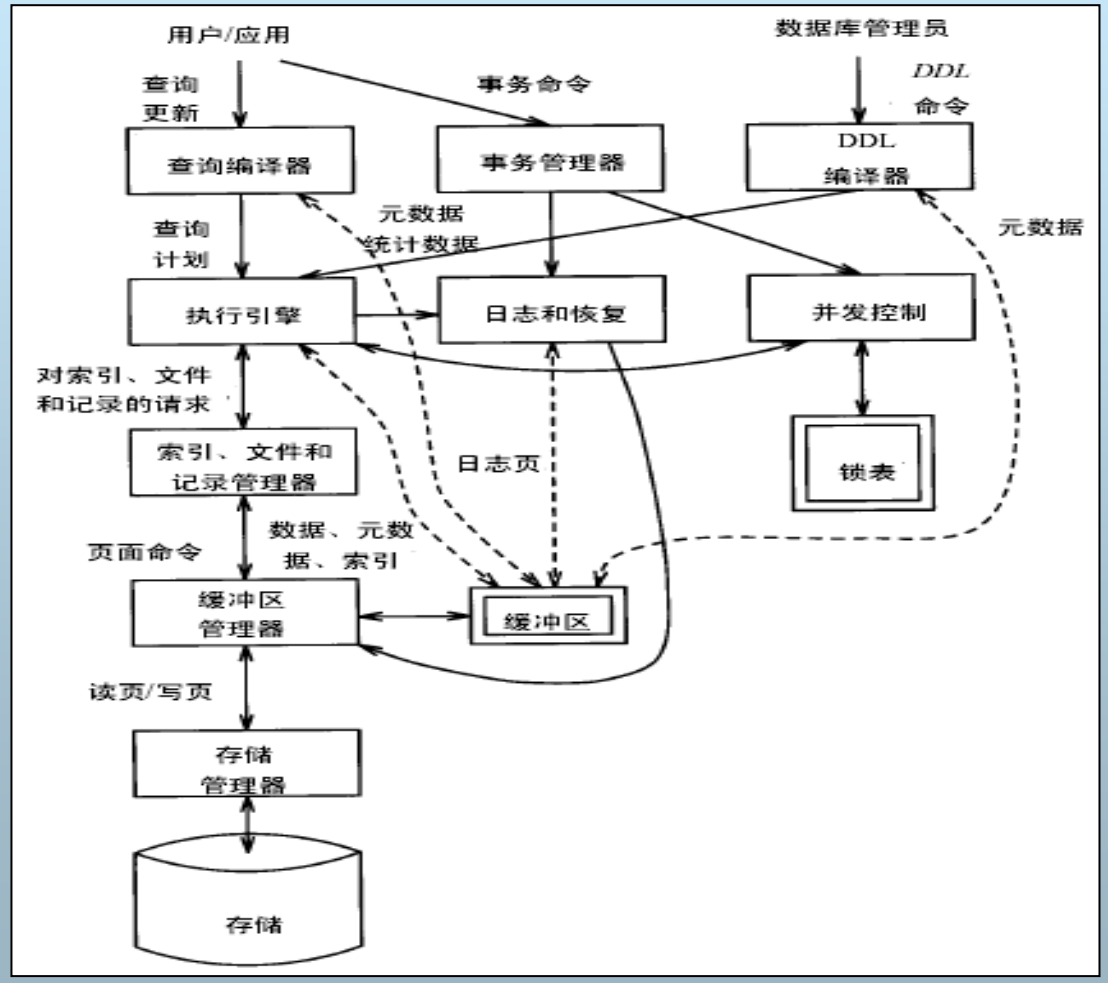
Index Structures

金培权（jpq@ustc.edu.cn）

数据库为什么需要索引？

- 没有索引，数据查询效率低

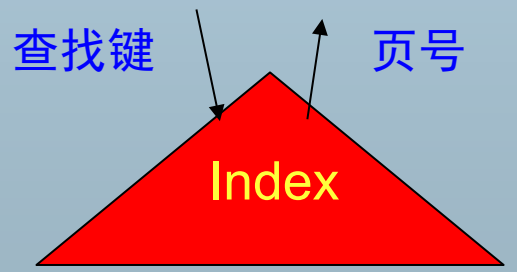


若 page size = 8 KB，page I/O 10ms

1 MB (128 pages)：1.28 s
128 MB (16384 pages)：163.8 s
1 GB (131072 pages)：1310.7 s ≈21.8 min

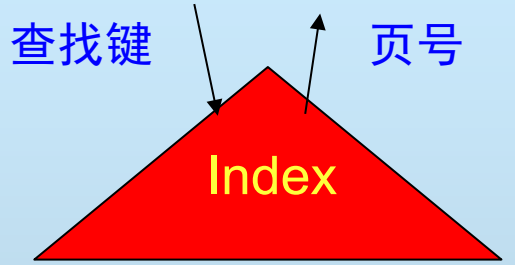
索引的动机：

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

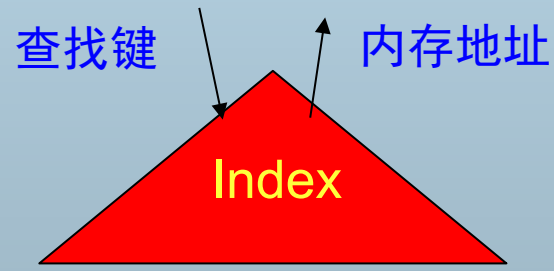


外存索引 vs. 内存索引

- 外存索引
  - 数据在外存，e.g. SSD or HDD
  - 目标是减少I/O代价
  - 数据库索引通常指外存索引



- 内存索引
  - 数据在内存，e.g. DRAM or NVM
  - 目标是减少内存cacheline访问次数

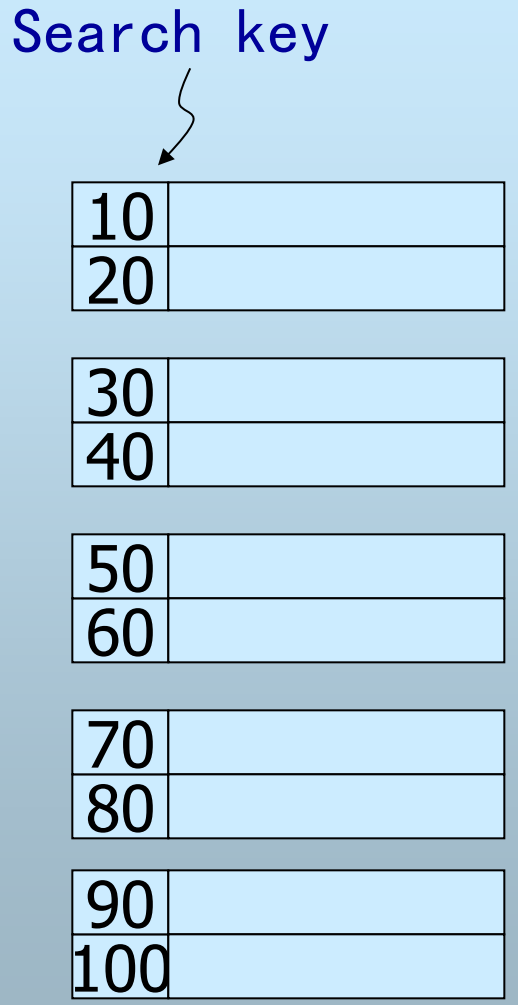


主要内容

- 顺序文件上的索引
- 辅助索引
- 一维索引
  - B+树
  - 散列表 (Hash Tables)

一、顺序文件上的索引

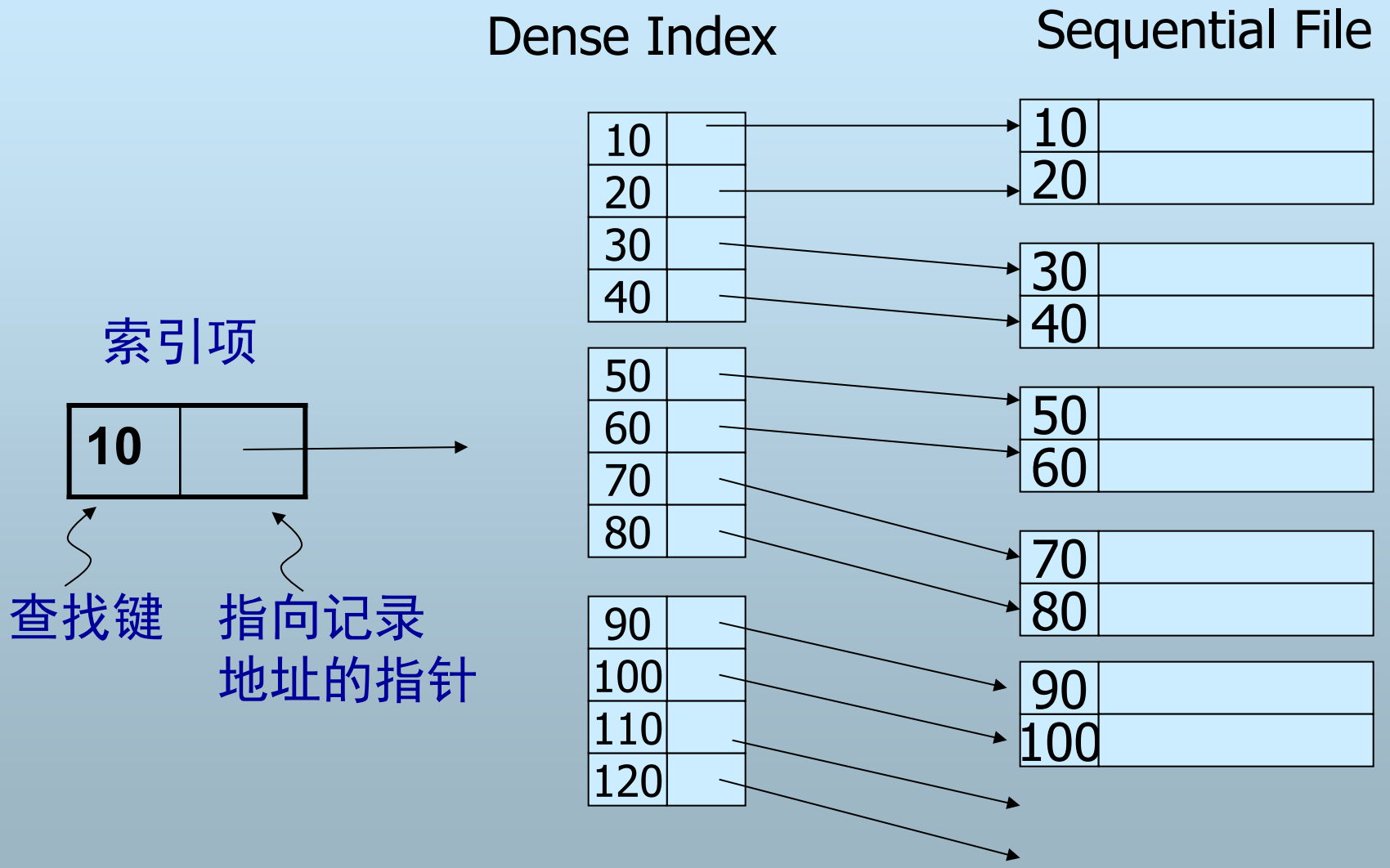
- 顺序文件
  - 记录按查找键排序



1、密集索引(Dense Index)

