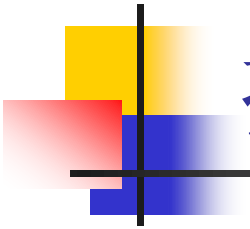
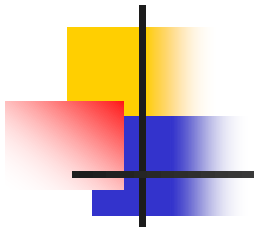


搜索/查找 (Search)

- Search基本概念
- Search算法
  - 静态搜索表
  - 二叉搜索树
  - 最优二叉搜索树
  - AVL树
- B树
- B+树
- Hashing
- Search算法的分析

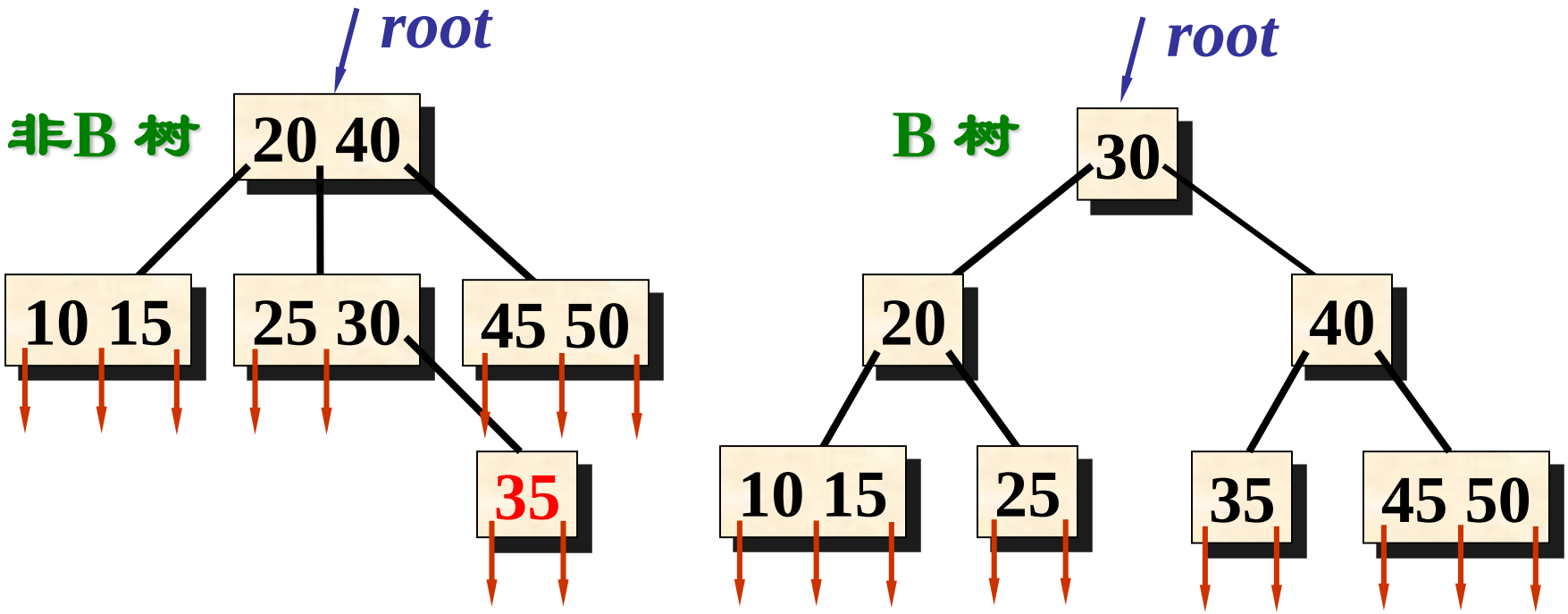
B树(平衡的 m 路搜索树)

- 一棵 m 阶B 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
- 在B 树中的“失败”结点
  - 当 x 不在树中时才能到达的结点
  - 实际不存在，指向它们的指针为NULL。不计入树的高度



- m阶B树继承了m路搜索树的定义。
  - m路搜索树定义中的规定在m阶B树中都保留。
  - 每个结点中还包含有一组指针recptr[m+1]，指向实际记录的存放地址。
  - key[i]与recptr[i] ($1 \le i \le n < m$) 形成一个索引项 (key[i], recptr[i])，
  - 通过key[i]可找到某个记录的存储地址recptr[i]。

