数据结构-15周

本周范围：二叉搜索树、平衡二叉树（AVL树）、B树以及相关编程题

张天戈

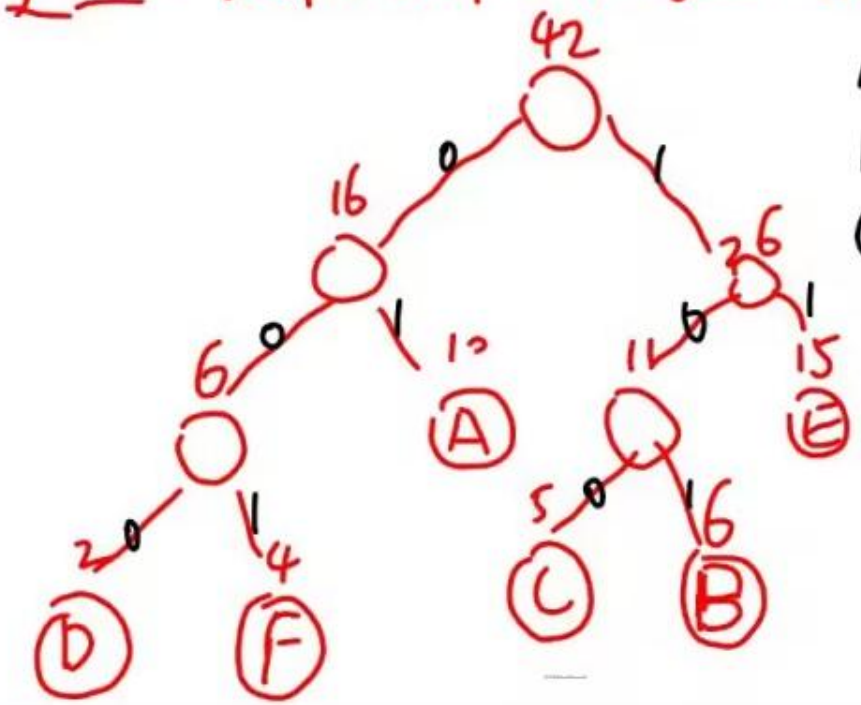
01 上周习题勘误

01 本周习题

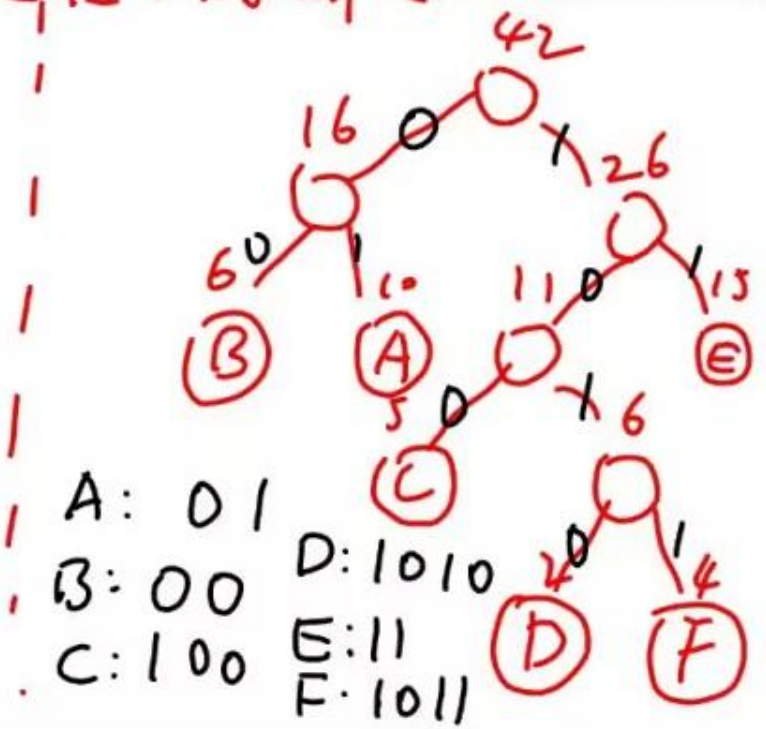
确保某字符编码前缀不是另一个字符前缀

5. 有一份电文中共使用 6 个字符：A、B、C、D、E、F，它们的出现频率依次为 10、6、5、2、15、4，试画出对应的赫夫曼树（请按左子树根节点的权小于等于右子树根节点的权的次序构造，左 0 右 1），并求出每个字符的赫夫曼编码。

更正：以下两种答案均正确（因为WPL值相同，都是最优二叉树）



A: 01
B: 101
C: 100
D: 000
E: 11
F: 001



A: 01
B: 00
C: 100
D: 1010
E: 11
F: 1011

5.5.1 二叉排序树（BST）

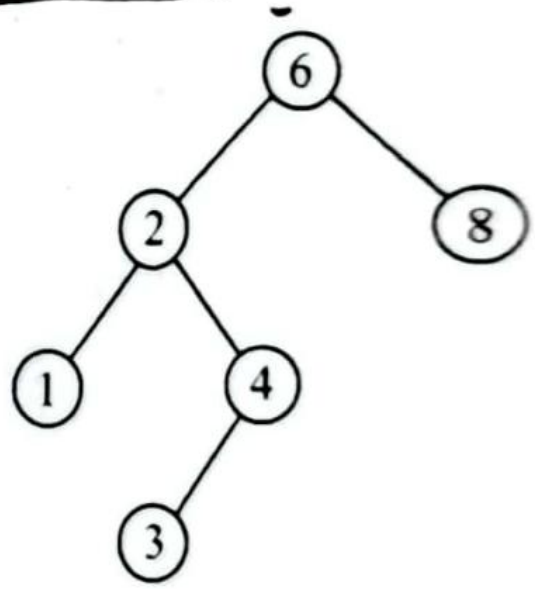
1. 二叉排序树的定义

二叉排序树（也称二叉查找树）或者是一棵空树，或者是具有下列特性的二叉树：

1）若左子树非空，则左子树上所有结点的值均小于根结点的值。

2）若右子树非空，则右子树上所有结点的值均大于根结点的值。

3）左、右子树也分别是一棵二叉排序树。



根据二叉排序树的定义，左子树结点值 < 根结点值 < 右子树结点值，所以对二叉排序树进行中序遍历，可以得到一个递增的有序序列。

例如，图 5.21 所示二叉排序树的中序遍历序列为 1 2 3 4 6 8。

01 本周习题

2. 在给出的函数体中编写二叉搜索树对某值为 key 的节点进行查找的 c++实现的代码，给出算法思想和关键注释。

```
BSTNode *BST_Search(BiTree T, int key){
    //在此处补充代码。T 的左子树为 lchild，右子树为 rchild，值为 data。
}
```