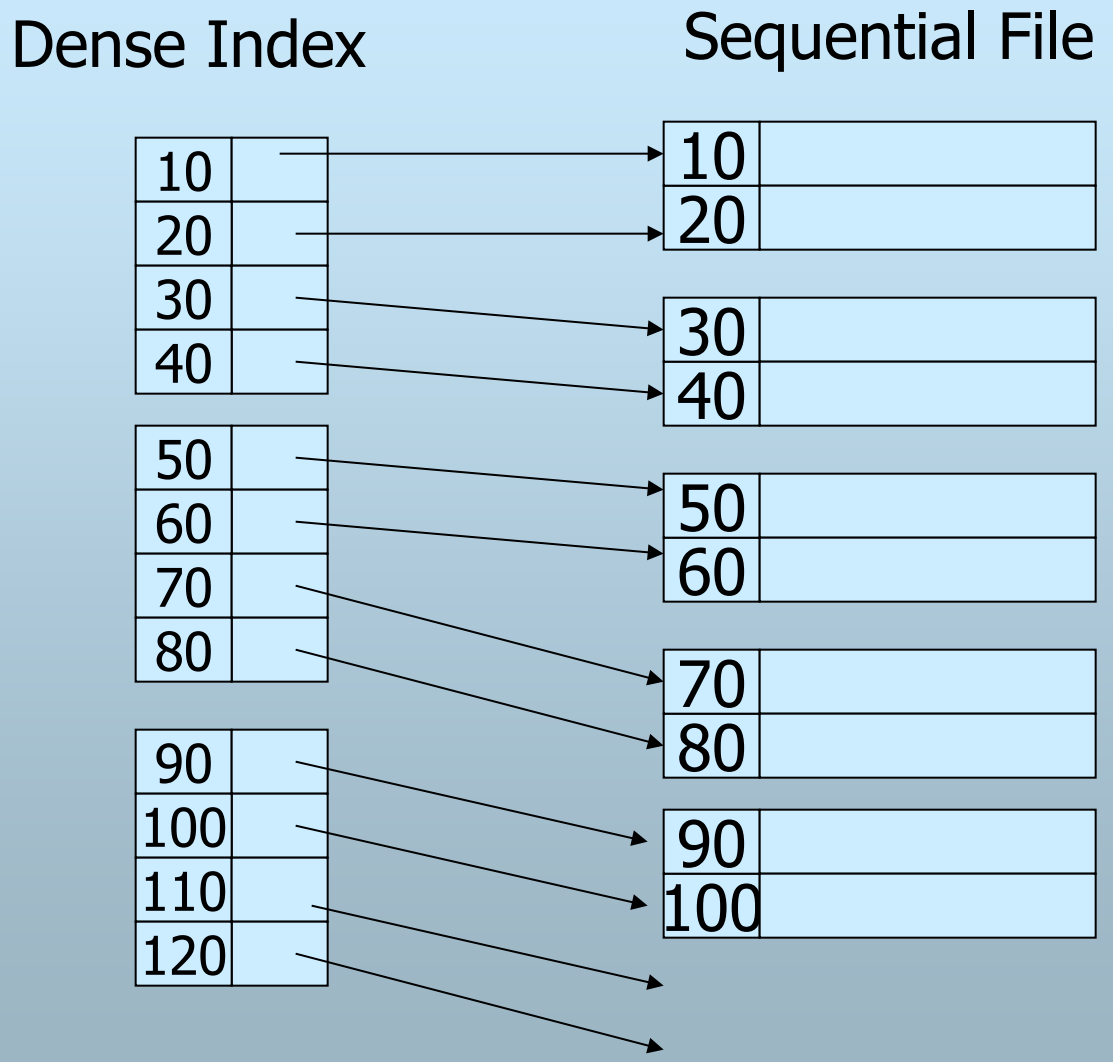
- 每个记录都有一个索引项
- 索引项按查找键排序

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



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

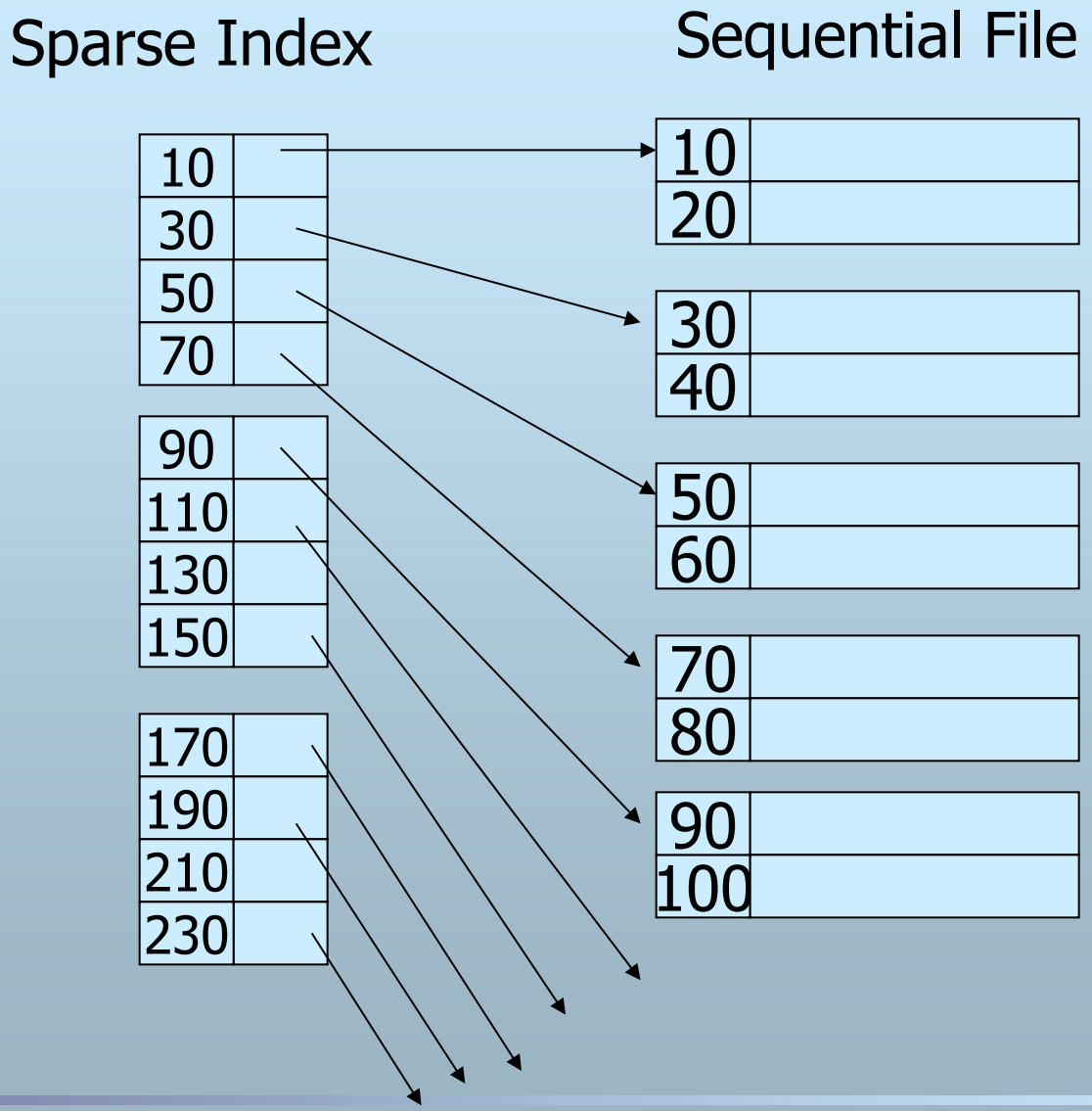
- 密集索引缺点？
  - 索引占用太多空间

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)



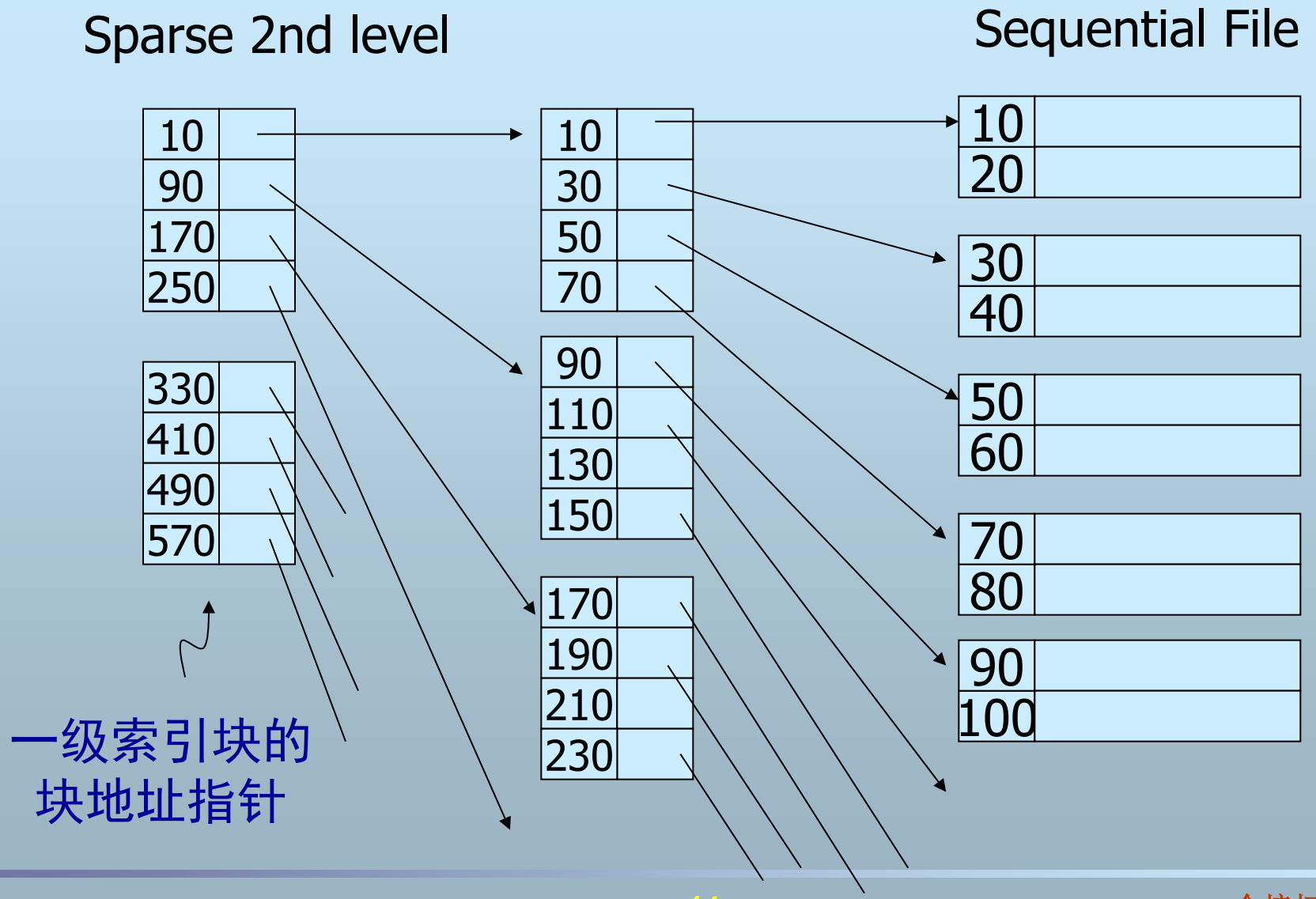
2、稀疏索引(Sparse Index)

- 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)



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 lg10000≈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

