

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

9.2.4 平衡二叉树

建立平衡二叉树旋转注意要点

1、首先确定属于何种类型的旋转；

- LL平衡旋转（单右旋）
- RR平衡旋转（单左旋）
- LR平衡旋转（先左后右）
- RL平衡旋转（先右后左）

2、确定旋转轴

LL和RR型的旋转轴在沿着失衡路径上，以失去平衡点的**直接后继结点**作为旋转轴；LR和RL的旋转轴是沿着失衡路径上，失去平衡点的**后两层结点**作为旋转轴。

9.2.4 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，

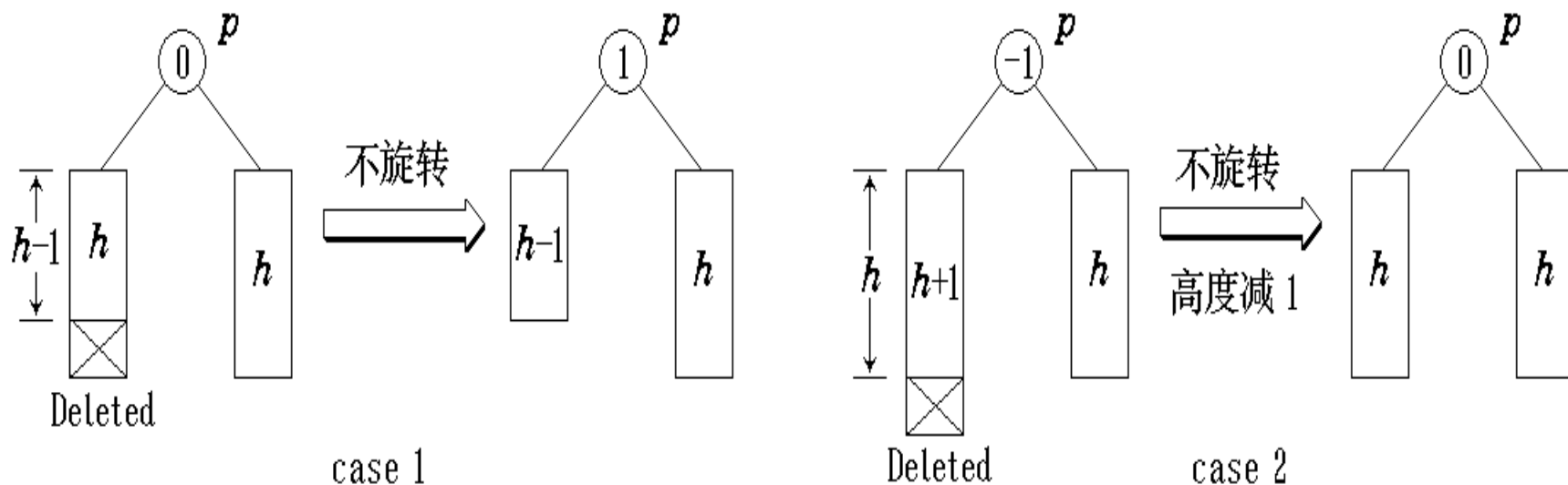
9.2.4 平衡二叉树

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。

9.2.4 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。

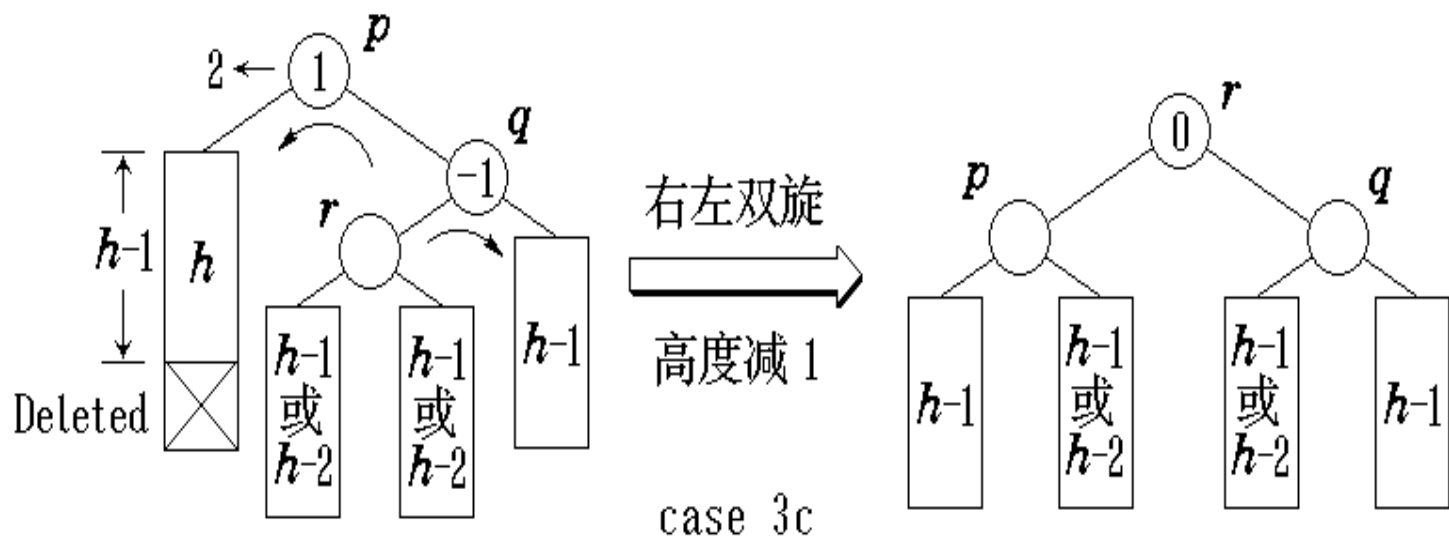
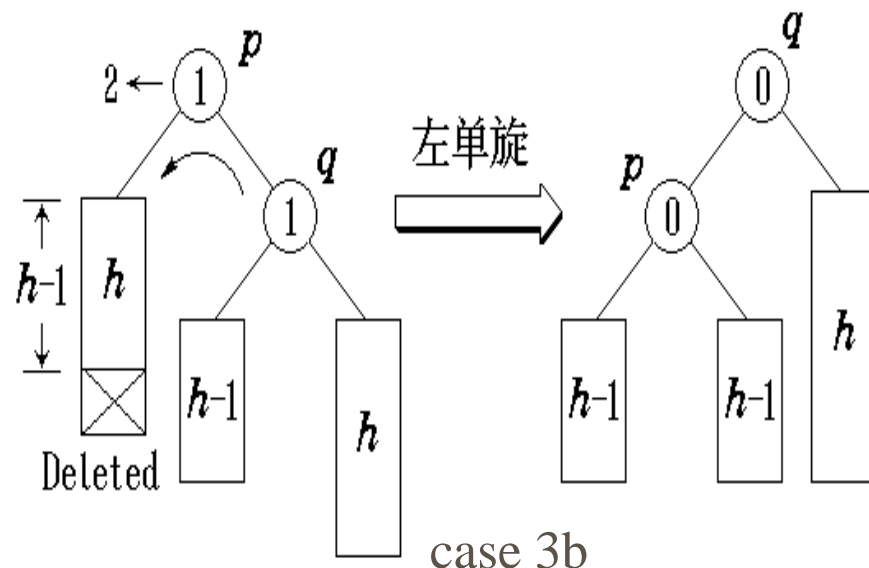
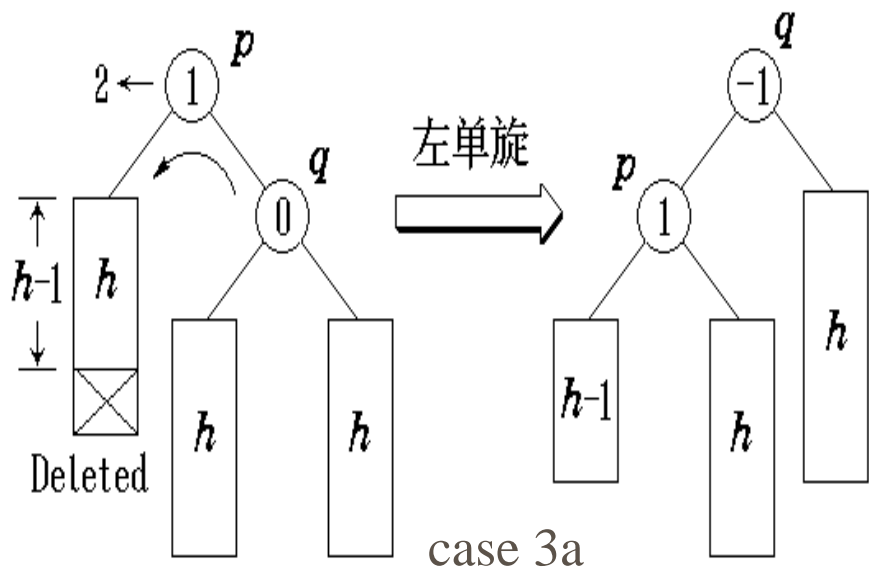


9.2.4 平衡二叉树

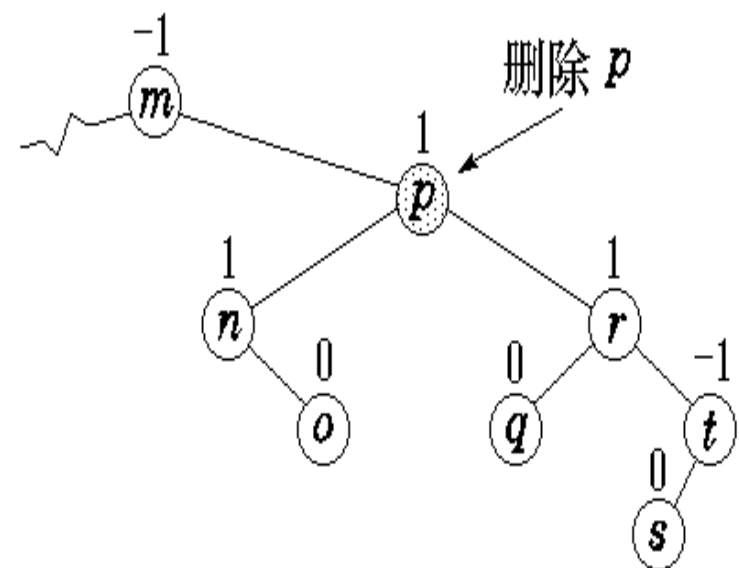
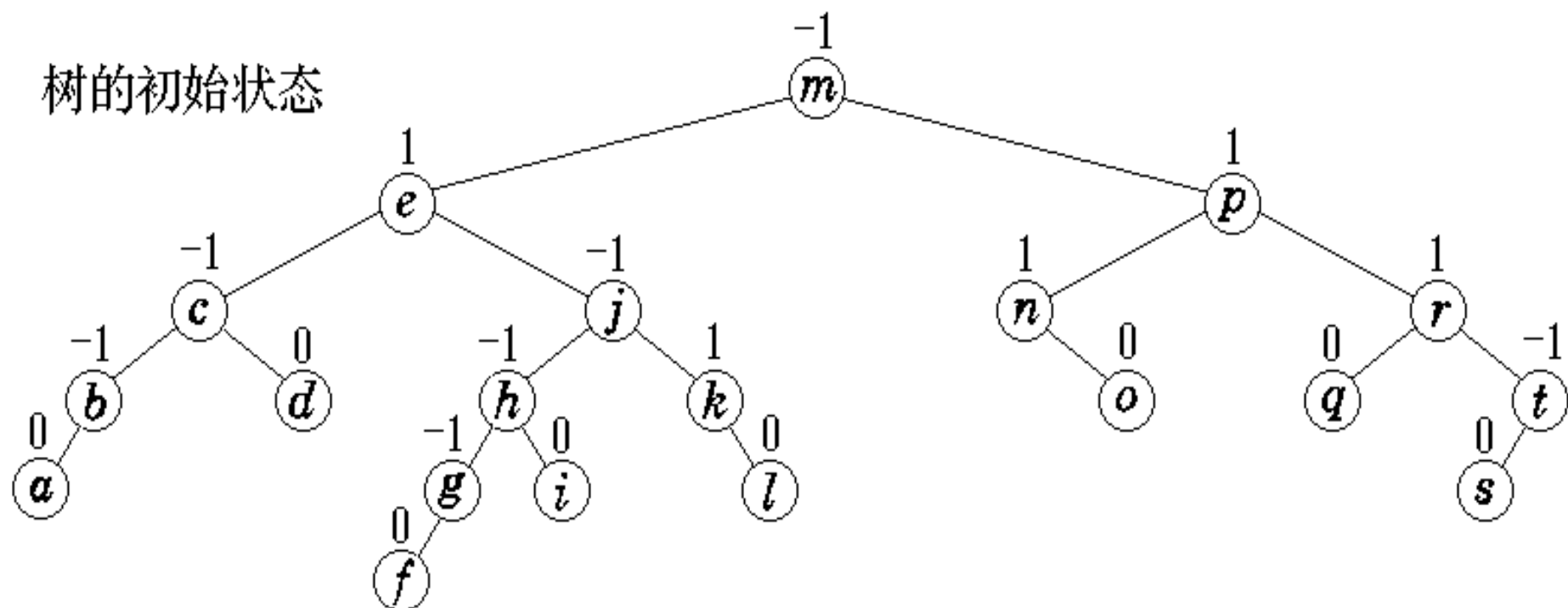
- case 3 : 结点 p 的 `balance` 不为0, 且较矮的子树又被缩短, 则在结点 p 发生不平衡。需要进行平衡化旋转来恢复平衡。令 p 的较高的子树的根为 q (该子树未被缩短), 根据 q 的 `balance`, 有如下 3 种平衡化操作。
- case 3a : 如果 q 的 `balance` 为0, 执行一个单旋转来恢复结点 p 的平衡, 置 `shorter` 为 `False`。
- case 3b : 如果 q 的 `balance` 与 p 的 `balance` 相同, 则执行一个单旋转来恢复平衡, 结点 p 和 q 的 `balance` 均改为0, 同时置 `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`。

9.2.4 平衡二叉树

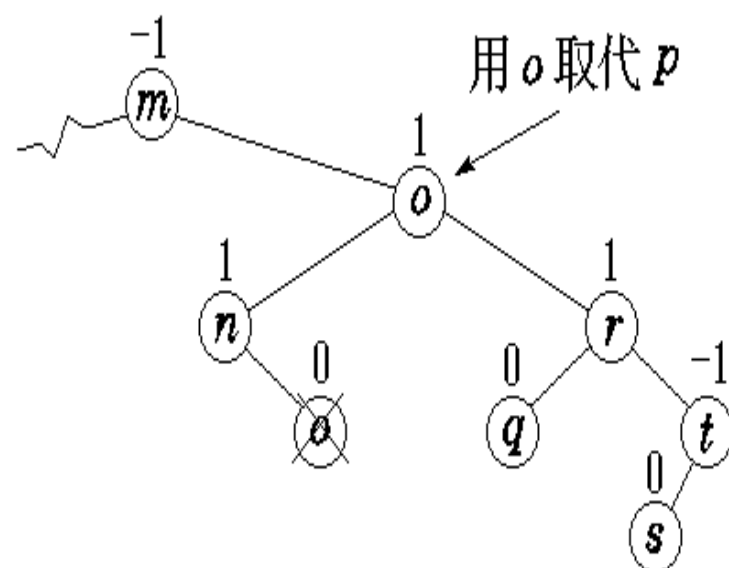
在case 3a, 3b和3c的情形中, 旋转的方向取决于结点 p 的哪一棵子树被缩短

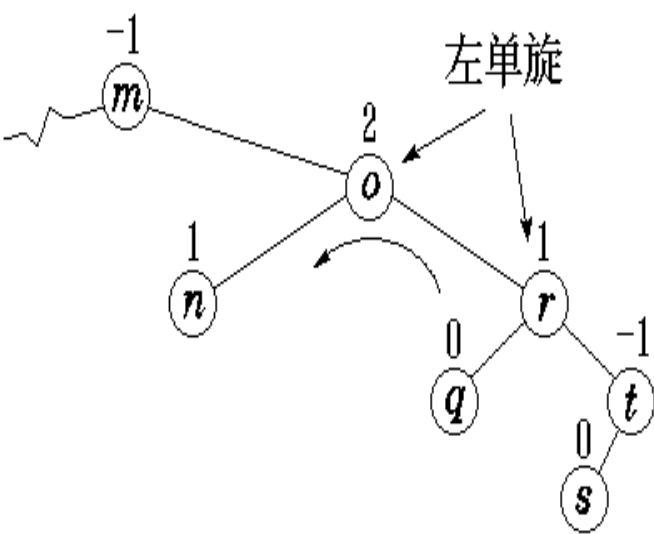


树的初始状态

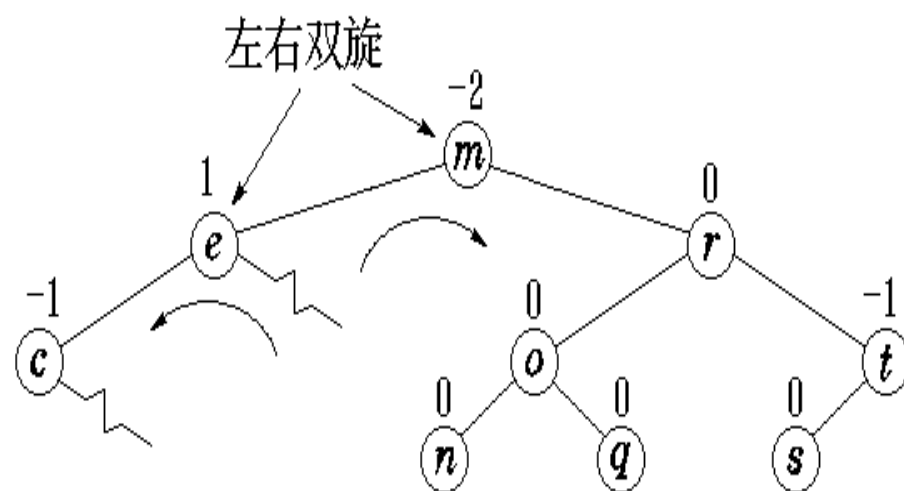


寻找 p 的中序下的直接前驱 o ，用 o 取代 p ，删除 o ，平衡旋转

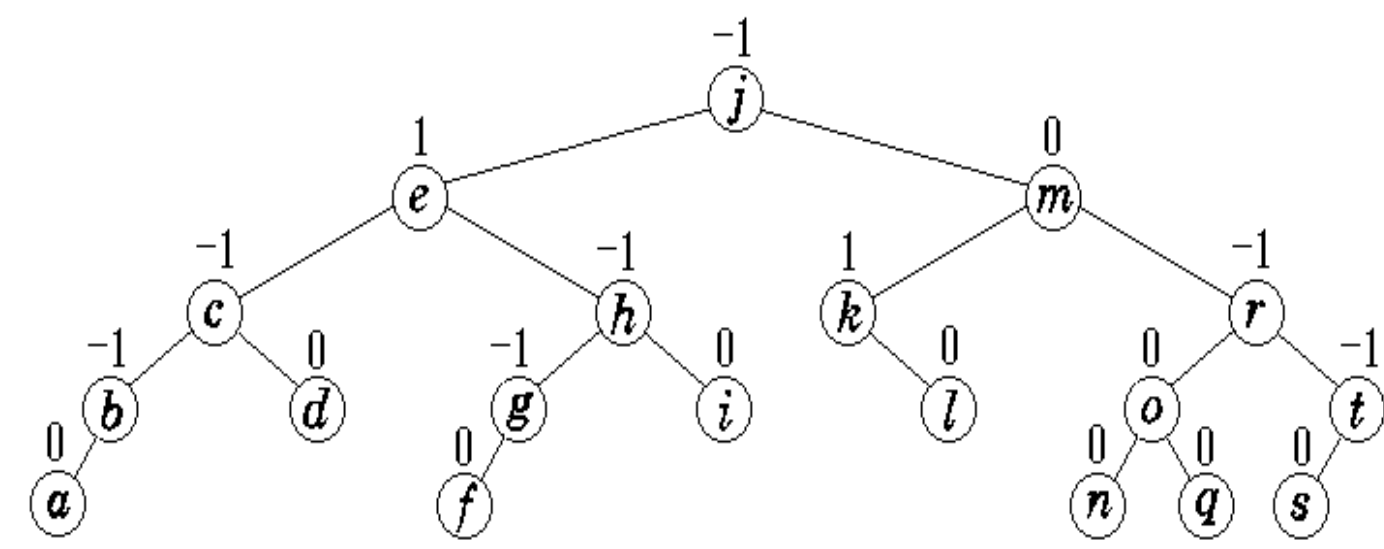




以 r 为旋转轴
作左单旋，子
树的高度减 1
 m 发生不平衡



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

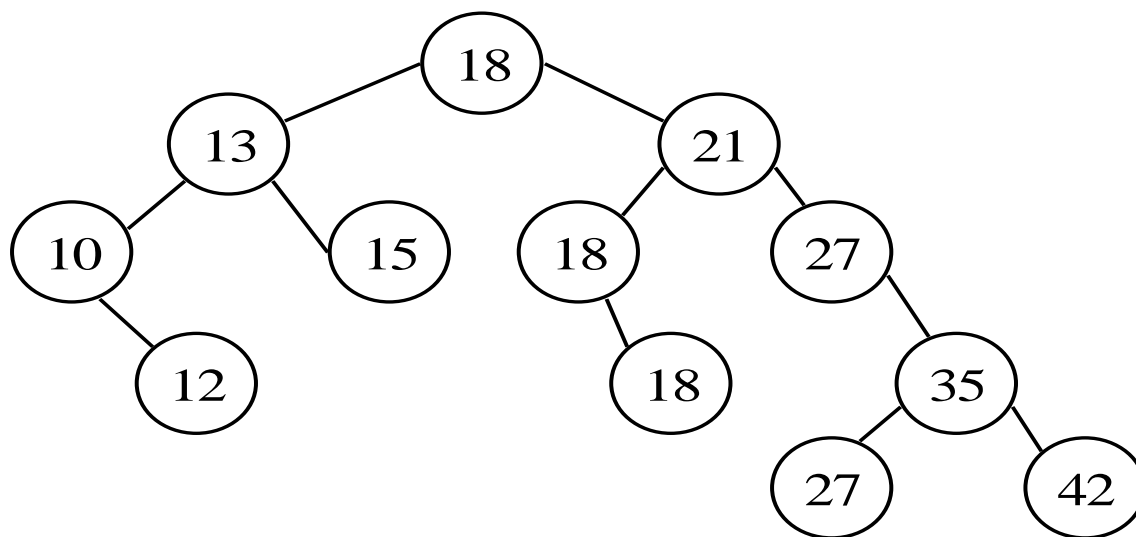


9.2.4 平衡二叉树

- 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 $h \geq 0$ 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 $O(\log n)$ 。

