数据结构与算法

Data Structures and Algorithms

第五部分 查找

回顾：查找--1

1. 查找的基本概念
2. 线性查找法
3. 折半/二分查找法
4. 分块查找
5. 二叉排序树
   - 基本概念
   - 查找算法
   - 插入新结点（建立）
   - 二叉排序树的结点删除- 难点
     - 删除叶子结点
     - 只有左/右子树非空
     - 左/右子树均非空

5.5 AVL树，又称平衡二叉树 (Balanced Binary Tree)

G.M.Adelson-Velsky和E.M.Landis,首次出现在1962年论文：
《An algorithm for the organization of information》

格奥尔吉·阿德尔森-韦利斯基，前苏联数学家，计算机科学家。1922年出生于俄罗斯萨马拉。1962年与叶夫吉尼·兰迪斯发明了AVL树。后移居以色列，现居住于阿什杜德。

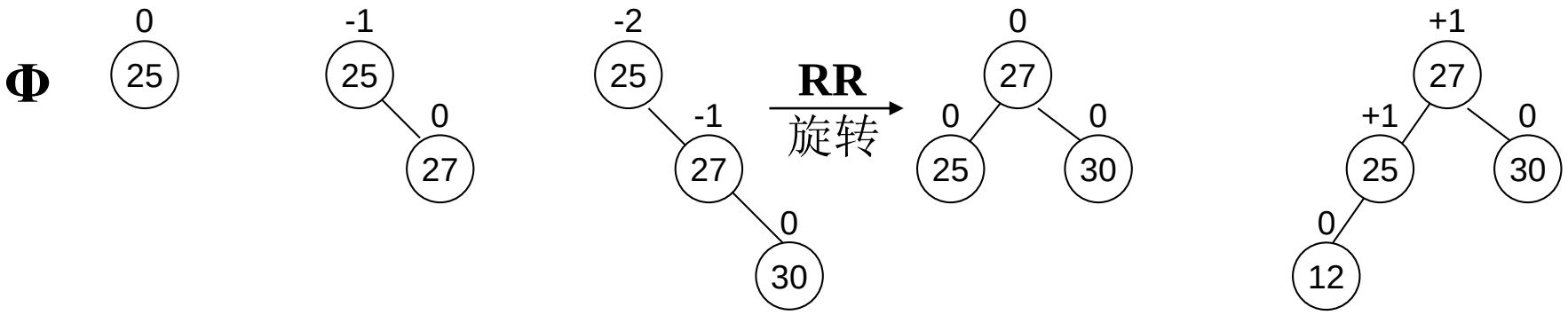
【定义】AVL树或者是一棵空二叉树，或者具有如下性质的二叉查找树：

其左子树和右子树都是高度平衡的二叉树，且左子树和右子树高度之差的绝对值不超过1。

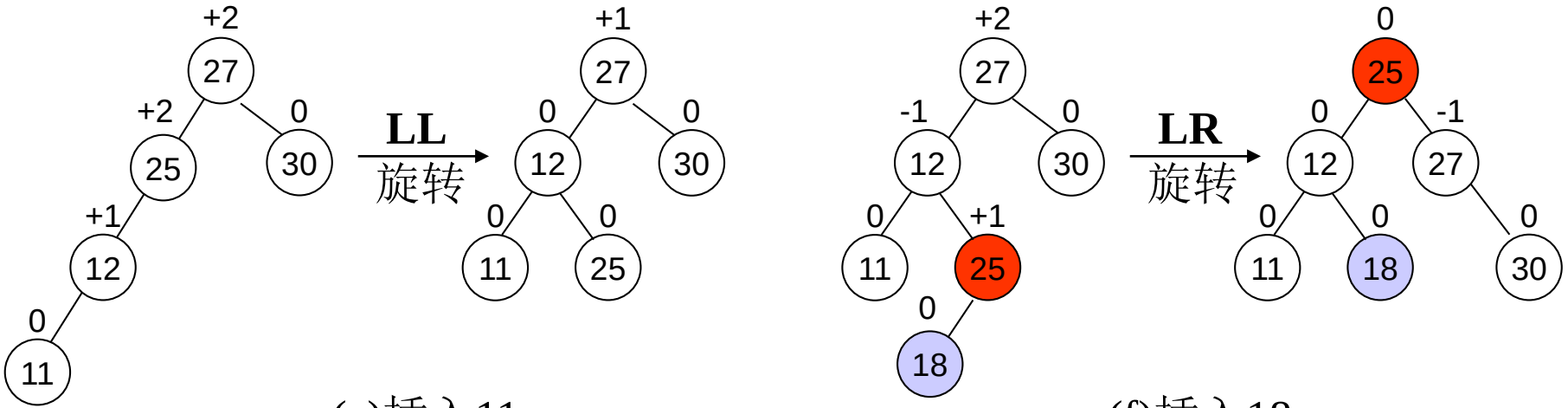
结点的平衡因子 BF (Banlanced Factor) 定义为：结点左子树与右子树的高度之差。

AVL中的任意结点的 BF 只可能是 -1，0，+1

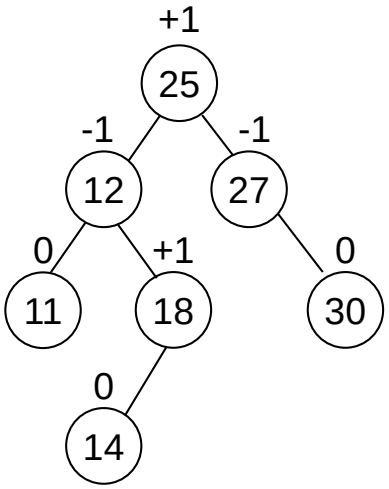
输入={25, 27, 30, 12, 11, 18, 14, 20, 15, 22}



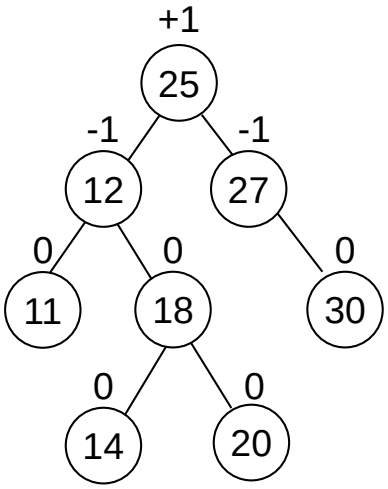
(a)插入25 (b)插入27 (c)插入30 (d)插入12



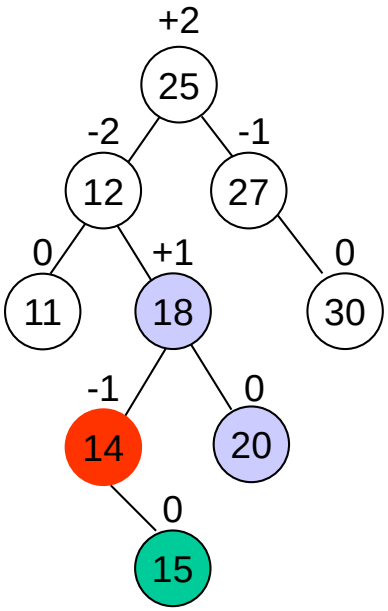
(e)插入11 (f)插入18



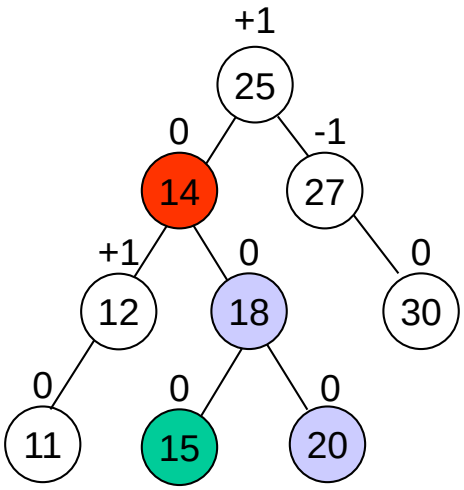
(g)插入14



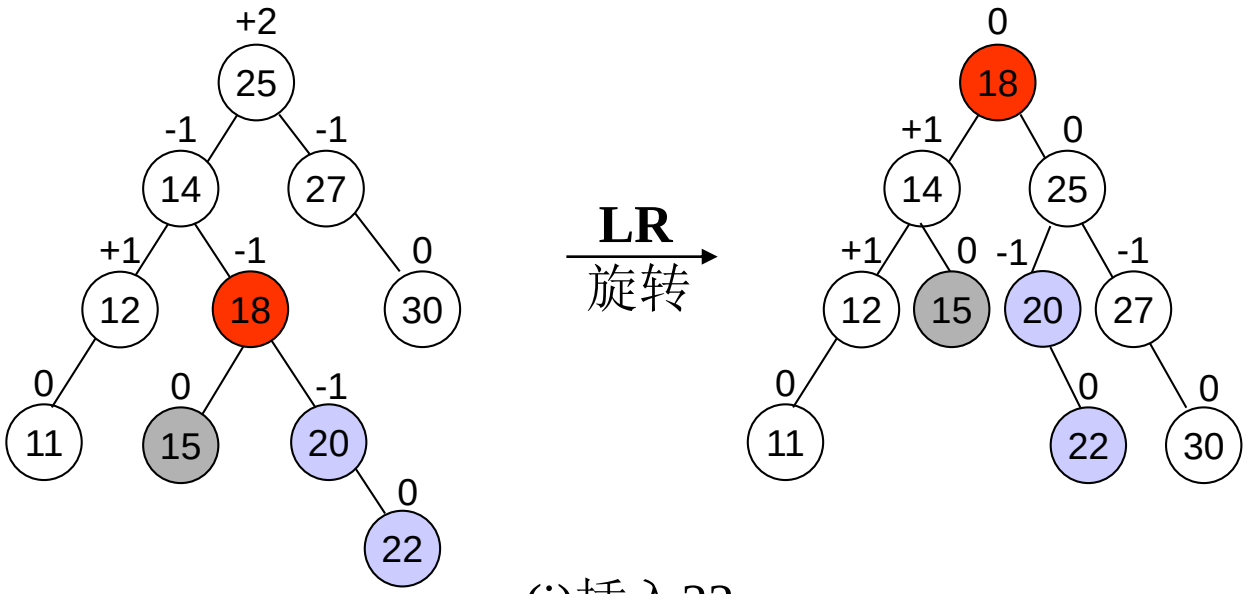
(h)插入20



RL
旋转



(i)插入15



(i)插入22

设：Y表示新插入的结点，

A表示离新插入结点Y最近的且平衡因子变为±2的祖先结点

对称

- LL：新结点Y被插入到A的左子树的左子树上；
- LR：新结点Y被插入到A的左子树的右子树上；
- RR：新结点Y被插入到A的右子树的右子树上；
- RL：新结点Y被插入到A的右子树的左子树上；