- 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（Secondary Index）
  - 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))
```

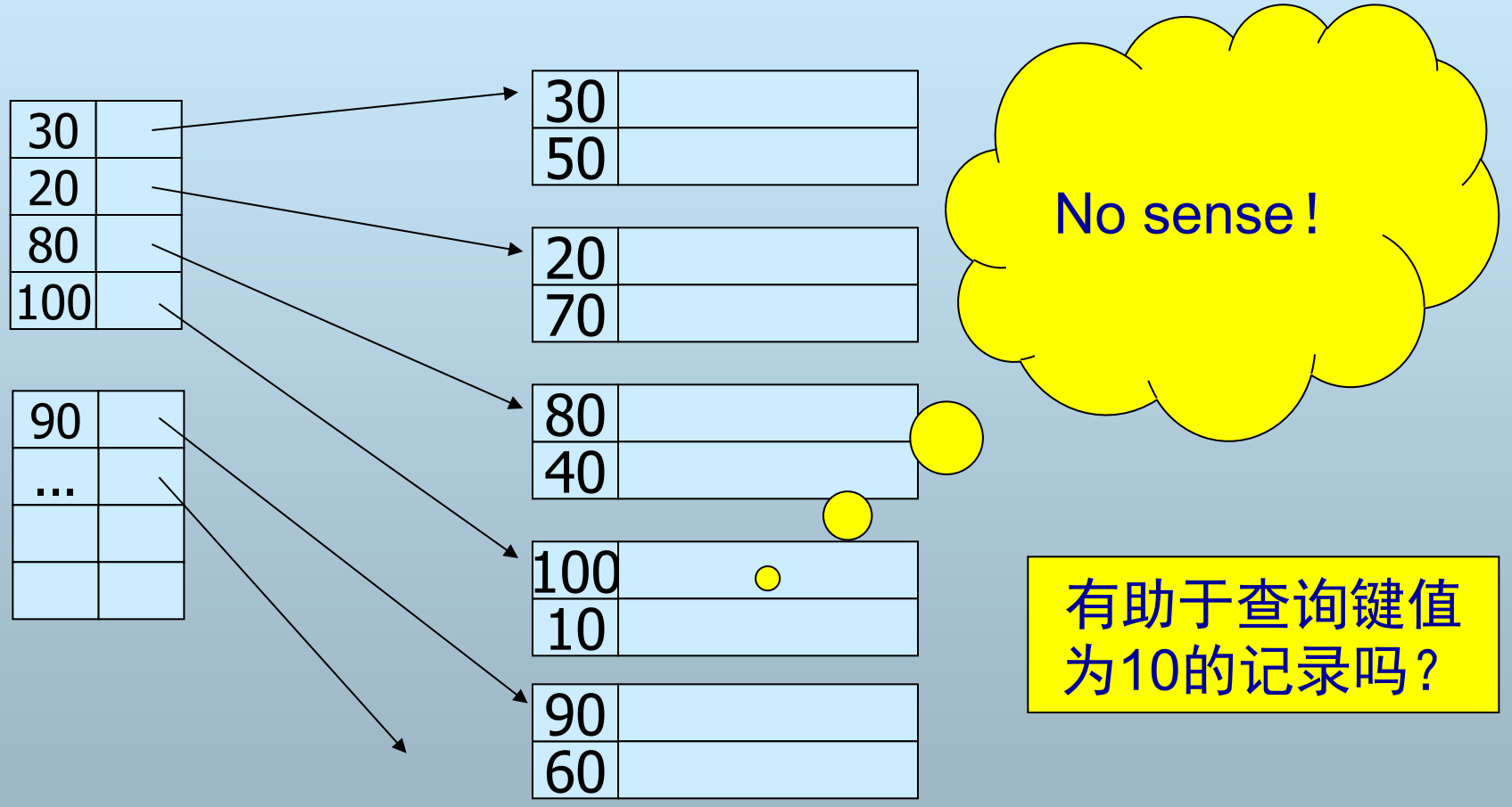
- **Name上创建了主索引，记录按name有序**
- **Address上创建辅助索引**

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

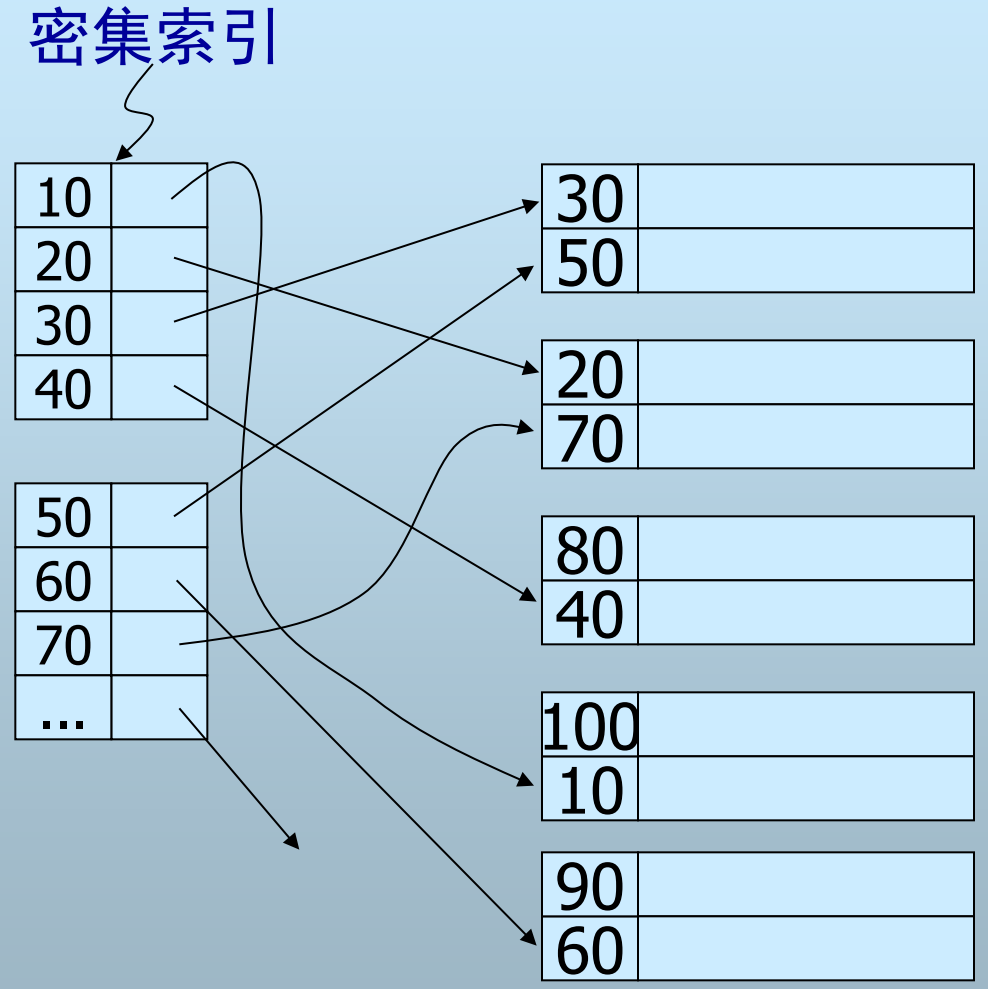
1、辅助索引概念

- 辅助索引只能是密集索引
  - 稀疏的辅助索引有意义吗？

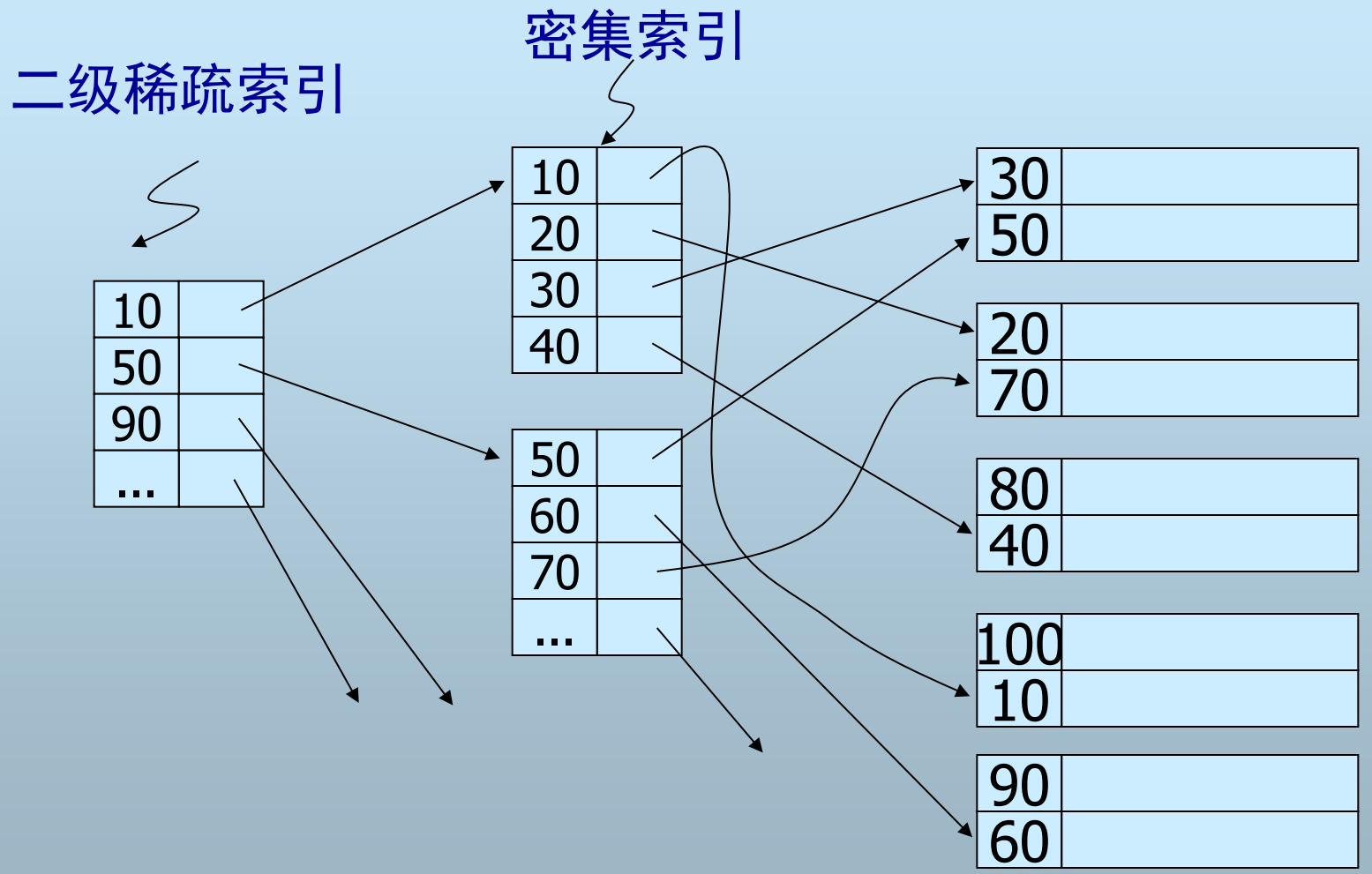
1、辅助索引概念


30
20
80
100
90
...
30
50
20
70
80
40
100
10
90
60
No sense!
有助于查询键值为10的记录吗？

2、辅助索引设计



2、辅助索引设计


二级稀疏索引
密集索引
10
50
90
...
10
20
30
40
50
60
70
...
30
50
20
70
80
40
100
10
90
60

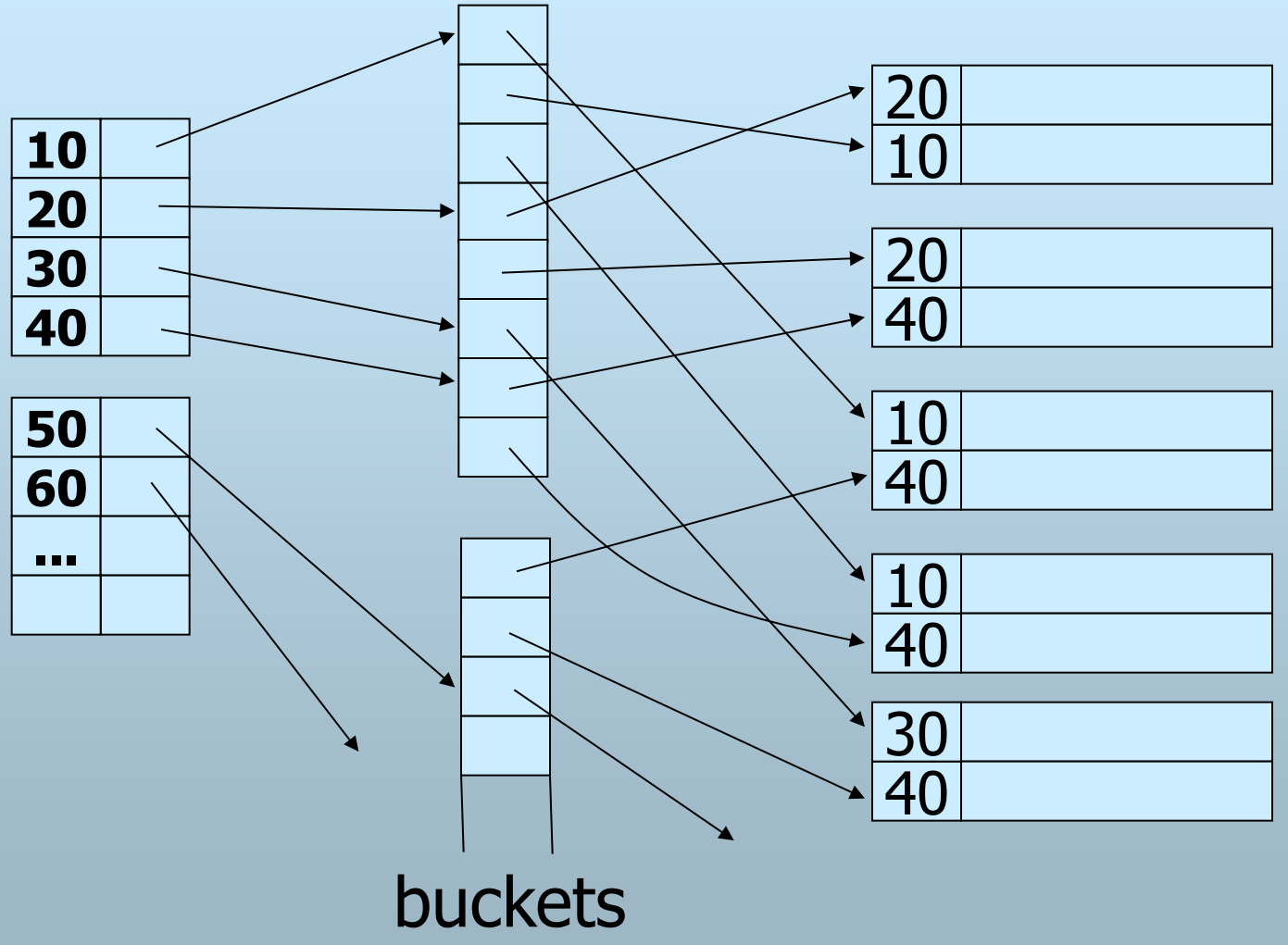
问 题