9.2.4 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。



9.2.4 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  
    }  
}
```

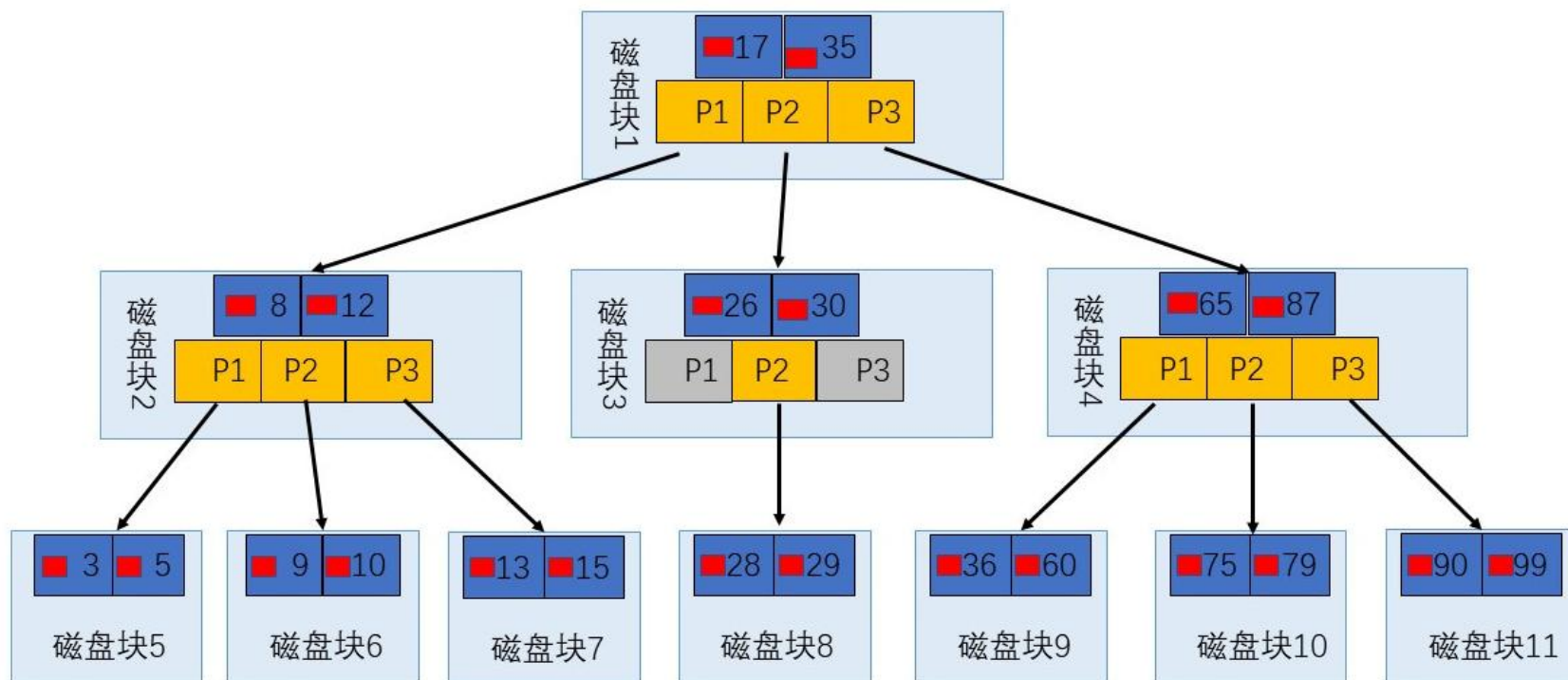
9.2 动态查找表

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)
- 五、B-树和B+树

9.2.5 B-树和B+树

多路查找树 (Muit-Way Search Tree)，其每一个**结点的孩子数**可以**多于两个**且**每一个结点处可以存储多个元素**。



9.2.5 B-树和B+树

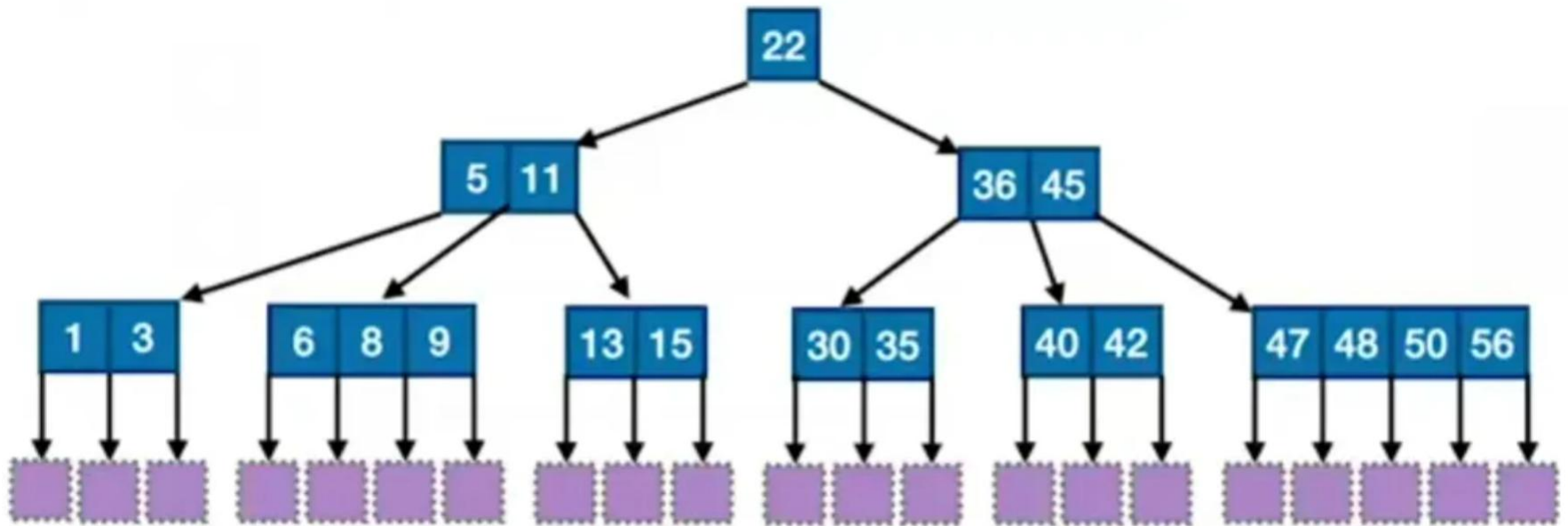
一、B-树的定义

B-树是一种平衡的多路查找树。一棵 m 阶的B-树，或为空树，或为满足下列特性的 m 叉树：

1. 树中每个结点至多有 m 棵子树， $m-1$ 个关键字；
2. 若根结点不是叶子结点，则至少有两棵子树；
3. 除根之外的所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树，至少有 $\lceil m/2 \rceil - 1$ 个关键字；

9.2.5 B-树和B+树

5阶B树



9.2.5 B-树和B+树

一、B-树的定义

4. 所有的非终端结点中包含下列信息数据：

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

其中： $K_i (i = 1, \dots, n)$ 为关键字，且 $K_i < K_{i+1} (i = 1, \dots, n-1)$

$$, \left(\left\lceil \frac{m}{2} \right\rceil - 1 \leq n \leq m - 1 \right);$$

$A_i (i = 1, \dots, n)$ 为指向子树根结点的指针，且指针 A_{i-1} 所指子树中所有结点的关键字均小于 $K_i (i = 1, \dots, n)$ ， A_i 所指子树中所有结点的关键字均大于 K_i ， n 为关键字的个数（或 $n+1$ 为子树个数）。

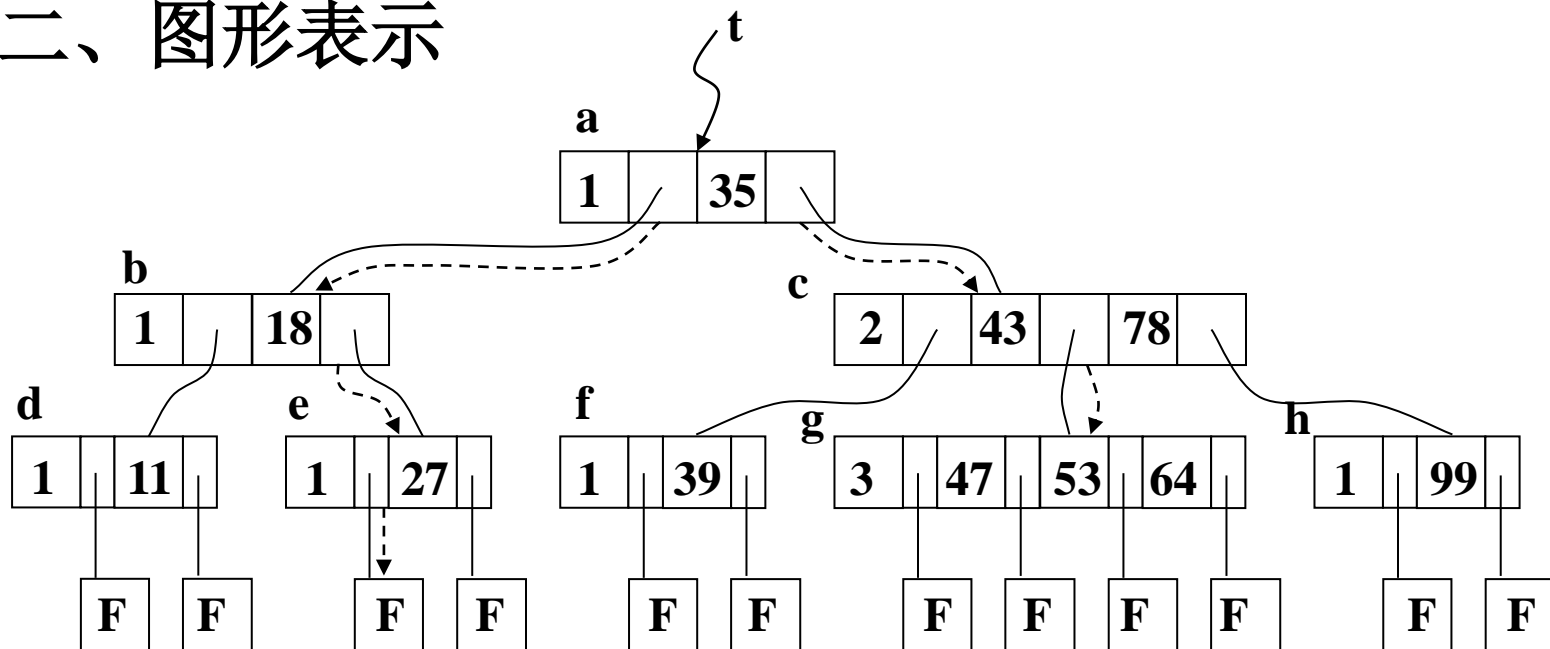
9.2.5 B-树和B+树

一、B-树的定义

5. 所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

9.2.5 B-树和B+树

二、图形表示

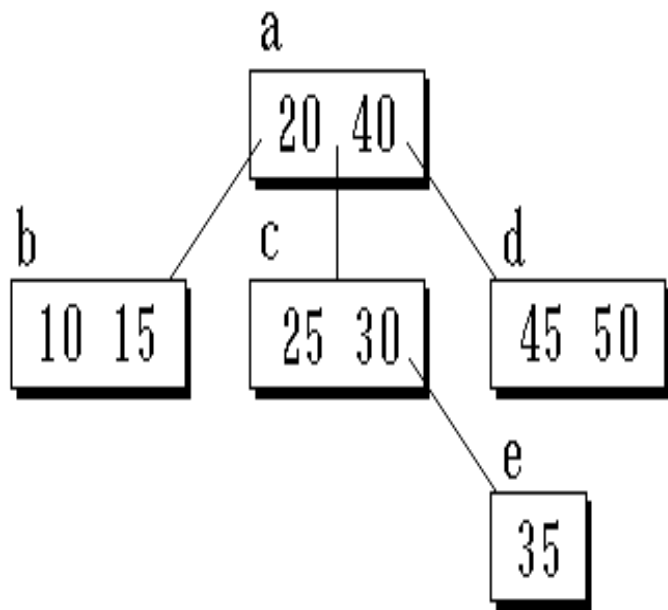


一棵4阶的B-树

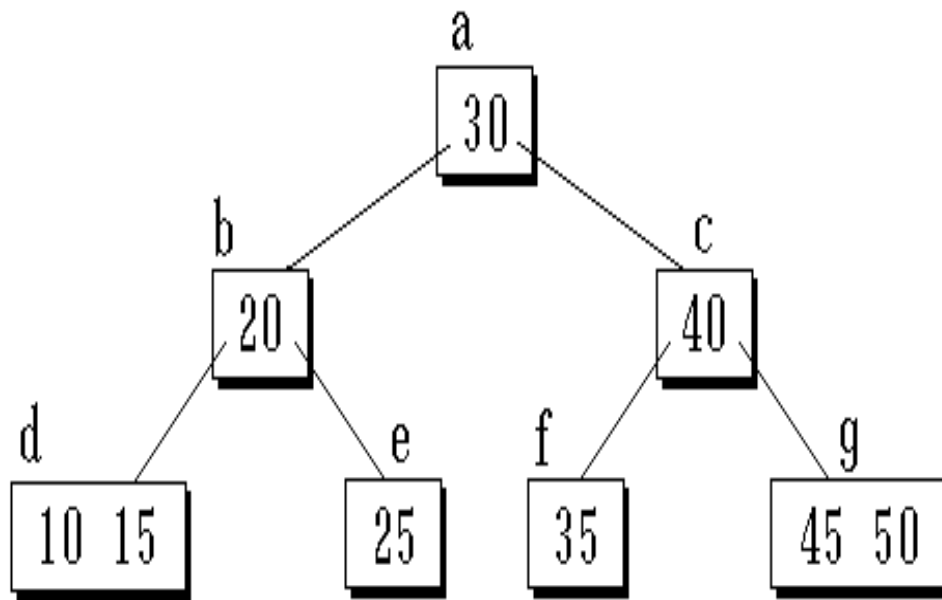
结点中关键字个数为 $1 \leq n \leq 3$

子树数量范围为 $2 \leq k \leq 4$

9.2.5 B-树和B+树



非B-树



B-树

9.2.5 B-树和B+树

B-树的类型定义

```
#define M 3 //B-树的阶，根据实际情况需要设定
typedef struct BTNode{
    int keynum; //结点中的关键字个数
    struct BTNode *parent; //指向双亲结点
    KeyType key[M+1]; //关键字向量，0号单元未用
    struct BTNode *ptr[M+1] //子树指针向量
    Record *recptr[M+1]; /*记录指针向量，0号单元
    未用*/
}BTNode, *Btree;
```

9.2.5 B-树和B+树

B-树查询结构体

```
typedef struct {  
    BTNode      *pt;    //指向找到的结点  
    int          i;    //1..m-1, 在结点中的关键字序号  
    int          tag;    //1:查找成功, 0:查找失败  
} Result;              //B-树的查找结果类型
```

9.2.5 B-树和B+树

三、B-树的查找

从根结点出发，沿指针搜索结点和在结点内进行顺序（或折半）查找两个过程交叉进行。

若查找成功，则返回指向被查关键字所在结点的指针和关键字在结点中的位置；

若查找不成功，则返回插入位置。

9.2.5 B-树和B+树

- 查找过程如下：

- 将根节点作为当前节点 //常驻RAM

- 只要当前节点非外部节点

- 在当前节点中顺序或折半查找 //RAM内部

- 若找到目标关键码，则返回查找成功

- 否则 //止于某一对下层引用

- 沿引用，转至对应子树

- 将其根节点读入内存

- ///O操作，最为耗时

- 更新当前节点

- 返回查找失败

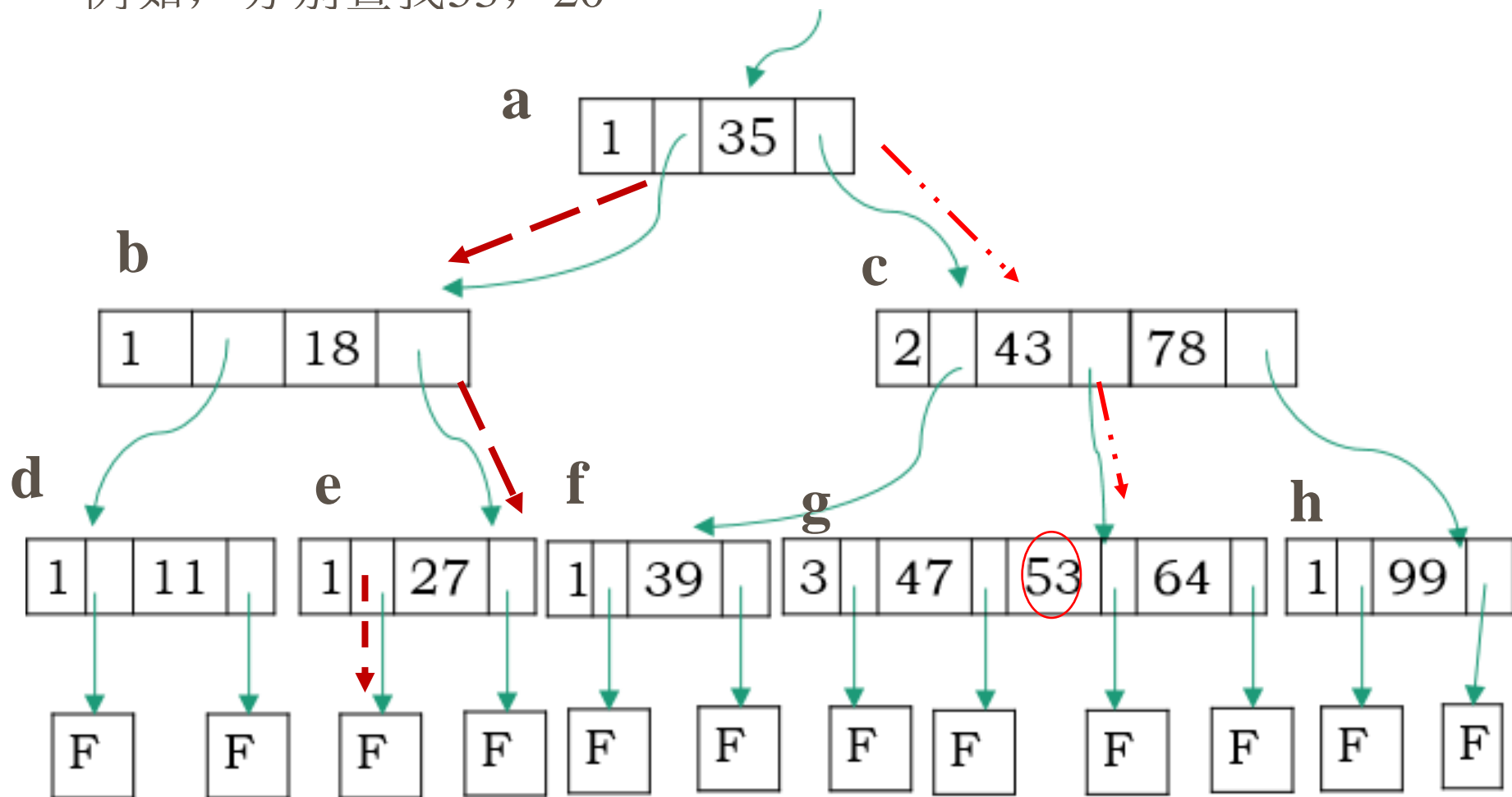

