第十章

- 静态索引结构
- 动态索引结构

静态索引结构

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

线性索引 (Linear Index List)

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

索引表

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

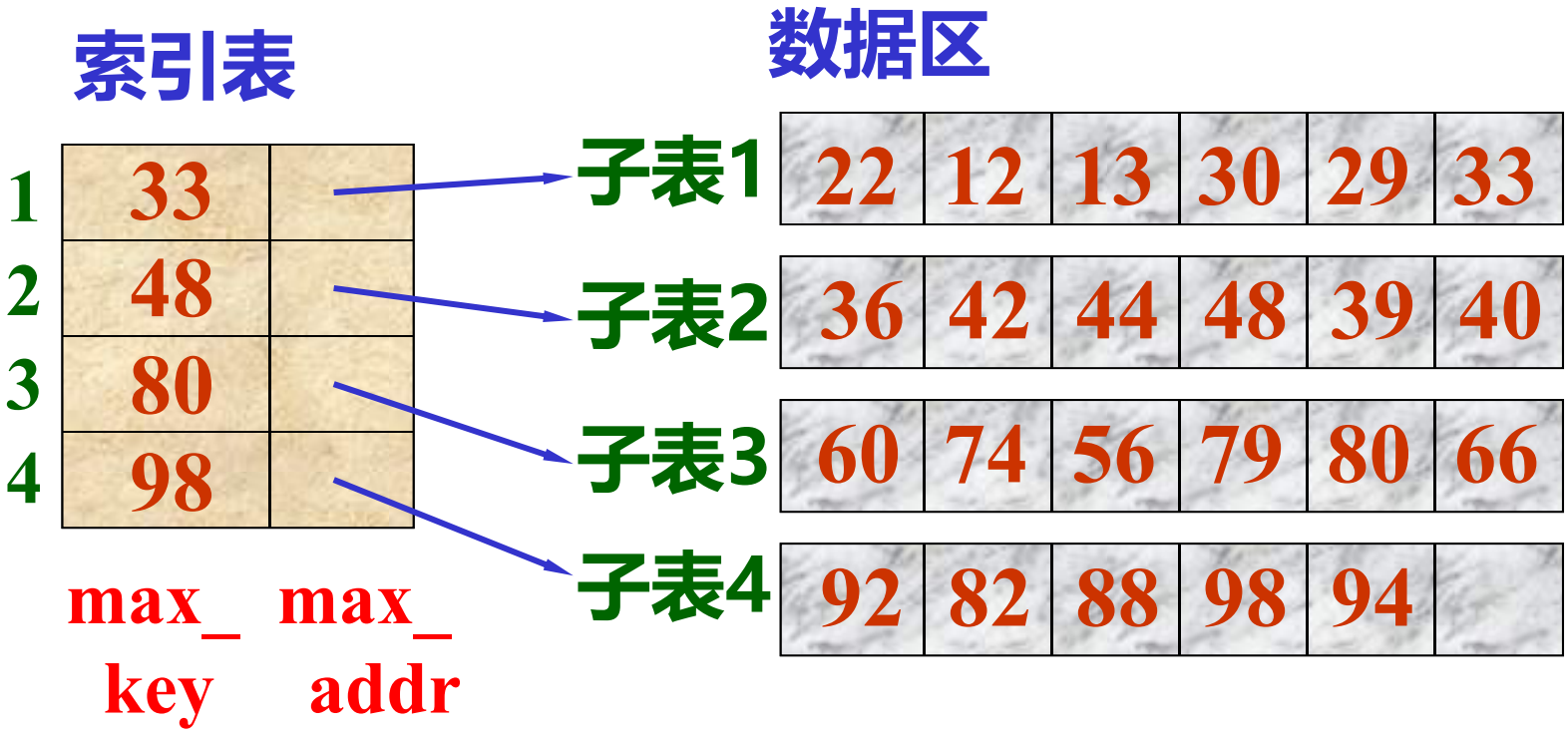
数据表

	职工号	姓名	性别	职务	婚否	
100	83	张珊	女	教师	已婚	…
140	08	李斯	男	教师	已婚	…
180	03	王鲁	男	教务员	已婚	…
220	95	刘琪	女	实验员	未婚	…
260	24	岳跋	男	教师	已婚	…
300	47	周斌	男	教师	已婚	…
340	17	胡江	男	实验员	未婚	…
380	51	林青	女	教师	未婚	…

- 假设内存工作区仅能容纳 64k 字节的数据，在某一时刻内存最多可容纳 64 个对象以供搜索。
- 如果对象总数有 14400 个，不可能把所有对象的数据一次都读入内存。无论是顺序搜索或折半搜索，都需要多次读取外存记录。
- 如果在索引表中每一个索引项占4个字节，每个索引项索引一个职工对象，则 14400 个索引项需要 56.25k 字节，在内存中可以容纳所有的索引项。

- 这样只需从外存中把索引表读入内存，经过搜索索引后确定了职工对象的存储地址，再经过 1 次读取对象操作就可以完成搜索。
- **稠密索引**：一个索引项对应数据表中一个对象的索引结构。当对象在外存中按加入顺序存放而不是按关键码有序存放时必须采用**稠密索引**结构，这时的索引结构叫做索引非顺序结构。
- **稀疏索引**：当对象在外存中有序存放时，可以把所有 n 个对象分为 b 个子表(块)存放，一个索引项对应数据表中一组对象(子表)。

- 在子表中，所有对象可能按关键码有序地存放，也可能无序地存放。但所有这些子表必须分块有序，后一个子表中所有对象的关键码均大于前一个子表中所有对象的关键码。它们都存放在数据区中。
- 另外建立一个索引表。索引表中每一表目叫做索引项，它记录了子表中最大关键码max _key以及该子表在数据区中的起始位置obj _ addr。
- 第 i 个索引项是第 i 个子表的索引项，$i = 0, 1, \ldots, n\text{-}1$。这种索引结构叫做索引顺序结构。


索引表
数据区
1
2
3
4
33
48
80
98
子表1
子表2
子表3
子表4
22 12 13 30 29 33
36 42 44 48 39 40
60 74 56 79 80 66
92 82 88 98 94
max_key
max_addr

- 对索引顺序结构进行搜索时，分为两级搜索:
  - 先在索引表 ID 中搜索给定值 K, 确定满足
    $$ID[i-1].max\_key < K \leq ID[i].max\_key$$
    的 i 值, 即待查对象可能在的子表的序号。
  - 然后再在第 i 个子表中按给定值搜索要求的对象。
- 索引表是按max_key有序的，且长度也不大，可以折半搜索，也可以顺序搜索。
- 各子表内各个对象如果也按对象关键码有序，可以采用折半搜索或顺序搜索；如果不是按对象关键码有序，只能顺序搜索。

- 索引顺序搜索的搜索成功时的平均搜索长度

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

- 其中，ASL_{Index} 是在索引表中搜索子表位置的平均搜索长度，$ASL_{SubList}$ 是在子表内搜索对象位置的搜索成功的平均搜索长度。
- 设把长度为 n 的表分成均等的 b 个子表，每个子表 s 个对象，则 $b = \lceil n/s \rceil$。又设表中每个对象的搜索概率相等，则每个子表的搜索概率为1/b，子表内各对象的搜索概率为 1/s。
- 若对索引表和子表都用顺序搜索，则索引顺序搜索的搜索成功时的平均搜索长度为

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

- 索引顺序搜索的平均搜索长度与表中的对象个数 n 有关，与每个子表中的对象个数 s 有关。在给定 n 的情况下，s 应选择多大?
- 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对象所在的子表，则搜索成功时的平均搜索长度为

$$ASL_{IndexSeq} = ASL_{Index} + ASL_{SubList}$$
$$\approx \log_2 (b+1)-1 + (s+1)/2$$
$$\approx \log_2(1+n/s) + s/2$$

倒排表 (Inverted Index List)

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

对象关键码 *key*	对象存放地址 *addr*

- 但在实际应用中有时需要针对其它属性进行搜索。例如，查询如下的职工信息：

  (1) 列出所有教师的名单；

  (2) 已婚的女性职工有哪些人？

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

- 次索引的索引项由次关键码、链表长度和链表本身等三部分组成。
- 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

次关键码	男	女
计　数	5	3
地址指针		

指针	03	08	17	24	47	51	83	95

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已婚的职工对象，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对象。

- 在次索引中**记录对象存放位置的指针**可以**用主关键码表示**：可通过搜索次索引确定该对象的主关键码，再通过搜索主索引确定对象的存放地址。
- 在倒排表中各个属性链表的长度大小不一,管理比较困难。为此引入单元式倒排表。
- 在单元式倒排表中，索引项中不存放**对象的存储地址**, 存放**该对象所在硬件区域的标识**。
- 硬件区域可以是磁盘柱面、磁道或一个页块，以**一次 I / O 操作能存取的存储空间作为硬件区域**为最好。

- 为使索引空间最小，在索引中标识这个硬件区域时可以使用一个能转换成地址的二进制数，整个次索引形成一个(二进制数的) **位矩阵**。
- 例如，对于记录学生信息的文件，次索引可以是如图所示的结构。二进位的值为 1 的硬件区域包含具有该次关键码的对象。
- 如果在硬件区域1，……中有要求的对象。然后将硬件区域1，……等读入内存，在其中进行检索，确定就可取出所需对象。

		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	…
	……											

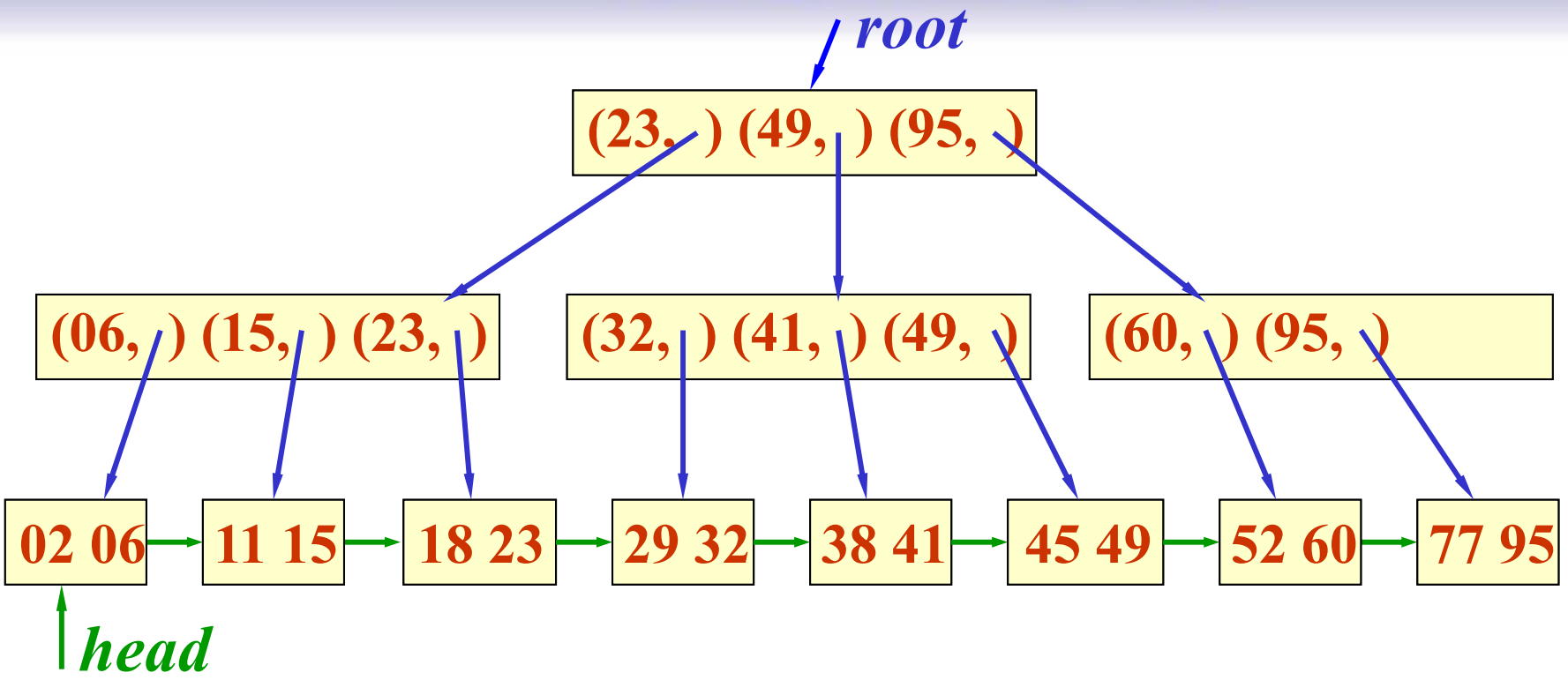
单元式倒排表结构

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

	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

m 路静态搜索树

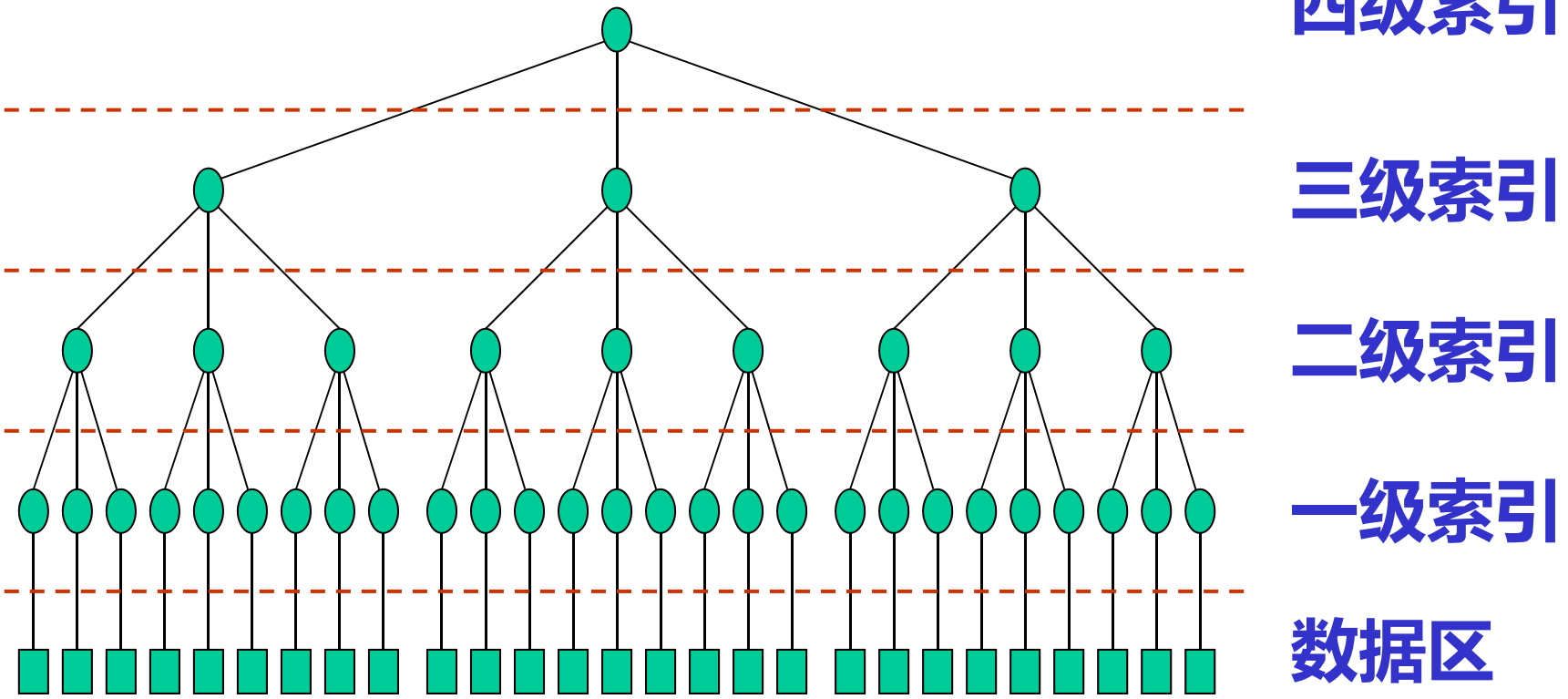
- 当数据对象数目特别大，索引表本身也很大，在内存中放不下，需要分批多次读取外存才能把索引表搜索一遍。
- 此时, 可以建立索引的索引(二级索引)。二级索引可以常驻内存，二级索引中一个索引项对应一个索引块，登记该索引块的最大关键码及该索引块的存储地址。



- 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次读取对象。

- 必要时, 还可以有4级索引, 5极索引, ….
- 这种多级索引结构形成一种 m 叉树。树中每一个分支结点表示一个索引块，它最多存放 m 个索引项，每个索引项分别给出各子树结点(低一级索引块) 的最大关键码和结点地址。
- 树的叶结点中各索引项给出在数据表中存放的对象的关键码和存放地址。这种m叉树用来作为多级索引，就是m路搜索树。
- m路搜索树可能是静态索引结构，即结构在初始创建，数据装入时就已经定型，在整个运行期间，树的结构不发生变化。