- 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。

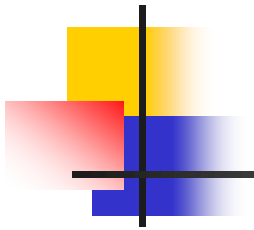


B树类和B树结点类的定义

```
template <class T>
class Btree : public Mtreetree<T>
{    //B树类定义
    //继承m又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

- 继承*m*路搜索树Mtree的搜索算法
- 搜索过程
  - 一个在结点内搜索和循某一条路径向下一层搜索交替进行的过程
  - 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
  - 搜索成功所需的时间取决于关键码所在的层次
  - 搜索不成功所需的时间取决于树的高度

- 定义：
  - B树的高度h为叶结点（失败结点的双亲）所在的层次
  - 树的高度h与树中的关键码个数 N 之间有什么关系？
  - 如果让B树每层结点个数达到最大（$m-1$），且设关键码总数为N，则树的高度达到最小：

$$N \leq m^h - 1 \Longrightarrow h \geqslant \lceil \log_m(N+1) \rceil$$

  - 如果让m阶B树中每层结点个数达到最少，则B树的高度可能达到最大

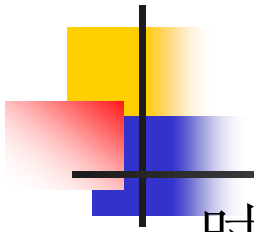
- 例若B 树的阶数 m = 199, 关键码总数 N = 1999999，则B 树的高度 h 不超过:

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

m值的选择

- 提高B 树的阶数 *m*-> 减少树的高度->减少读入结点的次数->减少读磁盘的次数。
- 事实:
  - *m* 受到内存可使用空间的限制
  - 当 *m* 很大超出内存工作区容量时，结点不能一次读入到内存，增加了读盘次数，也增加了结点内搜索的难度
- *m*值的选择：

  使得在B 树中找到关键码 *x* 的时间总量达到最小

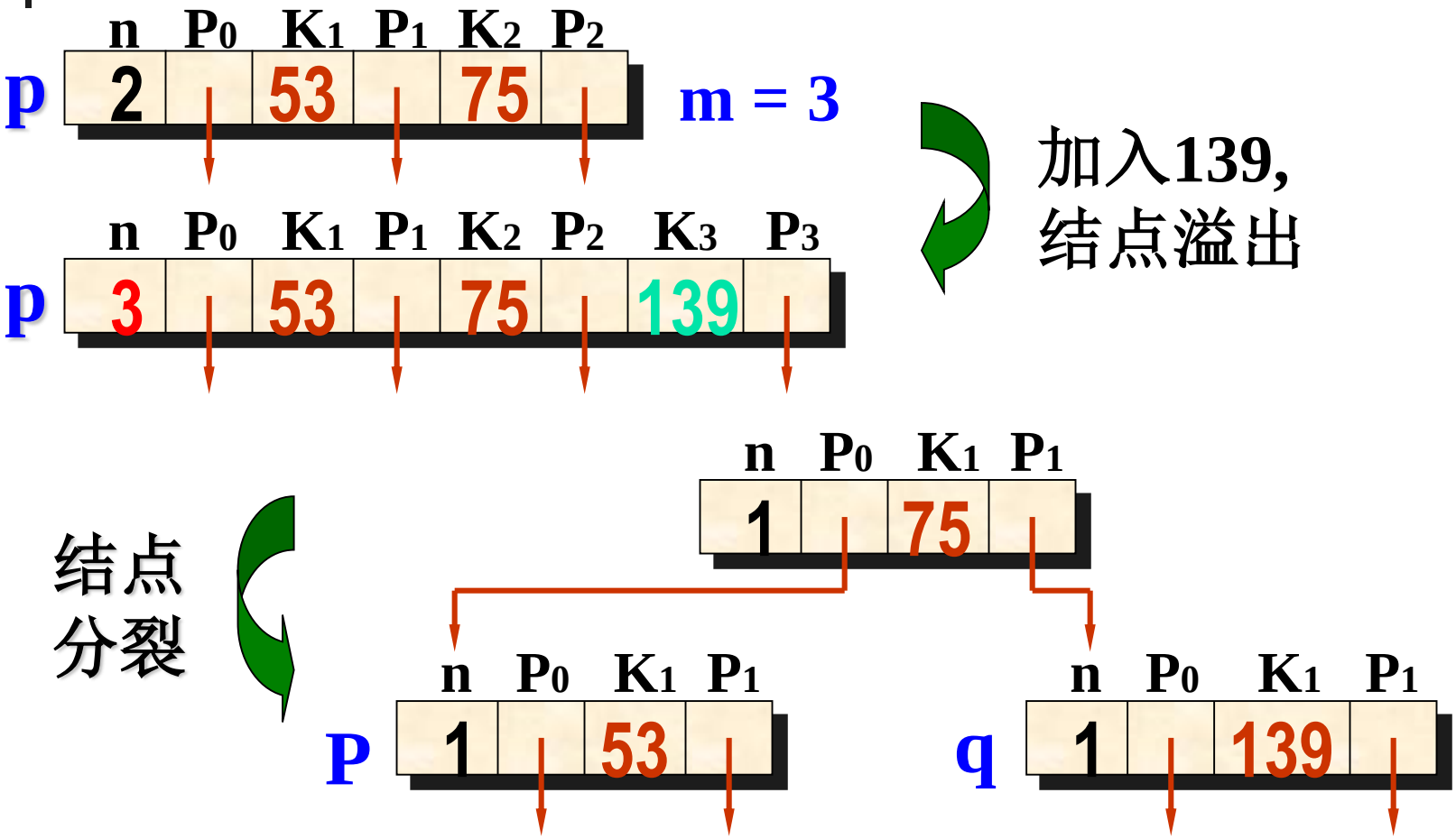


- 时间由两部分组成：
  - 从磁盘中读入结点所用时间
  - 在结点中搜索 x 所用时间

B 树的插入

- B 树
  - 从空树起, 逐个插入关键码而生成
  - 每个非失败结点的关键码个数都在 $[\lceil m/2 \rceil - 1, m-1]$ 之间
- 在某个叶结点开始插入
  - 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

结点“分裂”


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

- 结点“分裂”的原则：
  - 结点 p 中已经有 $m-1$ 个关键码，当再插入一个关键码后结点中的状态为:

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

