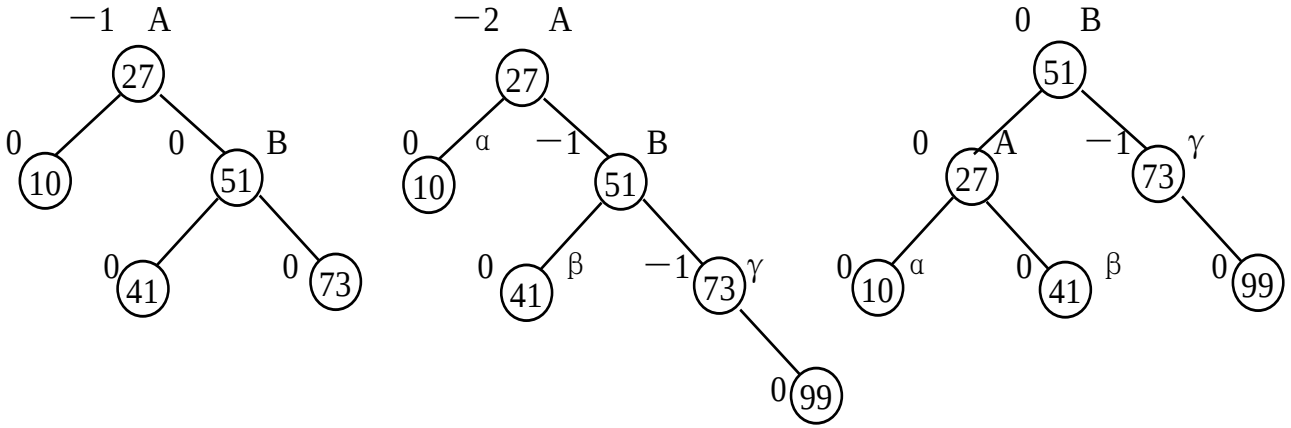


RR型调整（例）



输入 73 前　　输入 73 后　　调整后

(a) α 、 β 和 γ 都是空树的调整

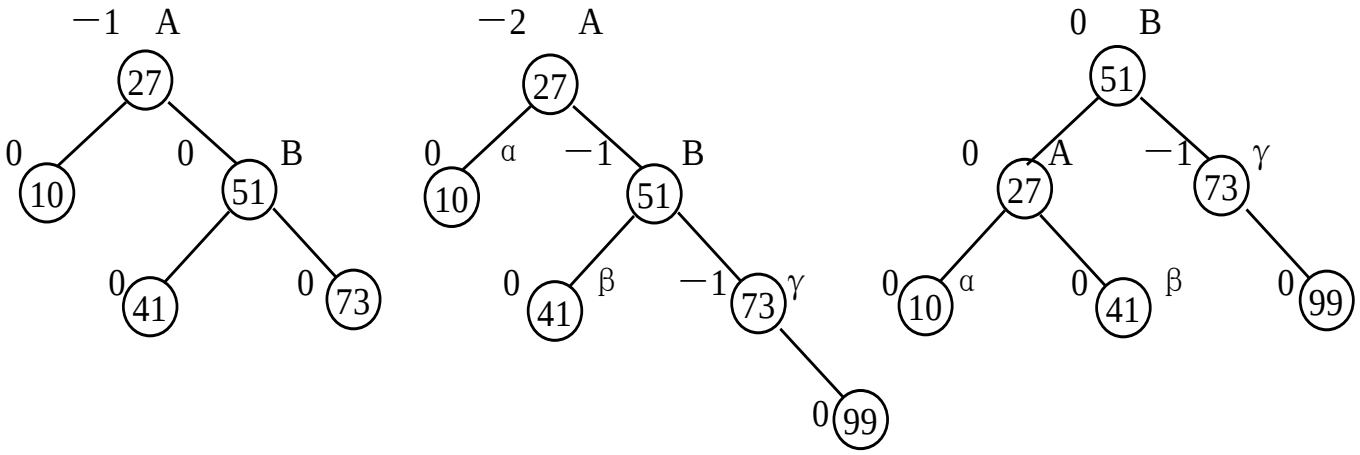


插入 99 前　　插入 99 后　　调整后

(b) α 、 β 和 γ 都是非空树的调整

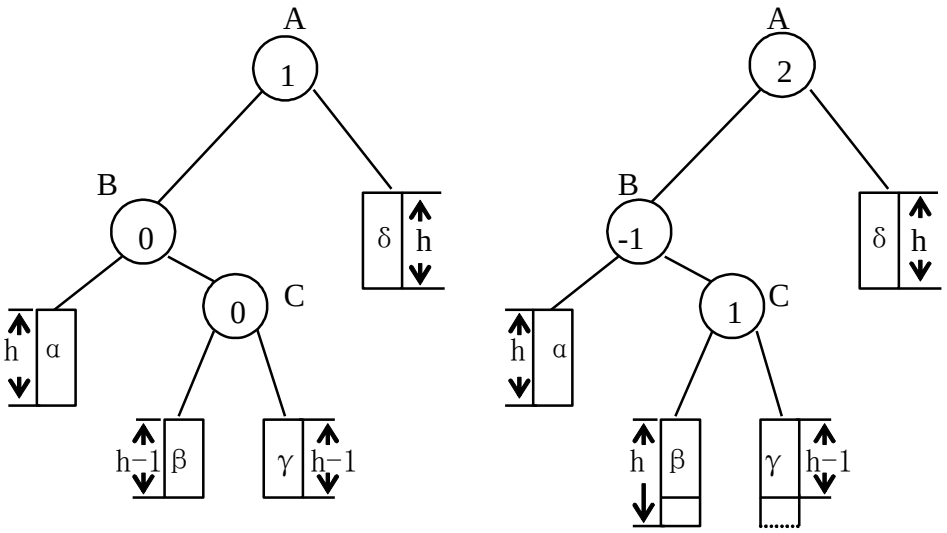
RR型调整算法

```
PAVLNode  rR(PAVLNode a, PAVLNode b)
{    /* a 指向最小不平衡子树根节点，b指向其右儿子*/
     a->bf=0;  a->rlink=b->llink;
     b->bf=0;  b->llink=a;
     return(b);
}
```



