数据结构与算法

Data Structures and Algorithms

第五部分 查找

回顾：查找--2

1. 平衡二叉树（AVL树）

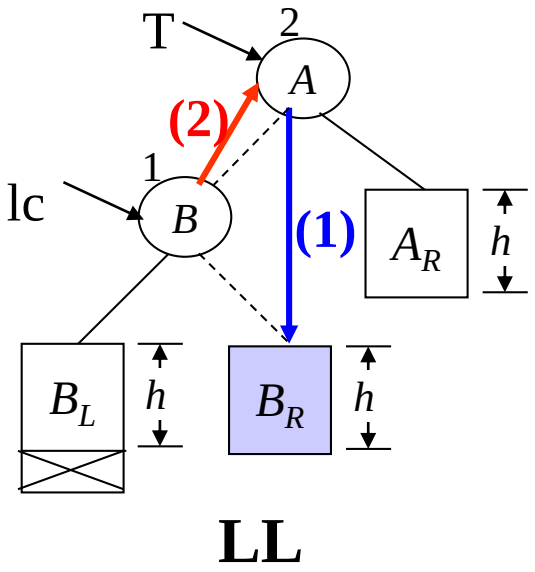
- 基本概念
- 插入算法（如何建立）

---LL、RR、LR、RL

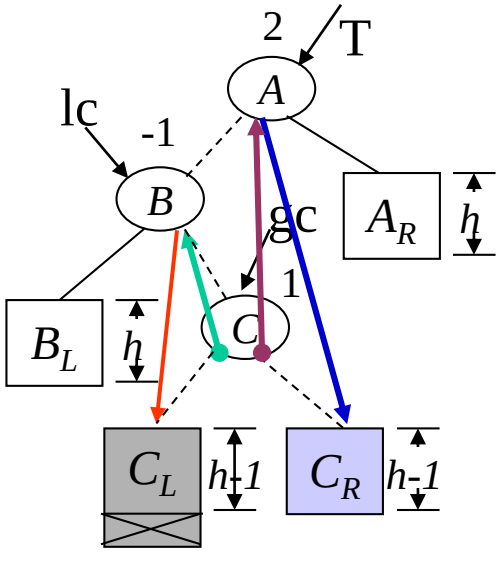
2. m-路查找树（概念）

3. B-树

- 基本概念
- B树的高度 $log_m(N+1) \le l \le log_{\lceil m/2 \rceil}\lfloor (N+1)/2 \rfloor + 1$
- B树阶数的选择
- B树的存储结构



LL



LR

5.6 B- 树 B-Tree Balanced-Tree

B-树：B-树是一种非二叉的查找树

除了要满足查找树的特性，还要满足以下结构特性。

一棵 *m* 阶的 B- 树：

（1）树的根或者是一片叶子(一个结点的树), 或者其儿子数在 2 和 *m* 之间；

（2）除根外，所有的非叶子结点的孩子数在$\lceil m/2 \rceil$和 *m* 之间；

（3）所有的叶子结点都出现在同一层上，且不带信息。

B-树的平均深度为$\log_{\lceil m/2 \rceil} N$。执行查找的平均时间为O(*logm*)；

B-树应用在数据库系统中的索引，它加快了访问数据的速度；

4、B-树的查找

（1）在B-树中找结点（读写磁盘）

（2）在结点内找关键字（内存）

一棵含有N个总关键字数的m阶B-树高度为：

$$log_m(N+1) \leq l \leq log_{\lceil m/2 \rceil}|(N+1)/2|+1$$

在含有N个关键字的B-树上进行查找时，从根结点到关键字所在结点的路径上涉及的结点数不超过 $l \leq log_{\lceil m/2 \rceil}|(N+1)/2|+1$。

4、B-树的查找

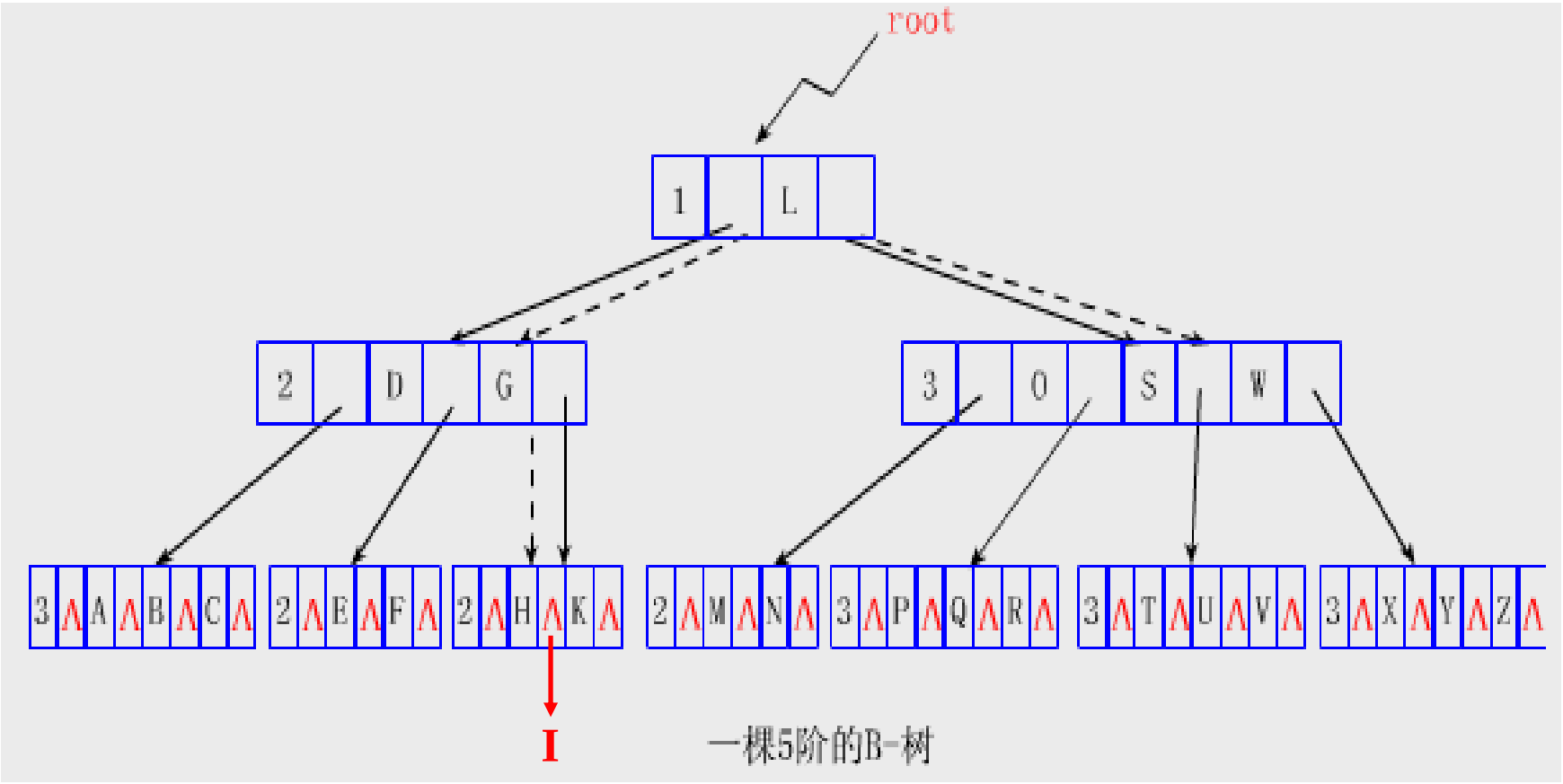
（1）B-树的查找方法

在B-树中查找给定关键字的方法类似于二叉排序树上的查找。不同的是在每个结点上确定向下查找的路径不一定是二路而是keynum+1路的。

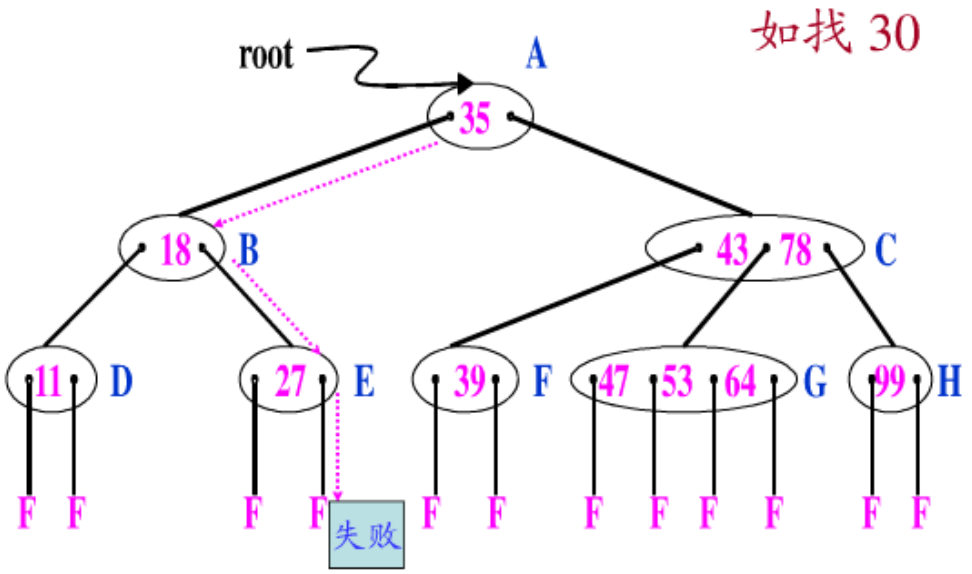
➢对结点内的存放有序关键字序列的向量key[l..keynum] 用顺序查找或折半查找方法查找；

➢若在某结点内找到待查的关键字K，则返回该结点的地址及K在key[1..keynum]中的位置； 否则，确定K在某个key[i]和key[i+1]之间结点后，从磁盘中读son[i]所指的结点继续找……。

