第十章 索引与散列

- 静态索引结构
- 动态索引结构
- *Trie*树
- 散列 (Hashing)
- 小结

静态索引结构

当数据对象个数 ***n*** 很大时，如果用无序表形式的静态搜索结构存储，采用顺序搜索，则搜索效率极低。如果采用有序表存储形式的静态搜索结构，则插入新记录进行排序，时间开销也很可观。这时可采用索引方法来实现存储和搜索。

线性索引 (Linear Index List)

- 示例：有一个存放职工信息的数据表，每一个职工对象有近 **1*k*** 字节的信息，正好占据一个页块的存储空间。

索引表

关键码	地址
03	180
08	140
17	340
24	260
47	300
51	380
83	100
95	220

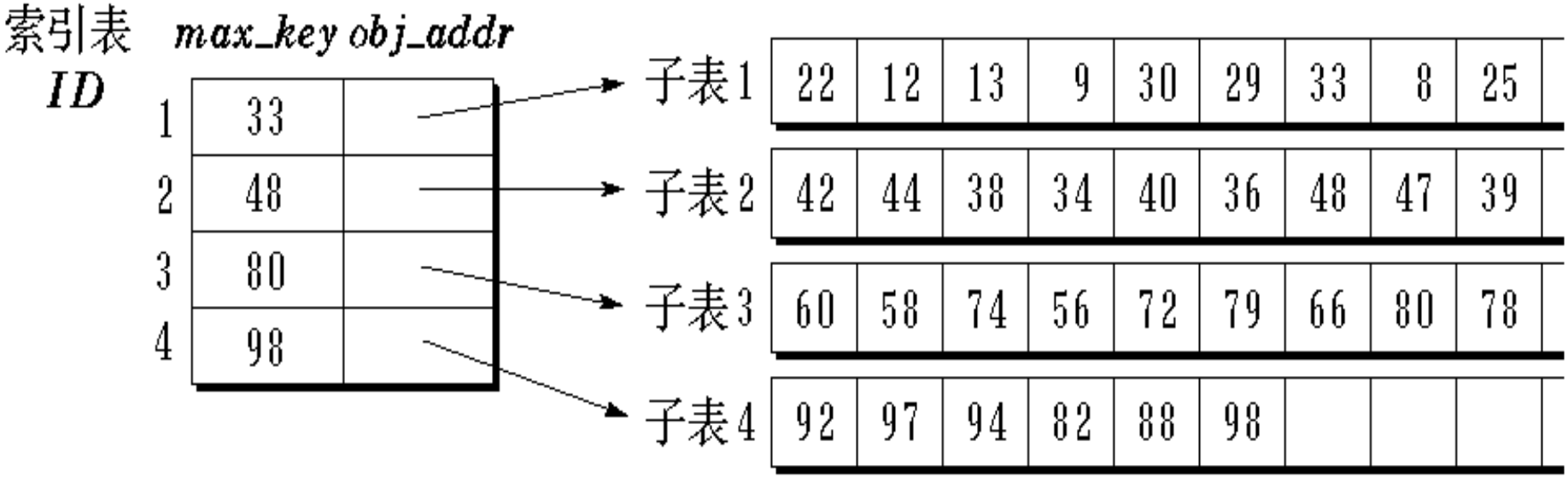
数据表

职工号	姓　名	性别	职　务	婚否	其它
83	张　珊	女	教　师	已婚	……
08	李　斯	男	教　师	已婚	……
03	王　鲁	男	行政助理	已婚	……
95	刘　琪	女	实验员	未婚	……
24	岳　跋	男	教　师	已婚	……
47	周　惠	男	教　师	已婚	……
17	胡　江	男	实验员	未婚	……
51	林　青	女	教　师	未婚	……

- 假设内存工作区仅能容纳 **64k** 字节的数据，在某一时刻内存最多可容纳 **64** 个对象以供搜索。
- 如果对象总数有 **14400** 个，不可能把所有对象的数据一次都读入内存。无论是顺序搜索或对分搜索，都需要多次读取外存记录。
- 如果在索引表中每一个索引项占**4**个字节，每个索引项索引一个职工对象，则 **14400** 个索引项需要 **56.25k** 字节，在内存中可以容纳所有的索引项。
- 这样只需从外存中把索引表读入内存，经过搜索索引后确定了职工对象的存储地址，再经过 **1** 次读取对象操作就可以完成搜索。

- 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)。
- 在子表中，所有对象可能按关键码有序地存放，也可能无序地存放。但所有这些子表必须分块有序，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。另外建立一个索引表。

- 索引表中每一表目叫做索引项，它记录了子表中最大关键码***max_key***以及该子表在数据区中的起始位置***obj_addr***。



- 各个索引项在索引表中的序号与各个子表的块号有一一对应的关系：第 ***i*** 个索引项是第 ***i*** 个子表的索引项，***i*** = **0, 1, …, *n*-1**。这样的索引结构叫做索引顺序结构。

- 对索引顺序结构进行搜索时，一般分为两级搜索。
  - 先在索引表 *ID* 中搜索给定值 *K*，确定满足

    $$ID[i-1].max\_key < K \le ID[i].max\_key$$

    的 *i* 值，即待查对象可能在的子表的序号。
  - 然后再在第 *i* 个子表中按给定值搜索要求的对象。
- 索引表是按***max_key***有序的，且长度也不大，可以对分搜索，也可以顺序搜索。
- 各子表内各个对象如果也按对象关键码有序，可以采用对分搜索或顺序搜索；如果不是按对象关键码有序，只能顺序搜索。

- 索引顺序搜索的搜索成功时的平均搜索长度

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

- 其中，ASL_{Index} 是在索引表中搜索子表位置的平均搜索长度，$ASL_{SubList}$ 是在子表内搜索对象位置的搜索成功的平均搜索长度。
- 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 $b = \lceil n/s \rceil$。又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

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

- 索引顺序搜索的平均搜索长度不仅与表中的对象个数 n 有关，而且与每个子表中的对象个数 s 有关。在给定 n 的情况下，s 应选择多大?
- 利用数学方法可以导出，当 $s = \sqrt{n}$ 时，$ASL_{IndexSeq}$ 取极小值 $\sqrt{n}+1$。 这个值比顺序搜索强，但比对分搜索差。但如果子表存放在外存时，还要受到页块大小的制约。
- 若采用对分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

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

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

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

倒排表 (Inverted Index List)

- 对包含有大量数据对象的数据表或文件进行搜索时，最常用的是针对对象的主关键码建立索引。主关键码可以唯一地标识该对象。用主关键码建立的索引叫做主索引。
- 主索引的每个索引项给出对象的关键码和对象在表或文件中的存放地址。

对象关键码 *key*	对象存放地址 *addr*

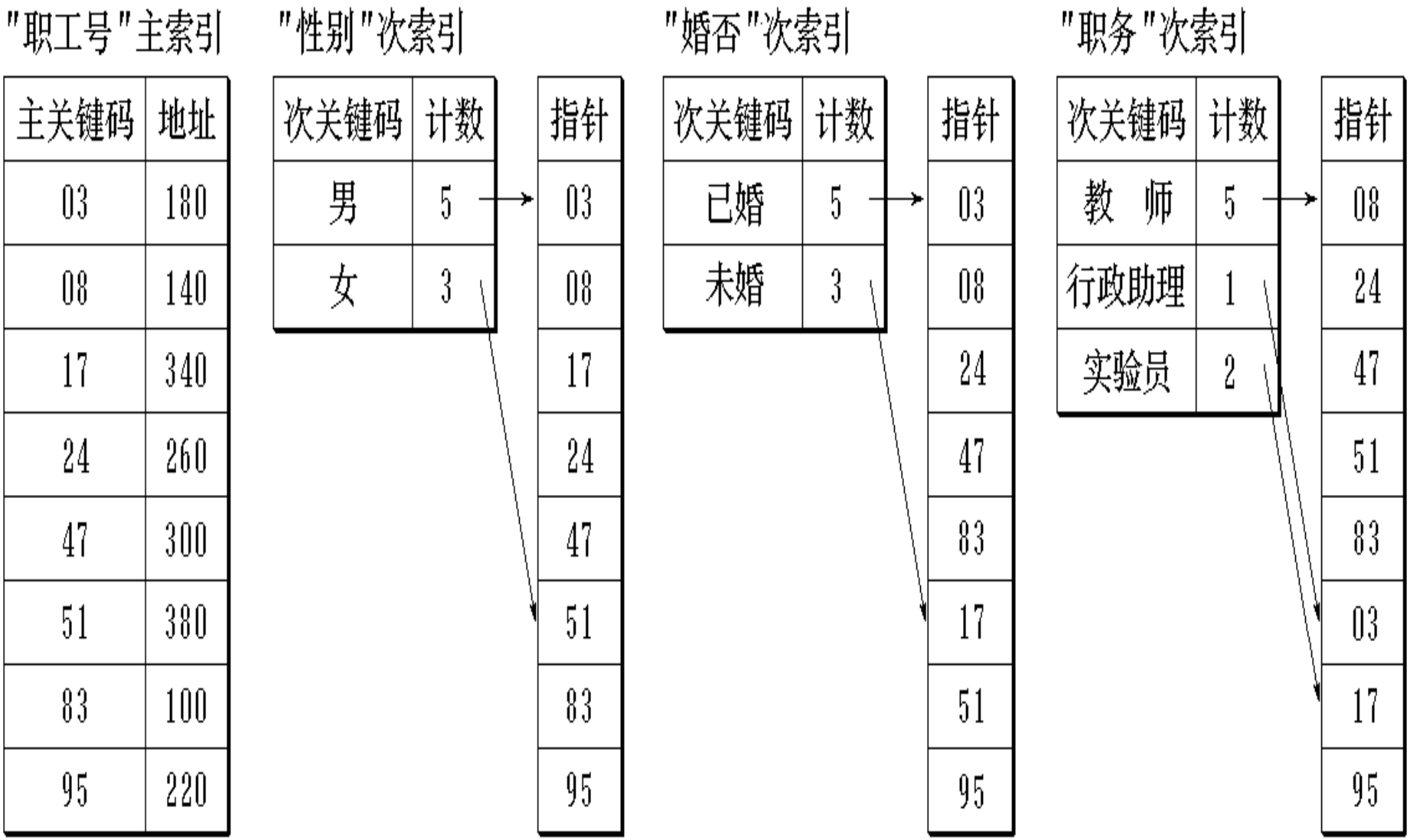
- 但在实际应用中有时需要针对其它属性进行搜索。例如，查询如下的职工信息:

  (1) 列出所有教师的名单;

  (2) 已婚的女性职工有哪些人?

- 这些信息在数据表或文件中都存在，但都不是关键码，为回答以上问题，只能到表或文件中去顺序搜索，搜索效率极低。
- 因此，除主关键码外，可以把一些经常搜索的属性设定为次关键码，并针对每一个作为次关键码的属性，建立次索引。
- 在次索引中，列出该属性的所有取值，并对每一个取值建立有序链表，把所有具有相同属性值的对象按存放地址递增的顺序或按主关键码递增的顺序链接在一起。
- 次索引的索引项由次关键码、链表长度和链表本身等三部分组成。

- 为了回答上述的查询，我们可以分别建立“性别”、“婚否”和“职务”次索引。



(1) 列出所有教师的名单;

(2) 已婚的女性职工有哪些人?

- 通过顺序访问“职务”次索引中的“教师”链，可以回答上面的查询**(1)**。
- 通过对“性别”和“婚否”次索引中的“女性”链和“已婚”链进行求“交”运算，就能够找到所有既是女性又已婚的职工对象，从而回答上面的查询**(2)**。

	1	0	1	1	1	……	1	0	1	1
	1	0	0	1	0	……	0	1	0	0
AND	1	1	0	0	1	……	0	1	0	1
	1	0	0	0	0	……	0	0	0	0

- 倒排表或倒排文件是一种次索引的实现。在倒排表中所有次关键码的链都保存在次索引中，仅通过搜索次索引就能找到所有具有相同属性值的对象。
- 在次索引中记录对象存放位置的指针可以用主关键码表示，可以通过搜索次索引确定该对象的主关键码，再通过搜索主索引确定对象的存放地址。
- 在倒排表中各个属性链表的长度大小不一，管理起来比较困难。为此引入单元式倒排表。

- 在单元式倒排表中，索引项中不存放对象的存储地址，存放该对象所在硬件区域的标识。
- 硬件区域可以是磁盘柱面、磁道或一个页块，以一次 I / O 操作能存取的存储空间作为硬件区域为最好。为使索引空间最小，在索引中标识这个硬件区域时可以使用一个能转换成地址的二进制数，整个次索引形成一个(二进制数的) 位矩阵。
- 例如，对于记录学生信息的文件，次索引可以是如图所示的结构。二进位的值为1的硬件区域包含具有该次关键码的对象。

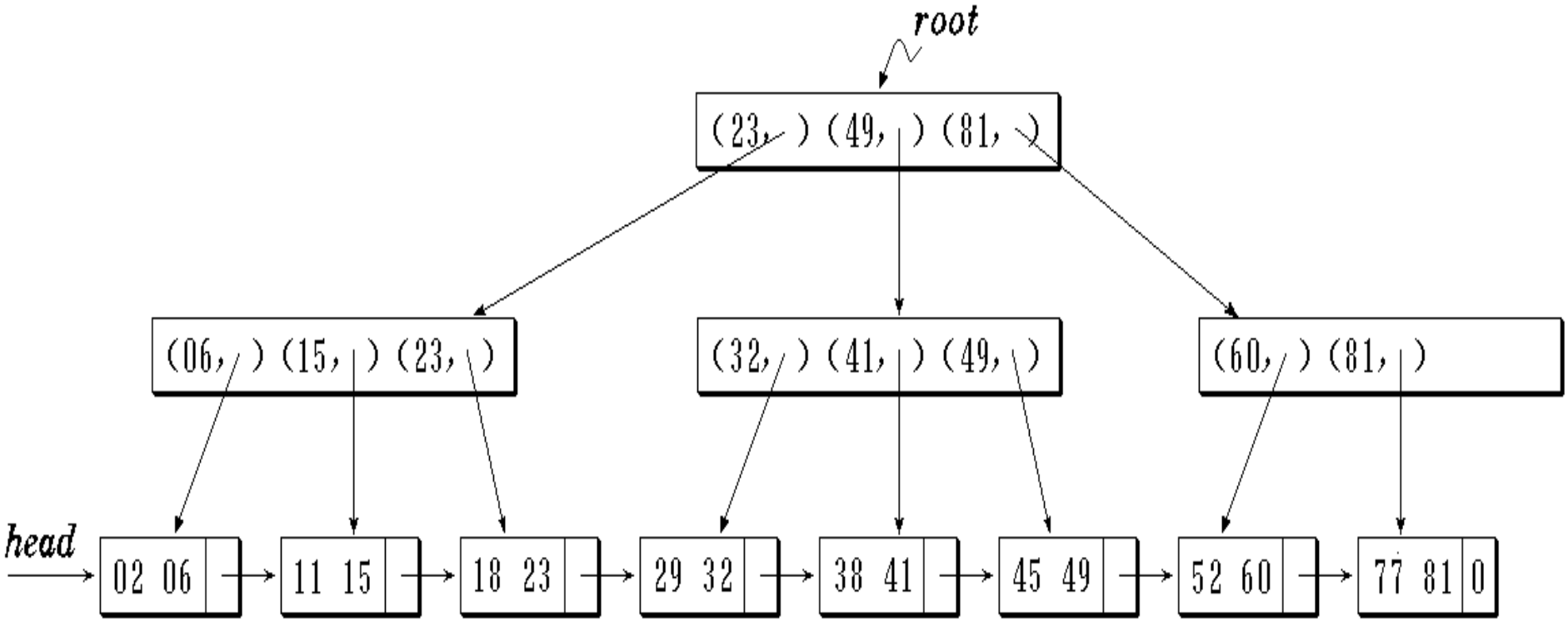
		硬 件 区 域										
		1	2	3	4	5	…	251	252	253	254	…
次关键码 1 (性别)	男	1	0	1	1	1	…	1	0	1	1	…
	女	1	1	1	1	1	…	0	1	1	0	…
次关键码 2 (籍贯)	广东	1	0	0	1	0	…	0	1	0	0	…
	北京	1	1	1	1	1	…	0	0	1	1	…
	上海	0	0	1	1	1	…	1	1	0	0	…
	……											
次关键码 3 (专业)	建筑	1	1	0	0	1	…	0	1	0	1	…
	计算机	0	0	1	1	1	…	0	0	1	1	…
	电机	1	0	1	1	0	…	1	0	1	0	…
	……											

单元式倒排表结构

- 如果在硬件区域1，……中有要求的对象。然后将硬件区域1，……等读入内存，在其中进行检索，确定就可取出所需对象。

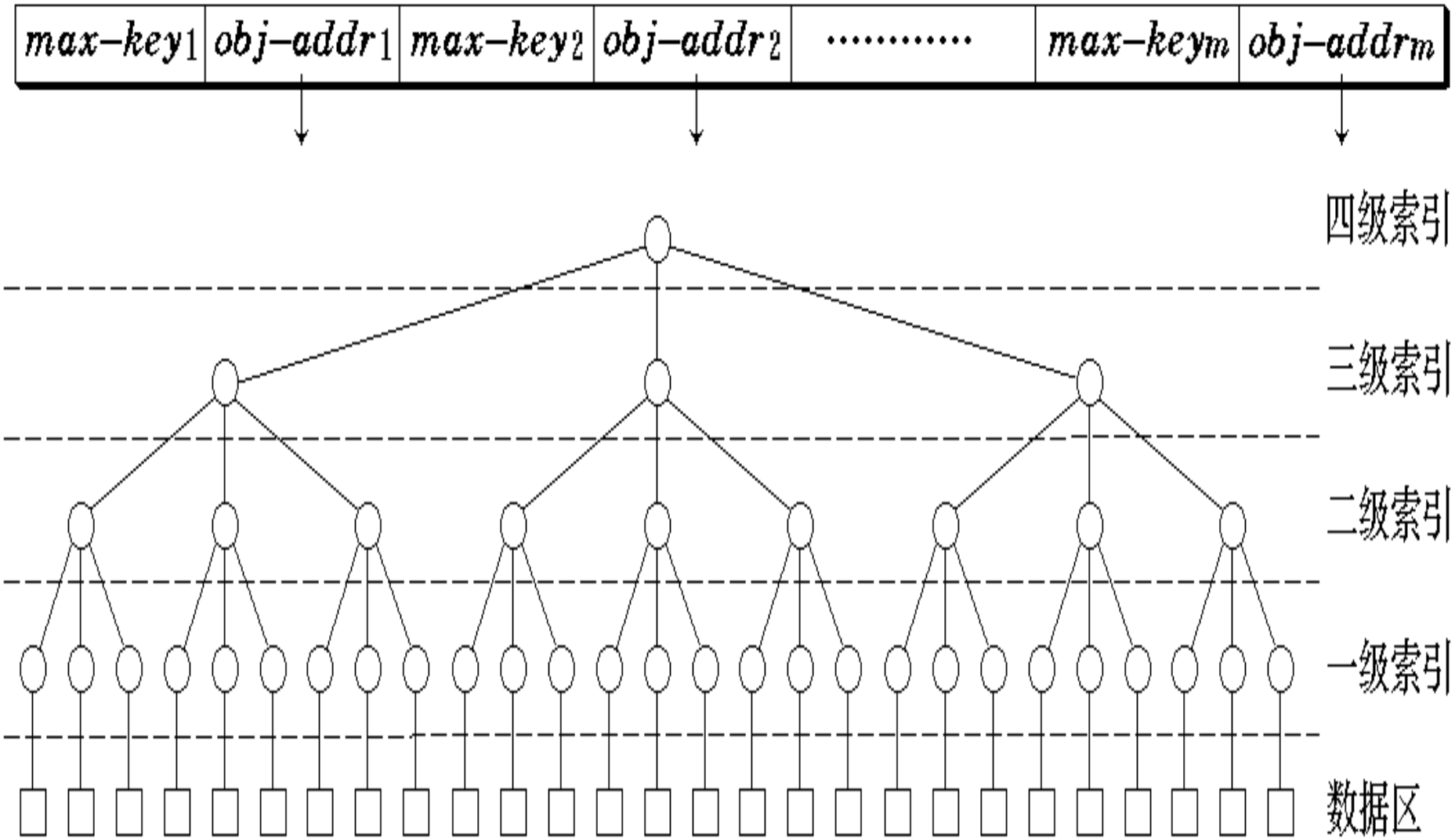
m 路静态搜索树

- 当数据对象数目特别大，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。
- 在此情况下，可以建立索引的索引，称为二级索引。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，叫做三级索引。这时，访问外存次数等于读入索引次数再加上1次读取对象。
- 必要的话，还可以有4级索引，5极索引，….

