

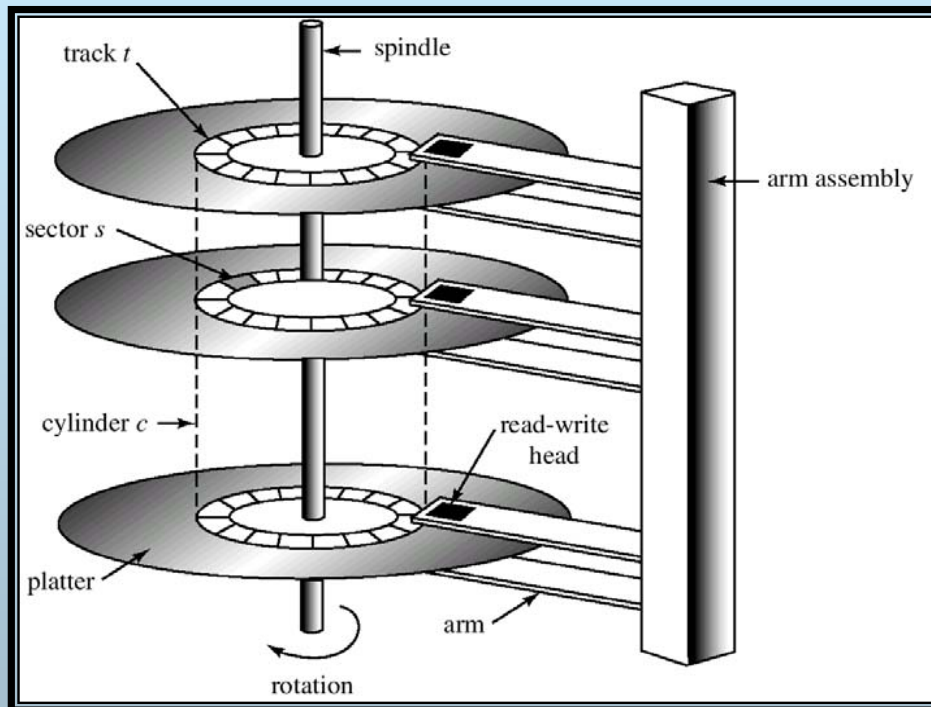
第8章 数据库索引



主要内容

- 顺序文件上的索引
- 辅助索引
- **B+树**
- **散列表 (Hash Tables)**

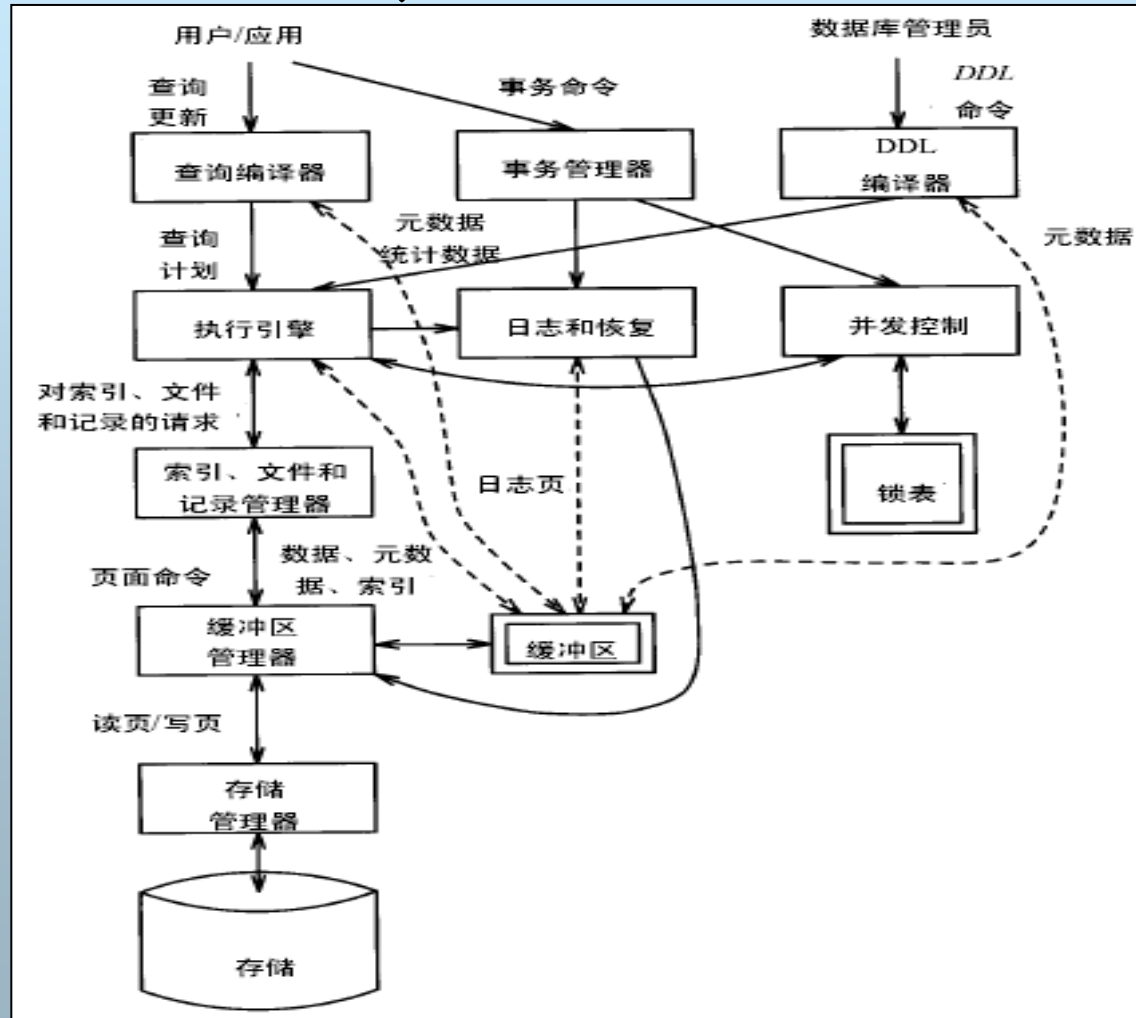
为什么DBMS底层需要按页存取？



- 一次磁盘访问的延迟包括：
 - 寻道时间 (10-40ms)
 - 旋转延迟 ($\sim 4\text{ms}$, 7200RPM)
 - 传输时间 ($< 1\text{ms}$)
 - 其它延迟 (忽略不计)
- 磁盘最小物理存取单位
 - 扇区 (512B)
- DBMS和OS的逻辑存取单位
 - 页 (磁盘块) —— 若干连续扇区
 - 16KB——MySQL
 - 8KB——MS SQL Server
- 减少数据存取时的寻道时间

为什么数据库需要索引?

■ 没有索引，数据查询效率低



若 page size = 8 KB, page I/O 10ms

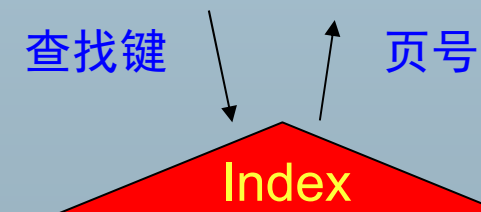
1 MB (128 pages): 1.28 s

128 MB (16384 pages): 163.8 s

1 GB (131072 pages): 1310.7 s
≈21.8 min

索引的动机:

提高按查找键 (Search Key) 查找的性能, 将记录请求快速定位到页地址

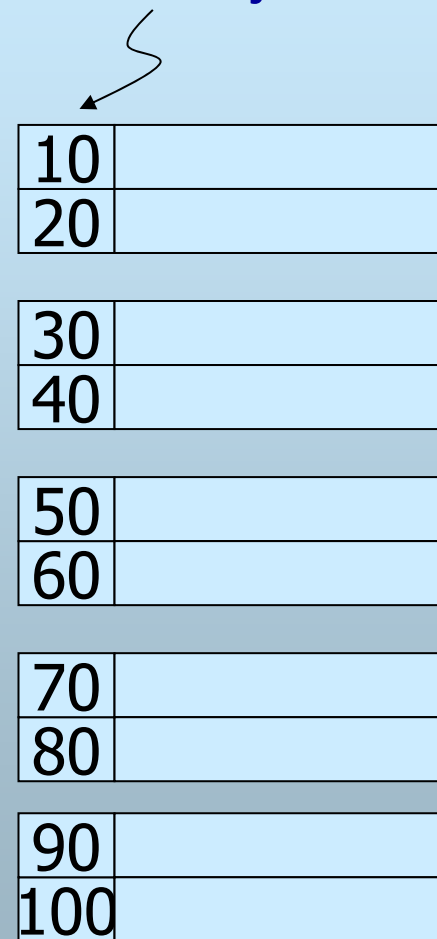


一、顺序文件上的索引

■ 顺序文件

- 记录按查找键排序

Search key

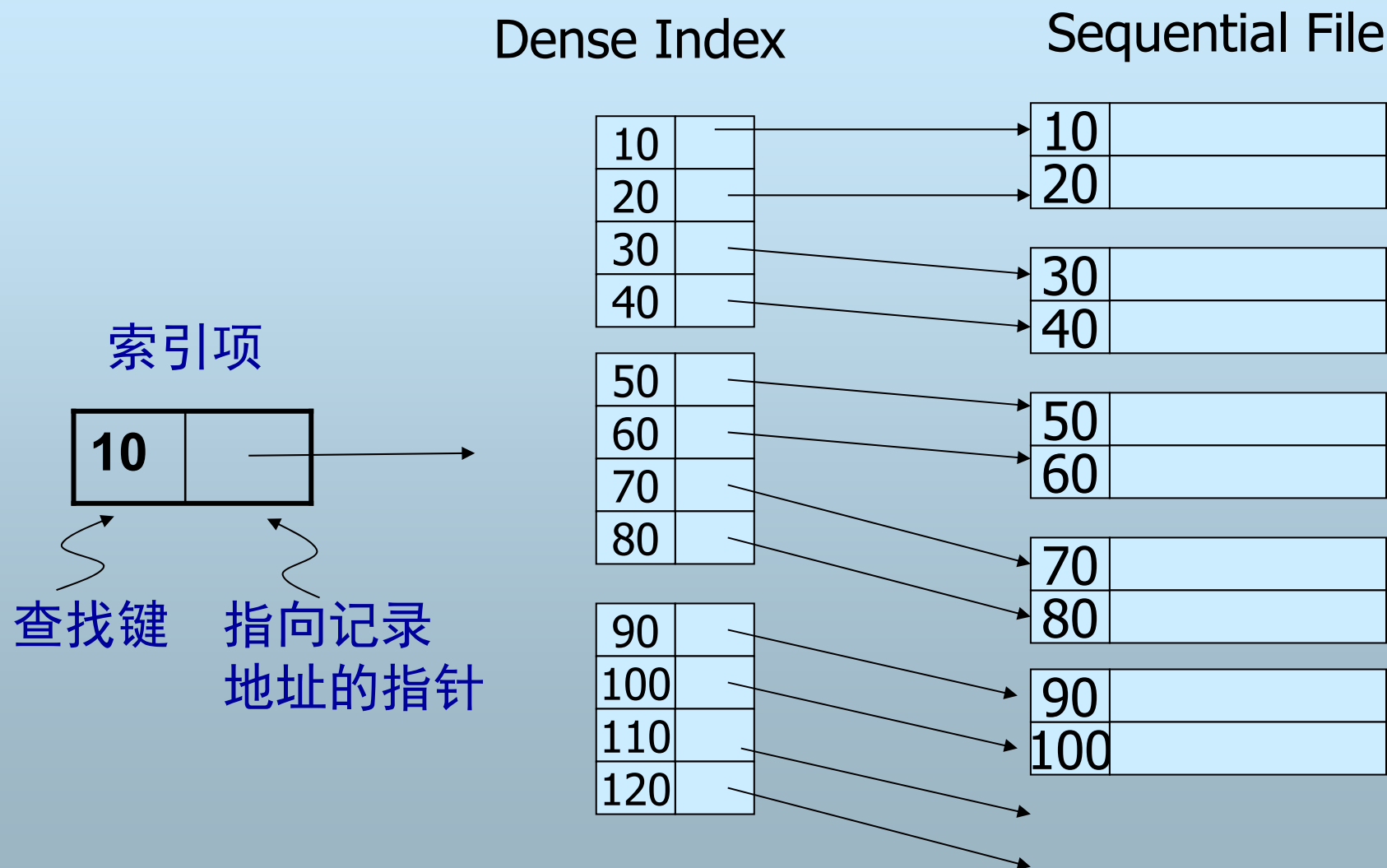


10	
20	
30	
40	
50	
60	
70	
80	
90	
100	

1、密集索引(Dense Index)

- 每个记录都有一个索引项
- 索引项按查找键排序

1、密集索引(Dense Index)

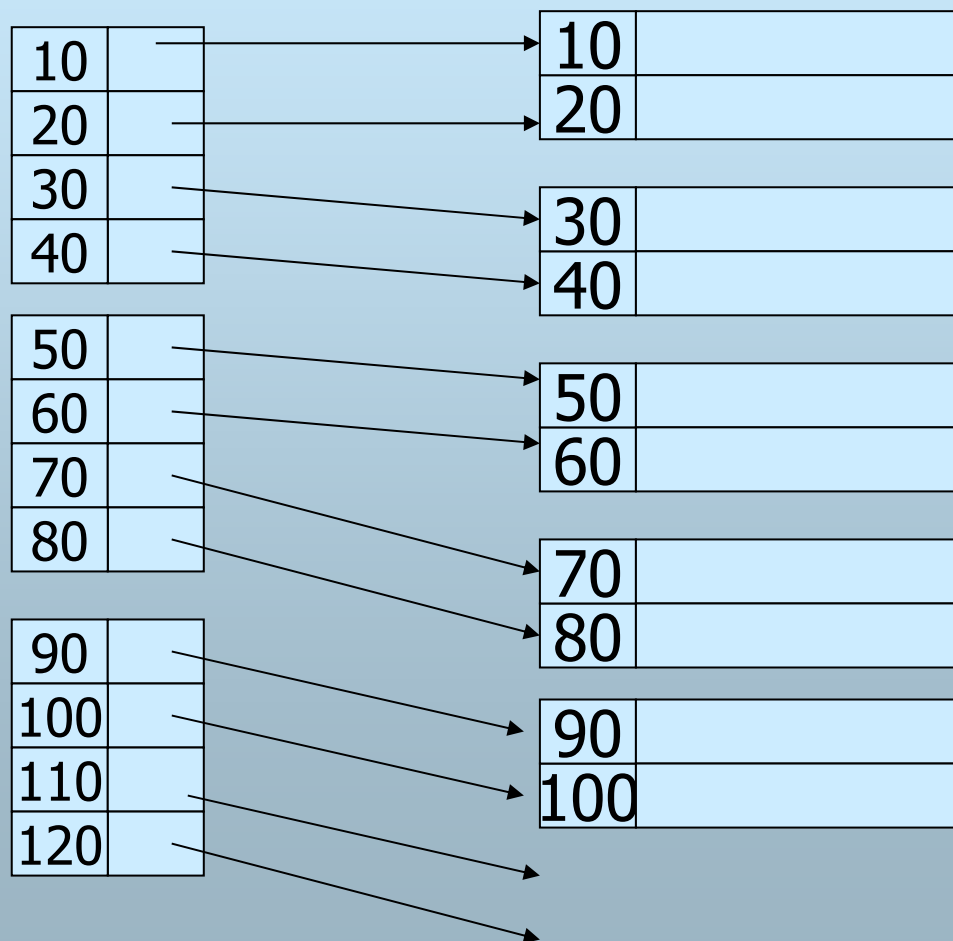


1、密集索引(Dense Index)

查找：
查找索引项，
跟踪指针即可

Dense Index

Sequential File



1、密集索引(Dense Index)

■ 为什么使用密集索引?