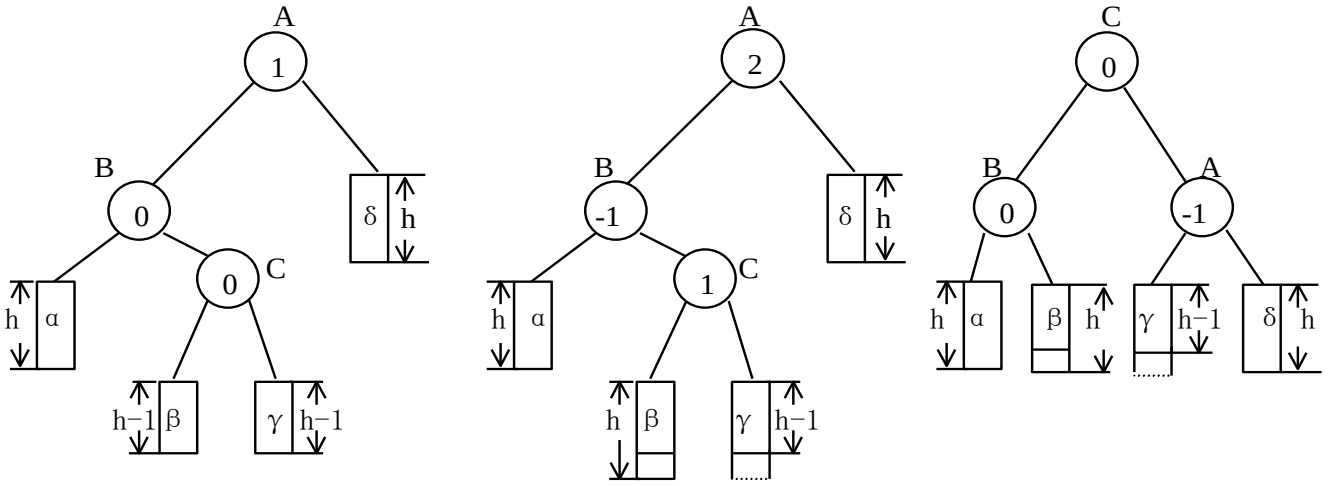
LR型调整

- 破坏平衡的原因：
  - 由于在A的左子女(L)的右子树(R)中插入结点，使A的平衡因子由1变为2而失去平衡

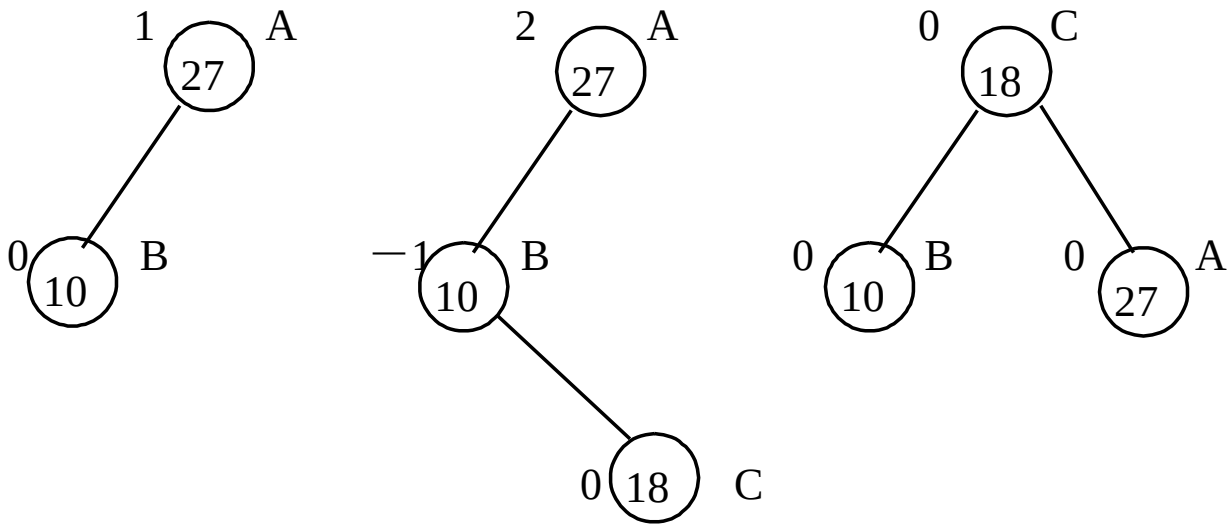


LR型调整

- 调整规则：
  - 设C为A的左子女的右子女，将A的孙子结点C提升为新二叉树的根；
  - 原C的父结点B连同其左子树α向左下旋转成为新根C的左子树，原C的左子树β成为B的右子树
  - 原根A连同其右子树δ向右下旋转成为新根C的右子树，原C的右子树γ成为A的左子树
- 结合律：((α)B(βCγ))A(δ)= (αBβ)C(γAδ)



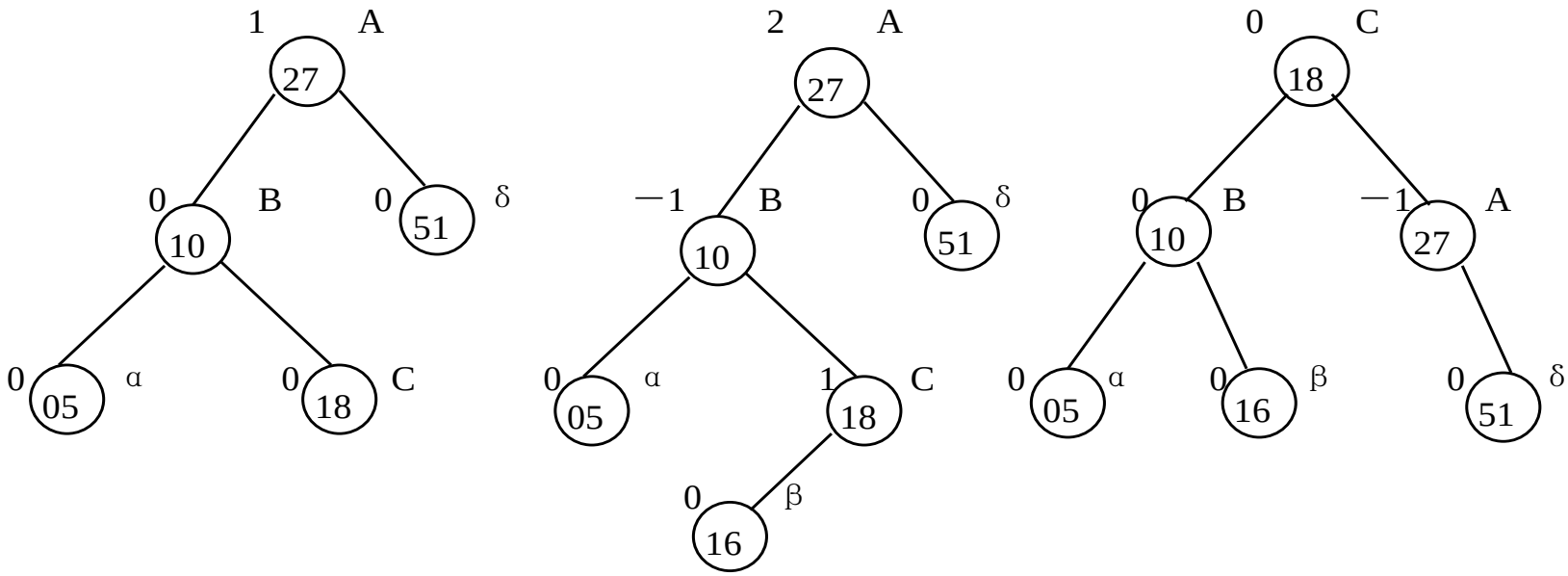
LR型调整(例)