2. 二叉排序树的查找

图 5.21 一棵二叉排序树

二叉排序树的查找是从根结点开始，沿某个分支逐层向下比较的过程。若二叉排序树非空，先将给定值与根结点的关键字比较，若相等，则查找成功；若不等，如果小于根结点的关键字，则在根结点的左子树上查找，否则在根结点的右子树上查找。这显然是一个递归的过程。

二叉排序树的非递归查找算法：

```
BSTNode *BST_Search(BiTree T,ElemType key){
    while(T!=NULL&&key!=T->data){   //若树空或等于根结点值，则结束循环
        if(key<T->data)  T=T->lchild;   //小于，则在左子树上查找
        else  T=T->rchild;              //大于，则在右子树上查找
    }
    return T;
}
```

3．二叉排序树的插入

二叉排序树作为一种动态树表，其特点是树的结构通常不是一次生成的，而是在查找过程中，当树中不存在关键字值等于给定值的结点时再进行插入的。

插入结点的过程如下：若原二叉排序树为空，则直接插入结点；否则，若关键字 k 小于根结点值，则插入到左子树，若关键字 k 大于根结点值，则插入到右子树。插入的结点一定是一个新添加的叶结点，且是查找失败时的查找路径上访问的最后一个结点的左孩子或右孩子。如图 5.22 所示在一个二叉排序树中依次插入结点 28 和结点 58，虚线表示的边是其查找的路径。

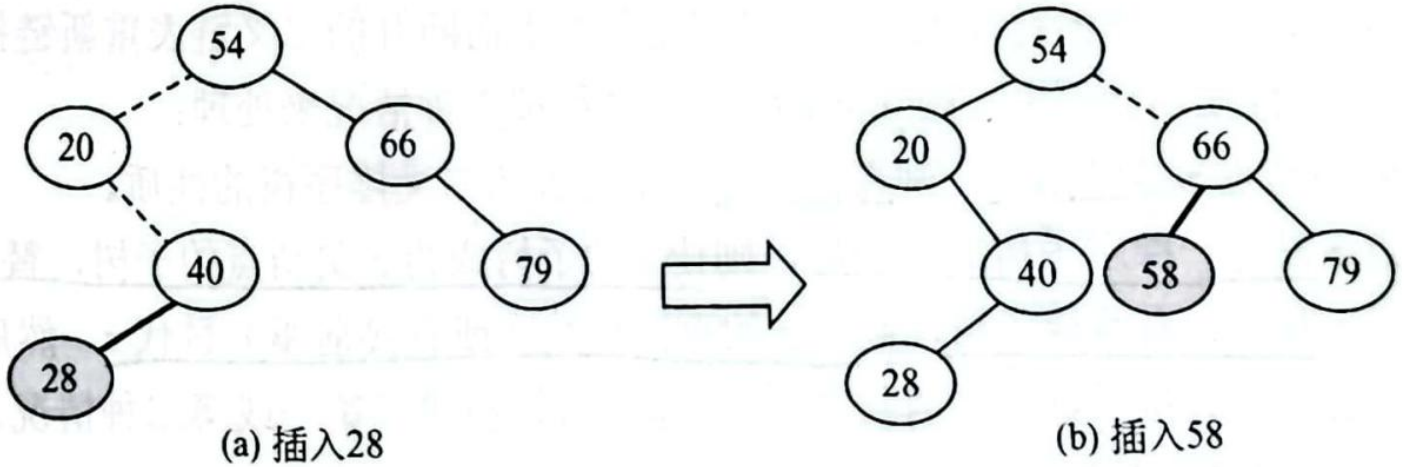


图 5.22　向二叉排序树中插入结点

二叉搜索树的增删改查等

二叉排序树插入操作的算法描述如下：

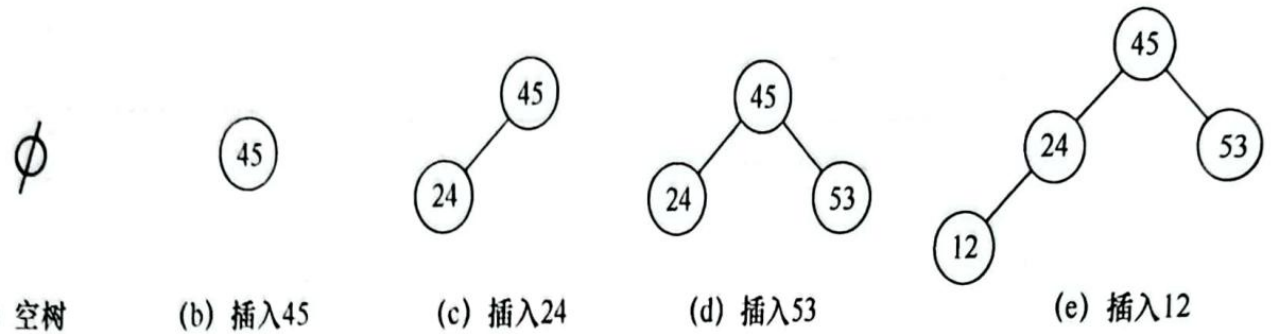
```
int BST_Insert(BiTree &T,KeyType k){
    if(T==NULL){                          //原树为空，新插入的记录为根结点
        T=(BiTree)malloc(sizeof(BSTNode));
        T->key=k;
        T->lchild=T->rhild=NULL;
        return  1;                        //返回 1，插入成功
    }
    else if(k==T->key)                    //树中存在相同关键字的结点，插入失败
        return  0;
    else if(k<T->key)                     //插入到 T 的左子树
        return  BST_Insert(T->lchild,k);
    else                                  //插入到 T 的右子树
        return  BST_Insert(T->rchild,k);
}
```

01 本周习题

1. 给定一输入序列{ 16, 3, 7, 11, 9, 26, 18, 14, 15 }，构建二叉搜索树，并给出等概率查找各节点的情况下，查找成功的平均查找长度。