- 记录通常比索引项要大
- 索引可以常驻内存
- 要查找键值为K的记录是否存在，不需要访问磁盘数据块

■ 密集索引缺点?

- 索引占用太多空间



用稀疏索引改进

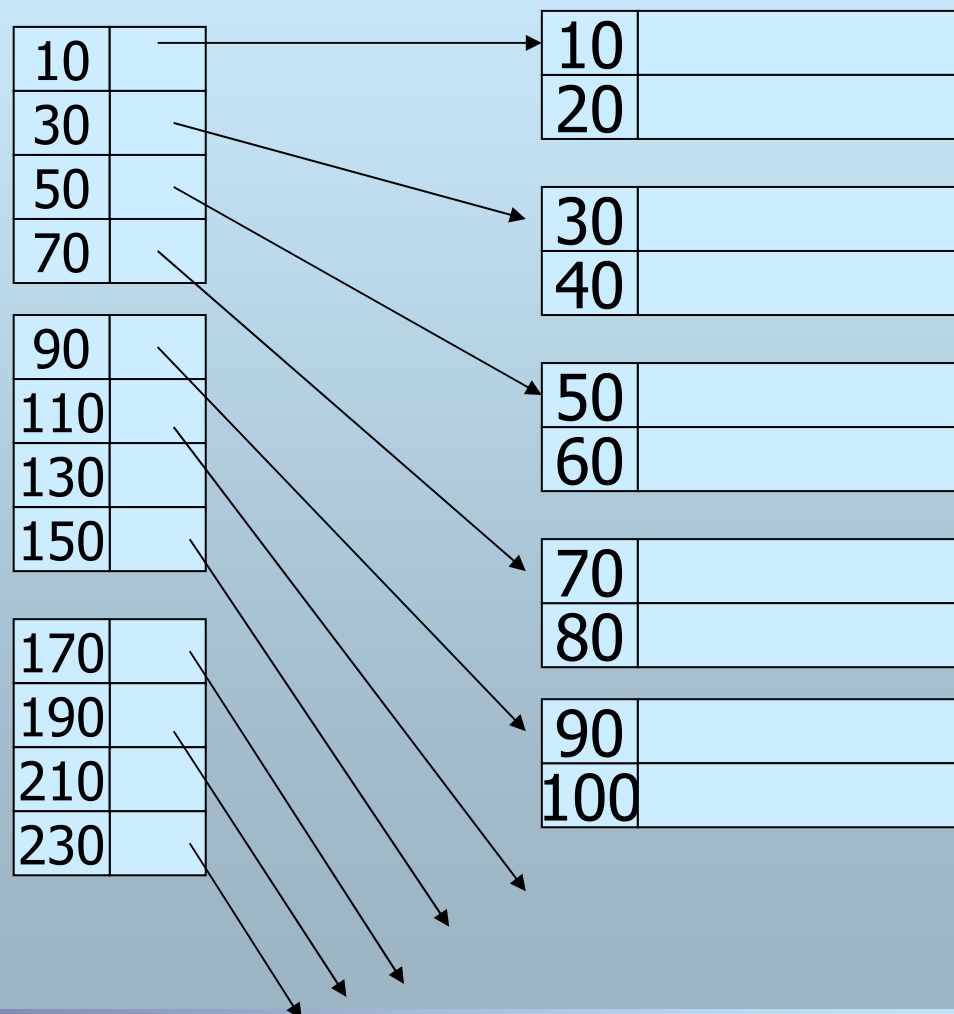
2、稀疏索引(Sparse Index)

- 仅部分记录有索引项
- 一般情况：为每个数据块的第一个记录建立索引

2、稀疏索引(Sparse Index)

Sparse Index

Sequential File



2、稀疏索引(Sparse Index)

■ 有何优点？

- 节省了索引空间
- 对同样的记录，稀疏索引可以使用更少的索引项

■ 有何缺点？

- 对于“是否存在键值为K的记录？”，需要访问磁盘数据块

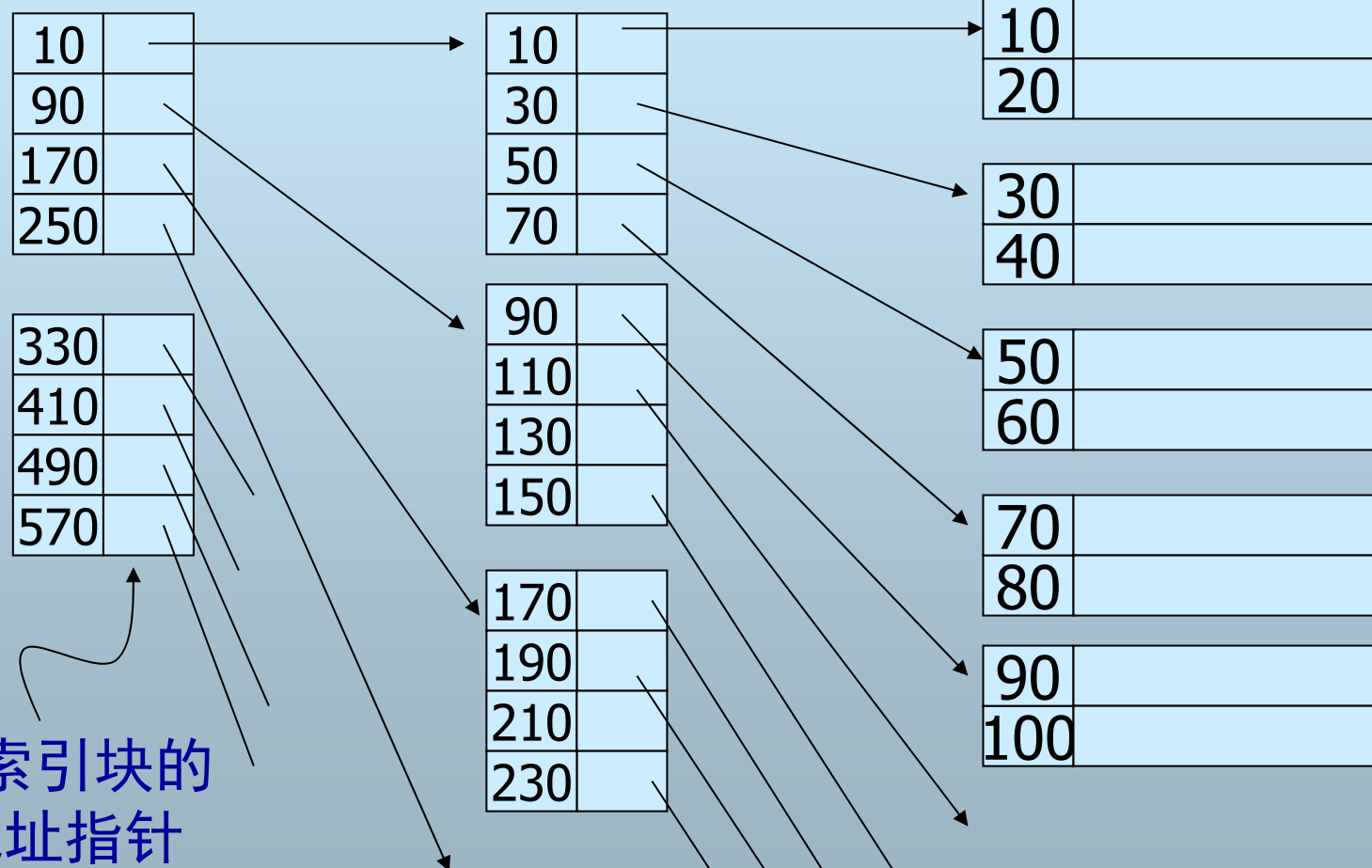
3、多级索引(Multi-level Index)

- 索引上再建索引
 - 二级索引、三级索引.....

3、多级索引(Multi-level Index)

Sparse 2nd level

Sequential File



3、多级索引(Multi-level Index)

■ 多级索引的好处？

- 一级索引可能还太大而不能常驻内存
- 二级索引更小，可以常驻内存
- 减少磁盘I/O次数

3、多级索引(Multi-level Index)

例：一块=4KB。一级索引10,000个块，每个块可存100个索引项，共40MB。二级稀疏索引100个块，共400KB。

按一级索引查找(二分查找)：平均 $\lg 10000 \approx 13$ 次I/O定位索引块，加一次数据块I/O，共约14次I/O

按二级索引查找：定位二级索引块0次I/O，读入一级索引块1次I/O，读入数据块1次I/O，共2次I/O

3、多级索引(Multi-level Index)

- 当一级索引过大而二级索引可常驻内存时有效
- 二级索引仅可用稀疏索引
 - 思考：二级密集索引有用吗？
- 一般不考虑三级以上索引
 - 维护多级索引结构
 - 有更好的索引结构——**B⁺树**

二、辅助索引(Secondary Index)

■ 主索引 (Primary Index)

- 顺序文件上的索引
- 记录按索引属性值有序
- 根据索引值可以确定记录的位置

■ 辅助索引

- 数据文件不需要按查找键有序
- 根据索引值不能确定记录在文件中的顺序

1、辅助索引概念

```
MovieStar(name char(10) PRIMARY KEY, address char(20))
```

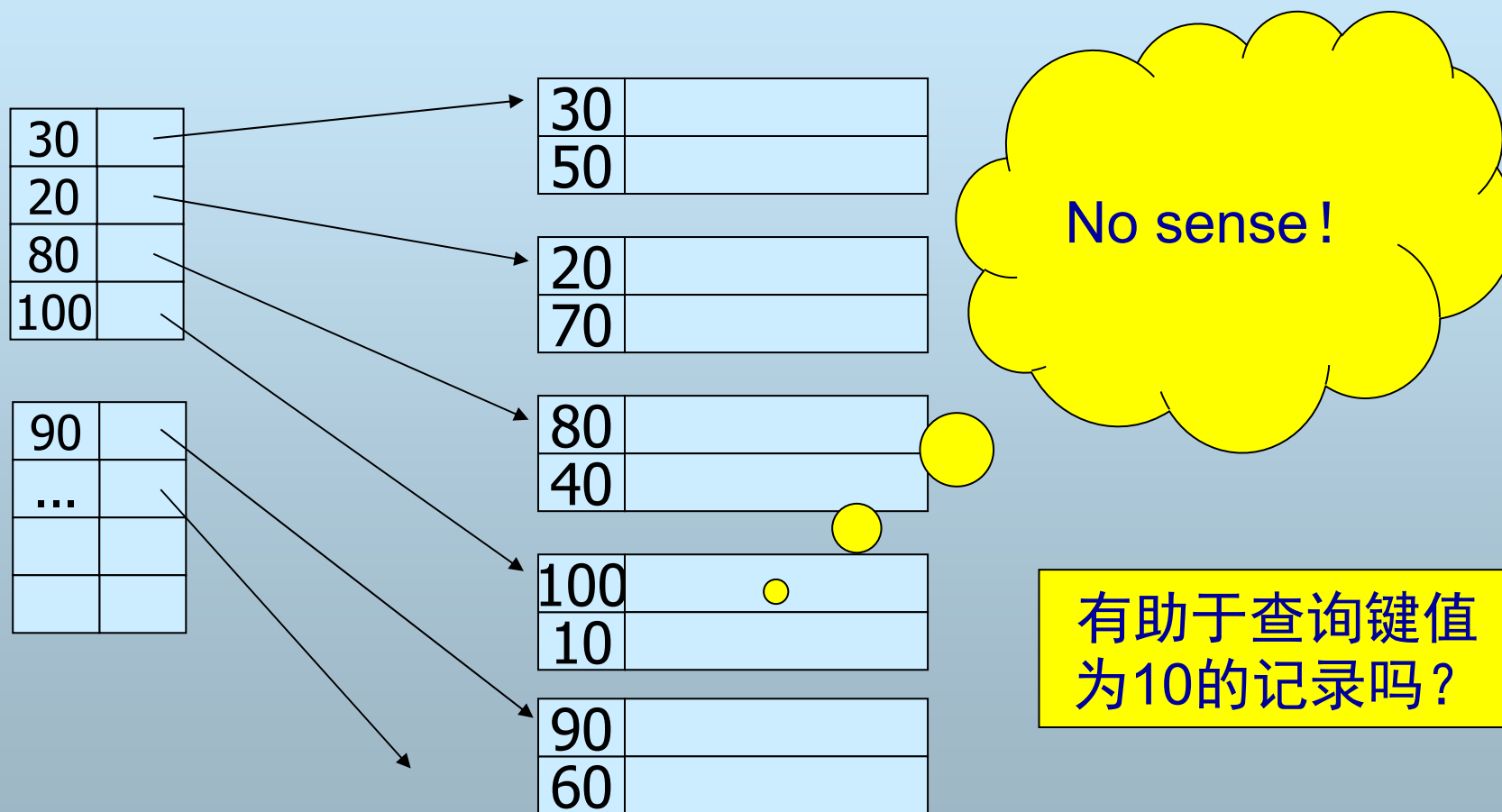
- **Name**上创建了主索引，记录按**name**有序
- **Address**上创建辅助索引

```
Create Index adIndex On MovieStar(address)
```

