



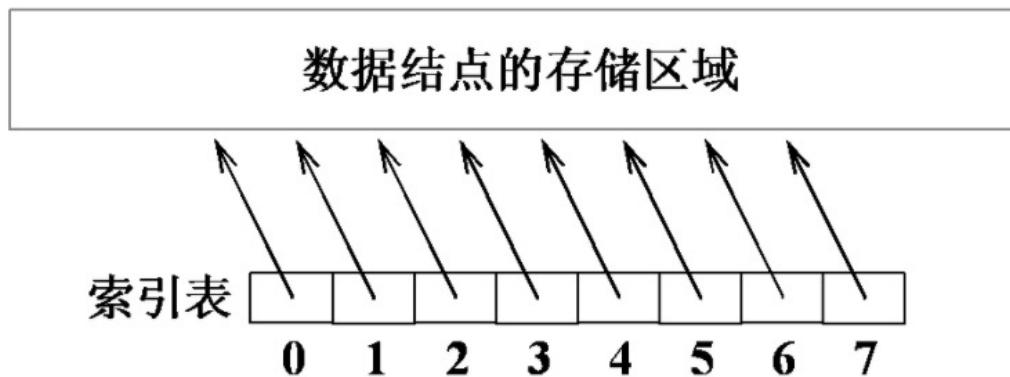
第十章 多级索引结构

- 多级索引结构
 - B 树
 - B+ 树 (自学)



索引结构

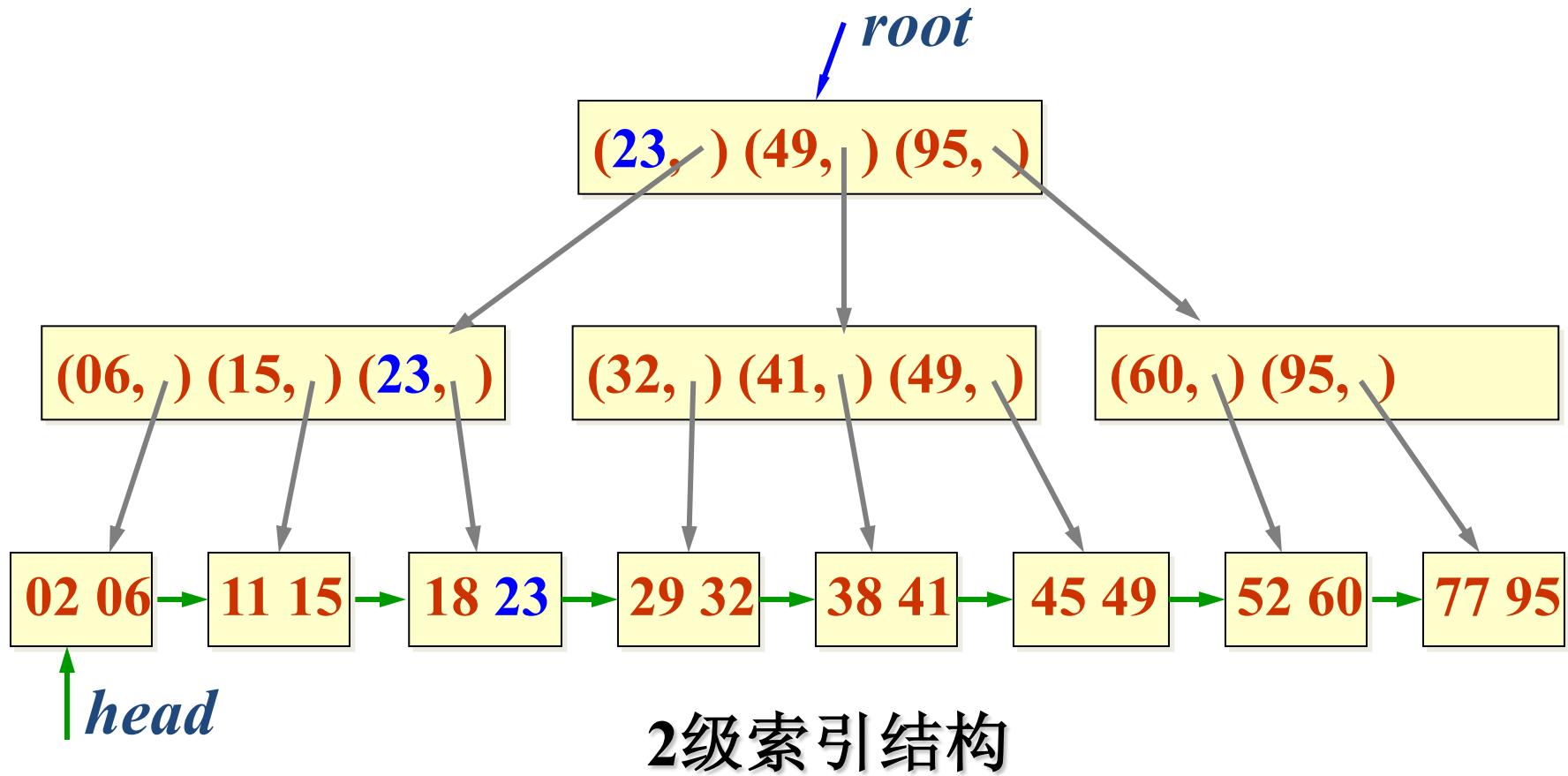
索引：建造一个由整数域Z映射到存储地址域D的函数Y： $Z \rightarrow D$





多级索引结构

- 当数据记录数目特别大，索引表本身也很大，在内存中放不下，需要分批多次读取外存才能把索引表搜索一遍。
- 此时，可以建立索引的索引（二级索引）。二级索引可以常驻内存，二级索引中一个索引项对应一个索引块，登记该索引块的最大关键码及该索引块的存储地址。
- 如果二级索引在内存中也放不下，需要分为许多块多次从外存读入。可以建立二级索引的索引（三级索引）。这时，访问外存次数等于读入索引次数再加上1次读取记录。



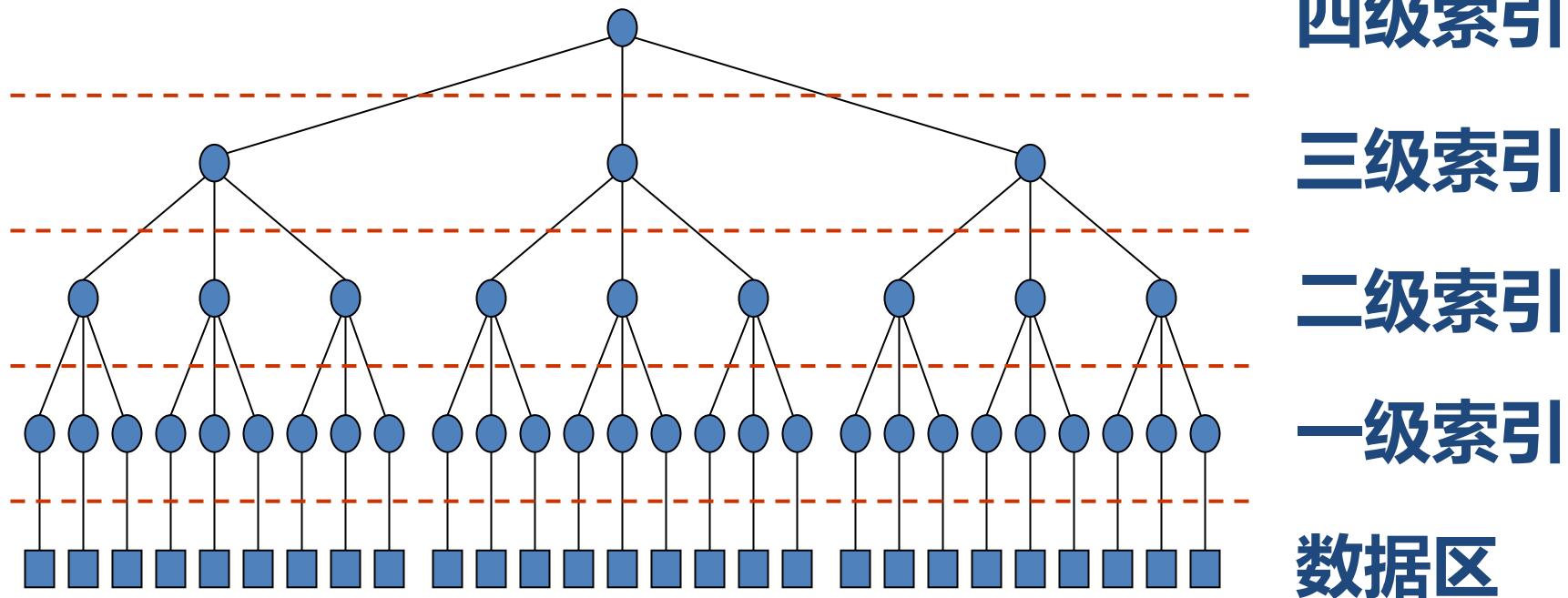
必要时, 还可以有4级索引, 5级索引, ...。



- 这种多级索引结构形成 m 叉树。树中每一个分支结点表示一个索引块，它**最多存放 m 个索引项**，每个索引项分别给出各子树结点(低一级索引块)的**最大关键码**和结点地址。
- 树的叶结点中各索引项给出在数据表中存放的记录的关键码和存放地址。这种 m 叉树用来作为多级索引，就是 **m 路搜索树**。
- **m 路搜索树可能是静态索引结构**，即结构在初始创建，数据装入时就已经定型，在整个运行期间，树的结构不发生变化。



- **m** 路搜索树还可能是**动态索引结构**, 即在整个系统运行期间, 树的结构随数据的增删及时调整, 以保持最佳的搜索效率。



多级索引结构形成 m 路搜索树



- 静态 m 路搜索树：结构在初始创建，数据装入时就已经定型，在整个运行期间，树的结构不发生变化。
- 动态 m 路搜索树：在整个系统运行期间，树的结构随数据的增删及时调整，以保持最佳的搜索效率。

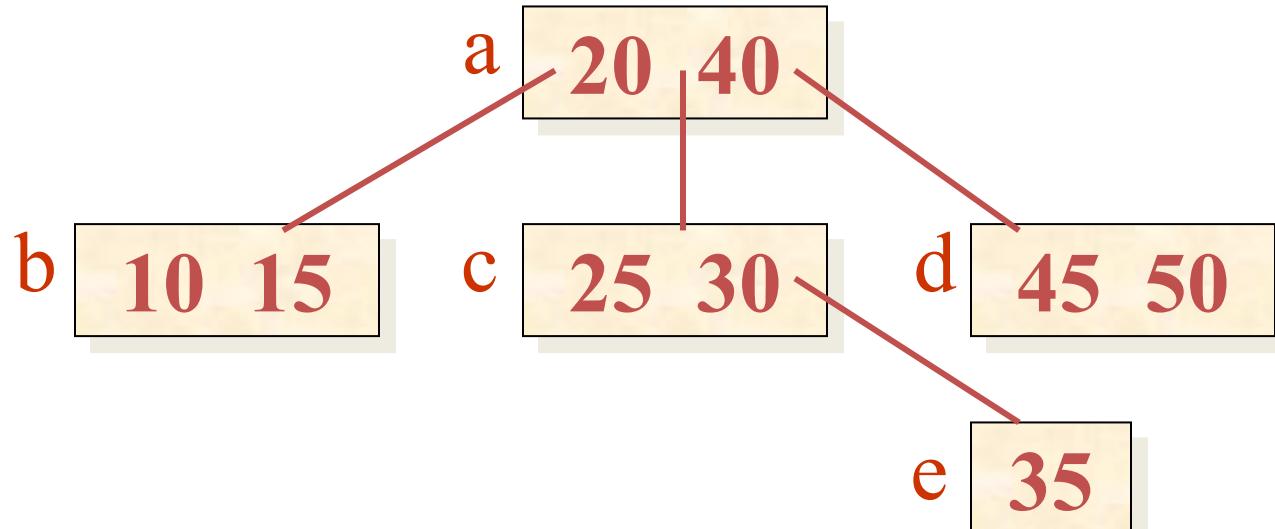


• 动态的 m 路搜索树（递归定义）：
它或者是一棵空树，或者是满足如下性质的树：

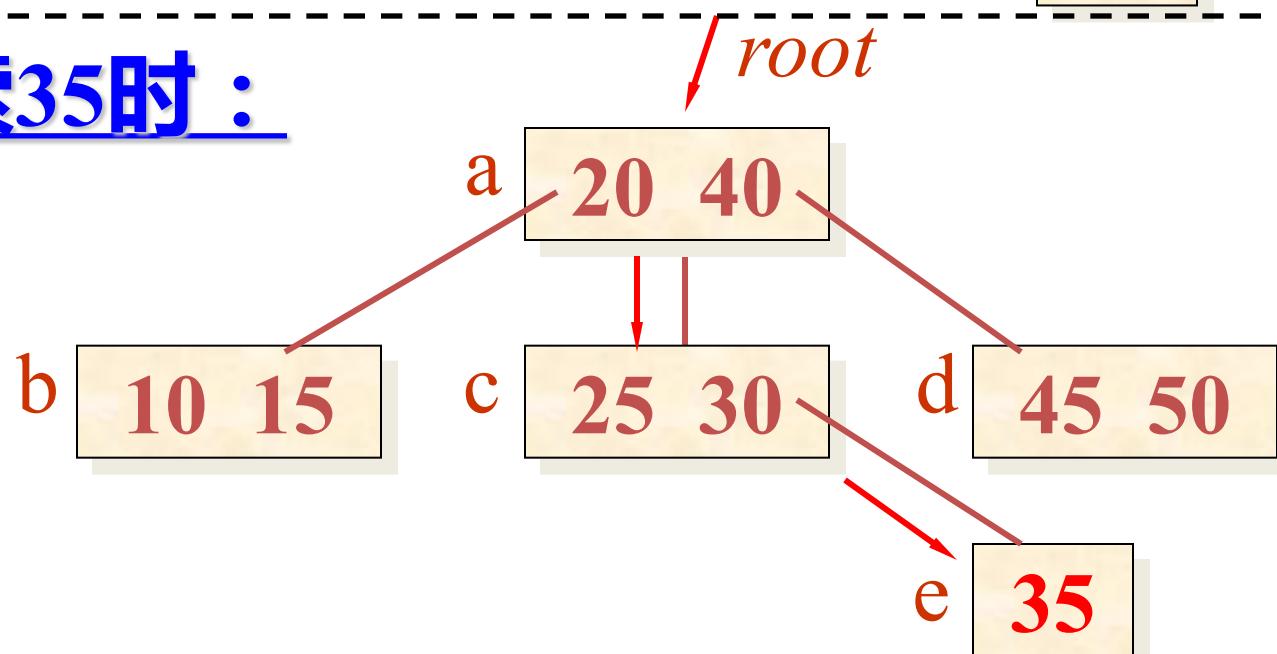
- ◆ 根最多有 m 棵子树，并具有如下的结构：
 $(n, P_0, K_1, P_1, K_2, P_2, \dots, K_n, P_n)$
其中， P_i 是指向子树的指针， K_i 是关键码，
 $K_i < K_{i+1}$, $0 \leq i \leq n < m$ ；
- ◆ 在子树 P_i 中所有的关键码都小于 K_{i+1} ，且大于 K_i ，
- ◆ 在子树 P_n 中所有的关键码都大于 K_n ；
- ◆ 在子树 P_0 中的所有关键码都小于 K_1 。
- ◆ 子树 P_i 也是 m 路搜索树， $0 \leq i \leq n$ 。



例：一棵3路搜索树



搜索35时：





- 提高搜索树的路数 m , 可以改善树的搜索性能。对于给定的关键码数 n , 如果搜索树是平衡的, 可以使 m 路搜索树的性能接近最佳。下面将讨论一种称之为**B树**的平衡的 m 路搜索树。



B树的定义

- 一棵 m 阶B 树或者是空树, 或者是满足下列性质的树:

- ① 根结点至少有 2 个子女
- ② (非根结点)至少有 $\lceil m/2 \rceil$ 个子女
- ③ (非根结点)最多有 m 个子女
- ④ 每个结点有如下结构

n	P_0	K_1	P_1	K_2	K_n	P_n
-----	-------	-------	-------	-------	-------	-------	-------

- P_i 是指向子树的指针, K_i 是关键码,

- $K_i < K_{i+1}$, $0 \leq i \leq n < m$;