AVL树的平衡化处理

■在一棵AVL树上插入结点可能会破坏树的平衡性，需要平衡化处理恢复平衡，且保持BST的结构性质。

■若用Y表示新插入的结点，A表示离新插入结点Y最近的，且平衡因子变为±2的祖先结点。

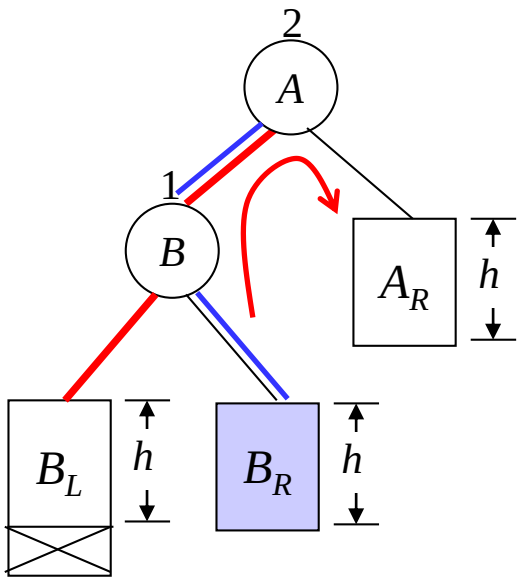
■可以用4种旋转进行平衡化处理：

①LL型：新结点Y 被插入到A 的左子树的左子树上（顺）

②RR型：新结点Y 被插入到A 的右子树的右子树上（逆）

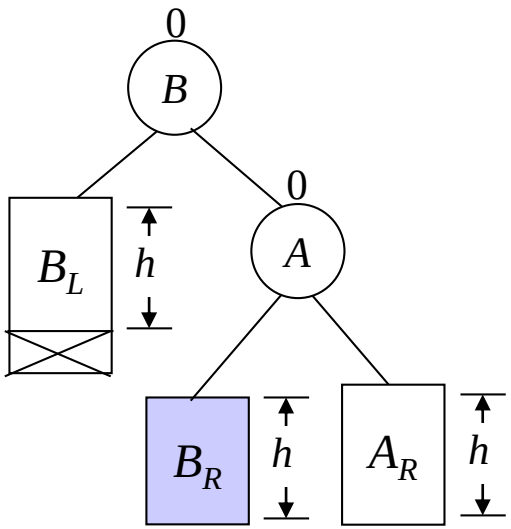
③LR型：新结点Y被插入到A的左子树的右子树上（逆、顺）

④RL型：新结点Y 被插入到A 的右子树的左子树上（顺、逆）



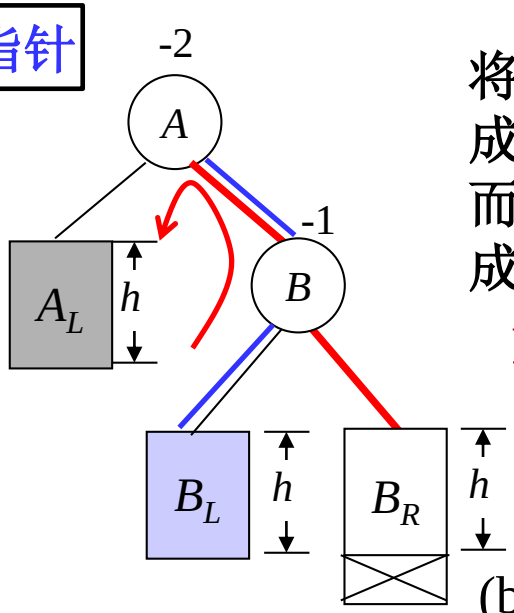
将A顺时针旋转，
成为B的右子树，
而原来B的右子树
成为A的左子树。

LL型（顺）



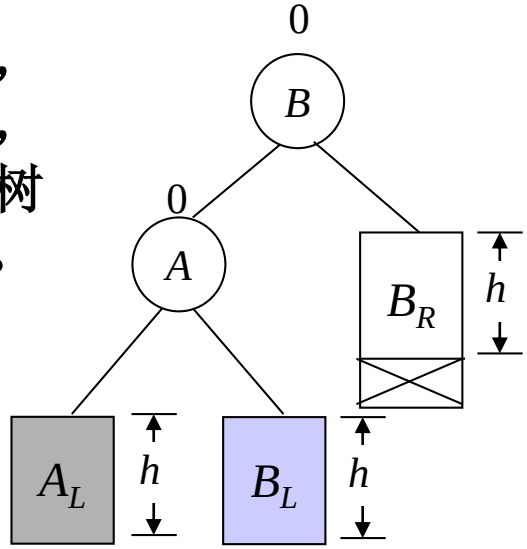
(a) LL型的旋转

修改指针



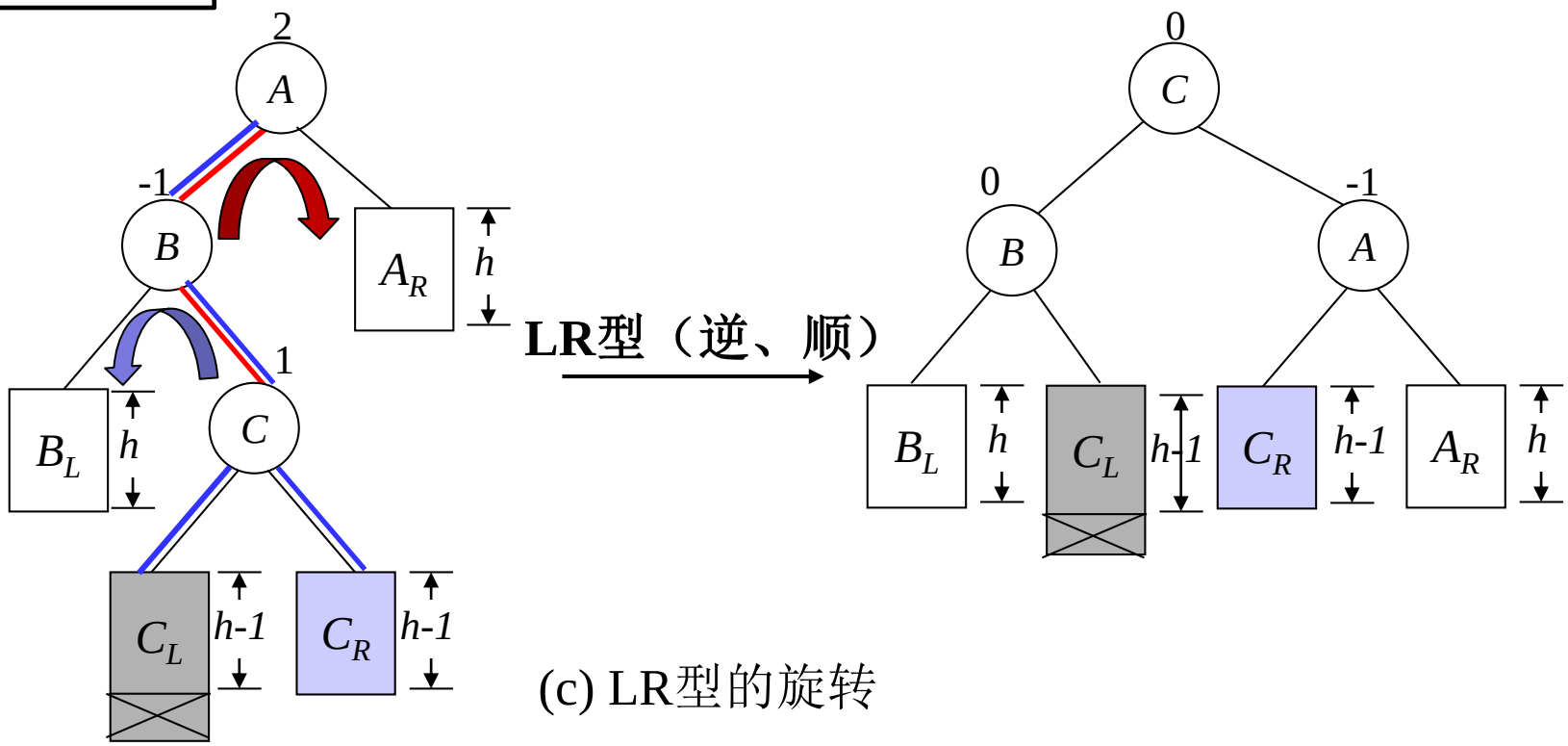
将A逆时针旋转，
成为B的左子树，
而原来B的左子树
成为A的右子树。

RR型（逆）



(b) RR型的旋转

修改指针

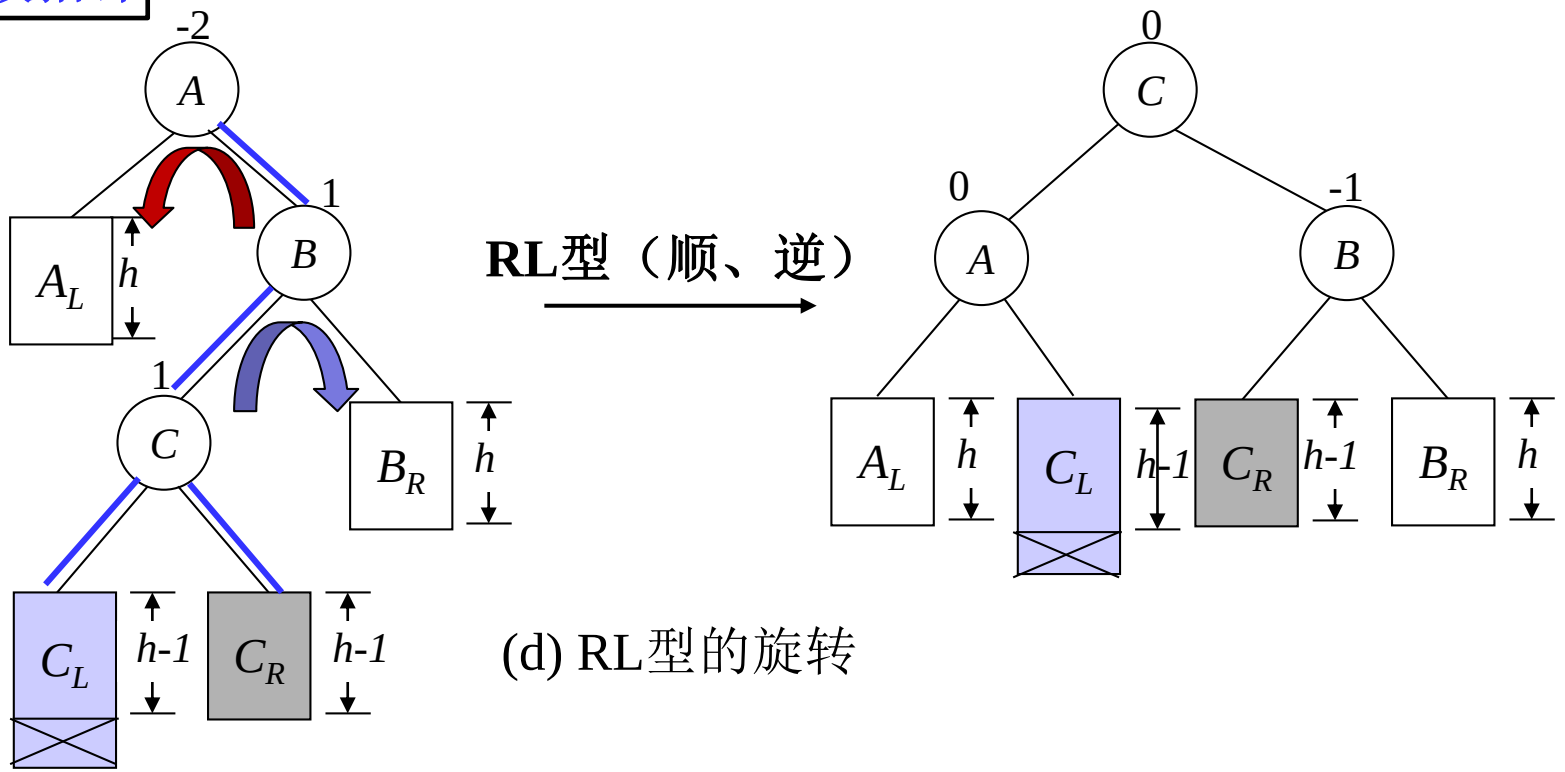


(c) LR型的旋转

（1）绕 *C*，将 *B* 逆时针旋转，C_L作为 B 的右子树；

（2）绕 *C*，将 *A* 顺时针旋转，C_R作为 A 的左子树。

修改指针



(d) RL型的旋转

（1）绕 *C*，将 *B* 顺时针旋转，C_R作为 B 的左子树；

（2）绕 *C*，将 *A* 逆时针旋转，C_L作为 A 的右子树。

AVL树的插入操作与建立

■对于一组关键字的输入，从空开始不断插入结点， 最后构成AVL树；

■每插入一个结点后，就应判断从该结点到根的路径上平衡因子有无结点发生不平衡；

■如有不平衡问题，利用旋转方法进行树的调整，使之平衡化；

■建AVL树过程是不断插入结点和必要时进行平衡化的过程。

AVL 树的结构类型

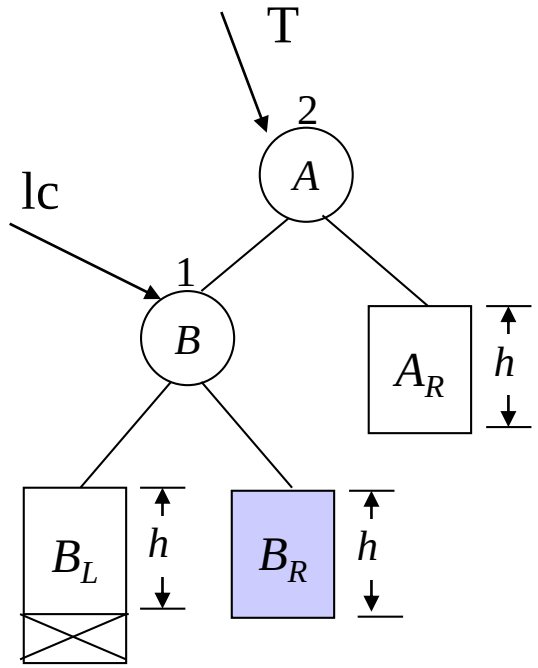
```
int unbalanced = FALSE ;
struct Node {
      ElementType  data ;
                   int   bf ;
 struct Node *lchild, *rchild ;
}
Typedef   Node  *AVLT ;
```

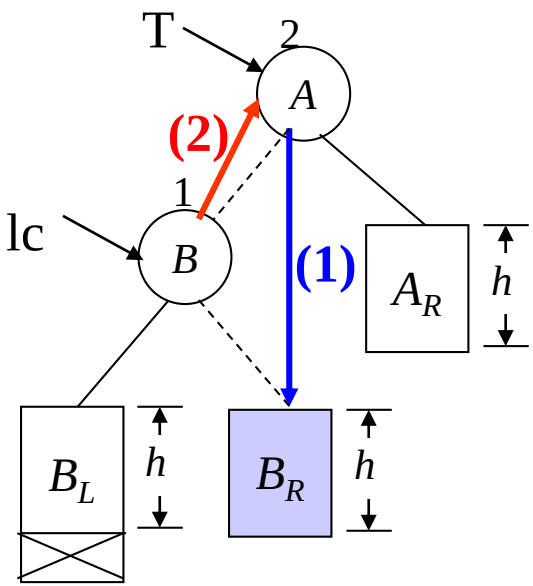
```
unbalanced = TRUE;

void AVLInsert ( AVLT  &T , ElementType R , int   &unbalanced )
{   if(  !T  )   //向空二叉树中插入元素
    {   unbalanced = TRUE ;
        T = new Node ;
        T->data = R ;
        T->lchild = T->rchild = NULL ;
        T->bf  = 0;
     }
     else if ( R.key  <  T->data.key )        //在左子树上插入
     {   AVLInser( T->lchild ,  R,  unbalanced ) ;
         if ( unbalanced )
           switch ( T->bf  )  {
                case -1:  T->bf = 0 ;
                          unbalanced = FALSE ;