1、辅助索引概念

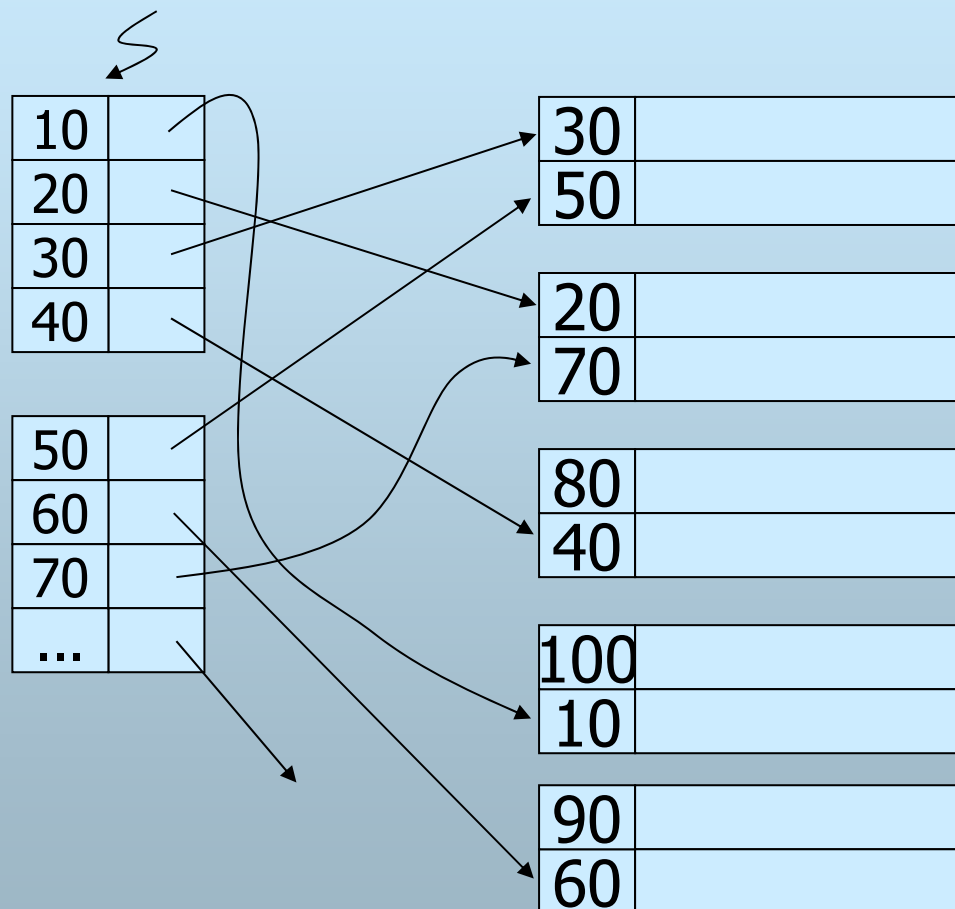
- 辅助索引只能是密集索引
 - 稀疏的辅助索引有意义吗？

1、辅助索引概念

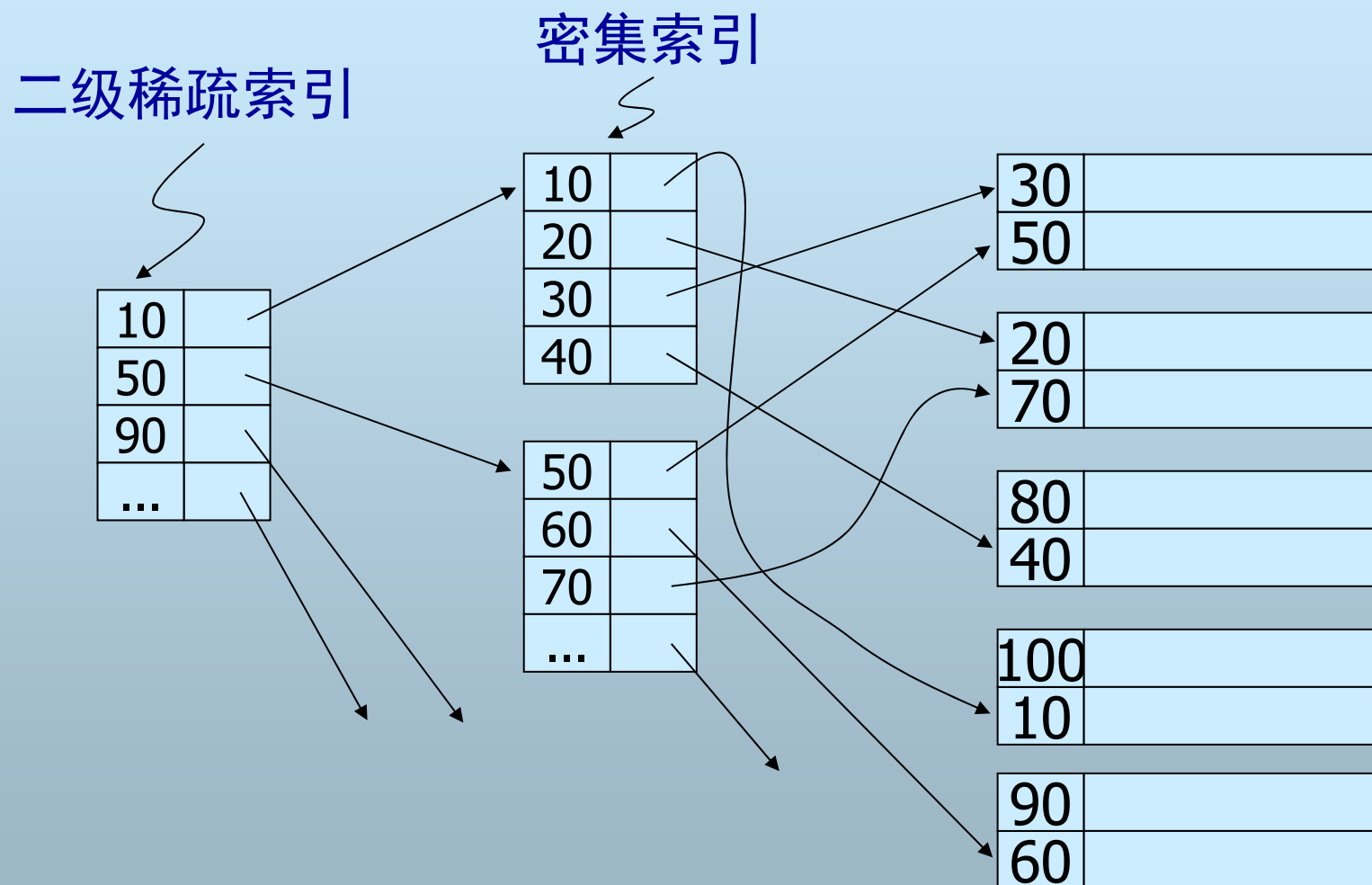


2、辅助索引设计

密集索引



2、辅助索引设计



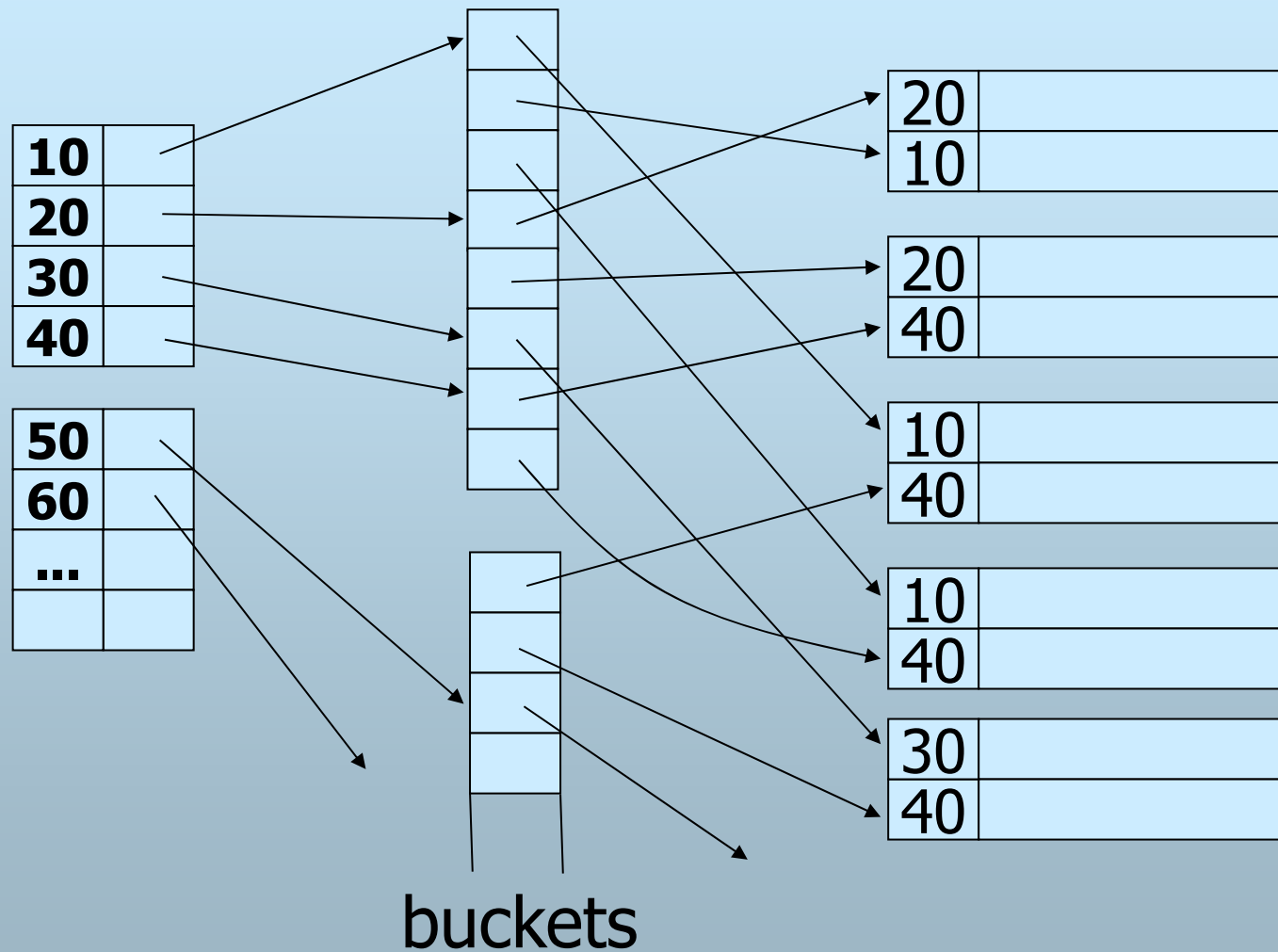
问题

- 重复键值怎么处理？

3、辅助索引中的间接桶

- **Indirect Bucket**
- **重复键值**
 - 采用密集索引浪费空间
- **间接桶**
 - 介于辅助索引和数据文件之间

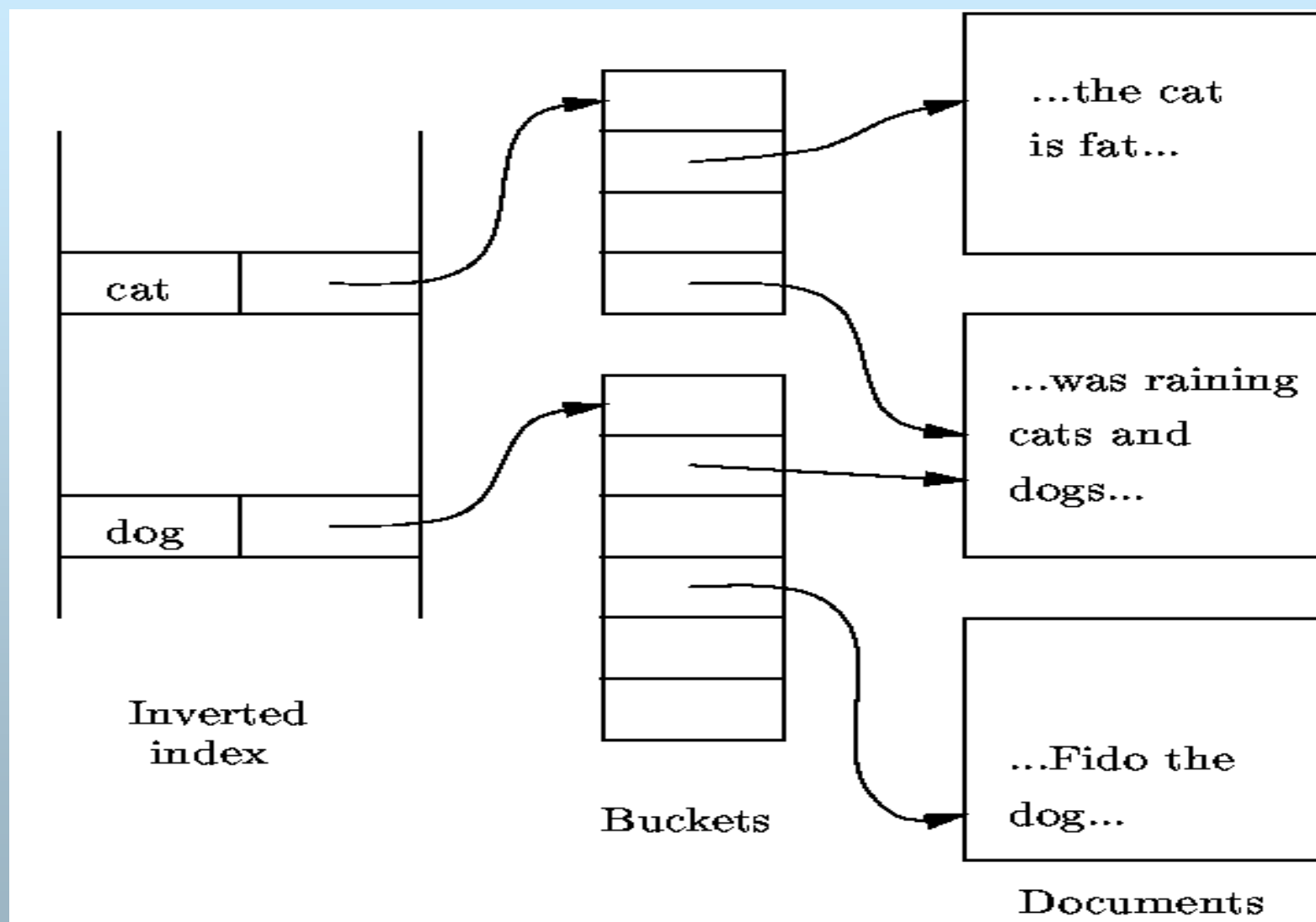
3、辅助索引中的间接桶



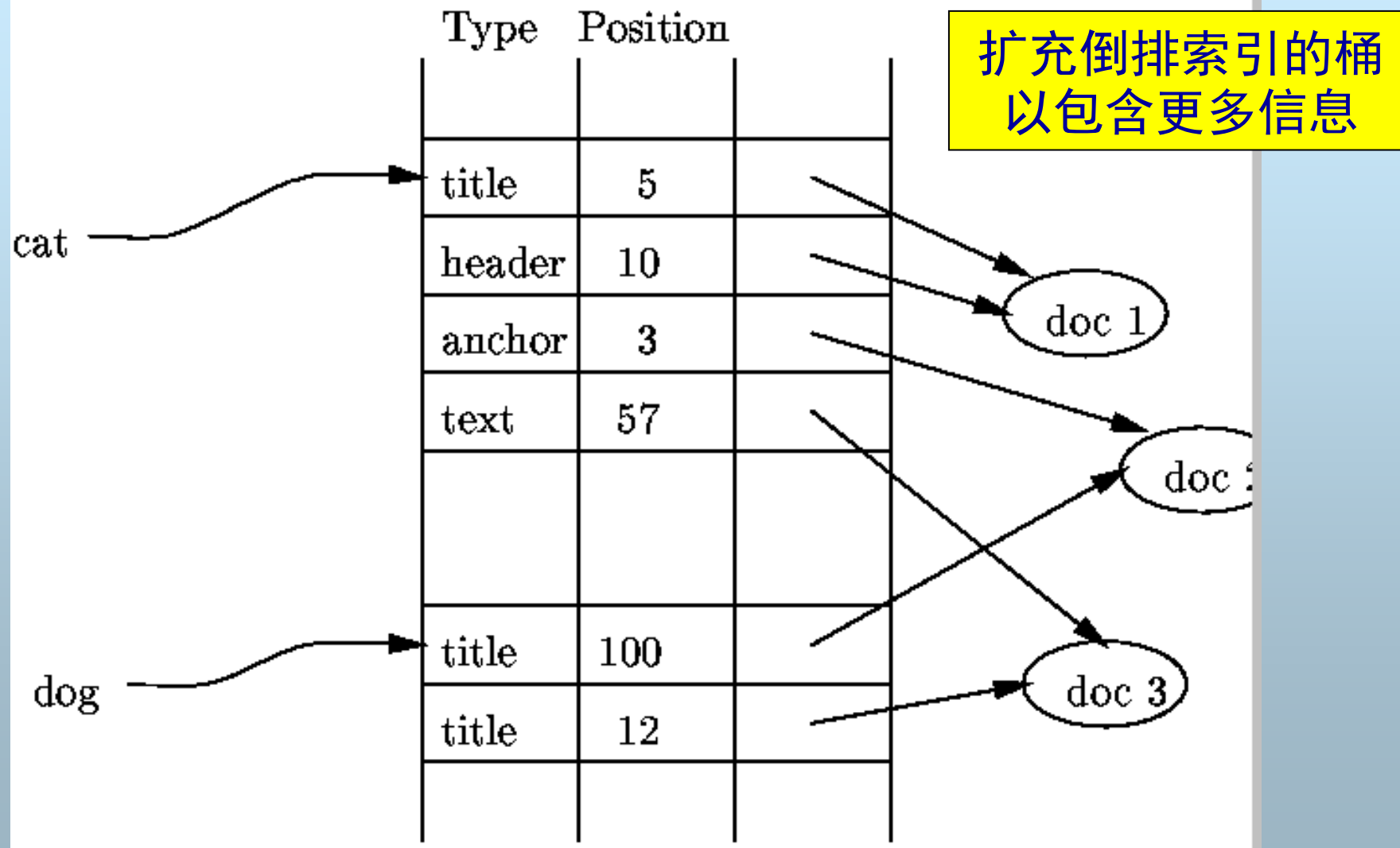
4、倒排索引(Inverted Index)