- 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

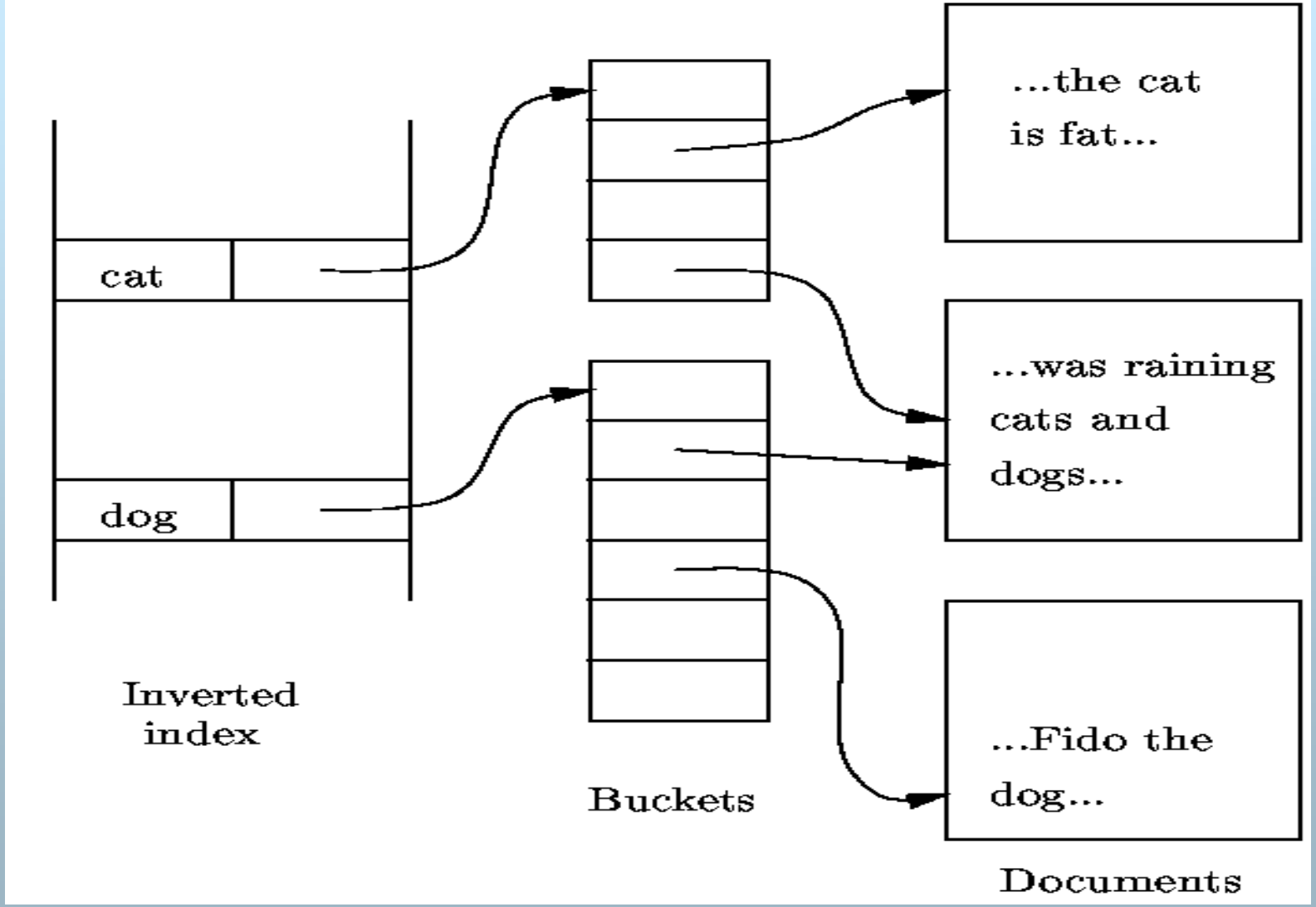
3、辅助索引中的间接桶



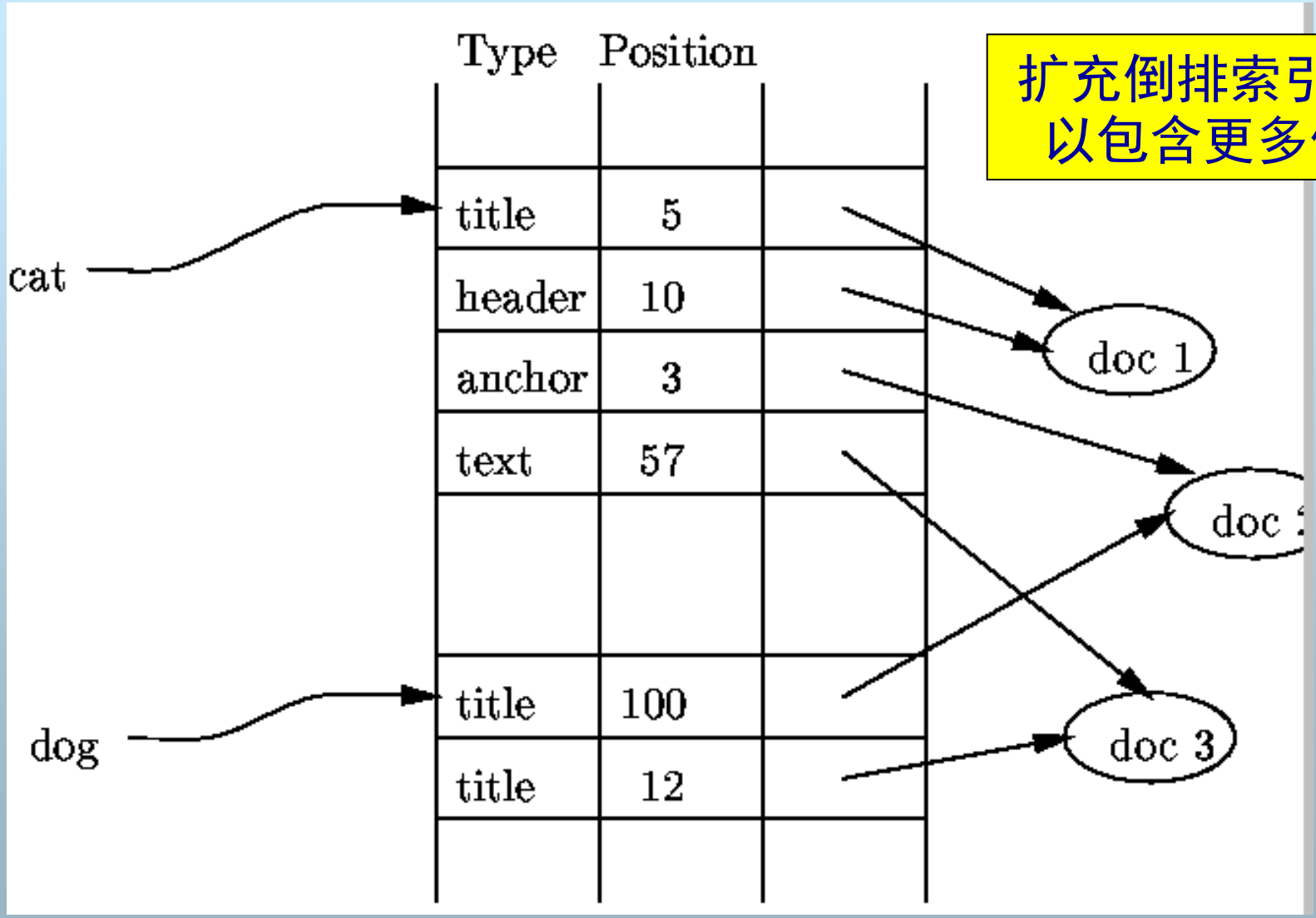
4、倒排索引(Inverted Index)

- 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)



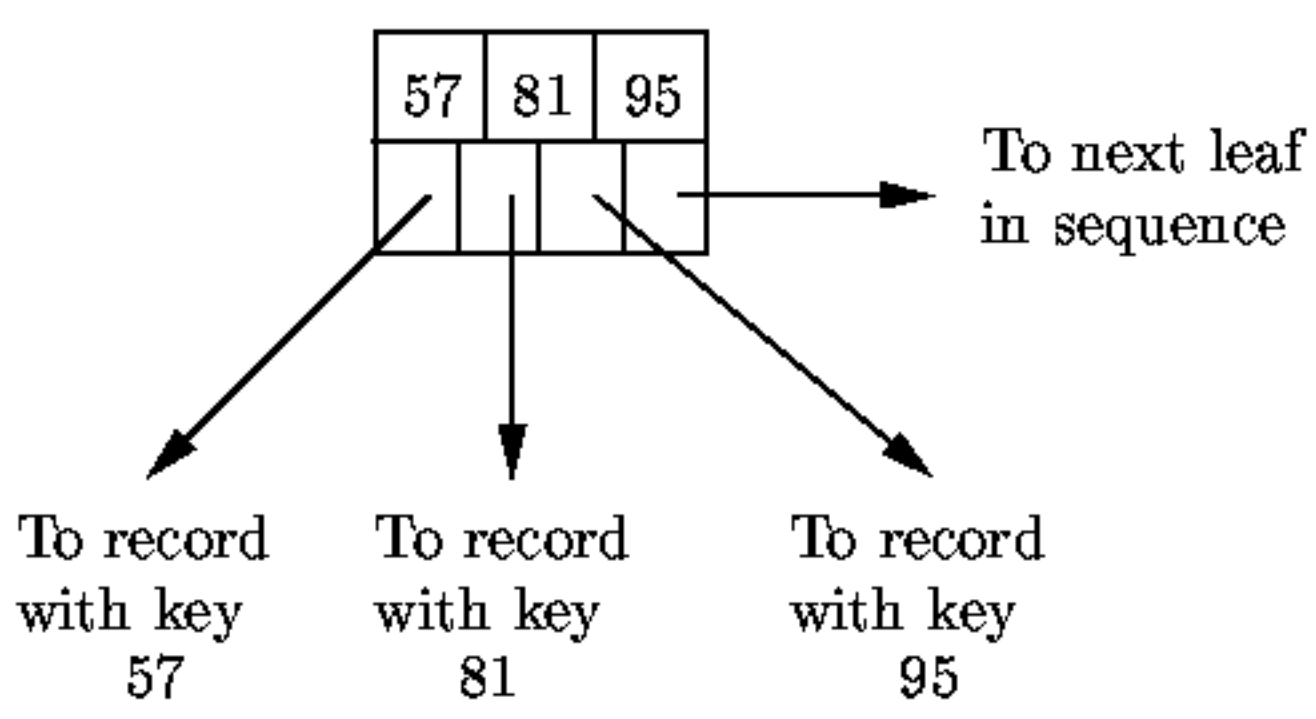
4、倒排索引(Inverted Index)



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

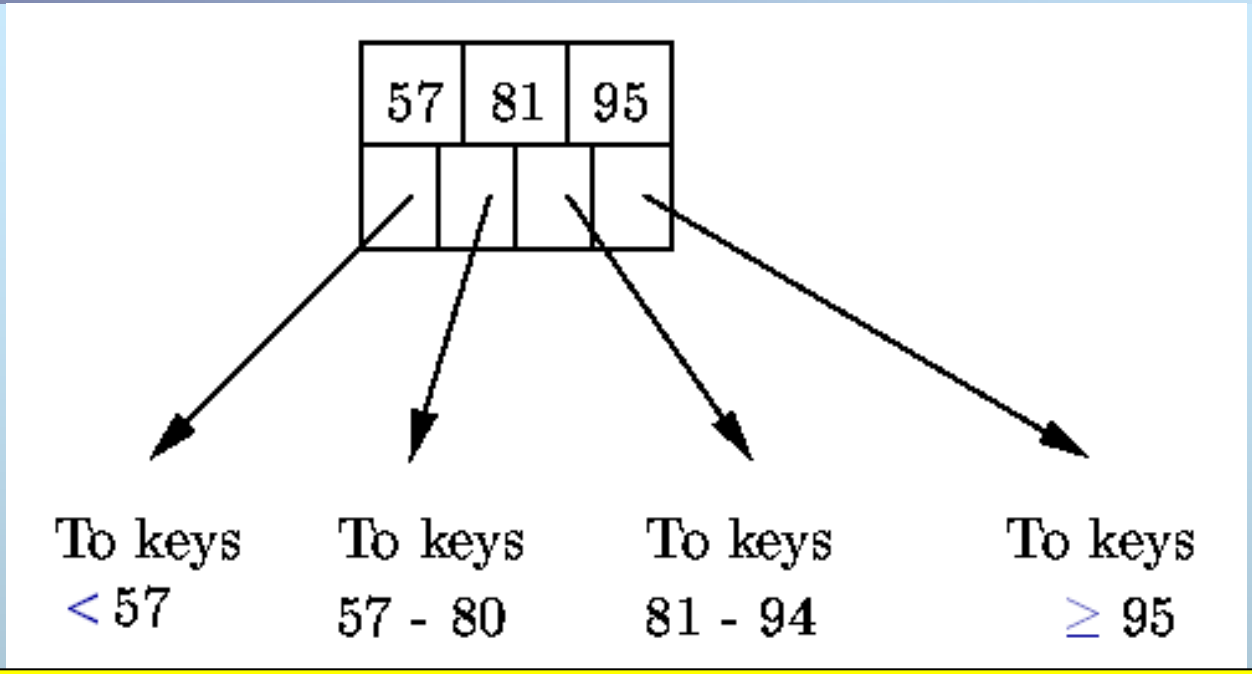
1、叶结点



- 1个指向相邻叶结点的指针
- n对键－指针对

- 至少⌊(n+1)/2⌋个指针指向键值

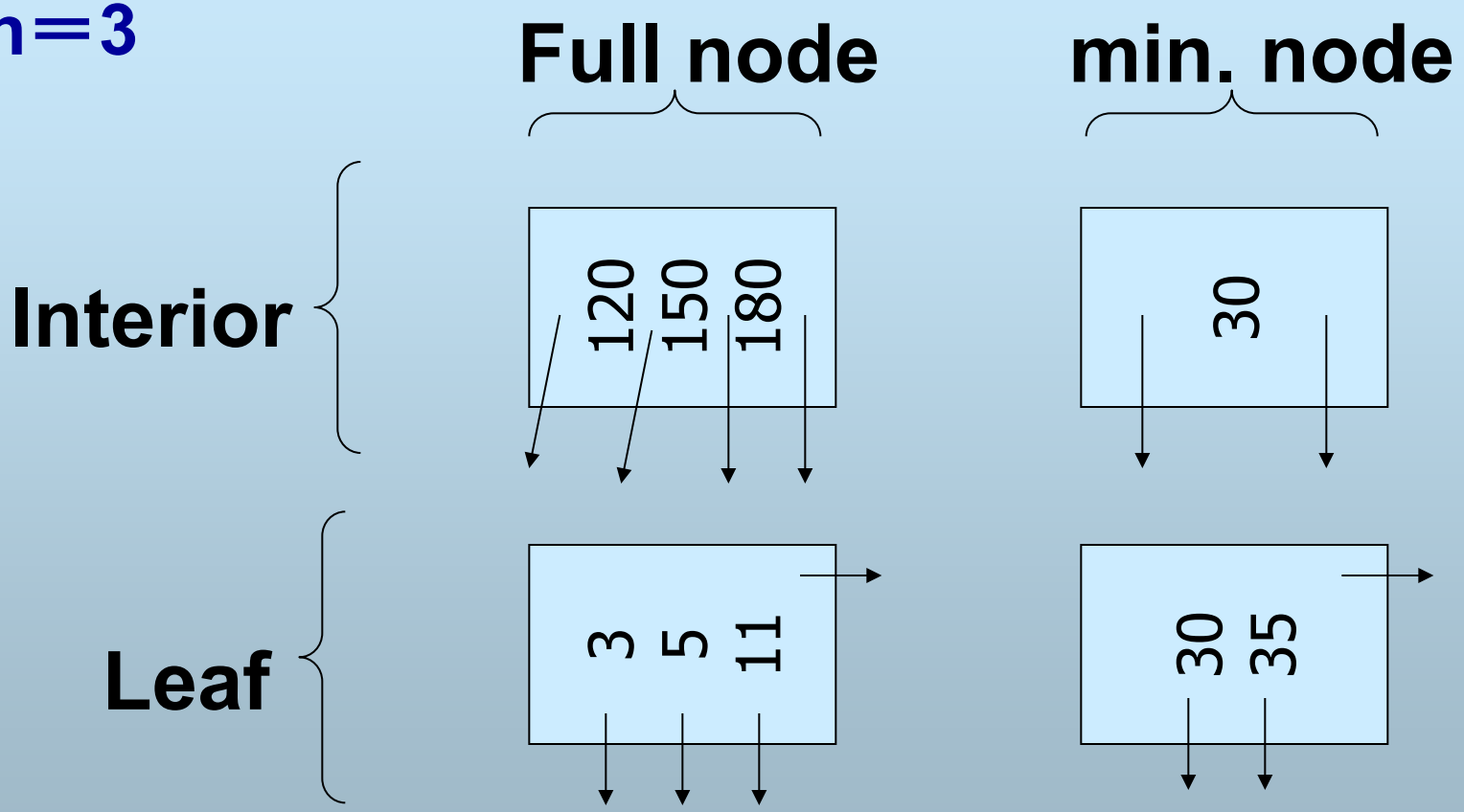
2、中间结点



- 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⌈(n+1)/2⌉个指针指向子树
- 根结点至少 2 个指针

B+树结点例子

n＝3



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