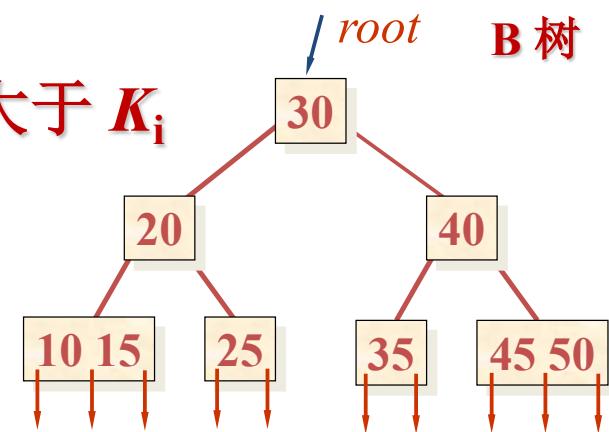
- P_i 中所有的关键码都小于 K_{i+1} , 且大于 K_i

- P_0 中的所有关键码都小于 K_1

- P_n 中所有的关键码都大于 K_n

- ⑤ 所有叶子结点都位于同一层

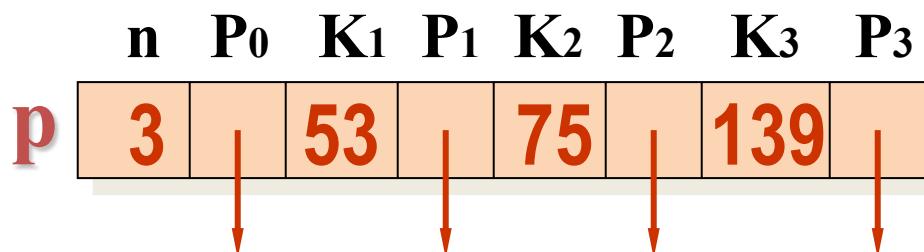
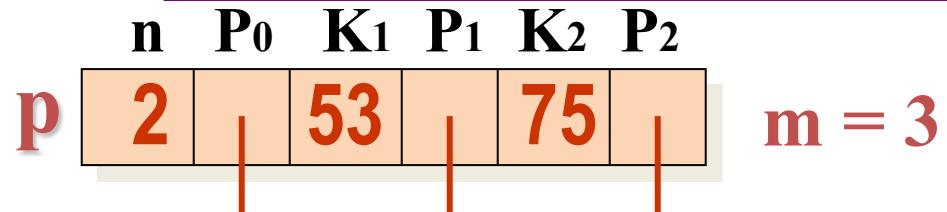
- ⑥ 所有子树 P_i 满足B树定义





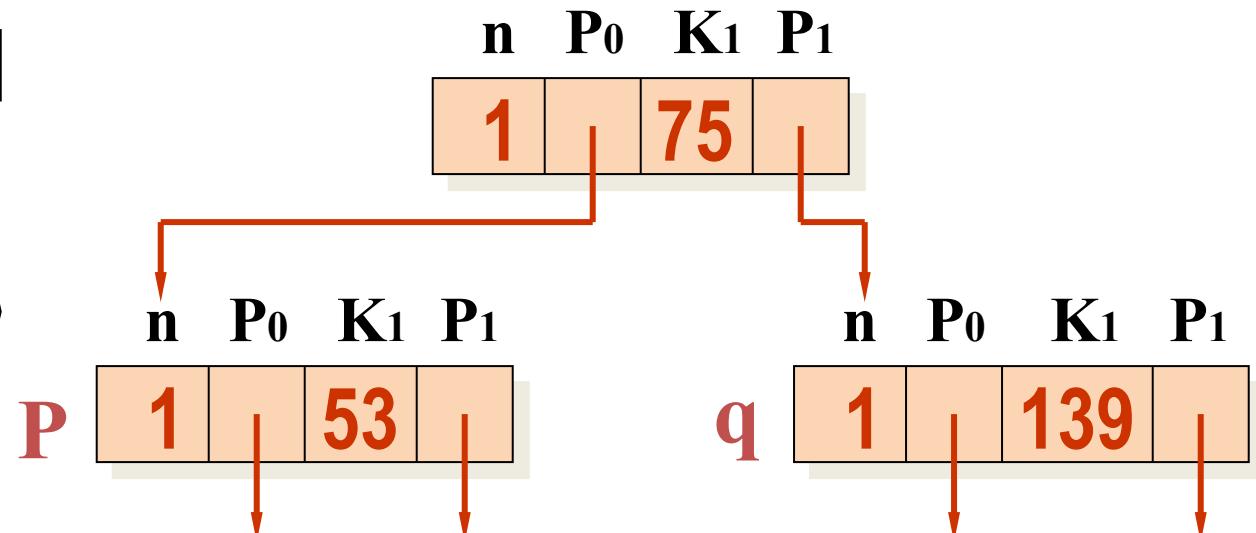
B 树的插入

- B 树是从空树起, 逐个插入关键码而生成的。
- 在B 树, 每个非失败结点的关键码个数都在 $\lceil m/2 \rceil - 1, m - 1$ 之间。
- 插入在某个叶结点开始。如果在关键码插入后结点中的关键码个数超出了上界 $m - 1$, 则结点需要“分裂”, 否则可以直接插入。
- 需要时, “分裂”可以一直进行到根结点。



加入139,
结点溢出

结点
分裂

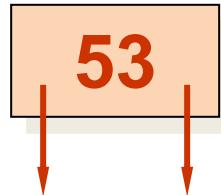


结点“分裂”的示例

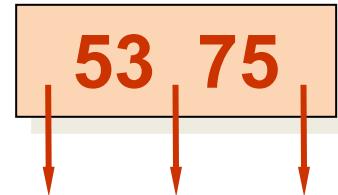


示例:从空树开始加入关键码建立3阶B 树

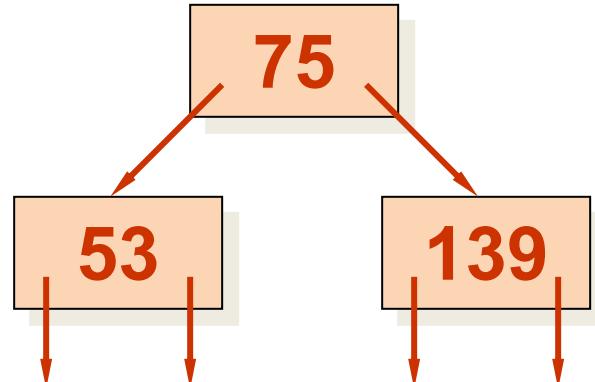
n=1 加入53



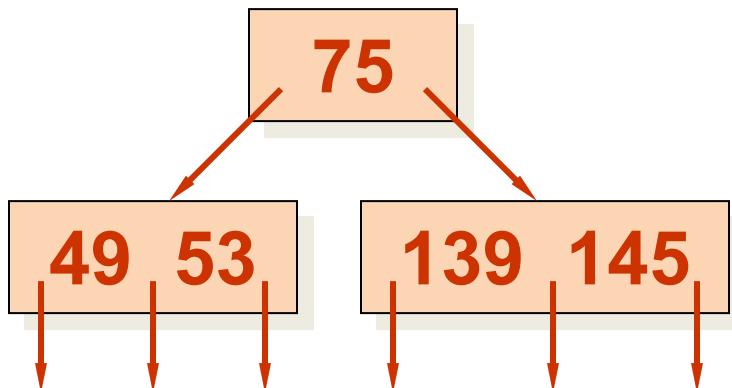
n=2 加入 75



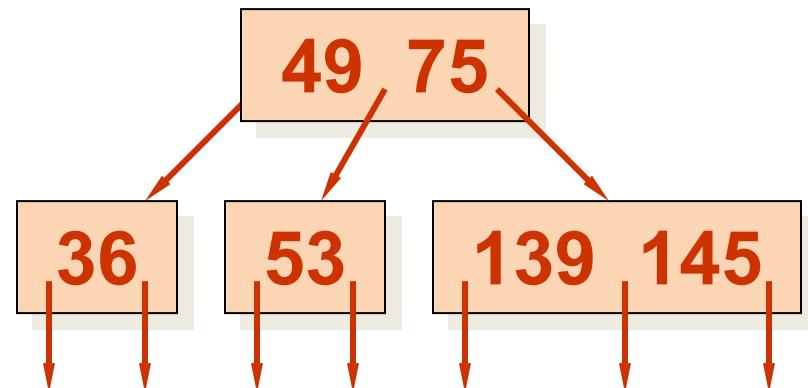
n=3 加入139



n=5 加入49,145

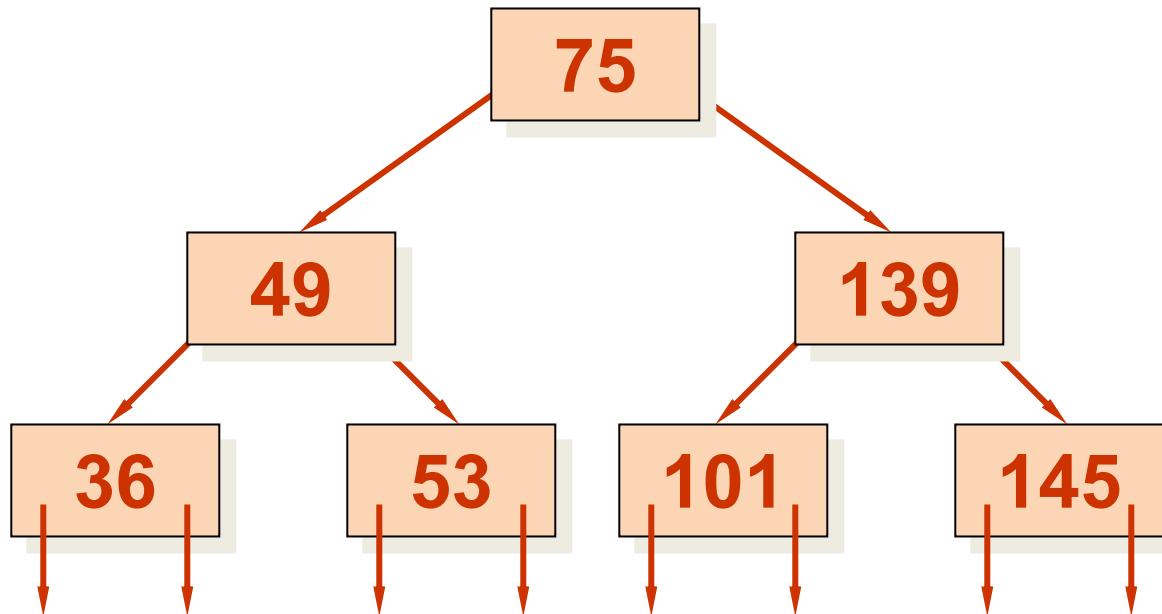


n=6 加入36





n=7 加入101



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

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



B 树的删除

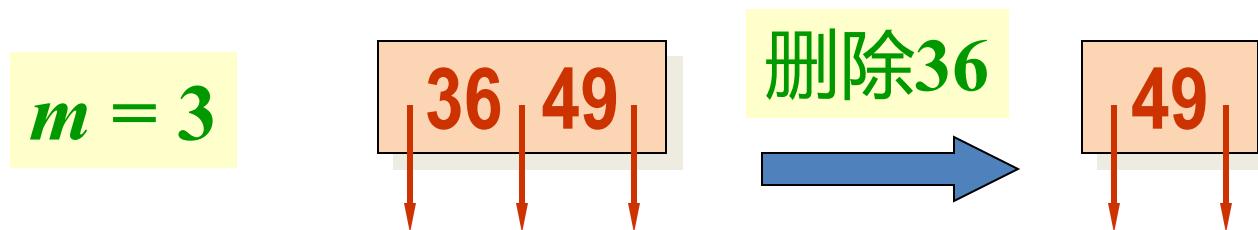
- 在B 树上删除一个关键码时，
 - ◆ 关键码不在叶子结点：

通过用它的**后继**（即比它大的最小数）或前驱（即比它小的最大数）来替换，将问题转化成删除叶子中的关键码。
 - ◆ 在叶结点删除关键码：
 - 1) 删除后，关键码够数：结束。
 - 2) 删除后，关键码不够数：若左（右）兄弟子树中有多余的，从兄弟子树中**借一个**，否则，进行**合并**。



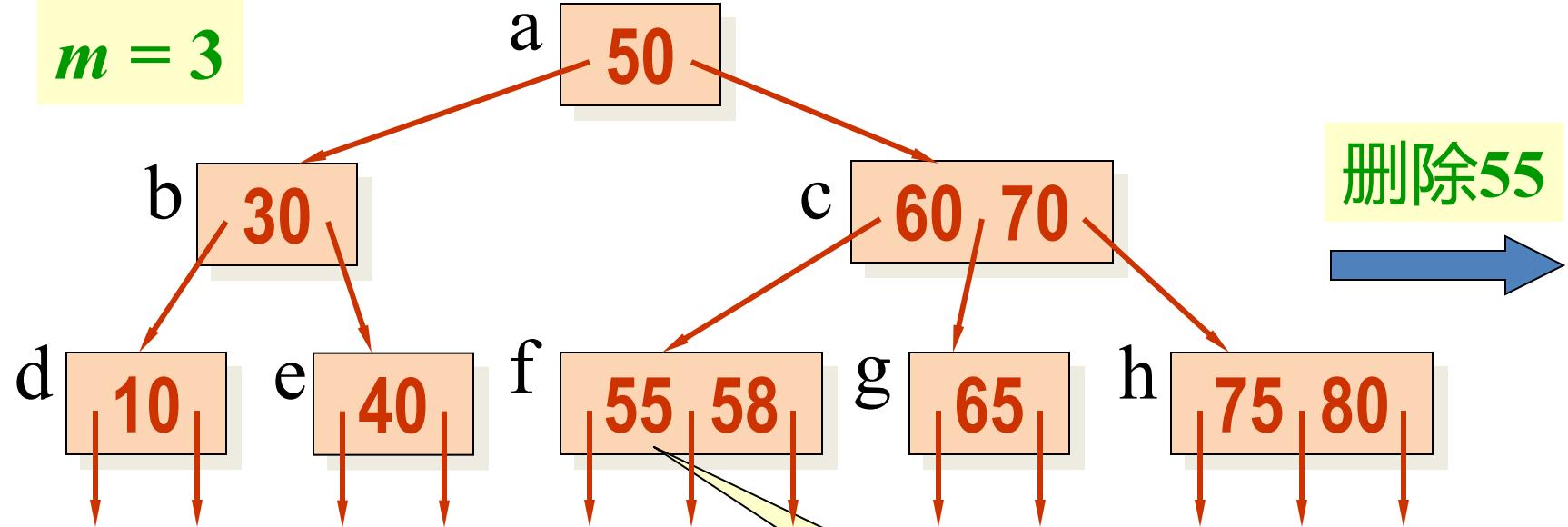
在叶结点上的删除有 4 种情况：

- ① 被删关键码所在叶结点同时又是根结点且删除前该结点中关键码个数 $n \geq 2$ ，则直接删去该关键码并将修改后的结点写回磁盘。

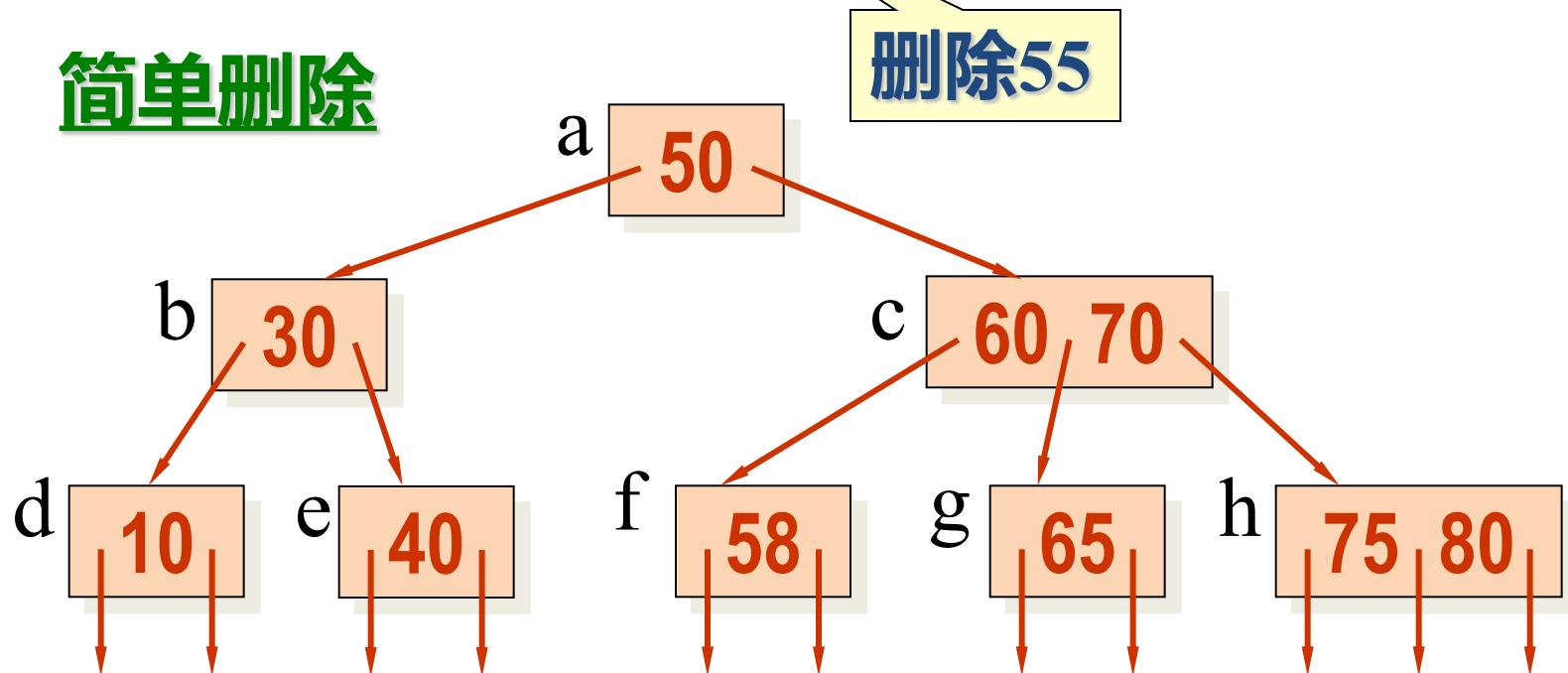


- ② 被删关键码所在叶结点不是根结点且删除前该结点中关键码个数 $n \geq \lceil m/2 \rceil$ ，则直接删去该关键码并将修改后的结点写回磁盘，删除结束。

