第十章 文件、外部排序与搜索

赵建华

南京大学计算机系

文件

- 什么是文件
  - 文件是存储在外存上的数据结构。
  - 文件分操作系统文件和数据库文件
    - 操作系统中的文件是流式文件：是没有结构的字符流
    - 数据库文件是具有结构的数据集合
  - 数据结构中讨论的是数据库文件。
- 操作系统对文件是按物理记录读写的，在数据库中文件按页块存储和读写。

文件的组成

- 文件由记录组成；记录由若干数据项组成。
  - 记录：文件存取的基本单位
  - 数据项：文件可使用的最小单位。
- 逻辑记录：面向用户的基本存取单位
- 物理记录：面向外设的基本存取单位
- 主关键码项：唯一标识一个记录的数据项或数据项集；

文件的结构

- 文件结构包括：逻辑结构、存储结构和操作。
- 逻辑结构是线性结构，各个记录以线性方式排列。
- 存储结构：文件在外存上的组织方式，与文件特性有关。
  - 顺序组织
  - 直接存取组织（散列组织）
  - 索引组织
- 文件的操作：
  - 在逻辑结构上定义，在存储结构上具体实现

顺序文件 (Sequential File)

- 顺序文件：
  - 记录按进入文件的先后顺序存放，
  - 其逻辑顺序与物理顺序一致。
  - 通常可存放在顺序存取设备（如磁带）或直接存取设备（如磁盘）。
    - 当存放在顺序存取设备上时只能按顺序搜索法存取；
    - 存放在直接存取设备上时，可以使用顺序搜索法、折半搜索法等存取。
- 顺序有序文件：文件的记录按主关键码有序
- 顺序无序文件：否则

直接存取文件
(Direct Access File)

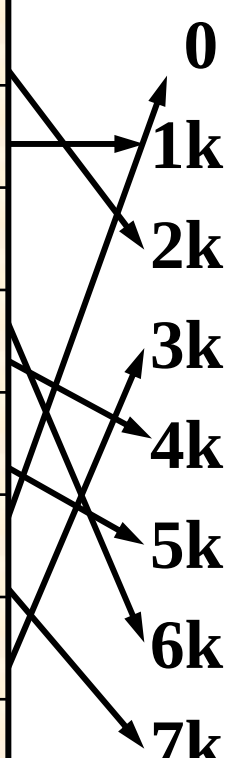
- 记录的逻辑顺序与物理顺序不一定相同
- 可通过关键码可直接确定记录地址
- 利用散列技术组织文件。
- 处理类似散列法，但它是存储在外存上的

索引文件 (Indexed File)

- 索引文件由索引表和主文件组成。

索引表

key	addr
03	2k
08	1k
17	6k
24	4k
47	5k
51	7k
83	0
95	3k



数据表

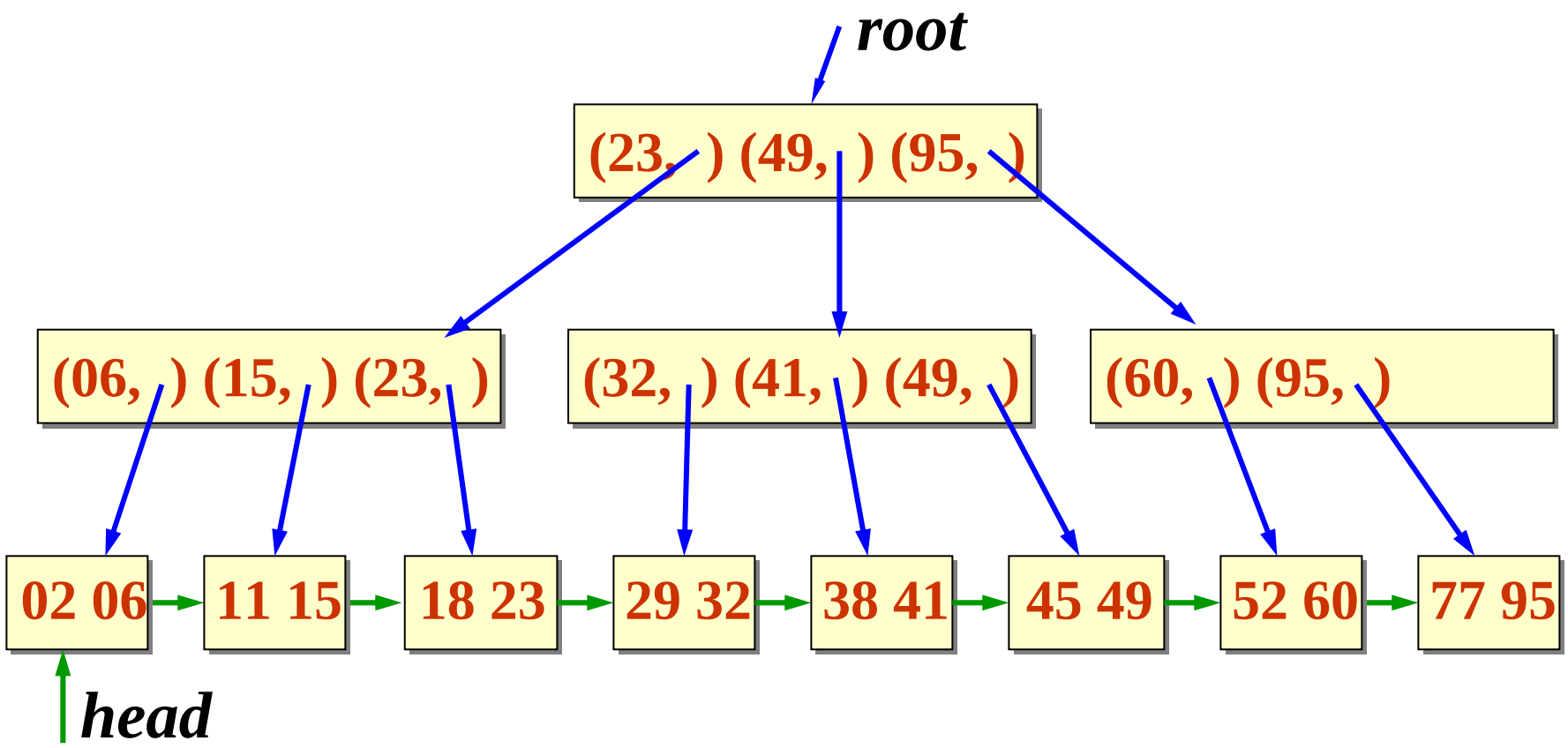
职工号	姓名	性别	职务	婚否	…
83	张珊	女	教师	已婚	…
08	李斯	男	教师	已婚	…
03	王璐	男	教务员	已婚	…
95	刘琪	女	实验员	未婚	…
24	岳跋	男	教师	已婚	…
47	周斌	男	教师	已婚	…
17	胡江	男	实验员	未婚	…
51	林青	女	教师	未婚	…

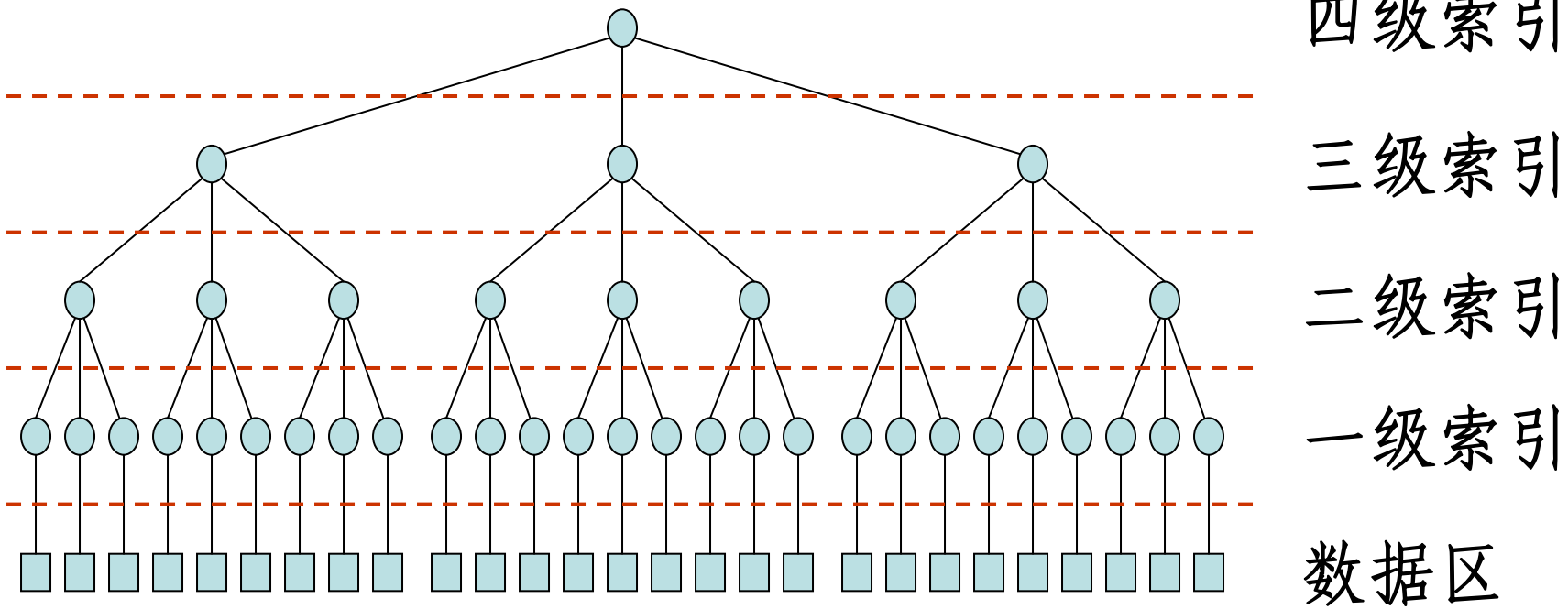
索引文件 (Indexed File)

- 索引表是按关键码有序的表，指示逻辑记录与物理记录间的对应关系
- 对顺序文件的索引：稀疏索引
  - 主文件也按关键码有序。一组记录只需要一个索引项。
- 对非顺序文件的索引：稠密索引
  - 主文件中记录未按关键码有序
  - 每个主文件记录必须对应一个索引项
  - 当对象在外存中按加入顺序存放时，必须采用稠密索引结构