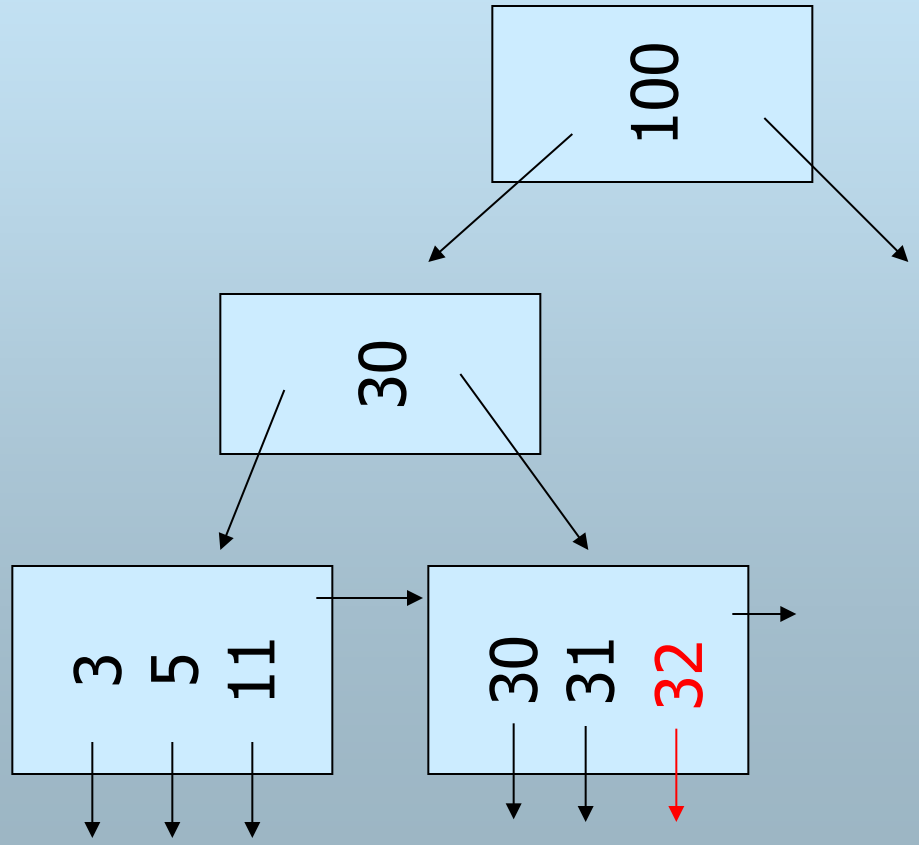
4、B+树插入

- 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

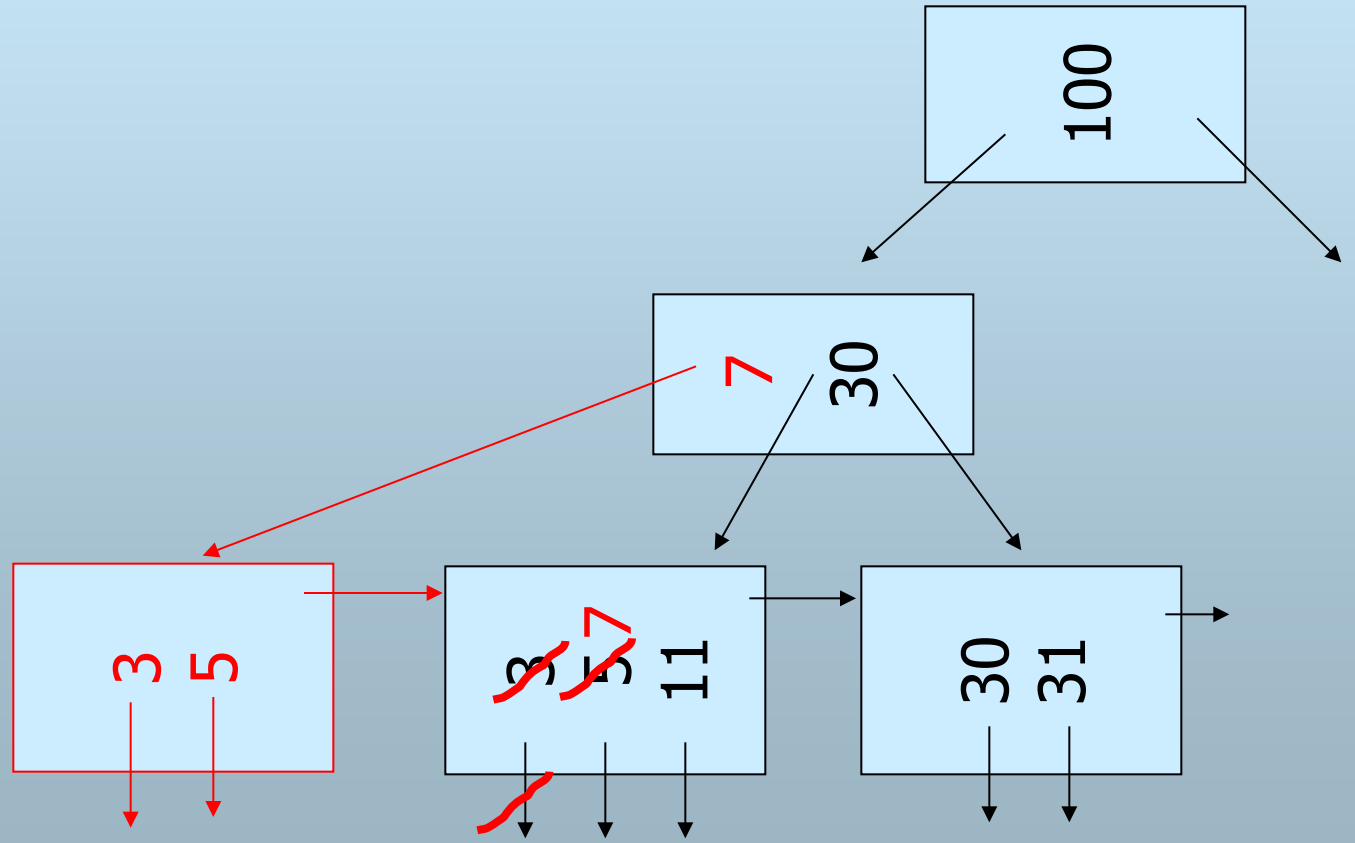
n=3



B+树插入例子

(b) Insert key = 7

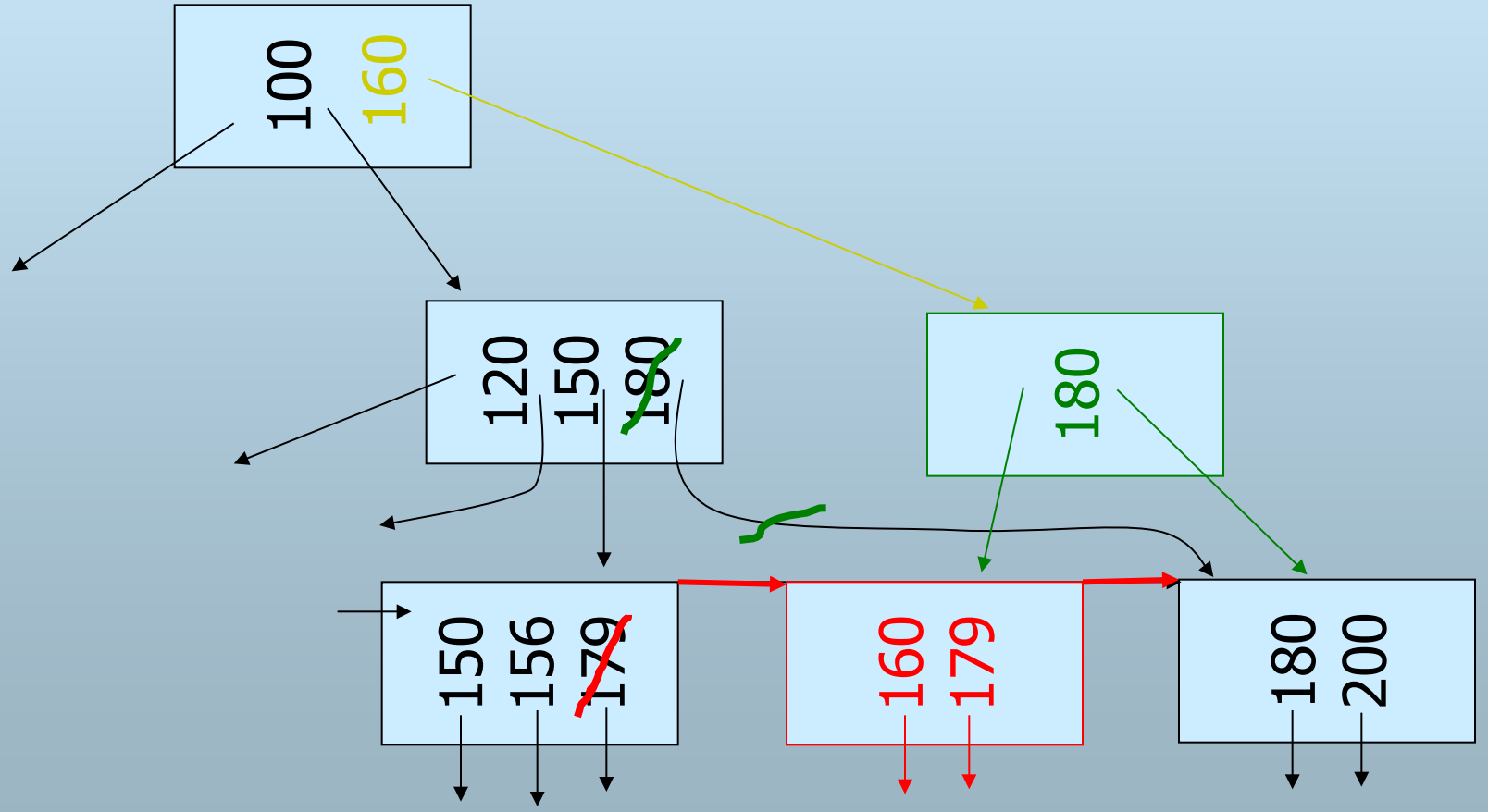
n=3



B+树插入例子

(c) Insert key = 160

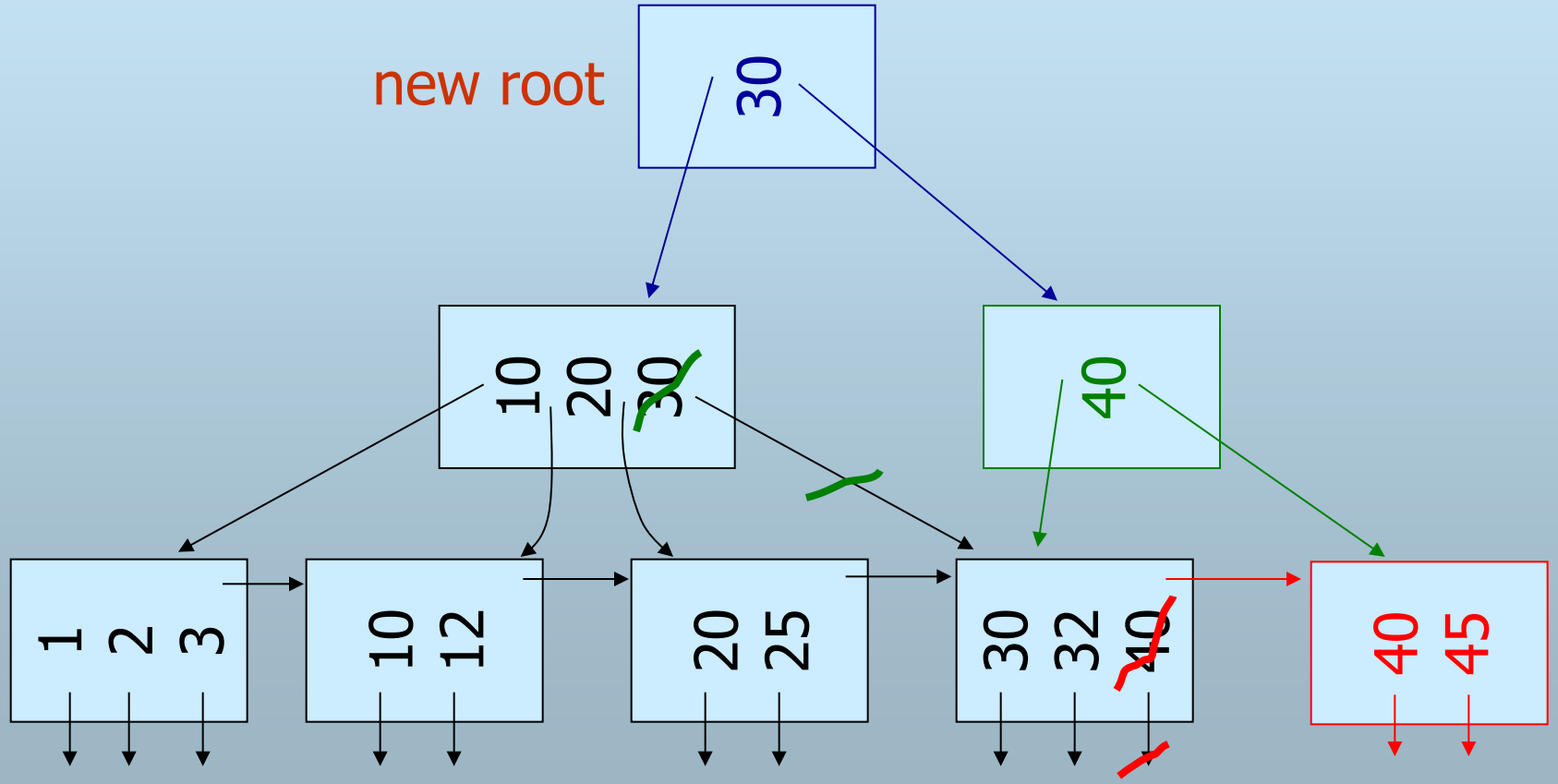
n=3



B+树插入例子

(d) New root, insert 45

n=3



5、B+树删除

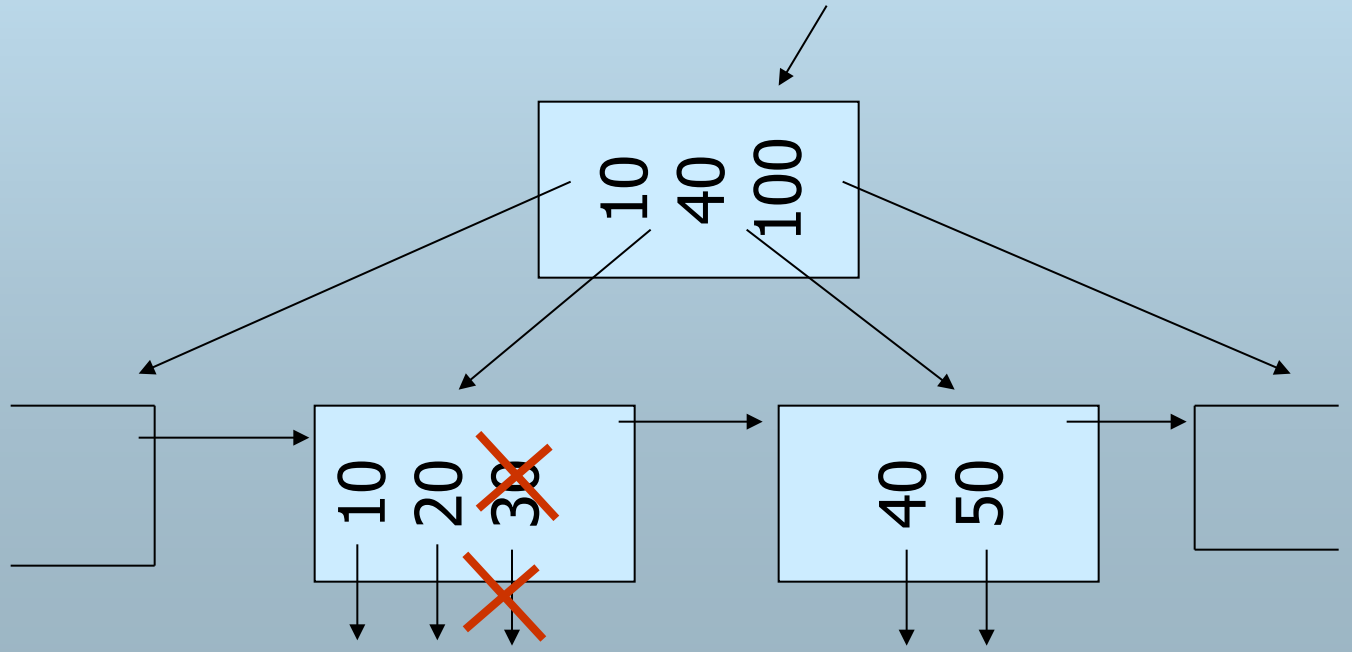
- 查找要删除的键值，并删除之
- 若结点的键值填充低于规定值，则调整
  - 若相邻的叶结点中键填充高于规定值，则将其中一个键值移到该结点中
  - 否则，合并该结点与相邻结点
    - 合并可视作在父结点中删除一个子结点
  - 递归向上删除
- 若删除的是叶结点中的最小键值，则需对父结点的键值加以调整

B+树删除例子

(a) Simple delete

- **Delete 30**

n=4

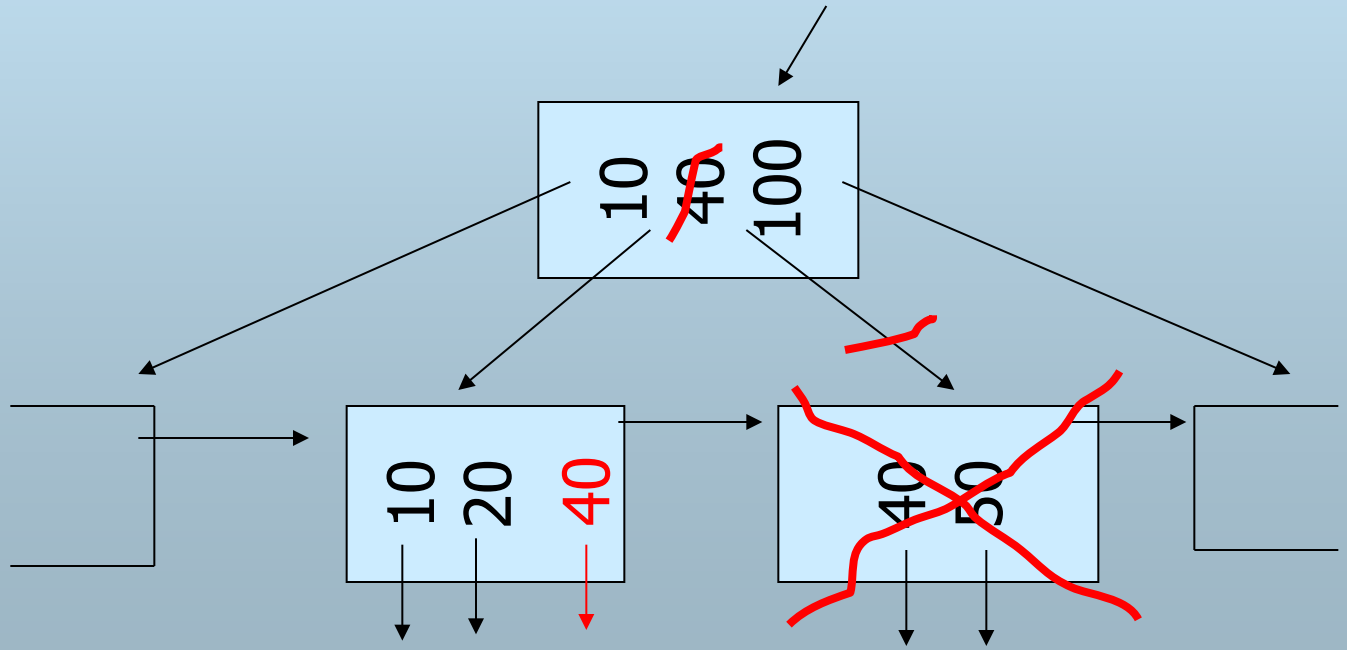


B+树删除例子