$m = 3$



简单删除





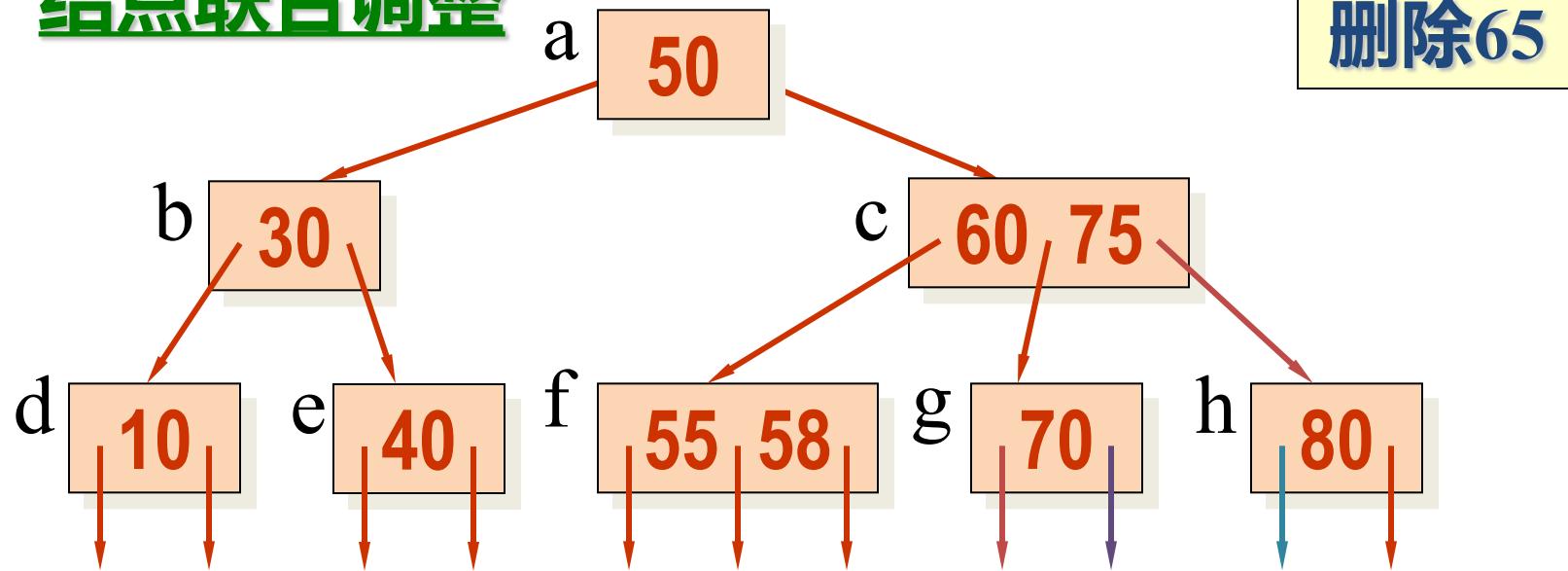
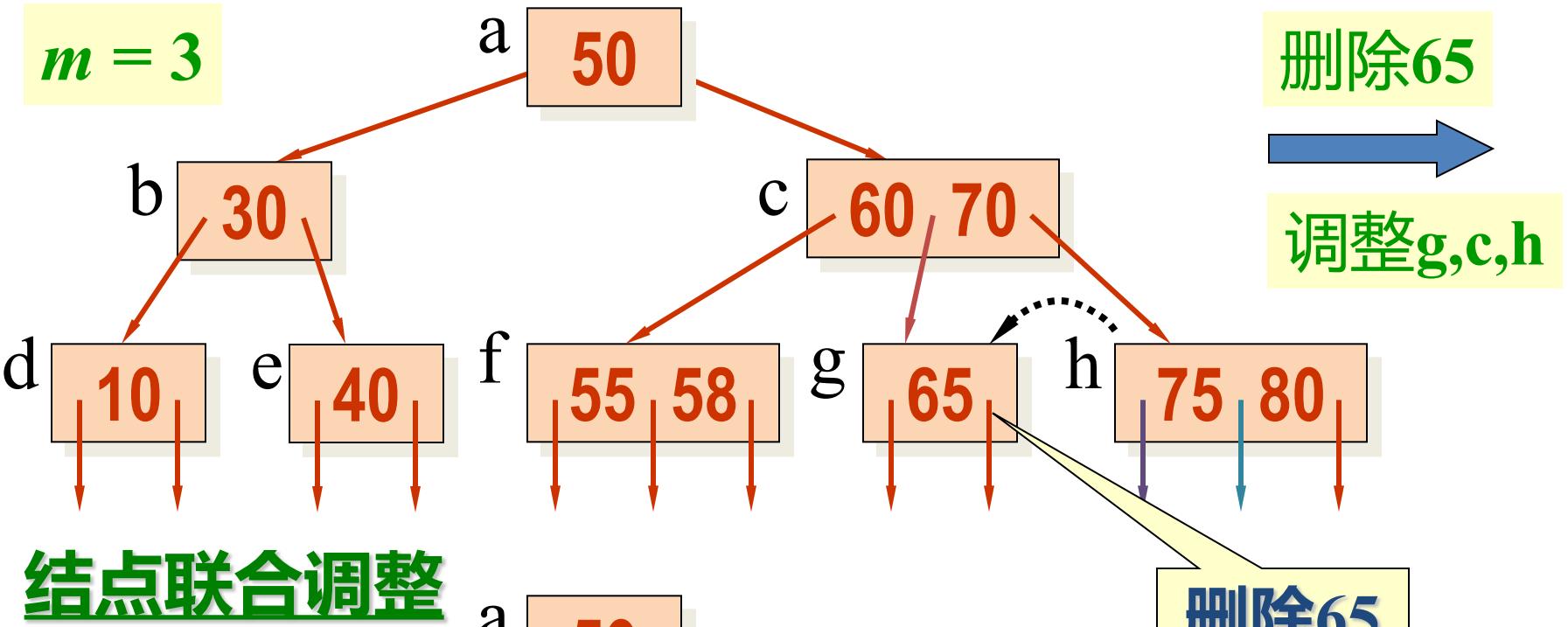
③ 被删关键码所在叶结点删除前关键码个数 $n = \lceil m/2 \rceil - 1$, 若这时与该结点相邻的右兄弟 (或左兄弟) 结点的关键码个数 $n \geq \lceil m/2 \rceil$, 则可按以下步骤调整该结点、右兄弟 (或左兄弟) 结点以及其双亲结点, 以达到新的平衡。

- ◆ 将双亲结点中刚刚大于 (或小于) 该被删关键码的关键码 K_i ($1 \leq i \leq n$) 下移;
- ◆ 将右兄弟 (或左兄弟) 结点中的最小 (或最大) 关键码上移到双亲结点的 K_i 位置;
- ◆ 将右兄弟 (或左兄弟) 结点中的最左 (或最右) 子树指针平移到被删关键码所在结点中最右 (或最左) 子树指针位置;

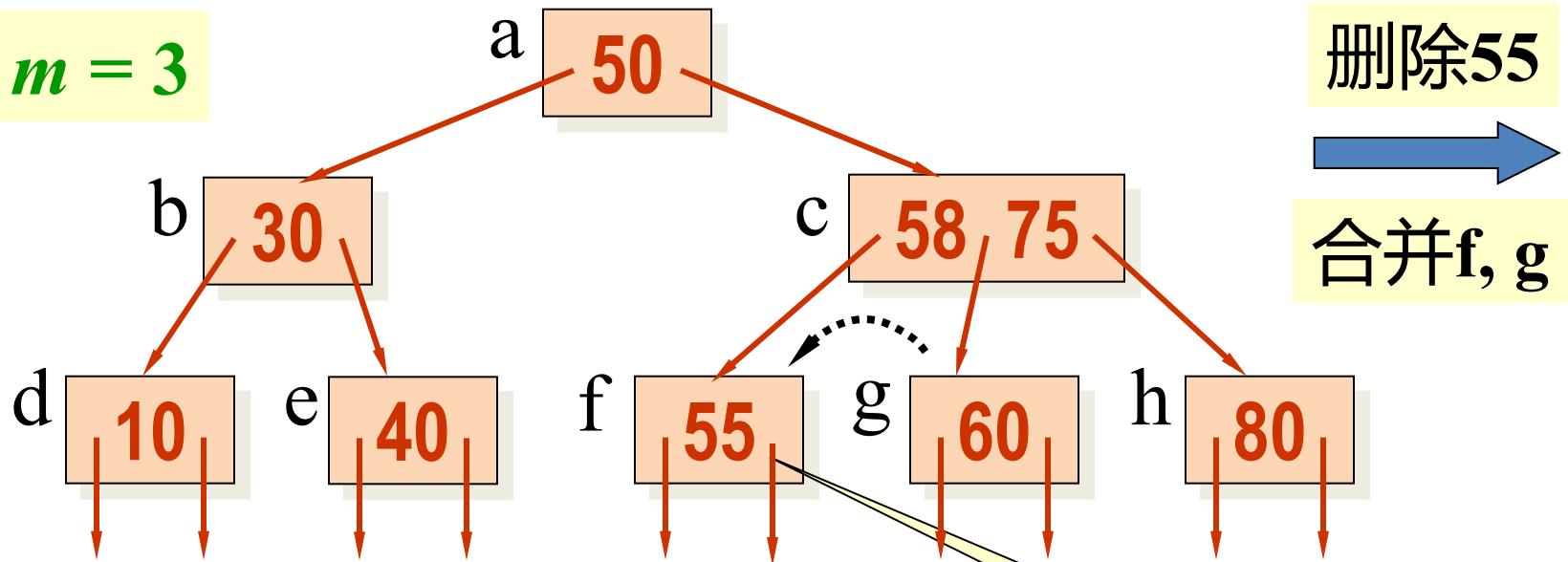


◆ 在右兄弟(或左兄弟)结点中，将被移走的关键码和指针位置用剩余的关键码和指针填补、调整。再将结点中的关键码个数减1。

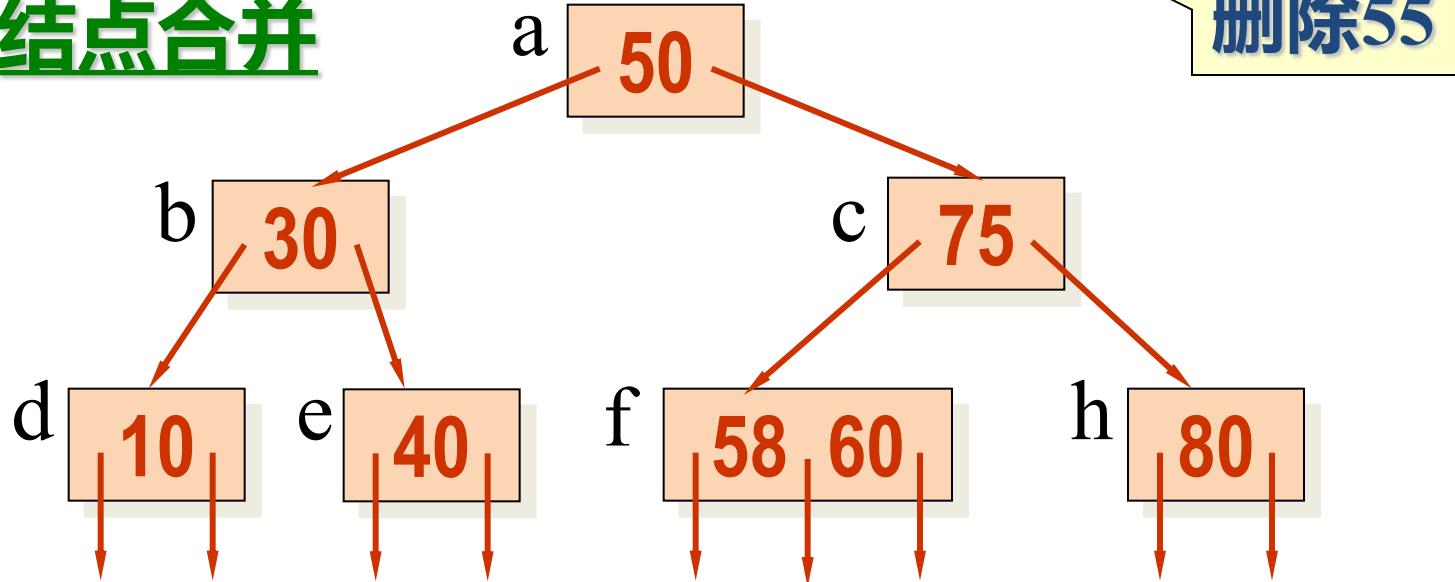
④ 被删关键码所在叶结点删除前关键码个数 $n = \lceil m/2 \rceil - 1$, 若这时与该结点相邻的右兄弟(或左兄弟)结点的关键码个数 $n = \lceil m/2 \rceil - 1$, 则必须按以下步骤合并这两个结点。