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



多级索引结构形成 *m* 路搜索树

动态索引结构

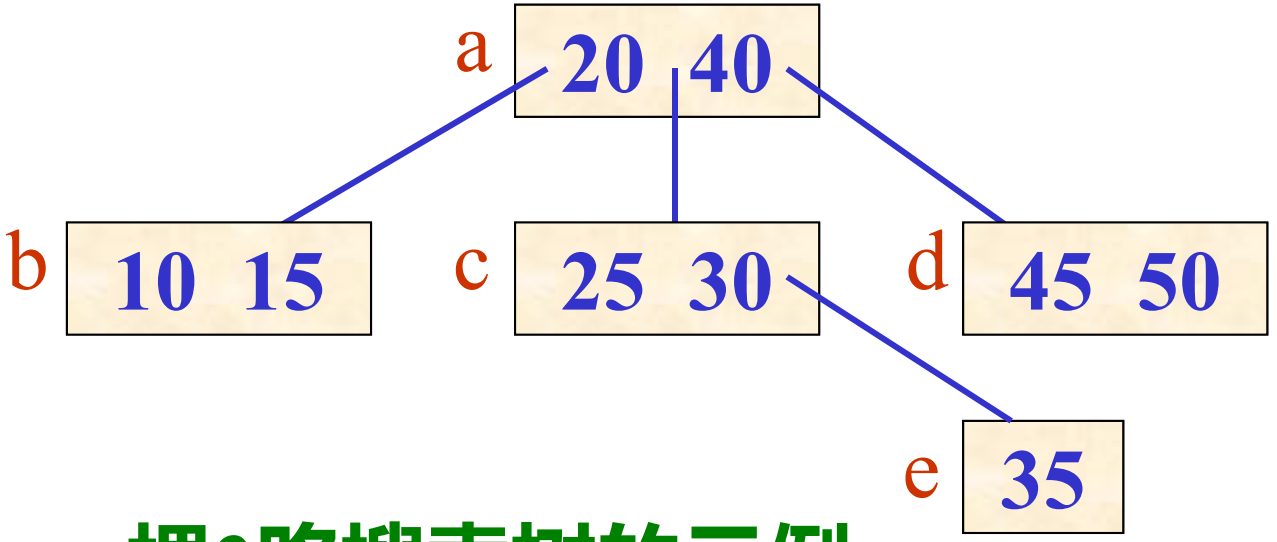
动态的 m 路搜索树

- 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，$0 \le i \le n$。



一棵3路搜索树的示例

m 路搜索树的类定义

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

AVL树是2路搜索树。如果已知 *m* 路搜索树的度 *m* 和它的高度 *h*, 则树中的最大结点数为

(等比级数前 *h* 项求和) $\sum_{i=0}^{h} m^i = \frac{1}{m-1}\left(m^{h+1}-1\right)$

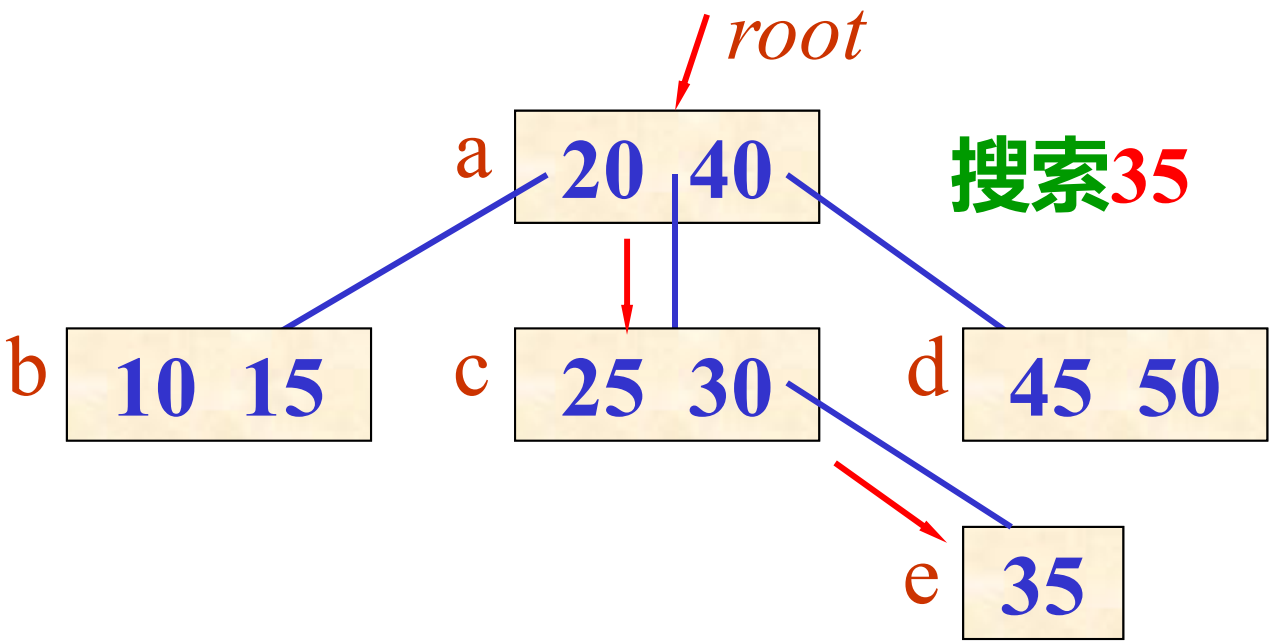
- 每个结点中**最多有 m-1 个关键码**，在一棵高度为 h 的 m 路搜索树中关键码的最大个数为 $m^{h+1}-1$。
  - 高度 h=2 的二叉树, 关键码最大个数为7；
  - 高度 h=3 的3路搜索树，关键码最大个数为 $3^4-1=80$。

标识 m 路搜索树搜索结果的三元组表示

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

m 路搜索树上的搜索算法

- 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 Type> Triple<Type> &
Mtree<Type> :: Search ( const Type& x ) {
    Triple result;                  //记录搜索结果三元组
    GetNode ( root );               //读入根结点
    Mnode <Type> *p = root, *q = NULL;
    while ( p != NULL ) {           //从根开始检测
        int i = 0;  p->key[(p->n)+1] = MAXKEY;
        while ( p->key[i+1] < x ) i++; //结点内搜索
        if ( p->key[i+1] == x ) {          //搜索成功
            result.r = p;  result.i =  i+1;  result.tag = 0;
            return result;
        }
```

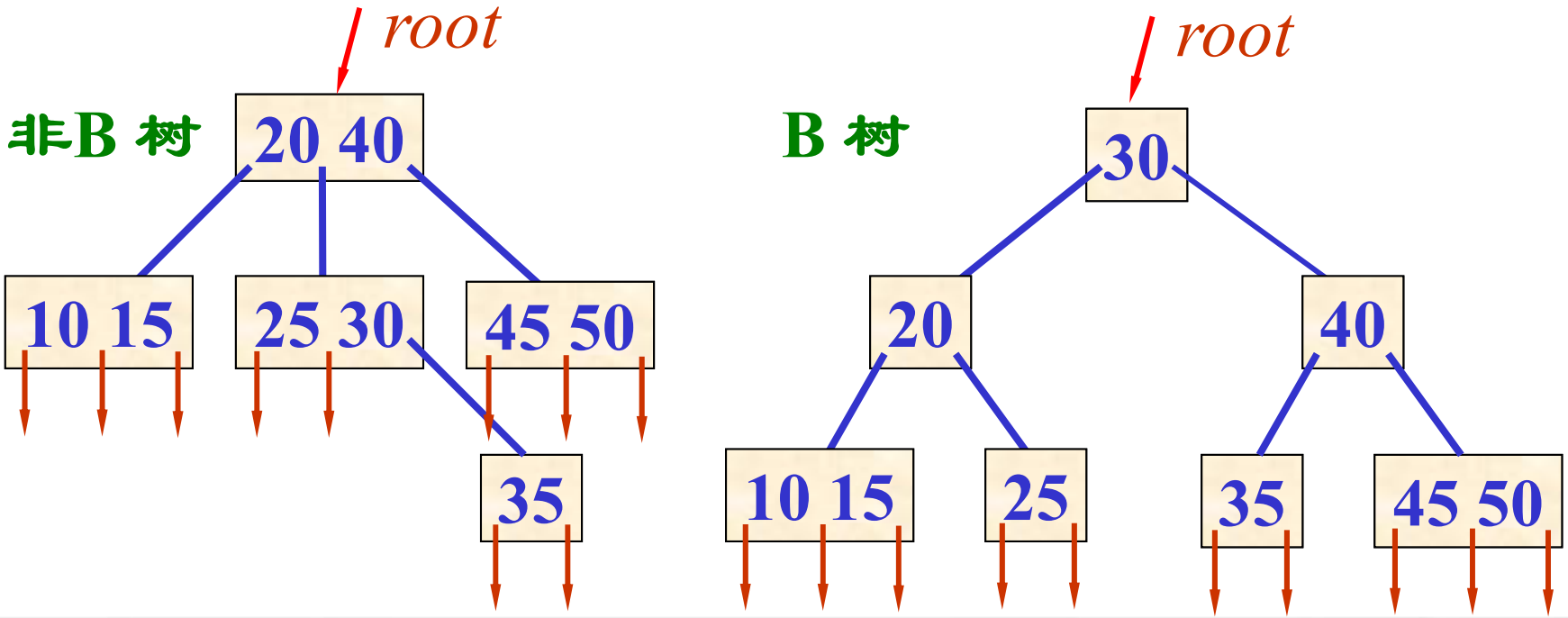
```
            q = p;   p = p->ptr[i]; //向下一层结点搜索
            GetNode (p);              //读入该结点
        }
        result.r = q;  result.i = i;  result.tag = 1;
        return result;          //搜索失败, 返回插入位置
}
```

- **提高搜索树的路数 m, 可以改善树的搜索性能。对于给定的关键码数 n，如果搜索树是平衡的，可以使 m 路搜索树的性能接近最佳。下面将讨论一种称之为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”结点是当搜索值x不在树中时才能到达的结点。
- 事实上，在B 树的每个结点中还包含有一组指针$D[m]$，指向实际对象的存放地址。

- $K[i]$与$D[i]$ $(1 \le i \le n < m)$ 形成一个索引项$(K[i], D[i])$，通过$K[i]$可找到某个对象的存储地址$D[i]$。
- 一棵B 树是平衡的 m 路搜索树，但一棵平衡的 m 路搜索树不一定是B 树。



B 树的类定义和B 树结点类定义

```
template <class Type> class Btree :
        public Mtree<Type> {
//继承 m 路搜索树的所有属性和操作
public:
    int Insert ( const Type& x );
    int Remove ( const Type& x );
};

template <class Type> class Mnode {
// B 树结点类定义
private:
```

```
    int n;                              //结点中关键码个数
    Mnode<Type> *parent;  //双亲指针
    Type key[m+1];                  //关键码数组 1~m-1
    Mnode<Type> *ptr[m+1];  //子树指针数组
};
```

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。

- B 树的搜索时间与B 树的阶数 m 和B 树的高度h直接有关, 必须加以权衡。
- 在B 树上进行搜索, 搜索成功所需的时间取决于关键码所在的层次; 搜索不成功所需的时间取决于树的高度。
- 如果定义B 树的高度h 为失败结点所在的层次，需要了解树的高度h 与树中的关键码个数 N 之间的关系。
- 如果让B 树每层结点个数达到最大，且设关键码总数为N, 则树的高度达到最小:

$$h = \lceil \log_m(N+1) \rceil - 1$$

高度h与关键码个数N之间的关系

- 设在 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:
  - 0层　1 个结点
  - 1层　至少 2 个结点
  - 2层　至少 $2\lceil m/2 \rceil$ 个结点
  - 3层　至少 $2\lceil m/2 \rceil^2$ 个结点
  - 如此类推，……
  - h-1 层 至少有 $2\lceil m/2 \rceil^{h-2}$ 个结点。所有这些结点都不是失败结点。

- 若树中关键码有N个, 则失败结点数为N+1。这是因为失败数据一般与已有关键码交错排列。因此，有

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

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

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

- 所有的非失败结点所在层次为$0 \sim h-1$。
- 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:

- 这个时间由两部分组成：
  - 从磁盘中读入结点所用时间
  - 在结点中搜索 x 所用时间
- 根据定义，B 树的每个结点的大小都是固定的，结点内有 m-1 个索引项 (K_i, D_i, P_i), $1 \le i < m$。其中，K_i 所占字节数为 α，D_i 和 P_i 所占字节数为 β，则结点大小近似为 $m(\alpha+2\beta)$ 个字节。读入一个结点所用时间为：

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

B 树的插入

- B 树是从空树起, 逐个插入关键码而生成的。
- 在B 树，每个非失败结点的关键码个数都在

$$[\lceil m/2 \rceil - 1, m-1]$$

之间。

- 插入在某个叶结点开始。如果在关键码插入后结点中的关键码个数超出了上界 $m-1$，则结点需要“分裂”，否则可以直接插入。
- 实现结点“分裂”的原则是：
  - 设结点 p 中已经有 $m-1$ 个关键码，当再插入一个关键码后结点中的状态为

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

其中 $K_i < K_{i+1}$, $1 \le i < m$

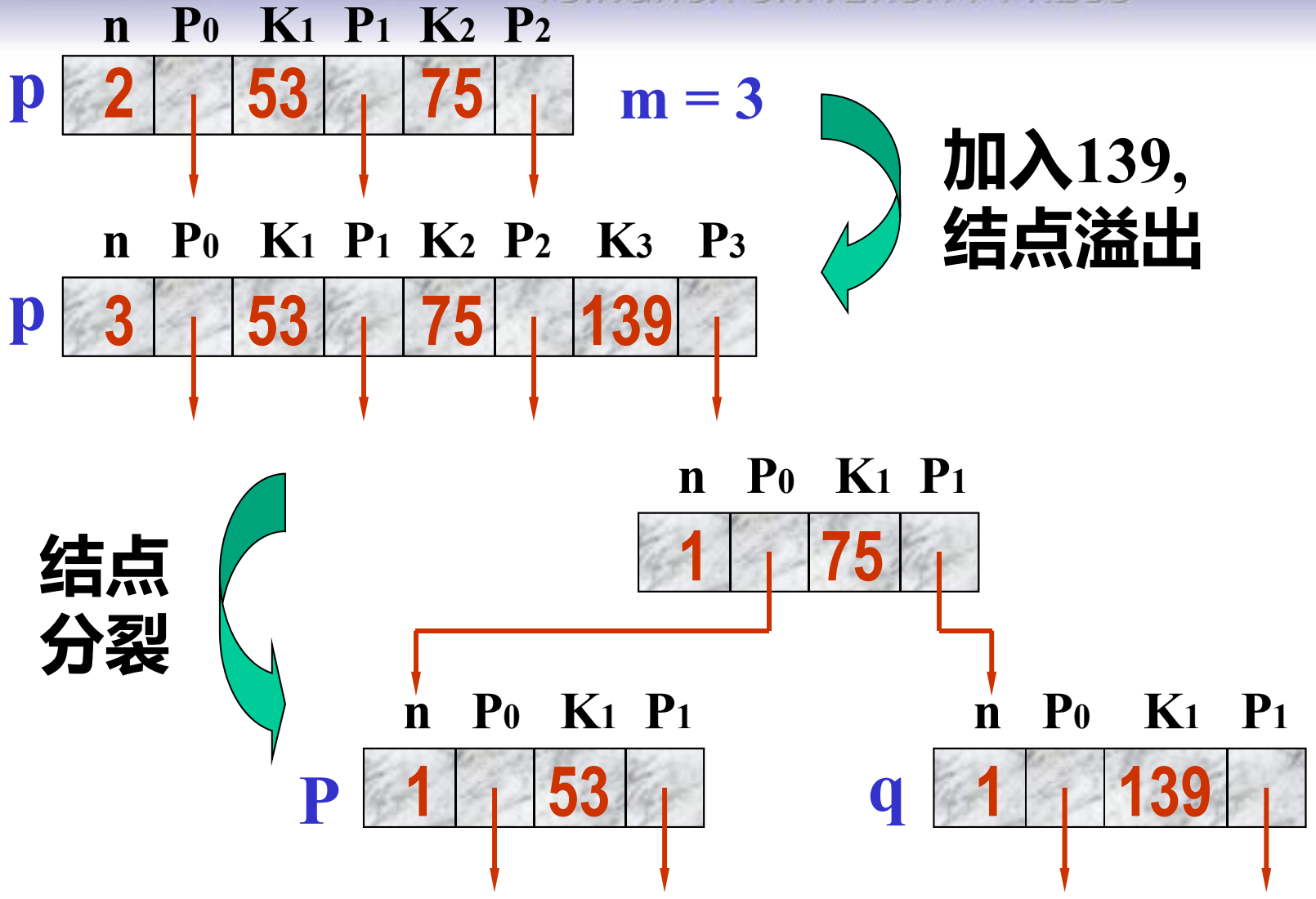
- 这时必须把结点 p 分裂成两个结点 p 和 q，它们包含的信息分别为：
  - 结点 p：

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

  - 结点 q：

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

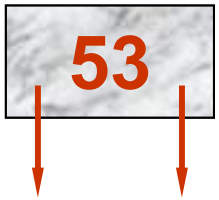
- 位于中间的关键码 $K_{\lceil m/2 \rceil}$ 与指向新结点 q 的指针形成一个二元组 $(K_{\lceil m/2 \rceil}, q)$，插入到这两个结点的双亲结点中去。


n P0 K1 P1 K2 P2
p 2 53 75
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

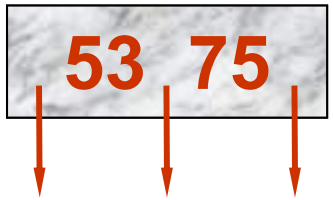
结点“分裂”的示例

示例:从空树开始加入关键码建立3阶B 树

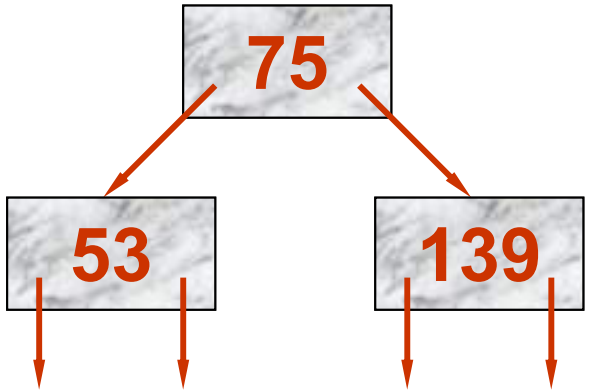
n=1 加入53



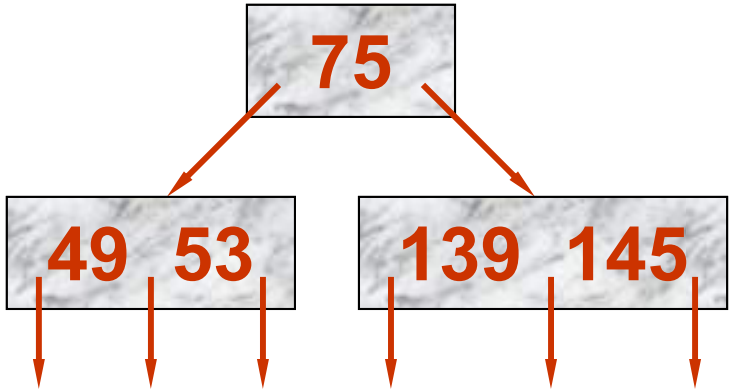
n=2 加入75



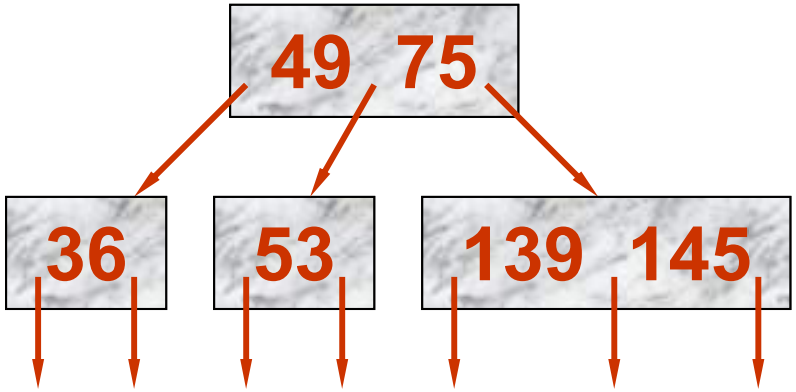
n=3 加入139



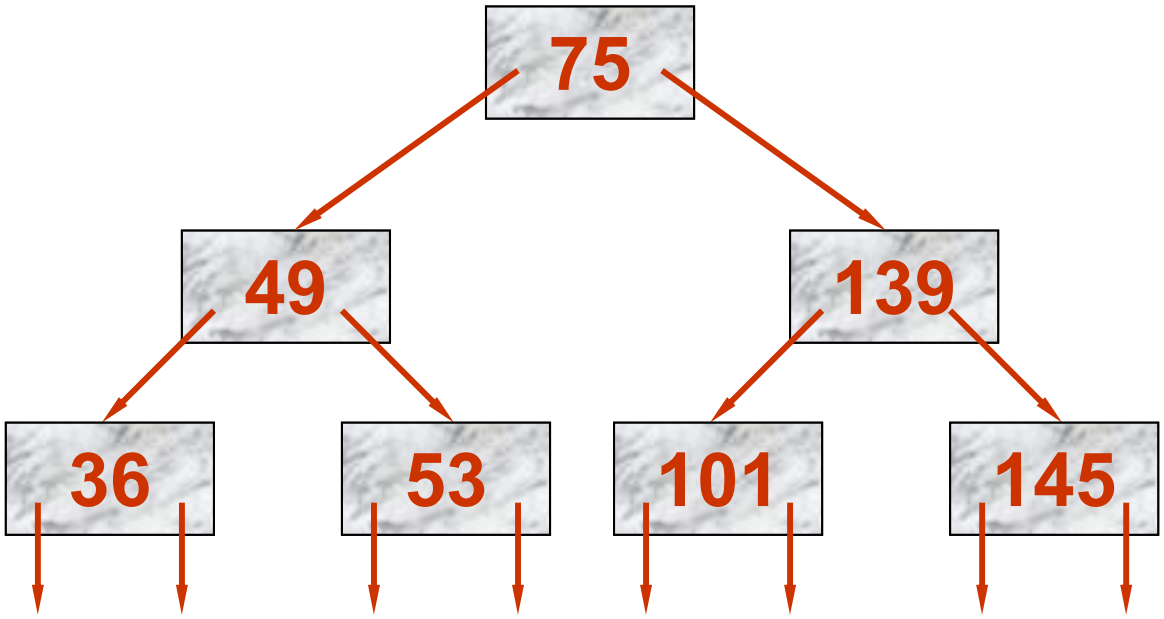
n=5 加入49,145



n=6 加入36



n=7 加入101



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

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

B 树的插入算法

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

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