- 应用于文档检索，与辅助索引思想类似
- 不同之处
 - 记录→文档
 - 记录查找→ 文档检索
 - 查找键→ 文档中的词
- 思想
 - 为每个检索词建立间接桶
 - 桶的指针指向检索词所出现的文档

4、倒排索引(Inverted Index)



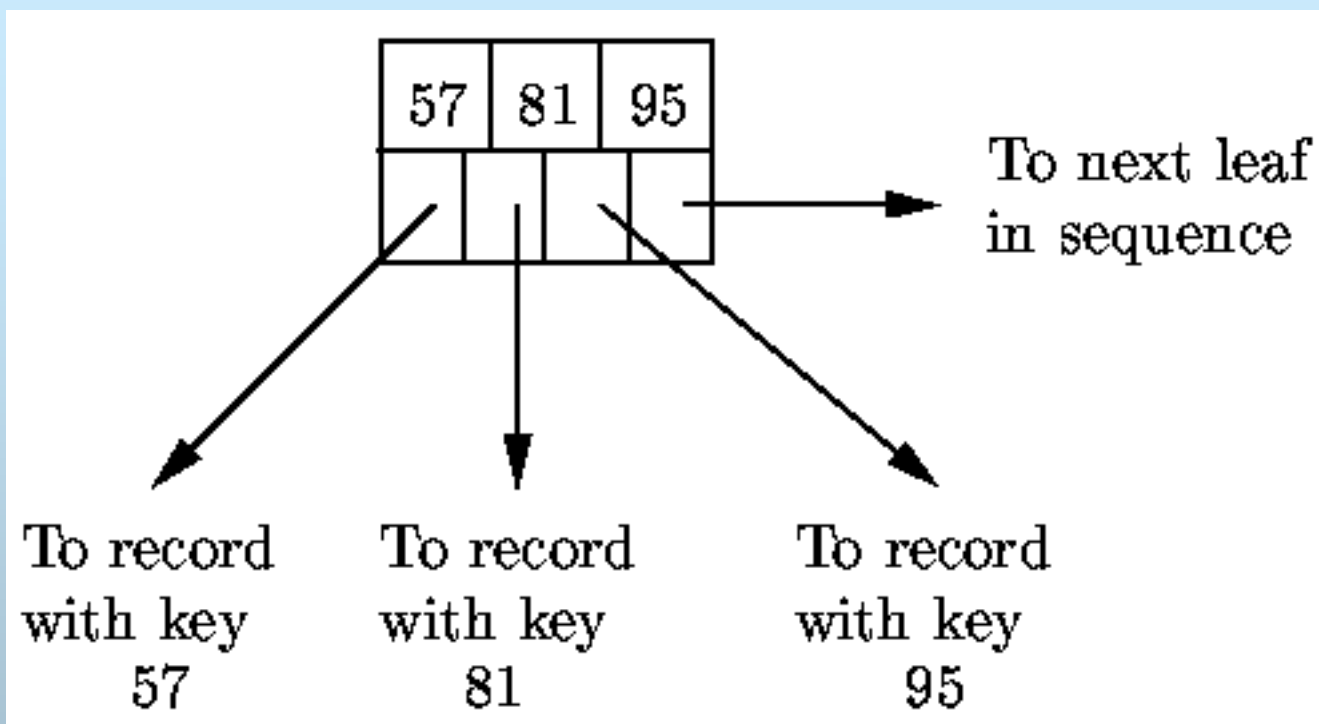
4、倒排索引(Inverted Index)



二、B+树

- 一种树型的多级索引结构
- 树的层数与数据大小相关，通常为**3**层
- 所有结点格式相同： **n** 个值， **$n+1$** 个指针
- 所有叶结点位于同一层

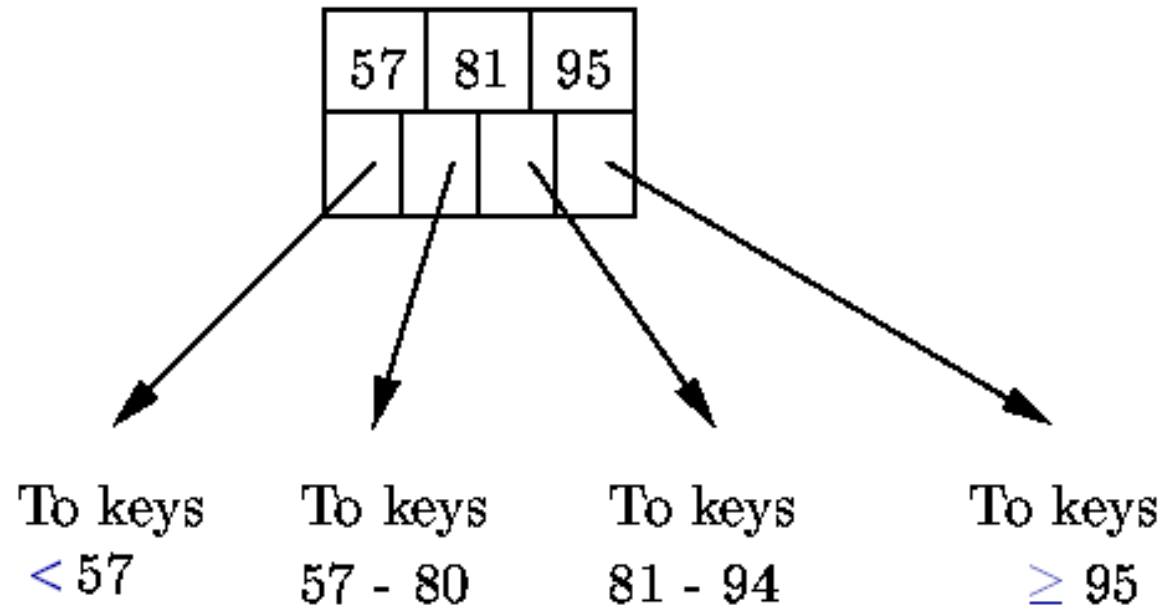
1、叶结点



- 1个指向相邻叶结点的指针
- n对键一指针对

- 至少 $\lfloor (n+1)/2 \rfloor$ 个指针指向键值

2、中间结点



- n 个键值划分 $n+1$ 个子树
- 第 i 个键值是第 $i+1$ 个子树中的最小键值
- 至少 $\lceil (n+1)/2 \rceil$ 个指针指向子树
- 根结点至少 2 个指针

B+树结点例子

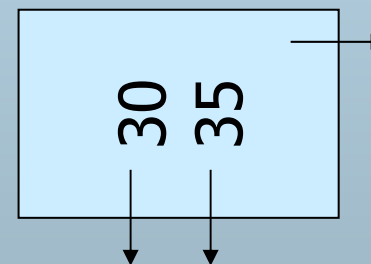
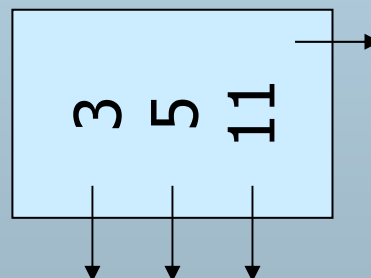
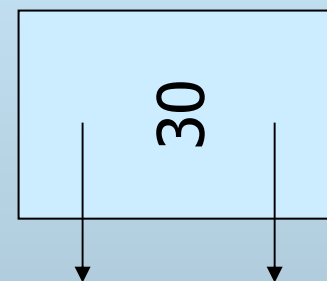
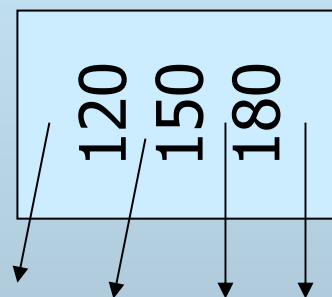
$n=3$

Interior

Full node

min. node

Leaf



3、B+树查找

- 从根结点开始
- 沿指针向下，直到到达叶结点
- 在叶结点中顺序查找

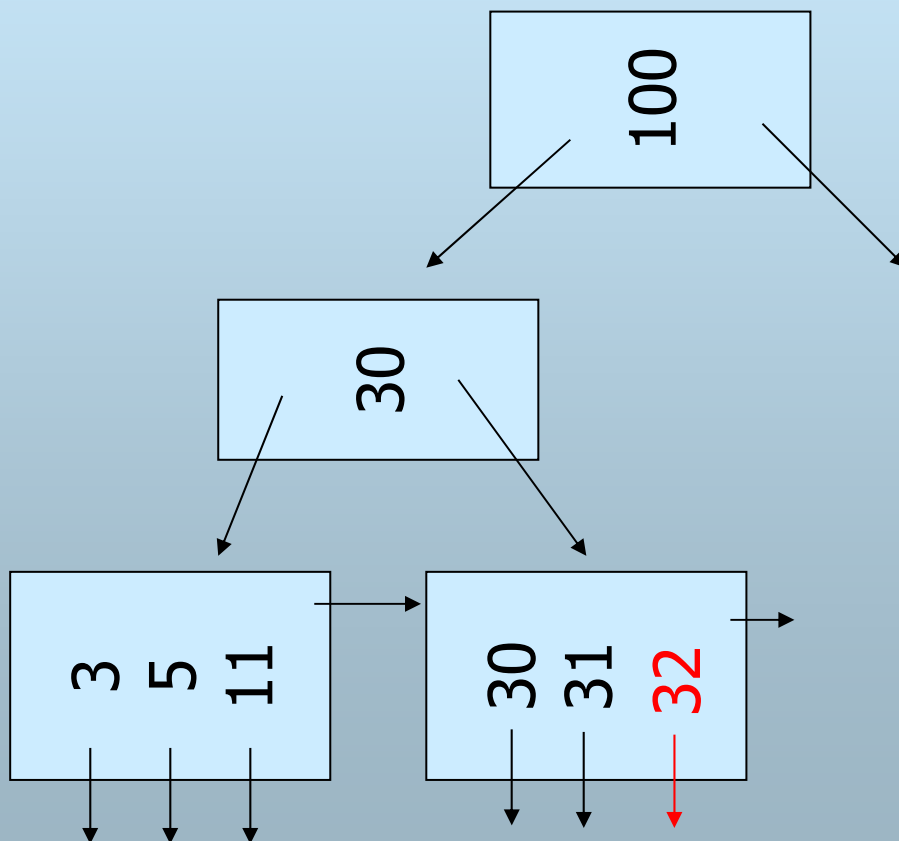
4、B+树插入

- 查找插入叶结点
- 若叶结点中有空闲位置（键），则插入
- 若没有空间，则分裂叶结点
 - 叶结点的分裂可视作是父结点中插入一个子结点
 - 递归向上分裂
 - 分裂过程中需要对父结点中的键加以调整
 - 例外：若根结点分裂，则需要创建一个新的根结点