```

Result SearchBTree (BTree T, KeyType K) {
    //在m阶B-树T上查找关键字K，返回结果(pt, i , tag)。若查找成功，
    //则特征值tag=1，指针pt所指结点中第i个关键字等于K；否则
    //特征值tag=0，等于K的关键字应插入在指针pt所指结点中第i和第i+1个关键字之间
    p = T;
    /*初始化，p指向待查结点，q指向p的双亲*/
    q = NULL;
    found = FALSE;
    i = 0;
    while (p && !found) {
        i = Search (p, K);
        /*在p->key[1..keynum]中查找，i使得： p->key[i] < K < p->key[i + 1]*/
        if (i > 0 && p->key[i] == K)
            found = TRUE;    //找到待查关键字
        else {
            q = p;
            p = p->ptr[i];
        }
    }
    if (found)
        return (p, i, 1);    //查找成功
    else
        return (q, i, 0);    //查找不成功，返回K的插入位置信息
} // SearchBTree

```

9.2.5 B-树和B+树

例如，分别查找53， 20



9.2.5 B-树和B+树

三、B-树的查找

查找分析

从上述算法可见，在B-树上进行查找包含两种基本操作：

- a.在B-树中找结点（操作在磁盘上进行）；
- b.在结点中找关键字（操作在内存中进行）。

显然，在磁盘上进行一次查找比在内存中进行一次查找耗费时间多得多，因此，在磁盘上进行查找的次数、即待查关键字所在结点在B-树上的层次数，是决定B-树查找效率的首要因素。

9.2.5 B-树和B+树

三、B-树的查找

查找分析

- 若定义在 m 阶B-树中，失败结点位于第 $h + 1$ 层。则在这棵B-树中关键字个数 N 最小能达到多少？

从B-树的定义知：

第一层 至少 1 个结点

第二层 至少 2 个结点

第三层 至少 $2 \lceil m / 2 \rceil$ 个结点

第四层 至少 $2 \lceil m / 2 \rceil^2$ 个结点

.....

第 h 层 至少有 $2 \lceil m / 2 \rceil^{h-2}$ 个结点。所有这些结点都不是失败结点。

第 $h+1$ 层 至少有 $2 \lceil m / 2 \rceil^{h-1}$ 个结点都是失败结点

9.2.5 B-树和B+树

四、B-树的查找

查找分析

- 而 $l+1$ 层的结点为叶子结点。若 m 阶B-树中具有 N 个关键字，则叶子结点即查找不成功的结点为 $N+1$ ，因此：

$$N + 1 \geq 2 * (\lceil m / 2 \rceil)^{l-1}$$

$$l \leq \log_{\lceil \frac{m}{2} \rceil} \left(\frac{n+1}{2} \right) + 1$$

含有 n 个关键字的B-树上查找时，从根结点到关键字结点的路径上涉及的结点数不超过上式。

9.2.5 B-树和B+树

五、B-树的插入

1 算法思想：由于B-树结点中的关键字个数必须
 $\geq \lceil m/2 \rceil - 1$ 。

因此，每次插入一个关键字不是在树中添加一个叶子结点，而是首先在最低层的某个非终端结点中添加一个关键字，若该结点的关键字个数不超过 $m-1$ ，则插入完成，否则要产生结点的“分裂”。

9.2.5 B-树和B+树

五、B-树的插入

2. 结点如何实现“分裂”：

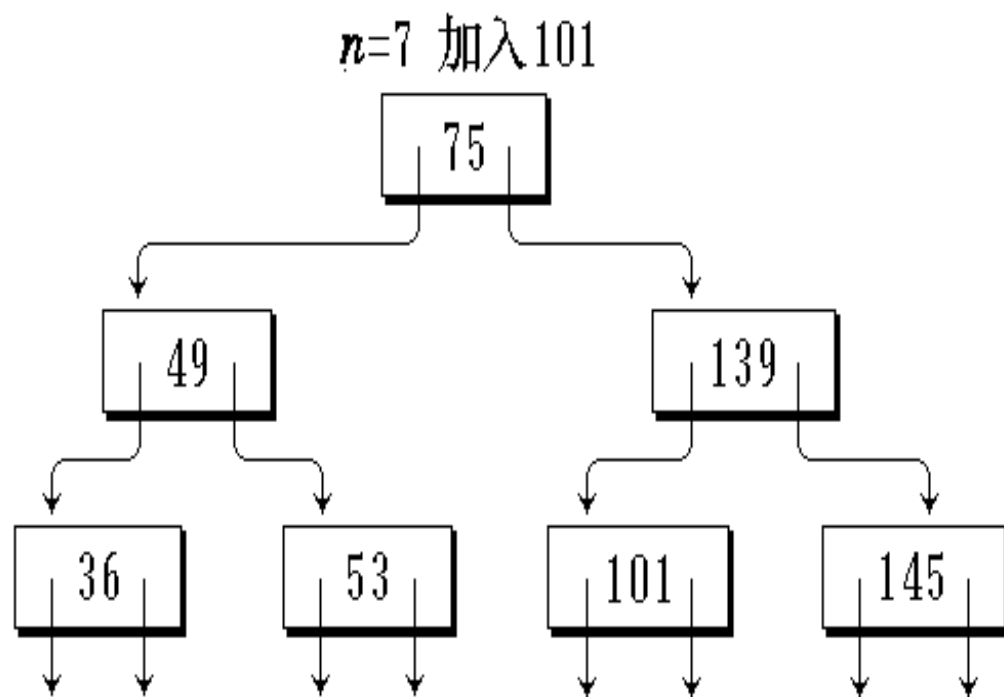
设结点 A 中已经有 $m-1$ 个关键字，当再插入一个关键字后，结点中的状态为 $(m, A_0, K_1, A_1, K_2, A_2, \dots, K_m, A_m)$ ，其中 $K_i < K_{i+1}, 1 \leq i < m$ 。这时必须把结点 p 分裂成两个结点 p 和 q，它们包含的信息分别为：