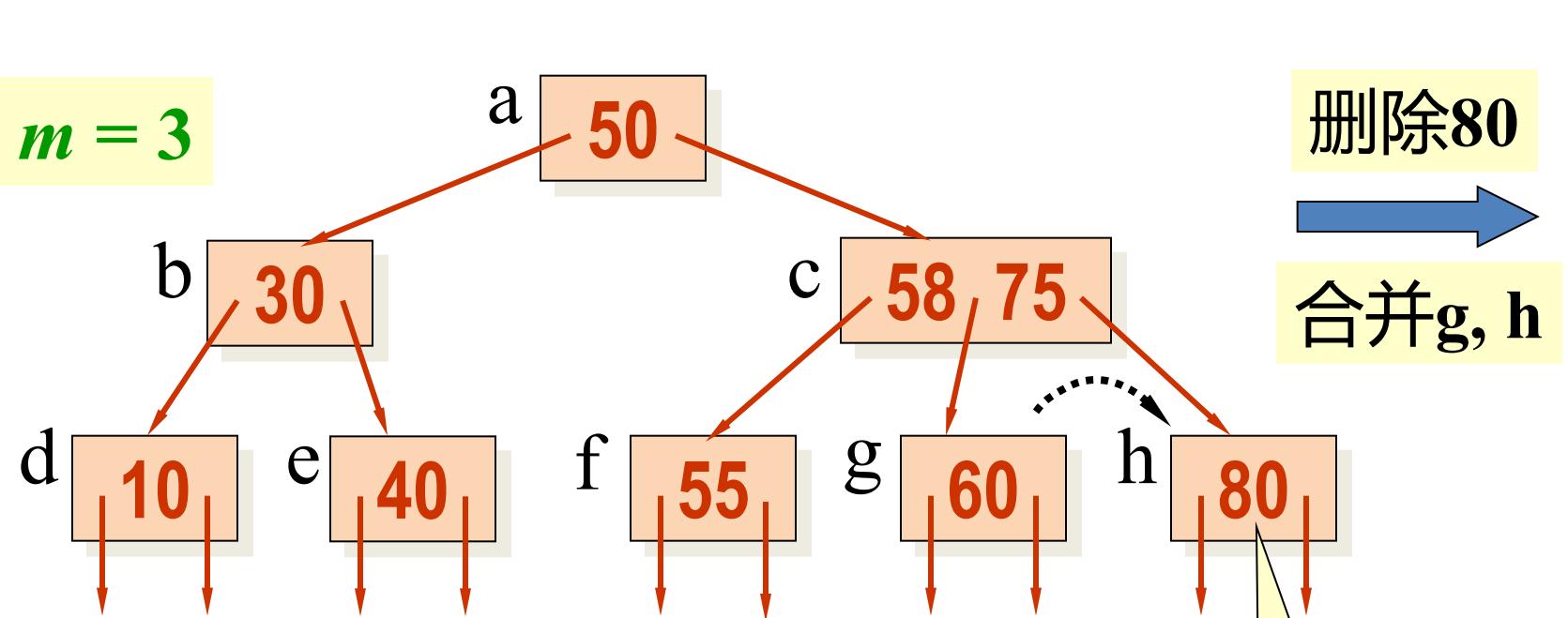


$$m = 3$$

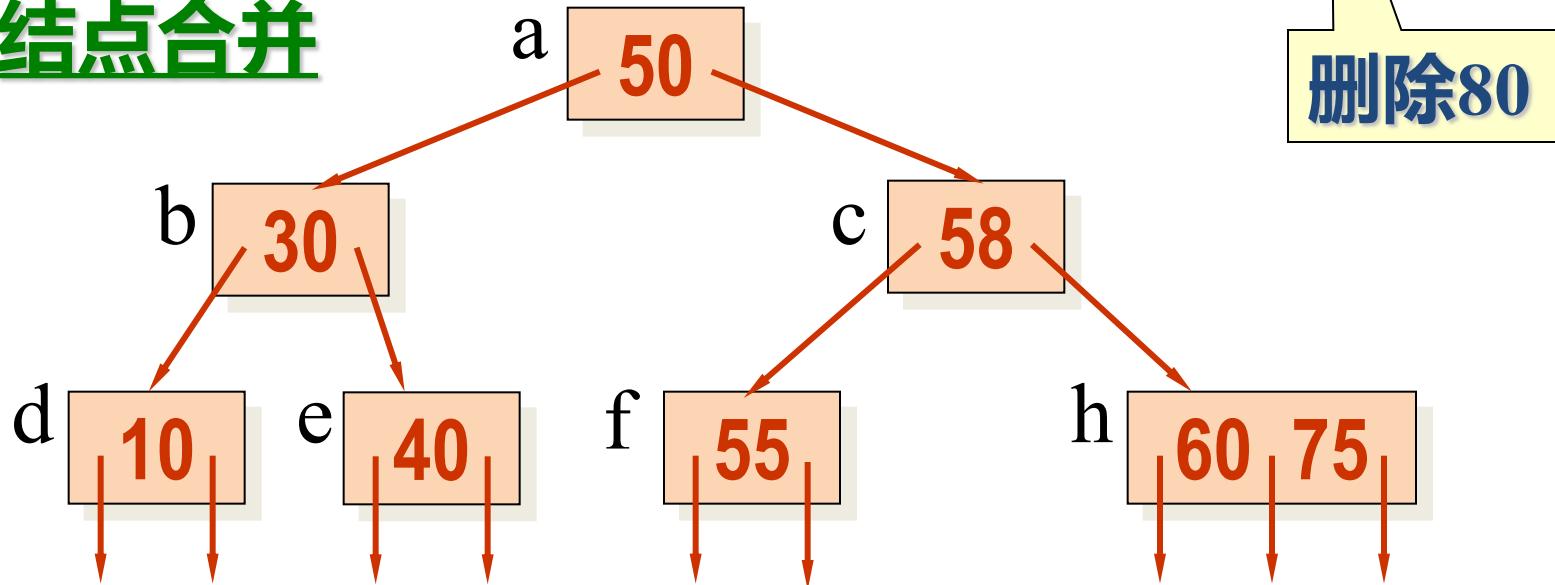


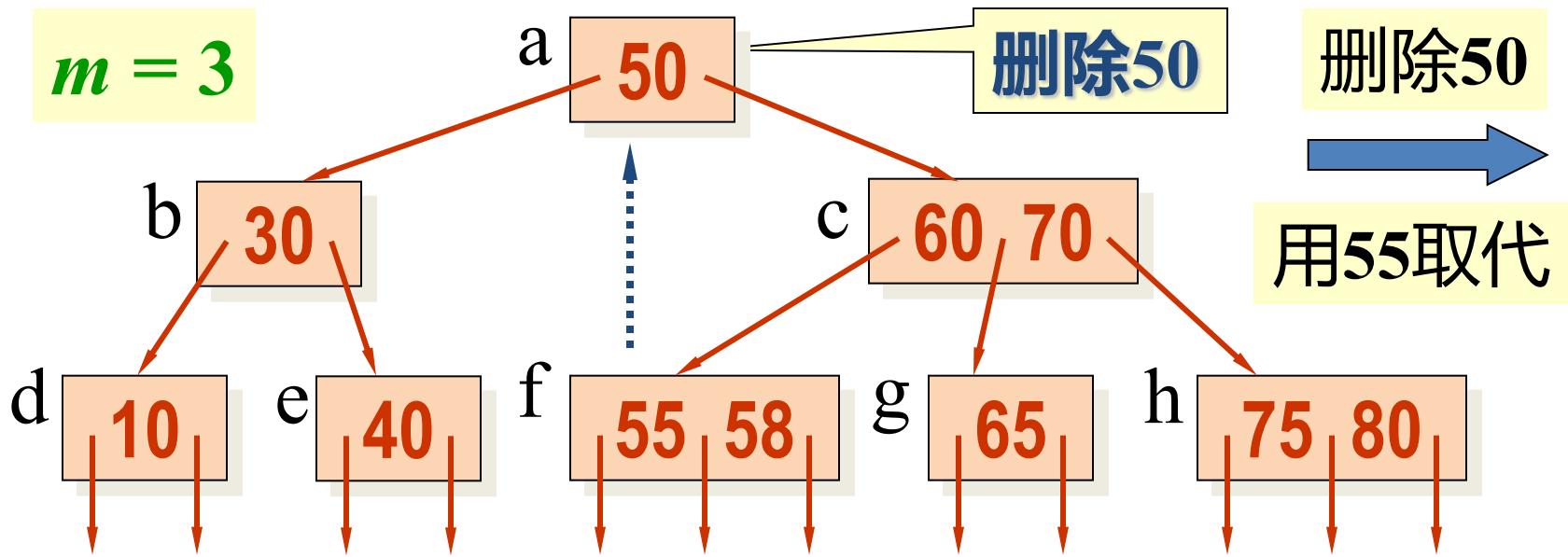
结点合并



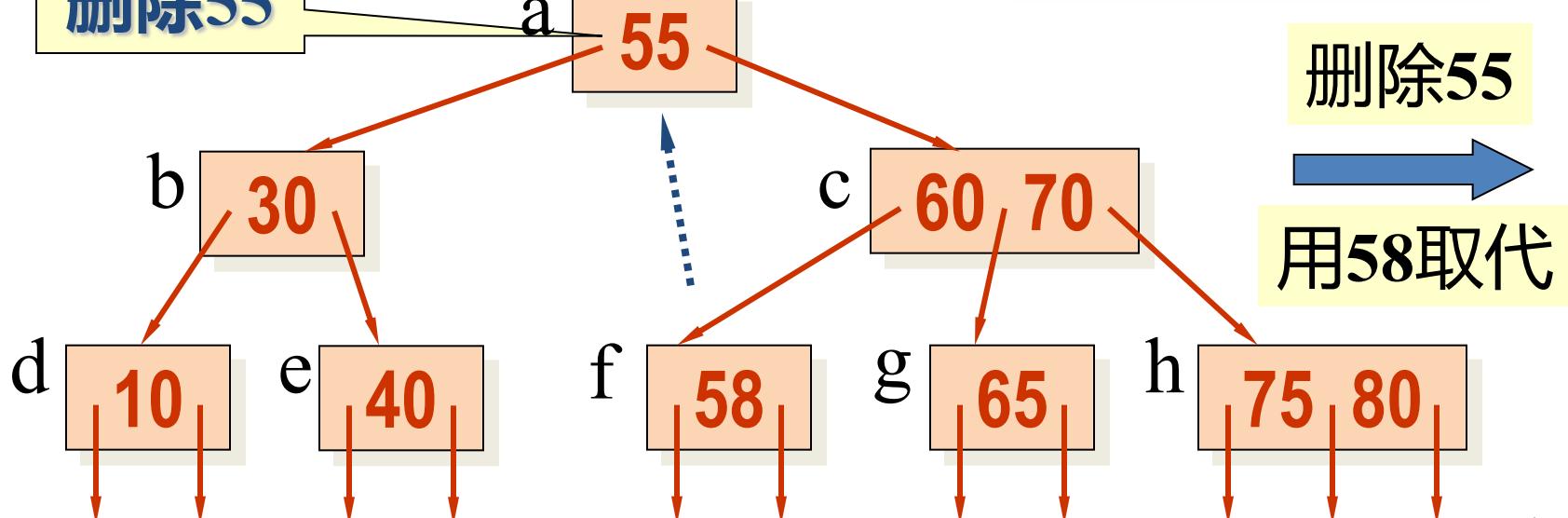


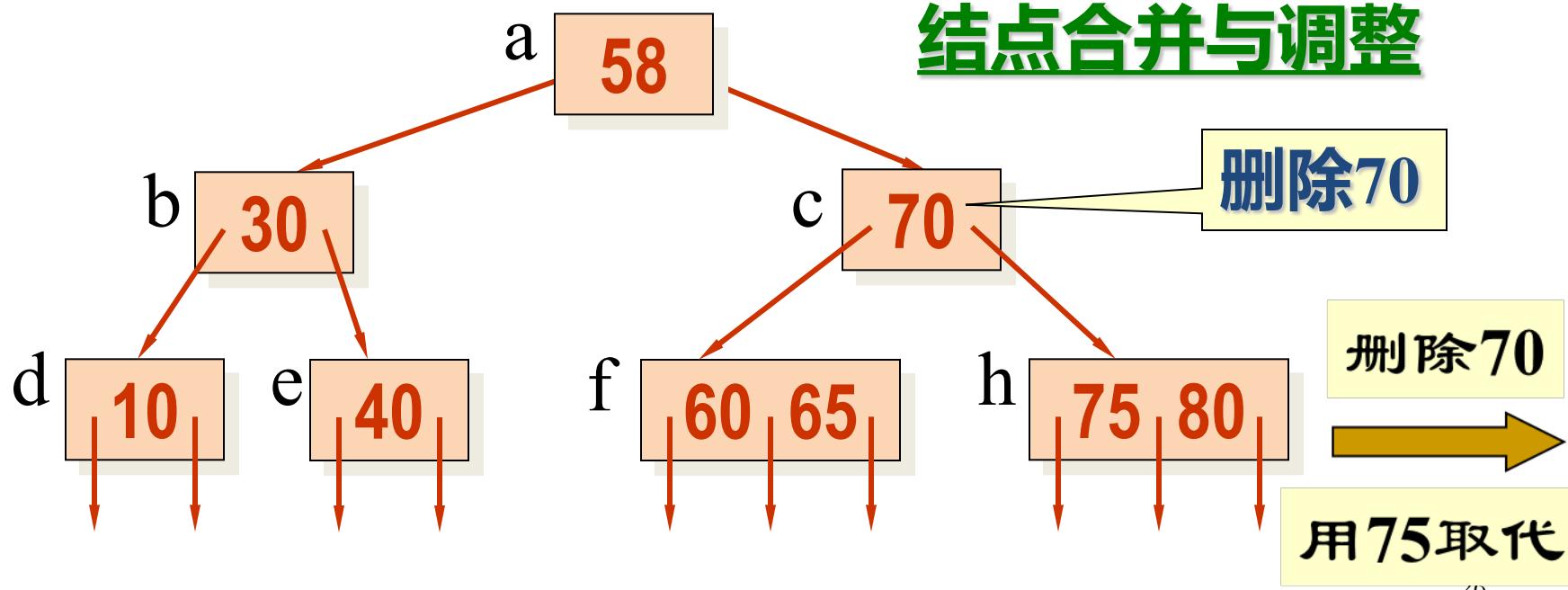
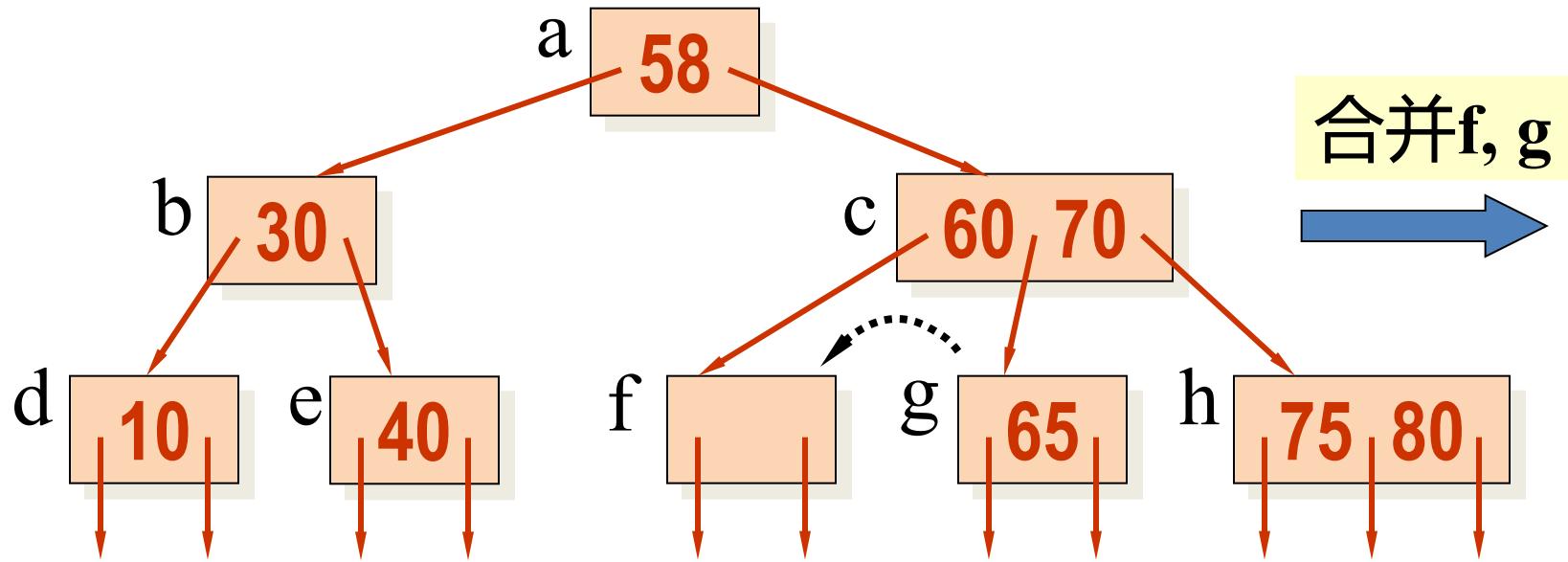
结点合并

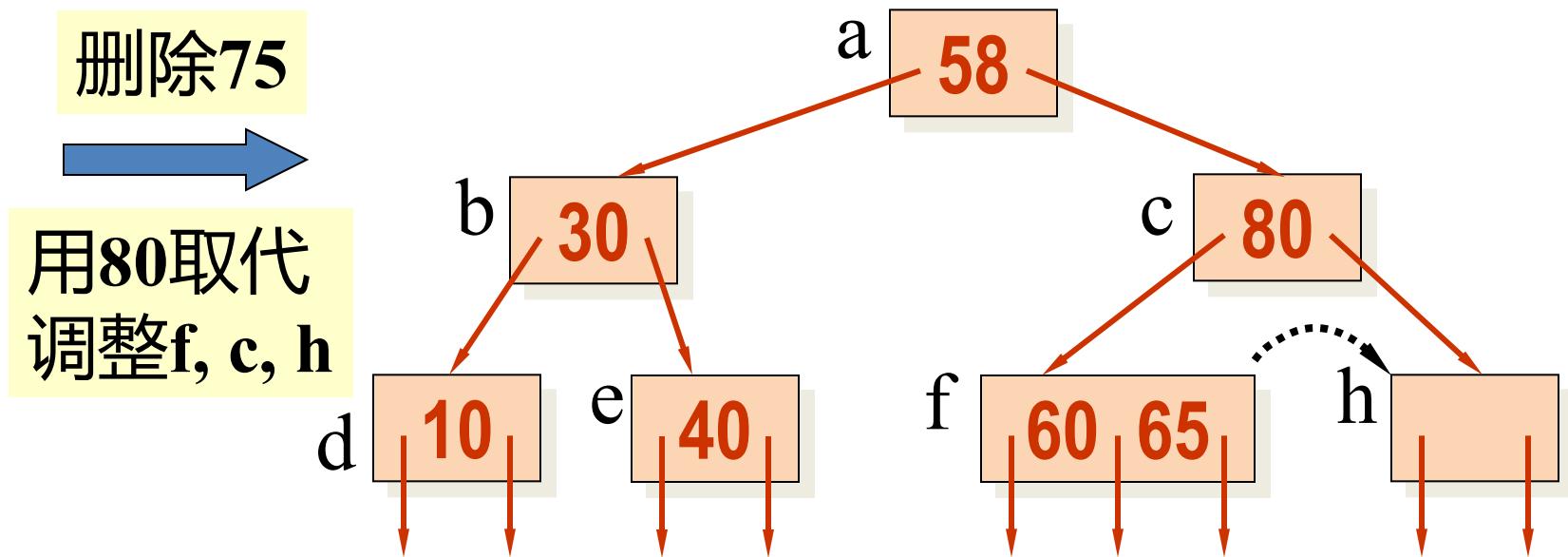
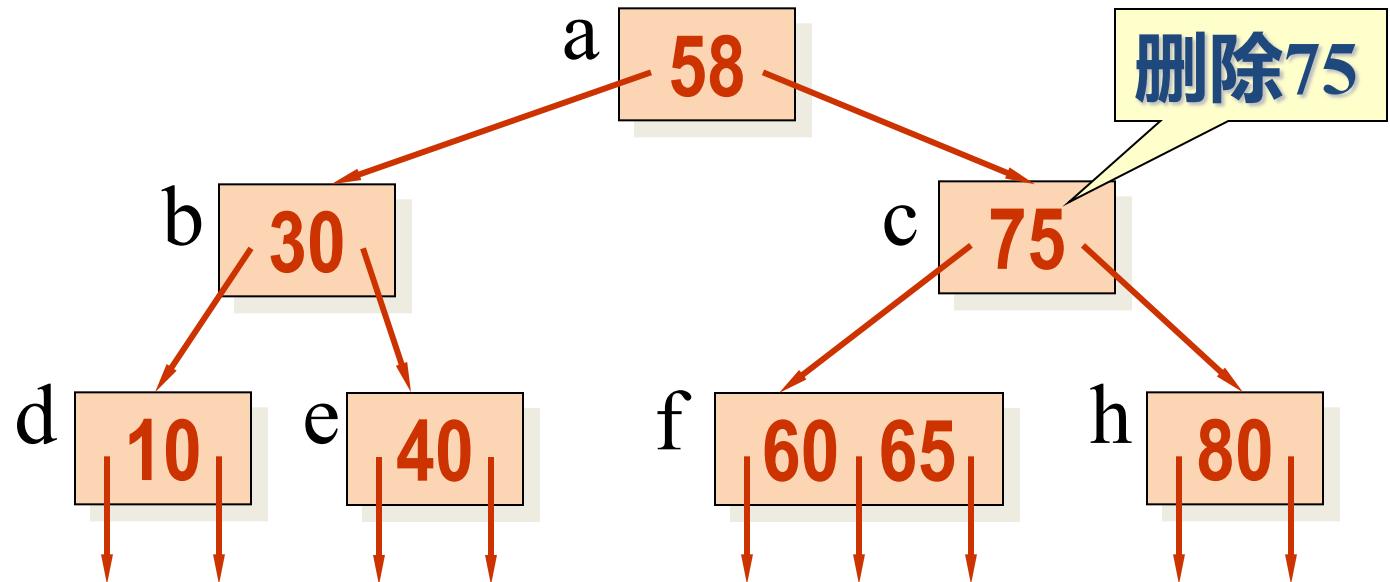