输入 18 前　　输入 18 后　　调整后

(a) LR(0)型调整

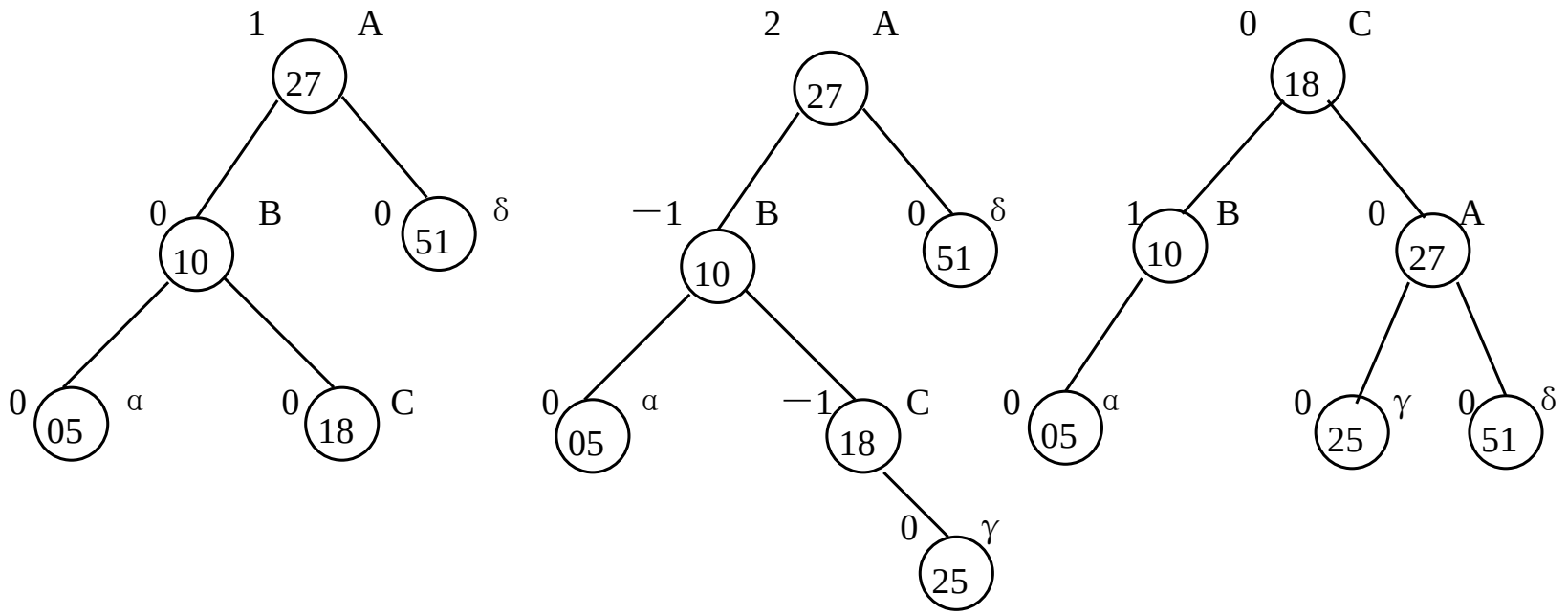
LR型调整(例)



输入 16 前　　输入 16 后　　调整后

(b) LR(L)型调整

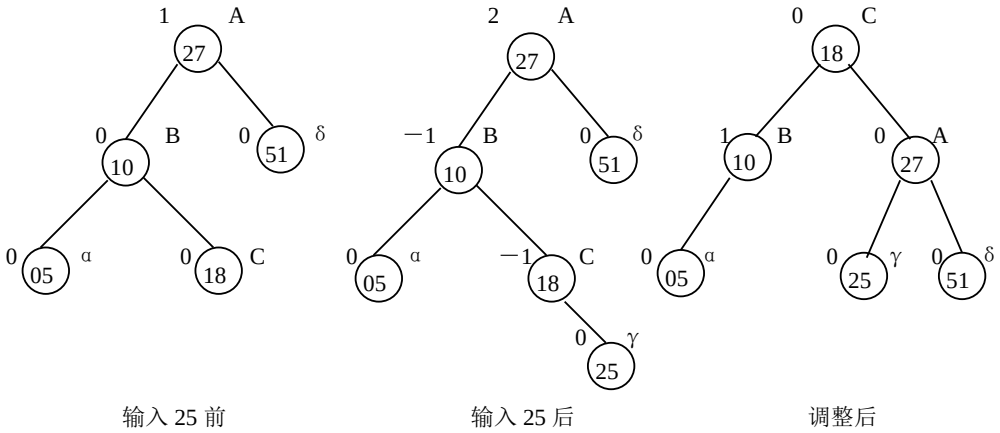
LR型调整(例)



