

高级数据库系统及其应用

第2部分 关系数据库系统实现

第5章 数据库索引技术

xshxie@ustc.edu.cn

LOGO

第5章 数据库索引技术

5.1

索引技术基础

5.2

B+树索引

5.3

散列索引

5.4

位图索引

5.5

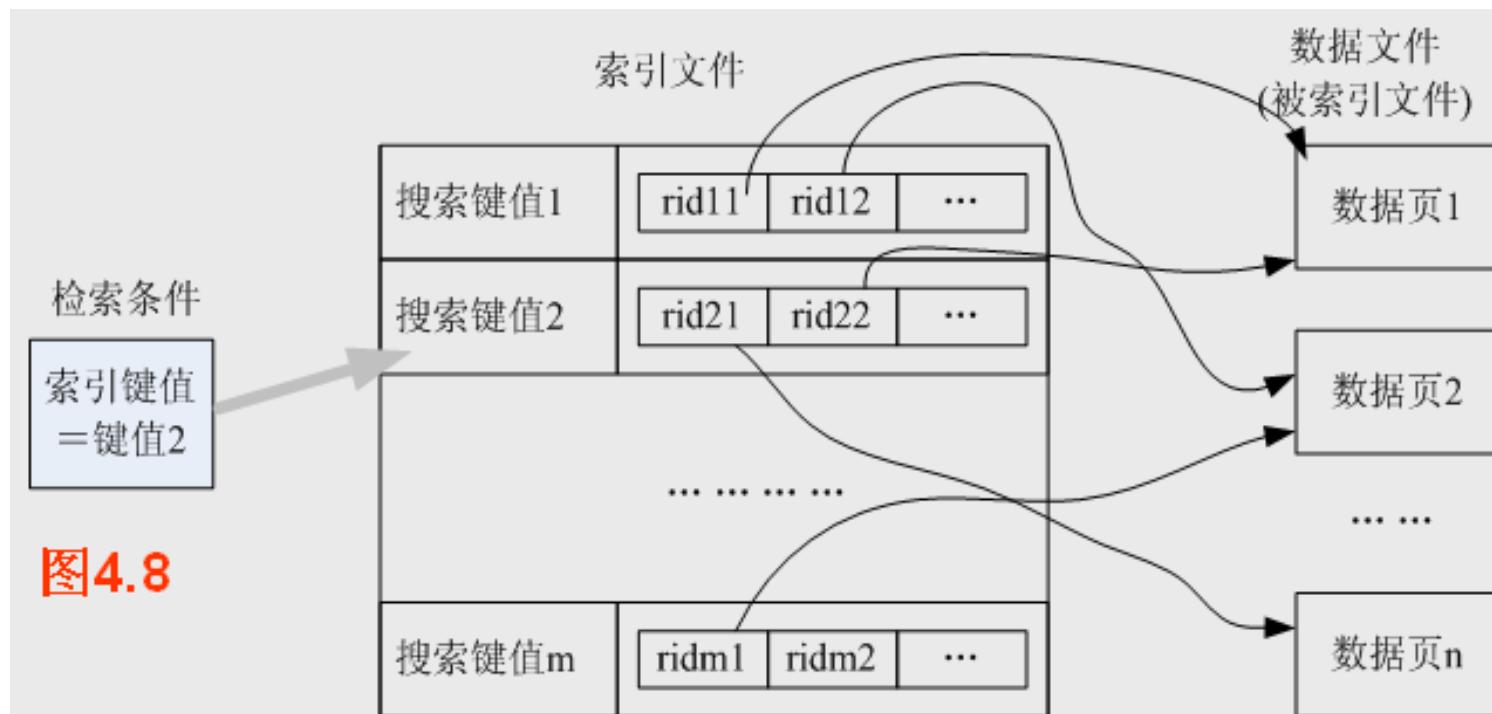
多维空间索引

5.2 索引技术基础

5.2.1 索引技术综述

5.2.2 顺序索引及其特性

5.2.3 创建索引语句



5.2.1 索引技术综述

❖ 索引定义

- 是一种能帮助我们有效找出满足指定条件记录rid的辅助数据结构，是一种特殊类型的记录结构文件。

❖ 索引记录

- 也称，索引项(index entry)，常简记为k*
- 索引项组织，除了可按索引键值顺序组织，还有按树结构(如B+树)和桶结构(散列)进行组织。

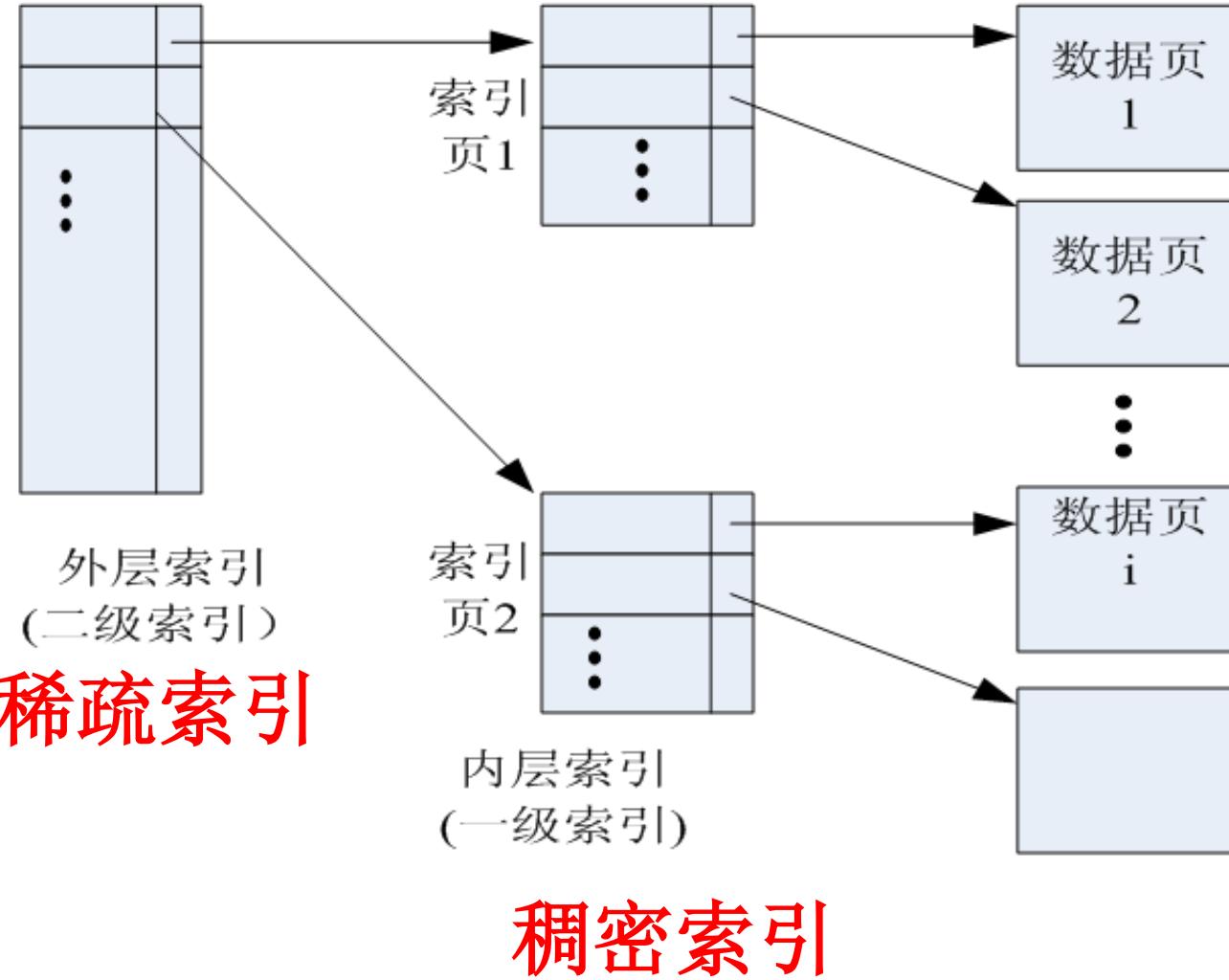
❖ RDBMS中，索引项可能具有的三种形式

- (1) 索引项k*是数据记录本身，无单独的索引文件。
 - 这种情况，数据文件，一般采用散列文件。
- (2) <k, rid>，有独立的索引文件。
- (3) <k, rid-list>，有独立的索引文件，且每个索引项中允许含多个rid

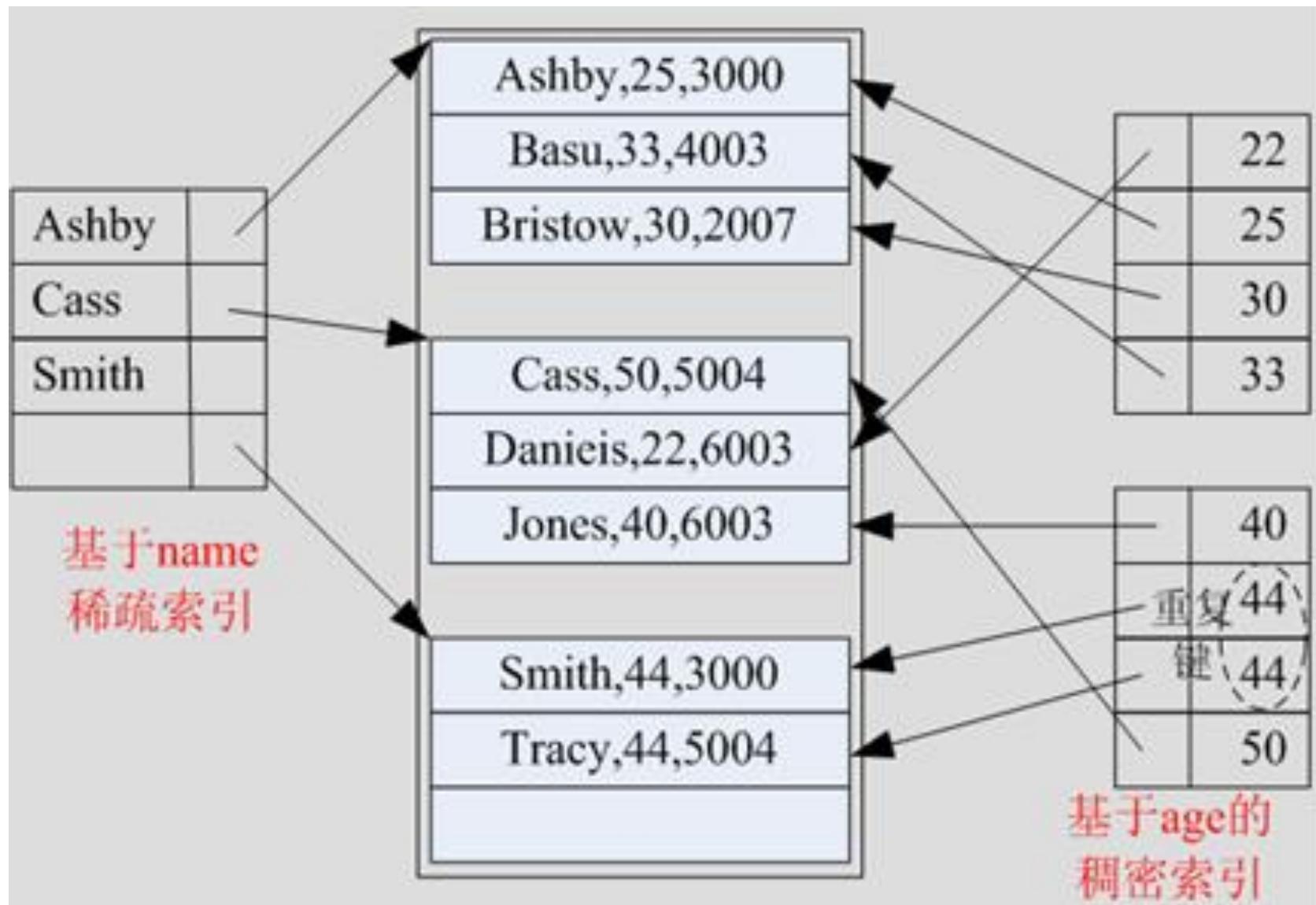
5.2.1 顺序索引及其特性

- ❖ **UNIQUE索引**--会确保索引列中的值是唯一的, 可null
- ❖ 聚集与非聚集索引 ([non]clustered index)
 - **聚集索引**: 索引键, 与数据文件的**排序键相同**, 即数据的物理存储顺序与索引顺序相同
- ❖ 稠密索引与稀疏索引
 - **稠密索引**: 每个索引键值都对应有一个索引项
 - **稀疏索引**: 只针对某些搜索键值--建立索引项
- ❖ 多级索引
 - 索引项过多, 会带来索引性能下降问题。可将索引文件本身当作一般顺序数据文件, 在其上再建一个索引, 即二级索引。
 - 如果需要建立三级或更多级的索引, 通常不如直接使用B树方便。
- ❖ 主索引与辅助索引
 - 以主码作为为搜索键的索引, 称为主索引。