- 这种多级索引结构形成一种 ***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对象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

动态搜索结构

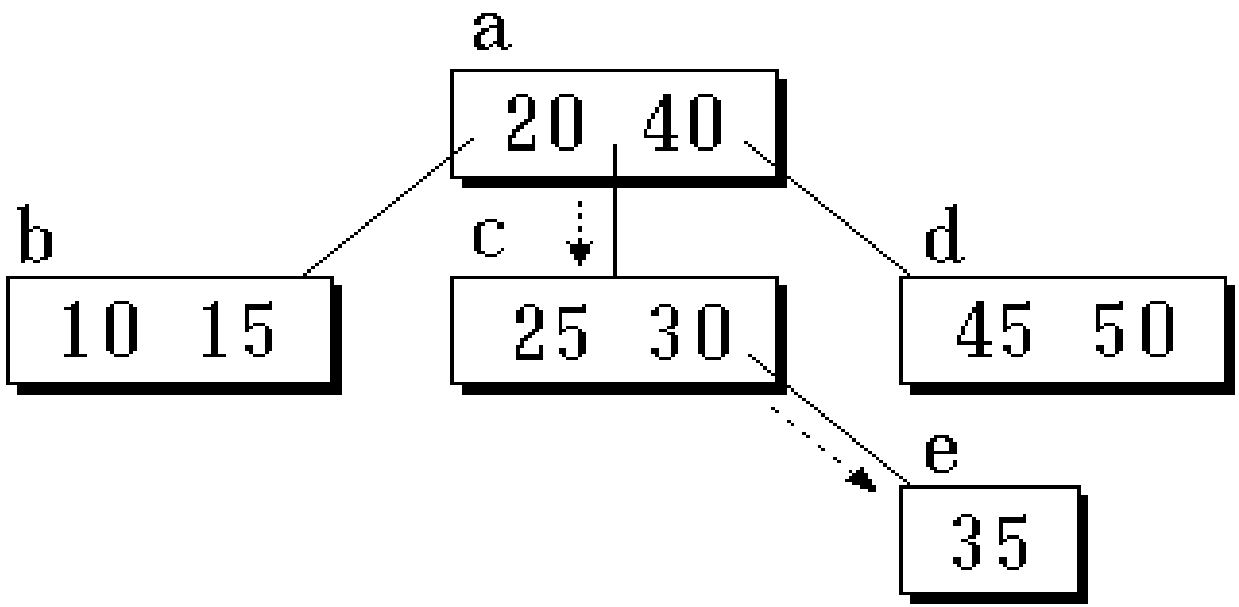
动态的 m 路搜索树

- 现在我们所讨论的 m 路搜索树多为可以动态调整的多路搜索树，它的一般定义为：
- 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：

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

其中，P_i 是指向子树的指针，$0 \le i \le n < m$；K_i 是关键码，$1 \le i \le n < m$。 $K_i < K_{i+1}$，$1 \le i < n$。

- 在子树 P_i 中所有的关键码都小于 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，$0 \le i \le n$。



一棵3路搜索树的示例

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)

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

- 每个结点中最多有 $m-1$ 个关键码，在一棵高度为 h 的 m 路搜索树中关键码的最大个数为 $m^{h+1}-1$。
  - 对于高度 $h=2$ 的二叉树，关键码最大个数为 7;
  - 对于高度 $h=3$ 的3路搜索树，关键码最大个数为 $3^4-1 = 80$。

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

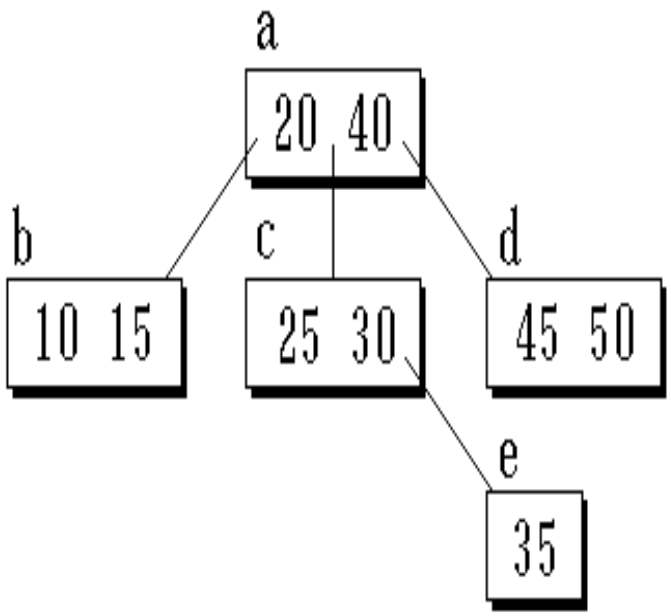
```
template <class Type> Triple<Type> &
Mtree<Type> :: Search ( const Type & x ) {
    Triple result;                          //记录搜索结果三元组
    GetNode ( root );                       //读入根结点
    Mnode <Type> *p = root, *q = NULL;
    while ( p != NULL ) {                   //从根开始检测
        int i = 0;  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
}
```

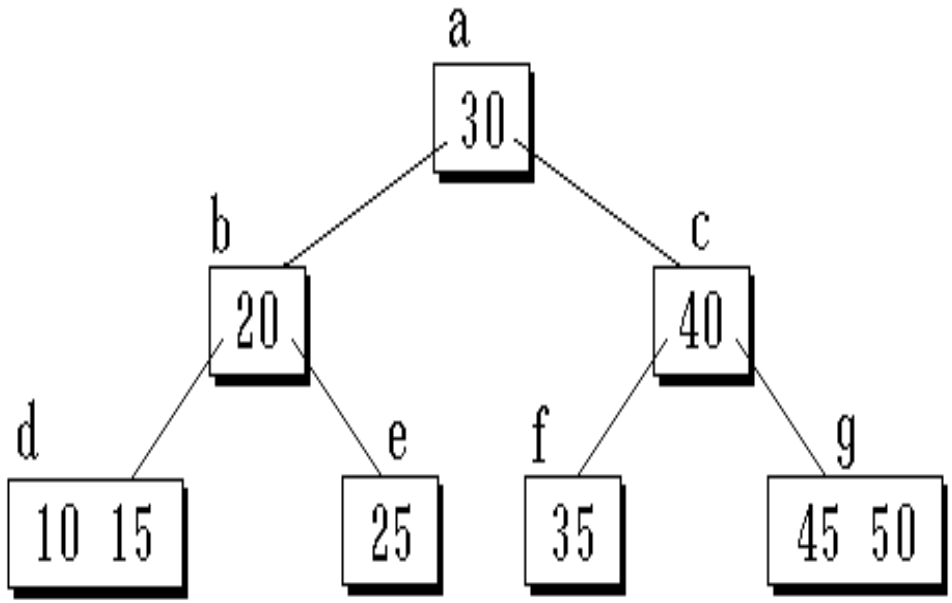
- 提高搜索树的路数 ***m***，可以改善树的搜索性能。对于给定的关键码数 ***n***，如果搜索树是平衡的，可以使 ***m*** 路搜索树的性能接近最佳。下面我们将讨论一种称之为**B_**树的平衡的 ***m*** 路搜索树。在**B_**树中我们引入“失败”结点。一个失败结点是当搜索值***x***不在树中时才能到达的结点。

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。
- 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$D[i]$ $(1 \le i \le n < m)$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$D[i]$。



非B_树



B_树

B_树的类定义和B_树结点类定义

```
template <class Type> class Btree :
        public Mtree<Type> {
//继承 m 路搜索树的所有属性和操作
```

```
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_树的搜索算法继承了在*m*路搜索树*Mtree*上的搜索算法。
- B_树的搜索过程是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。因此，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。如果我们定义B_树的高度*h*为失败结点所在的层次，需要了解树的高度*h* 与树中的关键码个数 *N* 之间的关系。

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？从B_树的定义知，
  - 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 $N+1$。这是因为失败一般发生在 $K_i < x < K_{i+1}$，$0 \le i \le N$，设 $K_0 = -\infty$，$K_{N+1} = +\infty$。因此，有

$$N+1 = \text{失败结点数} = \text{位于第 } h \text{ 层的结点数} \ge 2\lceil m/2 \rceil^{h-1}$$

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

- 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

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

- 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

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

- 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

- 如果提高B_树的阶数 m，可以减少树的高度，从而减少读入结点的次数，因而可减少读磁盘的次数。
- 事实上，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：应使得在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：
  - ☆ 从磁盘中读入结点所用时间
  - 在结点中搜索 *x* 所用时间
- 根据定义，B_树的每个结点的大小都是固定的，结点内有 $m-1$ 个索引项 $(K_i, D_i, P_i), 1 \le i < m$。其中，$K_i$ 所占字节数为α，D_i 和 P_i 所占字节数为β，则结点大小近似为 $m(\alpha+2\beta)$ 个字节。读入一个结点所用时间为：

$$t_{seek} + t_{latency} + m(\alpha + 2\beta)\, t_{tran} = a + bm$$

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。
- 实现结点“分裂”的原则是：
  - 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

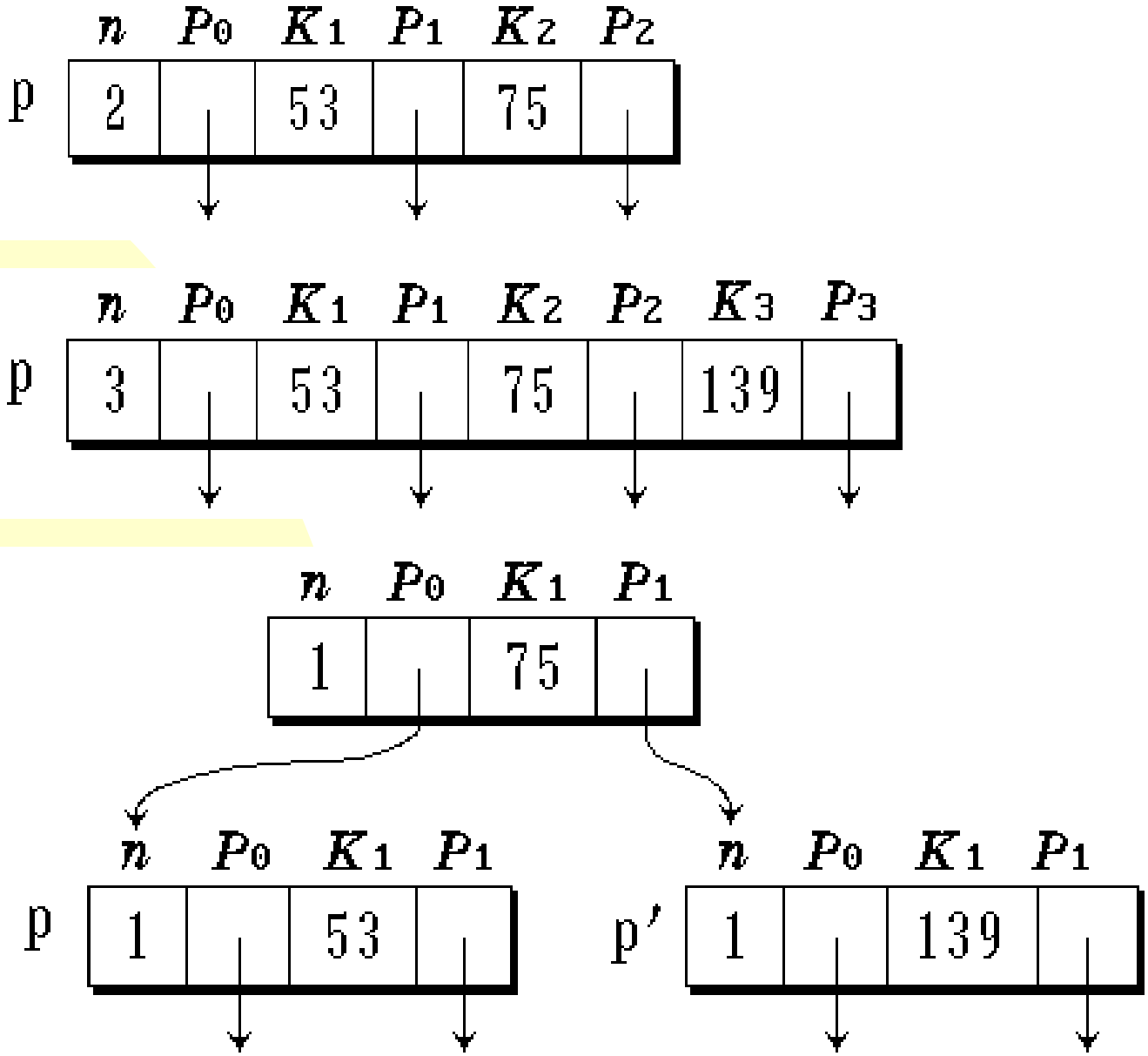
$(m, P_0, K_1, P_1, K_2, P_2, \cdots\cdots, K_m, P_m)$

其中 $K_i < K_{i+1}, 1 \le i < m$

- 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：

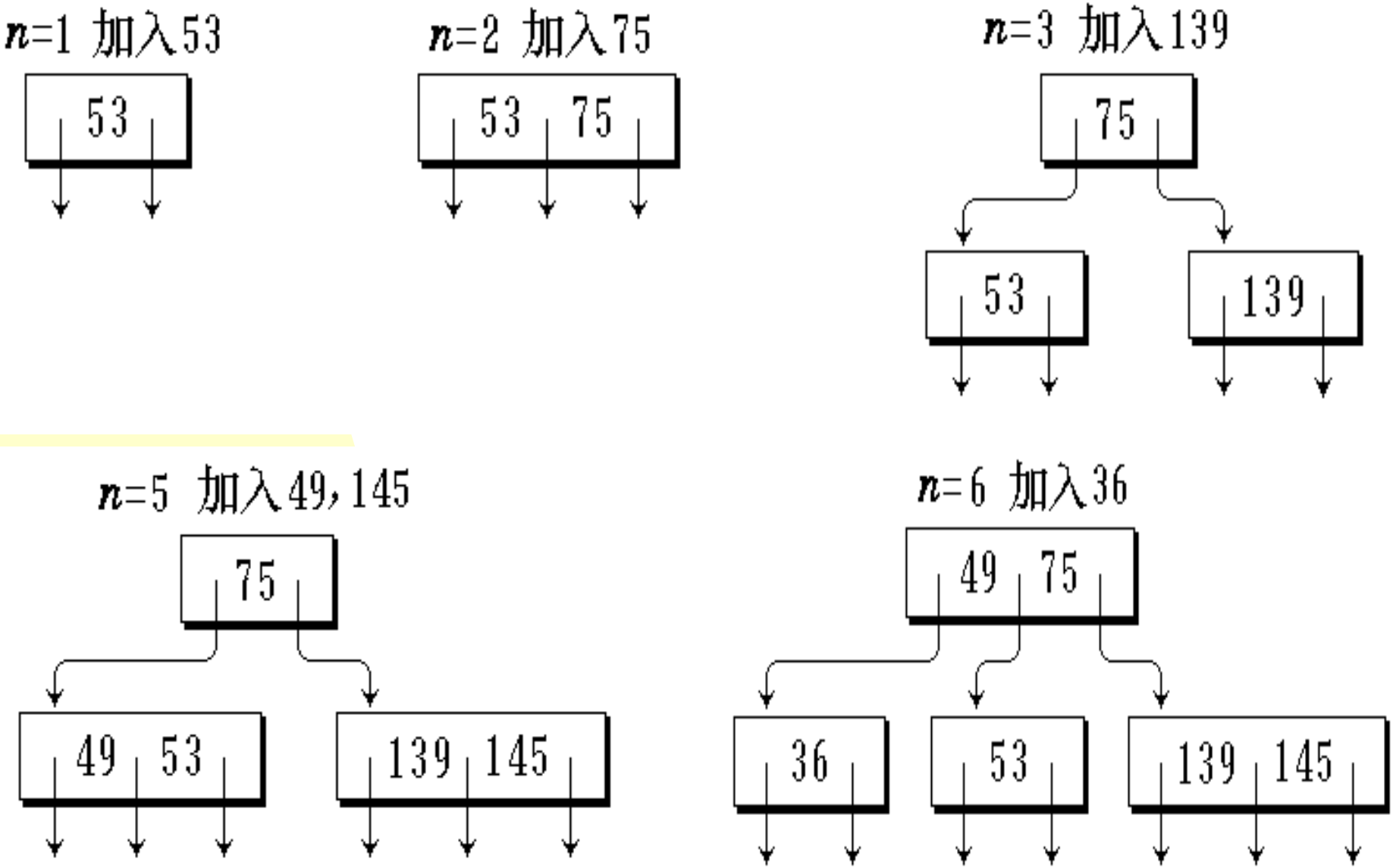
  $(\lceil m/2 \rceil - 1, P_0, K_1, P_1, \cdots\cdots, K_{\lceil m/2 \rceil - 1}, P_{\lceil m/2 \rceil - 1})$
- 结点 q：

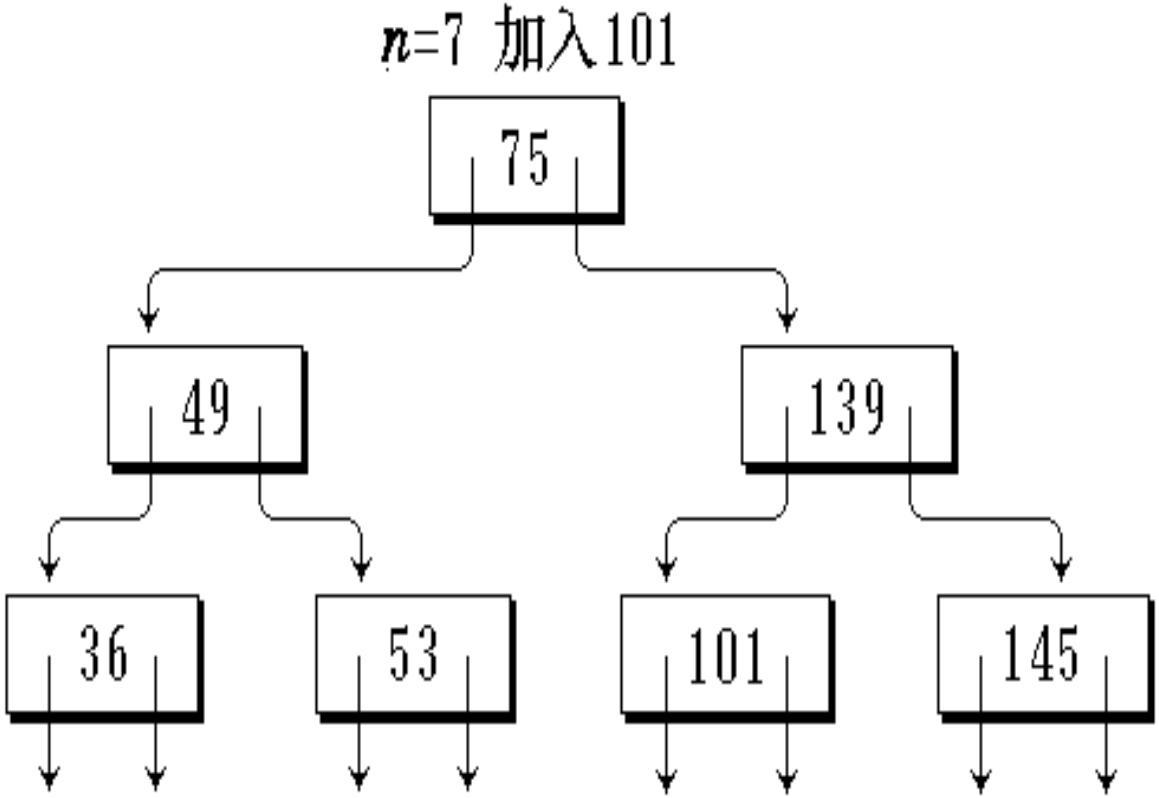
  $(m - \lceil m/2 \rceil, P_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, P_{\lceil m/2 \rceil + 1}, \cdots\cdots, K_m, P_m)$
- 位于中间的关键码 $K_{\lceil m/2 \rceil}$ 与指向新结点 q 的指针形成一个二元组 $(K_{\lceil m/2 \rceil}, q)$，插入到这两个结点的双亲结点中去。


n P0 K1 P1 K2 P2
p
2
53
75
n P0 K1 P1 K2 P2 K3 P3
p
3
53
75
139
n P0 K1 P1
1
75
n P0 K1 P1
p
1
53
n P0 K1 P1
p′
1
139

结点“分裂”的示例

示例：从空树开始逐个加入关键码建立3阶B_树





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_树的插入算法