```
            p = t;
        }    //继续 while(1) 循环,向上调整
        else {                                          //双亲是根
            root = new Mnode<Type>;  //创建新根
            root->n = 1;  root->parent = NULL;
            root->key[1] = K;  root->ptr[0] = p;
            root->ptr[1] = ap;
            q->parent = p->parent = root;
            PutNode (p);  PutNode (q);
            PutNode (root);   return 1;  //跳出返回
        }
    }
}
```

- 当分裂一个非根的结点时需要向磁盘写出 2 个结点, 当分裂根结点时需要写出 3 个结点。
- 如果我们所用的内存工作区足够大, 使得在向下搜索时, 读入的结点在插入后向上分裂时不必再从磁盘读入，那么，在完成一次插入操作时需要读/写磁盘的最大次数 =

  = 找插入结点向下读盘次数 +

  + 分裂非根结点时写盘次数 +

  + 分裂根结点时写盘次数 =

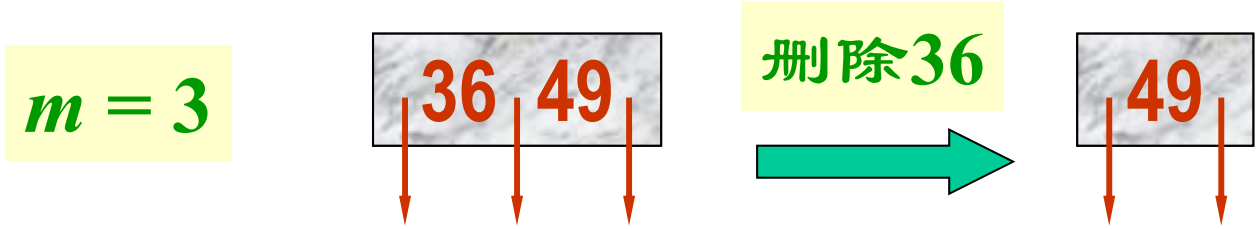
  = $h+2(h-1)+3$ =

  = $3h+1$。

B 树的删除

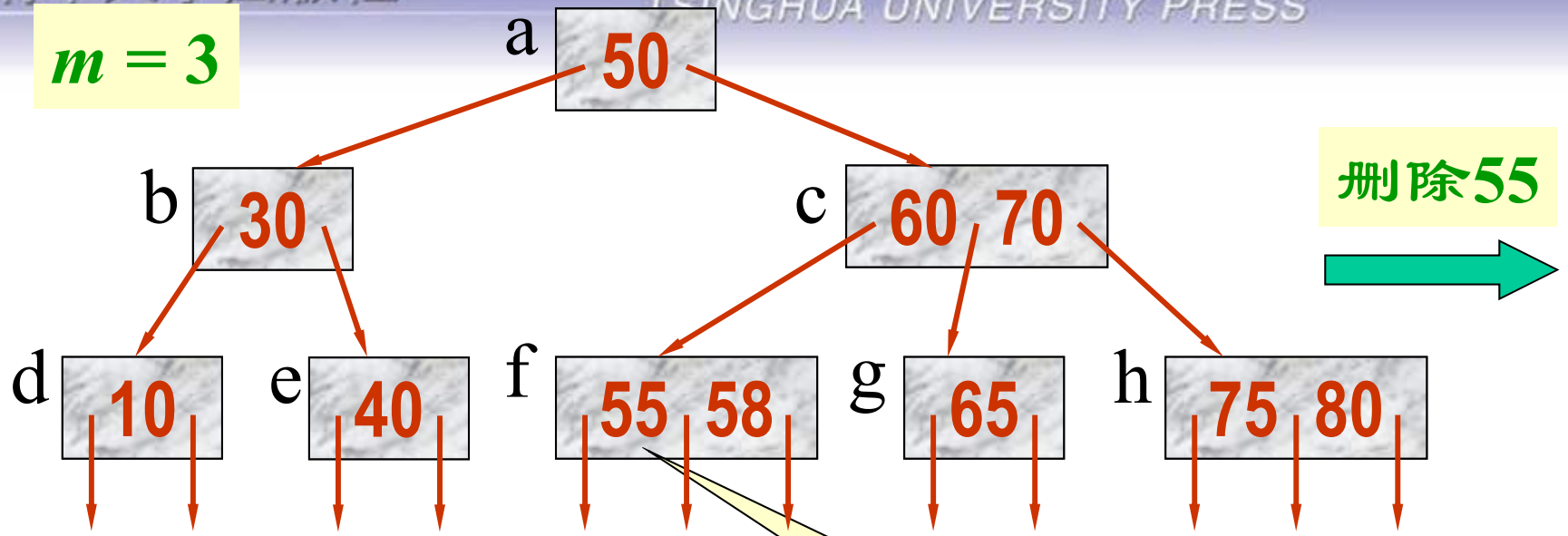
- 在B 树上删除一个关键码时,
  - 首先需要找到这个关键码所在的结点，从中删去这个关键码。
  - 若该结点不是叶结点, 且被删关键码为 K_i, $1 \le i \le n$, 则在删去该关键码之后, 应以该结点 P_i 所指示子树中的最小关键码 x 来代替被删关键码 K_i 所在的位置;
  - 然后在 x 所在的叶结点中删除 x。
- 在叶结点上的删除有 4 种情况。

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

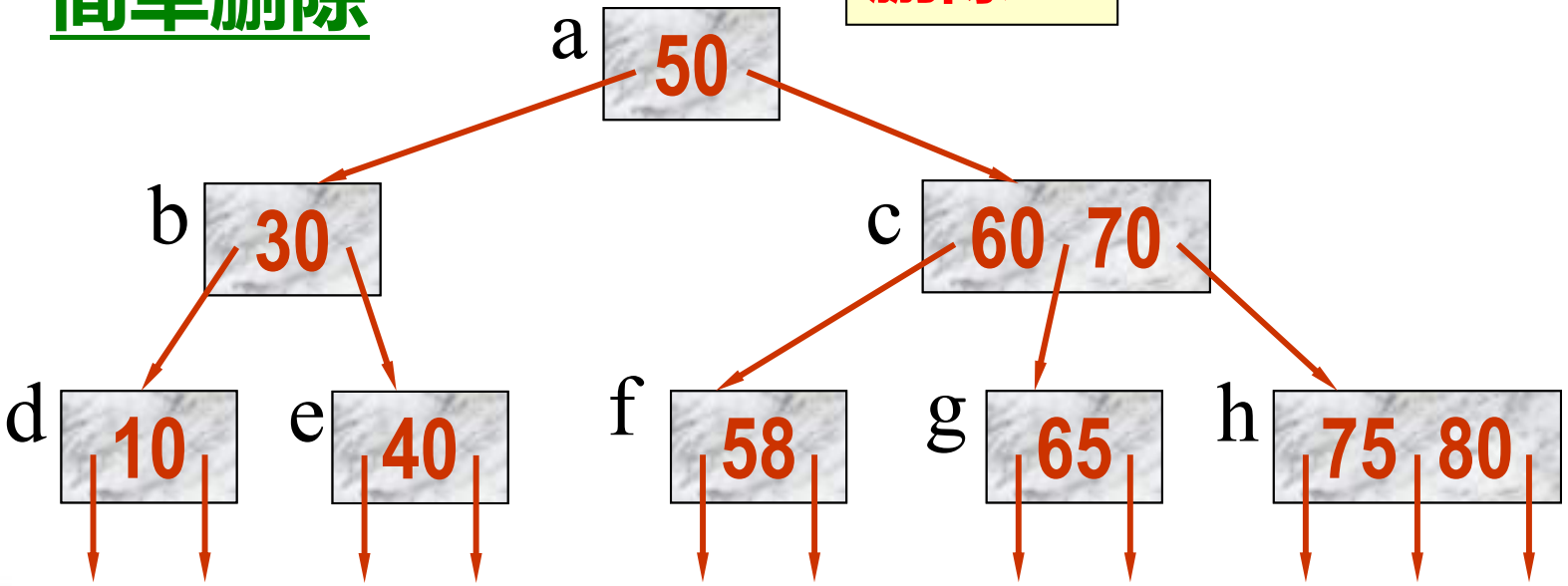


❷ 被删关键码所在叶结点不是根结点且删除前该结点中关键码个数 $n \geq \lceil m/2 \rceil$，则直接删去该关键码并将修改后的结点写回磁盘，删除结束。

$m = 3$


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

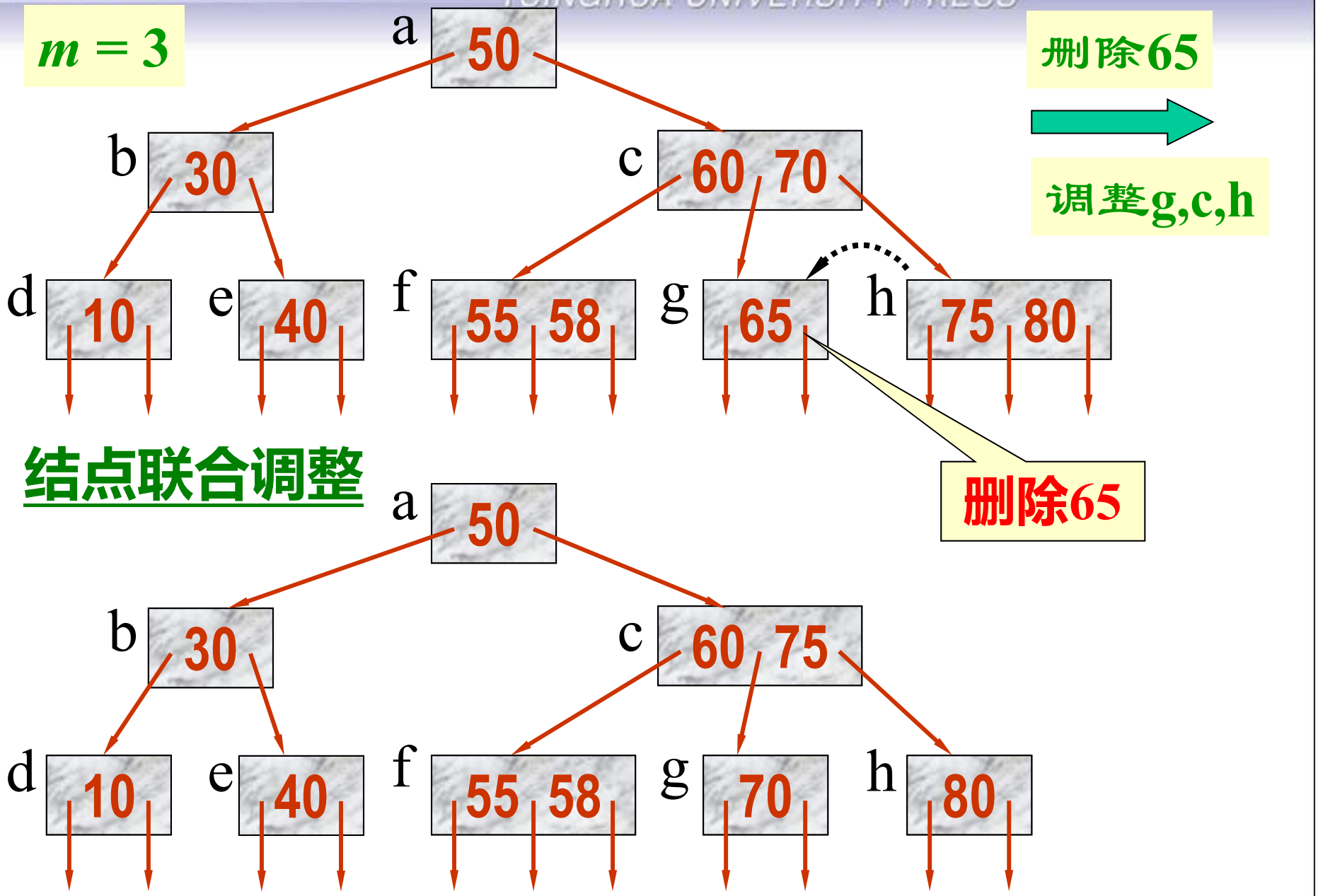
简单删除


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

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