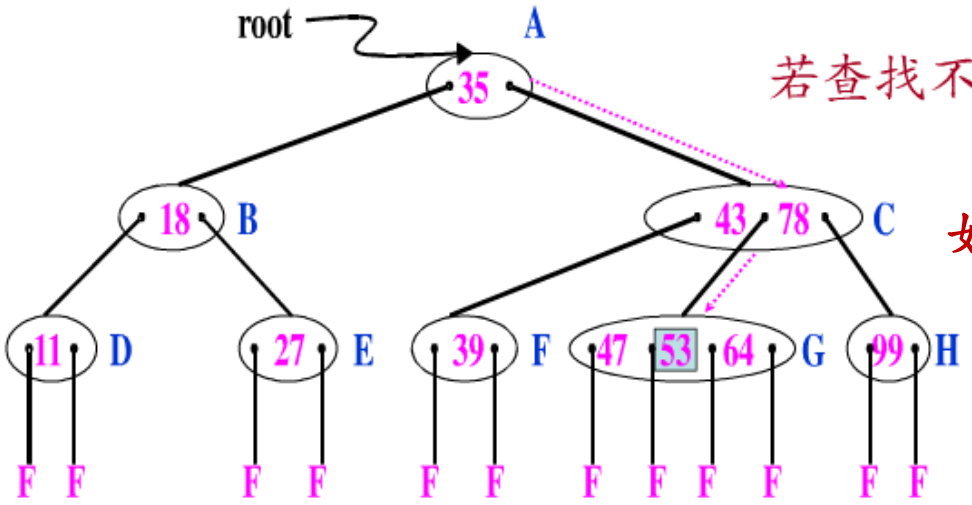
【例5-10】下图中左边的虚线表示查找关键字 I 的过程，它失败于叶结点的H和K之间空指针上；右边的虚线表示查找关键字S的过程，并成功地返回 S 所在结点的地址和S在key[1..keynum]中的位置2。



【例5-11】



若查找不成功，则返回插入位置。



如查找 53

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

（2）B-树的查找算法

```
BTreeNode *SearchBTree(BTree T，KeyType K，int *pos)
{  //在B-树T中查找关键字K，成功时返回找到的结点的地址及K在其中的位置*pos
   //失败则返回NULL
 int i;
 T->key[0]= K;                                  //设哨兵．下面用顺序查找key[1..keynum]
 for(i=T->keynum；  K< T->key[i]; i--)；  //从后向前找第1个小于等于K的关键字
 if(i>0 && T->key[i]==K){                       //查找成功，返回T及pos i
        *pos=i;
        return T;
    }          //结点内查找失败，但T->key[i]< K <T->key[i+1]，下一个查找的结点应为son[i]
 if(!T->son[i])                          //*T为叶子，在叶子中仍未找到K，则整个查找过程失败
     return NULL;
     //可增加插入操作,查找插入关键字的位置，则应令*pos=i，并返回T，见后面的插入操作
 else  DiskRead(T->son[i])；                    //在磁盘上读入下一查找的树结点到内存中
 return SearchBTree(T->son[i]，K，pos)；     //递归地继续查找于树T->son[i]
}
```

（3）查找操作的时间开销

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；

②在结点内查找，该查找属内查找。

查找操作的时间为：

①外查找的读盘次数不超过树高h，故其时间是$O(h)$；

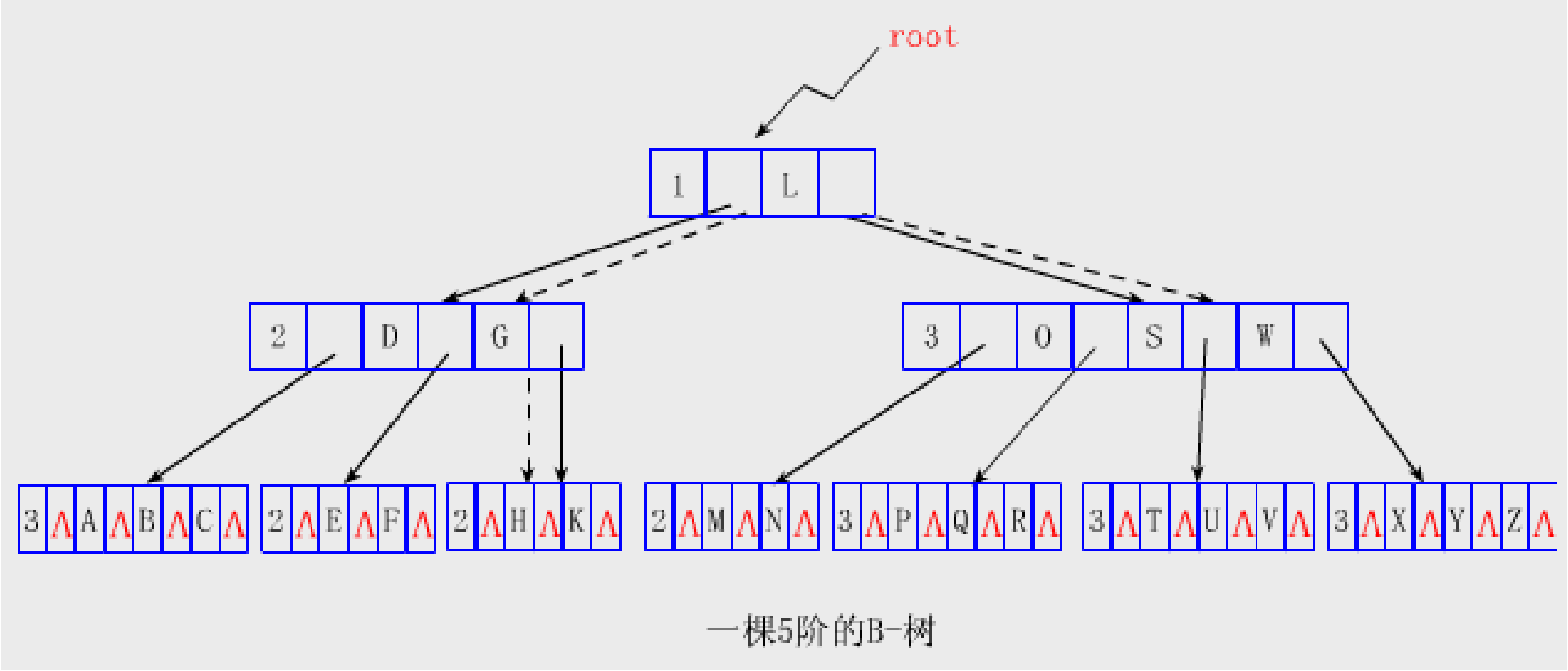
②内查找中，每个结点内的关键字数目$keynum<m$（m是B-树的阶数），故其时间为$O(nh)$。

注意：

①实际上外部查找时间可能远远大于内部查找时间；

②B-树作为数据库文件时，打开文件之后就必须将根结点读人内存，而直至文件关闭之前，此根一直驻留在内存中，故查找时可以不计读入根结点的时间。

【例5-12】一棵包含24个英文字母的5阶B-树的存储结构图。

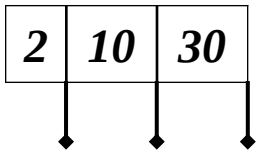


一棵5阶的B-树

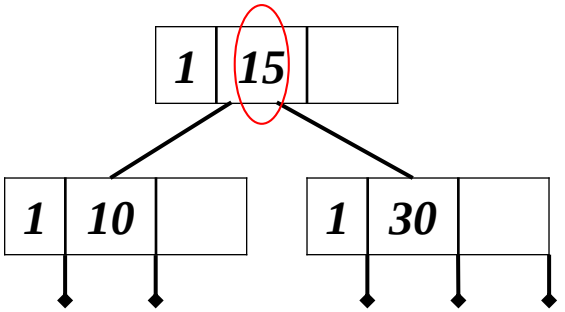
按照定义，在5阶B-树里，根中的关键字数目可以是1～4，子树数可以是2～5；其它的结点中关键字数目可以是2～4，若该结点不是叶子，则它可以有3～5棵子树。