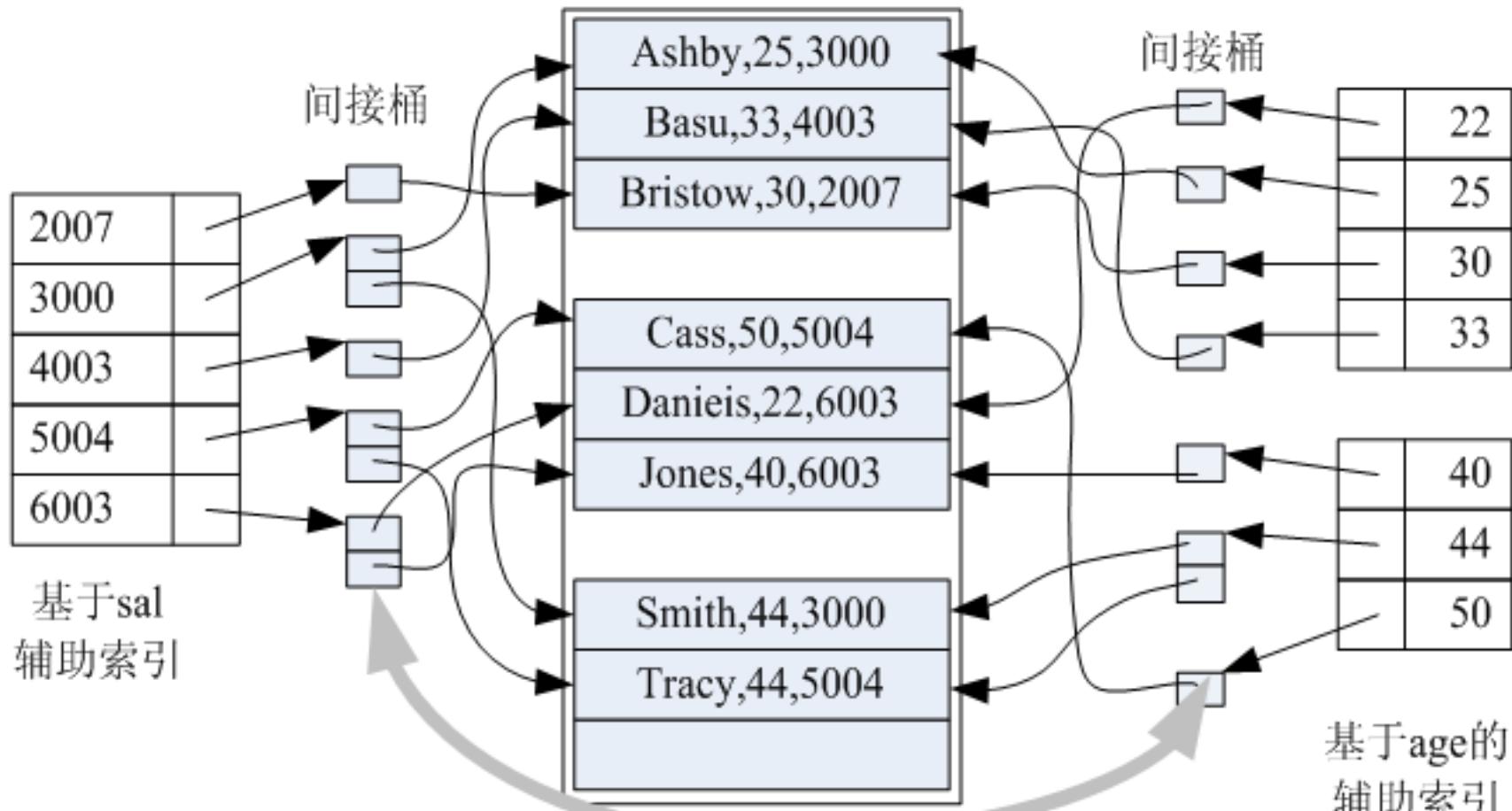
多级索引应用示例



稠密索引与稀疏索引应用示例



一种带有间接桶层的辅助索引结构



对形如 $(\text{sal} \leq V_x) \text{ and } (\text{age} \leq V_y)$ 的条件检索
可先求分别满足一条件的桶指针，再求两类桶
指针交集，再由交集中指针定位目标记录

创建索引语句

❖ 基本语法

```
CREATE [UNIQUE] [CLUSTERED|NONCLUSTERED]
    INDEX index_name USING {BTREE|HASH}
    ON table_name (col1 [ASC|DESC], col2...);
```

示例

```
CREATE INDEX user_name_idx
    ON users (first_name, last_name);
```

❖ 删除索引

```
DROP INDEX index_name ON table_name;
```

5.2 B+树

5.2.1 B+树概述

5.2.2 B+树操作

5.2.3 B+树的效率与实用化

5.2.1 B+树特点及约束（1）

❖ B+树的技术特点

- B树操作（插入/删除）能保持树平衡。从根节点到任一个叶节点路径都是等长的。
- B⁺树是传统B树的**增强**结构。采用平衡树来动态地组织索引项，树大小会因数据项的多少而动态地增长或收缩。
- 每个树节点用一个页来存储。内节点用于搜索导向，叶节点用来存储数据项或数据项指针。

❖ B+树的阶数（通常以字母m表示）

- 指B+树中节点允许容纳的最大索引键值个数。

5.2.1 B+树特点及约束（2）

❖ 根节点/内节点格式化

- 除了根节点外，所有树节点都必须保持50%的占用率（即半满）。
- 一个含有 j 个索引键值的节点，必含有 $j+1$ 个指针
 - 节点内容格式“ $p_0, K_1, p_1, \dots, K_j, p_j$ ”，其中，指针 p_i 指向一个键值 k 落在 $K_i \leq n < K_{i+1}$ 范围的子树。
- m 阶B+树的非叶节点【根 + 内节点】：
 - 至多只能有 $m+1$ 个子节点
 - 内节点至少要有 $\lfloor m/2 \rfloor + 1$ 个子节点；
根节点至少要有2个子节点

❖ 叶节点格式化

- 至多包含 m 个数据项，至少包含 $\lfloor (m+1)/2 \rfloor$ 个数据项；
- 每个项既可以存放实际数据记录，也可以是形式为 $\langle K, rid | pid \text{指针} \rangle$ （简记为 $k*$ ）的数据记录指针。
- 每个叶子节点与前后的叶子节点用双链连接在一起。

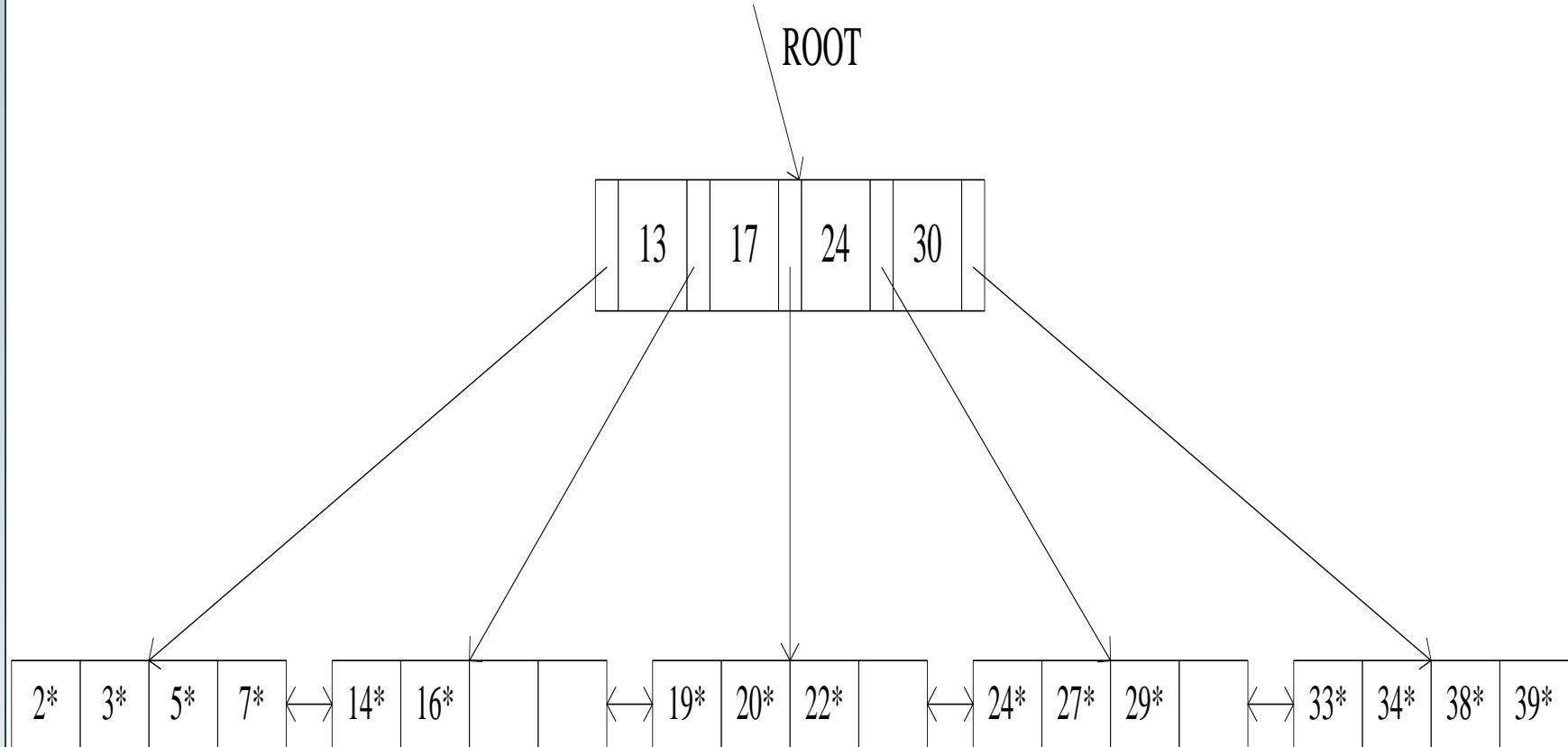
5.2.2.1 B+树搜索算法（算法5.1）

```
//主函数
func find (search-key-value K)
    returns nodePointer
    //给定搜索键值K，找所在的叶节点
    return tree-search (root, K);
endfunc
```

B+树搜索算法（算法5.1）

```
func tree-search(nodePointer, search-key-value K) return nodePointer
    if *nodePointer 是叶节点 then
        return nodePointer;
    else
        if K<K1 then
            return tree_search(P0, K);
        else if K≥ Kn then
            return tree_search(Pn, K);
        else
            find i such that Ki≤ K ≤ Ki+1 //选择子树
            return tree_search(Pi, K);
        endif
    endif
endfunc;
```

一个阶数m=4的B+树及其搜索示例



B+树插入算法

算法 5.2 B+树的插入算法。

```
proc insert (nodePointer, entry, newchildentry)
//插入一个项 entry 到一个：根节点为 * nodePointer, 阶数为 m, d = ⌊m/2⌋ 的子树
//newchildentry 代表下层中新分裂出来的一个节点项，它是一个<键值, 指针>对
//初始时, newchildentry 为 NULL,
if * nodePointer 不是叶节点, 不妨设为 N then
    find i such that Ki <= entry.K <= Ki+1; //选择子树
    insert (Pi, entry, newchildentry); //递归地插入项
    if newchildentry is NULL then //无分裂子节点, 返回
        return;
    else
        if N 中还有空间 then
            在 N 中放置 * newchildentry;
            newchildentry = NULL; return; //置为空值后返回
        else //N 中项已满: 已有 2d 个键值, 2d + 1 个指针
```

B+树插入算法（到达叶节点后的处理逻辑）

```
else if *nodePointer 是叶节点, 令为 L then
    if L 还有空间 then
        放置 entry 到 L 中;
        set newchildentry=null;
        return;
    else
        分裂 L: 前 d 个项留在原节点中, 其余项移到新节点 L2 中;
        newchildentry=&(L2 中的最小键值, 指向 L2 的指针);
        设置 L 和 L2 之间的链接指针;
        return;
    endif
endif
```

插入算法图解

B+树插入算法（非叶节点中分裂子节点处理逻辑）

```
if N 中还有空间 then
    在 N 中放置 *newchildentry;
    newchildentry=null; return; //置为空值后返回
else //N 中项已满：已有 2d 个键值, 2d+1 个指针
    split N; //加入新项后，共有 2d+1 个键值, 2d+2 个指针
        创建一个新节点，令为 N2;
        将前 d 个键值和前 d+1 个指针留在 N 中;
        将后 d 个键值和后 d+1 个指针移到新节点 N2 中;
    //将剩余的一个键(选中间的那个)与指向 N2 的指针组成一个新项;
    设置 newchildentry= & (<剩余的那个中间键值, 指向 N2 的指针>)
if N 是根节点 then
    创建一新节点，内容为<指向 N 的指针, *newchildentry>
    让树的原根节点指针指向这个新节点;
    return;
endif
endif
```

插入算法图解

5.2.2.3 B+树删除算法（算法5.3）

```
proc delete (nodePointer, entry, oldchildentry)
    //从一个根节点为 *nodepointer, 阶数为 d 的子树中删除指定的项 entry
    //oldchildentry 代表当前节点中(因相应下层子节点被删除)而必须删除的项
    //初始时, oldchildentry 为空
    if *nodePointer 不是叶节点,令为 N then
        find i such that Ki <= entry.K <= Ki+1; //选择子树
        delete (Pi, entry, oldchildentry); //递归删除
        if oldchildEntry is null then //子节点未被删除的情形;
            return;
        else
            //处理子节点被删除的情况
        endif
    else if *nodePointer 是叶节点,令为 L then
        //处理在叶节点中删除一个项的情况
    endif
endfunc
```

删除算法图解

B+树删除算法（到达叶节点后的处理逻辑）

```
if L 中有超过半满的多余项 then
    删除 entry 项;      set oldchildEntry=null; return;
else
    取 N 的相邻兄弟叶节点 S; //N,S 有相同父节点,可利用 parentPointer 来找
    set M=rhs ( N, S ); //令 M 代表 N,S 中位于右边的那个节点
    if S 中有多余的项 then
        重新分配 N 和 S 中的项;
        在父节点中找到 M 对应的项, 并用 M 的最小键值替换这个项的键值;
        set oldchildEntry=null; return;
    else //合并 N 与 S;
        oldchildEntry= &(M 在父节点中对应的项);
        将 M 中所有的项移到它的左边节点中;
        调整与 M 相邻的两叶节点链接指针,使它们链在一起后,丢弃释放空节点 M;
    endif
endif
```

删除算法图解

B+树删除算法（处理有子节点被删除逻辑）

从当前节点 N 中, ~~删除~~ oldchildEntry;

if N 中项数仍满足半满要求 then

 set oldchildEntry=null; return;

else

 取 N 的相邻兄弟节点 S; //N,S 有相同父节点,可用 parentPointer 来找

 set M = rhs(N,S); //令 M 代表 N,S 中位于右边的那个节

 if S 中有多余的项 then

 重新分配 N 和 S 中的项;

 在父节点中找到 M 对应的项, 并用 M 的最小键值替换这个项键值

 set oldchildEntry=null; return;

 else //合并 N 与 S;

 oldchildEntry = &(M 在父节点中对应的项);

 //将父节点的分支键压入 M 的左节点中;

 将 M 中所有的项移到它的左边节点中;

 丢弃释放空节点 M; return;

 endif

endif

删除算法图解

5.3.3 B树的效率与实用化

❖ B+树索引的优势

- 虽然B+树付出了在内节点存储索引项的开销，但能获得排序文件的所有好处，且还能保持很好的插入、删除性能。
- B+树没有溢出页；
- 实用条件下，B+树的每个页能容纳搜索键数可能很大，分裂/合并树节点的情况可能很少发生。
- 按索引键值检索一条记录，典型只需要2~3次磁盘I/O。