```
template <class Type>
int Btree<Type> :: Insert ( const Type & x ) {
    Triple<Type> loc = Search (x);   //找x的位置
    if ( !loc.tag ) return 0;                //找到,不再插入
    Mnode<Type> *p = loc.r,  *q;    //未找到,插入
    Mnode<Type> *ap = NULL,  *t;
    Type K = x;
    int j = loc.i;
    while (1) {
        if ( p→n < m−1) {                    //当前结点未满
            insertkey ( p, j, K, ap );      //(K,ap)插入j后
            PutNode (p);  return 1;     //写出
```

```
}                                          //结点已满,分裂
int s = (m+1)/2;                           //求⌈m/2⌉
insertkey ( p, j, K, ap );                 //(K,ap)插入 j后
q = new Mnode<Type>;                       //建立新结点
move ( p, q, s, m );                       //从 p向q 搬送
K = p→key[s];   ap = q;                    //分界关键码上移
if ( p→parent != NULL ) {                  //双亲结点不是根
    t = p→parent;  GetNode (t);            //读入双亲
    j = 0;  t→key[(t→n)+1] = MAXKEY;
    while ( t→key[j+1] < K ) j++;          //找插入点
    q→parent = p→parent;
    PutNode (p);  PutNode (q);
    p = t;
}
```

```
        else {                                  //双亲是根
            root = new Mnode<Type>;     //创建新根
            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 1;
        }
    }
}
```

- 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。
- 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插入后向上分裂时不必再从磁盘读入，那么，在完成一次插入操作时

需要读写磁盘的次数 =

= 找插入结点向下读盘次数 +

+ 分裂非根结点时写盘次数 +

+ 分裂根结点时写盘次数 =

$= h+2(h-1)+3 =$

$= 3h+1$。

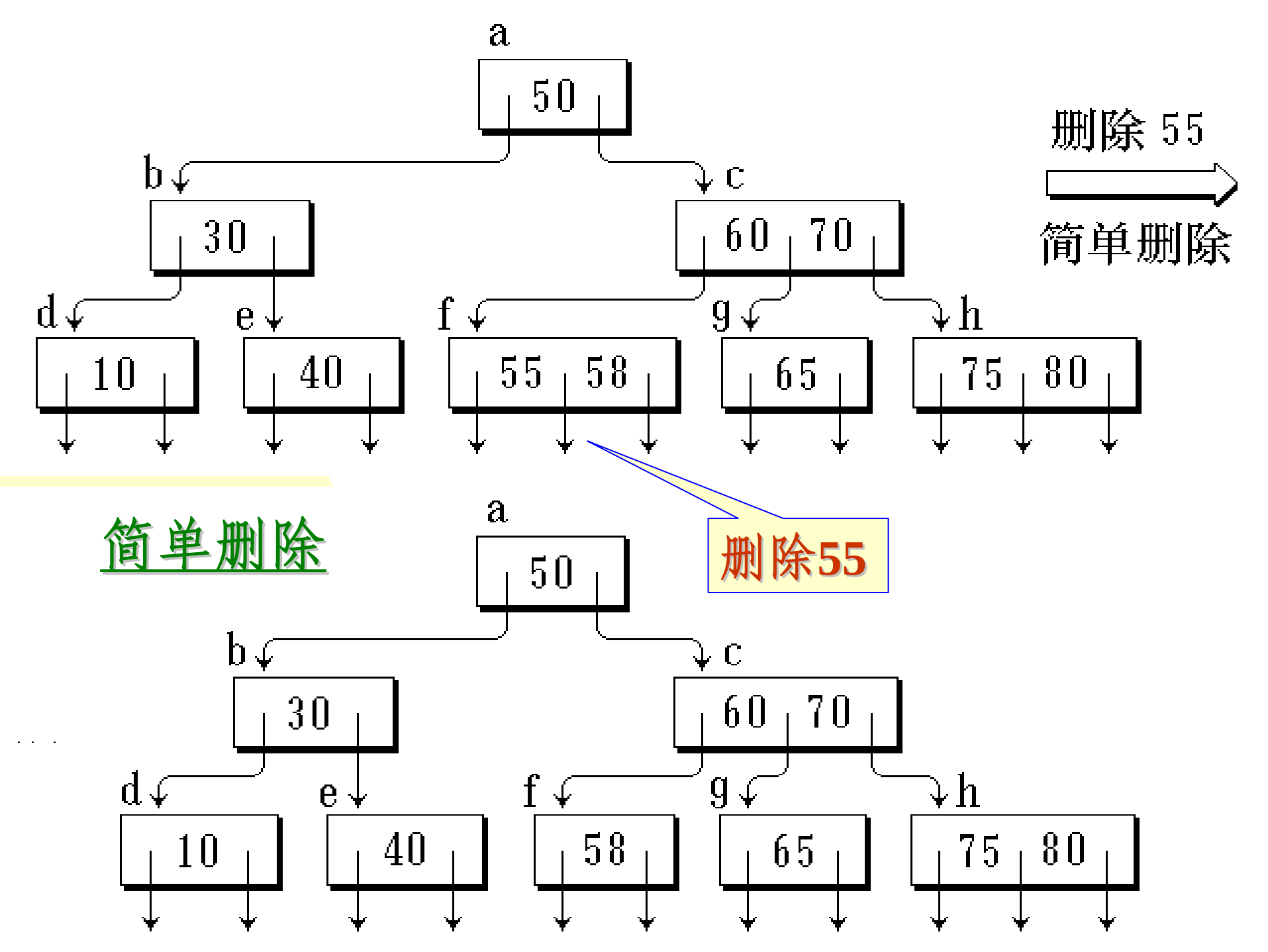
- 可以证明，在向一棵初始为空的B_树中插入 N 个关键码，并且非失败结点个数为 p 时，分裂的结点总数最多为 $p-2$。由于根结点至少有一个关键码，其它各非失败结点至少有 $\lceil m/2 \rceil - 1$ 个关键码，则一棵拥有 p 个结点的 m 阶B_树至少有 $1+(\lceil m/2 \rceil - 1)(p-1)$ 个关键码。
- 其平均分裂结点数 S_{avg} 为：

$$S_{avg} = \text{分裂结点总次数}/N$$
$$\leq (p-2)/(1+(\lceil m/2 \rceil - 1)(p-1))$$
$$< 1/(\lceil m/2 \rceil - 1)$$

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，$1 \le i \le n$，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。

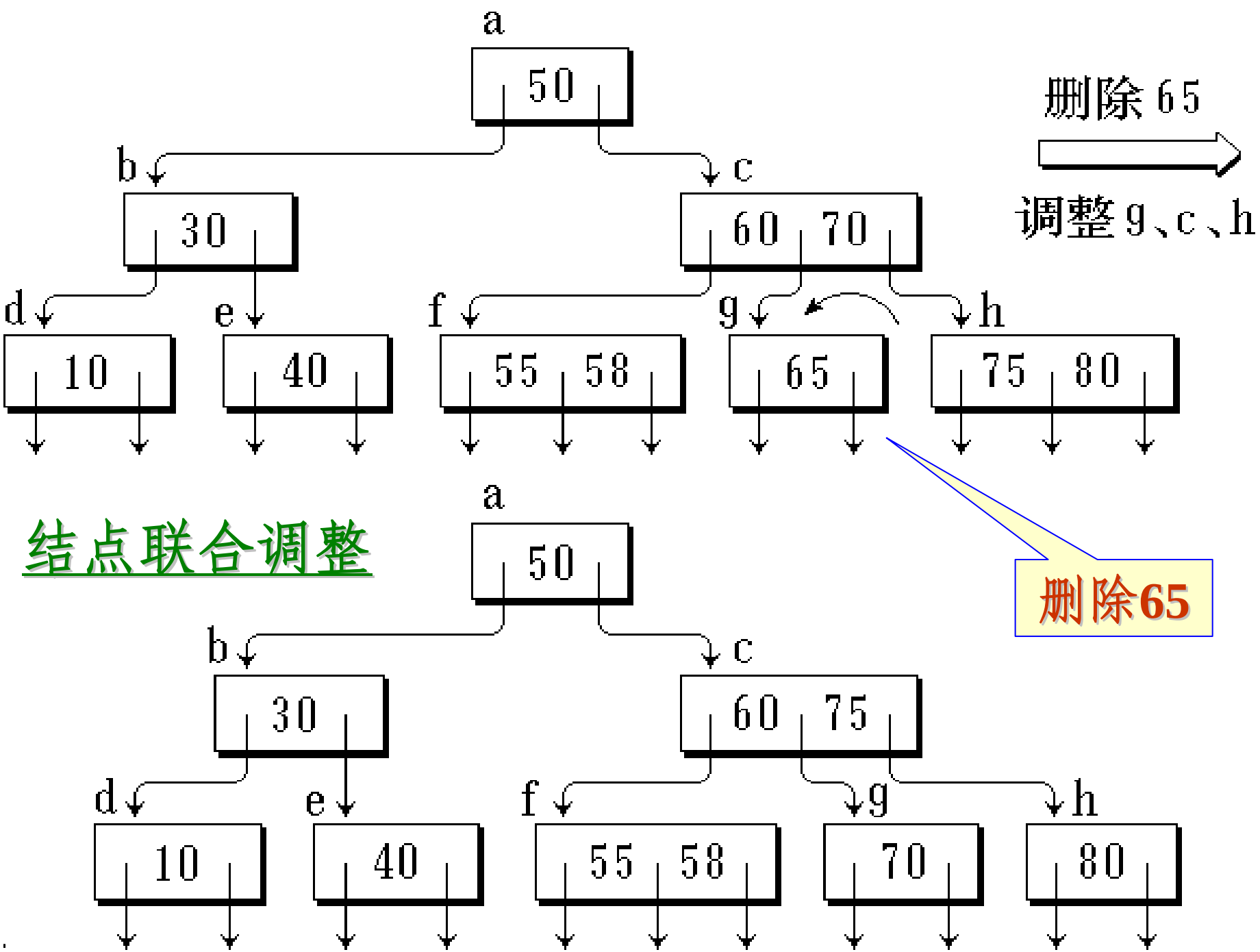
☆ 被删关键码所在叶结点同时又是根结点且删除前该结点中关键码个数 $n \ge 2$，则直接删去该关键码并将修改后的结点写回磁盘。

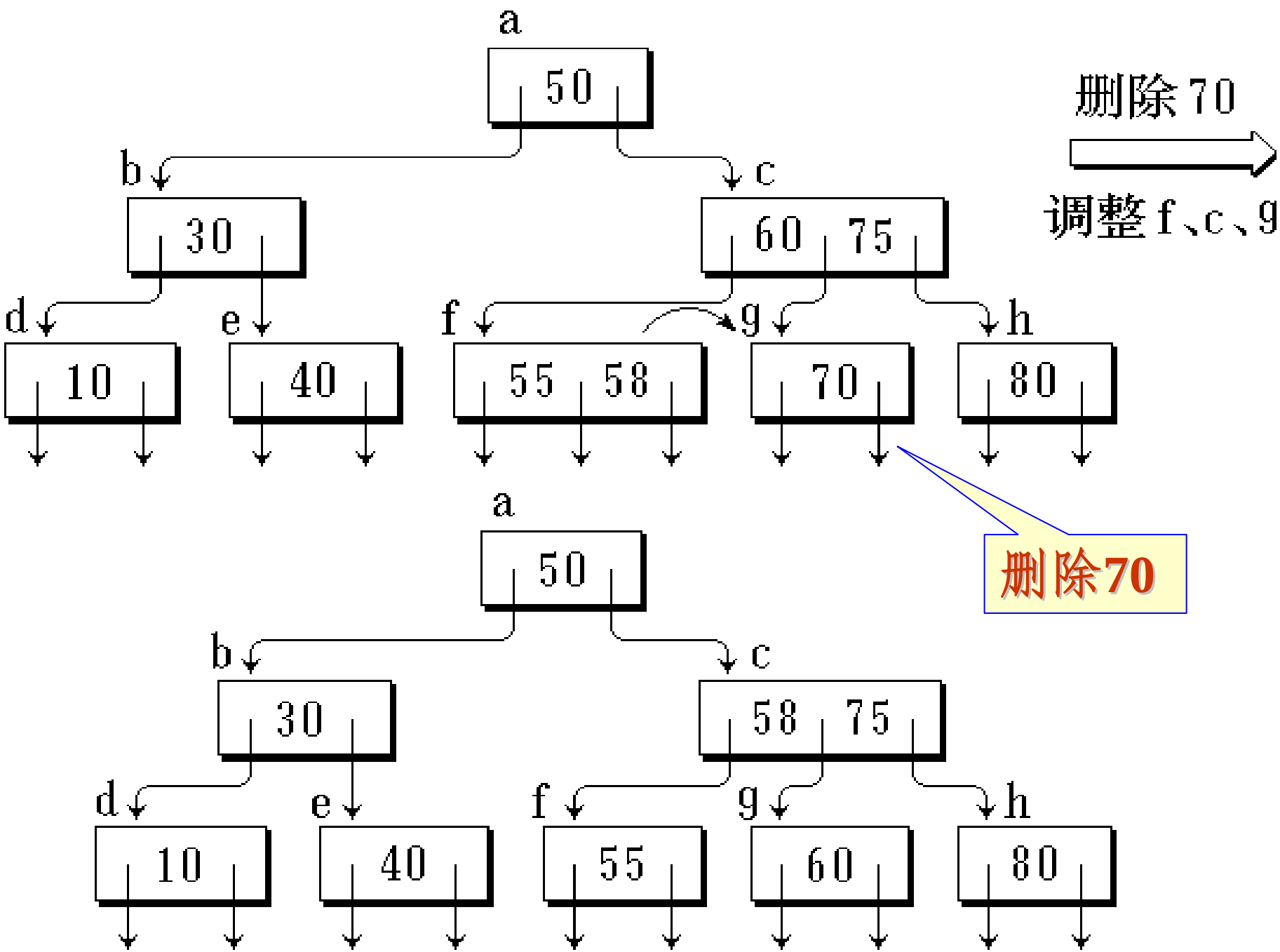

a
50
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
删除 55
简单删除
简单删除
删除55
a
50
b
30
c
60 70
d
10
e
40
f
58
g
65
h
75 80

- 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。
- 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。

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。

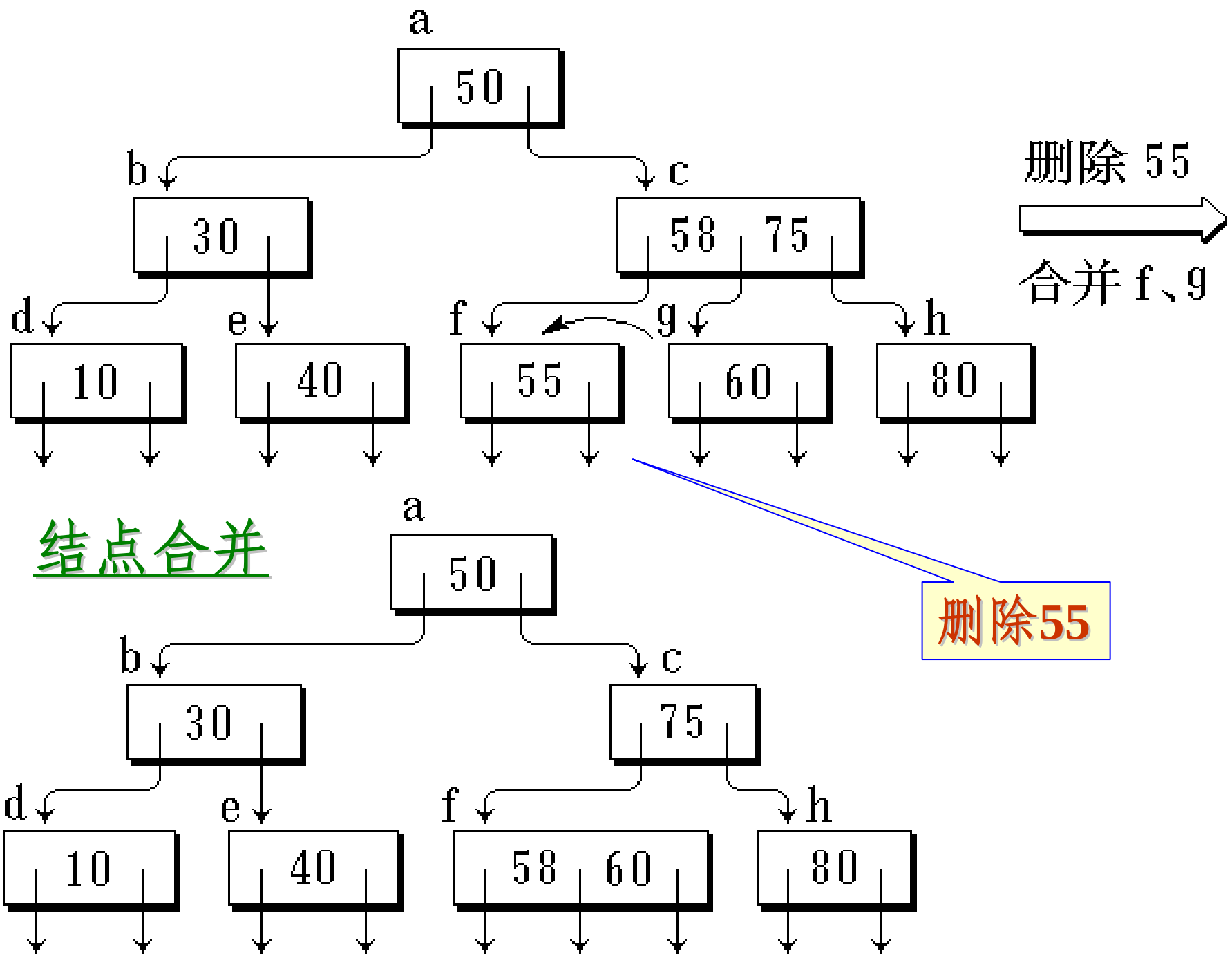

a
50
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
删除 65
调整 g、c、h
结点联合调整
删除65
a
50
b
30
c
60 75
d
10
e
40
f
55 58
g
70
h
80

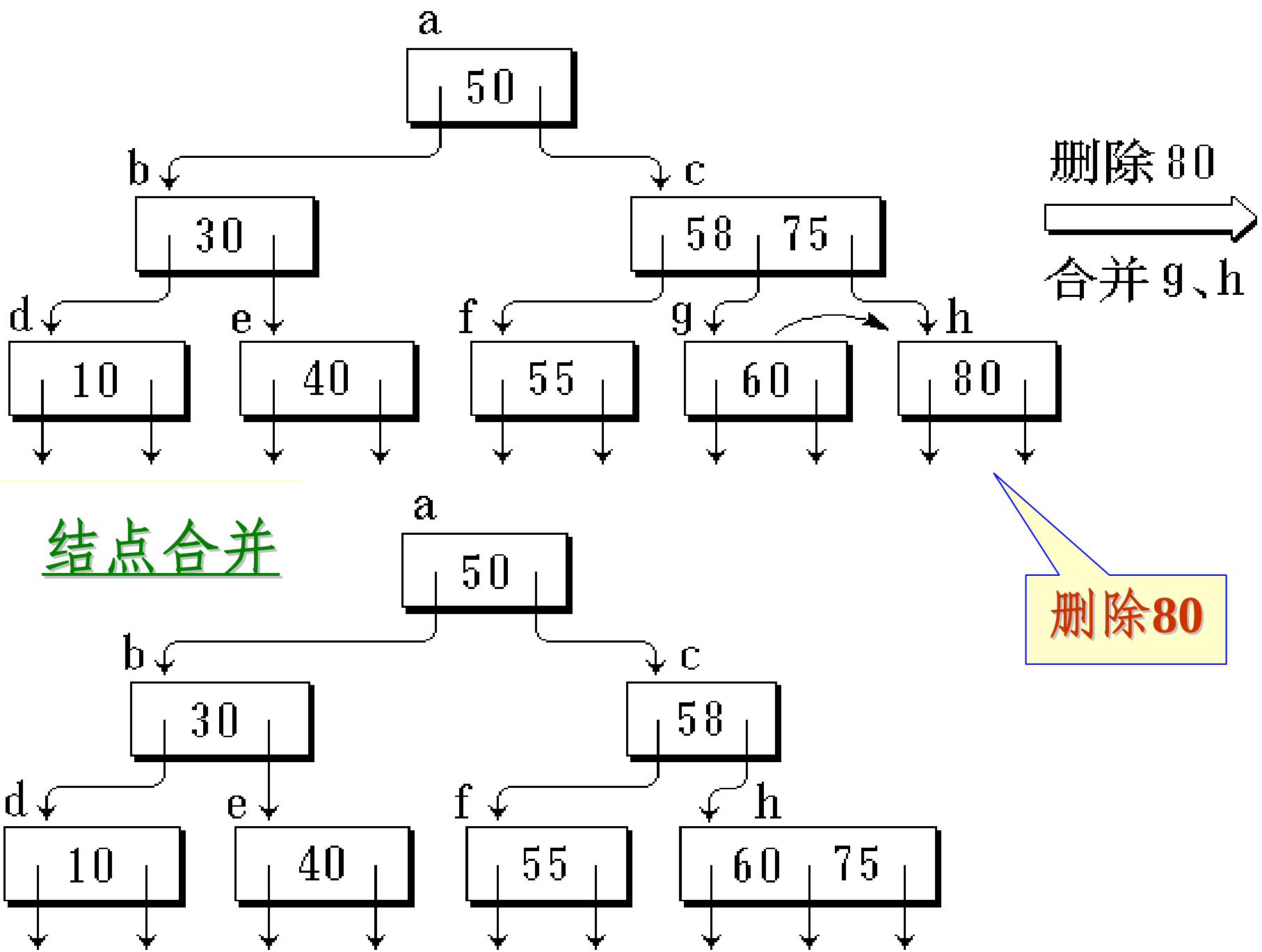

a
50
b
30
c
60 75
d
10
e
40
f
55 58
g
70
h
80
删除70
调整f、c、g
删除70
a
50
b
30
c
58 75
d
10
e
40
f
55
g
60
h
80

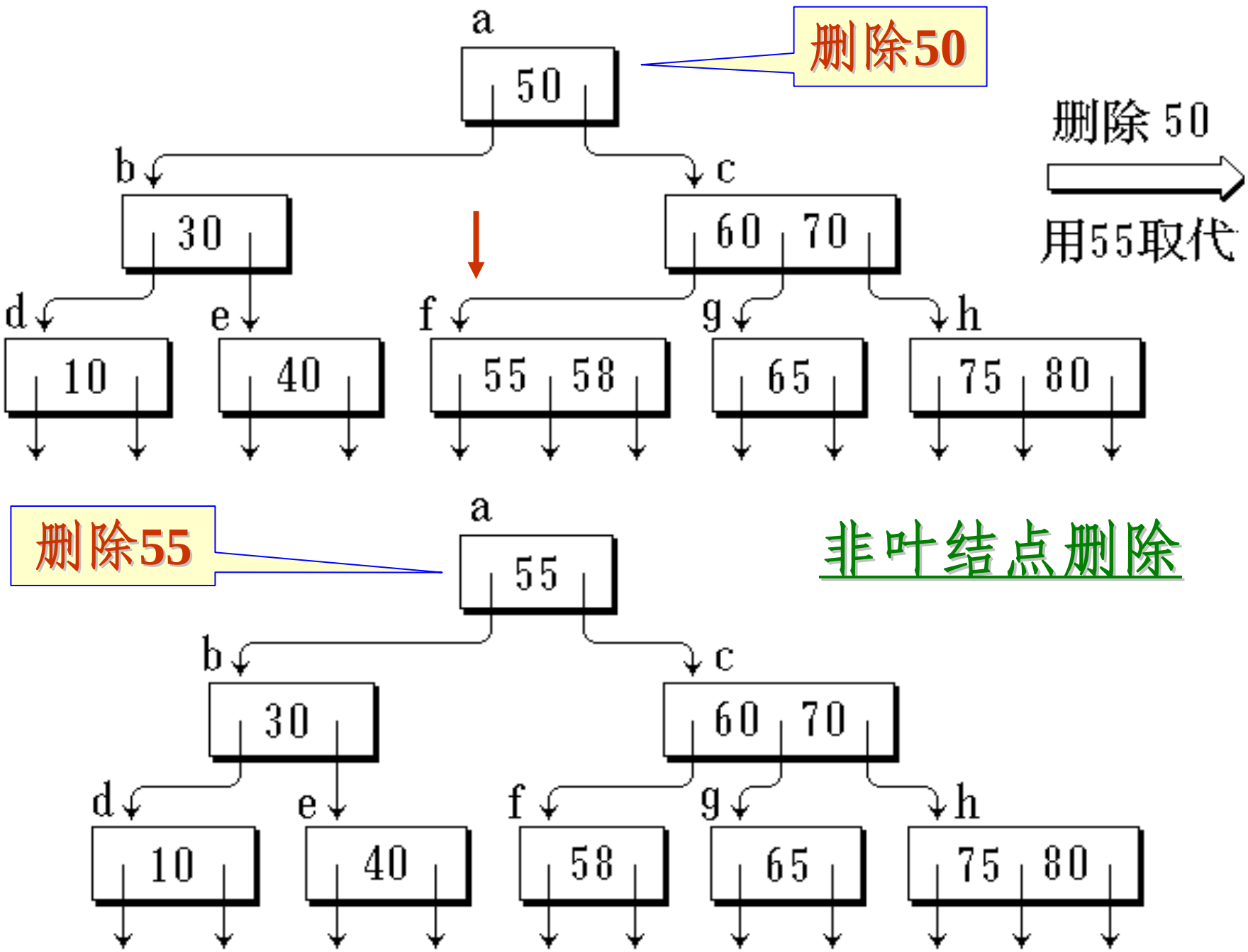
- 将双亲结点 p 中相应关键码下移到选定保留的结点中。若要合并 p 中的子树指针 P_i 与 P_{i+1} 所指的结点，且保留 P_i 所指结点，则把 p 中的关键码 K_{i+1} 下移到 P_i 所指的结点中。
- 把 p 中子树指针 P_{i+1} 所指结点中的全部指针和关键码都照搬到 P_i 所指结点的后面。删去 P_{i+1} 所指的结点。
- 在结点 p 中用后面剩余的关键码和指针填补关键码 K_{i+1} 和指针 P_{i+1}。
- 修改结点 p 和选定保留结点的关键码个数。

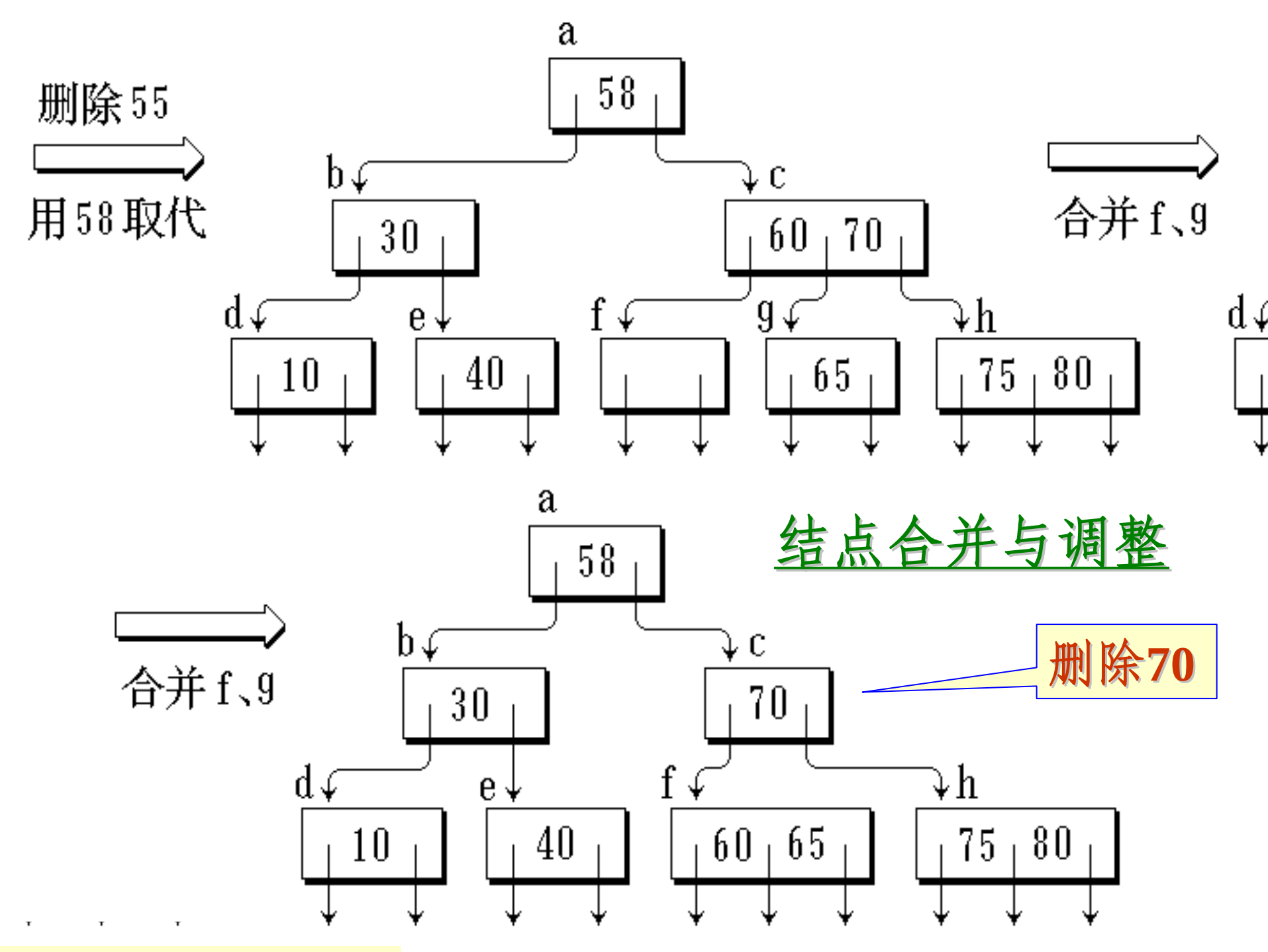
- 在合并结点的过程中，双亲结点中的关键码个数减少了。

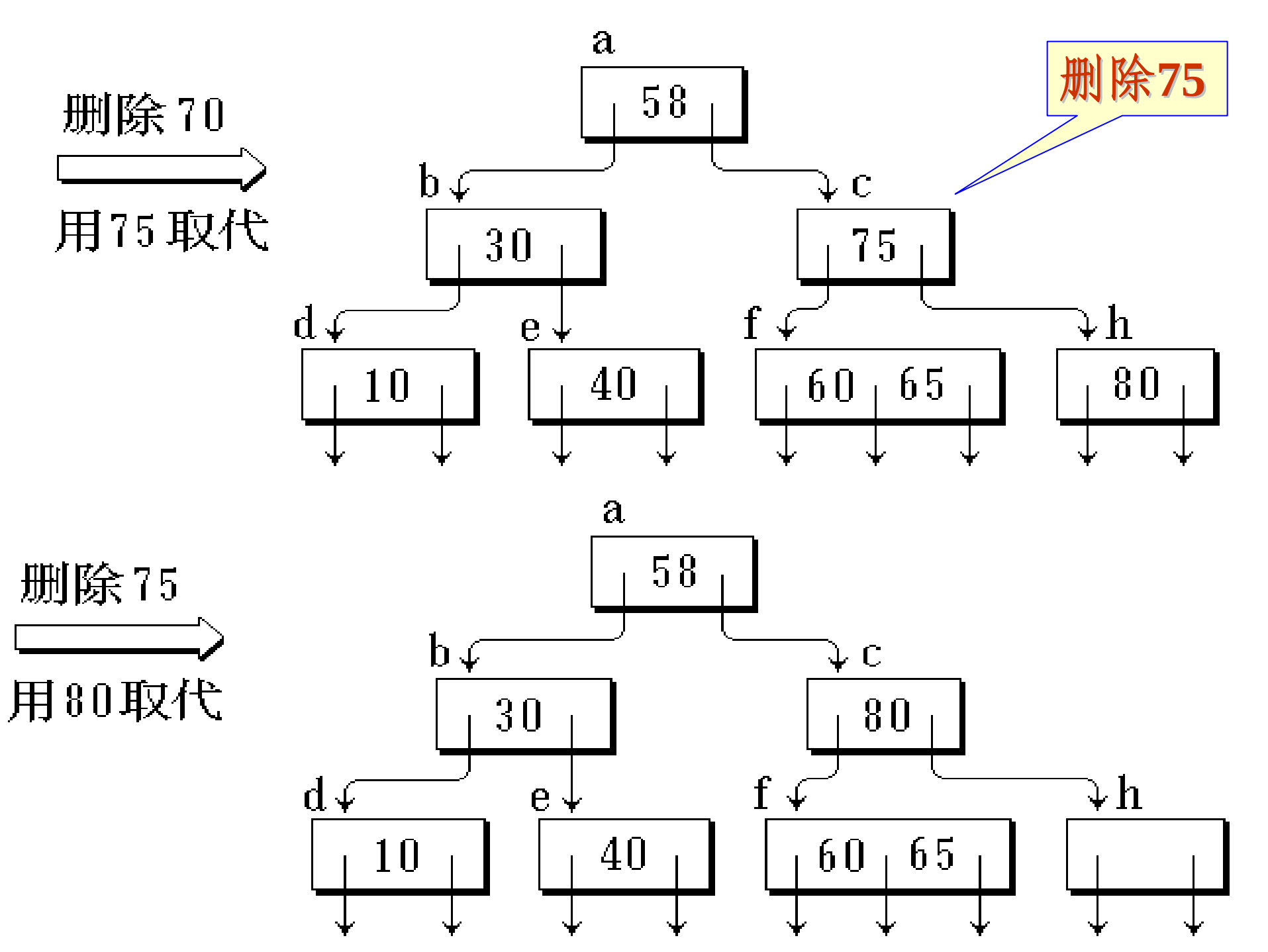
- 若双亲结点是根结点且结点关键码个数减到0，则该双亲结点应从树上删去，合并后保留的结点成为新的根结点；否则将双亲结点与合并后保留的结点都写回磁盘，删除处理结束。
- 若双亲结点不是根结点，且关键码个数减到⌈m/2⌉-2，又要与它自己的兄弟结点合并，重复上面的合并步骤。最坏情况下这种结点合并处理要自下向上直到根结点。

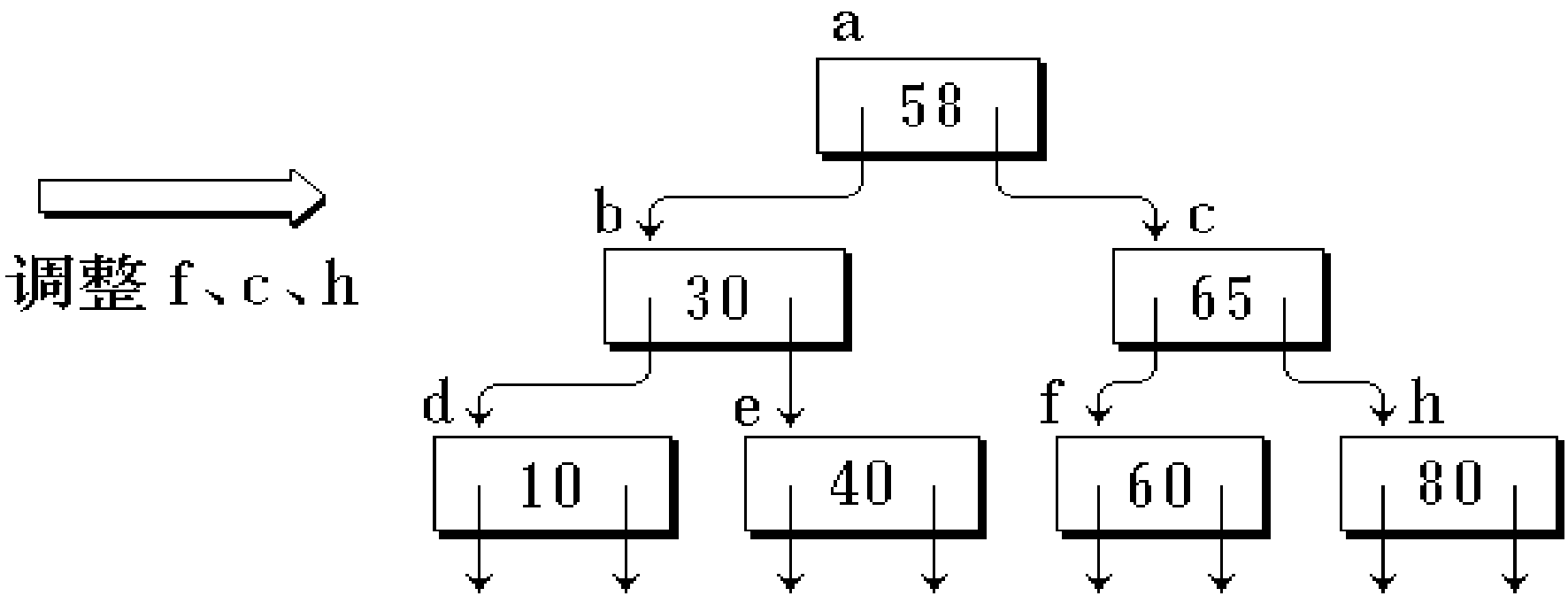

a
50
b
30
c
58 75
d
10
e
40
f
55
g
60
h
80
删除 55
合并 f、g
结点合并
删除55
a
50
b
30
c
75
d
10
e
40
f
58 60
h
80


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


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
删除55
a
55
非叶结点删除
b
30
c
60 70
d
10
e
40
f
58
g
65
h
75 80


删除55
用58取代
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
合并f、g
删除70


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



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⌈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);
            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
    }
}
```