                          break ;
                 case 0:   T->bf = 1 ;
                          break ;
                 case 1:   LeftRotation ( T, unbalanced ) ;   }
}
```

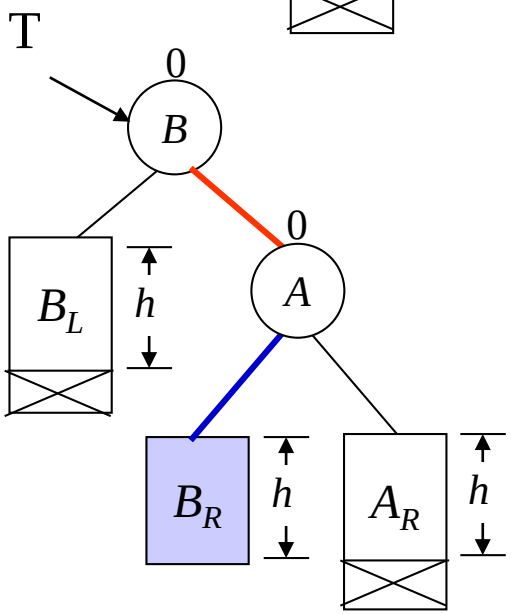
```
else if ( R.key  > =T->data.key )    //在右子树上插入
   {  AVLInsert ( T->rchild , R,  unbalanced ) ;
      if ( unbalanced )
         switch ( T->bf  ) {
            case   1:  T->bf  =  0 ;
                       unbalanced  =  FALSE ;
                       break ;
             case 0:   T->bf  =  -1 ;
                        break ;
            case -1:   RightRotation ( T,  unbalanced  ) ;
         }
      }
else
   unbalanced  =  FALSE  ;
}  //AVLInsert
```

```
void  LeftRotation ( AVL  &T ,  int  &unbalanced )
{
    AVLT  gc , lc ;
    lc  =  T->lchild ;
    if (  lc->bf  ==  1 )    //LL旋转
      {
          T->lchild  =  lc->rchild ;
          lc->rchild  =  T ;
          T->bf  =  0 ;
          T = lc
      }
     else                          //LR旋转
      {
          gc = lc->rchild ;
          lc->rchild  = gc->lchild ;
          gc->lchild  = lc ;
          T->lchild   = gc->rchild ;
          gc->rchild  = T ;
          switch ( gc->bf )  {
              case 1: T->bf   = -1 ;
                      lc->bf  = 0 ;
                      break ;
              case 0: T->bf  = lc->bf  = 0 ;
                      break ;
              case -1:T->bf  = 0 ;
                      lc->bf = 1 ;
            }
            T = gc ;
        }
      T->bf  = 0 ;
      unbalanced  = FALSE ;
}
```

```
lc = T->lchild ;
if ( lc->bf == 1 )   //LL旋转
  {   T->lchild = lc->rchild ;// (1)
      lc->rchild = T ;// (2)
      T->bf = 0 ;
      T = lc;
  }
…
```