4. 二叉排序树的构造

从一棵空树出发，依次输入元素，将它们插入二叉排序树中的合适位置。设查找的关键字序列为{45, 24, 53, 45, 12, 24}，则生成的二叉排序树如图 5.23 所示。



(a) 空树　(b) 插入45　(c) 插入24　(d) 插入53　(e) 插入12

图 5.23　二叉排序树的构造过程

构造二叉排序树的算法描述如下：

```
void Creat_BST(BiTree &T,KeyType str[],int n){
    T=NULL;          //初始时 T 为空树
    int i=0;
    while(i<n){      //依次将每个关键字插入到二叉排序树中
        BST_Insert(T,str[i]);
        i++;
    }
}
```

5. 二叉排序树的删除

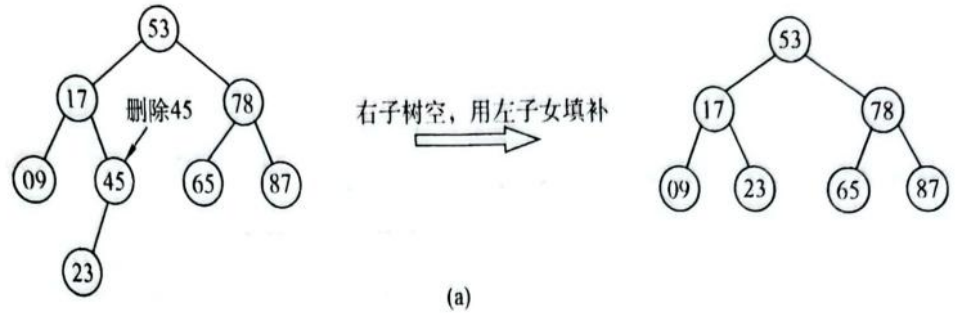
在二叉排序树中删除一个结点时，不能把以该结点为根的子树上的结点都删除，必须先把被删除结点从存储二叉排序树的链表上摘下，将因删除结点而断开的二叉链表重新链接起来，同时确保二叉排序树的性质不会丢失。删除操作的实现过程按 3 种情况来处理：

① 若被删除结点 z 是叶结点，则直接删除，不会破坏二叉排序树的性质。

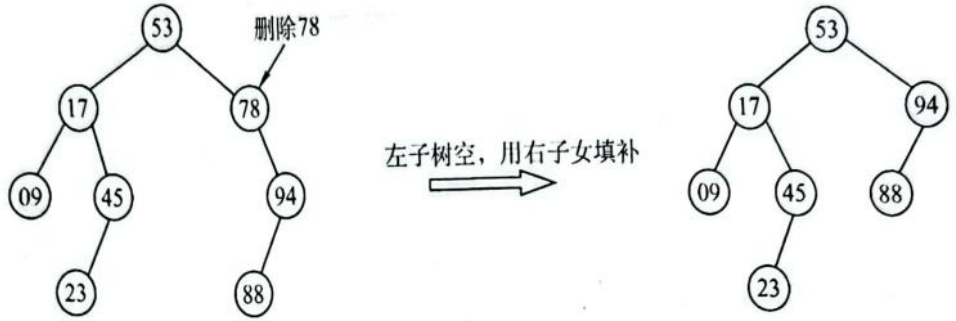
② 若结点 z 只有一棵左子树或右子树，则让 z 的子树成为 z 父结点的子树，替代 z 的位置。

③ 若结点 z 有左、右两棵子树，则令 z 的直接后继（或直接前驱）替代 z，然后从二叉排序树中删去这个直接后继（或直接前驱），这样就转换成了第一或第二种情况。

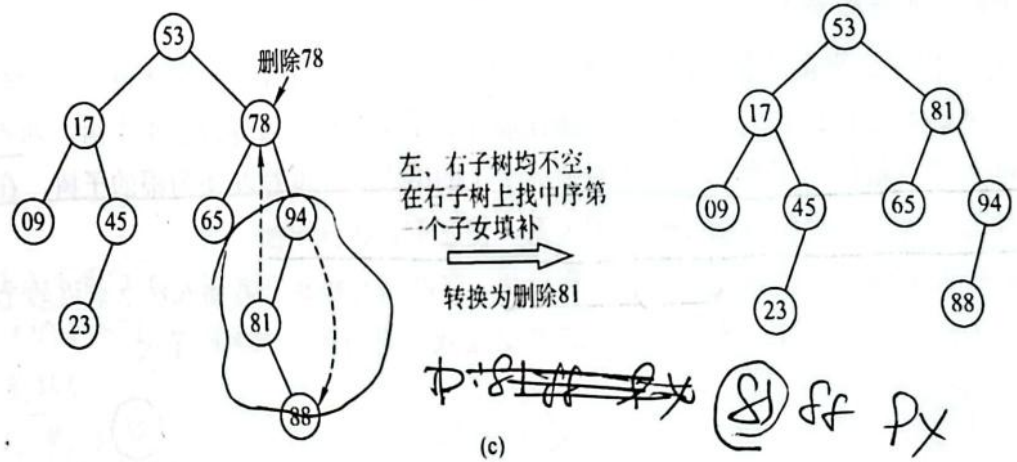
二叉搜索树的增删改查等



(a)



(b)



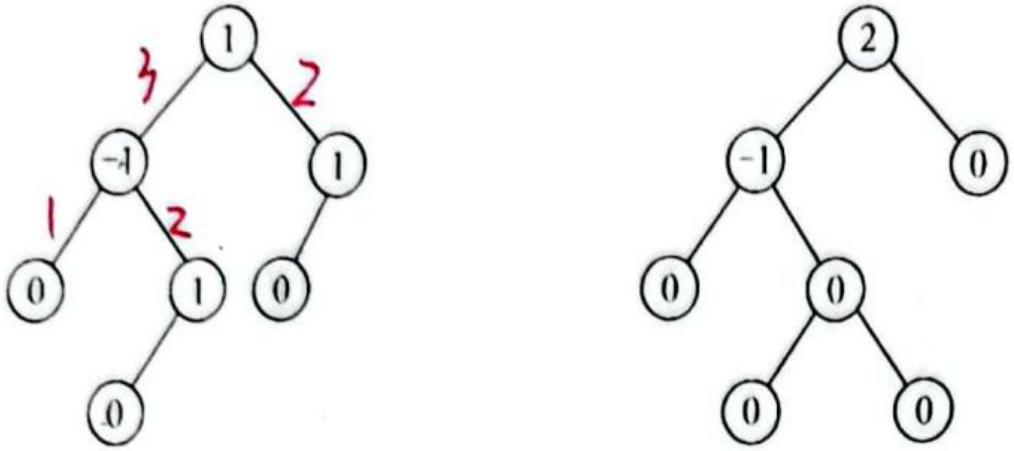
(c)

