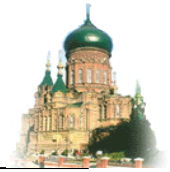
8.7 B-树和B+树

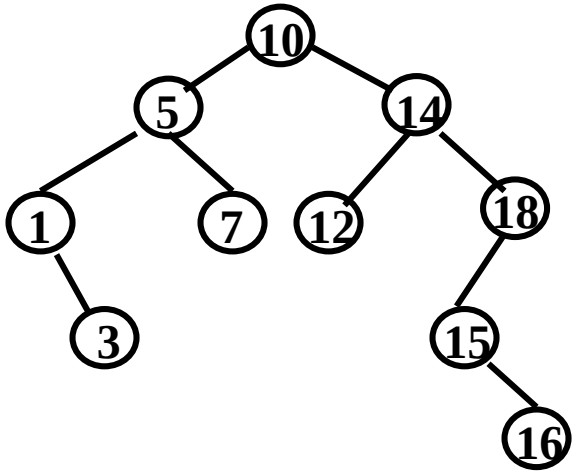
- 当查找表的大小超过内存容量时，不可能一次调入内存，要多次访问外存。硬盘的访问速度慢。主要矛盾变为减少访问外存次数。
- 从磁盘等辅助存储设备上去读取这些查找树结构中的结点，每次只能根据需要读取一个结点
- 用二叉树组织文件，若在查找时一次调入一个结点进内存，当文件的记录个数为100000时,要找到给定关键字的记录，平均需访问外存17次($\log_2 100000$)
- 因此，BST树性能就不是很高。



8.7 B-树和B+树

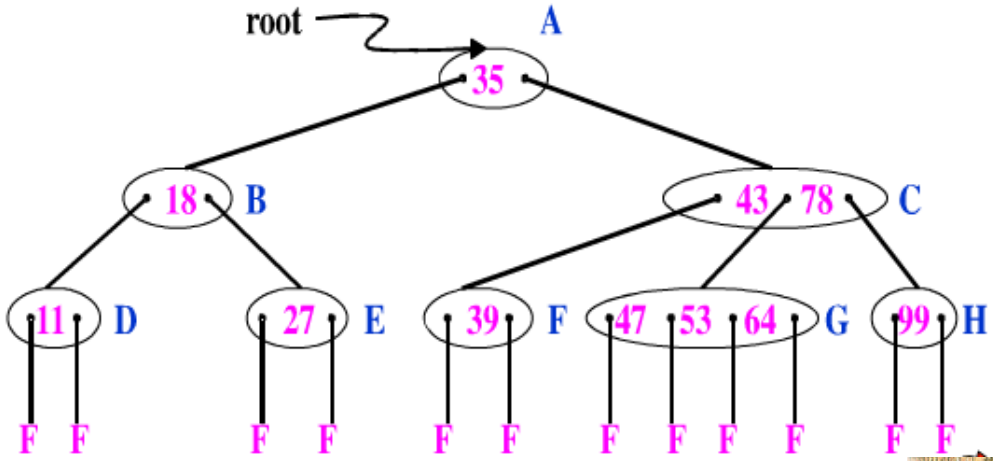
- 二叉查找树
  - 节点内一个关键字（2个叉）

```
typedef struct  celltype  {
        records   data ;
        struct celltype  *lchild,*rchild ;
} BSTNode;
```



- 四叉查找树
  - 最少1个关键字（2个叉）
  - 最多3个关键字（4个叉）
  - 节点内关键字有序

```
typedef struct  celltype  {
        records   data[3] ;
        struct celltype  *child[4];
        int num;
} BSTNode;
```



8.7 B-树和B+树

- ***m*-路查找树：**

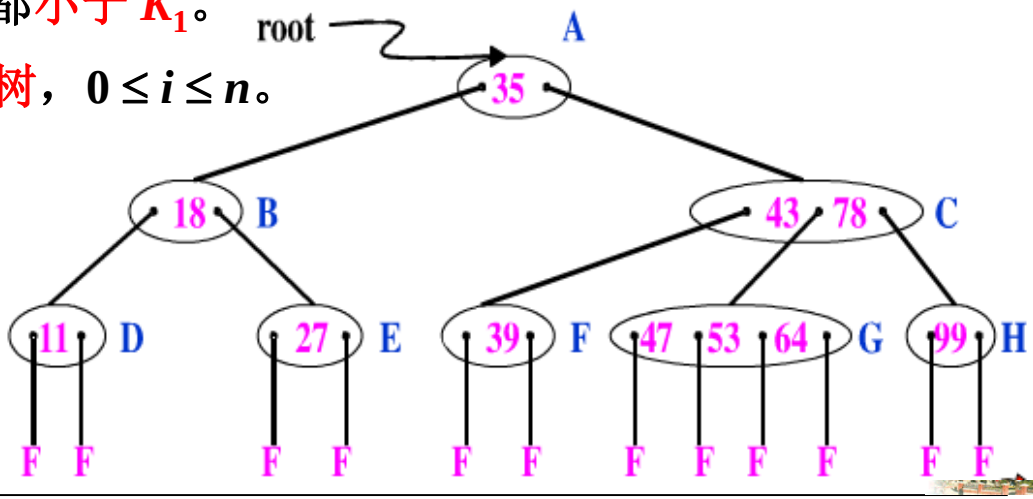
一棵*m*-路查找树或者是一棵空树，或者是满足如下性质的树：

- 根结点最多有 m 棵子树，并具有如下的结构：

$$A_0, (K_1, A_1), (K_2, A_2), \ldots, (K_i, A_i), \ldots, (K_n, A_n)$$

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

- 子树 A_i 中所有的关键字值都小于K_{i+1}而大于K_i，$0 < i < n$。
- 子树 A_n 中所有的关键字值都大于K_n；
- 子树 A_0 中的所有关键字值都小于 K_1。
- 每棵子树 A_i 也是 m -路查找树，$0 \le i \le n$。



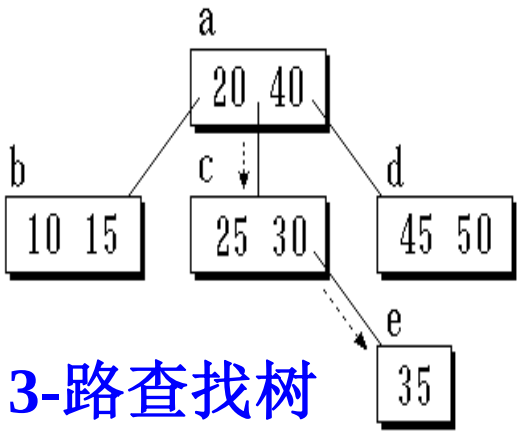
8.7 B-树和B+树

设 m -路查找树高度为h，最多有多少个关键字？

其结点数量的最大值是：

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

由于每个结点最多包含$m-1$个关键字，因此在一棵高度为h的m-路查找树中，最多可容纳m^h-1个关键字。



3-路查找树

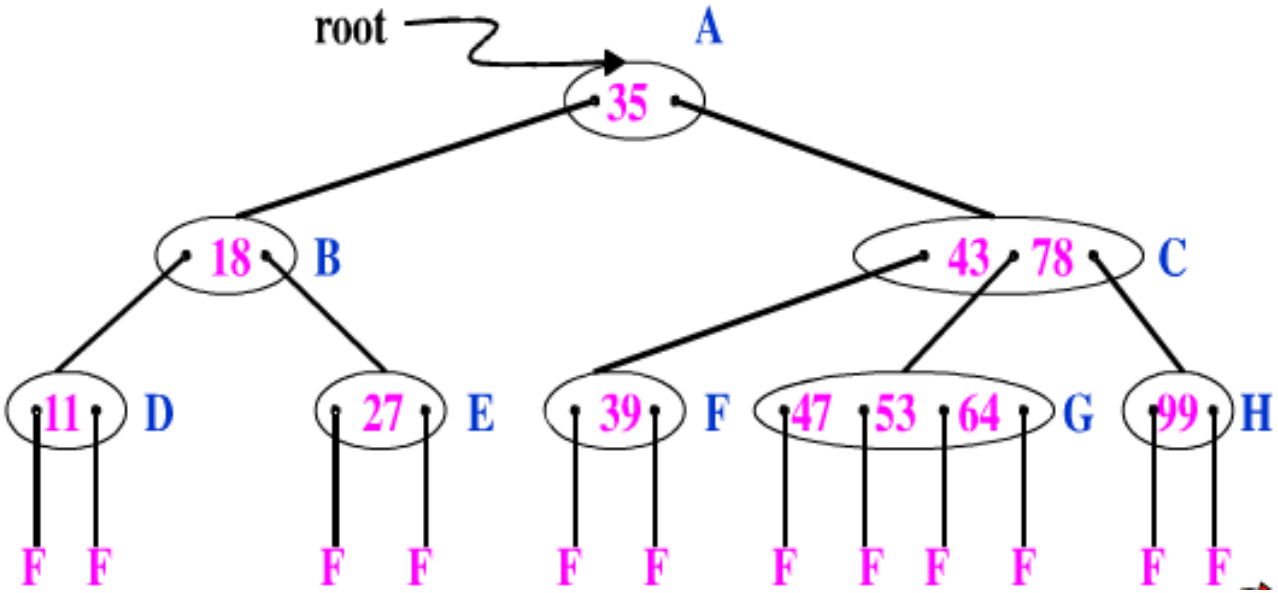
【例】对于$h=3$的二叉树，最多容纳7个关键字；

高度 h=3 的3路查找树，关键码最大个数为 $3^3-1 = 26$。

而对于h=3的200-路查找树，最多有$m^h-1=8*10^6-1$个关键字。

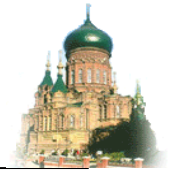
8.7 B-树和B+树

- 要想查找效率高，考虑两个方向：
  - 树的层数不能太多
    - 除根节点外，所有结点至少有⌈*m*/2⌉棵子树



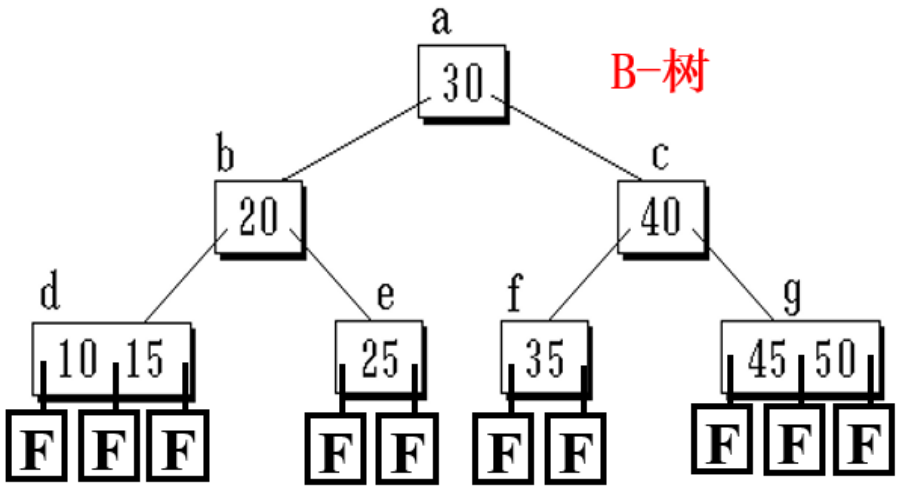
8.7 B-树和B+树

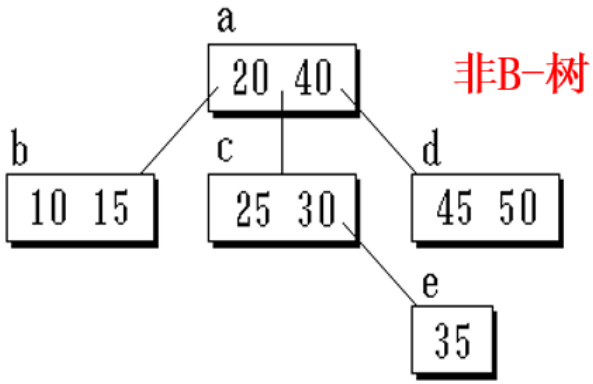
- 要想查找效率高，考虑两个方向：
  - 树的层数不能太多
  - 树要尽可能平衡
- 在AVL树在结点高度上采用相对平衡的策略，使其平均性能接近于BST的最好情况下的性能。
- 如果保持查找树在高度上的绝对平衡，而允许查找树结点的子树个数（分支个数）在一定范围内变化，能否获得很好的查找性能呢？
- 基于这样的想法，1970年，R.Bayer设计了许多在高度上保持绝对平衡，而在宽度上保持相对平衡的查找结构
- 如B-树及其各种变形结构，这些查找结构不再是二叉结构，而是m-路查找树（*m-way search tree*），且以其子树保持等高为其基本性质，在实际中都有着广泛的应用。



8.7 B-树和B+树

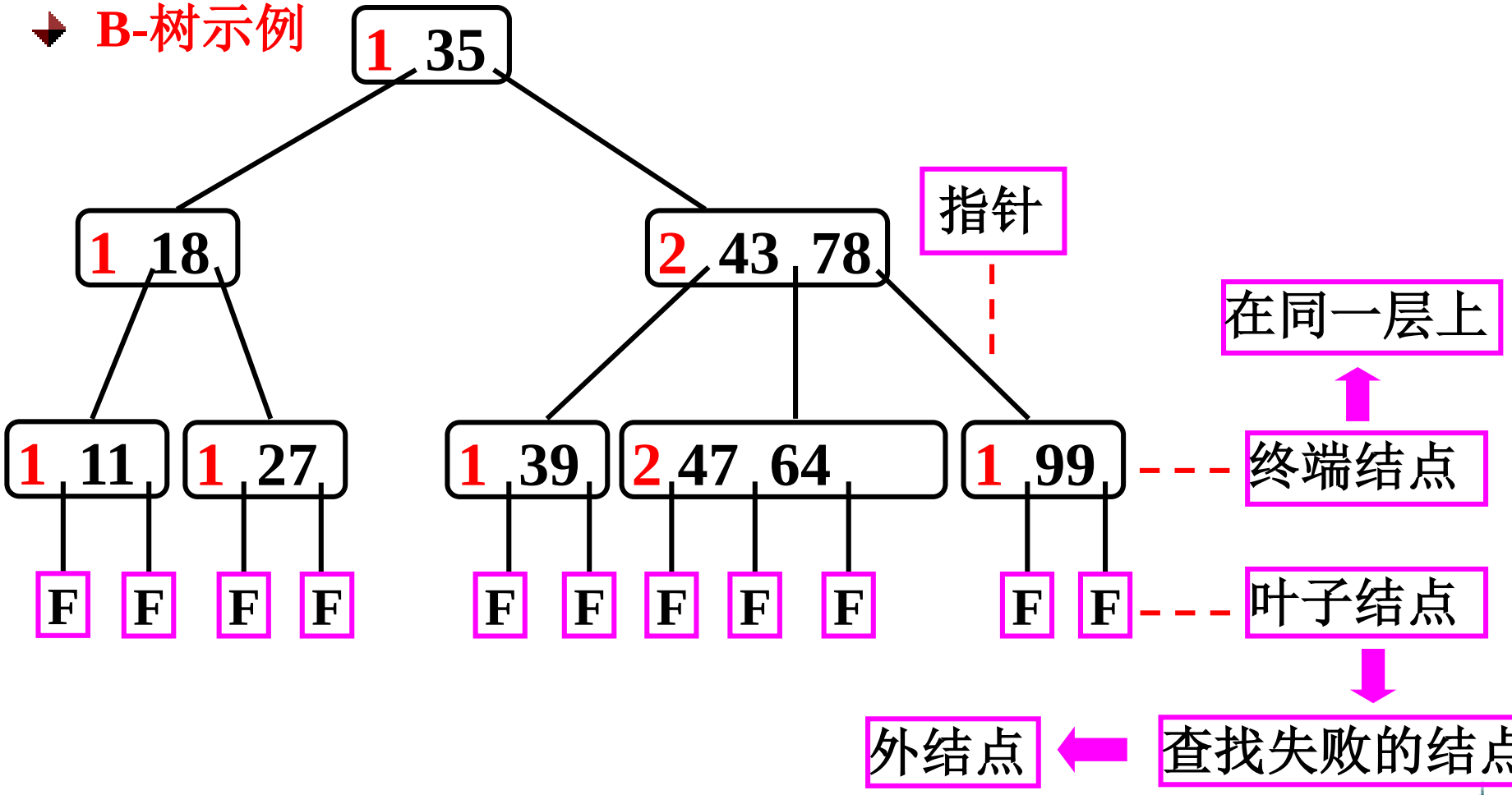
- **B-树**：一棵 *m* 阶**B-树**是一棵 *m*-路查找树，它或者是空树，或者是满足下列性质：
  - 树中每个结点至多有*m*棵子树；
  - 根结点至少有 2 棵子树； **(2~m)**
  - 除根结点和失败结点外，所有结点至少有⌈*m*/2⌉棵子树；⌈*m*/2⌉~m
  - 所有的终端结点和叶子结点（失败结点）都位于同一层。