B+树

- B+树可以看作是B_树的一种变形，在实现文件索引结构方面比B_树使用得更普遍。
- 一棵 *m* 阶B+树可以定义如下:
  - 树中每个非叶结点最多有 *m* 棵子树;
  - 根结点 (非叶结点) 至少有 2 棵子树。除根结点外，其它的非叶结点至少有⌈*m*/2⌉棵子树；有 *n* 棵子树的非叶结点有 *n*−1 个关键码。
  - 所有叶结点都处于同一层次上，包含了全部关键码及指向相应数据对象存放地址的指针，且叶结点本身按关键码从小到大顺序链接;

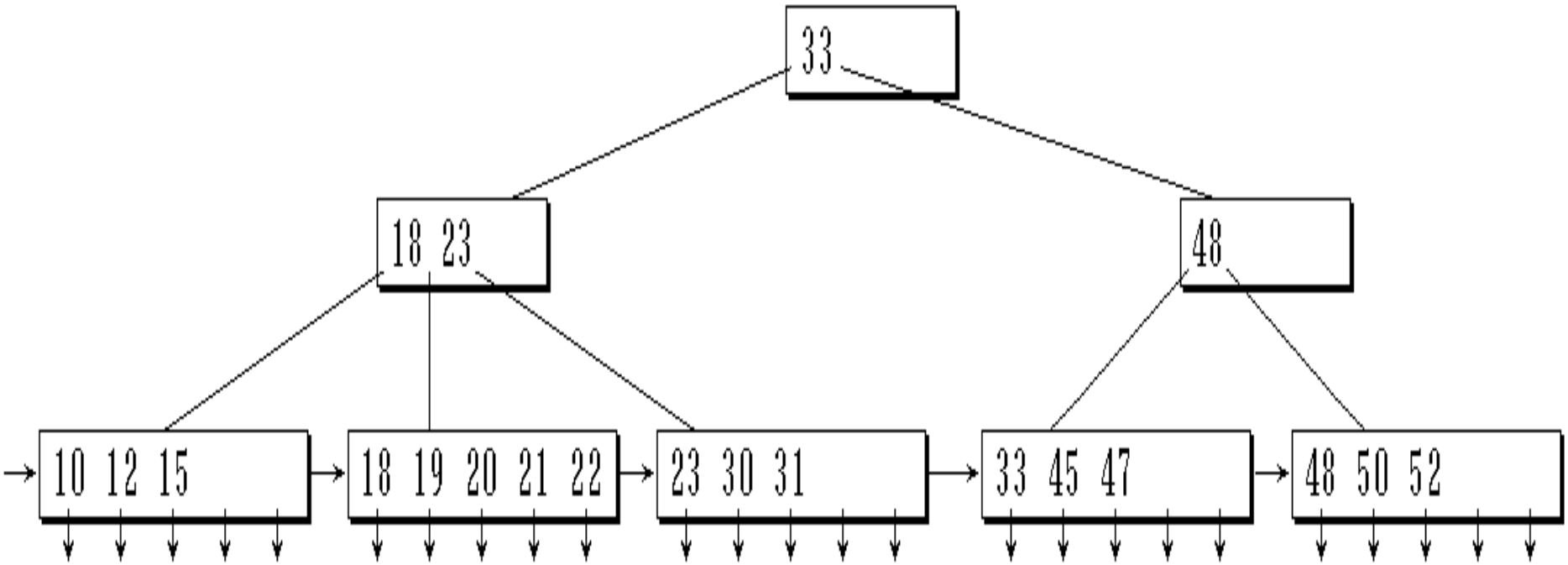
- 每个叶结点中的子树棵数 n 可以多于 m，可以少于 m，视关键码字节数及对象地址指针字节数而定。