01 平衡二叉树的定义及插入

山东大学 SHANDONG UNIVERSITY

1. 平衡二叉树的定义

为避免树的高度增长过快，降低二叉排序树的性能，规定在插入和删除二叉树结点时，要保证任意结点的左、右子树高度差的绝对值不超过 1，将这样的二叉树称为平衡二叉树（Balanced Binary Tree），简称平衡树。定义结点左子树与右子树的高度差为该结点的平衡因子，则平衡二叉树结点的平衡因子的值只可能是 -1、0 或 1。



(a) 平衡二叉树　　(b) 不平衡的二叉树

图 5.26　平衡二叉树和不平衡的二叉树

因此，平衡二叉树可定义为或者是一棵空树，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高度差的绝对值不超过 1。图 5.26(a)所示是平衡二叉树，图 5.26(b)所示是不平衡的二叉树。结点中的值为该结点的平衡因子。

2. 平衡二叉树的插入

二叉排序树保证平衡的基本思想如下：每当在二叉排序树中插入（或删除）一个结点时，首先检查其插入路径上的结点是否因为此次操作而导致了不平衡。若导致了不平衡，则先找到插入路径上离插入结点最近的平衡因子的绝对值大于 1 的结点 A，再对以 A 为根的子树，在保持二叉排序树特性的前提下，调整各结点的位置关系，使之重新达到平衡。

注意：每次调整的对象都是最小不平衡子树，即以插入路径上离插入结点最近的平衡因子的绝对值大于 1 的结点作为根的子树。图 5.27 中的虚线框内为最小不平衡子树。

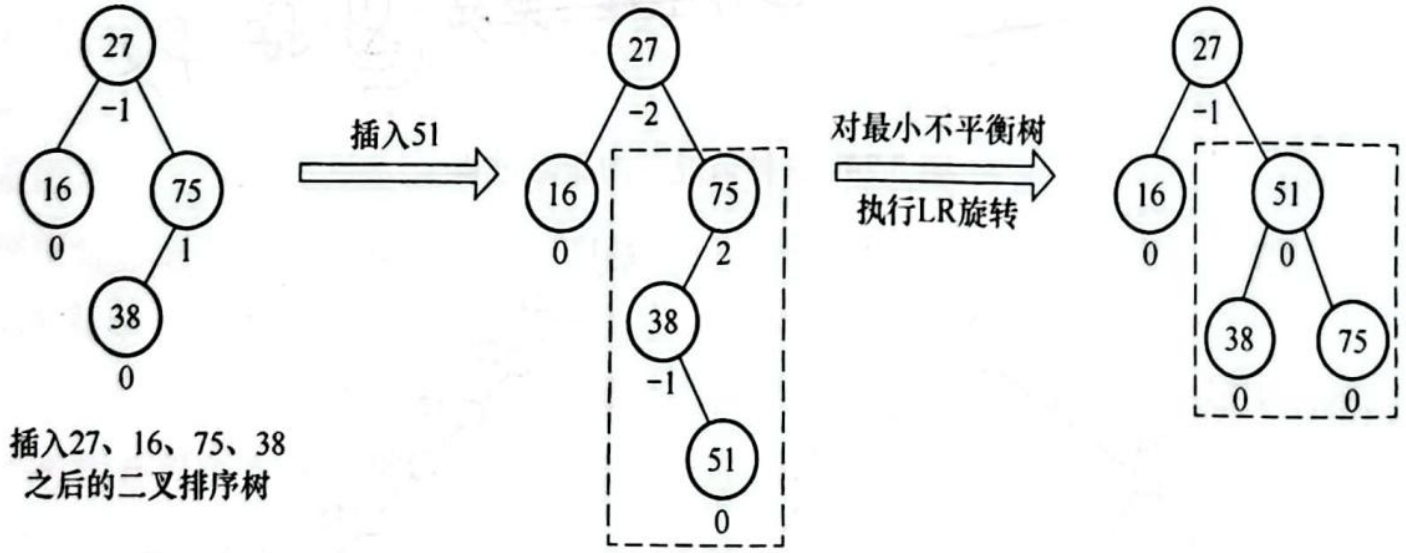


图 5.27　最小不平衡子树示意

平衡二叉树的定义及插入

平衡二叉树的插入过程的前半部分与二叉排序树相同，但在新结点插入后，若造成查找路径上的某个结点不再平衡，则需要做出相应的调整。可将调整的规律归纳为下列 4 种情况：

1）LL 平衡旋转（右单旋转）。由于在结点 A 的左孩子（L）的左子树（L）上插入了新结点，A 的平衡因子由 1 增至 2，导致以 A 为根的子树失去平衡，需要一次向右的旋转操作。将 A 的左孩子 B 向右上旋转代替 A 成为根结点，将 A 结点向右下旋转成为 B 的右子树的根结点，而 B 的原右子树则作为 A 结点的左子树。

