第十章 文件、外部排序与外部搜索

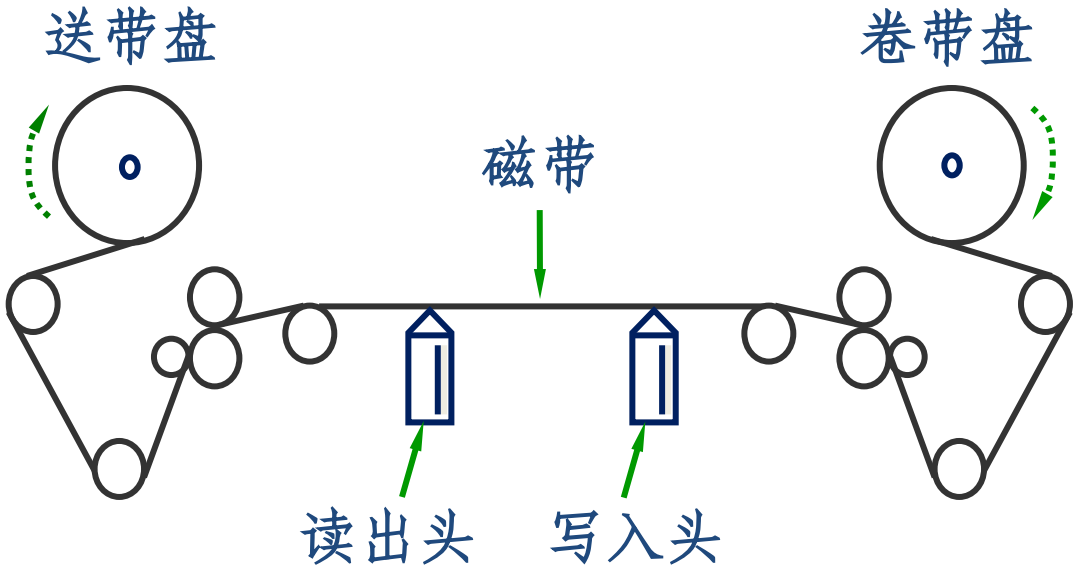
- 主存储器和外存储器
- 文件组织
- 多级索引结构
- 外排序

主存储器与外存储器

- 主存储器又叫内存储器，简称为内存；外存储器简称为外存。
- 外存储器与内存储器相比，优点是：
  - 价格较低
  - 永久的存储能力
- 缺点：
  - 访问外存储器上的数据比访问内存要慢5～6个数量级
- 要求我们在开发系统时必须考虑如何使外存访问次数达到最少。

磁带（tape）

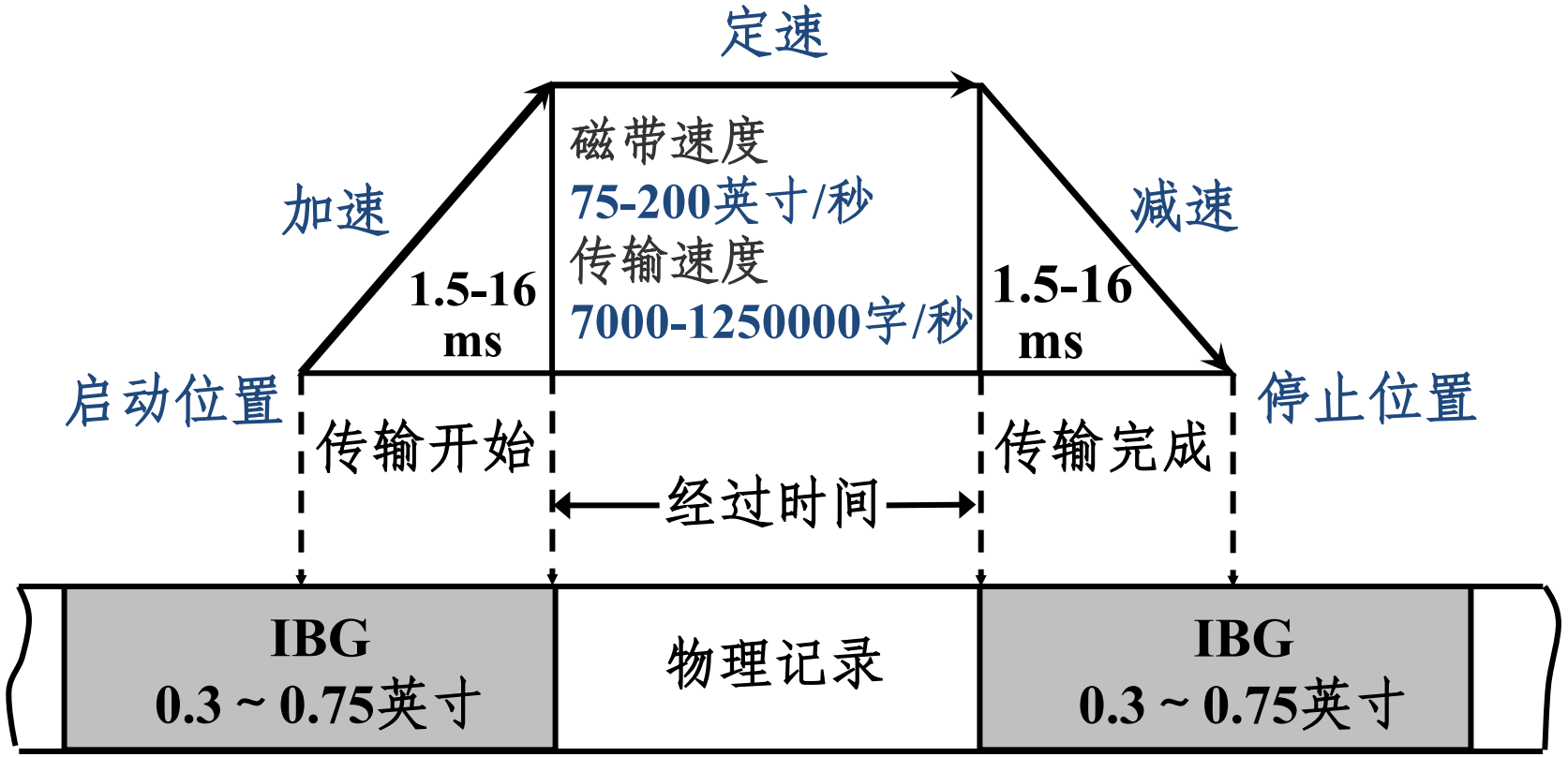
- 磁带是一种顺序存取设备。
- 磁带主要用于备份、存储不经常使用的数据，以及作为将数据从一个系统转移到另一个系统的脱机介质。



- 磁带卷在一个卷盘上，运行时磁带经过读写磁头，把磁带上的信息读入计算机，或者把计算机中的信息写到磁带上去。
- 数据记录在磁带带面上。在带面上并列存放有9个磁道的信息，即每一横排有9位二进制信息：8位数据加1位奇偶校验位。
- 磁带的存储密度用BPI（Bit Per Inch）为单位，典型的存储密度有3种：6250BPI（=246排/mm）、1600BPI（=64排/mm）、800BPI（32排/mm）。正常走带速度为3～5m/Sec，因设备而异。

- 数据的传送速度＝存储密度×走带速度。
- 在应用中使用文件进行数据处理的基本单位叫做逻辑记录，简称为记录；在磁带上物理地存储的记录叫做物理记录。
- 在使用磁带或磁盘存放逻辑记录时，常常把若干个逻辑记录打包进行存放，把这个过程叫做“块化”（**blocking**）。经过块化处理的物理记录叫做块化记录。
- 磁带设备是一种启停设备。磁带每次启停都有一个加速与减速的过程，在这段时间内走带不

稳定，只能走空带，这段空带叫做记录间间隙IRG（Inter Record Gap）或者块间间隙IBG（Inter Block Gap），其长度因设备而异。



- 如果每个逻辑记录是80个字符，IRG为0.75英寸，则对存储密度为1600BPI的磁带，一个逻辑记录仅占80/1600＝0.05英寸。每传输一个逻辑记录磁带走过0.05英寸，接着磁带要走过一个IRG占0.75英寸。结果大部分时间都花费在走空带上，存储利用率只有1/16。
- 如果将若干逻辑记录存放于一个块，将IRG变成IBG，可以提高存储利用率。例如，将50个有80个字符的逻辑记录放在一个块内，此块的长度将达到50×80/1600＝2.5英寸，存储利用率达到0.77。因此在磁带上采用按块读写。

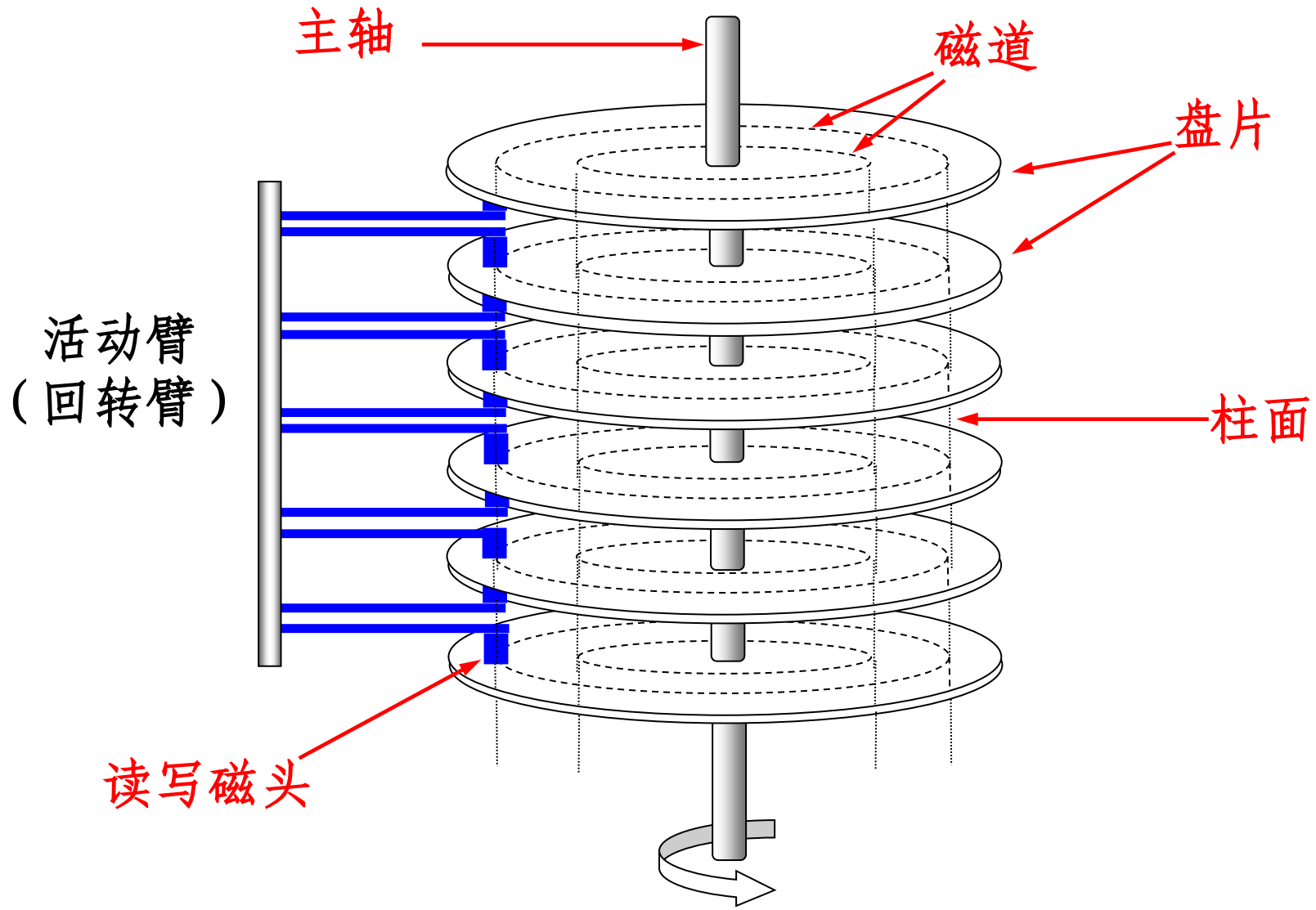
- 在磁带设备上读写一块信息所用时间

$$t_{IO} = t_a + t_b$$

- 其中，t_a是延迟时间，即读写磁头到达待读写块开始位置所需花费的时间，它与当前读写磁头所在位置有关。t_b是对一个块进行读写所用时间，它等于数据传输时间加上IBG时间。
- 磁带设备只能用于处理变化少，只进行顺序存取的大量数据。

磁盘（disc）

- 磁盘存储器通常称为直接存取设备，或随机存取设备，它访问外存上文件的任一记录的时间几乎相同。
- 磁盘存储器可以顺序存取，也可以随机存取。
- 目前使用较多的是活动臂硬盘组：若干盘片构成磁盘组，它们安装在主轴上，在驱动装置的控制下高速旋转。除了最上面一个盘片和最下面一个盘片的外侧盘面不用以外，其他每个盘片上下两面都可存放数据。将这些可存放数据的盘面称为记录盘面。


主轴
磁道
盘片
活动臂
（回转臂）
柱面
读写磁头

- 每个记录盘面上有很多磁道，数据就存放在这些磁道上。它们在记录盘面上形成一个个同心圆。
- 每个记录盘面都有一个读写磁头。所有记录盘面的读写磁头都安装在同一个动臂上，随动臂向内或向外做径向移动，从一个磁道移到另一个磁道。
- 任一时刻，所有记录盘面的读写磁头停留在各个记录盘面的半径相同的磁道上。运行时，由于盘面做高速旋转，磁头所在的磁道上的数据相继在磁头下，从而可以读写数据 。

- 各个记录盘面上半径相同的磁道合在一起称为柱面。动臂的移动实际上是将磁头从一个柱面移动到另一个柱面上。
- 一个磁道可以划分为若干段，称为扇区，一个扇区就是一次读写的最小数据量。这样，对磁盘存储器来说，从大到小的存储单位是：盘片组、柱面、磁道和扇区。
- 对磁盘存储器进行一次存取所需时间：

1. 当有多个盘片组时，要选定某个盘片组。这是由电子线路实现的，速度很快。

2. 选定盘片组后再选定某个柱面，并移动动臂把磁头移到此柱面上。这是机械动作，速度较慢。这称为“寻查（seek）”。
3. 选定柱面后，要进一步确定磁道，即确定由哪个读写磁头读写，由电子线路实现。
4. 确定磁道后，还要确定所要读写数据在磁盘上的位置（如在哪一个扇区）。这实际上就是在等待要读写的扇区转到读写磁头下面。这是机械动作。这段时间一般称为旋转延迟（rotational delay）时间。
5. 真正进行读写时间。

- 在磁盘组上一次读写的时间主要为：

$$t_{io} = t_{seek} + t_{latency} + t_{rw}$$

- 其中，t_{seek}是平均寻查时间，是把磁头定位到要求柱面所需时间，这个时间的长短取决于磁头移过的柱面数。$t_{latency}$是平均等待时间，是将磁头定位到指定块所需时间。t_{rw}是传送一个扇区数据所需的时间。
- 在MS-DOS系统中，多个扇区集结成组，称为簇。簇是文件分配的最小单位，其大小由操作系统决定。在UNIX系统中不使用簇，文件分配的最小单位和读写的最小单位是一个扇区，称为一个块（block）。

- 磁盘一次读写操作访问一个扇区，称为访问“一页”（page）或“一块”（block），又称为“一次访外”。

缓冲区（buffer）

- 为了实施磁盘读写操作，在内存中需要开辟一些区域，用以存放需要从磁盘读入的信息，或存放需要写出的信息。这些内存区域称为缓冲区。多数操作系统至少设置两个缓冲区，一个为输入缓冲区，一个为输出缓冲区。

- 例如，在从磁盘向内存读入一个扇区的数据时，数据被存放到输入缓冲区，如果下次需要读入同一个扇区的数据，就可以直接从缓冲区中读取数据，不需要重新读盘。
- 缓冲区大小应与操作系统一次读写的块的大小相适应，这样可以通过操作系统一次读写把信息全部存入缓冲区中，或把缓冲区中的信息全部写出到磁盘。
- 如果缓冲区大小与磁盘上的块大小不适配，就会造成存储空间的浪费。
- 缓冲区的构造可以看作一个先进先出的队列。

缓冲区的定义及其操作

```
#include <iostream.h>
#include <assert.h>
const int DefaultSize = 2048;
template <class T>
struct buffer {
    T *data;                          //缓冲区数组
    int current, maxSize;   //当前指针, 缓冲区容量
    buffer (int sz = DefaultSize) : maxSize(sz), current(0)
      { data = new T[sz];  assert (data != NULL); }
    ~buffer() { delete []data; }
```

```
    void OutputInfo (ostream& out, T x);   //缓冲区输出
    void InputInfo (istream& in, T& x);    //缓冲区输入
};

template <class T>
void buffer<T>::OutputInfo (ostream& out, T x) {
    if (current == maxSize) {
        for (int i = 0; i < maxSize; i++) out << data[i];
        current = 0;
    }
    data[current] = x;  current++;
};
```

```
template <class T>
void buffer<T>::InputInfo (istream& in, T& x) {
    if (current < maxSize) {
        x = data[current];
        current++;
    }
    else {
        for (int i = 0; i < maxSize; i++) in >> data[i];
        current = 0;
    }
};
```

文件组织

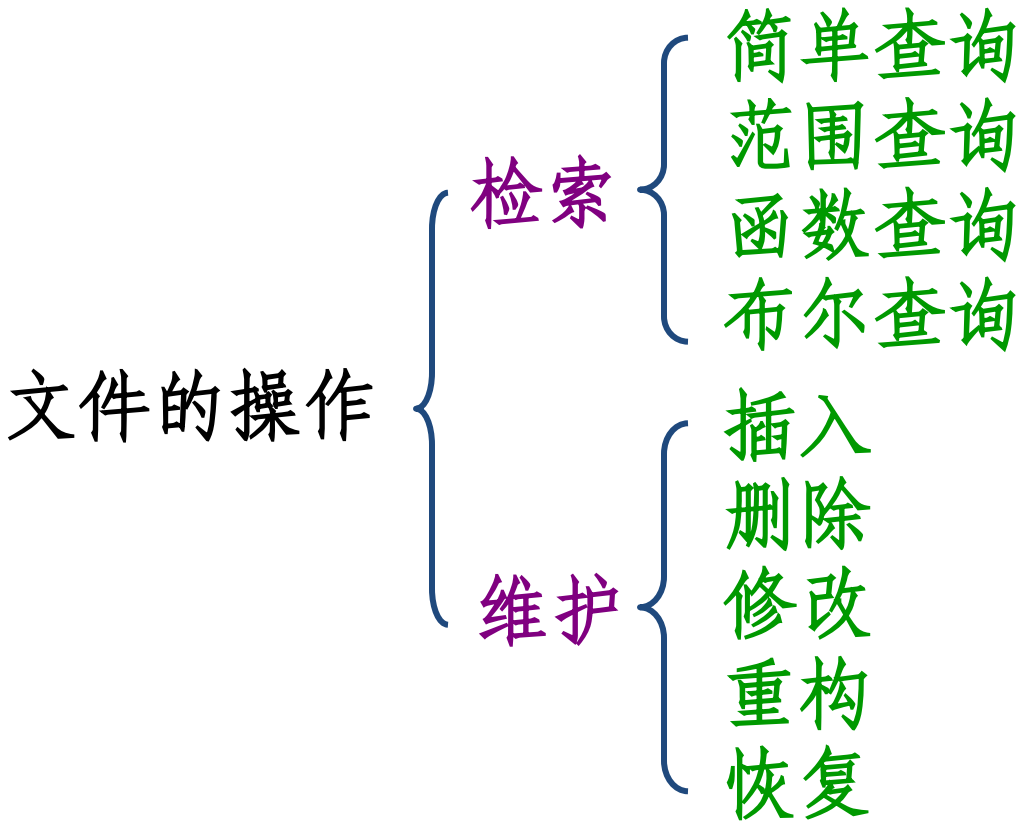
文件的基本概念

- 什么是文件
  - 文件是存储在外存上的数据结构，一般是在逻辑上具有完整意义的一组相关信息项的有序序列。
  - 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。
- 记录是文件存取的基本单位，数据项是文件可使用的最小单位。
- 从不同的观点，文件记录分为逻辑记录和物理记录。前者是面向用户的基本存取单位，后者是面向外设的基本存取单位。
- 能够唯一标识一个记录的数据项或数据项集称为主关键码项，其值称为主关键码；
- 不唯一标识一个记录的数据项或数据项集称为次关键码项，其值称为次关键码。

- 文件结构包括文件的逻辑结构、文件的存储结构和文件的操作。
- 文件的逻辑结构是线性结构，各个记录以线性方式排列。
- 文件的存储结构是指文件在外存上的组织方式，它与文件特性有关。
  - 顺序组织
  - 直接存取组织（散列组织）
  - 索引组织
- 文件的操作是定义在逻辑结构上的，但操作的具体实现要在存储结构上进行。



- 评价一个文件组织的效率
  - 执行文件操作所花费的时间
  - 文件组织所需要的空间

顺序文件 (Sequential File)

- 顺序文件中的记录按它们进入文件的先后顺序存放，其逻辑顺序与物理顺序一致。
- 如果文件的记录按主关键码有序，则称其为顺序有序文件，否则称其为顺序无序文件。
- 顺序文件通常存放在顺序存取设备（如磁带）上或直接存取设备（如磁盘）上。
- 当存放在顺序存取设备上时只能按顺序搜索法存取；当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。

- 顺序文件的存储方式

1. 连续文件：文件的全部记录顺序地存放于外存的一个连续的区域中。优点是存取速度快、存储利用率高、处理简单。缺点是区域大小需事先定义，不能扩充。
2. 串联文件：文件记录成块存放于外存中，在块中记录顺序连续存放，但块与块之间可以不连续，通过块链指针顺序链接。优点是文件可以扩充、存储利用率高。缺点是影响了存取和修改的效率。

直接存取文件 (Direct Access File)

- 又叫散列文件。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。
- 文件记录的逻辑顺序与物理顺序不一定相同。通过记录的关键码可直接确定该记录的地址。
- 使用散列函数把关键码集合映射到地址集合时，往往会产生地址冲突，处理冲突有两种处理方式：
  - 按桶散列
  - 可扩充散列