  若设结点可容纳最大关键码数为 $m1$，则指向对象的地址指针也有 $m1$ 个。

  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。
- 所有的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。
- 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。
- 可对B+树进行两种搜索运算：

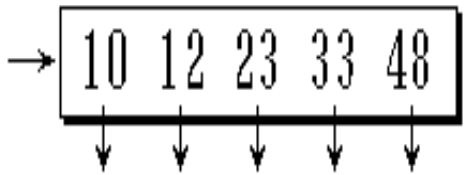


- 循叶结点链顺序搜索
- 另一种是从根结点开始，进行自顶向下，直至叶结点的随机搜索。

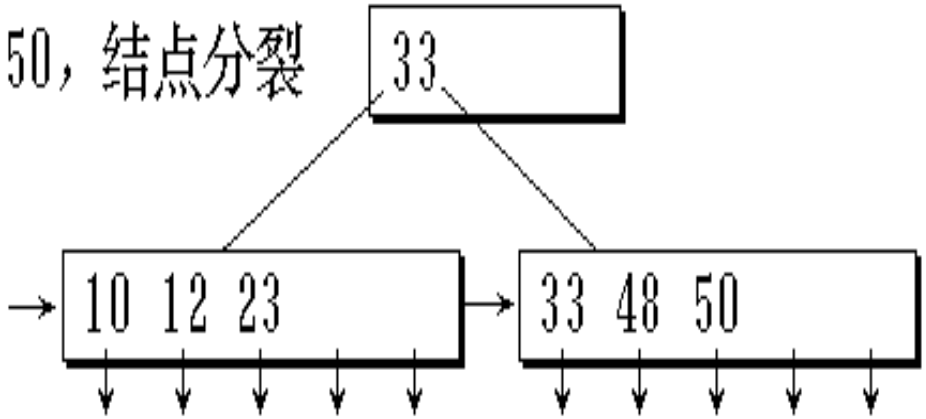
- 在B+树上进行随机搜索、插入和删除的过程基本上与B_树类似。只是在搜索过程中，如果非叶结点上的关键码等于给定值，搜索并不停止，而是继续沿右指针向下，一直查到叶结点上的这个关键码。
- B+树的搜索分析类似于B_树。
- B+树的插入仅在叶结点上进行。每插入一个关键码-指针索引项后都要判断结点中的子树棵数是否超出范围。

- 当插入后结点中的子树棵数 $n > m1$ 时，需要将叶结点分裂为两个结点，它们的关键码分别为$\lceil (m1+1)/2 \rceil$和$\lfloor (m1+1)/2 \rfloor$。并且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。此后，问题归于在非叶结点中的插入了。
- 在非叶结点中关键码的插入与叶结点的插入类似，但非叶结点中的子树棵数的上限为 m，超出这个范围就需要进行结点分裂。
- 在做根结点分裂时，因为没有双亲结点，就必须创建新的双亲结点，作为树的新根。这样树的高度就增加一层了。

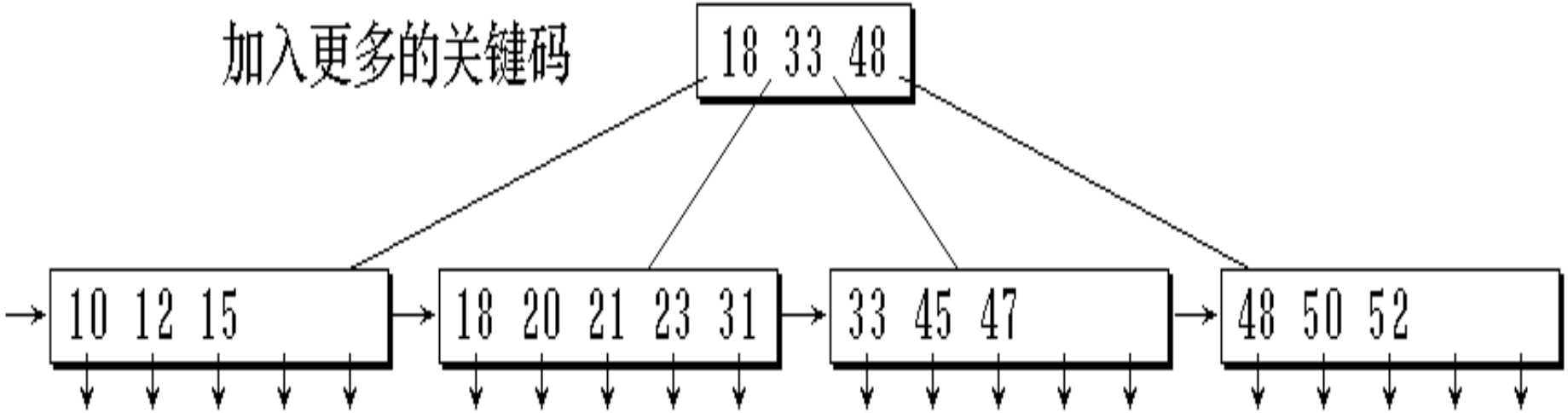
包含5个关键码的B+树



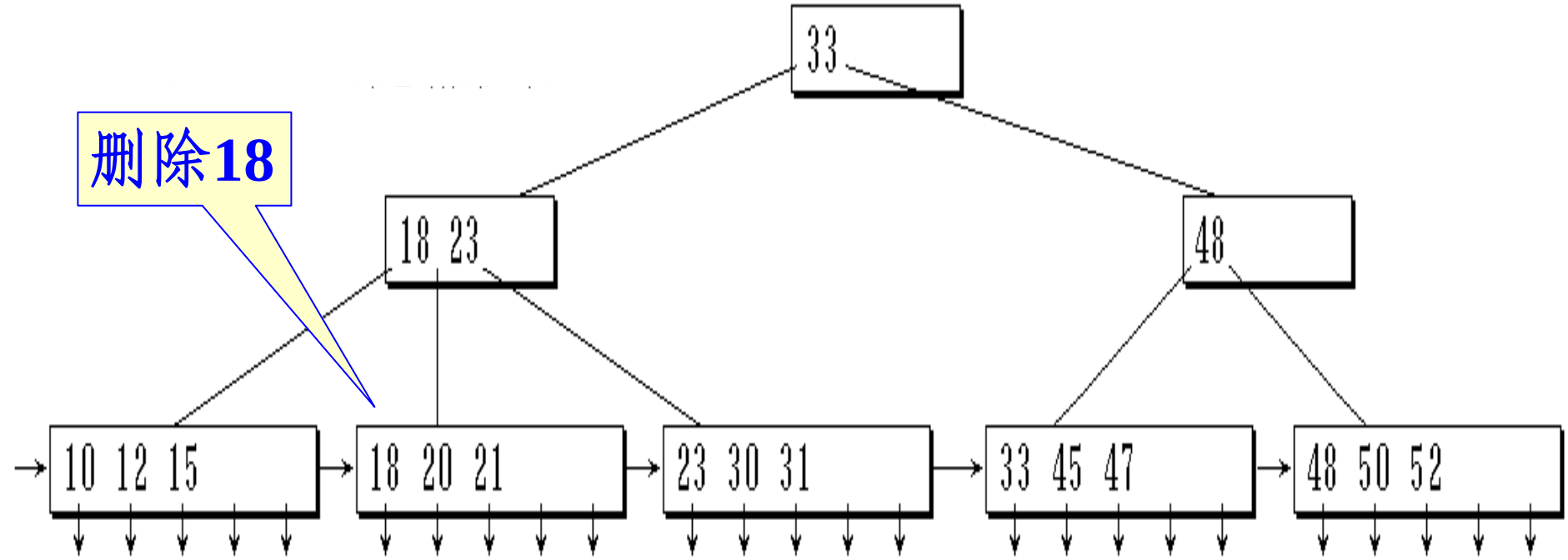
加入关键码50，结点分裂



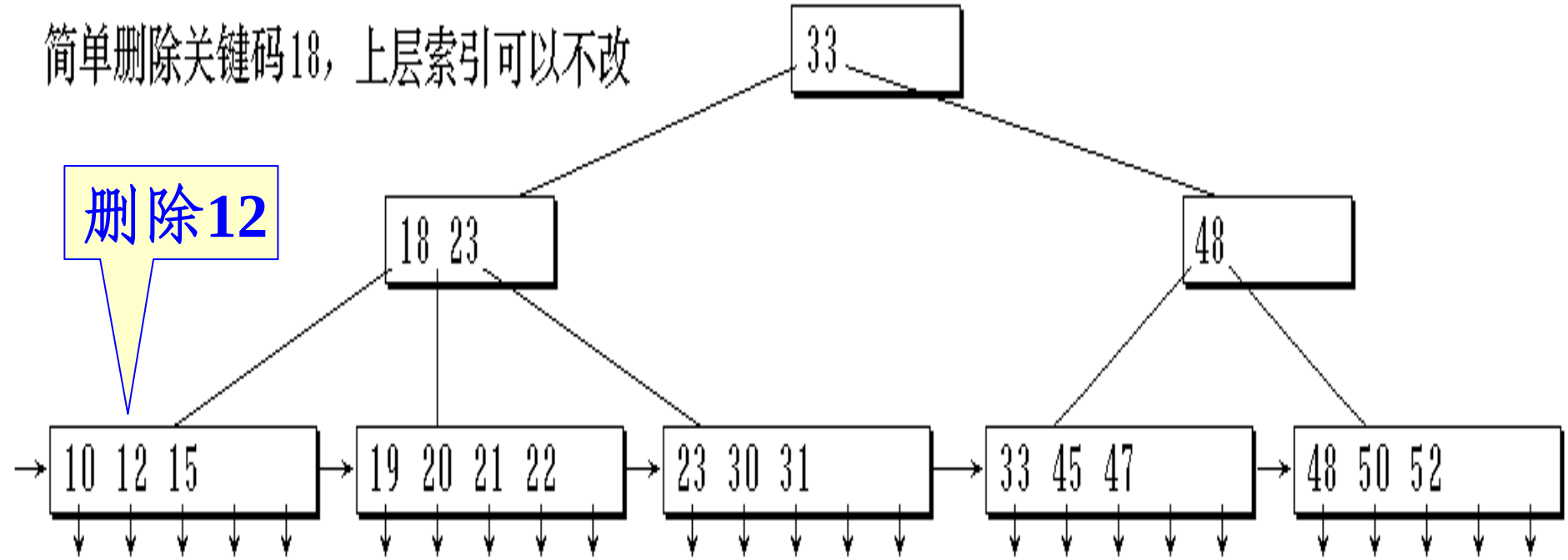
加入更多的关键码

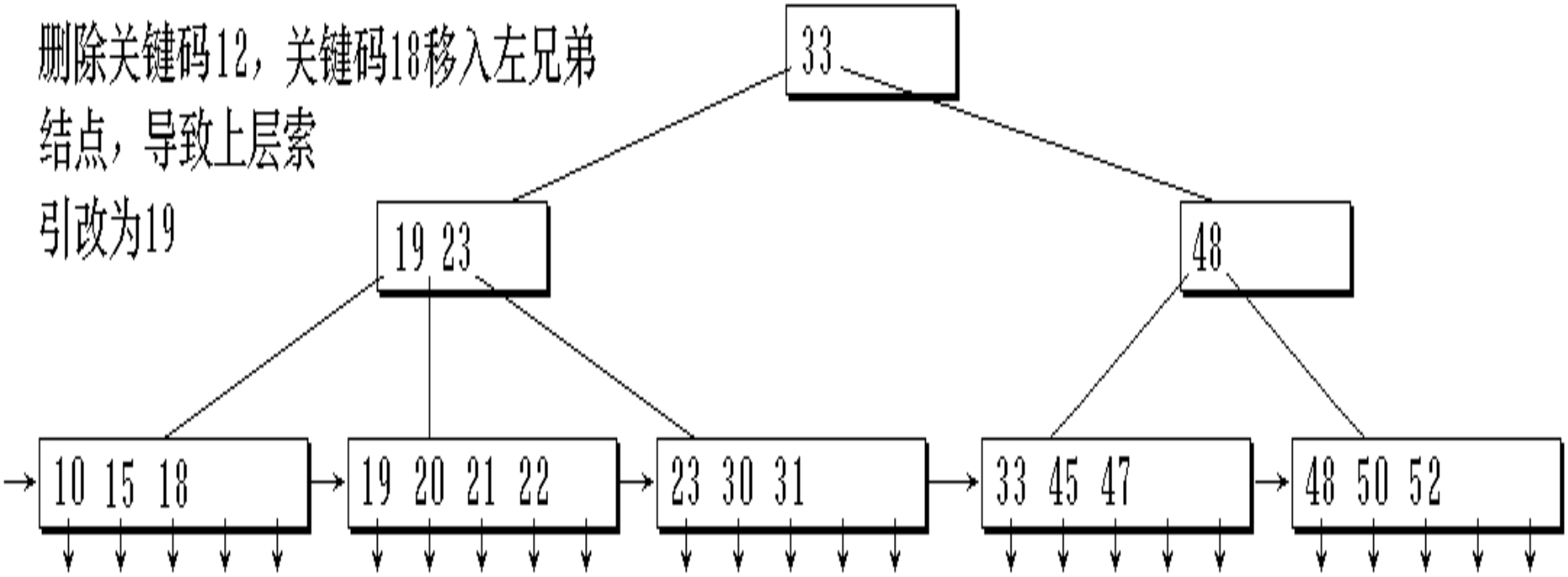


- B+树的删除仅在叶结点上进行。当在叶结点上删除一个关键码-指针索引项后，结点中的子树棵数仍然不少于$\lceil m1/2 \rceil$，这属于简单删除，其上层索引可以不改变。
- 如果删除的关键码是该结点的最小关键码，但因在其上层的副本只是起了一个引导搜索的“分界关键码”的作用，所以上层的副本仍然可以保留。
- 如果在叶结点中删除一个关键码-指针索引项后，该结点中的子树棵数 n 小于结点子树棵数的下限$\lceil m1/2 \rceil$，必须做结点的调整或合并工作。



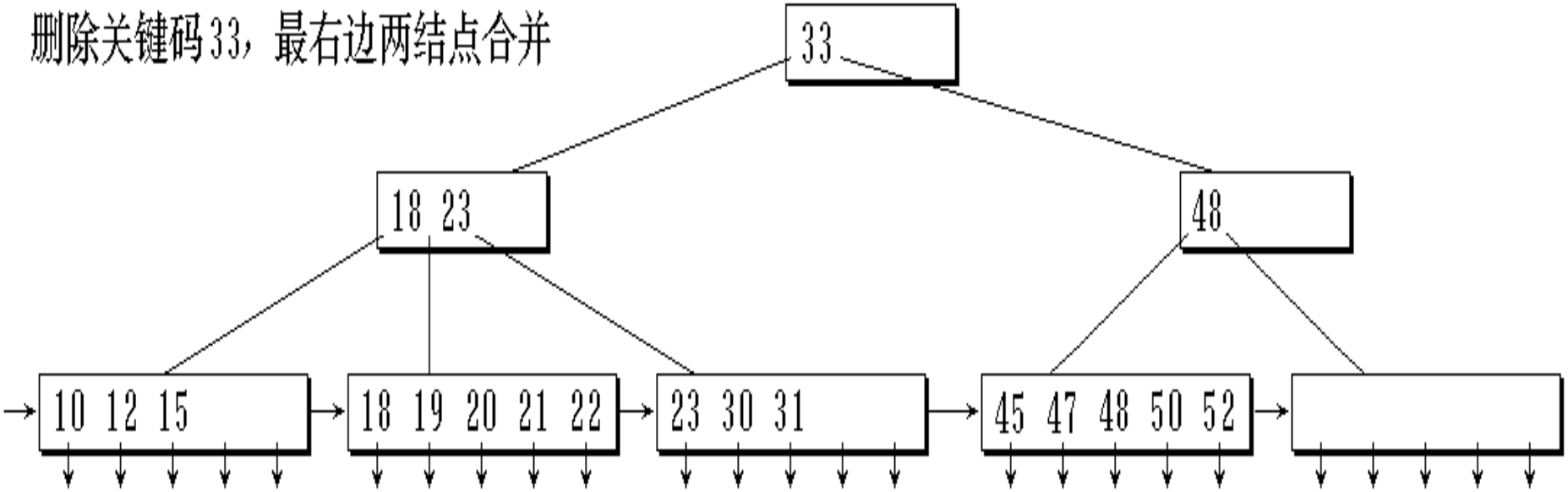
简单删除关键码18，上层索引可以不改



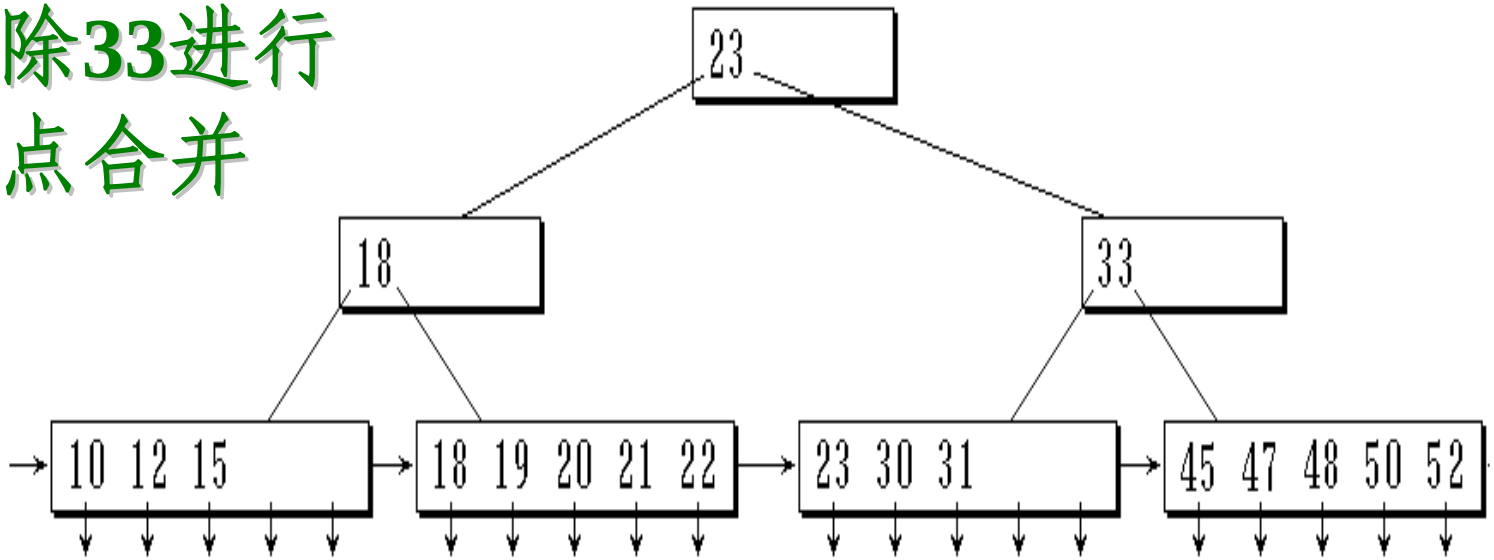


- 如果右兄弟结点的子树棵数已达到下限⌈*m*1/2⌉，没有多余的关键码可以移入被删关键码所在的结点，这时必须进行两个结点的合并。将右兄弟结点中的所有关键码-指针索引项移入被删关键码所在结点，再将右兄弟结点删去。

