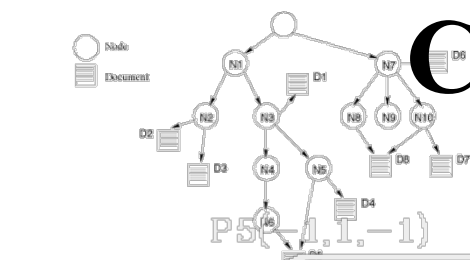
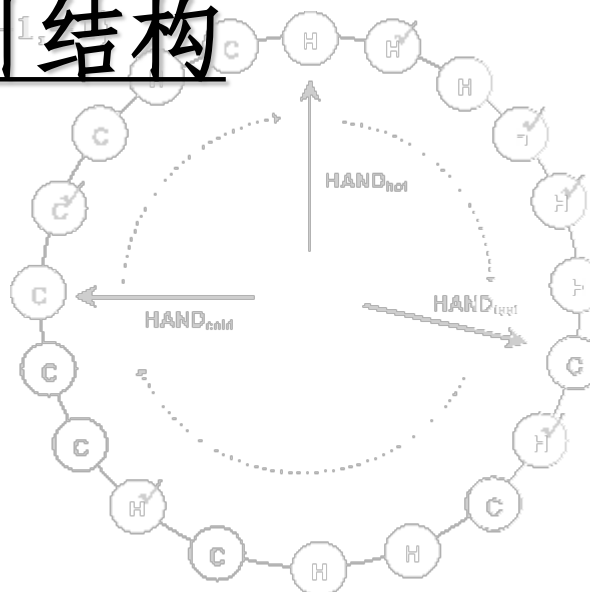


# Chapter2 B-树 红黑树



■ **m路静态搜索树**

■ **动态索引结构**



## 2.1 静态索引结构

- 线性索引 (Linear Index List)
- m路静态搜索树



# 线性索引

索引表

| key | addr |
|-----|------|
| 03  | 180  |
| 08  | 140  |
| 17  | 340  |
| 24  | 260  |
| 47  | 300  |
| 51  | 380  |
| 83  | 100  |
| 95  | 220  |

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  | 王鲁 | 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 | ... |

## •稠密索引:

——一个索引项对应数据表中一个对象的索引结构。

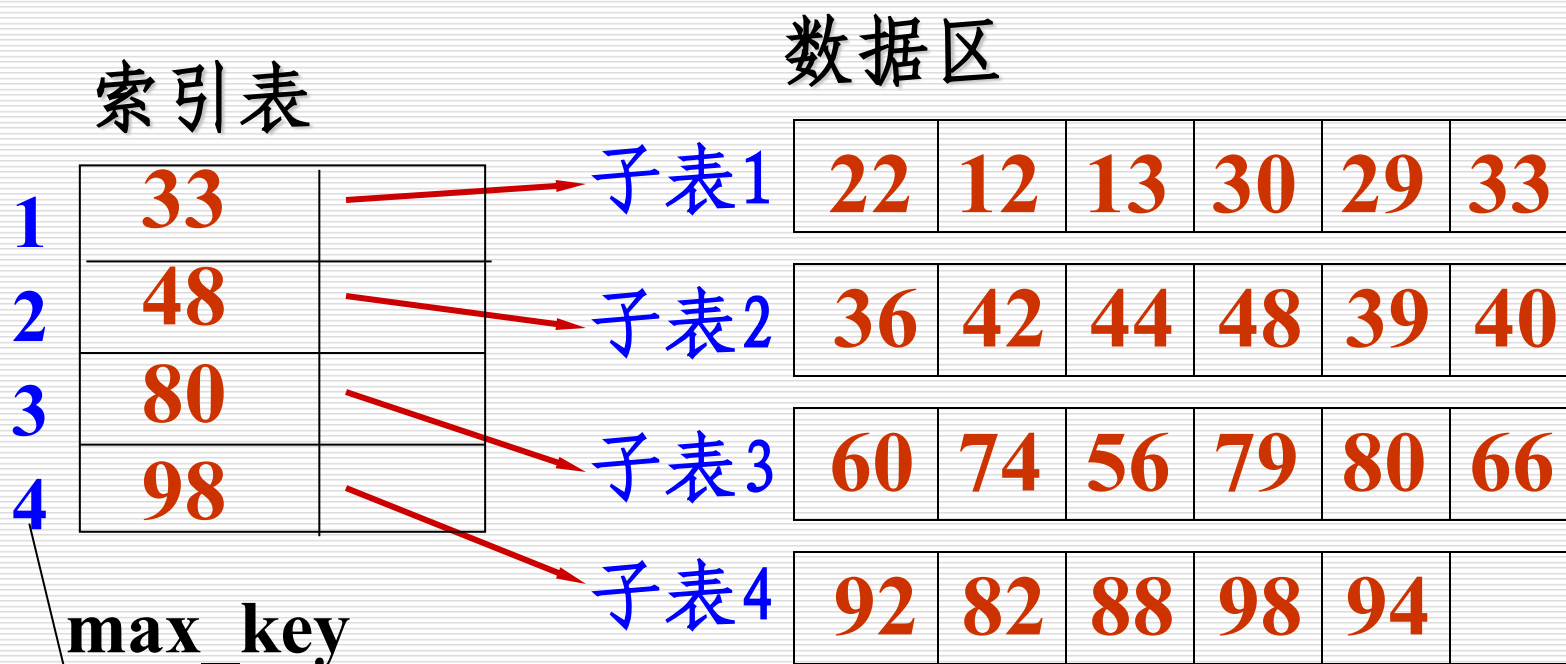
——当对象在外存中按加入顺序存放而不是按关键码有序存放时必须采用稠密索引结构,这时的索引结构叫做索引非顺序结构

## •稀疏索引:

当对象在外存中有序存放时,可以把所有  $n$  个对象分为  $b$  个子表(块)存放, 一个索引项对应数据表中一组对象(子表)。

# 子表的要求

- 所有对象：可能按关键码有序地存放，也可能无序地存放。
- 子表：必须分块有序，后一个子表中所有对象的关键码均大于前一个子表中所有对象的关键码。它们都存放在数据区中



**索引ID: 第  $i$  个索引项是第  $i$  个子表的索引项,  $i = 0, 1, \dots, n-1$**

## 索引顺序结构

## 索引顺序结构的搜索:

- 先在索引表 ID 中搜索给定值 K, 确定满足  $ID[i-1].max\_key < K \leq ID[i].max\_key$  的 i 值, 即待查对象可能在的子表的序号。
- 再在第 i 个子表中按给定值搜索要求的对象。

## 搜索成功时的平均搜索长度:

$$ASL_{IndexSeq} = ASL_{Index} + ASL_{SubList}$$

- 设把长度为  $n$  的表分成均等的  $b$  个子表，每个子表  $s$  个对象，则  $b = \lceil n/s \rceil$ 。
- 设表中每个对象的搜索概率相等，则每个子表的搜索概率为  $1/b$ ，子表内各对象的搜索概率为  $1/s$ 。
- 若对索引表和子表都用顺序搜索，则索引顺序搜索的搜索成功时的平均搜索长度为

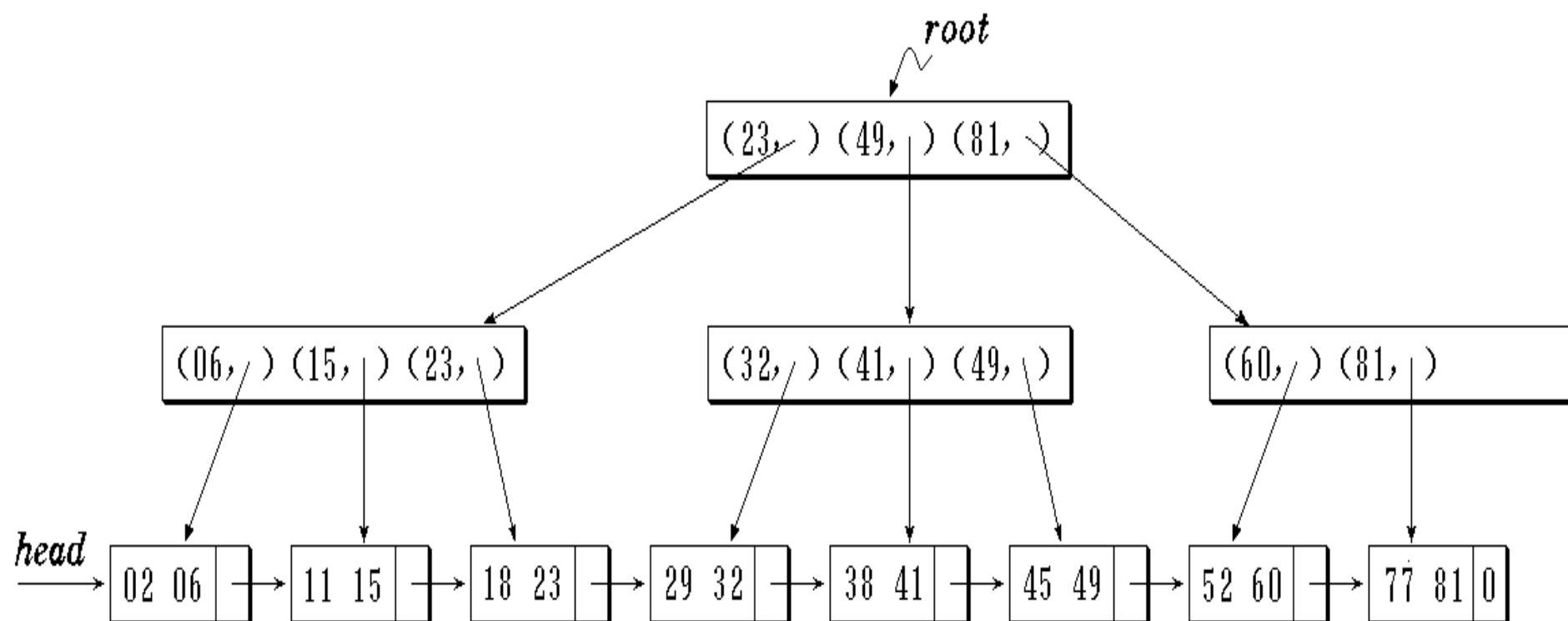
$$ASL_{IndexSeq} = (b+1)/2 + (s+1)/2 = (b+s)/2 + 1 = (n/s + s)/2 + 1$$

- 若采用折半搜索确定对象所在的子表，则搜索成功时的平均搜索长度为

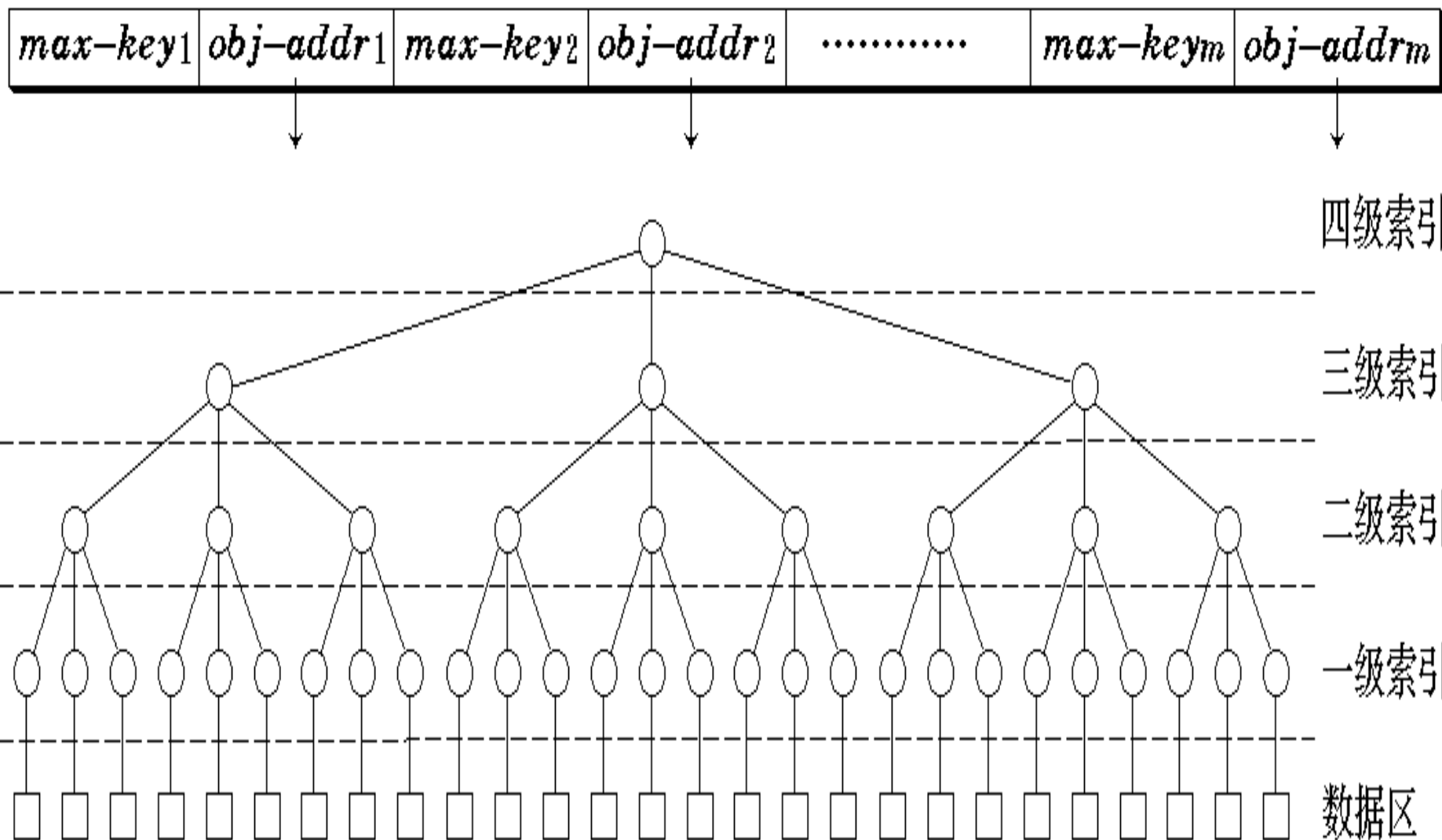
$$ASL_{IndexSeq} = ASL_{Index} + ASL_{SubList}$$

$$\approx \log_2(b+1) - 1 + (s+1)/2 \approx \log_2(1+n/s) + s/2$$

# m 路静态搜索树



二级索引结构的示例

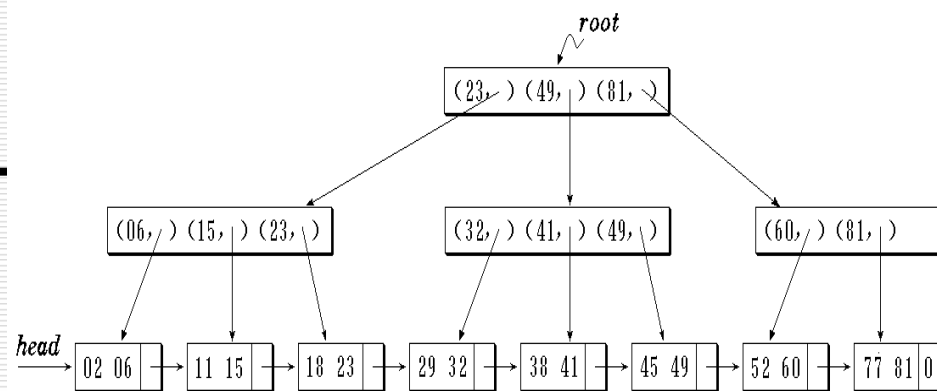


多级索引结构形成  $m$  路搜索树



每一个分支结点：一个索引块

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对象的**关键码**和**存放地址**。

## $m$ 路搜索树

### ▣ 静态索引结构

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

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

## 10.2 动态搜索结构

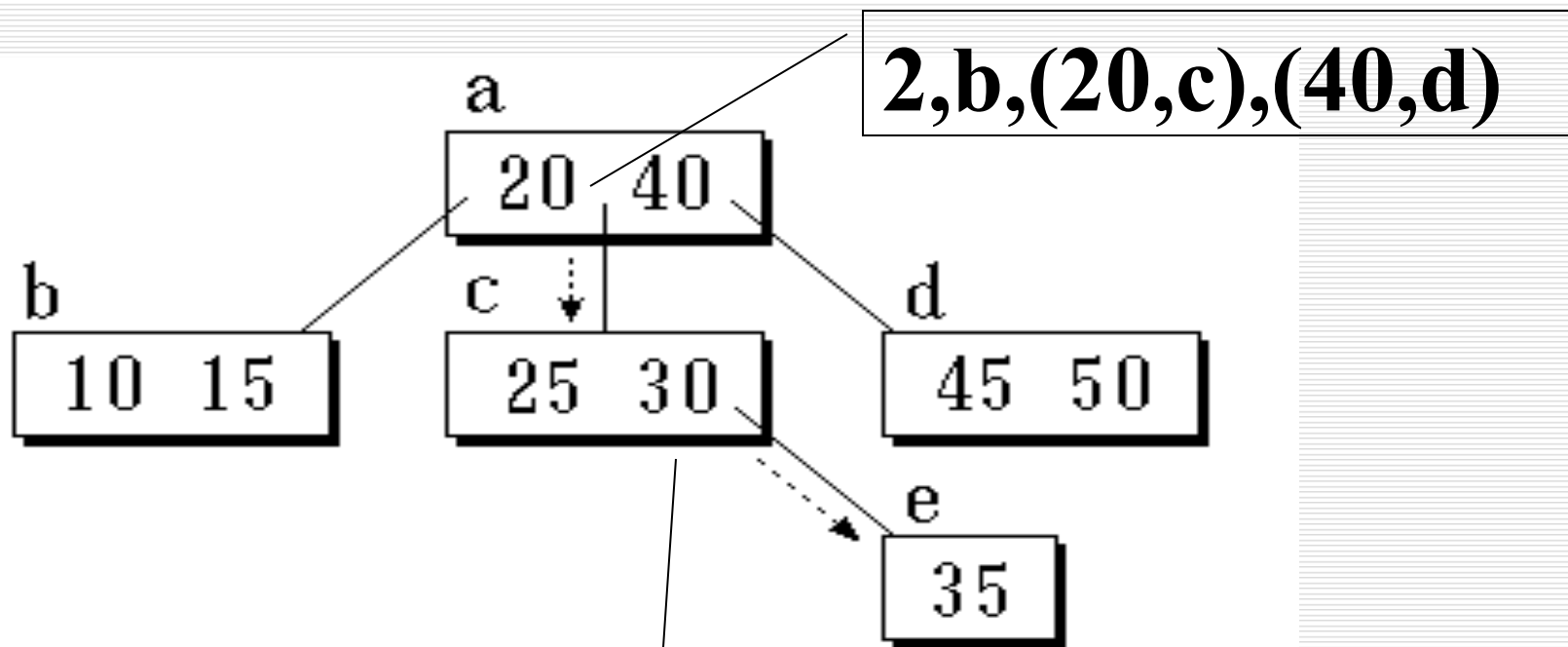
### 动态的 $m$ 路搜索树

一般定义：一棵  $m$  路搜索树，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， $0 \leq i \leq n < m$ ； $K_i$  是关键词，  
 $1 \leq i \leq n < m$ 。  $K_i < K_{i+1}, 1 \leq i < n$ 。

- 在子树  $P_i$  中所有的关键词都小于  $K_{i+1}$ ，且大于  $K_i$ ，  
 $0 < i < n$ 。
- 在子树  $P_n$  中所有的关键词都大于  $K_n$ ；
- 在子树  $P_0$  中的所有关键词都小于  $K_1$ 。
- 子树  $P_i$  也是  $m$  路搜索树， $0 \leq i \leq n$ 。



一棵3路搜索树的示例

$2, 0, (25, 0), (30, e)$

## *m* 路搜索树的类定义

```
template <class Type> class Mtree {    //基类
public:
    Triple<Type> & Search ( const Type & );
protected:
    Mnode<Type> root;
    int m;
}
```

*AVL*树是2路搜索树。

- 如果已知  $m$  路搜索树的度  $m$  和它的高度  $h$ , 则树中的最大结点数为:

(等比级数前  $h$  项求和)

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

- 每个结点中最多有  $m-1$  个关键码, 在一棵高度为  $h$  的  $m$  路搜索树中关键码的最大个数为?