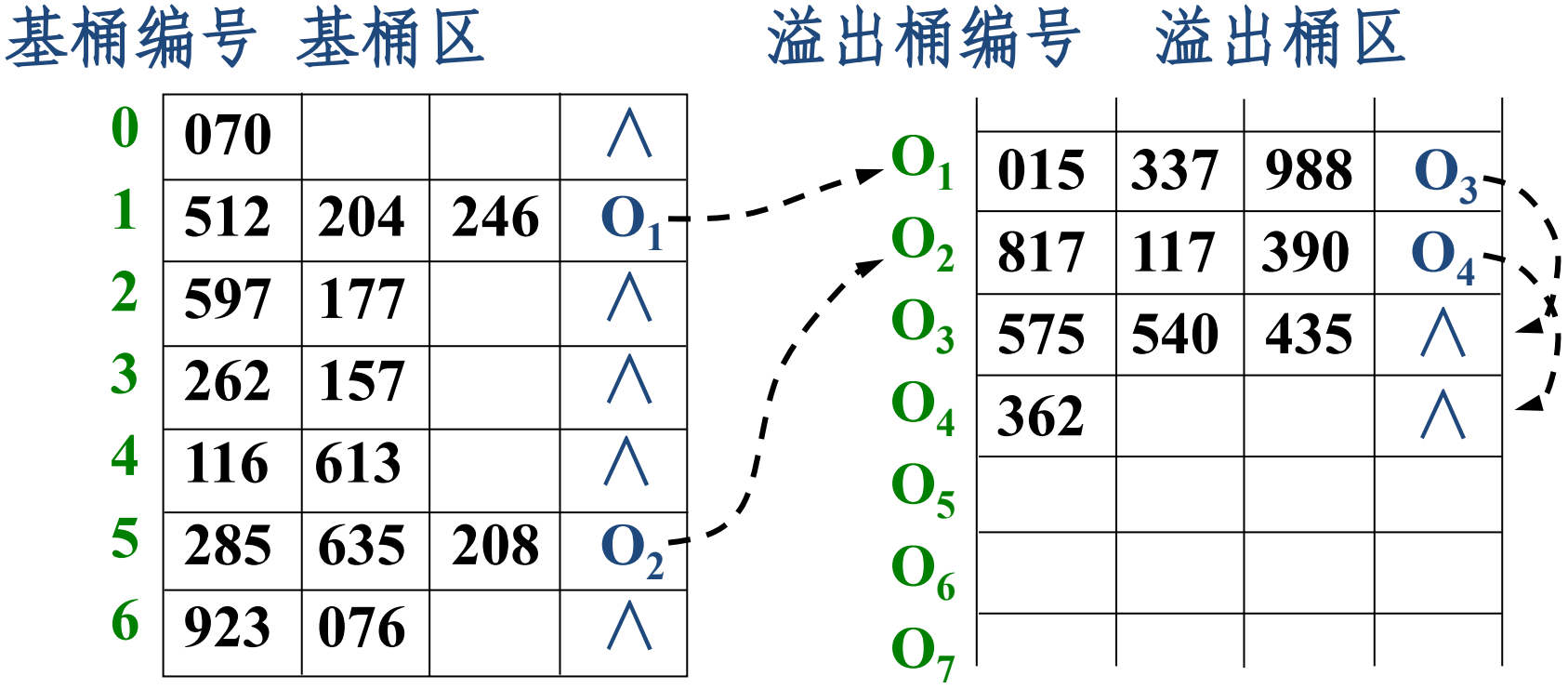
(1) 按桶散列

- 文件中的记录成组存放，若干个记录组成一个存储单位，称之为桶。假若一个桶能存放m个记录，则m个互为同义词的记录可以存放在同一地址的桶中。当第m+1个同义词出现时，才发生“溢出”。

(a) 溢出链

- 当发生“溢出”时，将第m+1个同义词存放到“溢出桶”。并称存放前m个同义词的桶为“基桶”。溢出桶和基桶大小相同。当在基桶中检索不成功，就循指针到溢出桶中检索。

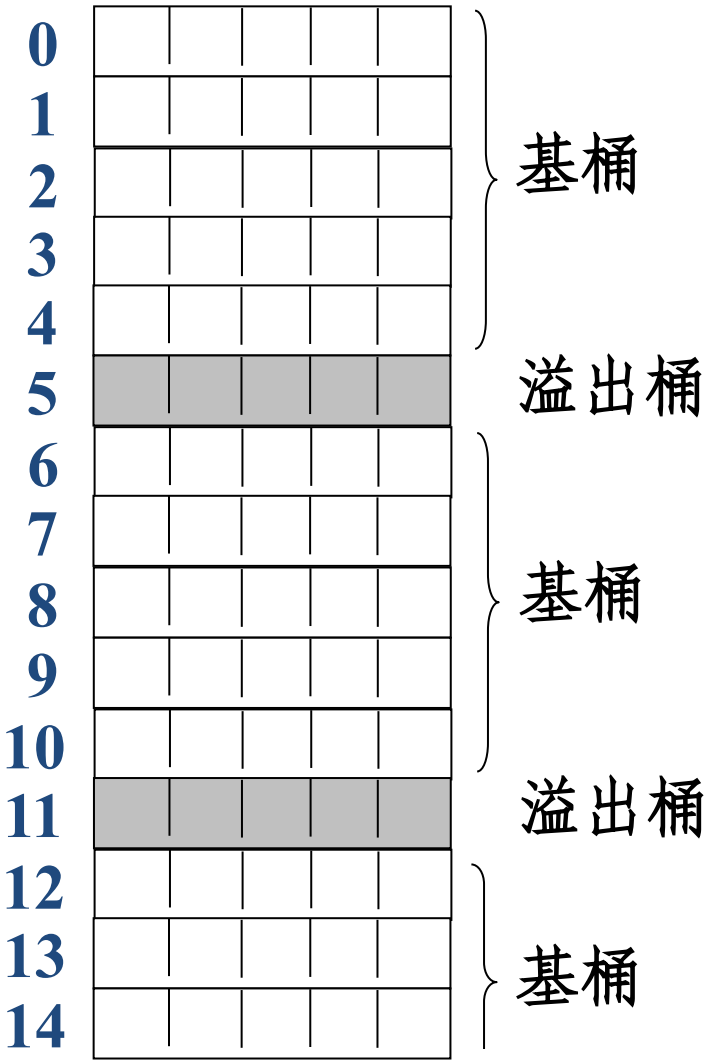
桶大小为3的溢出桶链表示例



- 在这种散列文件中删除记录时，因为可能需要重新链接，所以只需做一个逻辑删除标记即可，待系统做周期性重构时再做物理删除。

(b) 分布式溢出空间

- 溢出桶按照一定的间隔分布在基桶之间。如果有一个基桶溢出了，系统就将记录存放在下一个溢出桶中。
- 如果溢出桶自己溢出了，则使用下一个相继的溢出桶，这需要第二次溢出处理。



分布式溢出桶

- 如果系统对基桶按0, 1, 2, 3, 4, 5, …进行编号，在按间隔 G = 5 插入溢出桶后，可按下列公式按字节求出各个桶的实际存储地址:

$$\text{桶的地址} = B_0 + B \times \left(i + \left\lfloor \frac{i}{5} \right\rfloor \right).$$

- 其中，B_0是在文件中第0号桶的起始地址，B是每个桶的字节数。在括号中的除数5表示每隔5个基桶安排一个溢出桶。

(c) 相继溢出法

- 此方法不设置溢出桶。当记录应存放的桶溢出时，溢出记录存放到下一个相继的桶中。

- 如果该桶已满，就把它放到再下一个桶中，如此处理，直至把记录存放好。
- 相继溢出法的优点是对溢出不需要漫长的寻找。紧邻的桶通常相距不多于一次磁盘旋转。但当邻近的多个桶被挤满时，则为了查找空闲空间就需要检查许多桶。如果桶的容量很小更是如此。

H(key) = key % 11

0	362			
1	177			
2				
3	597	157	817	575
4	070	246	015	542
5	389			
6	116	204	512	435
7	337	117		
8	635	613		
9	262	988		
10	285	923	076	208

相继溢出法

(2) 可扩充散列

- 这是基于数字搜索树的一种散列方法，细节参见**10.5**节。

散列文件优缺点

- 散列文件具有随机存放、记录不需进行排序、插入删除方便、存取速度快、不需要索引区和节省存储空间等优点。

- 散列文件不能顺序存取，只能按关键码随机存取。在经过多次插入、删除后，可能出现溢出桶满而基桶内多数记录已被删除的情况。此时需要重新组织文件。

索引文件 (Indexed File)

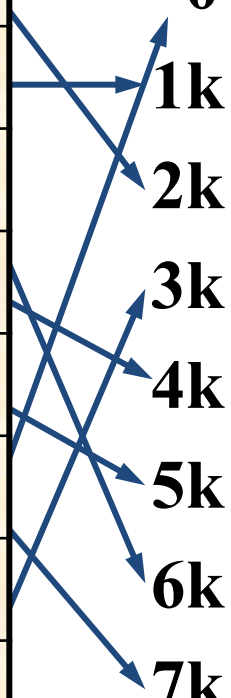
- 索引文件由索引表和数据表（主文件）组成。
- 索引表用于指示逻辑记录与物理记录间的对应关系，它是按关键码有序的表。
  - 索引顺序文件：主文件也按关键码有序。此时可对主文件分组，一组记录对应一个索引项。称这种索引表为稀疏索引。
  - 索引非顺序文件：主文件中记录未按关键码有序。此时，每一个主文件记录必须对应一个索引项。称这种索引表为稠密索引。

- 静态索引：采用多级索引结构，每一级索引均为有序表。优点是结构简单，缺点是修改很不方便，每次修改都要重组索引。
- 动态索引：采用可动态调整的平衡搜索树结构，如二叉搜索树、B树与B+树等。优点是插入、删除和搜索都很方便。
- 在文件中搜索时，访问外存所花费时间比在内存中搜索所需的时间大得多，因此，外存上搜索一个记录的时间代价主要取决于访问外存的次数，即索引树的高度。

索引非顺序文件示例

索引表

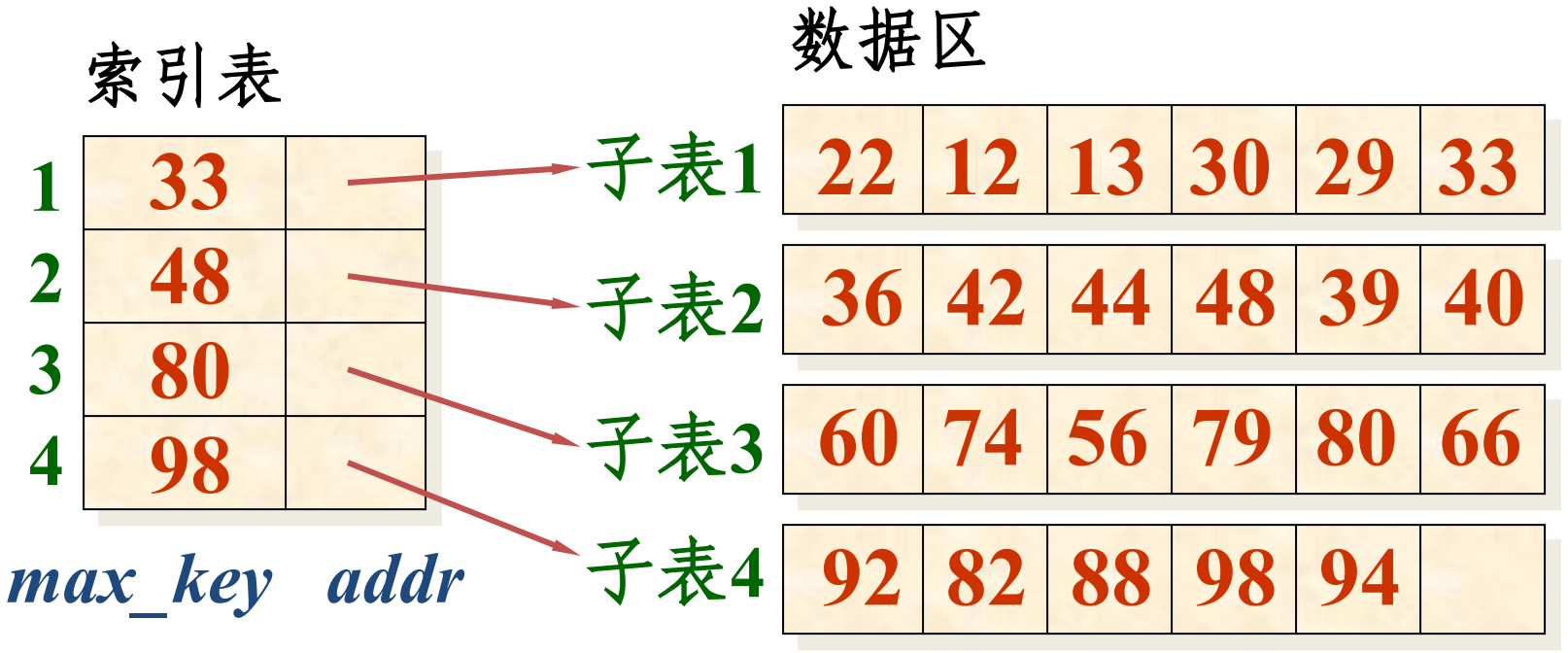
key	addr
03	2k
08	1k
17	6k
24	4k
47	5k
51	7k
83	0
95	3k



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	…

索引顺序文件示例



- 当记录在外存中有序存放时，可以把所有*n*个记录分为*b*个子表 (块) 存放，一个索引项对应数据表中一组记录 (子表)。

- 对索引顺序文件进行搜索，一般分为两级:
  - 先在索引表*ID*中搜索给定值*K*, 确定满足

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

    的 *i* 值, 即待查记录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记录。
- 索引表是按*max_key*有序的，且长度也不大,可以折半搜索，也可以顺序搜索。
- 各子表内各个记录如果也按关键码有序，可以采用折半搜索或顺序搜索；如果不是按关键码有序，只能顺序搜索。

- 索引顺序文件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，$ASL_{SubList}$ 是在子表内搜索记录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个记录，则$b = \lceil n/s \rceil$。又设表中每个记录的搜索概率相等，则每个子表的搜索概率为$1/b$，子表内各记录的搜索概率为 $1/s$。
- 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

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

- 索引顺序文件的平均搜索长度与表中的记录个数n有关，与每个子表中的记录个数s有关。在给定n的情况下，s 应选择多大?
- 用数学方法可导出，当 $s = \sqrt{n}$ 时，$ASL_{IndexSeq}$ 取极小值 $\sqrt{n}+1$。这个值比顺序搜索强，但比折半搜索差。但如果子表存放在外存时，还要受到页块大小的制约。
- 若采用折半搜索确定记录所在的子表，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)

- 对包含有大量数据记录的数据表或文件进行搜索时，最常用的是针对记录的主关键码建立索引。主关键码可以唯一地标识该记录。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记录的关键码和记录在表或文件中的存放地址。

记录关键码 *key*	记录存放地址 *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：

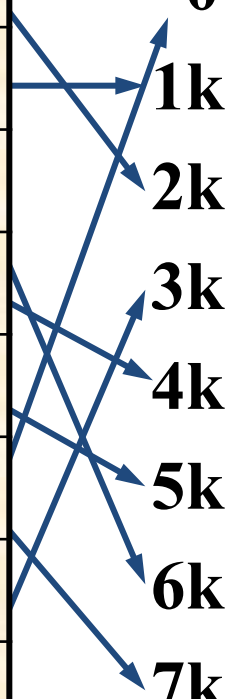
  (1) 列出所有教师的名单；
  (2) 已婚的女性职工有哪些人？

- 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记录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。

索引非顺序文件示例

索引表

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

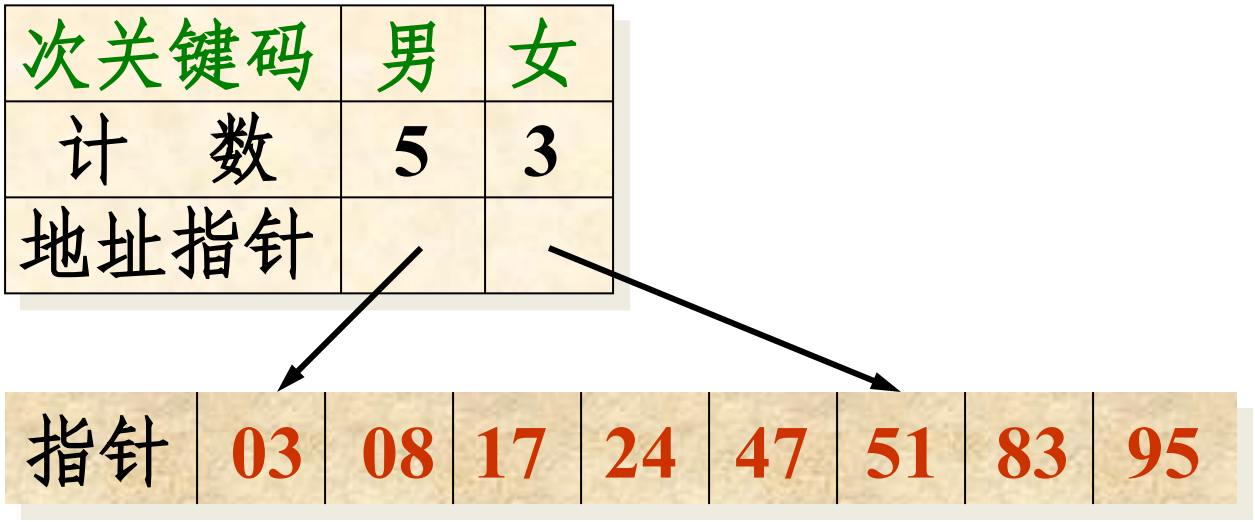


数据表

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	…

- 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。
- 例如，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。

性别次索引



婚否次索引

次关键码	已婚	未婚
计　数	5	3
地址指针		

指针	03	08	24	47	83	17	51	95

职务次索引

次关键码	教师	教务员	实验员
计　数	5	1	2
地址指针			

指针	08	24	47	51	83	03	17	95

(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是已婚的职工记录，从而回答上面的查询(2)。
- 倒排表是次索引的一种实现。在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记录。
- 在次索引中记录记录存放位置的指针可以用主关键码表示: 可通过搜索次索引确定该记录的主关键码, 再通过搜索主索引确定记录的存放地址。

- 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，索引项中不存放记录的存储地址,而是存放该记录所在硬件区域（即存储区域）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（下页）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记录。

		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

- 针对一个查询：找出所有广东籍学建筑的男学生。可以从“性别”、“籍贯”、“专业”三个次索引分别抽取属性值为“男”、“广东”、“建筑”的位向量，按位求交，求得满足查询要求的记录在哪些硬件区域中，再读入这些硬件区域，从中查找所需的数据记录。

	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

- 由运算结果可知，在硬件区域1，……中有所需的记录。然后将硬件区域1，……等读入内存，在其中进行检索，就可取出所需记录。

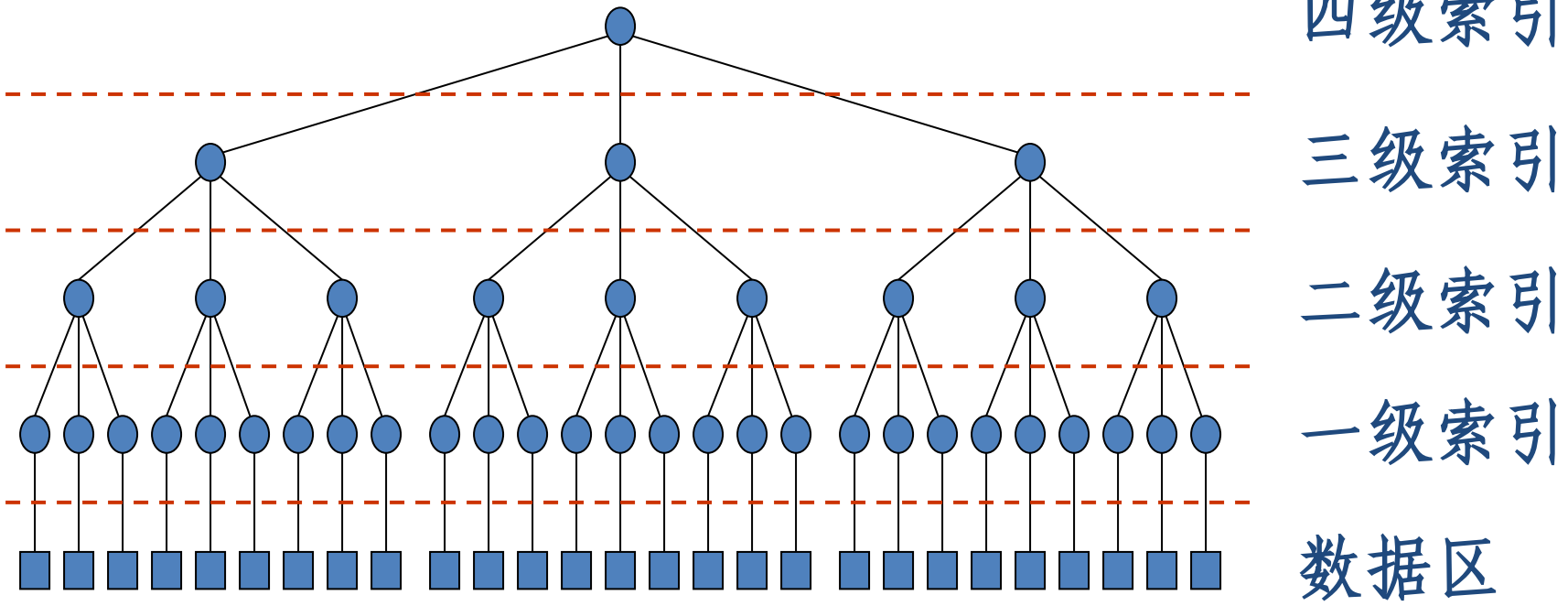
多级索引结构

- 当数据记录数目特别大，索引表本身也很大，在内存中放不下，需要分批多次读取外存才能把索引表搜索一遍。
- 此时，可以建立索引的索引(二级索引)。二级索引可以常驻内存，二级索引中一个索引项对应一个索引块，登记该索引块的最大关键码及该索引块的存储地址。
- 如果二级索引在内存中也放不下，需要分为许多块多次从外存读入。可以建立二级索引的索引(三级索引)。这时，访问外存次数等于读入索引次数再加上1次读取记录。
- 必要时，还可以有4级索引，5级索引，….