B+树插入例子

(a) Insert key = 32

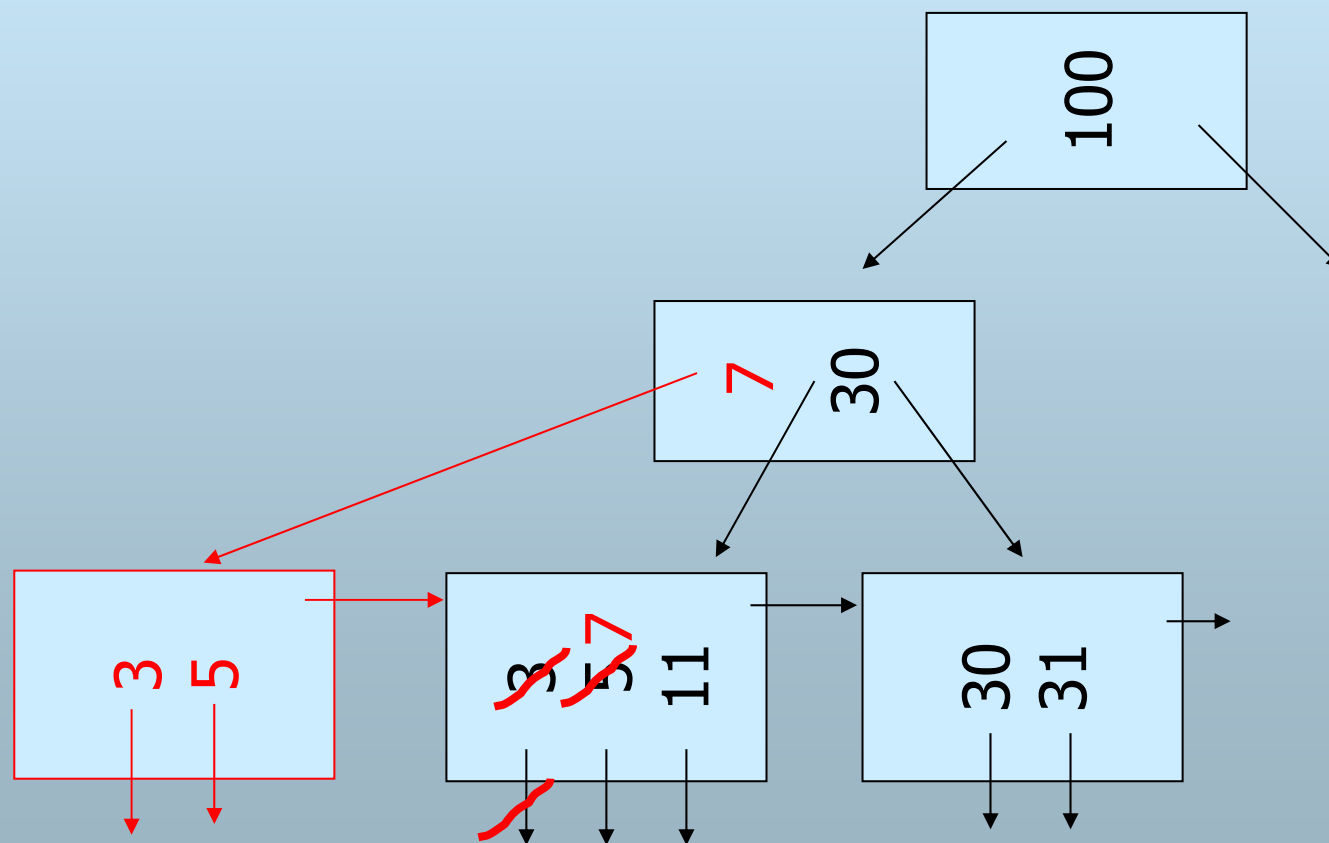
$n=3$



B+树插入例子

(b) Insert key = 7

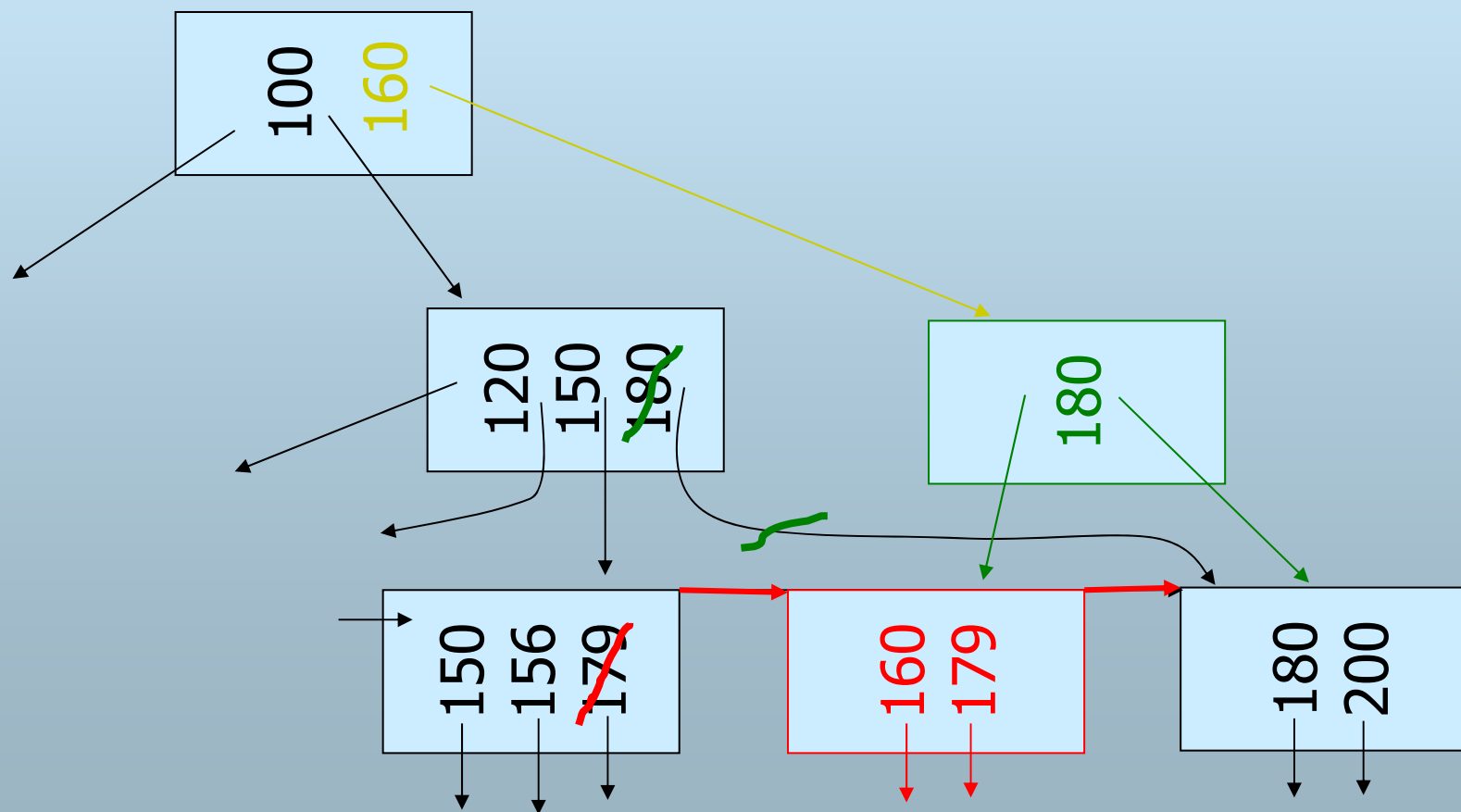
n=3



B+树插入例子

(c) Insert key = 160

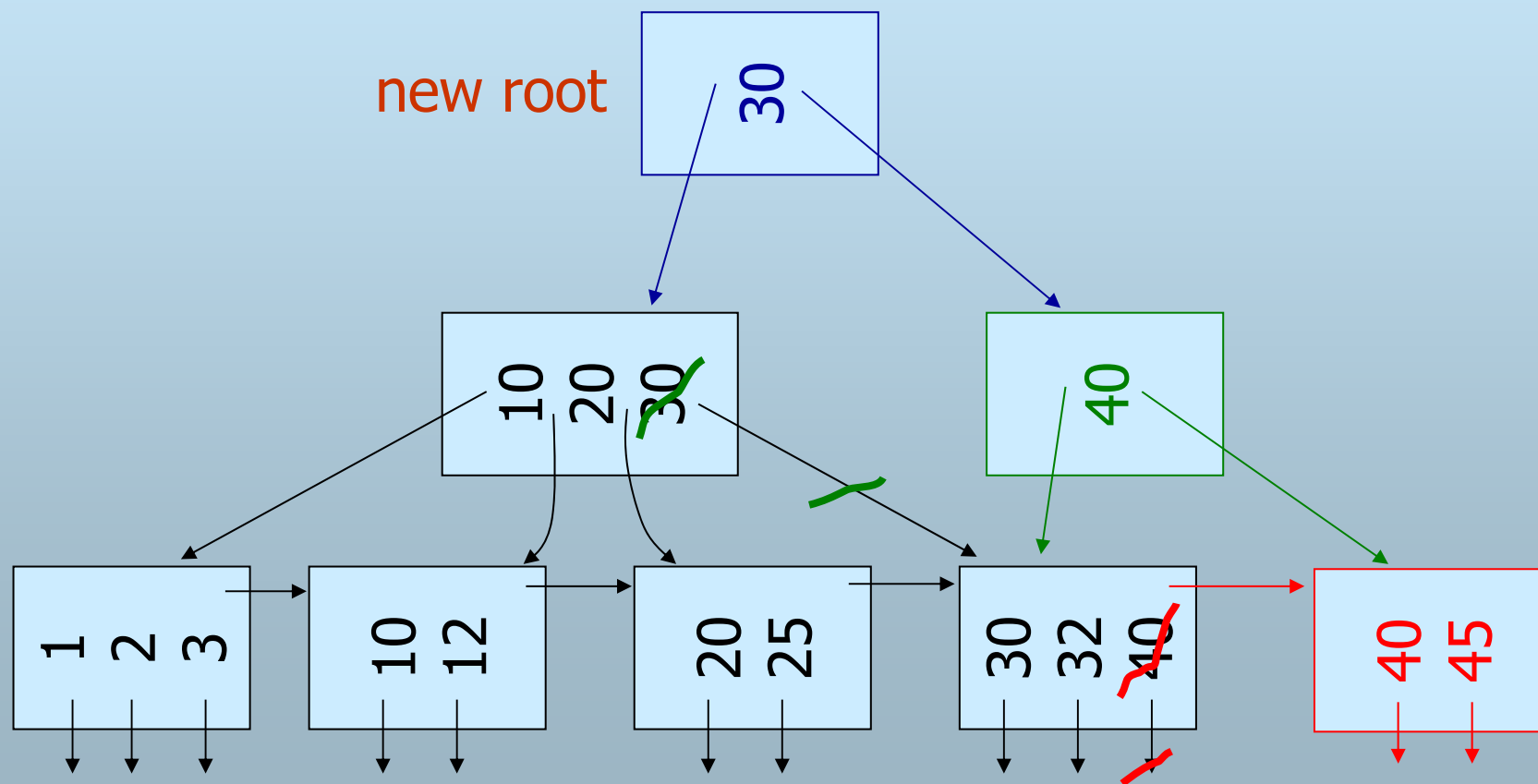
n=3



B+树插入例子

(d) New root, insert 45

n=3



5、B+树删除

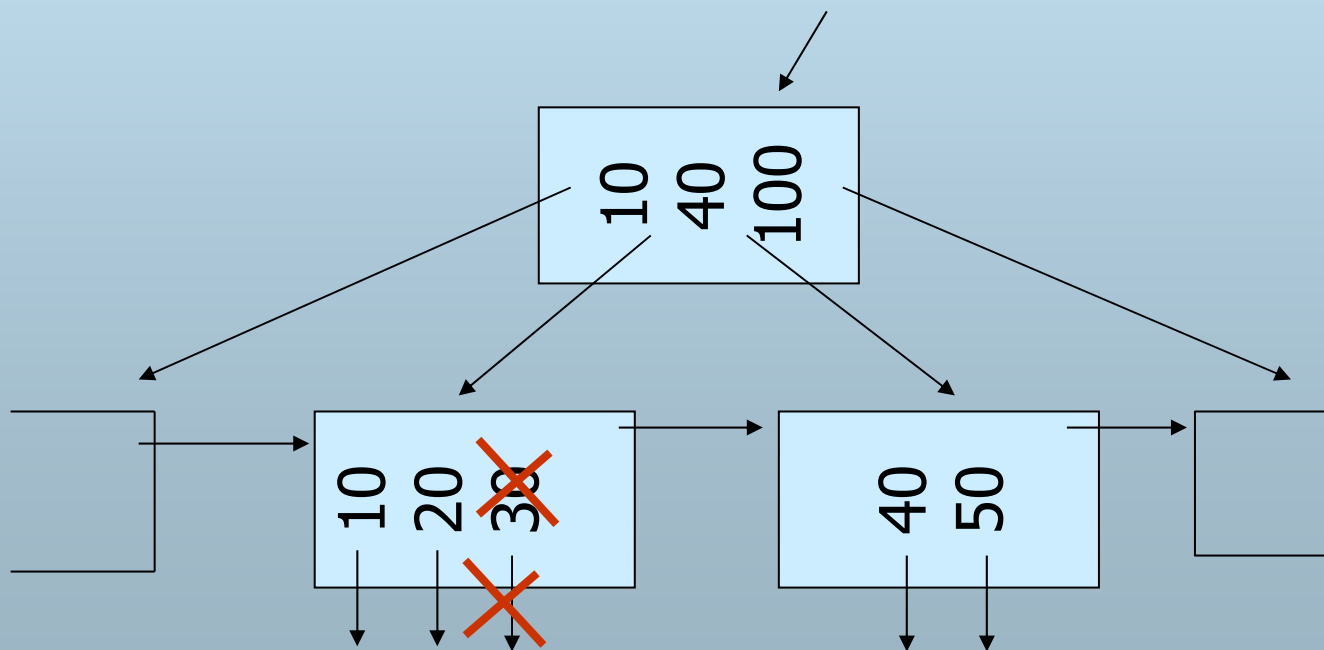
- 查找要删除的键值，并删除之
- 若结点的键值填充低于规定值，则调整
 - 若相邻的叶结点中键填充高于规定值，则将其中一个键值移到该结点中
 - 否则，合并该结点与相邻结点
 - ◆ 合并可视作在父结点中删除一个子结点
 - 递归向上删除
- 若删除的是叶结点中的最小键值，则需对父结点的键值加以调整