5.2.3 B树的效率与实用化

- ❖ **B+**树索引的优势
- ❖ **B+**树重复键问题及其处理
 - 当重复键很多时，可能会出现叶节点无法容纳具有给定键值所有记录项的情况。
 - 常用处理方法
 - 用溢出页来处理重复键值问题。
 - 把重复键按一般非重复键一样处理，这时重复键项将出现在一个或连续的多个页节点中。
 - 将rid值也作为搜索键的一个部分，这实际上相当于消除了重复键。

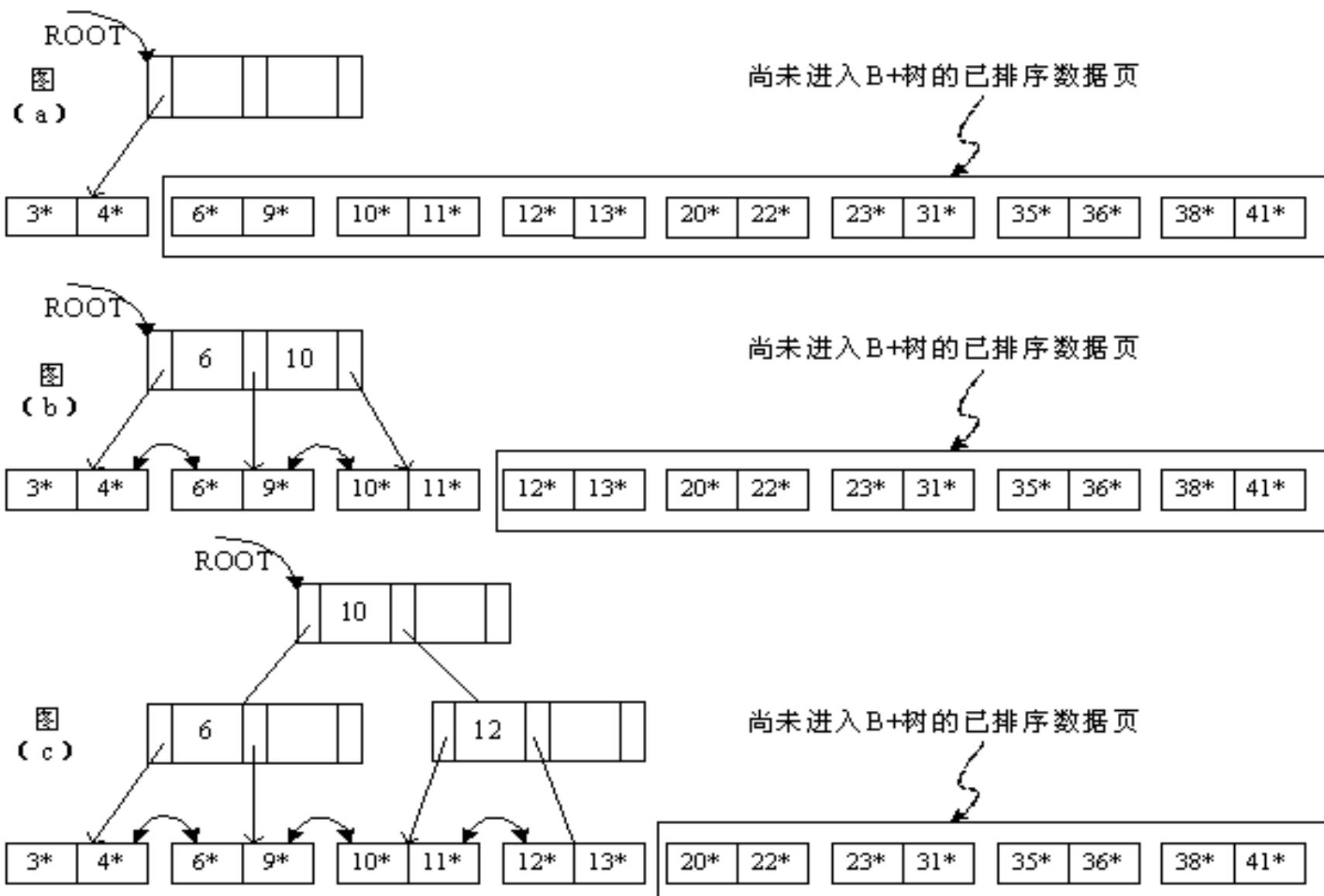
5.2.3 B树的效率与实用化

- ❖ **B+**树索引的优势
- ❖ **B+**树重复键问题及其处理
- ❖ 键值压缩处理（键压缩）
 - 如果搜索键值很长，页中能存储的索引项数就很少，树的宽度（fan-out）也就小。
 - 最大化fan-out以减小树的深度，对减少树操作磁盘I/O数非常重要。
 - 键值压缩原理
 - 只保留键的前缀。
 - 为确保能保持一个索引项中键值的比较语义，在压缩一个项时，除考虑它相邻项键值外，还要考虑左、右子树中的最大键值。

5.2.3 批量加载数据集到B+树

- ❖ 数据项加入到**B+树**索引可能会遇到两种情形：
 - 拟加入的数据记录集之前已建有B+树索引。这时，可利用标准的B+树插入算法，将数据项逐个加入数据集，同时更新相应的B+树索引。
 - 拟加入的数据集上还没有B+树索引。
- ❖ 对于后者，为减少操作代价，常采用批量加载方法
- ❖ 实现批量加载数据集到**B+树**的算法步
 - 参见教材，P159

图5.6 批量加载B+树过程演示



批量加载数据集到B+树的代价分析

这个操作算法可归纳为三大步骤：

- 第一步，从一个记录集创建要插入到索引的数据项；
 - 该步包括扫描关系记录集，并生成和写出相应的数据项。
 - 其代价为 $(R+E)$ 次 I/Os
 - R 是记录集数据文件的总页数， E 是包含数据项的总页数。
 - 第二步，排序数据项；
 - 外部排序含数据项的有 E 个页，保守估计需要 $3E$ 次 I/Os (参见 6.1 部分)。
 - 第三步，从排序好的数据项中建立 B+ 树索引。
 - 该步的代价只是写出所有索引页的代价。
- ❖ 总代价为： $R+4E+(B$ 树索引节点数 $)$

5.3 散列索引

5.3.1 静态散列存储表

5.3.2 可扩展的动态散列

5.3.3 线性散列表

散列存储技术概要

◆ 散列与散列函数

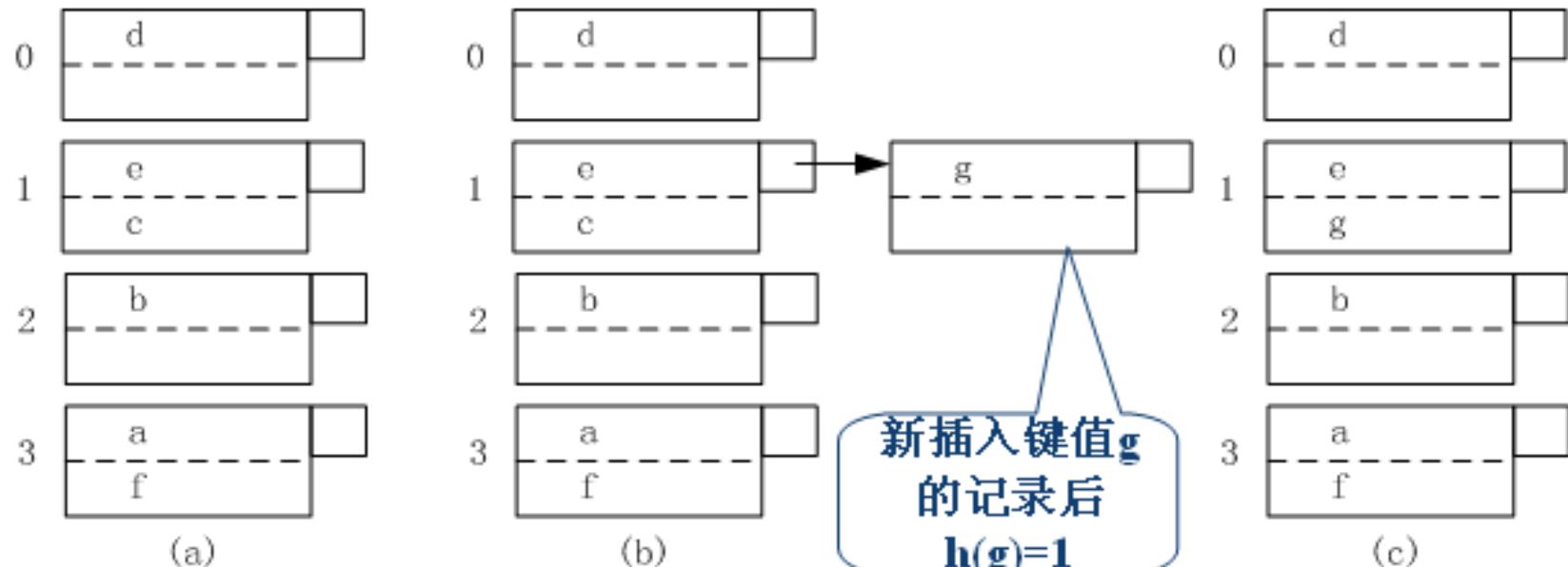
- 散列函数选择要求：随机分布好、易计算；
- 散列函数参数：查找键（搜索键）或散列键；
- 函数值：散列值（指定范围内的一个随机整数）

◆ 基于散列的存储结构

- 通常每个散列值对应一个存储桶（页/块）
 - 散列值对应桶编号（如果散列值范围是 $0 \sim B-1$ ，则桶总数为B）。
 - 根据被散列对象键值，计算散列值，然后保存到相应的桶中；
 - 当桶内对象不止一个时，按链连接起来，构成对象链。
- 存储到桶的对象，既可能是实际数据项或数据记录，也可以是数据记录指针；

5.4.1 静态散列存储表

例5.4 假定B=4，桶编号0~3；每个页只能存储两条记录。现有6个键值为字母a~f的记录，且有 $h(d)=0$, $h(c)=h(e)=1$, $h(b)=2$ 和 $h(a)=h(f)=3$ 。



初始时

新插入键值g
的记录后
 $h(g)=1$

删除键值c
的记录后

5.4.2 可扩展的动态散列（1）

- ❖ 静态散列一般通过增加溢出页来处理溢出问题。
 - 如不希望增加溢出页，也可修改散列函数，将桶的数目扩大（如扩大一倍），然后重组数据文件。
 - 但这种重组的代价可能很大：
 - 需要读入n个页，并写回 $2n$ 个页。
- ❖ 克服这个问题的一个方法
 - 一旦出现溢出桶，就分裂之；每次只分裂有溢出的桶。
 - 引入一个仅存储桶指针的**目录数组**，处理桶数扩展；用翻倍目录数组来取代翻倍数据桶数目；
 - 由于每个目录项只含有一个桶指针，目录所需存储页一般很少，这样翻倍的代价就很小。

5.4.2 可扩展的动态散列 (2)

- ❖ **H(索引键)=键值的二进制数，并取后d个比特位作为桶编码。**
- ❖ **d值是目录项的编码位数**
 - 目录项总数为 2^d 个； d值加1目录项数将增加一倍。
 - d值也称**全局位深度 (the global bit-depth)**。
 - 每个目录项中，含一个指向——数据桶——指针。
- ❖ **每个桶有一个局部位深度 (the local bit-depth)。**
 - 在局部位深度为l 的桶中，所存储的数据项的索引键值的后l位取值都相同。
 - 一般情况下，会有 2^{d-l} 个目录项指向这个桶；当l=d时，只有一个目录项指向这个桶。最初，所有桶都有l=d。

5.4.2 可扩展的动态散列（3）

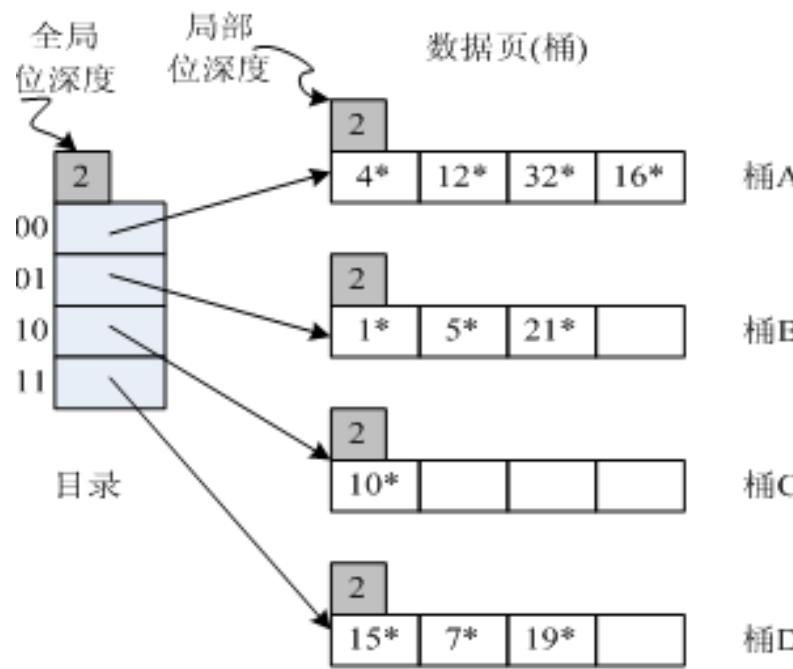
- ❖ 当向一个已满的、局部位深度为 ℓ 的桶插入数据项时，就要分裂这个桶。
 - 该桶及分裂产生的映像桶： $\ell \rightarrow \ell+1$ ；
 - 如果 $\ell+1 > d$ ，则还需要增加目录的编码位数，即要对目录数组——进行翻倍处理。
- ❖ 翻倍目录时，只需将原有的每个目录项分别复制产生**1个**对应的新目录项。
 - 一个目录项和由它复制产生的新目录项，互称为“对应元素”
 - “对应元素”开始时指向同一个桶，只是当桶分裂后，才分别指向原桶和原桶分裂映像。
- ❖ 可扩展散列存在的主要问题
 - 当散列值分布不均匀或偏斜很大时，会导致目录项数特别大和数据桶的空间利用率很低。
 - 每次目录调整都是翻倍调整，目录大小扩展过快，调整不平滑。

5.4.2 可扩展的动态散列 (4)