如图 5.28 所示，结点旁的数值代表结点的平衡因子，而用方块表示相应结点的子树，下方数值代表该子树的高度。

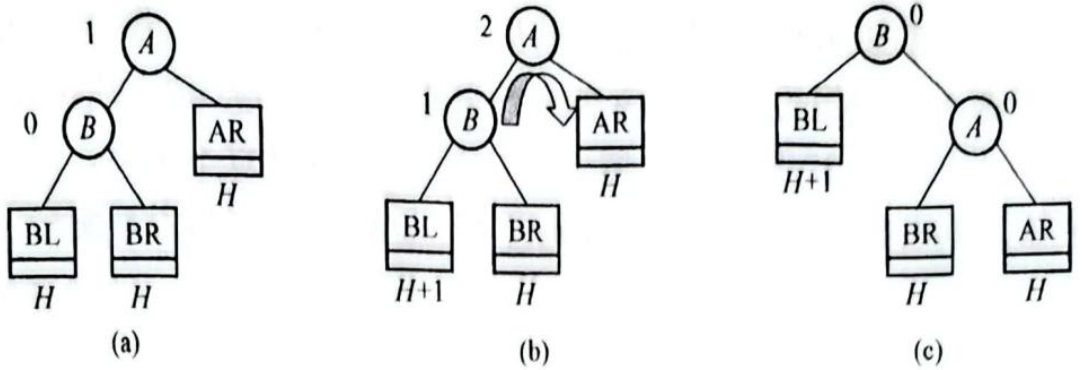
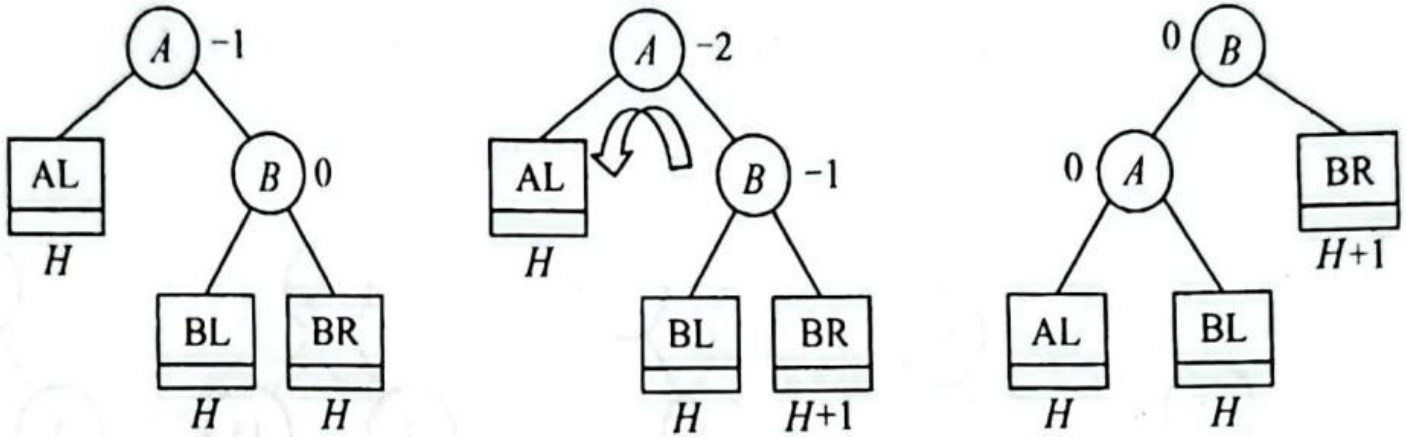


图 5.28　LL 平衡旋转

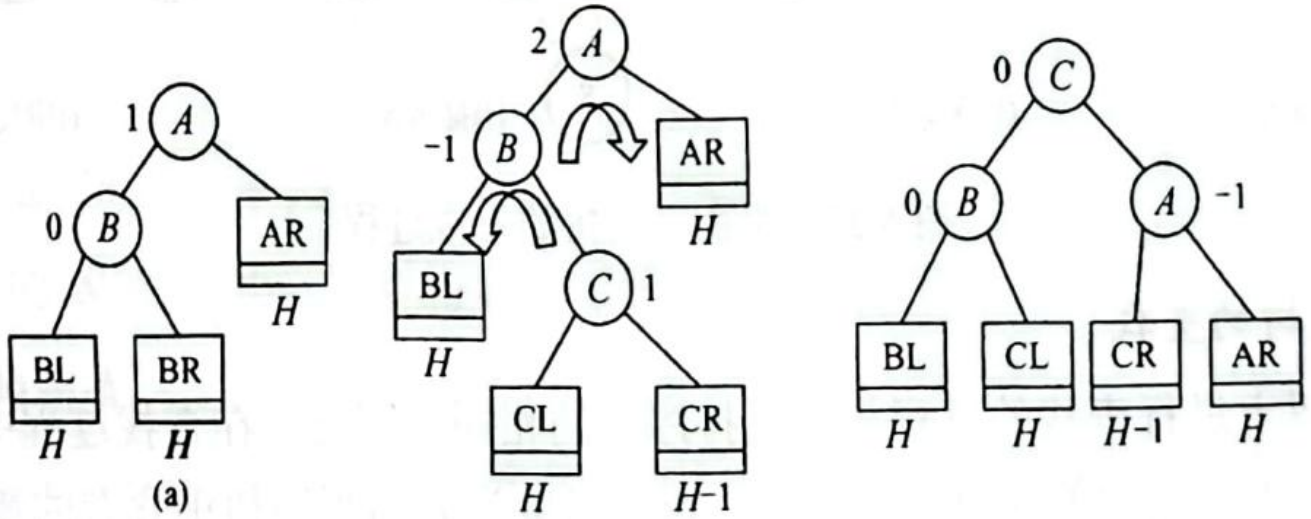
2）RR平衡旋转（左单旋转）。由于在结点 A 的右孩子（R）的右子树（R）上插入了新结点，A 的平衡因子由-1 减至-2，导致以 A 为根的子树失去平衡，需要一次向左的旋转操作。将 A 的右孩子 B 向左上旋转代替 A 成为根结点，将 A 结点向左下旋转成为 B 的左子树的根结点，而 B 的原左子树则作为 A 结点的右子树，如图 5.29 所示。



(a) 插入结点前　(b) 插入结点导致不平衡　(c) RR旋转(左单旋转)