多级索引结构

- 如果文件记录数目大到索引表本身也很大，则需要通过分批多次读取外存才能把索引表搜索一遍
- 解决方法
  - 建立索引的索引（二级索引）。
  - 二级索引的一个索引项对应一个一级索引块
    - 记录该一级索引块的最大关键码及存储地址。
  - 二级索引可以常驻内存，
- 如果二级索引仍不能放入内存
  - 建立二级索引的索引（三级索引）


root
(23,) (49,) (95,)
(06,) (15,) (23,)
(32,) (41,) (49,)
(60,) (95,)
02 06
11 15
18 23
29 32
38 41
45 49
52 60
77 95
head



多级索引结构形成 m 路搜索树

- 多级索引结构形成 m 叉树。
  - 每个分支结点表示一个索引块，最多存放 m 个索引项
  - 每个索引项给出各子树结点 (较低一级索引块) 的最大关键码和结点地址。
  - 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

- 静态索引结构的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。
- 动态索引结构的m路搜索树
  - 在系统运行期间，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 m 路搜索树的递归定义

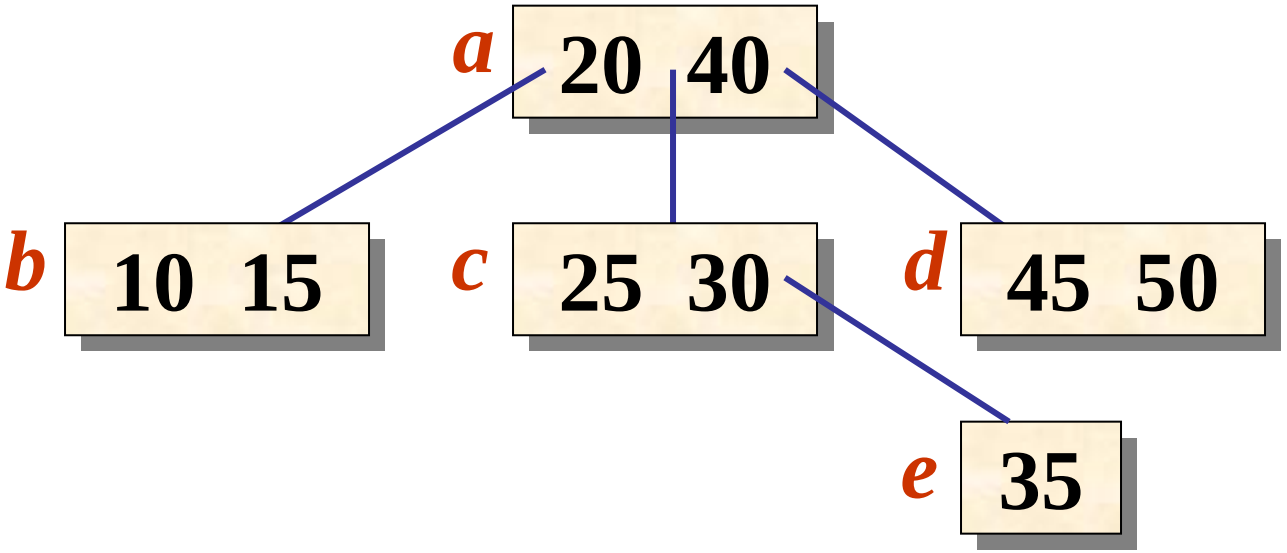
- 空树是m 路搜索树,
- 满足如下性质的树也是m 路搜索树：
  - 根最多有 m 棵子树, 并具有如下的结构：

    $(n, P_0, K_1, P_1, K_2, P_2, \ldots\ldots, K_n, P_n)$

    其中, P_i 是指向子树的指针, $0 \leqslant i \leqslant n < m$;

    K_i 是关键码, $1 \leqslant i \leqslant n < m$。 $K_i < K_{i+1}$, $1 \leqslant i < n$。
  - P_i 中的关键码都小于 K_{i+1}且大于 K_i， $0 < i < n$。 P_n 中的关键码都大于K_n； P_0 中的关键码都小于 K_1。
  - 子树 P_i 也是 m 路搜索树， $0 \leqslant i < n$ 。

一棵3路搜索树的示例



m路搜索树的最大结点数

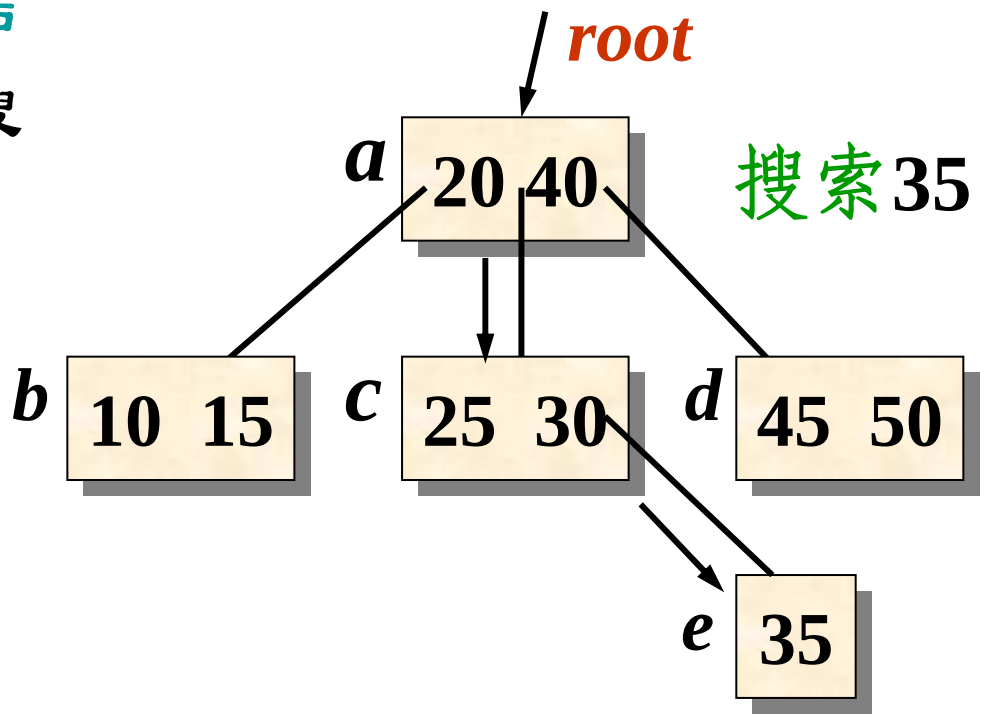
- 若已知 m 路搜索树的度 m 和它的高度 h, 则树中的最大结点个数为:

$$\sum_{i=1}^{h} m^{i-1} = \frac{1}{m-1}\left(m^h - 1\right)$$

  - 每个结点中最多有m-1个关键码，在一棵高度为 h 的m路搜索树中关键码最大个数为m^h-1。
  - 高度h=3的二叉搜索树, 关键码最大数为7;
  - 高度h=4的3路搜索树，关键码最大数为3^4-1 = 80。

m路搜索树的搜索算法

- 在m路搜索树上的搜索过程是一个在结点内搜索和自根结点向下逐个结点搜索的交替的过程。



```
template <class T>
Triple<T> Mtree<T>::Search (const T& x) {
/*用关键码 x 搜索驻留在磁盘上的m路搜索树
  各结点格式为n,p[0],(k[1],p[1]),…,(k[n],p[n]), n < m
  函数返回一个类型为Triple(r,i,tag)的记录。
```

- tag = 0, 表示 x 在结点r中找到, 该结点的k[i]等于x;
- tag = 1, 表示没有找到x, 可插入结点为r, 插入到该结点的k[i]与k[i+1]之间。

```
  */
```

```
    Triple result;                          //记录搜索结果三元组
    GetNode (root);                         //从盘上读取结点root
    MtreeNode<T> *p = root, *q = NULL;  //p是扫描指针,q是父结点指针

    while (p != NULL) {    //从根开始检测，直到叶子结点
        int i = 0;   p->key[(p->n)+1] = MaxValue;  //结束标记
        while (p->key[i+1] < x) i++;          //在结点内搜索
        //循环结束时，i指向结点中最后一个小于x的元素。

        if (p->key[i+1] == x) {      //结点内搜索成功
                result.r = p;  result.i = i+1;  result.tag = 0;
                return result;
        }
        q = p;  p = p->ptr[i];          //此时p->key[i+1]必大于x
                //本结点无x, q记下当前结点, p下降到子树
        GetNode(p);                     //从磁盘上读取结点p
    }
    result.r = q;  result.i = i;  result.tag = 1;
    return result;              //搜索失败,返回插入位置
};
```

m路搜索树的效率考虑

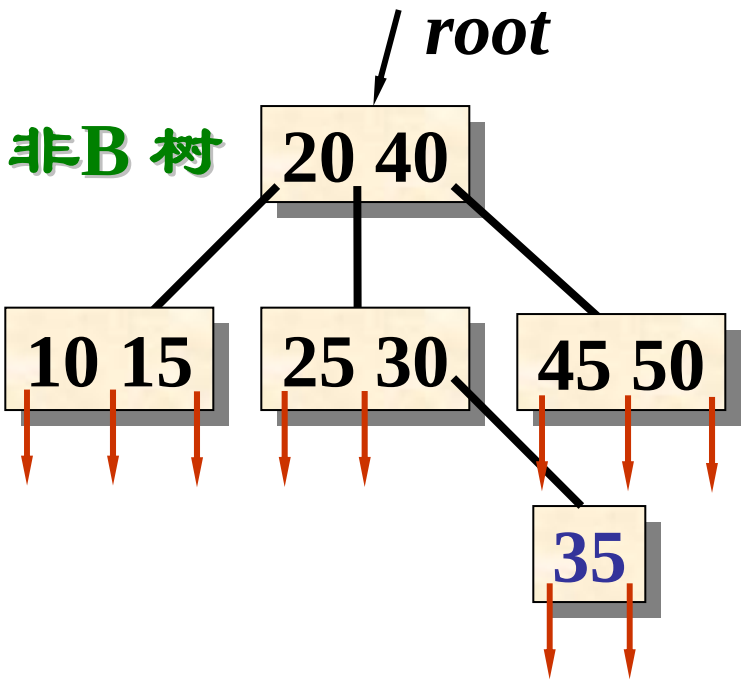
- 提高搜索树的路数m, 可改善搜索性能。
  - 但m过大可导致一个结点不能一次读入内存
  - 也导致读入一个结点的时间增加
- 对于给定的m
  - 当搜索树平衡时， m 路搜索树的性能接近最佳。
  - B树： 一种平衡的m路搜索树。