    $$K_i < K_{i+1},\ 1 \le i < m$$

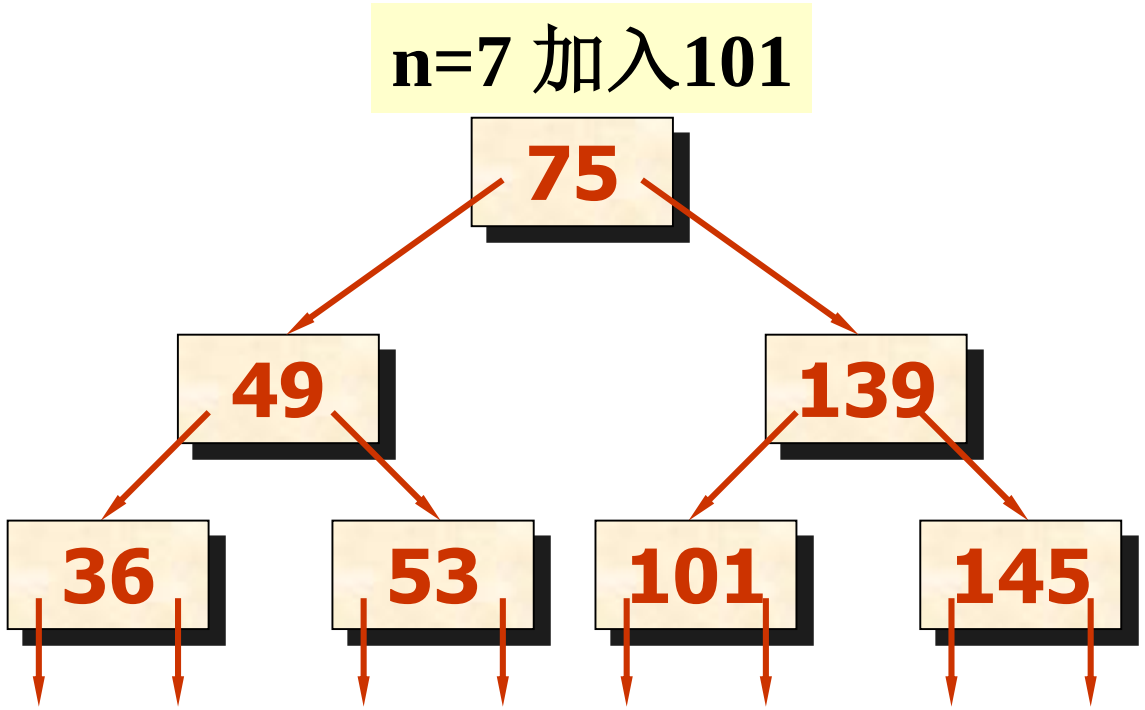
  - 必须把结点 p 分裂成两个结点 p 和 q，它们包含的信息分别为：
    - 结点 p：

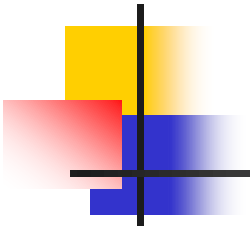
    $$(\lceil m/2 \rceil - 1, P_0, K_1, P_1, \ldots\ldots, K_{\lceil m/2 \rceil - 1}, P_{\lceil m/2 \rceil - 1})$$

- 结点 q：

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

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


n=7 加入101
75
49
139
36
53
101
145



- 插入新关键码
  - 可能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。
- 设B 树的高度为*h*，在自顶向下搜索到叶结点的过程中，需要进行*h* 次读盘。

- 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。
- 假设: 所用的内存工作区足够大, 使得在向下搜索时, 读入的结点在插入后向上分裂时不必再从磁盘读入。
- 则完成一次插入操作，需要读/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)形成插入组
while (1) {
    if (p->n < m-1) {                       //关键码个数未超出
        insertkey (p, j, K, ap);    //插入, 且p->n加1
        PutNode (p);  return true;          //输出结点p
    }
    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
    move(p, q, s, m);                       //向新结点移动
    K = p->key[s];   ap = q;
                        //(K,ap)形成向上插入二元组
```

```
if (p->parent != NULL) {                    //从下向上调整
    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
    p = t;                  //p上升到双亲, 继续调整
}
else {                              //原来p是根, 要产生新根
    root = new MtreeNode<T>;
    root->n = 1;  root->parent = NULL;
```