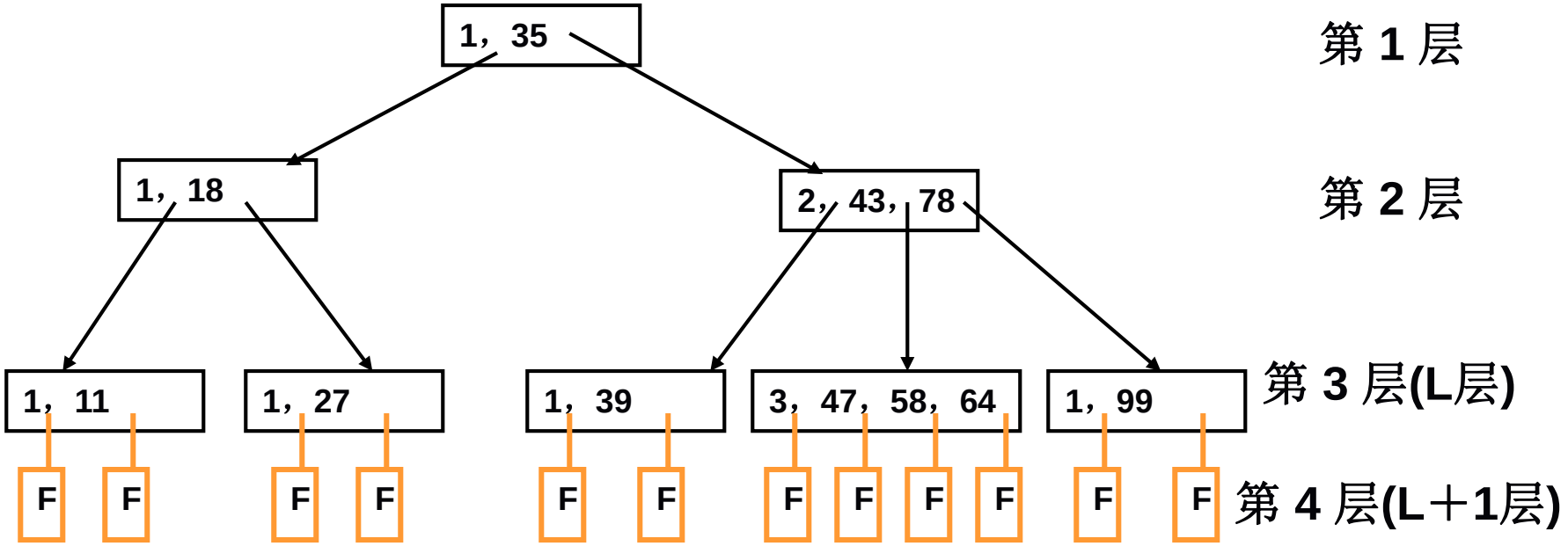


8.7 B-树和B+树

- B-树示例

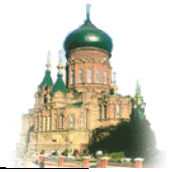


例： m = 4 阶 B- 树。除根结点和叶子结点之外，每个结点的儿子个数为 m/2 = 2 个~4个；结点的关键字个数为 1~3 。该 B- 树的深度为 4。叶子结点都在第 4 层上。



8.7 B-树和B+树

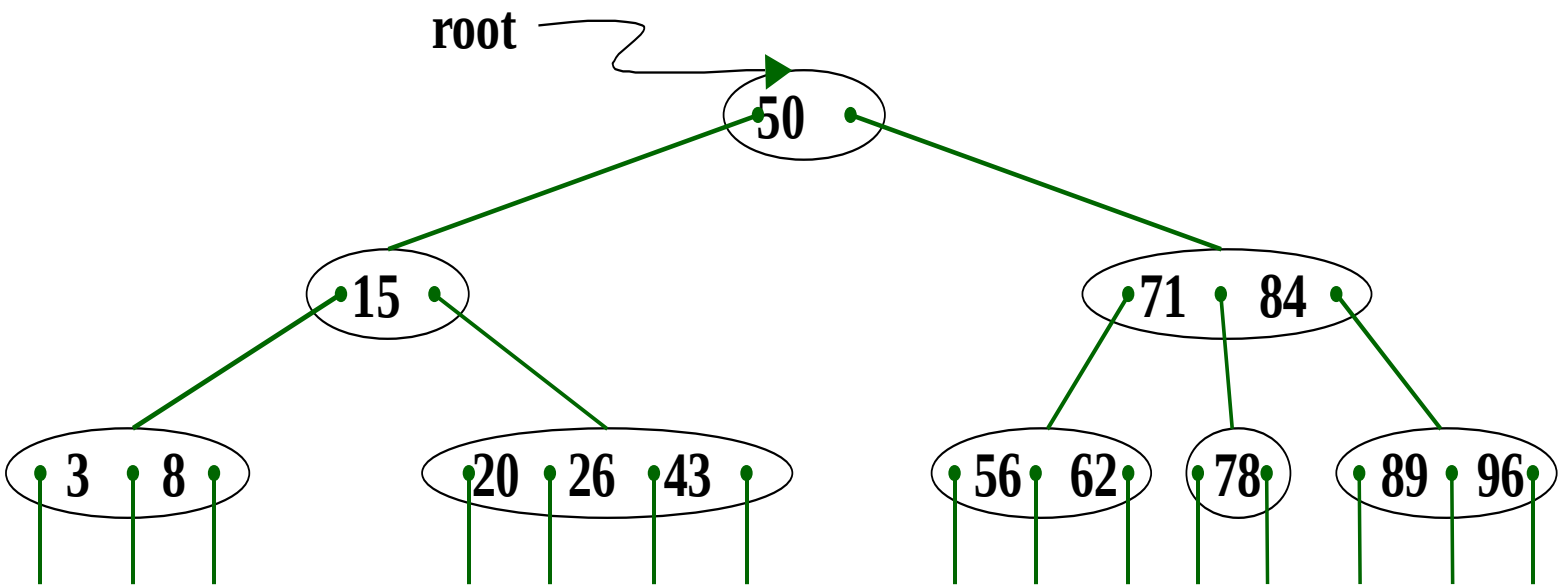
- B-树的查找操作
  - B-树的查找过程是一个顺指针（纵向）查找结点和在结点中（横向）查找交替进行的过程。
  - 若查找成功，则返回指向被查关键字所在结点的指针和关键字在结点中的位置；
  - 若查找不成功，则返回查找失败或插入位置。



8.7 B-树和B+树

B-树的查找操作

例如: 4 阶B-树，查找43，查找79



8.7 B-树和B+树

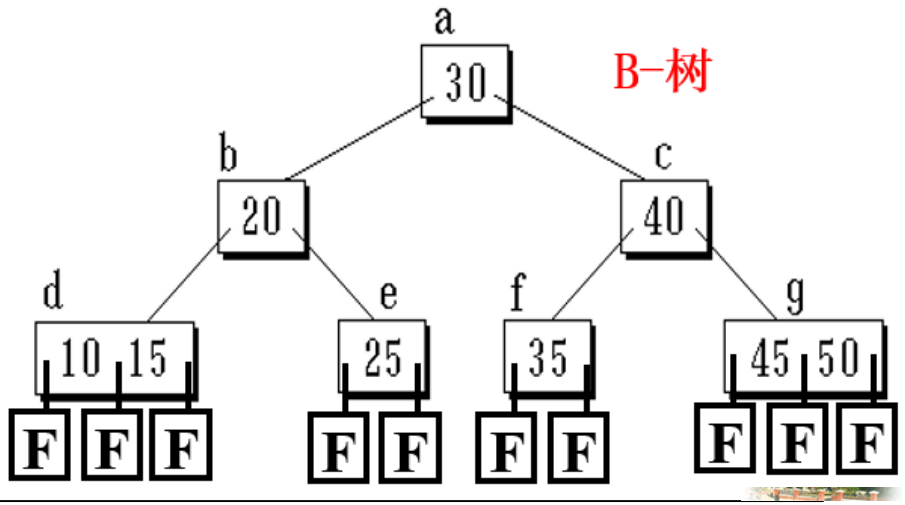
- B-树的查找性能
- B-树上的查找有两个基本步骤：
  - ①在B-树中查找结点，该查找涉及读盘操作，属外部查找；
    - 外查找的读盘次数不超过树高h，故其时间是O(h)；
  - ②在结点内查找，该查找属内查找。
    - 内查找中，每个结点内的关键字数目keynum<m (m是B-树的阶数)，故其时间为O(m)。
  - 查找关键字的时间为O(m*h)

- 注意：
  - 实际上外部查找时间可能远远大于内部查找时间
  - 在B-树中进行查找时，其查找时间主要花费在搜索结点（访问外存）上，即主要取决于B-树的深度。

8.7 B-树和B+树

B-树高度与关键字个数 *N* 之间的关系

- 设 *m* 阶B-树所有的失败结点都位于第 *h+1*层，在这棵B-树中关键字个数 *N* 最小能达到多少？
- 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。
  - h+1层至少有 $2\lceil m/2 \rceil^{h-1}$ 的结点



8.7 B-树和B+树

- h+1层至少有$2\lceil m/2\rceil^{h-1}$的结点
- 设 *m*若树中关键字有 *N* 个, 则失败结点数为 *N* +1
- *N* +1 = 位于第 *h+1* 层的结点数 ≥ $2\lceil m/2\rceil^{h-1}$

  $N \geq 2\lceil m/2\rceil^{h-1}-1$ (最少关键字的个数)
- 反之，如果在一棵 *m* 阶B-树中有 *N* 个关键字，则

  $h \leq \log_{\lceil m/2\rceil}((N+1)/2)+1$ (不含失败节点的最大高度)

- 例，若B-树的阶数 *m* = 199，关键字总数 *N* = 1999999，则在这棵B-树上进行一次查找，访问的结点个数不超过

  $\log_{\lceil 199/2\rceil}((1999999+1)/2)+1=\log_{100}1000000+1=4$
- 例，若B-树的阶数 *m* = 3，高度不含失败节点的高度为 *h* = 4，则关键字总数至少为

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

8.7 B-树和B+树

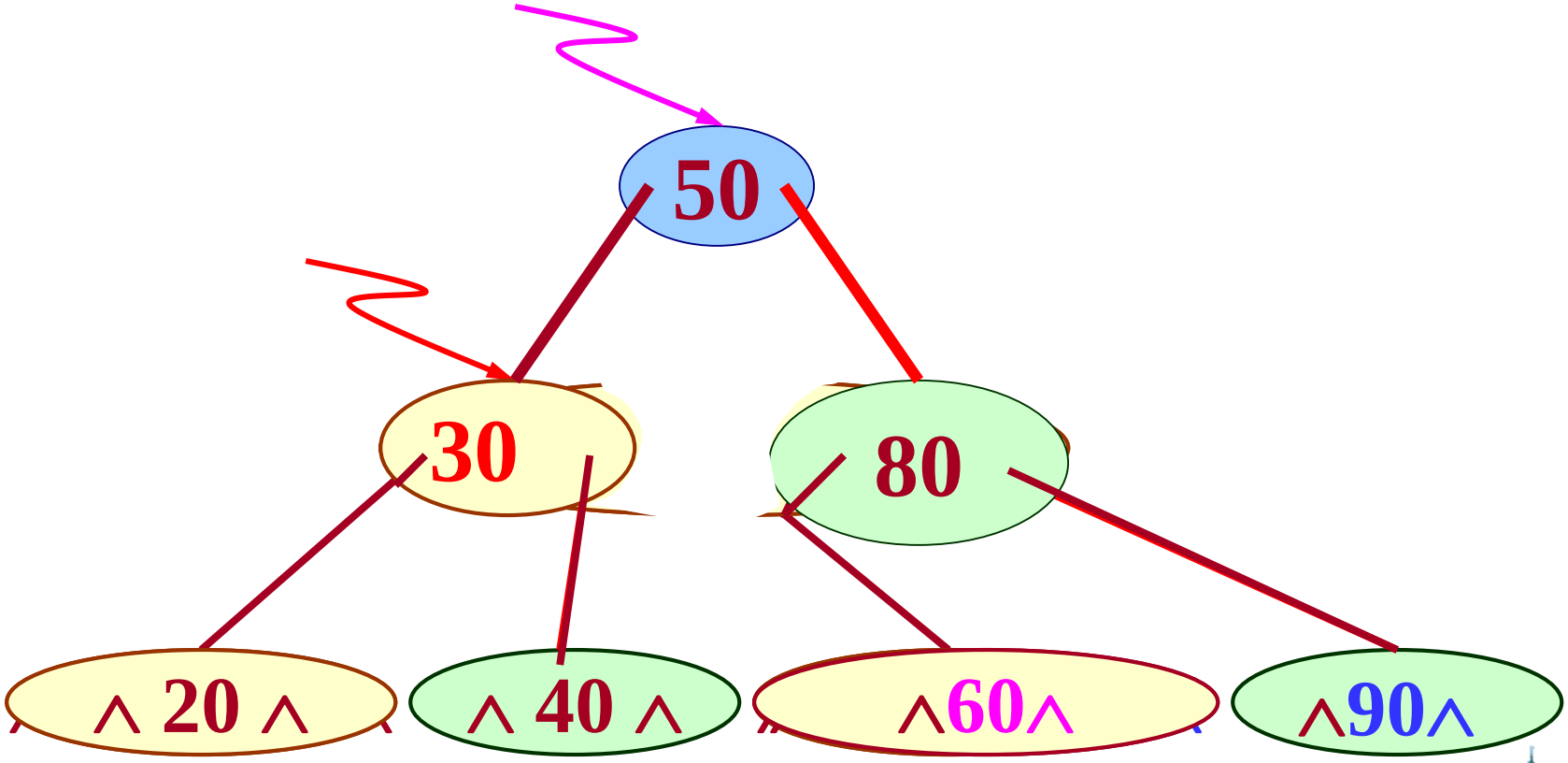
- B-树的插入操作与建立

B-树的是从空树起，逐个插入关键字而生成的。

- 在*m*阶B-树中，每个非失败结点的关键字个数都在[⌈*m*/2⌉-1, *m*-1]之间。
- 插入操作，首先执行查找操作以确定可以插入新关键字的终端结点 *p*；
- 如果在关键字插入后，结点中的关键字个数超出了上界 *m*-1，则结点需要“分裂”，否则可以直接插入。
- 如何分裂：令 s =⌈*m*/2⌉，
- 在原结点中保留
- （A0，K1，……，Ks-1，As-1）；
- 建新结点
- （As，Ks+1，……，Kn，An）；
- 将（Ks，p）插入双亲结点；

8.7 B-树和B+树

B-树的插入操作 例如:下列为 3 阶B-树，$s=\lceil 3/2 \rceil=2$



插入关键字 = 60, 90, 30,

【例5-15】以关键字序列：

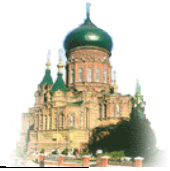
(a，g，f，b，k，d，h，m，i，
e，s，i，r，x，c，l，n，t，u，p)

建立一棵5阶B-树的生长过程。

注意：

①当一结点分裂时所产生的两个结点大约是半满的，这就为后续的插入腾出了较多的空间,尤其是当 m较大时，往这些半满的空间中插入新的关键字不会很快引起新的分裂；

②向上插人的关键字总是分裂结点的中间位置上的关键字，它未必是正待插入该分裂结点的关键字。因此，无论按何次序插入关键字序列，树都是平衡的。



关键字序列 :(a, g, f, b, k, d, h, m, j, e, s, i, r, x, c, l, n, t, u, p)

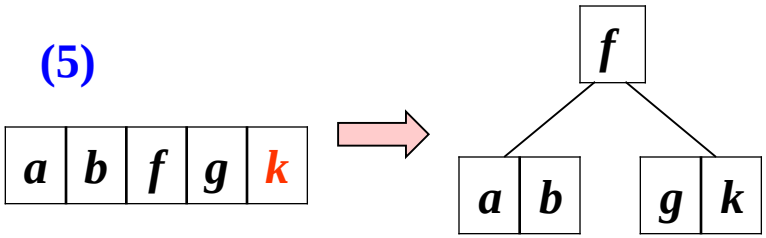
(1) a

(2) a g

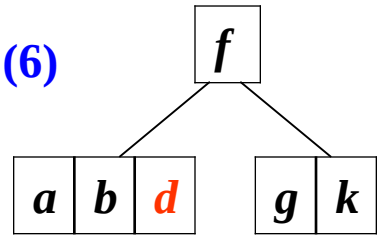
(3) a f g

(4) a b f g

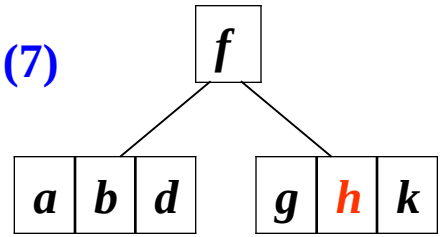
(5)



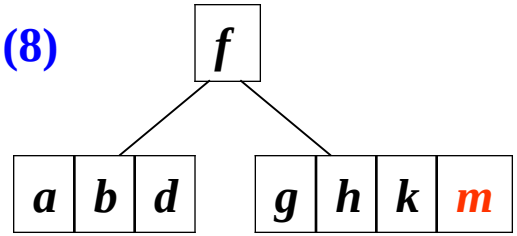
(6)