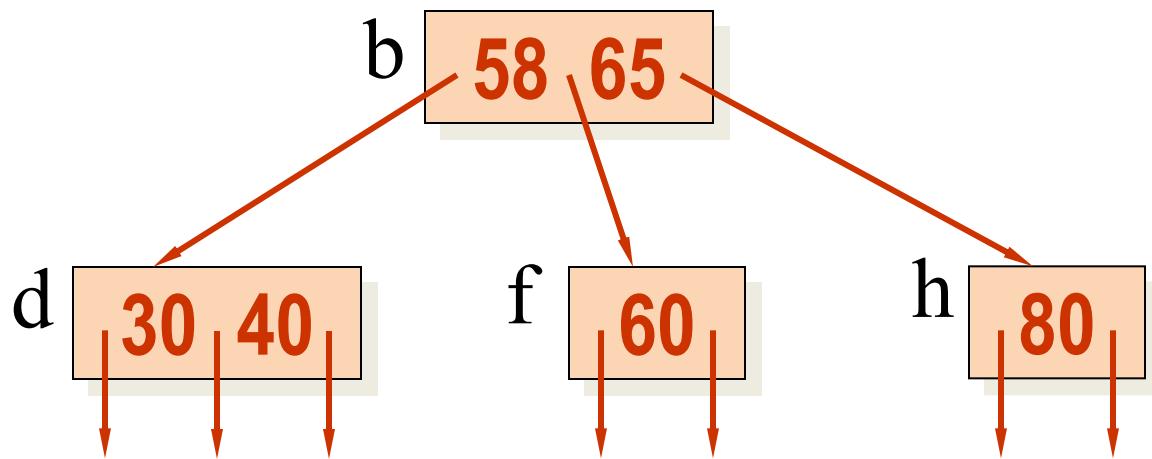
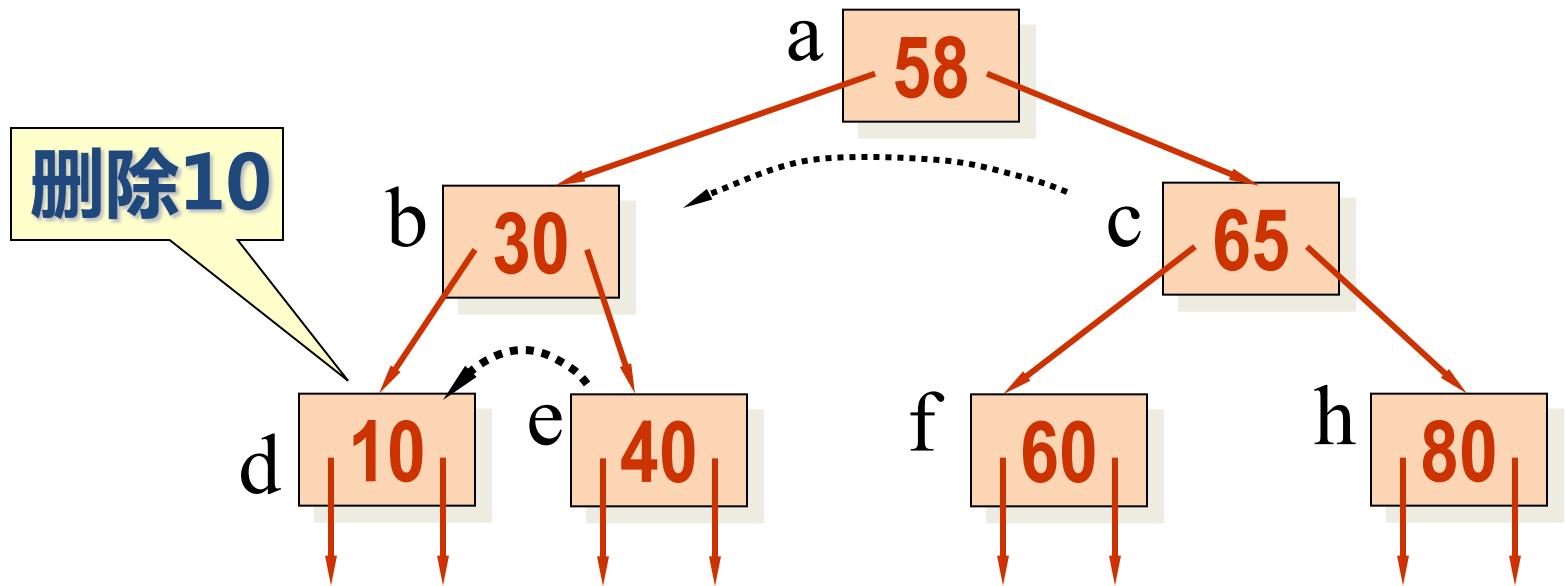


非叶结点删除



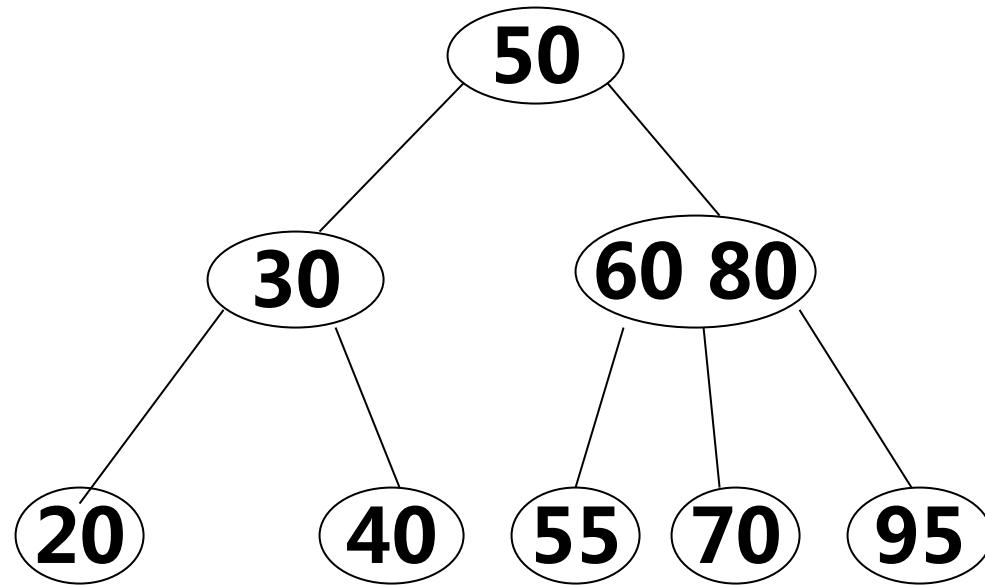


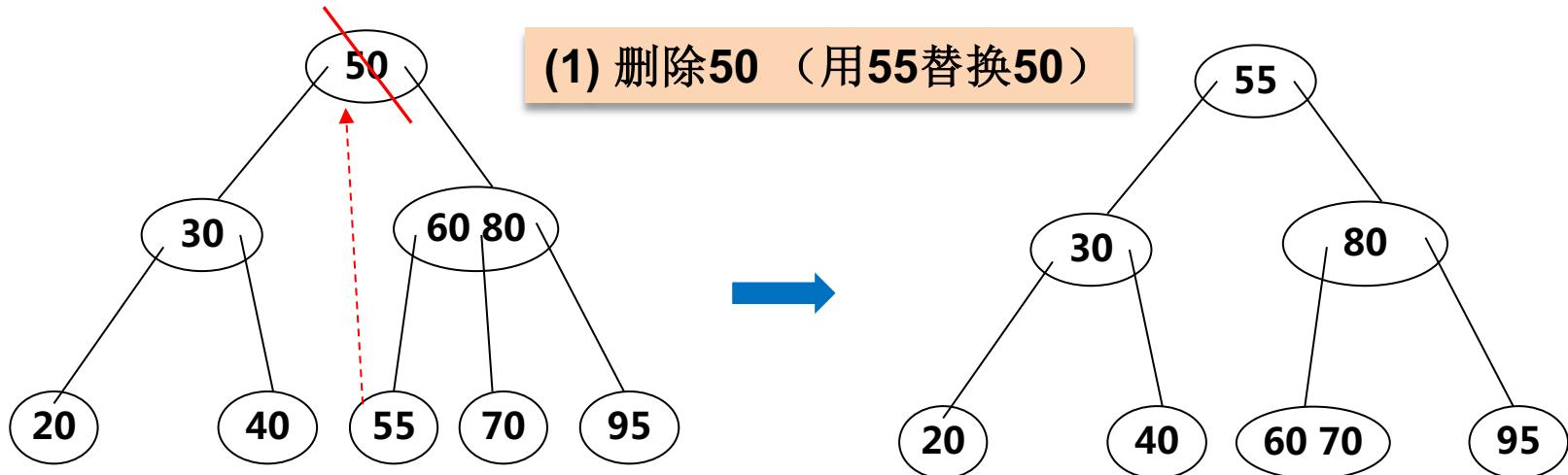




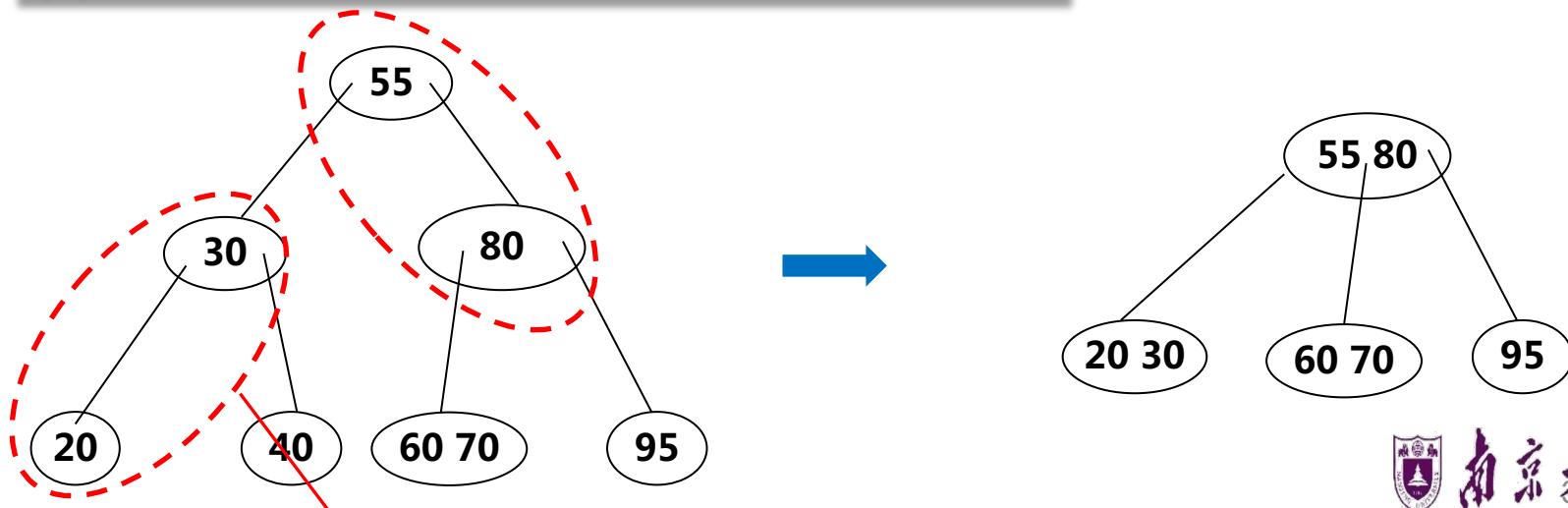


例1： 在3阶B-树中依次删除50 , 40。





(2) 删除40（20和30合并，并引起55和80合并）



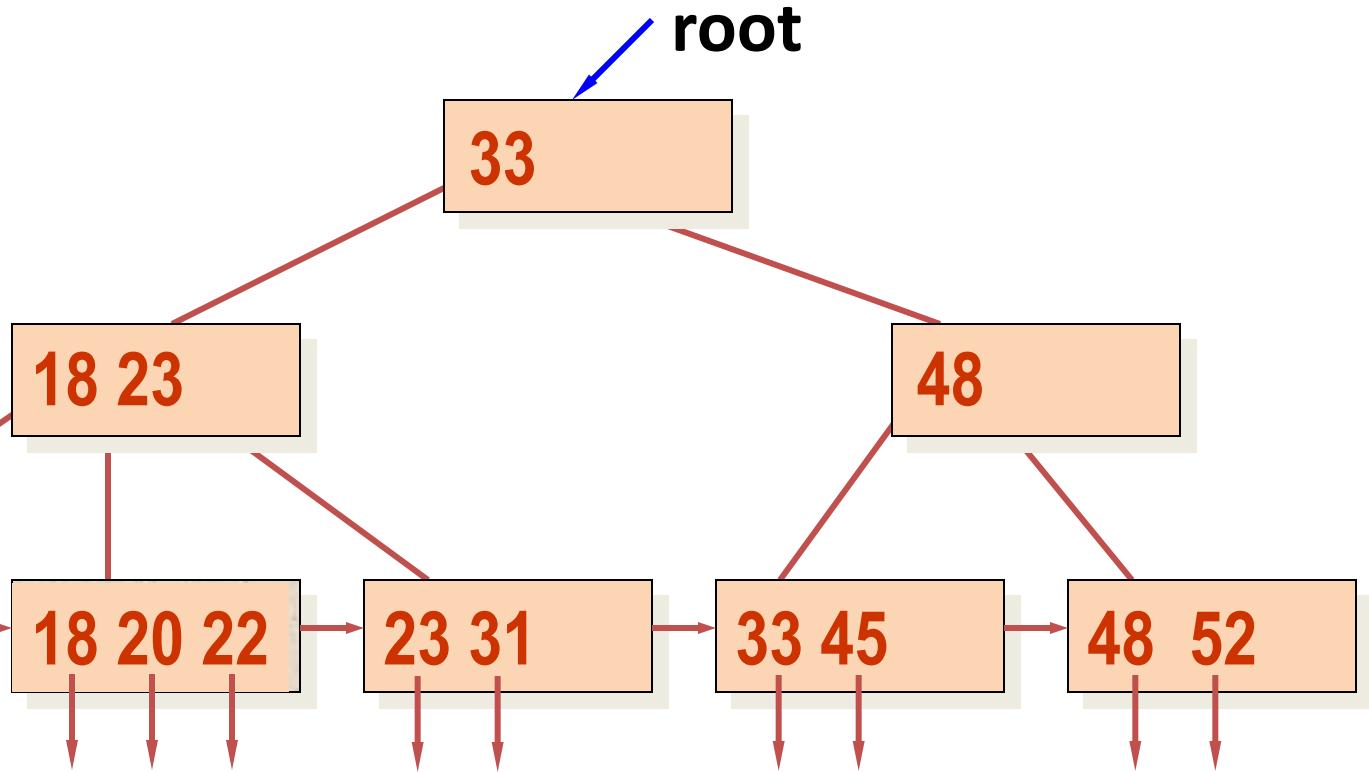


B+树

- B+树是B树的一种变体，查询性能往往更好。
- 一棵m阶的B+树和m阶的B树的差异在于：
 - 1) 叶子结点中包含了全部关键码的信息，及指向含有这些关键码记录的指针，且叶子结点本身依关键码的大小自小而大的顺序链接。
 - 2) 所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键码。