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

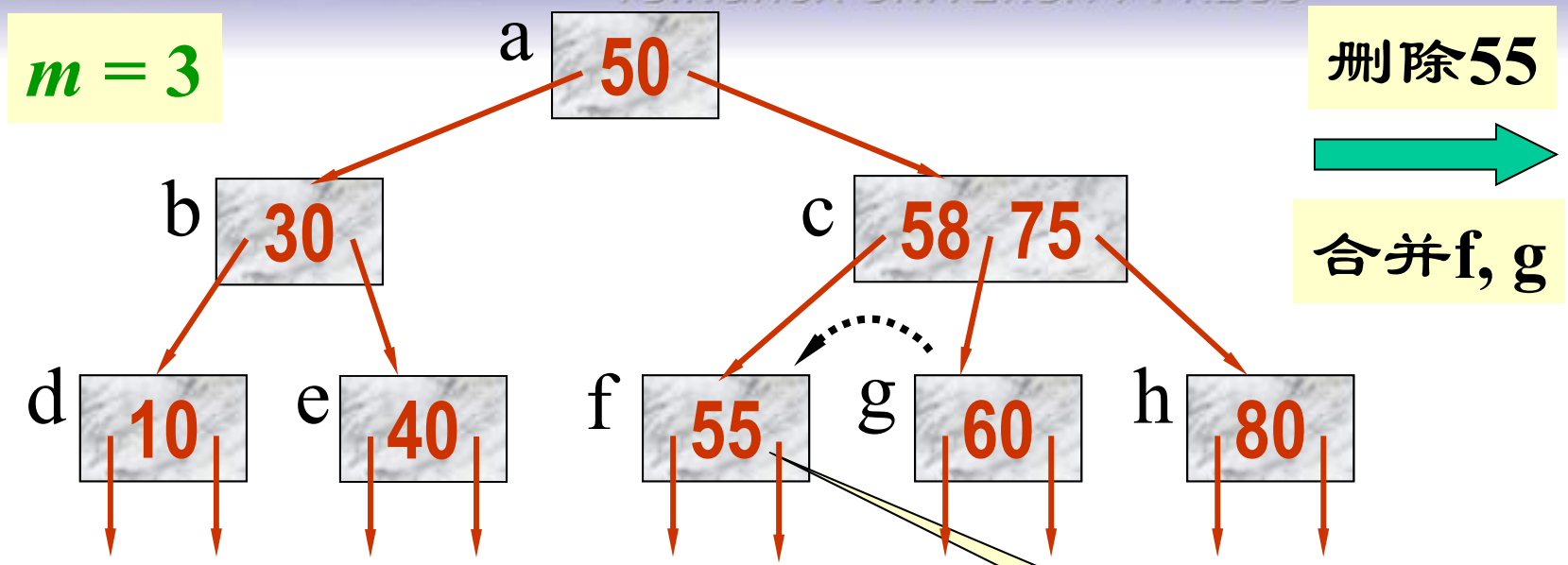
❹ 被删关键码所在叶结点删除前关键码个数 n =⌈m/2⌉-1, 若这时与该结点相邻的右兄弟 (或左兄弟) 结点的关键码个数 n =⌈m/2⌉-1, 则必须按以下步骤合并这两个结点。


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

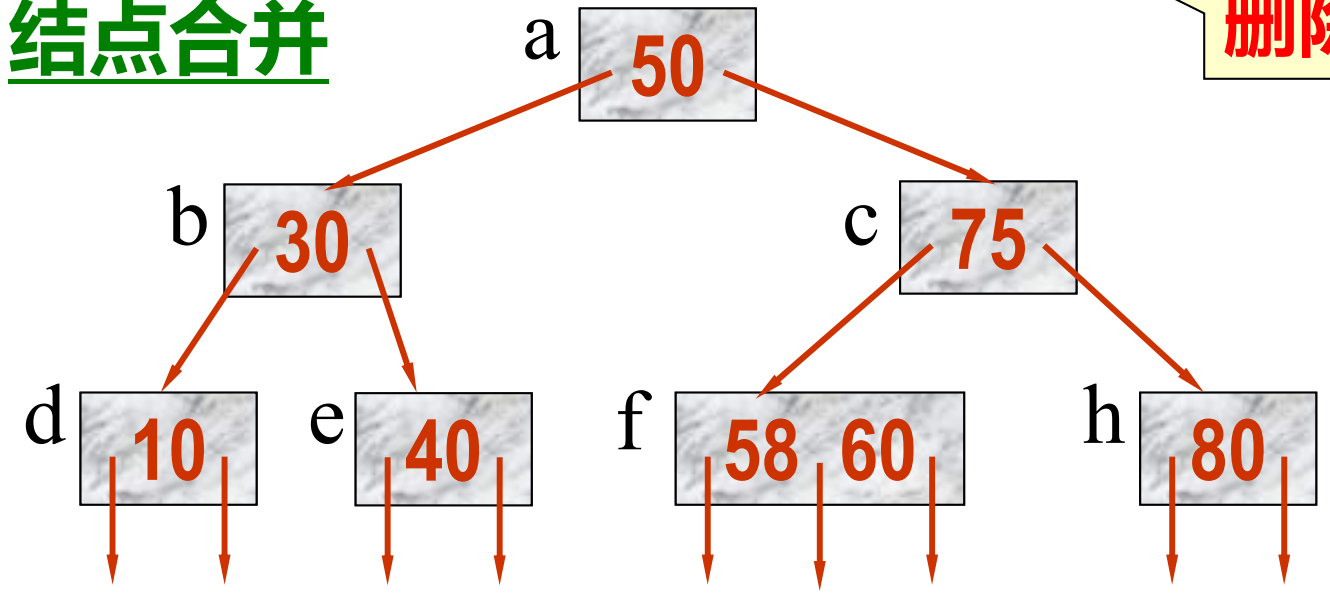
- 将双亲结点 p 中相应关键码下移到选定保留的结点中。
- 若要合并 p 中的子树指针 P_i 与 P_{i+1} 所指的结点, 且保留 P_i 所指结点, 则把 p 中的关键码 K_{i+1} 下移到 P_i 所指的结点中。
- 把 p 中子树指针 P_{i+1} 所指结点中的全部指针和关键码都照搬到 P_i 所指结点的后面。删去 P_{i+1} 所指的结点。
- 在结点 p 中用后面剩余的关键码和指针填补关键码 K_{i+1} 和指针 P_{i+1}。
- 修改结点 p 和选定保留结点的关键码个数。

- 在合并结点的过程中, 双亲结点中的关键码个数减少了。
- 若双亲结点是根结点且结点关键码个数减到0, 则将该双亲结点删去, 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⌈m/2⌉-2，又要与它自己的兄弟结点合并, 重复上面的合并步骤。最坏情况下这种结点合并处理要自下向上直到根结点。

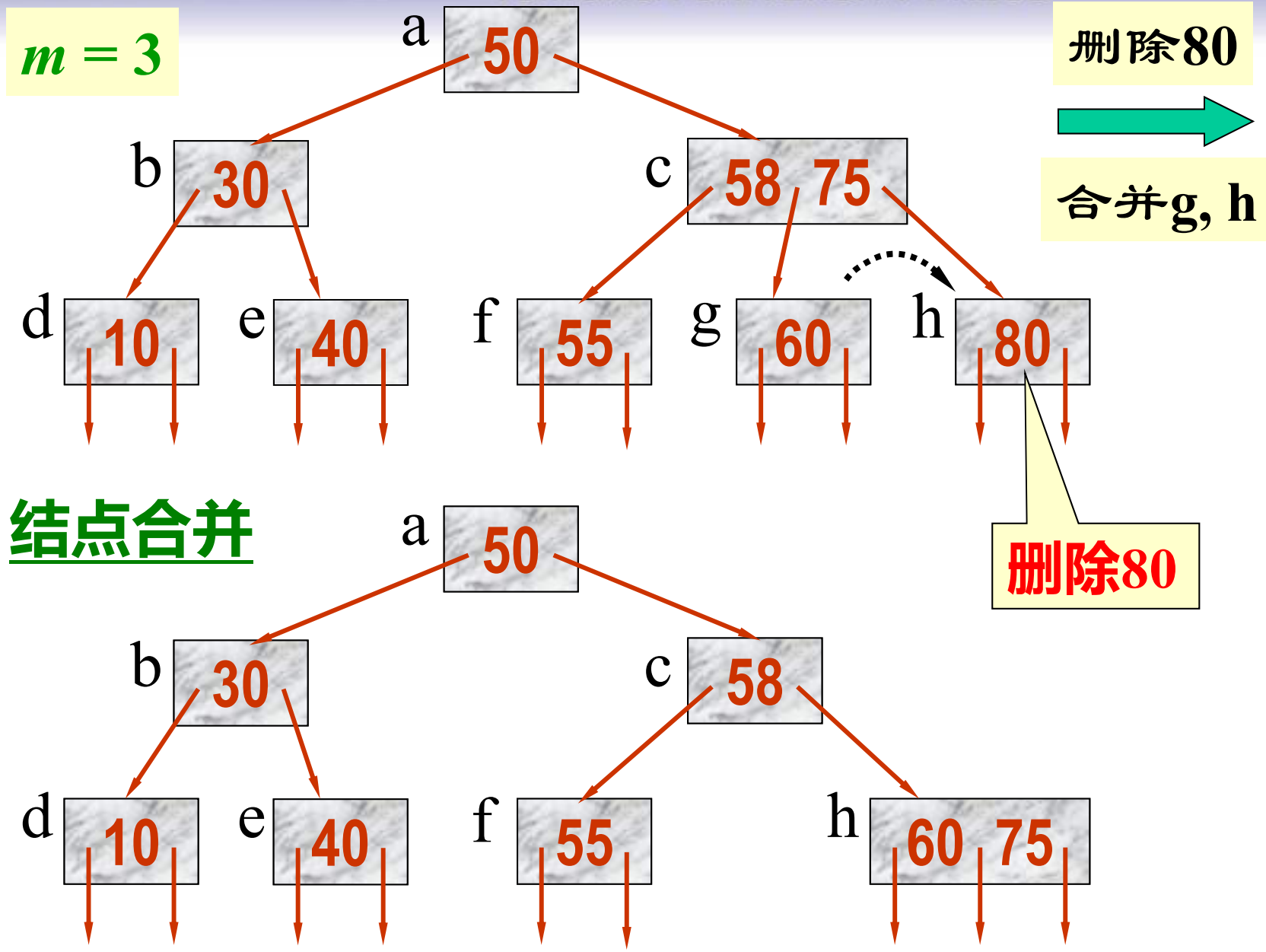
$m = 3$

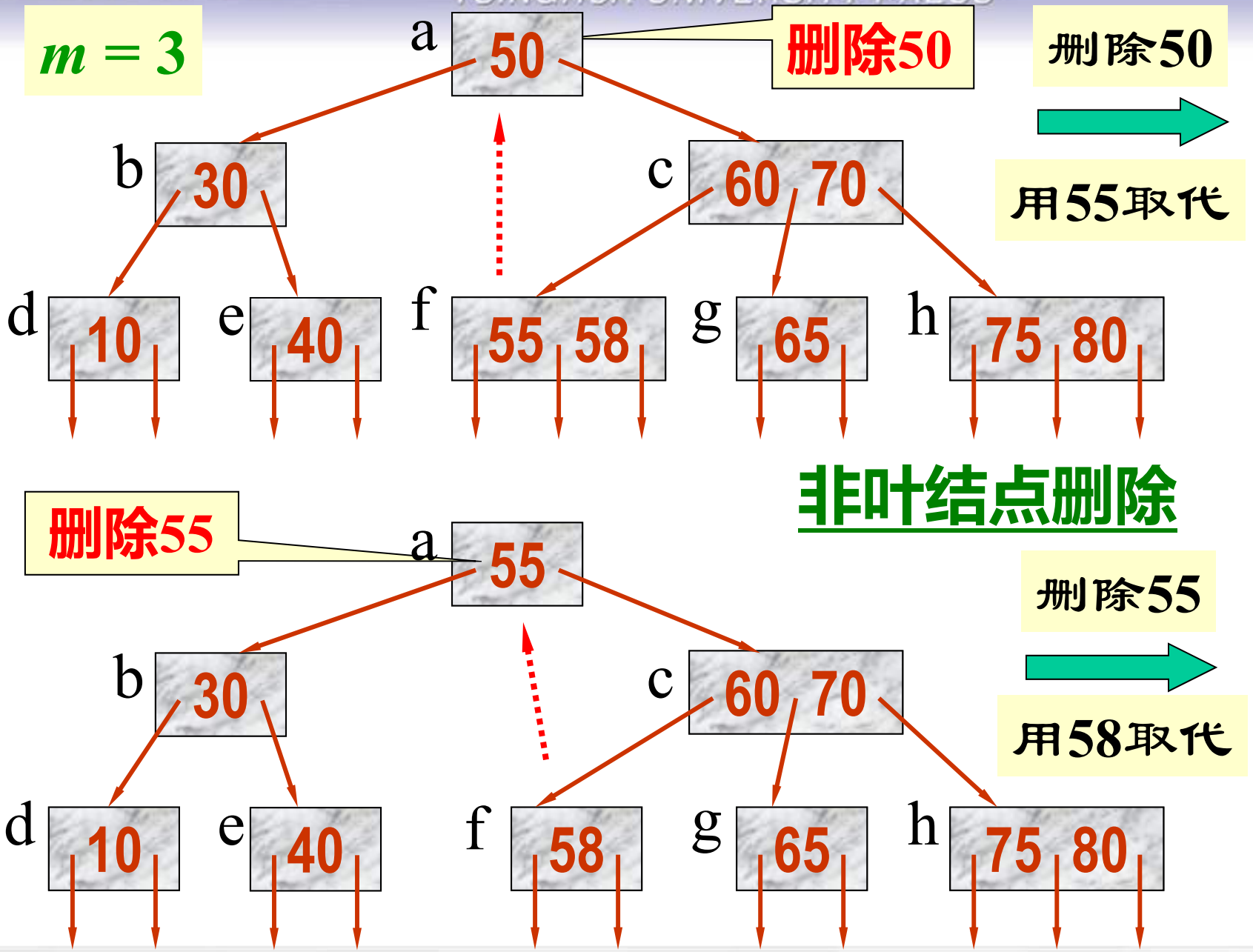

a
50
b
30
c
58
75
d
10
e
40
f
55
g
60
h
80
删除55
合并f, g
删除55

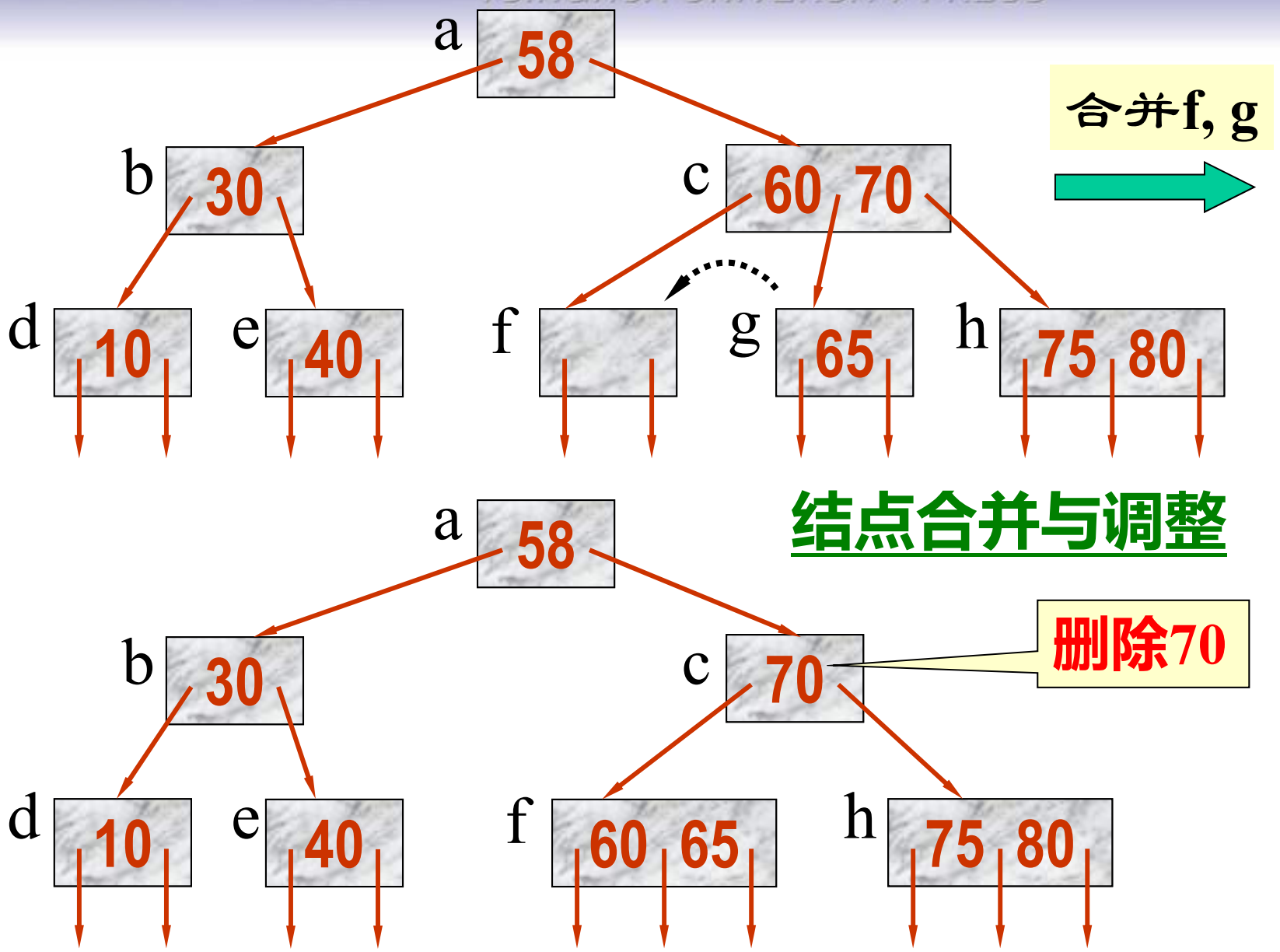
结点合并


a
50
b
30
c
75
d
10
e
40
f
58
60
h
80

$m = 3$

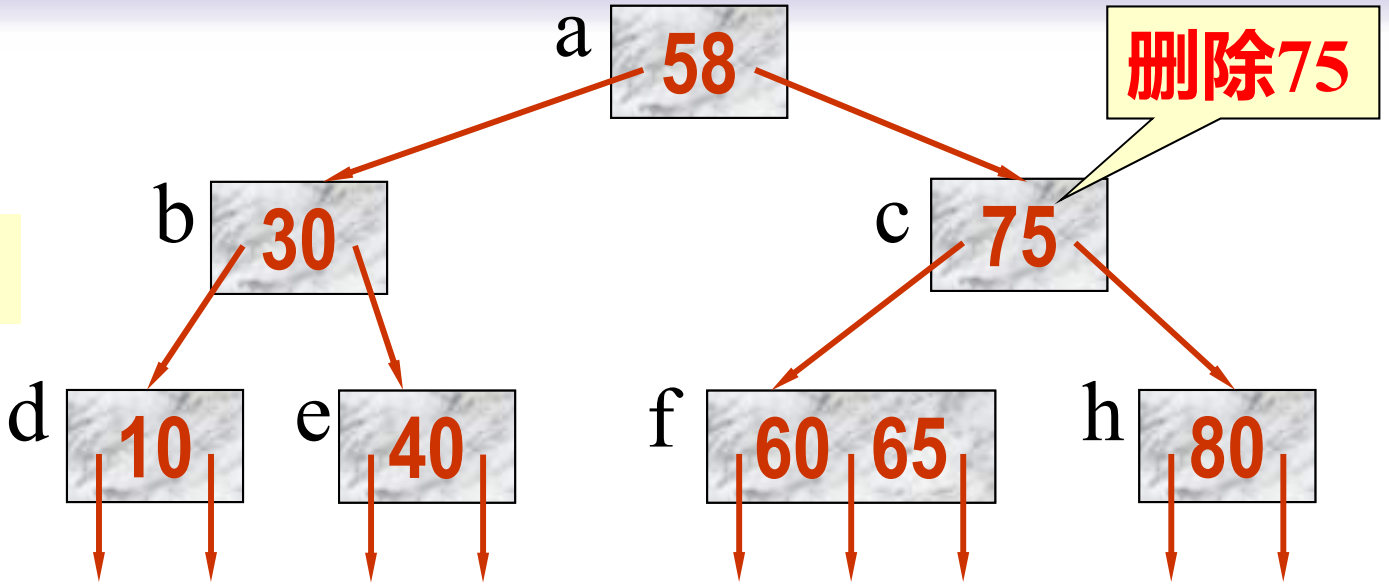

a
50
b
30
c
58
75
d
10
e
40
f
55
g
60
h
80
删除80
合并g, h
结点合并
a
50
b
30
c
58
d
10
e
40
f
55
h
60
75
删除80


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

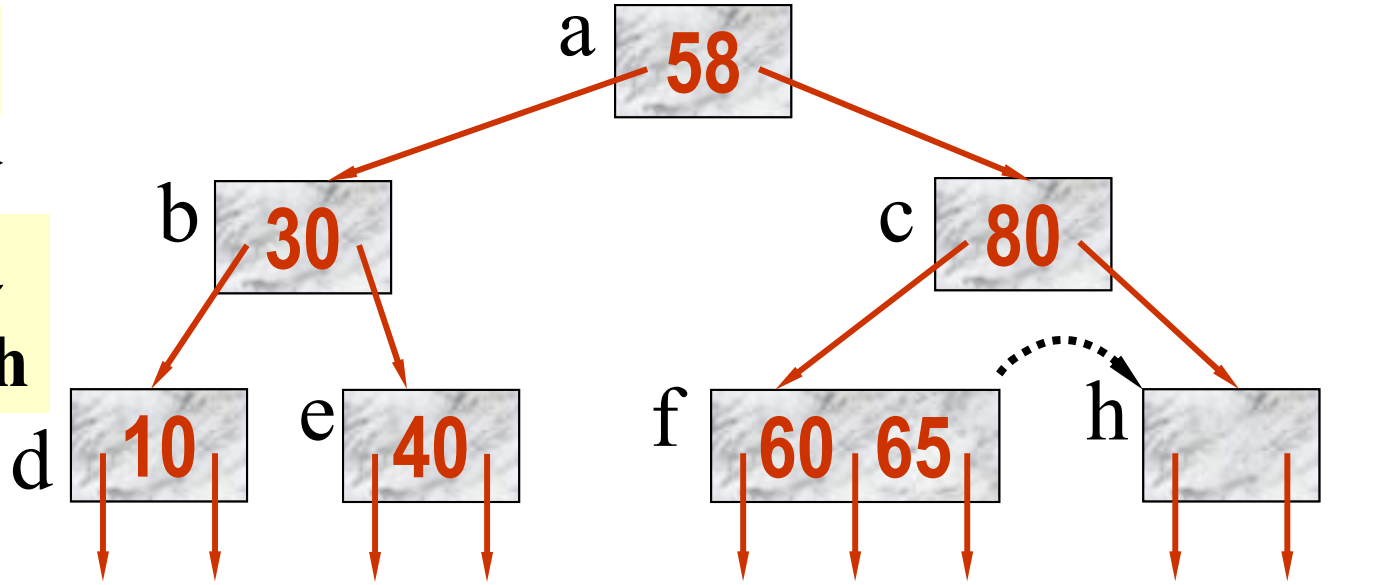
用75取代

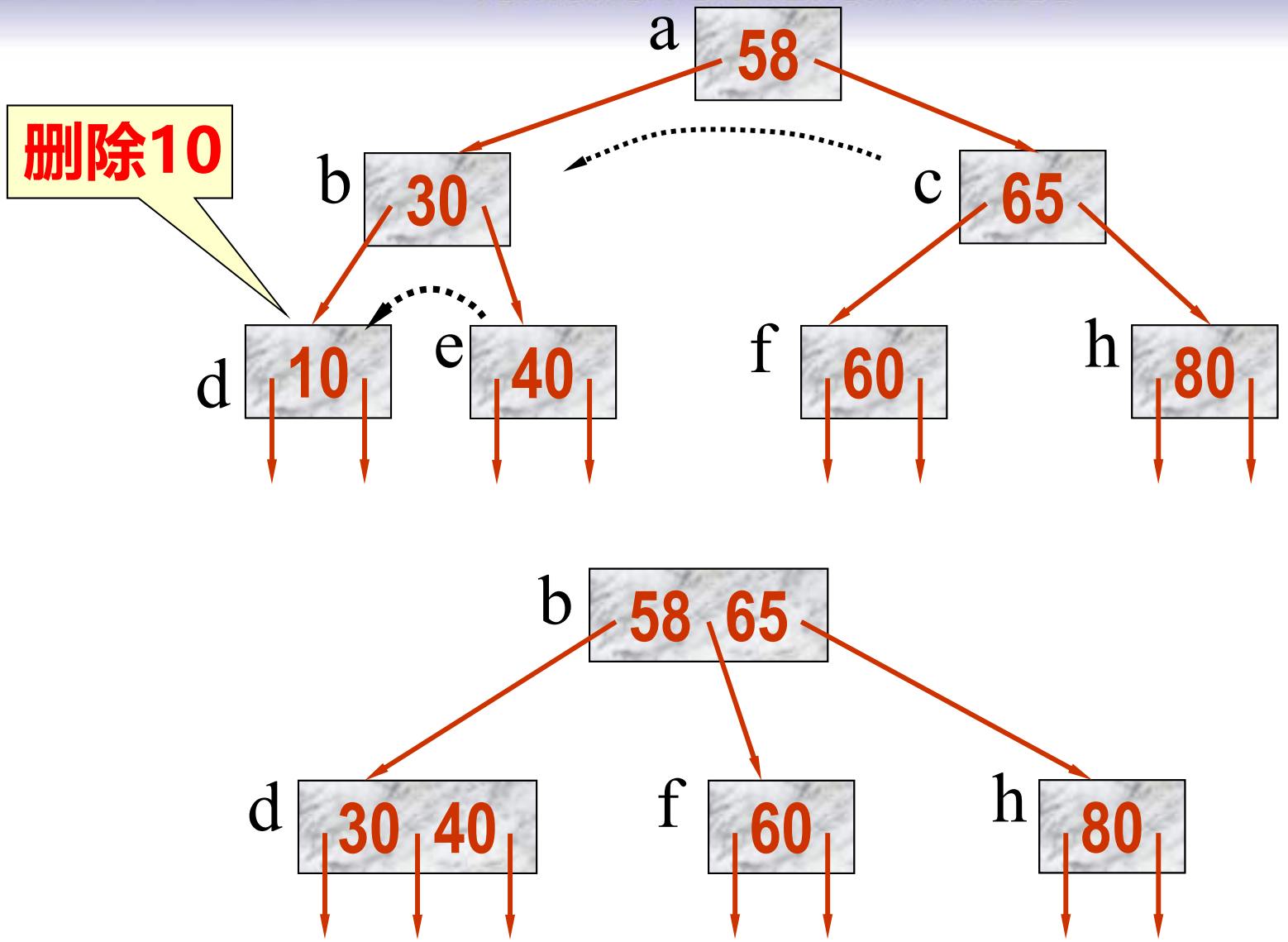


删除75

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 树的关键码删除算法