p: $(\lceil m/2 \rceil - 1, A_0, K_1, A_1, \dots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$

q: $(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}, \dots, K_m, A_m)$

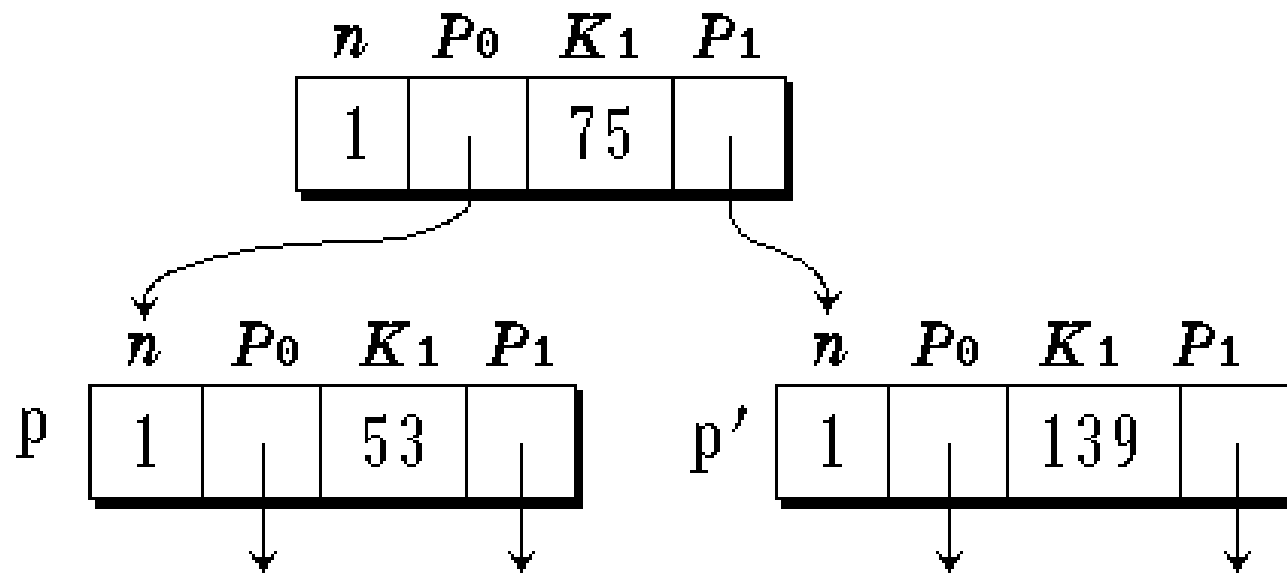
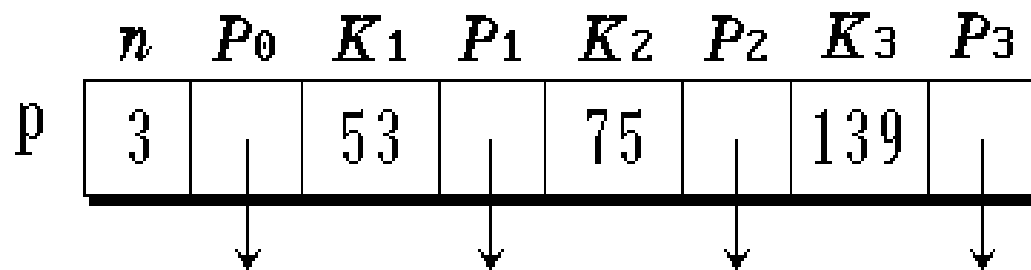
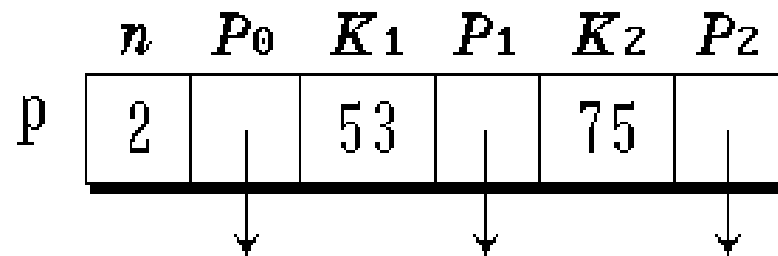
位于中间的关键字 $K_{\lceil m/2 \rceil}$ 与指向新结点 q 的指针形成一个二元组 $(K_{\lceil m/2 \rceil}, q)$ ，插入到这两个结点的双亲结点中去。

五、B-树的插入



若设B-树的高度为 h ，那么在自顶向下查找到叶结点的过程中需要进行 h 次读盘。

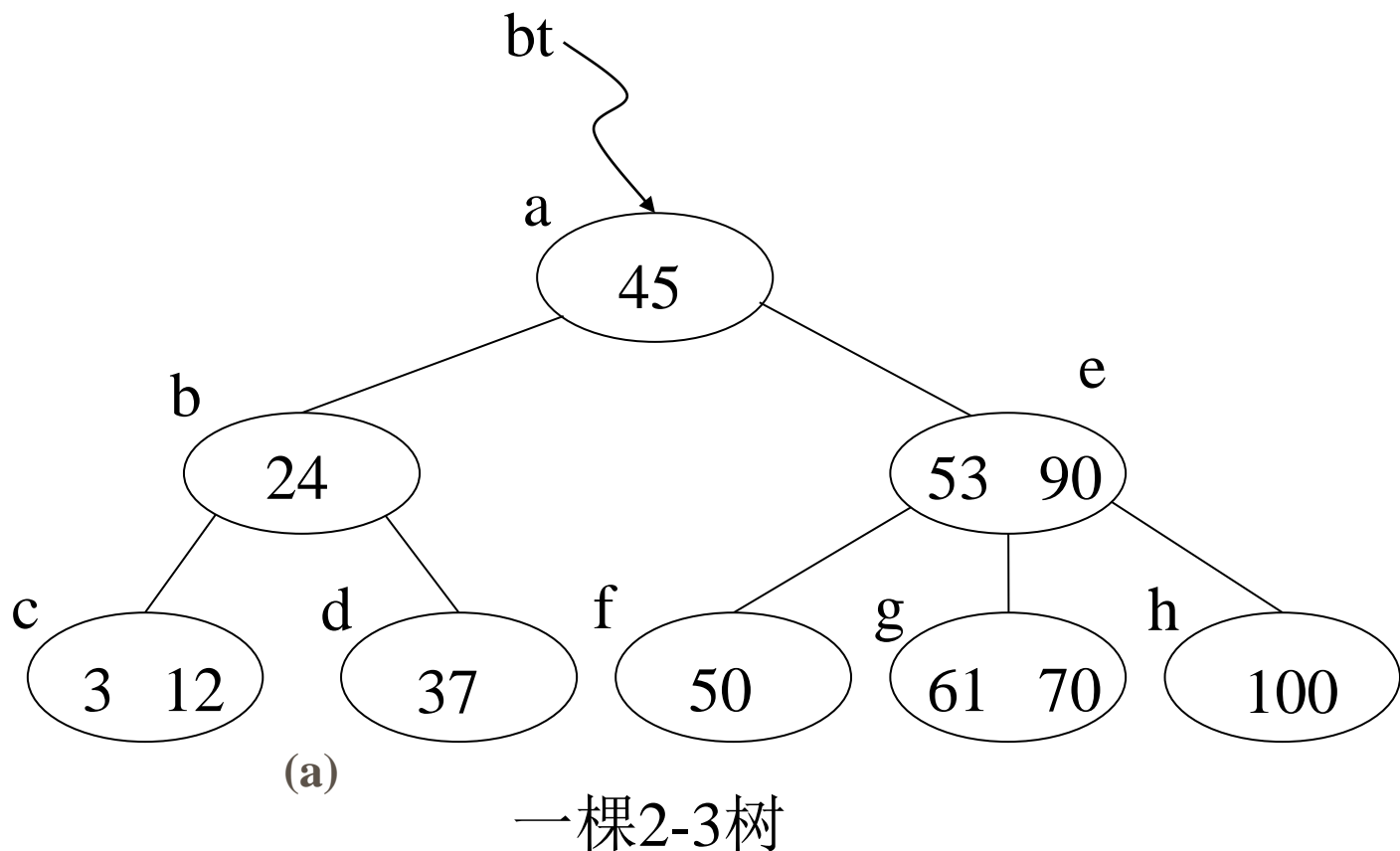
- 在插入新关键字时，需要自底向上分裂结点，最坏情况下从被插关键字所在叶结点到根的路径上的所有结点都要分裂。



五、B-树的插入

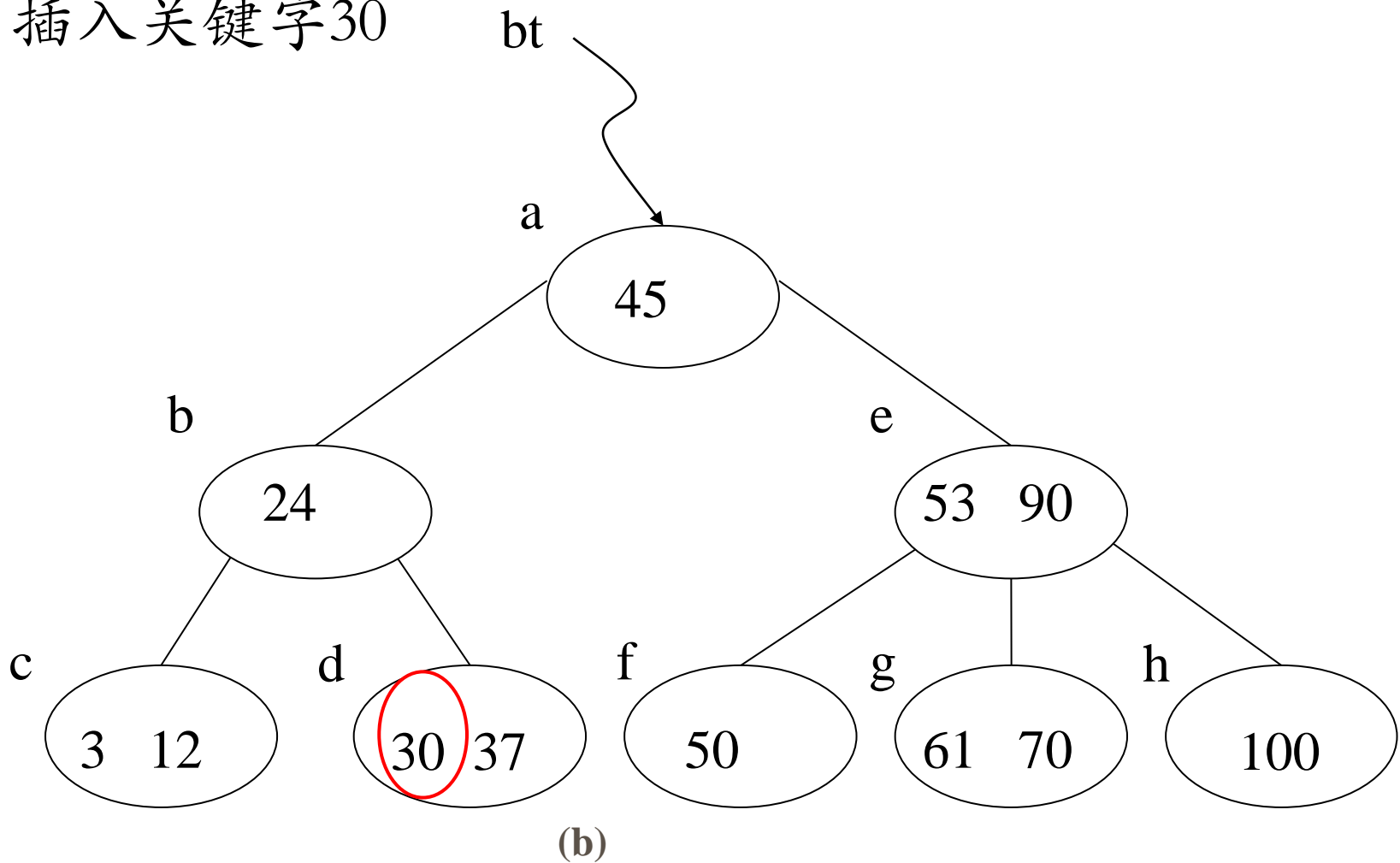
3. 例子

例如，图2所示为3阶B-树（图中略去F结点，即叶子结点），假设需一次插入关键字30，26，85和7。



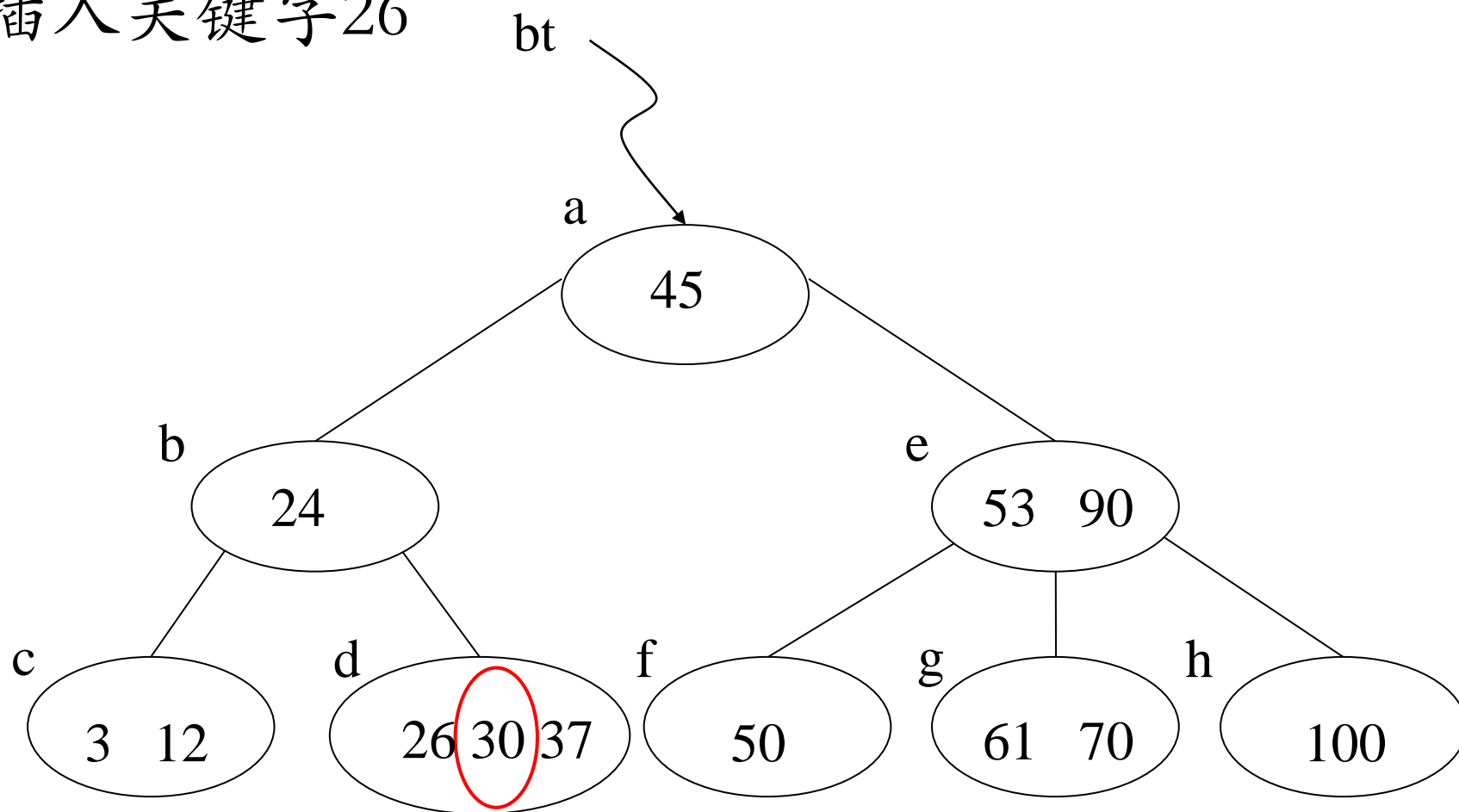
五、B-树的插入

插入关键字30



五、B-树的插入

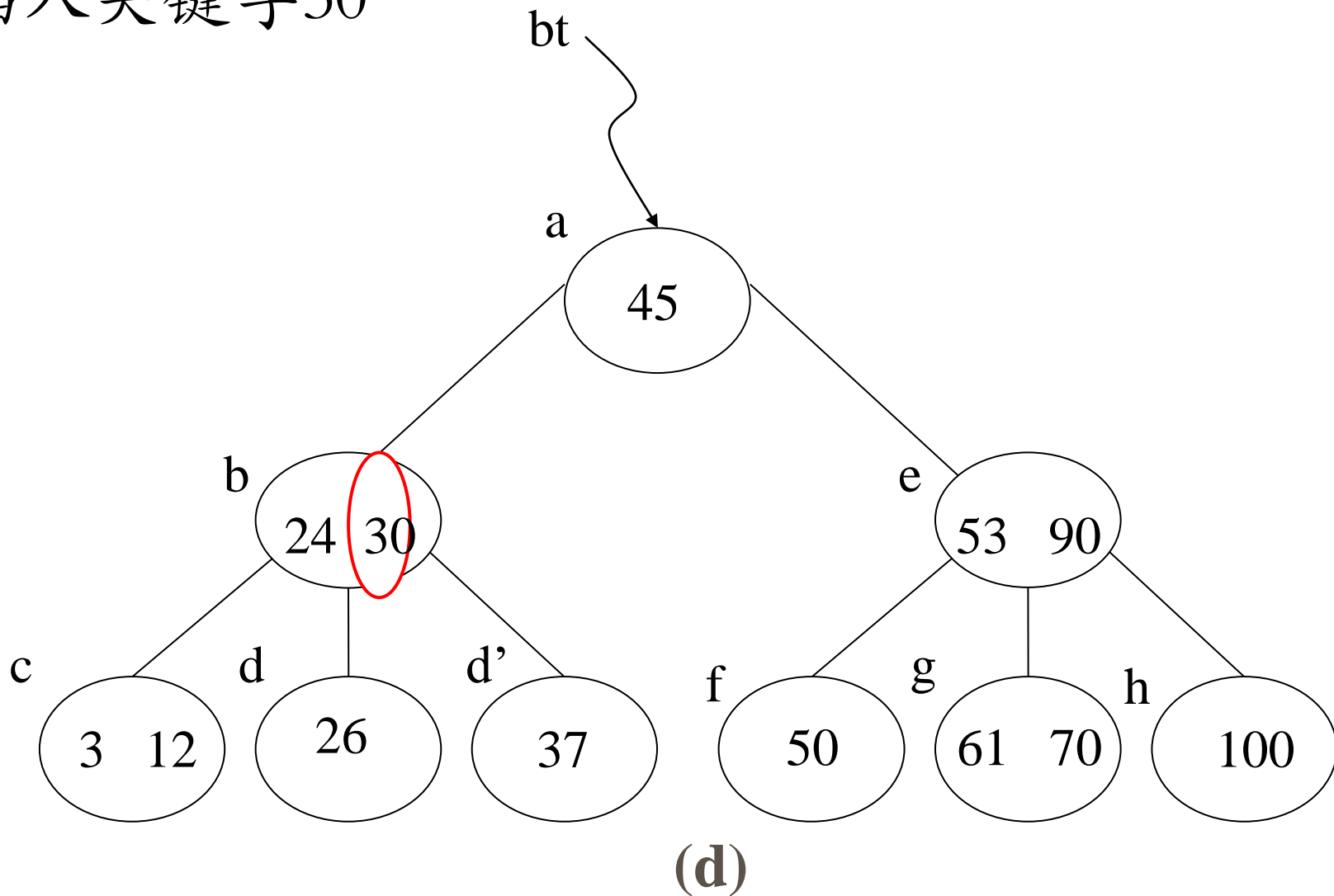
插入关键字26



(c)

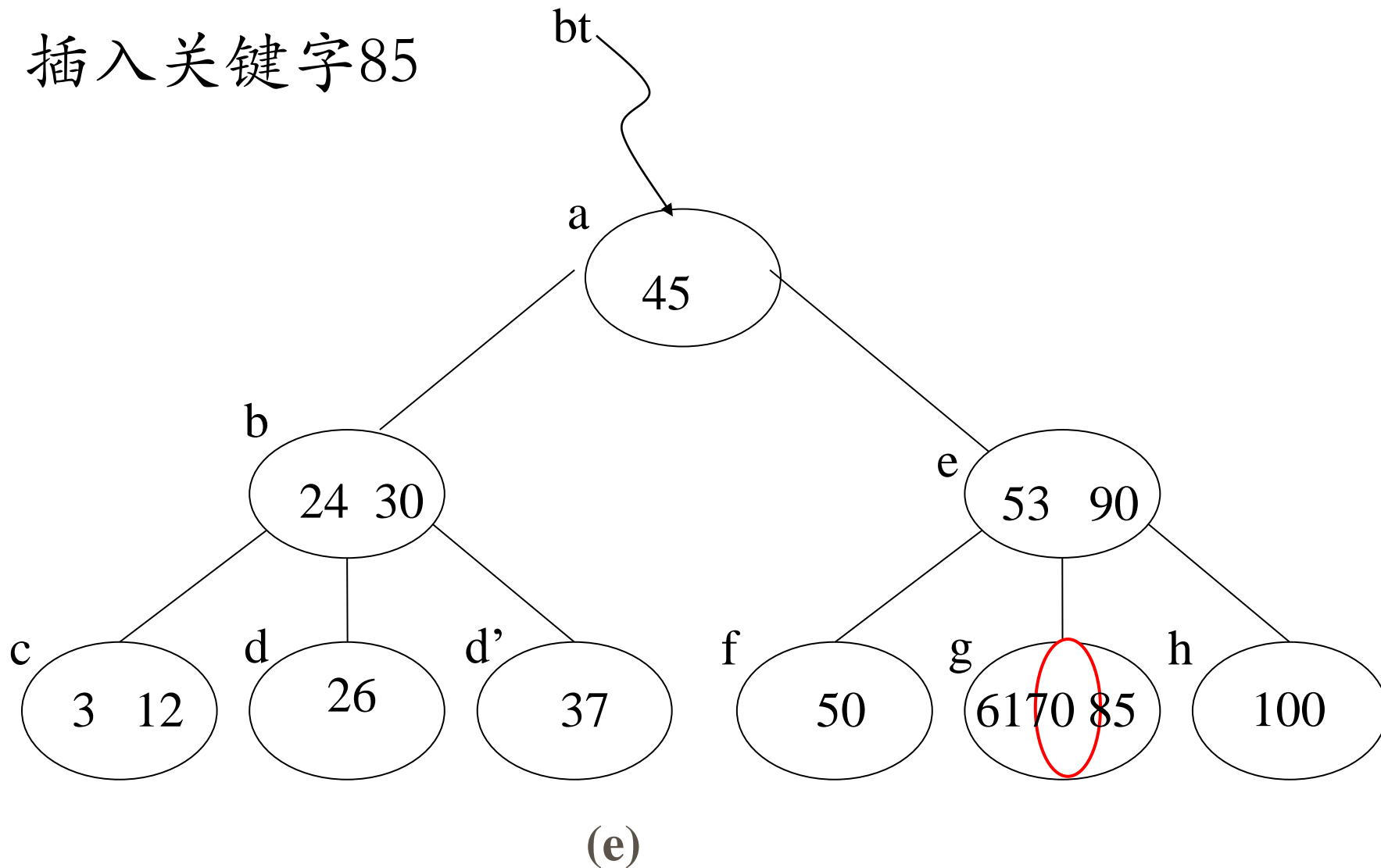
五、B-树的插入

插入关键字30



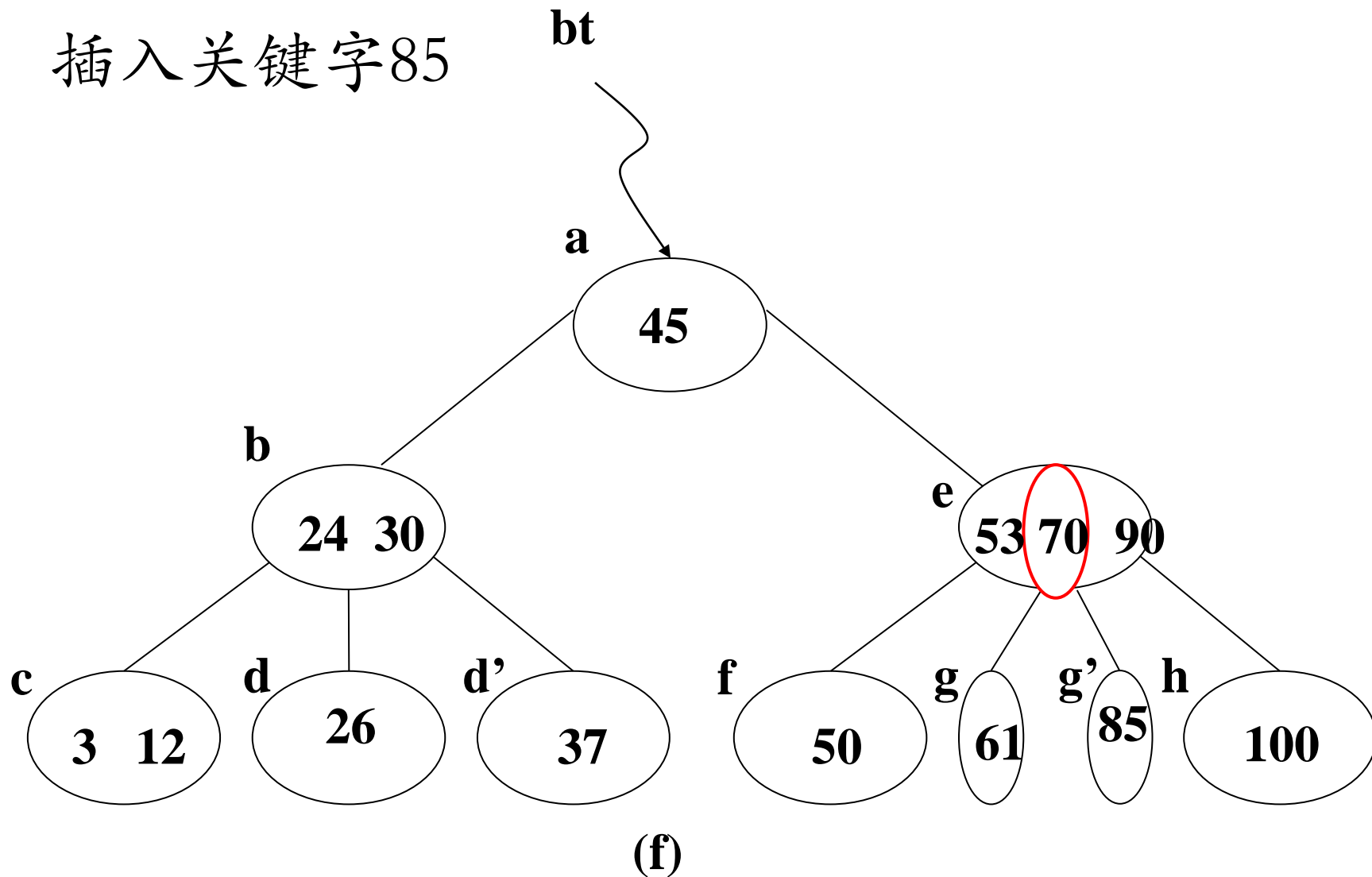
五、B-树的插入

插入关键字85



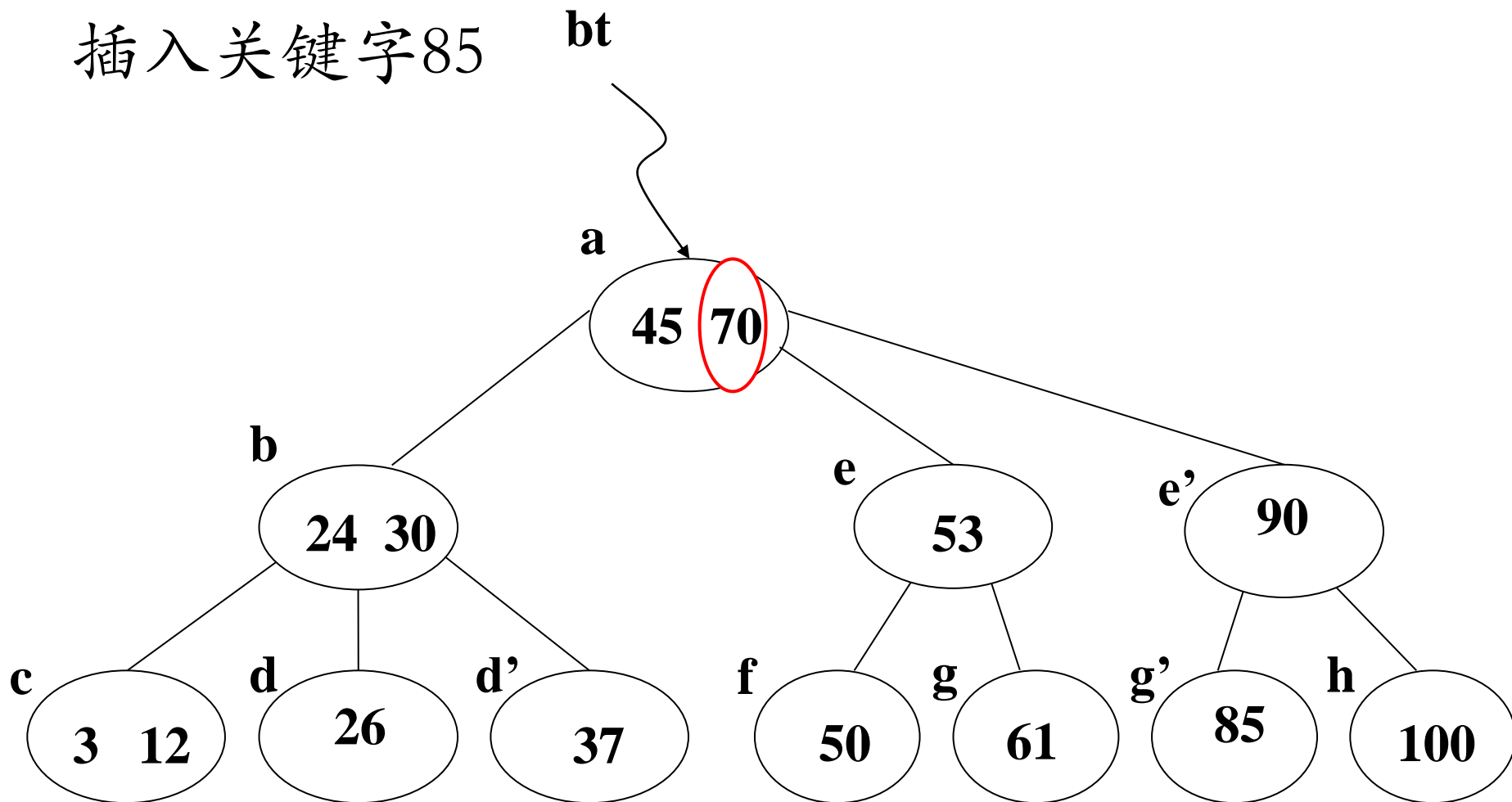
五、B-树的插入

插入关键字85



五、B-树的插入

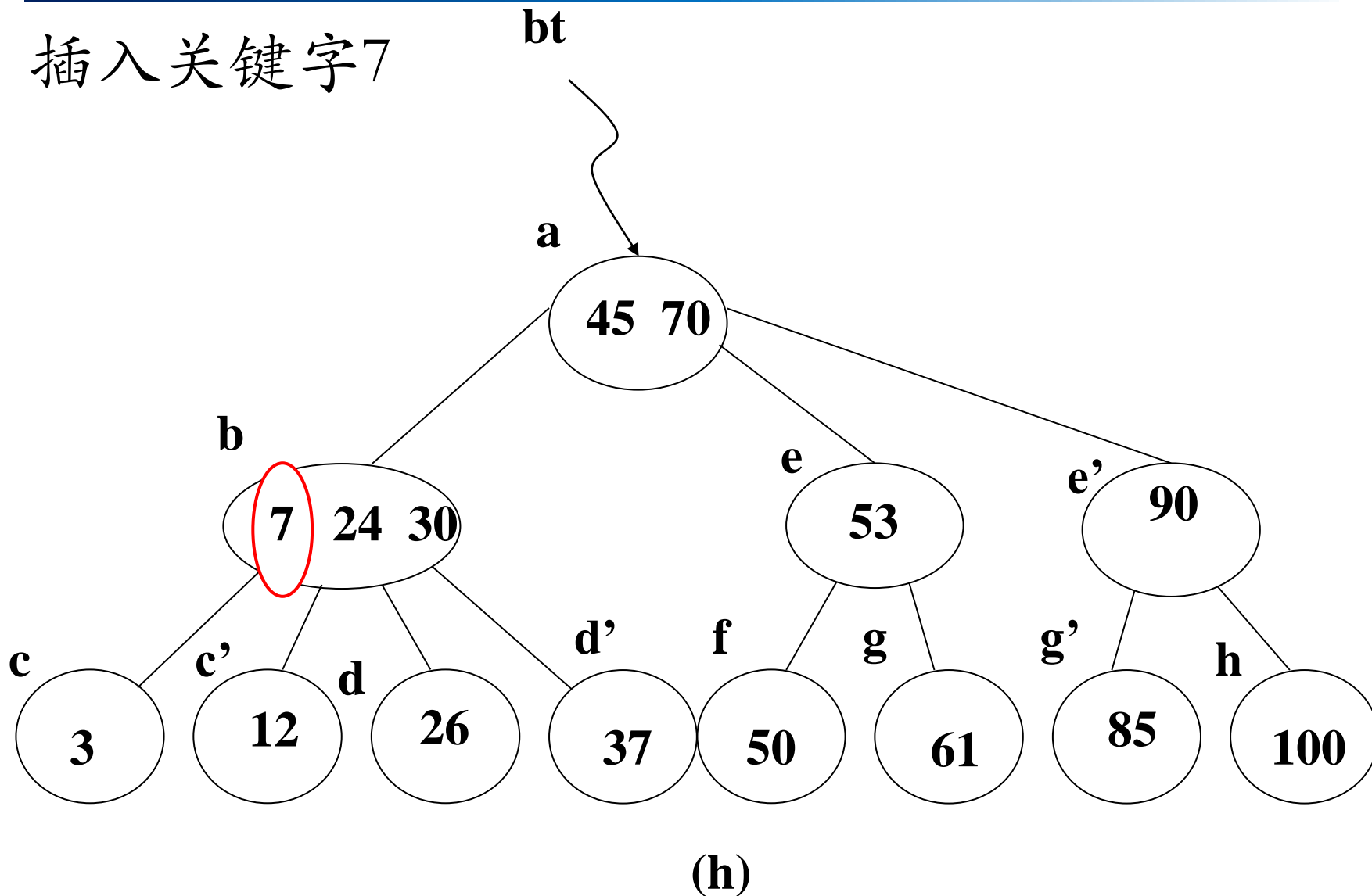
插入关键字85



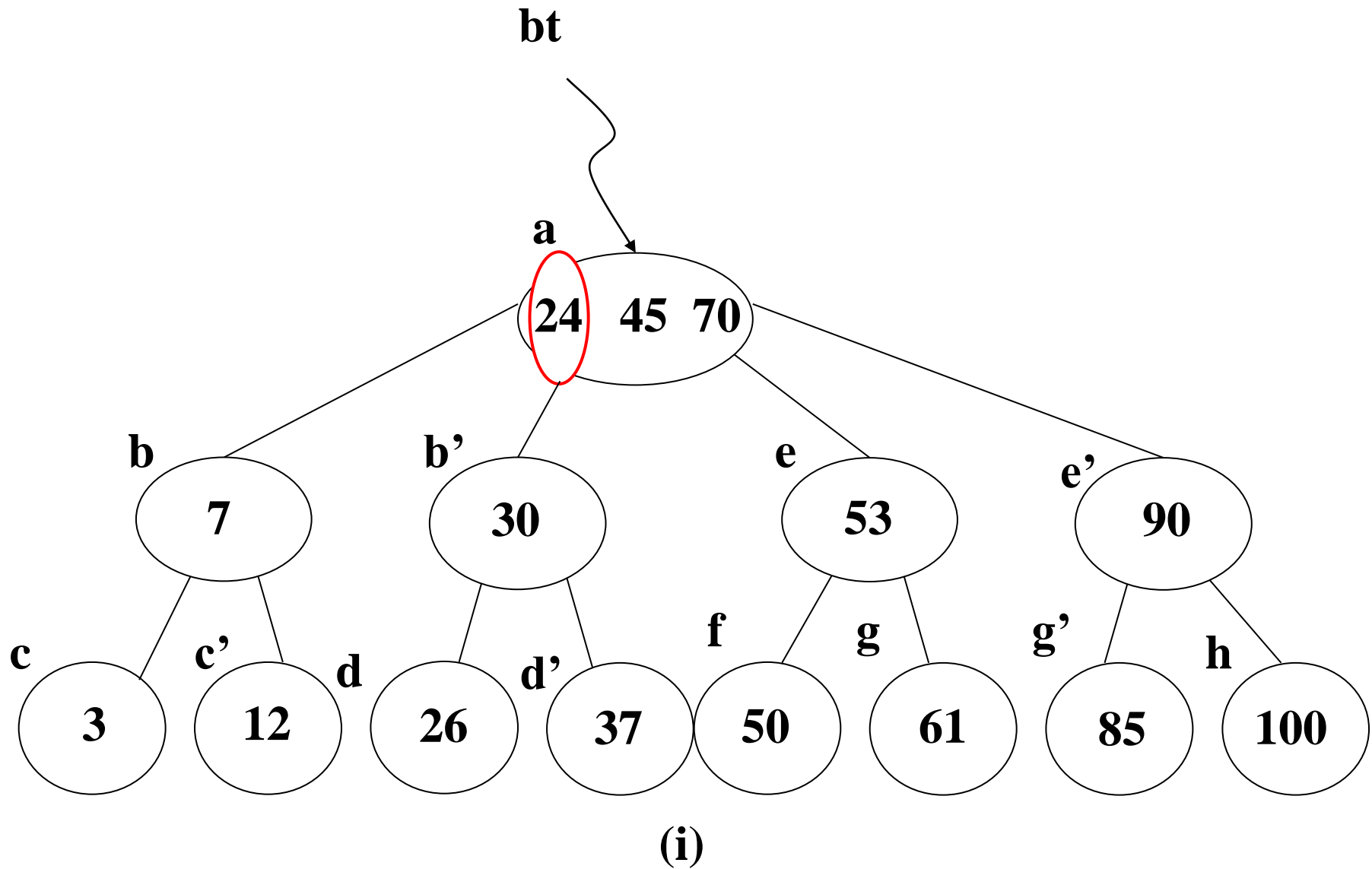
(g)

五、B-树的插入

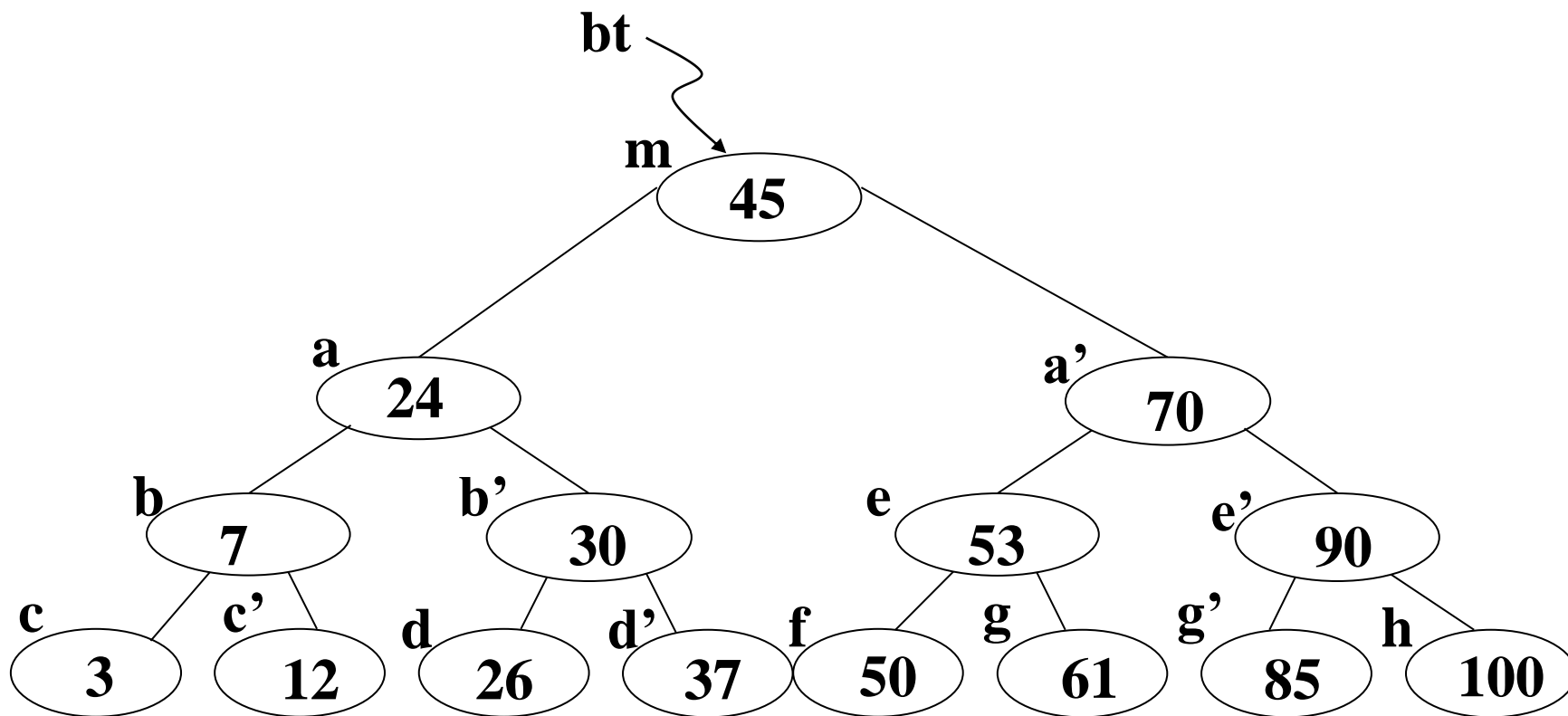
插入关键字7



五、B-树的插入



五、B-树的插入

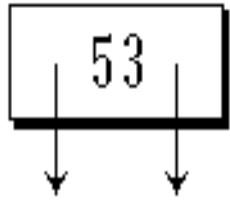


(j)

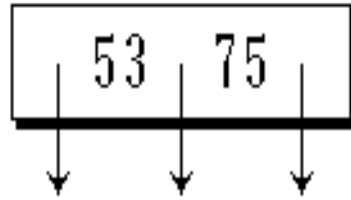
在B-树中进行插入（省略叶子结点）

示例：从空树开始逐个加入关键字建立3阶B-树

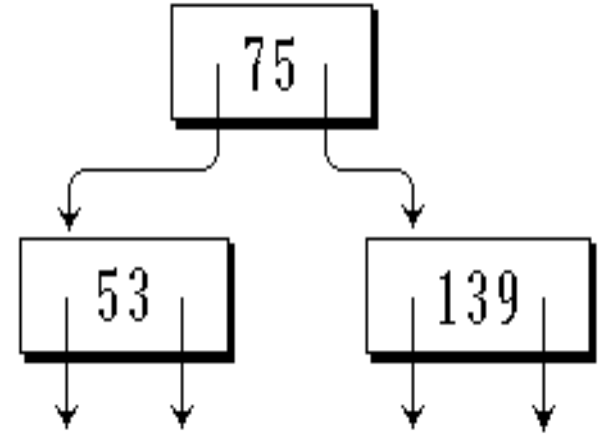
$n=1$ 加入 53



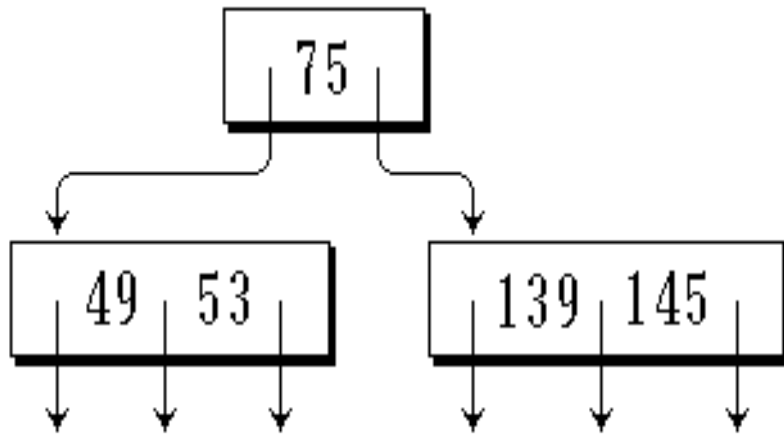
$n=2$ 加入 75



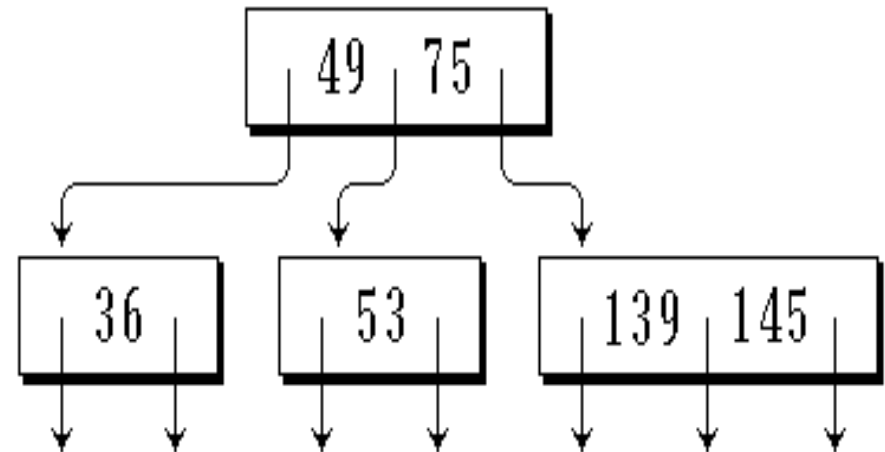
$n=3$ 加入 139



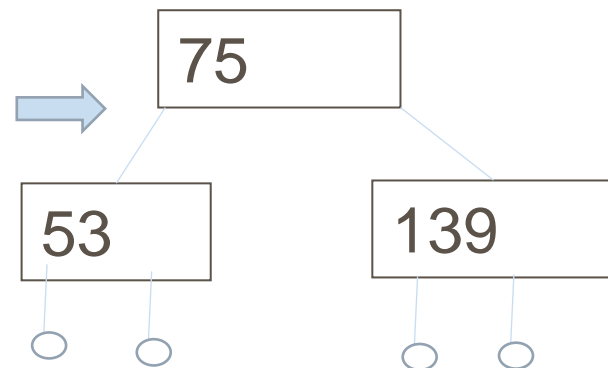
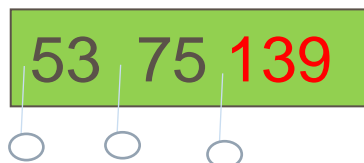
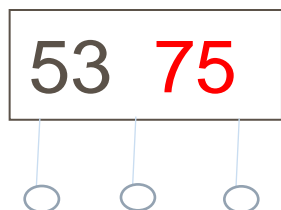
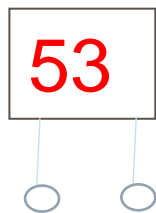
$n=5$ 加入 49, 145



$n=6$ 加入 36

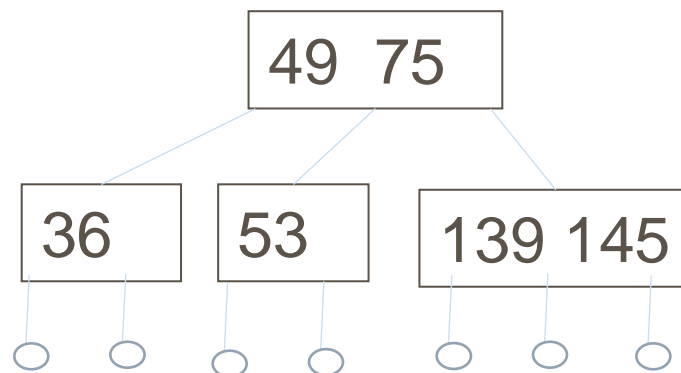
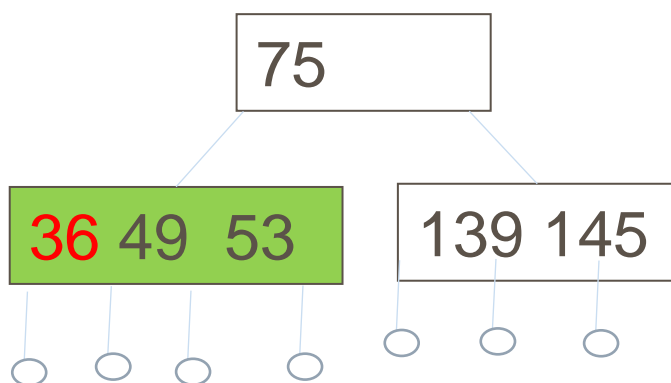
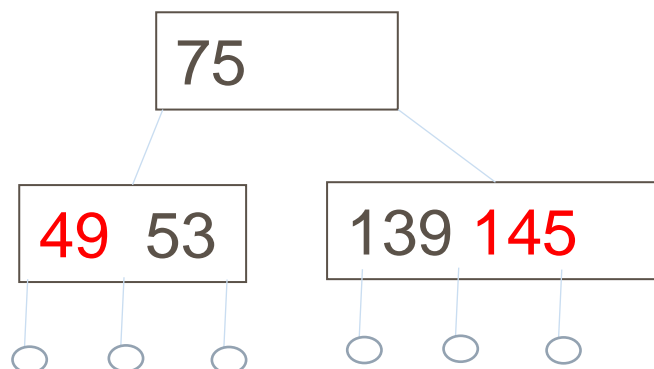


53, 75, 139, 49, 145, 36,

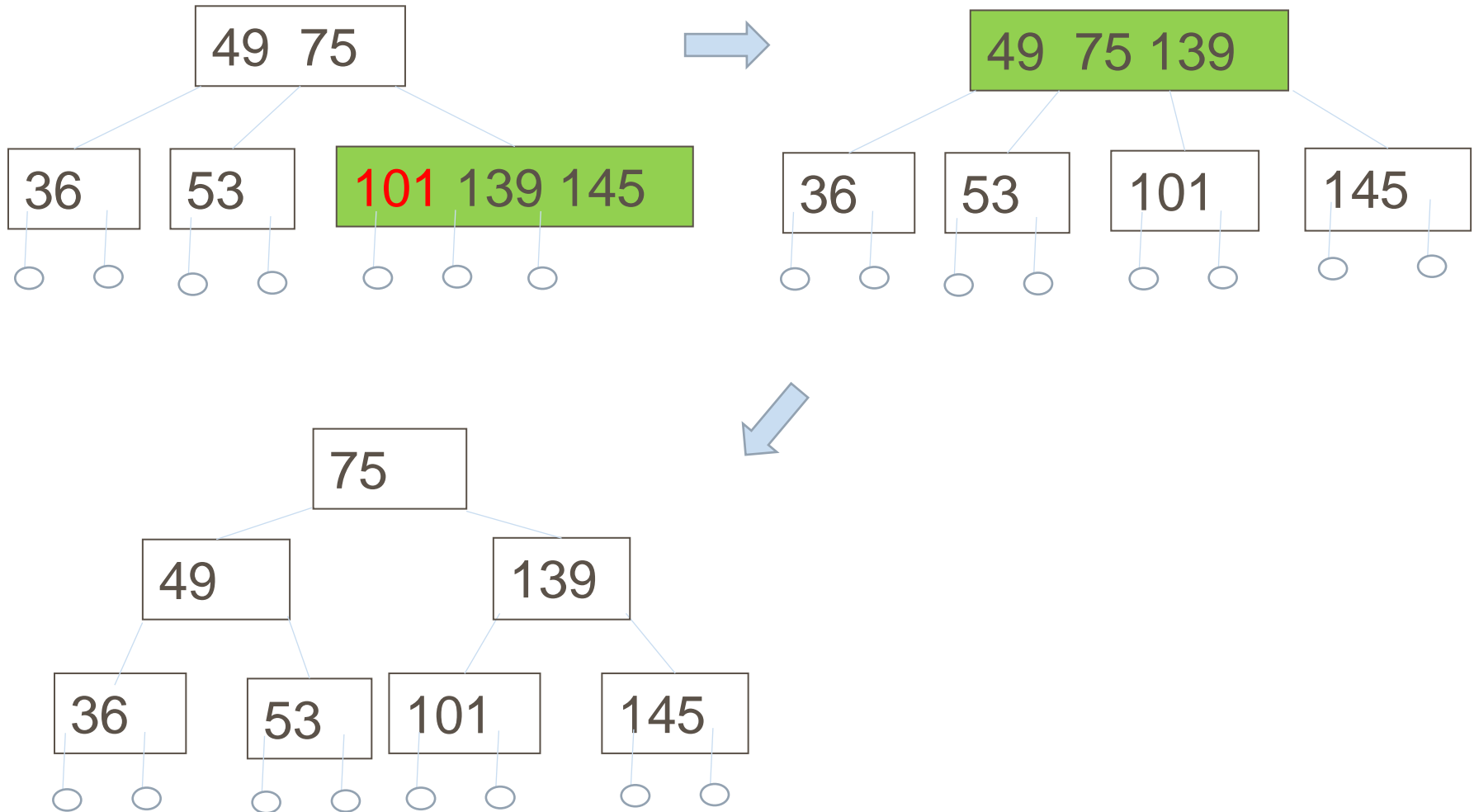


上溢时，分裂

取中间关键字，以及指向分裂后结点的指针，一起上移到双亲结点中



53, 75, 139, 49, 145, 36, 101,



五、B-树的插入