5、B-树的插入操作

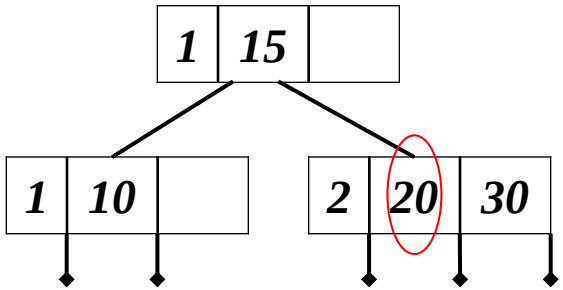
【例5-13】3阶的B-树的插入

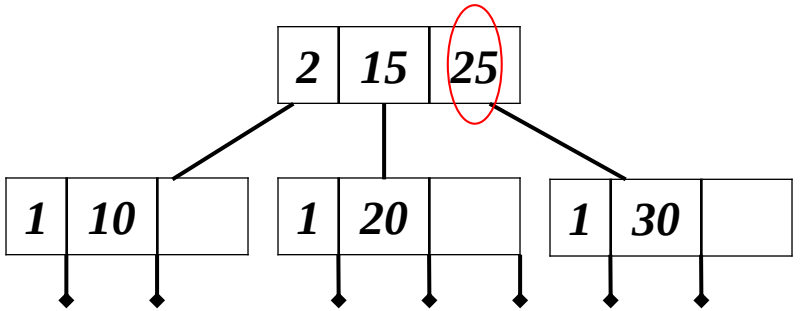


未分裂



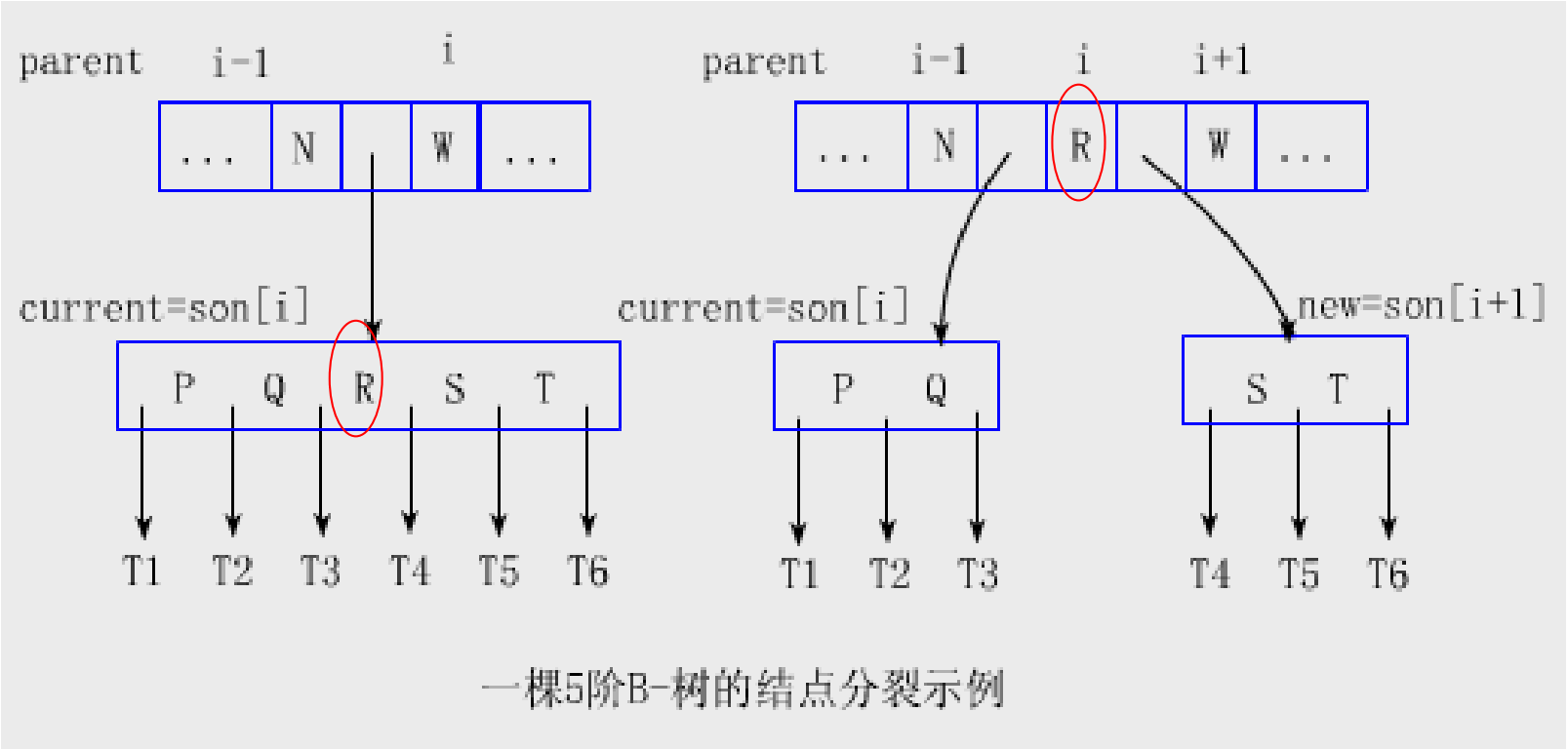
分裂1次





分裂2次

【例5-14】



一棵5阶B-树的结点分裂示例

【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向上插入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)

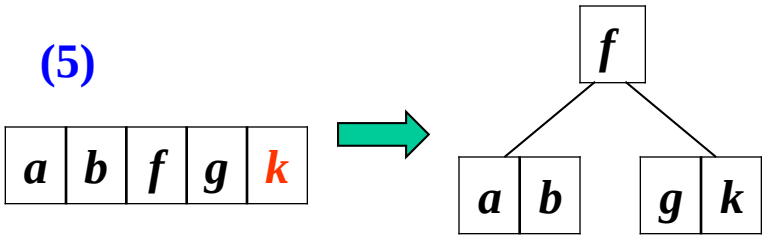
(1) a

(2) a g

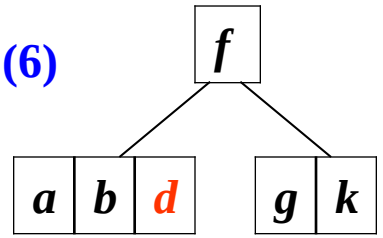
(3) a f g

(4) a b f g

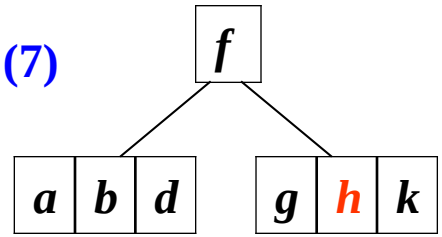
(5)



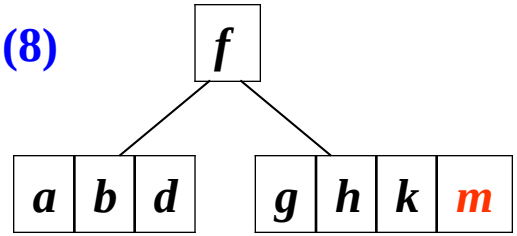
(6)

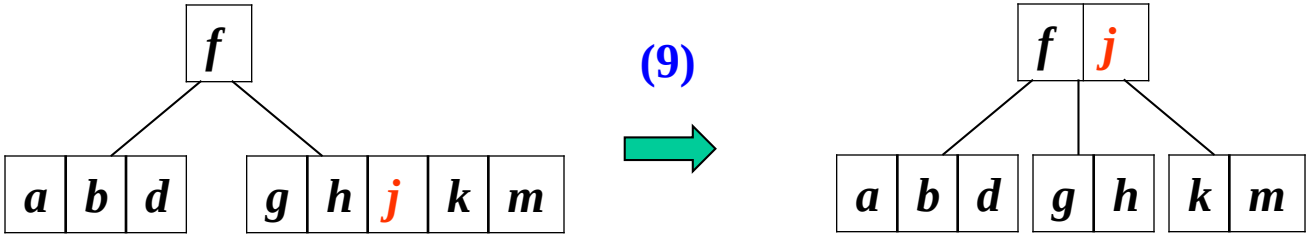


(7)

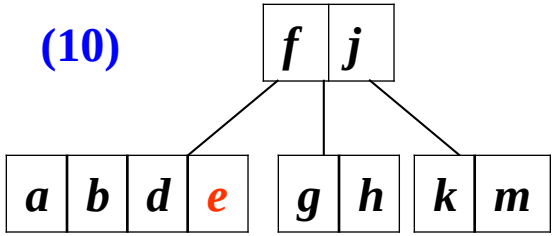


(8)

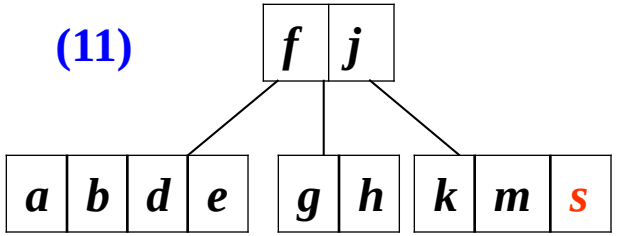




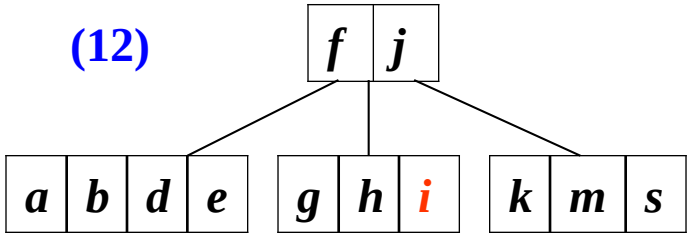
(10)


f j
a b d e g h k m

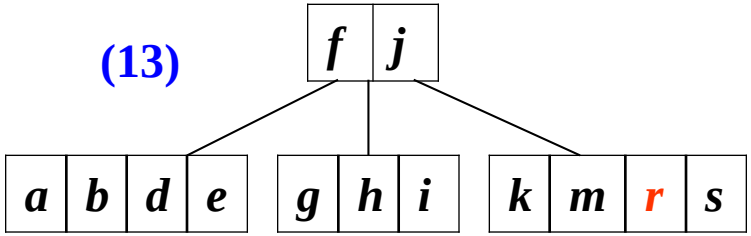
(11)

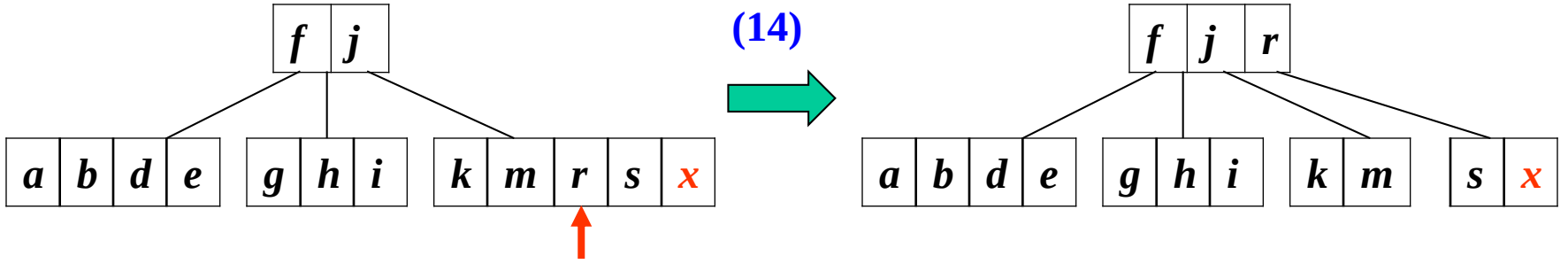

f j
a b d e g h k m s

(12)