- 多级索引结构常用 *m* 叉树表示，称为*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。

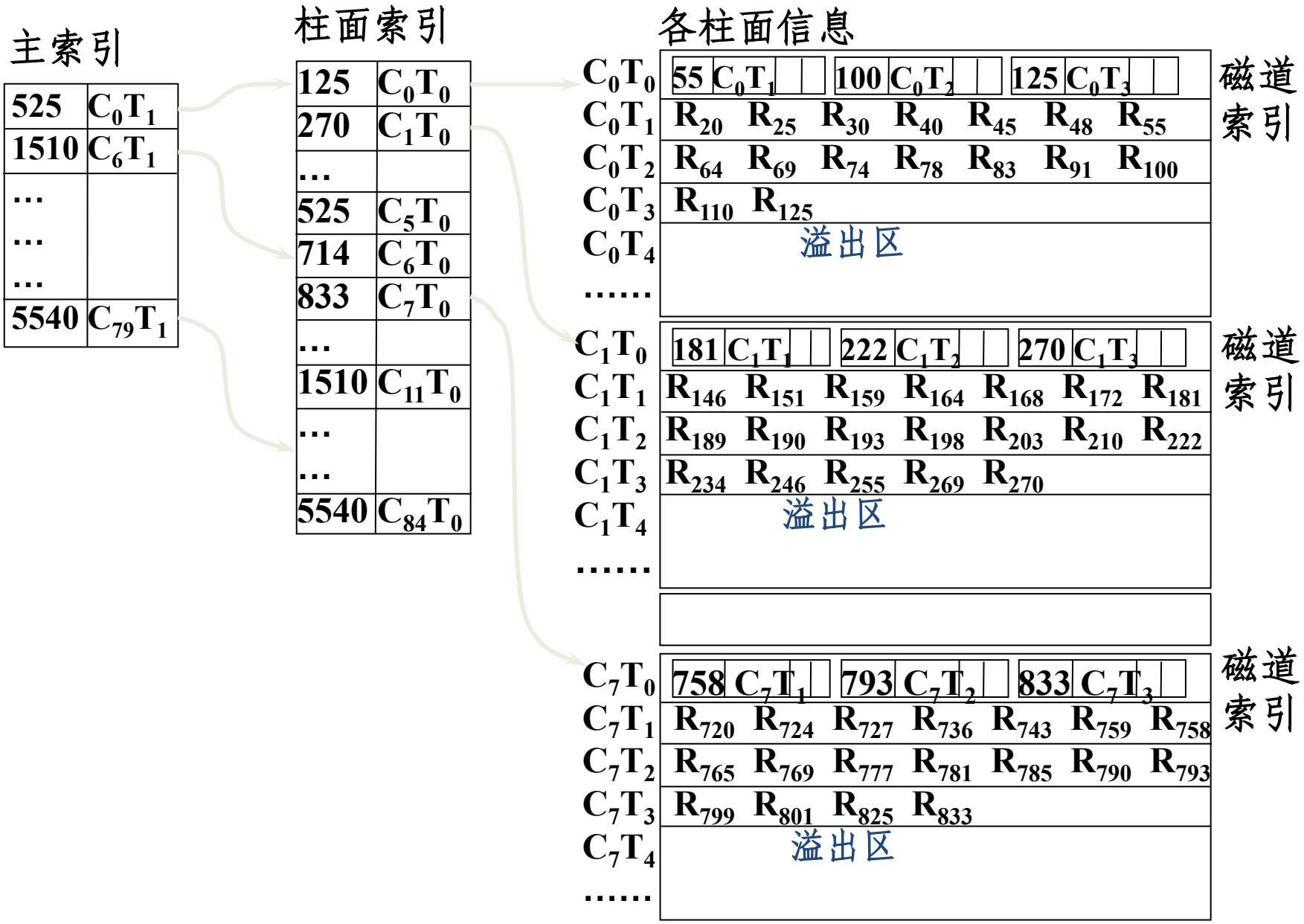
- *m*路搜索树还可能是动态索引结构，即在整个系统运行期间，树的结构随数据的增删及时调整，以保持最佳的搜索效率。



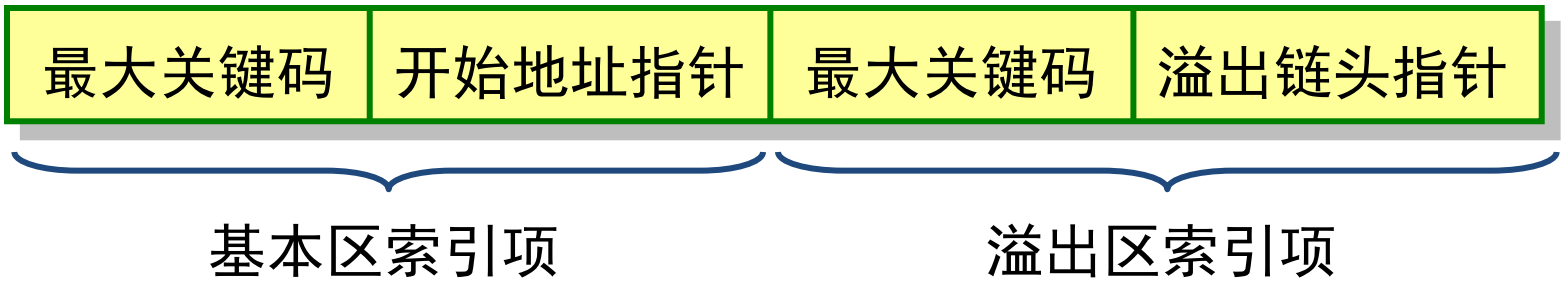
多级索引结构形成 *m* 路搜索树

ISAM（索引顺序存取方法文件）

- 它是静态索引结构。典型的例子是对磁盘上的数据文件建立盘组、柱面、磁道三级地址的多级索引。
- **ISAM**文件用柱面索引对各个柱面进行索引。一个柱面索引项保存该柱面上的最大关键码（最后一个记录）以及柱面开始地址指针。
- 如果柱面太多，可以建立柱面索引的分块索引，即主索引。主索引不太大，一般常驻内存。


主索引
525 C_0T_1
1510 C_6T_1
… … …
5540 $C_{79}T_1$
柱面索引
125 C_0T_0
270 C_1T_0
…
525 C_5T_0
714 C_6T_0
833 C_7T_0
…
1510 $C_{11}T_0$
… …
5540 $C_{84}T_0$
各柱面信息
C_0T_0 55 C_0T_1 100 C_0T_2 125 C_0T_3
C_0T_1 R_{20} R_{25} R_{30} R_{40} R_{45} R_{48} R_{55}
C_0T_2 R_{64} R_{69} R_{74} R_{78} R_{83} R_{91} R_{100}
C_0T_3 R_{110} R_{125}
C_0T_4 溢出区
……
磁道索引
C_1T_0 181 C_1T_1 222 C_1T_2 270 C_1T_3
C_1T_1 R_{146} R_{151} R_{159} R_{164} R_{168} R_{172} R_{181}
C_1T_2 R_{189} R_{190} R_{193} R_{198} R_{203} R_{210} R_{222}
C_1T_3 R_{234} R_{246} R_{255} R_{269} R_{270}
C_1T_4 溢出区
……
磁道索引
C_7T_0 758 C_7T_1 793 C_7T_2 833 C_7T_3
C_7T_1 R_{720} R_{724} R_{727} R_{736} R_{743} R_{759} R_{758}
C_7T_2 R_{765} R_{769} R_{777} R_{781} R_{785} R_{790} R_{793}
C_7T_3 R_{799} R_{801} R_{825} R_{833}
C_7T_4 溢出区
……
磁道索引

- 在每个柱面上，所有数据记录存放于基本区，此外保留一部分磁道作为溢出区。所有记录在基本区按关键码升序排列，后一磁道所有记录的关键码均大于前一磁道所有记录的关键码。
- 在一个柱面上所有记录分布在一系列磁道上，通过磁道索引进行搜索。磁道索引一般放在每个柱面上第0号磁道中，
- 每个磁道索引的索引项由两部分组成：



- 基本区索引项存放本磁道在基本区最大关键码（在基本区该磁道最后一个记录）和本磁道在基本区的开始地址，溢出区索引项存放本磁道在溢出区最大关键码和本磁道在溢出区中溢出记录链（有序链表）的第一个结点地址。
- 在某一磁道插入一个新记录时，如果原来该磁道基本区记录已经放满，则根据磁道索引项指示位置插入新记录后，把最后的溢出记录（具有最大关键码）移出磁道基本区，再根据溢出索引项将这个溢出记录放入溢出区，并以有序链表插入算法将溢出记录链入。

动态索引结构

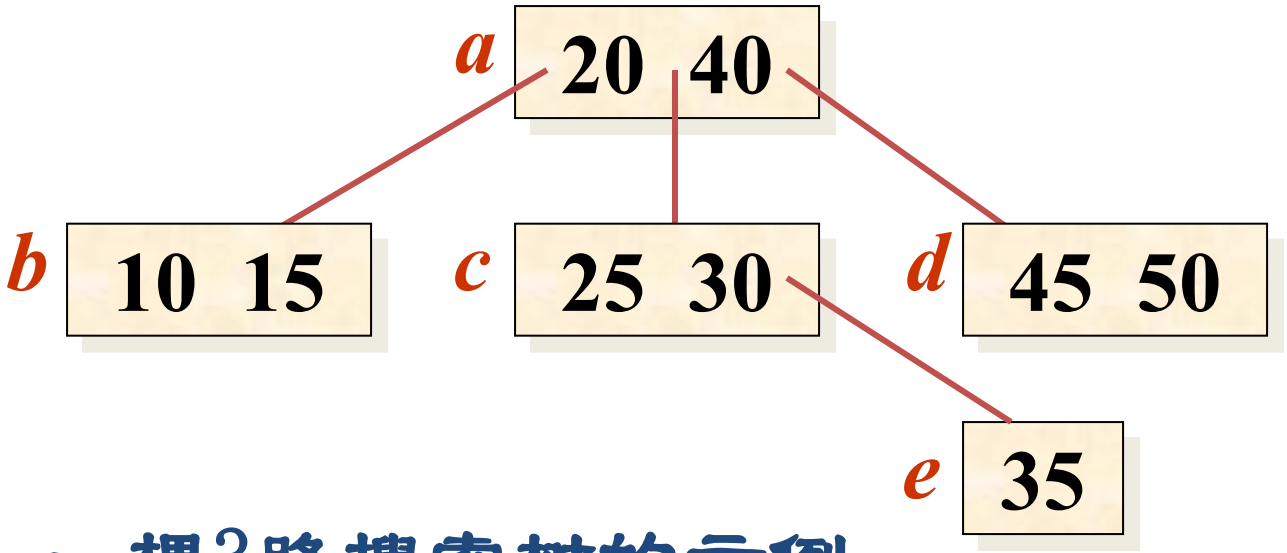
动态的 m 路搜索树

- 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，它的递归定义为：
- 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：

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

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，$0 \le i < n$ 。



一棵3路搜索树的示例

M路搜索树的C++描述

```
const int MaxValue = ……;
//关键码集合中不可能有的最大值
template <class T>
struct MtreeNode {                    //树结点定义
    int n;                            //索引项个数
    MtreeNode<T> *parent;             //父结点指针
    T key[m+1];  //key[m]为监视哨，key[0]未用
    int *recptr[m+1];                 //索引项记录起始地址指针
    MtreeNode<T> *ptr[m+1];//子树结点指针，ptr[m]
    在插入溢出时使用
};
```

```
template <class T>                  //搜索结果三元组
struct Triple {
    MtreeNode<T> *r;                //结点地址指针
    int i;                                  //结点中关键码序号i
    int tag;                                //tag=0,成功; =1,失败
};

template <class T>
class Mtree {                       //m叉搜索树定义
protected:
    MtreeNode<T> *root;                     //根指针
    int m;                                  //路数
public:
    Triple<T> Search(const T& x);   //搜索
};
```

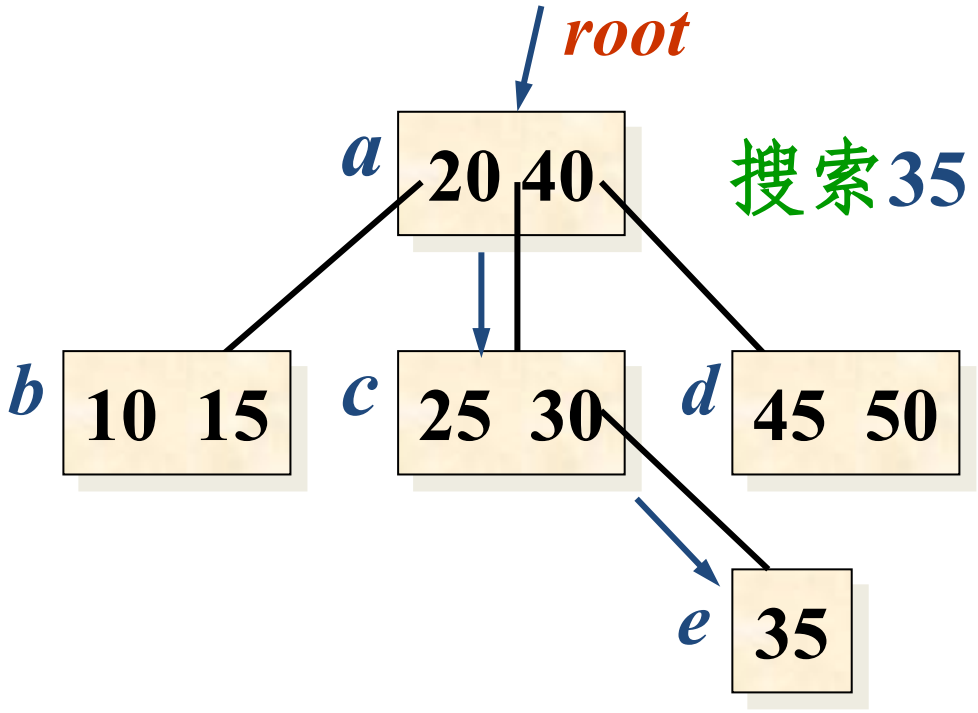
- AVL树是 2 路搜索树。若已知 m 路搜索树的度 m 和它的高度 h, 则树中的最大结点个数为:

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

- 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p的父结点指针
    int i = 0 ;
```

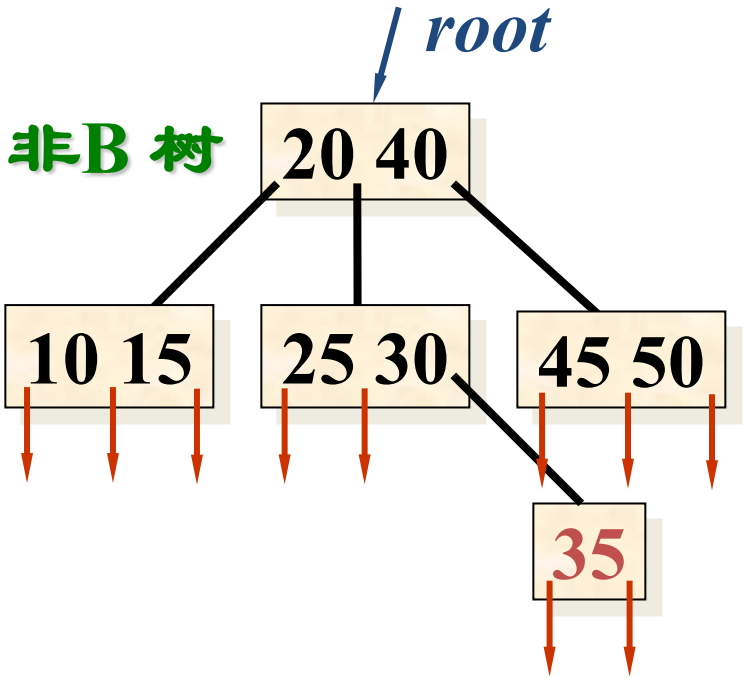
```
while (p != NULL) {                          //从根开始检测
    i = 0;   p->key[(p->n)+1] = MaxValue;
    while (p->key[i+1] < x) i++;             //在结点内搜索
    if (p->key[i+1] == x) {                  //搜索成功
        result.r = p;  result.i = i+1;  result.tag = 0;
        return result;
    }
    q = p;  p = p->ptr[i];
      //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
};
```

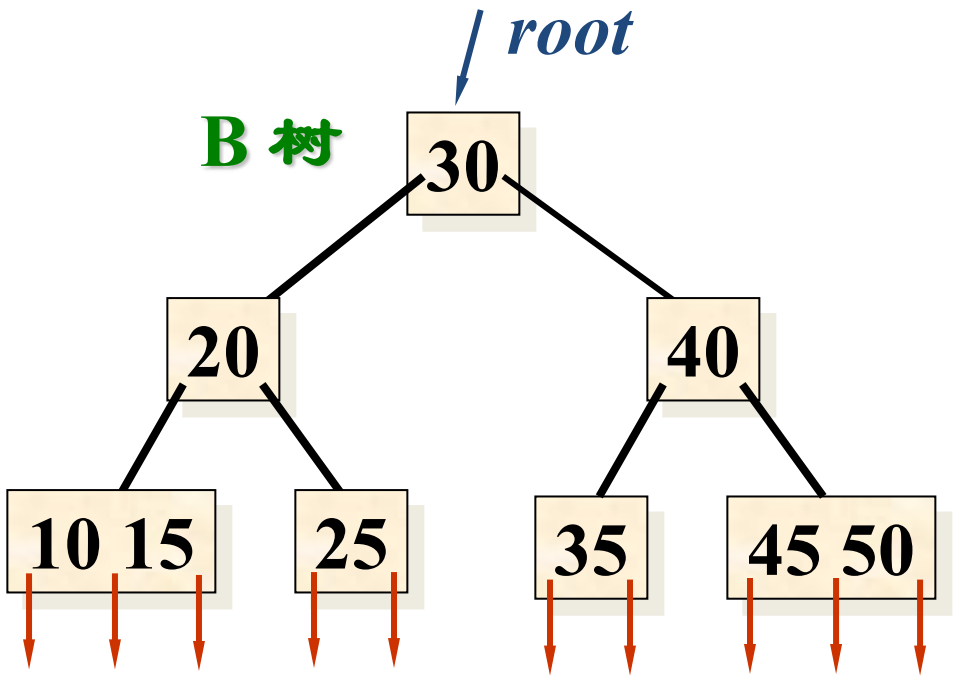
- 提高搜索树的路数 ***m***, 可以改善树的搜索性能。对于给定的关键码数 ***n***，如果搜索树是平衡的，可以使 ***m*** 路搜索树的性能接近最佳。下面将讨论一种称之为**B** 树的平衡的 ***m*** 路搜索树。

B 树

- 一棵 m 阶B 树是一棵平衡(balanced)的 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。
- 在B 树中的“失败”结点是当 x 不在树中时才能到达的结点。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。

- 注意，m阶B树继承了m路搜索树的定义。原来m路搜索树定义中的规定在m阶B树中都保留。
- 事实上，在B 树的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]。
- 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。


root
非B 树
20 40
10 15
25 30
45 50
35


root
B 树
30
20
40
10 15
25
35
45 50

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树的搜索算法继承了*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。
- 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，那么，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。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。

- 所有这些结点都不是失败结点。失败结点在第h+1层，失败结点个数为N+1。

- 这是因为树中关键码有N个，而失败数据一般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

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

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

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

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

- 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, 结点内最多有 m-1个索引项 $(key_i, recptr_i, P_i)$, $1 \le i < m$。
- 设 key_i 所占字节数为α，$recptr_i$ 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 树中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：

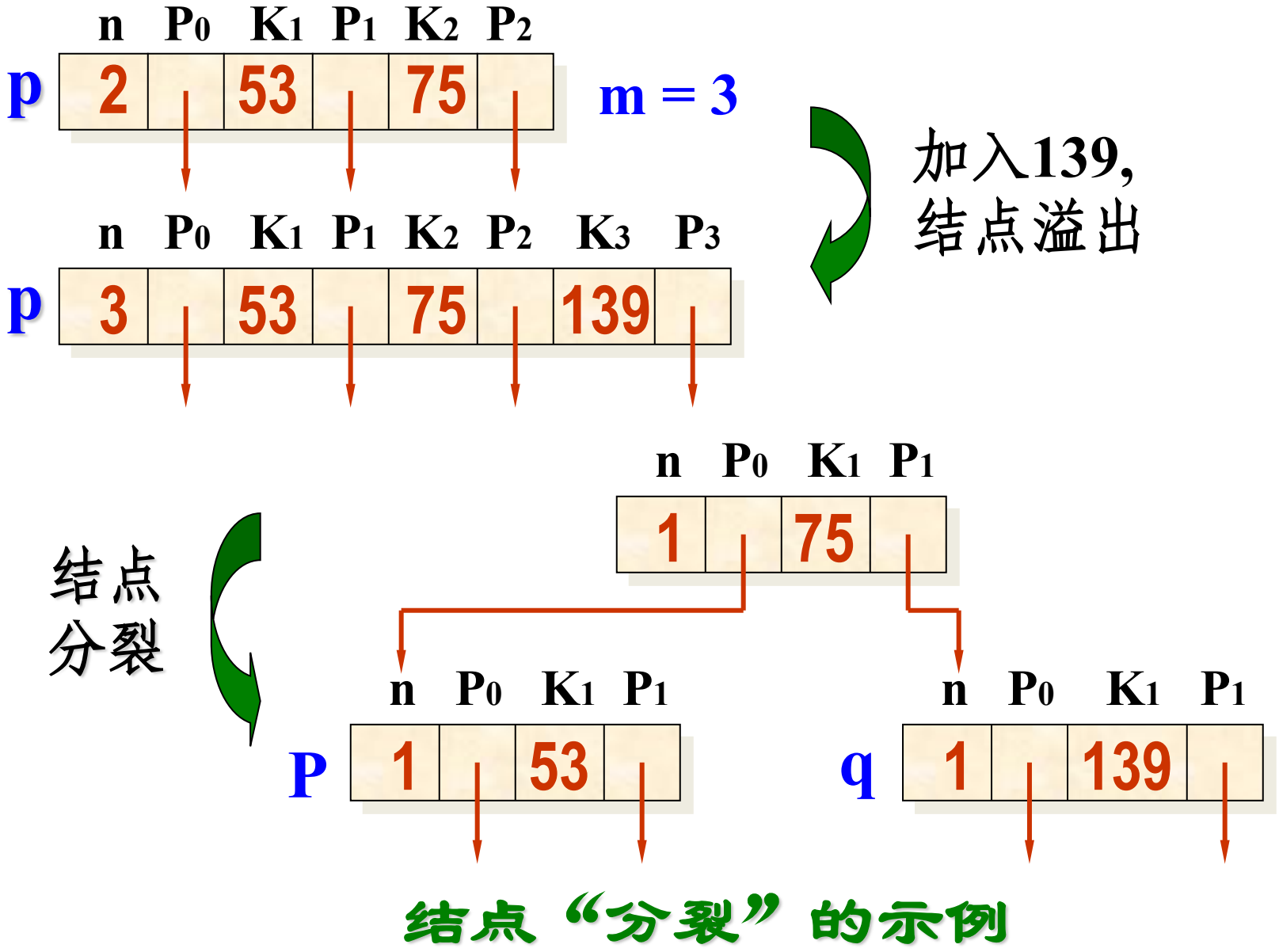
- 结点 p:

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

- 结点 q:

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

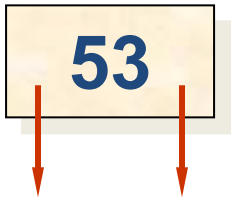
✓位于中间的关键码 $K_{\lceil m/2 \rceil}$ 与指向新结点 q 的指针形成一个二元组 $(K_{\lceil m/2 \rceil}, q)$，插入到这两个结点的双亲结点中去。


n P0 K1 P1 K2 P2
p 2 53 75
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