```
int InsertBTree(BTree &T,KeyType K,BTree q, int i){
//在m阶B树T上结点*q的 key[i]与 key[i+1]之间插入关键字K。
//若引起结点过大,则沿双亲链进行必要的结点分裂调整,使T仍是m阶B树。
    BTree ap=NULL; int finished=0;
    KeyType x;      x=K;
    while (q && !finished){
        Insert(q,i,x,ap); //将x和ap分别插入到q->key[i+1]和q->ptr[i+1]
        if (q->keynum<M) finished=1; //插入完成
        else{
            int s=(M+1)/2;
            split(q,s,ap); // 分裂结点 *q
            x=q->key[s];
            //将q->key[s+1..m],q->ptr[s..m]和q->recptr[s +1..m]移入新结点 *ap
            q=q->parent;
            if (q) i=Search(q,x); //在双亲结点 *q中查找x的插入位置
        } //else
    } //while
    if (!finished) //T是空树(参数q初值为 NULL)或者根结点已分裂为结点 *q和*ap
        NewRoot(T,q,x,ap); //生成含信息(T,x,ap)的新的根结点*T,原T和ap为子树指针
    return 1;
} //InsertBTree
```

9.2.5 B-树和B+树

六、B-树的删除

1. 算法思想:

在B-树中删除一个关键字，则首先应找到该关键字所在结点，并从中删除之。

(1) 若所删关键字为非终端结点中的 K_i ，则以指针 A_i 所指子树中的最小关键字 Y 替代 K_i ，然后在相应的结点中删去 Y 。

9.2.5 B-树和B+树

六、B-树的删除

1. 算法思想:

(2) 若所删关键字在**最下层非终端结点**中。有下列3种情况:

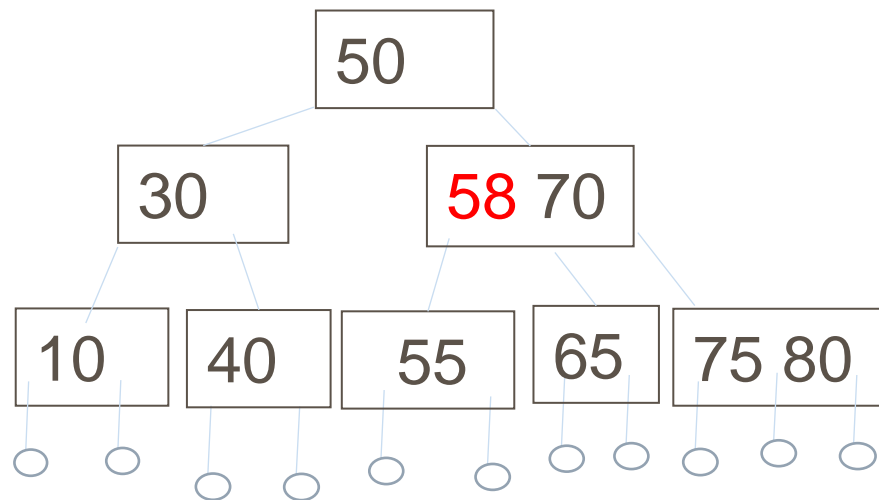
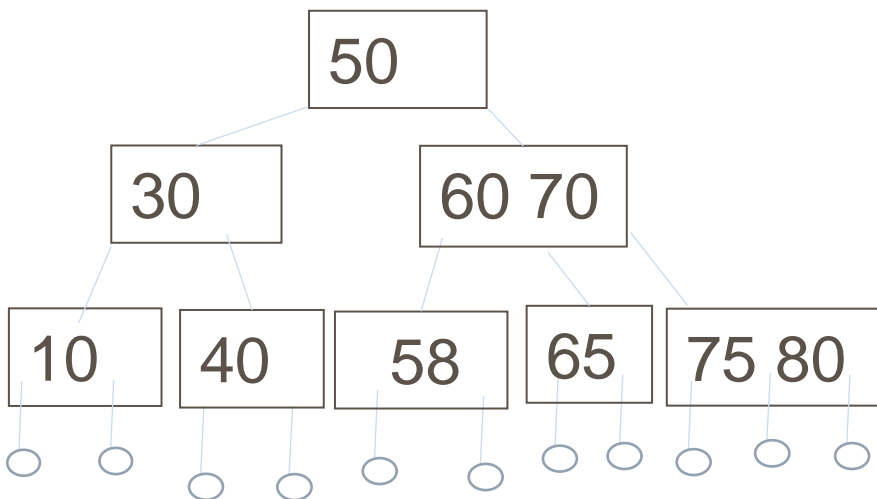
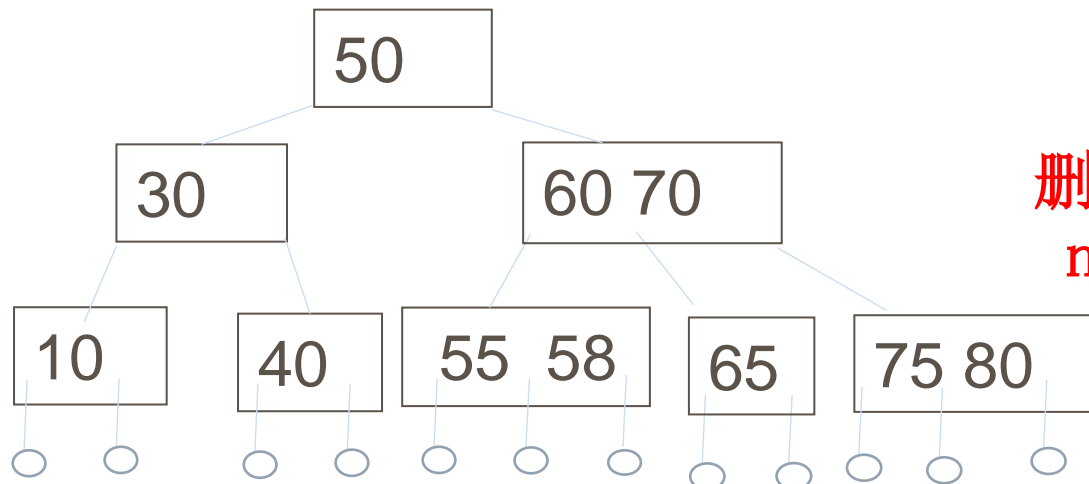