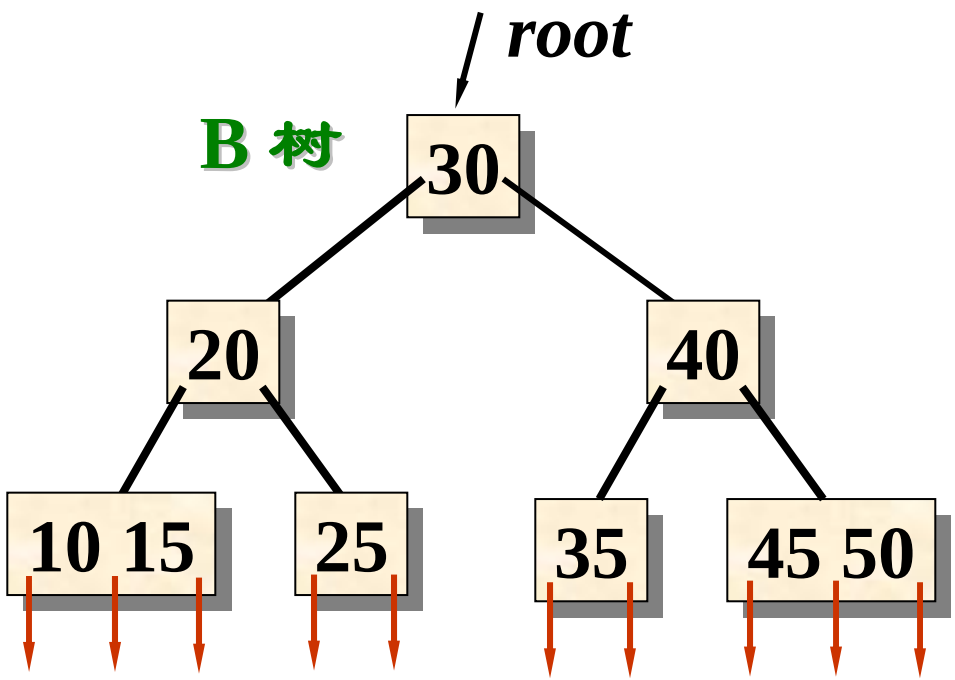
B 树

- 一棵 m 阶B 树是一棵平衡的 m 路搜索树
  - 或者是空树，
  - 或者是满足下列性质的树：
    - ✓根结点至少有 2 个子女。
    - ✓除根结点以外的所有结点 (不包括失败结点)至少有$\lceil m/2 \rceil$个子女。
    - ✓所有的失败结点都位于同一层。
- “失败”结点
  - 当 x 不在树中时到达的结点。这些结点实际不存在，指向它们的指针为NULL。
  - 它们不计入树的高度。

- *m*阶*B*树是特殊的*m*路搜索树。
  - 原来m路搜索树定义中的规定在m阶B树中都保留。
  - 额外的规定是为了提高搜索效率，避免退化。
- 在B 树的每个结点包含有一组指针recptr[*m*+1]，指向实际记录的存放地址。
  - key[*i*]与recptr[*i*] 形成索引项 (key[*i*], recptr[*i*]),
  - 通过key[*i*]可找到某个记录的存储地址recptr[*i*]。
- 在讨论B树结构的操作时先不涉及recptr[*i*]，因此在后续讨论中该指针不出现。

- 一棵B 树是平衡的 m 路搜索树，但一棵平衡的 m 路搜索树不一定是B 树。





B树类和B树结点类的定义

```
template <class T>
class Btree : public Mtree<T> {          //B树类定义
//继承m叉搜索树的所有属性和操作,
//Search从Mtree继承,  MtreeNode直接使用
public:
    Btree();                             //构造函数
    bool Insert (const T& x);            //插入关键码x
    bool Remove (T& x);                  //删除关键码x
};
```

B 树的搜索算法

- B树的搜索算法就是m路搜索树Mtree上的搜索算法。
  - 交替进行*结点内搜索*以及*循某一路径向下搜索*
  - 搜索成功：报告结点地址及在结点中的关键码序号；
  - 搜索不成功：报告最后停留的叶结点地址及新关键码在结点中可插入的位置。
- B 树的搜索时间与B 树的阶数 m 和B 树的高度 h直接有关, 必须加以权衡。

- 在B 树上进行搜索，
  - 搜索成功所需时间：关键码所在的层次；
  - 搜索不成功所需时间：树的高度。
- B树高度h与关键码个数N的关系如下：
  - 让B树每层结点个数达到最大，得到最小高度。每个结点有m个子结点
  - $N \leqslant (m^h-1)$ ➔

$$h \geqslant \lceil \log_m(N+1) \rceil$$

- 让m阶B树中每层结点个数最少，高度达到最大。设树中关键码个数为N，从B 树的定义知：
  - ✓ 1层：1个结点
  - ✓ 2层：至少2个结点
  - ✓ 3层：至少$2\lceil m/2 \rceil$个结点
  - ✓ 4层：至少$2\lceil m/2 \rceil^2$个结点
  - ✓ 如此类推，……
  - ✓ h层：至少有$2\lceil m/2 \rceil^{h-2}$个结点。
- 失败结点在第h+1层，失败结点个数为N+1。
  - – 关键码有N个，而失败数据与已有关键码交错排列。

- 因此，有

  ✓$N+1$ = 失败结点数 = 位于第$h+1$层的结点数

  $$\geqslant 2\lceil m/2\rceil^{h-1}$$

  $$\therefore\ N\geqslant 2\lceil m/2\rceil^{h-1}-1$$

  $$\therefore\ h-1\leqslant \log_{\lceil m/2\rceil}((N+1)/2)$$

  $$\therefore\ h\leqslant \log_{\lceil m/2\rceil}((N+1)/2)+1$$

- 示例：若B 树的阶数 $m = 199$, 关键码总数 $N = 1999999$，则B 树的高度 h 不超过

  $$\log_{100} 1000000 +1= 4$$

m值的选择

- 如何选择m值，使得在B 树中找到关键码 x 的时间总量达到最小。
- 这个时间由两部分组成：
  - ✓从磁盘中读入结点(耗时)
  - ✓在结点内搜索 x(较快)
- 因此搜索时间和高度相关
  - ✓提高B 树的阶数 m →
  - ✓减少高度→
  - ✓减少读入结点的次数, →
  - ✓减少读磁盘的次数→
  - ✓减少搜索时间

✓但是，m 受到内存可使用空间的限制。

B 树的插入

- B 树中，每个非失败结点的关键码个数

$$[\lceil m/2 \rceil - 1, m-1]$$

- 插入的过程在某个叶结点开始。
  - 如果插入后关键码个数超出m-1，则“分裂”结点；否则不需要其它操作；
  - 叶结点的分裂可能引起父结点的分裂；...
  - 分裂过程可能逐步向上传播。

节点分裂的原则

- 在已经有 m-1 个关键码的结点 p 中再插入一个关键码后：

$$(m, P_0, K_1, P_1, K_2, P_2, \ldots\ldots, K_m, P_m)$$

- 结点 p 分裂成两个结点 p 和 q：
  - 结点 p：

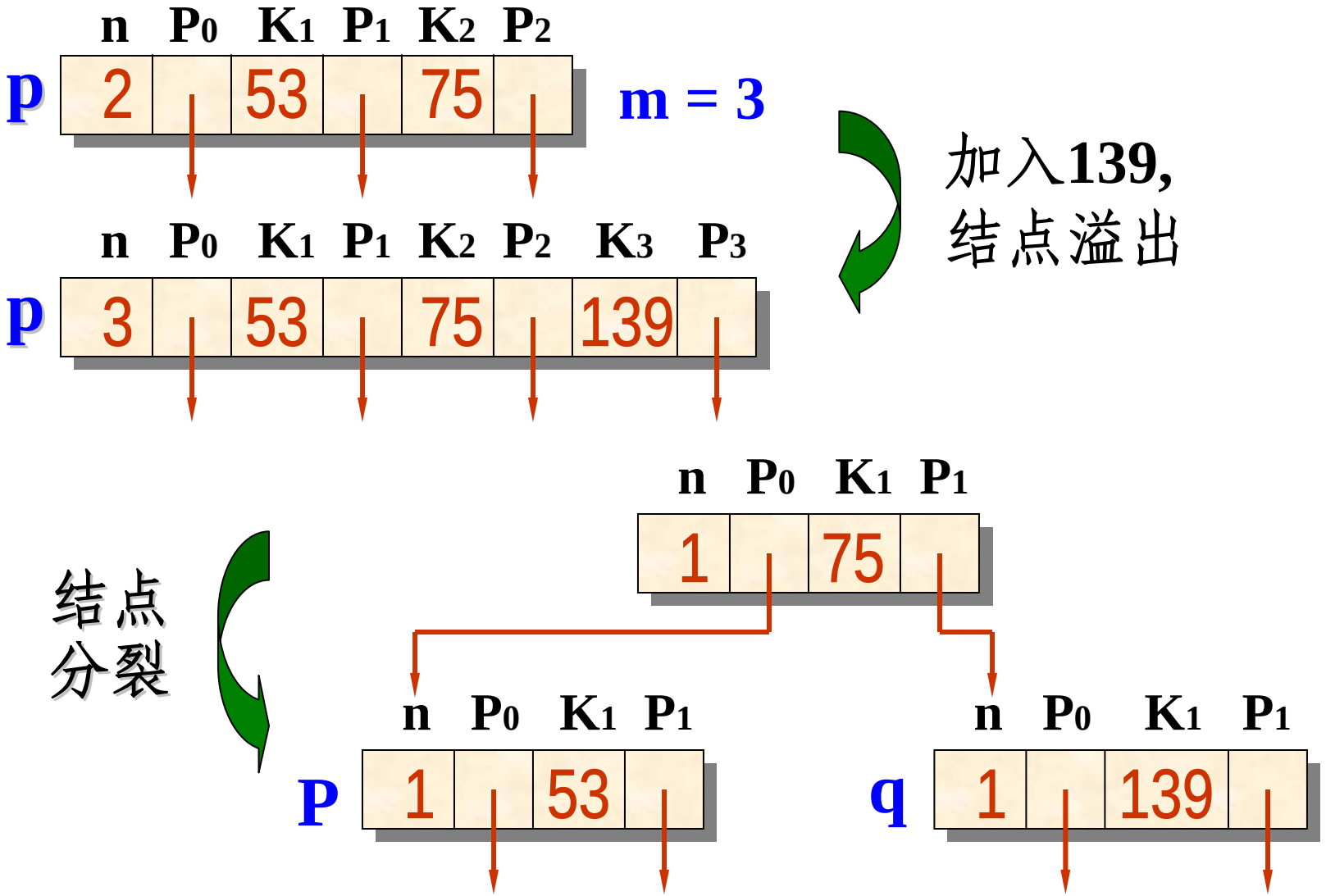
$$(\lceil m/2 \rceil -1, P_0, K_1, P_1, \ldots\ldots, K_{\lceil m/2 \rceil -1}, P_{\lceil m/2 \rceil -1})$$