(c) LR(R)型调整

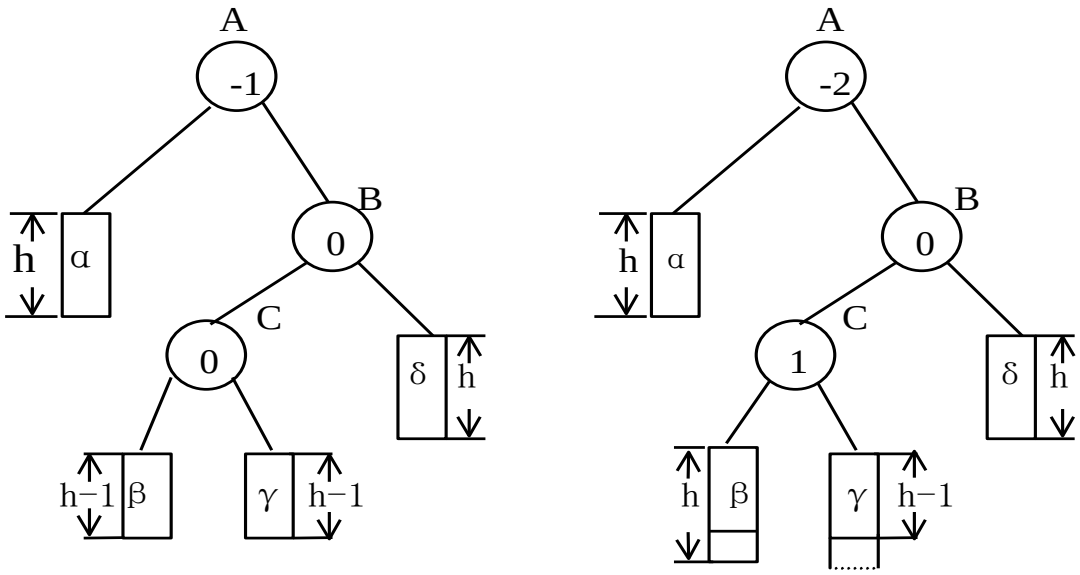
LR型调整算法

```
PAVLNode  lR(PAVLNode a,  PAVLNode b)
{
        PAVLNode c;
        c=b->rlink;
        a->llink=c->rlink;
        b->rlink=c->llink;
        c->llink=b;  c->rlink=a;
        switch(c->bf)
        {
        case 0:
          a->bf=0; b->bf=0;  break; /* LR(0)型调整 */
        case 1:
          a->bf=-1;b->bf=0; break; /* 新结点插在*c的左子树中，LR(L)型调整 */
        case -1:
          a->bf=0; b->bf=1; break; /* 新结点插在*c的右子树中，LR(R)型调整 */
        }
        c->bf=0;
        return(c);
 }
```



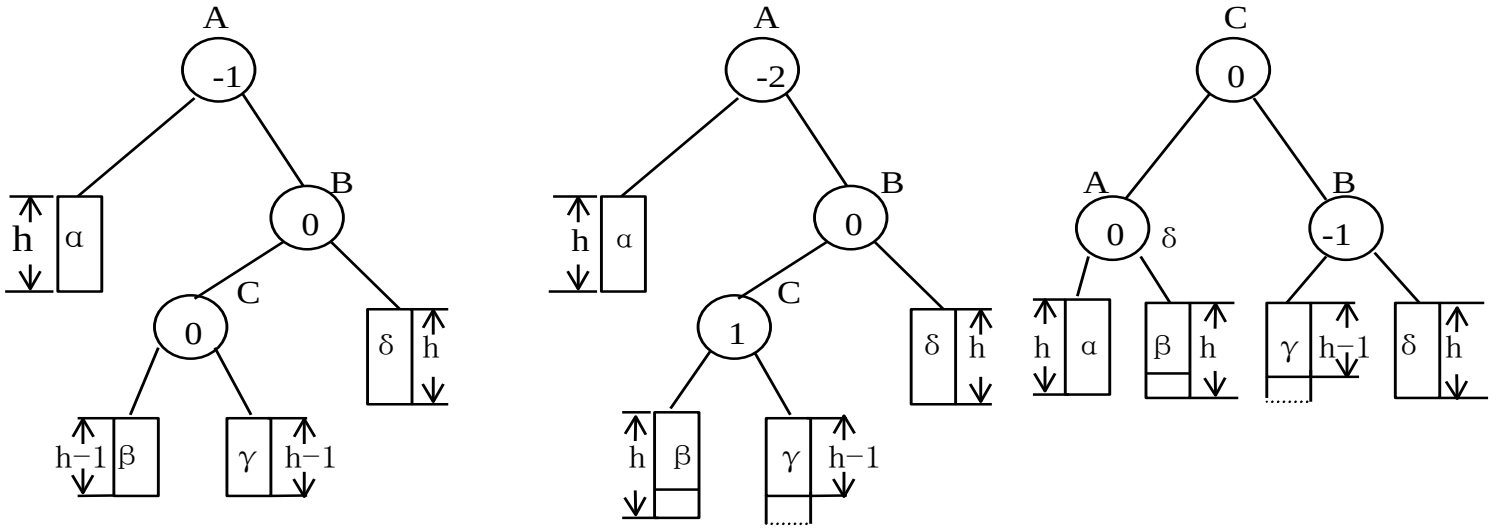
RL型调整

- 破坏平衡的原因：
  - 由于在A的右子女(R)的左子树(L)中插入结点，使A的平衡因子由－1变为－2而失去平衡
- RL型的调整规则与LR的对称

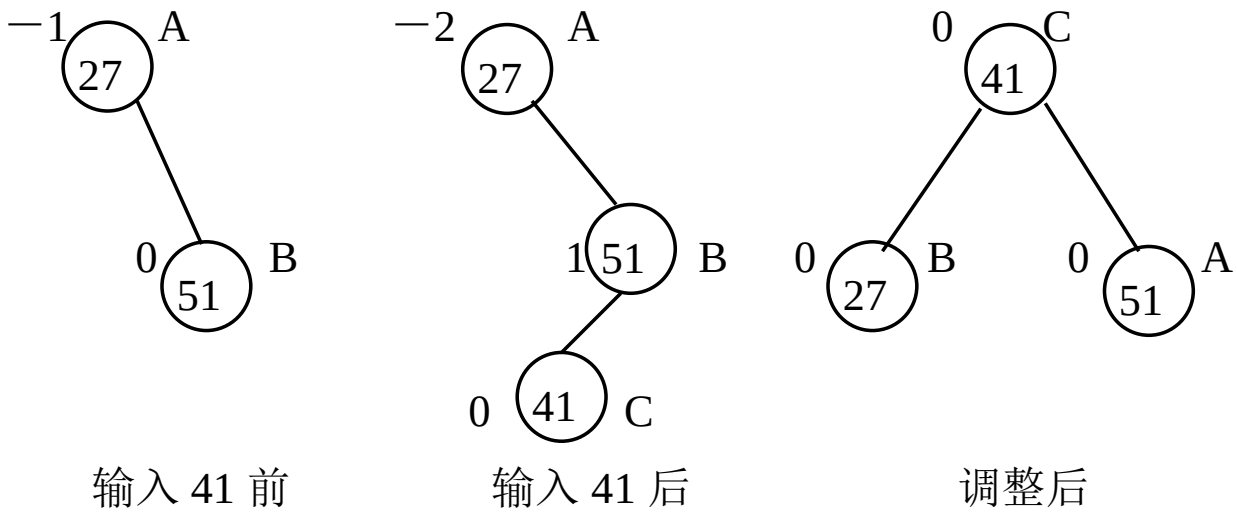


RL型调整

- 调整规则
  - 将A的孙子结点C提升为新二叉树的根，原来C的父结点B连同其右子树δ向右下旋转成为新根C的右子树，C的原右子树γ成为B的左子树
  - 原来的根A连同其左子树α向左下旋转成为新根C的左子树，原来C的左子树β成为A的右子树
- 结合律：(α)A((βCγ)B(δ))= (αAβ)C(γBδ)