```
template <class Type> int Btree<Type> ::
Delete ( const Type & x ) {
    Triple<Type> loc = Search (x);    //搜索x
    if ( loc.tag ) return 0;          //未找到, 不删除
    Mnode<Type> *p = loc.r, *q, *s;  //找到, 删除
    int j = loc.i;
    if ( p->ptr[j] != NULL ) {              //非叶结点
      s = p->ptr[j];  GetNode (s);  q = p;
      while ( s != NULL ) { q = s;  s = s->ptr[0]; }
      p->key[j] = q->key[1];          //从叶结点替补
      compress ( q, 1 );              //在叶结点删除
```

```
        p = q;                          //转化为叶结点的删除
    }
    else compress ( p, j );             //叶结点, 直接删除
    int d = (m+1)/2;                    //求⌈m/2⌉
    while (1) {                         //调整或合并
      if ( p->n < d -1 ) {              //小于最小限制
        j = 0;  q = p->parent;          //找到双亲
        GetNode (q);
        while ( j <= q->n && q->ptr[j] != p )
          j++;
        if ( !j ) LeftAdjust ( p, q, d, j );   //调整
        else RightAdjust ( p, q, d, j );
```

```
            p = q;                          //向上调整
            if ( p == root ) break;
        }
        else break;
    }
    if ( !root->n ) {                       //调整后根的n减到0
        p = root->ptr[0];
        delete root;  root = p;             //删根
        root->parent = NULL;                //新根
    }
}
```

- 当调整一个结点时需要从磁盘读入1个兄弟结点及写出2个结点。当调整到根的子女时则需要写出3个结点。
- 如果我们所用的内存工作区足够大, 使得在向下搜索时, 读入的结点在做删除时不必再从磁盘读入, 那么, 在完成一次删除操作时需要读写磁盘的最大次数 =

  = 找删除元素所在结点向下读盘次数 +

  + 调整结点时读写盘次数 =

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

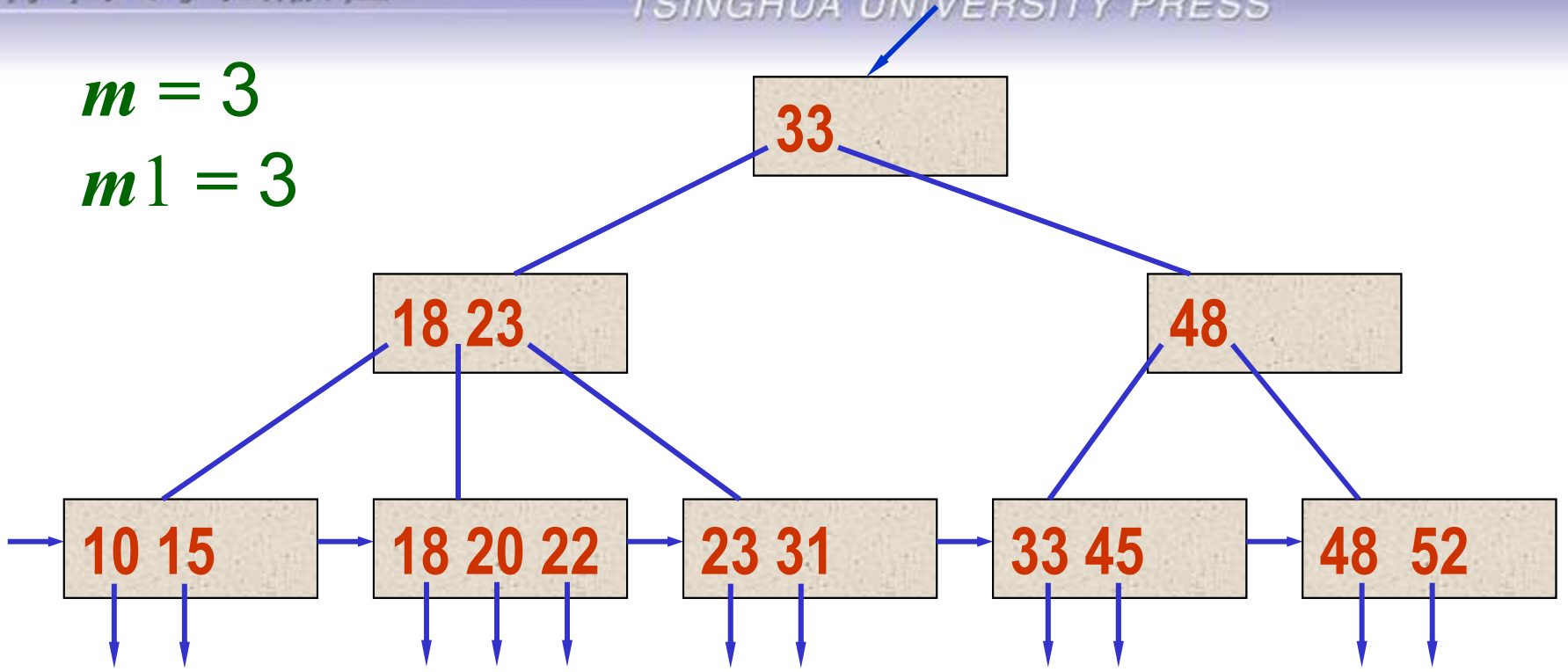
  = $4h-2$。

B+树

- 一棵 m 阶B+树可以定义如下：
  - 树中每个非叶结点最多有 m 棵子树；
  - 根结点至少有 2 棵子树。除根结点外，其它的非叶结点至少有$\lceil m/2 \rceil$棵子树；有 n 棵子树的非叶结点有 n-1 个关键码。
  - 所有叶结点都处于同一层次上，包含了全部关键码及指向相应数据对象存放地址的指针，且叶结点本身按关键码从小到大顺序链接；

$m = 3$

$m1 = 3$



- 叶结点的子树棵数 n 可以多于 m，可以少于 m，视关键码字节数及对象地址指针字节数而定。若设结点可容纳最大关键码数为 $m1$，则指向对象的地址指针也有 $m1$ 个。

- 叶结点中的子树棵数 n 应满足

$$n \in [\lceil m1/2 \rceil, m1]。$$

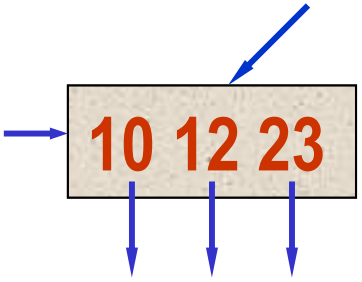
- 若根结点同时又是叶结点, 则结点格式同叶结点，但最少可有1棵子树。
- 所有的非叶结点可以看成是索引部分, 结点中关键码 K_i 与指向子树的指针 P_i 构成对子树(即下一层索引块)的索引项(K_i, P_i)， K_i 是子树中最小的关键码。