(b) Coalesce with sibling

n=4

- **Delete 50**

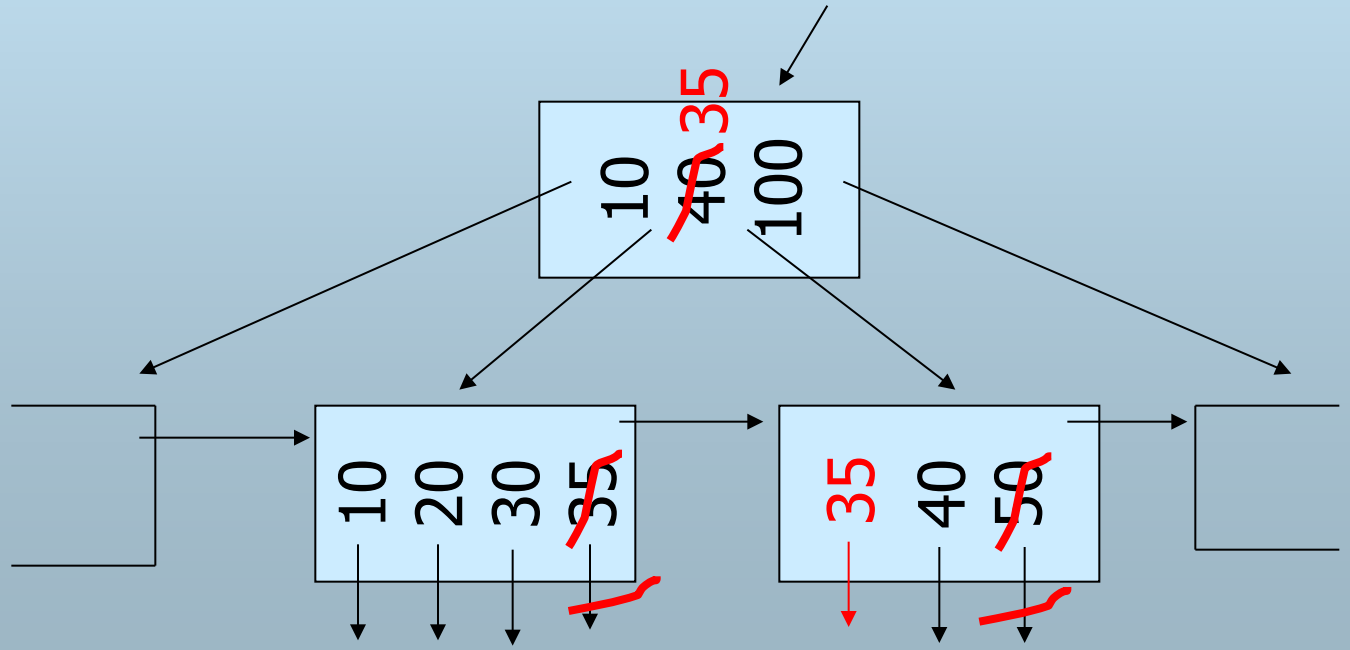


B+树删除例子

(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）

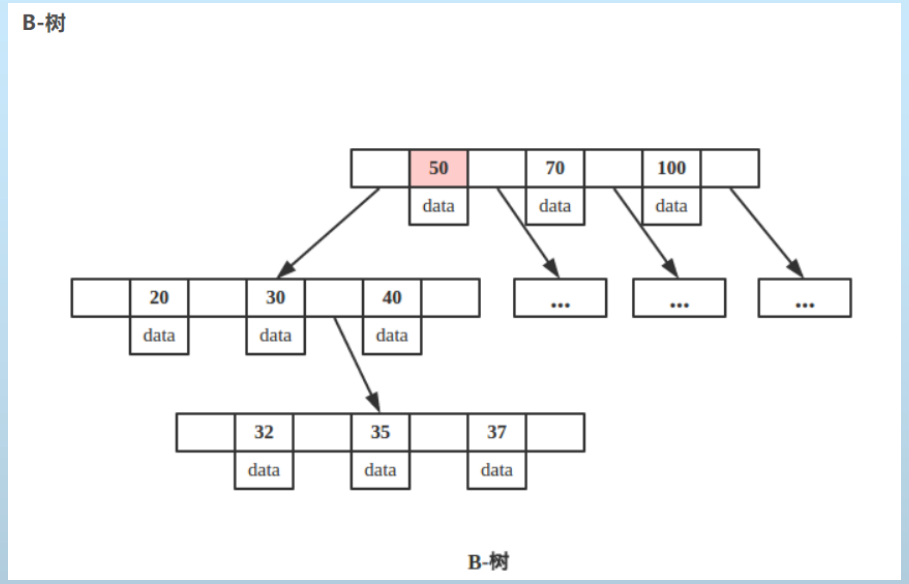
6、B＋树的效率

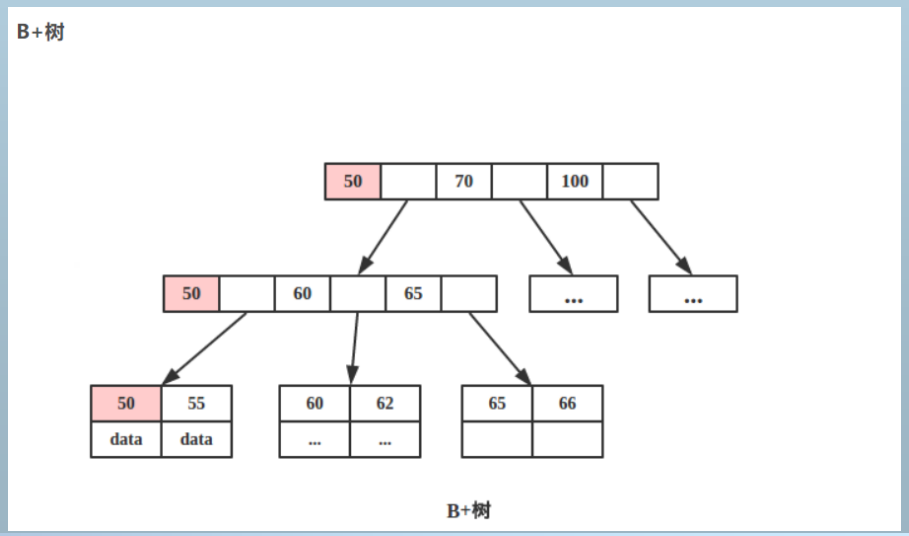
- 设块大小8KB，键2B（smallint），指针2B，则一个块可放2048个索引项

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

7、B树 vs. B+树

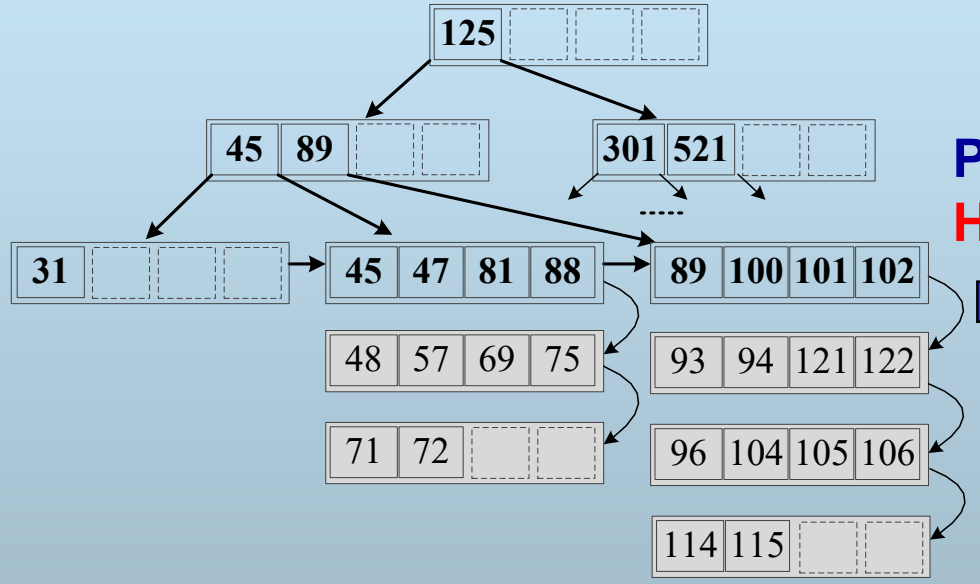
- **B-tree**
  - 所有节点都存储实际的数据（记录）
  - 键值无重复存储
  - 搜索有可能在非叶子结点结束
  - 是数据存储的一种文件结构
- **B+-tree**
  - 节点本身不存储数据





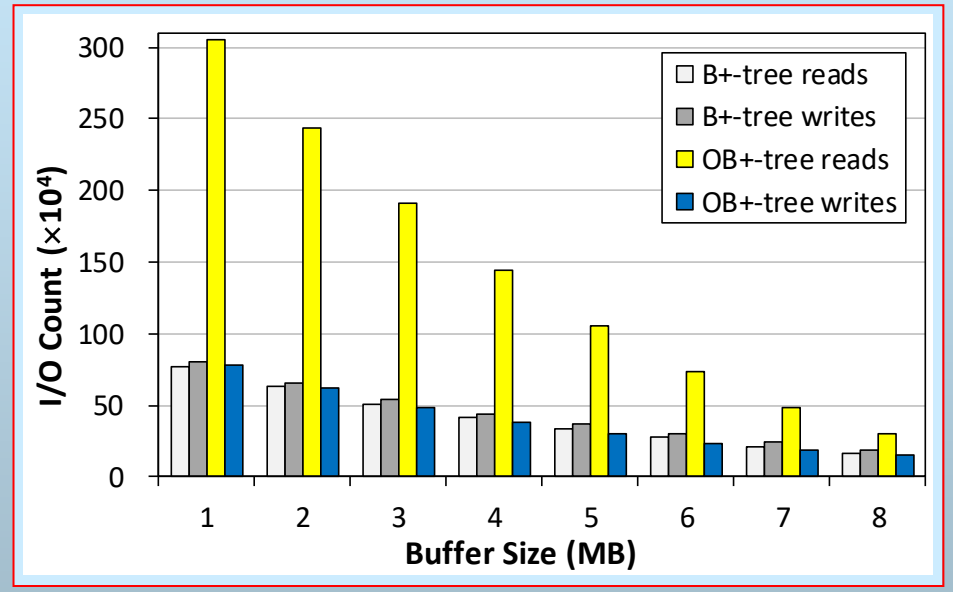
8、B+树的改进

- 如何提高B+树的写性能？



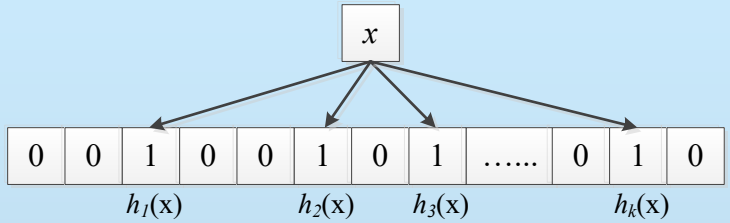
Overflow B+-tree