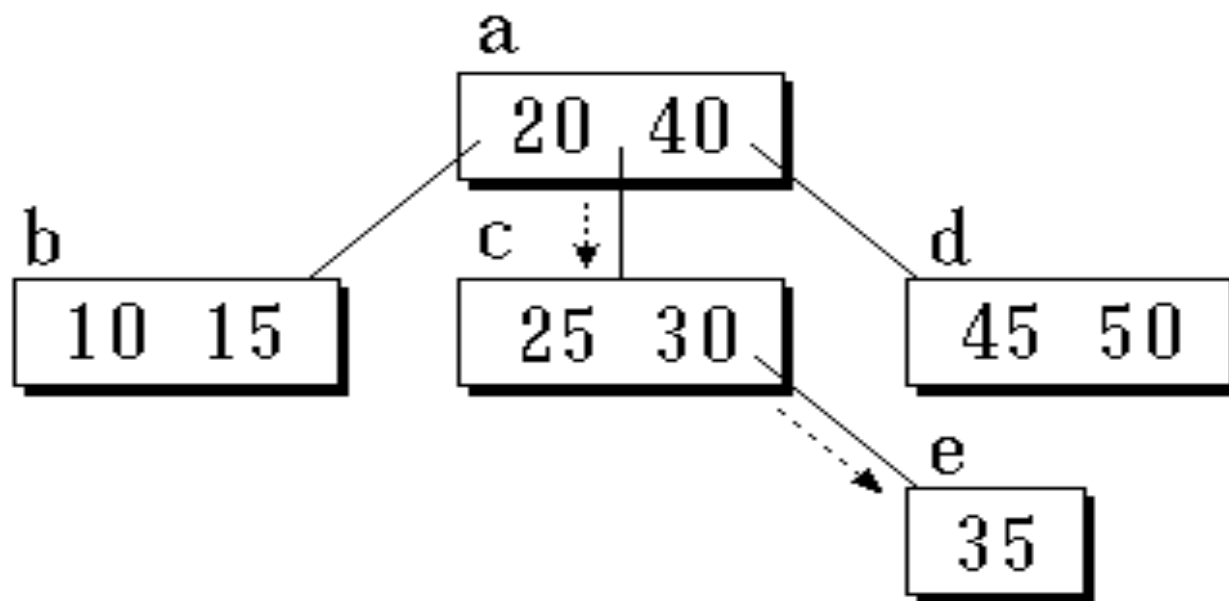
$$m^{h+1}-1$$

## 标识 $m$ 路搜索树结点的三元组表示

```
struct Triple {  
    Mnode<Type> * r;           //结点地址指针  
    int i; int tag;           //结点中关键码序号i  
};                             //tag=0,搜索成功; tag=1,搜索不成功
```

# $m$ 路搜索树上的搜索算法

搜索35





```
template <class Type> Triple<Type> & Mtree<Type> ::
```

```
Search ( const Type & x ) {
```

```
    Triple result;           //记录搜索结果三元组
```

```
    GetNode ( root );        //读入根结点
```

```
    Mnode <Type> *p = root, *q = NULL;
```

```
    while ( p != NULL ) {    //从根开始检测
```

```
        int i = 0; p->key[(p->n)+1] = MAXKEY;
```

```
        while ( p->key[i+1] < x ) i++; //结点内搜索
```

```
        if ( p->key[i+1] == x ) {      //搜索成功
```

```
            result.r = p; result.i = i+1; result.tag = 0;
```

```
            return result;
```

```
        }
```

```
        q = p; p = p->ptr[i]; //向下一层结点搜索
```

```
        GetNode (p);          //读入该结点
```

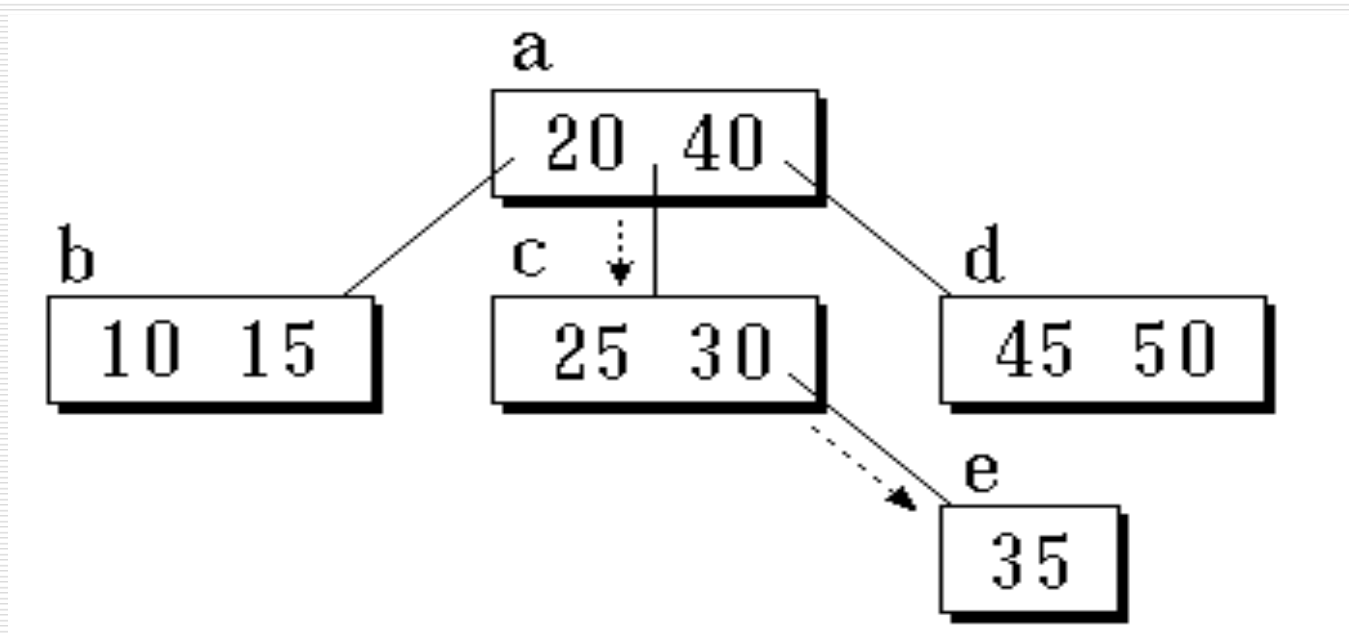
```
    }
```

```
    result.r = q; result.i = i; result.tag = 1;
```

```
    return result; } //搜索失败,返回插入位置
```

## 改善树的搜索性能？

- 提高搜索树的路数  $m$
- 对于给定的关键码数  $n$ ，搜索树是平衡的

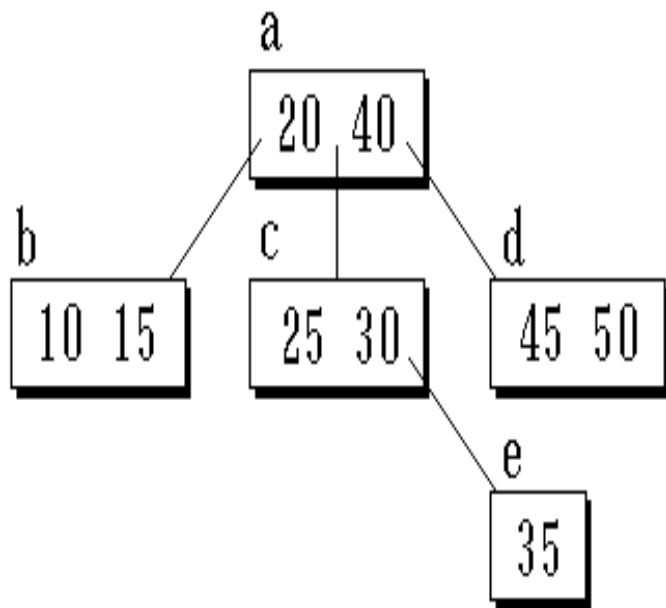


## B\_树

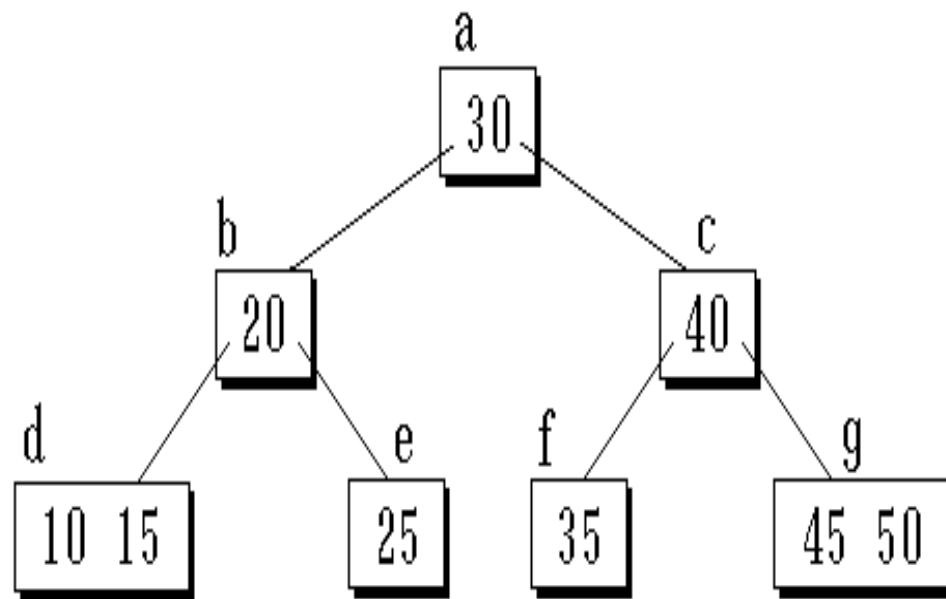
一棵  $m$  阶 B\_树 是一棵  $m$  路搜索树，它或者是空树，或者是满足下列性质的树：

- 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：是当搜索值  $x$  不在树中时才能到达的结点



## 非B\_树



## B\_树

- 事实上，在B\_树的每个结点中还包含有一组指针  $D[m]$ ，指向实际对象的存放地址。 $K[i]$  与  $D[i]$  ( $1 \leq i \leq n < m$ ) 形成一个索引项( $K[i]$ ,  $D[i]$ )，通过  $K[i]$  可找到某个对象的存储地址  $D[i]$ 。

# B\_树的类定义和B\_树结点类定义

```
template <class Type> class Btree :
```

```
    public Mtree<Type> {
```

```
public:
```

```
    int Insert ( const Type& x );
```

```
    int Remove ( const Type& x );
```

```
};
```

```
template <class Type> class Mnode { //结点定义
```

```
private:
```

```
    int n; //结点中关键码个数
```

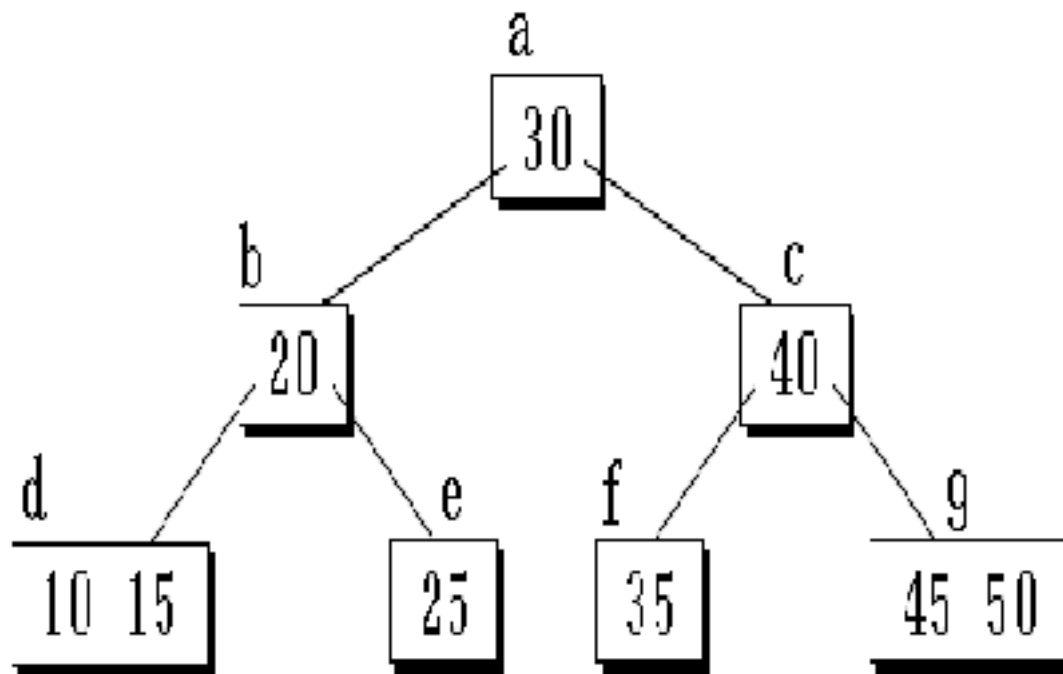
```
    Mnode<Type> *parent; //双亲指针
```

```
    Type key[m+1]; //关键码数组 1~m-1
```

```
    Mnode<Type> *ptr[m+1]; //子树指针数组 0~m
```

```
};
```

# B\_树的搜索算法



## ▣ 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。

▣ 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。

## 高度 $h$ 与关键码个数 $N$ 之间的关系

- 设在 $m$ 阶B\_树中，失败结点位于第 $h$ 层。  
在这棵B\_树中关键码个数 $N$ 最小能达到多少？

0层      1 个结点

1层      至少 2 个结点

2层      至少  $2 \lceil m/2 \rceil$  个结点

3层      至少  $2 \lceil m/2 \rceil^2$  个结点

如此类推，.....

$h-1$  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。



若树中关键码有  $N$  个, 则失败结点数为  $N+1$

$N+1 = \text{失败结点数} = \text{位于第 } h \text{ 层的结点数}$

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

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

□ 反之, 如果在一棵  $m$  阶B\_树中有  $N$  个关键码, 则所有的非失败结点所在层次都小于  $h$ , 则

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

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

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

示例1: 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?

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

示例2: 若B\_树的阶数  $m = 3$ , 高度  $h = 4$ , 则关键码总数至少为?

$$N = 2 \lceil 3 / 2 \rceil^{4-1} - 1 = 15$$

## $m$ 值的选择

应使得在B\_树中找到关键码  $x$  的时间总量达到最小。

- 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

生成B\_树的基本方法:

- 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。
- 插入在最底层的非失败结点开始。  
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。

## 实现结点“分裂”的原则:

- 设结点  $p$  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  
( $m, P_0, K_1, P_1, K_2, P_2, \dots, K_m, P_m$ )

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:

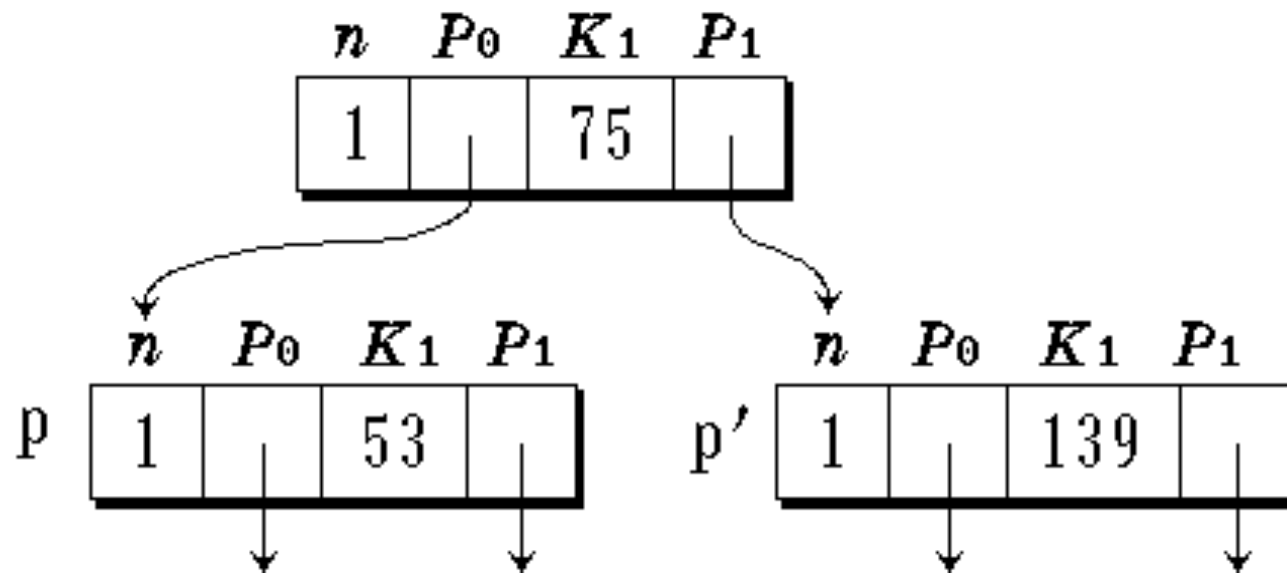
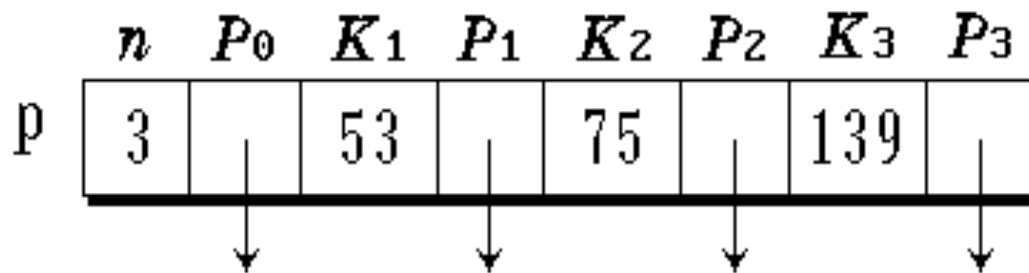
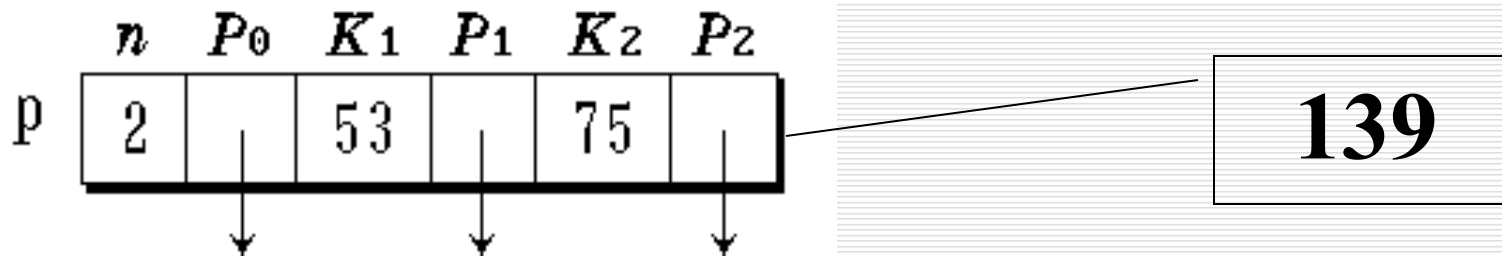
✓ 结点  $p$ :

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

✓ 结点  $q$ :

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

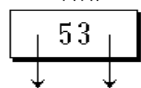
✓ 位于中间的关键码  $K_{\lceil m/2 \rceil}$  与指向新结点  $q$  的指针形成一个二元组 ( $K_{\lceil m/2 \rceil}, q$ ), 插入到这两个结