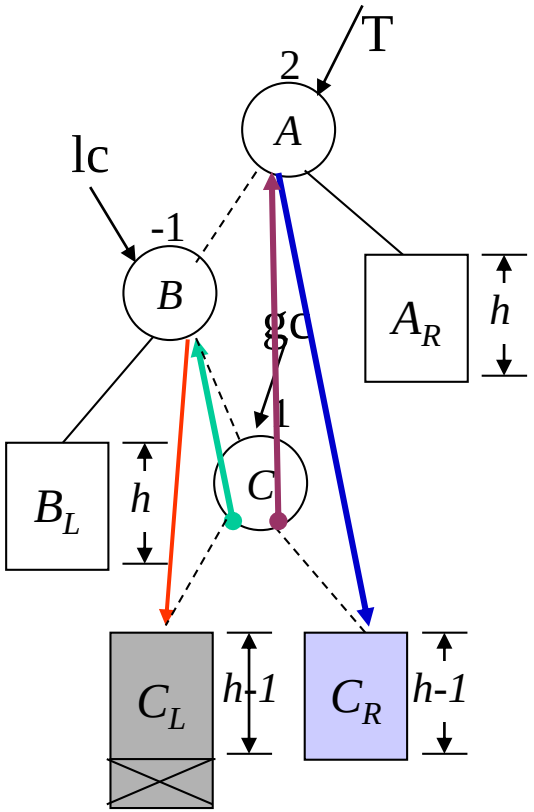




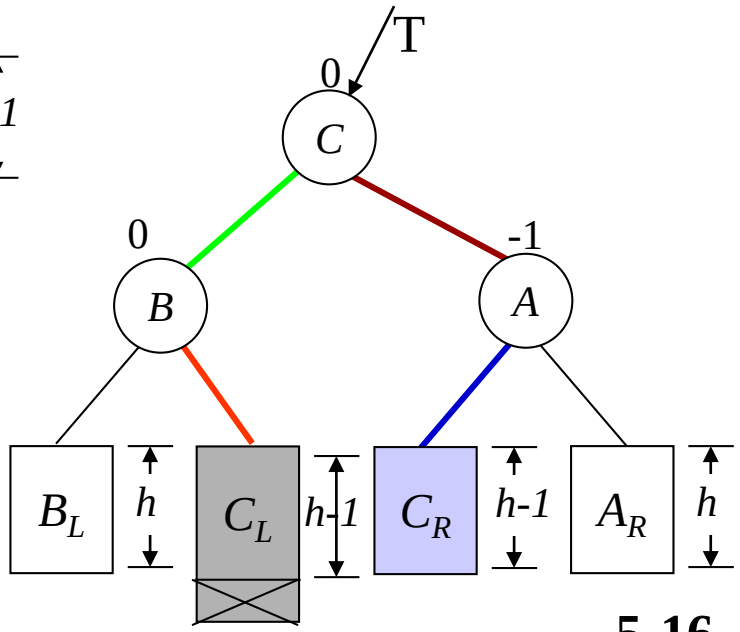


(a) LL型的旋转

```
lc  =  T->lchild ;
……
else    //LR旋转
{   gc = lc->rchild ;
    lc->rchild  = gc->lchild ;
    gc->lchild  = lc ;
    T->lchild   = gc->rchild ;
    gc->rchild  = T ;
……
T = gc ;
```



(c) LR型的旋转



```
void RightRotation(AVLT *&T)
{
   AVLT *gc,*rc ;
   rc=T->rchild ;
   if(rc->bf==-1)                    //RR旋转
   {
      T->rchild=rc->lchild ;
      rc->lchild=T ;
      T->bf=0;
      T=rc;
   }
   else                              //RL旋转
   {
      gc=rc->lchild ;
      rc->lchild=gc->rchild ;
      gc->rchild=rc ;
      T->rchild=gc->lchild ;
      gc->lchild=T;
```

```
   switch(gc->bf){     //调整平衡因子
      case -1: T->bf=-1;
               rc->bf=0 ;
               break ;
      case  0: T->bf=rc->bf=0 ;
               break ;
      case  1: T->bf =0;
               rc->bf=-1;
      }
      T=gc ;
   }
   T->bf=0 ;
   unbalanced=FALSE ;
}
```

分析：

- 高度为h的AVL树最多具有$N_{hmax}=2^h-1$个结点(二叉树性质一)
- 高度为h的AVL树最少具有多少个结点？

$N_0=1$

$N_1=1$

$N_2=2$

……

$N_h=N_{h-1}+N_{h-2}+1$

$F_i=F_{i-1}+F_{i-2}$ (Fibonacci polynomial)

$$N_{hmin}=\log_{\Phi}(\sqrt{5}*(N+1))-2$$

? $h \le \log_{\phi}(\sqrt{5}(n+1))-2$

T(n)=O(logn)

AVL树的删除操作

■删除操作与插入操作是对称的（镜像），但可能需要的平衡化次数多。

■因为平衡化不会增加子树的高度，但可能会减少子树的高。

■在有可能使树增高的插入操作中，一次平衡化能抵消掉树增高；

■而在有可能使树减低的删除操作中，平衡化可能会带来祖先结点的不平衡。

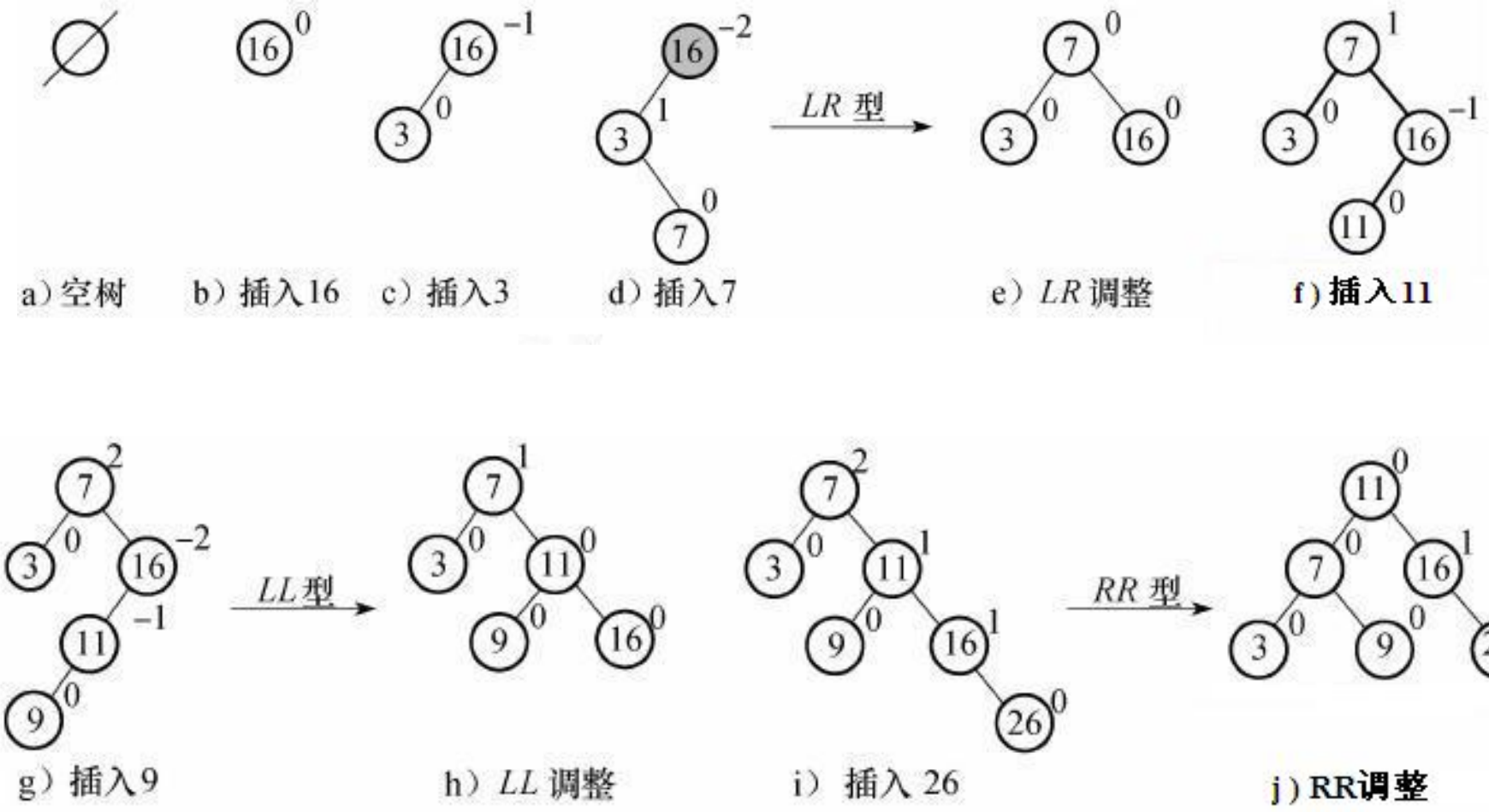
【例5-2】下述二叉树中,哪一种满足性质:从任一结点出发到根的路径上所经过的结点序列按其关键字有序。

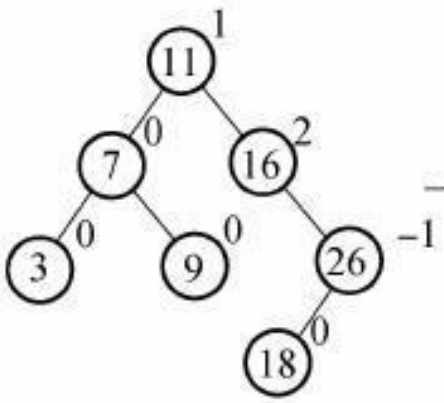
(A) 二叉排序树 (B) 赫夫曼树 (C) AVL树 (D) 堆

【分析】

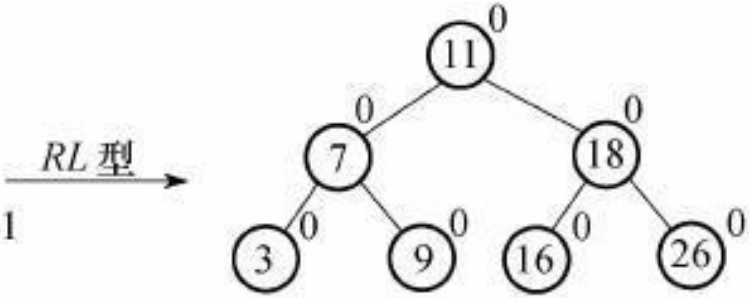
- 对于选项A，根据二叉排序树的结构特点我们可以知道，二叉排序树的中序遍历结果是一个有序序列，而在中序遍历中，父结点并不总是出现在孩子结点的前面（或后面），故该选项不正确。
- 对于选项B，根据赫夫曼树的结构特点我们可以知道，在赫夫曼树中所有的关键字只出现在叶结点上，其非叶结点上并没有关键字值，显然不正确。
- 对于选项C，AVL树其本质上也是一种二叉排序树，只不过是平衡化之后的二叉排序树，故该选项也是不正确的。
- 对于选项D，堆的概念我们会在堆排序中给大家介绍，根据建堆的过程，不断地把大者“上浮”，将小者“筛选”下去，最终得到的正是一个从任一结点出发到根的路径上所经过的结点序列按其关键字有序的树状结构。