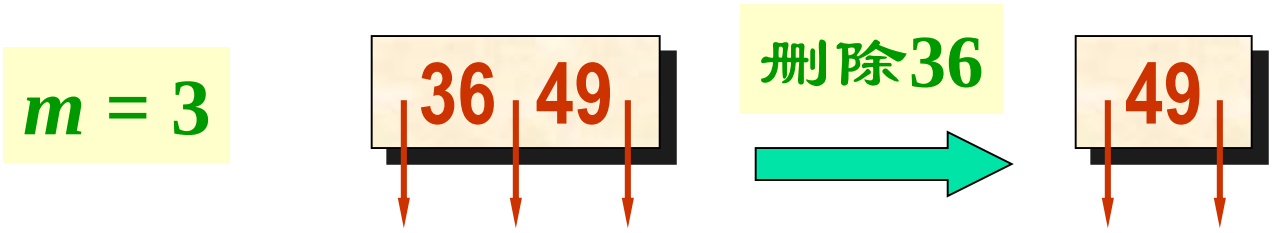
```
            root->key[1] = K;
            root->ptr[0] = p;  root->ptr[1] = ap;
            q->parent = p->parent = root;
            PutNode(p);  PutNode(q);  PutNode(root);
            return true;
        }
    }
};
```

B 树的删除

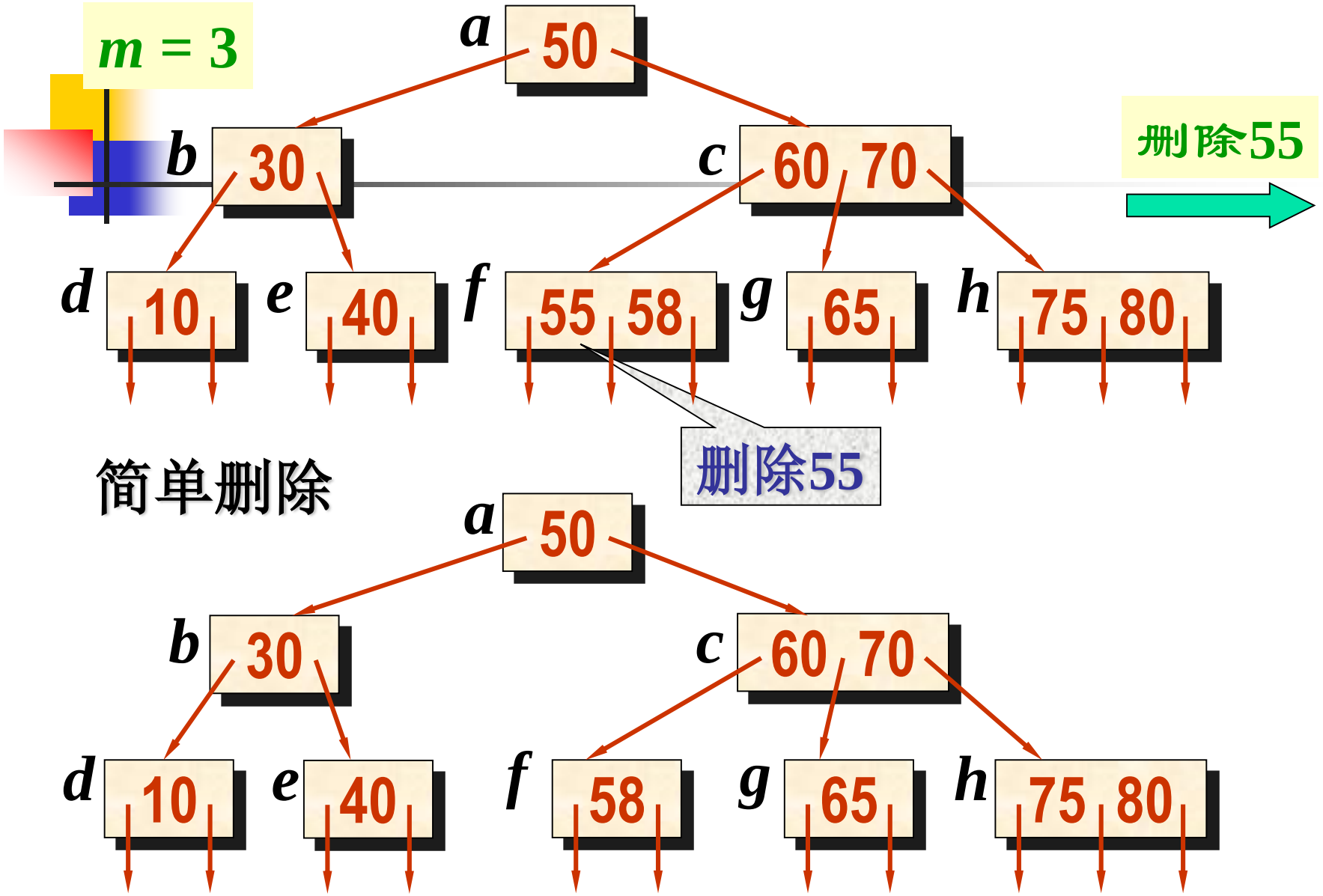
- 删除过程
  - ✓ 找到待删关键码所在的结点，从中删去该关键码
  - ✓ 若该结点不是叶结点，且被删关键码为 K_i，$1 \le i \le n$，
    则删去该关键码后，以该结点 P_i 所指示子树中的最小关键码 x 代替被删关键码 K_i 所在的位置。
    在 x 所在的叶结点中，删除 x。