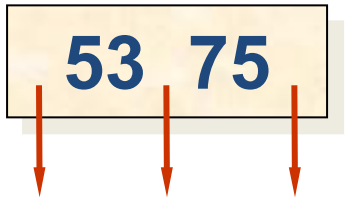
结点“分裂”的示例

示例：从空树开始加入关键码建立3阶B树

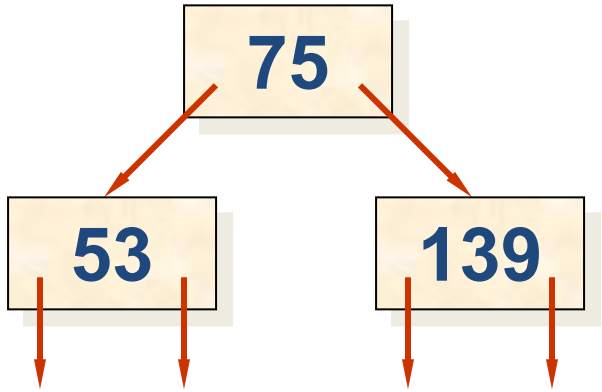
n=1 加入53



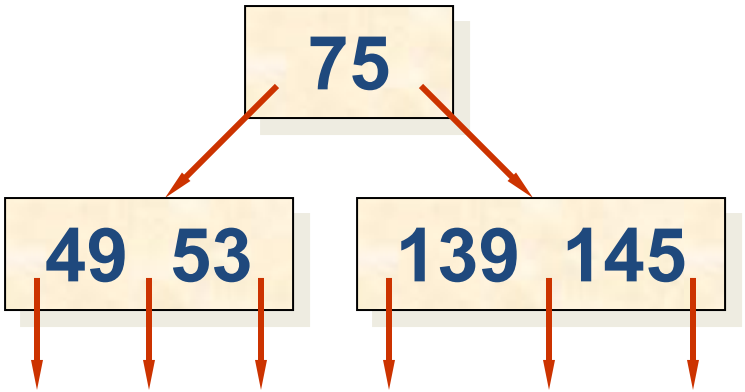
n=2 加入75



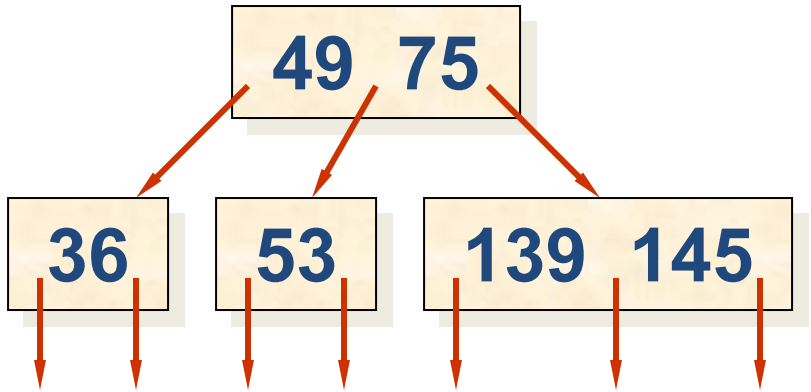
n=3 加入139

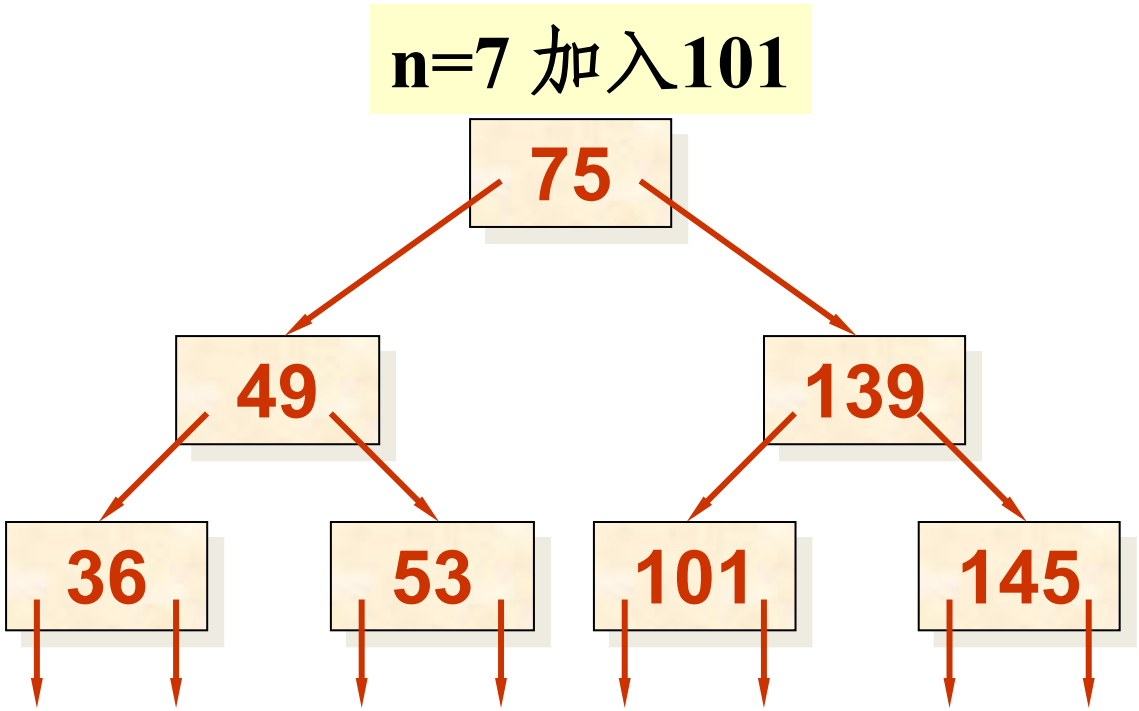


n=5 加入49,145



n=6 加入36





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

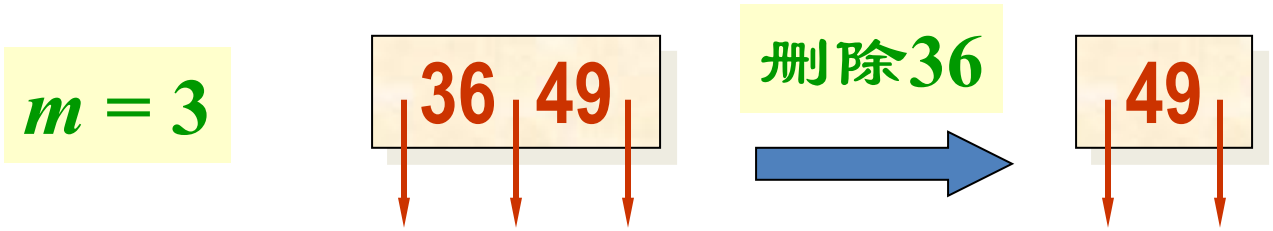
- 在插入新关键码时，需要自底向上地分裂结点，最坏情况下从被插关键码所在叶结点到根的路径上的所有结点都要分裂。

B 树的删除

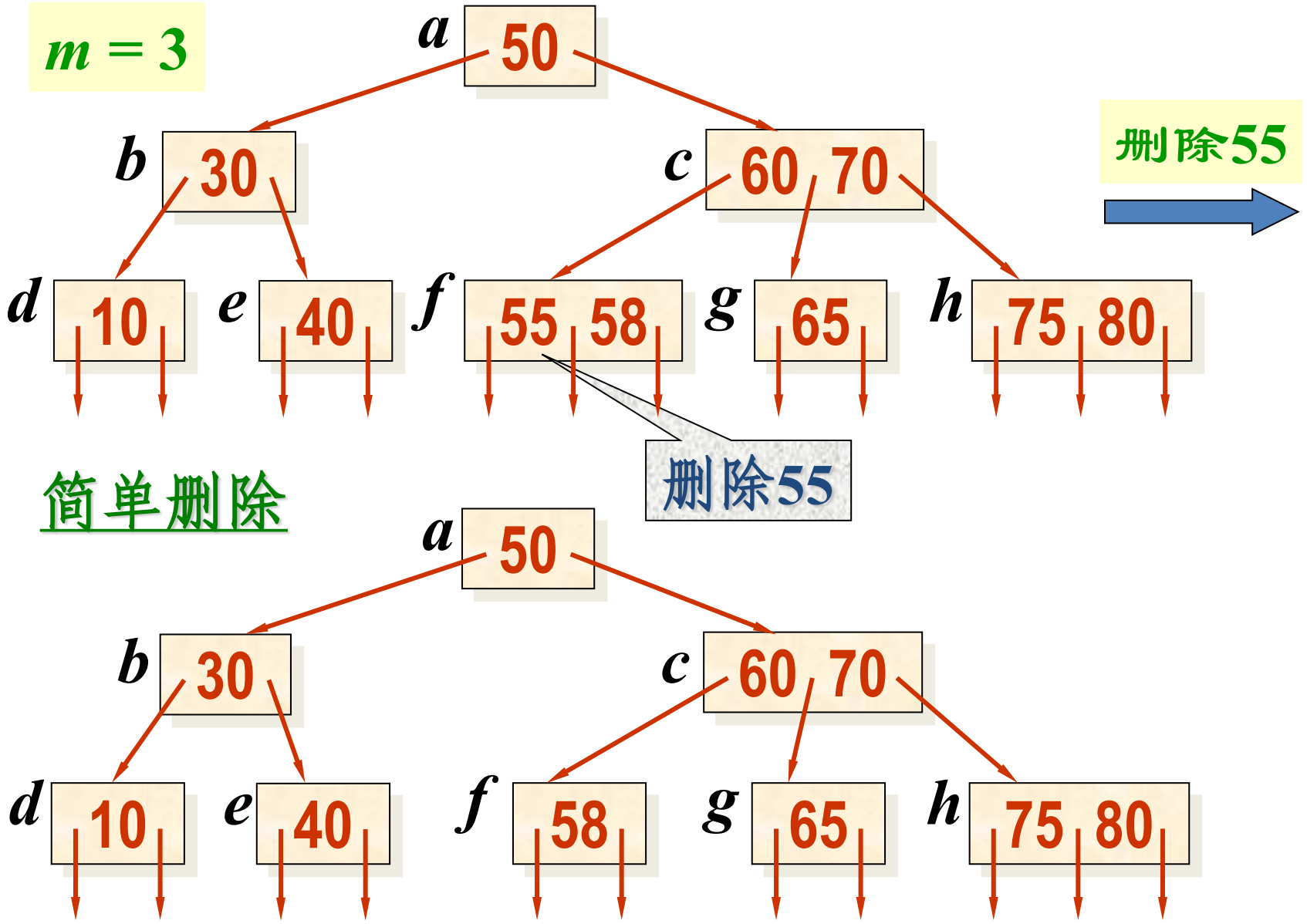
- 在B 树上删除一个关键码时，若结点中所剩关键码个数少于下限，要考虑结点的调整或合并问题，删除过程如下：
  - ✓首先需要找到这个关键码所在的结点，从中删去这个关键码。
  - ✓若该结点不是叶结点，且被删关键码为 K_i，$1 \le i \le n$，则在删去该关键码之后，应以该结点 P_i 所指示子树中的最小关键码 x 来代替被删关键码 K_i 所在的位置；
  - ✓然后在 x 所在的叶结点中删除 x。

在叶结点上的删除有 4 种情况。

① 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≥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 $n \geqslant \lceil m/2 \rceil$，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

③ 被删关键码所在叶结点删除前关键码个数 $n=\lceil m/2\rceil-1$，若这时与该结点相邻的右兄弟(或左兄弟)结点的关键码个数 $n\geqslant\lceil m/2\rceil$，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，以达到新的平衡。

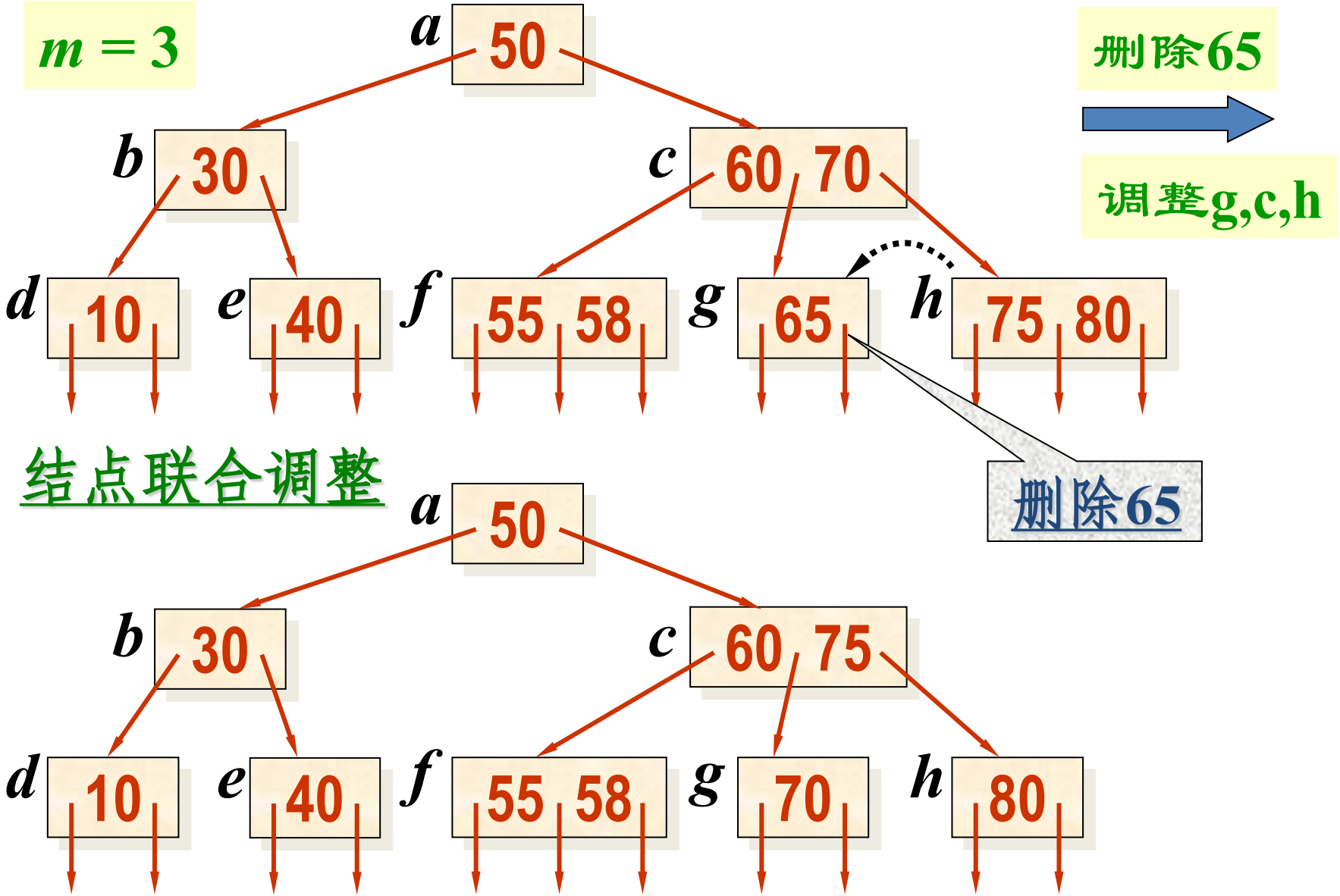
a) 将双亲结点中刚刚大于(或小于)该被删关键码的关键码 K_i $(1\leq i\leq n)$ 下移；

b) 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；

c) 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;

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。


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

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

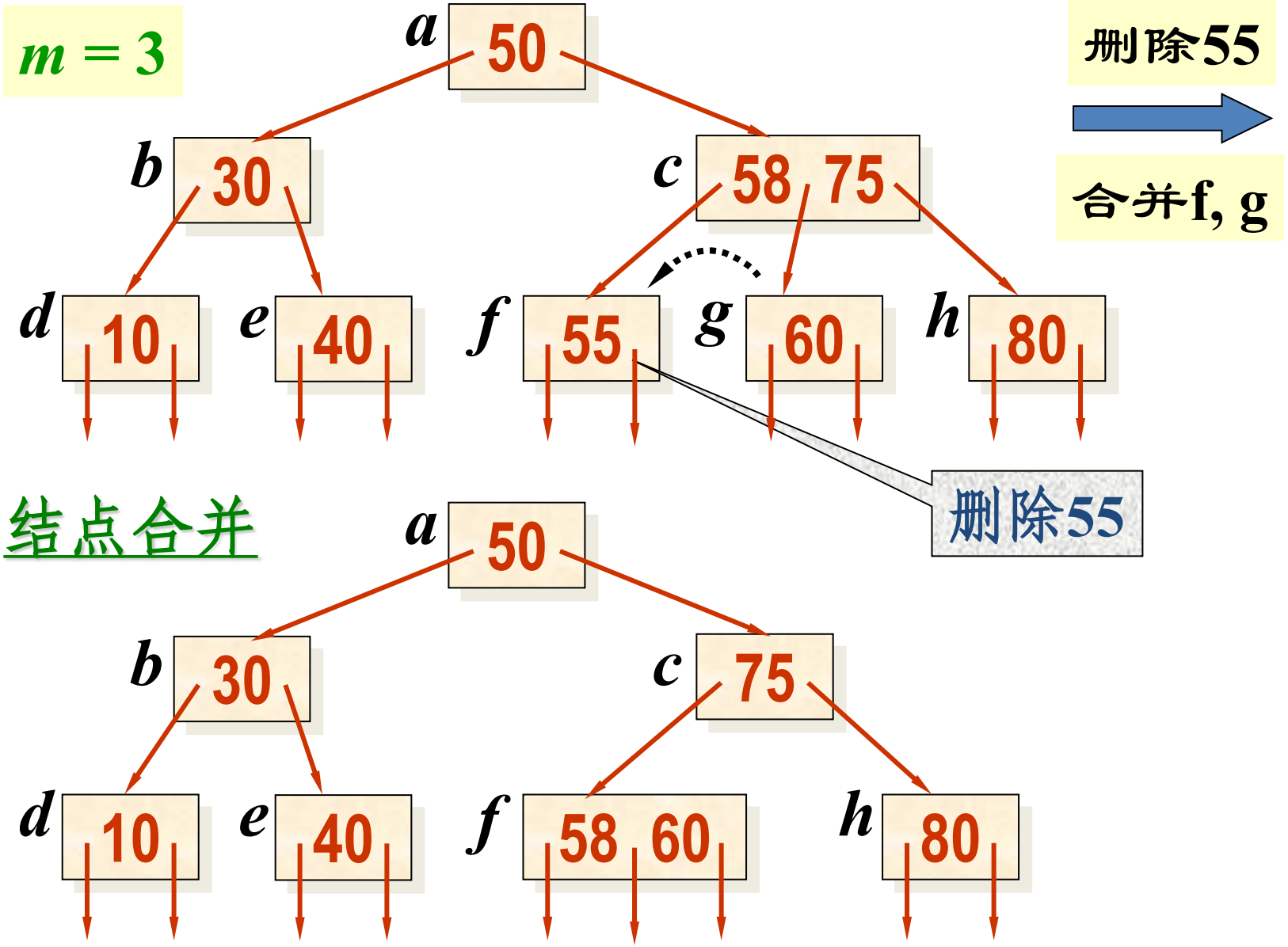
a) 若要合并 p 中的子树指针 P_i 与 P_{i+1} 所指的结点，且保留 P_i 所指结点，则把 p 中的关键码 K_{i+1} 下移到 P_i 所指的结点中。

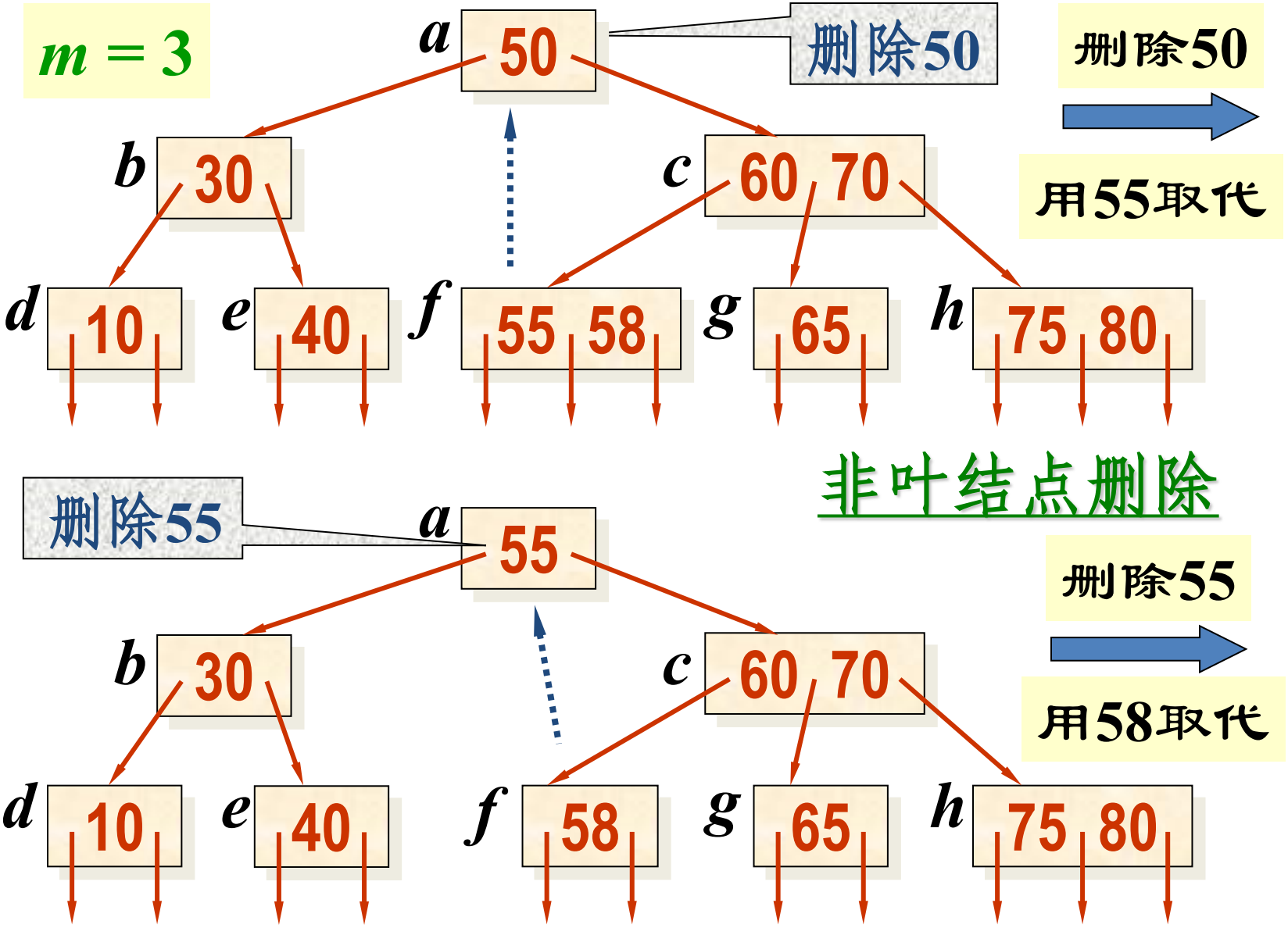
b) 把 p 中子树指针 P_{i+1} 所指结点中的全部指针和关键码都照搬到 P_i 所指结点的后面。删去 P_{i+1} 所指的结点。

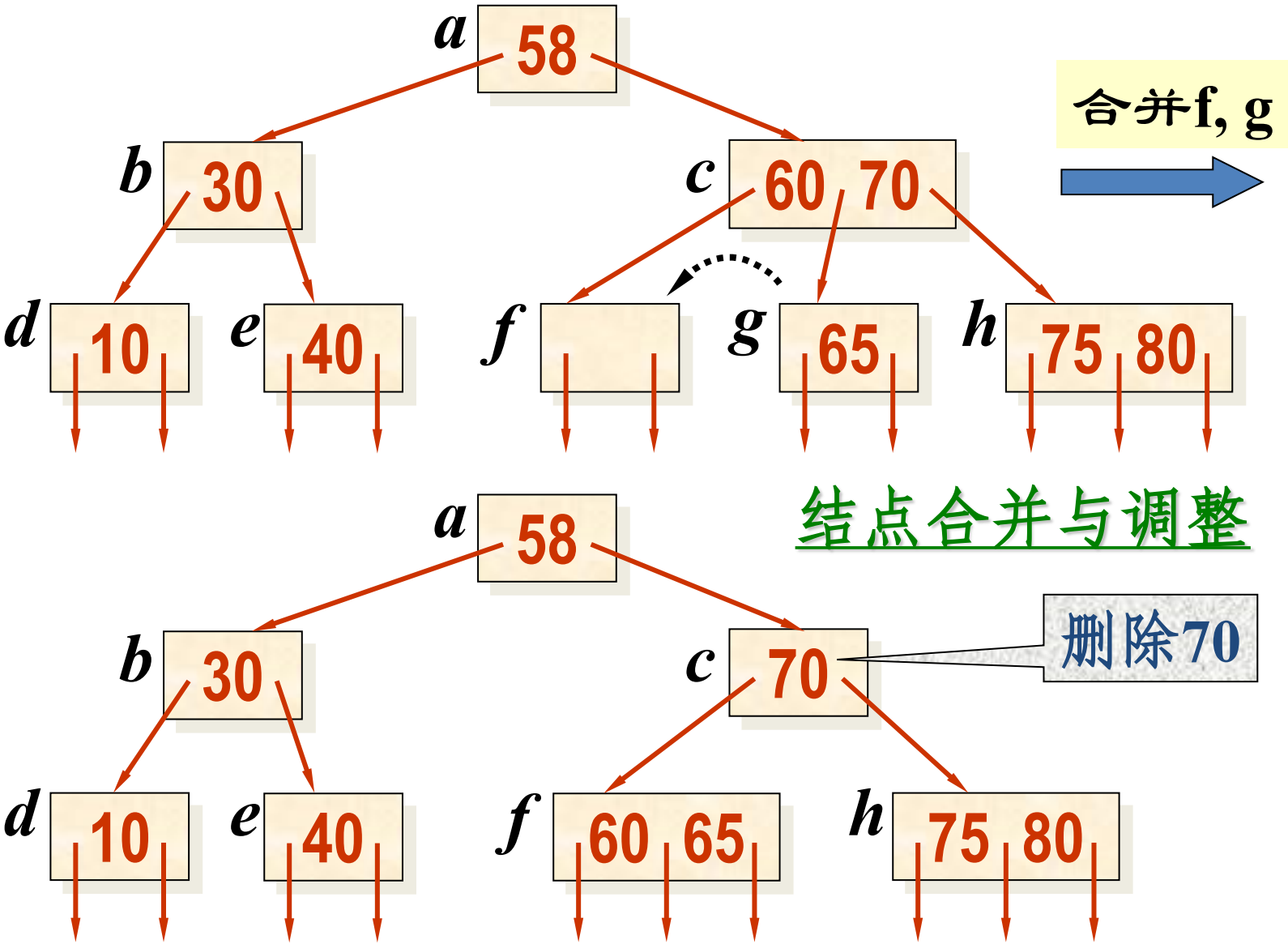
c) 在结点 p 中用后面剩余的关键码和指针填补关键码 K_{i+1} 和指针 P_{i+1}。

d) 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。


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

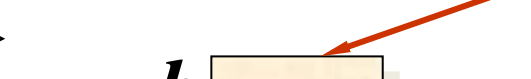

m = 3
a
50
删除50
删除50
用55取代
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
非叶结点删除
删除55
a
55
删除55
用58取代
b
30
c
60 70
d
10
e
40
f
58
g
65
h
75 80


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

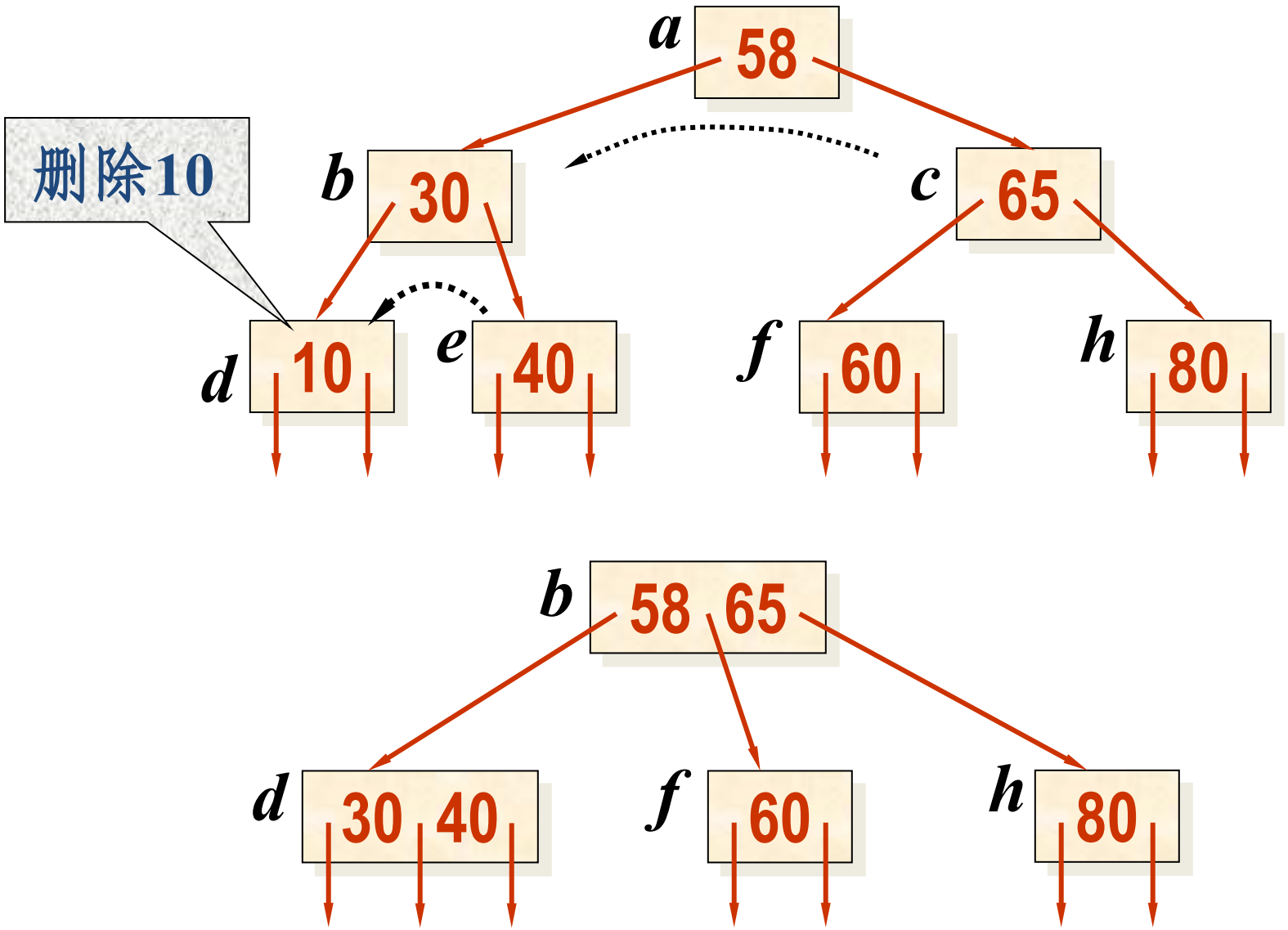
用75取代

删除75



用80取代

调整f, c, 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

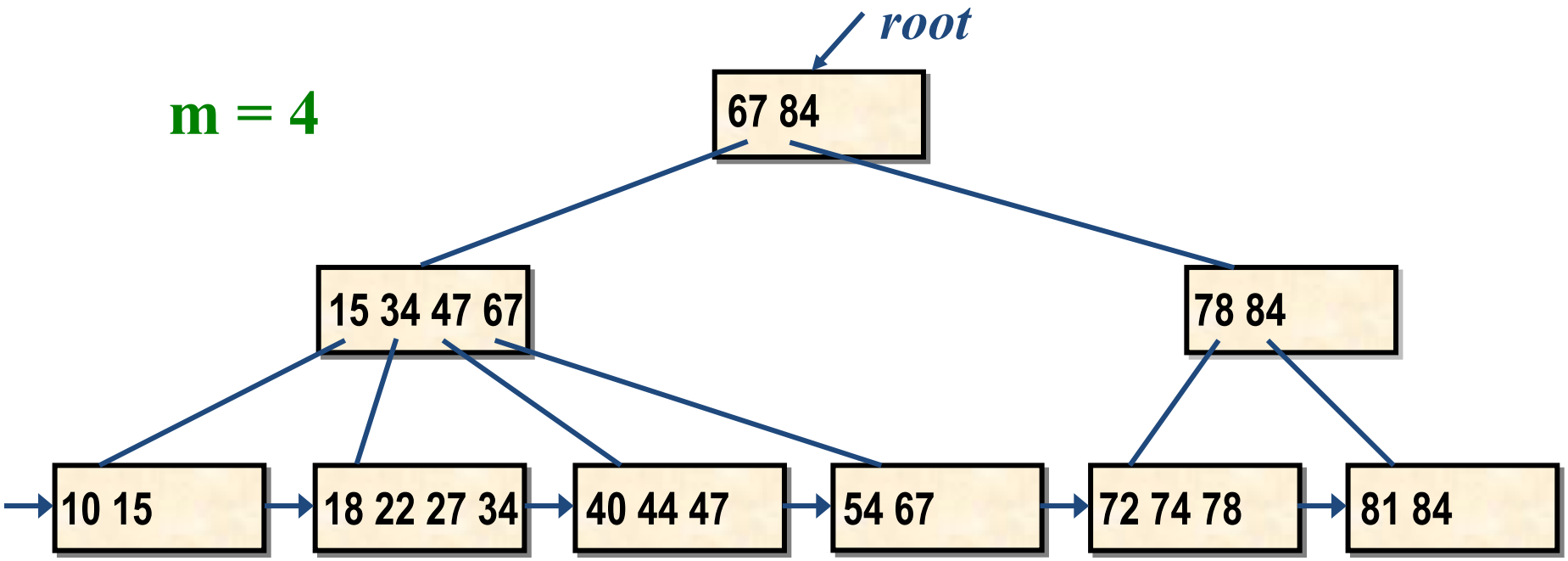
- B+ 树是B 树的变种，它与B 树的不同之处在于:
  - ✓所有关键码都存放在叶结点中，上层的非叶结点的关键码是其子树中最小（或最大）关键码的复写。