【例5-3】输入关键码序列为(16, 3, 7, 11, 9, 26, 18, 14, 15)，据此建立平衡二叉树，给出插入和调整的具体过程。

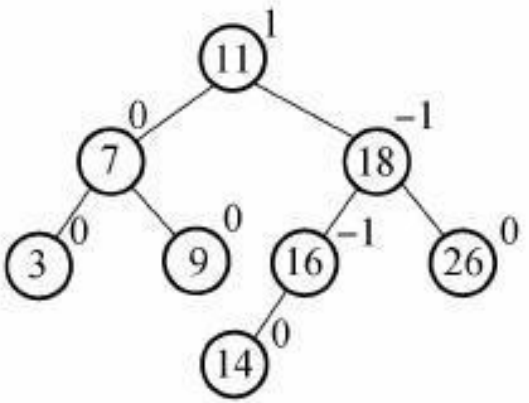




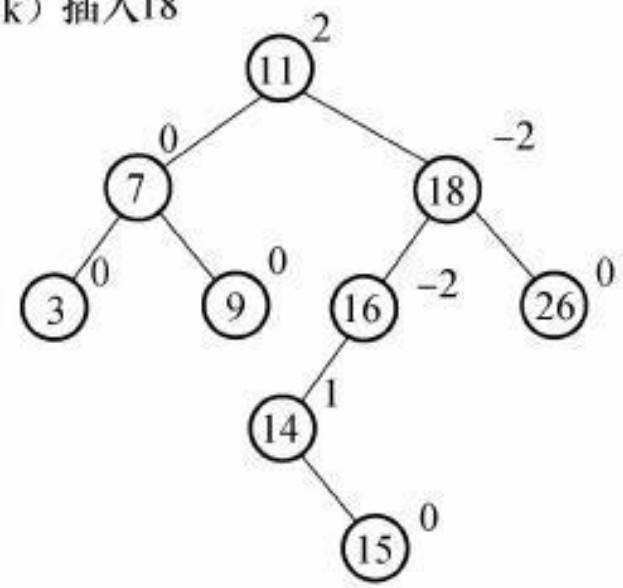
k）插入18



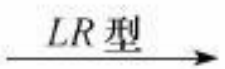
l） *RL* 调整

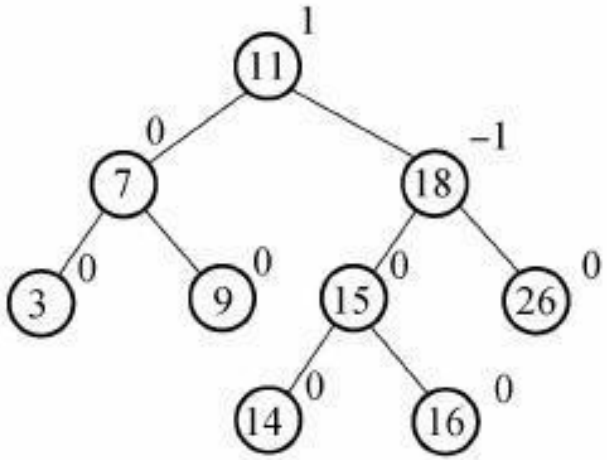


m）插入14



n）插入 15

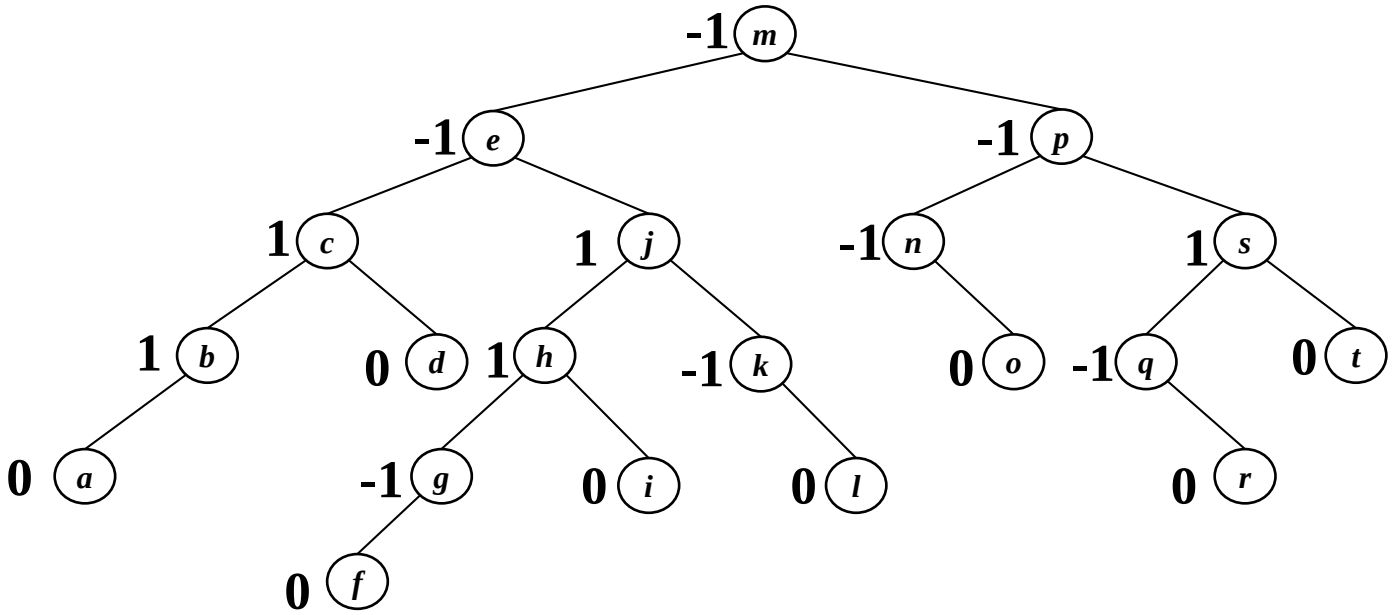


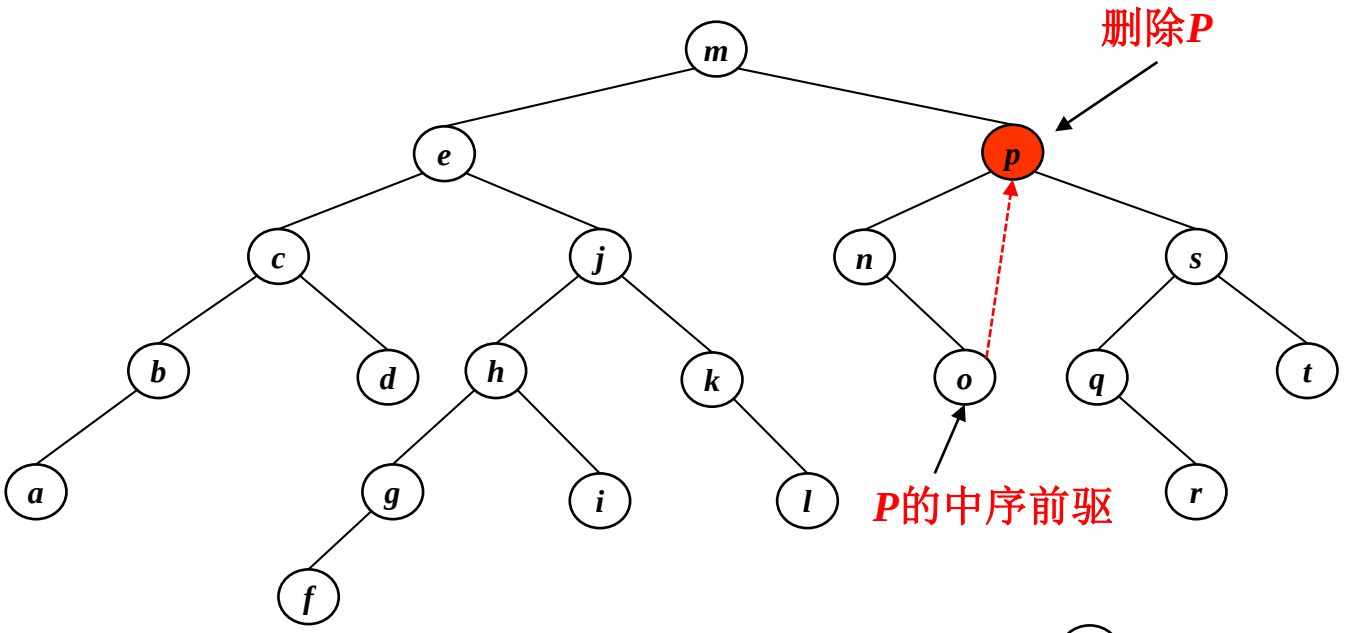


o） *LR* 调整

【例5-4】有一棵平衡二叉树的初始状态如图所示，

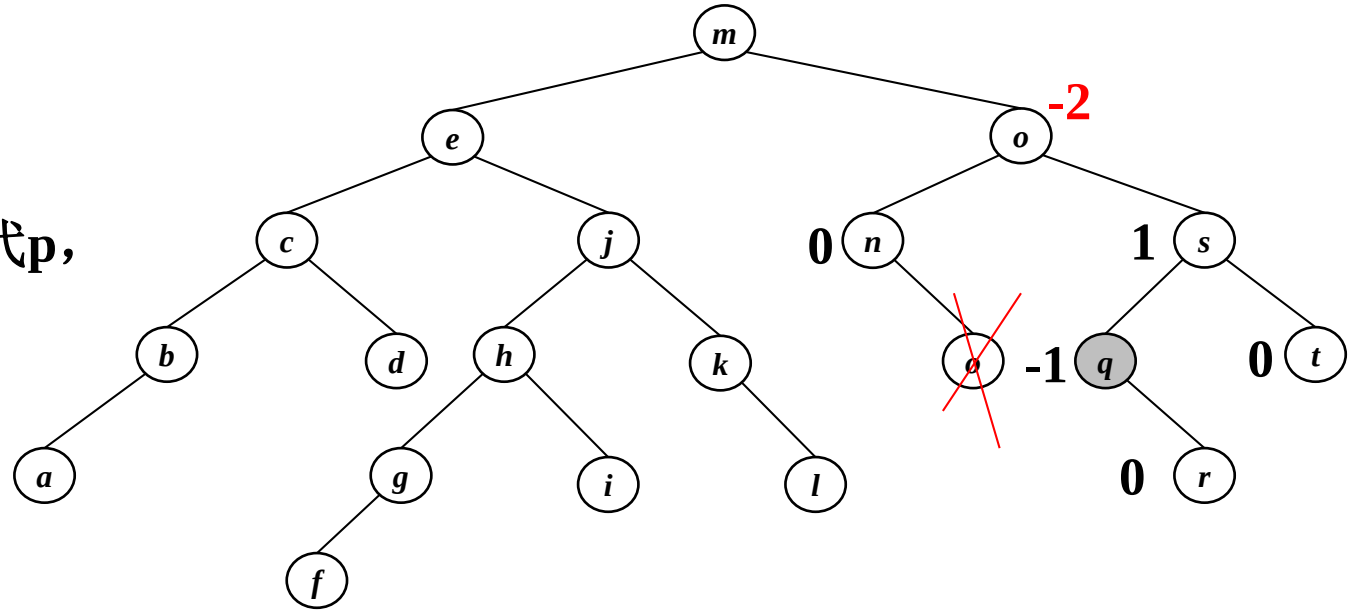
请给出删除图中结点p后经调整得到的新的平衡二叉树。

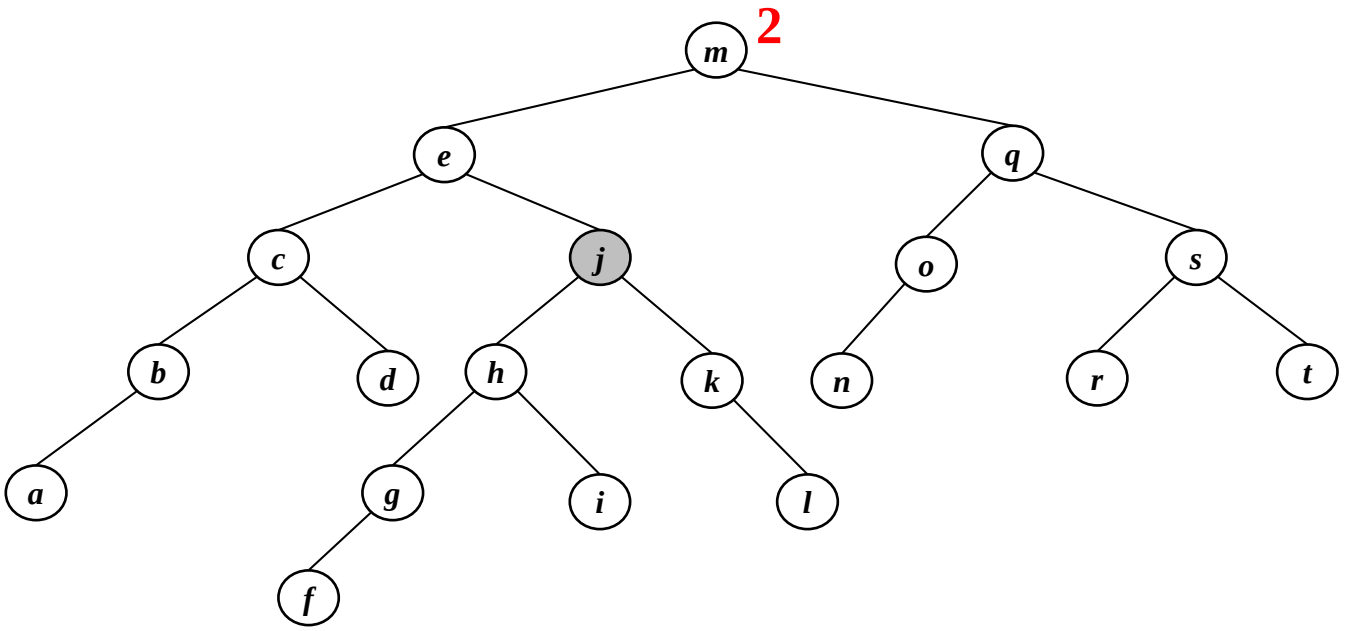




由于结点p既有左孩子，又有右孩子，故在删除结点p的时候应该先找到中序遍历中结点p的直接前驱结点，即结点o。

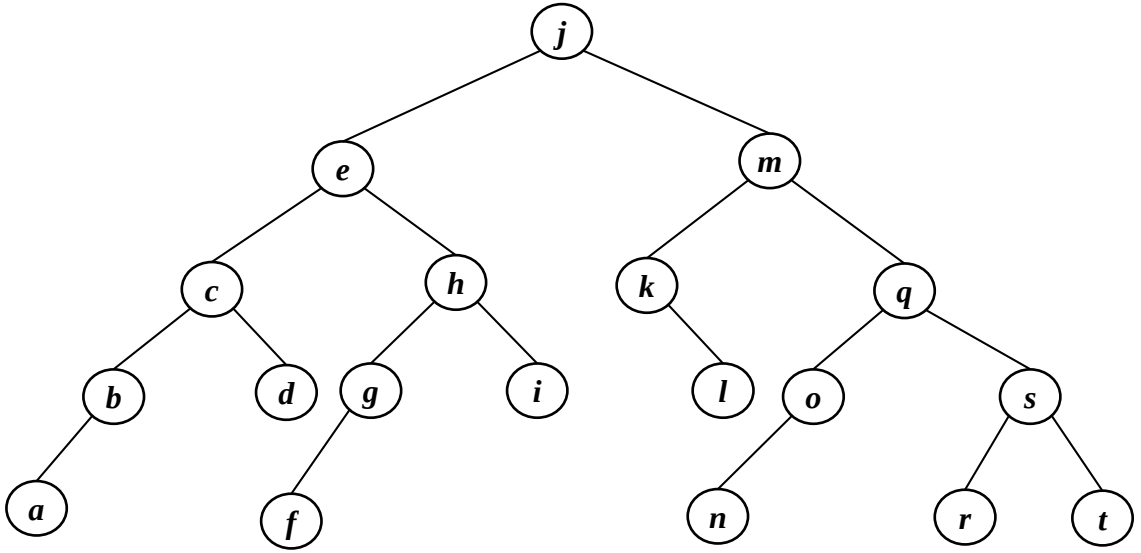
然后用o取代p，删除o，



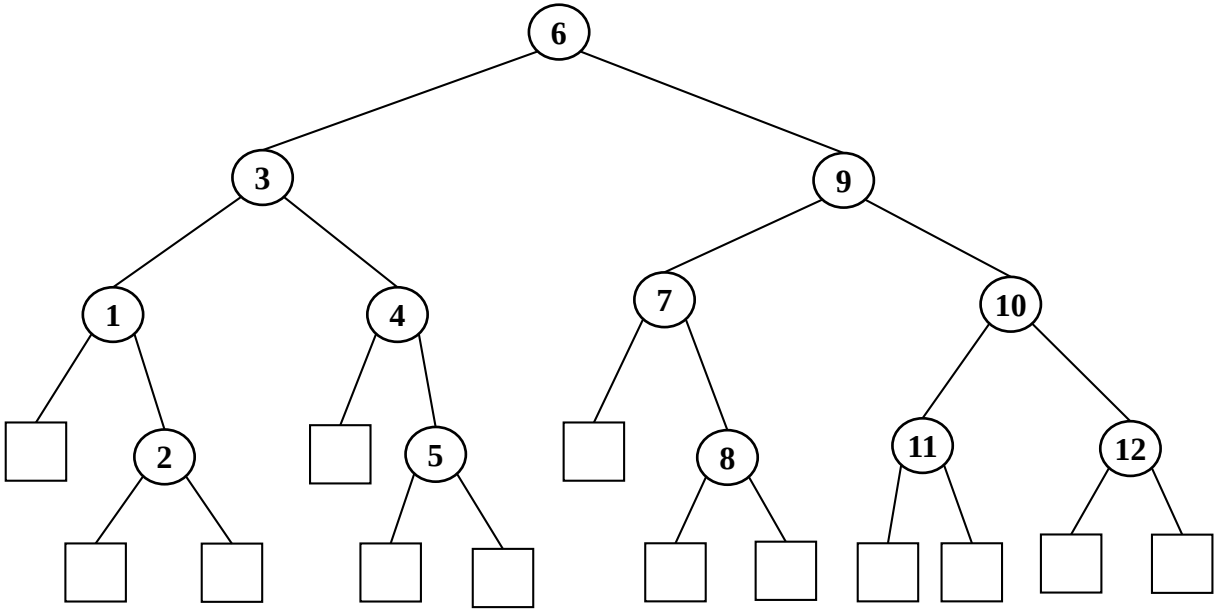


此时结点o的平衡度变为2，发生不平衡，故应对子树o,r,t进行RL型调整。

结点m的平衡度变为-2，发生不平衡，故应对子树m,e,j进行LR调整。

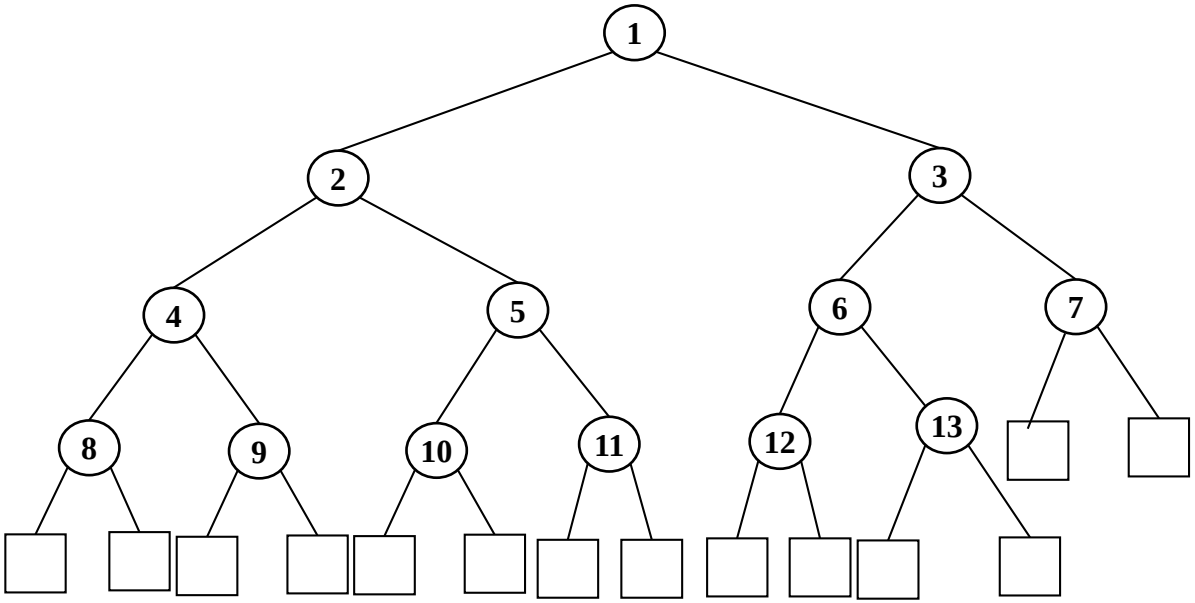


例：长度为12的有序表，按折半查找法对该表进行查找，以等概率查找表内各个元素，则查找成功时所需要的比较次数是多少？



$ASL_{成功}$ =（1*1+2*2+4*3+5*4）/ 12 = 37/12

$ASL_{失败}$ =（3*3+10*4）/ 13 = 62/13



$ASL_{成功}$ =（1*1+2*2+4*3+6*4）/ 13 = 41/13

$ASL_{失败}$ =（2*3+12*4）/ 14 = 54/14

【例5-5】判断任意二叉树是否是平衡二叉树。

```
int IsAVL(AVLT *T)   //判断平衡二叉树
{    int hl,hr;
     if(T==NULL)
            return(1);