  - 结点 q：

$$(m-\lceil m/2 \rceil, P_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil +1}, P_{\lceil m/2 \rceil +1}, \ldots\ldots, K_m, P_m)$$

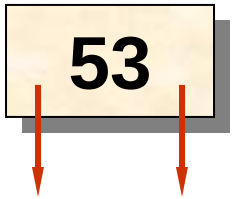
- 中间的关键码 $K_{\lceil m/2 \rceil}$ 与指向新结点 q 的指针形成一个二元组 $(K_{\lceil m/2 \rceil}, q)$，插入到父结点中去。
- 父结点中关键码加一，可能引起父结点的进一步分裂

结点“分裂”的示例

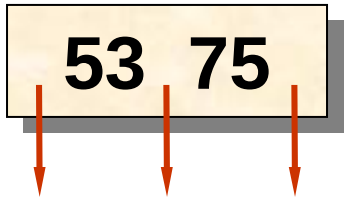

n P0 K1 P1 K2 P2
p 2 53 75
m = 3
加入139,
结点溢出
n P0 K1 P1 K2 P2 K3 P3
p 3 53 75 139
结点
分裂
n P0 K1 P1
1 75
n P0 K1 P1
P 1 53
n P0 K1 P1
q 1 139

示例:从空树开始建立3阶B树

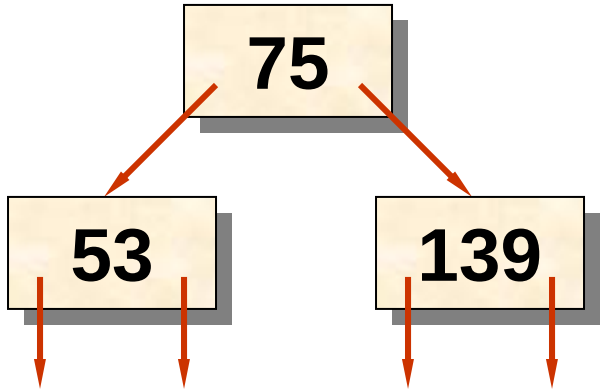
n=1 加入53



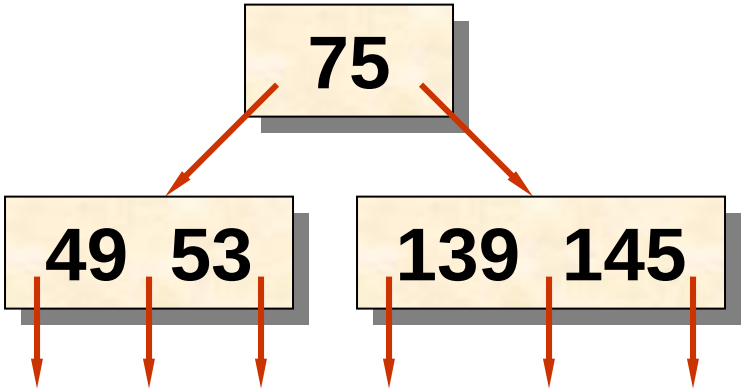
n=2 加入75



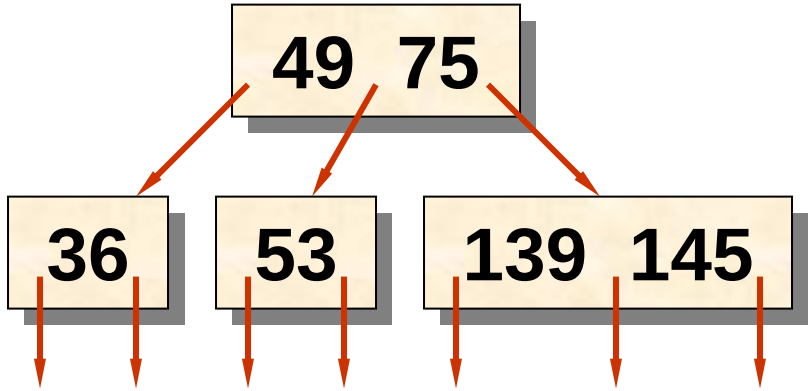
n=3 加入139

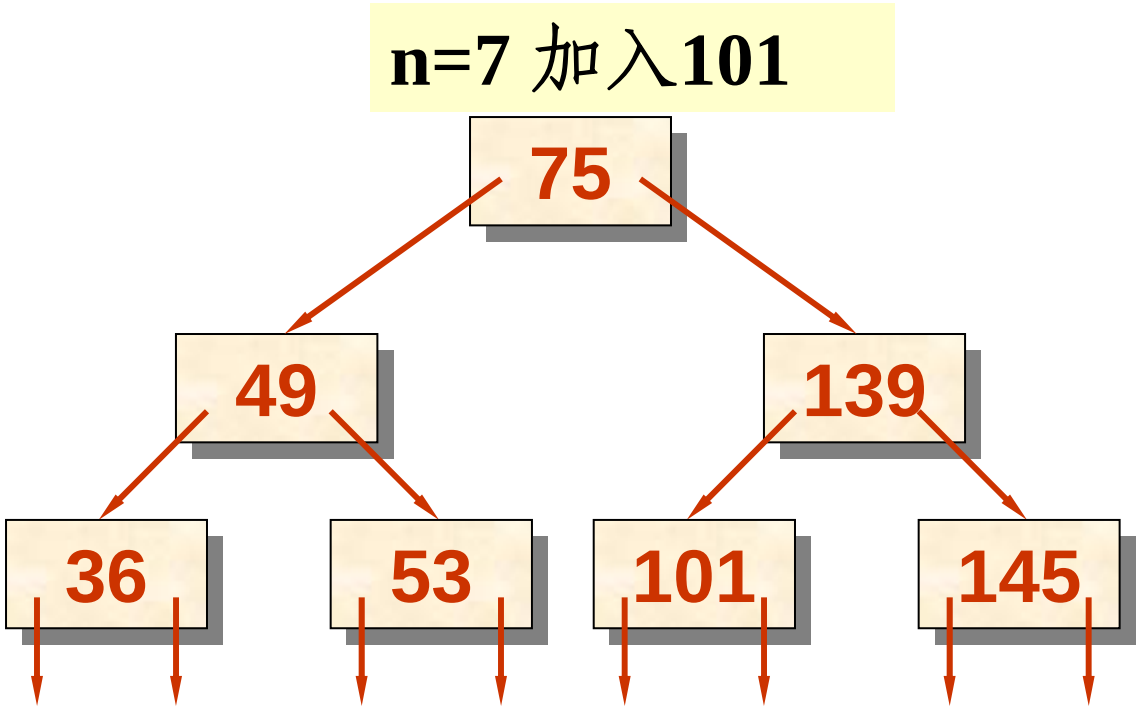


n=5 加入49,145



n=6 加入36





设B 树高h,
则搜索时最多
读盘h 次。

- 在插入时，需要首先进行搜索，然后可能自底向上地分裂结点
- 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 树的插入算法

```
template <class T>
bool Btree<T>::Insert (const T& x) {
//将关键码x插入到一个m阶B树中。
    Triple<T> loc = Search(x);        //搜索x的插入位置
    if (!loc.tag) return false;       //x已存在, 不插入

    MtreeNode<T> *p = loc.r, *q;      //r是插入结点
    MtreeNode<T> *ap = NULL, *t;      //ap是右邻指针
```

```
T K = x;  int j = loc.i;

//(K,ap)形成二元组，  插入到p指向的结点中
while (1) {
    if (p->n < m-1) {              //简单情况，
                                   //关键码个数未超出
        insertkey (p, j, K, ap);//插入，且p->n加1
        PutNode (p);               //输出结点p
        return true;}

    int s = (m+1)/2;               //准备分裂结点
    insertkey (p, j, K, ap);       //先插入
    q = new MtreeNode<T>;          //建立新结点q

    //把p中结点第s到第m个关键码
    //移动到q
    move(p, q, s, m);
```

```
K = p->key[s];   ap = q;
        //(K,ap)形成新二元组，插入到p的父结点

if (p->parent != NULL) {          //非根的情况
        t = p->parent;  GetNode(t);  //读取父结点t
        j = 0;
        t->key[(t->n)+1] = MAXKEY;//设监视哨
        while (t->key[j+1] < K) j++;     //顺序搜索
        q->parent = p->parent;    //新结点的双亲
        PutNode(p);  PutNode(q);//输出结点到外存
        p = t;              //p上升到双亲，继续调整
}
//else 处理根结点的分裂，见下页
```

```
        else {
                //原来p指向根结点, 需要要产生新根
                root = new MtreeNode<T>;
                root->n = 1;  root->parent = NULL;
                root->key[1] = K;
                root->ptr[0] = p;  root->ptr[1] = ap;
                q->parent = p->parent = root;
                PutNode(p);  PutNode(q);  PutNode(root);
                return true;
        }
    }//end of while(2)
};
```