$m = 3$

$m1 = 3$



- ◆ 叶结点的子树棵数可以多于 m , 可以少于 m ,
叶结点最大的关键码数由 $m1$ 而定。
- ◆ 叶结点每个关键码对应了一个指向对象的地址
指针。



- ◆ 叶结点中的子树棵数 n 应满足

$$n \in \lceil m1/2 \rceil, m1]。$$

- ◆ 若根结点同时又是叶结点，则结点格式同叶结点，但最少可有1棵子树。
- ◆ 非叶结点可以看成是索引部分，结点中关键码 K_i 与指向子树的指针 P_i 构成对子树(即下一层索引块)的索引项(K_i, P_i)， K_i 是子树中最小的关键码。
- ◆ 特别地，子树指针 P_0 所指子树上所有关键码均小于 K_1 。结点格式同B 树。
- ◆ 叶结点中存放的是对实际数据对象的索引。



- 可对B+树进行**两种搜索运算**:
 - ◆ 从(**sqt**)开始循叶结点链**顺序搜索**。
 - ◆ 另一种是从根结点(**root**)开始, 进行**自顶向下**, 直至叶结点的**随机搜索**。
- 在B+树上进行**随机搜索**、**插入**和**删除**的过程基本上与**B** 树类似。只是在**搜索过程中**, 如果非叶结点上的关键码等于给定值, 搜索并不停止, 而是继续沿右指针向下, 一直查到叶结点上的这个关键码。**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。

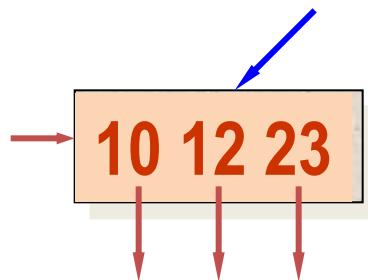


- B+树的搜索分析类似于B 树。
- B+树的插入仅在叶结点上进行。当插入后结点中的子树棵数 $n > m_1$ 时，需要将叶结点分裂为两个结点，它们的关键码分别为 $\lceil (m_1+1)/2 \rceil$ 和 $\lfloor (m_1+1)/2 \rfloor$ 。它们的双亲结点中应同时包含这两个结点的最小关键码和结点地址。



B+ 树的插入

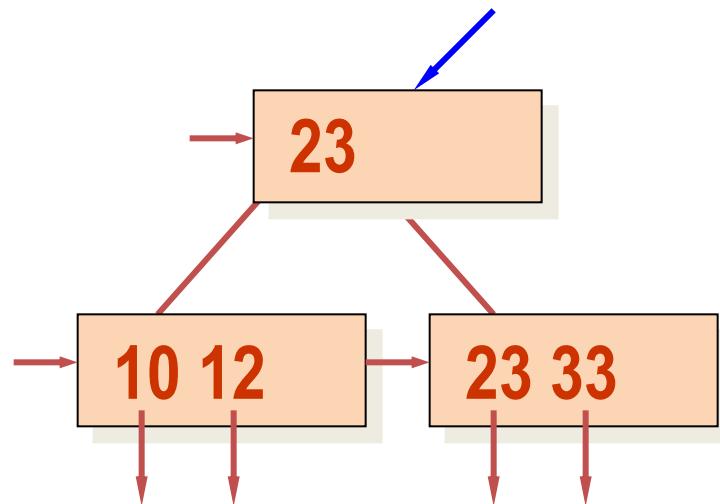
3个关键码的B+树



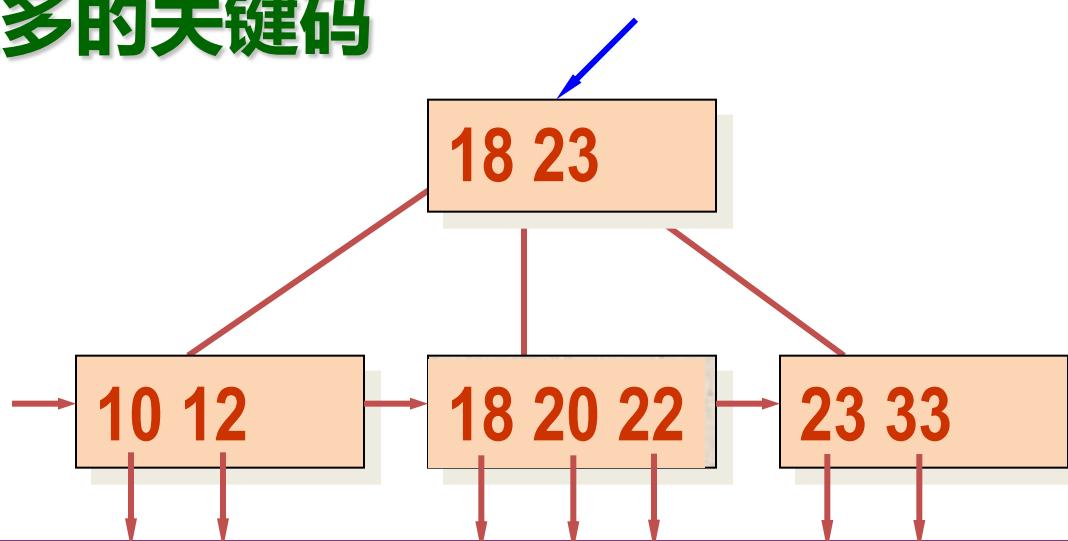
$$m = 3$$

$$m_1 = 3$$

加入关键码33, 结点分裂



加入更多的关键码



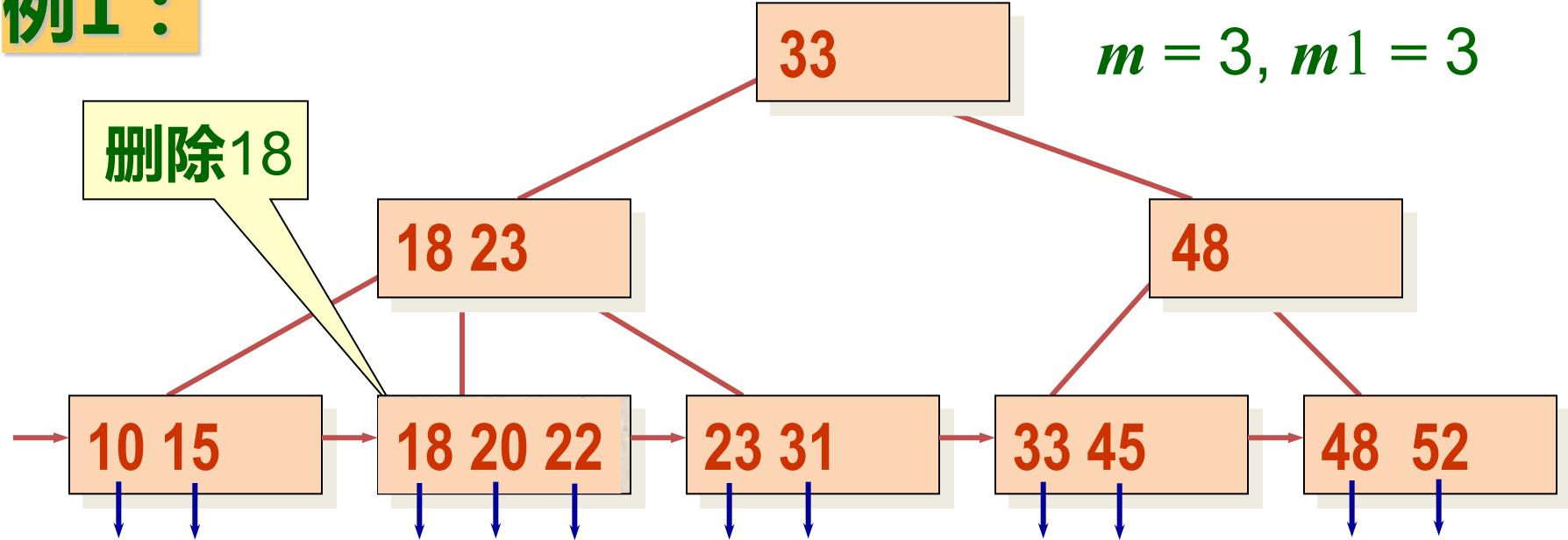


- B+ 树的删除仅在叶结点上进行。
 - 当叶子结点中的最大关键码被删除时，其在非终端结点中的值可以作为一个“**分界关键码**”存在。
 - 若因删除一个(关键码-指针)索引项后，结点中的子树棵数少于「 $m/2$ 」，兄弟结点的合并过程亦和B-树类似。