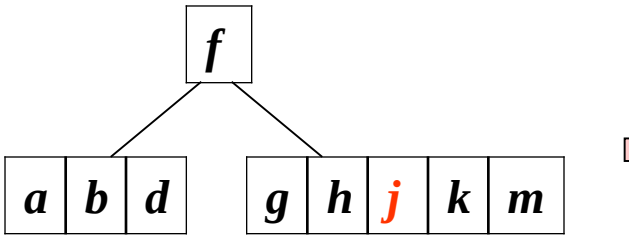


(7)

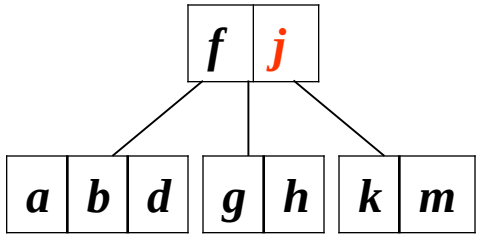


(8)

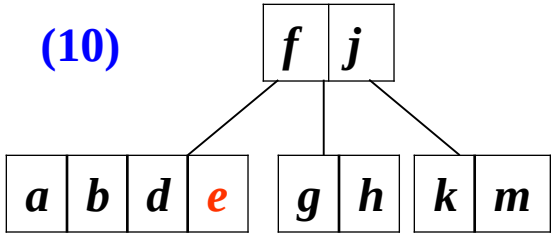




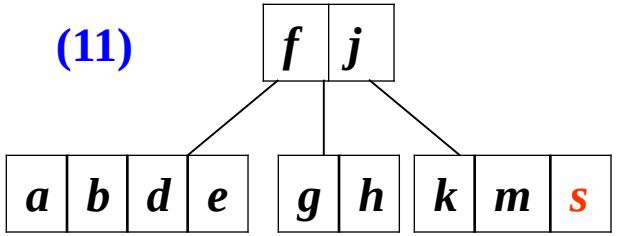
(9)



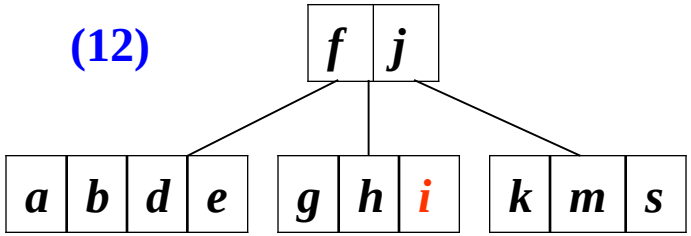
(10)


f j
a b d e
g h
k m

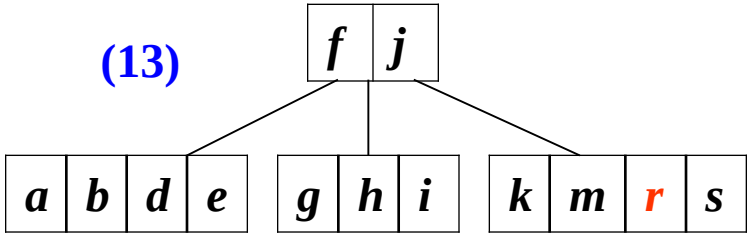
(11)

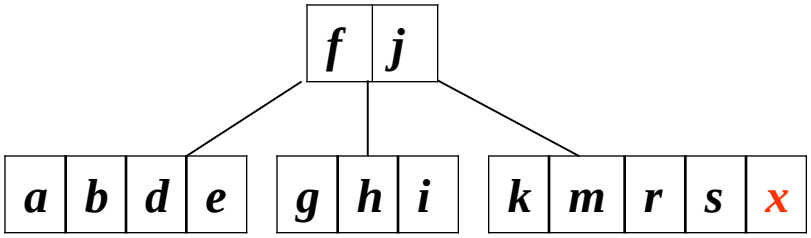

f j
a b d e
g h
k m s

(12)

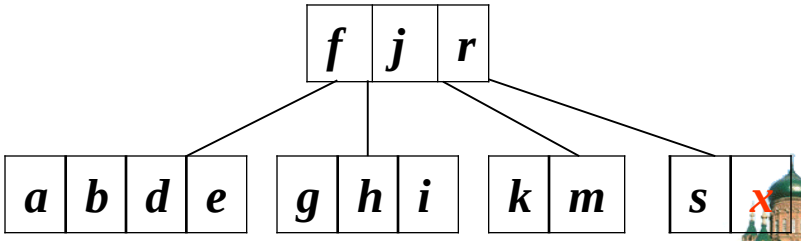

f j
a b d e
g h i
k m s

(13)

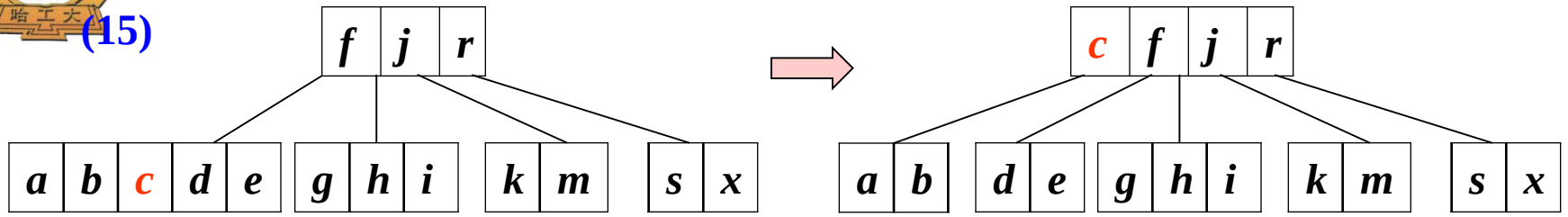

f j
a b d e
g h i
k m r s


f j
a b d e
g h i
k m r s x

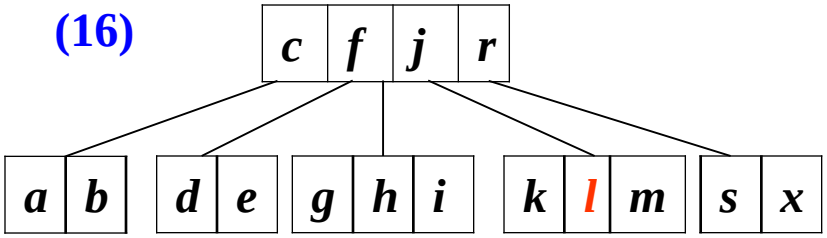
(14)


f j r
a b d e
g h i
k m
s x

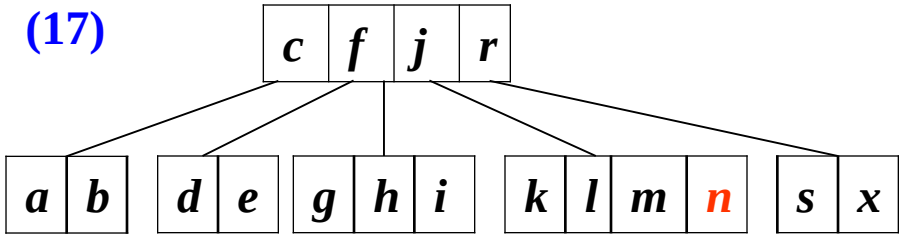
(15)


f j r
a b c d e g h i k m s x
c f j r
a b d e g h i k m s x

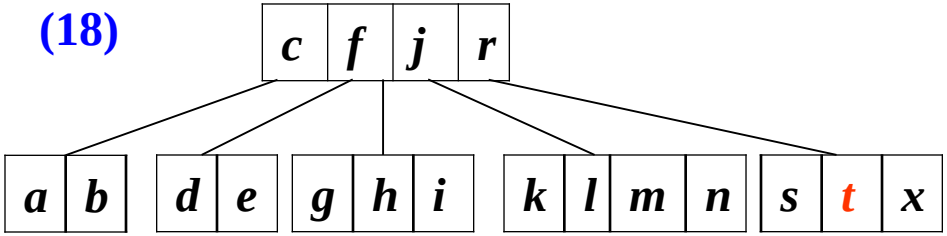
(16)


c f j r
a b d e g h i k l m s x

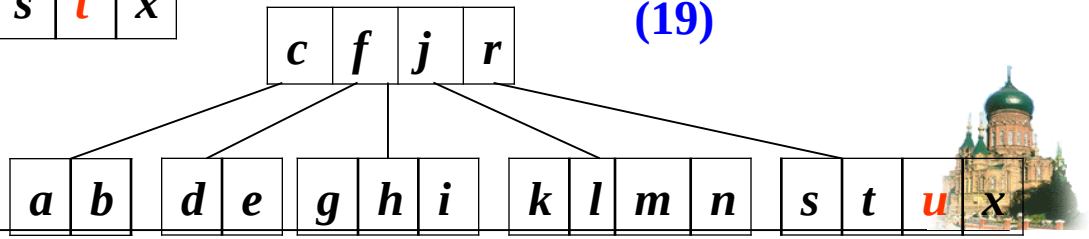
(17)


c f j r
a b d e g h i k l m n s x

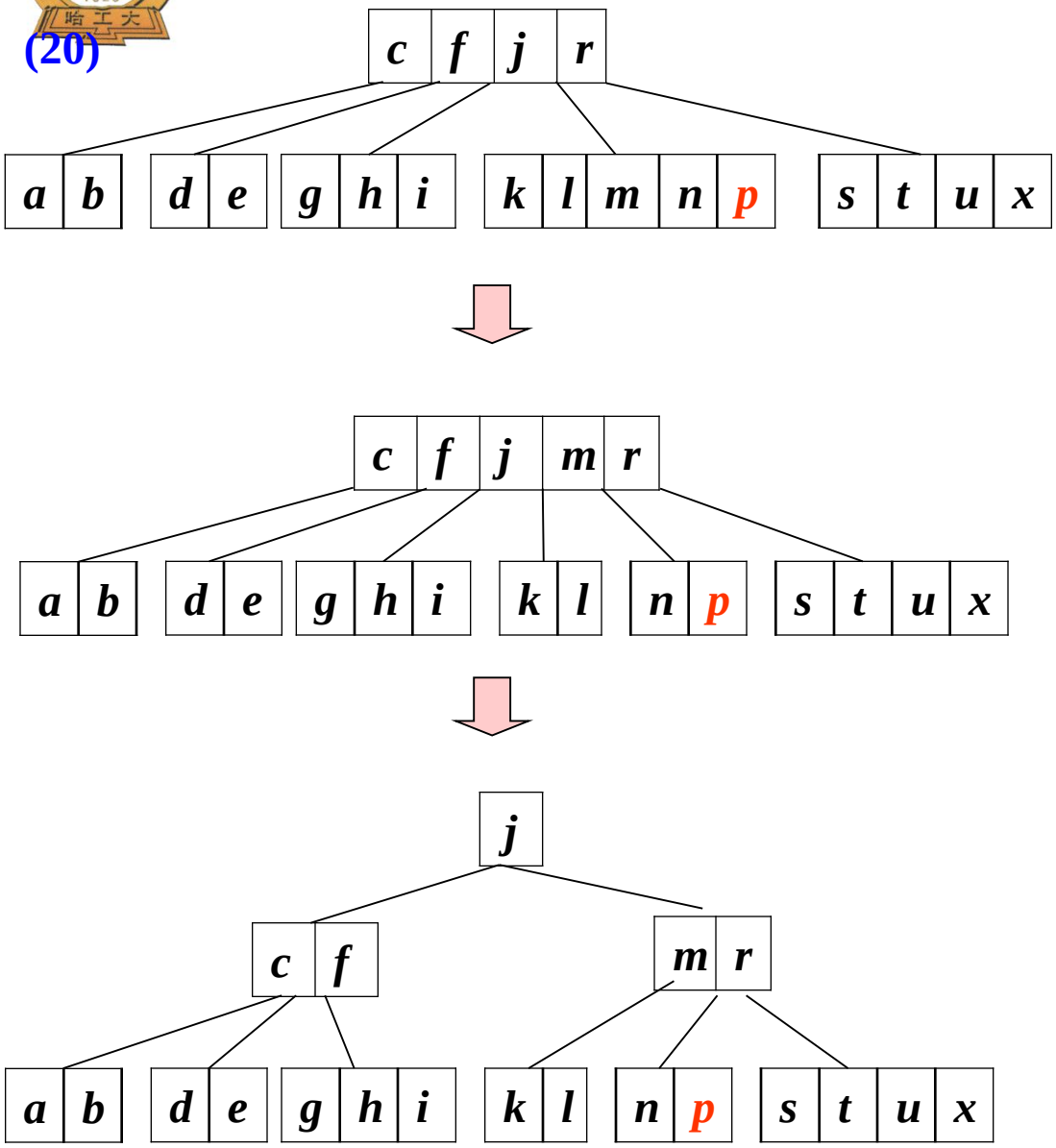
(18)


c f j r
a b d e g h i k l m n s t x

(19)


c f j r
a b d e g h i k l m n s t u x

(20)



重申：

①当一结点分裂时所产生的两个　结点大约是半满的，这就为后　续的插入腾出了较多的空间，尤其是当 m 较大时，往这些半满的空间中插入新的关键字不会很快引起新的分裂；

②向上插人的关键字总是分裂结点的中间位置上的关键字，它未必是正待插入该分裂结点的关键字。因此，无论按何次序插入关键字序列，树都是平衡的。

插入结点小结

首先执行插入操作以确定可以插入新关键字的终端结点*p*。

如果在结点*p*上插入新关键字是结点*p*的关键字达到*m*，则需要分裂结点*p*。

否则，只需将新结点*p*写入磁盘上，完成插入操作。

分裂节点：假定插入新关键字后，结点*p*具有如下结构：

$m, A_0, (K_1,A_1), (K_2, A_2), \ldots\ldots, (K_m, A_m)$, 且$K_i<K_{i+1}, 0 \le i < m$

分裂后，形成具有如下格式的两个结点*p*和*q*:

结点*p*： $m/2, A_0, (K_1,A_1), \ldots\ldots, (K_{m/2-1}, A_{m/2-1})$

结点*q*： $m-m/2, A_{m/2}, (K_{m/2},A_{m/2+1}), \ldots\ldots, (K_m, A_m)$

剩下的关键字$K_{m/2}$和指向新结点*q*的指针形成一个二元组$(K_{m/2},q)$。

该二元组将被插入到*p*的父结点中。插入前，将结点*p*和*q*写盘。

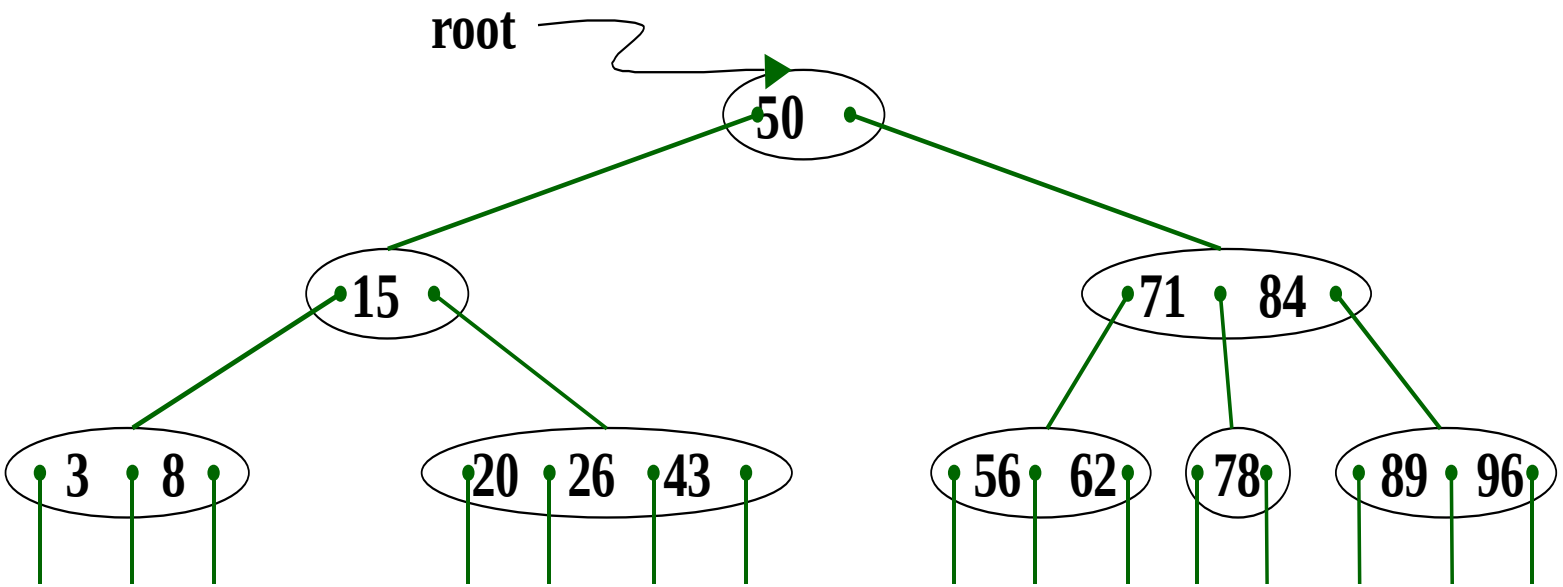
插入父结点有可能会导致这个父结点的分裂，而且此分裂过程可能会一直向上传播，直到根结点为止。