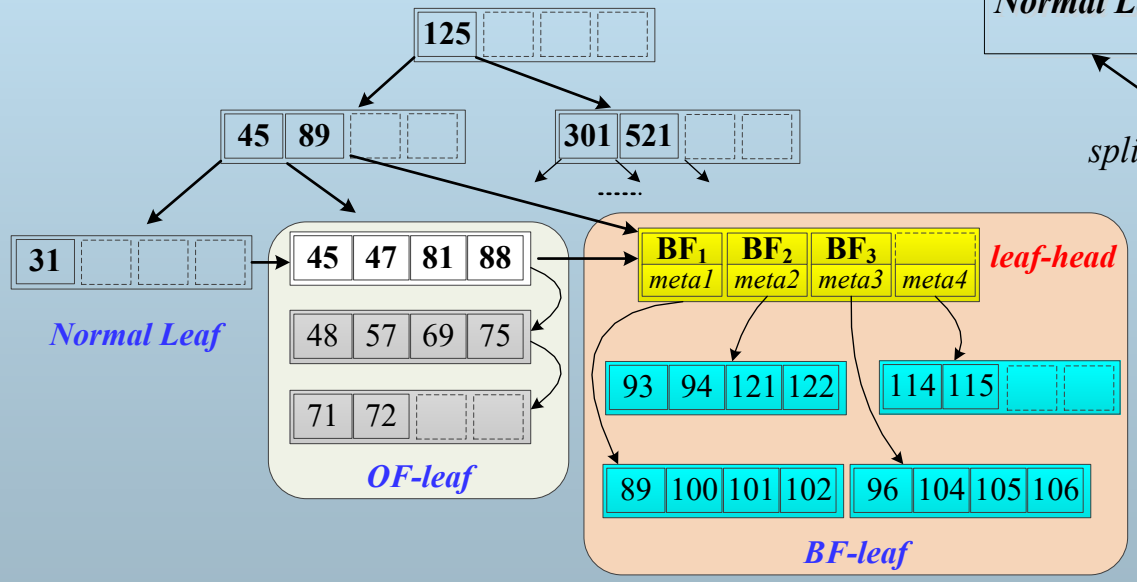
Problem:
High Read Cost

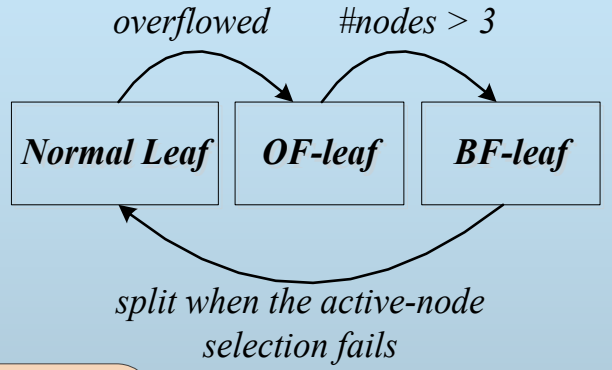


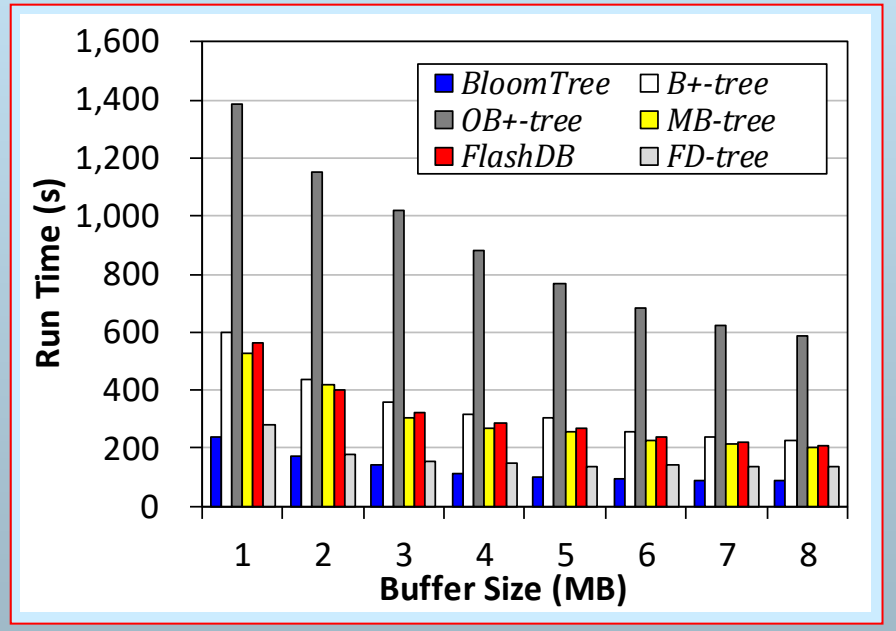
8、B+树的改进

- **BloomTree: Read/Write-Optimized B+-tree**
  - **Using Bloom Filters to reduce the read cost of Overflow Leaf**







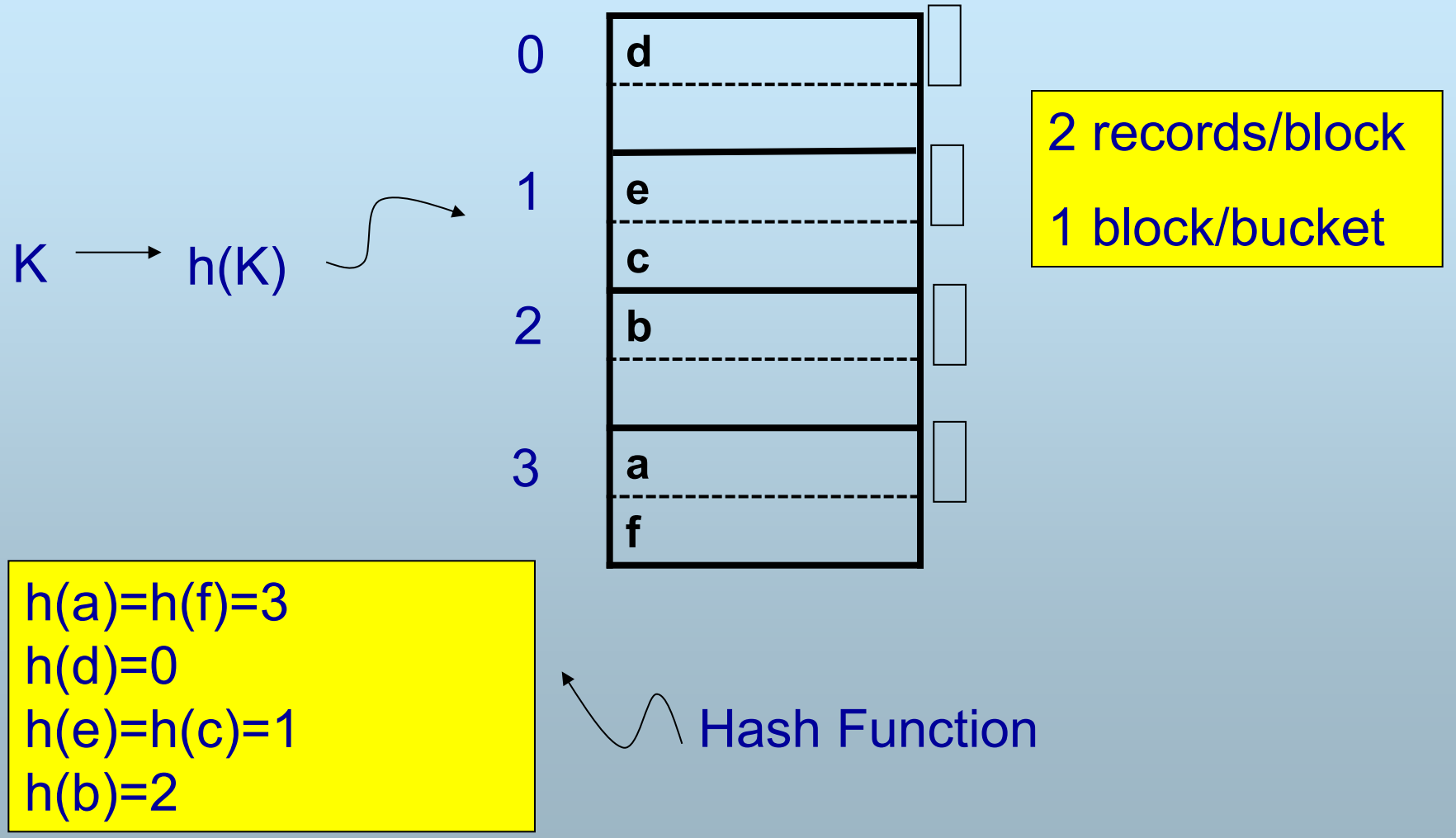


Peiquan Jin, Chengcheng Yang, et al. Read/write-optimized tree indexing for solid-state drives. VLDB J. 25(5): 695-717 (2016)

三、散列表(Hash Table)

- 散列函数(Hash Functions)
  - h：查找键(散列键) → [0...B – 1]
  - 桶(Buckets), numbered 0,1,..., 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


0
1
2
3
d
e
c
b
a
f
K
h(K)
2 records/block
1 block/bucket