B+树删除例子

■ (a) Simple delete

● Delete 30

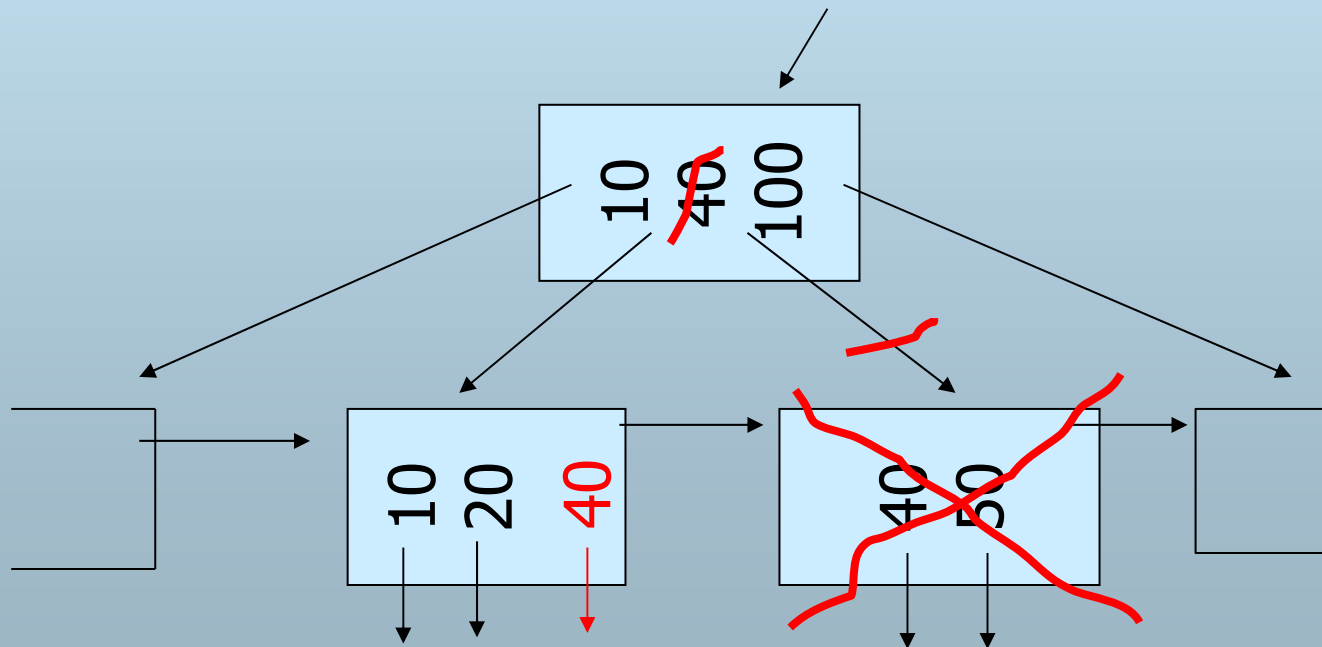
n=4



B+树删除例子

- (b) Coalesce with sibling
 - Delete 50

n=4

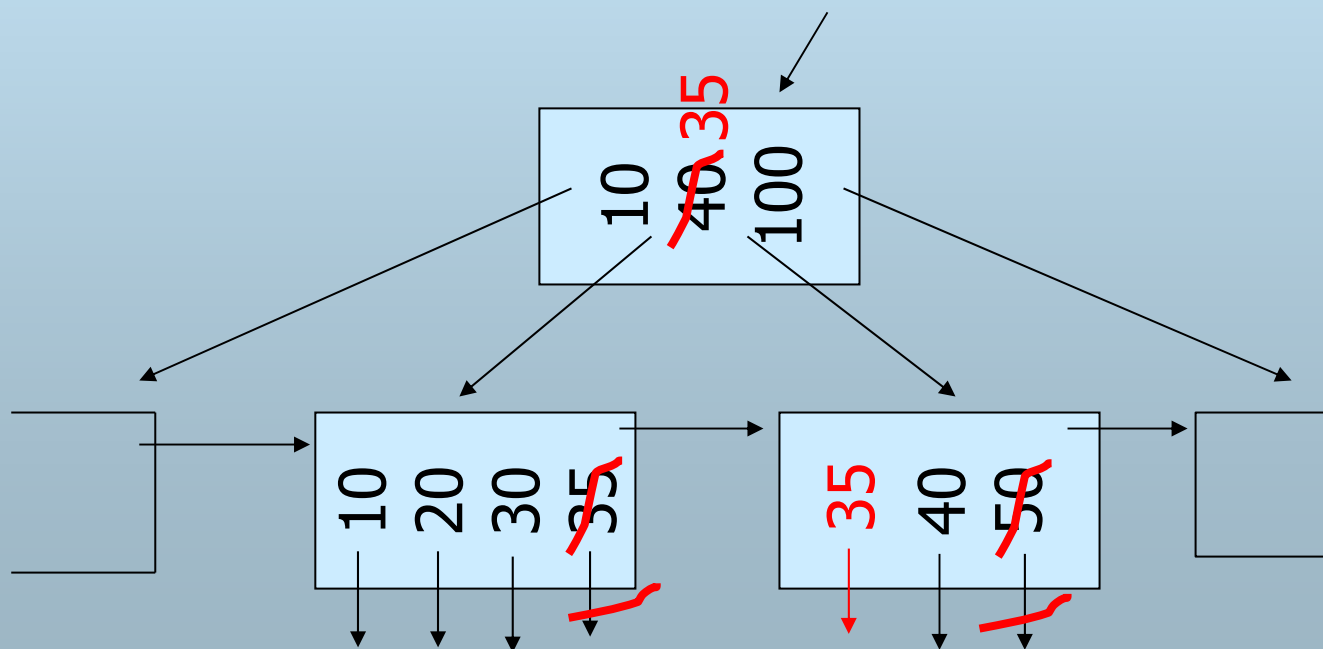


B+树删除例子

■ (c) Redistribute keys

● Delete 50

n=4



6、B+树的效率

- 访问索引的**I/O**代价=树高（**B+**树不常驻内存）或者**0**（常驻内存）
- 树高通常不超过**3**层，因此索引**I/O**代价不超过**3**（总代价不超过**4**）
 - 通常情况下，根节点常驻内存，因此索引**I/O**代价不超过**2**（总代价不超过**3**）

6、B+树的效率

- 设块大小**8KB**，键**2B**（**smallint**），指针**2B**，则一个块可放**2048**个索引项

层数	索引大小（块数/大小）	索引记录空间
1	1/8KB	2047
2	$(1 + 2048)/\text{约}16\text{M}$	约419万(2^{22})
3	$(1 + 2^{11} + 2^{22})/\text{约}32\text{G}$	约85亿(2^{33})

三、散列表(Hash Table)

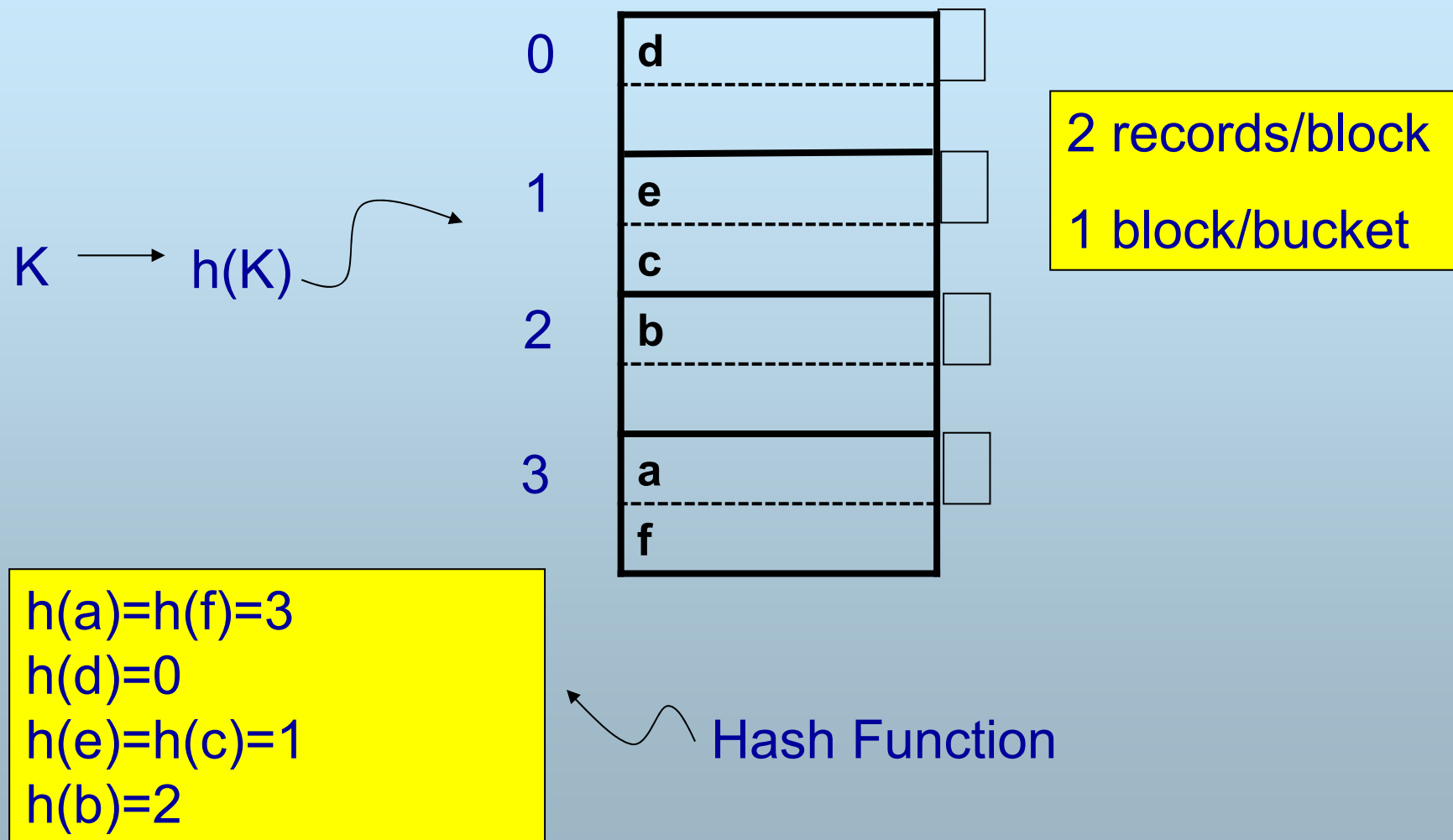
■ 散列函数(Hash Functions)

- h : 查找键(散列键) $\rightarrow [0 \dots B - 1]$
- 桶(Buckets), numbered $0, 1, \dots, B-1$

■ 散列索引方法

- 给定一个查找键 K , 对应的记录必定位于桶 $h(K)$ 中
- 若一个桶中仅一块, 则 I/O 次数 = 1
- 否则由参数 B 决定, 平均 = 总块数 / B

1、散列表概念



2、散列表查找

■ 查找

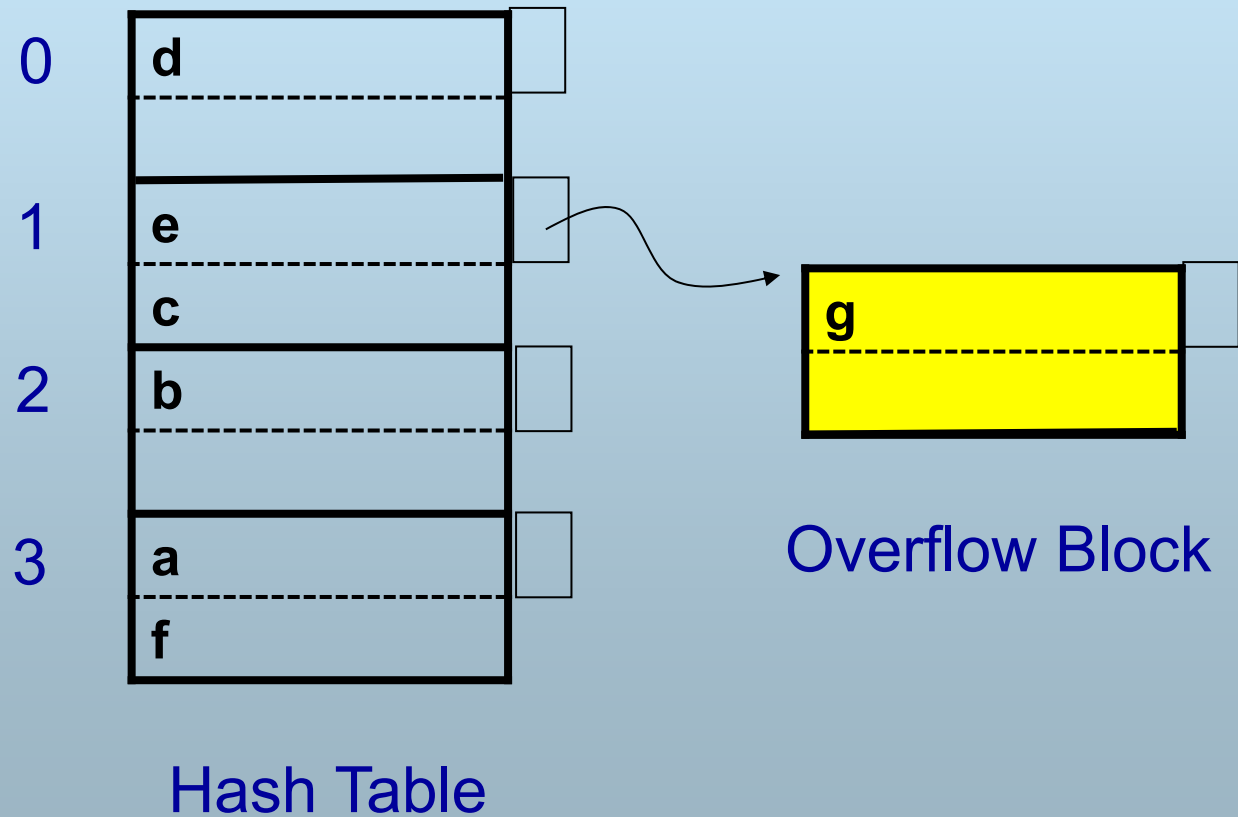
- 对于给定的散列键值 k ，计算 $h(K)$
- 根据 $h(K)$ 定位桶
- 查找桶中的块

3、散列表插入

- 计算插入记录的 $h(K)$ ，定位桶
- 若桶中有空间，则插入
- 否则
 - 创建一个溢出块并将记录置于溢出块中

插入例子

插入g, $h(k)=1$



3、散列表删除

- 根据给定键值 K 计算 $h(K)$ ，定位桶和记录
- 删除
 - 回收溢出块？

散列表删除例子

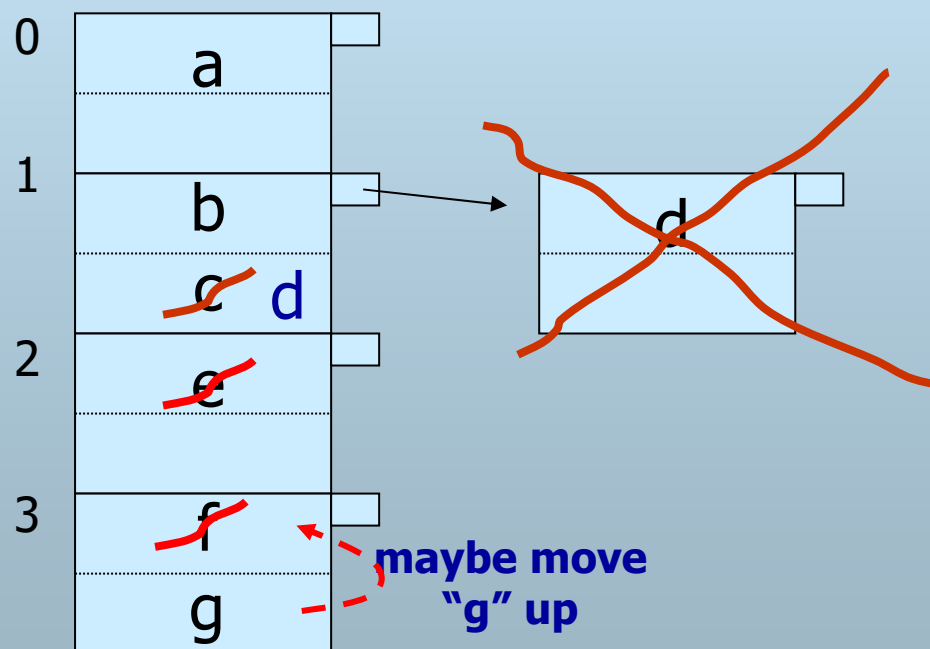
EXAMPLE: deletion

Delete:

e

f

C



4、散列表空间利用率问题

■ 空间利用率

- 实际键值数 / 所有桶可放置的键值数
- <50%：空间浪费
- >80%：容易产生溢出块，降低查询性能
- 50%到80%之间 (**GOOD!**)