在叶结点上的删除有 4 种情况：

❶ 被删关键码所在叶结点，同时是根结点，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



❷ 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。

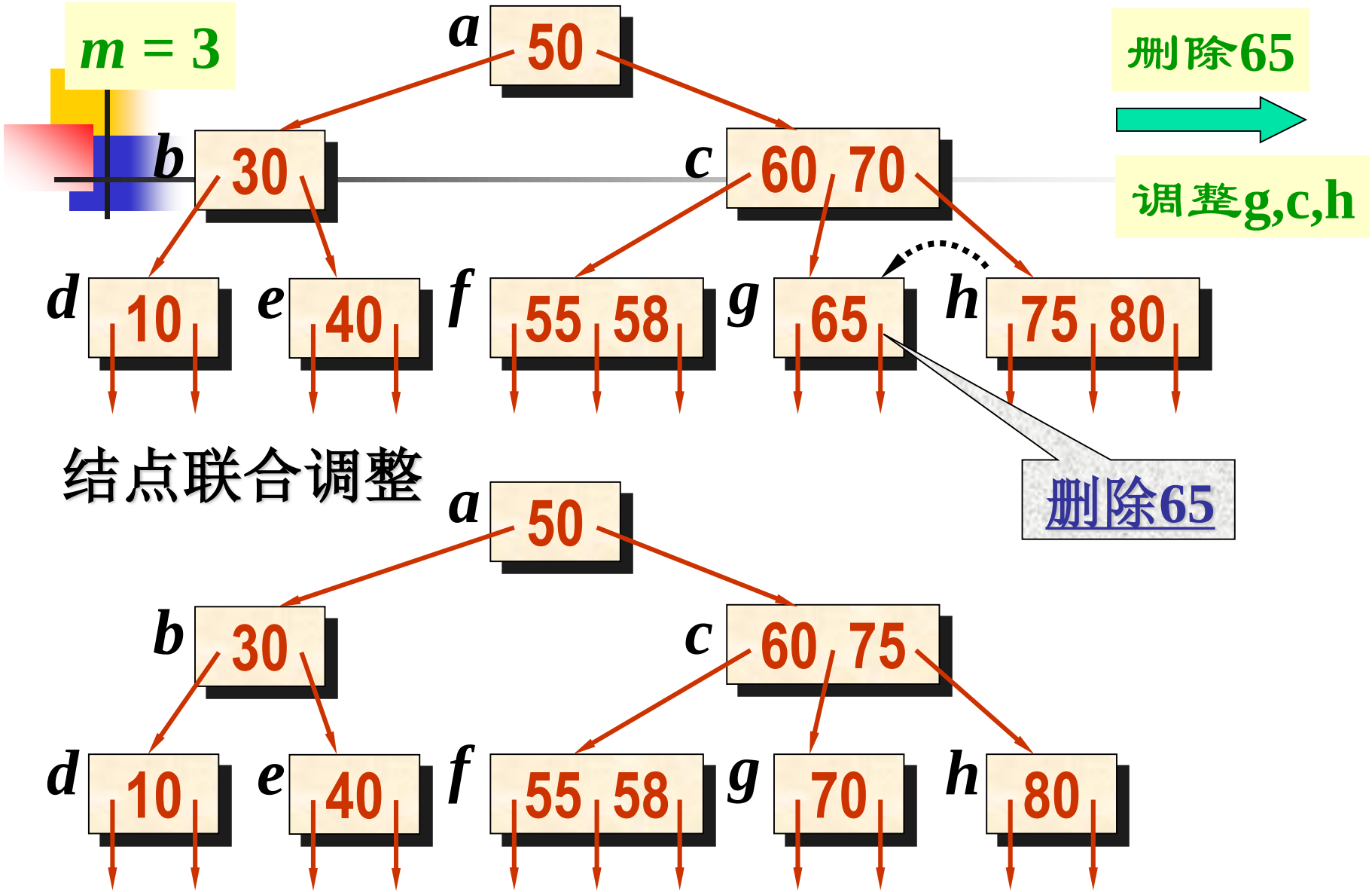

m = 3
a
50
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
删除55
删除55
简单删除
a
50
b
30
c
60 70
d
10
e
40
f
58
g
65
h
75 80

- 注意：删除一个关键码
  - 若结点中所剩关键码个数少于下限，考虑结点的调整或合并问题。

❸ 被删关键码所在叶结点，删除前关键码个数 $n=\lceil m/2\rceil-1$,

若此时与该结点相邻的右兄弟 (或左兄弟) 结点的关键码个数 $n\geqslant\lceil m/2\rceil$,

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


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

调整步骤：

a) 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 ($1 \le i \le n$) 下移；

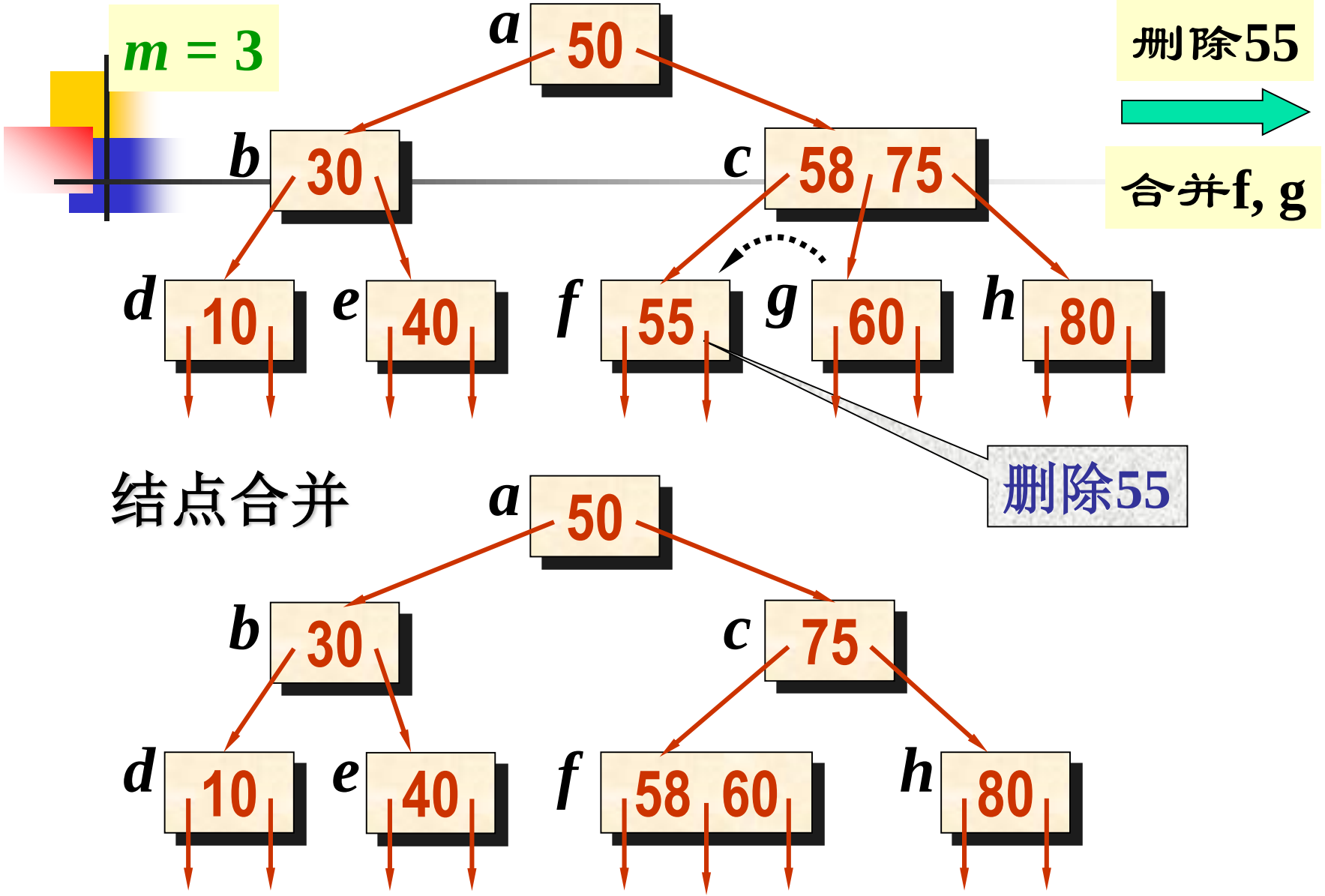
b) 将右兄弟 (或左兄弟) 结点中，最小 (或最大) 关键码，上移到双亲结点的 K_i 位置；

c) 将右兄弟 (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) 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。