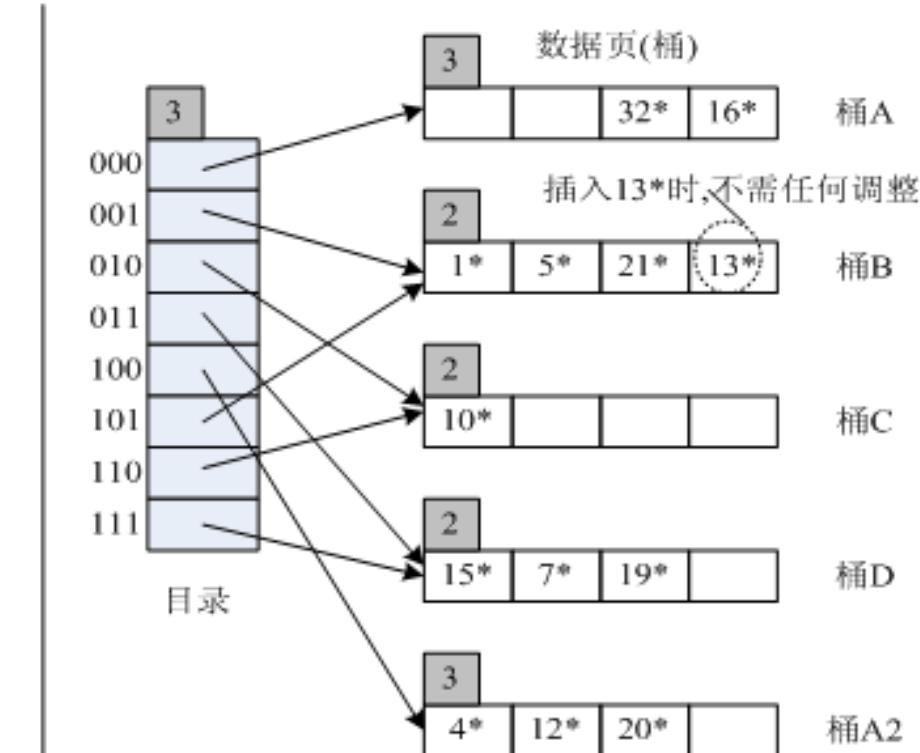
例5.5

$13=1101 \quad 20=10100$

- 每个目录项存1个指向桶的指针, 每个桶可存放4个数据项
- 初始时, 总共有4个目录项, $d=2$, $\ell=2$ 。



(a) 一个简单的扩展散列示例



(b) 在图(a)基础上, 插入数据项13*, 20*后的情况

5.4.2 可扩展的动态散列 (5)

算法 5.4 在可扩展散列中插入一个数据项

begin

 先定位数据项所属的桶。

 If 桶已满 then

 进行桶分裂

分配一个新桶，并将原桶中的数据项和新数据项，在原桶和分裂产生的映像桶中重新分布。调整目录项对应元素中的指针，使它们分别指向原桶和映像桶；

 将分裂桶及其映像桶的局部位深度 ℓ 增加 1；

 If $\ell >$ 全局位深度 d then

 全局位深度 d 增加 1；

 进行目录翻倍调整；

 Endif

Else

 将数据放入该桶中；

End if

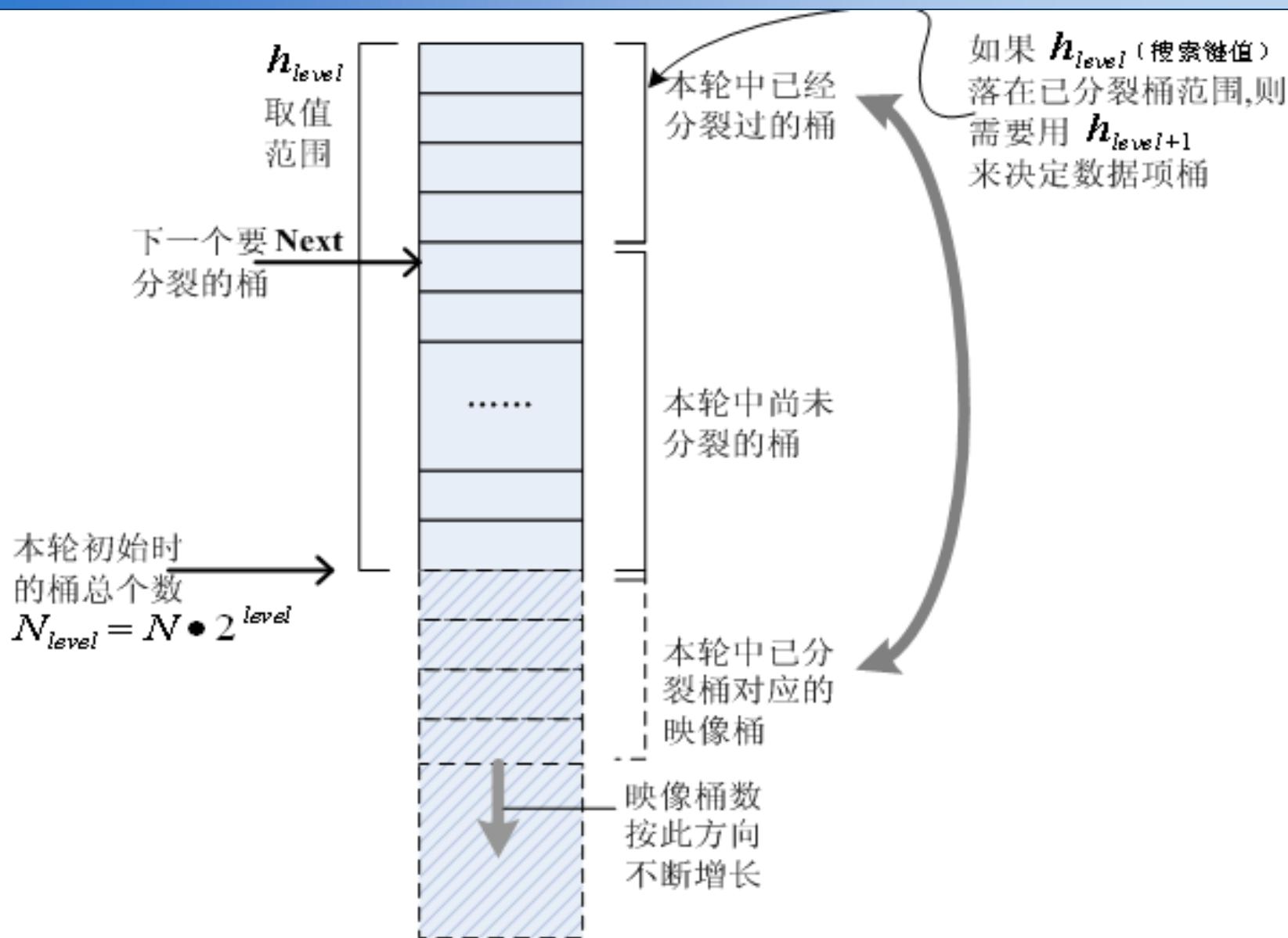
end begin

5.4.3 动态线性散列（1）

◆ 动态线性散列技术概要

- 也是一种动态散列存储技术，采用“**轮转分裂进化**”，适时地对存储桶的数目进行调整。
 - 每次桶分裂的触发条件，允许灵活选择（不一定非要选择——桶数据满——作为分裂该桶的条件）
 - 每次调整桶数——只增加1个桶。——平滑性
- 与可扩展散列相比，它不需要存放数据桶指针的辅助目录项数组。但需引入**索引元参数**：
 - 初始桶数 N_0 ——默认=4； 轮数level——初值=0；
 - 每轮进化指针Next——初值=0
- 以同等机会——**轮转分裂**每个数据桶
 - 每轮中，当每个桶都完成一次分裂，就进化到下一轮： $level \rightarrow level + 1$ ； 桶数 $(N_0 e^{level})$ 翻倍；

5.4.3 动态线性散列 (2)



5.4.3 动态线性散列 (4)

◆ 应用举例（例5.7）

- 假设：

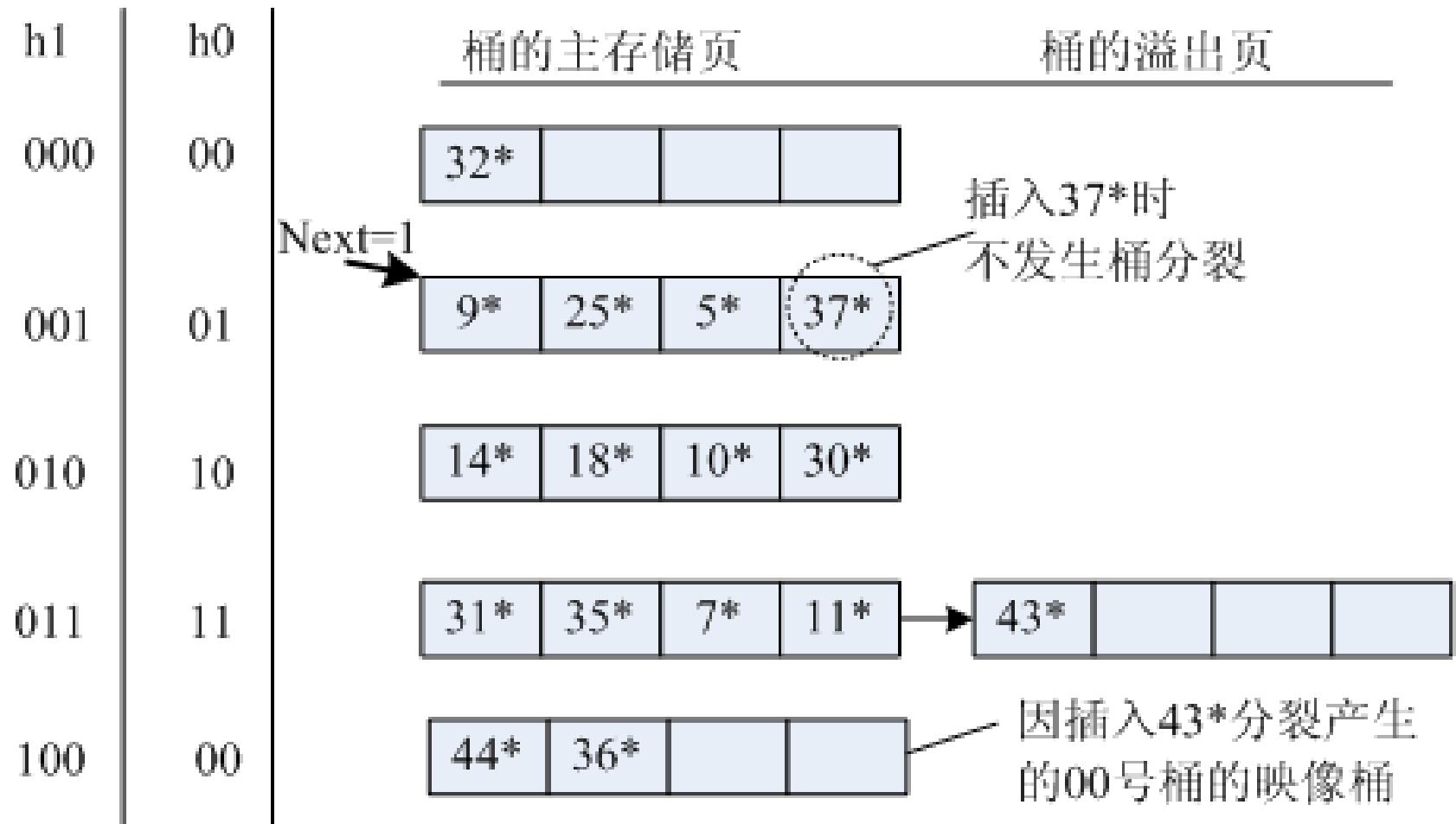
- 每个桶可存4个项，
初始桶数 $N_0=4$ ；
Level=0，
Next=0；
- 触发分裂条件：
“发生溢出插入”

仅用于介绍目的的信息		Level=0,N=4 数据文件桶的存储页			
h1	h0	000	00	32*	44*
			Next=0	36*	
001	01	9*	25*	5*	
010	10	14*	18*	10*	30*
011	11	31*	35*	7*	11*

- 分析插入新项 $h(r)=43(101011)$ 和 $h(r)=37(100101)$ 情况

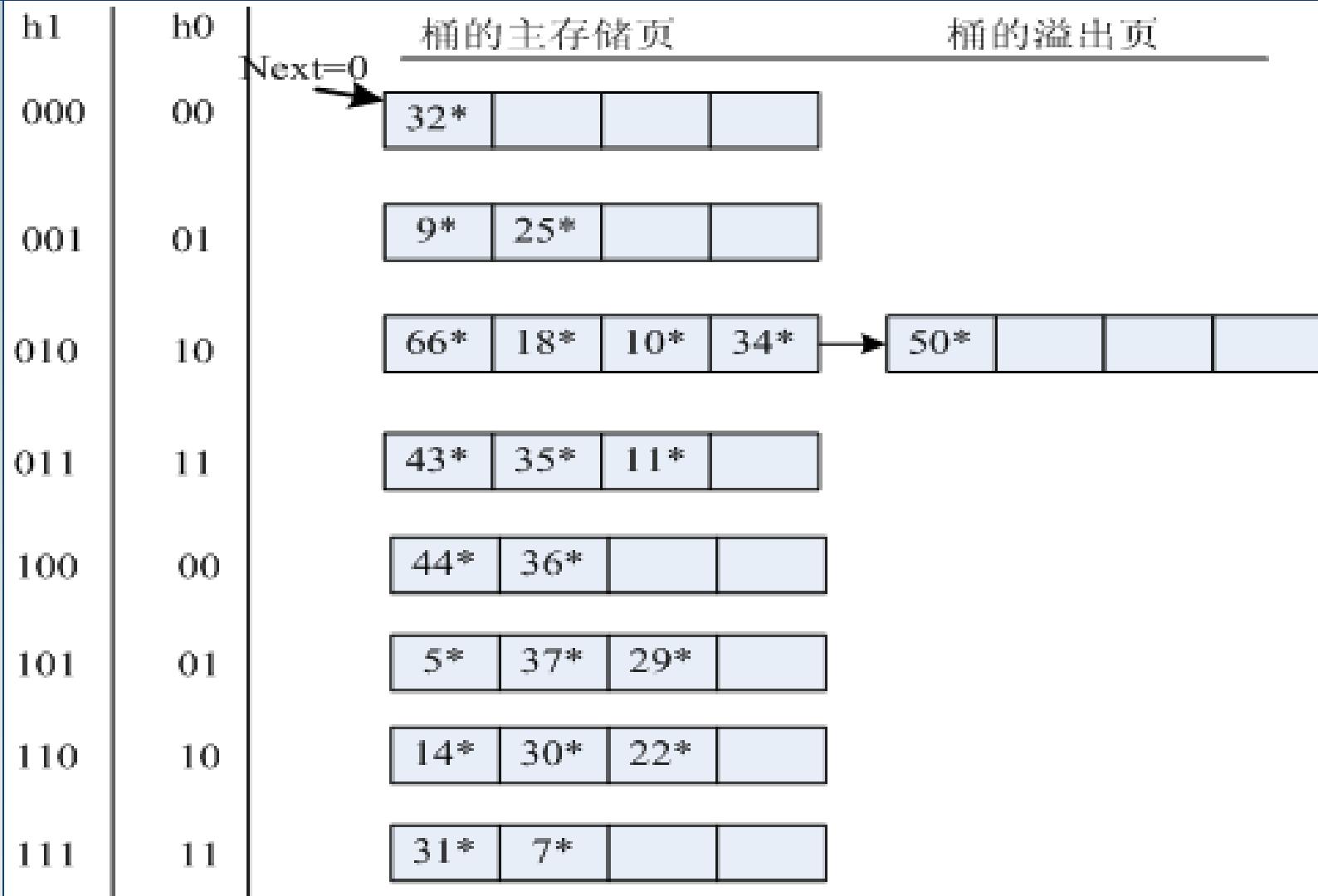
初始轮，level=0, N=4, $N_0=N*2^{Level}=4$; 桶位数 $d_0=\log_2 N$
桶位数 $d_0=2$; h_0 取 $h(k)$ 的最后2个bit位，作为桶索引指针；
桶位数 $d_1=d_0+1$, h_1 取 $h(k)$ 的最后3个二进制位。