  - ✓叶结点包含了全部关键码及指向相应数据记录存放地址的指针，且叶结点本身按关键码从小到大顺序链接。
- 每个非叶结点结构有两种方式处理。按下层结点“最大关键码复写”和“最小关键码复写”。

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棵子树的结点有 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$2 \le n \le 4$，其所有的关键码都出现在叶结点中，且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。



- 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。

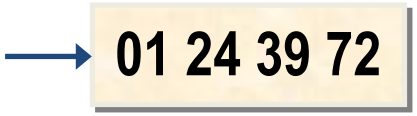
B+ 树的插入

- 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索引项个数是否超出范围m。
- 当插入后叶结点中的关键码个数n > m时，需要将叶结点分裂为两个结点：它们包含的关键码个数分别为⌈(m+1)/2⌉和⌊(m+1)/2⌋。并且它们的双亲结点中应同时包含这两个结点的最大关键码和结点地址。
- 在非叶结点中关键码的插入与叶结点的插入情

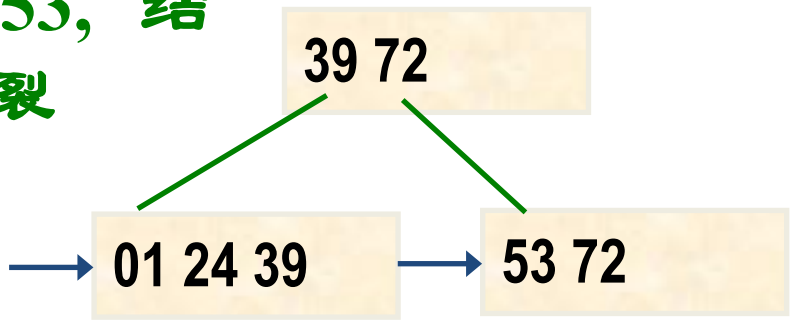
况类似，但在做根结点分裂时，必须创建新的父结点，作为树的新根。

- 例如，在一棵4阶B+树中的插入过程如下。

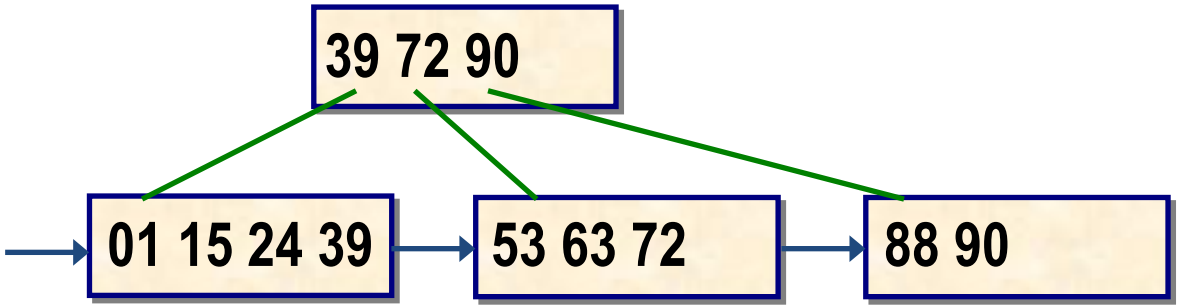
连续插入24, 72, 01, 39的B+树



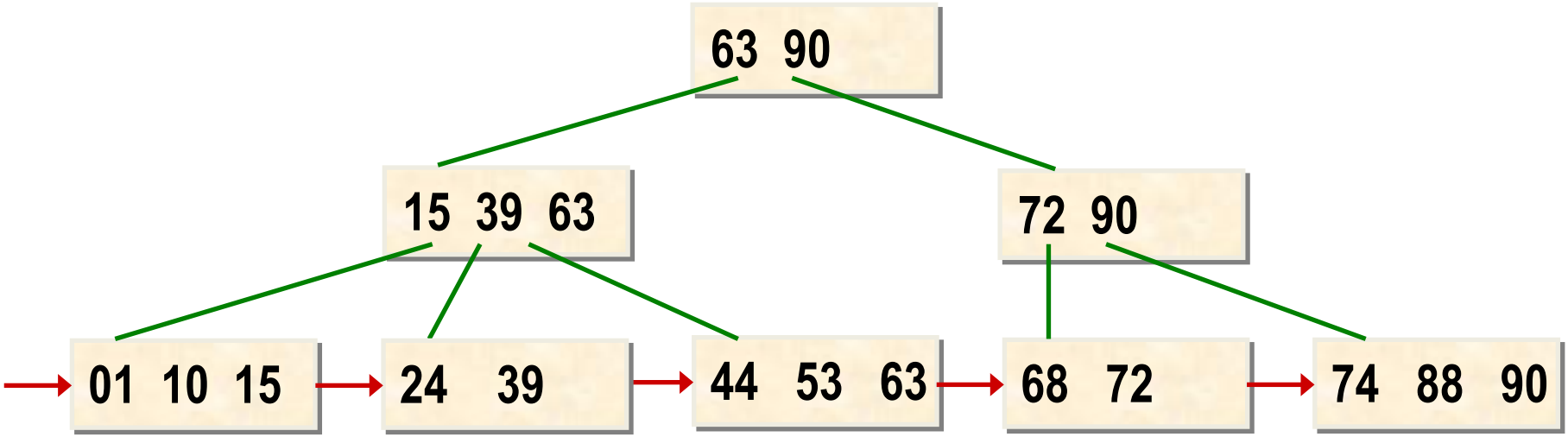
加入53，结点分裂



加入63, 90, 88, 15的B+树



加入10, 44, 68, 74的B+树

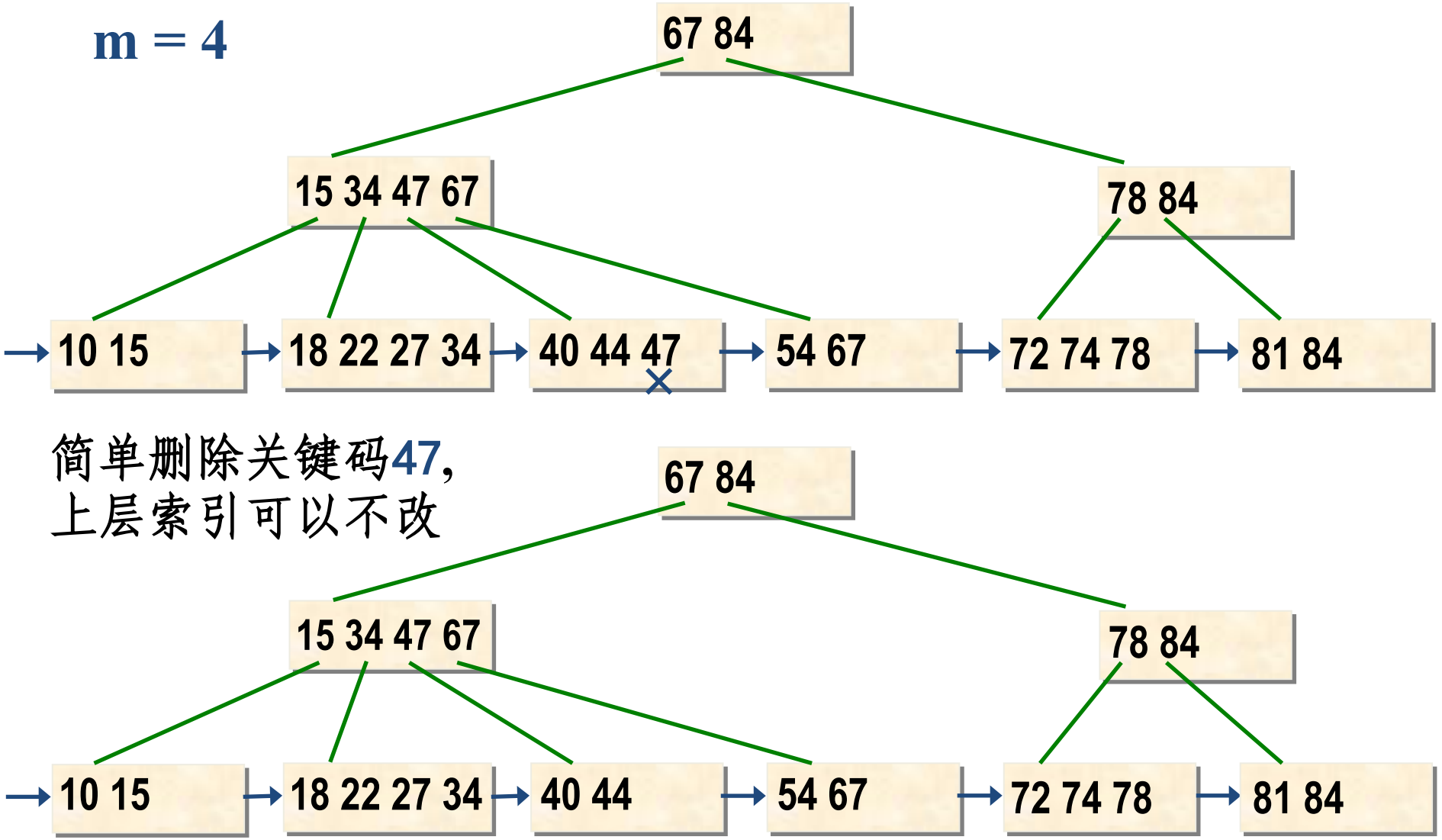


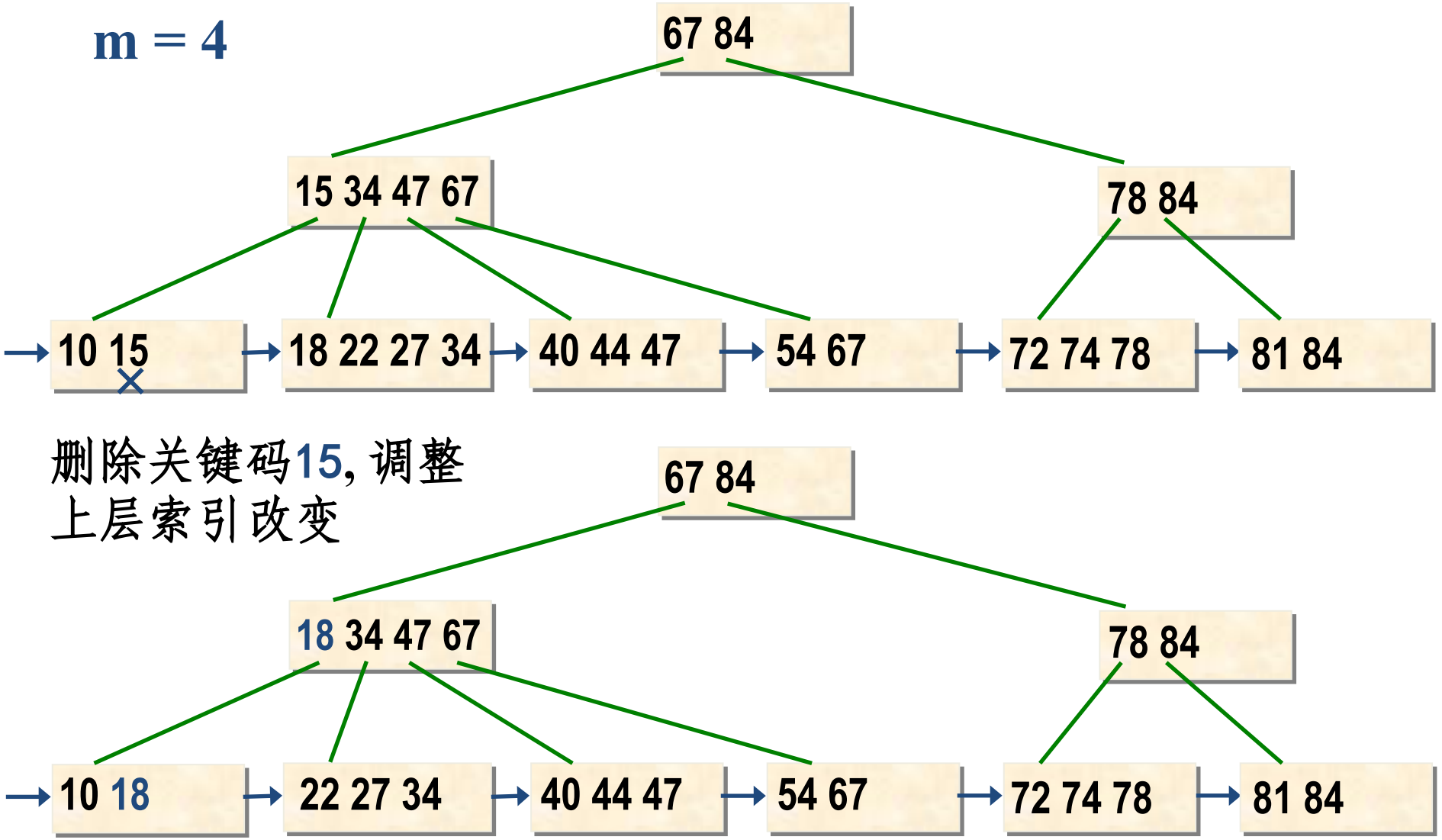
B+树的删除

- 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，结点中的索引项个数仍然不少于⌈m/2⌉，这属于简单删除，其上层索引可以不改变。

- 如果删除结点的最大关键码，但因在其上层的副本只起了一个引导搜索的“分界关键码”的作用，所以即使树中已经删除了关键码，但上层的副本仍然可以保留。

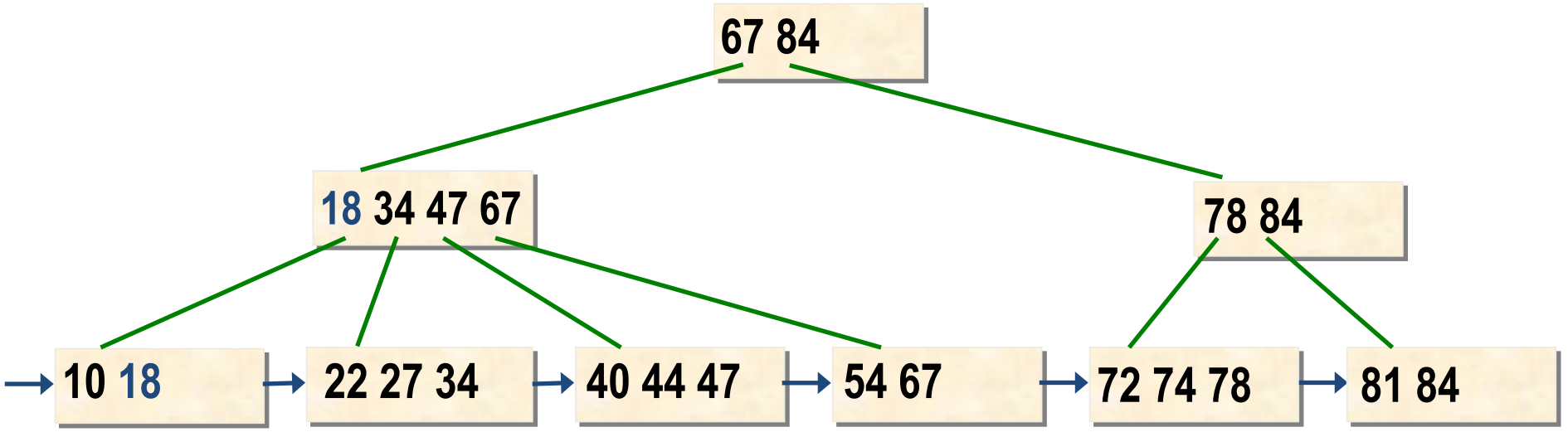
- 如果在叶结点中删除后，结点中索引项个数小于⌈m/2⌉，必须做结点的调整或合并工作。如果右兄弟结点中的子树棵数大于⌈m/2⌉，从右兄弟结点中移最左的（关键码-指针）索引项到这个被删关键码所在的结点，并修改上层的“分界关键码”的值。


m = 4
67 84
15 34 47 67
78 84
10 15
18 22 27 34
40 44 47
54 67
72 74 78
81 84
简单删除关键码47，
上层索引可以不改
67 84
15 34 47 67
78 84
10 15
18 22 27 34
40 44
54 67
72 74 78
81 84
从4阶B+树中做简单删除

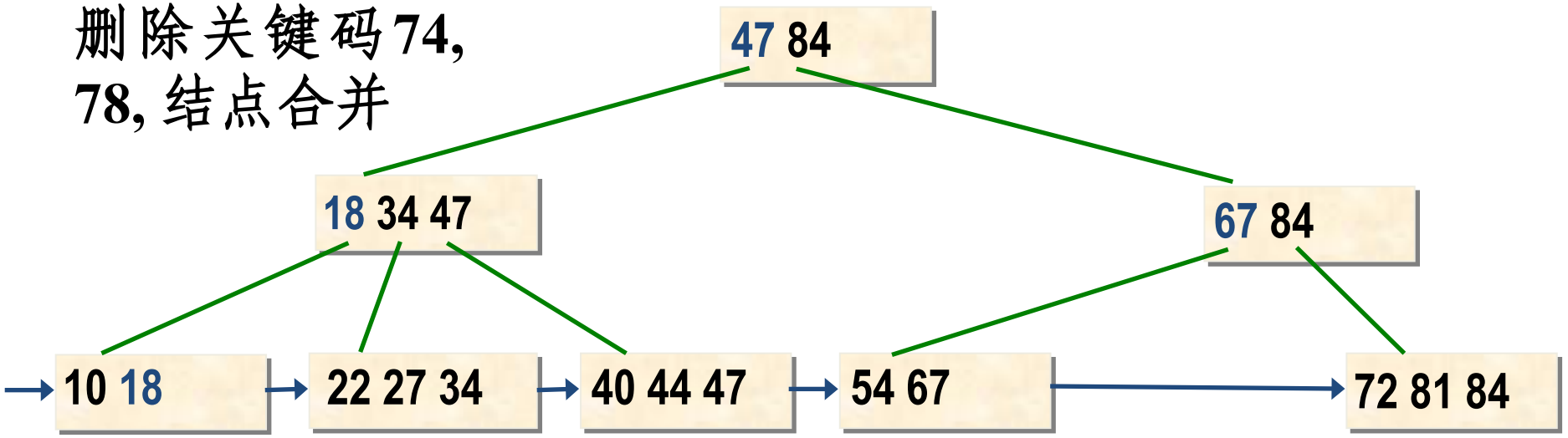


从4阶B+树中删除时的调整

- 如果右兄弟结点的关键码数已达到下限⌈m/2⌉，没有多余的关键码可以移入被删关键码所在的结点，必须进行结点的合并。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。
- 这种结点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关键码个数的下限⌈m/2⌉以下。这样将引起双亲结点的调整或合并。
- 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。



删除关键码74, 78, 结点合并



外排序

- 当待排序的记录数目特别多时，在内存中不能一次处理。必须把它们以文件的形式存放于外存，排序时再把它们一部分一部分调入内存进行处理。这样，在排序过程中必须不断地在内存与外存之间传送数据。这种基于外部存储设备（或文件）的排序技术就是外排序。

外排序的基本过程

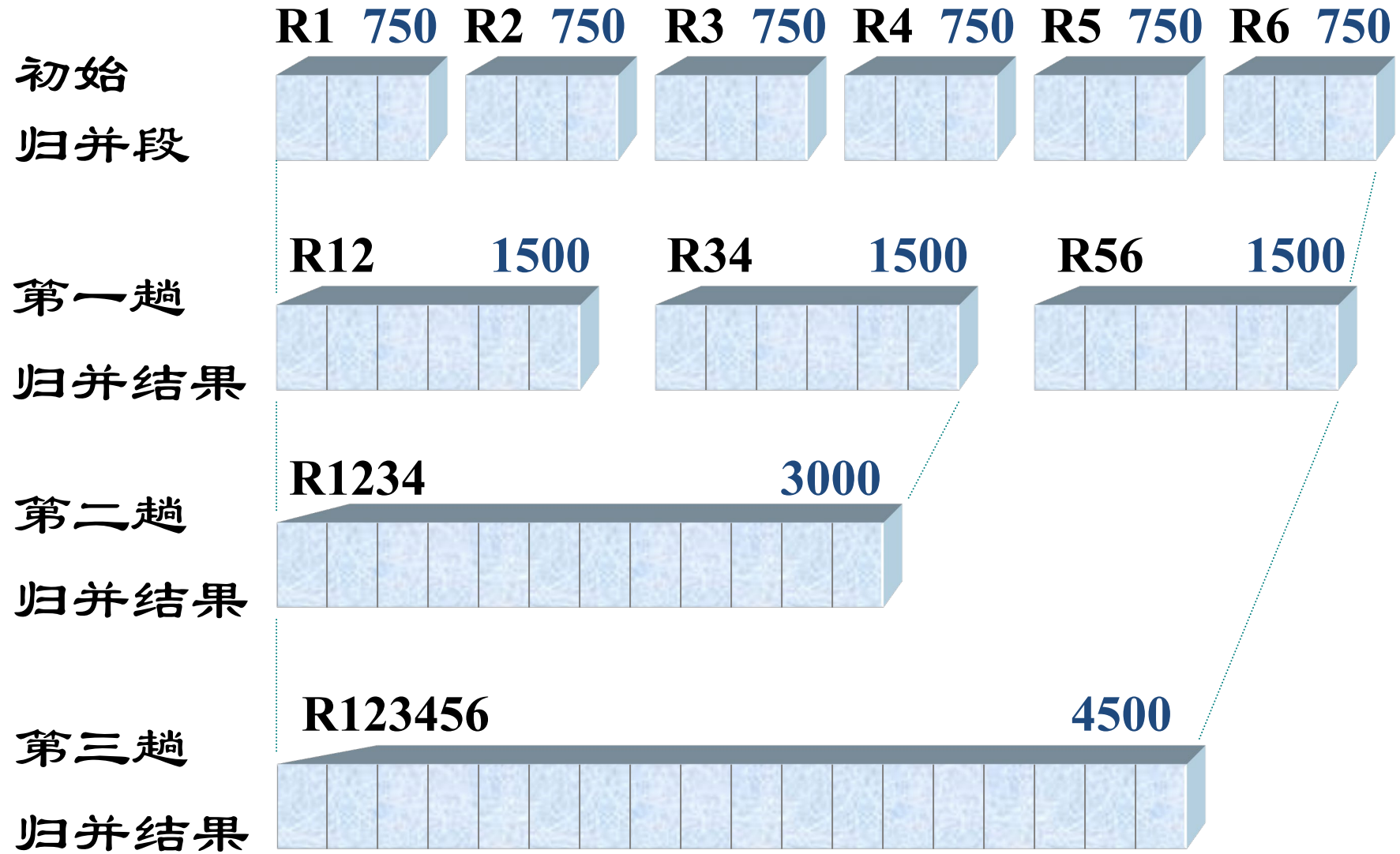
基于磁盘进行的排序大多使用归并排序方法。其排序过程主要分为两个阶段:

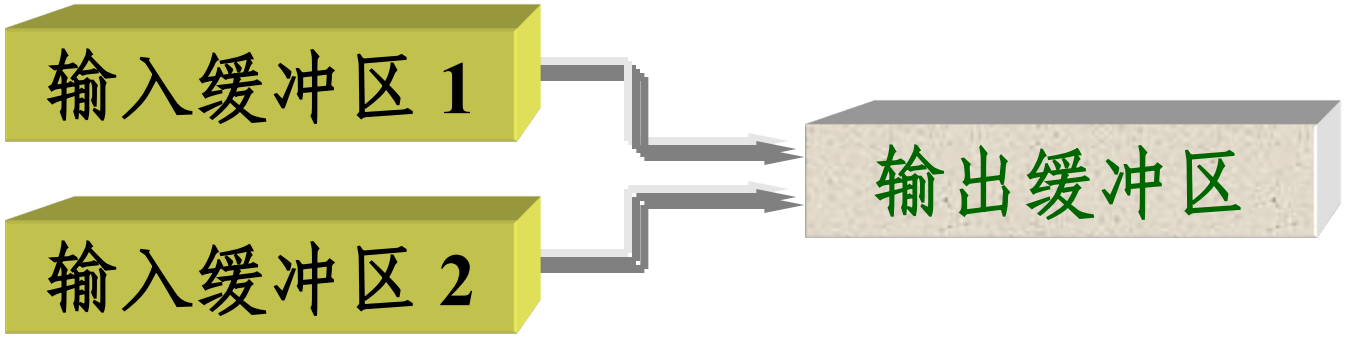
① 建立用于外排序的内存缓冲区。根据它们的大小将输入文件划分为若干段,用某种内排序方法对各段进行排序。经过排序的段叫做初始归并段 (Run)。当它们生成后就被写到外存中去。

② 按归并树模式,把①生成的初始归并段加以归并,一趟趟扩大归并段和减少归并段数,直到最后归并成一个大归并段为止。

- 示例：设有一个包含4500个记录的输入文件。现用一台其内存至多可容纳750个记录的计算机对该文件进行排序。输入文件放在磁盘上，磁盘每个页块可容纳250个记录，这样全部记录可存储在 4500 / 250 = 18 个页块中。输出文件也放在磁盘上，用以存放归并结果。
- 由于内存中可用于排序的存储区域能容纳750个记录，因此内存中恰好能存3个页块的记录。
- 在外排序一开始，把18块记录，每3块一组，读入内存。利用某种内排序方法进行内排序，形成初始归并段，再写回外存。总共可得到6个初始归并段。然后一趟一趟进行归并排序。