根结点分裂时，创建一个只包含一个关键字的新根结点，B-树的高度增1。

8.7 B-树和B+树

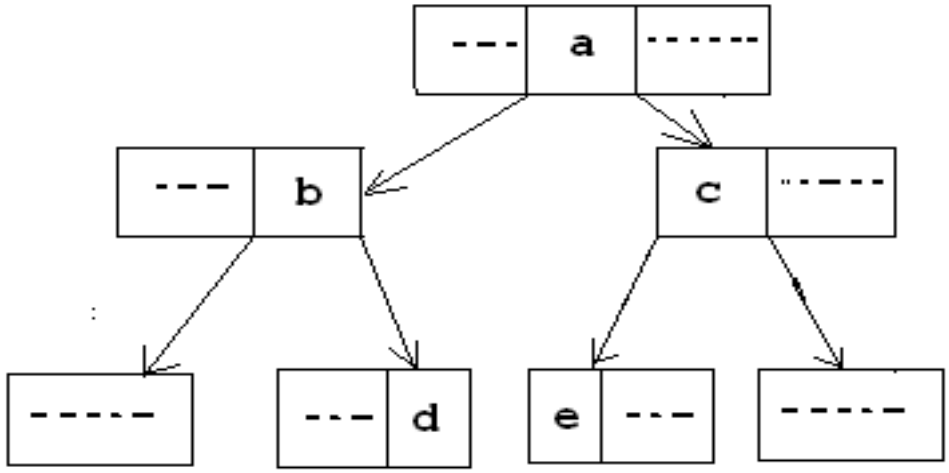
B-树的删除操作

首先，要找到该关键字 *x* 所在的结点*p*，从*p*中删去这个关键字。终端节点，非终端节点



8.7 B-树和B+树

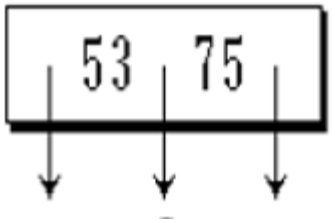
- **B-树的删除操作：** 在B-树上删除一个关键字*x*时，
  - 若该结点不是终端结点，比如a，找到其左子树中的最大关键字（关键字d）或者其右子树中的最小关键字(关键字e)来代替被删关键字a；
  - 然后在终端结点中删除d或e。
  - ①把所有删除操作转换为终端结点上的删除操作。



8.7 B-树和B+树

- **B-树的删除操作：**在终端结点上的删除分 4 种情况：
  - ①被删关键字所在终端结点 *p* 同时又是根结点，若删除后该结点中至少有一个关键字，则直接删去该关键字并将修改后的结点写回磁盘。否则，删除以后，B-树为空。

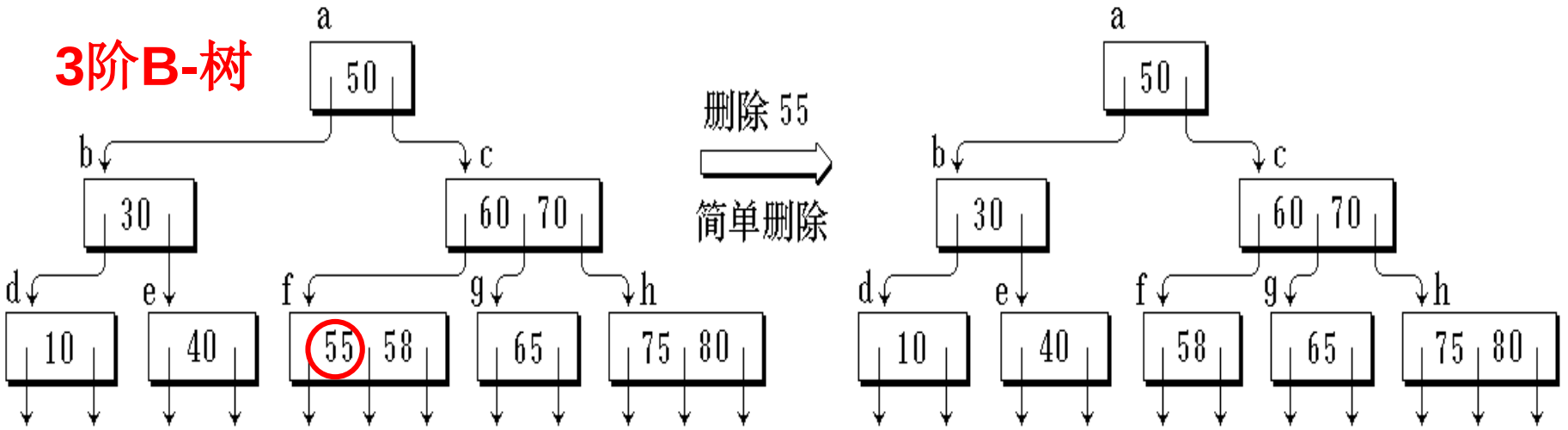
3阶B树



对于以下3种情况，*p* 都不是根结点：

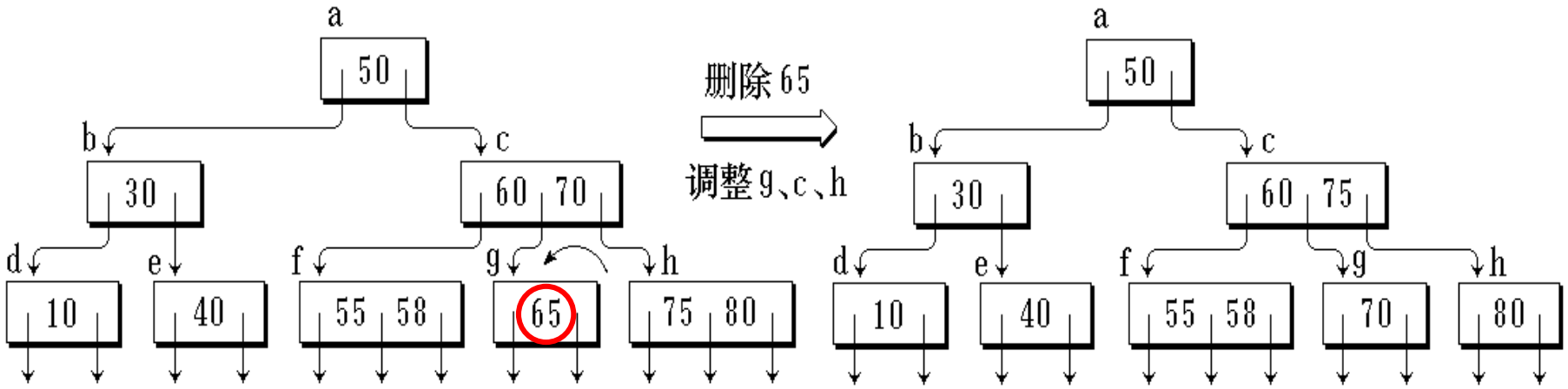
8.7 B-树和B+树

- **B-树的删除操作：**
- ②被删关键字所在终端结点 p 不是根结点，且删除前该结点中关键字个数 $n \geq \lceil m/2 \rceil$，则直接删去该关键字并将修改后的结点写回磁盘，删除结束。



8.7 B-树和B+树

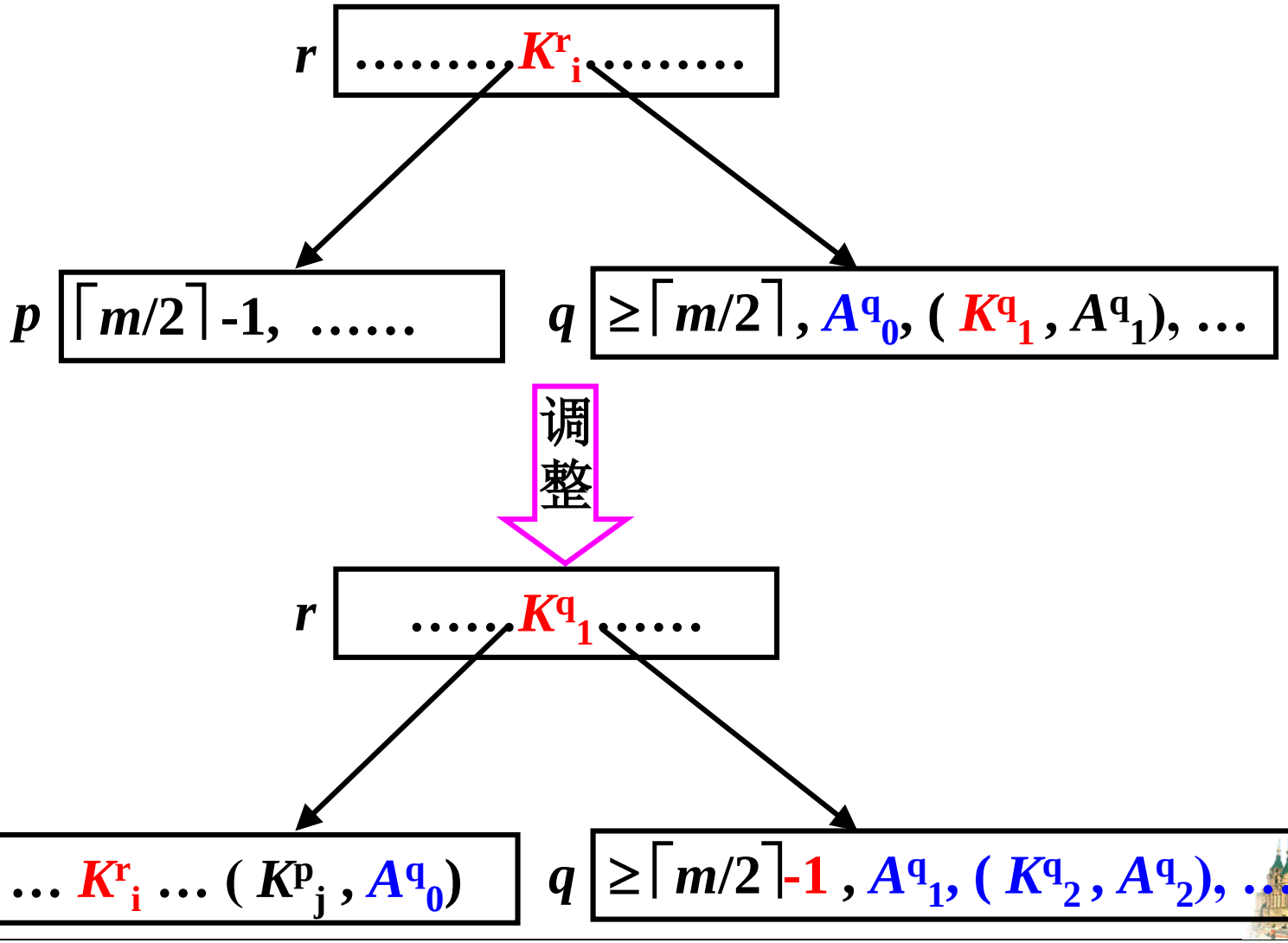
- B-树的删除操作：
  - ③被删关键字x所在的终端结点p删除前关键字个数 $n = \lceil m/2 \rceil - 1$，若此时其右兄弟（或左兄弟）$q$的关键字个数 $n \geq \lceil m/2 \rceil$，则可按以下步骤调整结点p、右兄弟（或左兄弟）q和其双亲结点r，以达到新的平衡。
    - 将双亲结点r中大于(或小于)被删的关键字的最小(最大)关键字 K_i($1 \leq i \leq n$)下移至结点p；
    - 将右兄弟（或左兄弟）结点中的最小（或最大)关键字上移到双亲结点r的 K_i 位置；
    - 将右兄弟（或左兄弟）结点中的最左（或最右）子树指针平移到被删关键字所在结点 p 中最后（或最前）子树指针位置；
    - 在右兄弟（或左兄弟）结点q中，将被移走的关键字和指针位置用剩余的关键字和指针填补、调整。再将结点q中的关键字个数减1。



8.7 B-树和B+树

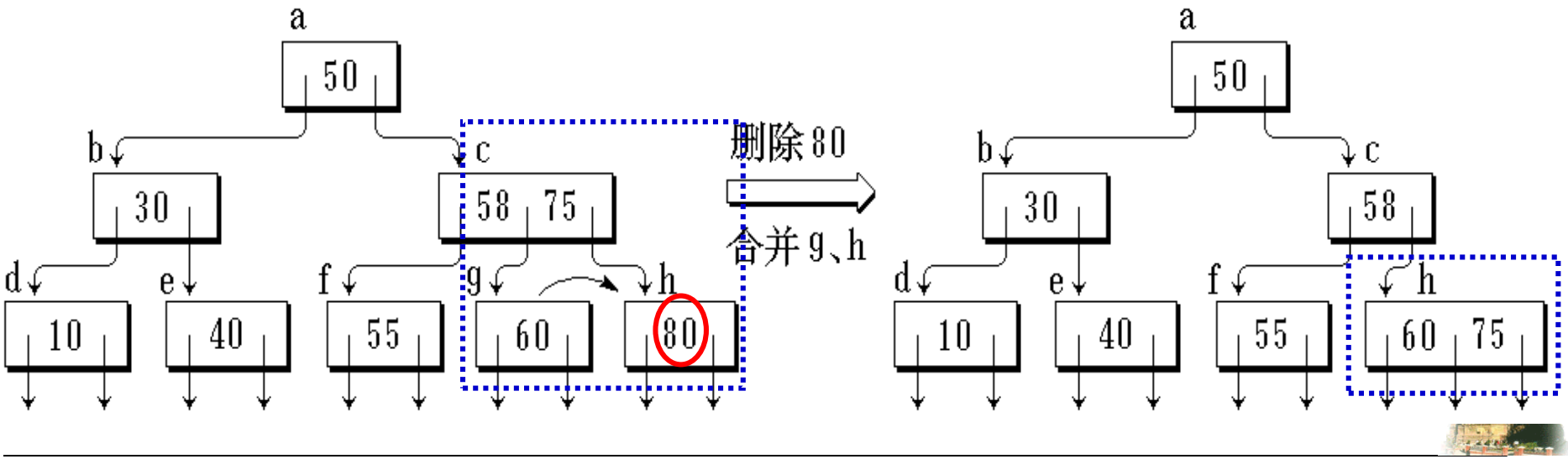
- **B-树的删除操作：** 在终端结点上的删除分 4 种情况：

兄弟够借，向兄弟借；调整



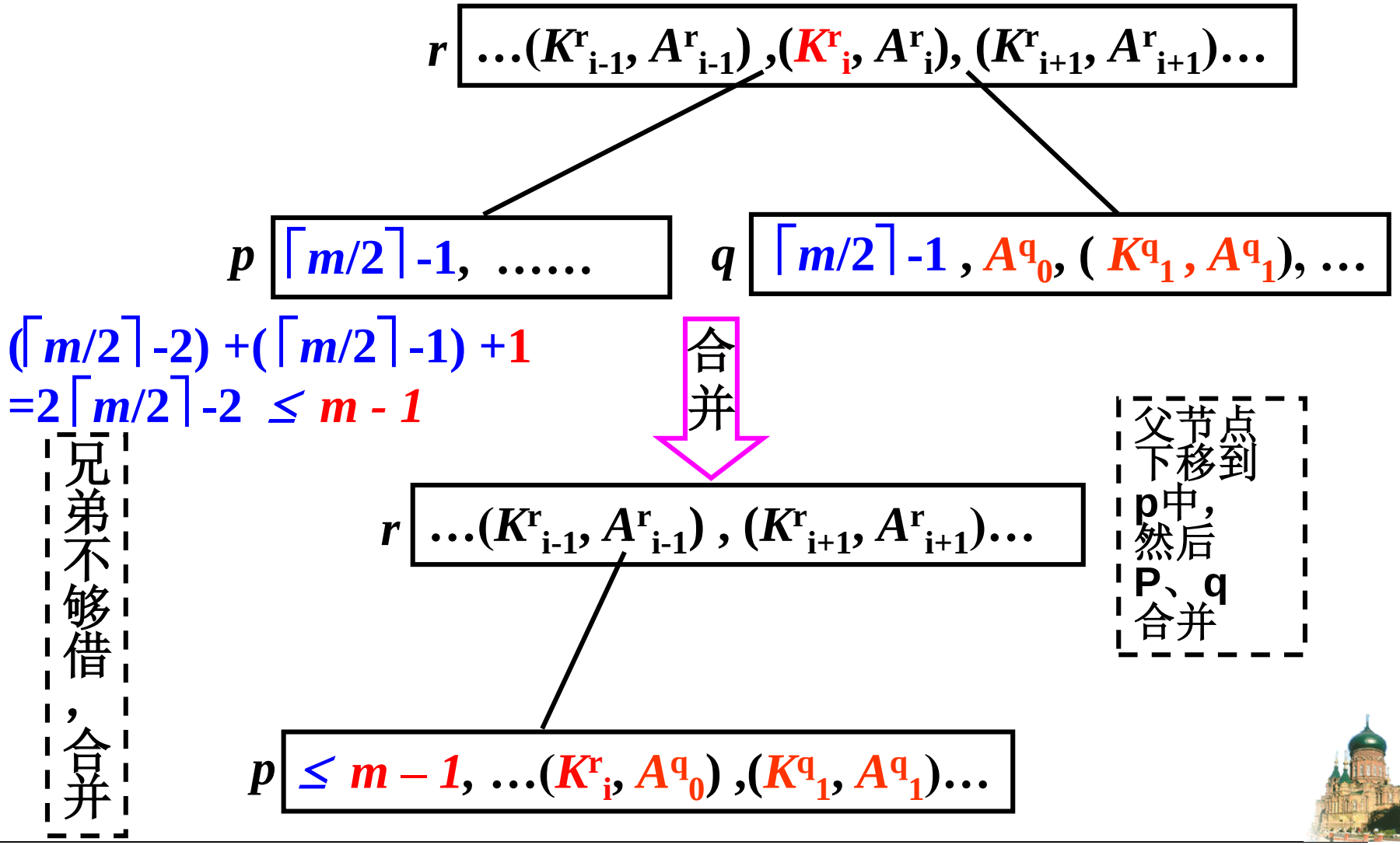
8.7 B-树和B+树

- **B-树的删除操作：**
  - ④被删关键字被删关键字*x*所在的终端结点*p*删除前关键字个数 $n = \lceil m/2 \rceil - 1$，若此时其右兄弟 (或左兄弟) 结点q的关键字个数$n = \lceil m/2 \rceil - 1$，则按以下步骤合并这两个结点。
    - 将双亲结点 *r* 中相应关键字K_i下移到选定保留的结点中。若要合并 *r* 中的子树指针 A_{i-1} 与 A_i 所指的结点，且保留 A_{i-1} 所指结点，则把 *r* 中的关键字 K_i 下移到 A_{i-1} 所指的结点中。
    - 把 *r* 中子树指针 A_i 所指结点中的全部指针和关键字都拷贝到 A_{i-1} 所指结点的后面。删去 A_i 所指的结点。
    - 在结点 *r* 中用后面剩余的关键字和指针填补关键字 K_i 和指针 A_i。
    - 修改结点 *r* 和选定保留结点的关键字个数。