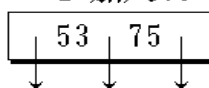
结点“分裂”的示例

# 示例：从空树开始逐个加入关键词建立3阶B\_树

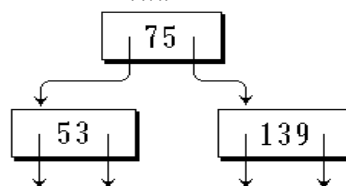
$n=1$  加入 53



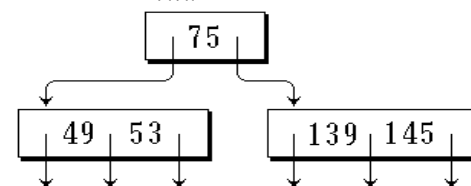
$n=2$  加入 75



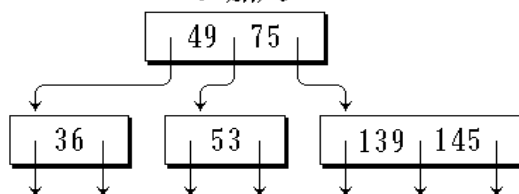
$n=3$  加入 139



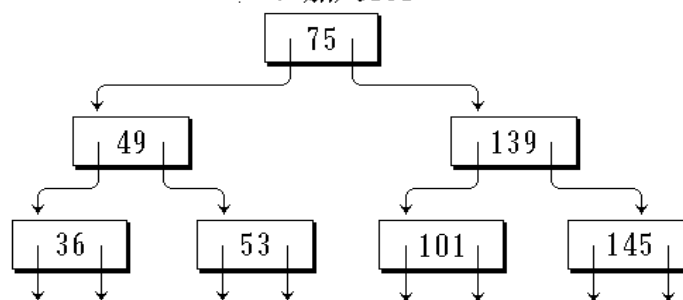
$n=5$  加入 49, 145



$n=6$  加入 36



$n=7$  加入 101



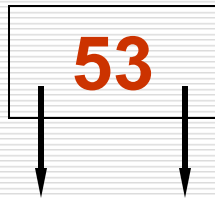
## m阶B\_树的性质

- 每个结点至多有m棵子树，m-1个关键码
- 若根结点不是叶结点，则至少有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
- 若树中关键码有N个，则失败结点数为N+1
- $N \geq 2 \lceil m / 2 \rceil^{h-1} - 1 \quad h-1 \leq \log_{\lceil m / 2 \rceil} ((N+1) / 2)$

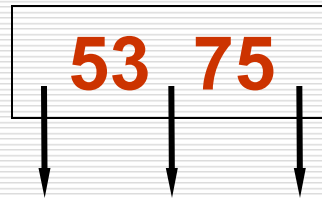


# 示例:从空树开始加入关键码建立3阶B 树

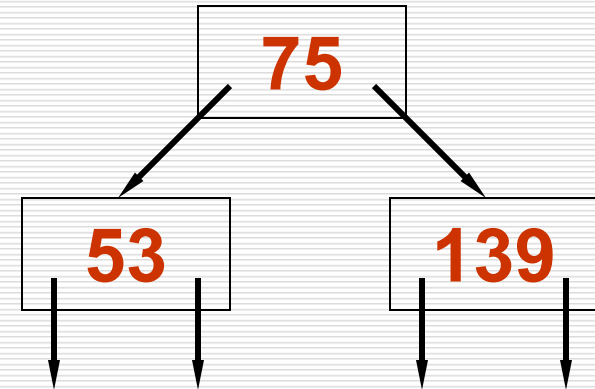
n=1 加入53



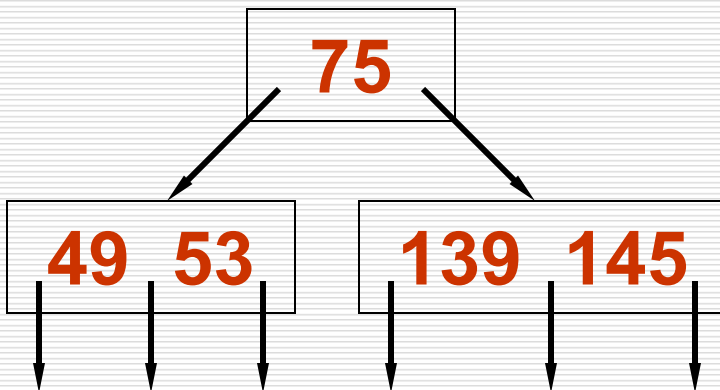
n=2 加入75



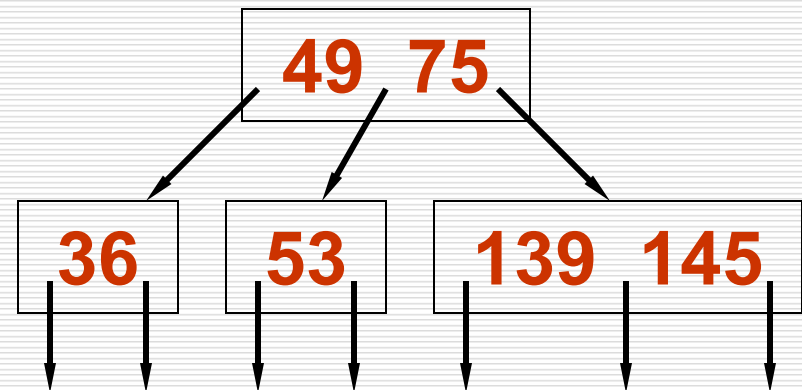
n=3 加入139



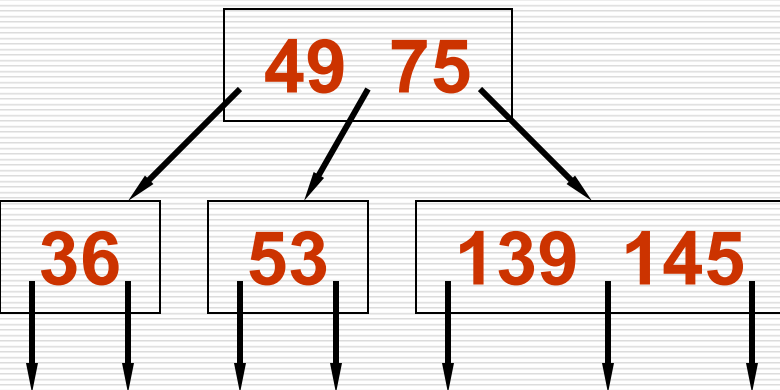
n=5 加入49,145



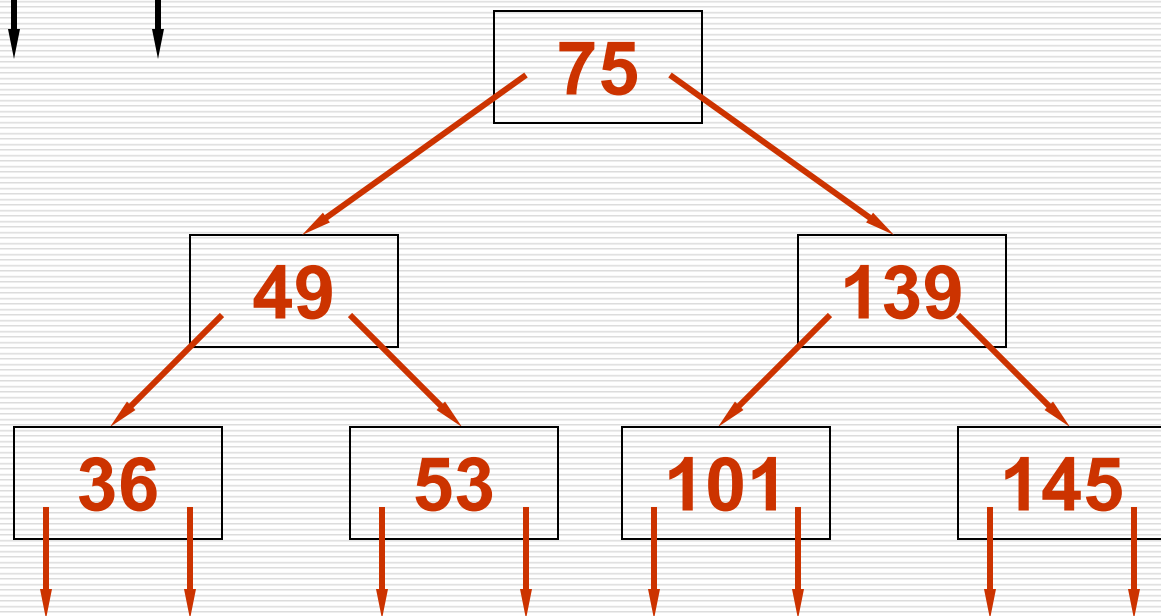
n=6 加入36



$n=6$  加入36



$n=7$  加入101



在插入新关键码时，需要自底向上分裂结点，  
**最坏情况：**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。

- 分裂一个非根的结点：
- 分裂根结点：

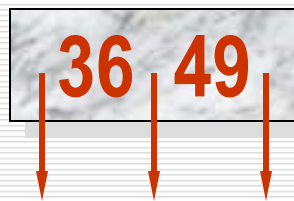
写出 2 个结点  
写出 3 个结点

## B 树的删除

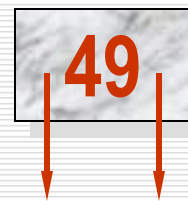
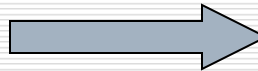
- 在B 树上删除一个关键码时，
  - ◆ 首先需要找到这个关键码所在的结点，从中删去这个关键码。
  - ◆ 若该结点不是叶结点，且被删关键码为  $K_i$ ,  $1 \leq i \leq n$ , 则在删去该关键码之后，应以该结点  $P_i$  所指示子树中的最小关键码  $x$  来代替被删关键码  $K_i$  所在的位置；
  - ◆ 然后在  $x$  所在的叶结点中删除  $x$ 。
- 在叶结点上的删除：有 4 种情况。

- 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。

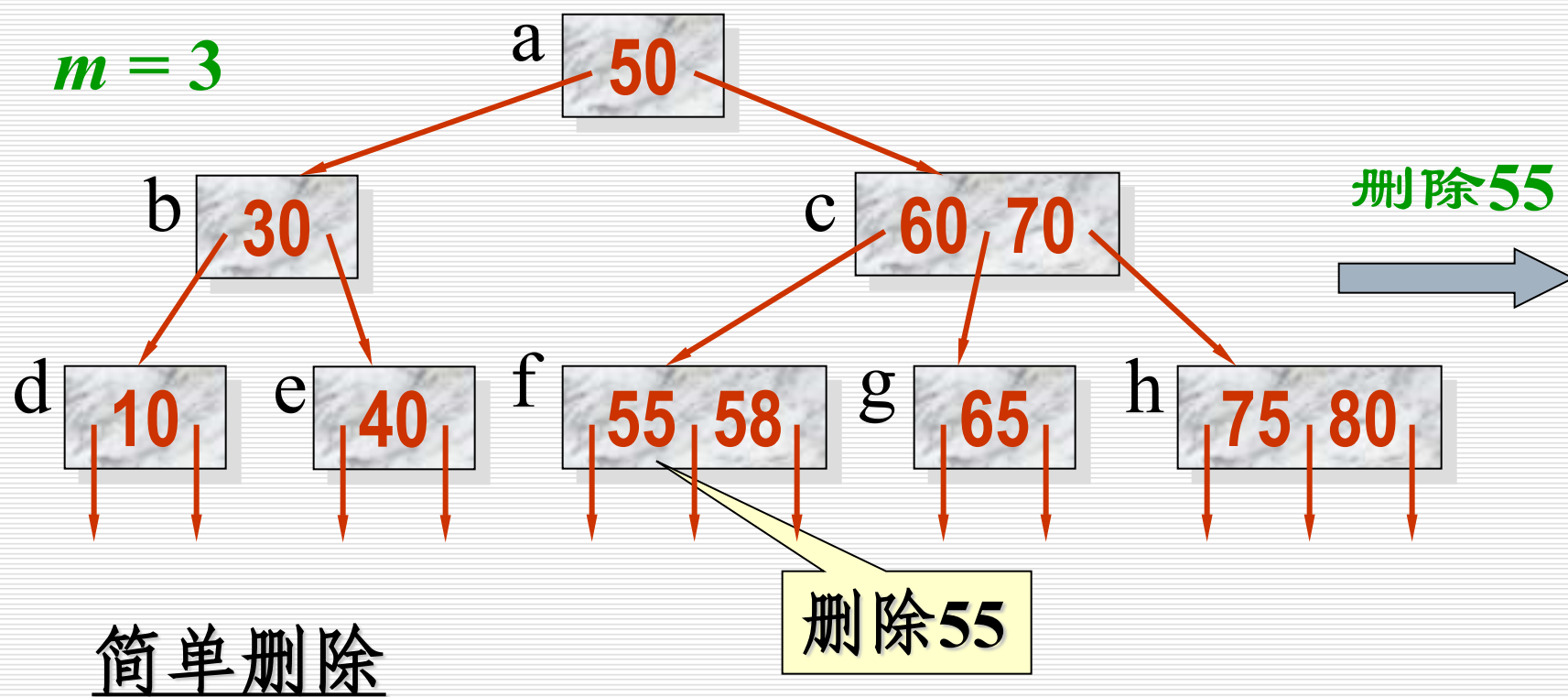
$m = 3$

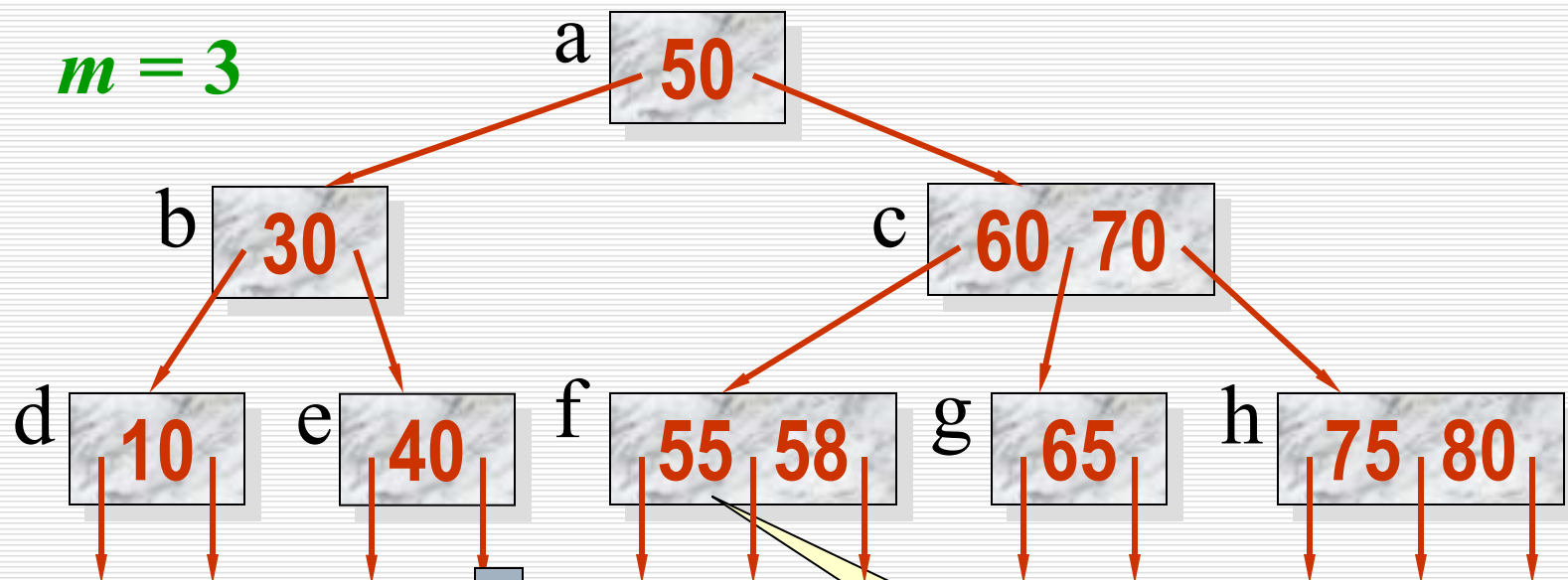


删除36

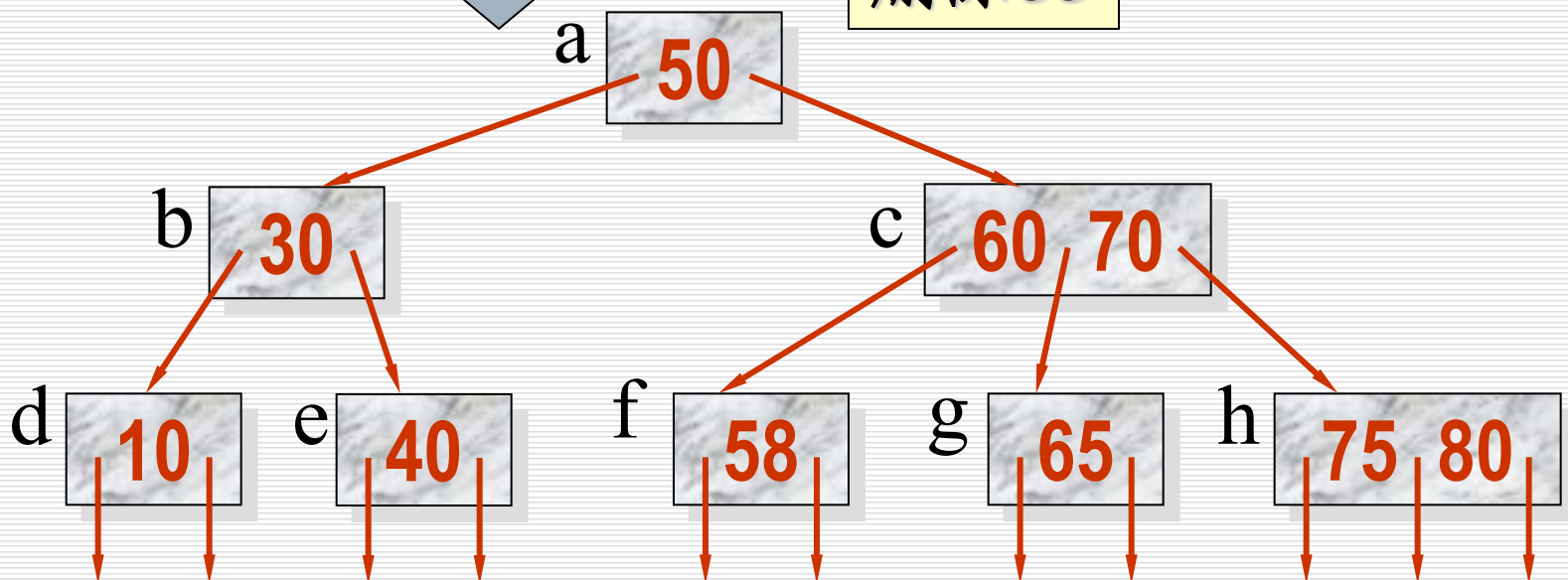
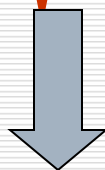


② 被删关键码所在叶结点不是根结点且删除前该结点中关键码个数  $n \geq \lceil m/2 \rceil$ , 则直接删去该关键码并将修改后的结点写回磁盘, 删除结束。