图 5.29　RR 平衡旋转

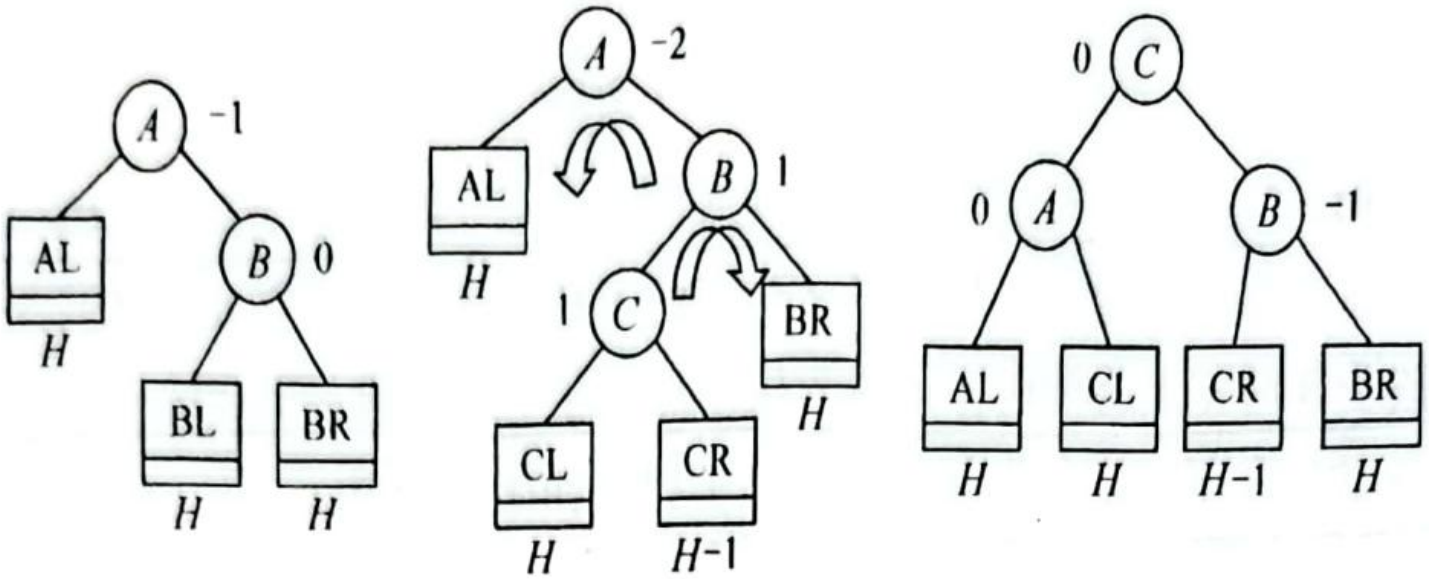
3）LR 平衡旋转（先左后右双旋转）。由于在 A 的左孩子（L）的右子树（R）上插入新结点，A 的平衡因子由 1 增至 2，导致以 A 为根的子树失去平衡，需要进行两次旋转操作，先左旋转后右旋转。先将 A 结点的左孩子 B 的右子树的根结点 C 向左上旋转提升到 B 结点的位置，然后再把该 C 结点向右上旋转提升到 A 结点的位置，如图 5.30 所示。



(a) 插入结点前　(b) 插入结点导致不平衡　(c) LR旋转(双旋转)

图 5.30　LR 平衡旋转

4）RL 平衡旋转（先右后左双旋转）。由于在 A 的右孩子（R）的左子树（L）上插入新结点，A 的平衡因子由-1 减至-2，导致以 A 为根的子树失去平衡，需要进行两次旋转操作，先右旋转后左旋转。先将 A 结点的右孩子 B 的左子树的根结点 C 向右上旋转提升到 B 结点的位置，然后再把该 C 结点向左上旋转提升到 A 结点的位置，如图 5.31 所示。



(a) 插入结点前　(b) 插入结点导致不平衡　(c) RL旋转(双旋转)

01 本周习题

山东大学 SHANDONG UNIVERSITY

3. 对给定输入序列{ 19, 5, 7, 11, 26, 18, 16, 17 }，构建 AVL 树，并给出等概率查找各节点的情况下，查找成功的平均查找长度。

B-树的定义及插入和删除

B 树，又称多路平衡查找树，B 树中所有结点的孩子个数的最大值称为 B 树的阶，通常用 m 表示。一棵 m 阶 B 树或为空树，或为满足如下特性的 m 叉树：

1）树中每个结点至多有 m 棵子树，即至多含有 $m-1$ 个关键字。

2）若根结点不是终端结点，则至少有两棵子树。

3）除根结点外的所有非叶结点至少有 $\lceil m/2 \rceil$ 棵子树，即至少含有 $\lceil m/2 \rceil - 1$ 个关键字。

4）所有非叶结点的结构如下：

n	P_0	K_1	P_1	K_2	P_2	$\cdots$	K_n	P_n

其中，K_i（$i=1,2,\cdots,n$）为结点的关键字，且满足 $K_1<K_2<\cdots<K_n$；P_i（$i=0,1,\cdots,n$）为指向子树根结点的指针，且指针 P_{i-1} 所指子树中所有结点的关键字均小于 K_i，P_i 所指子树中所有结点的关键字均大于 K_i，n（$\lceil m/2 \rceil - 1 \le n \le m-1$）为结点中关键字的个数。

5）所有的叶结点都出现在同一层次上，并且不带信息（可以视为外部结点或类似于折半查找判定树的查找失败结点，实际上这些结点不存在，指向这些结点的指针为空）。

B 树是所有结点的平衡因子均等于 0 的多路平衡查找树。

图 7.4 所示的 B 树中所有结点的最大孩子数 $m=5$，因此它是一棵 5 阶 B 树，在 m 阶 B 树中结点最多可以有 m 个孩子。可以借助该实例来分析上述性质：

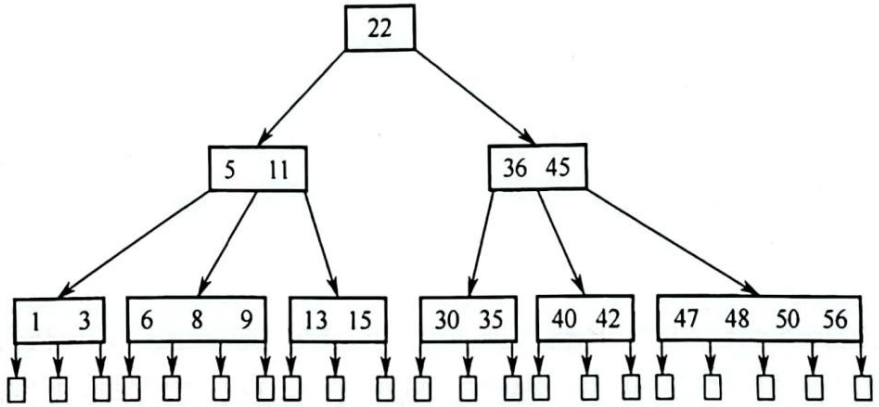


图 7.4 一棵 5 阶 B 树的实例

1）结点的孩子个数等于该结点中关键字个数加 1。

2）如果根结点没有关键字就没有子树，此时 B 树为空；如果根结点有关键字，则其子树必然大于等于两棵，因为子树个数等于关键字个数加 1。

3）除根结点外的所有非终端结点至少有$\lceil m/2\rceil=\lceil 5/2\rceil=3$ 棵子树（即至少有$\lceil m/2\rceil-1=\lceil 5/2\rceil-1=2$ 个关键字），至多有 5 棵子树（即至多有 4 个关键字）。