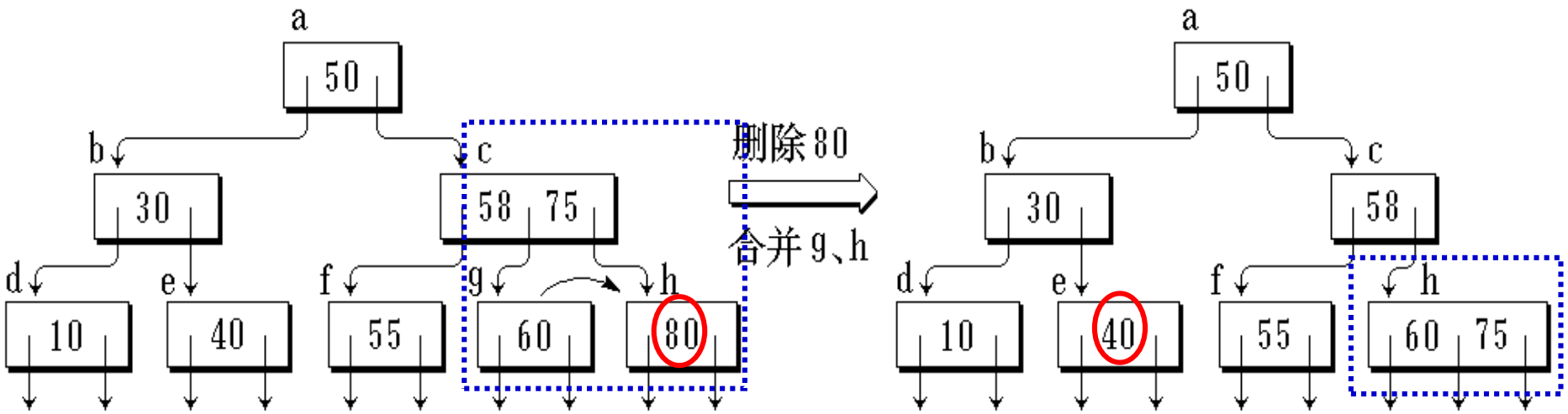
8.7 B-树和B+树

- B-树的删除操作：在终端结点上的删除分 4 种情况：



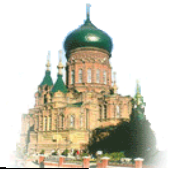
8.7 B-树和B+树

- **B-树的删除操作：**



8.7 B-树和B+树

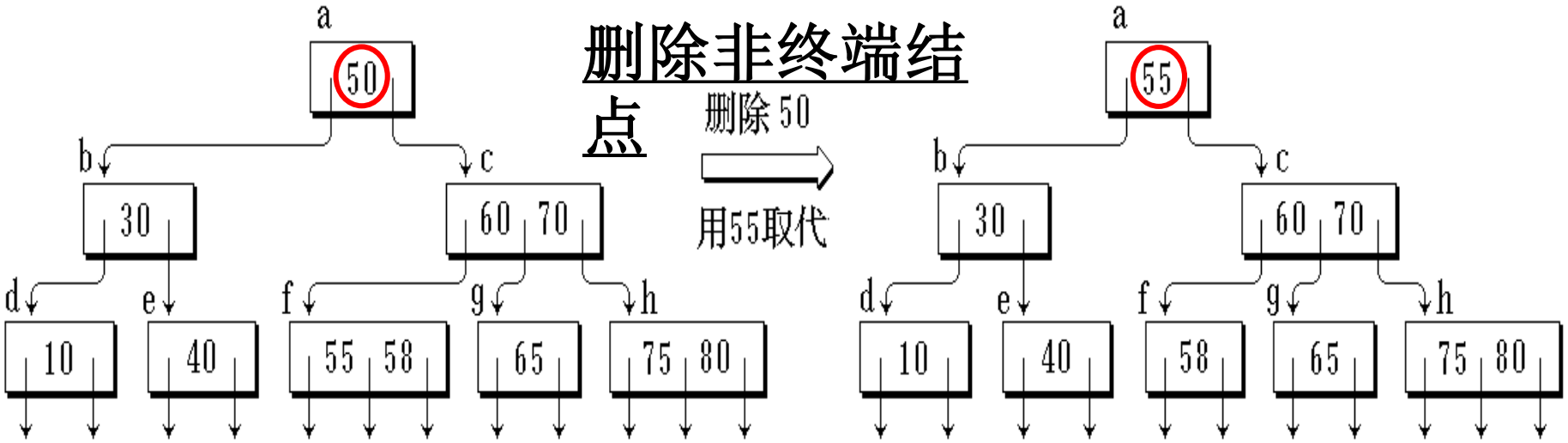
- **B-树的删除操作：**
  - 在在第四种情况④合并结点的过程中，双亲结点中的关键字个数减少了。
  - 若双亲结点不是根结点，且关键字个数减到$\lceil m/2 \rceil - 2$，则又要与它自己的兄弟结点合并，重复上面的合并步骤。最坏情况下这种结点合并处理要自下向上直到根结点。
  - 若双亲结点是根结点且结点关键字个数减到 0，则该双亲结点应从树上删去，合并后保留的结点成为新的根结点；否则将双亲结点与合并后保留的结点都写回磁盘，删除处理结束。