❹ 被删关键码所在叶结点，删除前关键码个数 $n = \lceil m/2 \rceil - 1$，若此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
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

合并调整

a) 合并 p 中的子树指针 P_i 与 P_{i+1} 所指的结点，且保留 P_i 所指结点，则把 p 中的关键码 K_{i+1} 下移到 P_i 所指的结点中。

b) 把 p 中子树指针 P_{i+1} 所指结点中，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。

a) 在结点 p 中，用后面剩余的关键码和指针填补关键码 K_{i+1} 和指针 P_{i+1}。

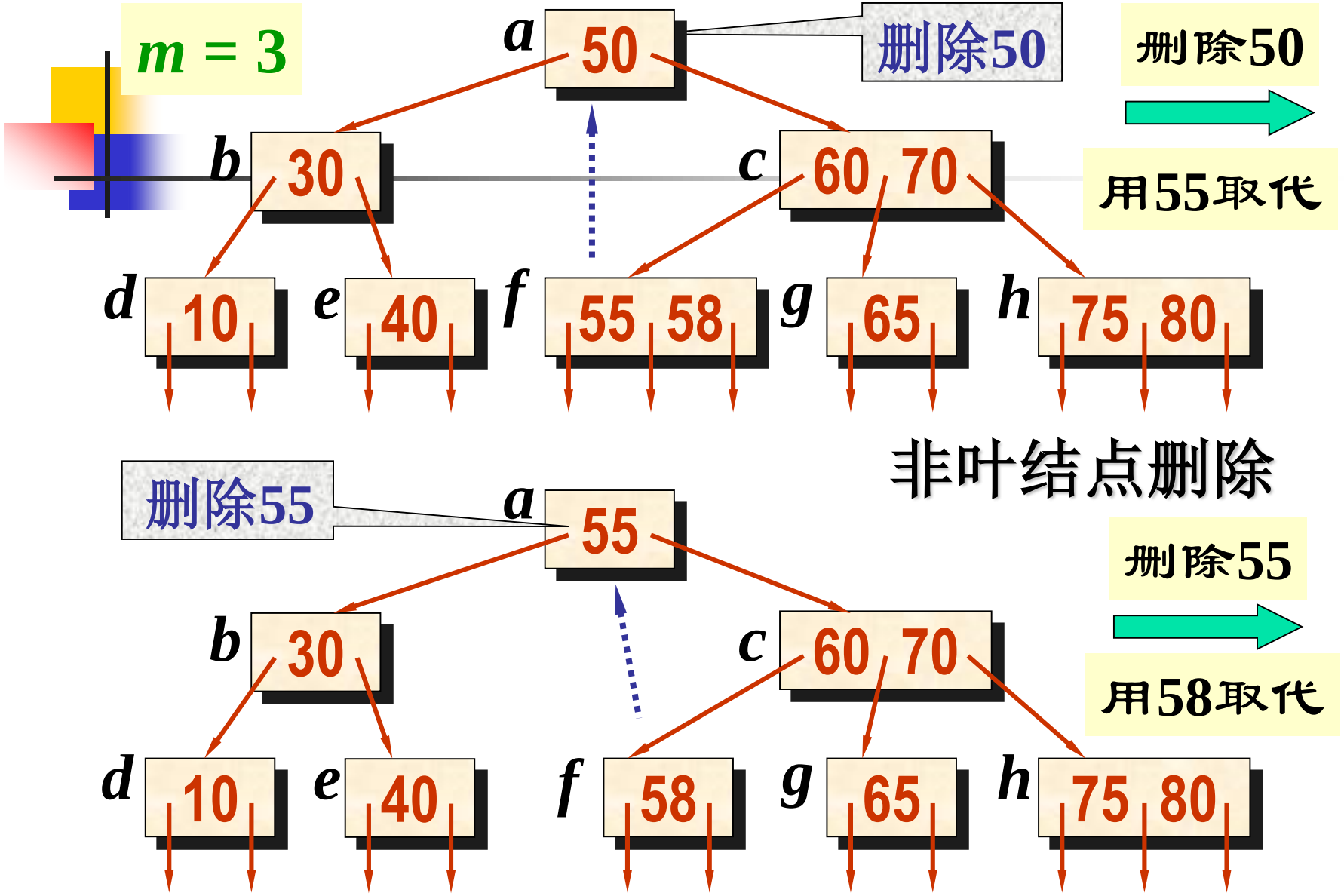
b) 修改结点 p 和选定保留结点的关键码个数。

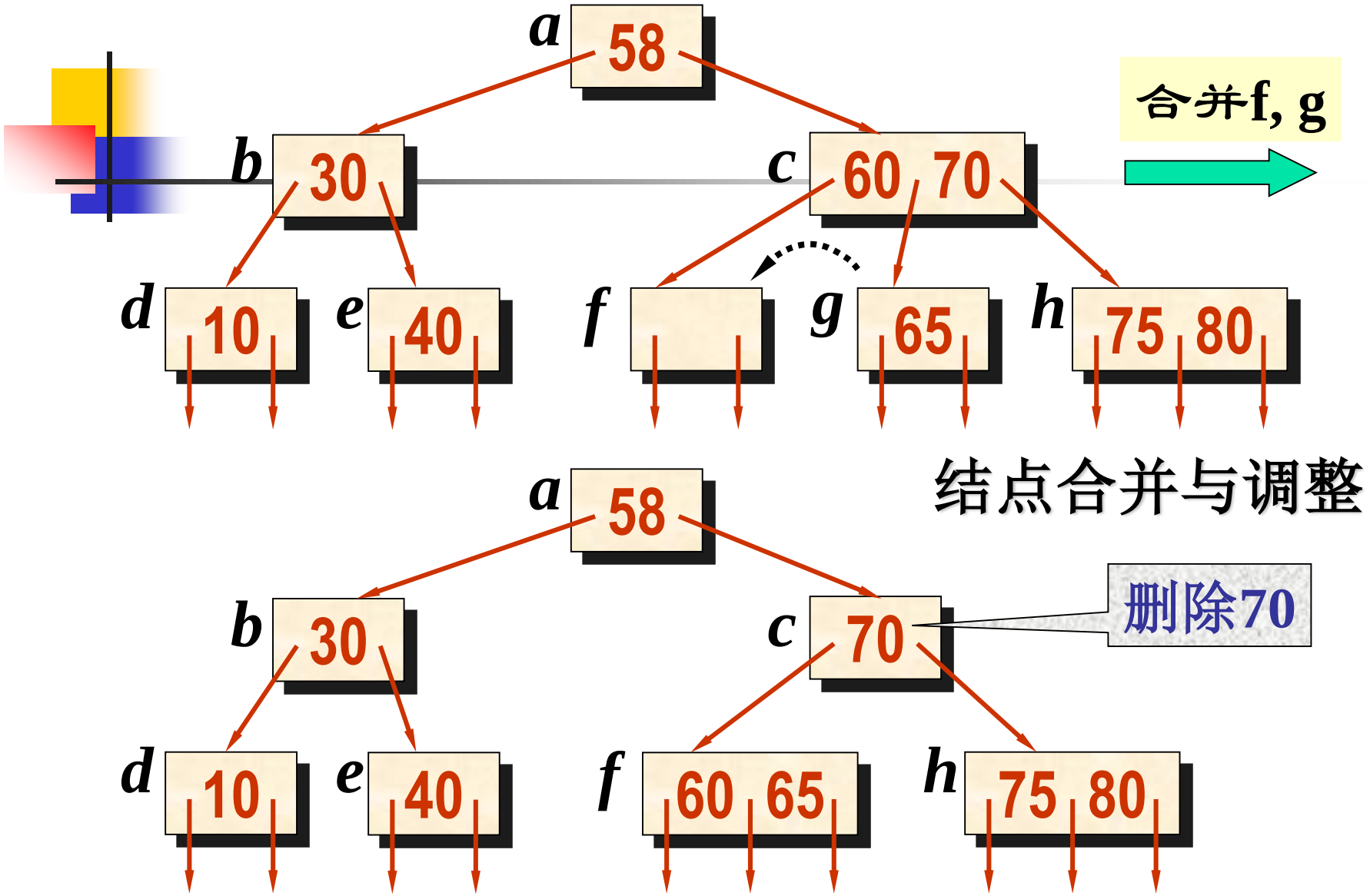
合并结点过程

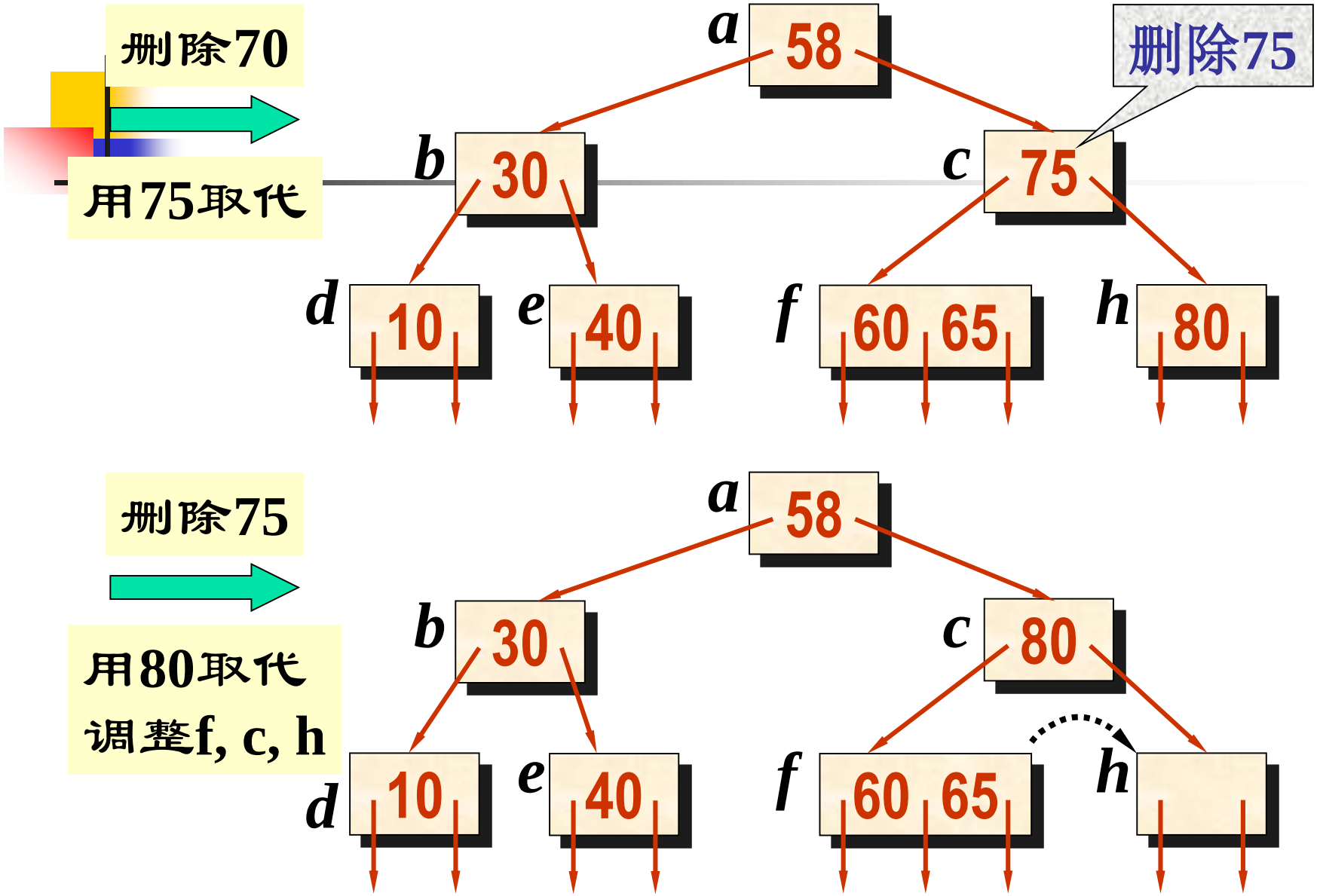
- 双亲结点中的关键码个数减少。
- 若双亲结点是根结点且结点关键码个数减到 0, 则将该双亲结点删去, 合并后保留的结点成为新的根结点;
- 否则将双亲结点与合并后保留的结点，都写回磁盘, 删除处理结束。
- 若双亲结点不是根结点、且关键码个数减到$\lceil m/2 \rceil - 2$，又要与它自己的兄弟结点合并, 重复上面的合并步骤。
- 最坏情况下：这种合并处理，自下向上直到根结点。