a. 被删关键字所在非终端结点不是根结点且删除前该结点中关键字个数 $n \geq \lceil m/2 \rceil$, 则直接删去该关键字 K_i 和相应的指针 A_i , 并将修改后的结点写回磁盘, 删除结束

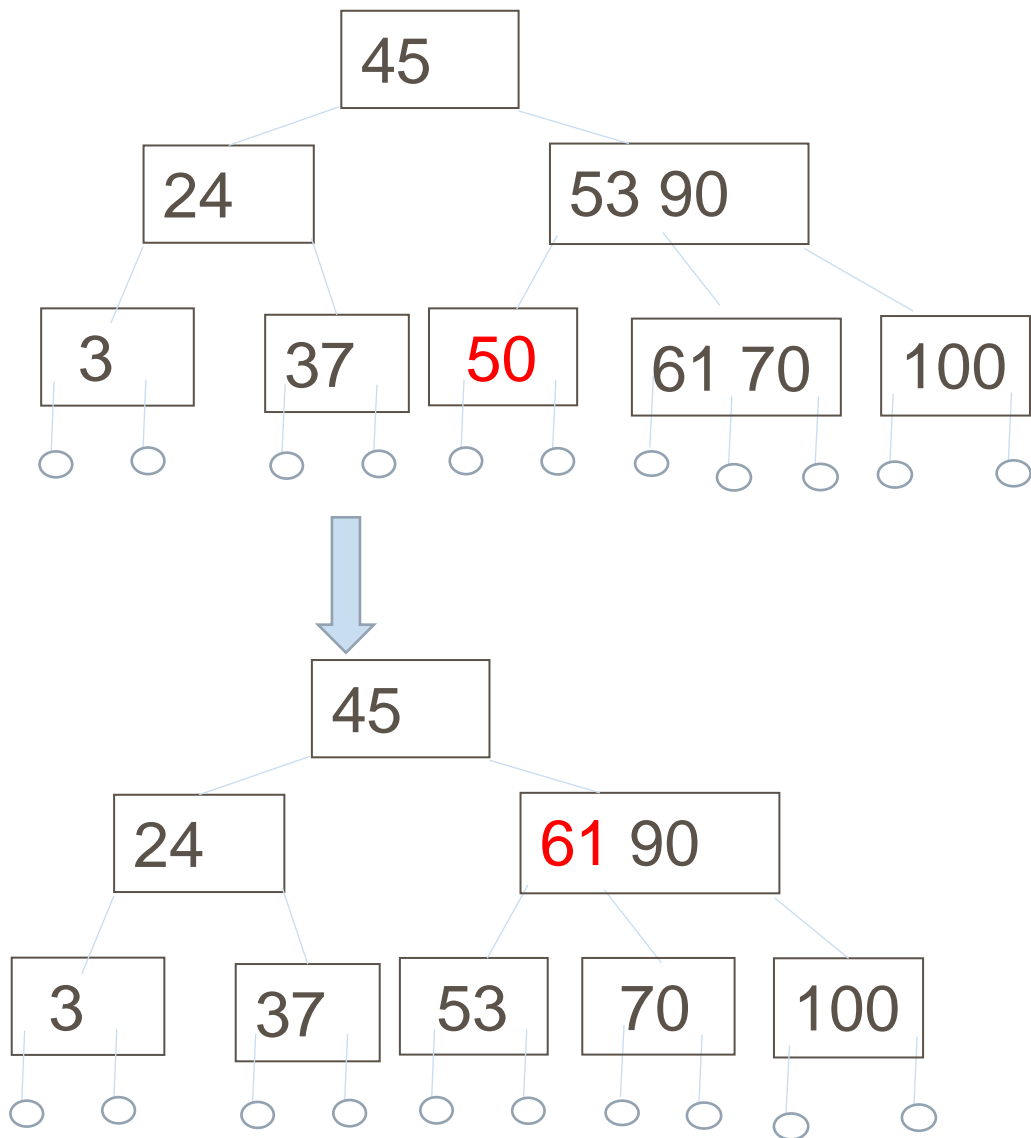
1. 简单删除

删除前该结点中关键字个数
 $n \geq \lceil m/2 \rceil$

例如：

1. 删除55
2. 删除60





2. 结点联合调整

若被删关键字所在结点
删除前关键字个数：

$n = \lceil m/2 \rceil - 1$, (下溢)

该结点相邻的右兄弟 (或左兄弟) 结点的关键
字个数：

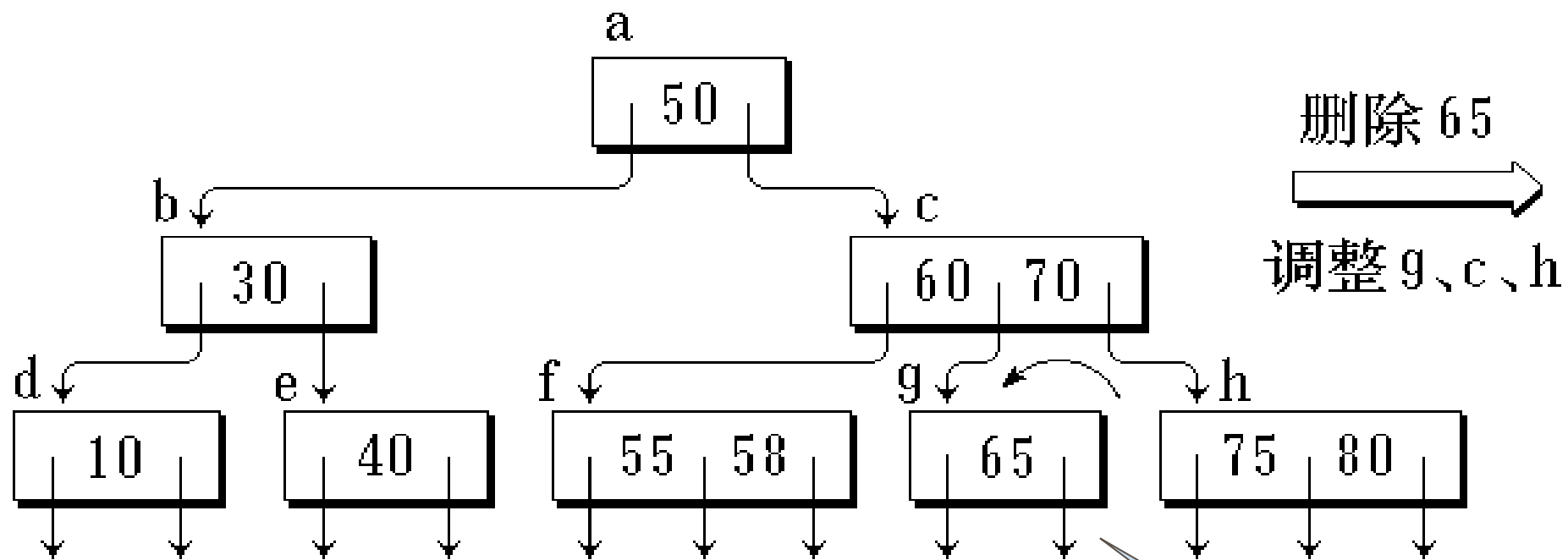
$n \geq \lceil m/2 \rceil$

办法：兄弟结点上移一个
关键码给双亲，双亲
下移一个相邻关键码到
溢出结点

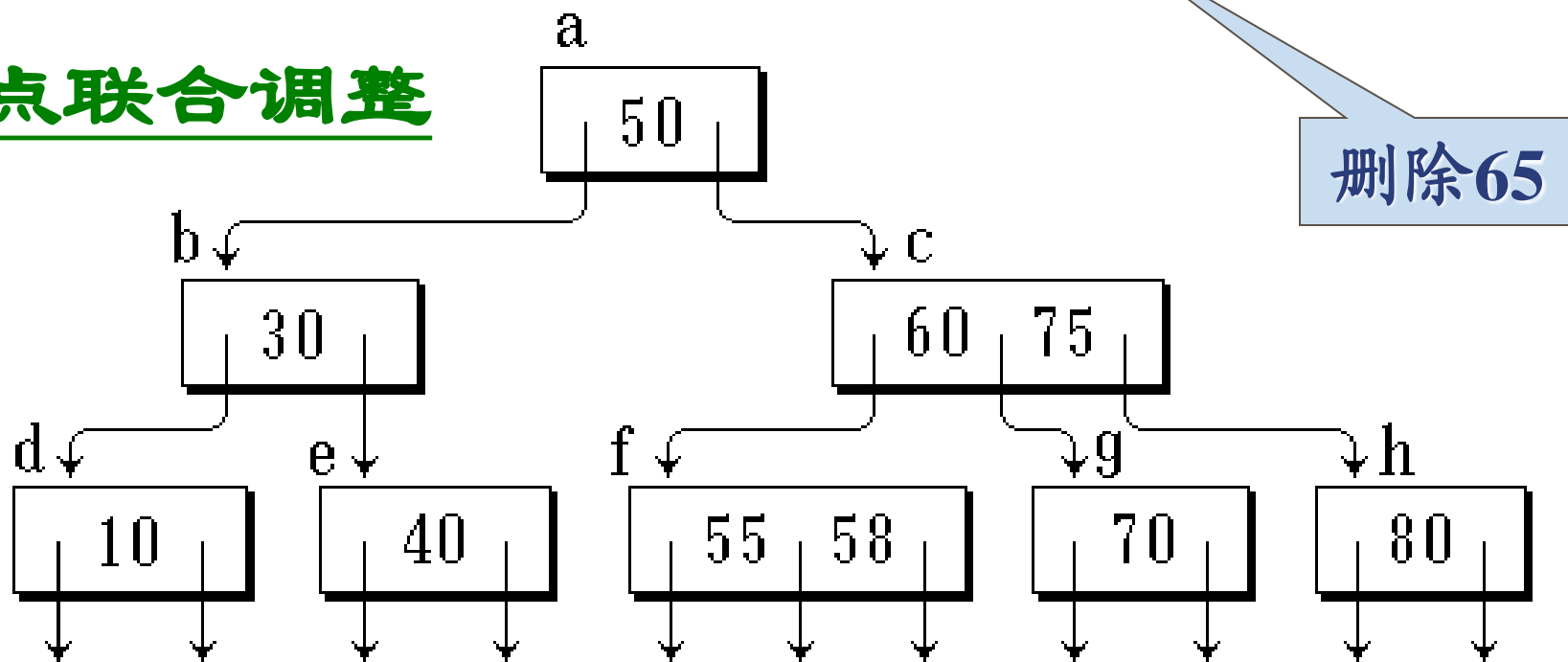
向兄弟“借”

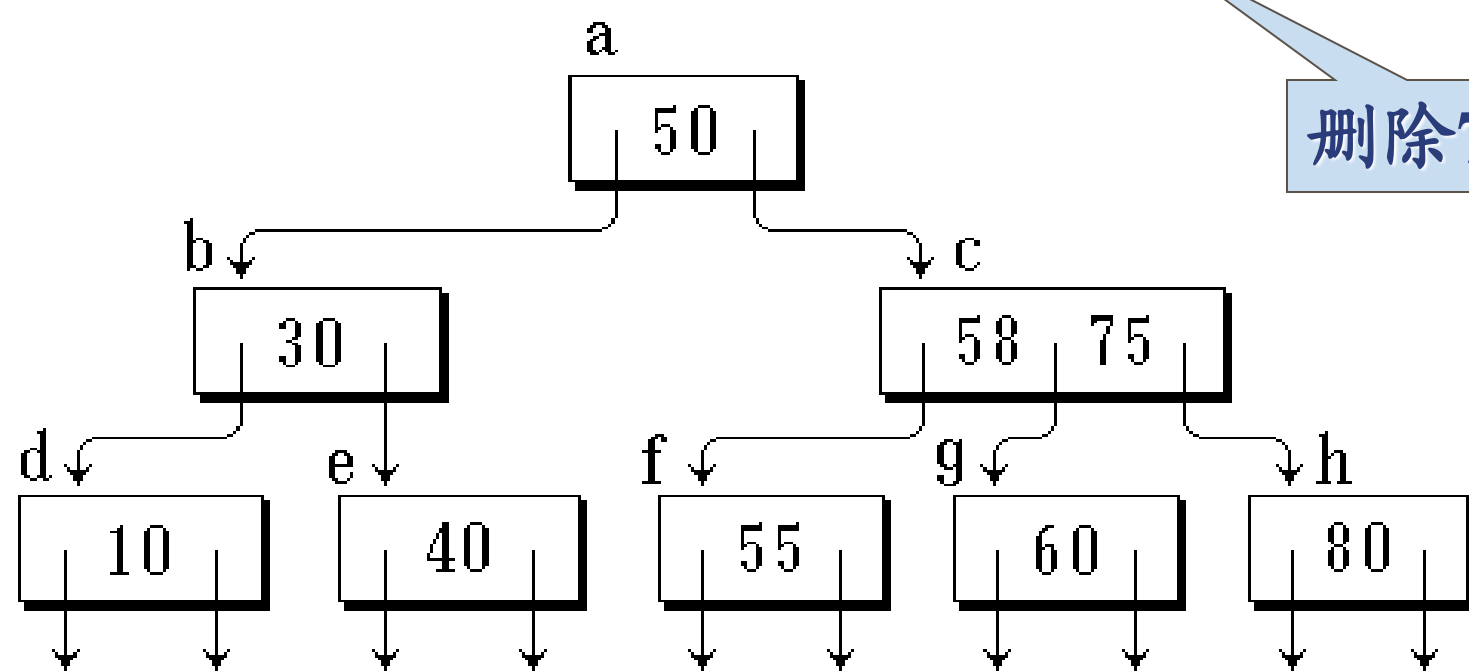
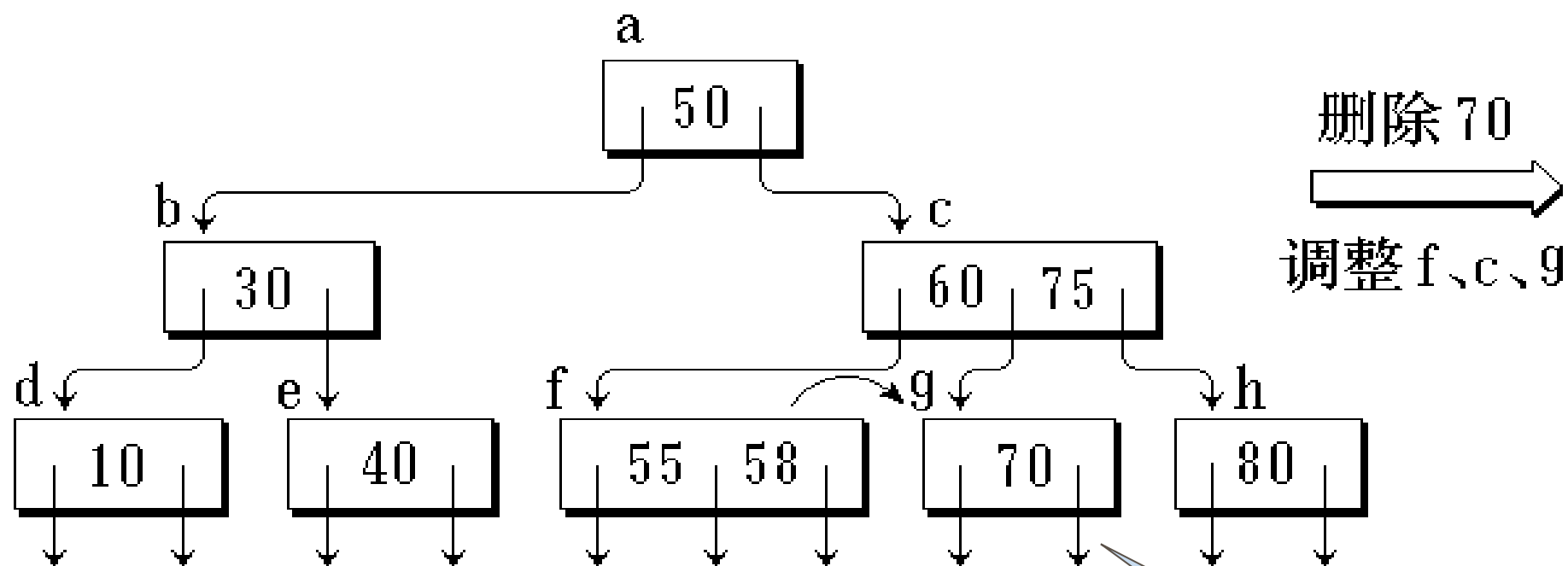
例如：

删除50



结点联合调整



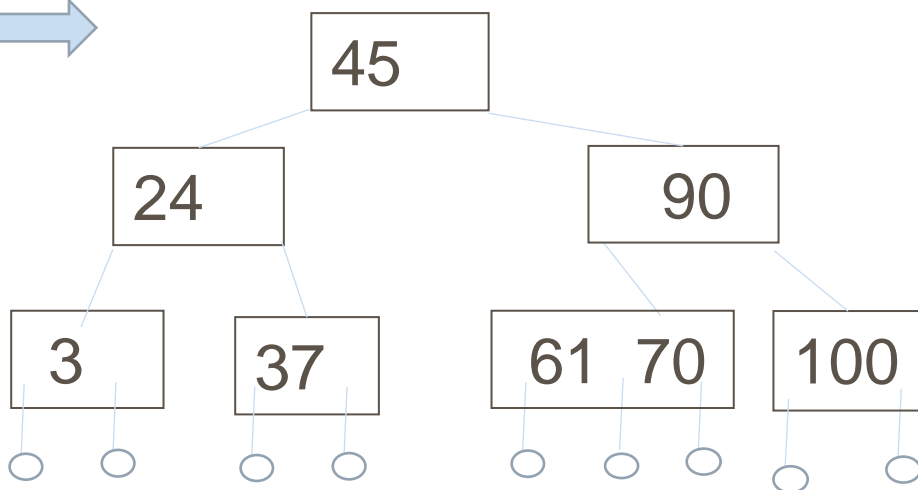


3. 结点合并

被删除关键字所在的结点和其相邻的兄弟结点中的关键字个数均等于 $\lceil m/2 \rceil - 1$

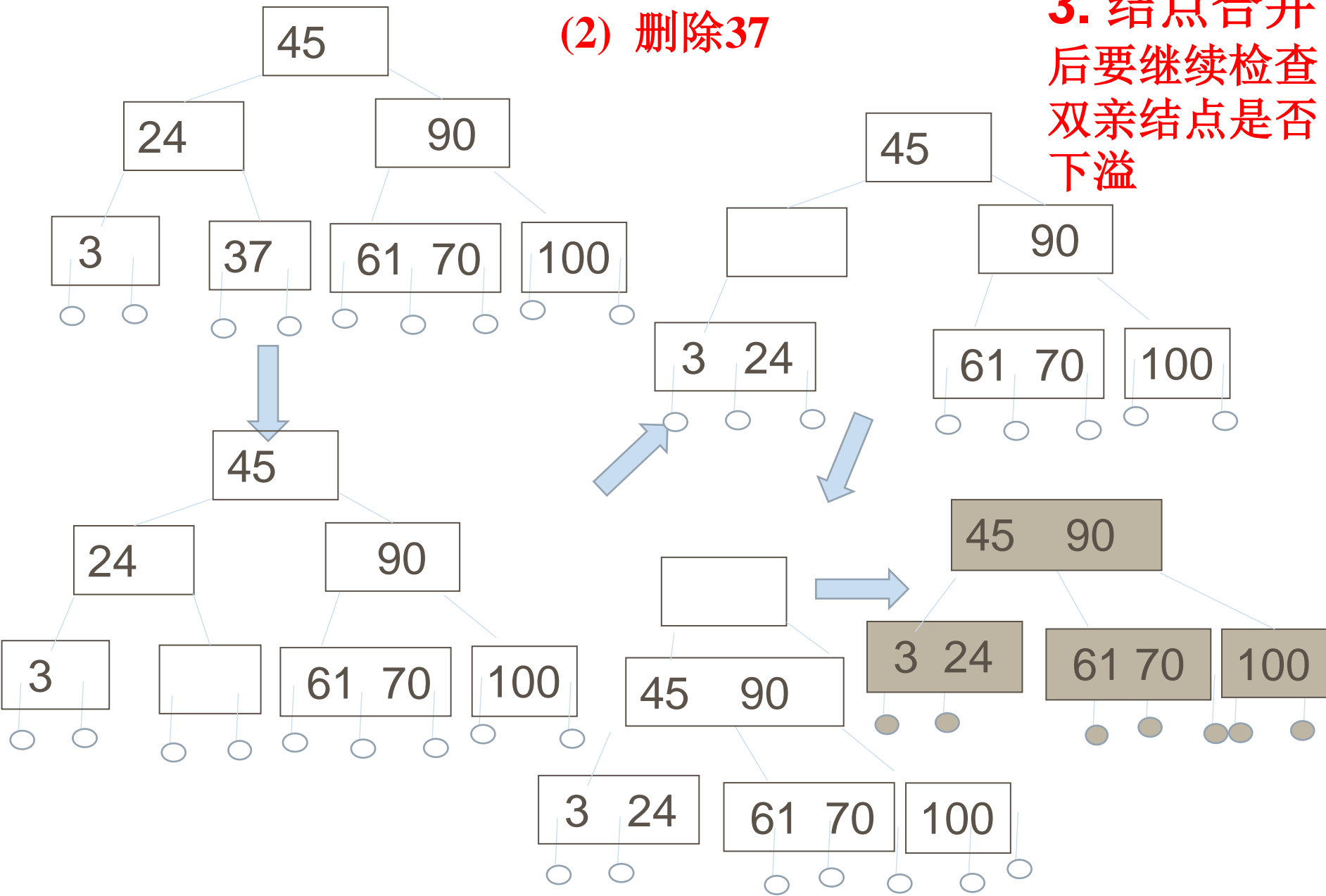
办法：从父结点中取出介于被删除结点和兄弟结点之间的关键字，连同被删结点剩余部分一起合并到兄弟结点中。

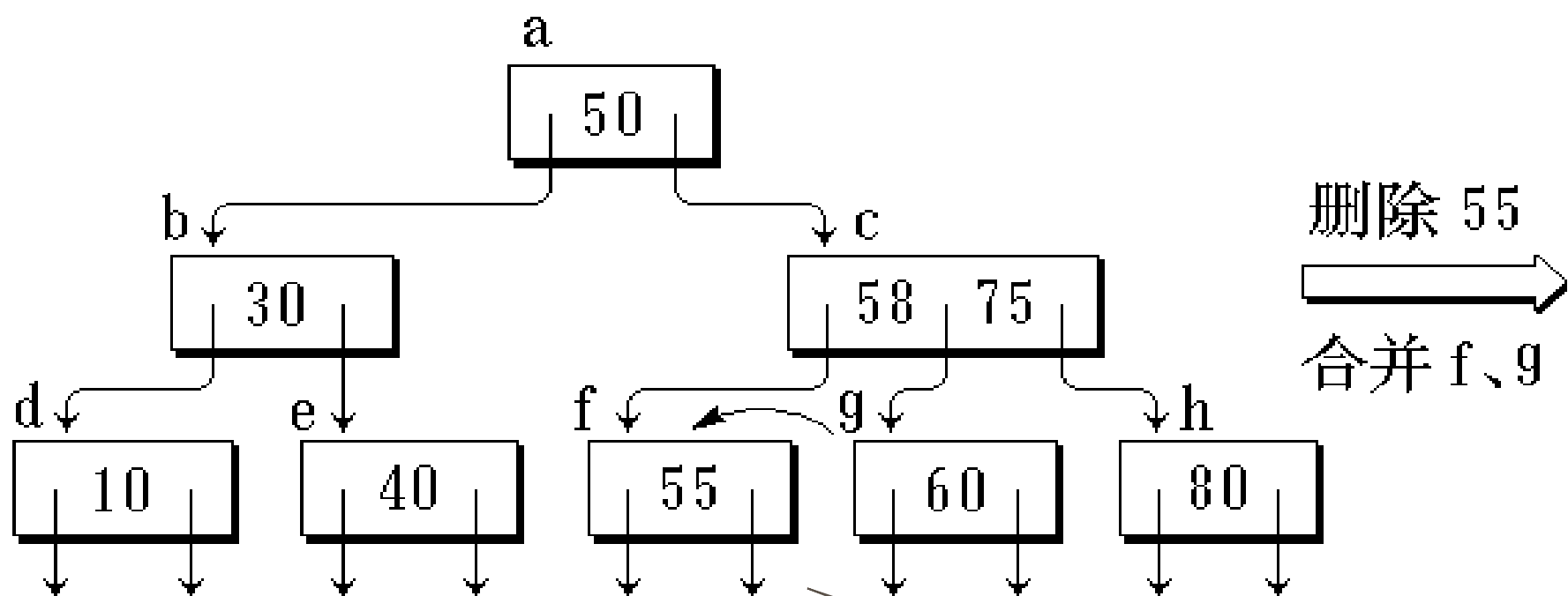
(1) 删除53



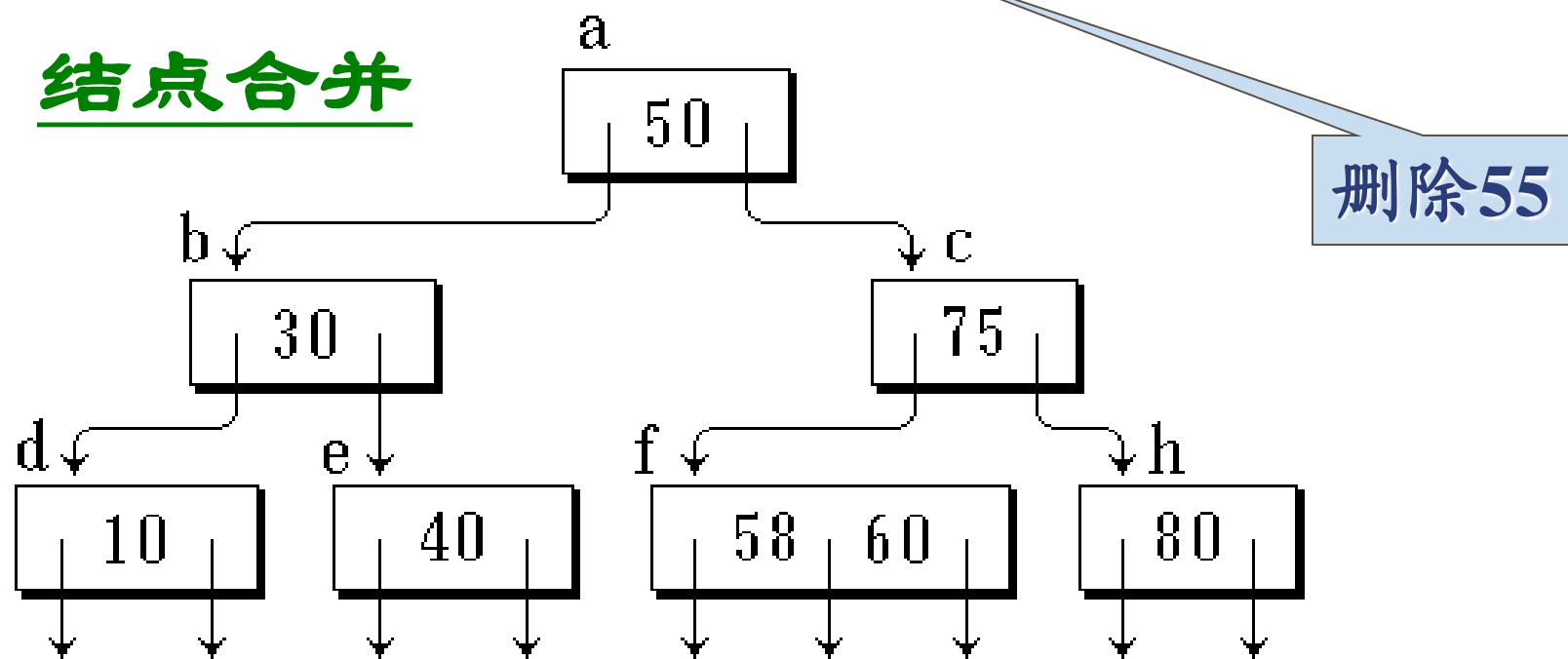
(2) 删除37

3. 结点合并
后要继续检查
双亲结点是否
下溢





结点合并



六、 B-树的删除

b. 被删关键字所在非终端结点删除前关键字个数 $n = \lceil m/2 \rceil - 1$ ，若这时与该结点相邻的右兄弟（或左兄弟）结点的关键字个数 $n \geq \lceil m/2 \rceil$ ：

则可按以下步骤调整该结点、右兄弟（或左兄弟）结点以及其双亲结点，以达到新的平衡。