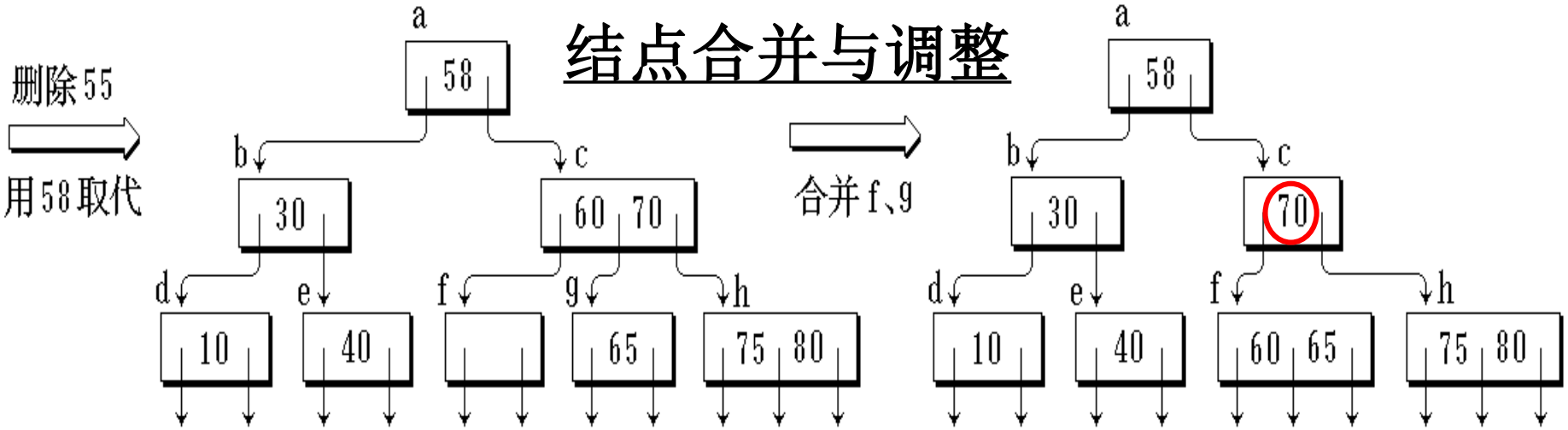
8.7 B-树和B+树

- **B-树的删除操作：** 在终端结点上的删除分 4 种情况：

删除非终端结点

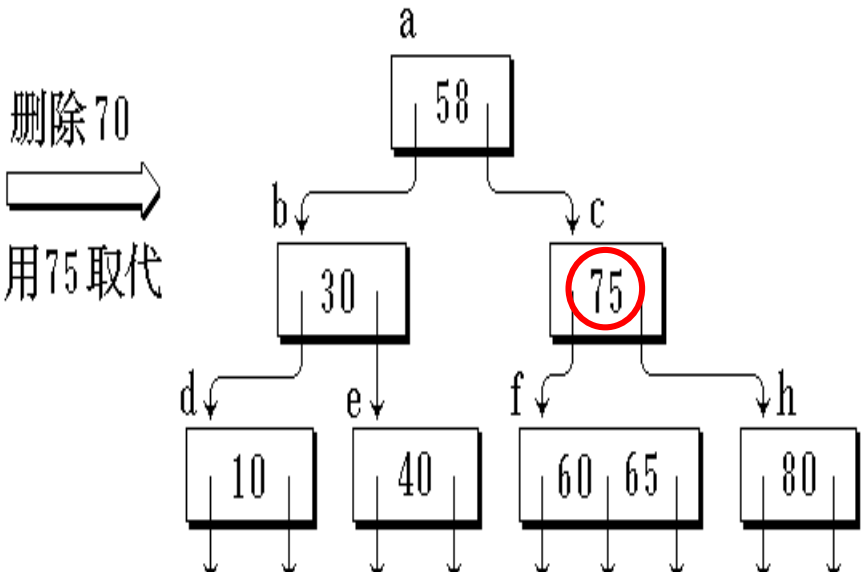


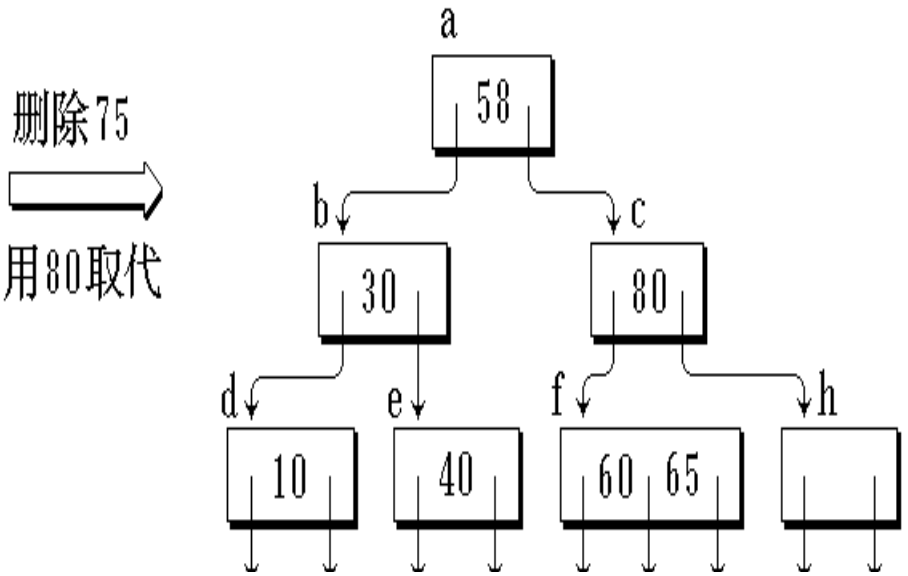
结点合并与调整

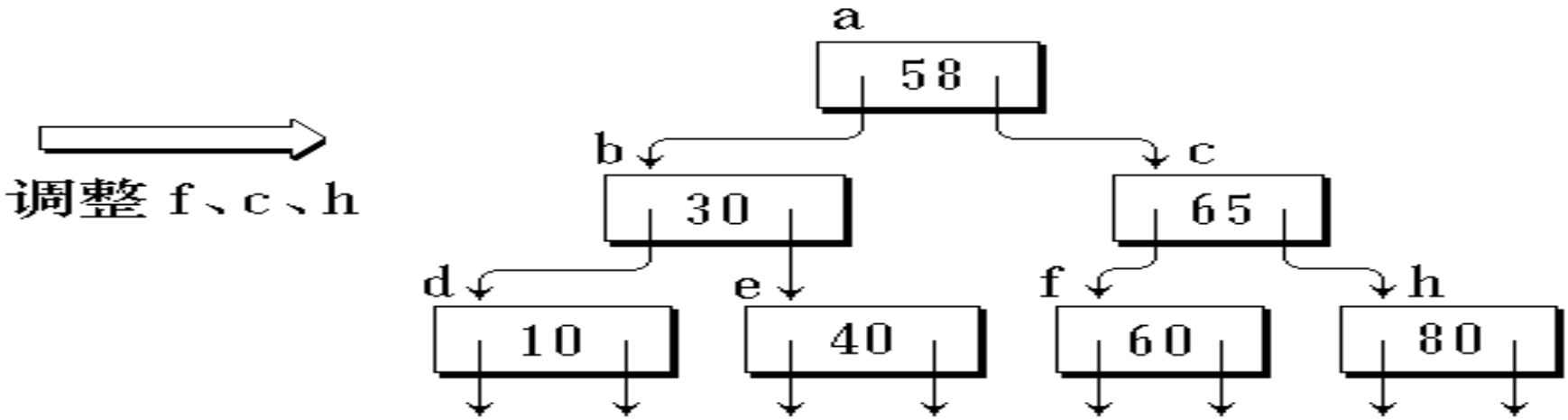


8.7 B-树和B+树

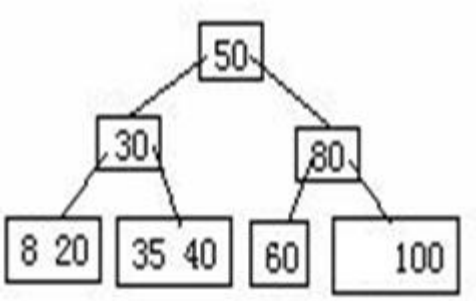
- **B-树的删除操作：在终端结点上的删除分 4 种情况：**

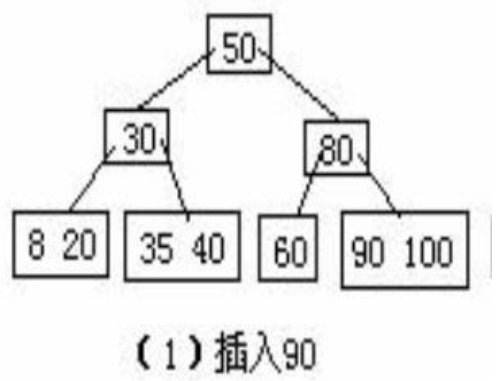




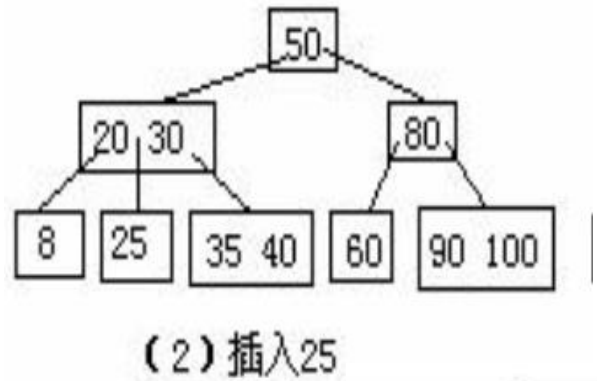


例.对下面的3阶B-树，依次执行下列操作，画出各步操作的结果。
1）插入90 （2） 插入25 （3） 插入45 （4） 删除60 （5）删除80

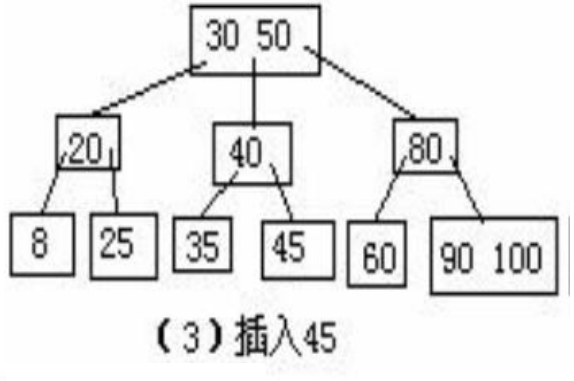




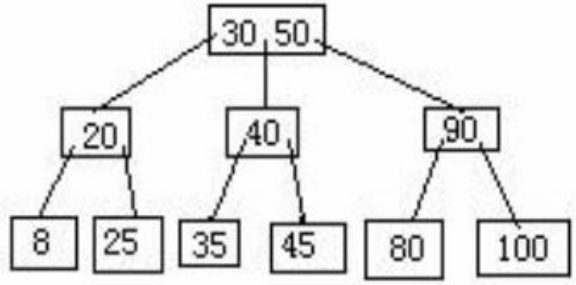
（1）插入90



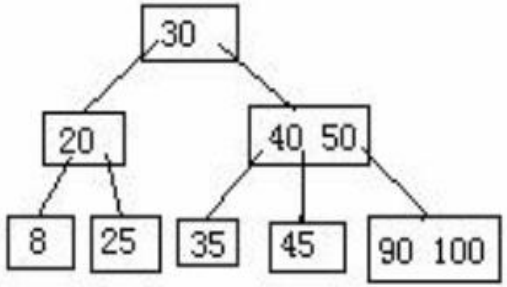
（2）插入25



（3）插入45



（4）删除60

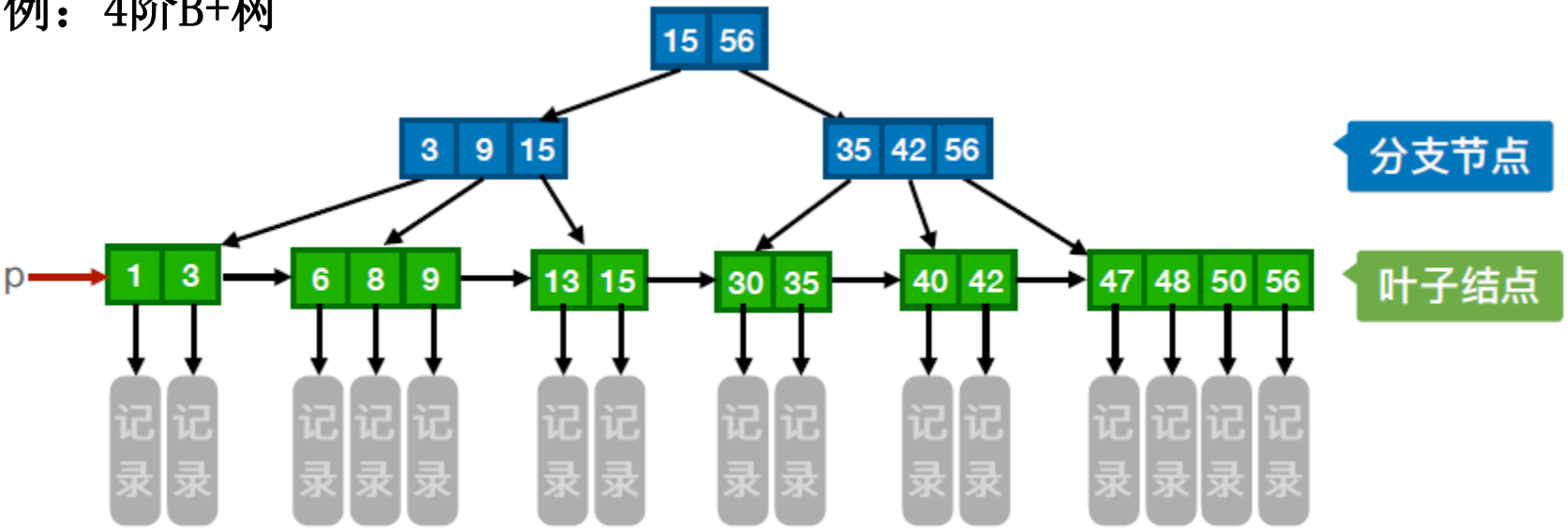


（5）删除80

8.7 B-树和B+树

B+树

例：4阶B+树



一棵m阶的B+树需满足下列条件：

1）每个分支结点最多有m棵子树（孩子结点）。

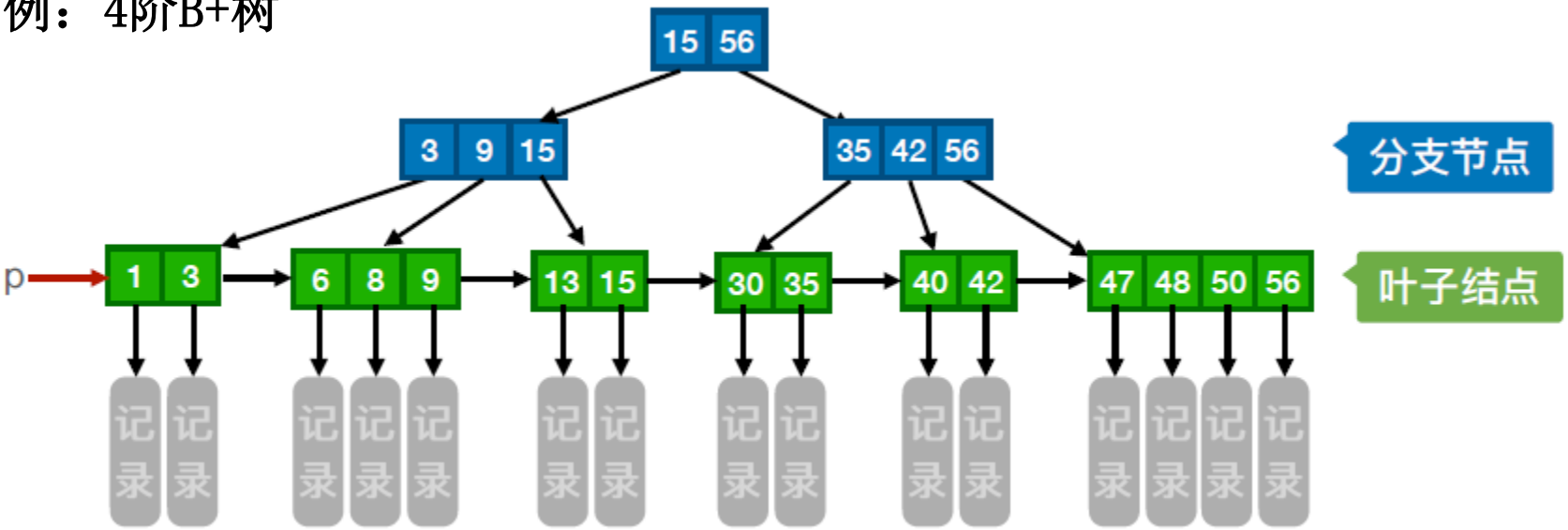
2）非叶根结点至少有两棵子树，其他每个分支结点至少有$\lceil m/2 \rceil$棵子树。

3）结点的子树个数与关键字个数相等。

8.7 B-树和B+树

B+树

例：4阶B+树



一棵m阶的B+树需满足下列条件：

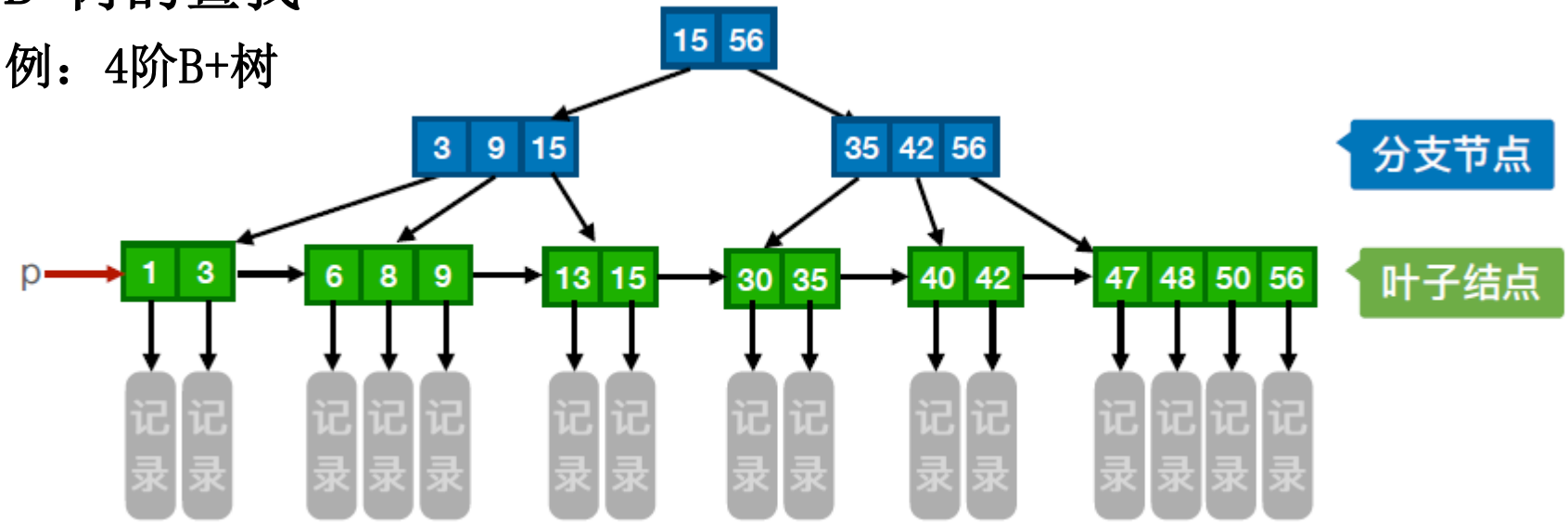
4）所有叶结点包含全部关键字及指向相应记录的指针，叶结点中将关键字按大小顺序排列，并且相邻叶结点按大小顺序相互链接起来。

5）所有分支结点中仅包含它的各个子结点中关键字的最大值及指向其子结点的指针。

8.7 B-树和B+树

B+树的查找

例：4阶B+树



◆所有的非终端结点可以看成是索引；

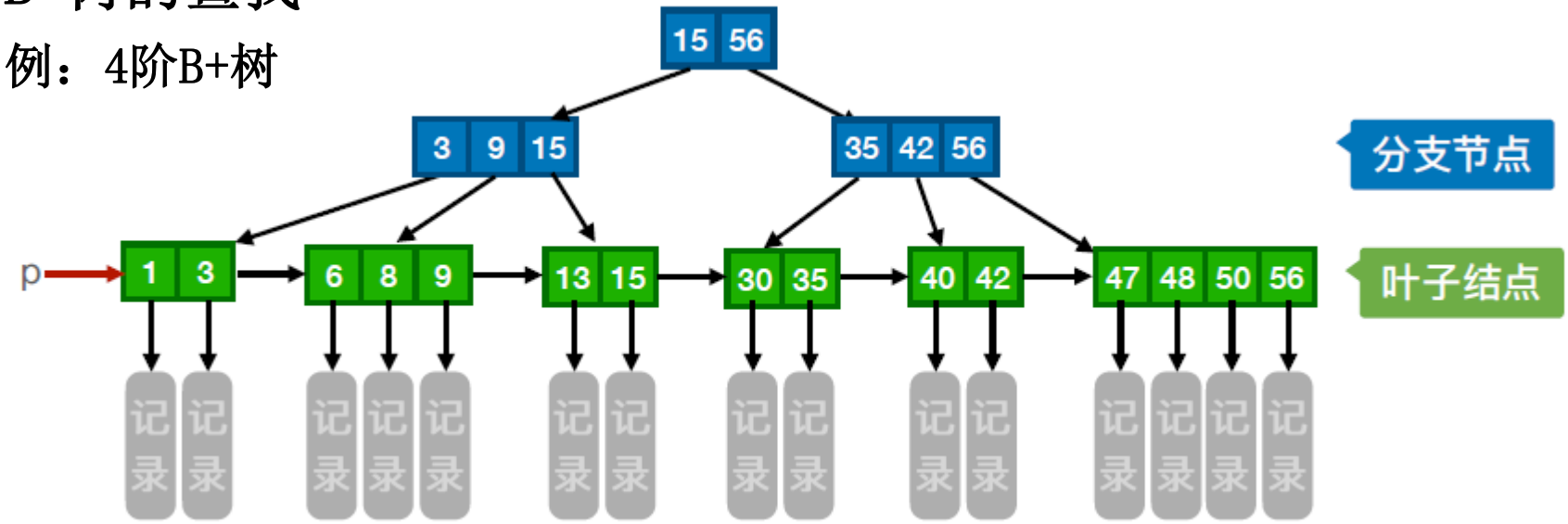
◆查找过程与B-树类似。只是在搜索过程中，若非终端结点上的关键码等于给定值，搜索并不停止，而是继续沿右指针向下，一直查到叶结点上的这个关键码。

◆在B+树，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。

8.7 B-树和B+树

B+树的查找

例：4阶B+树



- 在B^+树中有两个头指针：一个指向B^+树的根结点，一个指向关键字最小的终端结点。
- 可对B^+树进行两种查找操作：

  一种是沿终端结点链顺序查找，

  另一种是从根结点开始，进行自顶向下，直至终端结点的查找。

8.7 B-树和B+树

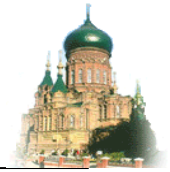
	m阶B树	m阶B+树
类比	二叉查找树的进化——>m叉查找树	分块查找的进化——>多级分块查找
关键字与分叉	n个关键字对应n+1个分叉（子树）	n个关键字对应n个分叉
结点包含的信息	所有结点中都包含记录的信息	只有最下层叶子结点才包含记录的信息
查找方式	不支持顺序查找。查找成功时，可能停在任何一层结点，查找速度“不稳定”	支持顺序查找。查找成功或失败都会到达最下一层结点，查找速度“稳定”
相同点	除根节点外，最少 $\lceil m/2 \rceil$ 个分叉（确保结点不要太“空”） 任何一个结点的子树都要一样高（确保“绝对平衡”）	

8.7 B-树和B+树

查找树

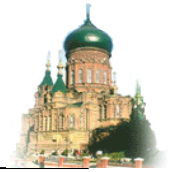
- 二叉排序树（二叉查找树）（BST）
- 平衡二叉排序树（AVL树） ---二叉平衡查找树
- B - 树
- B^+树

多路平衡查找树



8.8 散列技术

- 查找操作要完成什么任务？
  - 对于待查值*k*，通过比较，确定*k*在存储结构中的位置
- 基于关键字比较的查找的时间性能如何？
  - 其时间性能为O(log *n*)~O(*n*)。
  - 要向突破此下界，就不能仅依赖于基于比较来进行查找。
- 能否不用比较，通过关键字的取值直接确定存储位置？
  - 在关键字值和存储位置之间建立一个确定的对应关系
  - 理想情况下，随机存取，即时间复杂度为O(1)



8.8 散列技术

散列法（哈希查找法）——地址散列法

被查找元素的存储地址 = Hash (*Key*)

【例】全国30个地区的人口统计，每个地区为一个记录，内容如下：

地区名	总人口	汉族	回族	…

$f_1(key)$: 取关键字第一个字母在字母表中的序号；

$f_2(key)$: 求第一和最后字母在字母表中序号之和，然后取30的余数；

Key	BEIJING（北京)	TIANJIN (天津)	HEBEI (河北)	SHANXI (山西)	SHANGHAI (上海)	SHANGDONG (山东)	HENAN (河南)	SICHUAN (四川)
$f_1(key)$	02	20	08	19	19(?)	19(?)	08(?)	19(?)
$f_2(key)$	09	04	17	28	28(?)	26	22	03

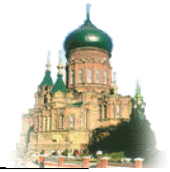
8.8 散列技术

- 散列技术仅仅是一种查找技术吗？
  - 散列既是一种查找技术，也是一种存储技术。
- 散列表是一种完整的存储结构吗？
  - 散列表只是通过记录的关键字的值定位该记录，没有表达记录之间的逻辑关系，所以散列主要是面向查找的存储结构
- 散列技术适合于哪种类型的查找？
  - 不适用于范围查找，如在散列表中，找最大或最小关键值的记录，也不可能找到在某一范围内的记录。
  - 散列技术最适合回答的问题是：如果有的话，哪个记录的关键字的值等于待查值。
- 散列技术有两种：一种是内散列法或称闭散列表，基本的数据结构是数组；另一种称为外散列表或称开散列表，基本数据结构是邻接表。