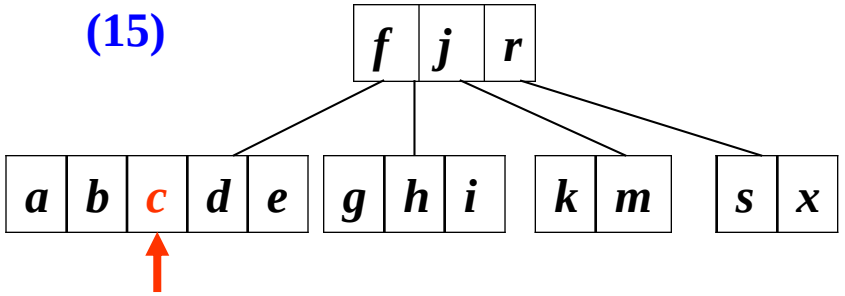

f j
a b d e g h i k m s

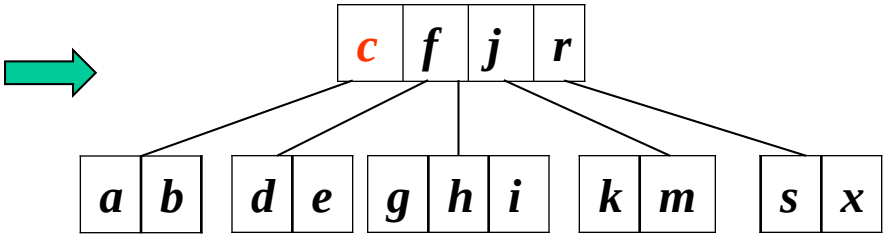
(13)


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

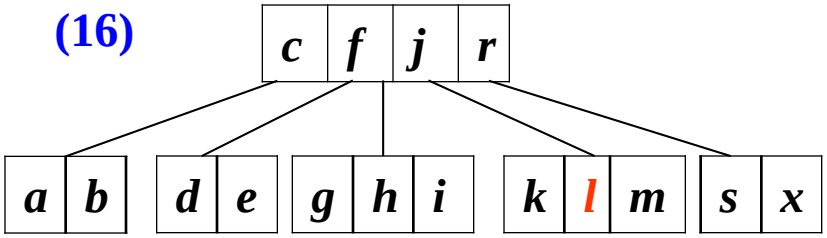

f j
a b d e g h i k m r s x
(14)
f j r
a b d e g h i k m s x

(15)

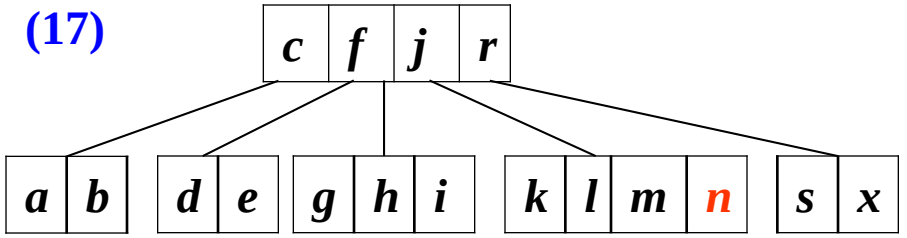

f j r
a b c d e g h i k m s x


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

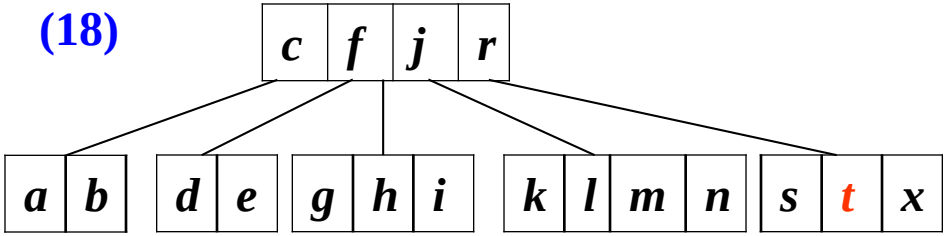
(16)


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

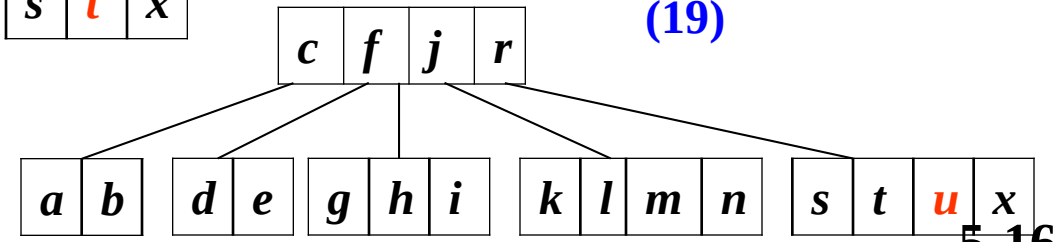
(17)


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

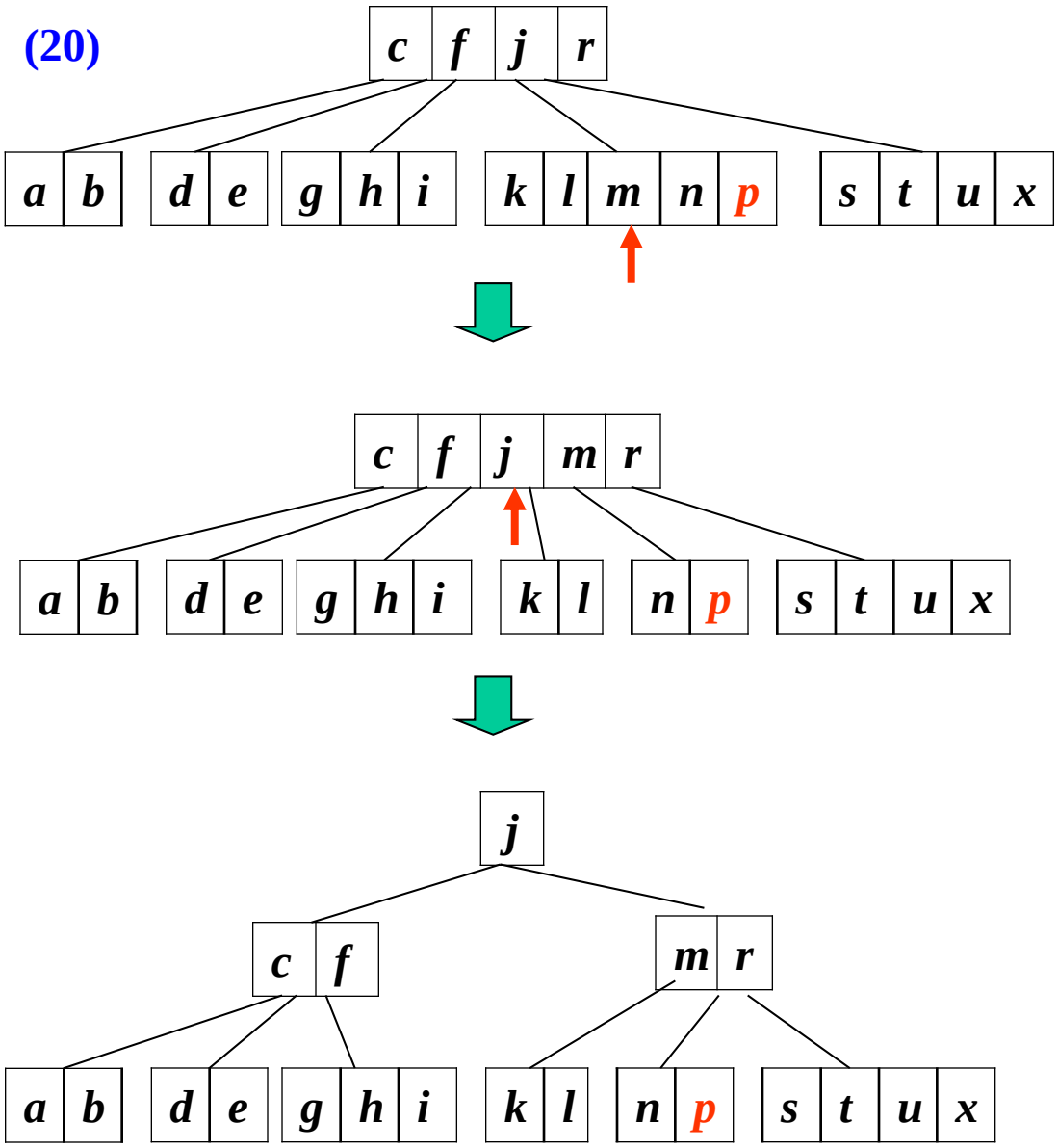
(18)


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向上插入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最下层非叶子结点 p。
如果在结点p上插入新关键字使得结点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：