4）结点中关键字从左到右递增有序，关键字两侧均有指向子树的指针，左边指针所指子树的所有关键字均小于该关键字，右边指针所指子树的所有关键字均大于该关键字。或者看成下层结点关键字总是落在由上层结点关键字所划分的区间内，如第二层最左结点的关键字划分成了 3 个区间：$(-\infty, 5), (5, 11), (11, +\infty)$，该结点 3 个指针所指子树的关键字均落在这 3 个区间内。

5）所有叶结点均在第 4 层，代表查找失败的位置。

5. 计算“含有 n 个非叶节点的 m 阶 B-树”中至少包含多少个关键字？

3. B 树的插入

与二叉查找树的插入操作相比，B 树的插入操作要复杂得多。在二叉查找树中，仅需查找到需插入的终端结点的位置。但是，在 B 树中找到插入的位置后，并不能简单地将其添加到终端结点中，因为此时可能会导致整棵树不再满足 B 树定义中的要求。将关键字 key 插入 B 树的过程如下：

1）定位。利用前述的 B 树查找算法，找出插入该关键字的最低层中的某个非叶结点（在 B 树中查找 key 时，会找到表示查找失败的叶结点，这样就确定了最底层非叶结点的插入位置。注意：插入位置一定是最低层中的某个非叶结点）。

2）插入。在 B 树中，每个非失败结点的关键字个数都在区间$[\lceil m/2\rceil-1, m-1]$内。插入后的结点关键字个数小于 m，可以直接插入；插入后检查被插入结点内关键字的个数，当插入后的结点关键字个数大于 $m-1$ 时，必须对结点进行分裂。

分裂的方法是：取一个新结点，在插入 key 后的原结点，从中间位置（$\lceil m/2\rceil$）将其中的关键字分为两部分，左部分包含的关键字放在原结点中，右部分包含的关键字放到新结点中，中间位置（$\lceil m/2\rceil$）的结点插入原结点的父结点。若此时导致其父结点的关键字个数也超过了上限，则继续进行这种分裂操作，直至这个过程传到根结点为止，进而导致 B 树高度增 1。

对于 $m=3$ 的 B 树，所有结点中最多有 $m-1=2$ 个关键字，若某结点中已有两个关键字，则结点已满，如图 7.5(a)所示。插入一个关键字 60 后，结点内的关键字个数超过了 $m-1$，如图 7.5(b)所示，此时必须进行结点分裂，分裂的结果如图 7.5(c)所示。

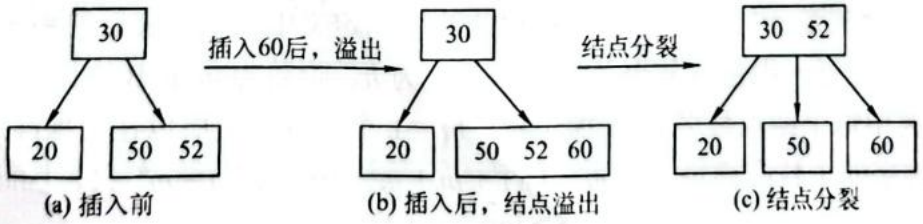


图 7.5　结点的“分裂”示意

4. B 树的删除

B 树中的删除操作与插入操作类似，但要稍微复杂一些，即要使得删除后的结点中的关键字个数≥$\lceil m/2 \rceil - 1$，因此将涉及结点的“合并”问题。

当被删关键字 k 不在终端结点（最低层非叶结点）中时，可以用 k 的前驱（或后继）k'来替代 k，然后在相应的结点中删除 k'，关键字 k'必定落在某个终端结点中，则转换成了被删关键字在终端结点中的情形。在图 7.6 的 4 阶 B 树中，删除关键字 80，用其前驱 78 替代，然后在终端结点中删除 78。因此只需讨论删除终端结点中关键字的情形。

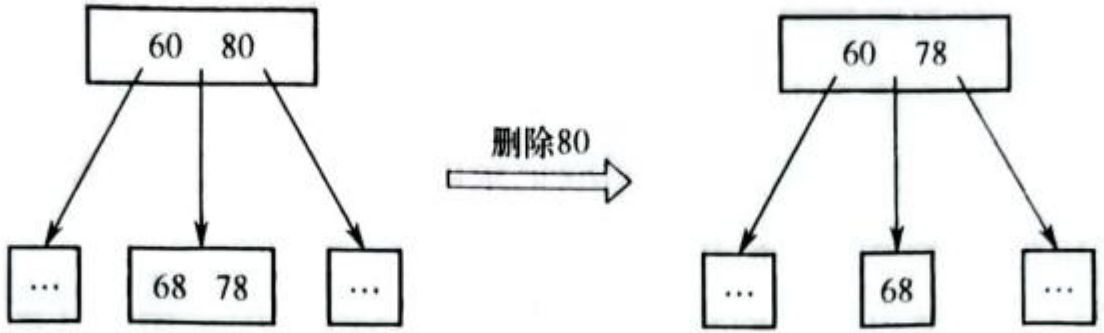
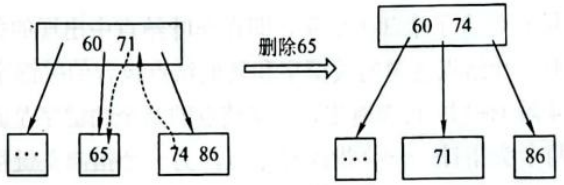


图 7.6 B 树中删除非终端结点关键字的取代

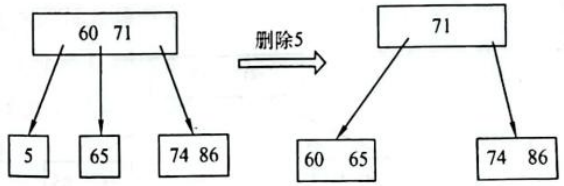
当被删关键字在终端结点（最低层非叶结点）中时，有下列三种情况：

1）直接删除关键字。若被删除关键字所在结点的关键字个数 $\geq\lceil m/2\rceil$，表明删除该关键字后仍满足 B 树的定义，则直接删去该关键字。