8.8 散列技术

相关概念

- 哈希函数：又叫散列函数，在关键字与记录在表中的存储位置之间建立一个函数关系，以 Hash(key) 作为关键字为 key 的记录在表中的位置，通常称这个函数 Hash (key) 为哈希函数。
- 哈希地址：由哈希函数得到的存储位置称为哈希地址。
- 装填因子：设散列表空间大小为n，填入表中的结点数为m，则称α=m/n为散列表的装填因子。
- 冲突（Collision）与同义词： 若Hash (k1)=Hash (k2),则称为冲突，发生冲突的两个关键字k1和k2称为同义词。



8.8 散列技术

相关概念

- 哈希表（散列表）
- 根据设定的哈希函数 Hash(key) 和所选中的处理冲突的方法，将一组关键字映象到一个有限的、地址连续的地址集 (区间) 上，并以关键字在地址集中的“象”作为相应记录在表中的存储位置，如此构造所得的查找表称之为“哈希表”。

在构造这种特殊的“查找表” 时：

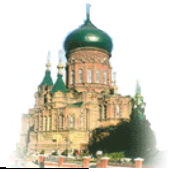
- 选择一个尽可能少产生冲突的哈希函数；
- 需要找到一种“处理冲突” 的方法。

Hash查找的关键问题
- ①构造**Hash函数(表长的确定）**
- ②制订解决**冲突**的方法

对数字关键字构造**散列函数（Hash）**的几种方法：

- 直接定址法：
- 质数除余法；
- 平方取中法；
- 折叠法；
- 数字分析法；
- 随机数法；

若是非数字关键字，则需先对其进行数字化处理。



8.8 散列技术

散列函数的构造

- 散列函数的构造方法----直接定址法
  - 散列函数是关键字值的线性函数，即：$h(key) = a \times key + b$ （a，b为常数）
  - 示例：关键字的取值集合为{10, 30, 50, 70, 80, 90}，选取的散列函数为$h(key)=key/10$，则散列表为：

0	1	2	3	4	5	6	7	8	9
	10		30		50		70	80	90

  - 优点：没有冲突
  - 缺点：若关键字取值集合很大或者不连续，浪费存储空间。

8.8 散列技术

散列函数的构造

- 散列函数的构造方法----质数除余法
  - 散列函数为：$h(key)=key \ \% \ m$
  - 示例：关键字的取值集合为{14, 21, 28, 35, 42, 49, 56, 63}，表长B=12。则选取m=11，散列函数为$h(key)=key$ % 11，则散列表为：

0	1	2	3	4	5	6	7	8	9	10	11
	56	35	14		49	28		63	42	21	

  - 一般情况下，选m为小于或等于表长B的最大质数。
    - Why？——用质数取模，分布更均匀，冲突更少。参见《数论》
  - 适用情况：质数除余法是一种最简单、也是最常用的构造散列函数的方法，适合对长度不等的关键字构造哈希函数，并且不要求事先知道关键码的分布。

8.8 散列技术

散列函数的构造

- 散列函数的构造方法----平方取中法
  - 取 key^2 的中间的几位数作为散列地址
  - 扩大相近数的差别，然后根据表长取中间几位作为散列值，使地址值与关键字的每一位都相关。
  - 散列地址的位数要根据B来确定，有时要设一个比例因子，限制地址越界。
  - 适用情况：事先不知道关键码的分布且关键码的位数不是很大

由于平方值中间的数据通常依赖于关键字的所有字符，因此不同的关键字映射为不同的桶地址的可能性很大。

记录	*key*	key^2	Hash
A	0100	0 010 000	010
I	1100	1 210 000	210
J	1200	1 440 000	440
I0	1160	1 370 400	370
P1	2061	4 310 541	310
P2	2062	4 314 704	314
Q1	2161	4 734 741	734
Q2	2162	4 741 304	741
Q3	2163	4 745 651	745

8.8 散列技术

散列函数的构造

- 散列函数的构造方法----折叠法
  - 若关键字位数较多，可根据B的位数将关键字分割成位数相同的若干段（最后一段位数可以不同），然后将各段叠加和(舍去进位)作为散列地址。
  - 示例：图书编号: 0-442-20586-4

```
   5864          5864
   4220          0224
+    04       +    04
-------       -------
  10088         6092
```

左：移位叠加，将分割后的每一部分的最低位对齐，然后相加

Hash(*key*) = 0088

右：间界叠加，从一端向另一端沿分割界来回折叠，然后对齐相加

Hash(*key*) = 6092

  - 适用情况：关键码位数很多，事先不知道关键码的分布。

8.8 散列技术

散列函数的构造

- 散列函数的构造方法----数字分析法
  - 根据关键字值在各个位上的分布情况，选取分布比较均匀的若干位组成散列地址。
  - 示例：

假设散列表长为10000，可取中间四位为散列地址。

8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7
8	1	3	5	4	1	5	7
8	1	3	6	8	5	3	7
8	1	4	1	9	3	5	5
①	②	③	④	⑤	⑥	⑦	⑧

  - 适用情况：若事先知道关键字集合，且关键字的位数比散列表的地址位数多

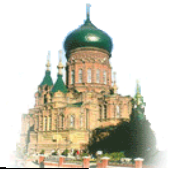
8.8 散列技术

散列函数的构造

- 散列函数的构造方法----随机数法
  - 选择一个随机函数，取关键字的随机函数值作为散列地址，即 **Hash (key) = random(key)**

其中**random**是某个伪随机函数，且函数值在**0, …, B-1**之间。

  - 适用情况：通常，当关键字长度不等时采用此法较恰当

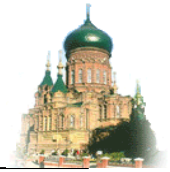


8.8 散列技术

- 构造Hash函数应注意以下几个问题：
  - 计算Hash函数所需时间　　计算简单
  - 关键字的长度
  - 散列表的大小
  - 关键字的分布情况
  - 记录的查找频率

  } 分布均匀

◆很难找到一个不产生冲突的哈希函数



8.8 散列技术

“处理冲突” 的实际含义是：

为产生冲突的地址寻找下一个哈希地址。

表序号	1	2	3	4	5	6	7	8	9	10	11	12
按i＝INT(k/3)＋1	01	05		09	13	16	19	21	26	27	31	
填入的关键字	02			11								

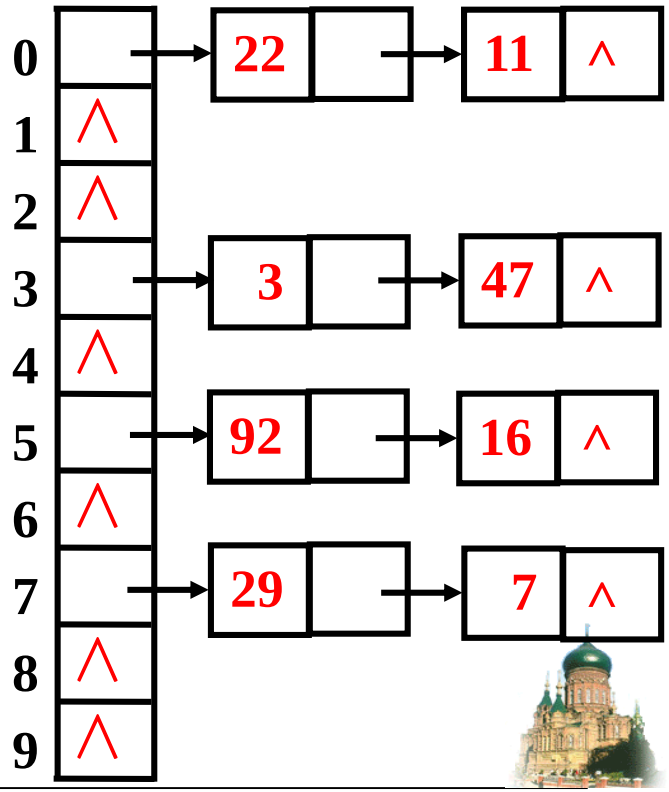
- **解决冲突的方法：**
  - 链地址法
  - 开放定址法
  - 再散列法

8.8 散列技术

冲突处理的方法----拉链法（链地址法）

- 基本思想：将所有散列地址相同的记录，即所有同义词的记录存储在一个单链表中（称为同义词子表），在散列表中存储的是所有同义词子表的头指针。

- 开散列表：用拉链法处理冲突构造的散列表也叫做开散列表。
- 示例：关键字取值集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为*h*(*key*)=*key* %11，用链地址法处理冲突，则散列表为：

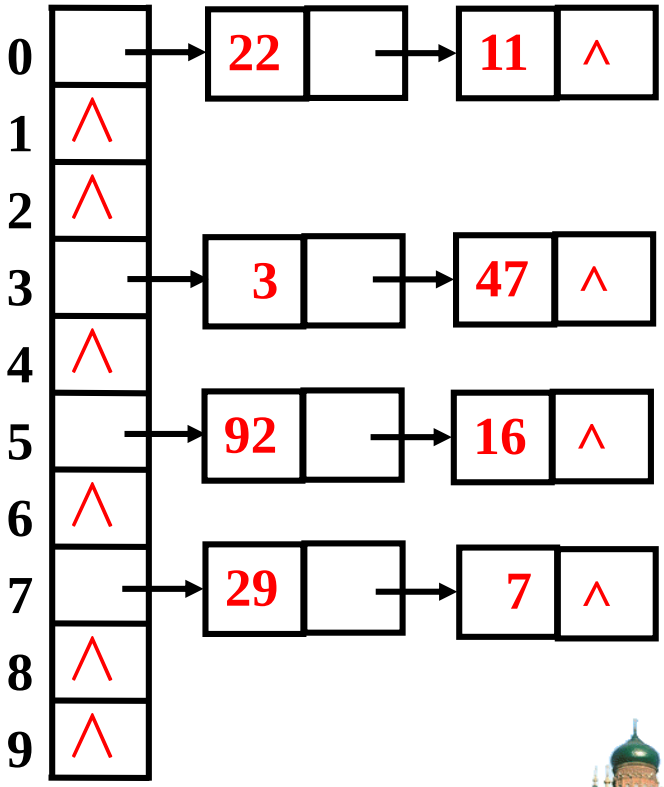


8.8 散列技术

冲突处理的方法----拉链法（链地址法）

- 开散列表的实现
  - 存储结构定义

```
struct celltype{
        records data;
        celltype *next;
};//链表结点类型
typedef  celltype  *cellptr;
//开散列表类型，B为桶数
typedef cellptr  HASH[B];
```

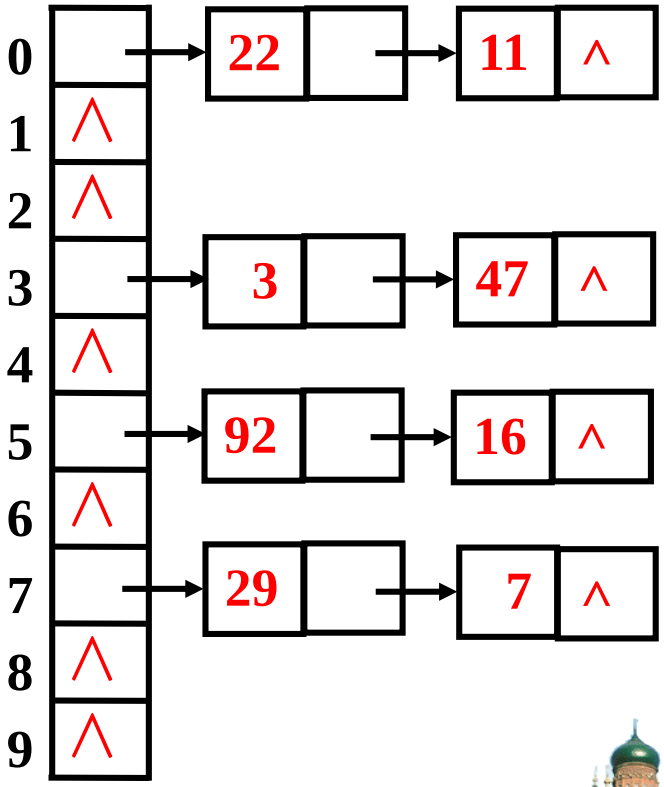


8.8 散列技术

冲突处理的方法----拉链法（链地址法）

- 开散列表的实现
  - 查找算法

```
cellptr Search(keytype k, HASH F)
{    cellptr bptr;
     bptr=F[h(k)];
     while(bptr!=NULL)
          if(bptr->data.key==k)
               return bptr;
          else
               bptr=bptr->next;
    return bptr;//没找到
}//Search
```

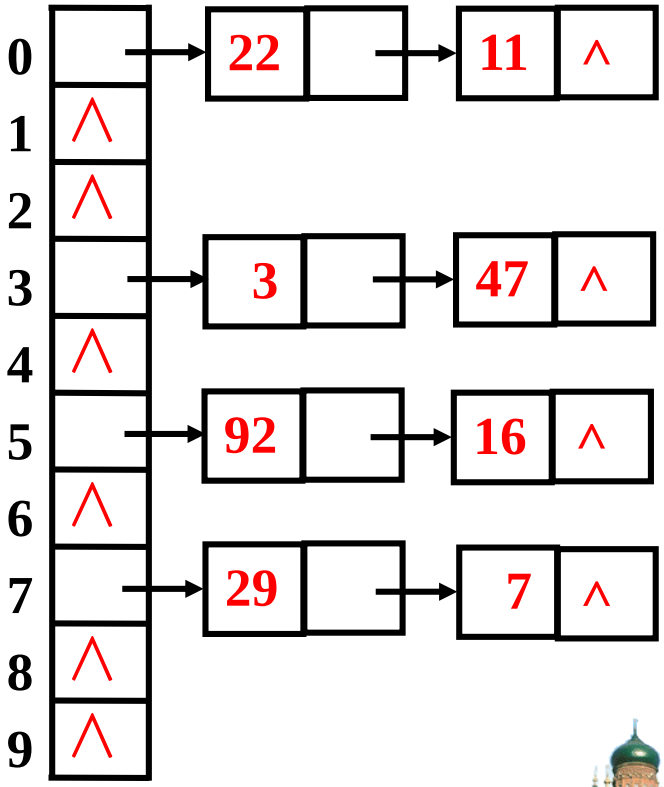


8.8 散列技术

冲突处理的方法----拉链法（链地址法）

- 开散列表的实现
  - 插入算法

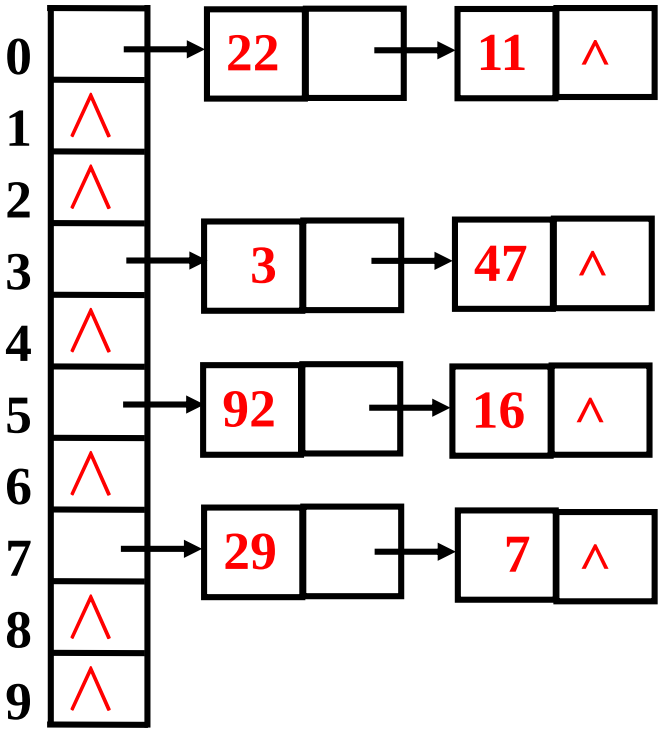
```
void Insert(records R, HASH  F)
{     int bucket;
      cellptr oldheader;
      if( SEARCH(R.key,F)==NULL){
           bucket=h(R.key);
           oldheader=F[bucket];
           F[bucket]=new celltype;
           F[bucket]->data=R;
           F[bucket]->next=oldheader;
   }
}//Insert
```



8.8 散列技术

冲突处理的方法----拉链法（链地址法）

- 开散列表的实现
  - 删除算法




```
void Delete(keytype k, HASH F)
{ int bucket=h(k); celltype bptr, p;
  if(F[bucket] != NULL)//可能在表中
    if(F[bucket]->data.key==k){//首元素就是
      bptr= F[bucket];
      F[bucket]=F[bucket]->next;
      free (bptr);
    }else{//可能在中间或不存在
       bptr=F[bucket];
       while(bptr->next!=NULL)
           if(bptr->next->data.key==k){
              p=bptr->next;
              bptr->next=p->next;
               free( p);
           }else
                bptr=bptr->next;
    }
}// Delete
```

8.8 散列技术

冲突处理的方法----开放定址法----线性探测法(c=1)

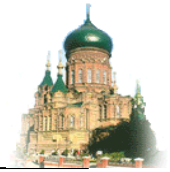
- 基本思想：当发生冲突时，从冲突位置的下一个位置起，依次寻找空的散列地址。
- 示例：关键字取值集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为 *h*(*key*)=*key* %11，用线性探测法处理冲突，则散列表为：

0	1	2	3	4	5	6	7	8	9
11	22		47	92	16	3	7	29	8
22			3	3	3		29	8	

- 堆积现象：在处理冲突的过程中出现的非同义词之间对同一个散列地址争夺的现象。

- 探测序列：设关键字值*key*的散列地址为*h*(*key*)，闭散列表的长度为*B*，则发生冲突时，寻找下一个散列地址的公式为：

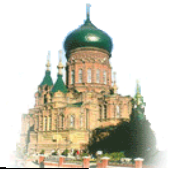
$$h_i=(h(key)+d_i)\ \%\ B \quad (d_i=1,\ 2,\ \ldots,\ m-1)$$



8.8 散列技术

冲突处理的方法----开放定址法

- 基本思想：
  - 当冲突发生时，使用某些探测技术在散列表中形成一个探测序列，沿此序列逐个单元查找，直到碰到一个开放地址（即该空的地址单元、空桶）为止。
- 常用的探测技术----如何寻找下一个空的散列地址?
  - 线性探测法
  - 线性补偿探测法
  - 二次探测法
  - 随机探测法
- 闭散列表：用开放定址法处理冲突得到的散列表也叫闭散列表.