     else
    {
        hl=Depth(T->lchild);
        hr=Depth(T->rchild);
        if(abs(hl-hr)<=1)
       {
          if(IsAVL(T->lchild))
            return(IsAVL(T->rchild));
       }
       else
         return(0);
    }
}
```

```
int Depth(AVLT *bt)  //求二叉树的深度
{   int ldepth,rdepth;
    if(bt==NULL)
        return(0);
    else
    {   ldepth=Depth(bt->lchild);
        rdepth=Depth(bt->rchild);
        if(ldepth>rdepth)
            return(ldepth+1);
        else
            return(rdepth+1);
    }
}
```

【定义】一颗m-路查找树或者为空，或者满足以下性质：

(1) 根结点至多包含m棵子树，且具有以下结构：

$$n, A_0, (K_1,A_1), (K_2, A_2), \ldots\ldots, (K_n, A_n)$$

其中，A_i是指向子树的指针，$0 \le i \le n < m$；K_i是关键字值，$0 \le i \le n < m$；n是关键字的个数。

(2) $K_i < K_{i+1}$，$1 \le i < n$；

(3) 子树A_i中的所有关键字值都小于K_{i+1}，而大于K_i, $1 < i < n$；

(4) 子树A_n中的所有关键字值都大于K_n，子树A_0中的所有关键字值都小于K_1；

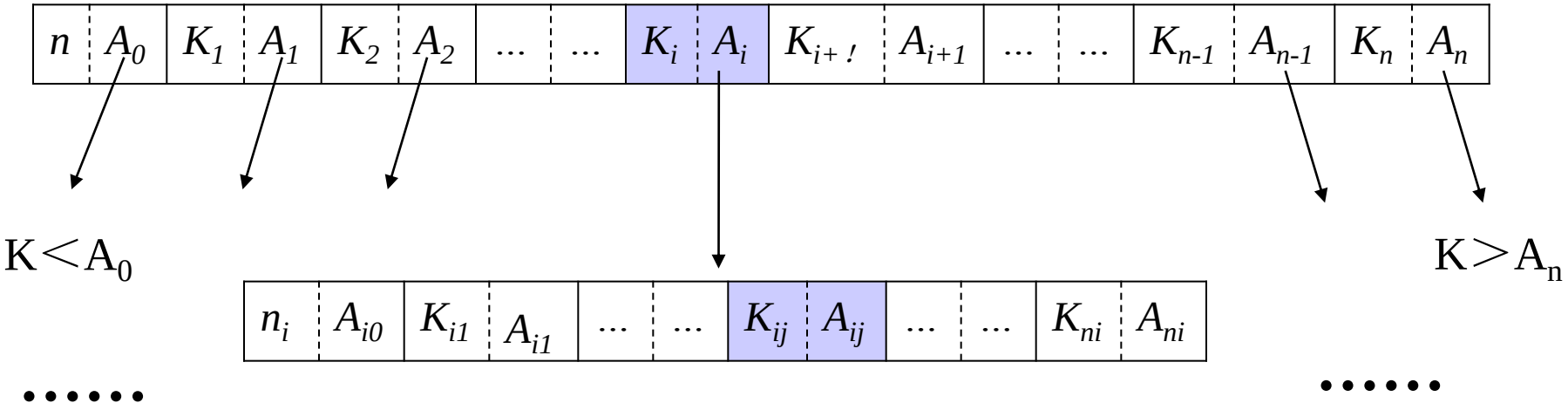
(5) 每棵子树A_i都是m-路查找树，$0 \le i \le n$。

设 m -路查找树高度为h，其结点数量的最大值是：

$$\sum_{i=0}^{h-1} m^i = \frac{m^h-1}{m-1}$$

由于每个结点最多包含$m-1$个关键字，因此在一棵高度为h的m-路查找树中，最多可容纳m^h-1个关键字。

【例5-6】$h=3$的二叉树，最多容纳7个关键字；而对于$h=3$的200-路查找树，最多有$m^h-1=8*10^6-1$个关键字。