- 当分裂一个非根的结点时需要向磁盘写出 2 个结点，当分裂根结点时需要写出 3 个结点。

算法效率分析

- 假设
  - 内存工作区足够大，向下搜索时读入的结点在插入完成前不需要再次读入
- 在完成插入操作时读/写磁盘的最大次数

  找插入（叶）结点向下读盘次数 +

  + 分裂非根结点时写盘次数 +

  + 分裂根结点时写盘次数 =

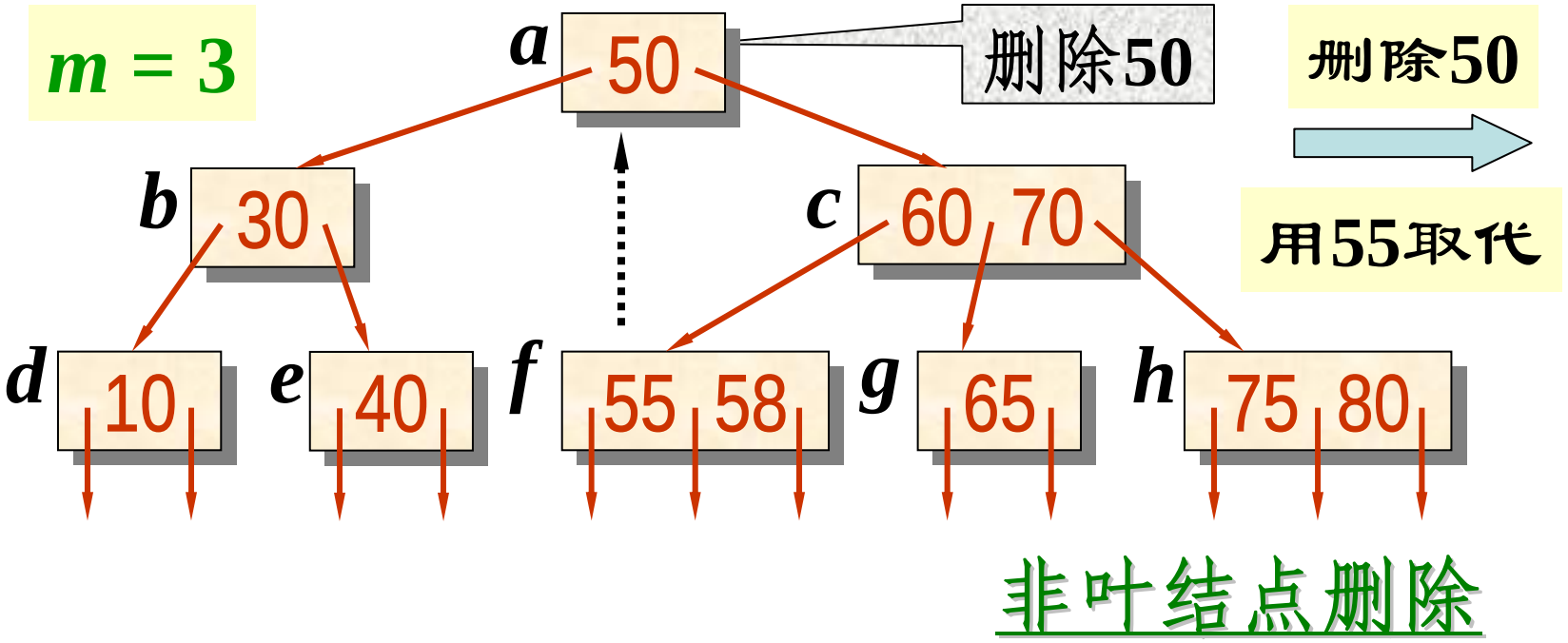
  = $h+2(h-1)+3 = 3h+1$。

B 树的删除

- 在B树中删除一个关键码分成两大步骤：
  - 第一步：
    - 找到并删除关键码。
    - 通过数据移动，换为叶结点上的删除操作。
  - 第二步：
    - 如果叶子结点的关键码数量低于下界，通过合并调整二叉树，使之满足B树的要求。

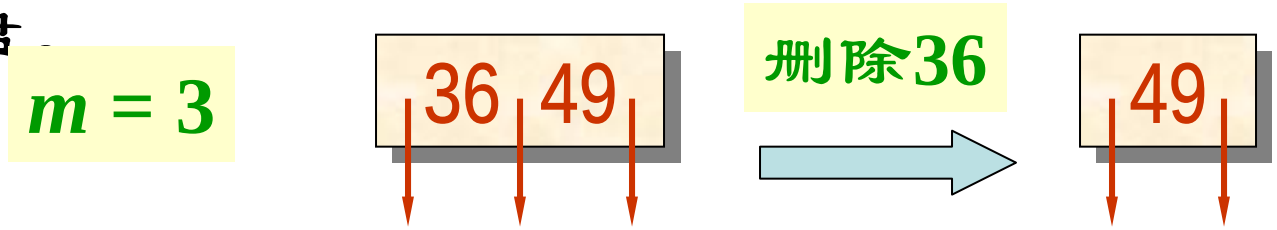
B 树的删除

- 第一步删除过程
  - ✓找到关键码所在结点，删去这个关键码。
  - ✓若所在结点不是叶结点，且被删关键码为 K_i，$1 \leqslant i \leqslant n$,
    - ✓将该结点的P_i所指子树中的最小关键码 x 移动到 K_i所在的位置;
    - ✓然后在 x 所在的叶结点中删除 x。
- 实际上把所有的删除操作变成了叶结点上的删除操作。

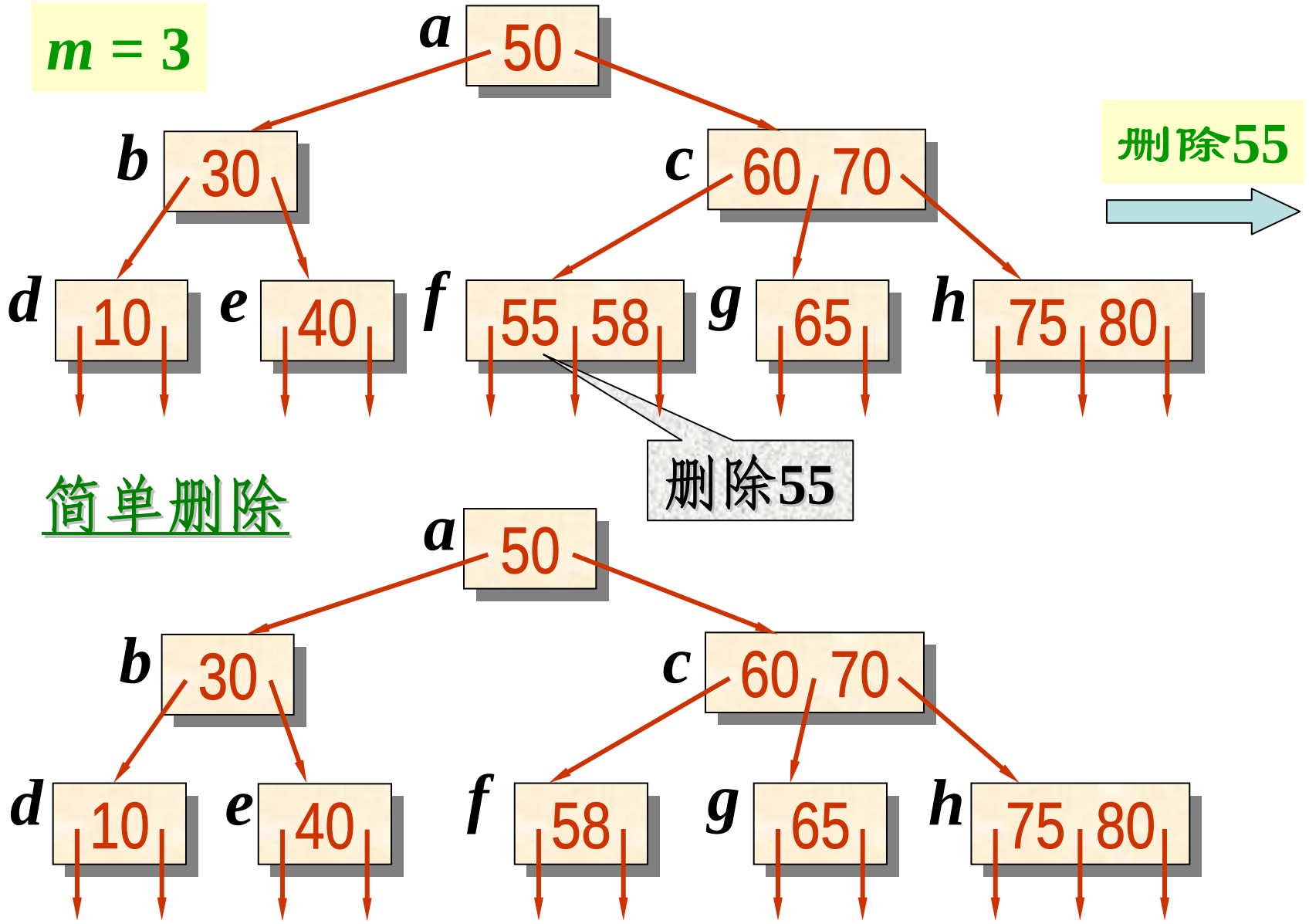

m = 3
a
50
删除50
删除50
用55取代
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
非叶结点删除

叶结点的删除

1. 被删关键码在根结点，且删除前关键码个数$n \geqslant 2$，则直接删去该关键码,并写回磁盘。



2. 被删关键码不在根结点，且删除前关键码个数 $n \geqslant \lceil m/2 \rceil$，则直接删去该关键码，写回磁盘。


m = 3
a
50
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
删除55
删除55
简单删除
a
50
b
30
c
60 70
d
10
e
40
f
58
g
65
h
75 80