8.8 散列技术

冲突处理的方法----开放定址法

- 二次探测法：当发生冲突时，寻找下一个散列地址的公式为：

$$h_i=(h(key)+d_i)\ \%\ B$$

（$d_i=1^2$，-1^2，2^2，-2^2，…，q^2，$-q^2$且$q\le B/2$）

例如：关键字集合 { 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数 H(key) = key MOD 11 (表长=11)

若采用二次探测再散列处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11

$d_i = 1^2,-1^2,2^2,-2^2,\ldots$

对于68，增量为2^2

8.8 散列技术

冲突处理的方法----开放定址法

- **线性补偿探测法**：当发生冲突时，寻找下一个散列地址的公式为：

$$h_i=(h(key)+d_i)\ \%\ B \quad （d_i=1c，2c，\ldots）$$

- **随机探测法**：当发生冲突时，下一个散列地址的位移量是一个随机数列，即寻找下一个散列地址的公式为：

$$h_i=(h(key)+d_i)\ \%\ B$$

（其中，$d_1, d_2, \ldots, d_{B-1}$是$1, 2, \ldots, B-1$的随机序列。）

注意：插入、删除和查找时，要使用同一个随机序列。

8.8 散列技术

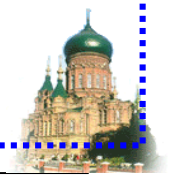
冲突处理的方法----开放定址法---线性探测法(c=1)的的实现

- 查找算法实现
  - 存储结构定义

```
struct records{
    keytype  key;
    fields other;
};
typedef records  HASH[B];
```

  - Search算法的实现

```
int  Search(keytype k, HASH F)
{   int locate=first= h(k),rehash=0;
    while((rehash<B)&&
          (F[locate].key!=empty)){
        if(F[locate].key==k)
            return locate;
        else
            rehash=rehash+1;
        locate=(first+rehash)%B
    }
    return -1;
}/*Search*/
```



8.8 散列技术

冲突处理的方法----开放定址法---线性探测法(c=1)的的实现

- 查找算法实现
  - Insert算法的实现
  - Delete算法的实现

```
void Delete(keytype k,HASH F)
{       int locate;
        locate = Search(k,F);
        if( locate != -1)
        F[locate].key = deleted;
}/*Delete*/
```

```
void Insert(records R, HASH F)
{   int locate = first=h(k), rehash= 0;
    while((rehash<B)&&
            (F[locate].key!=R.key)) {
      locate=(first+rehash)%B;
      if((F[locate].key==empty)||
            (F[locate].key==deleted))
          F[locate]=R;
      else
          rehash+=1; }
      if(rehash>=B)
          cout<<“hash table is full!";
} /*Insert */
```

8.8 散列技术

冲突处理的方法-----再散列法

- 基本思想：
  - 再散列法（再哈希法）：除了原始的散列函数 H(key) 之外，多准备几个散列函数， 当散列函数冲突时，用下一个散列函数计算一个新地址，直到不冲突为止

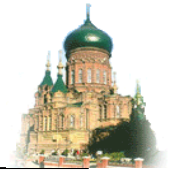
$$H_i = Rh_i(key) \quad i=1,2,\ldots,k$$

8.8 散列技术

散列查找的性能分析

- 由于冲突的存在，产生冲突后的查找仍然是给定值与关键码进行比较的过程。
- 在查找过程中，关键码的比较次数取决于产生冲突的概率。而影响冲突产生的因素有：
  - 散列函数是否均匀
  - 处理冲突的方法
  - 散列表的装载因子

    α=表中填入的记录数/表的长度



8.8 散列技术

散列查找的性能分析

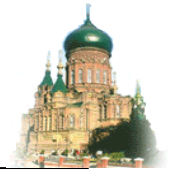
- 哈希表的平均查找长度是 α 的函数，用哈希表构造查找表时，选择适当的装填因子 α，使得平均查找长度限定在某个范围内。
- 一般情况下，可以认为选用的哈希函数是“均匀”的，也就是在讨论ASL时，认为各个位置被存数据的概率是相等的。

成功查找平均查找长度：ASL_s

查找到散列表中已存在结点的平均比较次数。

失败查找平均查找长度：ASL_u

查找失败，但找到插入位置的平均比较次数。



8.8 散列技术

散列查找的性能分析

- 几种不同处理冲突方法的平均查找长度

ASL / 处理冲突方法	查找成功时	查找不成功时
线性探测法	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$	$\frac{1}{2}(1+\frac{1}{1-\alpha^2})$
二次探测法 再散列法	$-\frac{1}{\alpha}\ln(1+\alpha)$	$\frac{1}{1-\alpha}$
拉链法	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$

【例】将序列13，15，22，8，34，19，21插入到一个初始时为空的哈希表，哈希函数采用H(x)=1+(x%7)：

(1)使用线性探测法解决冲突；

(2)使用步长为3的线性探测法解决冲突；

(3)使用再哈希法解决冲突，冲突时的哈希函数

H(x)=1+(x%6)。

解：设哈希表长度为8。

(1) 使用线性探测法解决冲突，即步长为1，对应地址为：

H(13)=1+(13%7)=7；
H(15)=1+(15%7)=2；
H(22)=1+(22%7)=2(冲突)，　$H_1(22)=(2+1)\%8=3$；
H(8)=1+(8%7)=2(冲突)，　$H_1(8)=(2+1)\%8=3$　(仍冲突)
　　$H_2(8)=(3+1)\%8=4$；
H(34)=1+(34%7)=7(冲突)，　$H_1(34)=(7+1)\%8=0$；
H(19)=1+(19%7)=6；
H(21)=1+(21%7)=1；

哈希表

地址	0	1	2	3	4	5	6	7
Key	34	21	15	22	8		19	13
探测次数	2	1	1	2	3		1	1

(2) 使用步长为3的线性探测法解决冲突，对应地址为：

H(13)=1+(13%7)=7；

H(15)=1+(15%7)=2；

H(22)=1+(22%7)=2(冲突)， $H_1(22)=(2+3)\%8=5$；

H(8)=1+(8%7)=2 (冲突)， $H_1(8)=(2+3)\%8=5$ (仍冲突)

$H_2(8)=(5+3)\%8=0$；

H(34)=1+(34%7)=7(冲突)， $H_1(34)=(7+3)\%8=2$ (仍冲突)

$H_2(34)=(2+3)\%8=5$ (仍冲突)

$H_3(34)=(5+3)\%8=0$ (仍冲突)

$H_4(34)=(0+3)\%8=3$；

H(19)=1+(19%7)=6；

H(21)=1+(21%7)=1；

哈希表

地址	0	1	2	3	4	5	6	7
Key	8	21	15	34		22	19	13
探测次数	3	1	1	5		2	1	1

(3) 使用再哈希法解决冲突， 再哈希函数为H(x)=1+(x%6)

H(13)=1+(13%7)=7；

H(15)=1+(15%7)=2；

H(22)=1+(22%7)=2(冲突)， H_1(22)=1+(22%6)=5；

H(8)=1+(8%7)=2 (冲突)， H_1(8)=1+(8%6)=3；

H(34)=1+(34%7)=7(冲突)， H_1(34)=1+(34%6)=5 (仍冲突)

H_2(34)=1+(34%5)=5 (仍冲突)

H_3(34)=1+(34%4)=3 (仍冲突)

H_4(34)=1+(34%3)=2 (仍冲突)

H_5(34)=1+(34%2)=1 ；

H(19)=1+(19%7)=6；

H(21)=1+(21%7)=1(冲突) ， H_1(21)=1+(21%6)=4；

哈希表

地址	0	1	2	3	4	5	6	7
Key		34	15	8	21	22	19	13
探测次数		6	1	2	2	2	1	1

【例】已知一组记录的键值为{1,9,25,11,12,35,17,29}，
请构造一个散列表。

（1）采用除留余数法构造散列函数，线性探测法处理冲突，要求新插入键值的平均探测次数不多于2.5次，请确定散列表的长度m及相应的散列函数。分别计算查找成功和查找失败的平均查找长度；$ASL_{失败} = \frac{1}{2}(1+\frac{1}{1-\alpha})$

（2）采用（1）中的散列函数，但用链地址法处理冲突，构造散列表，分别计算查找成功和查找失败的平均查找长度。

解：（1）$ASL_u = \frac{1}{2}(1+\frac{1}{1-\alpha}) \leq 2.5$ 得 $\alpha \leq 1/2$

因为 $8/m \leq 1/2$, 所以 $m \geq 16$

取 m=16 , 散列函数为 H(key)=key%13

给定键值序列为:{1,9,25,11,12,35,17,29}

散列地址为： d:1,9,12,11,12,9,4,3

用线性探测解决冲突，见下表：

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key		1		29	17					9	35	11	25	12		
成功探测次数		1		1	1					1	2	1	1	2		
失败探测次数	1	2	1	3	2	1	1	1	1	6	5	4	3			

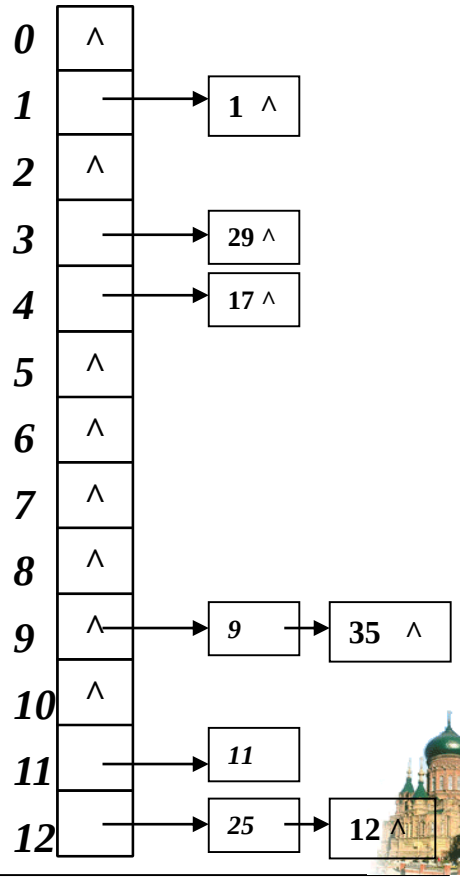
查找成功的平均查找长度：

$$ASL_s=(6+4)/8=1.25$$

查找失败的平均查找长度：

$$ASL_u=(6\times1+2+3+2+6+5+4+3)/13=2.4$$

(2)使用链地址法处理冲突构造散列表。



查找成功的平均查找长度：

$$ASL_s=(6\times1+2\times2)/8=1.25$$

查找失败的平均查找长度：

$$ASL_u=(4\times1+2\times2)/13\approx0.6$$

【例】将关键字序列（7，8，30，11，18，9，14）散列存储到散列表中，散列表的存储空间是一个下标从0开始的一维数组，散列函数为：H(key)=(key×3) MOD 7，处理冲突采用线性探测再散列法，要求装填（载）因子为0.7。

(1)请画出所构造的散列表；

(2)分别计算等概率情况下查找成功和查找不成功的平均查找长度。

下标	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	

查找成功的平均查找长度：$ASL_{成功}=12/7$

查找不成功的平均查找长度：$ASL_{不成功}=18/7$

关键字	7	8	30	11	18	9	14
散列地址	0	3	6	5	5	6	0
探查次数	1	1	1	1	3	3	2

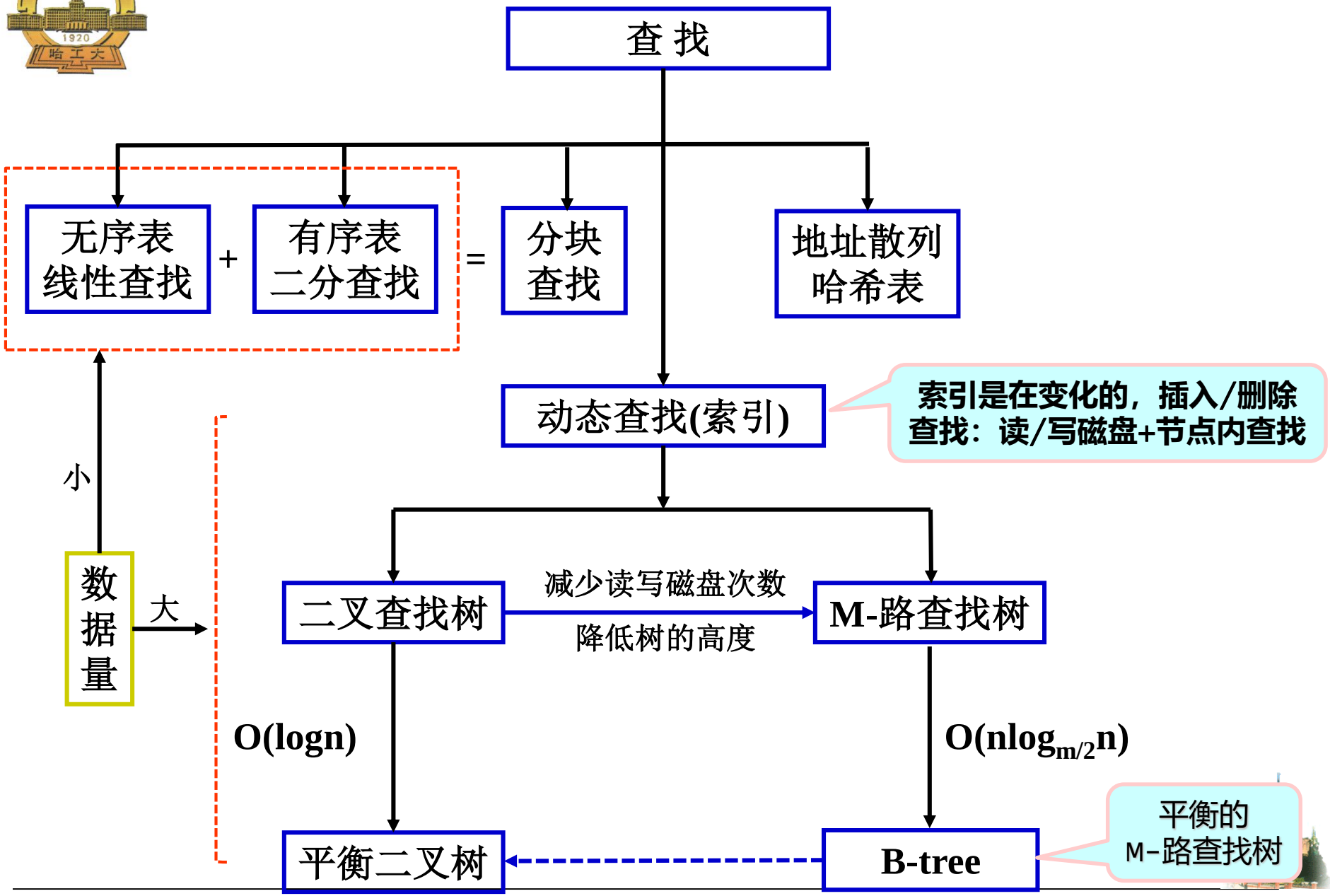
下标	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	

$ASL_{成功} = (1次\times4+2次\times1+3次\times2)/7 = 12/7$

各位置对应的探查次数如下表所示：

下标	0	1	2	3	4	5	6
探查次数	3	2	1	2	1	5	4

$ASL_{不成功} = (3+2+1+2+1+5+4)/7 = 18/7$。


查 找
无序表
线性查找
+
有序表
二分查找
=
分块
查找
地址散列
哈希表
动态查找(索引)
索引是在变化的，插入/删除
查找：读/写磁盘+节点内查找
小
数据量
大
二叉查找树
减少读写磁盘次数
降低树的高度
M-路查找树
O(logn)
O(nlog$_{m/2}$n)
平衡二叉树
B-tree
平衡的
M-路查找树

本章小结

- ✓ 熟练掌握：

  (1) 熟练掌握顺序查找、二分查找、二叉排序树基本思想、算法和应用。。

  (2) 熟练掌握散列表构造及应用。

  (3)熟练掌握二叉排序树的生成、插入和删除的注意算法；平衡二叉树的、B-树和B+树的基本知识。