两路归并排序的归并树


R1 750
R2 750
R3 750
R4 750
R5 750
R6 750
初始
归并段
R12 1500
R34 1500
R56 1500
第一趟
归并结果
R1234 3000
第二趟
归并结果
R123456 4500
第三趟
归并结果



- 若把内存区域等份地分为 3 个缓冲区。其中的两个为输入缓冲区, 一个为输出缓冲区, 可以在内存中利用简单两路归并函数 merge( ) 实现两路归并。
- 首先, 从参加归并排序的两个输入归并段 R_1 和 R_2 中分别读入一块, 放在输入缓冲区1 和输入缓冲区2 中。然后在内存中进行两路归并，归并结果顺序存放到输出缓冲区中。

- 若总记录个数为 n，磁盘上每个页块可容纳 b 个记录，内存缓冲区可容纳 i 个页块，则每个初始归并段长度为 $len = i * b$，可生成 $m = \lceil n / len \rceil$ 个等长的初始归并段。
- 在做 2 路归并排序时，第一趟从 m 个初始归并段得到 $\lceil m/2 \rceil$ 个归并段，以后各趟将从 l ($l > 1$) 个归并段得到 $\lceil l/2 \rceil$ 个归并段。总归并趟数等于归并树的高度减一：$\lceil \log_2 m \rceil$。
- 估计两路归并排序时间 t_{ES} 的上界为：

$$t_{ES} = m*t_{IS} + d*t_{IO} + S*n*t_{mg}$$

t_{IS}为每个初始归并段进行内排序的时间；d为访问外存块的次数；

t_{mg}是取得一个记录的时间；S是归并趟数

- 对 4500 个记录排序的例子, 各种操作的计算时间如下:
  - ✓ 读18个输入块, 内部排序6段, 写18个输出块
    $= 6\ t_{IS} + 36\ t_{IO}$
  - ✓ 成对归并初始归并段 $R_1 \sim R_6$
    $= 36\ t_{IO} + 4500\ t_{mg}$
  - ✓ 归并两个具有1500个记录的归并段R_{12}和R_{34}
    $= 24\ t_{IO} + 3000\ t_{mg}$
  - ✓ 最后将 R_{1234} 和 R_{56} 归并成一个归并段
    $= 36\ t_{IO} + 4500\ t_{mg}$
- 合计 $t_{ES} = 6\ t_{IS} + 132\ t_{IO} + 12000\ t_{mg}$

- 由于 $t_{IO} = t_{seek} + t_{latency} + t_{rw}$，其中，$t_{seek}$和$t_{latency}$是机械动作，而$t_{rw}$、$t_{IS}$、$t_{mg}$是电子线路的动作，所以$t_{IO}$远远大于$t_{IS}$和$t_{mg}$。想要提高外排序的速度，应着眼于减少$d$。
- 若对相同数目的记录，在同样页块大小的情况下做 3 路归并或做 6 路归并(当然, 内存缓冲区的数目也要变化)，则可做大致比较：

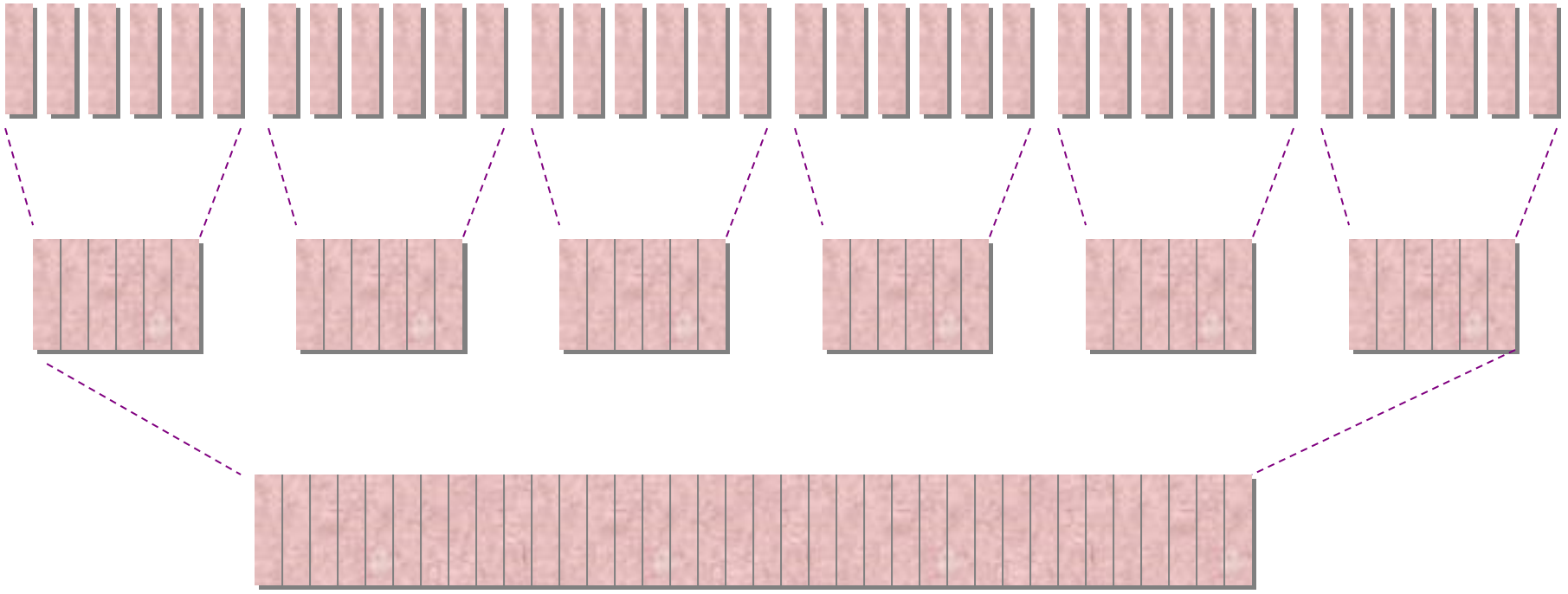
归并路数 k	总读写磁盘次数 d	归并趟数 S
2	132	3
3	108	2
6	72	1

- 增大归并路数, 可减少归并趟数, 从而减少总读写磁盘次数d。

- 对 m 个初始归并段, 做 k 路平衡归并, 一趟可将 m 个初始归并段归并为 $l=\lceil m/k \rceil$ 个归并段, 以后每一趟归并将 l 个归并段归并成 $l=\lceil l/k \rceil$ 个归并段，直到最后形成一个大的归并段为止。归并趟数$S=\lceil \log_k m \rceil$= 树的高度减一 。

k路平衡归并 (k-way Balanced merging)

- 做 k 路平衡归并时, 如果有 m 个初始归并段, 则相应的归并树有 $\lceil \log_k m \rceil + 1$ 层, 需要归并 $\lceil \log_k m \rceil$ 趟。下图给出对有 36 个初始归并段的文件做 6 路平衡归并时的归并树。



- 做内部k路归并时，在k个记录中选择最小者，需要顺序比较 $k-1$ 次。每趟归并n个记录需要做 $(n-1)*(k-1)$ 次比较，S 趟归并总共需要的比较次数为：

$$S*(n-1)*(k-1) = \lceil \log_k m \rceil * (n-1) * (k-1)$$
$$= \lceil \log_2 m \rceil * (n-1) * (k-1) / \lceil \log_2 k \rceil$$

- 在初始归并段个数 m 与记录个数 n 一定时，$\lceil \log_2 m \rceil*(n-1) = \text{const}$，而 $(k-1) / \lceil \log_2 k \rceil$ 随着k的增大而增大。因此，增大归并路数 k，会使得内部归并的时间增大。

- 使用“败者树”从k个归并段中选最小者，当k较大时 $(k \geq 6)$，选出排序码最小的记录只需比较$\lceil \log_2 k \rceil$次。

$$S*(n-1)*\lceil \log_2 k \rceil = \lceil \log_k m \rceil * (n-1) * \lceil \log_2 k \rceil$$
$$= \lceil \log_2 m \rceil * (n-1) * \lceil \log_2 k \rceil / \lceil \log_2 k \rceil$$