被删关键码所在叶结点删除前关键码个数 $n = \lceil m/2 \rceil - 1$，则需要进行调整。

a)向兄弟结点借关键码：右兄弟 (或左兄弟) 结点的关键码个数 $n \geqslant \lceil m/2 \rceil$；

- 从父结点获取一个关键码；
- 再从兄弟结点移动一个关键码到父结点；
- 这个过程不会向上传递。

b)兄弟也没有余粮：右兄弟 (或左兄弟) 结点的关键码个数 $n = \lceil m/2 \rceil - 1$；

- 合并两个兄弟结点。
- 父结点中两个兄弟之间的关键码一起移动到新结点。
- 父结点的关键码数量减少，可能会引起进一步的调整！

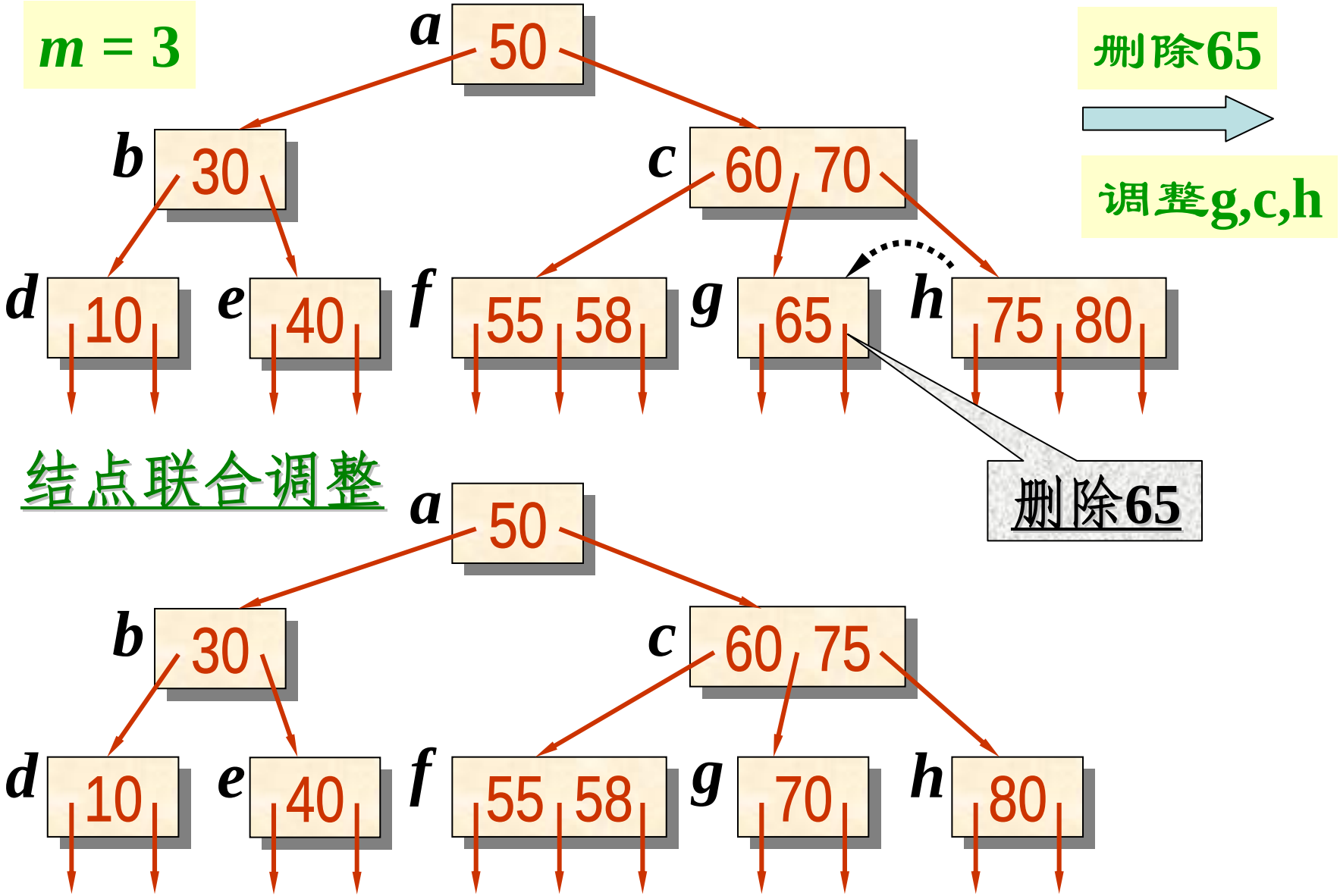
3. 所在叶结点删除前关键码个数 $n=\lceil m/2\rceil-1$, 。右兄弟 (或左兄弟) 结点的关键码个数 $n\geqslant\lceil m/2\rceil$:

a) 父结点中刚大于 (或小于) 被删关键码的关键码 K_i $(1\leqslant i\leqslant n)$ 下移;

b) 右兄弟 (或左兄弟) 结点中最小 (或最大) 关键码上移到父结点中K_i位置;

c) 右兄弟 (或左兄弟) 结点中最左 (或最右) 子树指针平移到被删关键码所在结点中最后 (或最前) 子树指针位置;

d) 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。


m = 3
a
50
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
删除65
调整g,c,h
结点联合调整
删除65
a
50
b
30
c
60 75
d
10
e
40
f
55 58
g
70
h
80

4. 所在叶结点删除前关键码个数 $n = \lceil m/2 \rceil - 1$，若右兄弟 (或左兄弟) 结点的关键码个数 $n = \lceil m/2 \rceil - 1$。设它们的父结点是p，两个结点分别是$P_i$与 P_{i+1}。按如下方法合并结点。

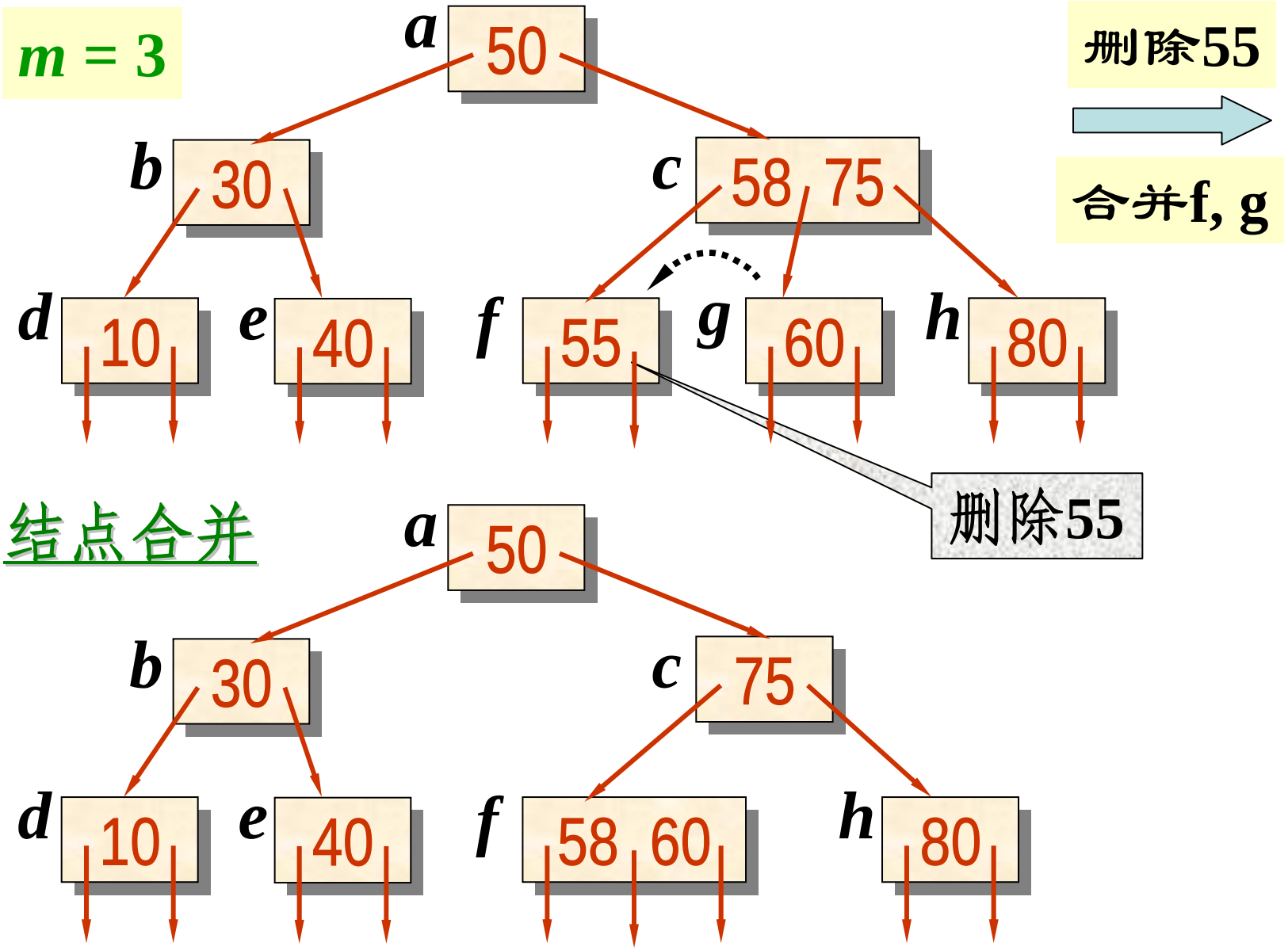
a) 保留 P_i 所指结点, 把 p 中的关键码 K_{i+1}下移到 P_i 所指的结点中。

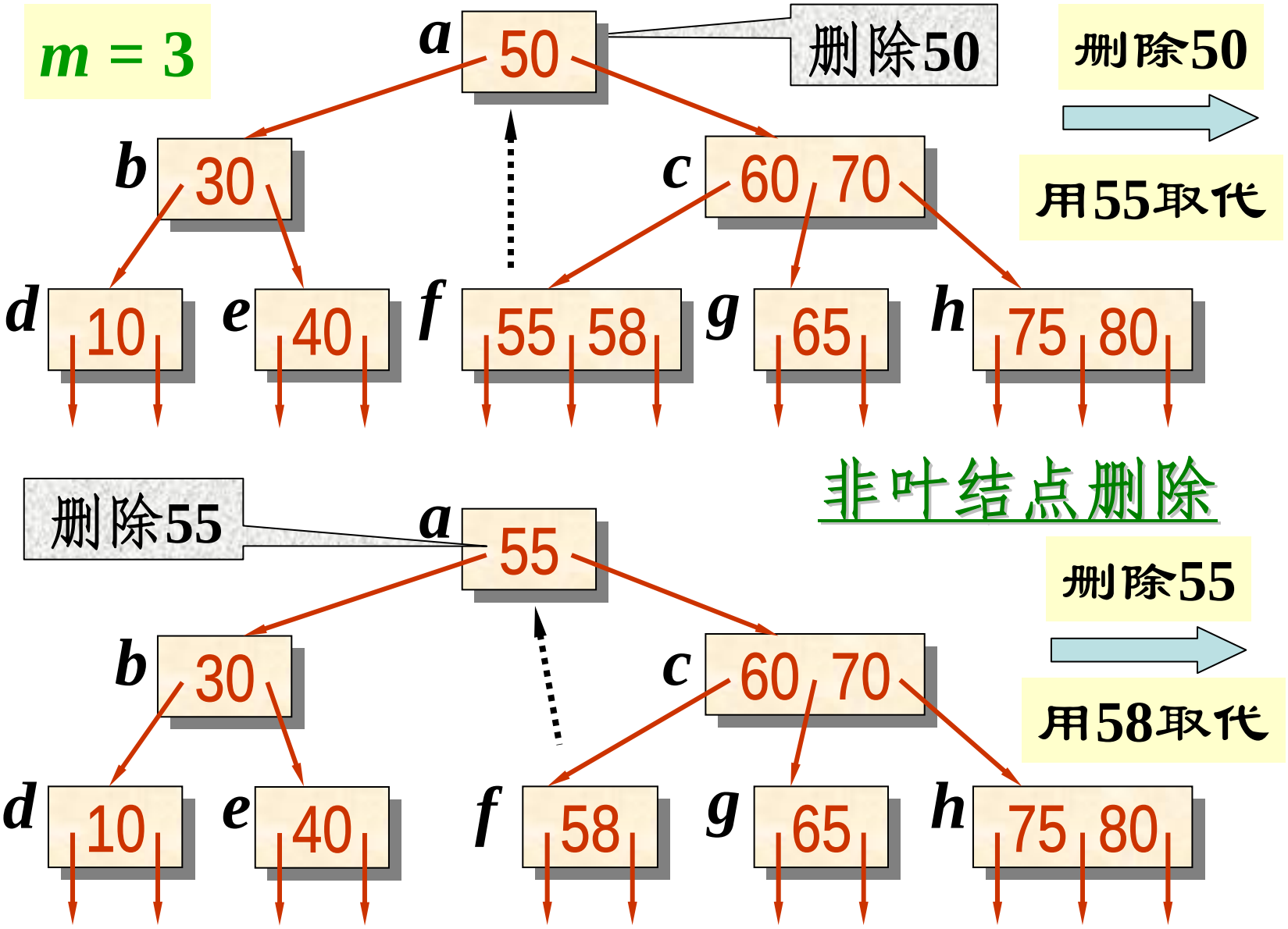
b) 把 P_{i+1} 所指结点中的全部指针和关键码都照搬到 P_i 所指结点的后面。删去 P_{i+1} 所指的结点。