- 特别地, 子树指针 P_0 所指子树上所有关键码均小于 K_1。结点格式同B 树。

- 叶结点中存放的是对实际数据对象的索引。

- 在B+树中有两个头指针: 一个指向 B+ 树的根结点，一个指向关键码最小的叶结点。
- 可对B+树进行两种搜索运算：
  - 循叶结点链顺序搜索
  - 另一种是从根结点开始, 进行自顶向下, 直至叶结点的随机搜索。
- 在B+树上进行随机搜索、插入和删除的过程基本上与B 树类似。只是在搜索过程中，如果非叶结点上的关键码等于给定值，搜索并不停止，而是继续沿右指针向下，一直查到叶结点上的这个关键码。

- B+树的搜索分析类似于B 树。
- B+树的插入仅在叶结点上进行。 每插入一个 (关键码-指针) 索引项后都要判断结点中的子树棵数是否超出范围。
- 当插入后结点中的子树棵数 $n > m1$ 时, 需要将叶结点分裂为两个结点, 它们的关键码分别为 $\lceil (m1+1)/2 \rceil$ 和 $\lfloor (m1+1)/2 \rfloor$。
- 它们的双亲结点中应同时包含这两个结点的最小关键码和结点地址。 此后, 问题归于在非叶结点中的插入了。

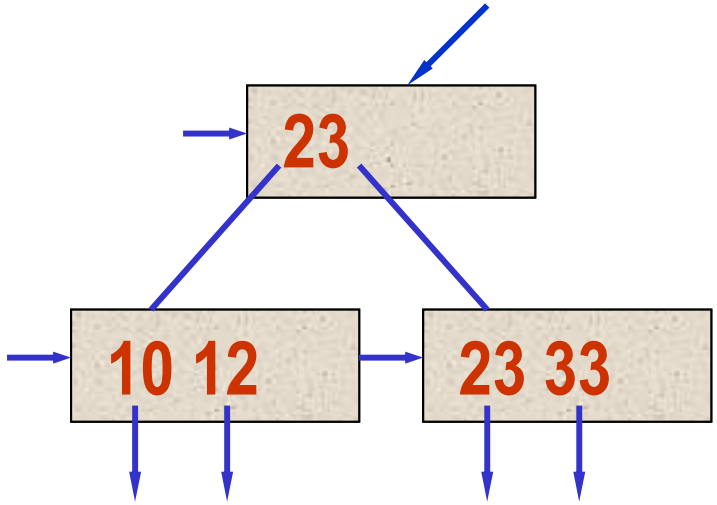
3个关键码的B+树



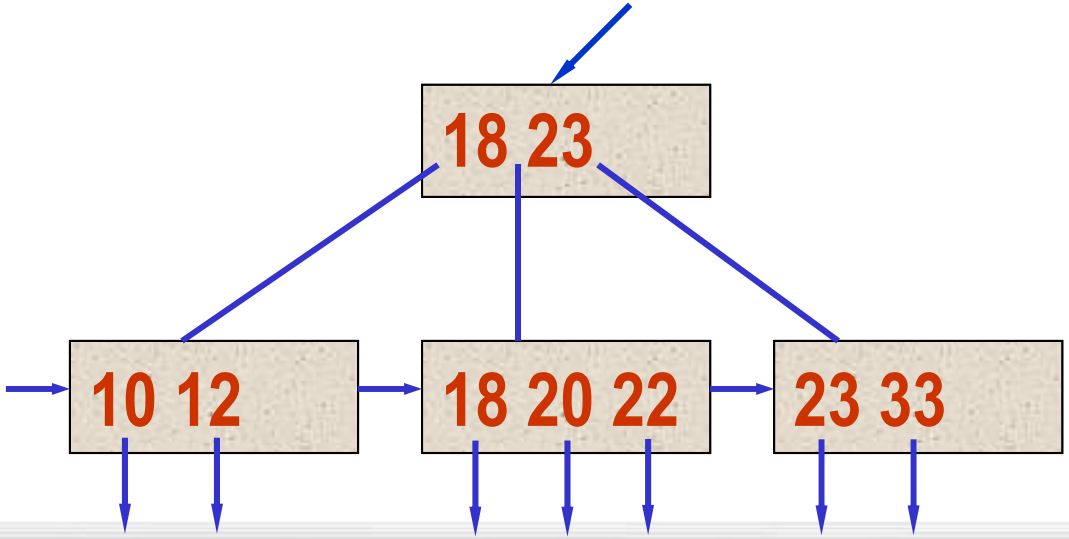
$m = 3$

$m1 = 3$

加入关键码33, 结点分裂



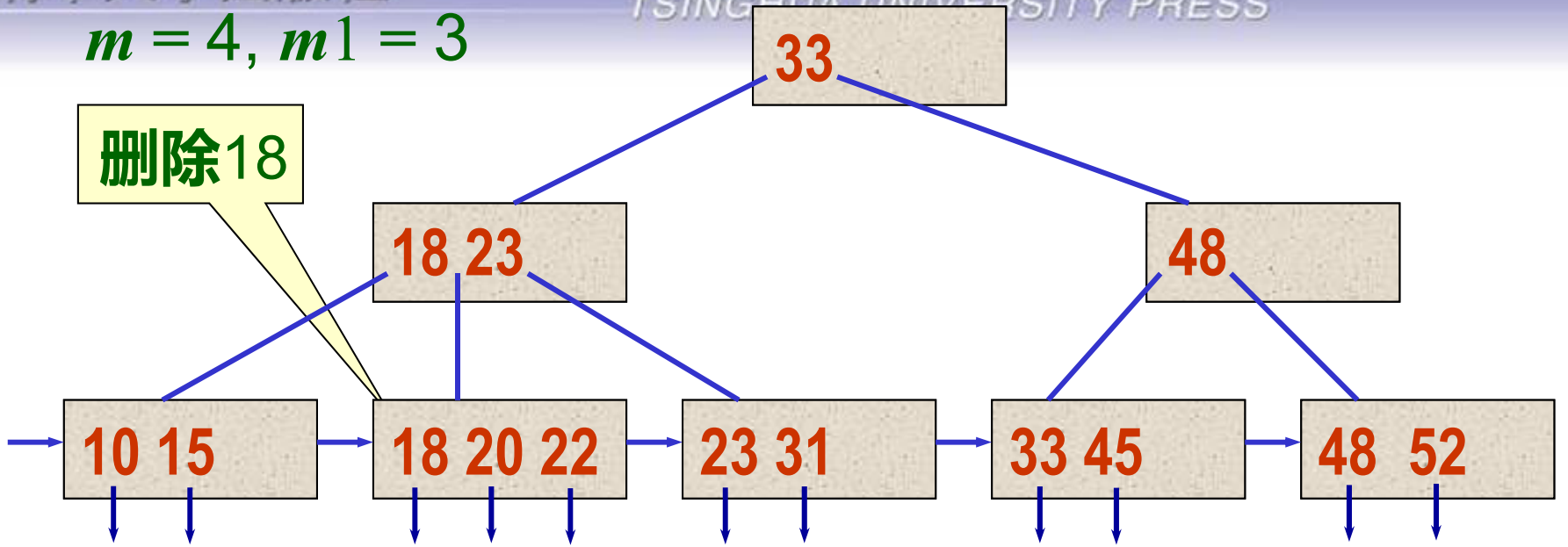
加入更多的关键码



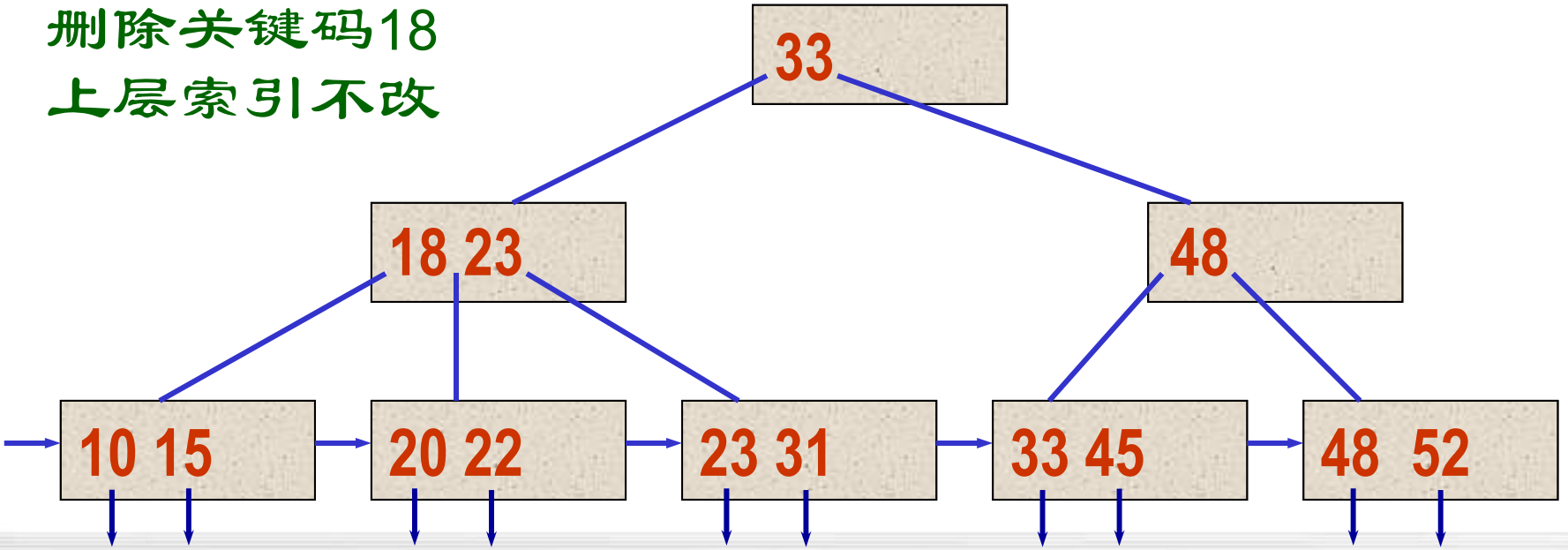
- 在非叶结点中关键码的插入与叶结点的插入类似, 但非叶结点中的子树棵数的上限为 m, 超出这个范围就需要进行结点分裂。
- 在做根结点分裂时, 因为没有双亲结点, 就必须创建新的双亲结点，作为树的新根。这样树的高度就增加一层了。
- B+ 树的删除仅在叶结点上进行。当在叶结点上删除一个 (关键码-指针) 索引项后, 结点中的子树棵数仍然不少于$\lceil m1/2 \rceil$，这属于简单删除, 其上层索引可以不改变。

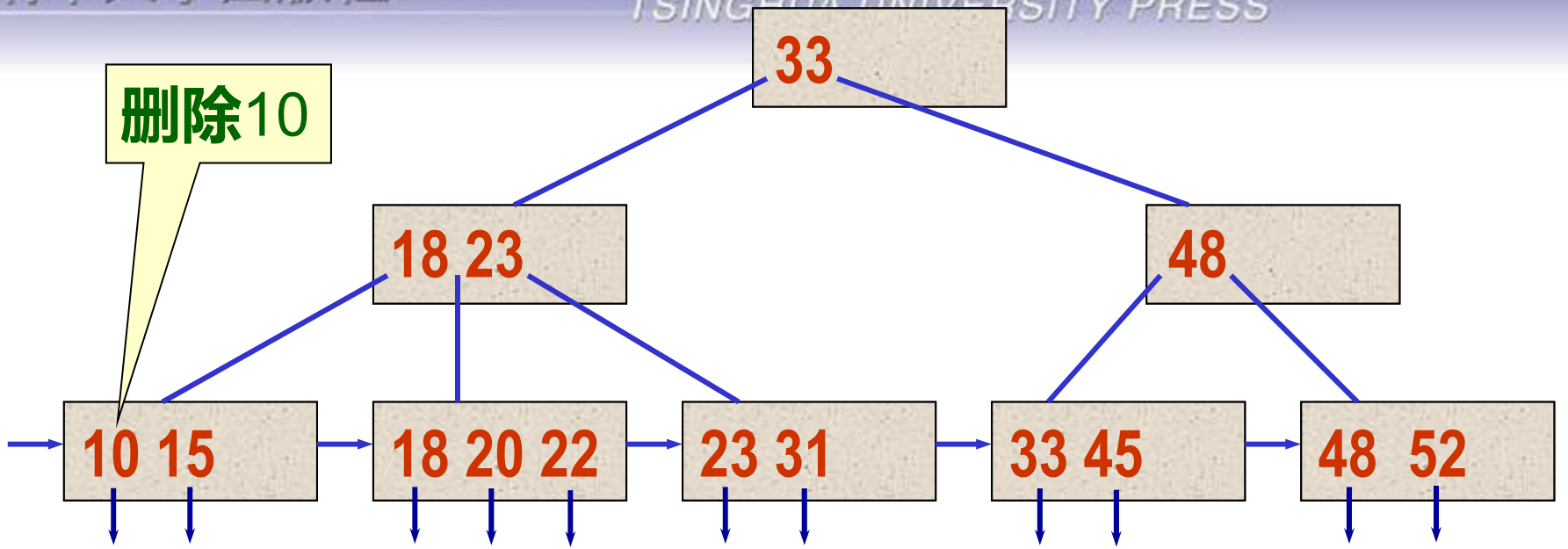
- 如果删除的关键码是该结点的最小关键码，但因在其上层的副本只是起了一个**引导搜索的“分界关键码”**的作用，所以上层的副本仍然可以保留。
- 如果在叶结点中删除一个 (关键码–指针) 索引项后，该结点中的子树棵数 n **小于结点子树棵数的下限**$\lceil m1/2 \rceil$，必须做结点的调整或合并工作。

$m = 4,\ m1 = 3$

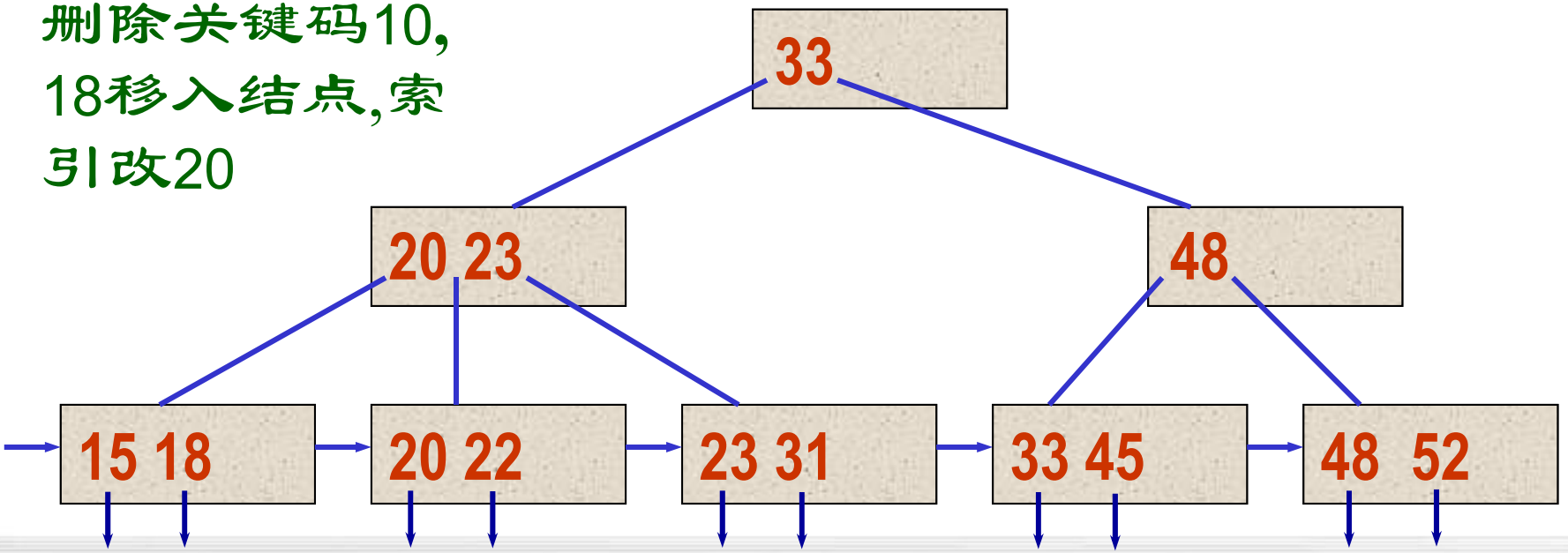


删除关键码18
上层索引不改

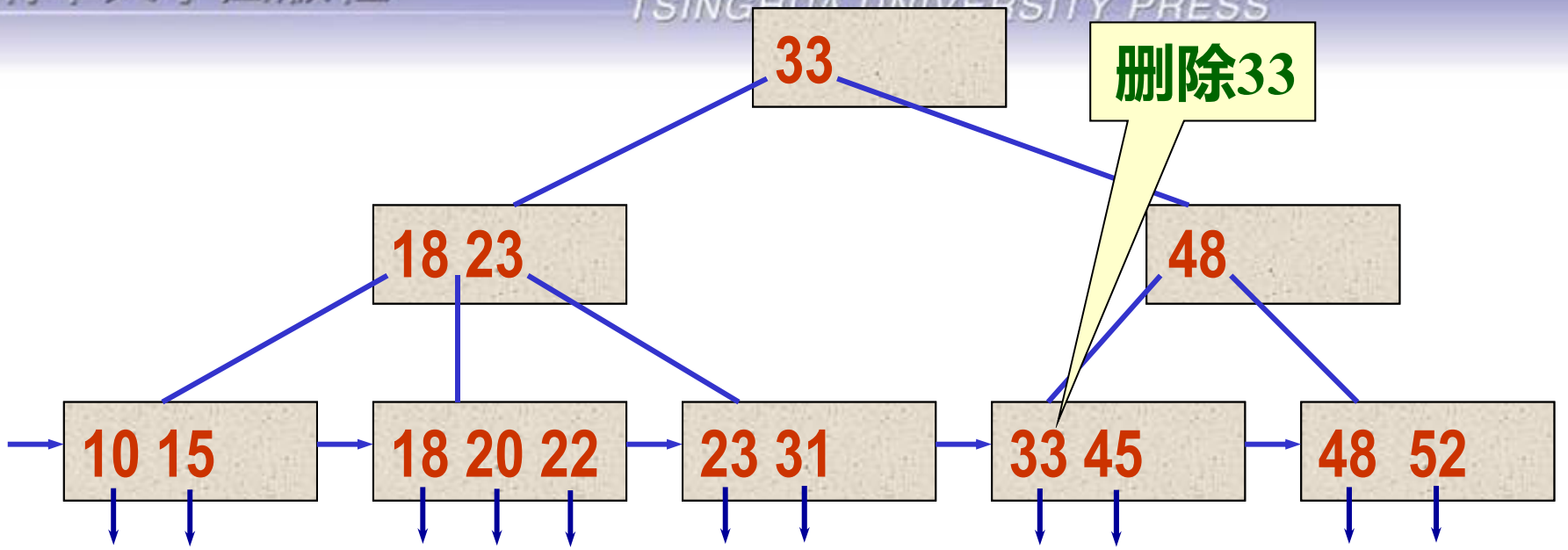




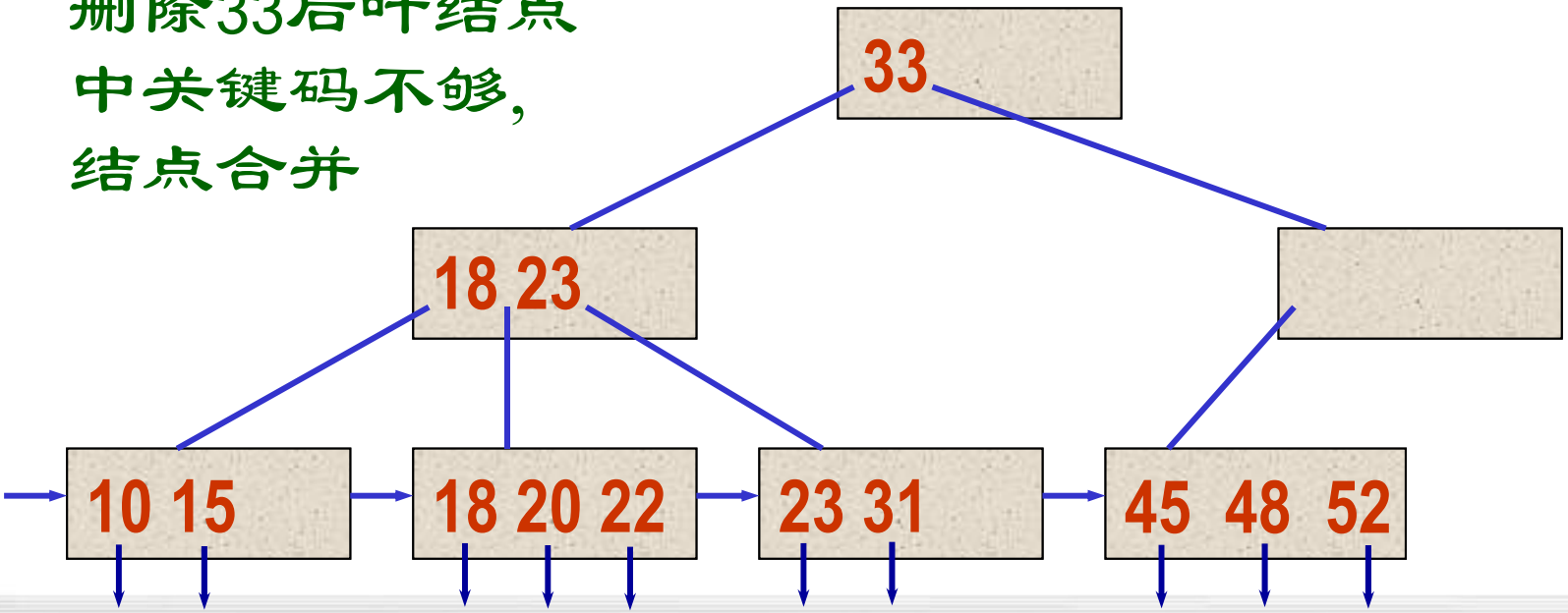
删除关键码10，18移入结点，索引改20

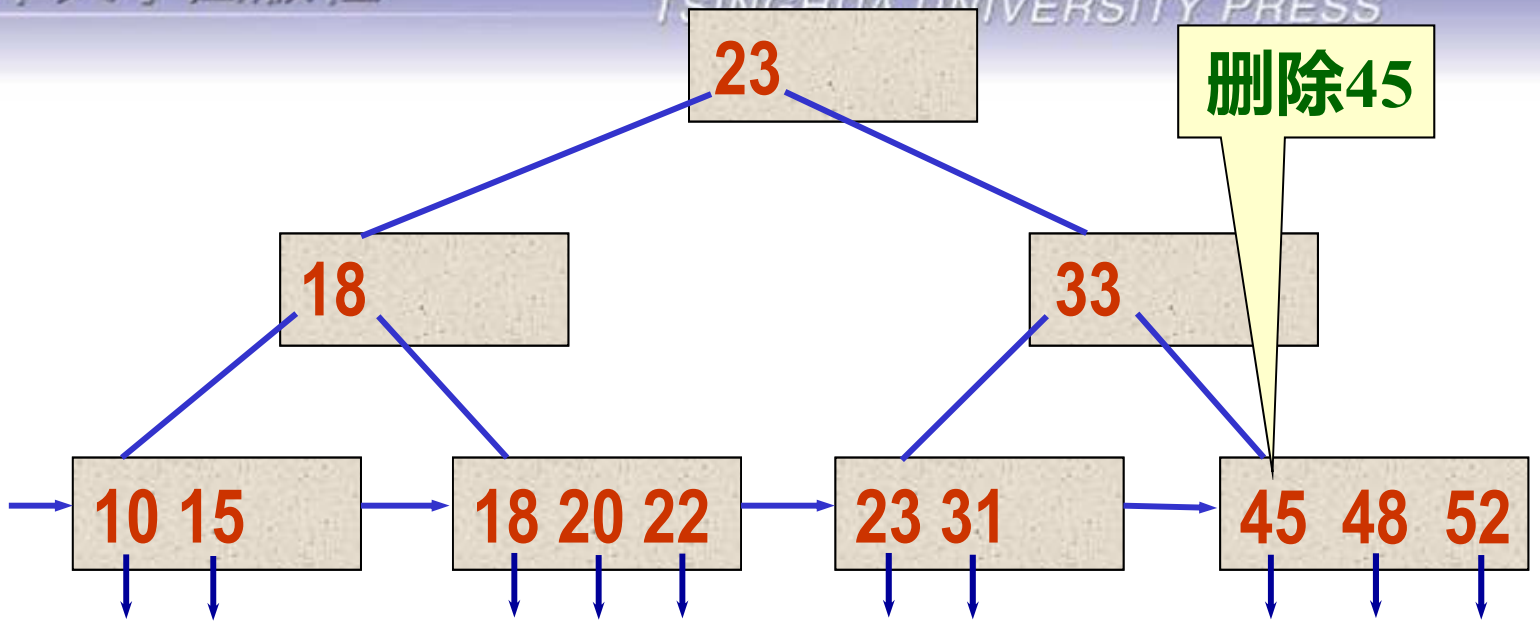


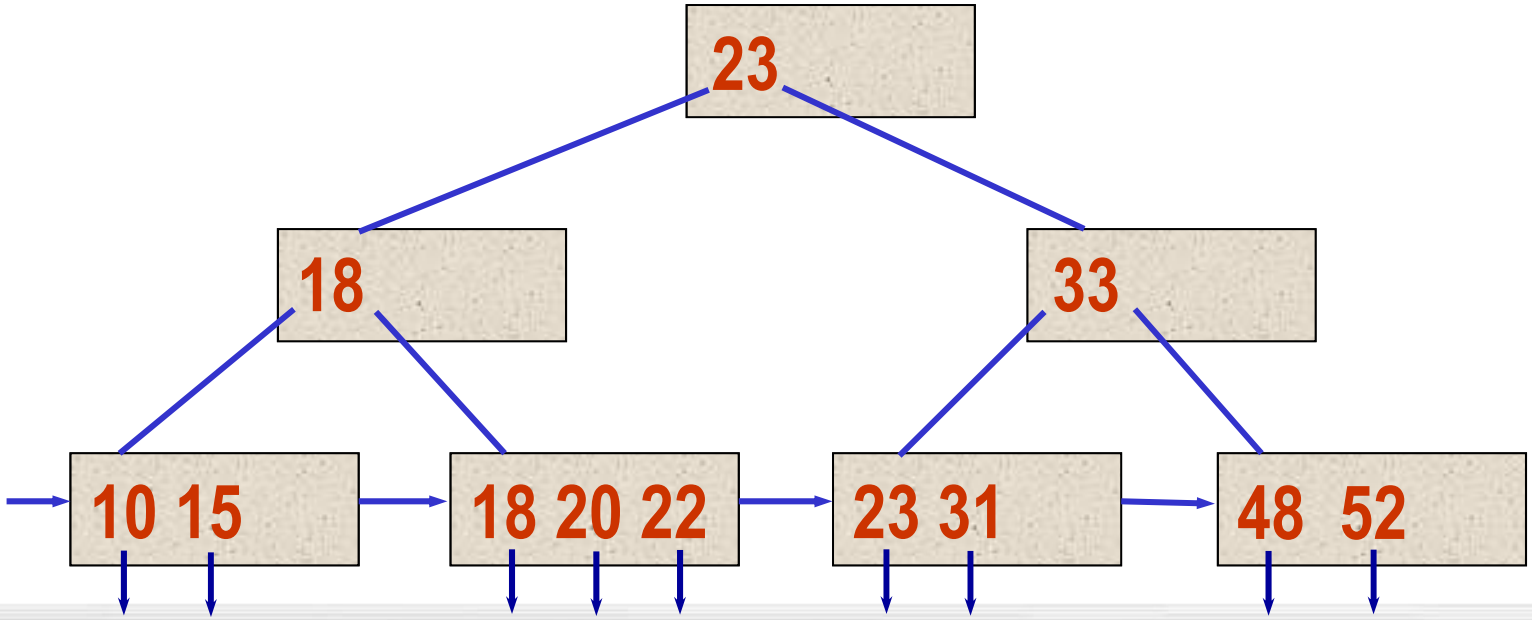
- 如果右兄弟结点的子树棵数已达到下限⌈*m*1/2⌉，没有多余的关键码可以移入被删关键码所在的结点，必须进行**两个结点的合并**。将右兄弟结点中的所有(关键码-指针)索引项移入被删关键码所在结点，再将右兄弟结点删去。
- 结点的合并将导致双亲结点中“分界关键码”的减少，有可能减到非叶结点中子树棵数的下限⌈*m*/2⌉以下。这样将引起非叶结点的调整或合并。如果根结点的最后两个子女结点合并，树的层数就会减少一层。

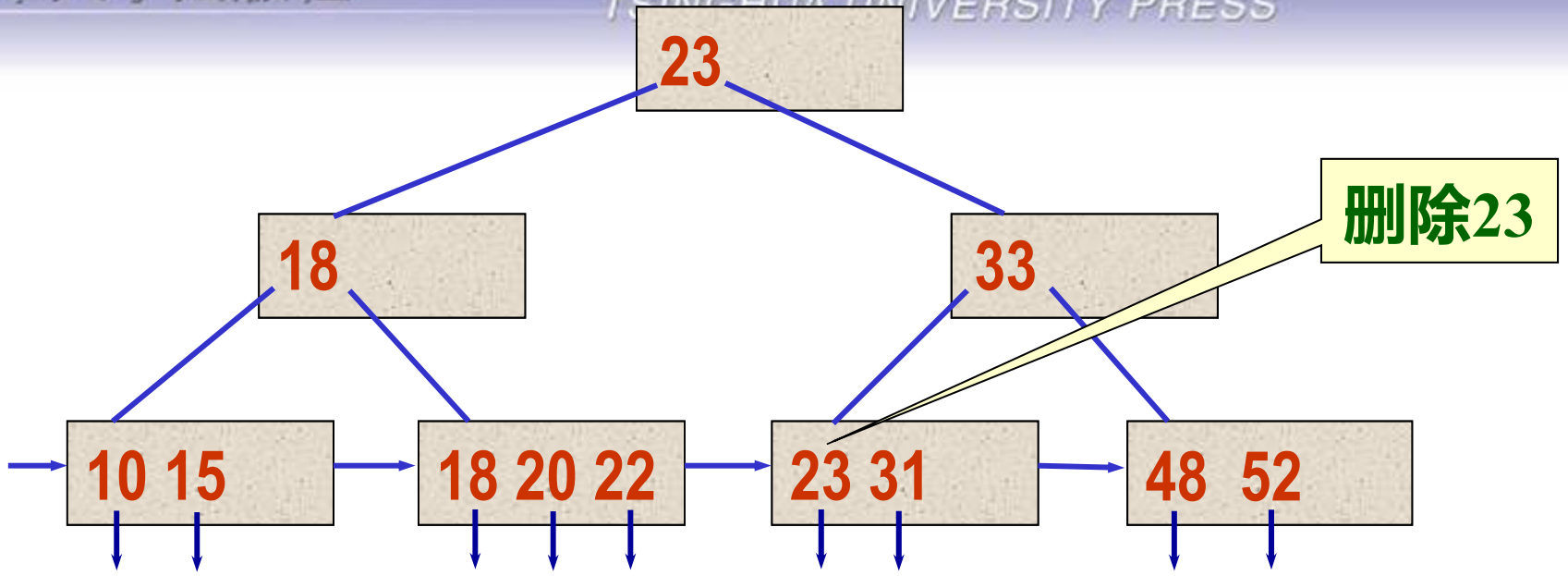


删除33后叶结点
中关键码不够，
结点合并

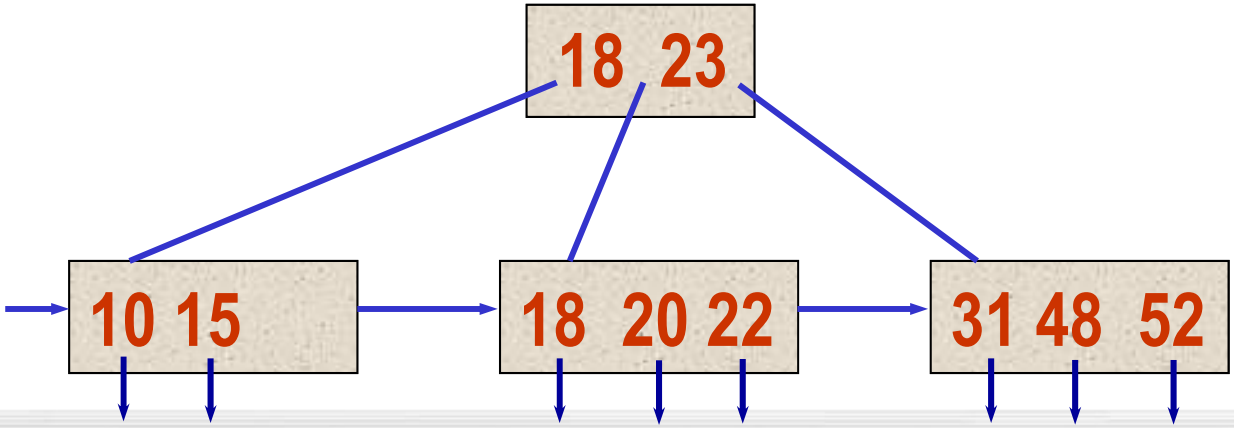



23
删除45
18
33
10 15
18 20 22
23 31
45 48 52


23
18
33
10 15
18 20 22
23 31
48 52



删除关键码23，右边两结点合并，影响到非叶结点合并，可能直到根结点，导致高度减少



可扩充散列

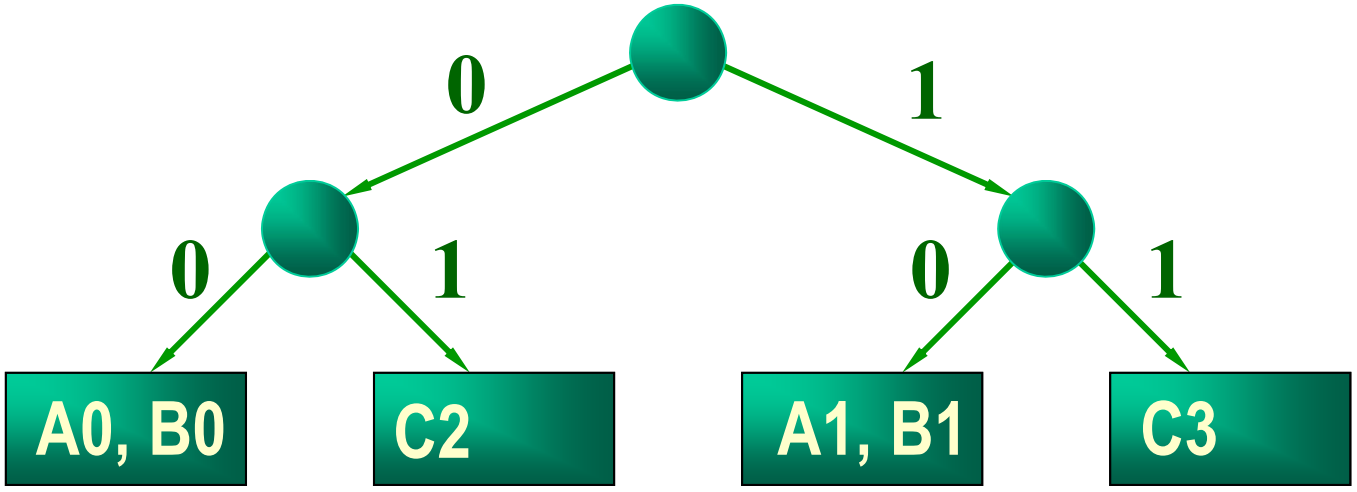
- 可扩充散列是一种动态散列方法，它对传统的散列技术进行了扩充，使之能够动态地适应对文件存储容量的需求，并能保持高效的搜索效率。

二叉Trie树

- 若设每一个关键码由2个字符组成, 而每一个字符用 3 个二进位表示。则有