例5.7 (续)



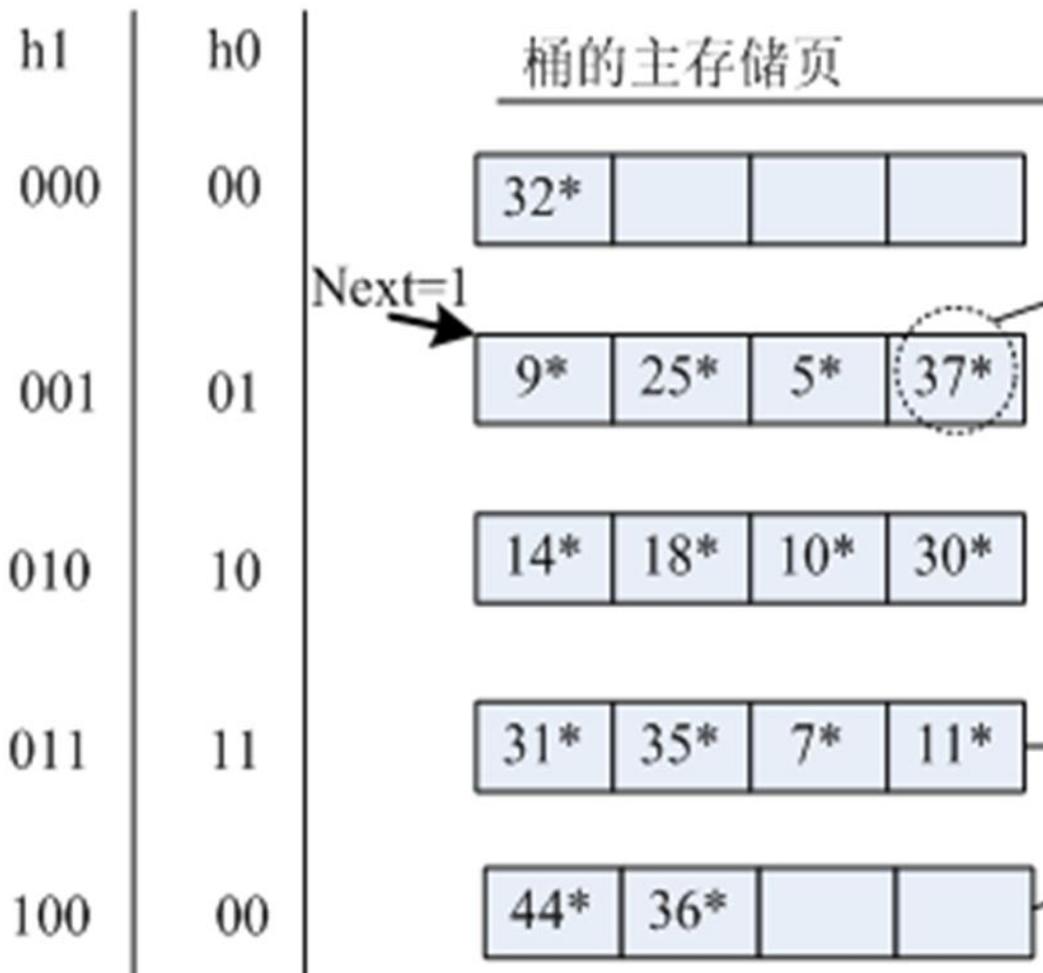
(b) 先插入 $h(r)=43=101011$, 再插入 $h(r)=37=100101$ 后的情况

例5.7 (续)



(c) 继续一次插入 $h(r)=29, 22, 66, 34, 50$ 后，将进化到下一轮

例5.8 线性散列查询应用



设level=0; Next=1

(1)查 $h(r)=18=10010$

∴ level=0,h0取后2位

h1取后3位

∴ 10在Next(=1) ~ 4

的未分裂区

∴ 直接查10获目标桶

(2)查 $h(r)=36=11100$

h0=00在0~ Next
的已分裂区

∴ 应查100获目标桶

5.5 位图索引

5.5.1 位图索引的结构

5.5.2 位图索引的应用

5.5.3 压缩位图

5.5.4 压缩位图的游程解码操作

5.5.5 *位图索引的维护

位图
索引

- ◆ 是一种主要针对多键查询设计的特殊类型索引，尽管每个位图索引都是建立在单键码之上。
- ◆ 为了使用位图索引，关系或数据文件中的记录必须进行顺序编号 $1, 2, \dots, n$ ，使得给定一个编号 i ，能方便检索到第 i 个数据记录。

5.5.1 位图索引的结构

❖ 考虑一个共有n个记录的关系R和它的一个属性A，假设R.A上只有m个的不同取值，

—— v_1, v_2, \dots, v_m 。

❖ 则R在A上的位图索引

- 是一组位图向量：
 - R.A的每个取值 v_j ($j=1..m$)，分别对应一个位图向量 $bmap_{v_j}$ ；
 - 共有m个这样的位图向量。
- $bmap_{v_j}$ 是一宽度为n的二进制数，该数的第i位值
 - $bmap_{v_j}[i] = 1$ ，如果第i条记录的R.A取值 = v_j ；
 - $bmap_{v_j}[i] = 0$ ，如果第i条记录的R.A取值 $\neq v_j$ ；

位图索引示例（图5.11）

Record number	name	gender	address	Income-level	gender 的位图	Income-level 的位图
0	John	m	Perryridge	L1	m 10010	L1 10000
1	Diana	f	Brooklyn	L2	f 01101	L2 01000
2	Mary	f	Jonestown	L3		L3 00100
3	Peter	m	Brooklyn	L4		L4 00010
4	Kathy	f	Perryridge	L5		L5 00001

5.5.2 位图索引的应用

❖ 例5.10 (续例5.9, 多键码检索应用示例)

1. 条件匹配查询

$\sigma_{\text{gender}=\text{f} \text{ and } \text{income-level}=\text{L2}} (\text{r})$

2. 范围查询

$\sigma_{\text{gender}=\text{f} \text{ and } \text{income-level}>\text{L2}} (\text{r})$

5.5.3 压缩位图

◆ 位图压缩的必要性

- 当用于建立位图索引的属性不同值数很多时，位图索引也可能会很大。

◆ 位图压缩技术的基本思想

- 对任何二进制数，我们总可以用 i_1 个0、 i_2 个0、 i_3 个0、... 来描述。由这个描述我们可以准确重构出这个二进制数。
- 但若直接对 i_1, i_2, \dots 进行二进制编码，就会出现问题。
 - 比如，对长度为7和4的两个位段，编码合为111100。
 - 我们无法唯一解码还原原来的两个位图向量。
- 解决这个问题的一个有效方案：
 - 在数值*i*的二进制编码前，加上一个前缀码。
 - 前缀码长度与值*i*的二进制编码长相同（令为 j , $j = \lceil \log_2 i \rceil = 4$ ），前缀码的取值模式为 $j-1$ 个1后跟一个0。
 - 对 $i \leq 1$ 特殊处理，规定： $i=0$ 的编码为00， $i=1$ 的编码为01。
- 例如，对 $i=13$: $j=\lceil \log_2 13 \rceil=4$ ，编码为1110,1101。

5.5.3 压缩位图

◆ 例5.11

- 对位图0001000,0000110000,0000,0001进行编码;
 - 1011,110111,00,11101011,
- 解压缩码11101101001011

◆ 位图压缩的效率分析:

- 一段长为 i 的码长为 $2 \log_2 i$,
一个 n 位向量, 平均编码长度约为 $2 \log_2 n$
- 因为最大向量个数 $m \leq n$, 故一个位图索引文件压缩后的长度上限约为: $2 n \log_2 n$
- 当 $n > 4$ 时, 恒有 $2 n \log_2 n < n^2$ 成立, 且 n 越大压缩效果越好。

5.5.4 压缩位图的游程解码操作

- ❖ 当要对两个压缩的位图向量运算时，必须先解码后运算。但不必先执行全部解码，可以结合运算的进行，一次一个段，即一次一个游程地交替解码。
- ❖ **例5.12**
 - 对两个压缩位图向量A:00110110(字段序列0, 5)和B:1101111101101(字段序列为7, 13)进行与运算，结果向量为C。

5.6 多维索引

5.6.1 多维空间索引技术综述

5.6.2 网格文件

5.6.3 R树

5.6.4 k-d树与四叉树

5.6.1 多维空间索引技术综述

1. 为什么需要多维空间索引

2. 空间数据及其查询的主要类型

点数据(point data)与区域数据(region data)

3. 需要多维空间索引的应用简介

4. 已建议的空间索引结构综述

5.6.1.1 为什么需要多维空间索引（1）

◆ 例5.13

- 设有一个存放购买金首饰顾客信息记录的关系表 Customers (age, salary)，假定在age和salary上分别建有B+索引。现考虑 $age < 30 \wedge salary < 2K$ 条件查询。
- 处理该查询的可选策略：
 - 1) 先基于Customers:age上的B+索引找出所有 $age < 30$ 的记录；再检查这些记录，进一步挑选出 $salary < 2K$ 的记录。
 - 2) 先基于Customers:salary上的B+索引找出所有 $salary < 2K$ 的记录；再检查这些记录，选出 $age < 30$ 的记录。
 - 3) 先根据两个B+索引，分别找出满足 $age < 30$ 和 $salary < 2K$ 的记录指针；然后，在内存中求这两组指针交集，并由交集指针找出所有目标记录。**利用位图索引可加快这种方法的交集操作。**
 - 基于组合键 $\langle age, salary \rangle$ 建立B+树索引（仍是一维的）

5.6.1.1 为什么需要多维空间索引（2）

P187 习题5.16 【解答】

分析：数据文件的总页数 = $1000,000/100 = 1$ 万

$R(x, y)$ 共2个维， x, y 对应维分别建有B+树索引

B+树索引的叶节点数 = $100,000/200 = 500$ ，其层数为 3 层

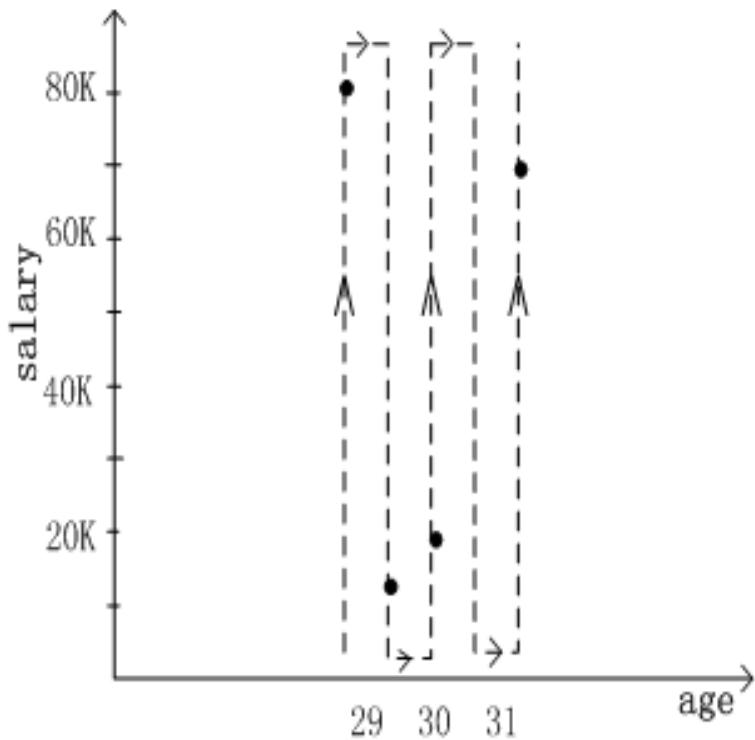
假设B+树的根保持在主存，则

- 1) 访问每个维的10万个指针，只需要检查一个B 树中间层结点，
以及包含所需指针的叶结点。最多 501次I/O
 \therefore 检索B+树，获得**1 万个记录指针**，大约需要进行1002 次的I/O
- 2) 但检索这1 万个目标记录指针对应的数据记录，
最坏情况下，可能还需要进行1 万次的I/O 操作。