删除关键码33，最右边两结点合并



删除33进行结点合并



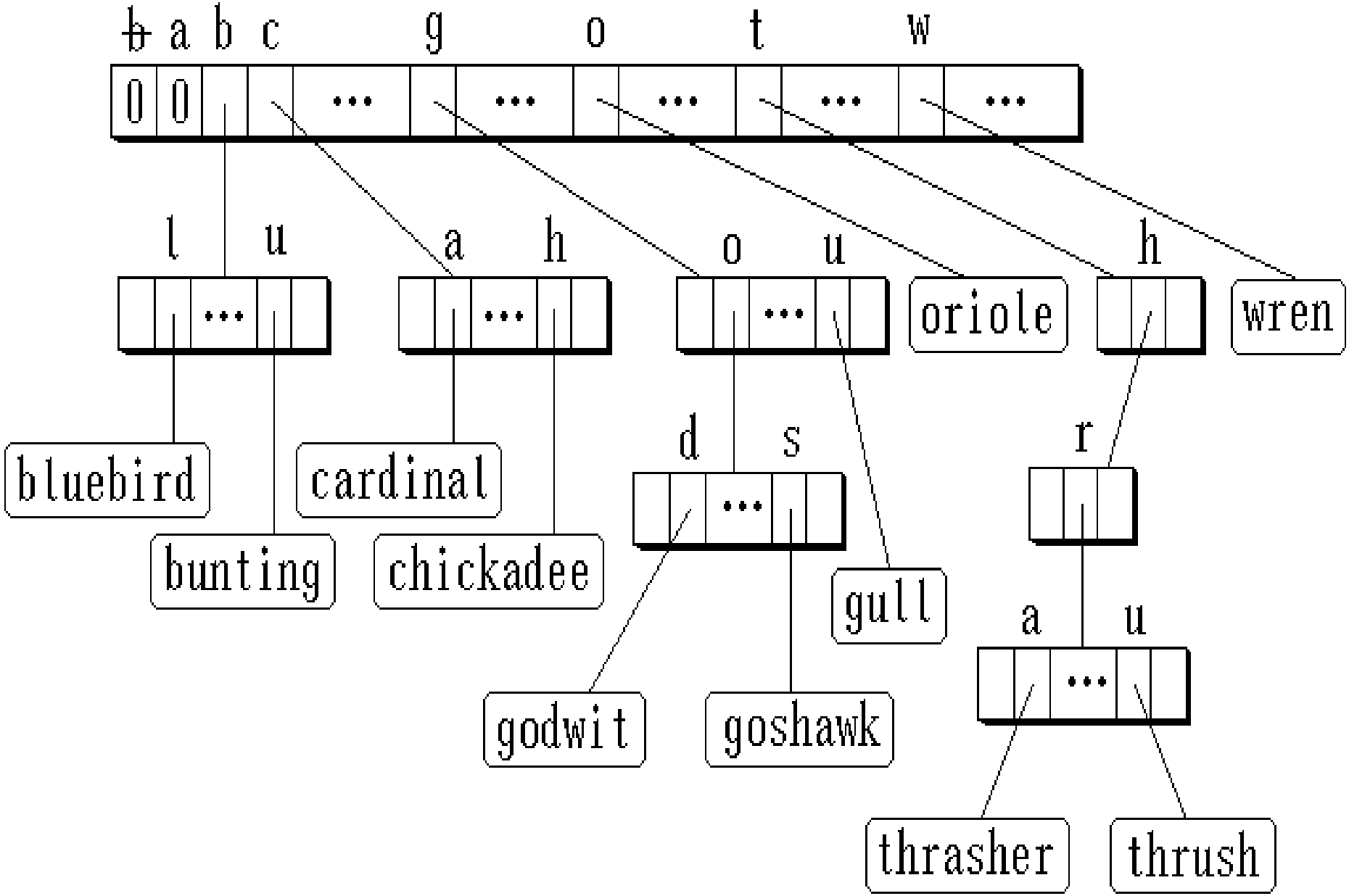
- 结点的合并将导致双亲结点中“分界关键码”的减少，有可能减到非叶结点中子树棵数的下限⌈*m*/2⌉以下。这样将引起非叶结点的调整或合并。
- 如果根结点的最后两个子女结点合并，树的层数就会减少一层。

*Trie*树

当关键码是可变长时，*Trie*树是一种特别有用的索引结构。

Trie 树的定义

- *Trie*树是一棵度 $m \geq 2$ 的树，它的每一层分支不是靠整个关键码的值来确定，而是由关键码的一个分量来确定。
- 图示*Trie*树，关键码由英文字母组成。它包括两类结点：元素结点和分支结点。元素结点包含整个*key*数据；分支结点有27个指针，其中有一个空白字符‘b’，用来终结关键码；其它用来标识‘a’, ‘b’,..., ‘z’等26个英文字母。


b a b c g o t w
0 0 ... ...
l u a h o u h
... ... oriole wren
bluebird bunting cardinal chickadee
d s r
... gull
godwit goshawk
a u
...
thrasher thrush

- 在第0层，所有的关键码根据它们第0位字符，被划分到互不相交的27个类中。
- 因此，*root*→*brch.link*[*i*] 指向一棵子*Trie*树，该子*Trie*树上所包含的所有关键码都是以第 *i* 个英文字母开头。
- 若某一关键码第 *j* 位字母在英文字母表中顺序为 *i* (*i* = 0, 1, …, 26), 则它在*Trie*树的第 *j* 层分支结点中从第 *i* 个指针向下找第 *j*+1 位字母所在结点。当一棵子*Trie*树上只有一个关键码时，就由一个元素结点来代替。在这个结点中包含有关键码，以及其它相关的信息，如对应数据对象的存放地址等。

*Trie*树的类定义

```
const int MaxKeySize = 25;    //关键码最大位数

typedef struct {                        //关键码类型
    char ch[MaxKeySize];                //关键码存放数组
    int currentSize;                    //关键码当前位数
} KeyType;

class TrieNode {                        //Trie树结点类定义
friend class Trie;
protected:
    enum { branch, element } NodeType;  //结点类型
```

```
union NodeType {    //根据结点类型的两种结构
    struct {                        //分支结点
        int num;                    //本结点关键码个数
        TrieNode *link[27];         //指针数组
    } branch;
    struct {                        //元素结点
        KeyType key;                //关键码
        recordNode *recptr;         //指向数据对象指针
    } element;
}
}
```

```
class Trie {                                    //Trie树的类定义
private:
    TrieNode *root, *current;
public:
    RecordNode* Search ( const keyType & );
    int Insert ( const KeyType & );
    int Delete ( const KeyType & );
}
```

*Trie*树的搜索

- 为了在*Trie*树上进行搜索，要求把关键码分解成一些字符元素，并根据这些字符向下进行分支。
- 函数 ***Search*** 设定 ***current = NULL***, 表示不指示任何一个分支结点，如果 ***current*** 指向一个元素结点 ***element***，则 ***current→element.key*** 是 ***current*** 所指结点中的关键码。

***Trie*树的搜索算法**

```
RecordNode* Trie::Search ( const KeyType & x ) {
    KeyType k = x.key;   concatenate ( k, '⊥' );
```

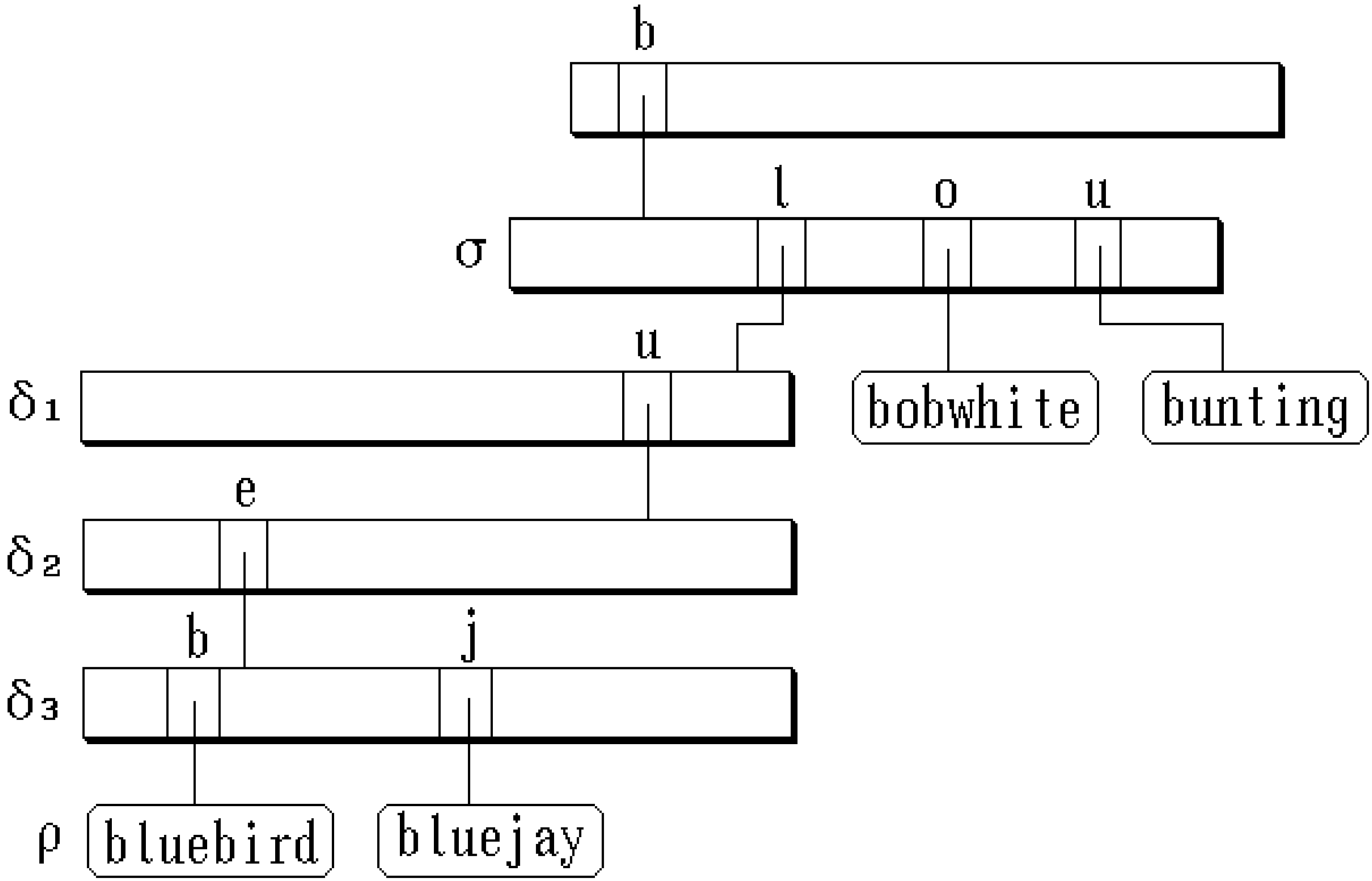
```
current = root;  int i = 0;          //扫描初始化
while ( current != NULL &&
        current→NodeType != element &&
        i <= currentSize ) {
   current = current→branch.link[ord (x.ch[i])];
   // ord( )将x.ch[i]转换成它在字母表中序号
   i =  i++;
};
if ( current != NULL &&
     current→NodeType == element &&
     current→element.key == x )
   return current→recptr;
else return NULL;
```

}

- 经验证，*Trie*树的搜索算法在最坏情况下搜索的时间代价是 **O(*l*)**。其中，*l* 是*Trie*树的层数(包括分支结点和元素结点在内)。
- 在用作索引时，*Trie*树的所有结点都驻留在磁盘上。搜索时最多做 *l* 次磁盘存取。
- 当结点驻留在磁盘上时，不能使用C++的指针 (*pointer*) 类型，因为C++不允许指针的输入 / 输出。在结点中的 *link* 指针可改用整型 (*integer*) 实现。

在*Trie*树上的插入和删除

- 示例：插入关键码**bobwhite**和**bluejay**。
  - 插入 *x* = **bobwhite** 时，首先搜索*Trie*树寻找 **bobwhite** 所在的结点。
  - 如果找到结点，并发现该结点的 ***link*[‘o’] = *NULL***。*x*不在*Trie*树中，可以在该处插入。插入结果参看图。
  - 插入 *x* = **bluejay**时，用*Trie*树搜索算法可找到包含有 **bluebird** 的元素结点，关键码 **bluebird** 和 **bluejay** 是两个不同的值，它们在第5个字母处不匹配。从 Trie树沿搜索路径，在第4层分支。插入结果参看图。



在*Trie*树上插入bobwhite和bluejay后的结果

- 示例：考虑在上图所示*Trie*树中删除**bobwhite**。此时，只要将该结点*link*[‘o’]置为0 (*NULL*)即可，*Trie*树的其它部分不需要改变。
- 考虑删除 **bluejay**。删除之后在标记为 δ_3 的子*Trie*树中只剩下一个关键码，这表明可以删去结点δ_3，同时结点 ρ 向上移动一层。对结点 δ_2 和 δ_1 可以做同样的工作，最后到达结点σ。因为以 σ 为根的子*Trie*树中有多个关键码，所以它不能删去，令该结点*link*[‘*l*’] = ρ即可。
- 为便于*Trie*树的删除，在每个分支结点中设置了一个 ***num*** 数据成员，它记载了结点中子女的数目。

散列 (Hashing)

在现实中，经常遇到按给定的值进行查询的事例。为此，必须在开发相应软件时考虑在记录的存放位置和用以标识它的数据项（称为关键码）之间的对应关系，从而选择适当的数据结构，很方便地根据记录的关键码检索到对应记录的信息。

词典(Dictionary)的抽象数据类型

- 在计算机科学中把词典当作一种抽象数据类型。
- 在讨论词典抽象数据类型时，把词典定义为<名字-属性>对的集合。根据问题的不同，可以为名字和属性赋予不同的含义。

- 例如，在图书馆检索目录中，名字是书名，属性是索书号及作者等信息；在计算机活动文件表中，名字是文件名，属性是文件地址、大小等信息；而在编译程序建立的变量表中，名字是变量标识符，属性是变量的属性、存放地址等信息。

词典的抽象数据类型

```
const int DefaultSize = 26;
template<class Name, Attribute>class Dictionary{
public:
    Dictionary ( int size = DefaultSize );
    int IsIn ( Name name );
```

```
    Attribute *Find ( Name name );
    void Insert ( Name name, Attribute attr );
    void Remove ( Name name );
}
```

- 在词典的所有操作中，最基本的只有3种：
  - ***Find* (搜索)**
  - ***Insert* (插入)**
  - ***Remove* (删除)**
- 在选择词典的表示时，必须确保这几个操作的实现。

- 通常，用文件 (表格) 来表示实际的对象集合，用文件记录 (表格的表项) 来表示单个对象。这样，词典中的<名字–属性>对将被存于记录 (表项) 中，通过表项的关键码 (<名字–属性>对的名字) 来标识该表项。
- 表项的存放位置及其关键码之间的对应关系可以用一个二元组表示:

  (关键码***key***，表项位置指针***addr***)
- 这个二元组构成搜索某一指定表项的索引项。考虑到搜索效率，可以用顺序表的方式组织词典，也可以用二叉搜索树或多路搜索树的方式组织词典，本节讨论另一种搜索效率很高的组织词典的方法，即散列表结构。

静态散列方法

- 散列方法在表项的存储位置与它的关键码之间建立一个确定的对应函数关系***Hash( )***，使每个关键码与结构中一个唯一存储位置相对应：

$$Address = Hash(Rec.key)$$

- 在搜索时，首先对表项的关键码进行函数计算，把函数值当做表项的存储位置，在结构中按此位置取表项比较。若关键码相等，则搜索成功。在存放表项时，依相同函数计算存储位置，并按此位置存放。这种方法就是散列方法。在散列方法中使用的转换函数叫做散列函数。而按此种想法构造出来的表或结构就叫做散列表。

- 使用散列方法进行搜索不必进行多次关键码的比较，搜索速度比较快，可以直接到达或逼近具有此关键码的表项的实际存放地址。
- 散列函数是一个压缩映象函数。关键码集合比散列表地址集合大得多。因此有可能经过散列函数的计算，把不同的关键码映射到同一个散列地址上，这就产生了冲突 **(Collision)**。
- 示例：有一组表项，其关键码分别是

  12361，07251，03309，30976

  采用的散列函数是

  $hash(x) = x \% 73 + 13420$

  其中，“%”是除法取余操作。

则有：*hash*(12361) = *hash*(07250) = *hash*(03309) = *hash*(30976) = 13444。就是说，对不同的关键码，通过散列函数的计算，得到了同一散列地址。我们称这些产生冲突的散列地址相同的不同关键码为同义词。

- 由于关键码集合比地址集合大得多，冲突很难避免。所以对于散列方法，需要讨论以下两个问题：
  - 对于给定的一个关键码集合，选择一个计算简单且地址分布比较均匀的散列函数，避免或尽量减少冲突；
  - 拟订解决冲突的方案。

散列函数

构造散列函数时的几点要求：

- 散列函数的定义域必须包括需要存储的全部关键码，如果散列表允许有 ***m*** 个地址时，其值域必须在 **0** 到 ***m*−1** 之间。
- 散列函数计算出来的地址应能均匀分布在整个地址空间中：若 ***key*** 是从关键码集合中随机抽取的一个关键码，散列函数应能以同等概率取 **0** 到 ***m*−1** 中的每一个值。
- 散列函数应是简单的，能在较短的时间内计算出结果。

直接定址法

此类函数取关键码的某个线性函数值作为散列地址:

$$Hash(key) = a * key + b \quad \{a, b\text{为常数}\}$$

这类散列函数是一对一的映射，一般不会产生冲突。但是，它要求散列地址空间的大小与关键码集合的大小相同。

示例：有一组关键码如下：{942148, 941269, 940527, 941630, 941805, 941558, 942047, 940001}。散列函数为

$$Hash(key) = key - 940000$$

则有

Hash (942148) = 2148　　*Hash* (941269) = 1269

Hash (940527) = 527　　*Hash* (941630) = 1630

Hash (941805) = 1805　　*Hash* (941558) = 1558

Hash (942047) = 2047　　*Hash* (940001) = 1

可以按计算出的地址存放记录。

- 数字分析法

设有 ***n*** 个 ***d*** 位数，每一位可能有 ***r*** 种不同的符号。这 ***r*** 种不同的符号在各位上出现的频率不一定相同，可能在某些位上分布均匀些；在某些位上分布不均匀，只有某几种符号经常出现。可根据散列表的大小，选取其中各种符号分布均匀的若干位作为散列地址。

- 计算各位数字中符号分布的均匀度 λ_k 的公式:

$$\lambda_k = \sum_{i=1}^{r} (\alpha_i^k - n/r)^2$$

- 其中，α_i^k 表示第 i 个符号在第 k 位上出现的次数，n/r 表示各种符号在 n 个数中均匀出现的期望值。计算出的 λ_k 值越小，表明在该位 (第 k 位) 各种符号分布得越均匀。
- 若散列表地址范围有 3 位数字, 取各关键码的④⑤⑥位做为记录的散列地址。也可以把第①，②，③和第⑤位相加，舍去进位位，变成一位数，与第④，⑥位合起来作为散列地址。还有其它方法。

9	4	2	1	4	8	①位，$\lambda_1 = 57.60$
9	4	1	2	6	9	②位，$\lambda_2 = 57.60$
9	4	0	5	2	7	③位，$\lambda_3 = 17.60$
9	4	1	6	3	0	④位，$\lambda_4 = 5.60$
9	4	1	8	0	5	⑤位，$\lambda_5 = 5.60$
9	4	1	5	5	8	⑥位，$\lambda_6 = 5.60$
9	4	2	0	4	7	
9	4	0	0	0	1	
①	②	③	④	⑤	⑥	

数字分析法仅适用于事先明确知道表中所有关键码每一位数值的分布情况，它完全依赖于关键码集合。如果换一个关键码集合，选择哪几位要重新决定。

除留余数法

- 设散列表中允许的地址数为 m, 取一个不大于 m，但最接近于或等于 m 的质数 p, 或选取一个不含有小于 20 的质因数的合数作为除数，利用以下公式把关键码转换成散列地址。散列函数为:

$$hash(key) = key \% p \quad p \le m$$

- 其中,“%”是整数除法取余的运算，要求这时的质数 p 不是接近2的幂。
- 示例：有一个关键码 $key = 962148$，散列表大小 $m = 25$，即 $HT[25]$。取质数 $p = 23$。散列函数 $hash(key) = key \% p$。则散列地址为

$hash(962148) = 962148 \% 23 = 12$。

- 可以按计算出的地址存放记录。需要注意的是，使用上面的散列函数计算出来的地址范围是 0到 22，因此，从23到24这几个散列地址实际上在一开始是不可能用散列函数计算出来的，只可能在处理溢出时达到这些地址。

乘余取整法

- 使用此方法时，先让关键码 *key* 乘上一个常数 $A\ (0 < A < 1)$，提取乘积的小数部分。然后，再用整数 *n* 乘以这个值，对结果向下取整，把它做为散列的地址。散列函数为：

$$hash(key) = \lfloor n * (A * key \% 1) \rfloor$$

其中，“*A* * *key* % 1”表示取 *A* * *key* 小数部分：

$$A * key \,\%\, 1 = A * key - \lfloor A * key \rfloor$$

示例：设关键码 *key* = 123456，*n* = 10000，且取 $A = \left(\sqrt{5}-1\right)/2 = 0.6180339$，则

$$hash(key) = \lfloor 10000*(0.6180339*key \,\%\, 1) \rfloor$$

因此有

$$hash(123456) =$$

$$= \lfloor 10000*(0.6180339*123456 \,\%\, 1) \rfloor =$$

$$= \lfloor 10000 * (76300.0041151\ldots \,\%\, 1) \rfloor =$$

$$= \lfloor 10000 * 0.0041151\ldots \rfloor = 41$$

此方法的优点是对 *n* 的选择不很关键。

平方取中法

- 此方法在词典处理中使用十分广泛。它先计算构成关键码的标识符的内码的平方，然后按照散列表的大小取中间的若干位作为散列地址。
- 设标识符可以用一个计算机字长的内码表示。因为内码平方数的中间几位一般是由标识符所有字符决定，所以对不同的标识符计算出的散列地址大多不相同，即使其中有些字符相同。
- 在平方取中法中，一般取散列地址为 2 的某次幂。例如，若散列地址总数取为 $m = 2^r$，则对内码的平方数取中间的 r 位。如果 $r = 3$，所取得的散列地址参看图的最右一列。