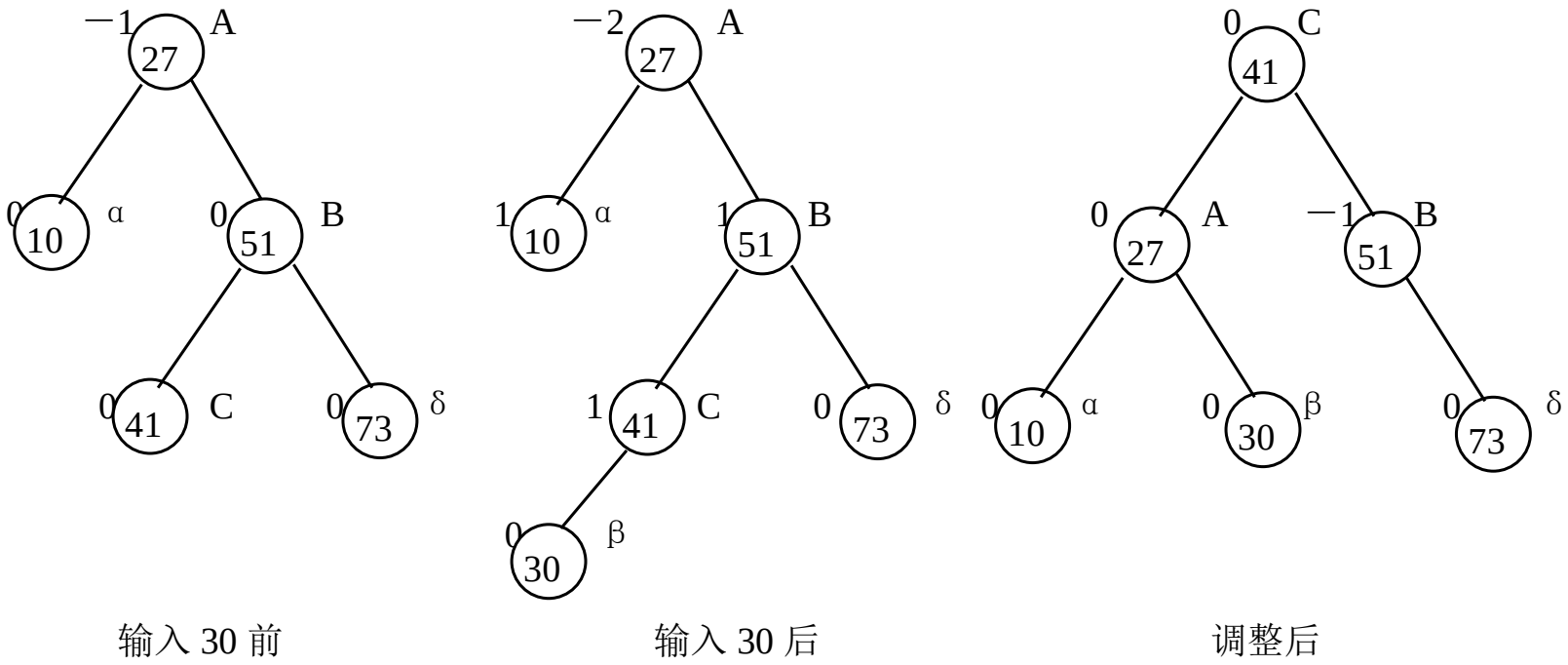
RL型调整（例）



输入 41 前　　输入 41 后　　调整后

(a) RL(0)型调整

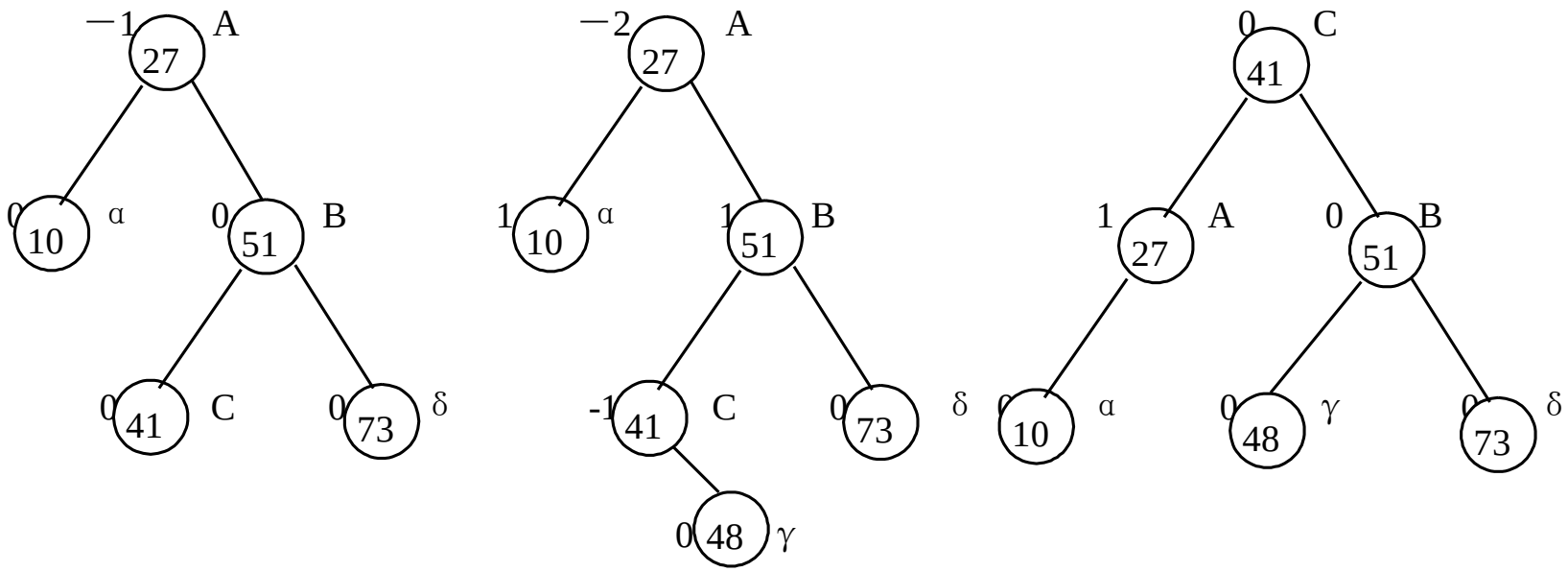
RL型调整（例）



输入 30 前　　输入 30 后　　调整后

(b) RL(L)型调整

RL型调整（例）

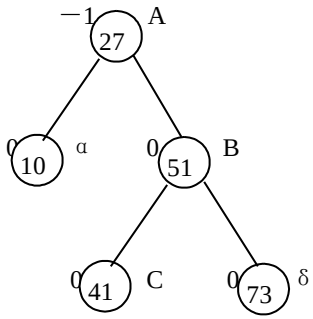


输入 48 前　　输入 48 后　　调整后

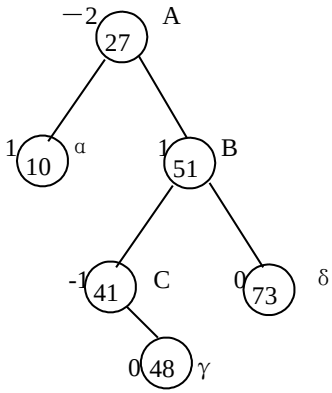
(c) RL(R)型调整

RL型调整算法

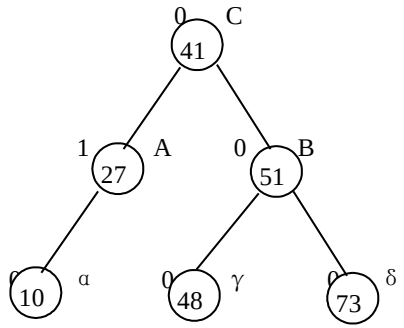
```
PAVLNode  rL(PAVLNode a, PAVLNode b)
{
        PAVLNode c;
        c=b->llink;
        a->rlink=c->llink;  b->llink=c->rlink;
        c->llink=a;  c->rlink=b;
        switch(c->bf)
        {
          case 0:
            a->bf=0; b->bf=0; break; /* *c本身就是插入结点，RL(0)型调整 */