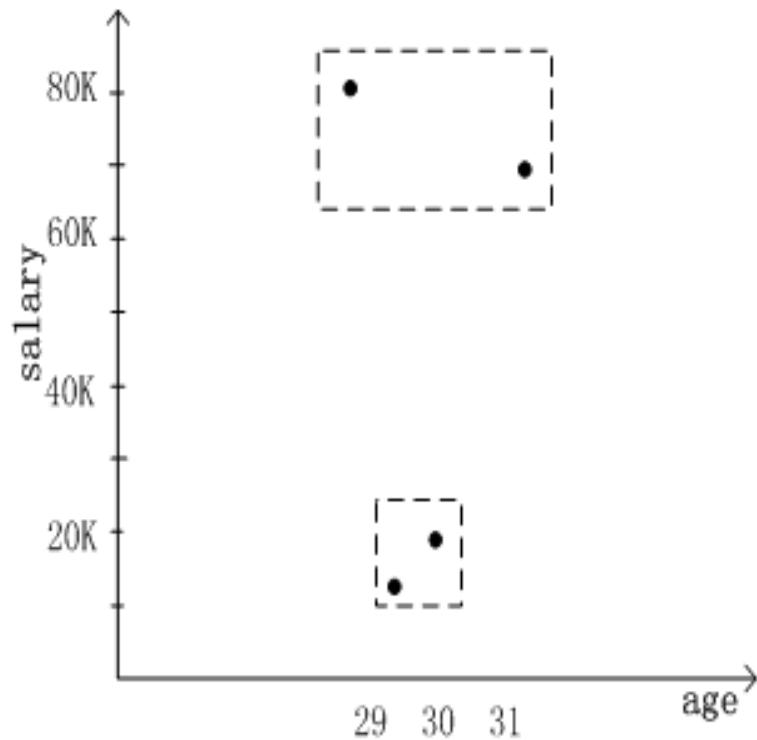
这通常已达到或超过数据文件本身的所有页！

5.6.1.1 为什么需要多维空间索引（3）

- 与B+树相比，多维或空间索引则往往利用某种空间关系（在空间中的相近性或邻近性）来组织数据项，如图5.12（b）所示。



(a) 基于组合键<age, salary>B+树聚集索引时
数据项的聚集方式



(b) 二维空间索引下数据项的聚集方式

5.6.1.2 空间数据及其查询的主要类型

(一) 空间数据的主要类型

(1) 空间点数据

- 基本特点

- 只有位置，没有大小、边界，不占空间。

- 常见点数据类型

- 光栅数据 (raster data)。
 - 多维特征向量 (feature vector)。

(2) 区域数据

- 同时具有位置和边界的空间延展。

- 存储在DB中的区域数据，通常是一些用来逼近实际对象形体的系列简单几何体。

空间查询-----的主要类型

❖ 范围查询(**range queries**)

- 空间范围查询通常关联着一个区域，并要求返回与目标区域范围重叠(overlap)或位于目标区域内的、指定类型的所有区域对象。

❖ 最邻近点查询(**nearest-neighbor queries**)

- 要求找出离指定点最近的某些对象。例如，查询“找距某位置最近的10个城市”。
- 在多媒体数据处理中，这种查询显得特别重要。

❖ 空间连接查询(**spatial join queries**)

- 典型例子包括“找相互间距离不超过200公里的城市组对”，“找靠近某区域（如一个湖泊）的所有城市”

❖ 部分(维)查询

- 只指定部分维的值，查找匹配这些维值的所有对象；

基于范围查询的最邻近点查询实现算法

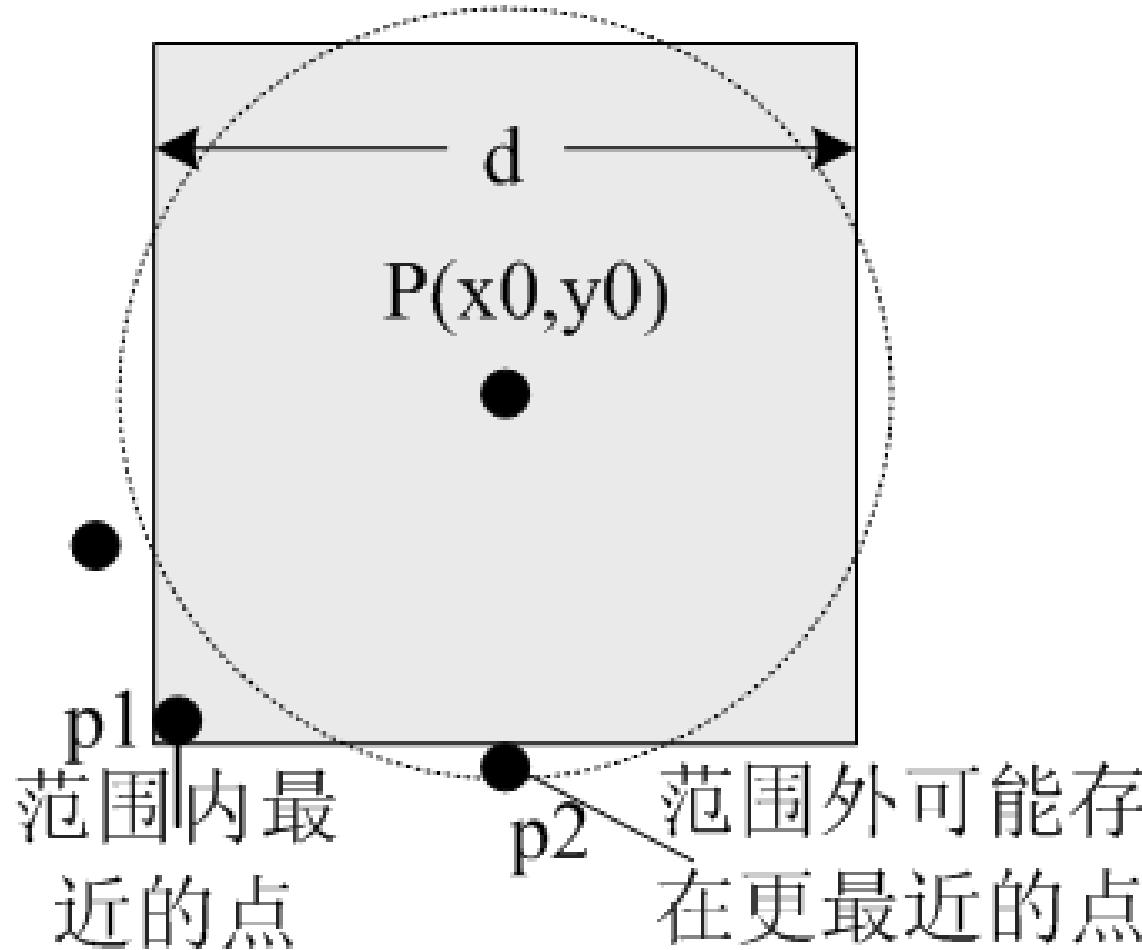


图5.13 处理最邻近点查询的示意图解

5.6.1.3 需要多维空间索引的应用简介



(一) 数据仓库的数据立方体



(二) 地理信息系统(GIS)



(三) CAD/CAM 系统



(四) 多媒体信息处理

5.6.1.3 (一)数据仓库的数据立方体

- 在数据仓库中，通常需要建立一种称为“数据立方体”的多维数据结构，以更好支持决策分析。

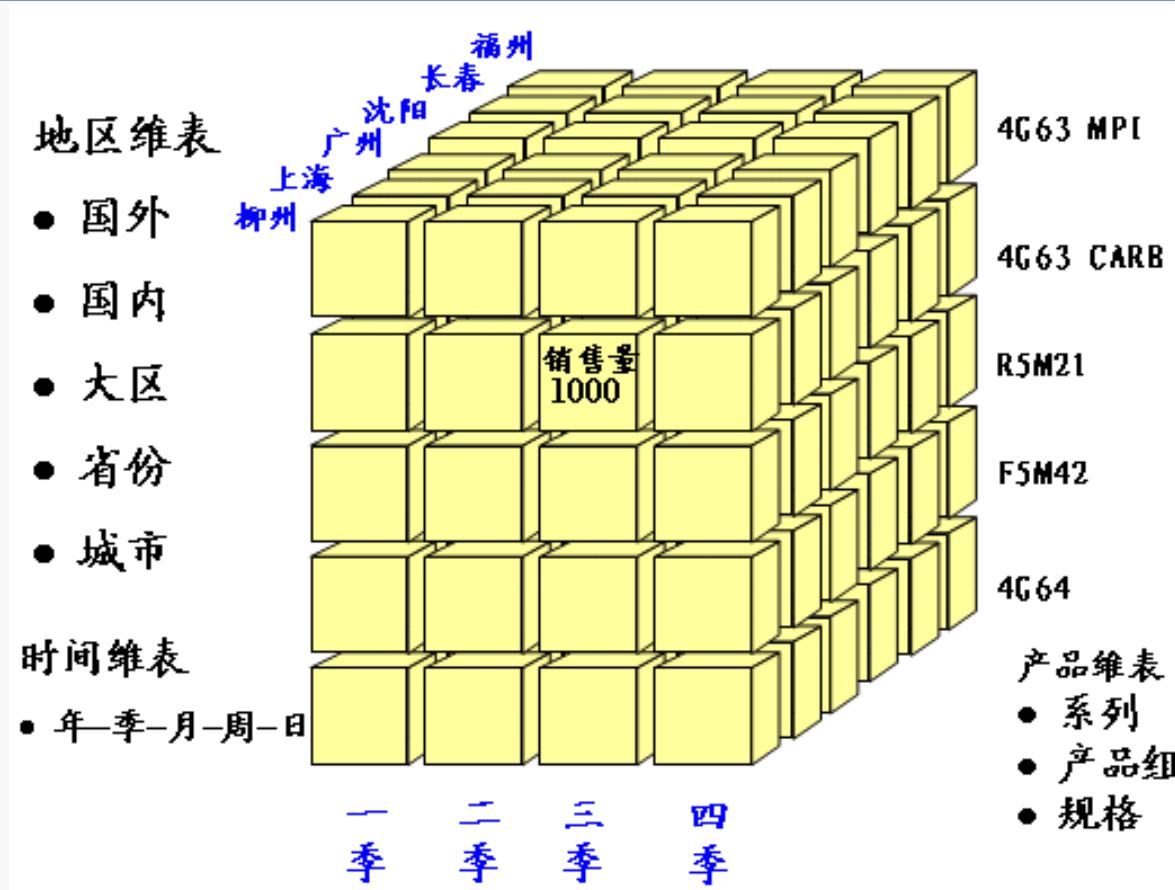


图5.14

5.6.1.3 (二)地理信息系统(GIS)

- ❖ **GIS**被广泛用来处理各种空间数据，包括点、线、二维/三维-区域。
 - 例如，一幅地图中，可能同时包含小目标（点）、河流/公路（线），以及城市/湖泊（区域）等。
- ❖ **GIS**能自然提出所有空间查询类型，它必须能有效管理二维、三维数据集，必须能有效处理空间点数据和区域数据。
- ❖ 当前许多对象数据库系统的都已能很好支持常见的**GIS**类应用。

5.6.1.3 (三)CAD/CAM系统

- ❖ 这类系统中通常要存储和处理大量的空间对象。类似**GIS**, 这类系统中也必须存储和处理空间点 / 区域数据。
- ❖ 范围查询和空间连接查询可能是这类应用中最常见的查询。
- ❖ **CAM/CAD**也是对象数据库系统发展的一个主要动因。

5.6.1.3 (四)多媒体信息处理

- ❖ 多媒体涵盖诸如图像、文本和各种类型时间序列数据（音频/视频）等各类对象，也需要空间管理方式。
- ❖ 在多媒体数据库(***multimedia databases***)中，使用象“查找与特定对象相似的所有对象”这类相似查询可能极为普遍。
- ❖ 回答相似查询的一个通行方法是首先映射/变换多媒体对象到特征向量点，将查找相似对象问题转换为关于特征向量点集的最小邻近点查询问题。

基于内容的图像检索技术

◆ 医疗/生物图像数据库

- 可能要存储大量数字化的二维/三维图像，如X-射线或MRI图像，形成相对完整的、涵盖各种案例的样本图像库；
- 可基于图像相似匹配技术，处理新采集图像的模式识别问题。
- ◆ 基于指纹数据库，进行给定指纹的匹配搜索，处理指纹识别问题。
- ◆ 基于人脸图像数据库，进行给定人脸的匹配搜索。
- ◆ 视频剪辑数据库。在视频DB中，搜索有场景变化的特别帧，或搜索包含特别对象的视频帧序列，来跟踪处理运动对象。
- ◆ 存储文本文档集，并处理“在文档集中搜索包含相似主题文档”等有关问题。

以上应用,本质上都要处理相似图像的匹配/识别问题。

5.6.1.4 已建议的空间索引结构综述

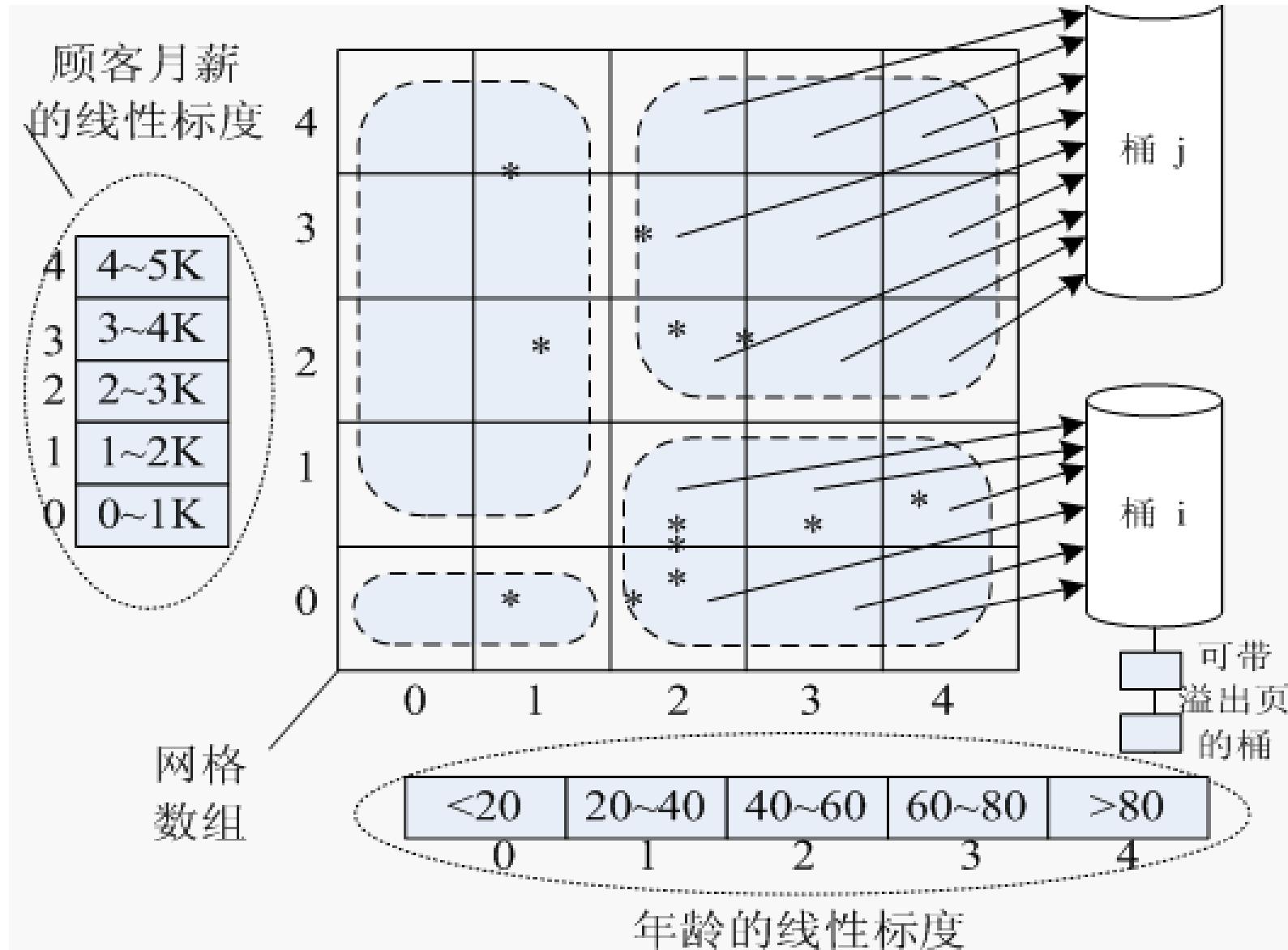
- ❖ 目前已提出的主要多空间索引结构
 - 有些索引结构主要是为满足**空间数据点**检索需要而设计，这类多维索引主要有：
 - 网格文件、kd树、**点四叉树**(point quad trees)和SR树等。
 - 有些能自然处理**区域数据**的索引结构，例如，
 - 四叉树(region quad trees)、R树和SKD树等。