$m = 3$ 

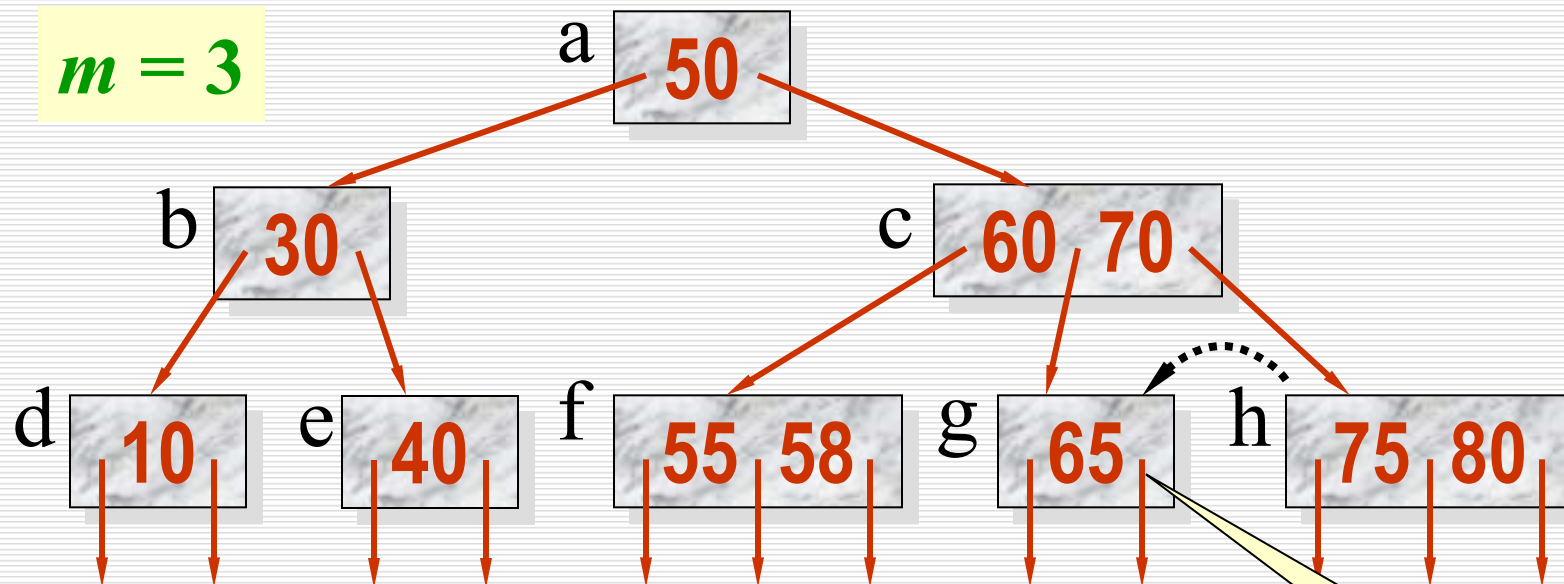
删除55



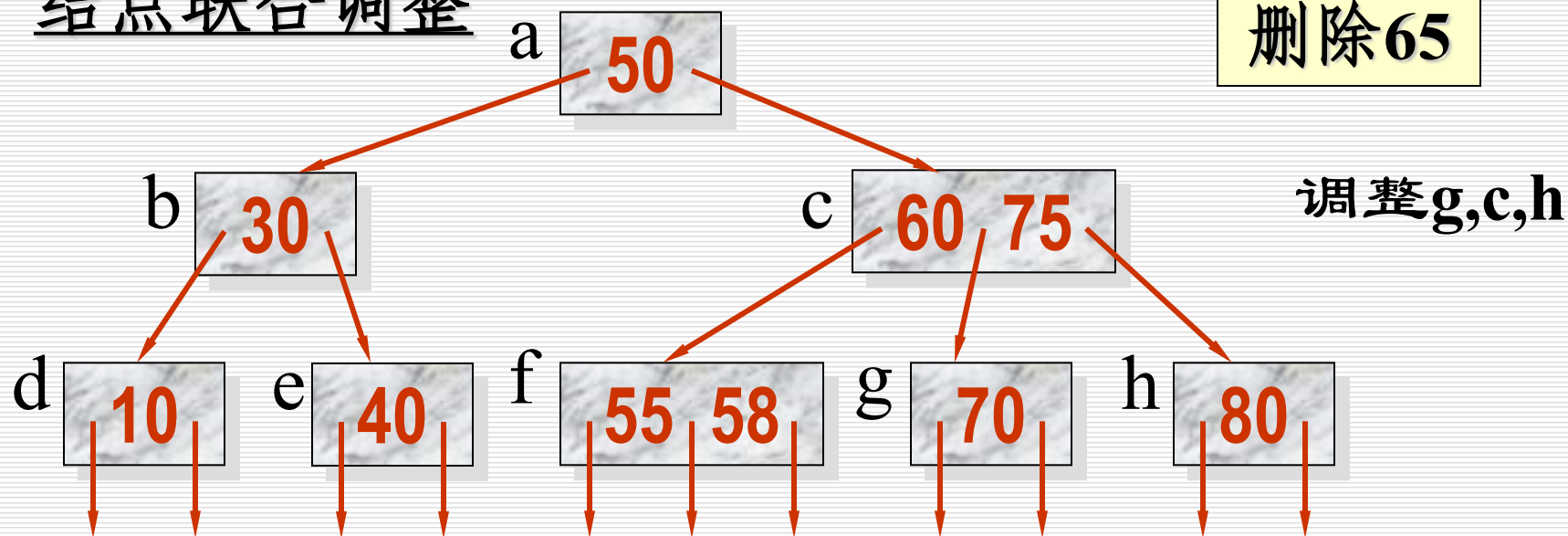
③ 被删关键码所在叶结点删除前关键码个数  $n = \lceil m/2 \rceil - 1$ , 若这时与该结点相邻的右兄弟（或左兄弟）结点的键码个数  $n \geq \lceil m/2 \rceil$ , 则可按以下步骤调整该结点、右兄弟（或左兄弟）结点以及其双亲结点, 以达到新的平衡。

- 将双亲结点中刚刚大于（或小于）该被删键码的键码  $K_i$  ( $1 \leq i \leq n$ ) 下移;
- 将右兄弟（或左兄弟）结点中的最小（或最大）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键码所在结点中最后（或最前）子树指针位置;
- 在右兄弟（或左兄弟）结点中, 将被移走的键码和指针位置用剩余的键码和指针填补、调整。再将结点中的键码个数减1



$m = 3$ 

结点联合调整



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) 结点的键码个数  $n = \lceil m/2 \rceil - 1$ , 则 “合并”。

- 假设合并至左兄弟, 且其结点地址由双亲结点中的指针  $P_{i+1}$  所指, 则在删除关键字后, 它所在结点中的剩余信息, 加上双亲结点中关键字  $K_{i+1}$  一起, 合并至  $P_i$  所指结点中。
- 假设合并至右兄弟, 且其结点地址由双亲结点中的指针  $P_i$  所指, 则在删除关键字后, 它所在结点中的剩余信息, 加上双亲结点中关键字  $K_{i+1}$  一起, 合并至  $P_{i+1}$  所指结点中。

具体方法（合并至左兄弟）：

- 将双亲结点  $p$  中相应关键码下移到选定保留的结点（左兄弟）中。若要合并  $p$  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。
- 把  $p$  中子树指针  $P_{i+1}$  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。
- 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  $P_{i+1}$ 。
- 修改结点  $p$  和选定保留结点的关键码个数。

具体方法（合并至右兄弟）：

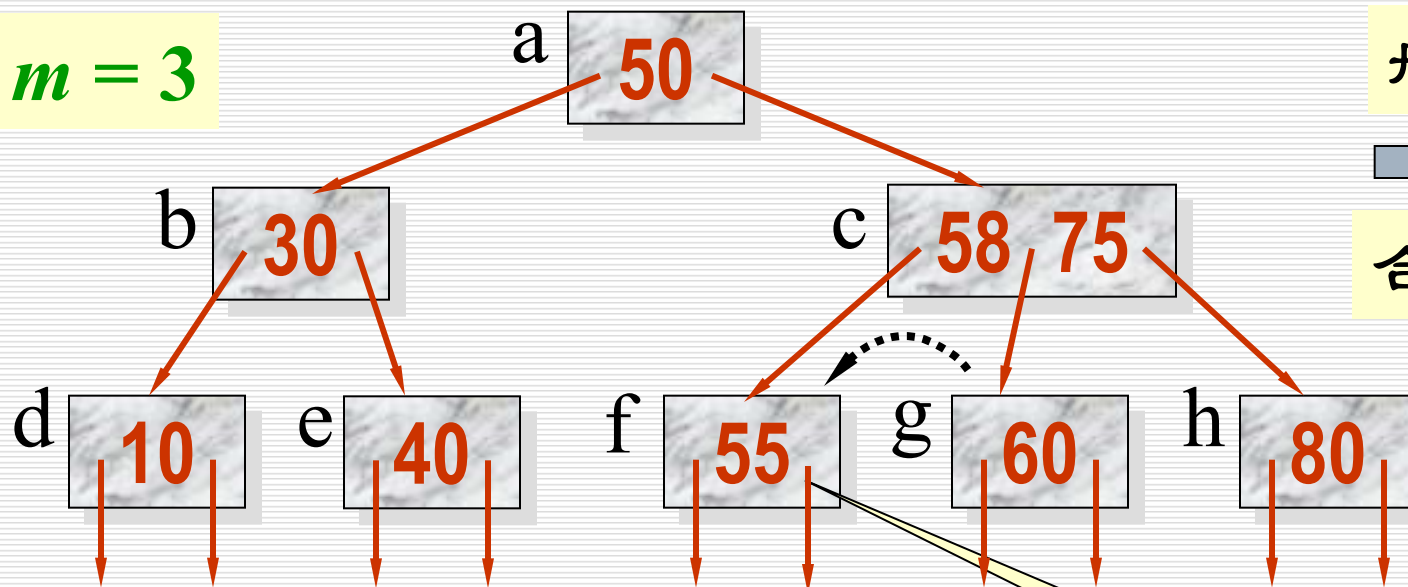
- 将双亲结点  $p$  中相应关键码下移到选定保留的结点（左兄弟）中。若要合并  $p$  中的子树指针  $P_i$  与  $P_{i+1}$  所指的结点，且保留  $P_{i+1}$  所指结点，则把  $p$  中的关键码  $K_{i+1}$  下移到  $P_{i+1}$  所指的结点中。
- 把  $p$  中子树指针  $P_i$  所指结点中的全部指针和关键码都照搬到  $P_{i+1}$  所指结点的前面。删去  $P_i$  所指的结点。
- 在结点  $p$  中用前面剩余的关键码和指针填补关键码  $K_{i+1}$  和指针  $P_i$ 。
- 修改结点  $p$  和选定保留结点的关键码个数。

$m = 3$ 

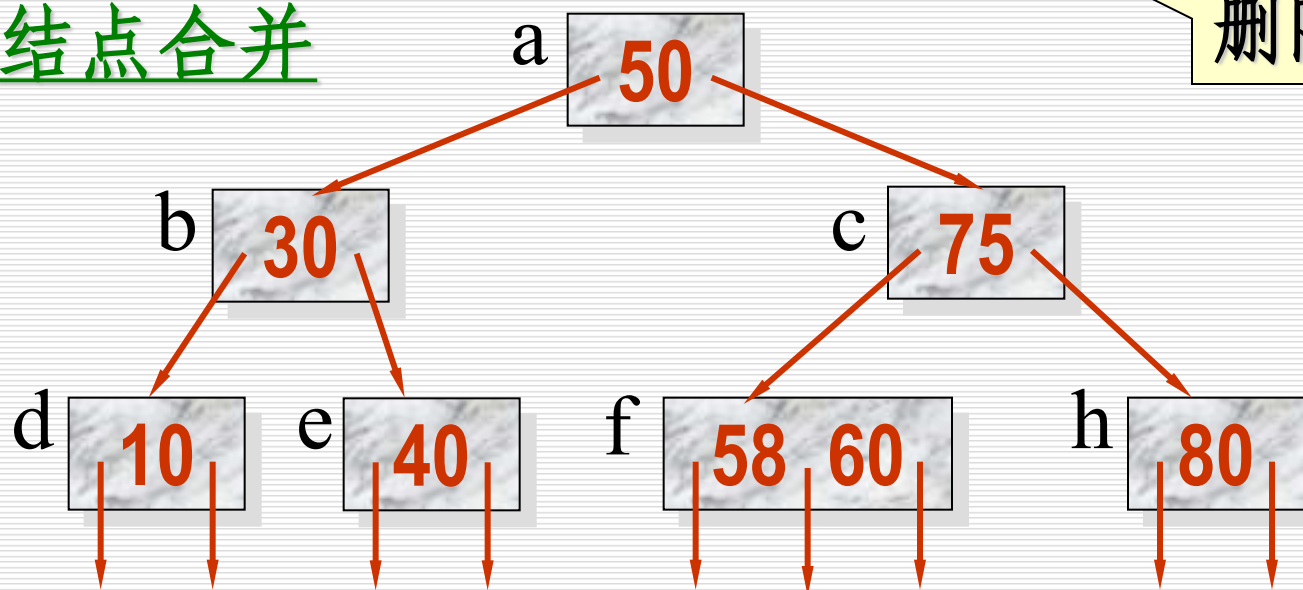
删除55

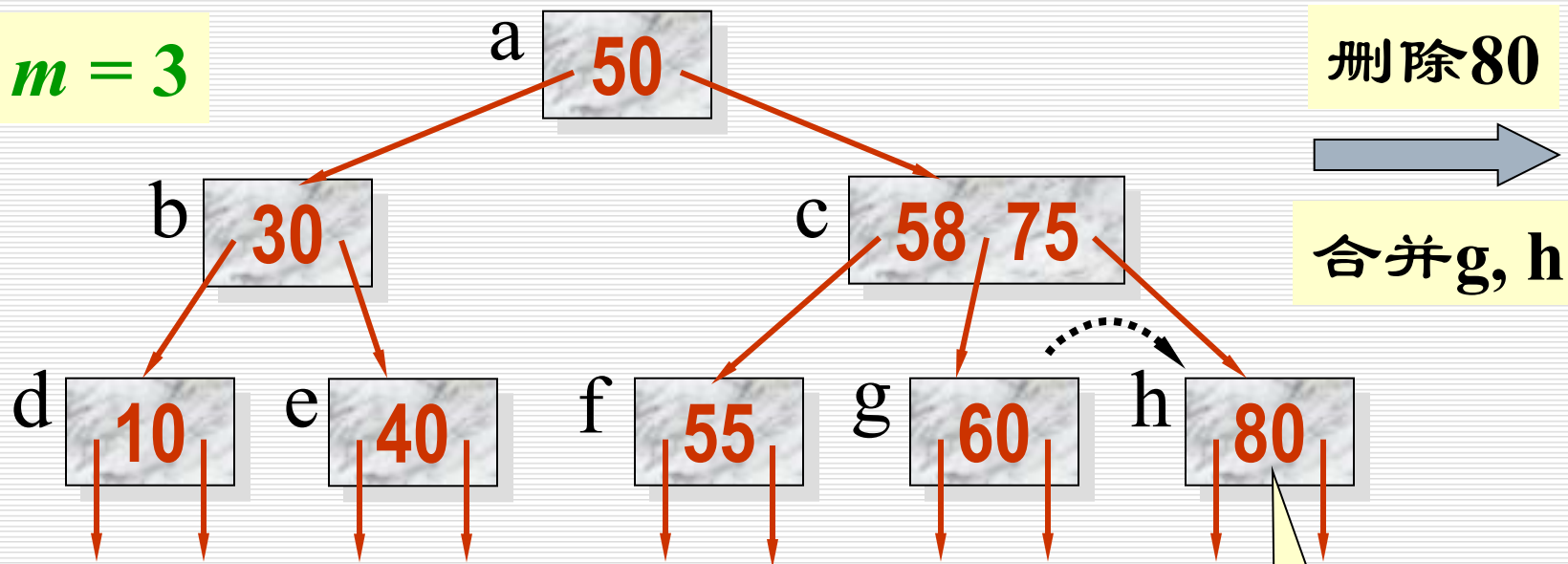
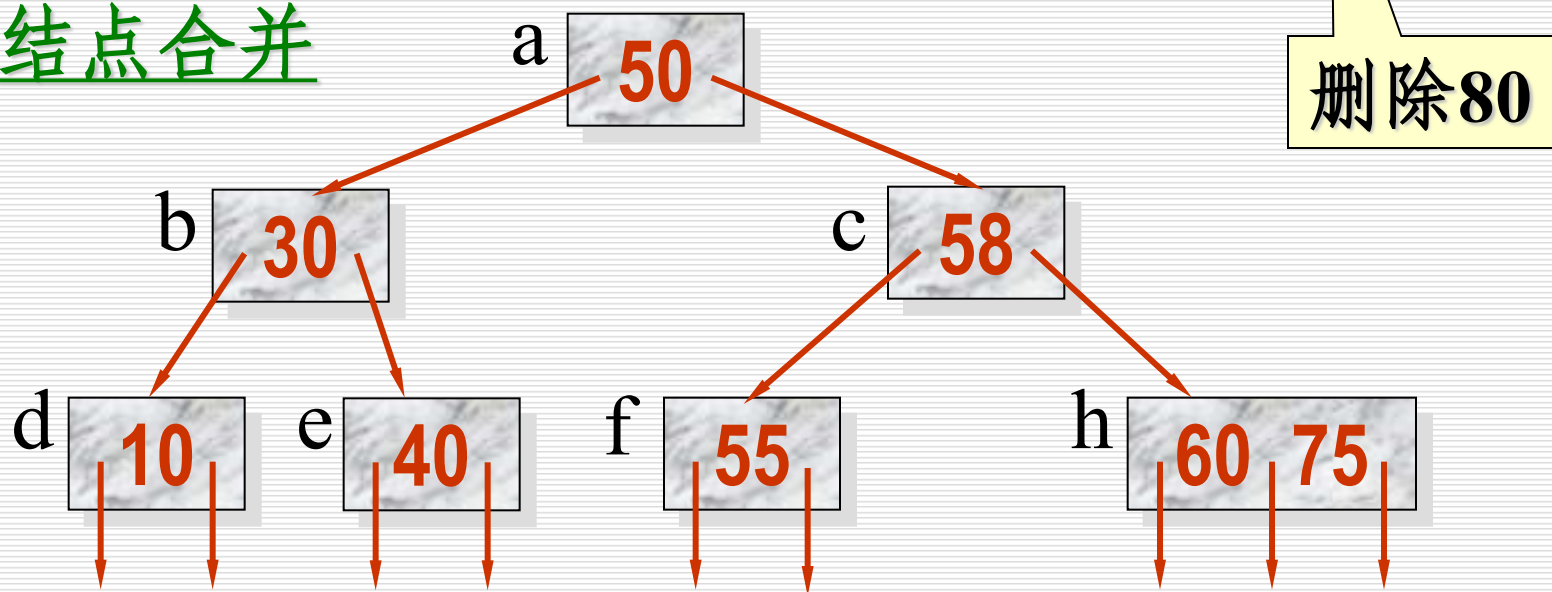