$$K_i < (K_{i1}, K_{i2}, \ldots\ldots K_{ij}, K_{ni}) < K_{i+1}$$

5.6 B- 树 B-Tree Balanced-Tree

B-树：B-树是一种非二叉的查找树

除了要满足查找树的特性，还要满足以下结构特性。

一棵 ***m*** 阶的 **B- 树**：

（1）树的根或者是一片叶子(一个结点的树), 或者其儿子数在 2 和 ***m*** 之间；

（2）除根外，所有的非叶子结点的孩子数在 ***m/2*** 和 ***m*** 之间；

（3）所有的叶子结点都在相同的深度，且不带信息。

Ceil(m/2)

B-树的平均深度为$\log_{m/2}N$。执行查找的平均时间为O(*logm*)；

B-树应用在数据库系统中的索引，它加快了访问数据的速度；

【B-树定义】

一颗*m* 阶B-树是一颗m-路查找树，它或为空，或满足以下性质：

(1) 根结点或为叶子结点，或至少包含2个儿子；

(2) 除根结点和失败结点以外，所有结点都至少具有⌈m/2⌉个儿子；

(3) 所有失败结点都在同一层。

~~➢所有2阶B-树都是满二叉树，只有当关键字个数等于2^k-1时，才存在2阶B-树~~

➢对于任意给定的关键字和任意的*m(>2)*,一定存在一棵包含上述所有关键字的*m*阶B-树。

1、B-树中的关键字值个数

所有失败结点都在 *l+1* 层的 *m* 阶B-树至多包含 m^l-1 个关键字。

B-树的第 *1* 层至少包含 *2* 个儿子

B-树的第 *2* 层至少包含 *2* 个结点

B-树的第 *3* 层至少包含 *2*(m/2)* 个结点

B-树的第 *4* 层至少包含 $2*(m/2)^2$ 个结点

…

B-树的第 *l (>1)* 层至少包含 $2*(m/2)^{l-2}$ 个结点

} 每个结点又至少包含*m/2*个儿子

注：上述所有结点都不是失败结点

B树的高度

若B-树中的关键字分别为$K_1, K_2, \ldots, K_N$，且有 $K_i < K_{i+1}$，$(1 \le i < N)$

则失败结点的个数为 $N+1$ 。

因为当 x 满足 $K_i<x<K_{i+1}$，就会查找失败，其中：$K_0=-\infty$，$K_{N+1}=+\infty$

使得查找不在B-树中的关键字 x 时，就可能到达 $N+1$ 个不同的失败结点。

因此：

$N+1$=失败结点个数=第 $l+1$ 层的结点个数 $\ge 2*(m/2)^{l-1}$

故有$N+1 \ge 2*(m/2)^{l-1}$，$l \ge 1$.