5、文件增长

- 数据文件的增长使桶的溢出块数增多，增加 I/O

- 采用动态散列表解决

- ◆ 可扩展散列表（**Extensible Hash Tables**）

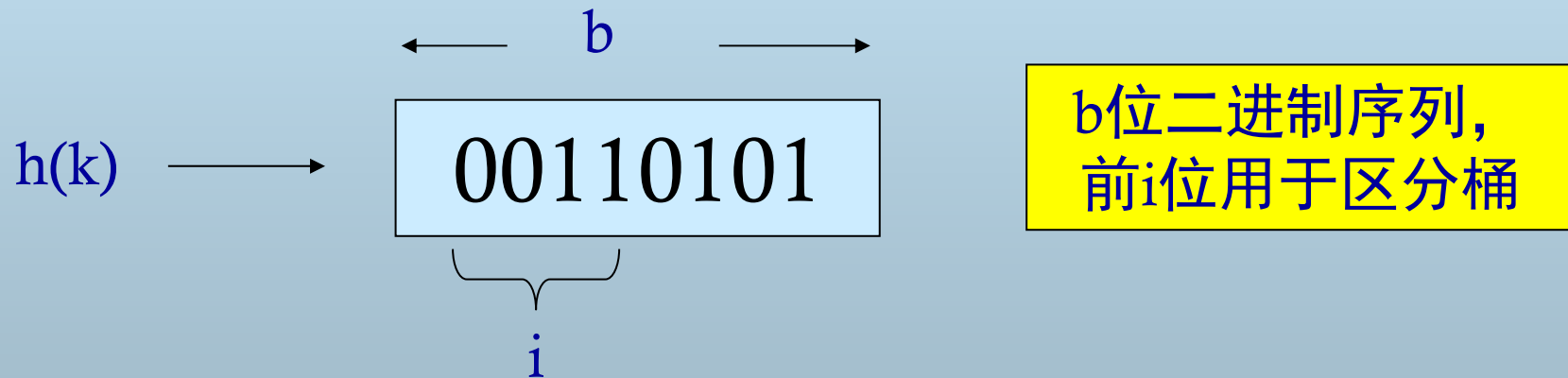
- 成倍增加桶数目

- ◆ 线性散列表（**Linear Hash Tables**）

- 线性增加

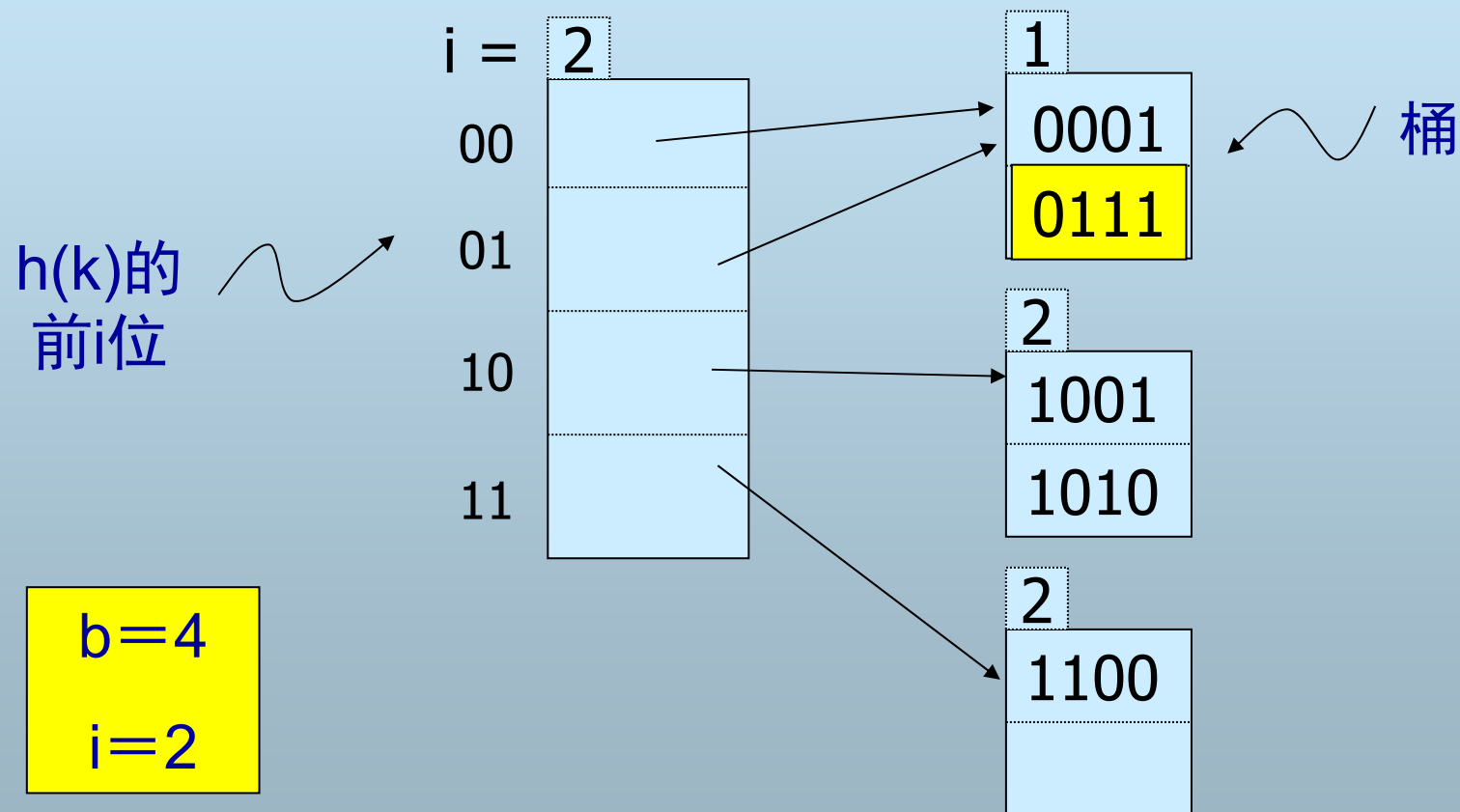
6、可扩展散列表

- 散列函数 $h(k)$ 是一个 b (足够大)位二进制序列，前 i 位表示桶的数目。
- i 的值随数据文件的增长而增大



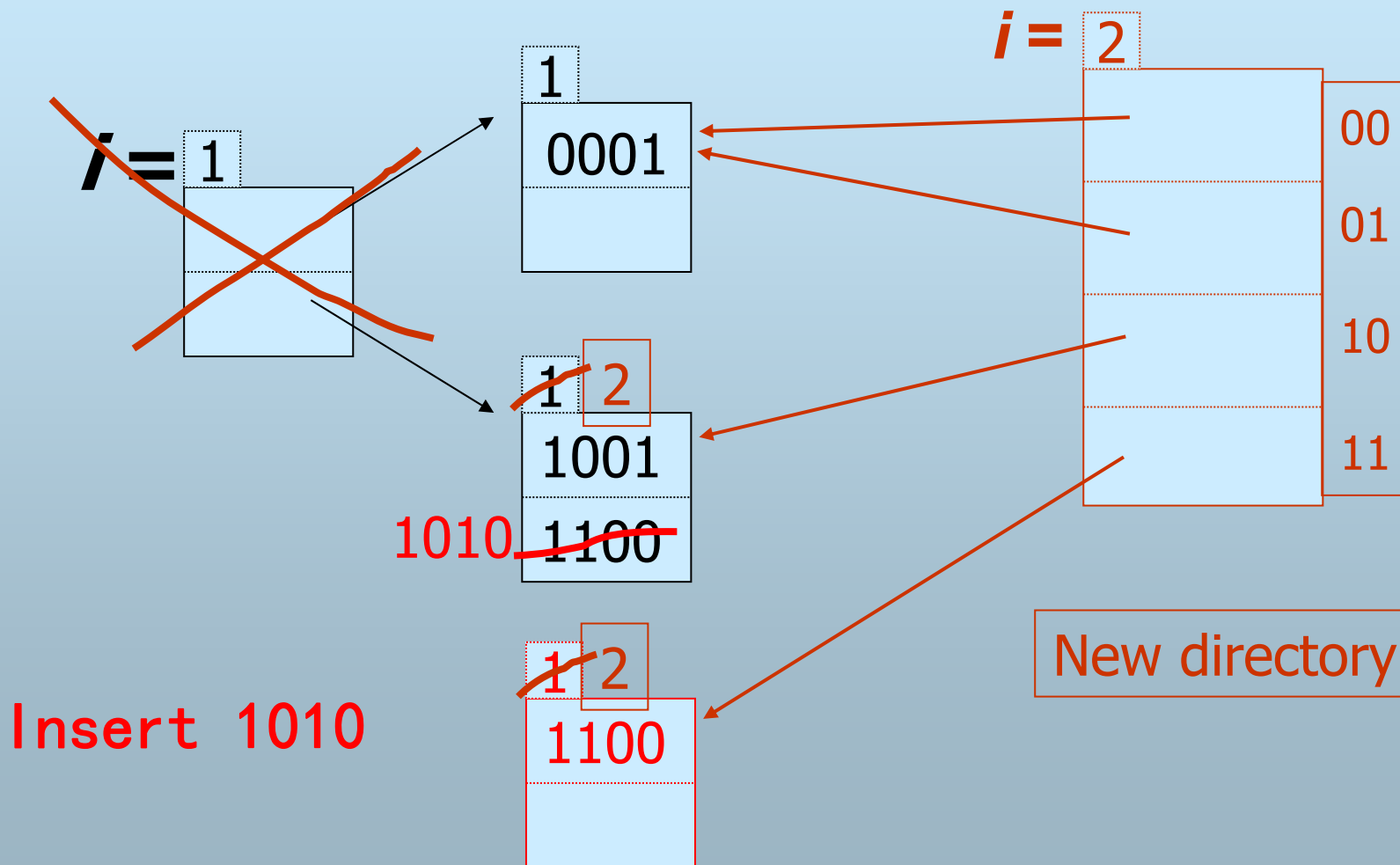
6、可扩展散列表

■ 前*i*位构成一个桶数组



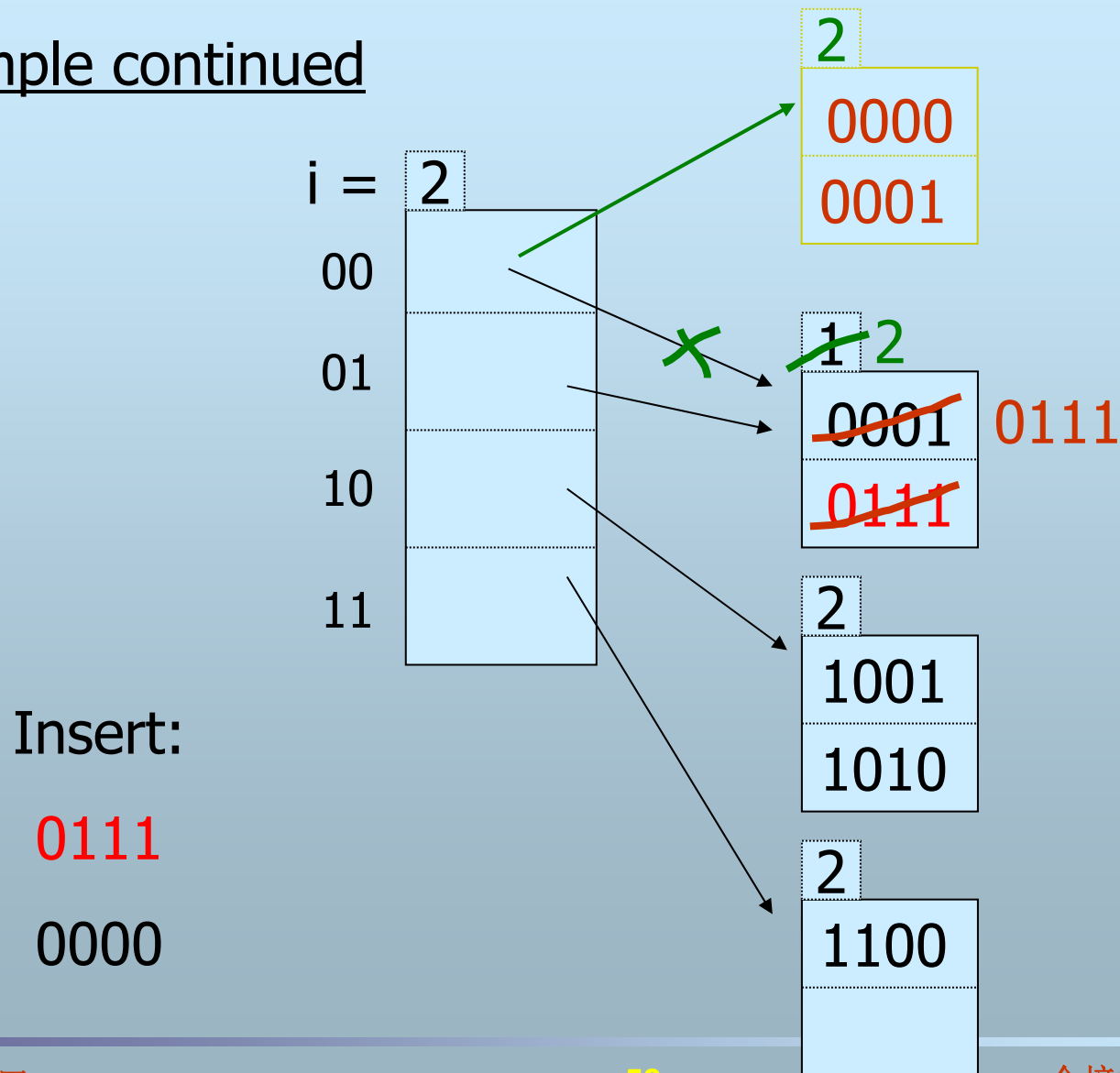
6、可扩展散列表

Example: $h(k)$ is 4 bits; 2 keys/bucket



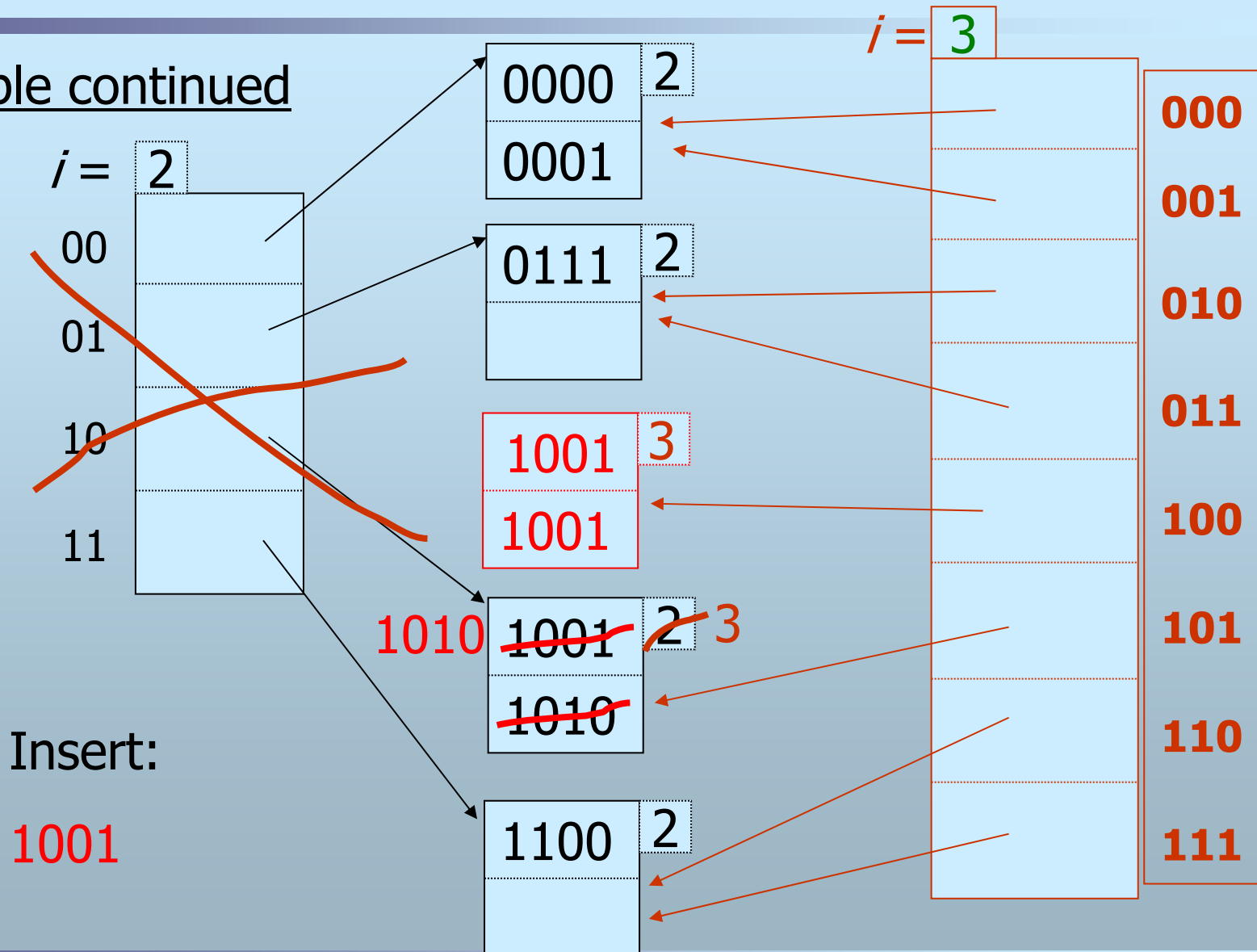
6、可扩展散列表

Example continued



6、可扩展散列表

Example continued



6、可扩展散列表

- 优点：

当查找记录时，只需查找一个存储块。

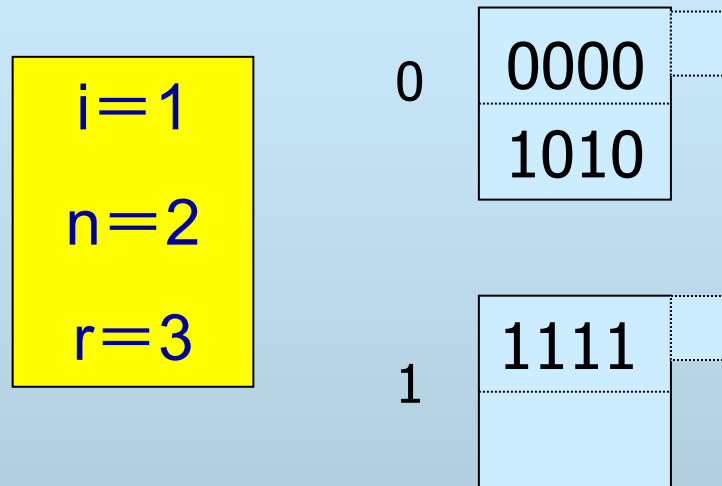
- 缺点：

桶增长速度快，可能会导致内存放不下整个桶数组，影响其他保存在主存中的数据，波动较大。

7、线性散列表