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

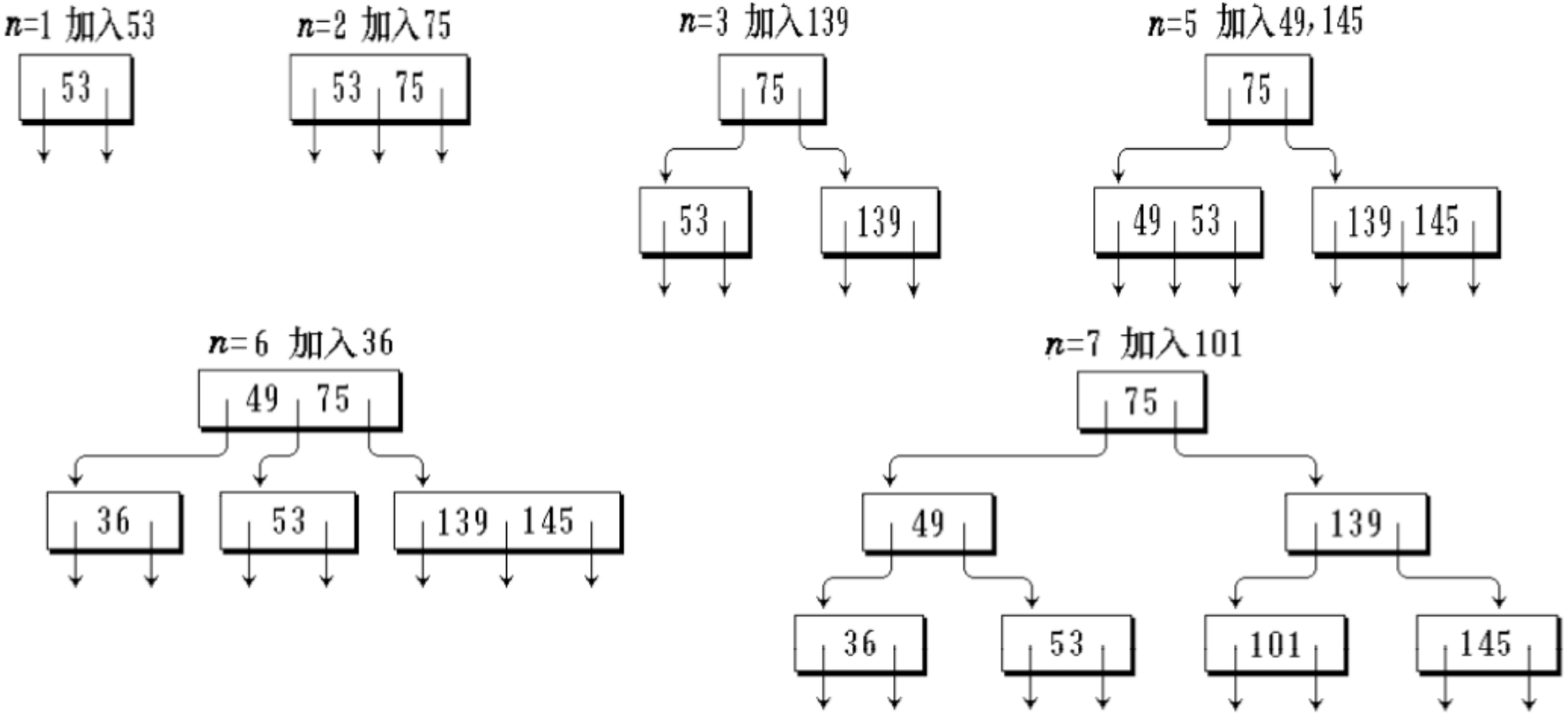
分裂后，形成具有如下格式的两个结点 p 和 q：

结点 p：$\lceil m/2\rceil-1, A_0, (K_1,A_1), \ldots\ldots, (K_{\lceil m/2\rceil-1}, A_{\lceil m/2\rceil-1})$
结点 q：$m-\lceil m/2\rceil, A_{\lceil m/2\rceil}, (K_{\lceil m/2\rceil+1}, A_{\lceil m/2\rceil+1}), \ldots\ldots, (K_m, A_m)$

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

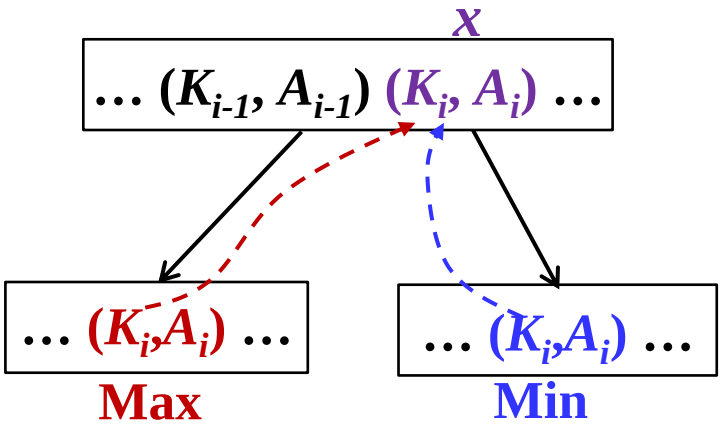
B-树的插入操作与建立



- 在插入新关键字时，需要自底向上分裂结点，最坏情况下从被插关键字所在终端结点到根的路径上的所有结点都要分裂。

6、B-树的删除操作

- 首先, 找到要删除的关键字*x*所在的结点 *z* ;
- 若 *z* 不是叶结点，则用该 B-树中某个叶结点中的关键字来代替结点 *z* 中的 *x*;
- 假定 *x* 是 *z* 中的第 *i* 个关键字（$x=K_i$），就可以用子树A_i中的最小关键字或者子树A_{i-1}中的最大关键字来替换 *x*，而这两个关键字均在叶结点中。 =〉把从非叶结点中删除*x*的操作转化成从终端结点中删除某个结点的操作。



【例5-16】删除5阶B-树的h、r、p、 d等关键字的过程

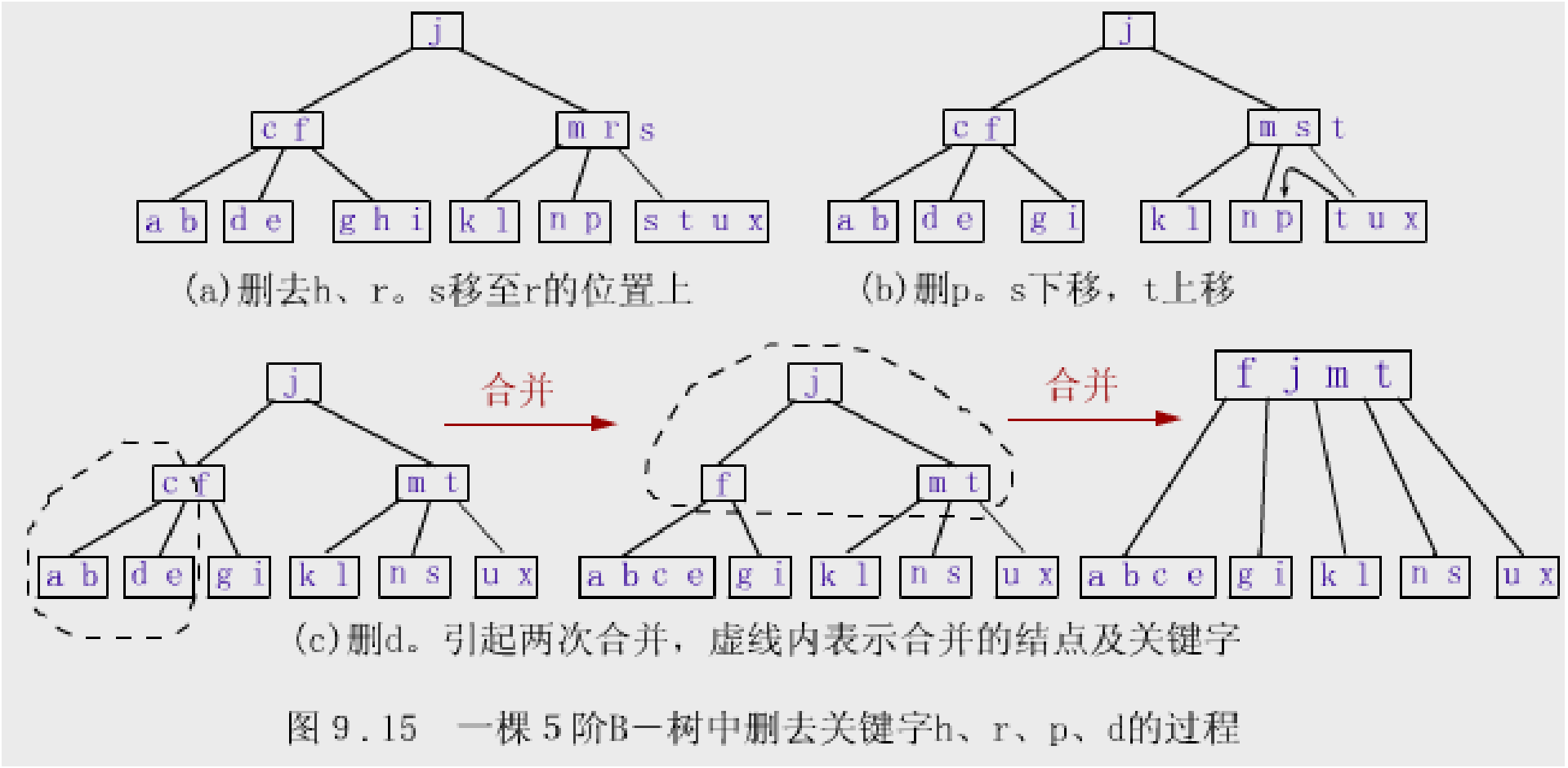


图 9.15 一棵 5 阶B—树中删去关键字h、r、p、d的过程

例16的过程分析：

第1个被删的关键字h是在叶子中，且该叶子的关键字个数>⌈m/2⌉-1(5阶B-树的Min=2)，故直接删去即可；

第2个删去的r不在叶子中，故用中序后继s取代r，即把s复制到r的位置上，然后从叶子中删去s；

第3个删去的p所在的叶子中的关键字数目是最小值Min，但其右兄弟的关键字个数 >⌈m/2⌉-1,故可以通过左移，将双亲中的s移到p所在的结点，而将右兄弟中‘最小(即最左边)的关键字t上移至双亲取代s；

当删去d时，d所在的结点及其左右兄弟均无多余的关键字，故需将删去d后的结点与这两个兄弟中的一个(图中是选择左兄弟(ab))及其双亲中分隔这两个被合并结点的关键字 c 合并在一起形成一个新结点(abce)。但因为双亲中失去c后关键字个数 <m/2-1，故必须对该结点做调整操作，此时它只有一个右兄弟，且右兄弟无多余的关键字，不可能通过移动关键字来解决。因此引起再次合并，因根只有一个关键字，故合并后树高度减少一层，从而得到上图的最后一个图。