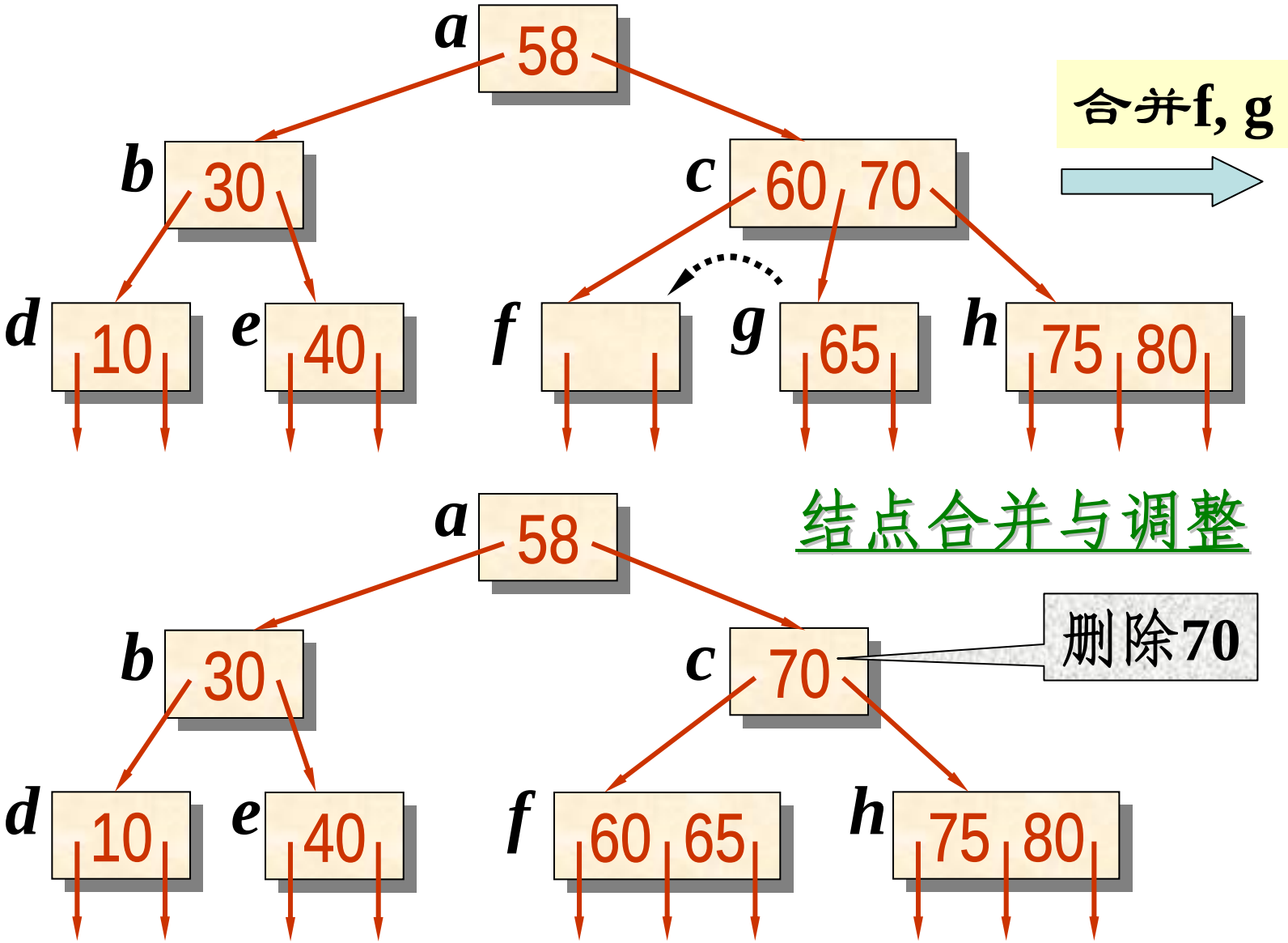
c) 移动结点 p 中后面的关键码和指针，填补因删除关键码 K_{i+1} 和指针 P_{i+1}留下的空位。