          case 1:
            a->bf=0; b->bf=-1; break; /* 插在*c的左子树中，RL(L)型调整 */
          case -1:
            a->bf=1;  b->bf=0;  break;/* 插在*c的右子树中，RL(R)型调整 */
        }
        c->bf=0;
  return(c);
}
```



输入 48 前



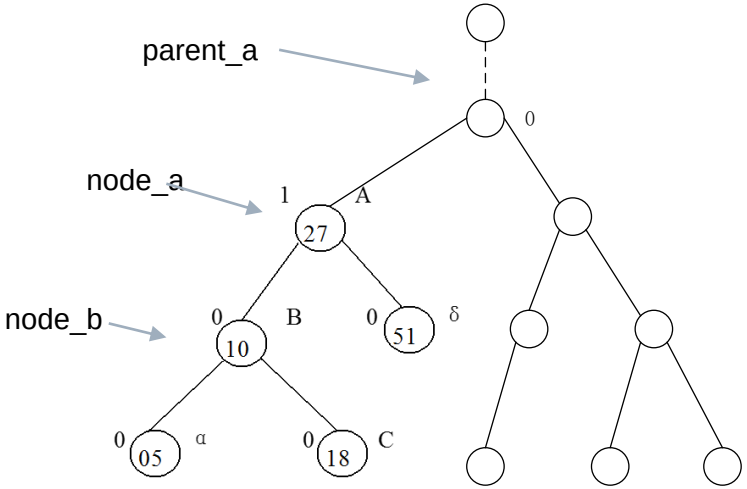
输入 48 后

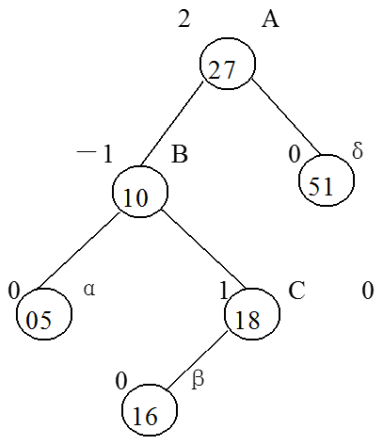


调整后

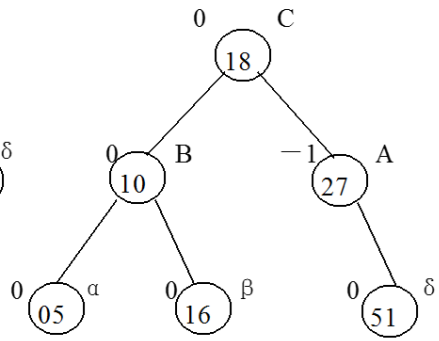
AVL树的平衡调整

- 上述所有的调整操作中，A为根的最小不平衡子树的深度在插入结点之前和调整之后相同，对A为根的子树之外的其它结点的平衡性无影响，调整后二叉排序树成为平衡二叉排序树
- AVL树的结点插入算法：
  - 新结点插入后，则要找到最小不平衡子树
    - 始终令A指向离插入位置最近、且平衡因子不为零的结点，如果这样的结点不存在，则A指向根结点；
    - 若新结点插入后使AVL树失去平衡，则*A是最小不平衡子树的根，parent A指向*A的父结点。
  - 新结点插入后，要修改一些结点的平衡因子