标识符	A0	A1	B0	B1	C0	C1	C2	C3
二进位表示	100 000	100 001	101 000	101 001	110 000	110 001	110 010	110 011

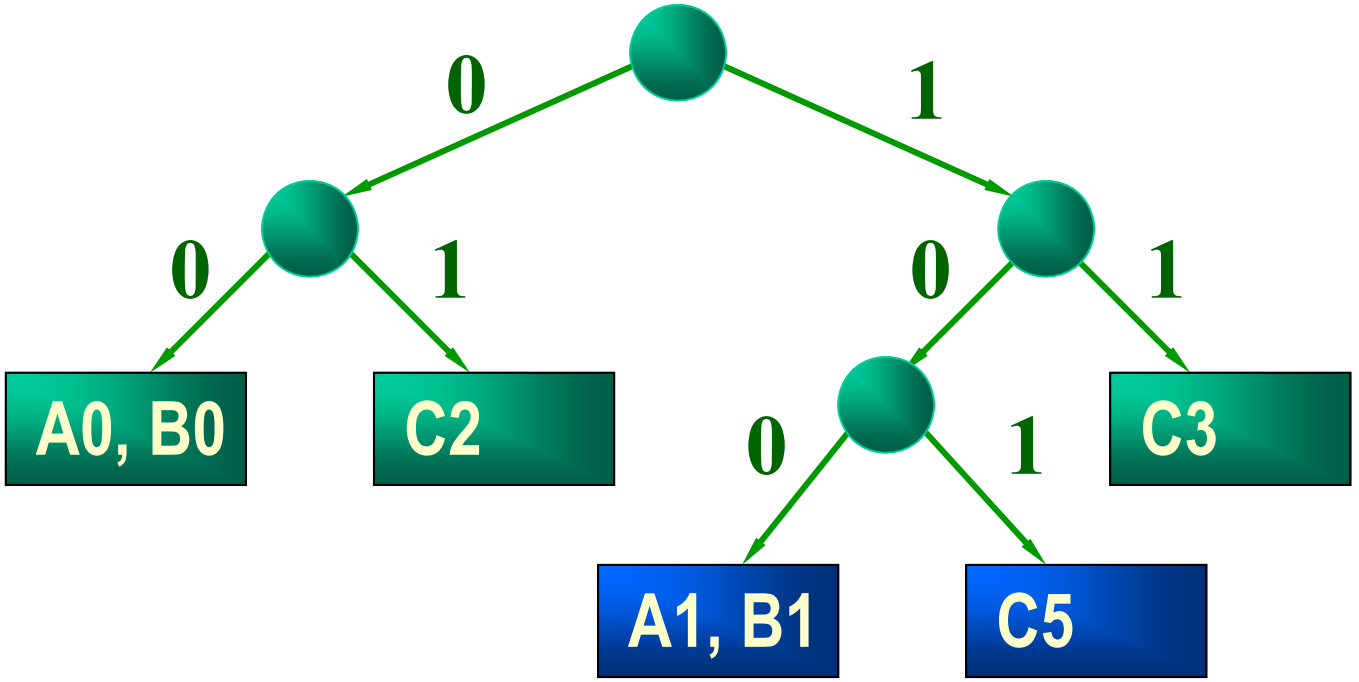
- 假设把这些关键码放到有 4 个页块的文件中, 每页最多可以容纳 2 个关键码, 各页可通过2 个二进位 00, 01, 10, 11 加以索引。现在利用各关键码的最低 2 位来决定它应放到哪个页块上。



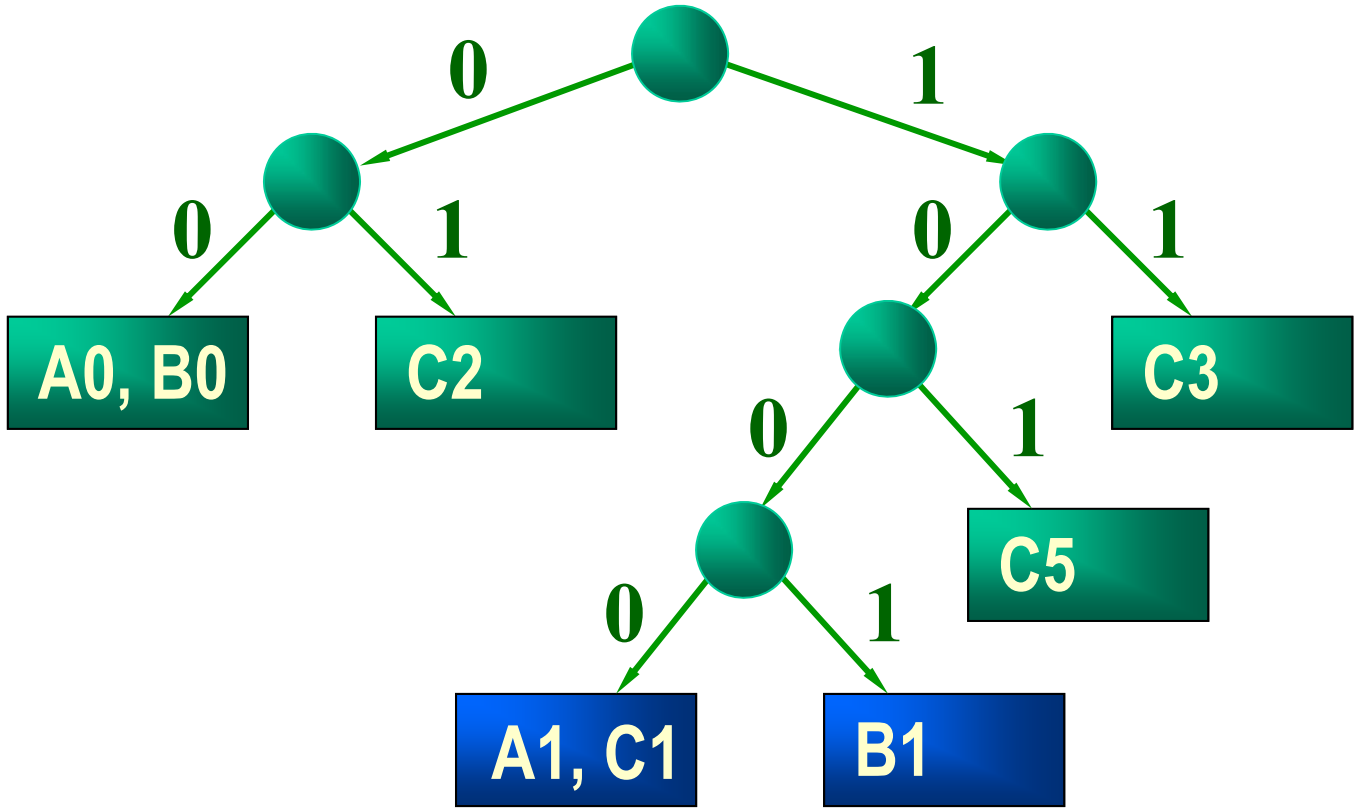
基于4个页块的2层索引

- 一般地, 根据各关键码的二进位表示, 从最低位到最高位进行划分，各页块的索引构成一 个二叉树结构。 这就是二叉Trie树。
- 首先, 在根结点处按各关键码的最低位是 1 还是 0，划分为两组。 对于每一组再按次低位是 1 还是 0 继续分组, 如此分下去, 直到每组剩下不多于 2 个关键码为止。
- 因此，在二叉Trie树中有两类结点：分支结点和叶结点。分支结点按其相关二进位是 1还是 0, 分为两个方向; 仅在叶结点包含指向关键码存放页块的指针。

- 插入新的关键码 C5 时, 因为 C5 的二进制表示110101的最低两位二进位是01，所以应把它放到A1, B1所在的第3页中。但是此时该页块已满，发生了溢出。为此，将第3页扩充一倍，一页分裂为两页。



- 再插入一个新关键码 C1, 它应插入到A1, B1所在的页块中。这时又发生溢出, 需要再分裂 A1, B1 所在页块：根据 A1, B1, C1 的最低 4 位将它们重新分配到这两个页块中。

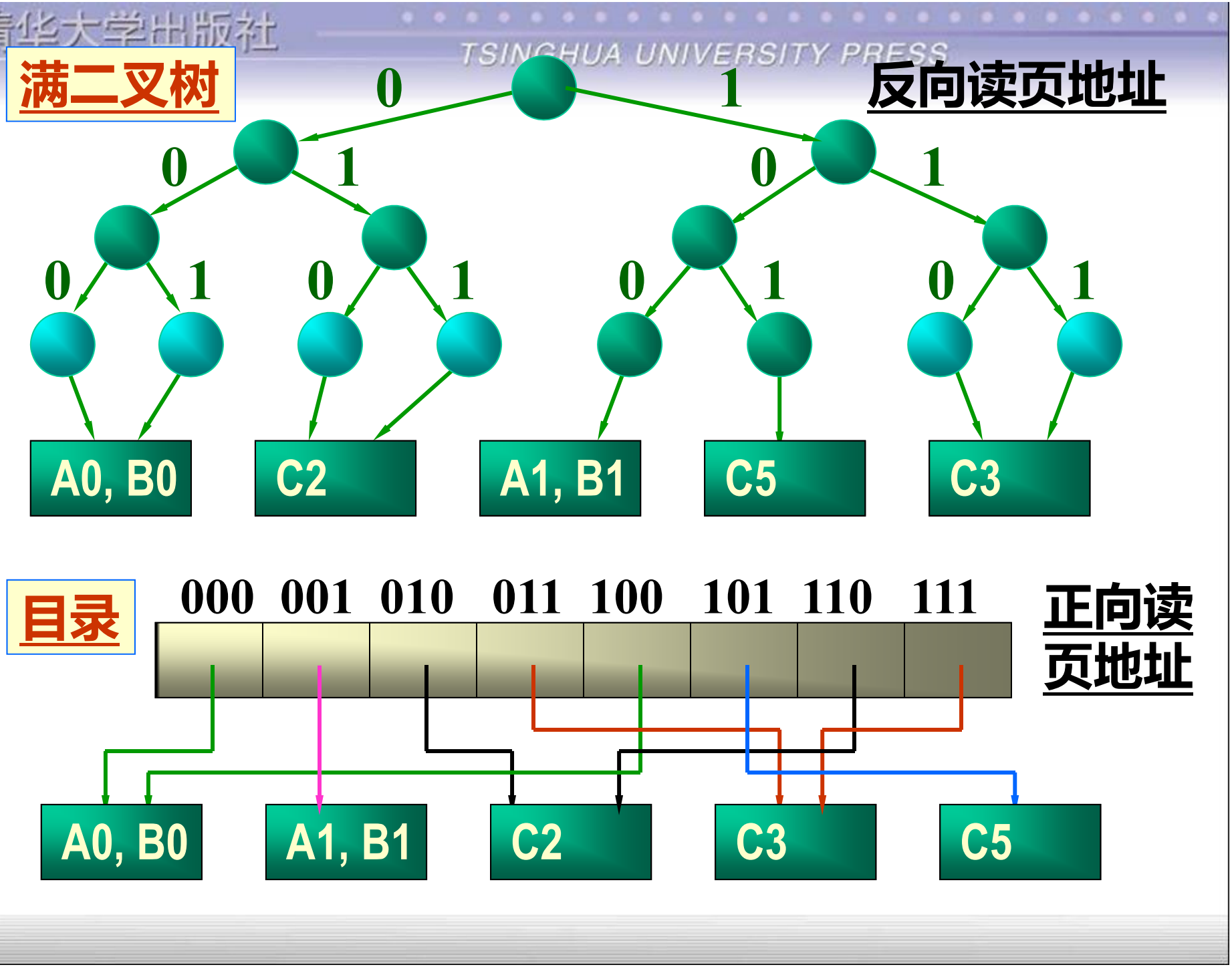


- 以上两个插入结果降低了树搜索效率。为解决上述问题，1979年Fagin等人提出了可扩充散列方法。
- 该方法采用一种特殊的散列函数，把关键码转换成二进制数字表示的伪随机数集合，取其最低若干位作为页块地址，从而避免了关键码的不均匀分布。并且把二叉Trie树映射为目录表，以避开在二叉Trie树中的长时间搜索过程。

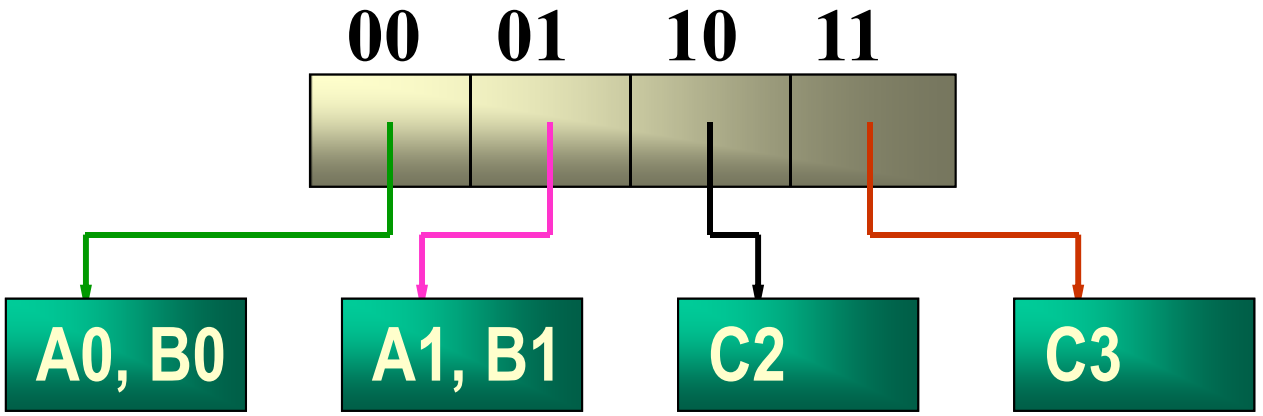
- 利用可扩充散列技术组织的文件为散列文件。
- 若设有 n 个记录，每个记录有一个关键码域用以唯一标识这个记录。又设每个页块有 p 个记录，整个文件有 m 个页块，则散列文件的存储利用率可用 n/(m*p) 来度量。
- 从上面的例子可知，存在着两个问题：
  - 对某个页块的访问时间取决于区分关键码所需的二进位数。
  - 如果关键码的分布不均匀，最后产生的二叉Trie树是倾斜的。

将二叉Trie树转换为目录表

- 为将二叉Trie树转换为目录表，首先需要将二叉Trie树的分支结点部分 (索引部分) 转换为一棵满二叉树——即所有指向叶结点的分支结点都位于同一层次上。然后再把这棵二叉树映射为指示各个页块的目录表。
- 一般地，目录表由一组指向页块的指针组成。如果区分所有的关键码需要 k 位二进制数，则目录项个数应为 2^k，其编址为 0 到 2^k-1。

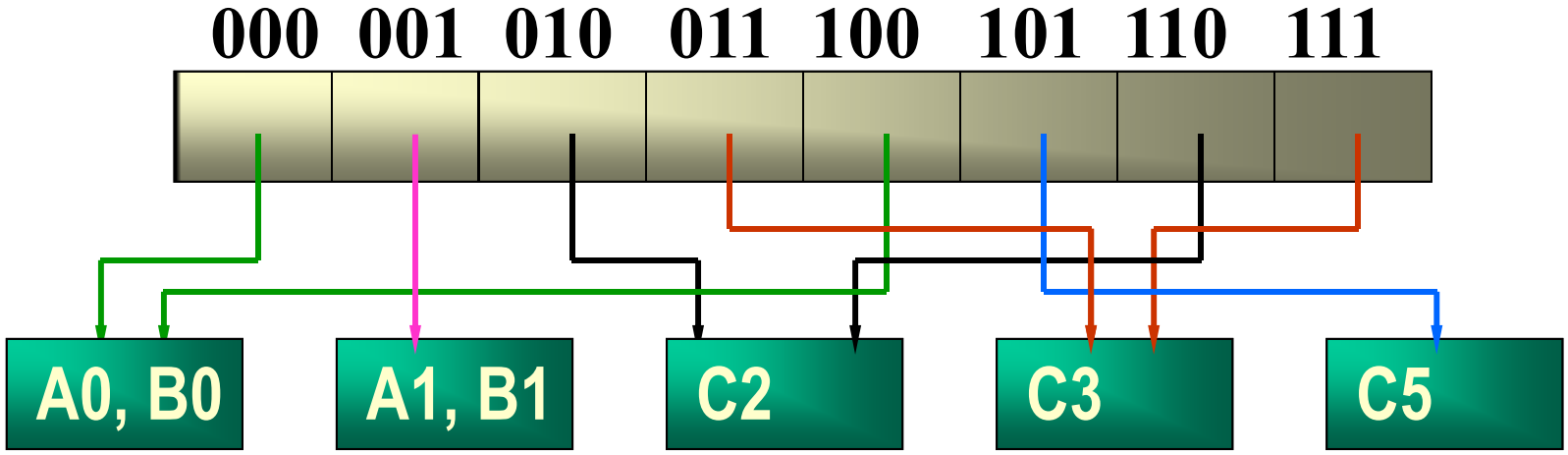

满二叉树
反向读页地址
0
1
A0, B0
C2
A1, B1
C5
C3
目录
000 001 010 011 100 101 110 111
正向读页地址
A0, B0
A1, B1
C2
C3
C5

- 一般使用关键码的最后 k 个二进位作为目录项下标来搜寻该关键码所在页块的指针, 然后对这个页块进行搜索。



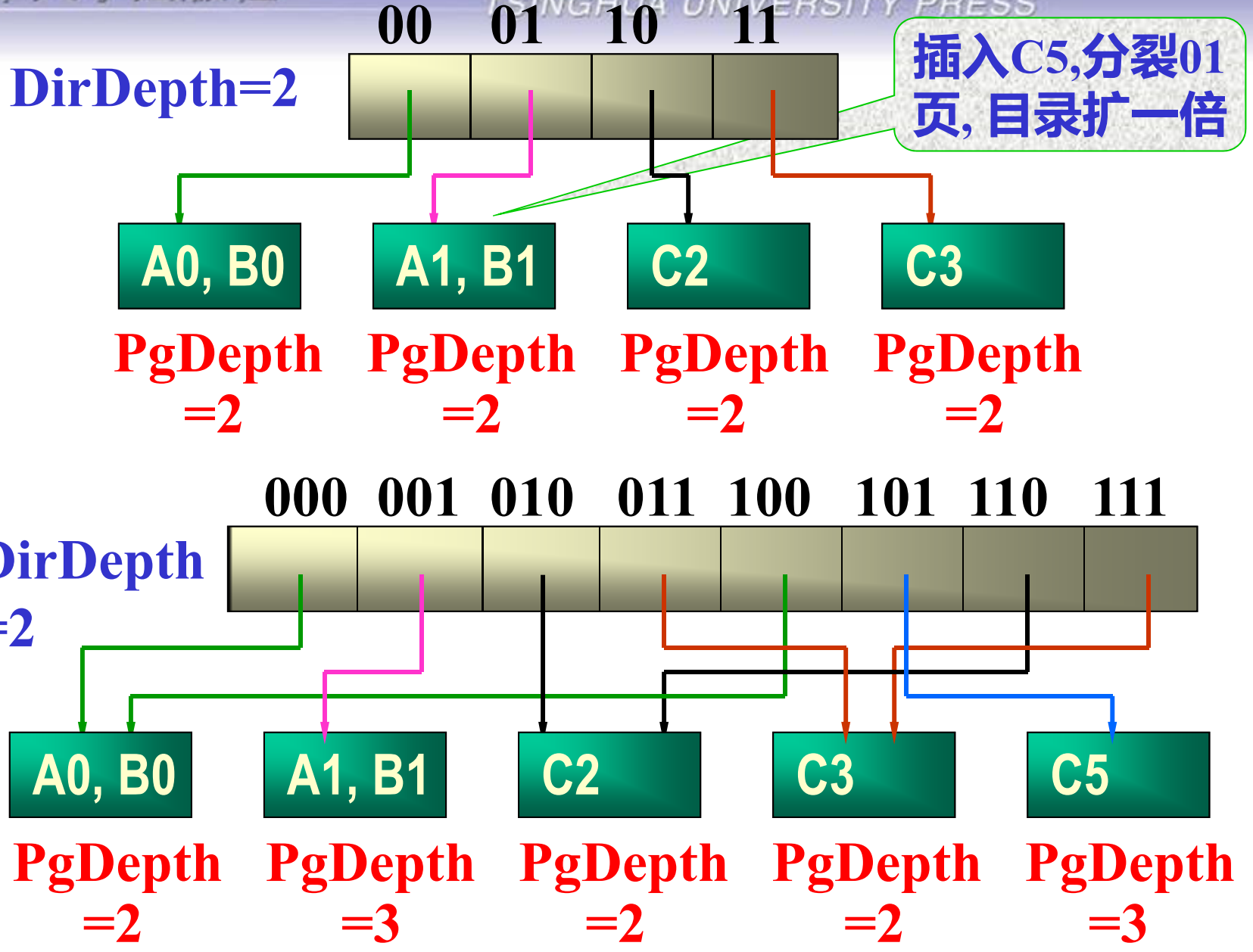
- 上面的目录由 4 个目录项组成, 其编址从 0 到 3。通过目录项的指针, 可找到相应页块的地址, 从而读取该页块的内容。
- 目录或页块的可区分编址位数叫做其深度。

- 在目录表中插入一个新关键码后，其深度可能增加 1。为了保持最高搜索效率，需要压缩目录的深度，使其深度仍为 2。
- 由于有 5 个目录项，需要由最低 3 位来区分它们，为此，设置8个目录项，其下标从0到7。这样，某些页块就有多个指针指向它。



- 使用目录表来表示一棵二叉Trie树，允许关键码序列动态地增长和收缩。
- 这里假设操作系统能够有效地为人们提供页块，并能够回收页块到可利用空间中去。
- 访问一个页块的步骤为：
  - 利用散列函数找到目录项的地址；
  - 按此地址检索相关的页块。
- 如果所有的关键码在各个页块中的分配不均衡，则目录表可能会增长得相当大，且多数指针指到同一个页块。为防止出现此种情况，使用一个均匀分布的散列函数得到一个随机地址，用它作为二进位下标。