删除小结

从终端结点 *p* (最低层非叶子结点)删除关键字后，分 *4* 种不同情况分别进行处理：

① p同时也是根结点，若删除x后还至少剩一个关键字，写盘，结束；

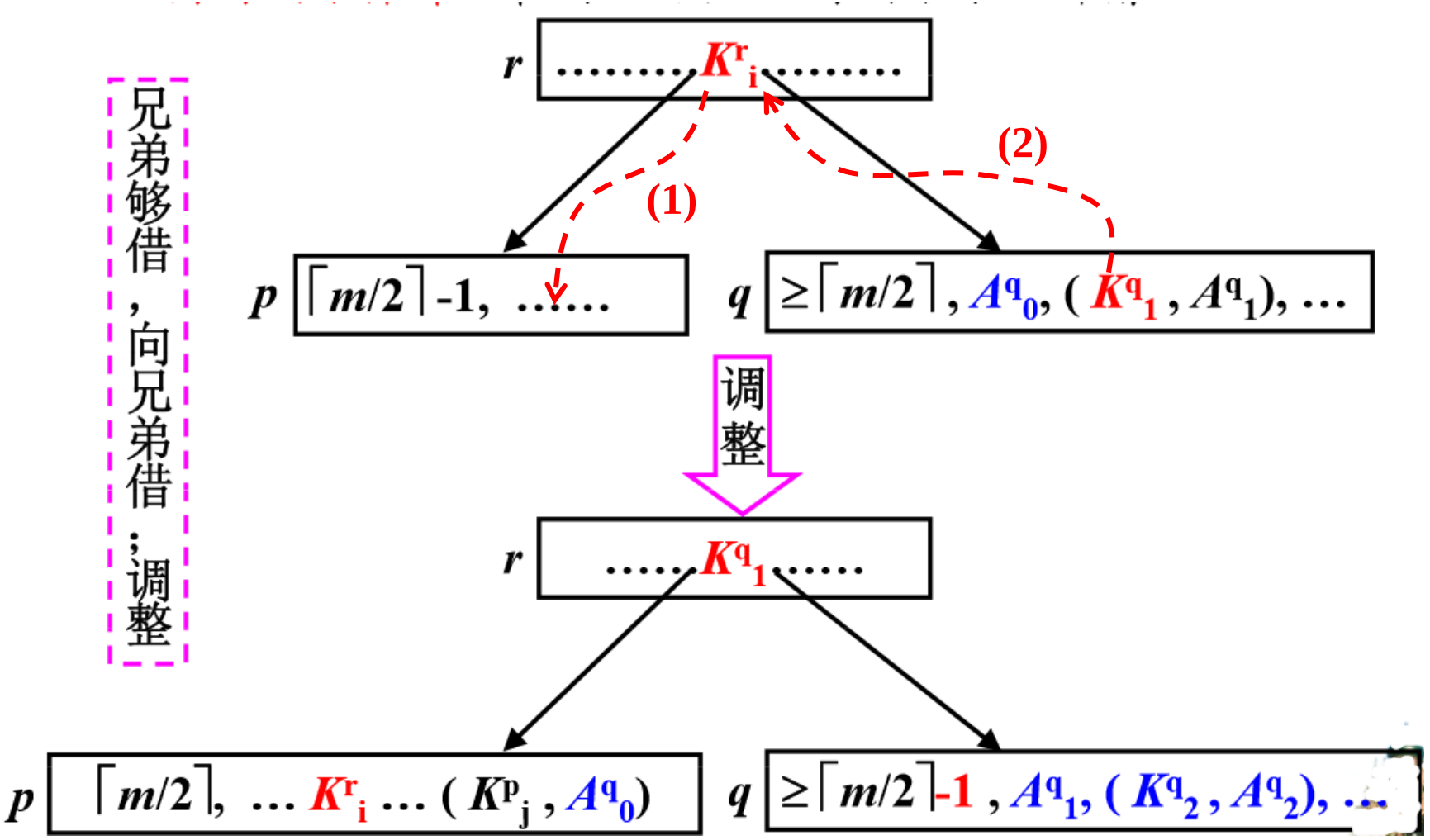
否则，B-树变成空树；

② 删除*x*后，结点*p*至少还有$\lceil m/2 \rceil - 1$个关键字，写盘，结束；

③ 删除*x*后，结点*p*有$\lceil m/2 \rceil - 2$个关键字，其邻近的右兄弟（左兄弟）结点*q*至少还有$\lceil m/2 \rceil$个关键字。

假定*r*是*p*和*q*的父结点，设K_i是*r*中大于（或小于）*x*的最小（或最大）关键字，将K_i下移至*p*，把 *q* 的最小（或最大）关键字上移到*r*的K_i处，把*q*的最左（或最右）子树指针平移到*p*中最后（或最前）子树的指针处，在*q*中，将被移走的关键字和指位置有剩余的关键字和指针补齐，再将其中的关键字个数减*1*。将*p*、*q*、*r*写盘，结束；

B-树-删除-在终端结点上的删除第三种情况（父子换位法）

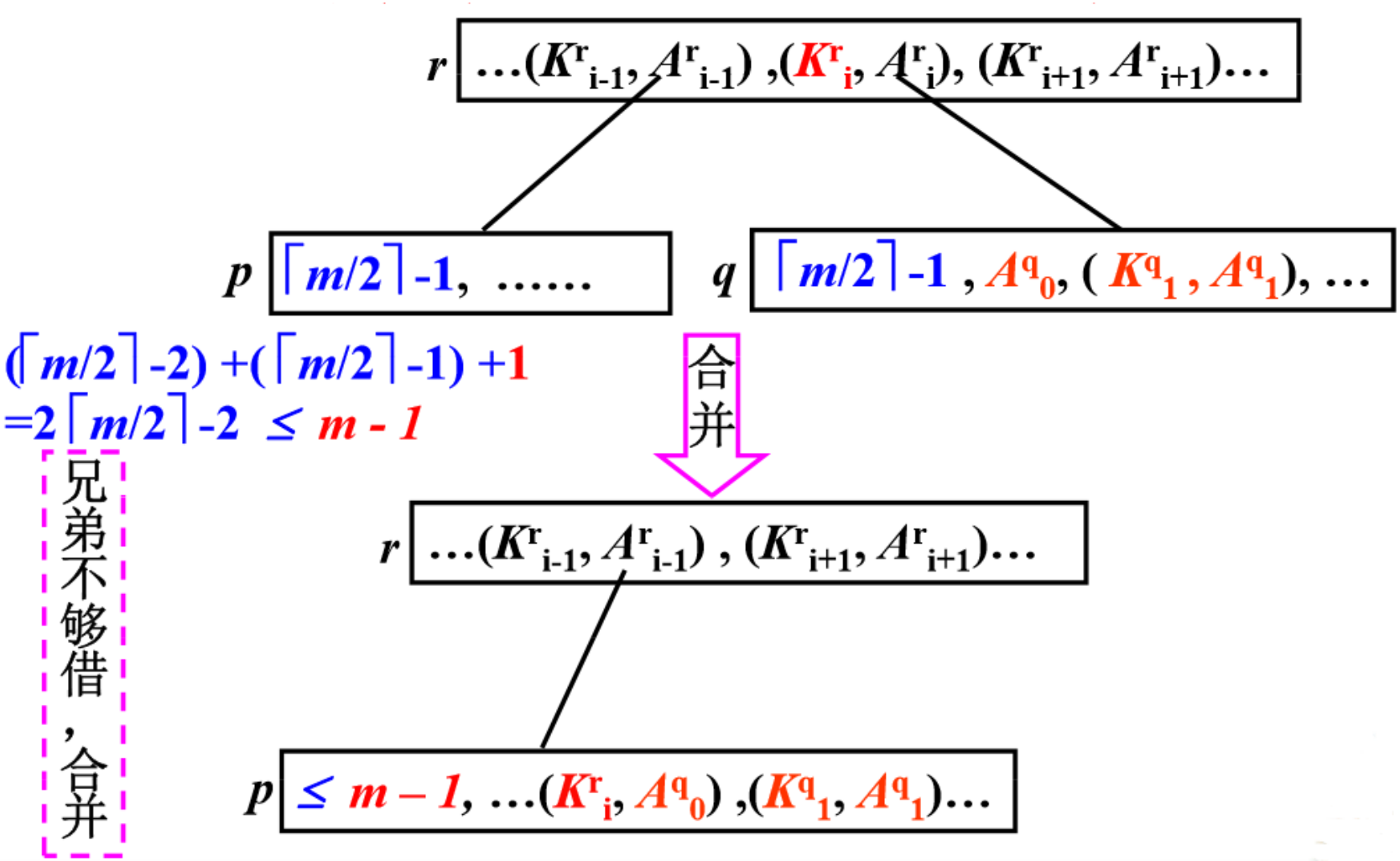


删除小结（续）

④ 删除x后结点p有$\lceil m/2 \rceil -2$个关键字，其邻近的右兄弟（左兄弟）结点q至少还有$\lceil m/2 \rceil -1$ 个关键字。

将p、q和关键字K_i合并，形成一个结点，有少于$m-1$个关键字。合并操作使父结点r的关键字个数减1。若r中关键字个数仍满足要求（根至少1个关键字，非根结点至少m/2-1个关键字），写盘，结束。否则，父结点r的关键字不足，若是根，其中无关键字，删除；若r不是根结点，其关键字个数应为m/2-2。要与其兄弟结点合并，并沿B-树向上进行，直到根结点的儿子被合并为止。

B-树-删除-在终端结点上的删除第四种情况



B-树的高度及性能分析

B-树上操作的时间由存取磁盘的时间和CPU计算时间构成。

➢B-树上大部分基本操作所需访问盘的次数均取决于树高*h*。关键字总数相同的情况下B-树的高度越小，磁盘I/O所花的时间越少；

➢与高速的CPU计算相比，磁盘I/O要慢得多，所以可忽略CPU的计算时间，只分析算法所需的磁盘访问次数：

（磁盘访问次数）×（读写盘的平均时间）。

(1) B-树的高度

若n≥1，m≥3，则对任意一棵具有n个关键字的m阶B-树，其树高h至多为：$\log_t((n+1)/2)+1$。

t 是每个(除根外)内部结点的最小度数，即⌈m/2⌉

➢B-树的高度为$O(\log_t n)$;
➢在B-树上查找、插入和删除的读、写盘的次数为$O(\log_t n)$;
➢CPU计算时间为$O(m\log_t n)$。