输入 16 后



调整后

问题

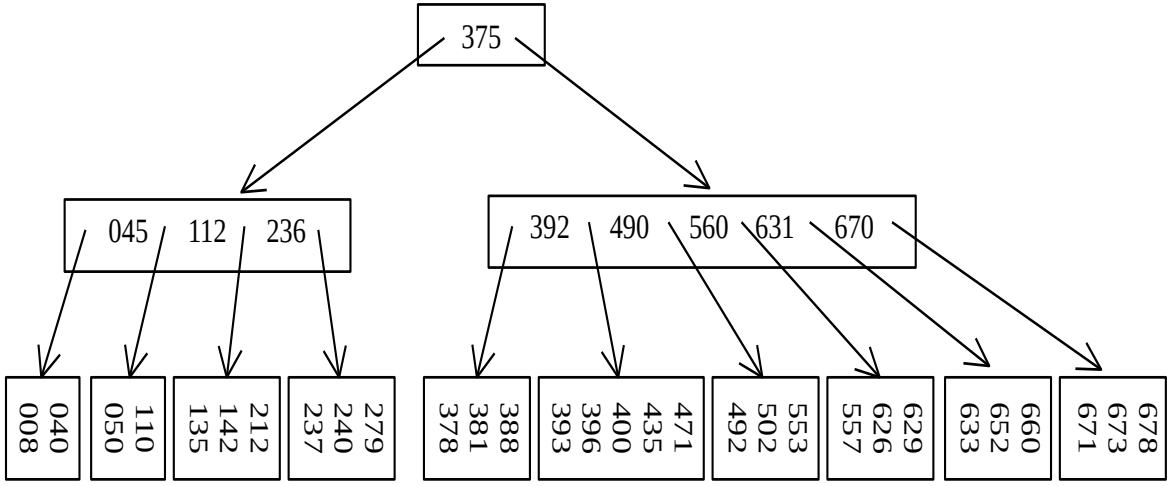
- 二叉排序树适用于组织较小的、内存中能容纳的字典。对于较大的必须存放在外存贮器上的字典，用二叉排序树组织索引就不合适。

- 外存文件索引中大量使用的是每个结点包括多个关键码的B树和B+树 （尽量减少节点的数量）

B树 （选学）

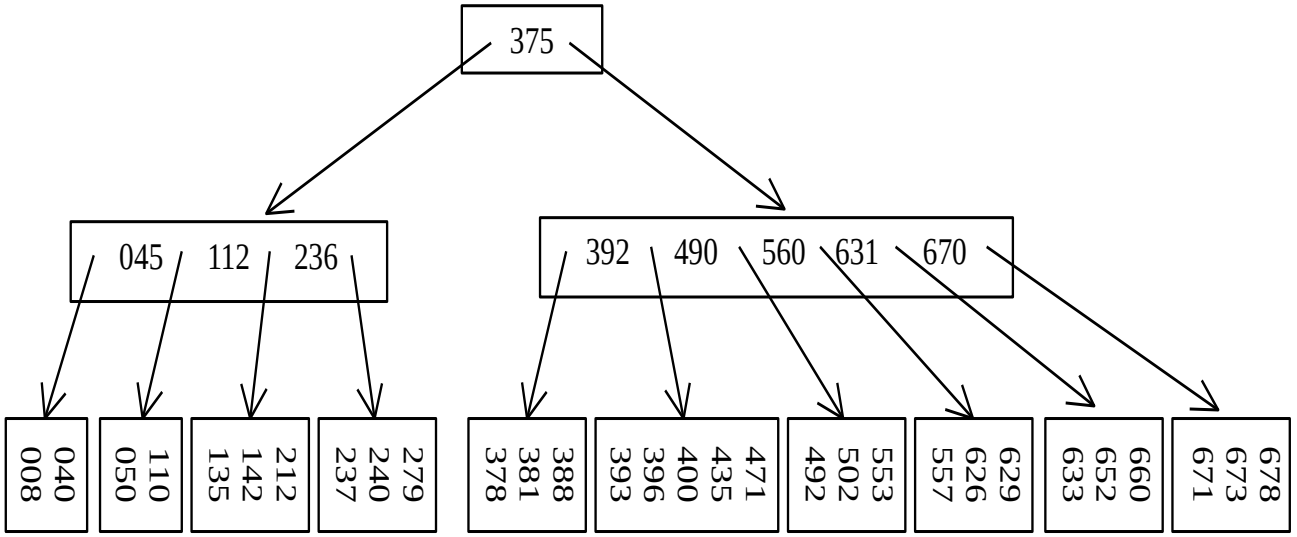
- 一棵m阶的B树满足下列条件：
  - ① 每个结点至多有m棵子树
  - ② 除根结点外，其它每个分支至少有$\lceil m/2 \rceil$棵子树
  - ③ 根结点至少有两棵子树(除非B树只有一个结点)
  - ④ 所有叶结点在同一层上
  - ⑤ 有j个孩子的非叶结点恰好有j-1个关键码，关键码按递增次序排列。
- 结点中包含的信息为：$(p_0,k_1,p_1,k_2,p_2,\ldots,k_{j-1},p_{j-1})$， 其中，$k_i$为关键码，且满足$k_i<k_{i+1}$；$p_i$为指向子树根结点的指针

6阶的B树



B树的检索

- B树上的检索过程与二叉排序树的相似
  - 根据要查找的关键码key，在根结点的关键码集合中进行顺序或二分法检索，
  - 若key=k_i，则检索成功；
  - 否则，key一定在某k_i和k_{i+1}之间，取指针p_i所指结点继续查找。
  - 重复上述检索过程，直到检索成功，或指针p_i为空，检索失败。