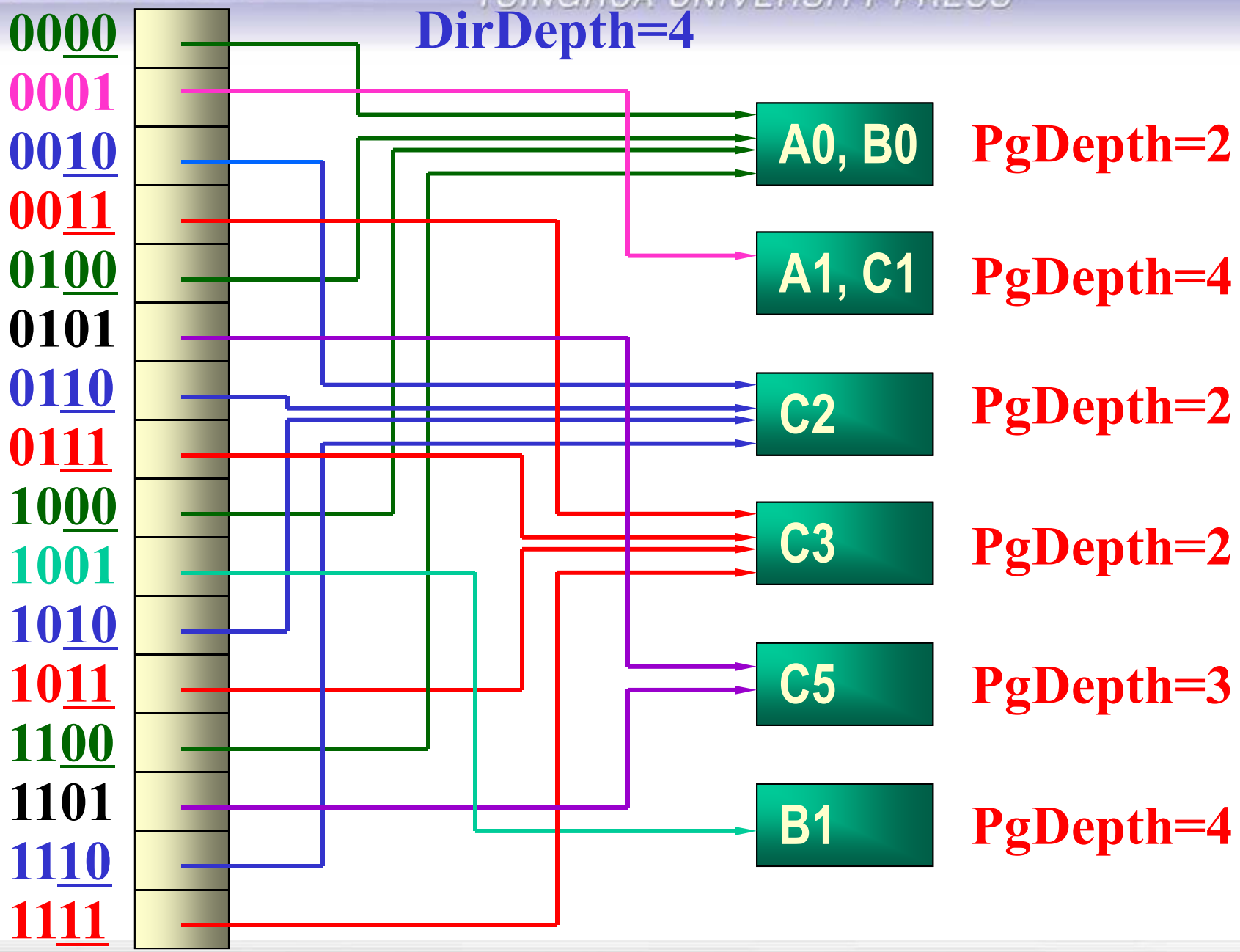
- 可扩充散列结构由一组页块和一个目录表构成。每一个页块包括的信息有
  - 该页块的局部深度PgDepth， 它指明该页块中存放的对象的二进位地址的低位部分有多少位是相同的；
  - 该页块关键码个数Count和存放关键码的数组Names。页块都存储在文件中。
- 目录表由目录项组成。每个目录项是一个指示某一页块的地址的指针。一般用一个数组Directory来存放它们，其下标范围从 0 到 MaxDir-1。目录表深度为DicDepth, 它指明为区分各个页块所需的二进位数。


00 01 10 11
DirDepth=2
插入C5,分裂01页, 目录扩一倍
A0, B0
A1, B1
C2
C3
PgDepth =2
PgDepth =2
PgDepth =2
PgDepth =2
000 001 010 011 100 101 110 111
DirDepth =2
A0, B0
A1, B1
C2
C3
C5
PgDepth =2
PgDepth =3
PgDepth =2
PgDepth =2
PgDepth =3

关键码插入及页块分裂

- 在向一个页块插入关键码 key 时, 如果该页块不满，可以直接将关键码插入；如果该页块已满，需要分裂页块。
- 若原来页块的PgDepth = d，分裂后两个伙伴页块的二进制地址都增加一位，它们的局部深度PgDepth = d+1。除了低阶共享的d位外, 用最高阶的一位来区分它们。
- 在页块分裂后，如果原来指向它的指针不为1，则必须扩充目录表。

- 扩充目录表时：
  - 让目录表的深度加1，所有的目录项增加一倍；
  - 原来地址为××的目录项中的页块指针分别放到 0×× 和 1×× 的两个目录项中。
- 如果目录项的二进制地址有 d 位, 则整个目录表的深度DicDepth = d, 目录项个数有 2^d 个。
- DicDepth = $\log_2$(目录表中目录项个数) = 目录表的二进位数。


DirDepth=4
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
A0, B0
PgDepth=2
A1, C1
PgDepth=4
C2
PgDepth=2
C3
PgDepth=2
C5
PgDepth=3
B1
PgDepth=4

- 有时有多个指针指向同一个页块, 它表明该页块的局部深度小于目录表深度:

  $PgDepth < DicDepth$

- 如果页块的$PgDepth = d'$，则目录表中同时指向该页块的指针个数必为$2^{d-d'}$。
- 初始时$DicDepth=0$, 目录项个数$n = 2^{DicDepth} = 1$, 所有关键码都在同一个页块上。

关键码删除及页块合并

- 在执行关键码删除时：若两个伙伴页块中的关键码个数总和少于或等于其中一个单独页块的容量时，就应把这两个页块合并成一个。
- 求二进制地址为 p 的某目录项所指页块的伙伴页块的方法：
  - 若设目录项 p 所指页块的 PgDepth=d'，把地址 p 的第 d' 位求反，得到一个新的二进制地址 p'，地址为 p' 的目录项所指页块即为原页块的伙伴页块。
- 页块中关键码减少可能会导致目录表的紧缩。

- 合并伙伴页块的工作包括：
  - 将关键码合并到其中任一个页块中；
  - 释放空闲的页块；
  - 将原来的指针都指向合并后的页块。
- 如果目录中指向每一个页块的指针个数都大于或等于2，即

  所有页块的 PgDepth < DicDepth
- 需要紧缩目录。其过程与目录扩充过程相反。

性能分析

- 可扩充散列技术是一种非常好的问题解决方法：其散列文件的地址空间可以扩充和收缩，文件自身能够增长和缩小。
- 在时间方面：用基于目录表的可扩充散列方法进行检索，仅需要2次磁盘访问。因为不存在溢出问题，访问时间与文件大小无关，时间开销达到O(1)。
- 在空间方面：因增加不均匀分布的关键码可能会导致目录表扩充一倍，许多指针指到同一页块，这样会消耗许多存储空间。

- 在空间利用率方面评价散列技术的方式仍然用装载因子α来衡量:

$$\alpha = \frac{\text{表中已存储的记录数}}{\text{表中最大可容纳记录数}}$$

- Fagin, Nievergelt等人做了一系列的试验和模拟。他们发现，可扩充散列的页块空间利用率在 0.53 到0 .94 之间周期性地变动。若给定的记录数为 r, 一个页块可容纳 b 个记录, 则所需的平均页块数 N 和装载因子 α 为:

$$\alpha = \frac{r}{b \cdot N} \qquad N = \frac{r}{b \cdot \ln 2}$$