(2) 性能分析

① 树高。n个结点的平衡的二叉排序树的高度H（即lgn）比B-树的高度h约大lgt倍。

【例5-17】若m=1024，则lgt=lg512=9。此时若B-树高度为4，则平衡的二叉排序树的高度约为36。

显然，若m越大，则B-树高度越小；

② 结点内查找。若要作为内存中的查找表，B-树却不一定比平衡的二叉排序树好，尤其当m较大时更是如此。因为查找等操作的CPU计算时间在B-树上是: $O(m\log_t n)=0(\lg n\cdot(m/\lg t))$;

而m/logt>1，所以m较大时$O(m\log_t n)$比平衡的二叉排序树上相应操作的时间 O(lgn)大得多。因此，仅在内存中使用的B-树必须取较小的m。

③ 阶数。通常取最小值m=3，此时B-树中每个内部结点可以有2或3个孩子，这种3阶的B-树称为2-3树。

B+树（了解基本概念）

B+树是B-树的一种变形，二者区别在于：

(1) 有k个子结点的结点必然有k个关键码；

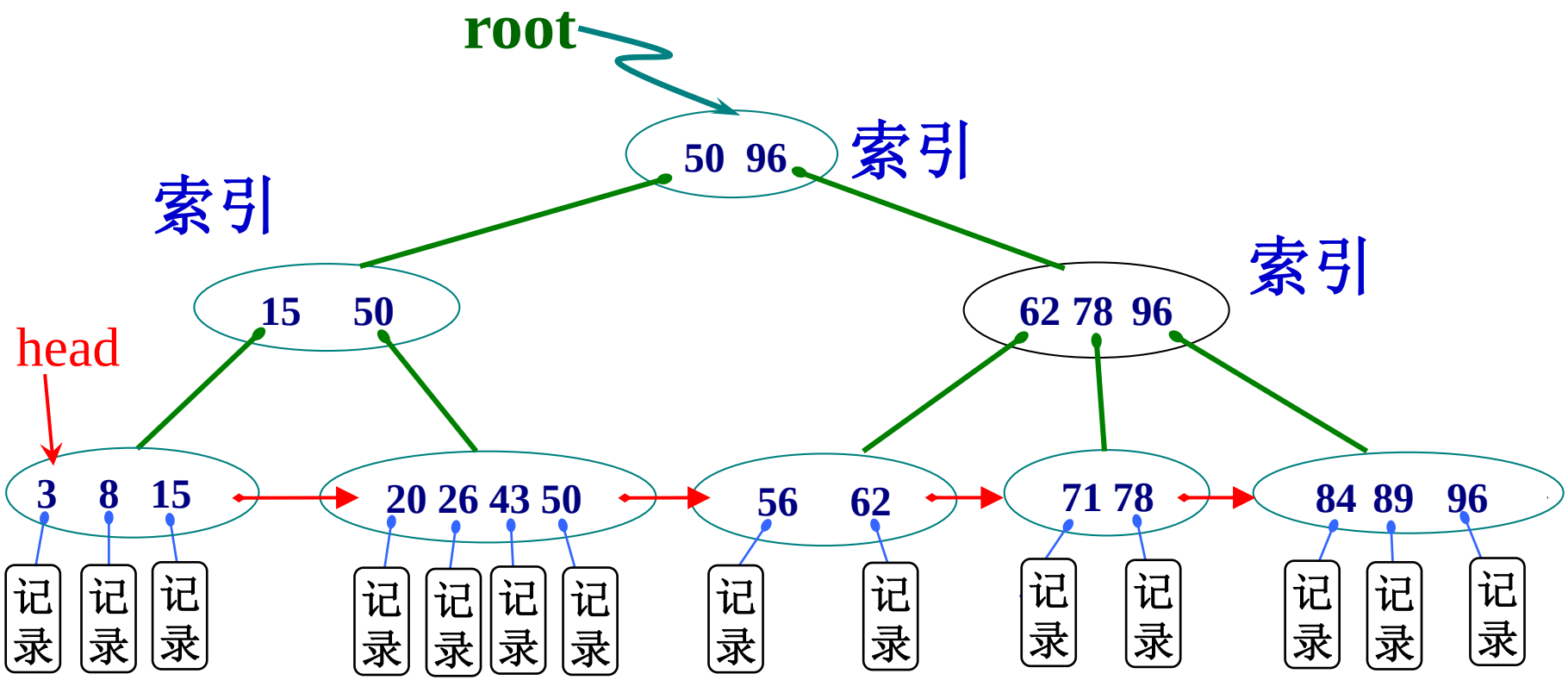
(2) 非叶子结点仅具有索引作用(存该子树最大或最小关键码)，与记录有关的信息均放在叶结点中。

查找：B+树查找过程中，即使找到关键码，也必须向下找到叶子结点；

插入：插入在叶子上，分裂，改变上层两个最大关键码；

删除：删除在叶结点，上层关键码可以保留，删除后若关键字少于m/2，则与B-同样处理。

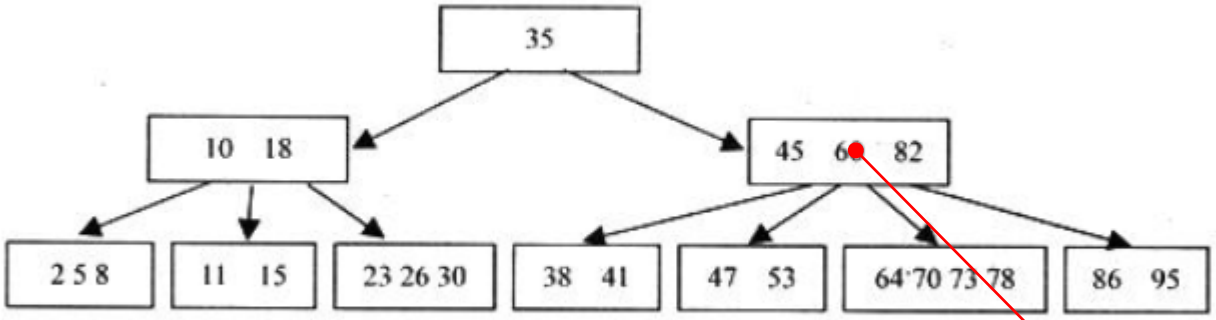
B-树只适合随机检索，但B+树同时支持随机检索和顺序检索。



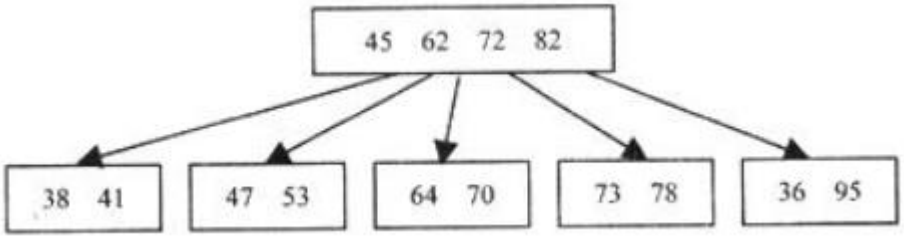
所有叶子结点都处在同一层次上构成一个有序链表；

头指针指向最小关键字的结点

试题基于以下的5阶B树结构，该B树现在的层数为2。



往该B树中插入关键码72后，该B树的第2层的结点数为_____。



从该B树中删除关键码15后，该B树的第2层的结点数为_____。

5.7 散列法——哈希（Hash）查找

散列法——地址散列法

被查找元素的存储地址 = Hash (*Key*)

【例5-17】假设有30个地区的人口统计，每个地区为一个记录，内容如下：

编号	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；

$f_3(key)$：将第一个字母的八进制看成十进制，再取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_B	20_T	08_H	19_S	19_S	19_S	08_H	19_S
$f_2(key)$	09_{B+G}	04_{T+N}	17_{H+I}	28_{S+I}	28_{S+I}	26_{S+G}	22_{H+N}	03_{S+N}
$f_3(key)$	04_8	26_8	02_8	13_8	23_8	17_8	16_8	16_8

Hash查找的关键问题 { ①构造Hash函数 ②制订解决冲突的方法

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 H(key) 作为关键字为 key 的记录在表中。

哈希地址：由哈希函数得到的存储位置称为哈希地址。

散列（Hash）函数（表）分类 { 内散列表（数组） 外散列表（链表）

冲突（Collision）与同义词：具有不同关键字而有相同散列地址的元素称为同义词, 即key1≠key2, 但H(key1)=H(key2), 该现象称为冲突。

哈希表：根据设定的哈希函数 H(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”。

哈希（散列）技术仅仅是一种查找技术吗？

答：哈希既是一种查找技术，也是一种存储技术。

哈希是一种完整的存储结构吗？

答：哈希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，所以哈希主要是面向查找的存储结构。

哈希技术适用于何种场合？

答：通常用于实际出现的关键字的数目远小于关键字所有可能取值的数量。

构造散列函数（Hash）的几种方法：

- 直接定址法：Hash(*key*) = *key* ; 或 Hash(*key*) = a·*key*+b ;
- 质数除余法；
- 平方取中法；
- 折叠法；
- 数字分析法；
- 随机数法；

解决冲突的方法：
- 开放定址法
- 再散列法
- 链地址法
- 建立一个公共溢出区

➢直接定址法：**Hash(*key*) = *key* ; 或 Hash(*key*) = a·*key*+b ;**

【例5-18】： Hash(*key*) = *key*

地址	01	02	03	04	05	…	25	26	27	…	100
年龄	1	2	3	4	5	…	25	26	27	…	100
人数	3000	2000	5000			…	6000			…	…
:											