合并f, g

结点合并

删除55



$m = 3$ 结点合并

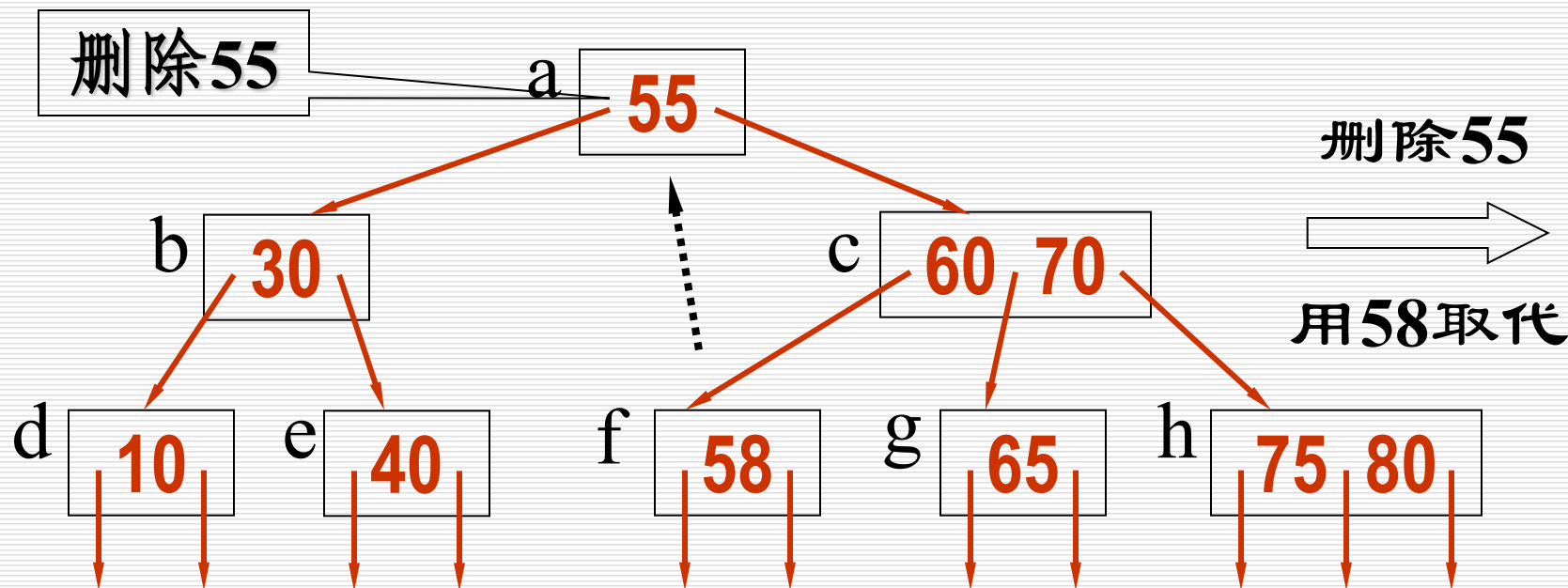
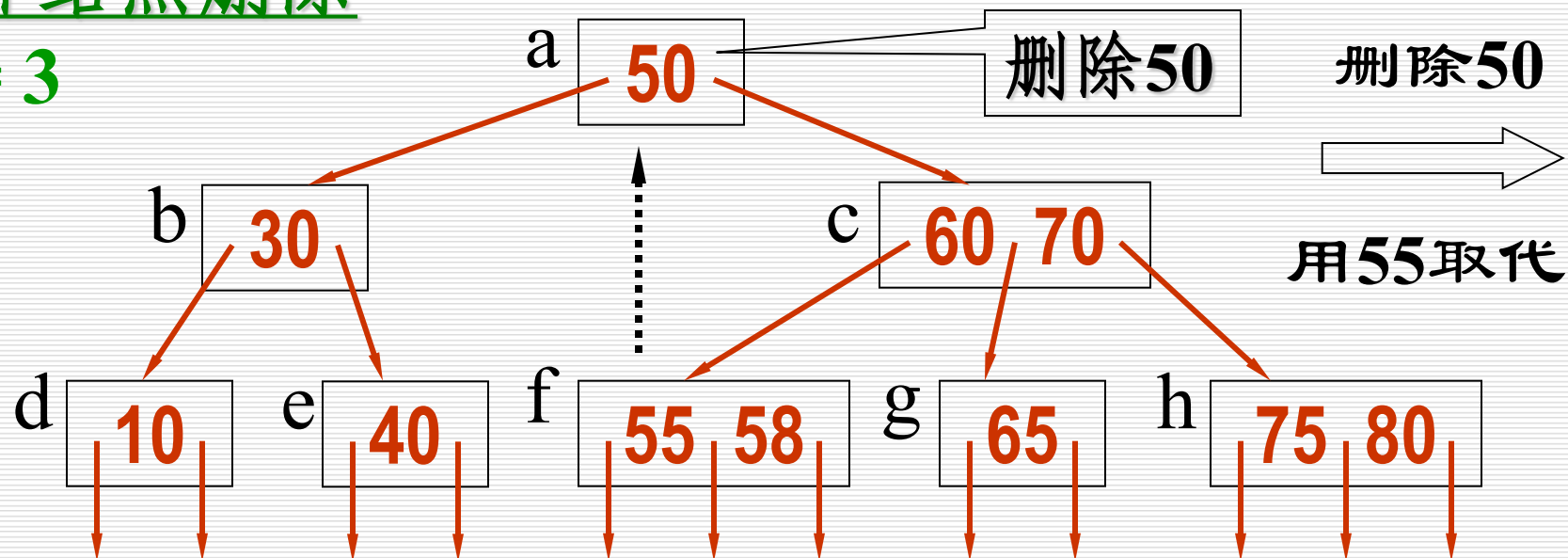


- 在合并结点的过程中, 双亲结点中的关键码个数减少了。
- 若双亲结点是根结点且结点关键码个数减到 0, 则将该双亲结点删去, 合并后保留的结点成为新的根结点; 否则将双亲结点与合并后保留的结点都写回磁盘, 删除处理结束。
- 若双亲结点不是根结点且关键码个数减到  $\lceil m/2 \rceil - 2$ , 又要与它自己的兄弟结点合并, 重复上面的合并步骤。

最坏情况下这种结点合并处理要自下向上直到根结点。

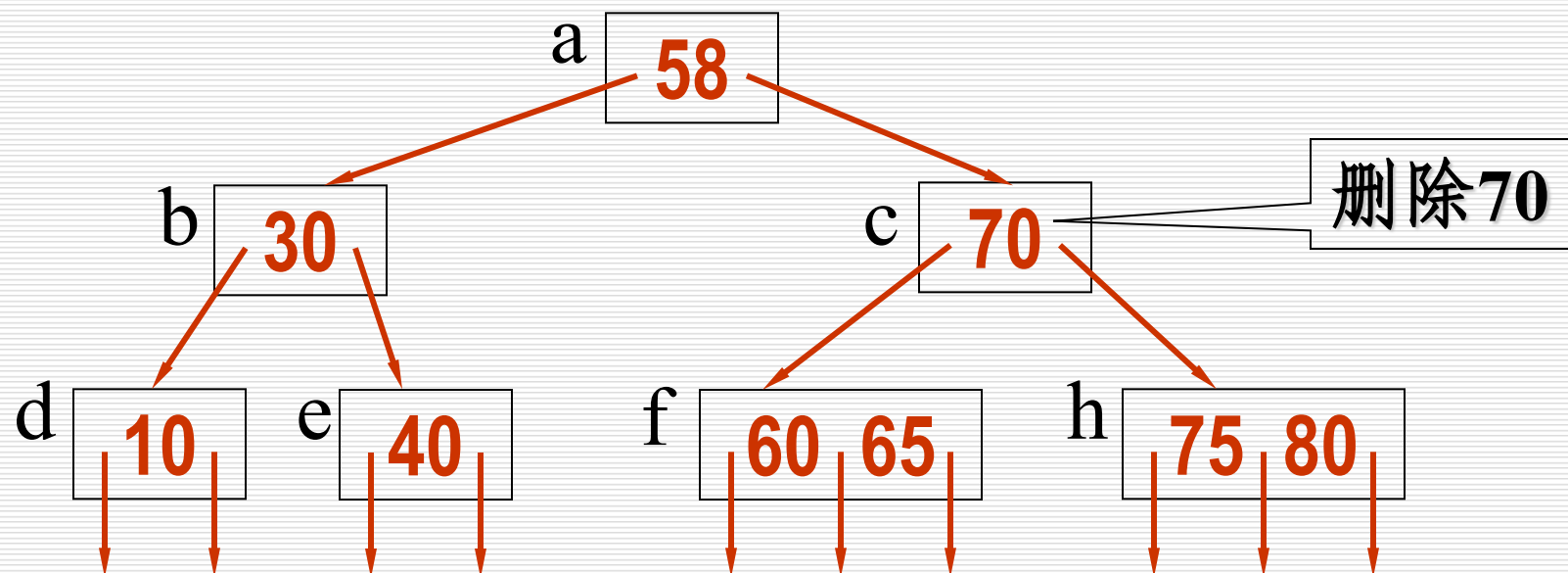
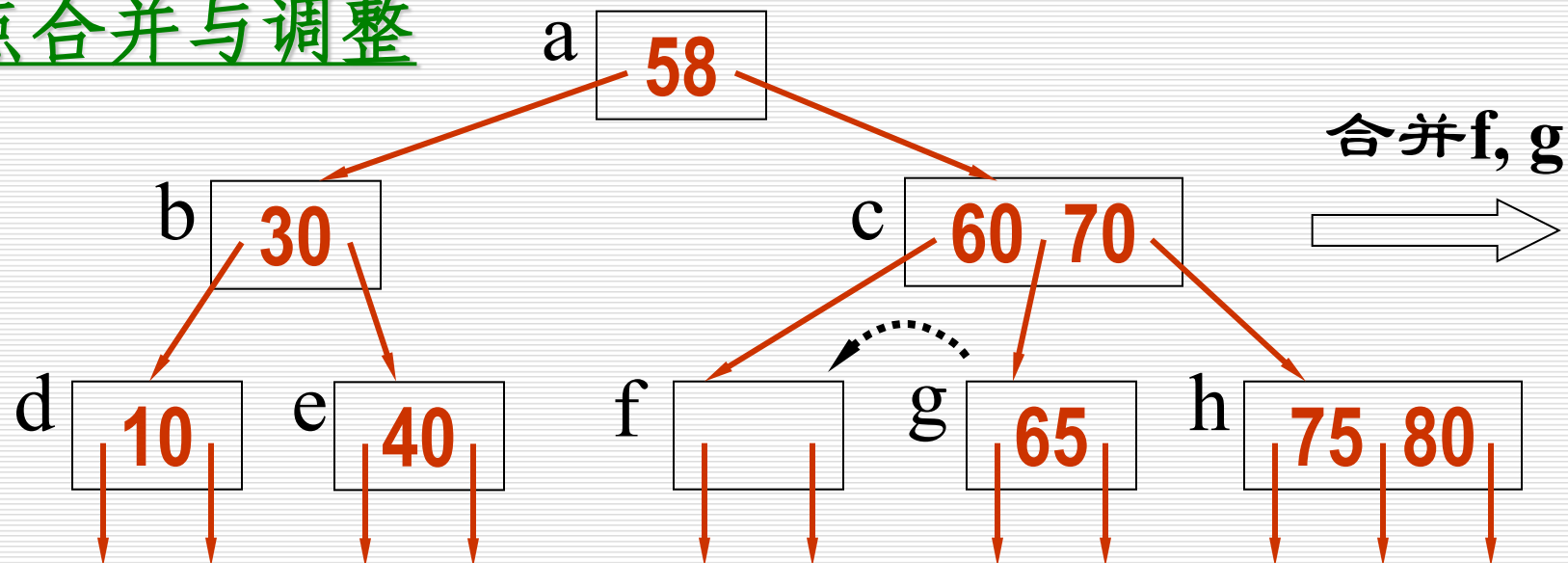
# 非叶结点删除

$m = 3$

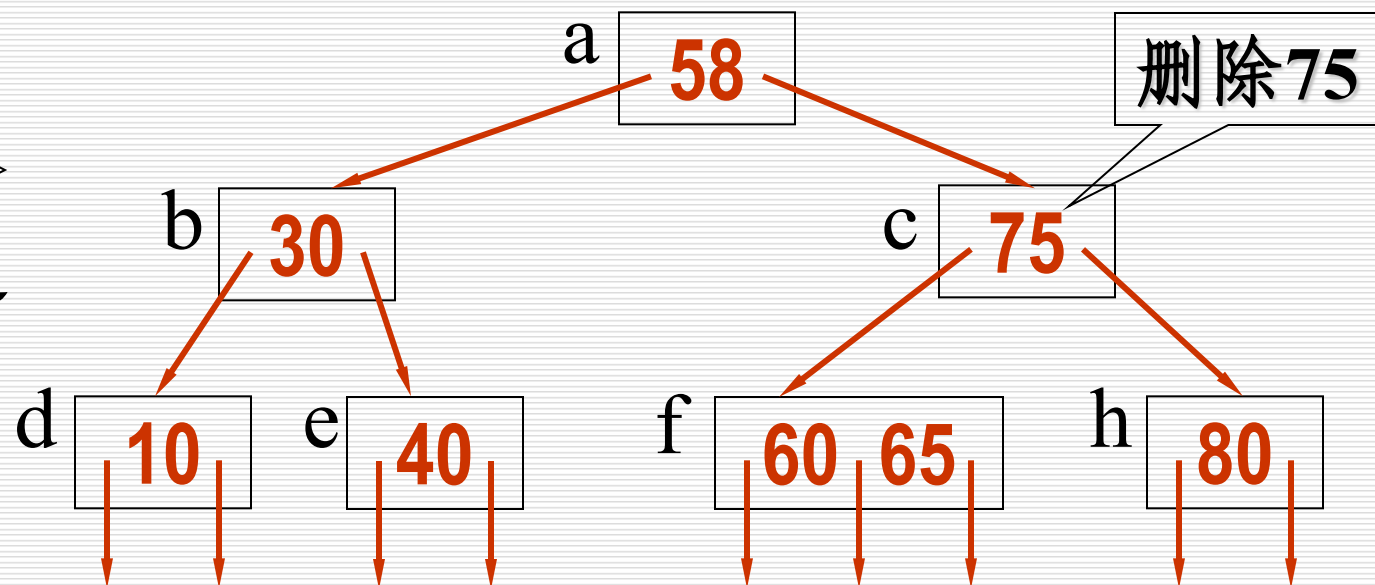




# 结点合并与调整



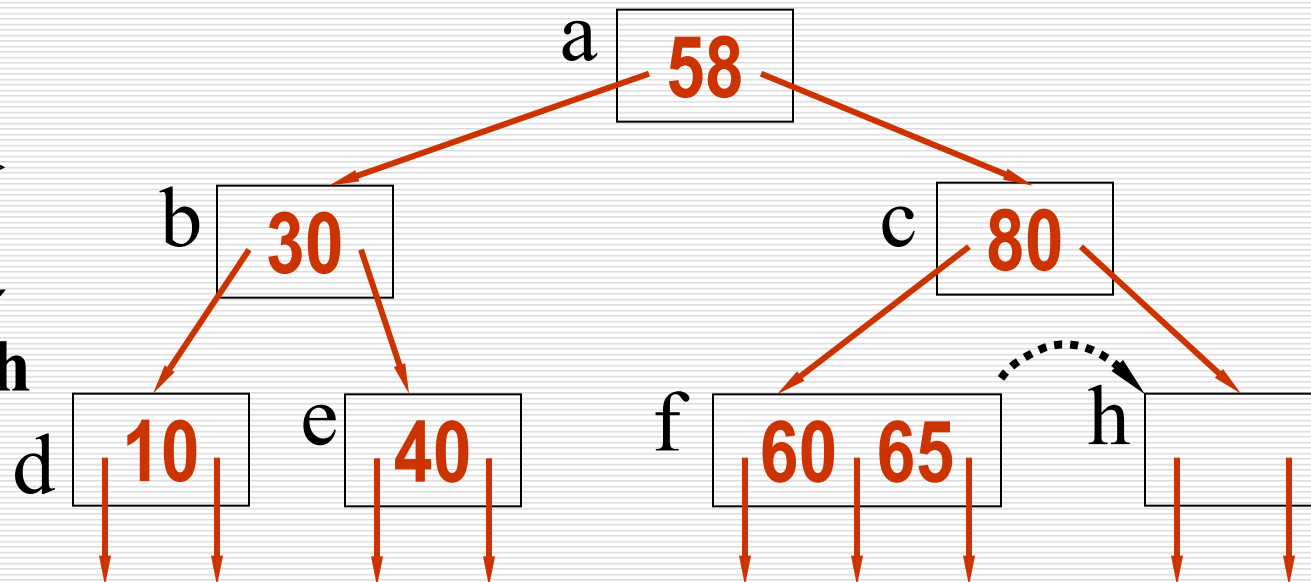
删除70  
→  
用75取代



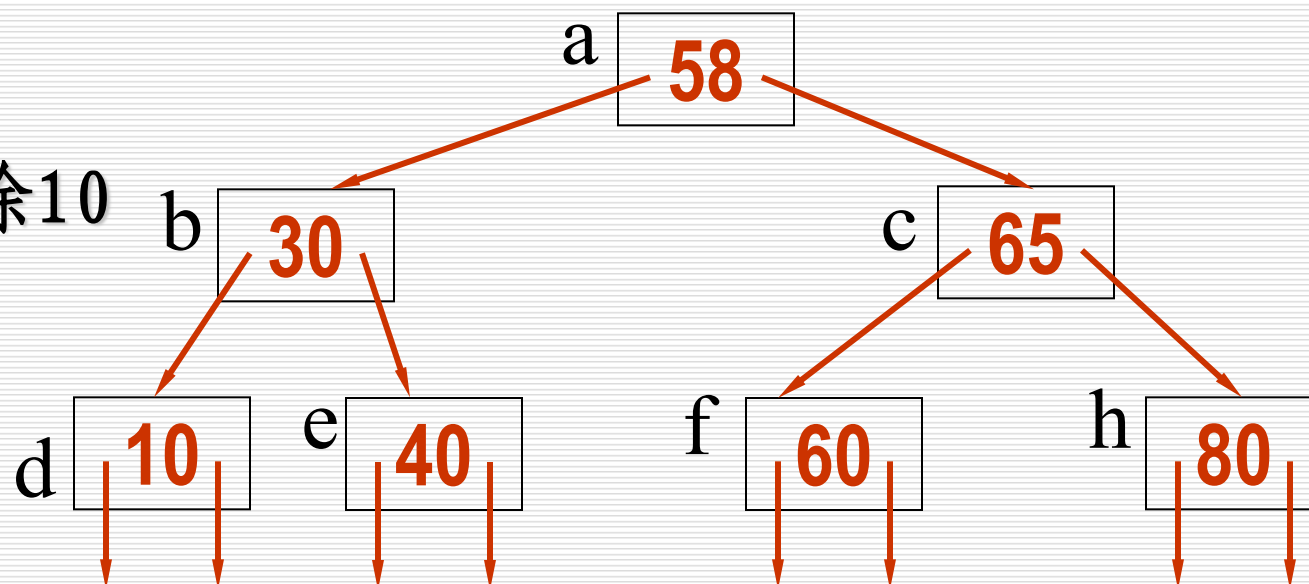
删除75

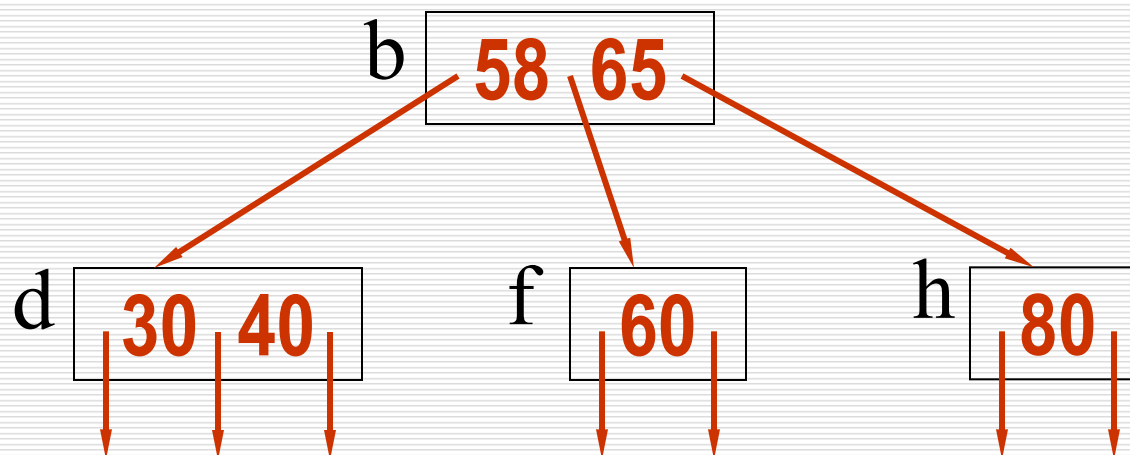
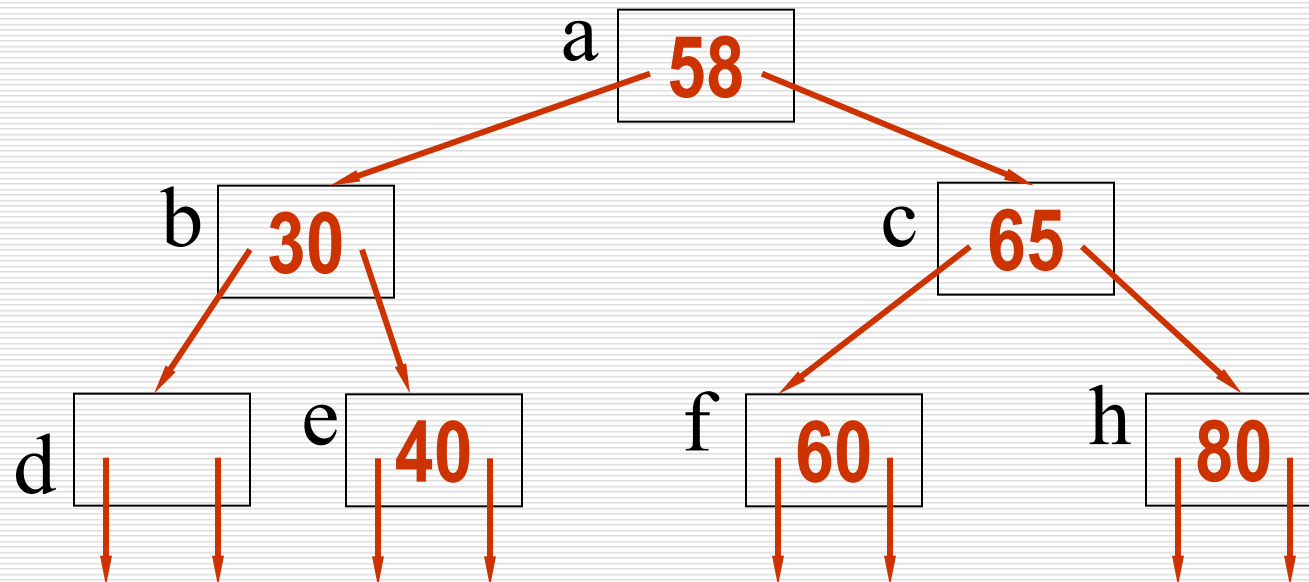