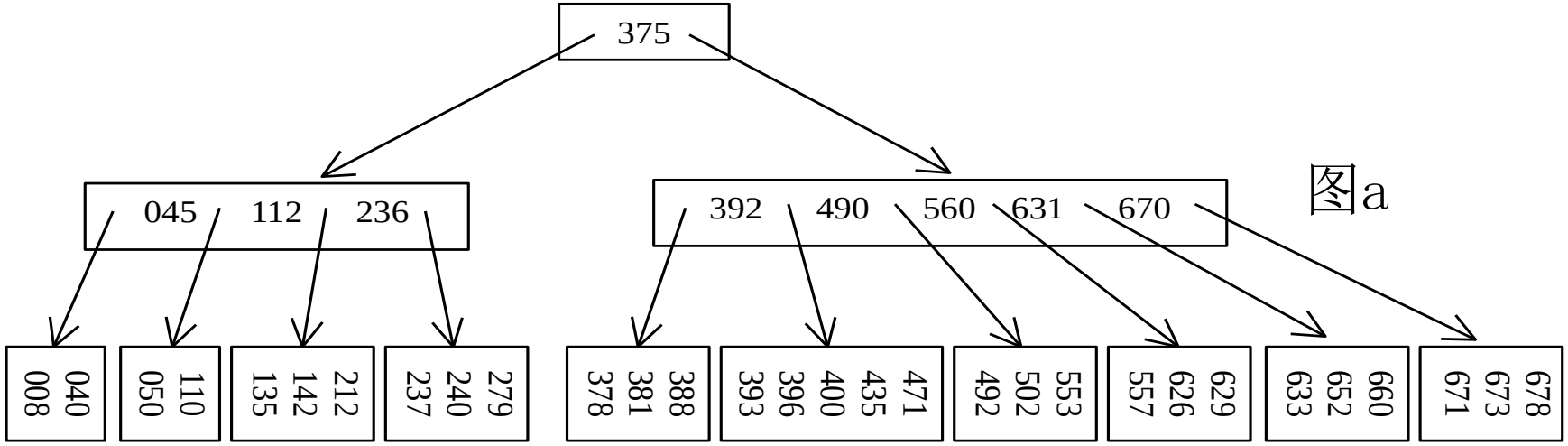


B树的插入运算

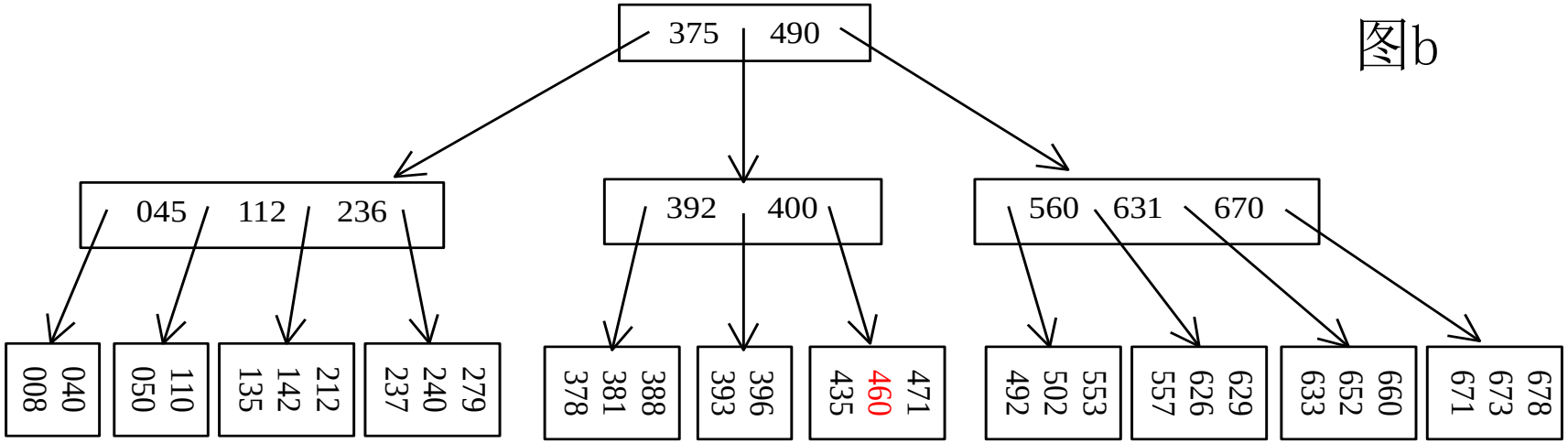
- 深度为h的m阶B树，结点一般是插在第h层。
- 首先检索到第h层，确定关键码应插入的结点：
- 若该结点中关键码个数小于m-1，则直接插入即可；
- 若该结点中关键码个数等于m-1，则将引起结点的分裂：
  - 以中间关键码为界将结点一分为二，产生一个新结点，并把中间关键码插入到父结点中
  - 若父结点中关键码数也为m-1，则需要再次分裂，最坏情况一直分裂到根结点，建立一个新的根结点，整个B树增加一层

例:在6阶B树（图a）中插入460

得到的B树如图b所示



图a



图b

树索引小结

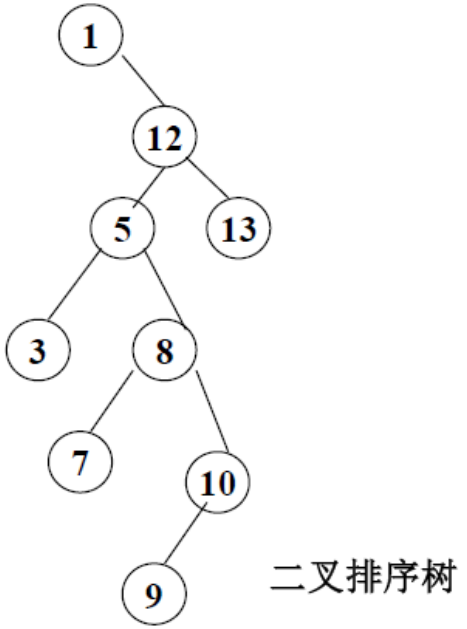
- 用树（诸如BST, AVL）作索引的好处：
  - 支持插入、删除操作
  - 保持平衡的开销相当小，尽管在设计上比较复杂
- 难点也是明显的：
  - log (N)的检索时间远逊于一个好的散列表
  - 为取得好的性能，树必须保持平衡
  - 若索引量巨大时，BST&AVL难于有效地在磁盘上存储

小结：字典的组织与检索算法

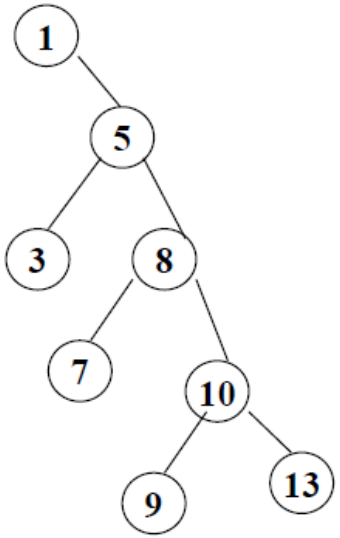
- 顺序存储：顺序检索、二分法检索、分块检索；
- 散列存储：散列函数、碰撞处理方法；
- 元素类型不同的字典：建立索引
  - 以关键码为结点的索引—二叉排序树和AVL树，熟悉它们的性质、检索、插入、构造、删除
- 各种检索算法的时间复杂度分析（ASL）
- 概念：静态字典，动态字典，ASL

练习

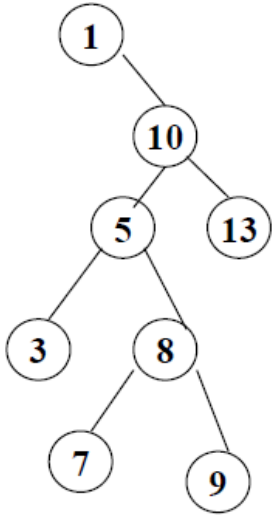
- 设数据集合d={1,12,5,8,3,10,7,13,9}
  - 依次取d中各数据，构造一棵二叉排序树；
  - 画出在二叉树中删除结点“12”后的树结构。



二叉排序树



删除12(第一种方法)



删除12(第二种方法)