【例5-19】Hash(*key*) = *key* + （-1949）+ 1

地址	01	02	03	04	05	…	25	26	27	…	100
年份	1949	1950	1951			…	1973			…	
人数	3000	2000	5000			…	6000			…	…
:											

优点：没有冲突；　　缺点：关键字集合很大时，浪费存储空间严重。

- **质数除余法**

  设桶数B，取质数 m ≤ B

  Hash (*key*) = *key* % m

- **平方取中法**

  取 key^2 的中间的几位数作为散列地址

- **折叠法**

  图书编号: 0-442-20586-4

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

平方取中法

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

左：移位叠加

Hash(*key*) = 0088

右：间界叠加

Hash(*key*) = 6092

➢数字分析法

如右图所示
假设散列表长为100_{10}
可取中间4位中的两位为散例地址。

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

➢随机数法

应对长度不等的关键字。

构造Hash函数应注意以下几个问题：

- 计算Hash函数所需时间；
- 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；

解决冲突的几种方法：

➢开放定址法

$$H_i = (H(key) + d_i) \ MOD \ m \quad i = 1,2,\ldots,k \ (k \le m-2)$$

H(*key*) 为哈希函数，m为表长，d_i 为增量序列：

(1) $d_i = 1, 2, 3, \ldots, m-1$　　(线性探测再散列)

(2) $d_i = 1^2,-1^2,2^2,-2^2,\ldots,\pm k^2 \ (k \le m/2)$　(二次探测再散列 或 平方探测再散列)

(3) d_i 为伪随机序列　(伪随机探测再散列)

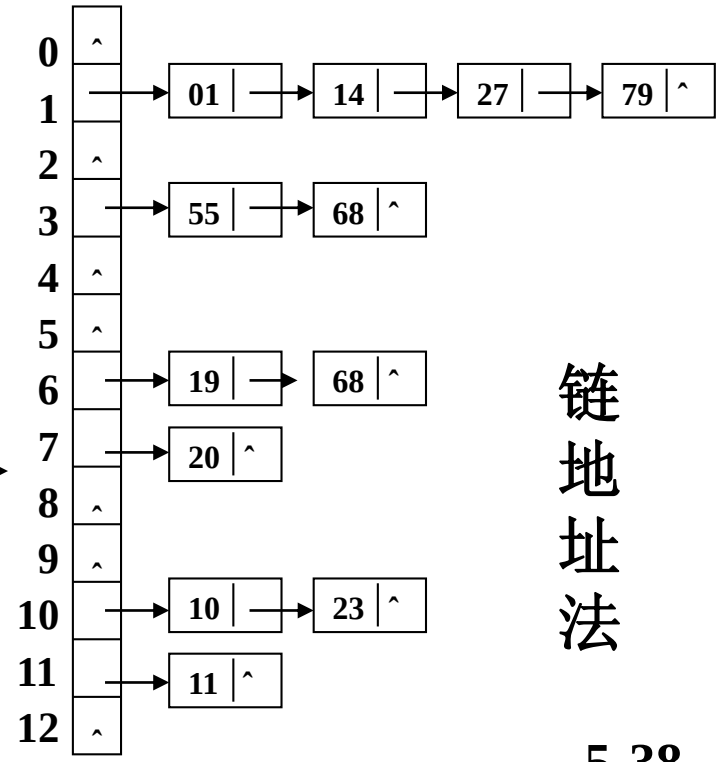
➢再散列法

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

➢链地址法（拉链法）

例：(19,14,23,01,68,20,84,27,55,11,10,79)

H(key) = key MOD 13



链地址法

➢建立一个公共溢出区

装填因子： $\alpha=\dfrac{\text{表中装入的记录数}}{\text{哈希表的长度}}$

装填因子α标志着哈希表的装满程度，α越小，发生冲突的可能性越小，反之，发生冲突的可能性越大。

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。

几种处理冲突方法的平均查找长度

处理冲突方法	查找成功	查找失败（插入记录）
线性探测法	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
二次探测法 再散列法	$-(\frac{1}{\alpha})\ln(1-\alpha)$	$\frac{1}{1-\alpha}$
链地址法	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$

【例5-20】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)，以下三种方案解决冲突问题：

(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), H(x)=1+(x%5) , …。

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5-21】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；

（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			

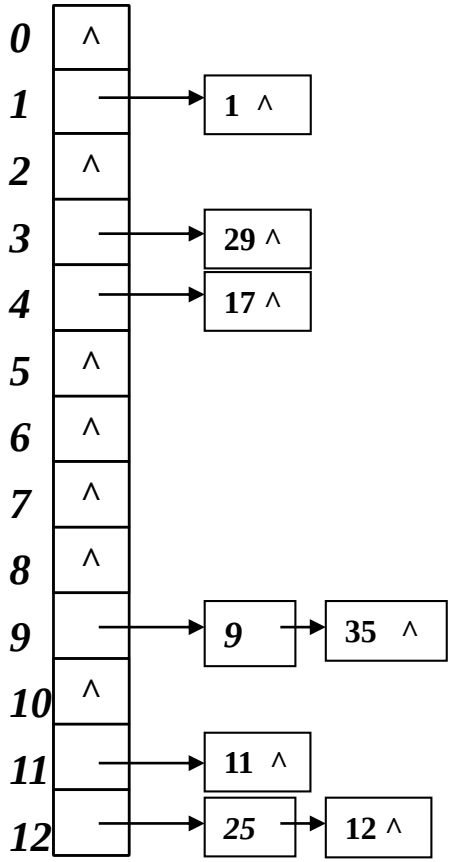
查找成功的平均查找长度：

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

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：

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

查找失败的平均查找长度：

$ASL_u=(4\times1+2\times2)/13\approx2.4$

【例5-22】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	

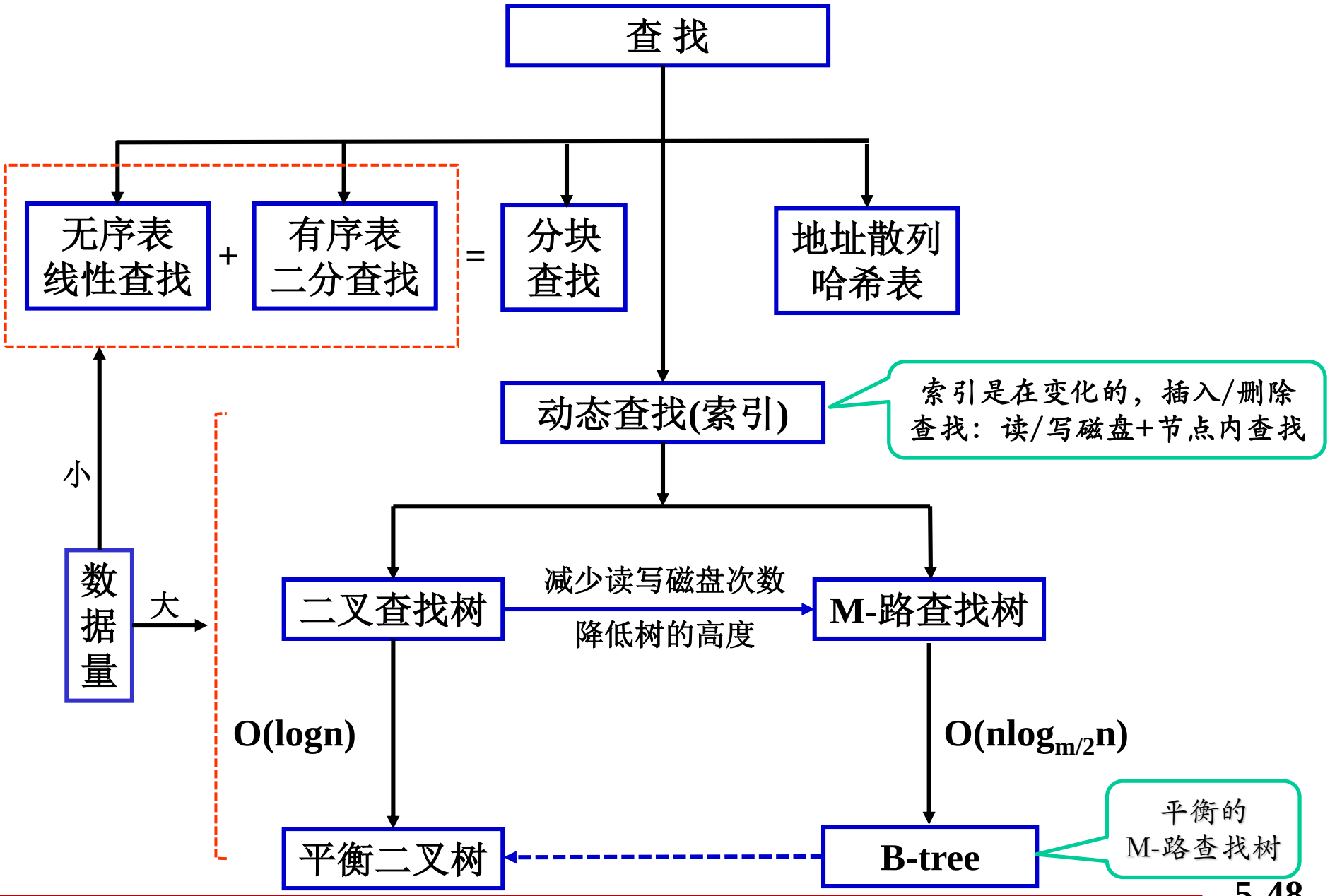
$ASL_{成功} = (1次\times4+2次\times1+3次\times2)/7 = 12/7$

查找不成功情况：根据哈希函数地址为MOD7，因此任何一个数经散列函数计算以后的初始地址只可能在0-6的位置

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

红-黑树（R-B Tree）

全称是Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树是一种近似平衡的二叉查找树，能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一陪。红黑树的特性:

（1）每个节点或者是黑色，或者是红色。
（2）根节点是黑色。
（3）每个叶子节点（NIL）是黑色。
（4）如果一个节点是红色的，则它的子节点必须是黑色的。
（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是**O(lgn)**。