5.6.2 网格索引结构（1）

例5.14 设有一个存放顾客购买金首饰记录的关系表 (age, salary)。为使问题简化，我们假设该关系只有顾客年龄和月薪两个属性。

---实例数据中有12个顾客，相关记录被表示成下列的年龄-薪水对： (26, 0. 6) (45, 0. 6)
(51, 0. 75) (51, 1) (51, 1. 28) (70, 1. 30)
(85, 1. 4) (30, 2. 6) (26, 4. 0)
(45, 3. 5) (51, 2. 75) (60, 2. 6)

5.6.2 网格索引结构（2）



5.6.2 网格索引结构（3）

- ❖ 用平行于坐标轴的分割线来划分空间到条形区域
 - 分割线相对于轴的位置不要求均匀；二维网格必须交替增加两个维度的分割线。
- ❖ 每个条形单元(**Cell**)含有一个指向桶的指针，每个桶可以是一个页或页组，桶中直接存放记录。
 - 为了节省空间，网格的多个单元可以指向同一个桶。
- ❖ 网格文件的插入算法
 - 举例：在图5.16网格中，插入记录(70, 3.5K)。
- ❖ 网格文件对多维查询支持及性能
 - 对指定点的查找，若无溢出块页，仅需1次I/O；
 - 部分匹配：可能需要查找桶矩阵的某行或某列的所有桶，I/O数可能很大；
 - 范围查询：检查与范围区域有相交的所有桶；

5.6.3 R树

- ❖ R-树也是一种平衡树结构，其中被索引的多边形存储在叶节点上（这一点类似 B+树）。
- ❖ 每个树节点（叶节点 / 内节点）都对应有一个平行于坐标轴的矩形边界框。
- ❖ 叶节点
 - 负责存储位于其内的所有被索引多边形，**边界框**是一个能涵盖其内所有存储对象的最小矩形。
- ❖ （含m个子节点的）内节点，存储：
 - 子节点 j 的边界框，及指向 j 子树的指针
 - $j = 1, \dots, m$ 。
- ❖ R树的基本操作：查找、删除和插入

R树的两种视图

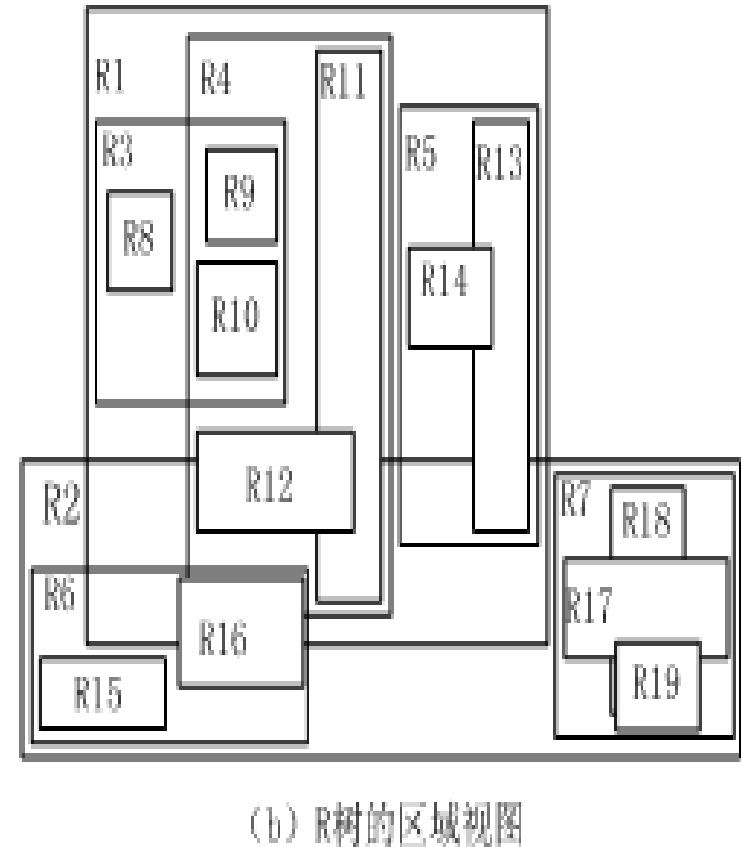
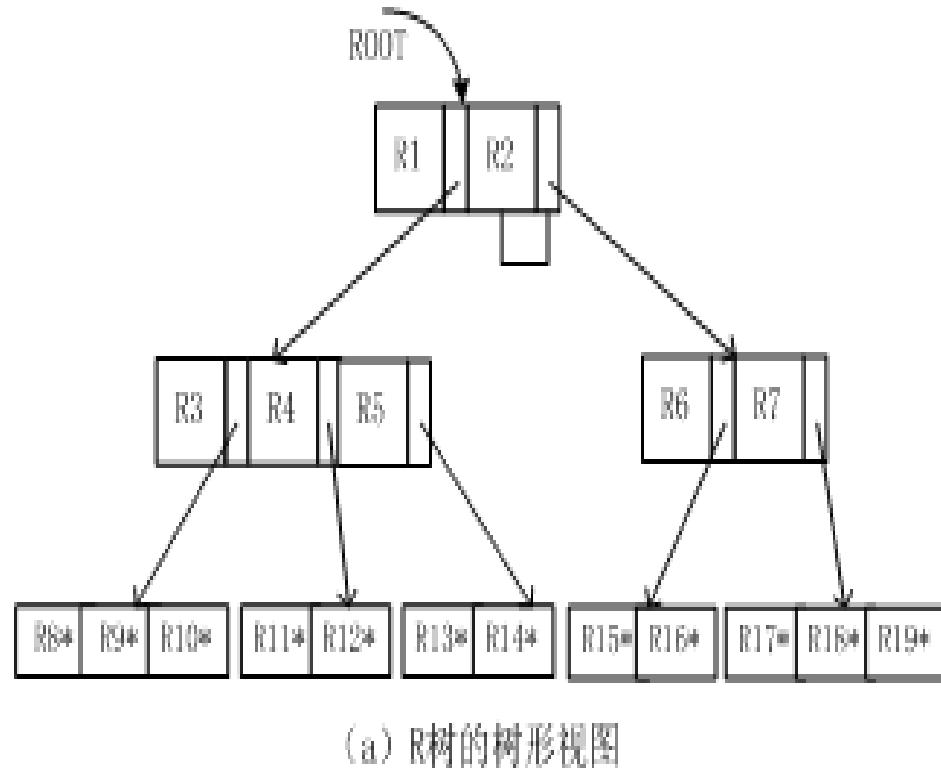


图 5.18 R 树的两种视图

5.6.3 R树-----插入算法

算法 5.6 R 树插入

Proc Insert(多边形 P)

 set TP-border= getBorder(多边形 P);

 set C=根节点

 while C 不是叶节点 do begin

 if 存在边界框包含 TP-border 的子节点 then

 从中选择一个;

 else

 选择一个其边界框与 TP-border 重叠最大的子节点 N;

 set C=N;

 end if;

 end while;

 //到达叶节点

 if 叶节点内有空间 then

 在叶节点内插入新对象;

 else

 类似 B 树，分裂叶节点，必要的话，可能还要向上传播；

 end if

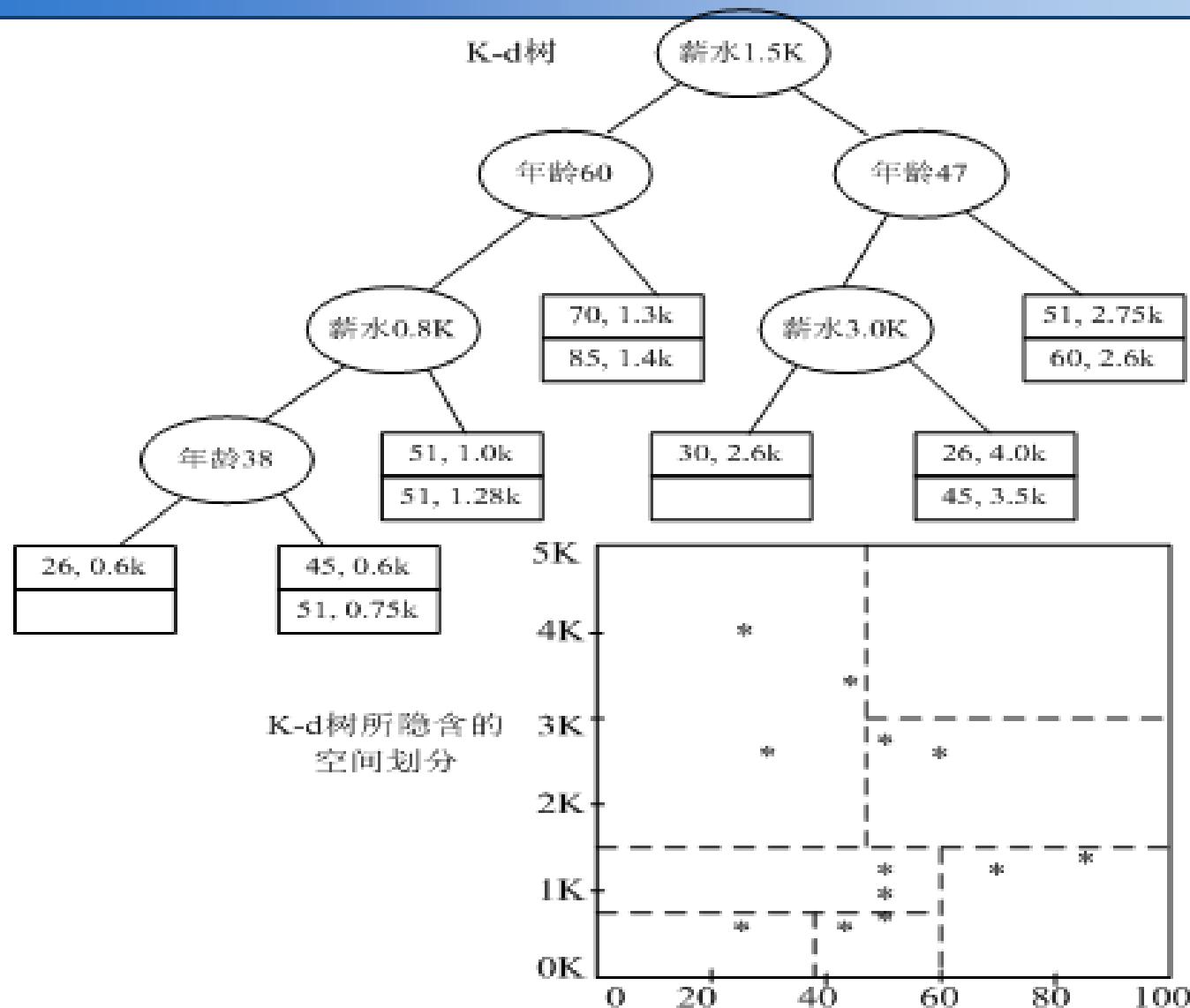
 调整叶节点边界框，同时回溯调整上层各节点的边界框；

end proc

5.6.4 KD树

- ❖ 考虑一维数据点的索引：
 - 树结构（如二叉树或B-树）都是通过不断划分来分割空间的。
- ❖ **k-d树(k维搜索树)**是早期用来建立多维空间索引的一种简单树形结构。
 - 每个节点将一个空间划分为两个子空间。
 - 划分首先通过树顶层节点的一个维进行，然后在紧接的下一个层节点上用另一个维进行划分，以此类推。
- ❖ **k-d树**数据结构特点（参见书本**P181-182**）