- $h(k)$ 仍是二进制位序列，但使用右边(低) i 位区分桶
 - 桶数= n ， $h(k)$ 的右 i 位= m
 - 若 $m < n$ ，则记录位于第 m 个桶
 - 若 $n \leq m < 2^i$ ，则记录位于第 $m - 2^{i-1}$ 个桶
 - n 的选择：总是使 n 与当前记录总数 r 保持某个固定比例
 - ◆ 意味着只有当桶的填充度达到超过某个比例后桶数才开始增长

7、线性散列表

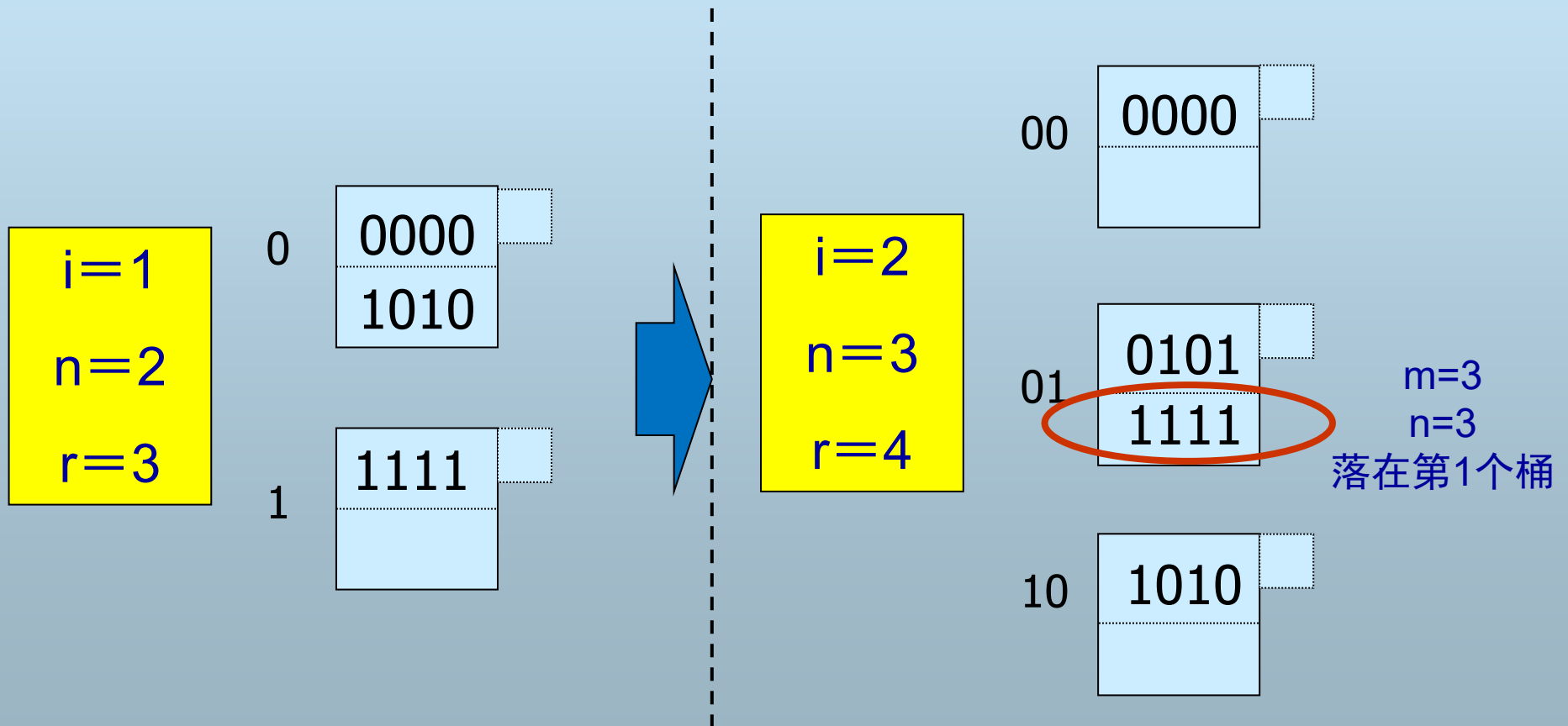


i : 当前被使用的散列函数值的位数，从低位开始
 n : 当前的桶数
 r : 当前散列表中的记录总数 $r/n < 1.7$

7、线性散列表

■ 插入0101

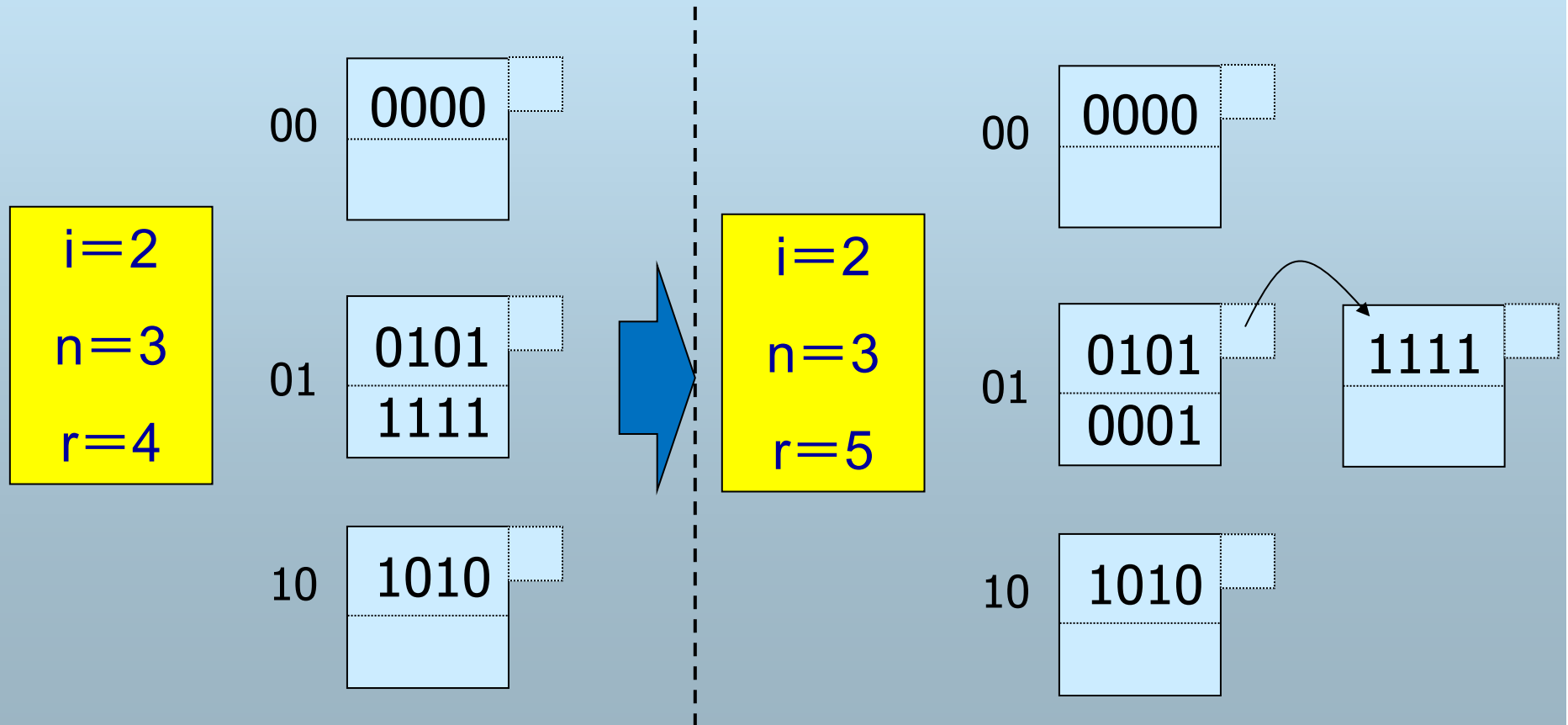
$r/n=4/2$, 超过了1.7, 所以增加新桶



7、线性散列表

■ 插入0001

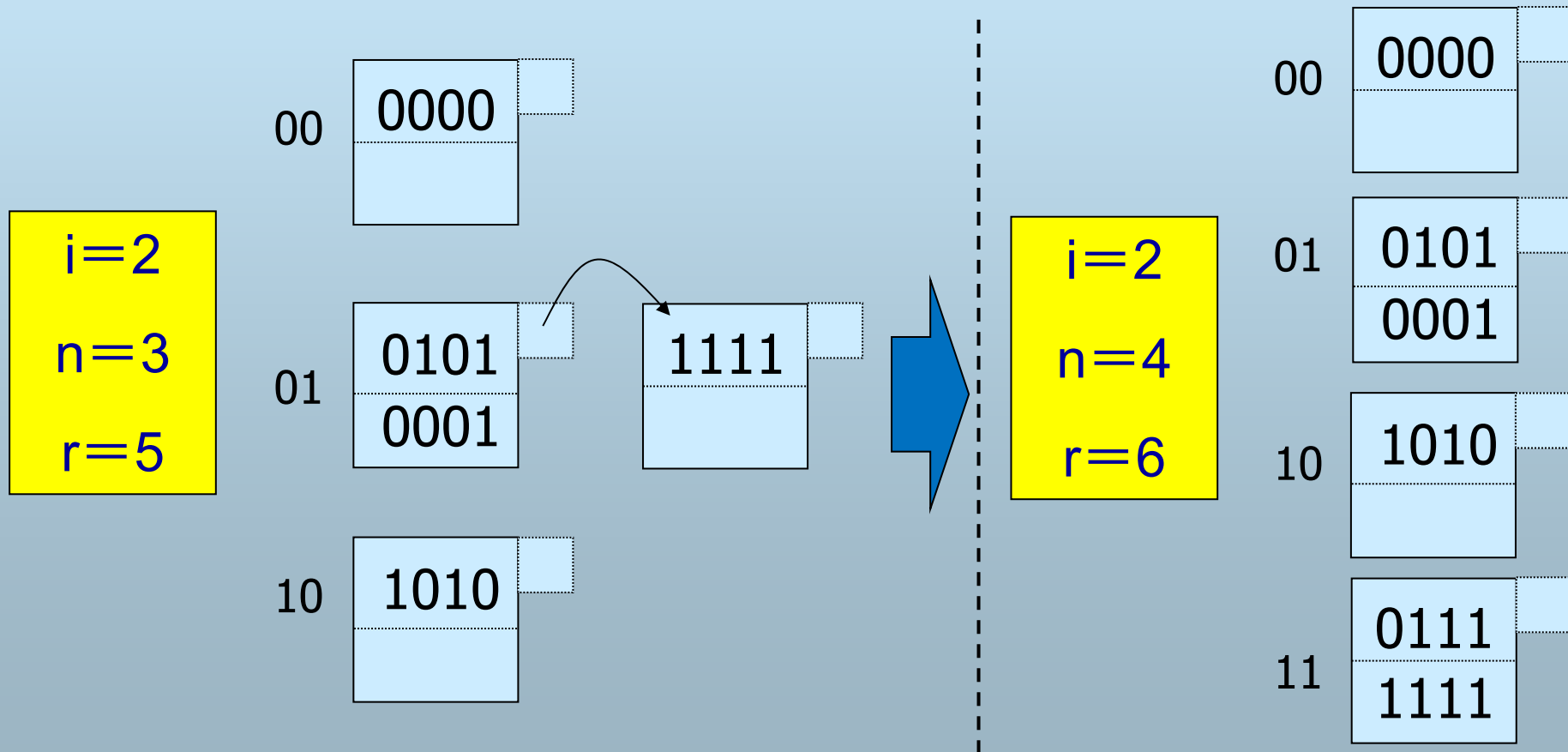
$r/n=5/3$, 小于1.7, 所以不增加新桶而使用溢出块



7、线性散列表

■ 插入0111

$r/n=6/3$, 大于1.7, 所以增加新桶



7、线性散列表

■ 总结

- 空间效率优于可扩展散列表
- 查找性能比可扩展散列表差
- 综合性能较好

小结

- 树形索引结构
 - **B+-Tree**
- 散列型索引
 - **Static Hashing**
 - **Linear Hashing**
 - **Extensible Hashing**