标识符	内码	内码的平方	散列地址
A	01	01	001
A1	0134	20420	042
A9	0144	23420	342
B	02	4	004
DMAX	04150130	21526443617100	443
DMAX1	0415013034	526447352151420	352
AMAX	01150130	1354236171 00	236
AMAX1	0115013034	345424652151420	652

标识符的八进制内码表示及其平方值

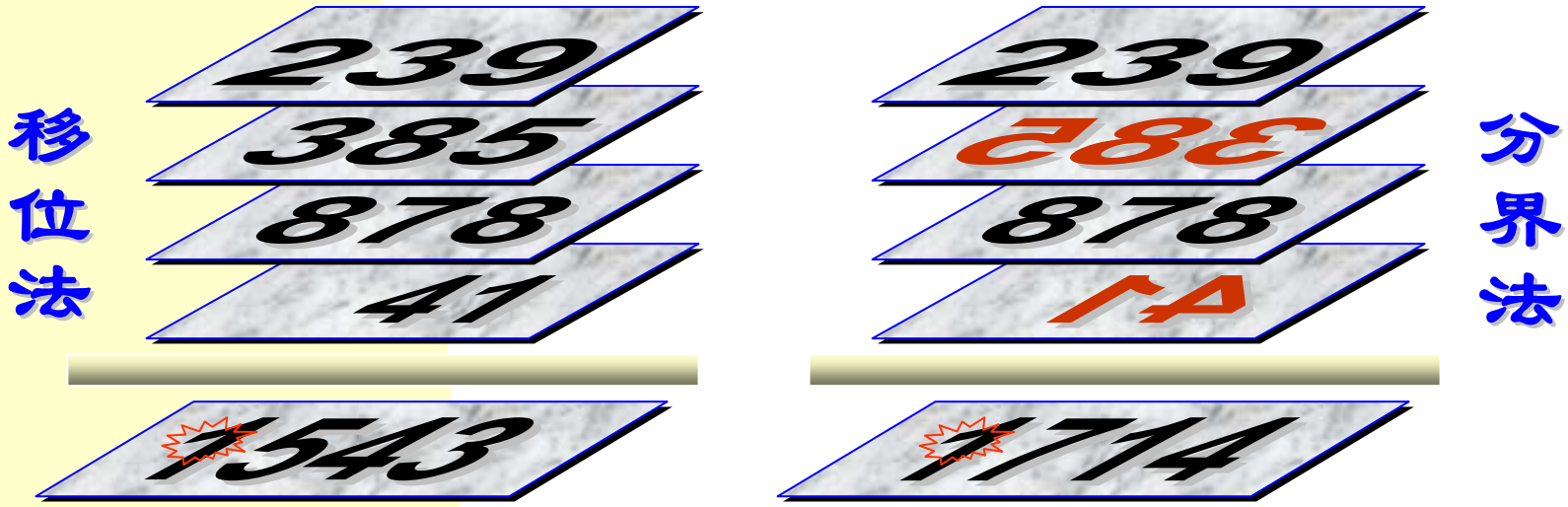
折叠法

- 此方法把关键码自左到右分成位数相等的几部分，每一部分的位数应与散列表地址位数相同，只有最后一部分的位数可以短一些。
- 把这些部分的数据叠加起来，就可以得到具有该关键码的记录的散列地址。
- 有两种叠加方法：
  - 移位法 — 把各部分的最后一位对齐相加；
  - 分界法 — 各部分不折断，沿各部分的分界来回折叠，然后对齐相加，将相加的结果当做散列地址。

- 示例：设给定的关键码为 **key = 2938587841**，若存储空间限定 3 位，则划分结果为每段 3 位. 上述关键码可划分为 4段:

  239　385　878　41

- 把超出地址位数的最高位删去，仅保留最低的3位，做为可用的散列地址。



- 一般当关键码的位数很多，而且关键码每一位上数字的分布大致比较均匀时，可用这种方法得到散列地址。
- 以上介绍了几种常用的散列函数。在实际工作中应根据关键码的特点，选用适当的方法。有人曾用“轮盘赌”的统计分析方法对它们进行了模拟分析，结论是平方取中法最接近于“随机化”。

处理溢出的闭散列方法

解决冲突的方法又称为溢出处理技术。因为任一种散列函数也不能避免产生冲突，因此选择好的解决冲突溢出的方法十分重要。

为了减少冲突，对散列表加以改造。若设散列表 HT 有 m 个地址，将其改为 m 个桶。其桶号与散列地址一一对应，第 $i\ (0 \le i < m)$ 个桶的桶号即为第 i 个散列地址。

每个桶可存放 s 个表项，这些表项的关键码互为同义词。如果对两个不同表项的关键码用散列函数计算得到同一个散列地址，就产生了冲突，它们可以放在同一个桶内的不同位置。

- 只有当桶内所有 s 个表项位置都放满表项后再加入表项才会产生溢出。
- 通常桶的大小 s 取的比较小， 因此在桶内大多采用顺序搜索。
- 闭散列就是处理溢出的一种常用的方法，也叫做开地址法。
- 在闭散列情形，所有的桶都直接放在散列表数组中。因此每个桶只有一个表项 ($s = 1$)。
- 若设散列表中各桶的编址为 0 到 $m-1$，当要加入一个表项 R_2时，用它的关键码 $R_2.key$，通过散列函数 $hash(R_2.key)$ 的计算，得到它的存放桶号 j。

- 但在存放时发现这个桶已经被另一个表项 R_1 占据，发生了冲突，必须处理溢出。为此，需把 R_2 存放到表中“下一个”空桶中。如果表未被装满，则在允许的范围内必定还有空桶。

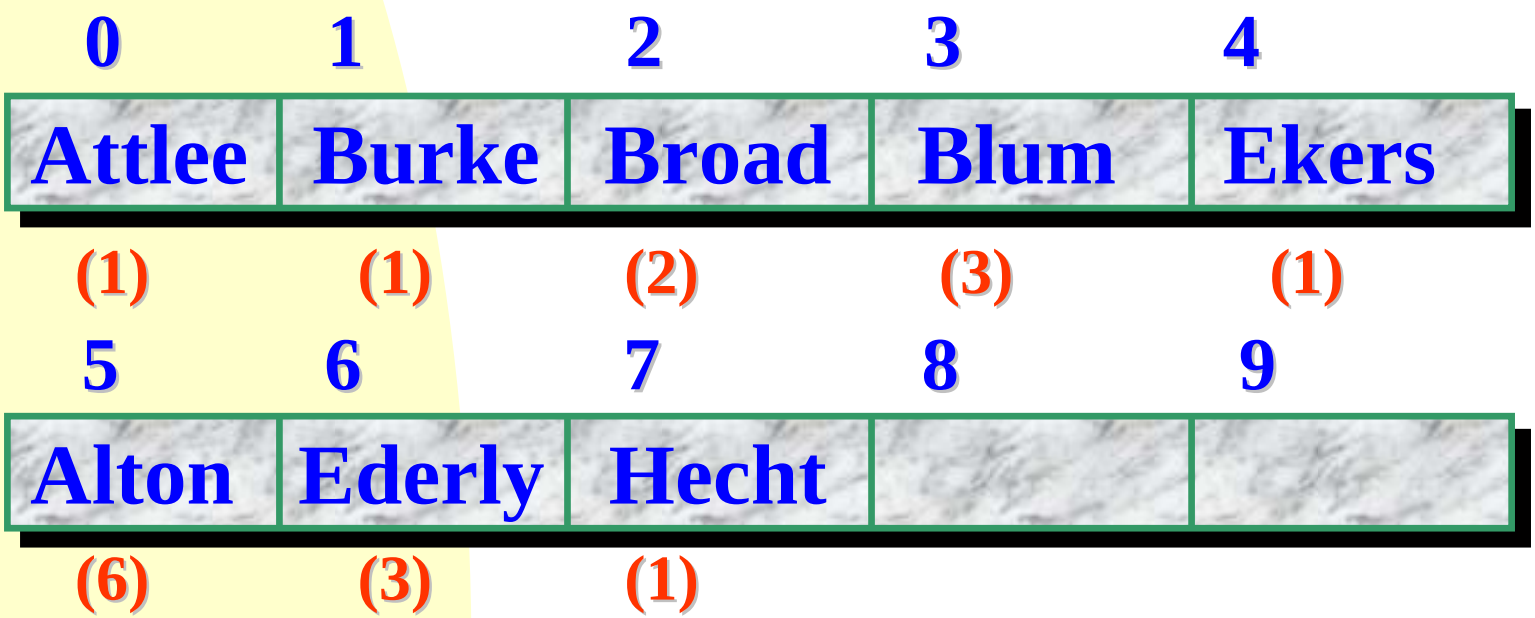
(1) 线性探查法 (Linear Probing)

假设给出一组表项，它们的关键码为 **Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly**。采用的散列函数是：取其第一个字母在字母表中的位置。

Hash (*x*) = *ord* (*x*) - *ord* (‘*A*’)

//ord ( ) 是求字符内码的函数

- 可得 *Hash* (Burke) = 1　　*Hash* (Ekers) = 4
  Hash (Broad) = 1　　*Hash* (Blum) = 1
  Hash (Attlee) = 0　　*Hash* (Hecht) = 7
  Hash (Alton) = 0　　*Hash* (Ederly) = 4
- 设散列表为*HT*[26]，*m* = 26。采用线性探查法处理溢出，则上述关键码在散列表中散列位置如图所示。



- 需要搜索或加入一个表项时，使用散列函数计算桶号：

$$H_0 = hash(key)$$

- 一旦发生冲突，在表中顺次向后寻找“下一个”空桶 H_i 的递推公式为：

$$H_i = (H_{i-1} + 1) \% m, \quad i = 1, 2, \ldots, m-1$$

即用以下的线性探查序列在表中寻找“下一个”空桶的桶号：

$$H_0+1, H_0+2, \ldots, m-1, 0, 1, 2, \ldots, H_0-1$$

亦可写成如下的通项公式：

$$H_i = (H_0 + i) \% m, \quad i = 1, 2, \ldots, m-1$$

- 当发生冲突时，探查下一个桶。当循环 ***m*−1**次后就会回到开始探查时的位置，说明待查关键码不在表内，而且表已满，不能再插入新关键码。
- 用平均搜索长度***ASL* (*Averagy Search Length*)**衡量散列方法的搜索性能。
- 根据搜索成功与否，它又有搜索成功的平均搜索长度***ASLsucc***和搜索不成功的平均搜索长度***ASLunsucc***之分。
- 搜索成功的平均搜索长度 ***ASLsucc*** 是指搜索到表中已有表项的平均探查次数。它是找到表中各个已有表项的探查次数的平均值。

- 搜索不成功的平均搜索长度 ***ASLunsucc*** 是指在表中搜索不到待查的表项，但找到插入位置的平均探查次数。它是表中所有可能散列到的位置上要插入新元素时为找到空桶的探查次数的平均值。
- 在使用线性探查法对示例进行搜索时，搜索成功的平均搜索长度为：

$$ASLsucc = \frac{1}{8}\sum_{i=1}^{8} Ci = \frac{1}{8}(1+1+2+3+1+6+3+1) = \frac{18}{8}.$$

- 搜索不成功的平均搜索长度为：

$$ASLunsucc = \frac{1}{26}\left(\sum_{i=0}^{7}(9-i)+\sum_{i=8}^{25}1\right) = \frac{9+8+7+6+5+4+3+2+18}{26} = \frac{62}{26}.$$

- 下面给出用线性探查法在散列表 ***ht*** 中搜索给定值 ***x*** 的算法。如果查到某一个 ***j***，使得 ***ht[j].info == Active*** 同时 ***ht[j].Element == x***，则搜索成功；否则搜索失败。造成失败的原因可能是表已满，或者是原来有此表项但已被删去，或者是无此表项且找到空桶。

```
class HashTable {
//用线性探查法处理溢出时散列表类的定义
public:
    enum KindOfEntry { Active, Empty, Deleted };
    HashTable ( ) : buckets ( DefaultSize )
                    { AllocateHt ( ); CurrentSize = 0; }
```

```
    ~HashTable ( ) { delete [ ] ht; }
    const HashTable & operator =
                                ( const HashTable & ht2 );
    int Find ( const Type & x );
    int Insert ( const Type & x );
    int Remove ( const Type & x );
    int IsIn ( const Type & x )
          { return ( i =  Find (x) ) >= 0 ? 1 : 0; }
    void MakeEmpty ( );
private:
    struct HashEntry {
       Type Element;
       KindOfEntry info;
```

```
        int operator== ( HashEntry &, HashEntry & );
        int operator!= ( HashEntry &, HashEntry & );
        HashEntry ( ) : info (Empty ) { }
        HashEntry (const Type & E, KindOfEntry
                        i =  Empty ) : Element (E), info (i) { }
    };
    enum { DefualtSize = 11 }
    HashEntry *ht;
    int CurrentSize, TableSize;
    void AllocateHt ( )
                        { ht = new HashEntry[TableSize ]; }
    int FindPos ( const Type & x ) const;
}
```

```
template <class Type> int HashTable<Type>::
Find ( const Type & x ) {
//线性探查法的搜索算法，函数返回找到位置。
//若返回负数可能是空位，若为-TableSize则失败。
   int i =  FindPos ( x ),  j = i;
   while ( ht[j].info != Empty && ht[j].Element != x ){
      j = ( j + 1 ) % TableSize;
      if ( j == i ) return -TableSize;
   }
   if ( ht[j].info == Active ) return j;
   else -j;
}
```

- 在利用散列表进行各种处理之前，必须首先将散列表中原有的内容清掉。只需将表中所有表项的***info***域置为***Empty***即可。
- 散列表存放的表项不应有重复的关键码。在插入新表项时，如果发现表中已经有关键码相同的表项，则不再插入。
- 在闭散列情形下不能真正删除表中已有表项。删除表项会影响其它表项的搜索。若把关键码为**Broad**的表项真正删除，把它所在位置的***info***域置为***Empty***，以后在搜索关键码为**Blum**和**Alton**的表项时就查不下去，从而会错误地判断表中没有关键码为**Blum**和**Alton**的表项。

- 若想删除一个表项，只能给它做一个删除标记***deleted***，进行逻辑删除，不能把它真正删去。
- 逻辑删除的副作用是：在执行多次删除后，表面上看起来散列表很满，实际上有许多位置没有利用。

```
template <class Type>
void HashTabe<Type> :: MakeEmpty ( ) {
//置表中所有表项为空
    for ( int i = 0; i < TableSize; i++)
        ht[i].info = Empty;
    CurrentSize = 0;
}
```

```
template <class Type> const HashTable <Type>
& HashTable<Type> ::
operator = ( const HashTable<Type> &ht2 ) {
//重载函数：从散列表ht2复制到当前散列表
    if ( this != &ht2 ) {
       delete [ ] ht;
       TableSize = ht2.TableSize;    AllocateHt ( );
       for ( int i = 0; i < TableSize; i++ )
            ht[i] = ht2.ht[i];
       CurrentSize = ht2.CurrentSize;
    }
    return *this;
}
```

```
template <class Type> int HashTable<Type>::
Insert (const Type & x ) {
//将新表项 x 插入到当前的散列表中
    if ( ( int i = Find (x) ) >= 0 ) return 0;    //不插入
    else if ( i != -TableSize && ht[-i].info != Active )
          {       //在 -i 处插入表项x
             ht[-i].Element = x;   ht[-i].info = Active;
             CurrentSize++;
             return 1;
          }
          else return 0;
}
```

```
template <class Type> int HashTable<Type> ::
Remove ( const Type & x ) {
//在当前散列表中删除表项x
    if ( ( int i =  Find (x) ) >= 0 ) {     //找到,删除
       ht[i].info = deleted;                //做删除标记
       CurrentSize--;
       return 1;
    }
    else return 0;
}
```

线性探查方法容易产生“堆积”，不同探查序列的关键码占据可用的空桶，为寻找某一关键码需要经历不同的探查序列，导致搜索时间增加。

算法分析

- 设散列表的装填因子为 $\alpha = n / (s*m)$，其中 n 是表中已有的表项个数，s 是每个桶中最多可容纳表项个数，m 是表中的桶数。
- 可用 α 表明散列表的装满程度。α 越大，表中表项数越多，表装得越满，发生冲突可能性越大。
- 通过对线性探查法的分析可知，为搜索一个关键码所需进行的探查次数的期望值 P 大约是 $(2-\alpha)/(2-2\alpha)$。虽然平均探查次数较小，但在最坏情况下的探查次数会相当大。

(2) 二次探查法 (quadratic probing)

- 为改善“堆积”问题，减少为完成搜索所需的平均探查次数，可使用二次探查法。
- 通过某一个散列函数对表项的关键码 x 进行计算，得到桶号，它是一个非负整数。

$$H_0 = hash(x)$$

- 二次探查法在表中寻找“下一个”空桶的公式为：

$$H_i = (H_0 + i^2) \% m,$$

$$H_i = (H_0 - i^2) \% m, \quad i = 1, 2, \ldots, (m-1)/2$$

- 式中的 m 是表的大小，它应是一个值为 $4k+3$ 的质数，其中 k 是一个整数。这样的质数如 3, 7, 11, 19, 23, 31, 43, 59, 127, 251, 503, 1019, ….

- 探查序列形如 $H_0, H_0+1, H_0-1, H_0+4, H_0-4, \ldots$。
- 在做 $(H_0 - i^2) \% m$ 的运算时，当 $H_0 - i^2 < 0$ 时，运算结果也是负数。实际算式可改为

  $j = (H_0 - i^2) \% m$, **while** $(j < 0)\ j += m$
- 示例：给出一组关键码 { **Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly** }。

  散列函数为：***Hash*** **(*x*) = *ord* (*x*) − *ord* ('*A*')**

  用它计算可得

  Hash **(Burke) = 1**　　***Hash*** **(Ekers) = 4**

  Hash **(Broad) = 1**　　***Hash*** **(Blum) = 1**

  Hash **(Attlee) = 0**　　***Hash*** **(Hecht) = 7**

  Hash **(Alton) = 0**　　***Hash*** **(Ederly) = 4**

- 因为可能桶号是 0 ~ 25, 取满足 $4k+3$ 的质数，表的长度为 *TableSize* = 31，利用二次探查法得到的散列结果如图所示。

利用二次探查法处理溢出

- 使用二次探查法处理溢出时的搜索成功的平均搜索长度为:

$$ASLsucc=\frac{1}{8}\sum_{i=1}^{8}Ci=\frac{1}{8}(3+1+2+1+2+1+5+3)=\frac{18}{8}.$$

- 搜索不成功的平均搜索长度为:

$$ASLunsucc=\frac{1}{26}(6+5+2+3+2+2+2+3+18)=\frac{43}{26}$$

适用于二次探查散列表类的定义

```
template <class Type> class HashTable {
public:
    enum KindOfEntry { Active, Empty, Deleted };
    const int &Find ( const Type & x );
```

```
    int IsEmpty ( ) { return !CurrentSize ? 1 : 0; }
    int IsFull ( )
            { return CurrentSize == TableSize ? 1 : 0; }
    int WasFound ( ) const { return LastFindOK; }
    ……
private:
    struct HashEntry {
       Type Element;
       KindOfEntry info;
       HashEntry ( ) : info (Empty ) { }
       HashEntry ( const Type &E, KindOfEntry i =
                    Empty ) : Element (E), info (i) { }
    };
```

```
    enum { DefualtSize = 31; }
    HashEntry *ht;
    int TableSize;
    int CurrentSize;
    int LastFindOK;
    void AllocateHt ( );
    int FindPos ( const Type & x );
};
```