5.6.4 KD树---示例



5.6.4 KD树---对多维查询支持

❖ 部分匹配查询

- 如果给定某属性的值，那么，当处在恰好是按该属性划分的层结点时，只需考察一个方向的子结点；否则要考察两个方向的子结点。

❖ 范围查询

- 如范围跨越结点划分值时，需考察两个方向的子结点树。

❖ 最邻近查询

- 可通过多次的范围查询来实现。

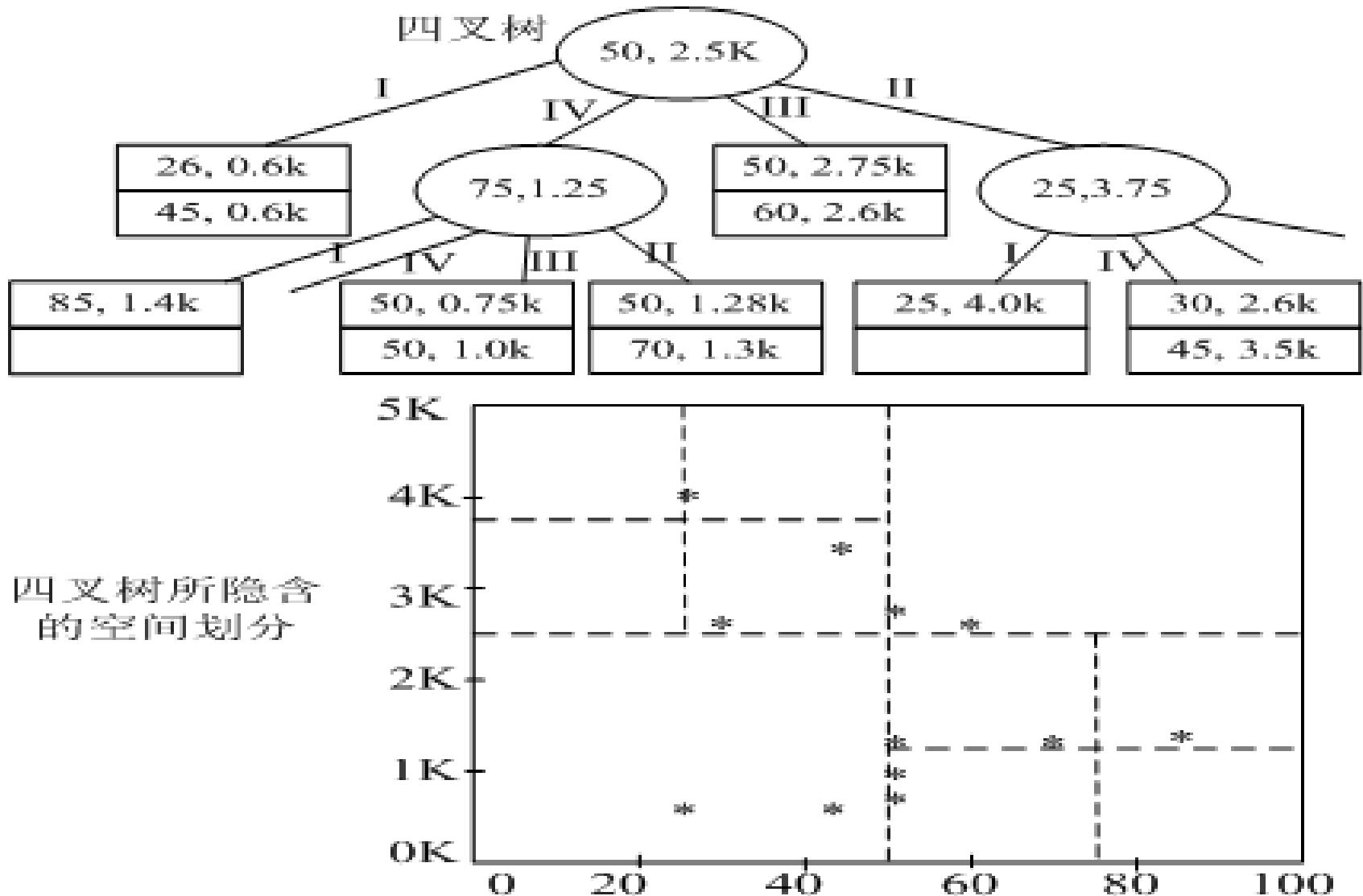
❖ 主要存在问题

- 每结点占用1个页，空间利用率低；
- 查询路径可能会很长；

解决方案：

- 采用多分支内结点（但这已不是k-d树了）；
- 聚集多个内结点到一个页

5.6.5 四叉树



本章作业

❖ 5.2

❖ 5.7

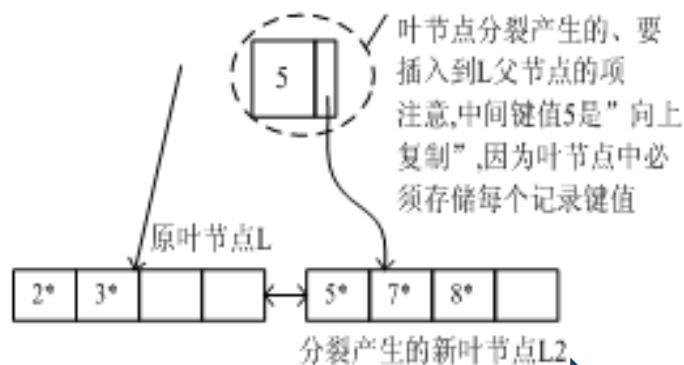
❖ 5.13

❖ 补充题：

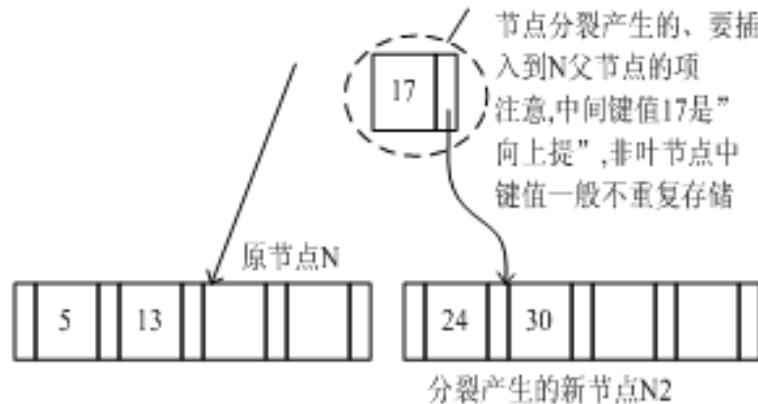
对图5.23所示的那组数据，给出基于线性散列的存储组织结构图，并回答习题5.7中的(4) (5) (6) 三个子问题。

插入算法应用示例演示

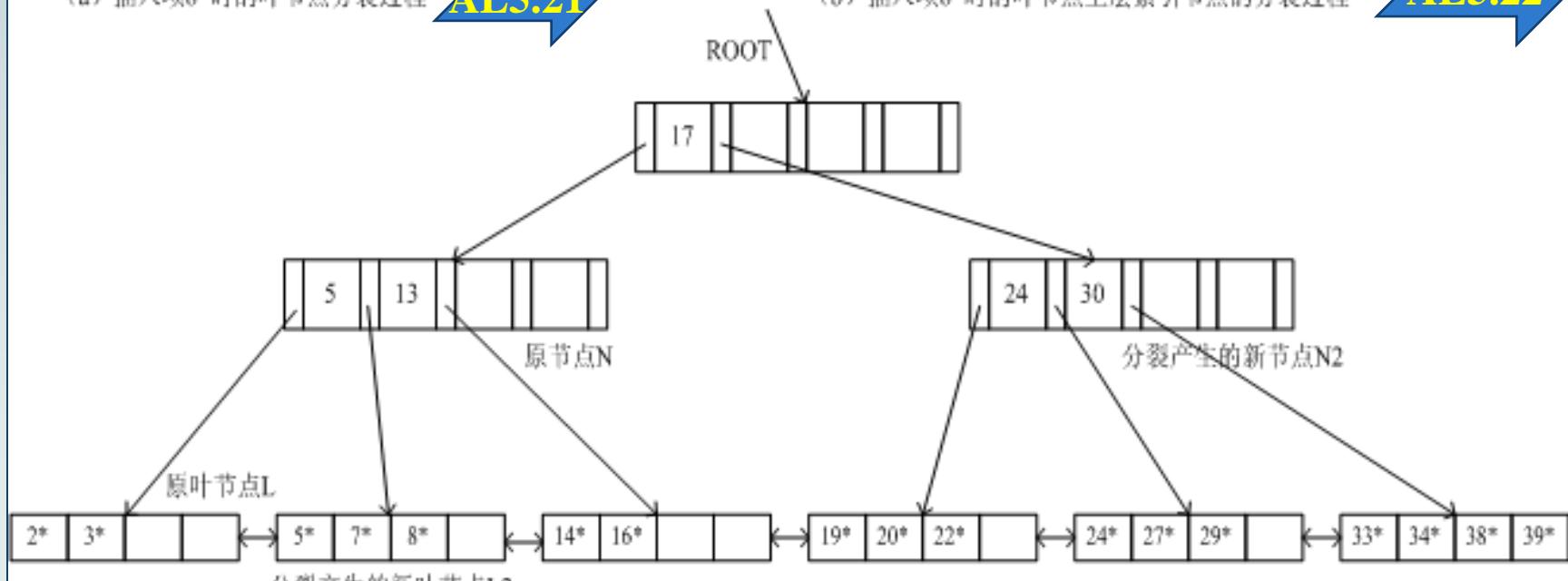
删除算法图解



(a) 插入项8*时的叶节点分裂过程 **AL5.21**

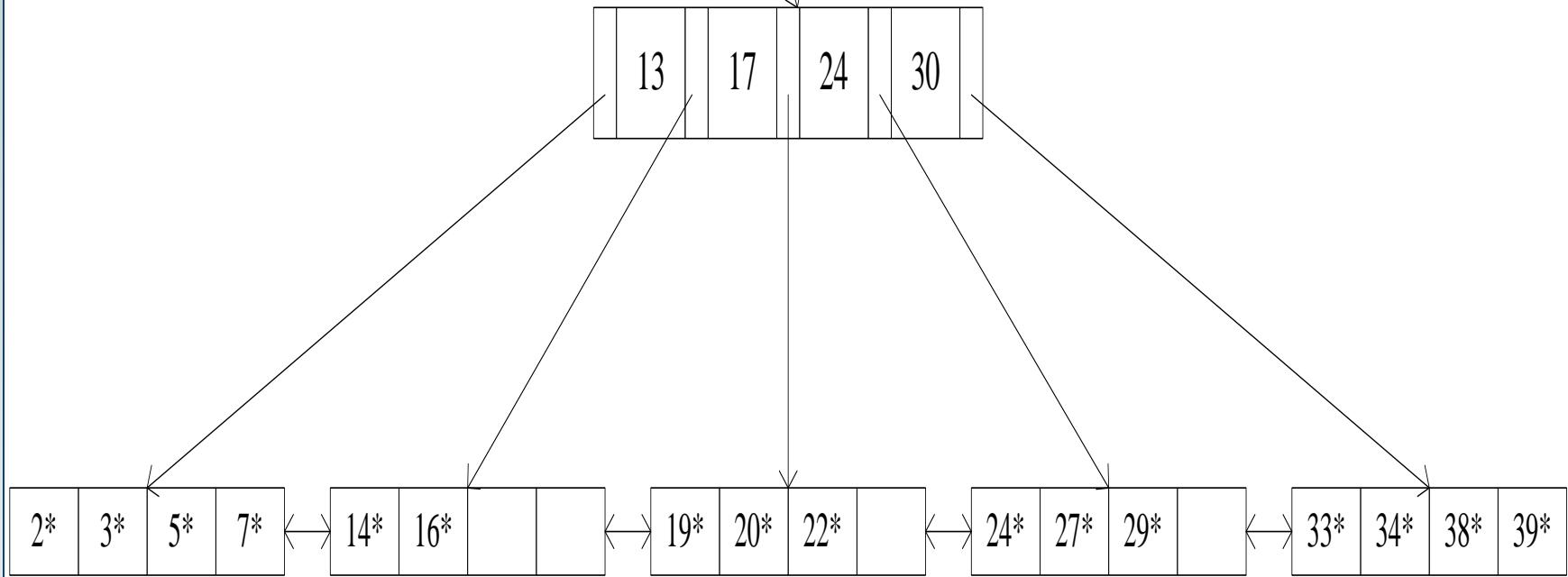


(b) 插入项8*时的叶节点上层索引节点的分裂过程 **AL5.22**



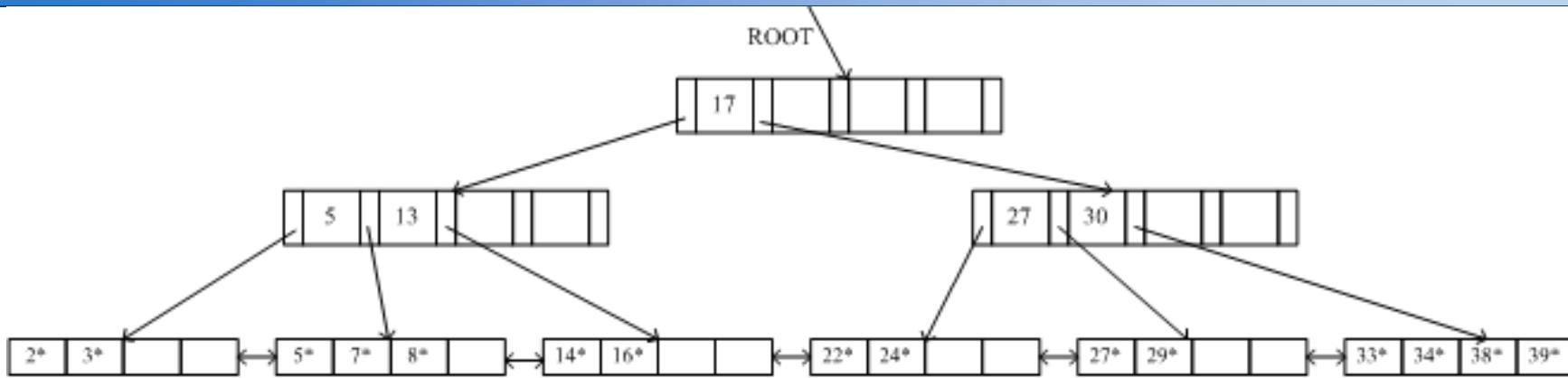
(c) 插入项8*后的完整B+树结构

ROOT



删除算法应用示例演示

插入算法图解



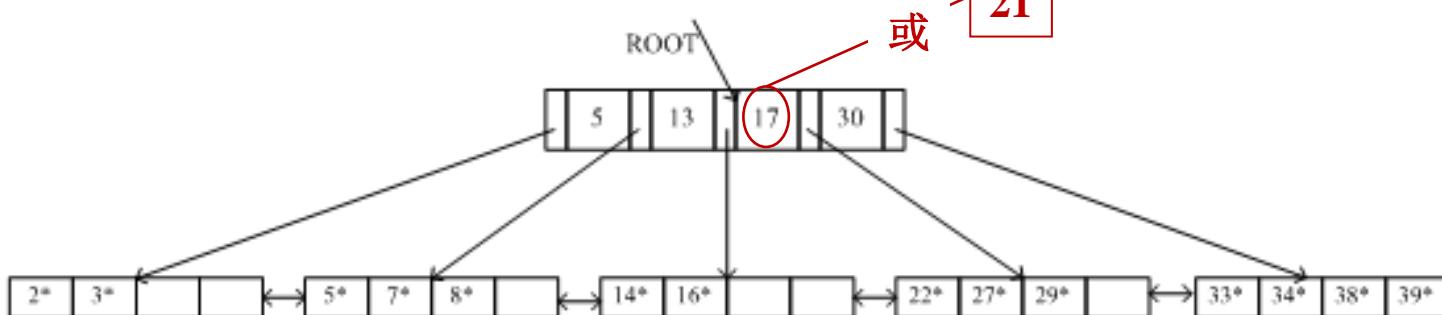
(a) 删除19*,20*后的B+树结构



AL5.3.1

AL5.3.2

(b) 删除项24*后的部分B+树结构



或
21

(c) 删除项24*并全部调整完成后的完整B+树结构