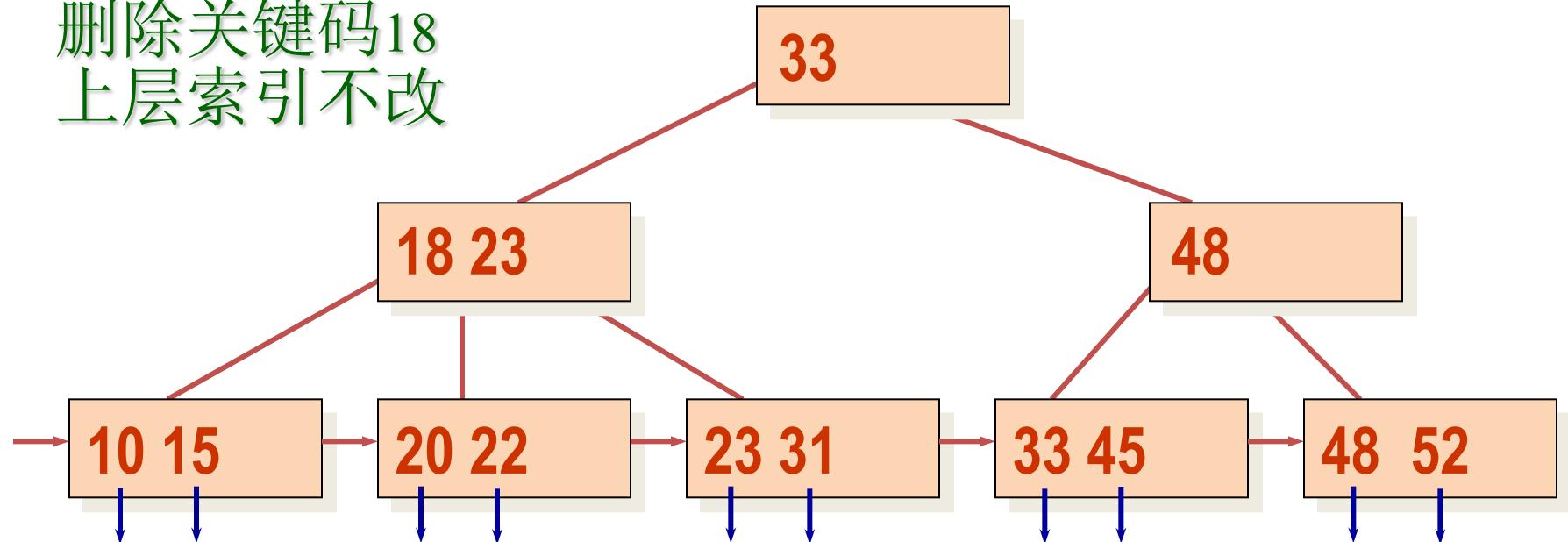
B+树的删除

例1：

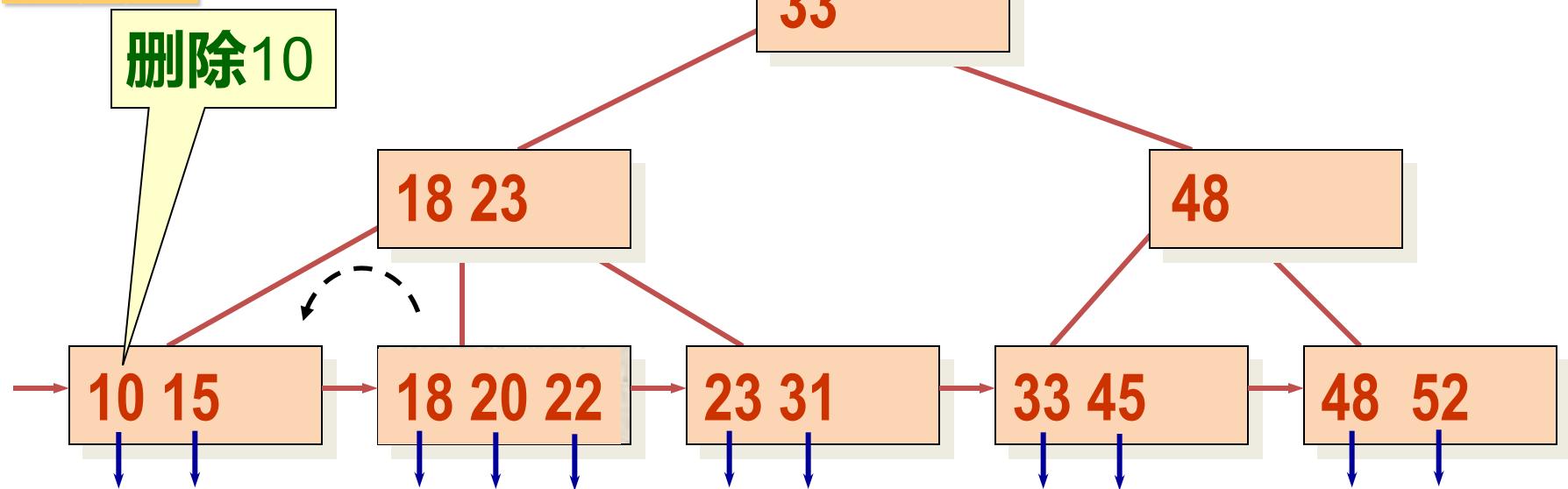
$$m = 3, m_1 = 3$$



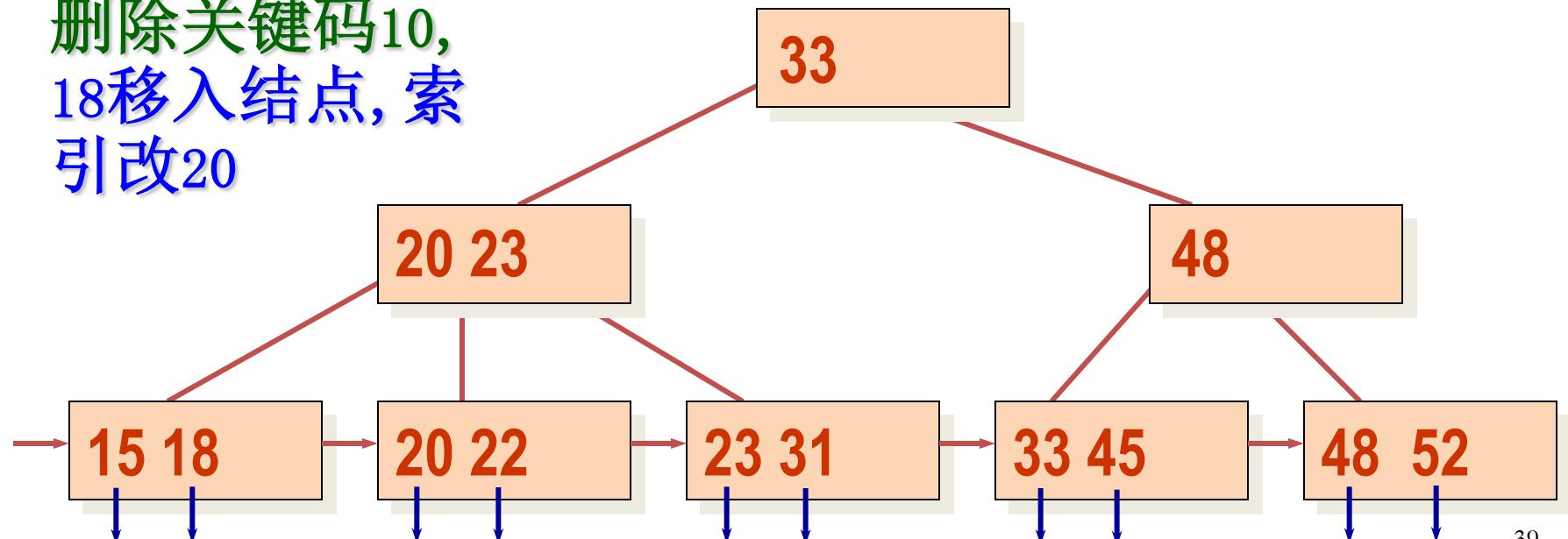
删除关键码18
上层索引不改



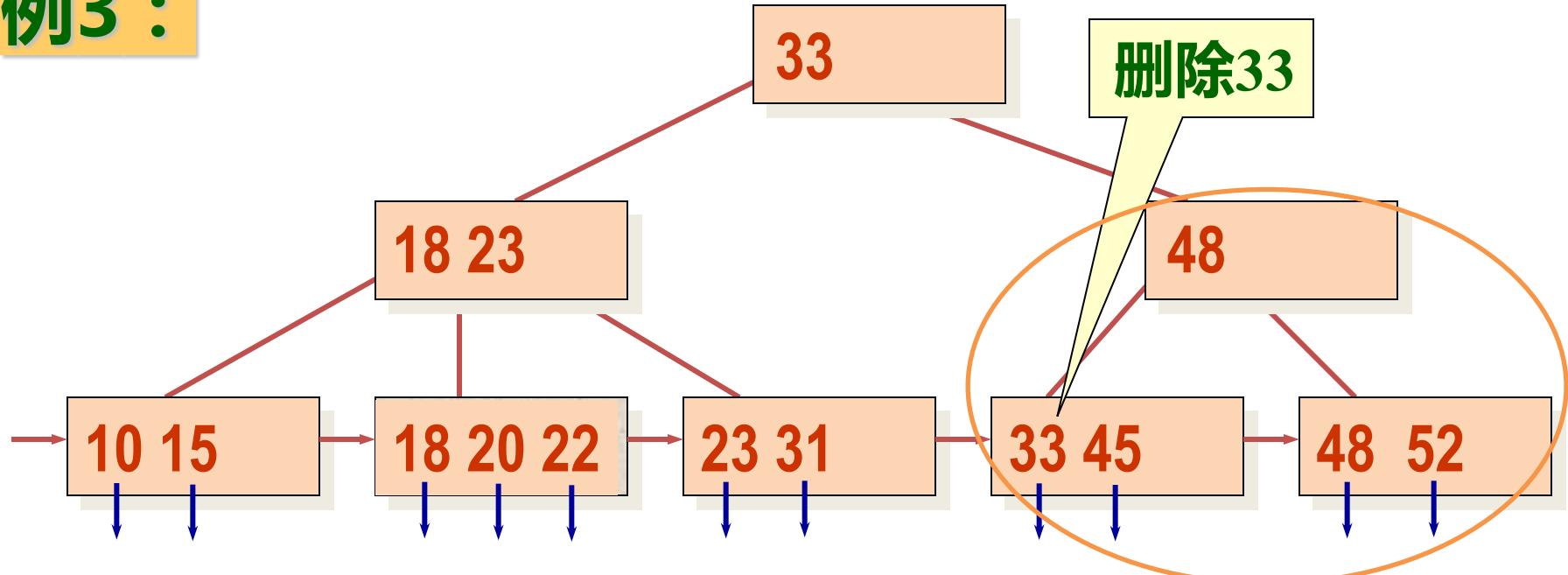
例2：



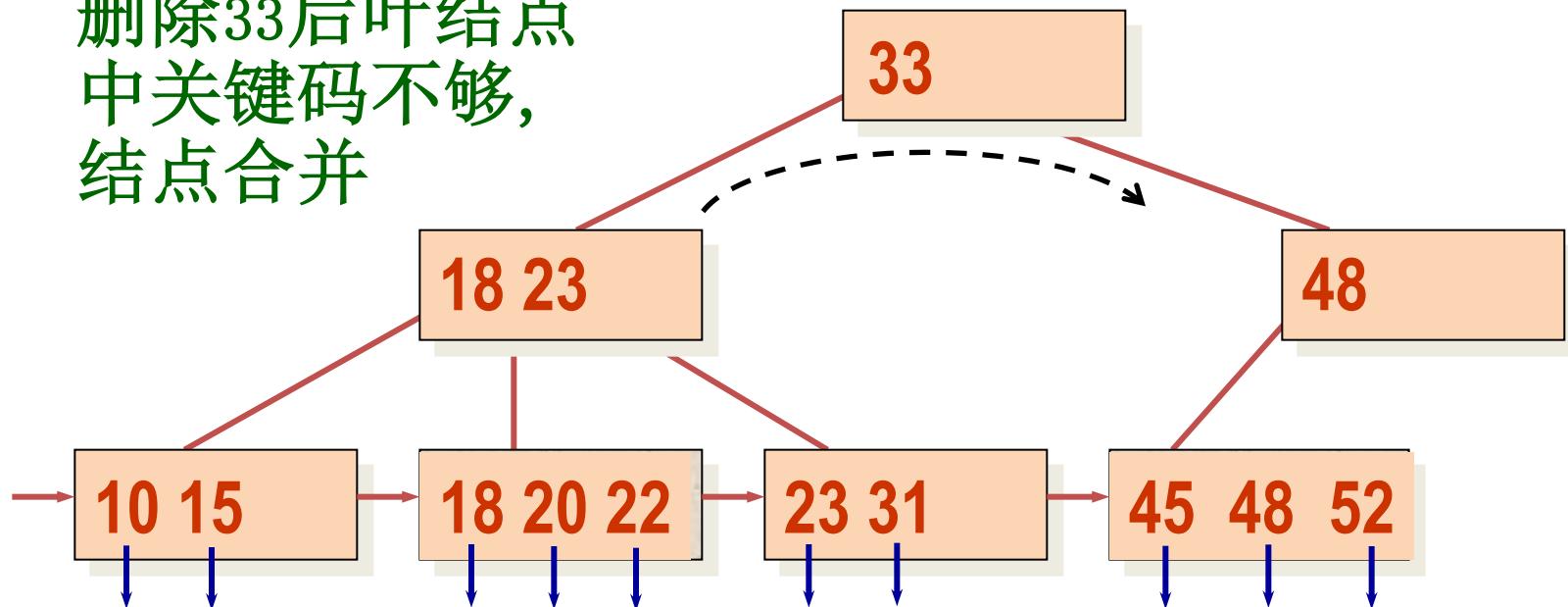
删除关键码10,
18移入结点, 索
引改20

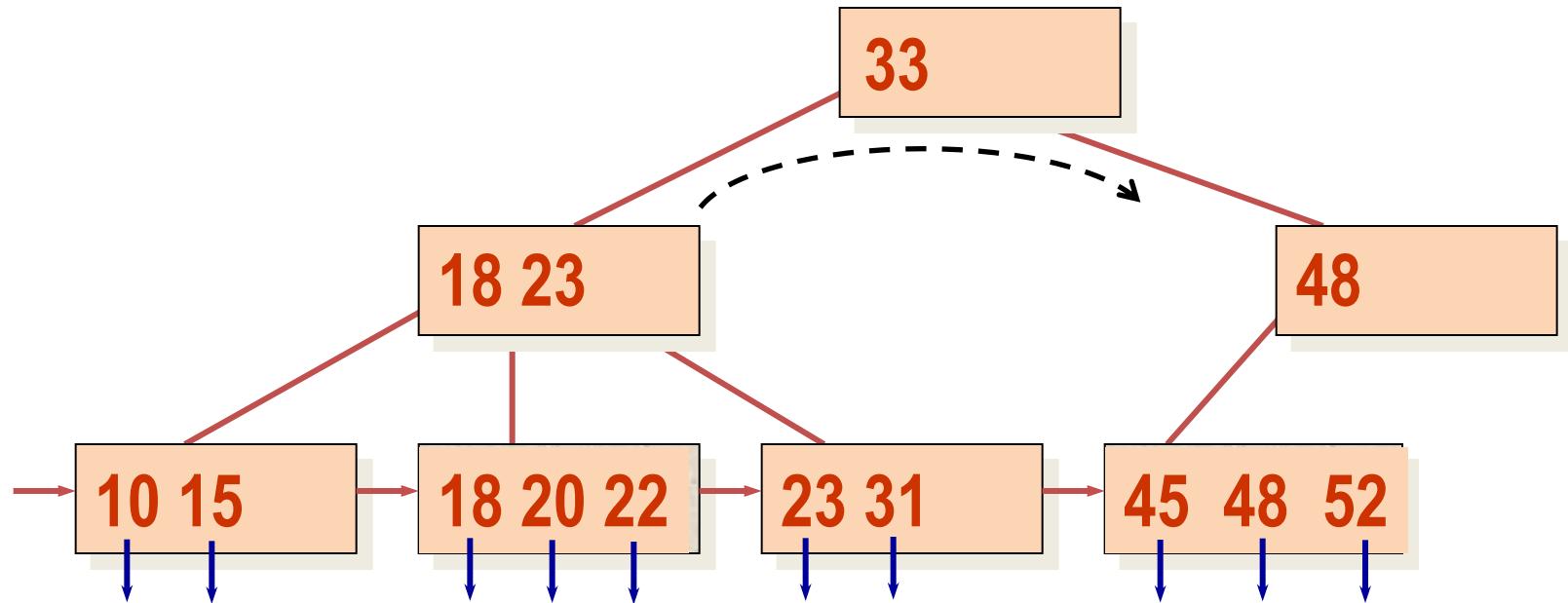


例3：

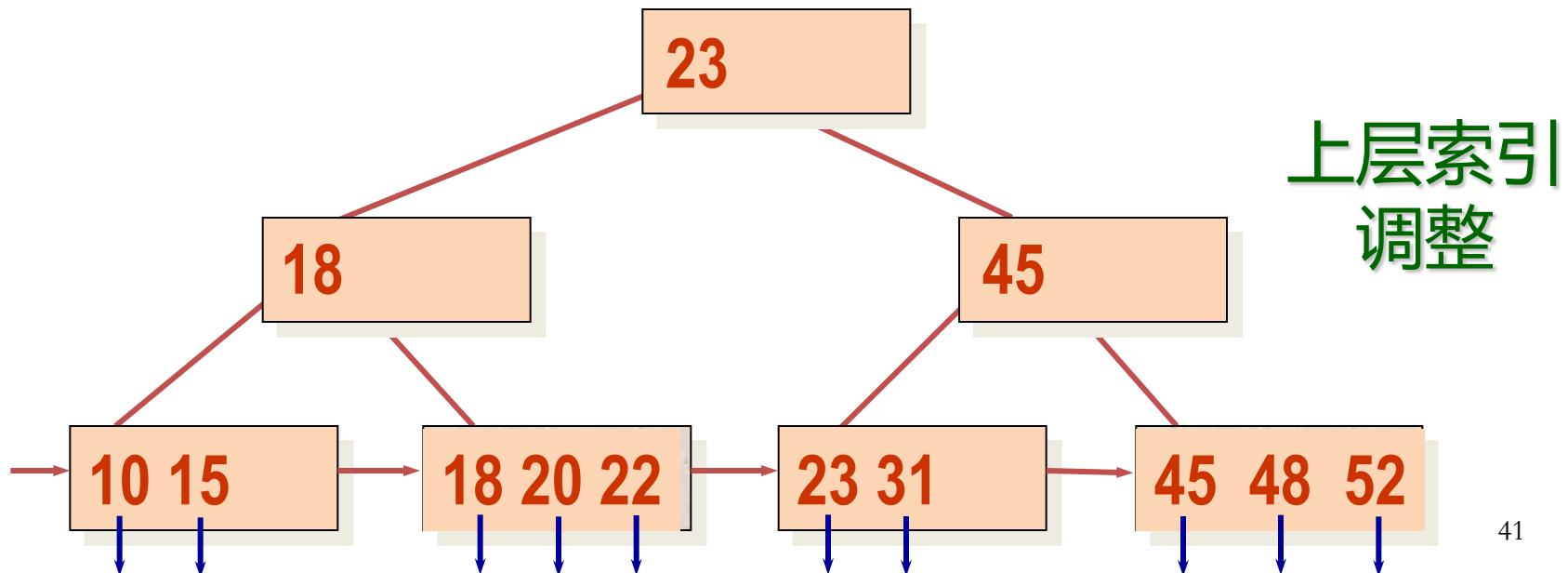


删除33后叶结点
中关键码不够，
结点合并

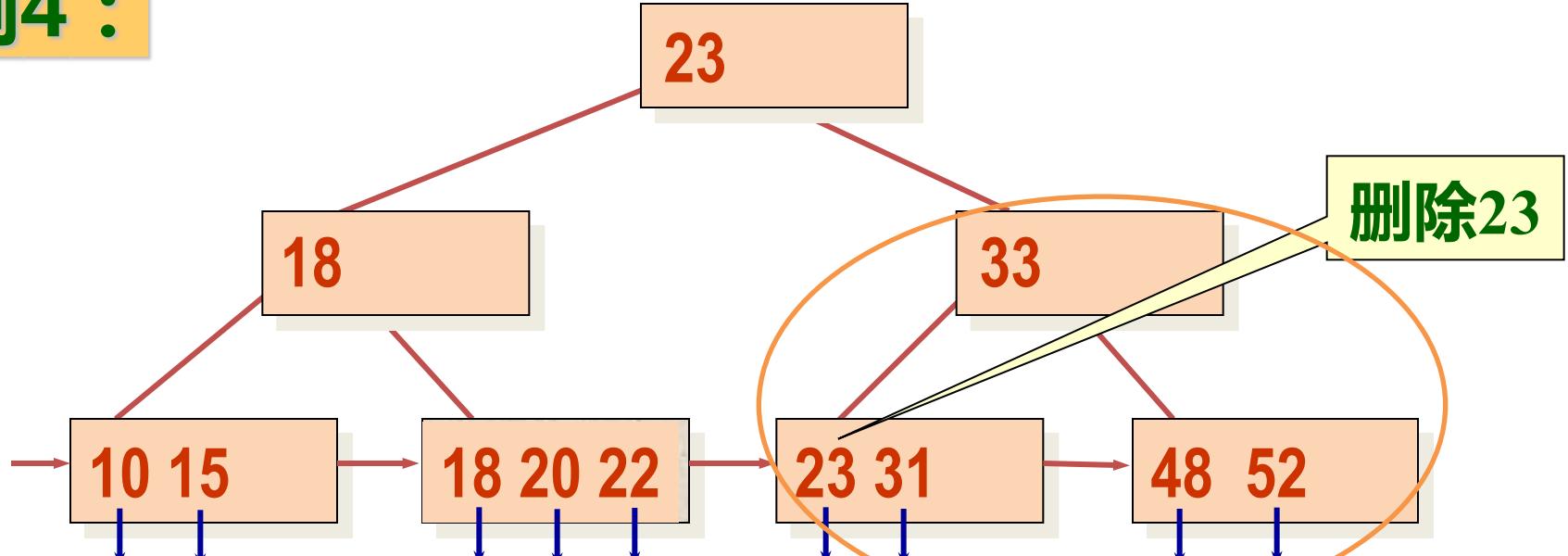




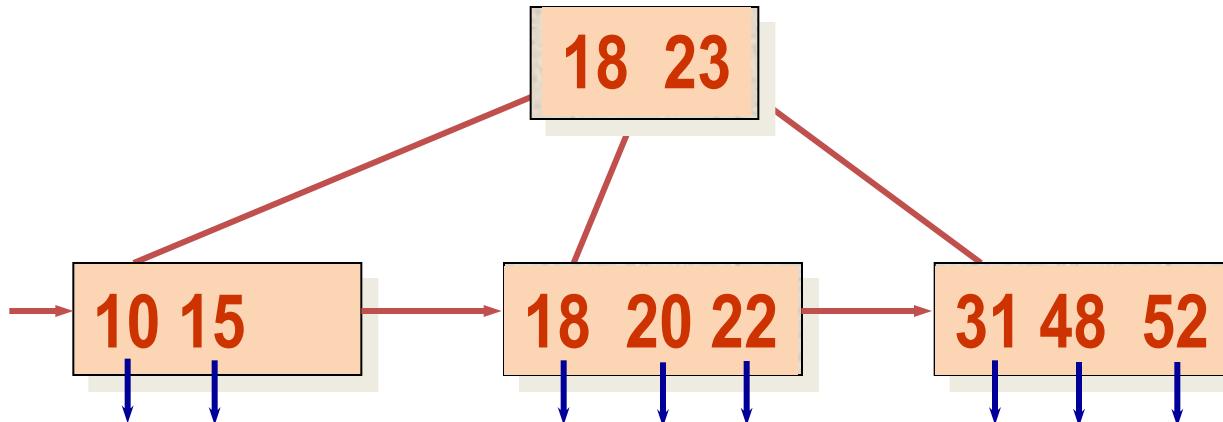
删除33后叶结点合并，从左兄弟借一个分支



例4：



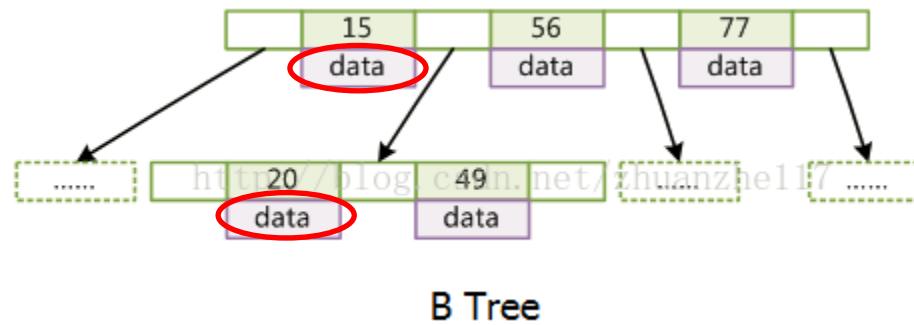
删除关键码23，右边两结点合并，影响到非叶结点合并，可能直到根结点，导致高度减少