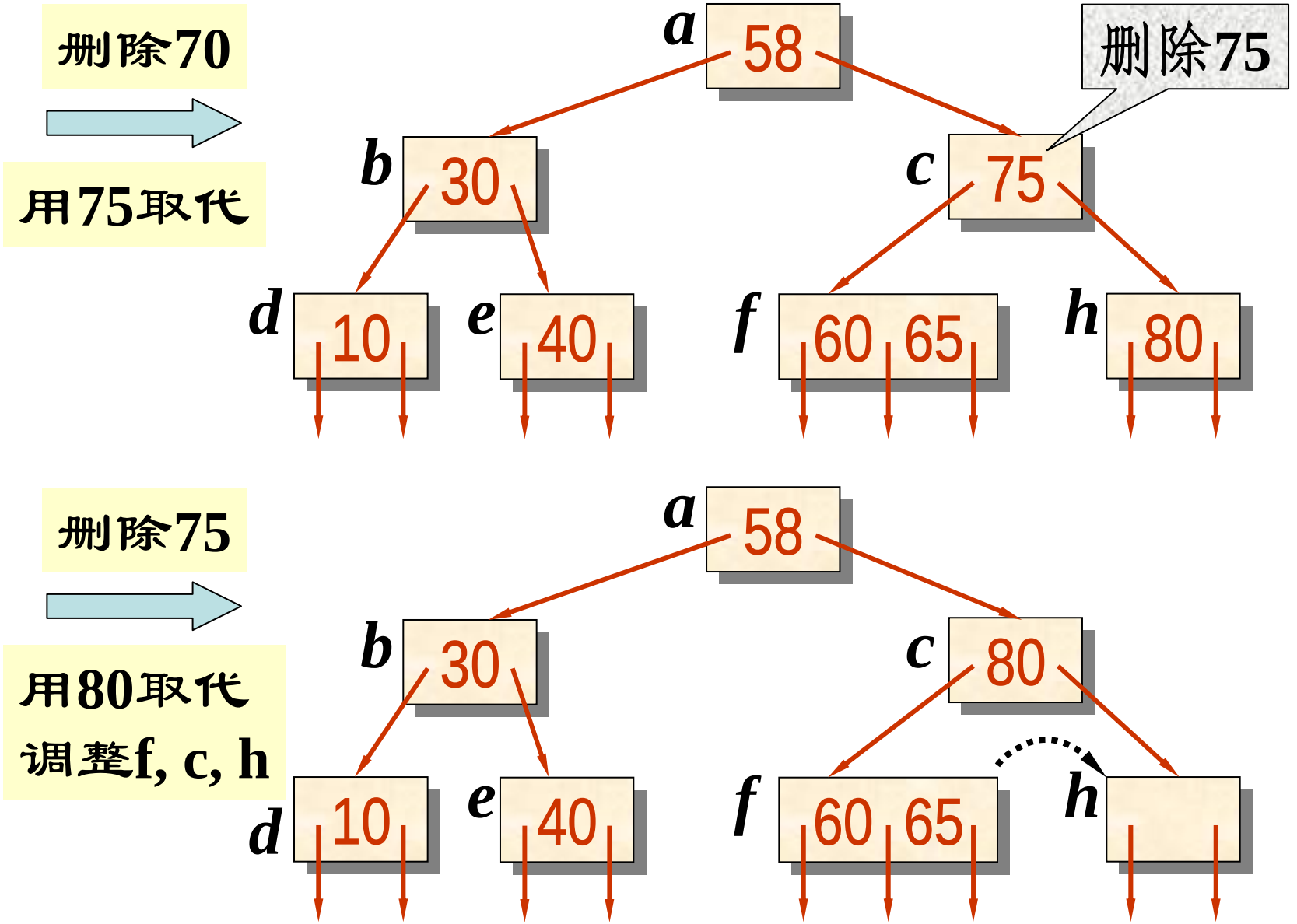
d) 修改结点 p 和选定保留结点的关键码个数。

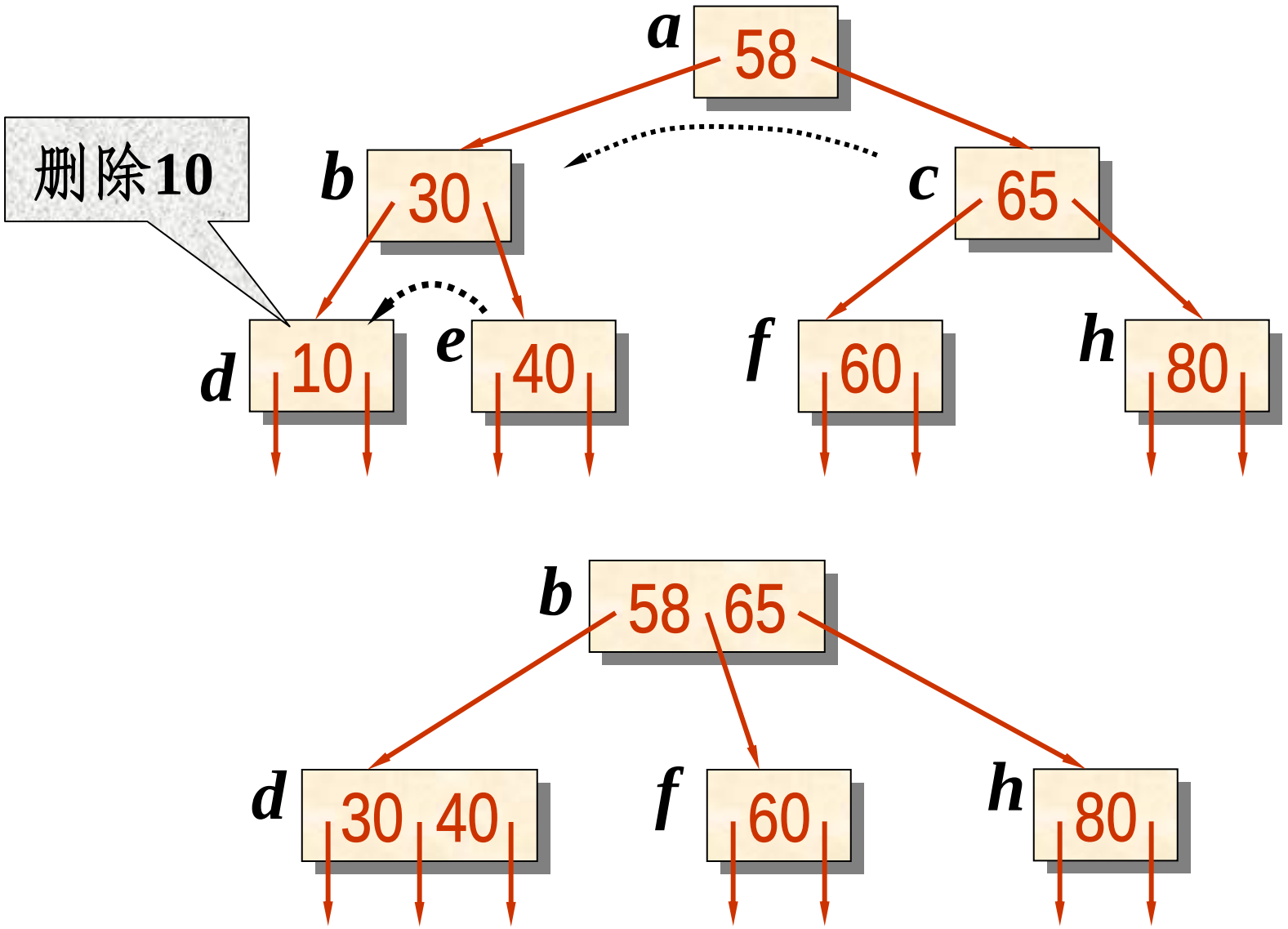
- 双亲结点中的关键码个数减少了，可能引起进一步合并和移动。
- 若双亲结点是根结点
  - 如果结点关键码个数减到 0：删去双亲结点, 合并得到的结点成为新的根结点;
  - 否则将双亲结点与合并后保留的结点都写回磁盘。删除处理结束。
- 若双亲结点不是根结点
  - 如果关键码个数减到$\lceil m/2 \rceil -2$，又要进行下一轮调整。最坏情况下这种结点合并处理要自下向上直到根结点。
  - 否则将需要的结点写回磁盘。删除处理结束。


m = 3
a
50
b
30
c
58 75
d
10
e
40
f
55
g
60
h
80
删除55
合并f, g
删除55
结点合并
a
50
b
30
c
75
d
10
e
40
f
58 60
h
80


m = 3
a
50
删除50
删除50
用55取代
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
非叶结点删除
删除55
a
55
删除55
用58取代
b
30
c
60 70
d
10
e
40
f
58
g
65
h
75 80


a
58
b
30
c
60
70
d
10
e
40
f
g
65
h
75
80
合并f, g
结点合并与调整
a
58
b
30
c
70
删除70
d
10
e
40
f
60
65
h
75
80


删除70
用75取代
a
58
删除75
b
30
c
75
d
10
e
40
f
60
65
h
80
删除75
用80取代
调整f, c, h
a
58
b
30
c
80
d
10
e
40
f
60
65
h


删除10
a
58
b
30
c
65
d
10
e
40
f
60
h
80
b
58 65
d
30 40
f
60
h
80

B 树的删除算法

```
template <class T>
bool Btree<T>::Remove (const T& x) {
    Triple<T> loc = Search (x);                    //搜索x
    if (loc.tag == 1) return false;        //未找到,不删除
    MtreeNode<T> *p = loc.r, *q, *s; //找到,开始删除

    //接下来:
    //首先删除关键码, 如果关键码在内部结点中,
    //则通过移动转换成为对叶结点的删除。

    //如果有必要, 进行调整/合并处理
```

删除步骤

```
int j = loc.i;
if (p->ptr[j] != NULL) {                    //非叶结点
      s = p->ptr[j];  GetNode (s);          //读取子树结点
      q = p;                                //找最左下结点
      while (s != NULL) {q = s;  s = s->ptr[0];}
      p->key[j] = q->key[1];                //从叶结点替补
      compress (q, 1);                      //在叶结点删除
      p = q;                                //p指向叶结点
}
else compress (p, j);                       //直接删除
```

```
int d = (m+1)/2;                          //求⌈m/2⌉
while (1) {                               //调整或合并
    if ( p->n < d-1 ) {                       //小于最小限制
        j = 0;  q = p->parent;                //找到双亲
        GetNode (q);                      //读取双亲结点
        while ( j <= q->n && q->ptr[j] != p ) j++;
                //在双亲结点中确定p子树的序号
        if (j == 0) LeftAdjust (p, q, d, j);  //调整
        else RightAdjust (p, q, d, j);
        p = q;                                    //向上调整
        if (p == root) break;
    }
    else break;
}
```

```
    if (root->n == 0) {                  //调整后根的n减到0
        p = root->ptr[0];
        delete root;  root = p;          //删根
        root->parent = NULL;             //新根
    }
}
```

- 当调整一个结点时需要从磁盘读入1个兄弟结点及写出2个结点。当调整到根的子女时则需要写出3个结点。
- 如果我们所用的内存工作区足够大，使得在向下搜索时，读入的结点在做删除时不必再从磁盘读入，那么，在完成一次删除操作时需要读写磁盘的<u>最大次数</u>

  = 找删除元素所在结点向下读盘次数

  + 调整结点时读写盘次数

  $\leqslant h+3(h-1)+1=4h-2$。

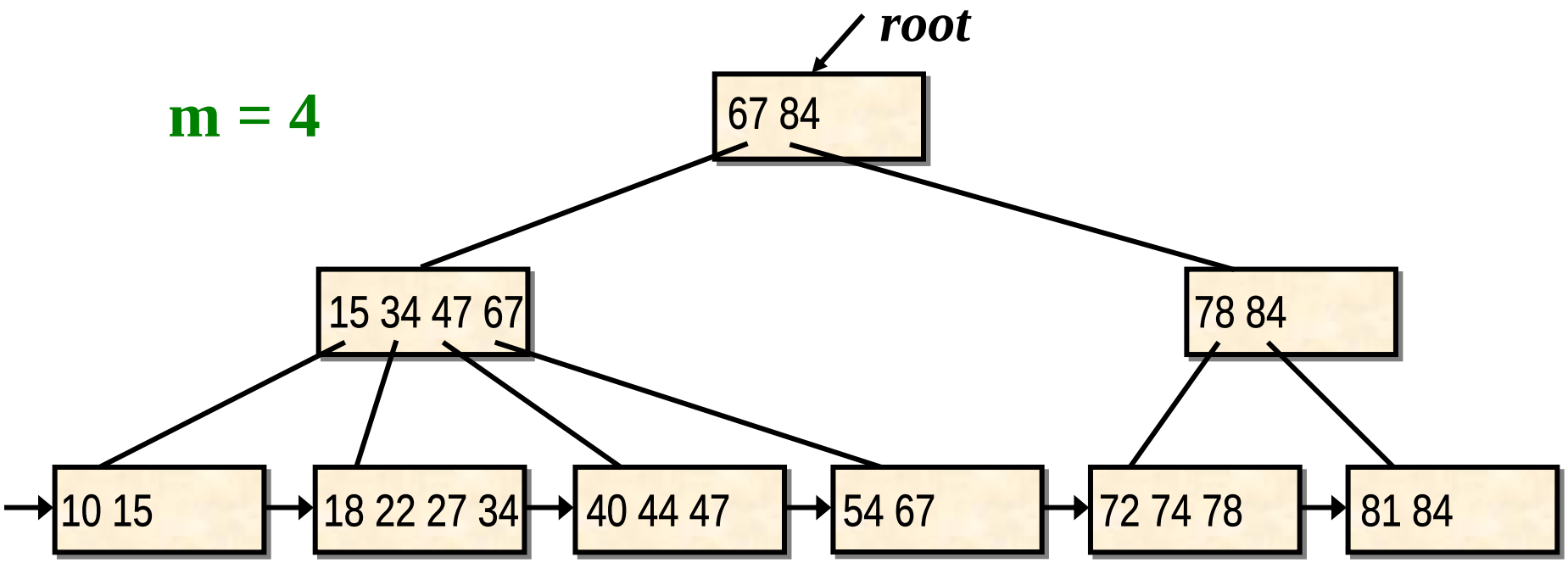
B+树

- 一棵m阶B+ 树是B 树的特殊情形，它与B 树的不同之处在于：
  - ✓所有关键码都存放在叶结点中
  - ✓上层的非叶结点的关键码是其子树中最小（或最大）关键码的复写。
  - ✓叶结点包含了全部关键码及指向相应数据记录存放地址的指针，
  - ✓叶结点本身按关键码从小到大顺序链接。
- 每个非叶结点结构有两种方式处理。按下层结点“最大关键码复写”和“最小关键码复写”。

按“最大关键码复写”原则组织

- 一棵m阶B+ 树的结构定义如下：
  - 每个结点最多有 m 棵子树；
  - 根结点最少有 1 棵子树，除根结点外，其他结点至少有⌈m/2⌉棵子树；
  - 有 n 个子树的结点有 n 个关键码。
  - 所有非叶结点可以看成是叶结点的索引，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是子树中最大的关键码。
  - 所有叶结点在同一层，按从小到大的顺序存放全部关键码，各个叶结点顺序链接。

- 叶结点中存放的是对实际数据记录的索引，每个索引项 (K_i, P_i) 给出数据记录的关键码及实际存储地址。
- 例如，在一棵4阶B+ 树中
  - 所有非叶结点中的子树棵数$2 \leqslant n \leqslant 4$,
  - 所有的关键码都出现在叶结点中，
  - 在叶结点中关键码有序地排列。
  - 上面各层结点中的关键码都是其子树上最大关键码的副本。


m = 4
root
67 84
15 34 47 67
78 84
10 15
18 22 27 34
40 44 47
54 67
72 74 78
81 84

- 通常在B+ 树中有两个头指针：
  - 一个指向B+ 树的根结点，
  - 一个指向关键码最小的叶结点。
- 因此，可以对B+ 树进行两种搜索运算：
  - 循叶结点自己拉起的链表顺序搜索；
  - 从根开始进行自顶向下直到叶结点的随机搜索。

B+树与B树的不同

- 关键码的分布，叶结点包含了所有的关键码
- 叶结点的定义、失败结点的定义

(叶结点不一定符合m阶，它依赖于关键码字节数与指针字节数而定为ml)