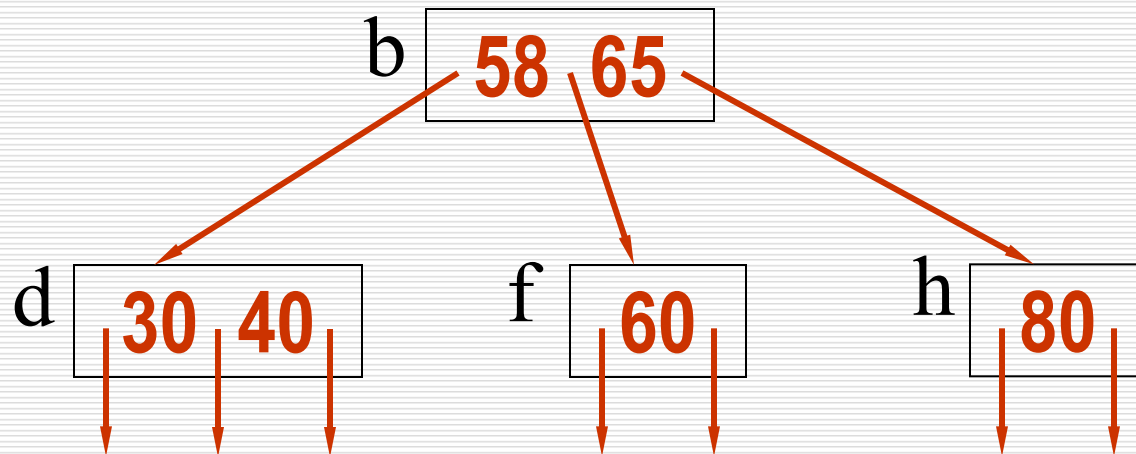
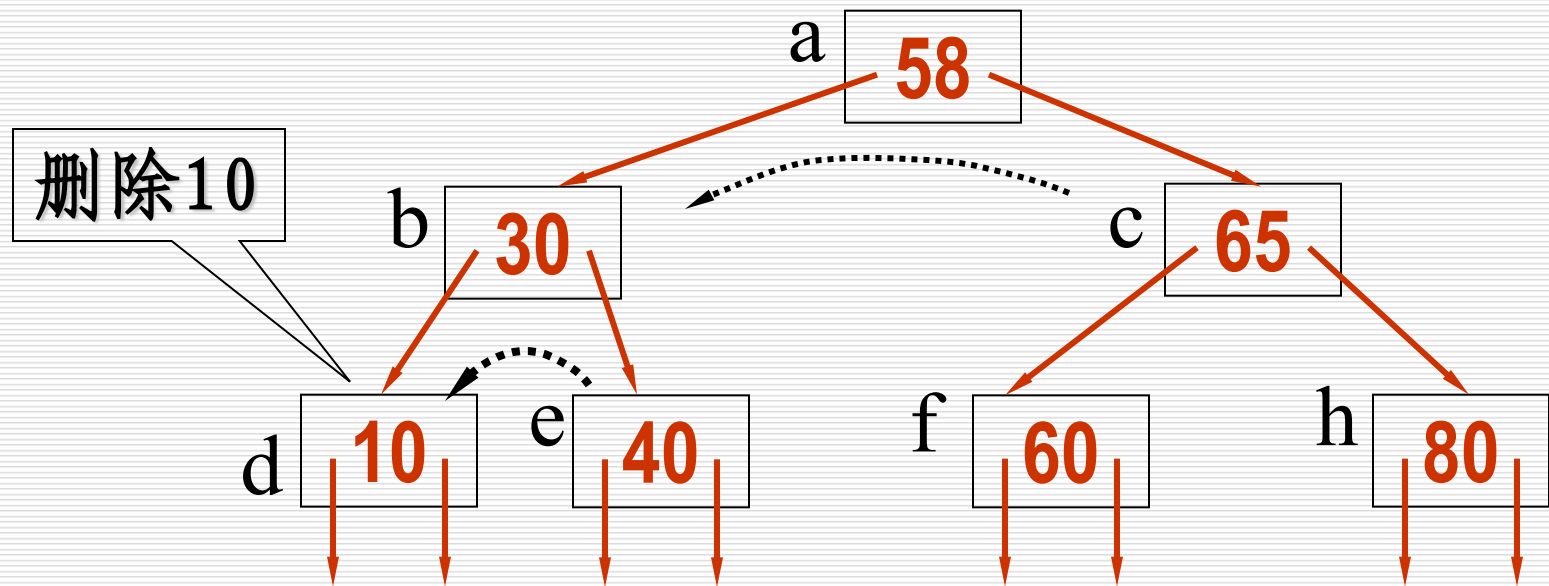
用80取代  
调整f, c, h



删除10







## B 树的关键码删除算法

```
template <class Type> int Btree<Type> ::  
Delete ( const Type & x ) {  
    Triple<Type> loc = Search (x);    //搜索x  
    if ( loc.tag ) return 0;           //未找到, 不删除  
    Mnode<Type> *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); q = p;  
        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;                                //转化为叶结点的删除
}
else compress ( p, j );                  //叶结点, 直接删除
int d = (m+1)/2;                          //求 $\lceil m/2 \rceil$ 
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);
        while ( j <= q->n && q->ptr[j] != p )
            j++;
        if ( !j ) 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 ) {                             //调整后根的n减到0
    p = root->ptr[0];
    delete root; root = p;                  //删根
    root->parent = NULL;                      //新根
}
}

```



# 红黑树的删除 和B树的删除

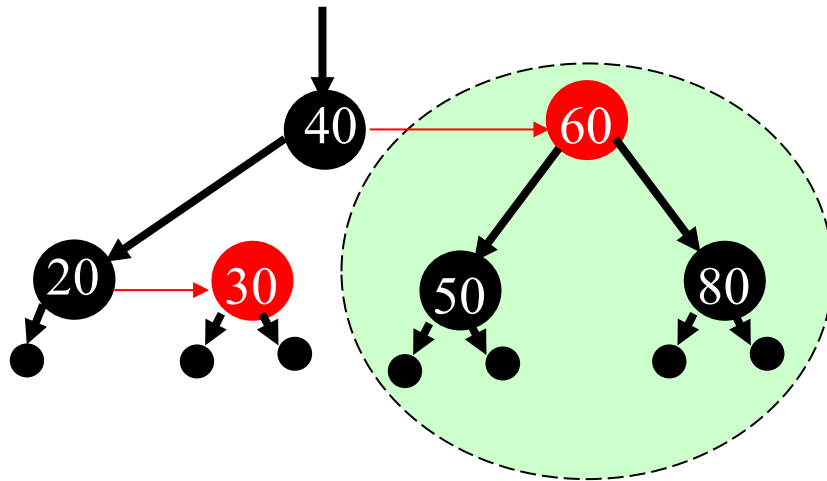
# B\_树

一棵  $m$  阶B\_树是一棵  $m$  路搜索树，它或者是空树，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。

# 红黑树和B树的对应

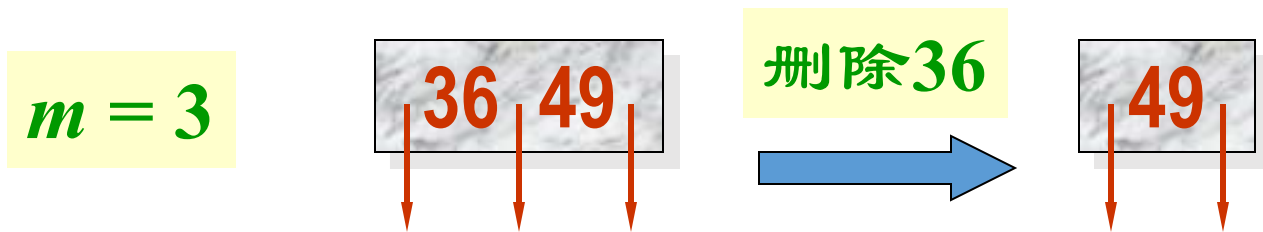
- 红黑树实际上对应于4阶B树
  - 每个黑色结点和它的红色子女结点可以看作是相应的B树的一个结点。这些结点的子树看作是B树中对应结点的子树。
  - 每个结点最多4个子树，也就是3个Key
  - 每个结点最少2个子树，也就是1个Key



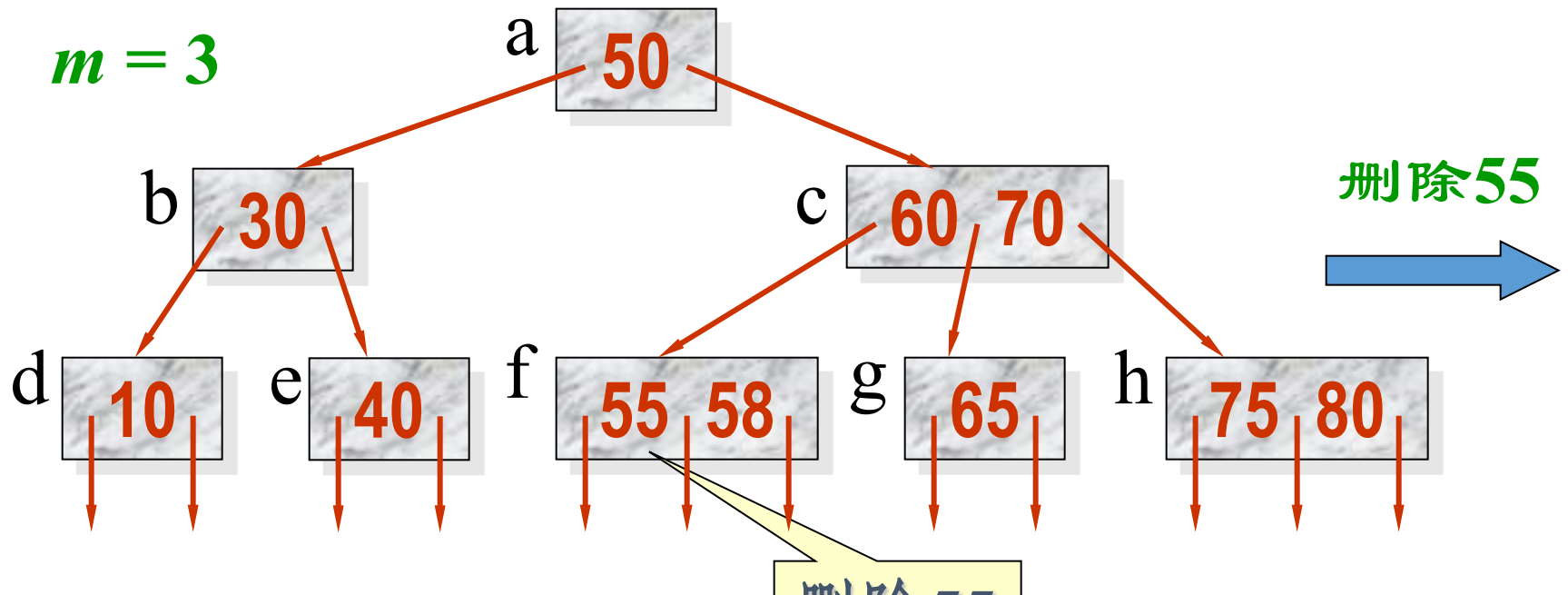
# B 树的删除

- 在B 树上删除一个关键码时，
  - ◆ 首先需要找到这个关键码所在的结点，从中删去这个关键码。
  - ◆ 若该结点不是叶结点，且被删关键码为  $K_i$ ,  $1 \leq i \leq n$ , 则在删去该关键码之后，应以该结点  $P_i$  所指示子树中的最小关键码  $x$  来代替被删关键码  $K_i$  所在的位置；
  - ◆ 然后在  $x$  所在的叶结点中删除  $x$ 。
- 在叶结点上的删除：有 4 种情况。

- 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。

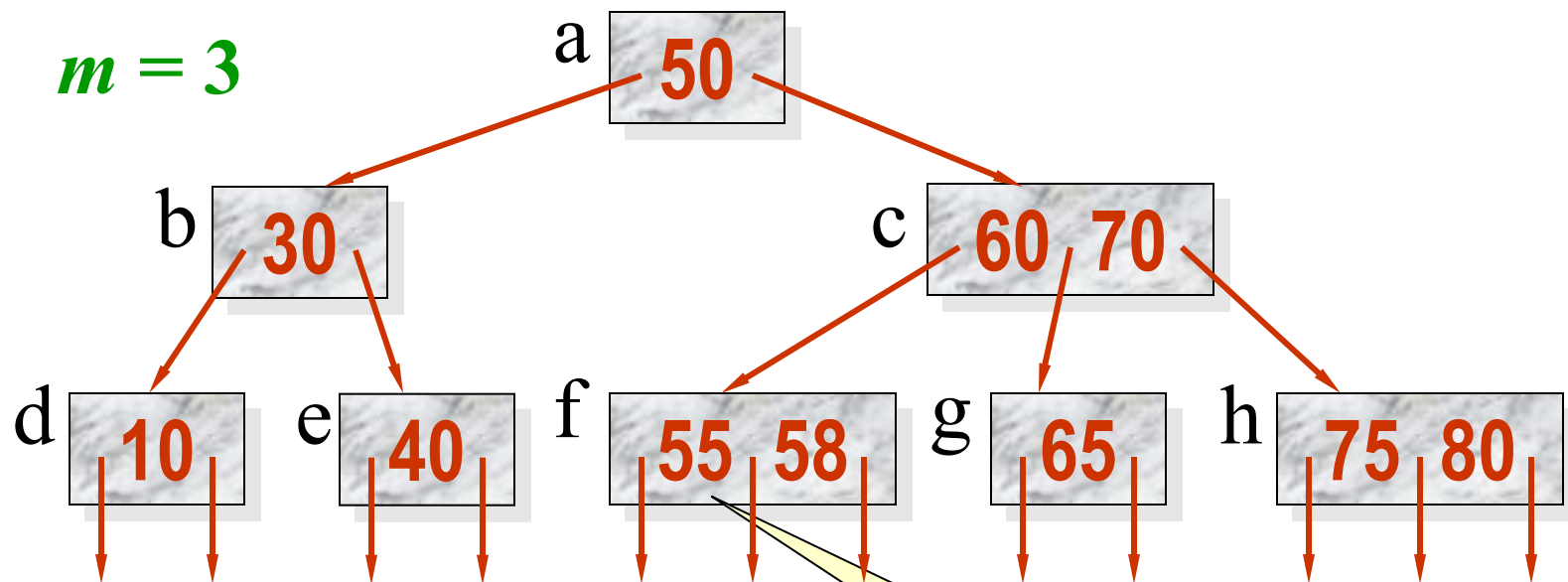


简单删除

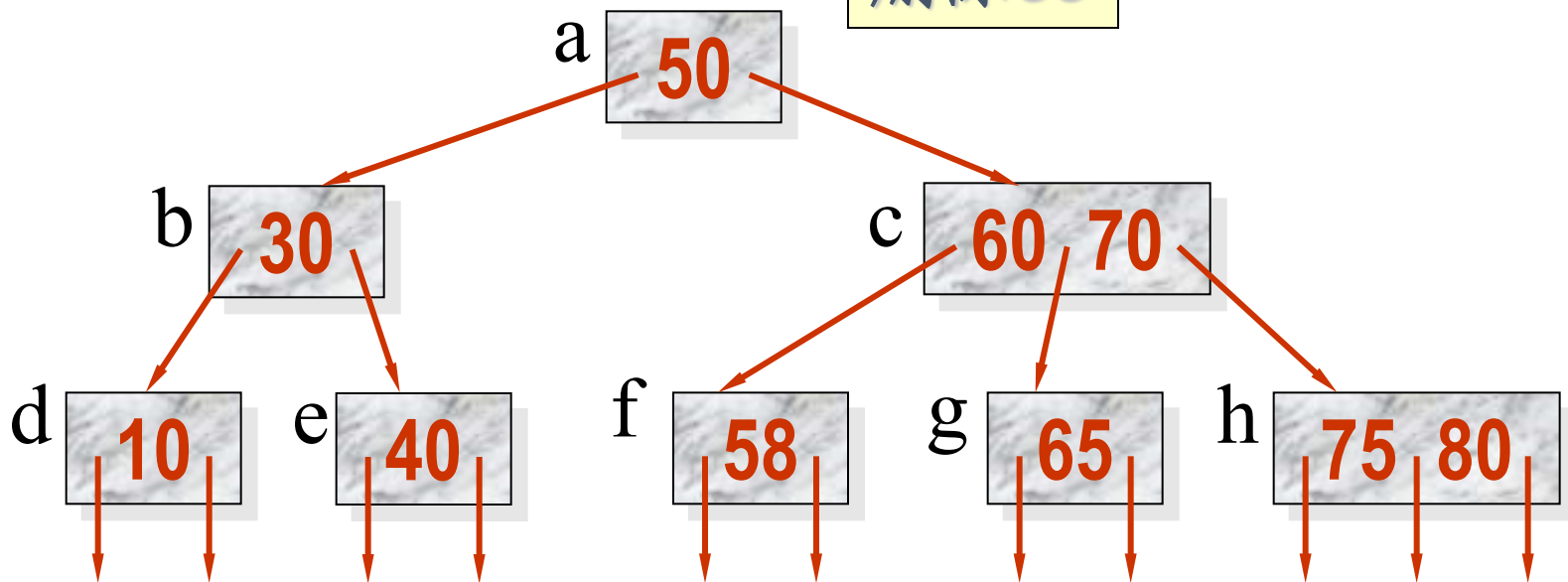
这个实际上相当于在红黑树中删除了

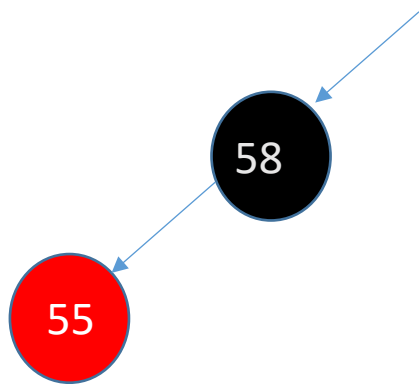
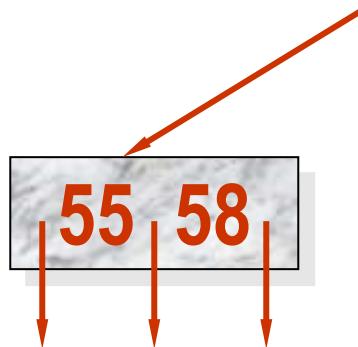
- 1、一个红结点
- 2、或者删除了一个黑色结点, 但是该黑色结点有一个红色子节点