反之，如果一棵m阶B-树包含N个关键字，则所有非失败结点所在的层号都小于或等于l，$l \le \log_{m/2}|(N+1)/2|+1$ $\log_m(N+1) \le l \le \log_{\lceil m/2 \rceil}|(N+1)/2|+1$

即每次查找所需进行至多 $l+1$ 次的结点访问。

【例5-7】$m=200$的B-树，当$N \le 2 \times 10^6-2$时，有$l \le \log_{100}|(N+1)/2|+1$

由于 l 为整数，所以 $l \le 3$。而当$N \le 2 \times 10^8-2$时，有 $l \le 4$。

因此，采用高阶的B-树可以用很少的磁盘访问次数来查找大规模的索引。

2、B-树阶的选择

在大多数系统中，B-树上的算法执行时间主要由读、写磁盘的次数来决定，每次读写尽可能多的信息可提高算法得执行速度。

B-树中的结点的规模一般是一个磁盘页，而结点中所包含的关键字及其孩子的数目取决于磁盘页的大小。

高阶B-树会降低在查找索引时所需的磁盘访问次数
——人们期望使用高阶B-树。

如果索引中包含 *N* 个元素，则 *m=N+1* 阶B-树就只有一层
——但 *m* 的选择受到内存可用空间的限制。

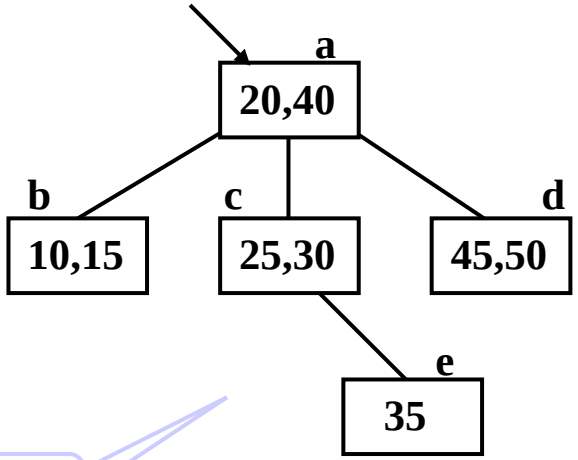
选取较大的结点度数可降低树的高度，以及减少查找任意关键字所需的磁盘访问次数；

如果 *m* 的值很大，使得索引不能一次性地全部读入内存，就会增加读盘次数，也会增加结点内的查找时间，*m* 的选择不合理；

合理地选择 *m* 的目标可使得在B-树中查找关键字 *x* 所需的总时间最小！

B-树上查找 *x* 的时间 { 从磁盘上读入结点的时间
在结点中查找 x 所需要的时间

【例5-8】 3 分支(路)查找树，元素为（10,15,20,25,30,35,40,45,50）。



非B-树

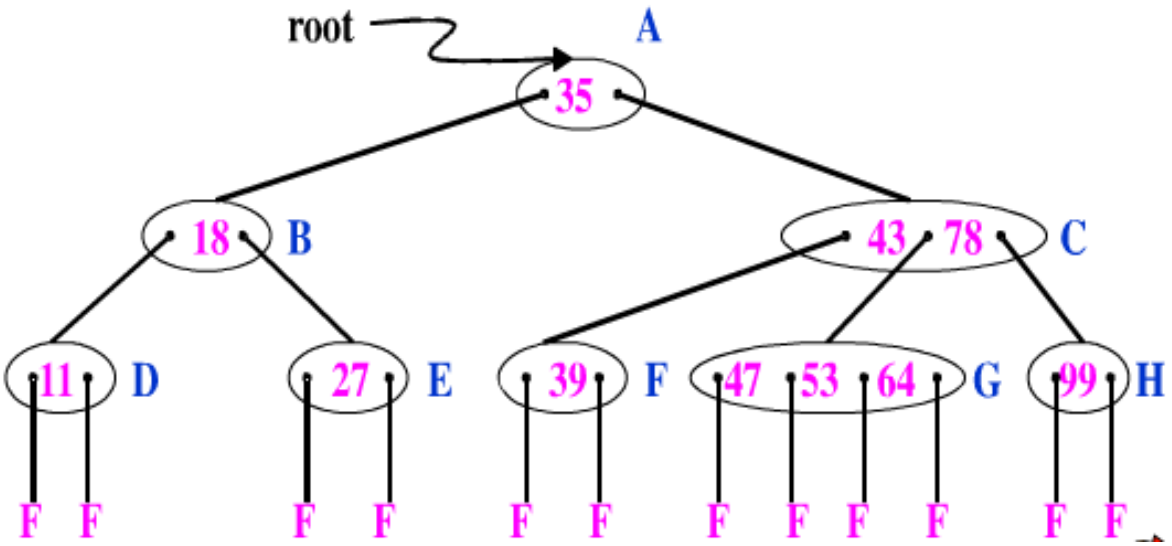
结点	n	A_0	(K_1,A_1)	(K_2,A_2)
a	2	b	(20,C)	(40,d)
b	2	^	(10, ^)	(15, ^)
c	2	^	(25, ^)	(30,e)
d	2	^	(45, ^)	(50, ^)
e	1	^	(35, ^)	

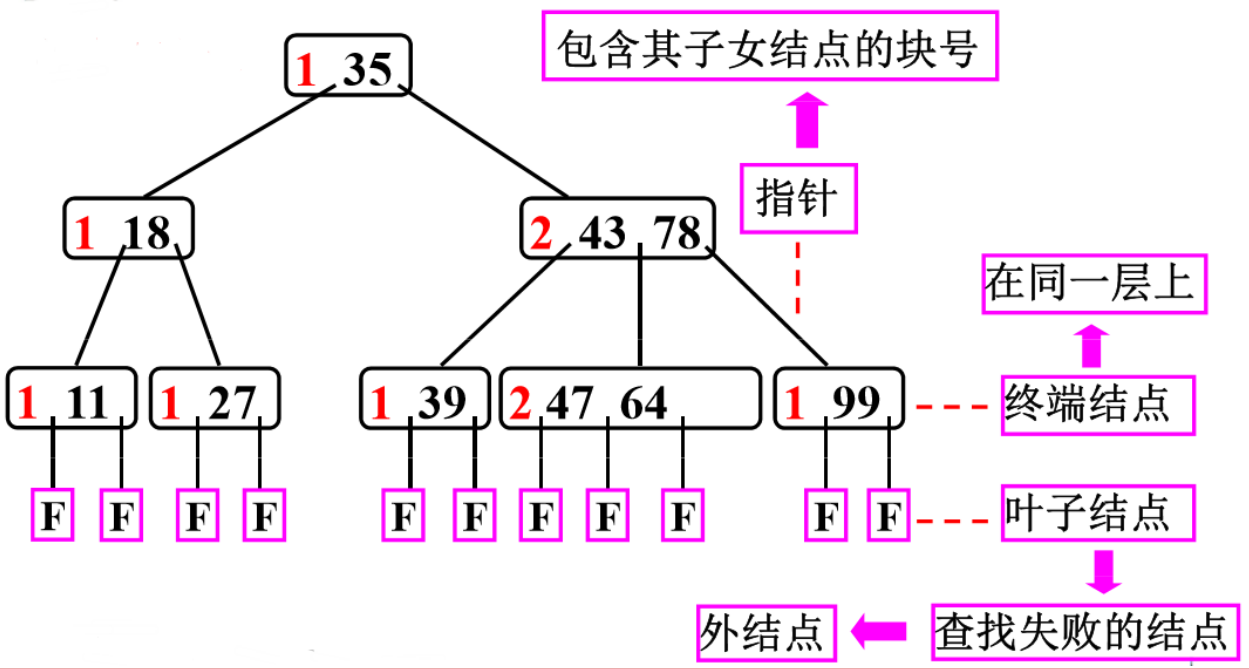
【重申】 m 阶的 B- 树 T 是这样一棵 m 分支查找树：T 或者为空；或者其高度不小于 1 且满足下述性质：

（1） 根结点至少有两个儿子；

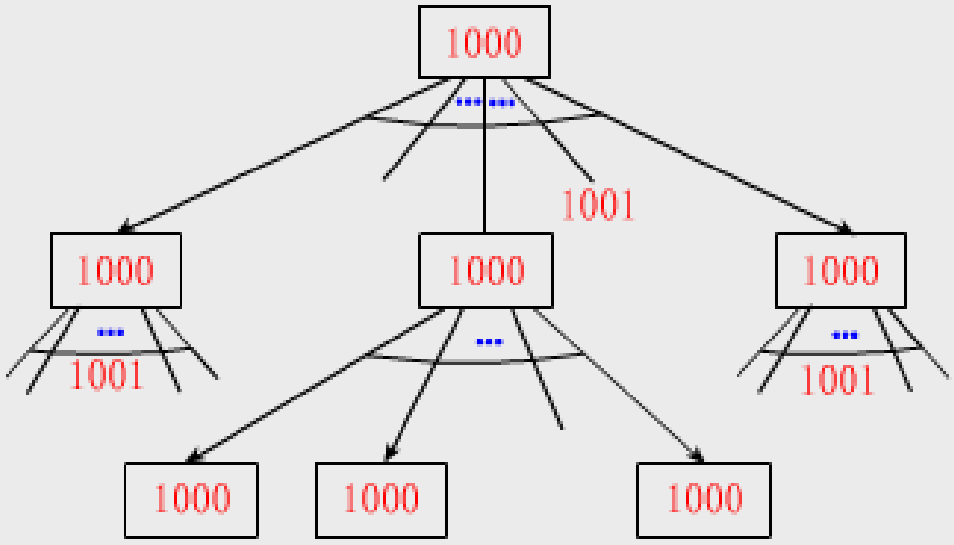
（2） 除根和叶子结点外的所有结点，至少有 m/2 个儿子；

（3） 所有的叶子结点都在同一层上。


root
A
35
18
B
43
78
C
11
D
27
E
39
F
47
53
64
G
99
H
F


1 35
1 18
2 43 78
1 11
1 27
1 39
2 47 64
1 99
F
包含其子女结点的块号
指针
在同一层上
终端结点
叶子结点
查找失败的结点
外结点

【例5-9】一棵高度为3的1001阶B-树。



一棵高度为3的1001阶B-树示例

【说明】

①每个结点包含1000个关键字，故在第三层上有100多万个叶结点，这些叶节点可容纳10亿多个关键字；

②图中各结点内的数字表示关键字的数目；

③通常根结点可始终置于主存中，因此在这棵B-树中查找任一关键字至多只需二次访问外存。

3、B-树的存储结构

```
#define  Max   l000
                        //结点中关键字的最大数目：Max=m-1，m是B-树的阶
#define  Min   500
                        //非根结点中关键字的最小数目：Min= ⌈m/2⌉ -1
typedef  int  KeyType;                  //KeyType应由用户定义
typedef struct node     //结点定义中省略了指向关键字代表的记录的指针
{    int keynum          //结点中当前拥有的关键字的个数，keynum<Max
     KeyType key[Max+1] //关键字向量为key[1..keynum]，key[0]不用
     struct node *parent;               //指向双亲结点
     struct node *son[Max+1];           //孩子指针向量为son[0..keynum]
 } BTreeNode;
typedef   BTreeNode   *BTree;
```