2）兄弟够借。若被删除关键字所在结点删除前的关键字个数 $=\lceil m/2\rceil-1$，且与此结点相邻的右（或左）兄弟结点的关键字个数 $\geq\lceil m/2\rceil$，则需要调整该结点、右（或左）兄弟结点及其双亲结点（父子换位法），以达到新的平衡。在图 7.7(a)中删除 4 阶 B 树的关键字 65，右兄弟关键字个数 $\geq\lceil m/2\rceil=2$，将 71 取代原 65 的位置，将 74 调整到 71 的位置。



(a) 兄弟够借



(b) 兄弟不够借

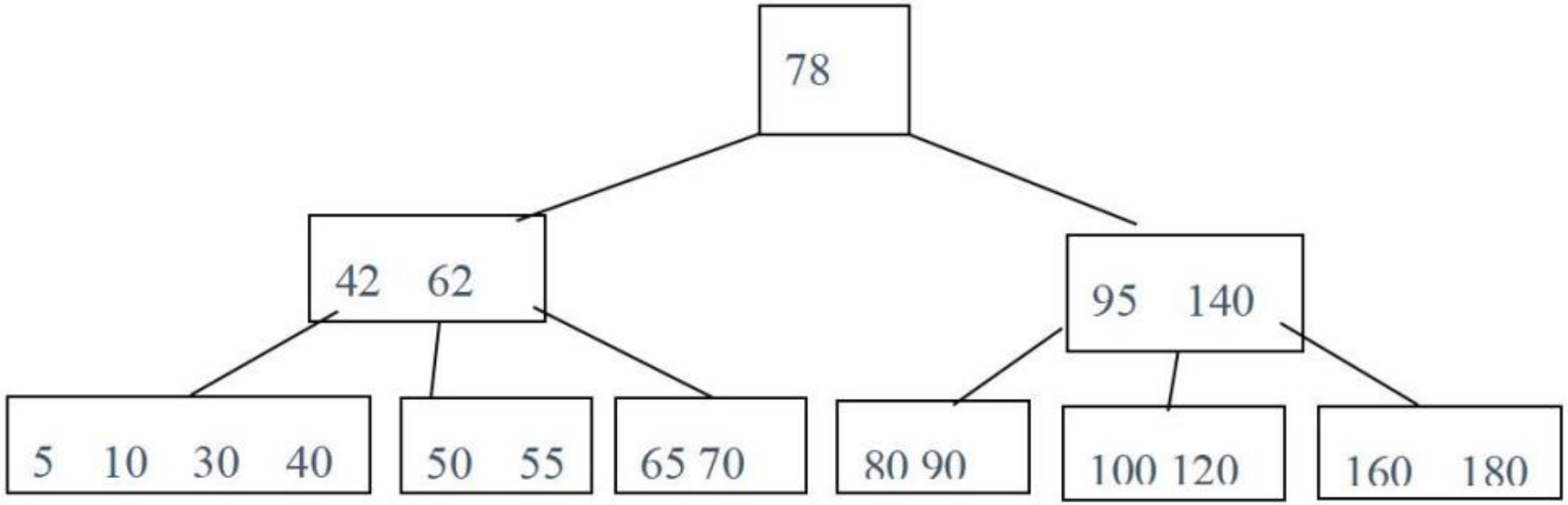
图 7.7　B 树中删除终端结点关键字的示意图

3）兄弟不够借。若被删除关键字所在结点删除前的关键字个数 $=\lceil m/2\rceil-1$，且此时与该结点相邻的左、右兄弟结点的关键字个数均 $=\lceil m/2\rceil-1$，则将关键字删除后与左（或右）兄弟结点及双亲结点中的关键字进行合并。在图 7.7(b)中删除 4 阶 B 树的关键字 5，它及其右兄弟结点的关键字个数=$\lceil m/2\rceil-1=1$，故在 5 删除后将 60 合并到 65 结点中。

在合并过程中，双亲结点中的关键字个数会减 1。若其双亲结点是根结点且关键字个数减少至 0（根结点关键字个数为 1 时，有 2 棵子树），则直接将根结点删除，合并后的新结点成为根；若双亲结点不是根结点，且关键字个数减少到 $\lceil m/2\rceil-2$，则又要与它自己的兄弟结点进行调整或合并操作，并重复上述步骤，直至符合 B 树的要求为止。

01 本周习题

4. 在下列 5 阶 B-树中首先插入关键字 85，然后删除关键字 70，画出插入元素和删除元素后的 B-树。



01 本周习题

6. 已知一棵二叉树按顺序存储结构进行存储，设计一个算法，求编号分别为 i 和 j 的两个节点的最近的公共祖先节点的值。叙述算法思想，给出用 c++语言实现的代码，必要时给出注释。最后分析算法执行的时空复杂度。

01 本周习题

首先，必须明确二叉树中任意两个结点必然存在最近的公共祖先结点，最坏的情况下是根结点（两个结点分别在根结点的左右分支中），而且从最近的公共祖先结点到根结点的全部祖先结点都是公共的。由二叉树顺序存储的性质可知，任一结点 i 的双亲结点的编号为 $i/2$。求解 i 和 j 最近公共祖先结点的算法步骤如下（设从数组下标 1 开始存储）：

1）若 $i>j$，则结点 i 所在层次大于等于结点 j 所在层次。结点 i 的双亲结点为结点 $i/2$，若 $i/2=j$，则结点 $i/2$ 是原结点 i 和结点 j 的最近公共祖先结点，若 $i/2\neq j$，则令 $i=i/2$，即以该结点 i 的双亲结点为起点，采用递归的方法继续查找。

2）若 $j>i$，则结点 j 所在层次大于等于结点 i 所在层次。结点 j 的双亲结点为结点 $j/2$，若 $j/2=i$，则结点 $j/2$ 是原结点 i 和结点 j 的最近公共祖先结点，若 $j/2\neq i$，则令 $j=j/2$。

重复上述过程，直到找到它们最近的公共祖先结点为止。

本题代码如下：

```
ElemType Comm_Ancestor(SqTree T,int i,int j){
//本算法在二叉树中查找结点 i 和结点 j 的最近公共祖先结点
    if(T[i]!='#'&&T[j]!='#'){          //结点存在
        while(i!=j){                   //两个编号不同时循环
            if(i>j)
                i=i/2;                 //向上找 i 的祖先
            else
                j=j/2;                 //向上找 j 的祖先
        }
        return  T[i];
    }
}
```

Q&A