$m = 3$

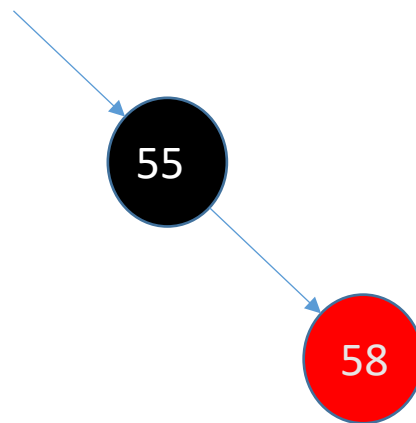


删除55





情况一



情况二

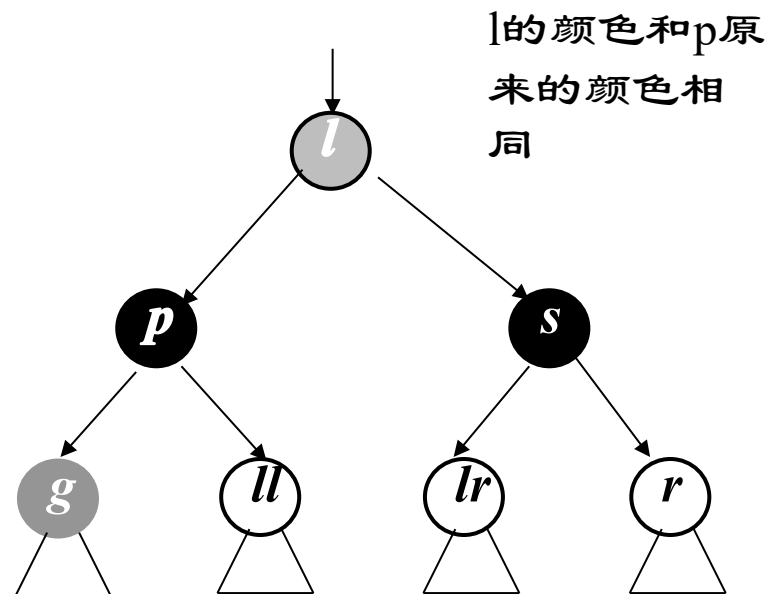
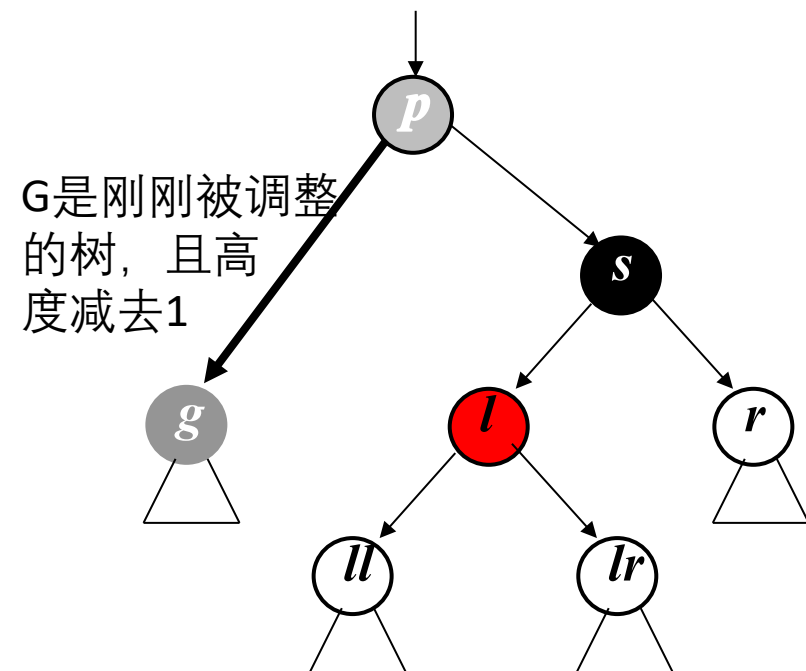


- 被删关键码所在叶结点删除前关键码个数  $n = \lceil m/2 \rceil - 1$ , 那么删除之后Key的数量为个数  $n = \lceil m/2 \rceil - 2$ , 不满足B树的要求
  - 若这时与该结点相邻的右兄弟 (或左兄弟) 结点的键码个数  $n \geq \lceil m/2 \rceil$ , 则可从兄弟结点 “借” 一个Key来保证B树的结点中至少有  $n = \lceil m/2 \rceil - 1$  个Key, 满足B树的要求。
  - 被删关键码所在叶结点删除前关键码个数  $n = \lceil m/2 \rceil - 1$ , 若这时与该结点相邻的右兄弟 (或左兄弟) 结点的键码个数  $\lceil m/2 \rceil - 1$ , 则 “合并”。合并时会把父亲结点中的一个Key加入到合并后的结点中, 因此合并后的结点有  $\lceil m/2 \rceil * 2 - 2$  个Key。  $\lceil m/2 \rceil * 2 - 2 \leq m - 1$ 。

## □ 特殊情况:

- 如果  $m$  为偶数, 兄弟结点的Key的数量为  $\lceil m/2 \rceil$ , 将当前结点和兄弟结点合并也是可行的。合并后的Key数量为有  $\lceil m/2 \rceil * 2 - 1$  个。因为  $m$  为偶数, 所以仍然保证了合并后的结点的Key小于等于  $m - 1$  个。
- 红黑树相当于4阶B树, 所以当有一个结点删除前有1个Key, 兄弟结点2个Key时, 仍然可以合并。

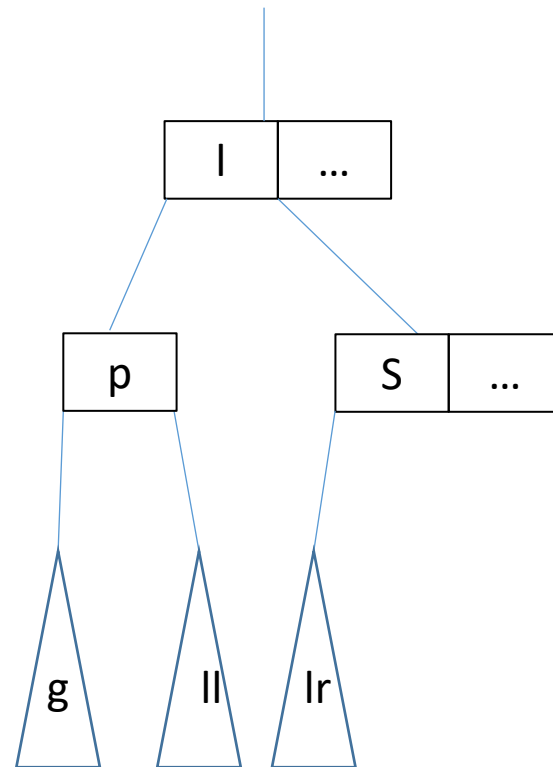
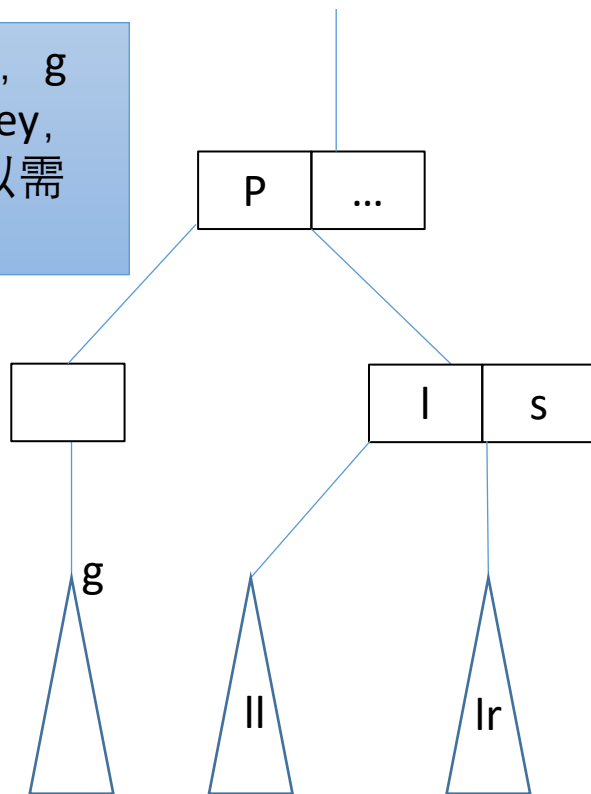
# Repair: L is Red



- 从小到大：g, p, ll, l, lr, s, r
- ll, lr的颜色为黑色
- g, ll, lr, r的Black Height相等
- 调整后，Black Height和原来的子树高度不变

# L is Red的B树视图

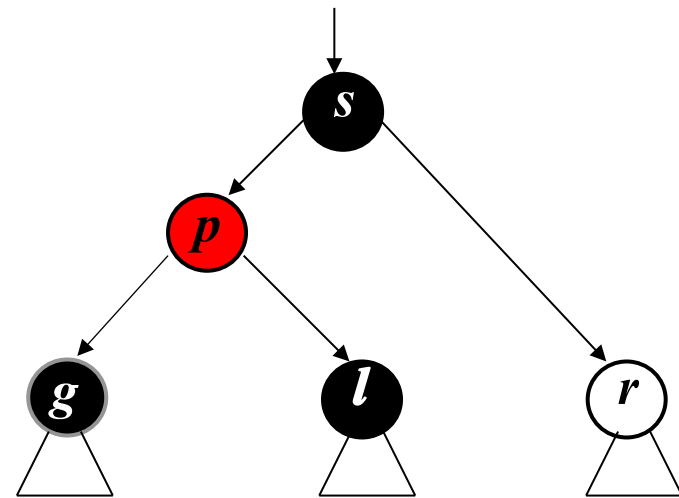
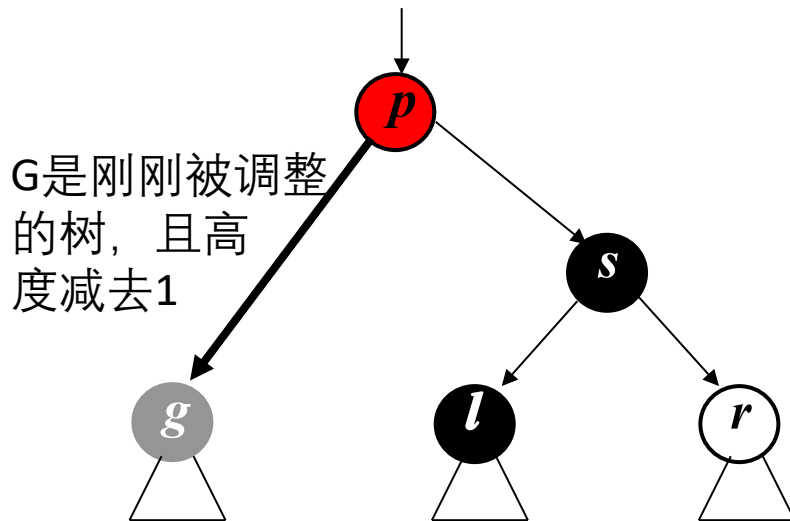
从B树的视点看，g  
的父结点没有Key，  
是空结点，所以需要  
处理。



实际上就是按照上面B树删除结点的过程：

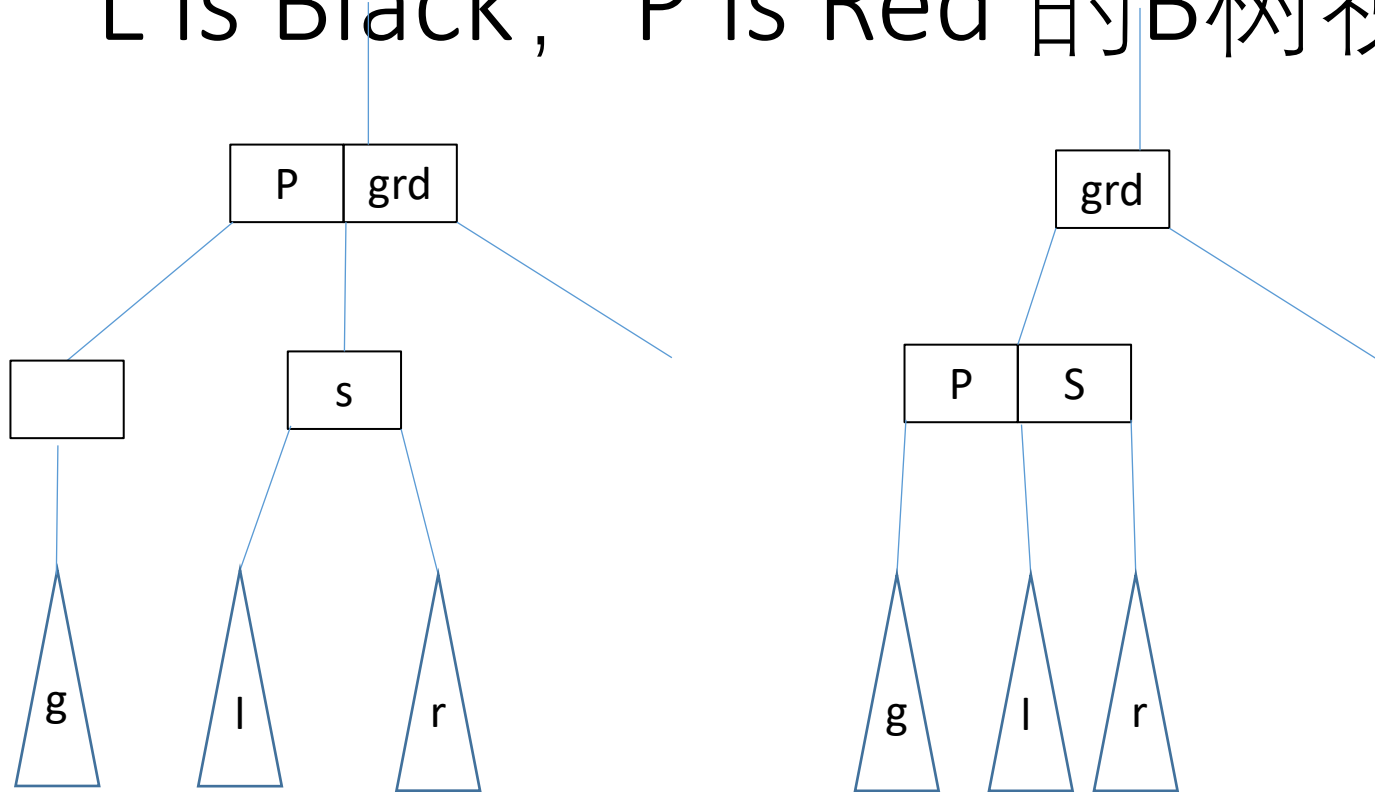
- 1、P下移
- 2、l上移
- 3、子树相应地调整

# Repair: l is black, p is red



- 如果p为红色，则s必然为黑色
- 从小到大：g,p,l,s,r
- l和r的Black Height等于g的Black Height
- 转换后，新树的高度和原来树的Black Height不变

# L is Black, P is Red 的B树视图

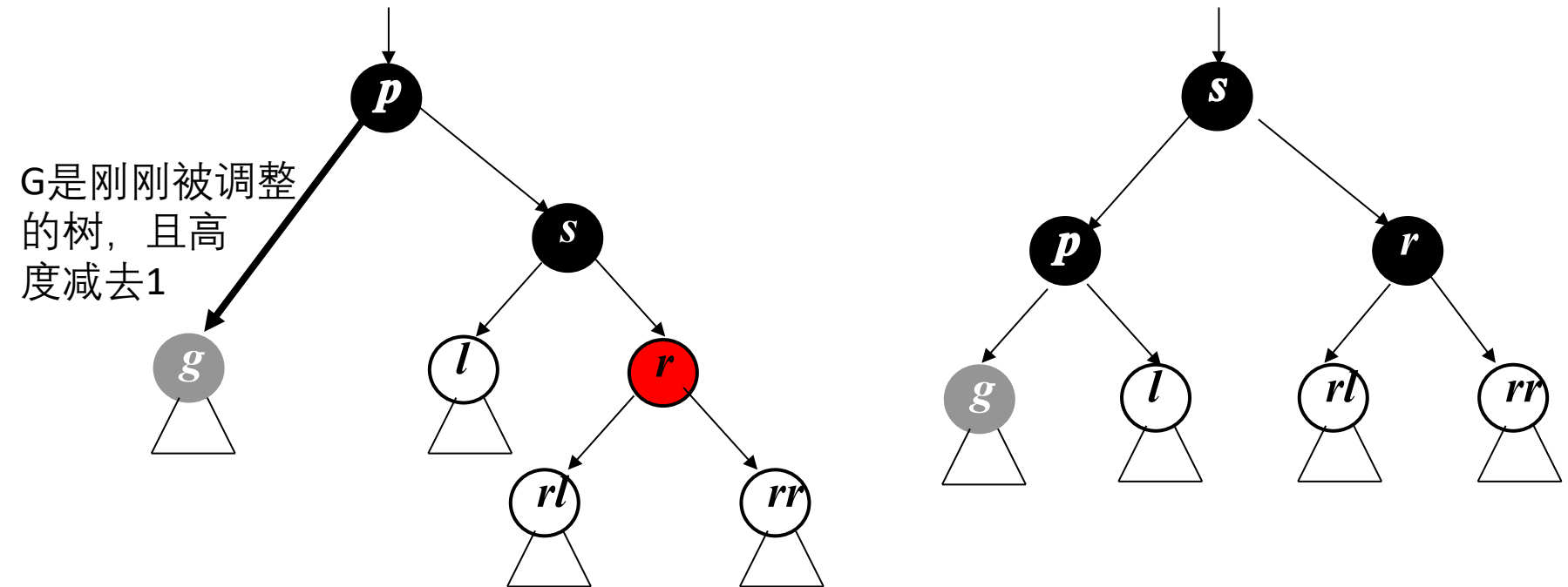


这个实际上是和兄弟结点合并的处理：

- 1、 P是红色，所以P必然是P所在B树结点的左边结点。
- 2、 l是红色，所以s最多有一个红色子节点，即兄弟结点中最多两个Key。因此可以合并

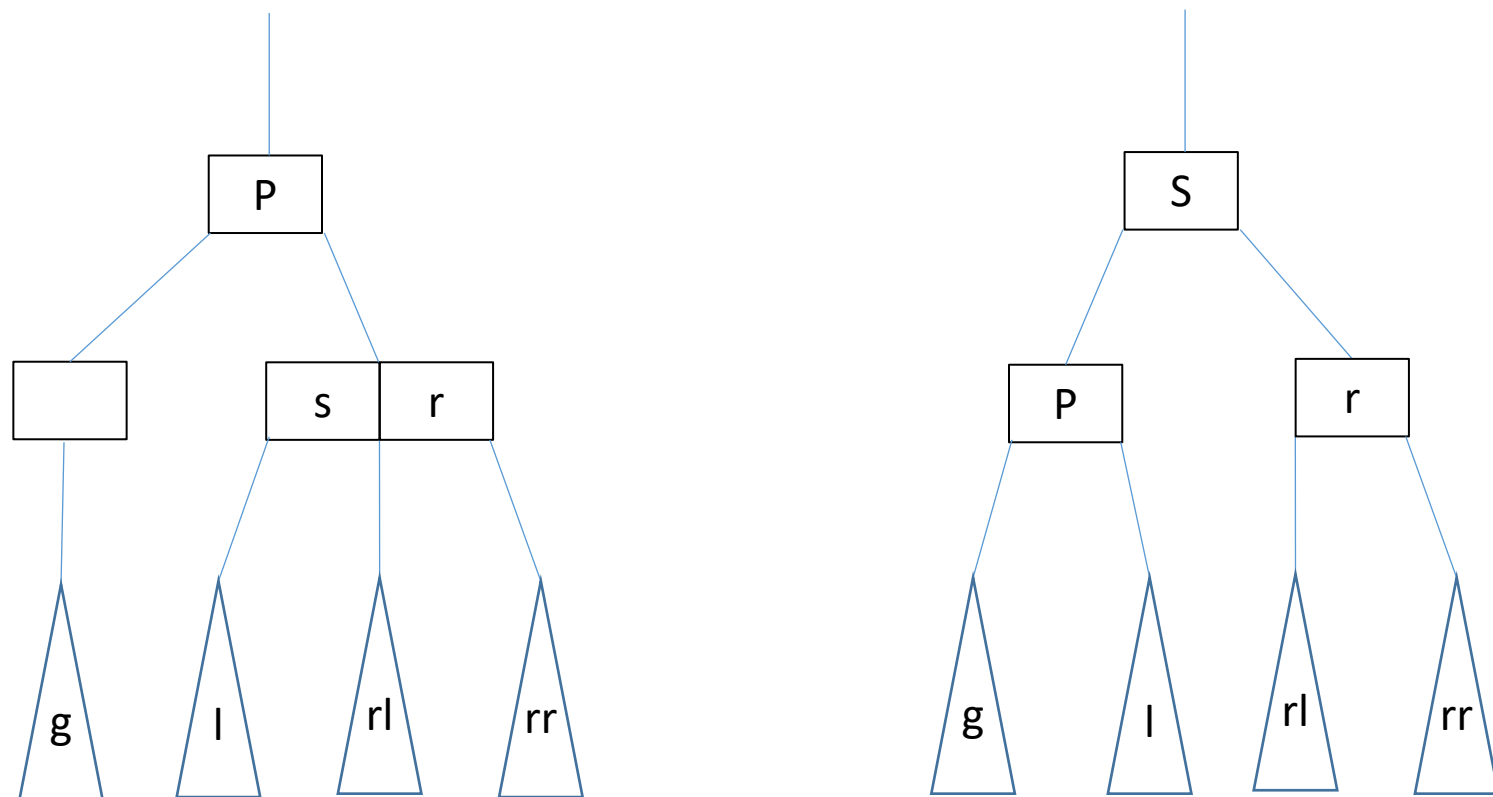
注意：这个和B树的操作不同。B树中，如果s的右子结点为红色，是可以从s所在的B树结点借一个Key的。