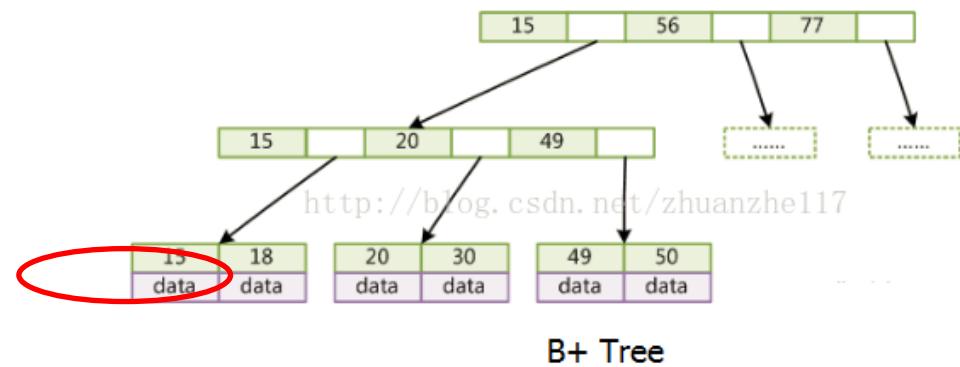
总结

- B树和B+树是文件系统的常用存储结构
- B树的每个结点都存储key和data



B Tree

- B+树只有叶子结点存储data，叶子结点包含了这棵树的所有关键值

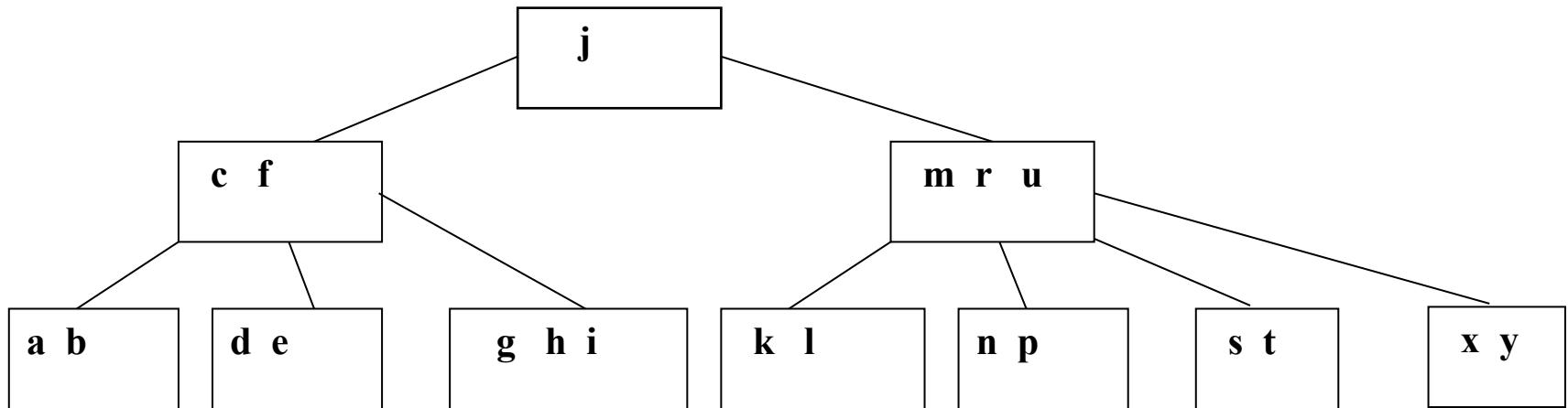
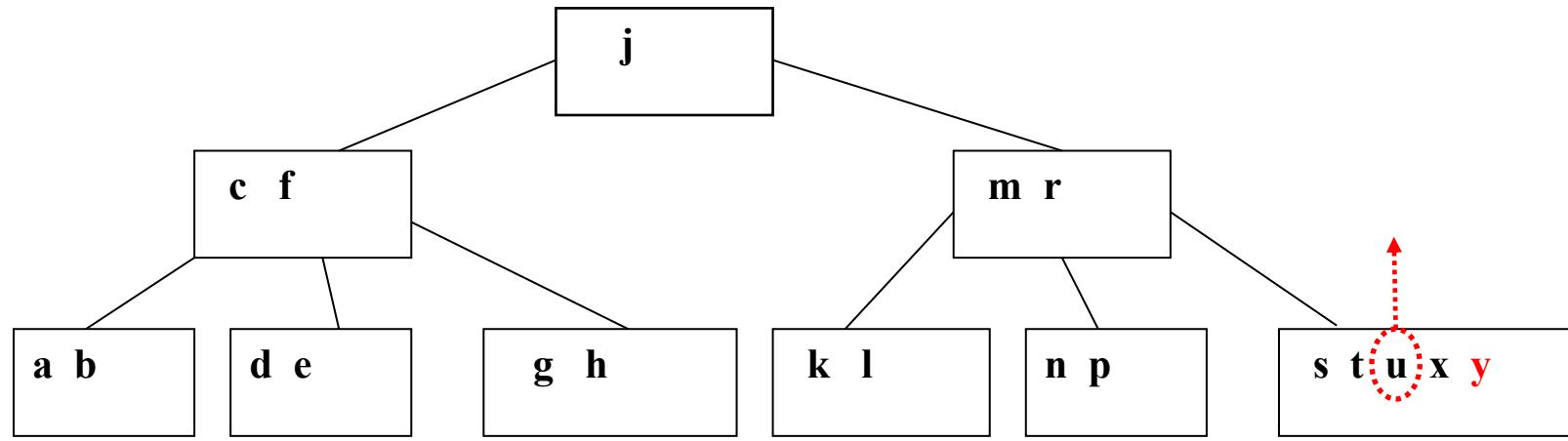


B+ Tree

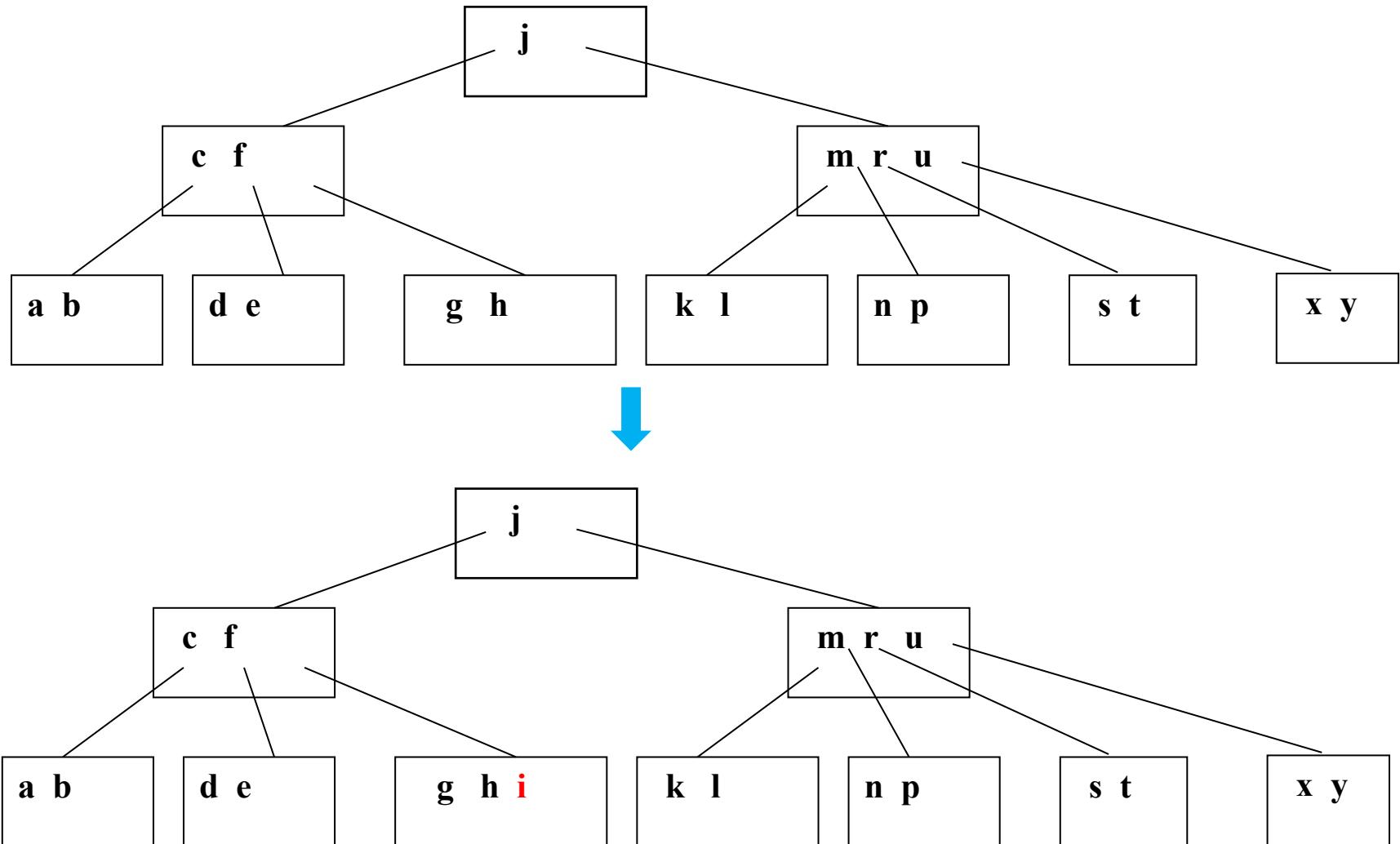


- **B**树的查找从根部开始向下，越靠近根结点的记录查找时间越快，只要找到关键字即可确定记录的存在；
- 随机查找时，**B+**树中每个记录的查找时间基本是一样的，都需要从根结点走到叶子结点。在实际应用中，因为**B+**树的非叶子结点不存放实际的数据，这样每个结点可容纳的元素个数比B树多，树高比B树小。实际应用中**B+**树的性能要好些。

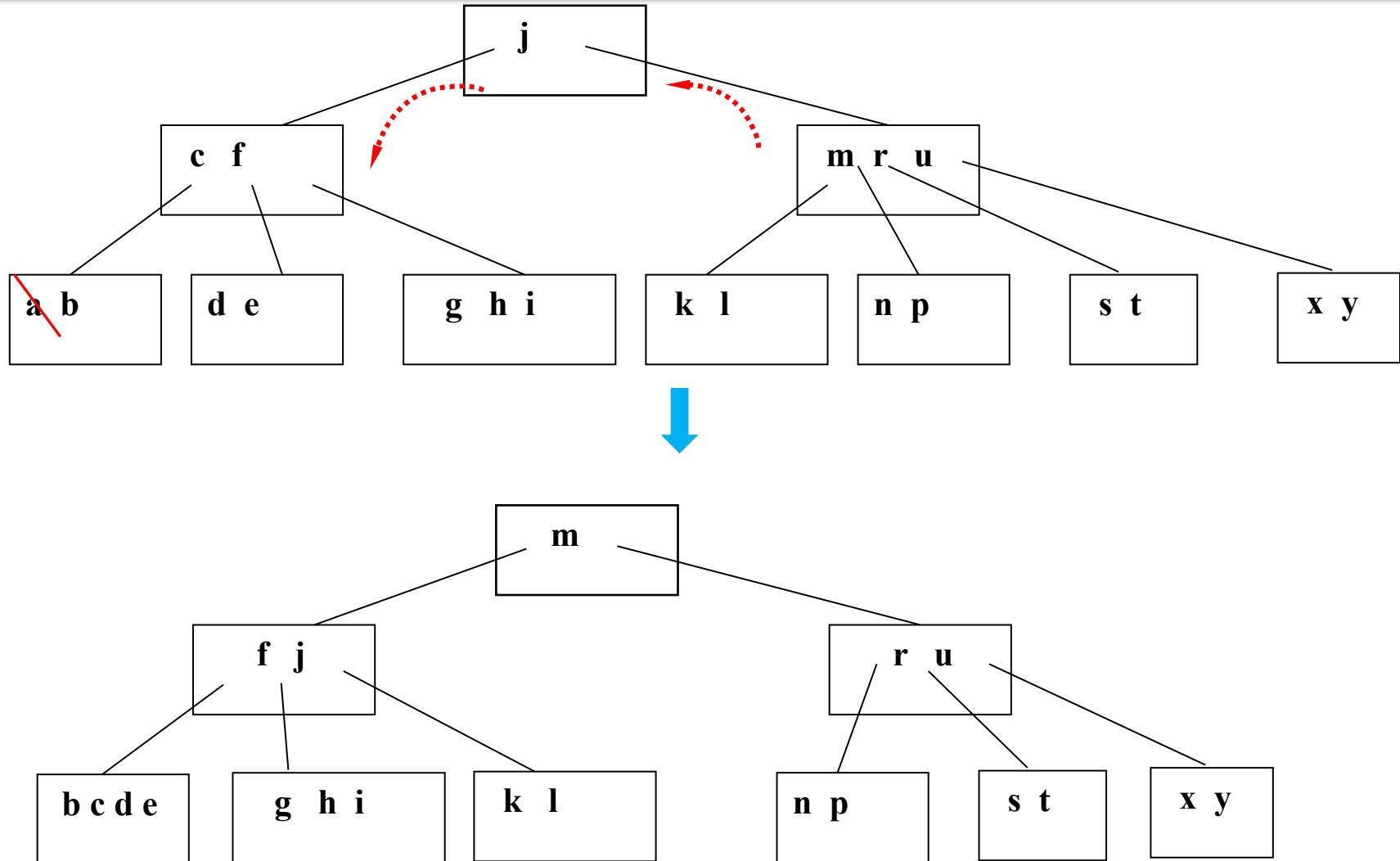
参考答案：(1) 插入y (引起结点分裂)



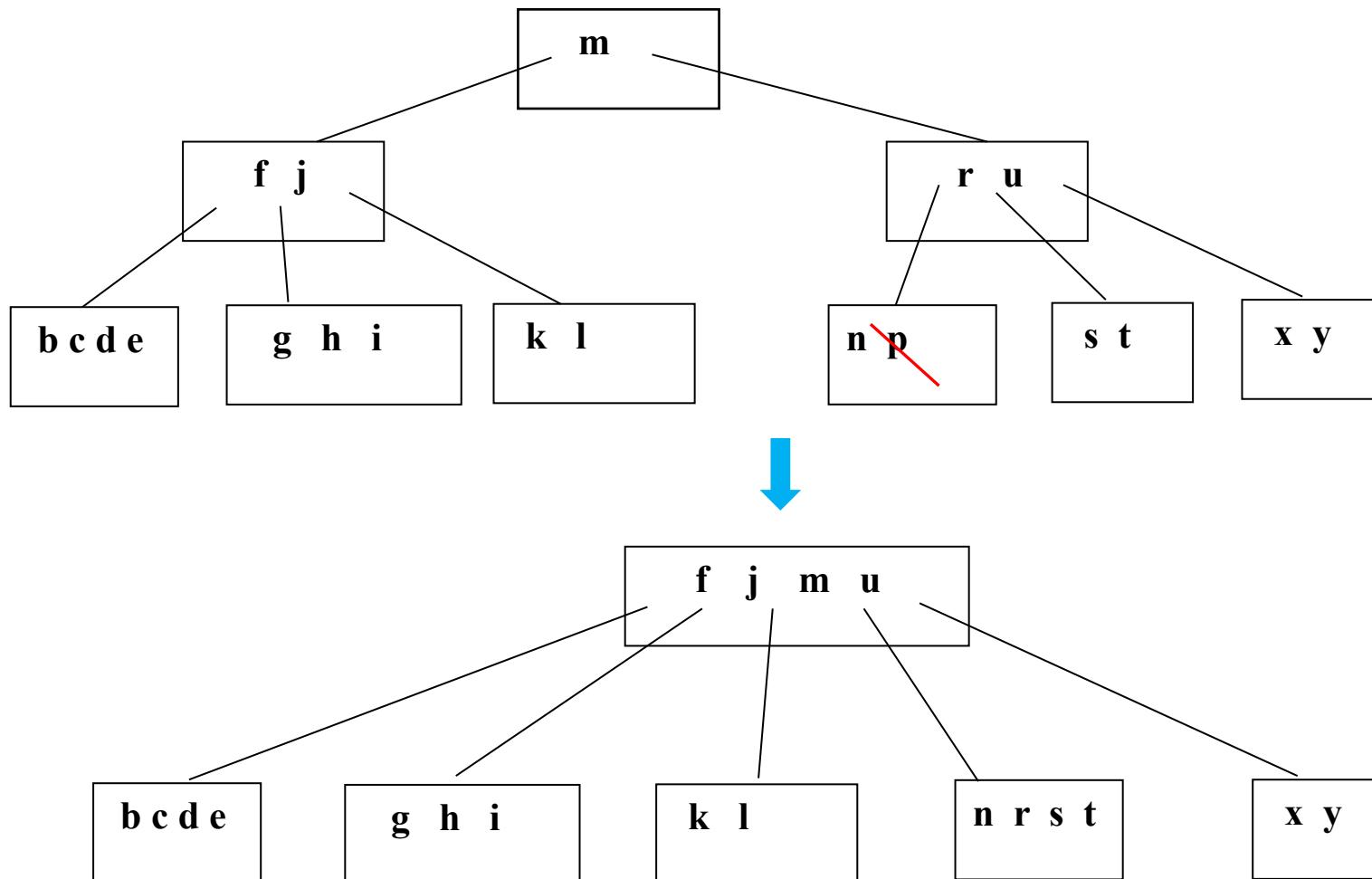
(2) 插入i



(3) 删除a (bcde合并, f所在结点Key不够数, 从兄弟借, j下移一层, m上移一层)



(4) 删除p (n r s t合并, u所在结点Key不够数, 引起f j m u合并)





练习

对下面的5阶B-树，依次执行下列操作，画出各步操作的结果。

- (1)插入y
- (2)插入i
- (3)删除a
- (4)删除p