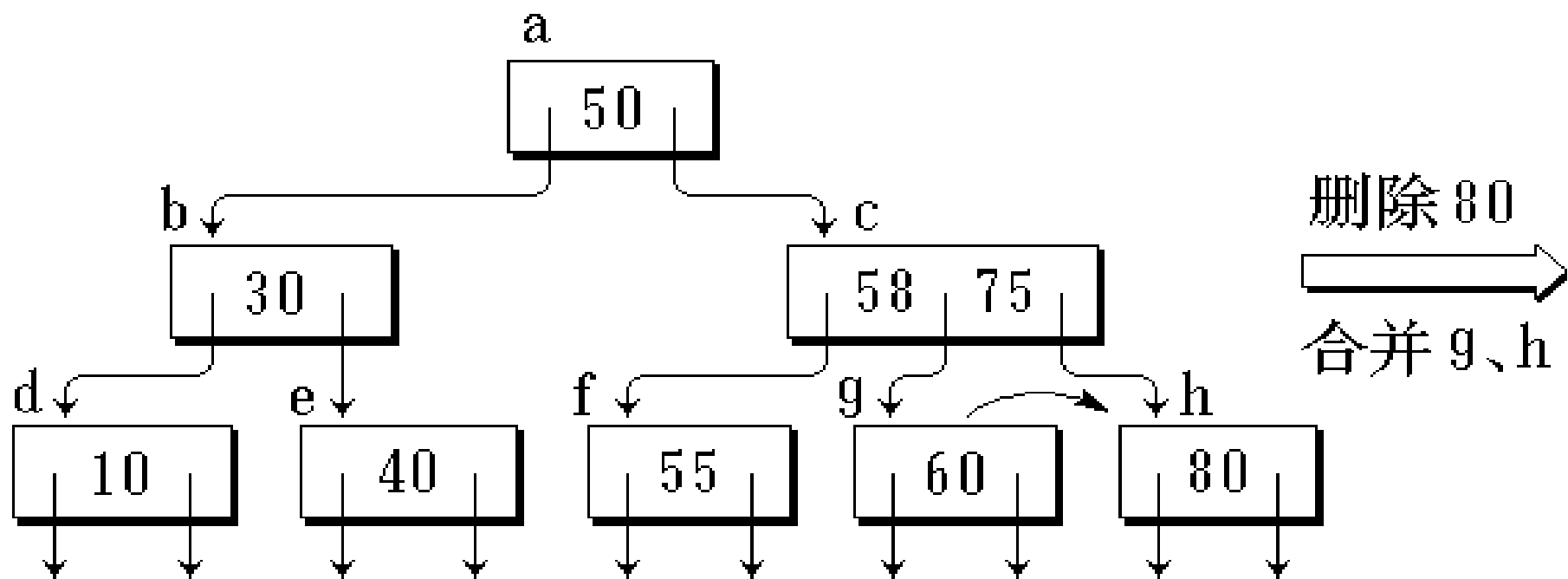
- 将右兄弟（或左兄弟）结点中的最小（或最大）关键字**上移**到双亲结点的 K_i 位置；
- 将双亲结点中刚刚大于（或小于）该被删关键字的关键字 K_i ($1 \leq i \leq n$) **下移**至被删关键字所在的结点中。

六、 B-树的删除

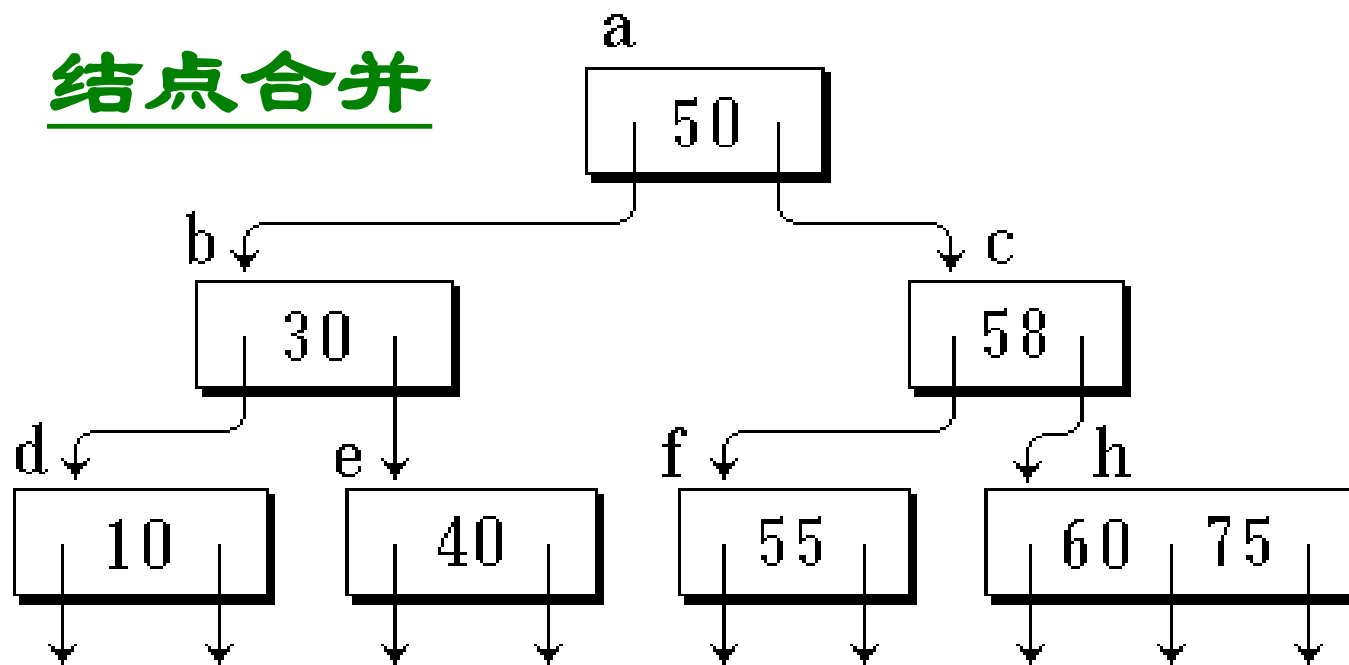
c. 被删除关键字所在的结点和其相邻的兄弟结点中的关键字个数均等于 $\lceil m/2 \rceil - 1$ 。

若该结点有右兄弟，且其右兄弟结点地址由双亲结点中的指针 A_i 所指，则在删除关键字之后，它所在结点中剩余的关键字和指针，加上双亲结点中的关键字 K_i 一起，合并到 A_i 所指的兄弟结点中（若没有右兄弟，则合并到左兄弟结点中）

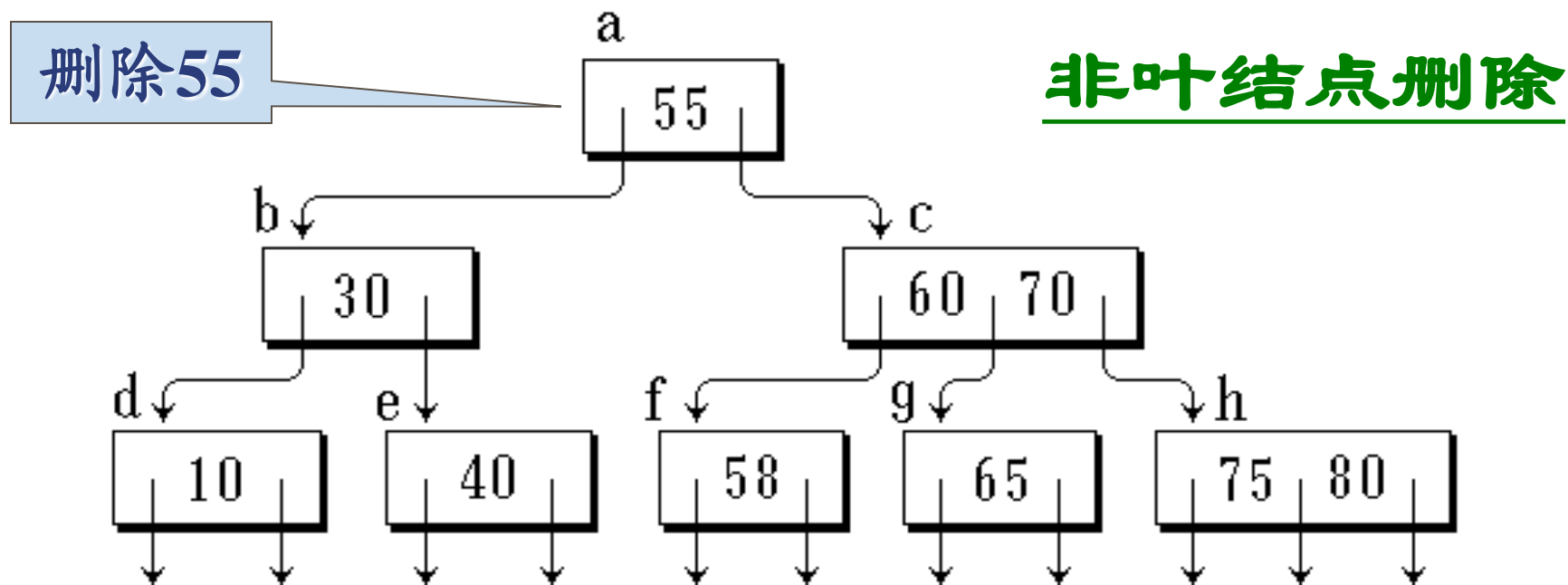
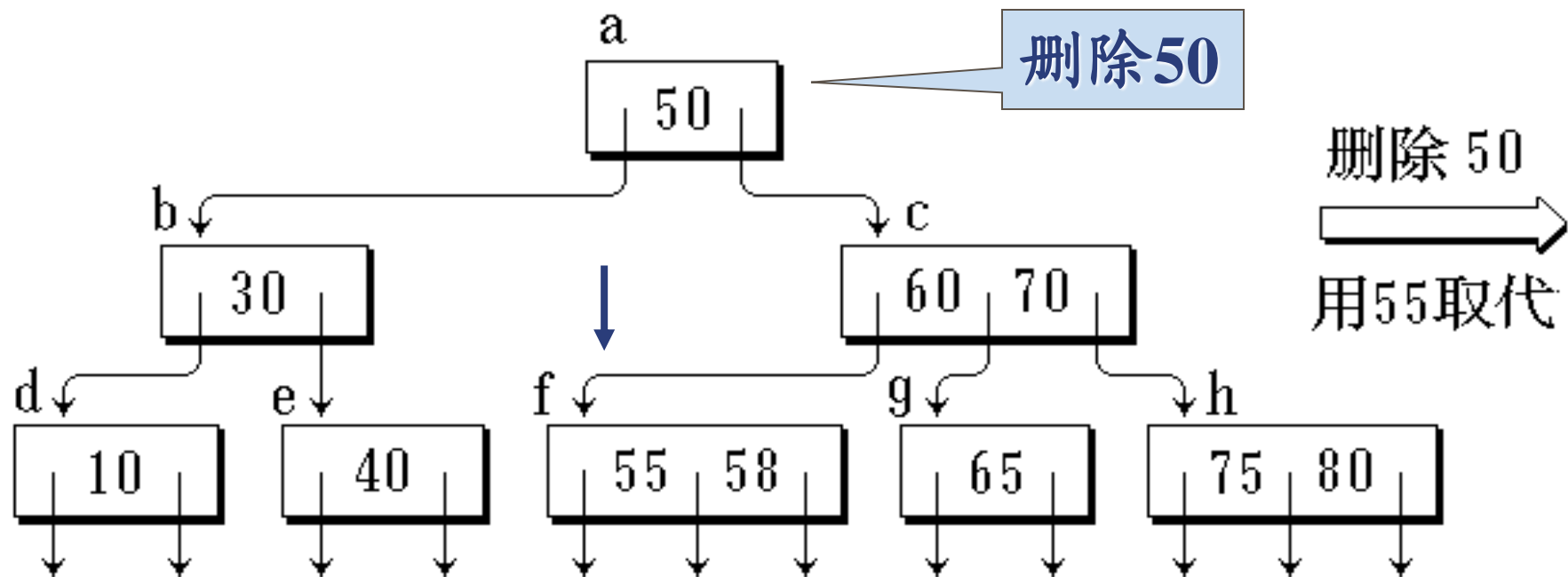
如果因此导致双亲结点中的关键字数目小于 $\lceil m/2 \rceil - 1$ ，则以此类推，做相应的处理。



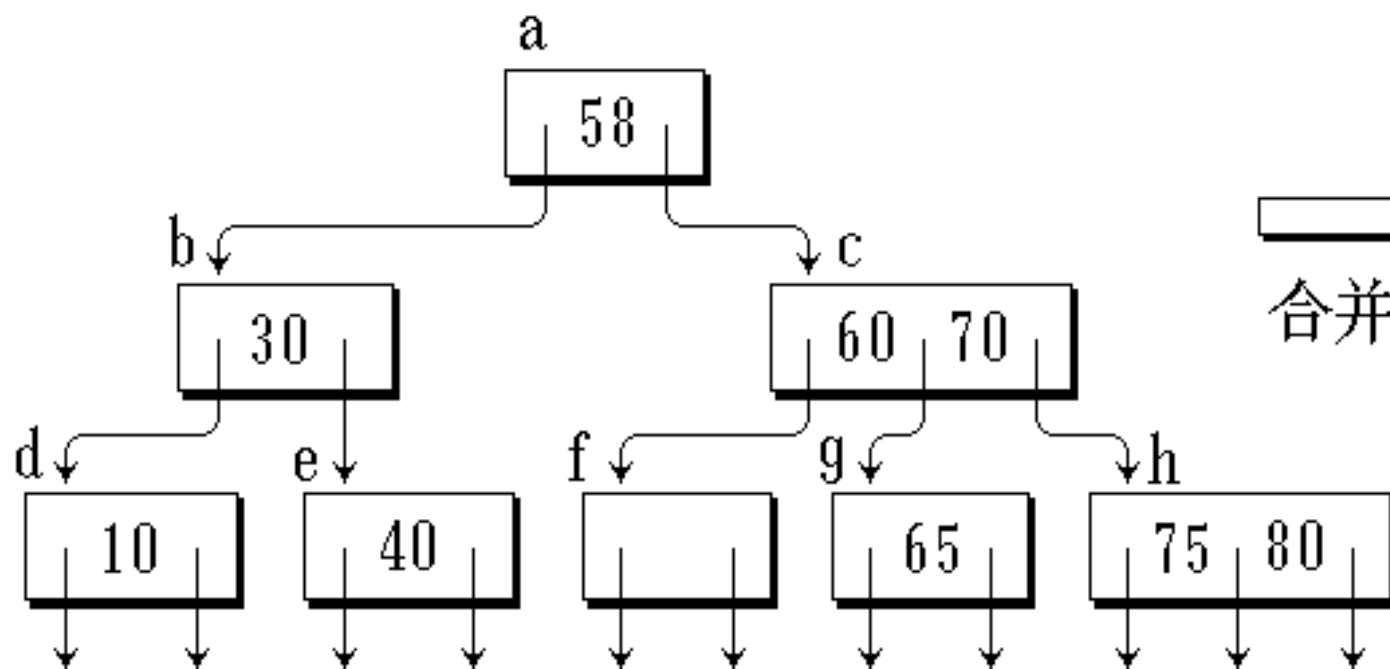
结点合并



删除 80

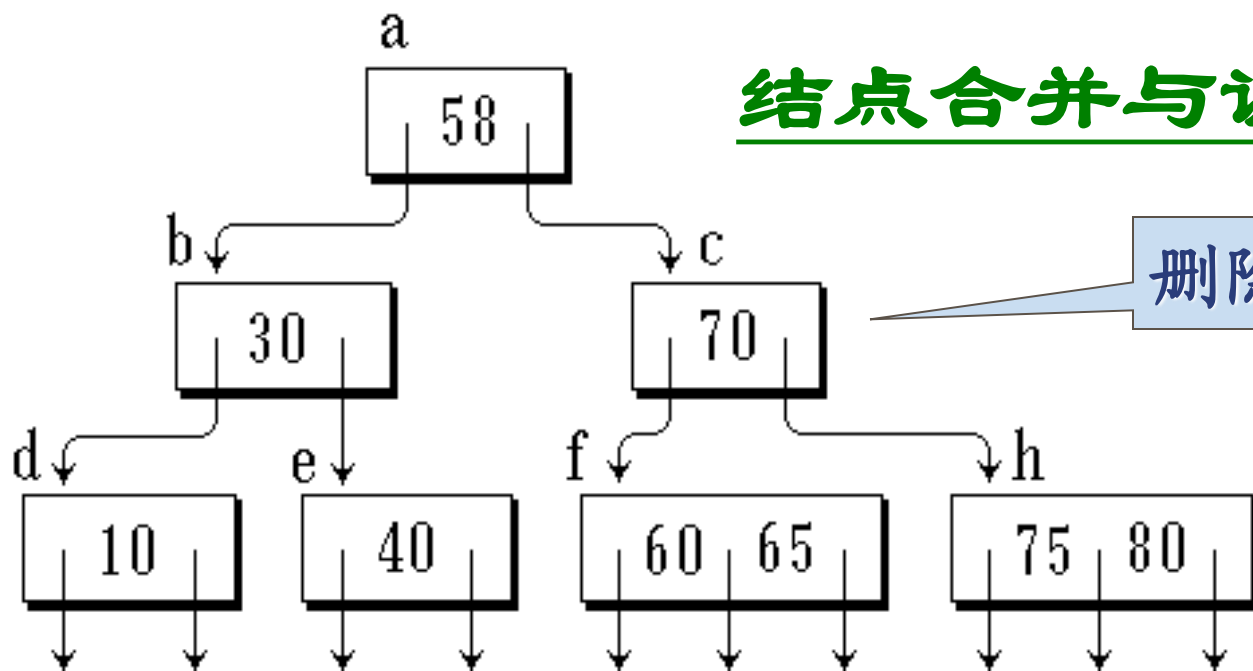


删除 55
用 58 取代



合并 f、g

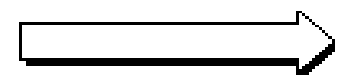
合并 f、g



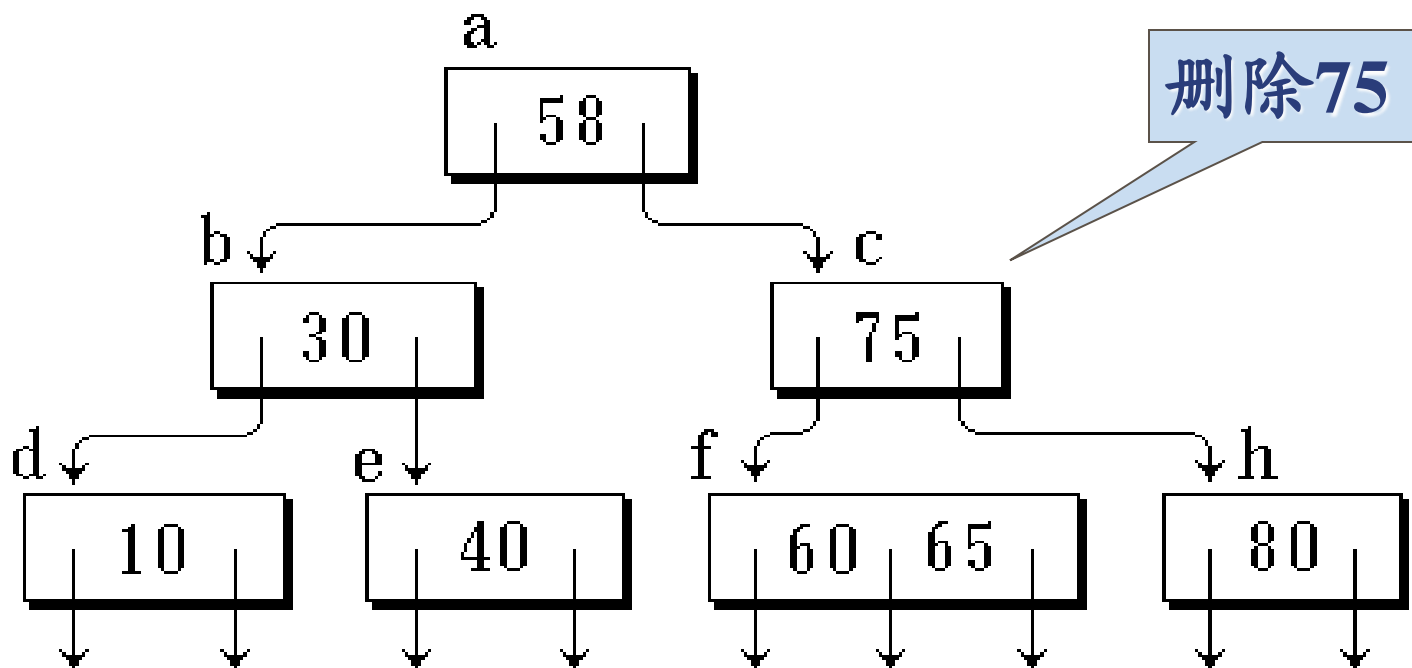
结点合并与调整

删除 70

删除 70



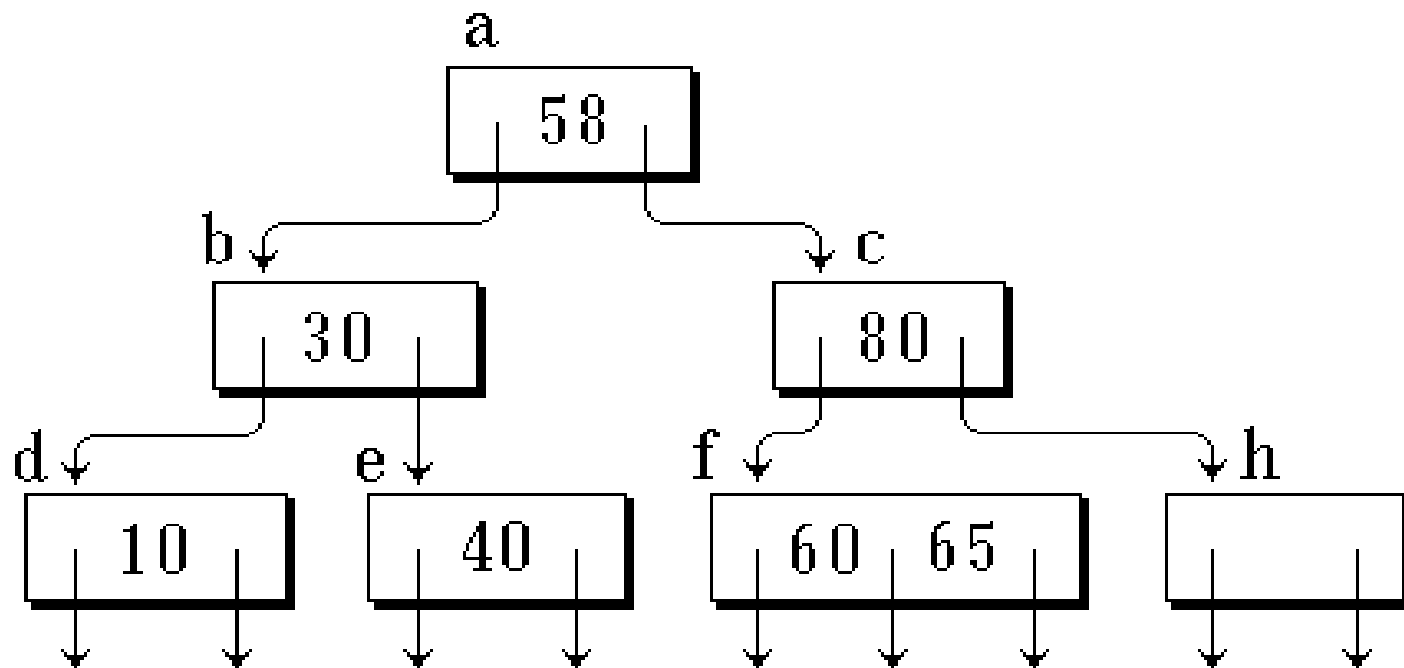
用 75 取代



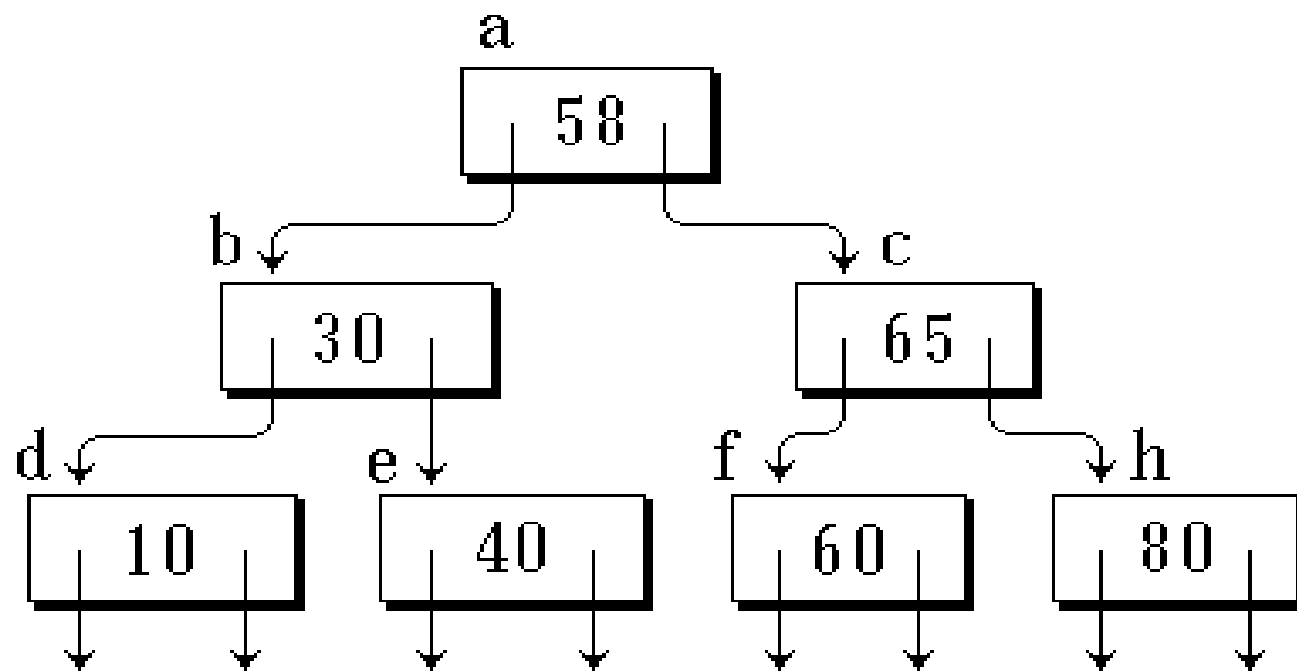
删除 75



用 80 取代

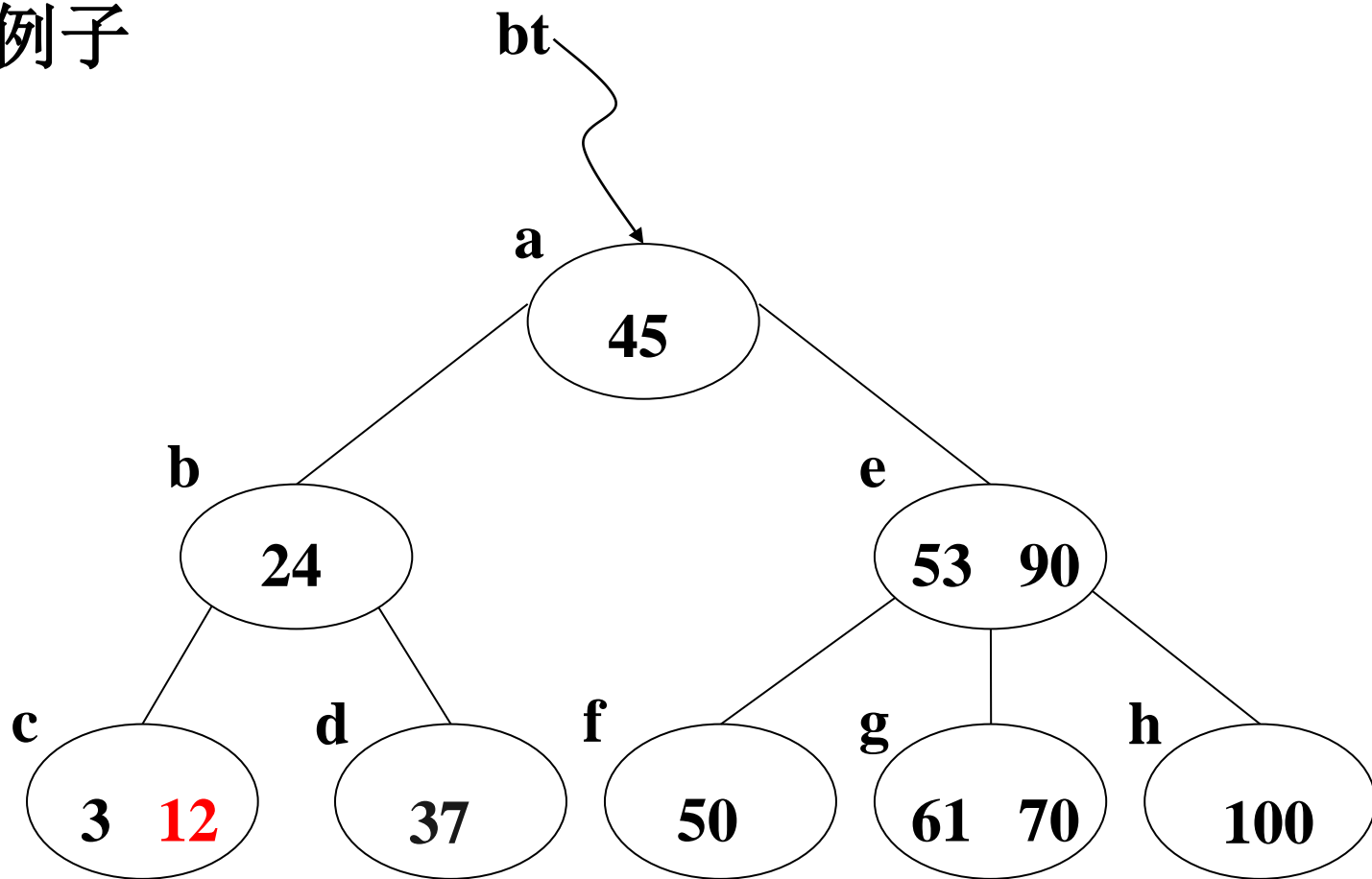


调整 f、c、h



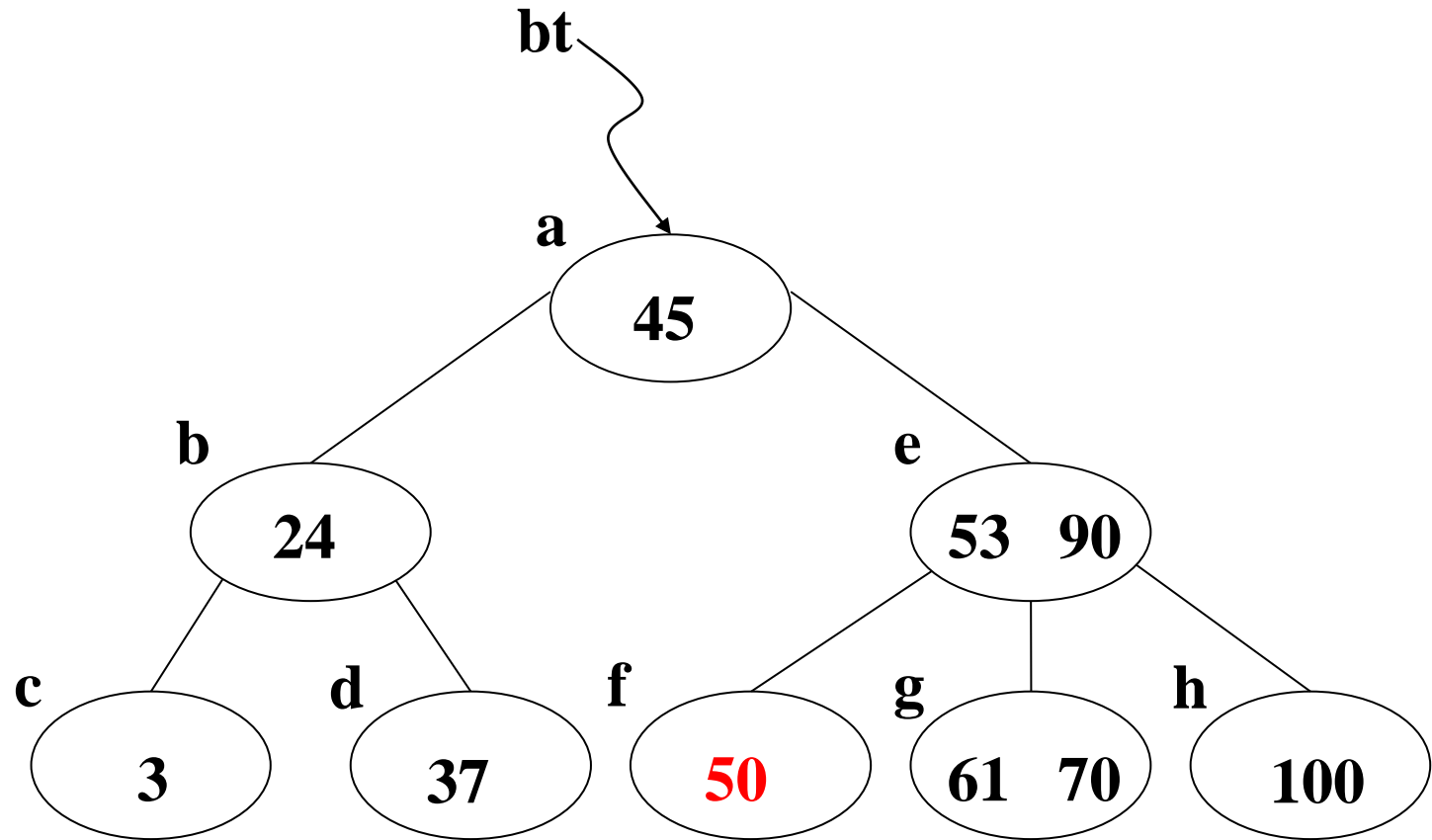
六、 B-树的删除

2. 例子



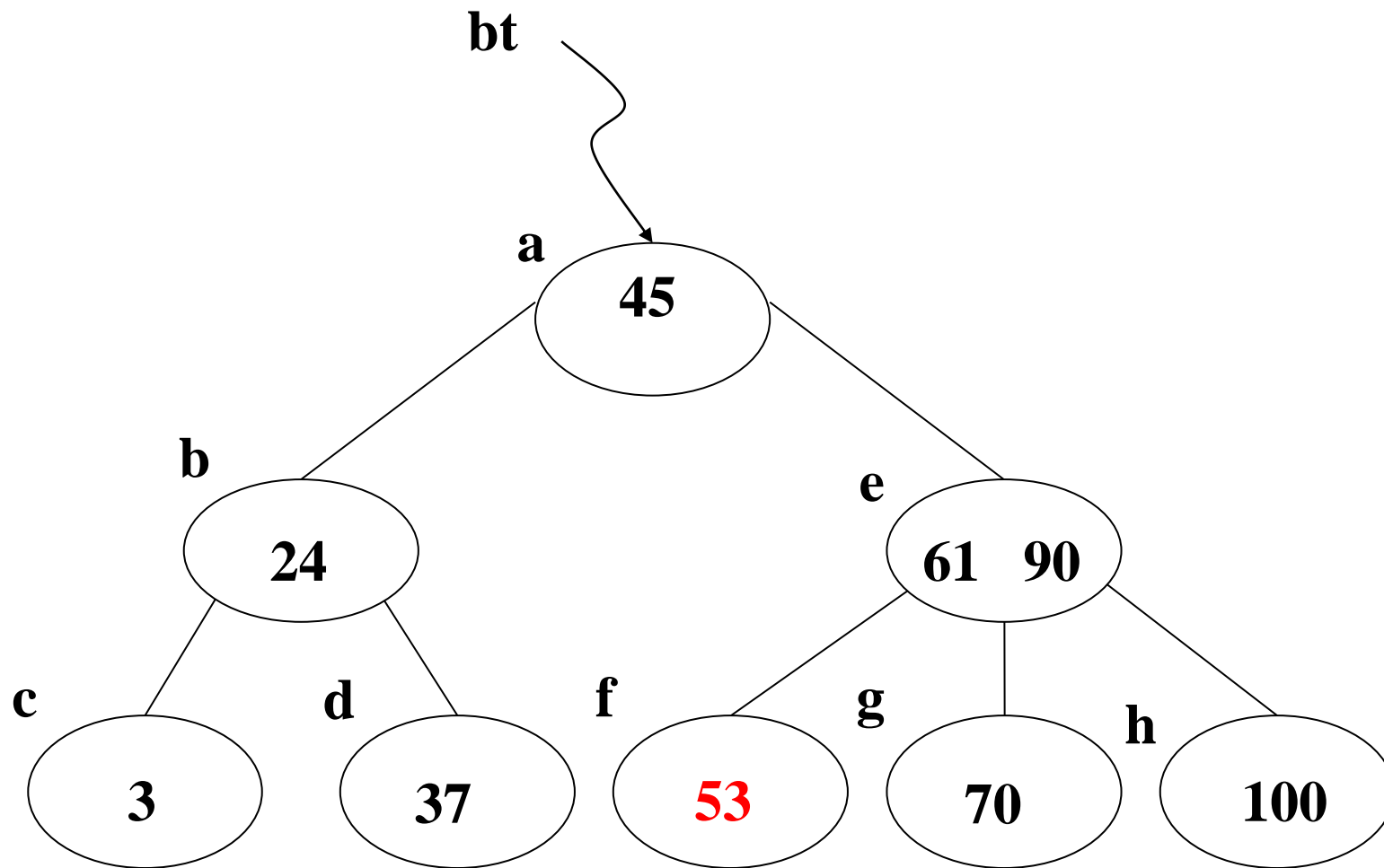
删去关键字12

六、 B-树的删除



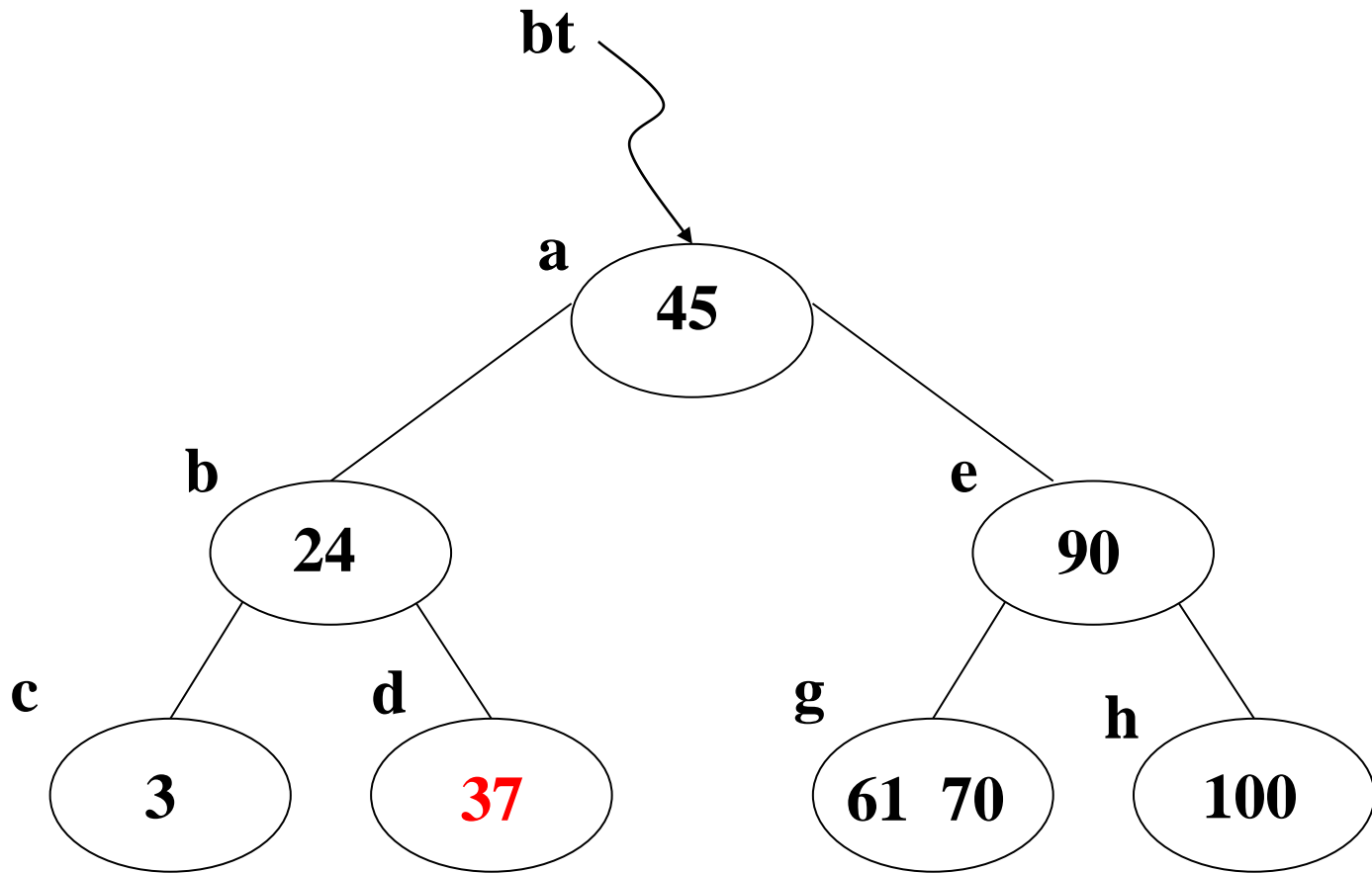