设散列表桶数为 m，待查表项关键码为 x，第一次通过散列函数计算出来的桶号为 $H_0=hash(x)$。当发生冲突时，第 $i-1$ 次和第 i 次计算出来的“下一个”桶号分别为：

$$H_{i-1}^{(0)} = (H_0 + (i-1)^2) \% m,$$

$$H_{i-1}^{(1)} = (H_0 - (i-1)^2) \% m.$$

$$H_i^{(0)} = (H_0 + i^2) \% m,$$

$$H_i^{(1)} = (H_0 - i^2) \% m.$$

相减，可以得到：

$$H_i^{(0)} - H_{i-1}^{(0)} = (2 * i - 1) \% m,$$

$$H_i^{(1)} - H_{i-1}^{(1)} = (-2 * i + 1) \% m.$$

从而

$$H_i^{(0)} = (H_{i-1}^{(0)} + 2 * i - 1) \% m,$$

$$H_i^{(1)} = (H_{i-1}^{(1)} - 2 * i + 1) \% m.$$

- 只要知道上一次的桶号 $H_{i-1}^{(0)}$和 $H_{i-1}^{(1)}$，当 i 增加 1 时可以从 $H_{i-1}^{(0)}$和 $H_{i-1}^{(1)}$简单地导出 $H_i^{(0)}$和 $H_i^{(1)}$不需要每次计算 i 的平方。
- 在溢出处理算法 *Find* 中，首先求出 H_0作为当前桶号 *CurrentPos*，当发生冲突时求“下一个”桶号，$i = 1$。
- 此时用一个标志 *odd* 控制是加 i^2还是减 i^2。
  - 若 *odd* == 0 加 i^2，并置 *odd* = 1;
  - 若 *odd* == 1 减 i^2，并置 *odd* = 0。
- 下次 i 进一后，又可由 *odd* 控制先加后减。

处理溢出的算法

```
template <class Type> int HashTable<Type>::
Find ( const Type & x ) {
    int i = 0,  odd = 0 ;
    int CurrentPos = HashPos ( x );                    //桶号
    while ( ht[CurrentPos].info != Empty &&
                    ht[CurrentPos].Element != x ) {  //冲突
        if ( !odd ) {                          // odd == 0 加 i²
            CurrentPos += 2*++i −1;   odd = 1;
            while ( CurrentPos >= TableSize )
                CurrentPos −= TableSize;
        }
```

```
else {                                  // odd == 1 减 i²
        CurrentPos −= 2*i−1;   odd = 0;
        while ( CurrentPos < 0 )
            CurrentPos += TableSize;
    }
  }
  LastFindOK = ht[CurrentPos].info == Active;
  return CurrentPos;
}
```

可以证明，当表的长度***TableSize***为质数且表的装填因子 α 不超过 **0.5** 时，新的表项 ***x*** 一定能够插入，而且任何一个位置不会被探查两次。因此，只要表中至少有一半空的，就不会有表满问

- 在搜索时可以不考虑表装满的情况；但在插入时必须确保表的装填因子α不超过**0.5**。如果超出，必须将表长度扩充一倍，进行表的分裂。
- 在删除一个表项时，为确保搜索链不致中断，也只能做表项的逻辑删除，即将被删表项的标记***info***改为***Deleted***。

二次散列的插入操作

```
template <class Type> int HashTable<Type>::
Insert ( const Type & x ) {
    int CurrentPos = Find (x);
    if (LastFindOK ) return 0;
```

```
    ht[CurrentPos] = HashEntry ( x, Active );
    if ( ++CurrentSize < TableSize/2) return 1;
    HashEntry *Oldht = ht;
    int OldTableSize = TableSize;
    CurrentSize = 0;
    TableSize = NextPrime ( 2 * OldTableSize );
    Allocateht ( );
    for ( i = 0; i < OldTableSize; i++)
        if ( Oldht[i].info == Active )
            Insert ( Oldht[i].Element );
    delete [ ] Oldht;
    return 1;
}
```

二次散列的删除和判存在操作

```
template <class Type> int HashTable<Type> ::
IsIn ( const Type & x ) {
    int CurrentPos = Find ( x );
    return LastFindOK.;
}

template <class Type> int HashTable<Type> ::
Remove ( const Type & x ) {
    int CurrentPos = Find (x);
    if (!LastFindOK ) return 0;
    ht[CurrentPos].info = Deleted;  return 1;
}
```

(3) 双散列法

- 使用双散列方法时，需要两个散列函数。
- 第一个散列函数 ***Hash( )*** 按表项的关键码 ***key*** 计算表项所在的桶号 $H_0 = Hash(key)$。
- 一旦冲突，利用第二个散列函数 ***ReHash( )*** 计算该表项到达“下一个”桶的移位量。它的取值与 ***key*** 的值有关，要求它的取值应当是小于地址空间大小 ***TableSize***，且与 ***TableSize*** 互质的正整数。
- 若设表的长度为 m = ***TableSize***，则在表中寻找“下一个”桶的公式为：

$j = H_0 = Hash(key)$，　$p = ReHash(key)$;

$j = (j + p) \% m$;

p是小于m且与m互质的整数

- 利用双散列法，按一定的距离，跳跃式地寻找“下一个”桶，减少了“堆积”的机会。
- 双散列法的探查序列也可写成:

$$H_i = (H_0 + i * ReHash(key)) \% m,$$

$$i = 1, 2, \ldots, m-1$$

- 最多经过$m-1$次探查，它会遍历表中所有位置，回到H_0位置。

- 示例：给出一组表项关键码{ 22, 41, 53, 46, 30, 13, 01, 67 }。散列函数为：

  $Hash(x) = (3x) \% 11$。

- 散列表为 $HT[0..10]$，$m = 11$。因此，再散列函数为 $ReHash(x) = (7x) \% 10 + 1$。

  $H_i = (H_{i-1} + (7x) \% 10 + 1) \% 11,\ i = 1, 2, \ldots$

- $H_0(22) = 0$　$H_0(41) = 2$　$H_0(53) = 5$

  $H_0(46) = 6$　$H_0(30) = 2$ 冲突 $H_1 = (2+1) = 3$

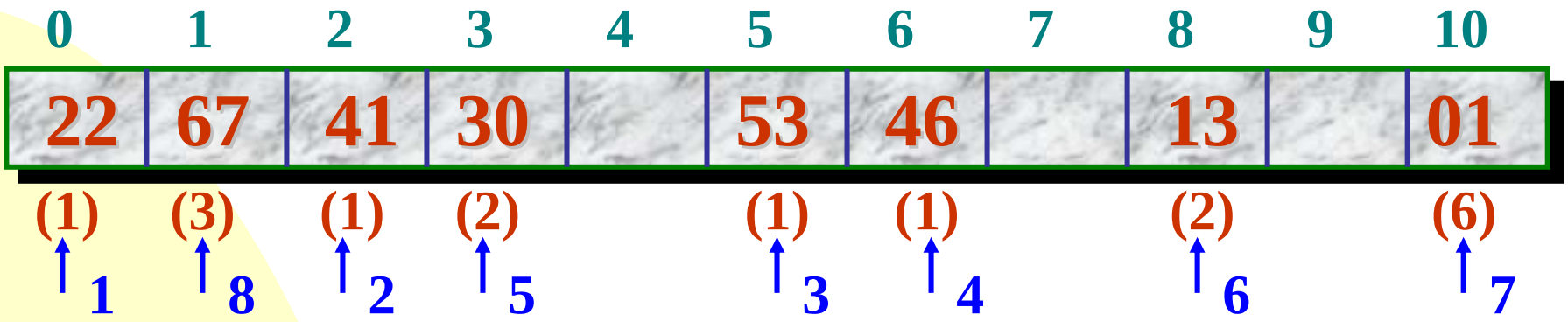
  $H_0(13) = 6$ 冲突 $H_1 = (6+2) = 8$

  $H_0(01) = 3$ 冲突 $H_1 = (3+8) = 0$

  $H_2 = (0+8) = 8$　$H_3 = (8+8) = 5$

  $H_4 = (5+8) = 2$　$H_5 = (2+8) = 10$

$H_0(67) = 3$ 冲突 $H_1 = (3+10) = 2$ $H_2 = (2+10) = 1$



- 搜索成功的平均搜索长度

$$ASLsucc = \frac{1}{8}(1+3+1+2+1+1+2+6) = \frac{17}{8}$$

- 搜索不成功的平均搜索长度
  - 每一散列位置的移位量有10种：1, 2, …, 10。先计算每一散列位置各种移位量情形下找到下一个空位的比较次数，求出平均值；
  - 再计算各个位置的平均比较次数的总平均值。

- *Rehash( )*的取法很多。例如，当*m*是质数时，可定义
  - $ReHash(key) = key \% (m-2) + 1$
  - $ReHash(key) = \lfloor key / m \rfloor \% (m-2) + 1$
- 当 m 是 2 的方幂时，$ReHash(key)$ 可取从 0 到 $m-1$ 中的任意一个奇数。

处理溢出的开散列方法 — 链地址法

- 开散列方法首先对关键码集合用某一个散列函数计算它们的存放位置。
- 若设散列表地址空间的所有位置是从**0**到***m*-1**，则关键码集合中的所有关键码被划分为***m***个子集，具有相同地址的关键码归于同一子集。我们称同一子集中的关键码互为同义词。每一个子集称为一个桶。
- 通常各个桶中的表项通过一个单链表链接起来，称之为同义词子表。所有桶号相同的表项都链接在同一个同义词子表中，各链表的表头结点组成一个向量。

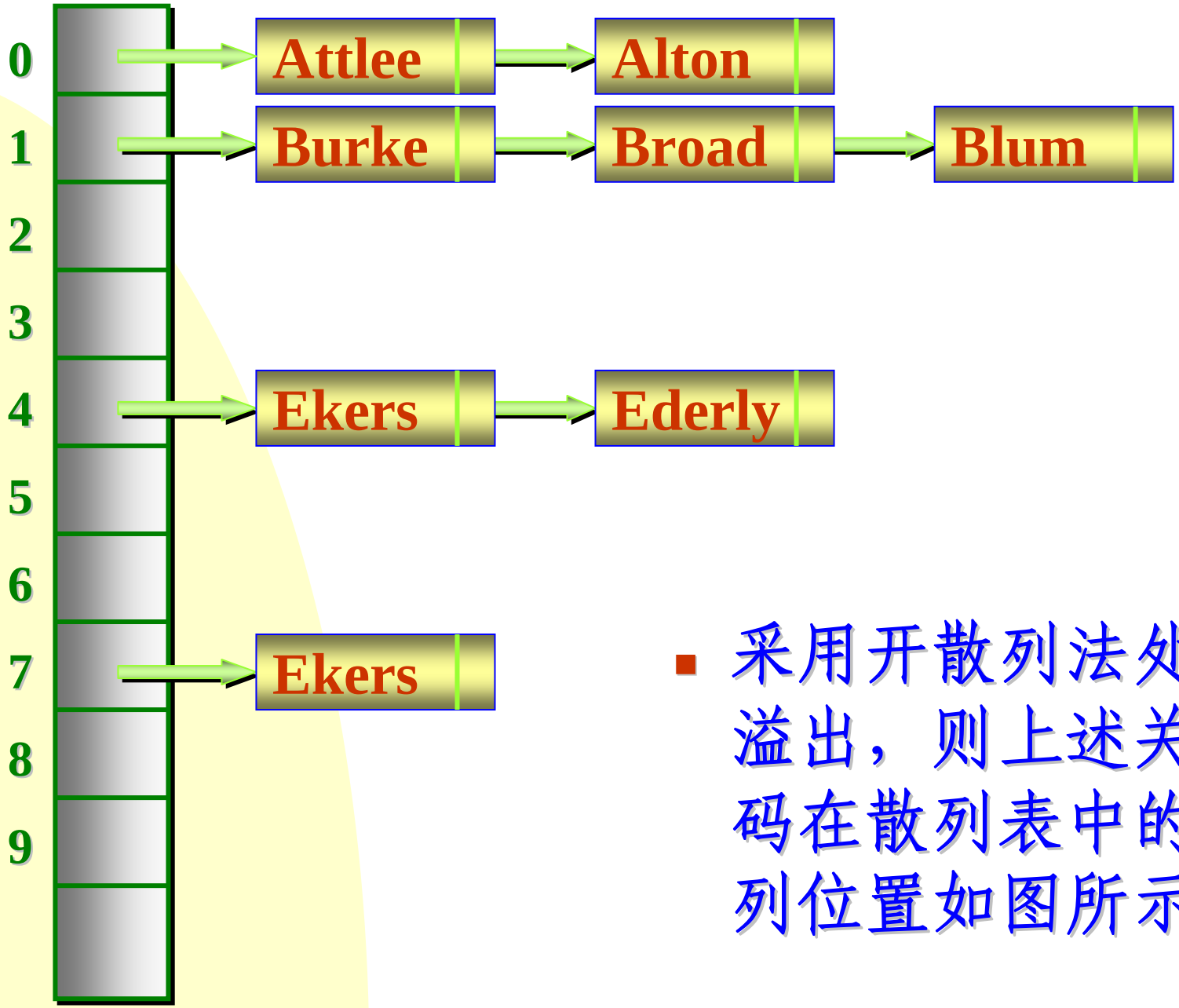
- 向量的元素个数与桶数一致。桶号为 *i* 的同义词子表的表头结点是向量中的第 *i* 个元素。
- 示例：给出一组表项关键码{ **Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly** }。散列函数为：$Hash(x) = ord(x) - ord('A')$。
- 用它计算可得：

  Hash **(Burke) = 1**　　***Hash*** **(Ekers) = 4**

  Hash **(Broad) = 1**　　***Hash*** **(Blum) = 1**

  Hash **(Attlee) = 0**　　***Hash*** **(Hecht) = 7**

  Hash **(Alton) = 0**　　***Hash*** **(Ederly) = 4**
- 散列表为 ***HT*****[0..25]**，***m*** **= 26**。



- 采用开散列法处理溢出，则上述关键码在散列表中的散列位置如图所示。

- 通常，每个桶中的同义词子表都很短，设有***n***个关键码通过某一个散列函数，存放到散列表中的 ***m*** 个桶中。那么每一个桶中的同义词子表的平均长度为 ***n / m***。这样，以搜索平均长度为 ***n / m*** 的同义词子表代替了搜索长度为 ***n*** 的顺序表，搜索速度快得多。

利用链地址法处理溢出时的类定义

```
template <class Type> class ListNode {  //链表结点
friend HashTable;
private:
    Type key;                  //关键码
    ListNode *link;            //链指针
```

```
};

typedef ListNode<Type> *ListPtr;

class HashTable {                    //散列表的类定义
public:
    HashTable( int size = defaultsize )
        { buckets = size;  ht = new ListPtr[buckets]; }
private:
    int buckets;                     //桶数
    ListPtr<Type> *ht;               //散列表数组的头指针
};
```

循链搜索的算法

```
template <class Type>
Type *HashTable<Type>:: Find ( const Type & x,)
{
    int j = HashFunc ( x, buckets );
    ListPtr<Type> *p = ht[j];
    while ( p != NULL )
        if ( p→key == x ) return & p→key;
        else p = p→link;
    return 0;
}
```

其它如插入、删除操作可参照单链表的插入、删除等算法来实现。

- 应用链地址法处理溢出，需要增设链接指针，似乎增加了存储开销。事实上，由于开地址法必须保持大量的空闲空间以确保搜索效率，如二次探查法要求装填因子$\alpha \leq 0.5$，而表项所占空间又比指针大得多，所以使用链地址法反而比开地址法节省存储空间。

散列表分析

- 散列表是一种直接计算记录存放地址的方法，它在关键码与存储位置之间直接建立了映象。
- 当选择的散列函数能够得到均匀的地址分布时，在搜索过程中可以不做多次探查。

- 由于很难避免冲突，就增加了搜索时间。冲突的出现，与散列函数的选取 (地址分布是否均匀)，处理溢出的方法 (是否产生堆积) 有关。
- 在实际应用中，在使用关键码进行散列时，如在用作关键码的许多标识符具有相同的前缀或后缀，或者是相同字符的不同排列的场合，不同的散列函数往往导致散列表具有不同的搜索性能。
- 下图给出一些实验结果，列出在采用不同的散列函数和不同的处理溢出的方法时，搜索关键码所需的对桶访问的平均次数。实验数据为 **{ 33575, 24050, 4909, 3072, 2241, 930, 762, 500 }**。

$\alpha = n/m$	0.50		0.75		0.90		0.95	
散列函数种类	开散列法	闭散列法	开散列法	闭散列法	开散列法	闭散列法	开散列法	闭散列法
平方取中	1.26	1.73	1.40	9.75	1.45	310.14	1.47	310.53
除留余数	1.19	4.52	1.31	10.20	1.38	22.42	1.41	25.79
移位折叠	1.33	21.75	1.48	65.10	1.40	710.01	1.51	118.57
分界折叠	1.39	22.97	1.57	48.70	1.55	69.63	1.51	910.56
数字分析	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
理论值	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

搜索关键码时所需对桶的平均访问次数

从图中可以看出，开散列法优于闭散列法；在散列函数中，用除留余数法作散列函数优于其它类型的散列函数，最差的是折叠法。

- 当装填因子 α 较高时，选择的散列函数不同，散列表的搜索性能差别很大。在一般情况下多选用除留余数法，其中的除数在实用上应选择不含有**20**以下的质因数的质数。
- 对散列表技术进行的实验评估表明，它具有很好的平均性能，优于一些传统的技术，如平衡树。
- 但散列表在最坏情况下性能很不好。如果对一个有 ***n*** 个关键码的散列表执行一次搜索或插入操作，最坏情况下需要 **O(*n*)** 的时间。
- **Knuth**对不同的溢出处理方法进行了概率分析。

- 若设 α 是散列表的装填因子:

$$\alpha = \frac{\text{表中已装有记录的桶数}}{\text{表中预设的最大桶数}} = \frac{n}{m}$$

- 用地址分布均匀的散列函数*Hash*( )计算桶号。
- S_n是搜索一个随机选择的关键码 x_i $(1 \le i \le n)$ 所需的关键码比较次数的期望值
- U_n是在长度为 m 的散列表中 n 个桶已装入表项的情况下，装入第 $n+1$ 项所需执行的关键码比较次数期望值。
- 前者称为在 $\alpha = n / m$ 时的搜索成功的平均搜索长度，后者称为在$\alpha = n / m$时的搜索不成功的平均搜索长度。

- 用不同的方法溢出处理冲突时散列表的平均搜索长度如图所示。

处理溢出的方法		平均搜索长度 ASL	
		搜索成功 Sn	搜索不成功 Un (登入新记录)
闭散列法	线性探查法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
	伪随机探查法 二次探查法 双散列法	$-\left(\frac{1}{\alpha}\right)\log_e(1-\alpha)$	$\frac{1}{1-\alpha}$
链地址法 (同义词子表法)		$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}\approx\alpha$

- 散列表的装填因子 α 表明了表中的装满程度。越大，说明表越满，再插入新元素时发生冲突的可能性就越大。
- 散列表的搜索性能，即平均搜索长度依赖于散列表的装填因子，不直接依赖于 ***n*** 或 ***m***。
- 不论表的长度有多大，我们总能选择一个合适的装填因子，以把平均搜索长度限制在一定范围内。

小结 需要复习的知识点

- 掌握：静态索引结构，包括线性索引、倒排索引、静态索引树的搜索和构造方法。
- 掌握：动态索引结构，包括B_树、B+树的搜索；B_树的插入和删除方法； B+树的插入与删除方法。
- 掌握：散列表与散列方法，包括散列函数的构造、处理溢出的闭散列方法；处理溢出的开散列方法；散列表分析。

- B_树的定义、平衡 *m* 路搜索树与 *m* 阶B_树的关系
- 有 *n* 个关键码的 *m* 阶B_树的最大高度和最小高度
  - 最大高度为 $\lfloor \log_{\lceil m/2 \rceil}((n+1)/2) \rfloor + 1$
  - 最小高度为 $\lceil \log_m(n+1) \rceil$

    (包括失败结点所在层次)
- B_树的插入 (包括结点分裂)、删除 (包括结点调整与合并)方法
- 散列表中散列函数的设计原则及除留余数法的设计 (注意除数的选择)

- 解决地址冲突的线性探查法的运用，平均探查次数计算
- 线性探查法的删除问题、散列表中必须为各地址设置的三个状态 ***Active, Empty, Deleted***
  - 解决地址冲突的双散列法的运用，平均探查次数的计算
  - 双散列法中再散列函数的设计：要求 计算结果与表长 ***m*** 互质 **(*m* 设计为质数)**
  - 解决地址冲突的二次散列法的运用，平均探查次数计算
  - 二次散列法中装填因子 α 与表长 ***m*** 的设置： α 不大于**0.5, *m***为满足 **4*j* + 3** 的质数

例 求散列表大小并设计散列函数

- 设有一个含200个表项的散列表，用二次探查法解决冲突，按关键码查询时找到一个新表项插入位置的平均探查次数不超过1.5，则散列表项应能够至少容纳多少个表项。再设计散列函数（设搜索不成功的平均搜索长度为 $U_n = 1 / (1 - \alpha)$，其中 α 为装填因子）
- 解答：设表中表项个数 $n = 200$，搜索不成功的平均搜索长度

$$U_n = 1 / (1 - \alpha) \leq 1.5 \Rightarrow \alpha \leq 1/3$$

$$\therefore n / m = 200 / m = \alpha \leq 1/3 \qquad m \geq 600$$

- 又 二次散列要求 m 必须是满足 $4j+3$的质数，故 m 可以取 607
- 散列函数 $hash(key) = key \% m$