# Repair: $p, l$ is Black, $r$ is Red



- 从小到大：  $g, p, l, s, rl, r, rr$
- $g, l, rl, rr$ 的Black Height相同
- 调整后，Black Height和原来的子树的BlackHeight相同

# L, P is Black, R is Red 的B树视图

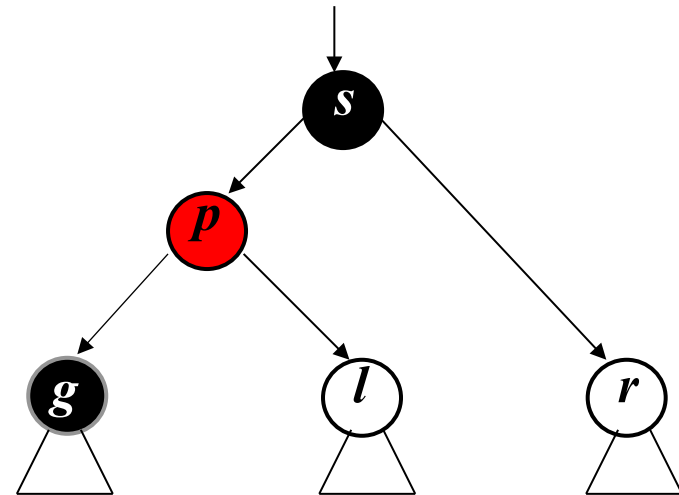
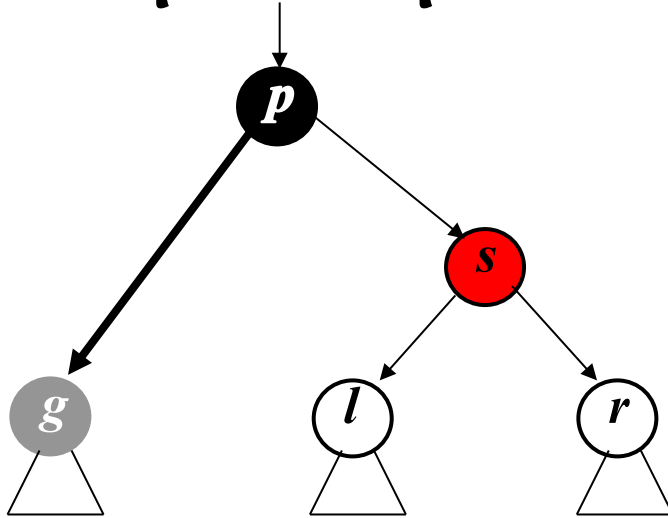


实际上和L is Red的情况一样，就是说兄弟结点有两个Key，所以操作其实是一样的。不过在二叉树结构和颜色上不一样：

- 1、P下移
- 2、l上移
- 3、子树相应地调整

其实即使P是红色，只要r是红色，我们仍然能够进行按照这个方法处理的。

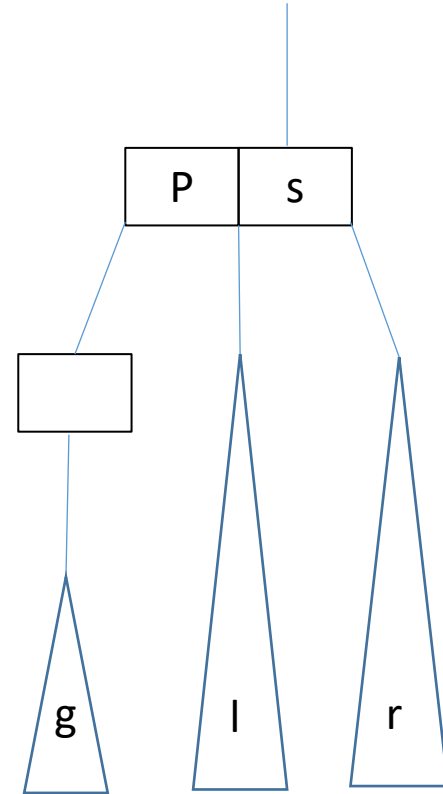
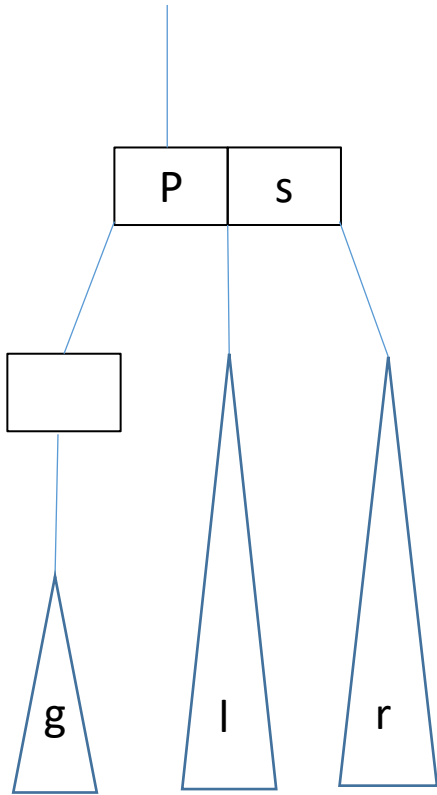
# Repair: $p, l, r$ is Black, $s$ is Red



- 如果  $s$  为红色，则  $p, l, r$  都是黑色
- $g$  的 BlackHeight 比  $l, r$  少 1
- 调整后仍然不平衡，但是可以看作以下情况：
  - 以  $p$  为根， $g$  和  $l$  为子树，且  $g$  因为删除而 Black Height 比  $l$  的小 1
  - 因为  $p$  为红色，只需要考虑前面列举的 2 种情况，经过调整后， $p$  所在子树的 Black Height 是  $l$  的 Black Height

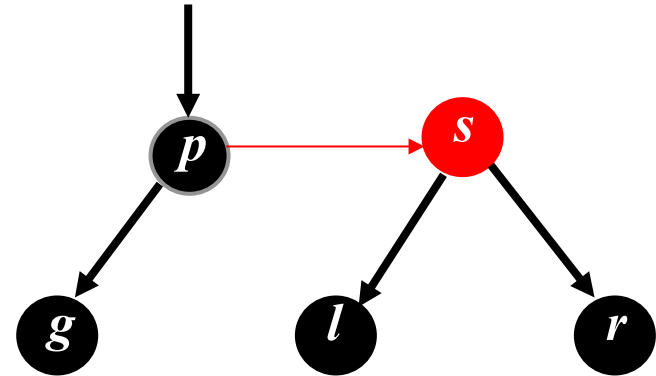
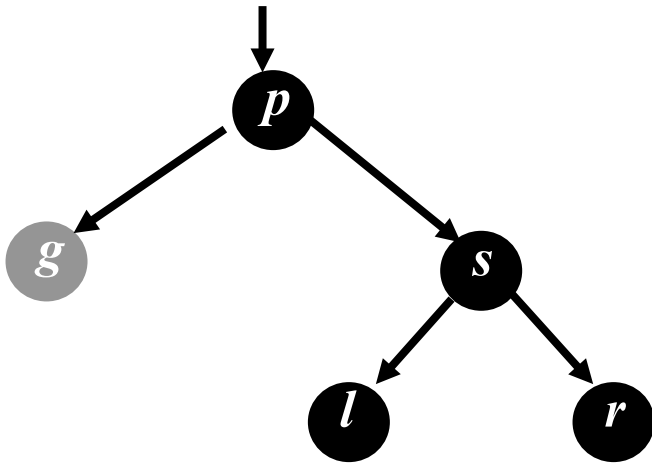


L, P, R is Black, S is Red



这个初步处理，实际上只是在红黑树中改变了一下结构，B树的结构没有改变后面的处理，会根据子树l的根结点的情况进行。

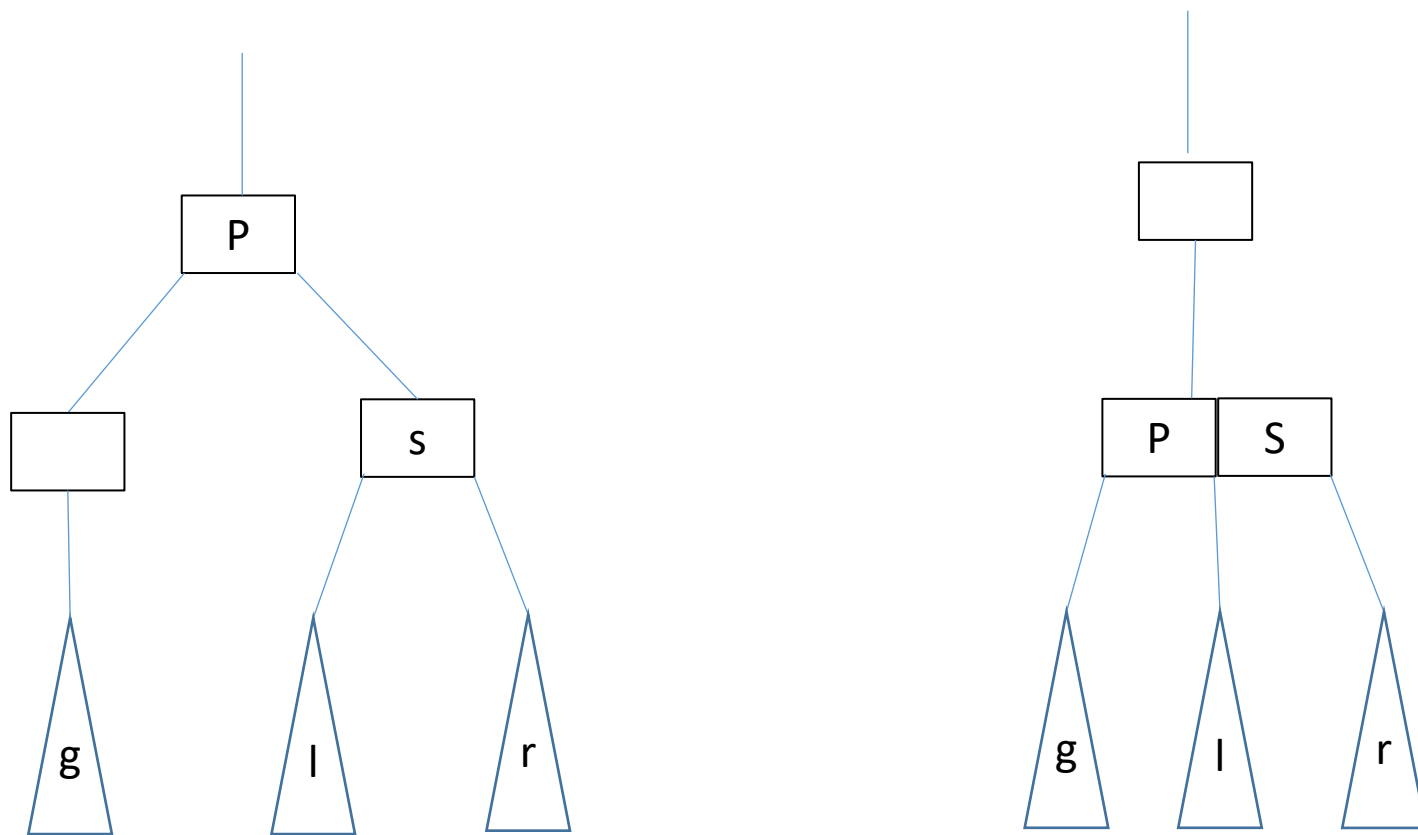
# Repair: All are black



- 1、 $g, l, r$ 的Black Height相等
- 2、调整后 $p$ 所在的子树的Black Height减去1

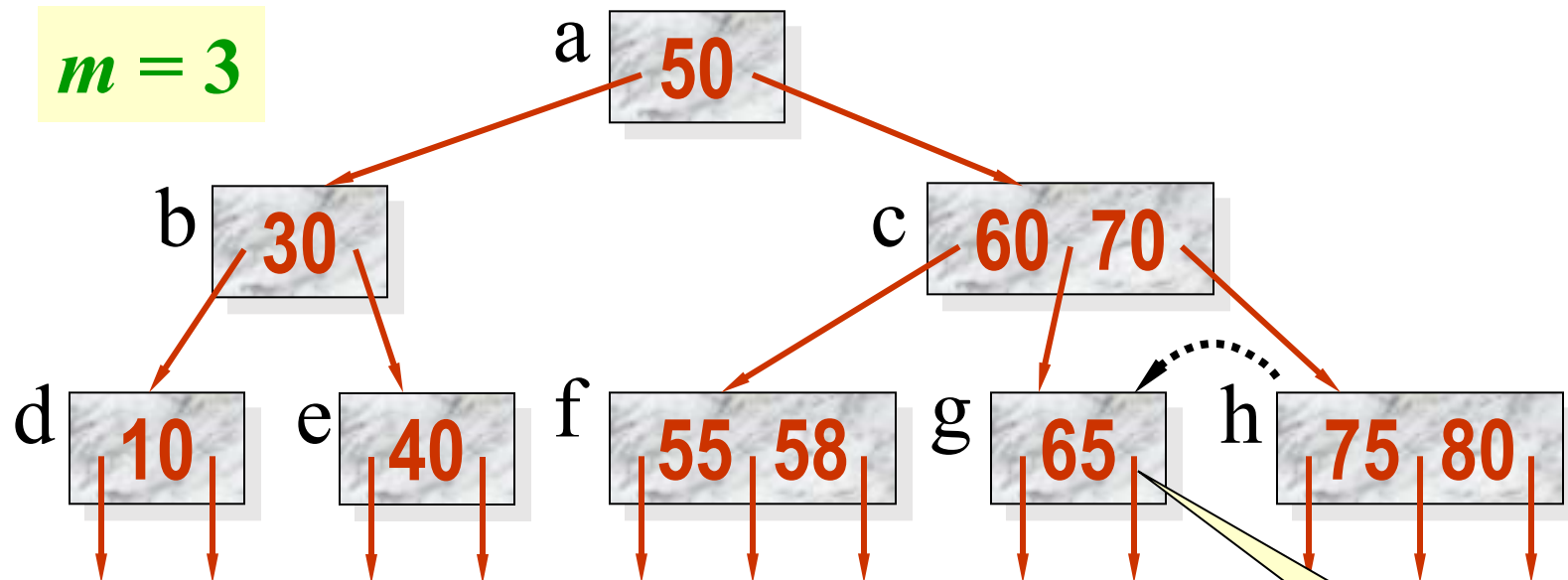
因此可能向上传递！

# L,P,S,R are Black的B树视图

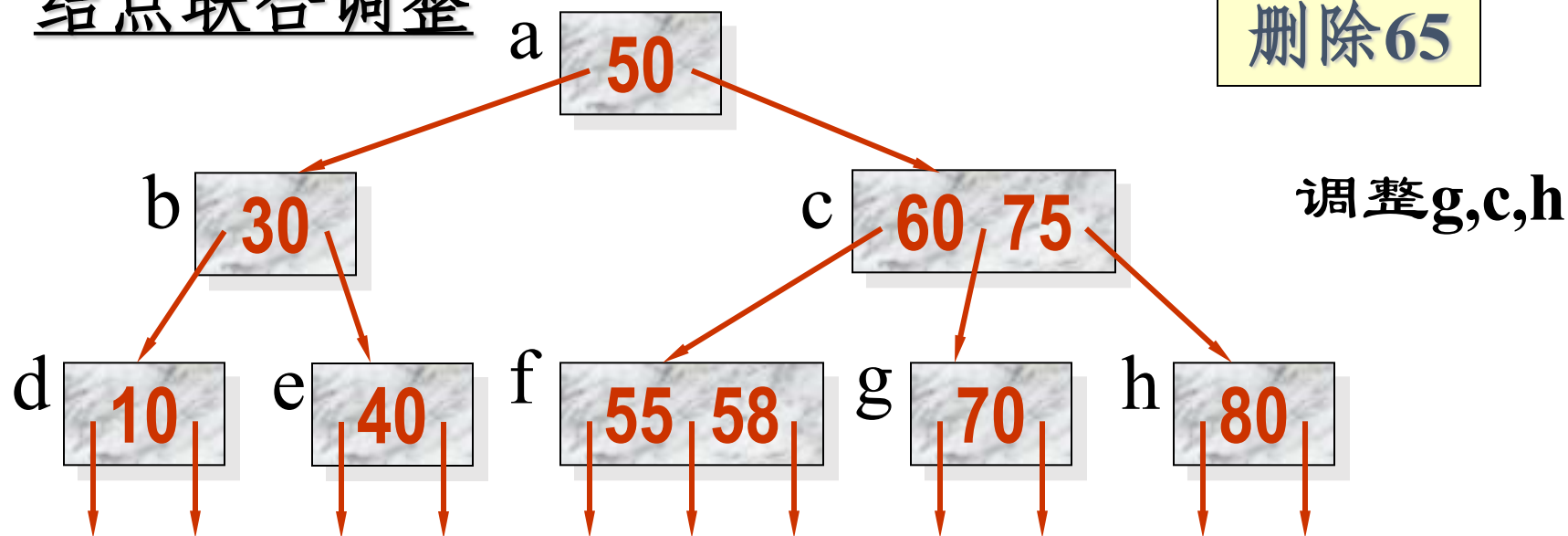


兄弟结点只有一个Key，所以只能够合并；  
同时，父亲结点也只有一个Key，合并后高度降低1.

$m = 3$



结点联合调整



# 直接从B树角度实现红黑树的操作

- 假设当前结点为g，找到它在红黑树中的兄弟结点：  
假设g是父节点的左子结点  
parent = g的父亲结点  
sibling = parent->right;  
if (sibling ->color == RED) {sibling = sibling ->left}  
//sibling是其兄弟结点  
  
if(sibling ->left->color == RED || sibling->right->color == RED)  
... //从兄弟结点借一个key  
else  
... //和兄弟结点合并为一个结点
- 这样记忆起来比较方便，但是可能实现的时候也很麻烦。
- 等我有空实现一下看看。反正我们的编程实习需要有红黑树的编程练习。