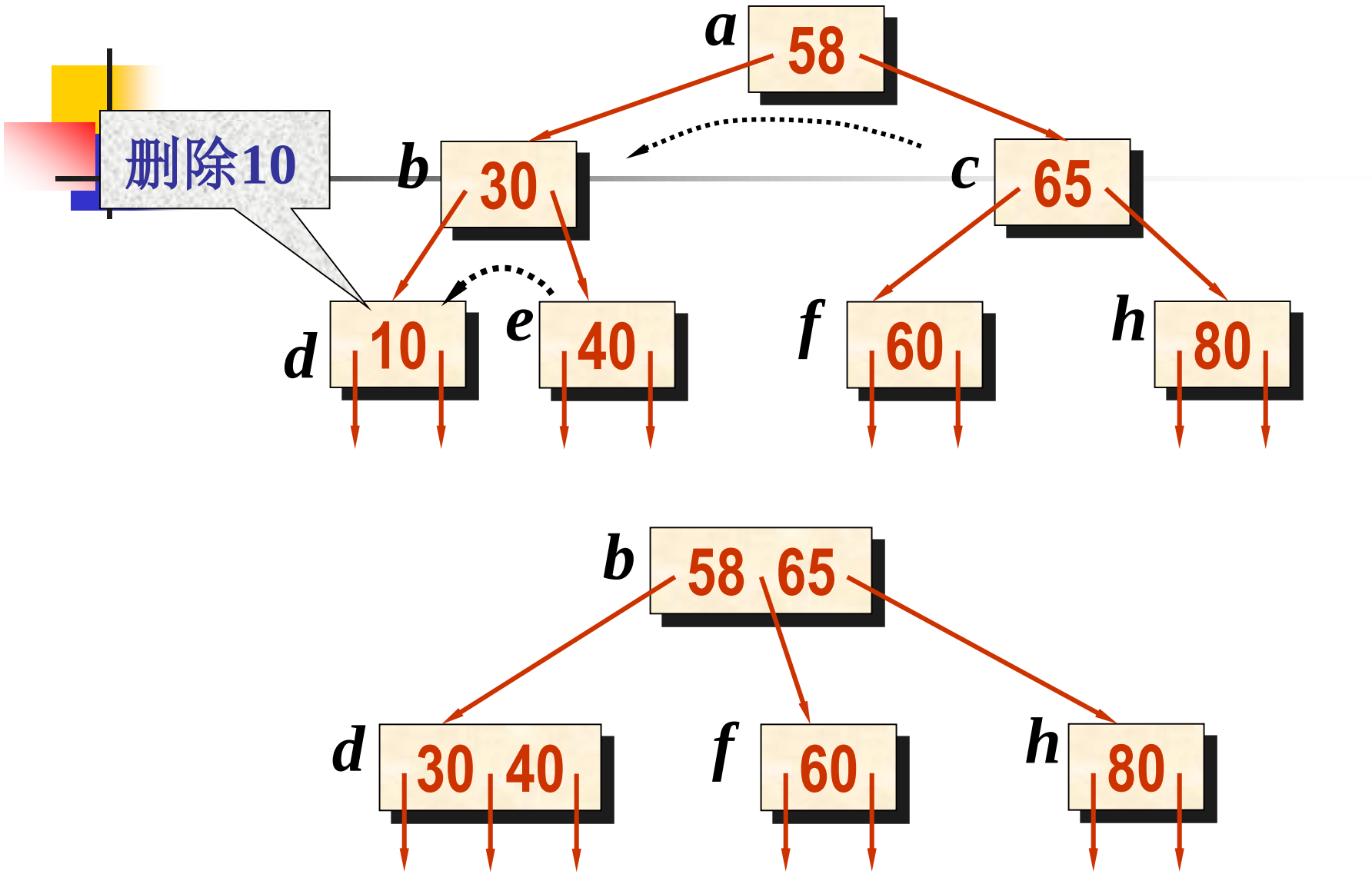
- 调整一个结点时
  - 需要从磁盘读入1个兄弟结点、写出2个结点。
- 调整到根的子女时
  - 需要写出3个结点。
- 如果：所用的内存工作区足够大, 使得在向下搜索时, 读入的结点在做删除时不必再从磁盘读入。
- 那么：完成一次删除操作，读写磁盘的最大次数 = 找删除元素所在结点向下读盘次数 + 调整结点时读写盘次数

$$\leq h + 3(h-1)+1 = 4h-2$$


m = 3
a
50
删除50
删除50
b
30
c
60 70
用55取代
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
b
30
c
60 70
用58取代
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
b
30
c
75
删除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,删除
    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;                                  //到叶结点删除
}
else compress (p, j);               //结点直接删除
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
    }
}
```