$$= \lceil \log_2 m \rceil * (n-1)$$

- 排序码比较次数与 k 无关，总的内部归并时间不会随 k 的增大而增大。
- 下面讨论利用败者树在 k 个输入归并段中选择最小者，实现归并排序的方法。

- 败者树是一棵完全二叉树。其中
  - 每个叶结点存放各归并段在归并过程中当前参加比较的记录;
  - 每个非叶结点存放它两个子女结点中记录排序码大的结点(即败者); （假设需要按从小到大排序）

- 在根结点的上一层另外增加一个结点，存放树中当前记录排序码最小的结点 (最小记录)。

- 败者树与胜者树的区别在于一个选择了败者(排序码大者), 一个选择了胜者(排序码小者)。

示例：设有 5 个初始归并段，它们中各记录的排序码分别是（增加一个关键码为∞的记录表示末尾）：

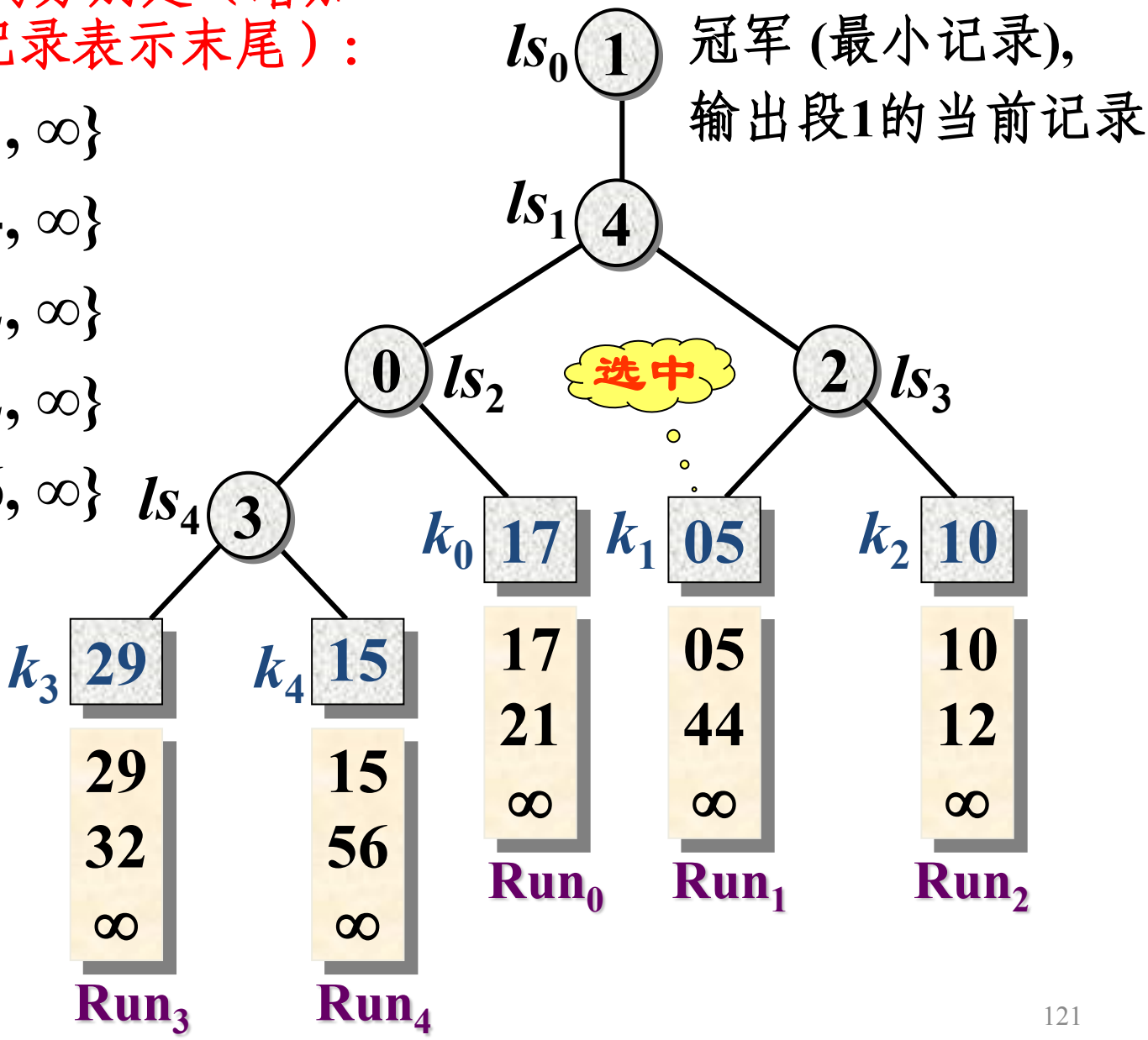
Run0: {17, 21, ∞}

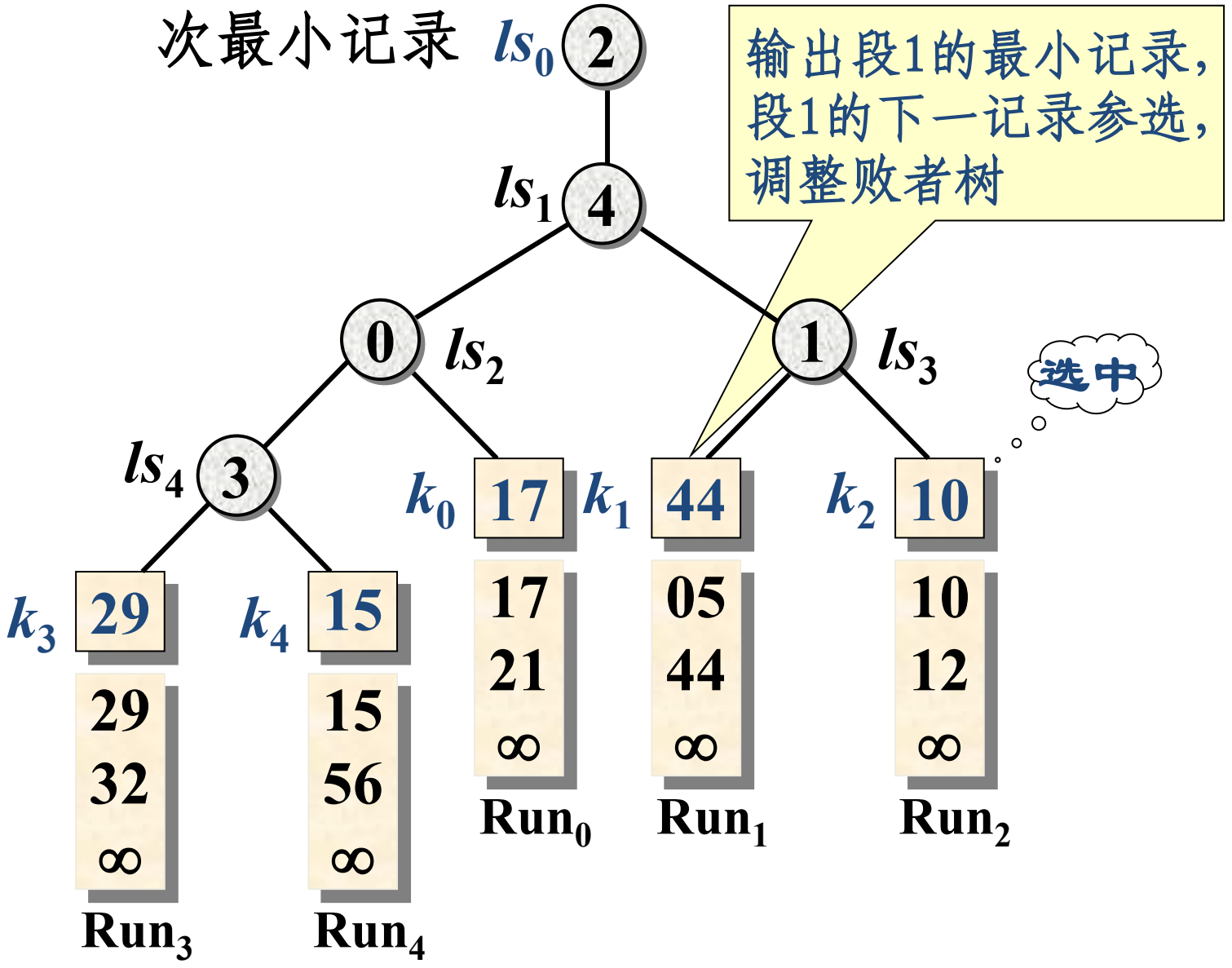
Run1: {05, 44, ∞}

Run2: {10, 12, ∞}

Run3: {29, 32, ∞}

Run4: {15, 56, ∞}




次最小记录
ls_0 2
输出段1的最小记录，
段1的下一记录参选，
调整败者树
ls_1 4
0 ls_2
1 ls_3
选中
ls_4 3
k_0 17
k_1 44
k_2 10
k_3 29
k_4 15
17
21
∞
Run$_0$
05
44
∞
Run$_1$
10
12
∞
Run$_2$
29
32
∞
Run$_3$
15
56
∞
Run$_4$

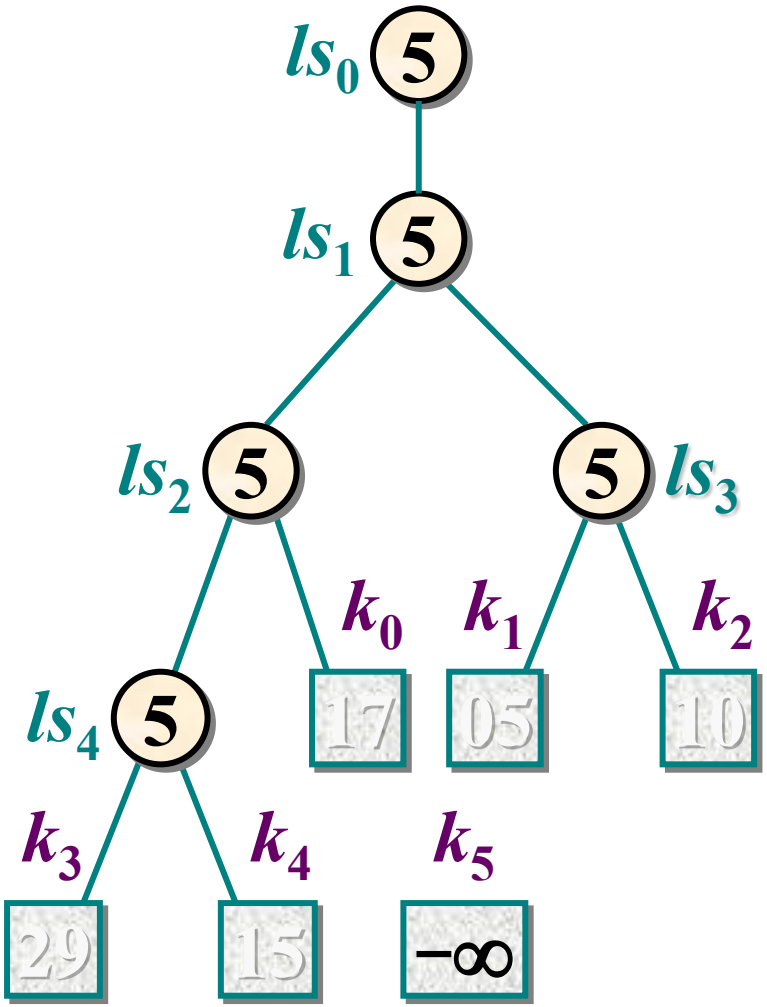
- 败者树的高度为$\lceil \log_2 k \rceil+1$，在每次调整、找下一个具有最小排序码记录时，最多做$\lceil \log_2 k \rceil$次排序码比较。
- 在内存中应为每一个归并段分配一个输入缓冲区，其大小应能容纳一个页块的记录，编号与归并段号一致。每个输入缓冲区应有一个指针，指示当前参加归并的记录。
- 在内存中还应设立一个输出缓冲区，其大小相当于一个页块大小。它也有一个缓冲区指针，指示当前可存放结果记录的位置。每当一个记录 *i* 被选出，就执行**OutputRecord(*i*)**操作，将记录存放到输出缓冲区中。

- 在实现利用败者树进行多路平衡归并算法时，把败者树的叶结点和非叶结点分开定义。
- 败者树叶结点key[]有k+1个，key[0]到key[k−1]存放各归并段当前参加归并的记录的排序码，key[k]是辅助工作单元，在初始建立败者树时使用：存放一个最小的在各归并段中不可能出现的排序码：−MaxValue。
- 败者树非叶结点loser[]有k个，其中loser[1]到loser[k−1]存放各次比较的败者的归并段号，loser[0]中是最后胜者所在归并段号。另外还有一个存放各归并段参加归并记录的数组$r[k]$。

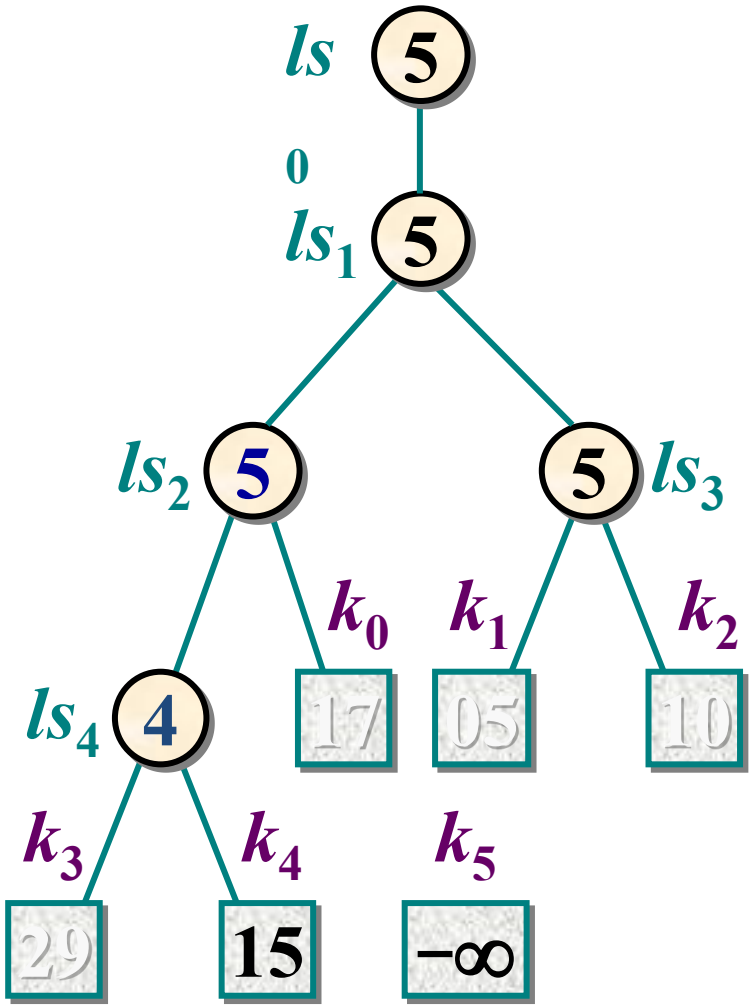
- 每选出一个当前排序码最小的记录，就需要在将它送入输出缓冲区之后，从相应归并段的输入缓冲区中取出下一个参加归并的记录，替换已经取走的最小记录，再从叶结点到根结点，沿某一特定路径进行调整，将下一个排序码最小记录的归并段号调整到**loser[0]**中。

- 段结束标志**MaxNum**升入**loser[0]**，排序完成。

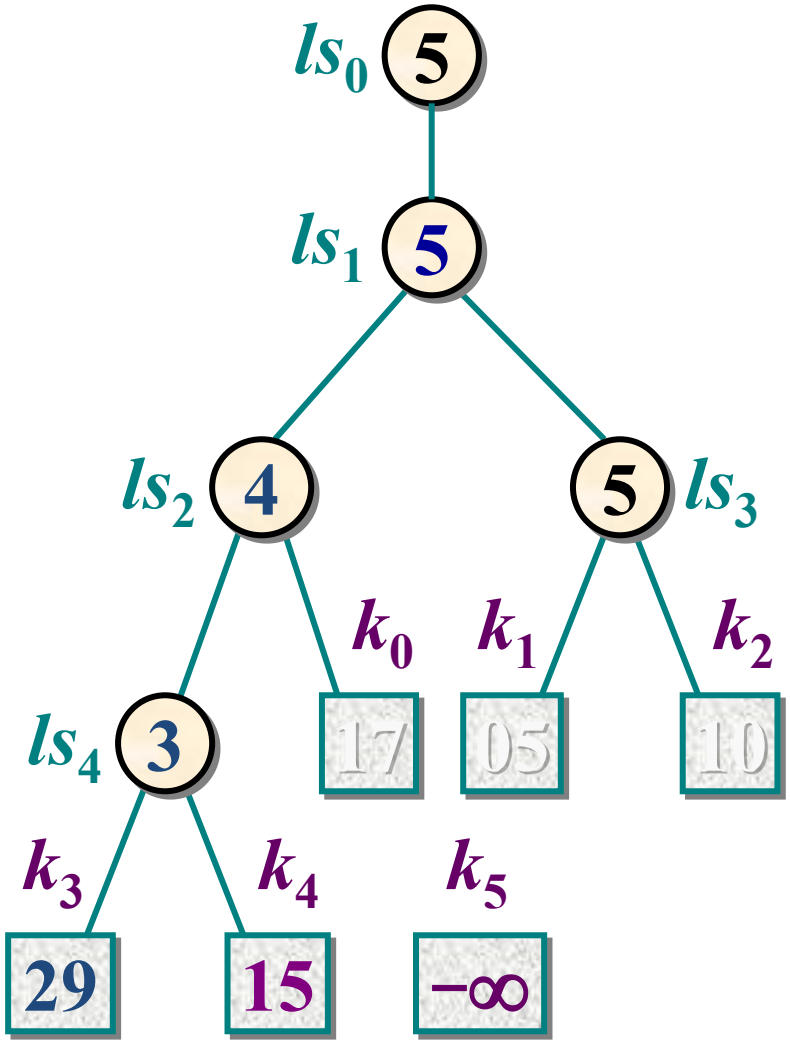
利用败者树进行 5 路平衡归并的过程



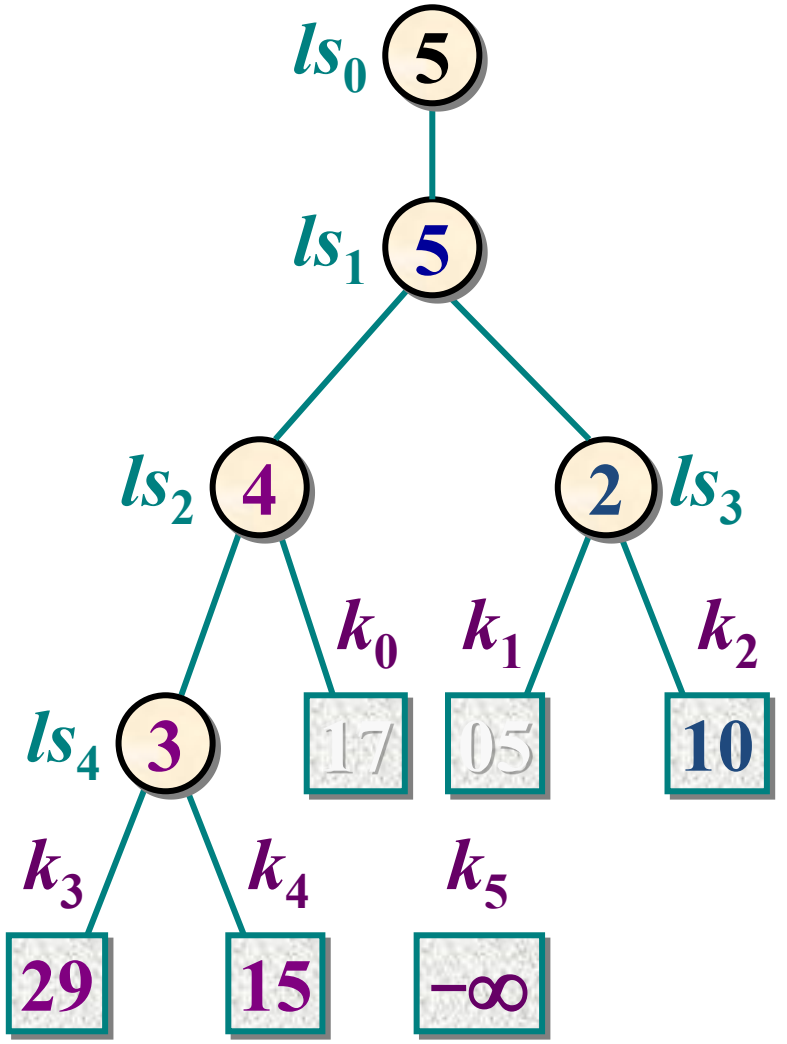
(1) 初始状态



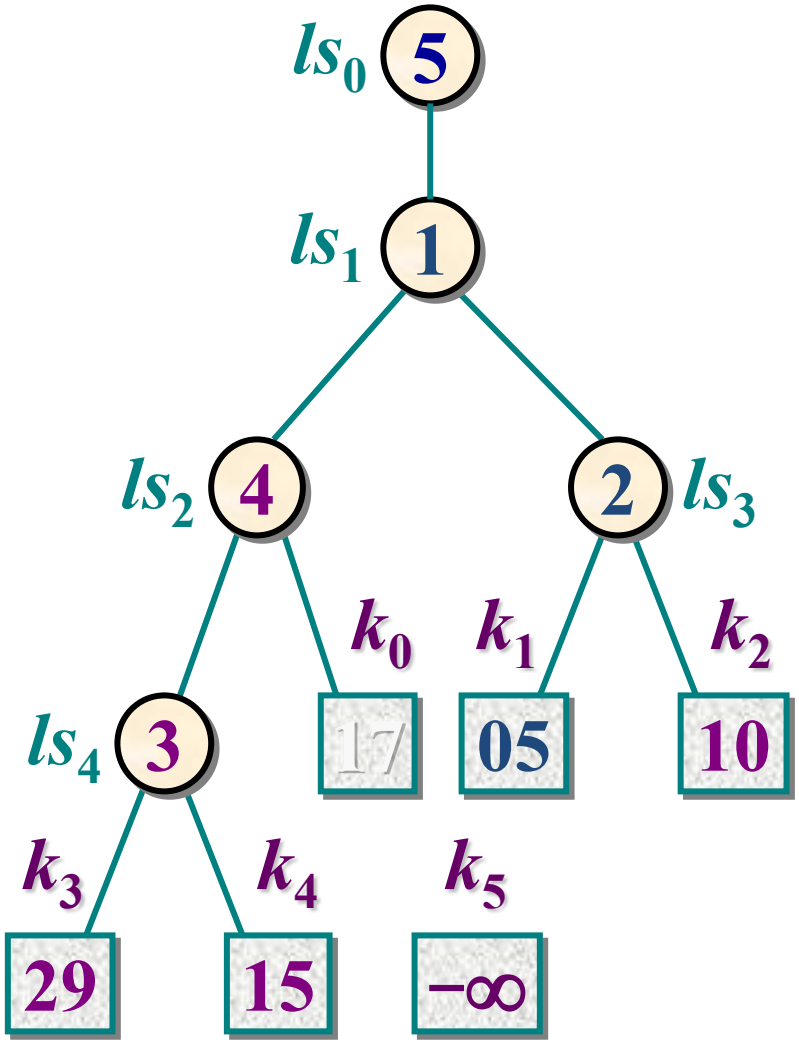
(2) 加入15, 调整



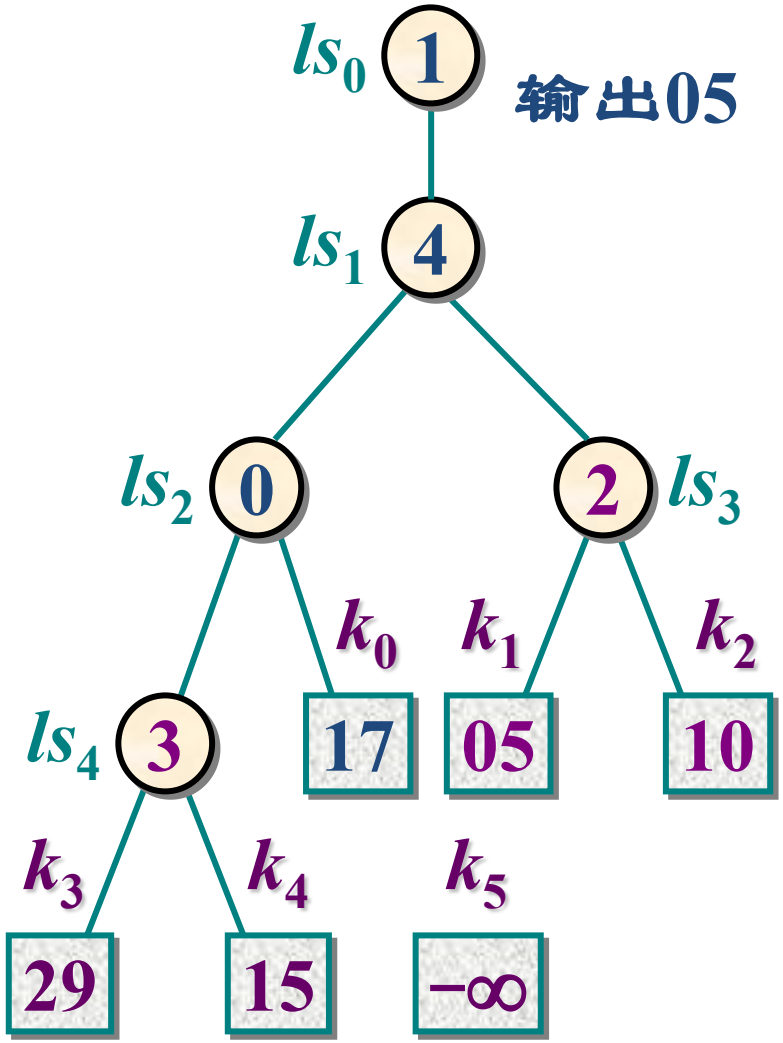
(3)加入29, 调整



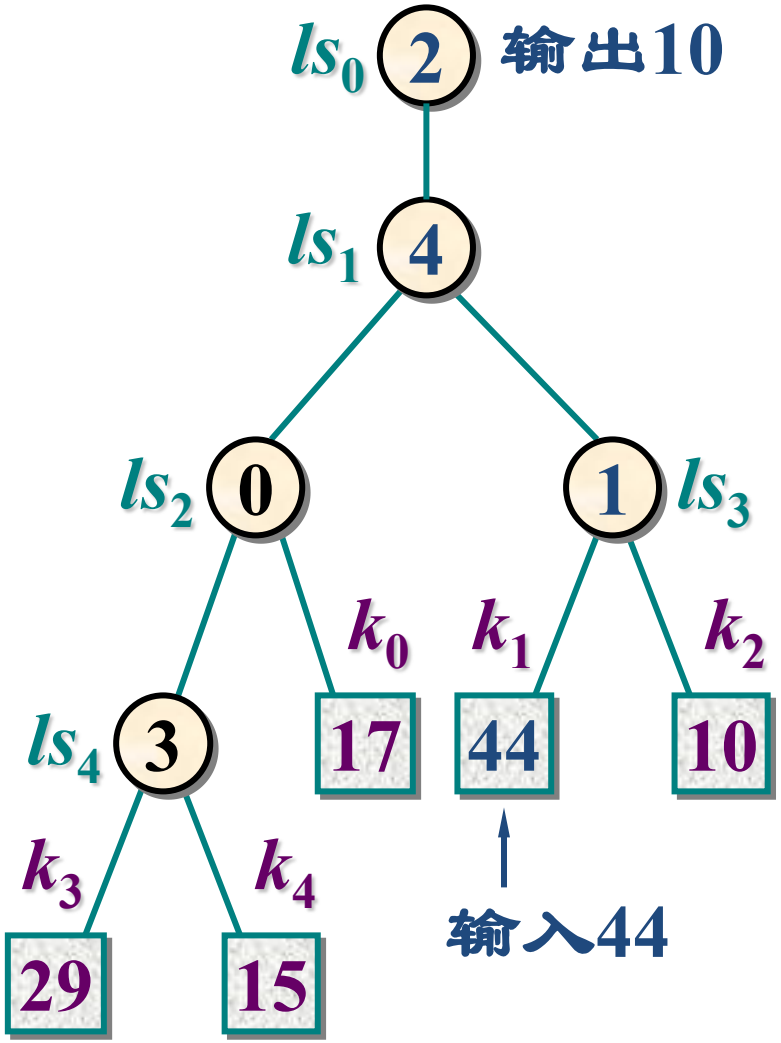
(4) 加入10, 调整



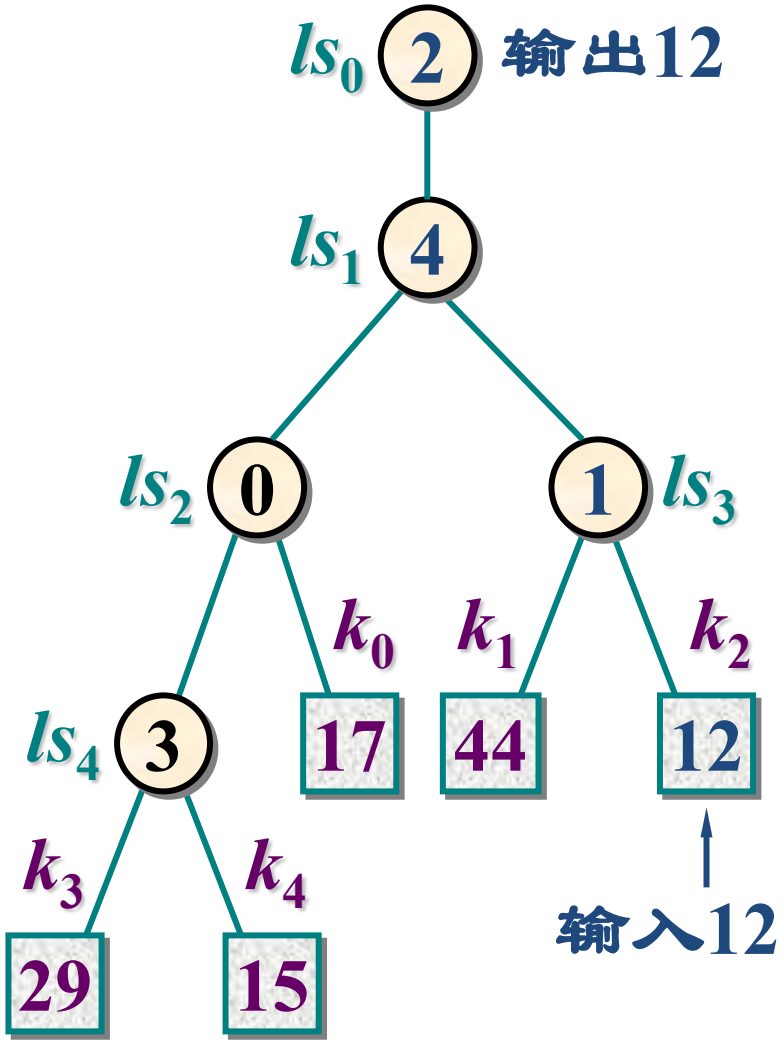
(5) 加入05, 调整



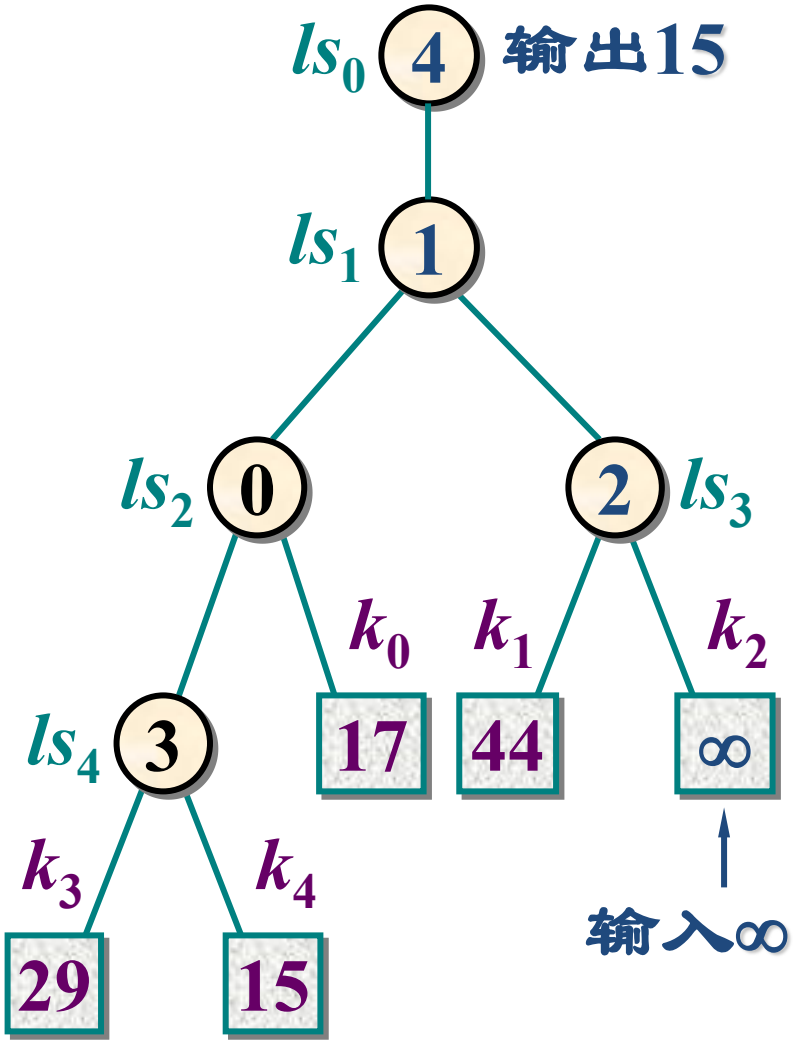
(6) 加入17, 调整



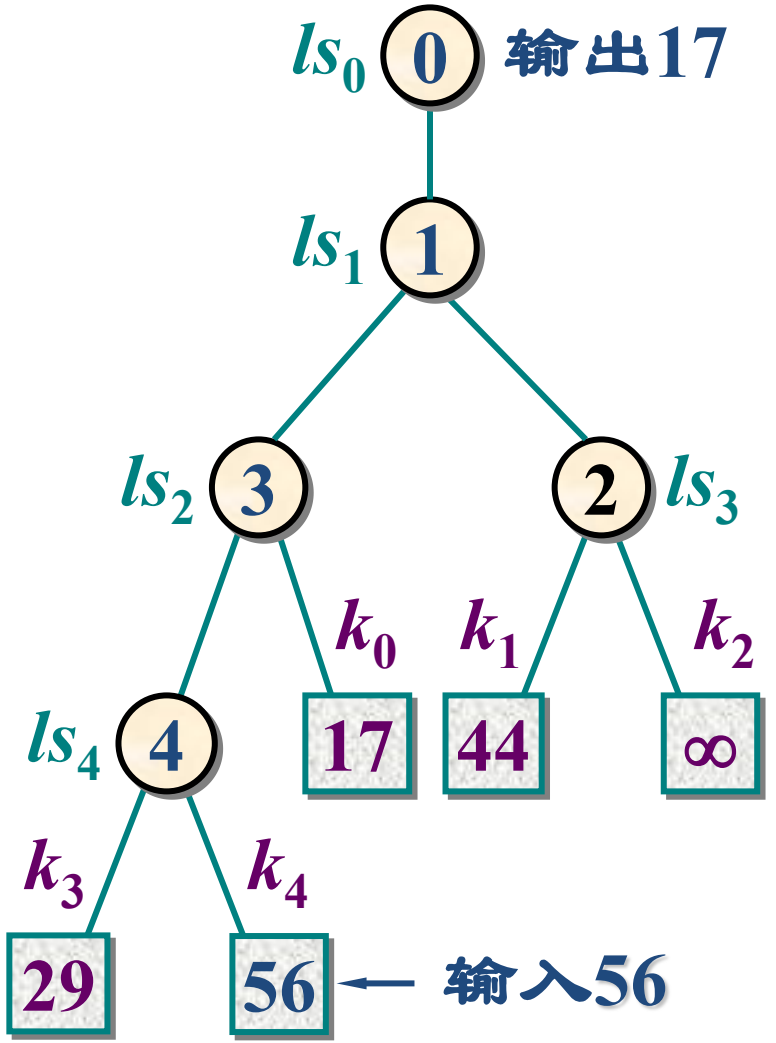
(7) 输出05后调整



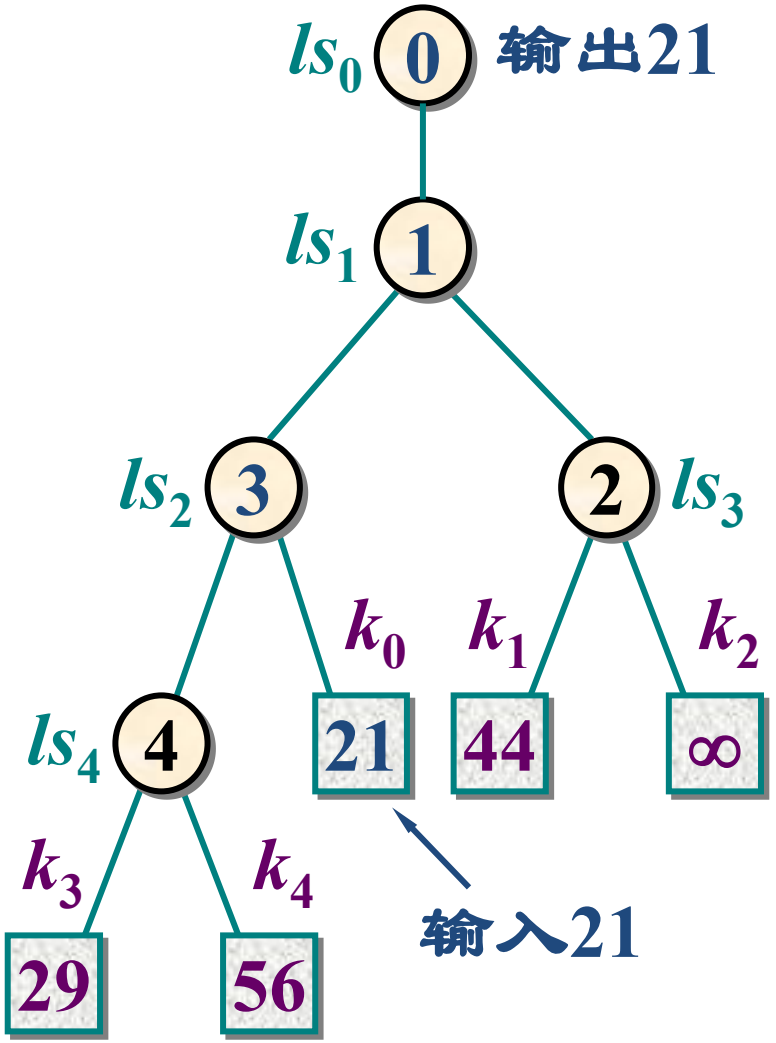
(8) 输出10后调整



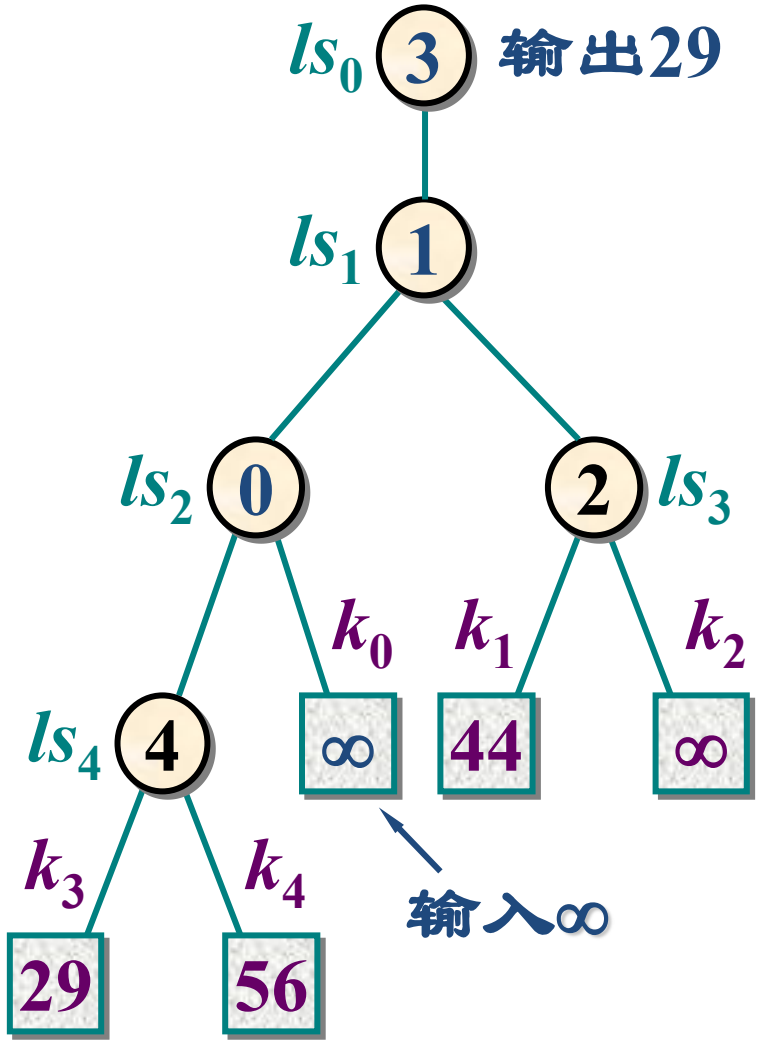
(9) 输出12后调整



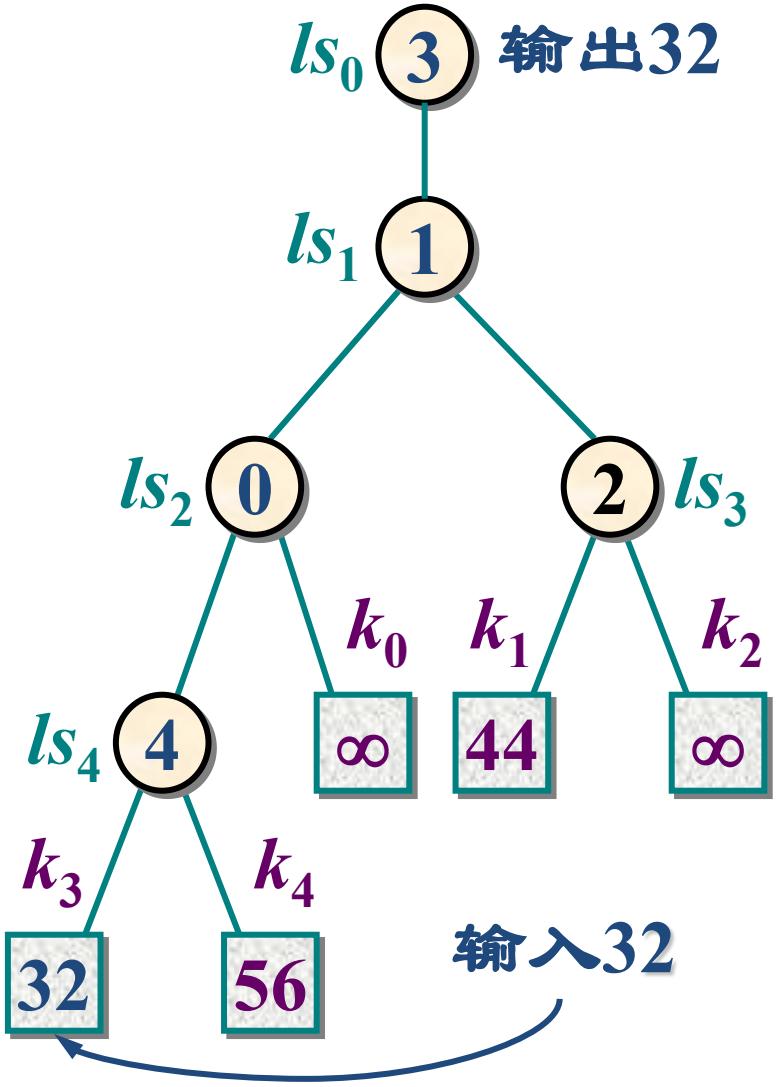
(10) 输出15后调整



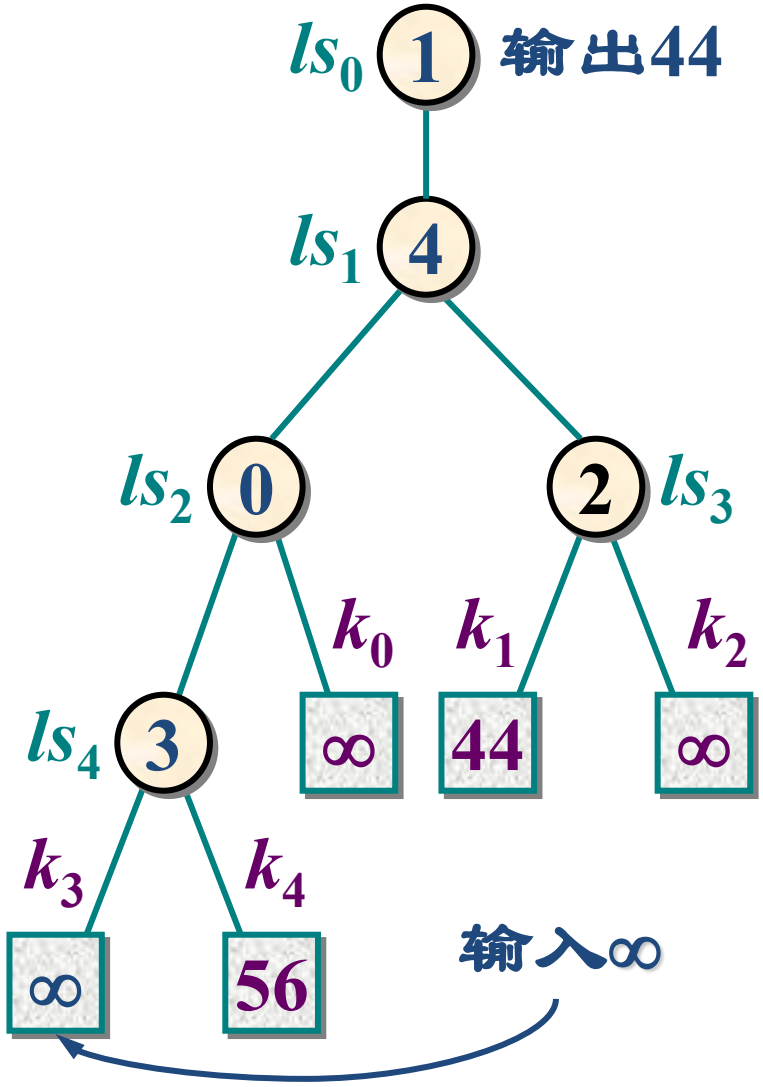
(11) 输出17后调整



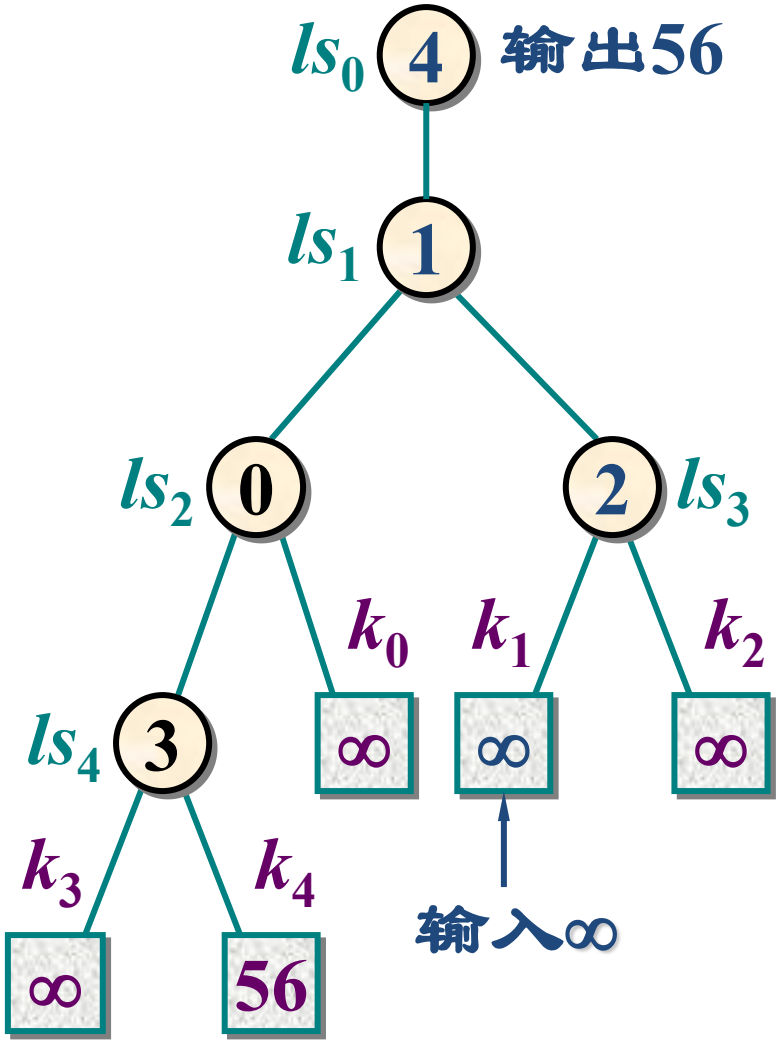
(12) 输出21后调整



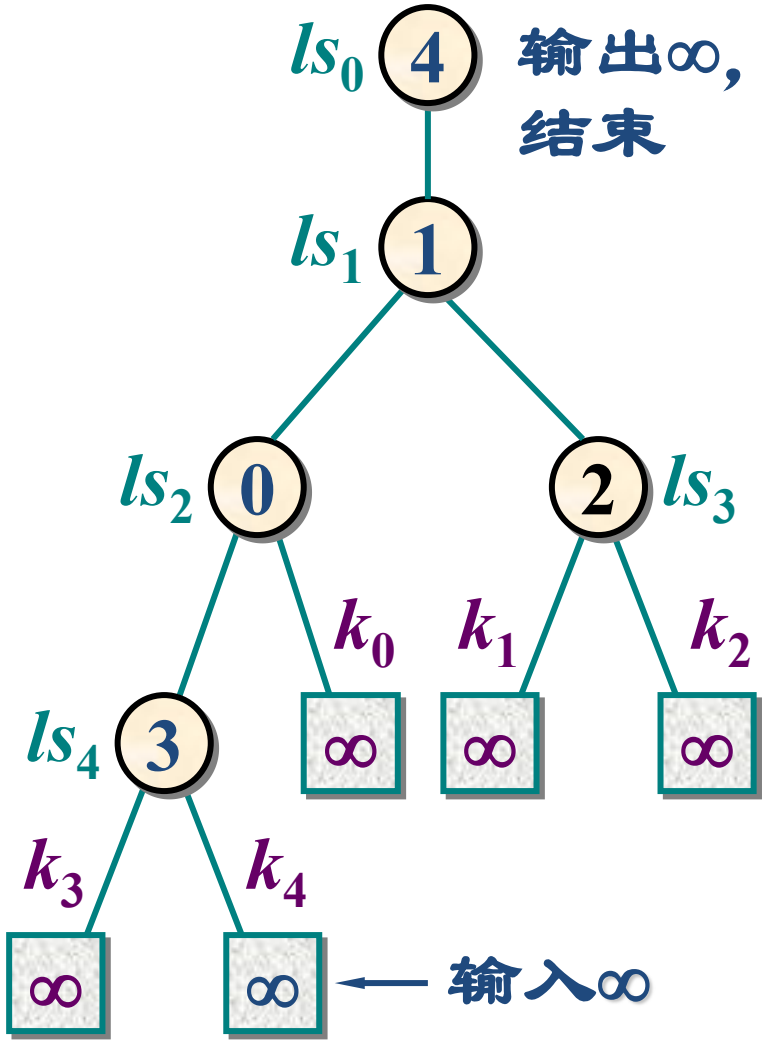
(13) 输出29后调整



(14) 输出32后调整



(15) 输出44后调整



(16) 输出56后调整

- 归并路数 k 不是越大越好。归并路数 k 增大，相应需增加输入缓冲区个数。如果可供使用的内存空间不变，势必要减少每个输入缓冲区的容量，使内外存交换数据的次数增大。