删去50

六、 B-树的删除



删去53

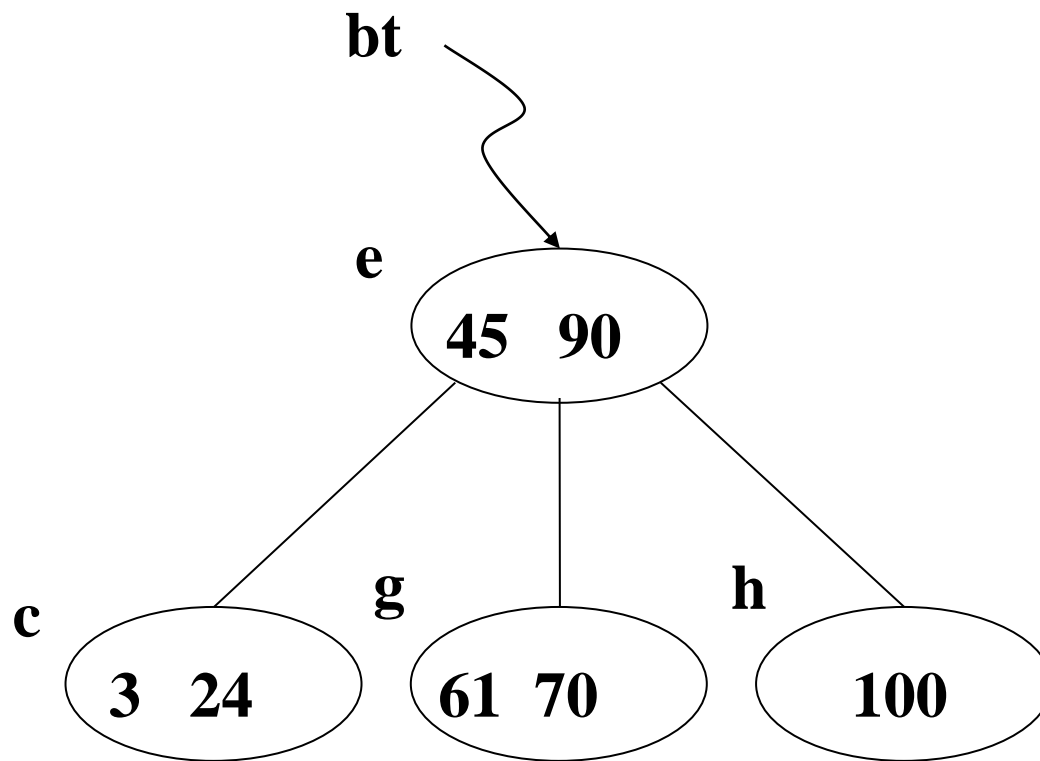
六、 B-树的删除



删去37

(c)

六、 B-树的删除



(d)

在B-树中删除关键字的情形

9.2.5 B-树和B+树

一、定义

B^+ 树是应文件系统所需而出的一种B-树的变型树。

一棵 m 阶的 B^+ 树和 m 阶的B-树的差异在于：

(1) 每个结点最多 m 棵子树，每个结点中最多含有 m 个关键字；

9.2.5 B-树和B+树

(2) 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

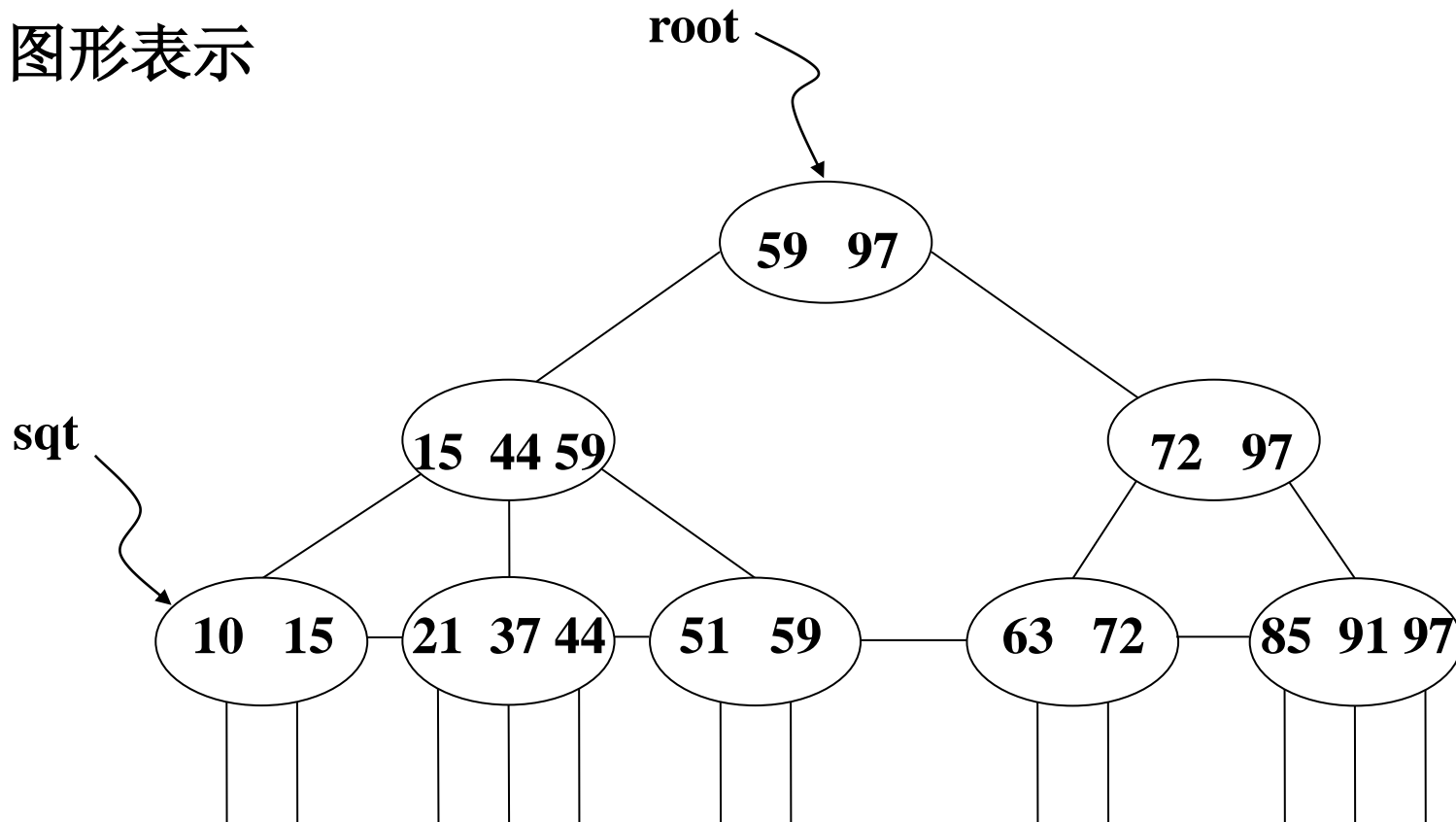
(3) 所有的非终端结点可以看成是索引部分，结点仅含其子树中的最大（或最小）关键字。

9.2.5 B-树和B+树

通常在B⁺树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。

9.2.5 B-树和B+树

二、图形表示



一棵3阶的B⁺树

9.2.5 B-树和B+树

三、B⁺树的查找

对B⁺树可以进行两种查找运算：

- 1.从最小关键字起顺序查找；
- 2.从根结点开始，进行随机查找。

在查找时，若非终端结点上的索引值等于给定值，并不终止，而是继续向下直到叶子结点。因此，在B⁺树中，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。其余同B-树的查找类似。

9.2.5 B-树和B+树

四、B⁺树的插入

B⁺树的插入仅在叶子结点上进行，当结点中的关键字个数大于m时要分裂成两个结点，它们所含关键字的个数分别为 $\left\lceil \frac{m+1}{2} \right\rceil$ 和 $\left\lfloor \frac{m}{2} \right\rfloor$ 。并且，它们的双亲结点中应同时包含这两个结点中的最大关键字。其余同B-树的插入类似。

9.2.5 B-树和B+树

五、B⁺树的删除

B⁺树的删除也仅在叶子结点进行，当叶子结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个“分界关键字”存在。若因删除而使结点中关键字的个数少于 $\left\lceil \frac{m}{2} \right\rceil$ 时，其和兄弟结点的合并过程亦和B-树类似。