h(a)=h(f)=3
h(d)=0
h(e)=h(c)=1
h(b)=2
Hash Function

2、散列表查找

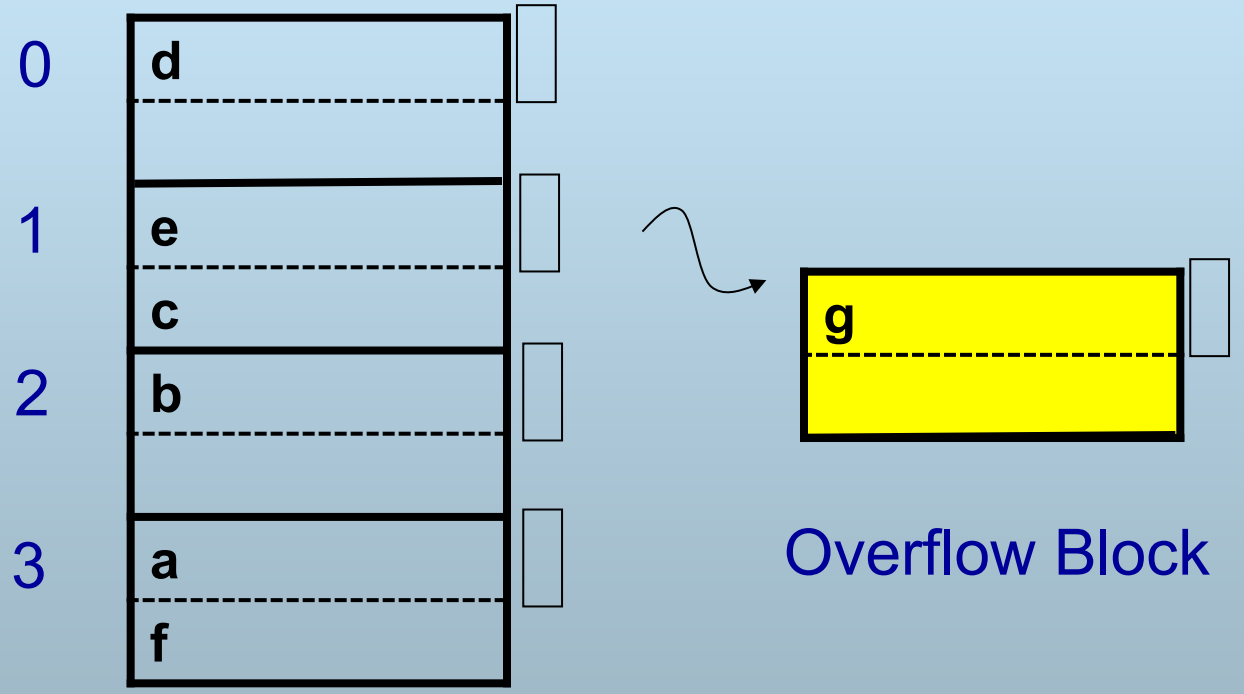
- 查找
  - 对于给定的散列键值k，计算h(K)
  - 根据h(K)定位桶
  - 查找桶中的块

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

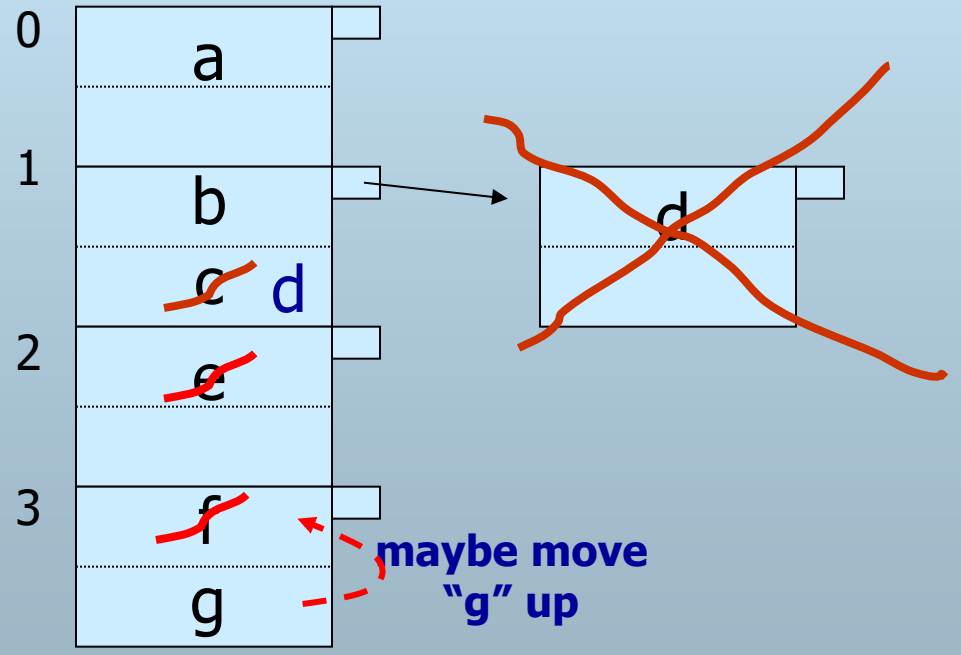
EXAMPLE: deletion

Delete:

e

f

c



4、散列表空间利用率问题

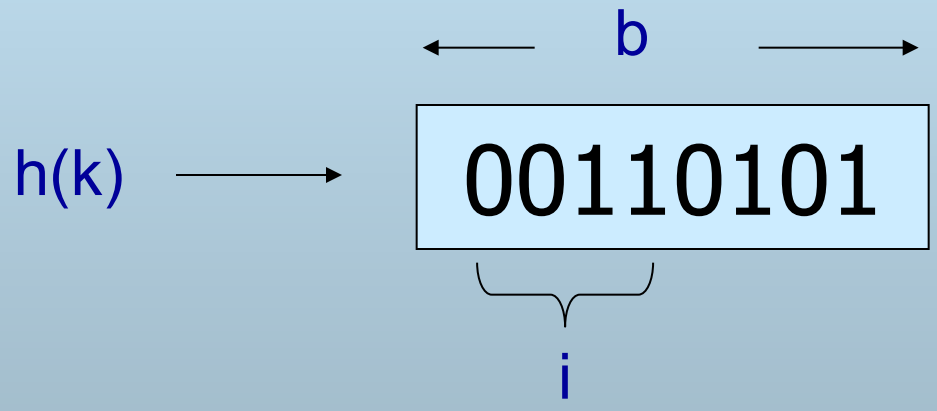
- 空间利用率
  - 实际键值数 / 所有桶可放置的键值数
  - <50%：空间浪费
  - >80%：溢出问题
  - 50%到80%之间（GOOD！）

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

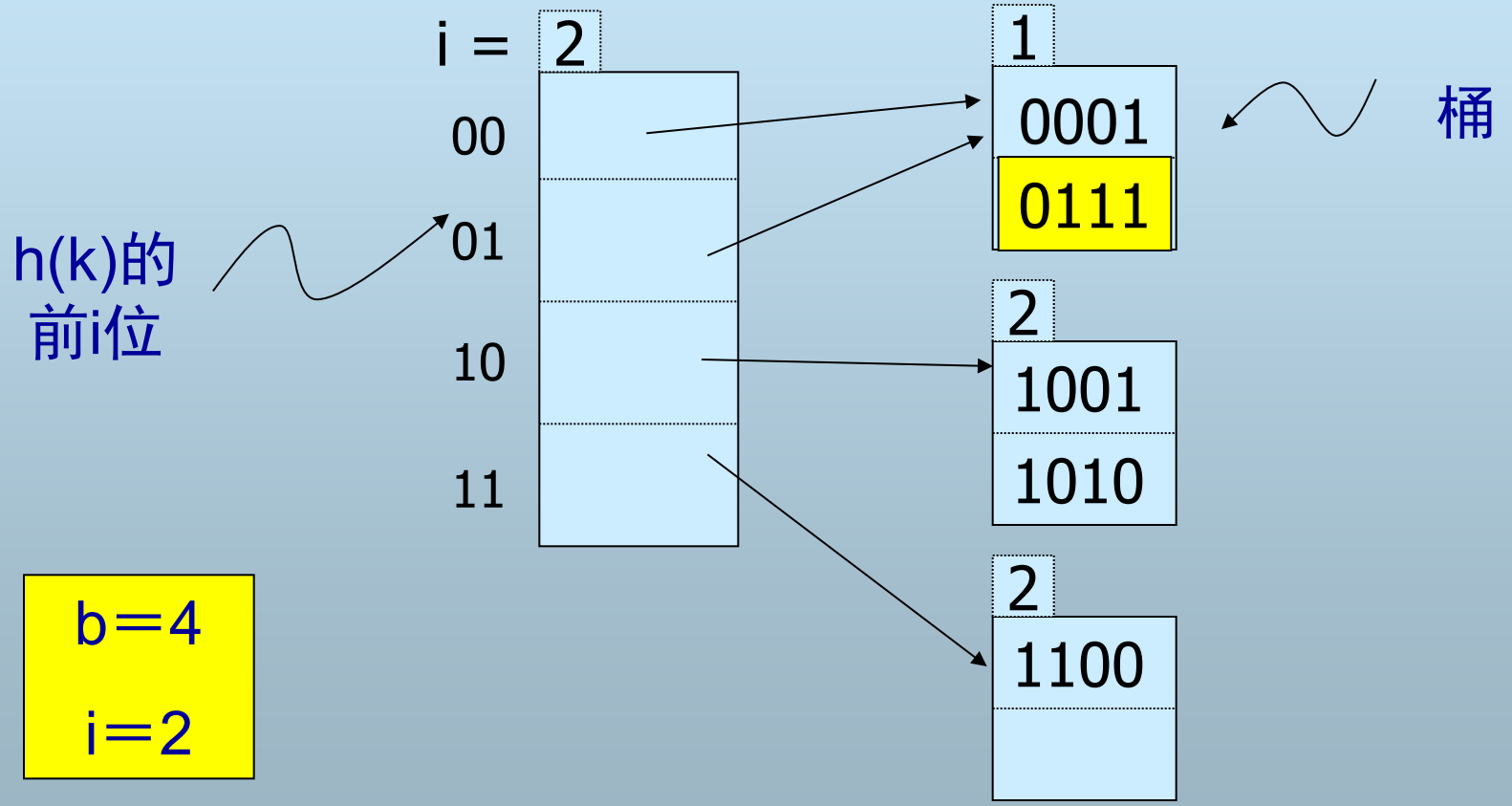
- 散列函数h(k)是一个b(足够大)位二进制序列，前i位表示桶的数目。
- i的值随数据文件的增长而增大



b位二进制序列，
前i位用于区分桶

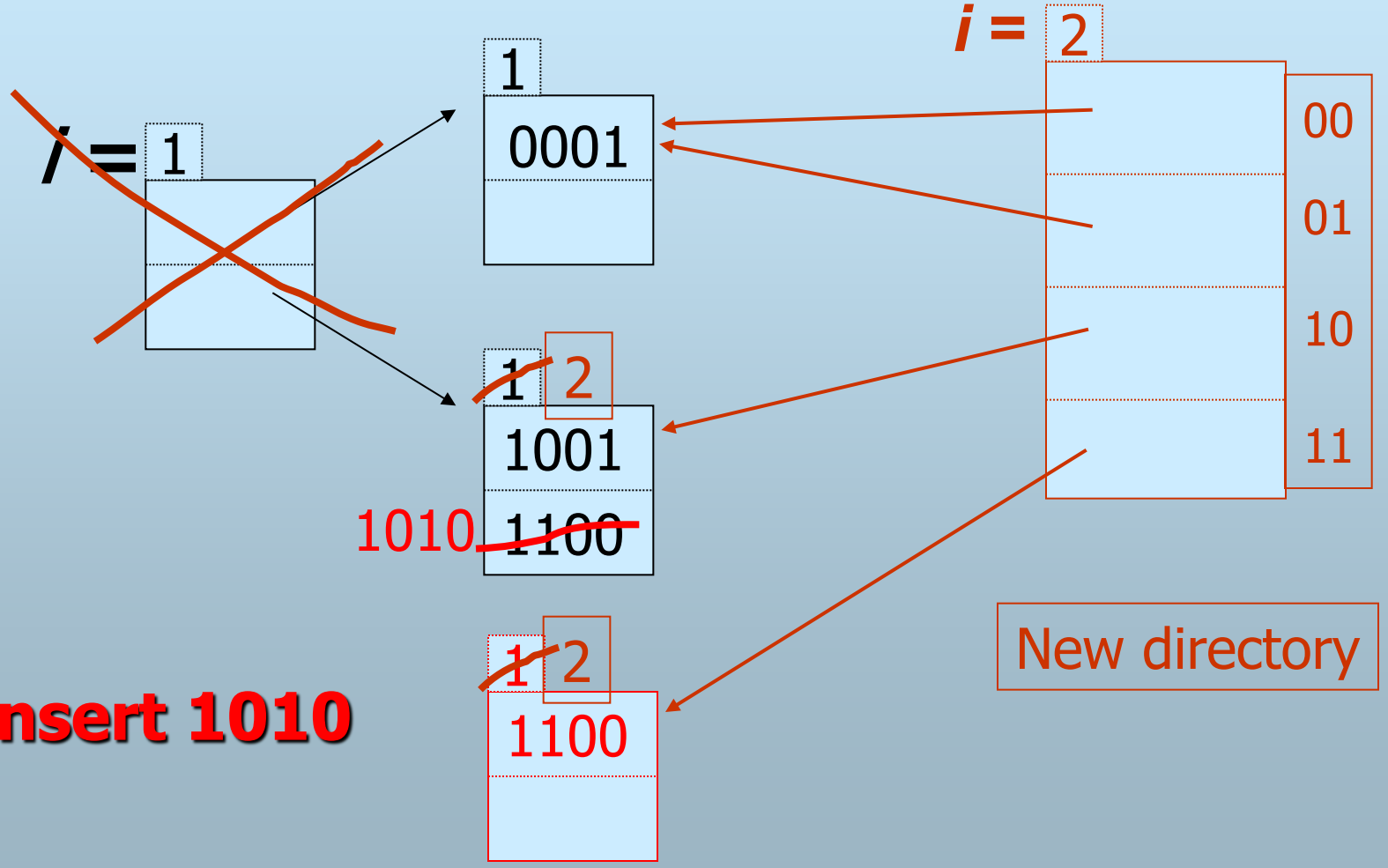
6、可扩展散列表

- 前i位构成一个桶数组



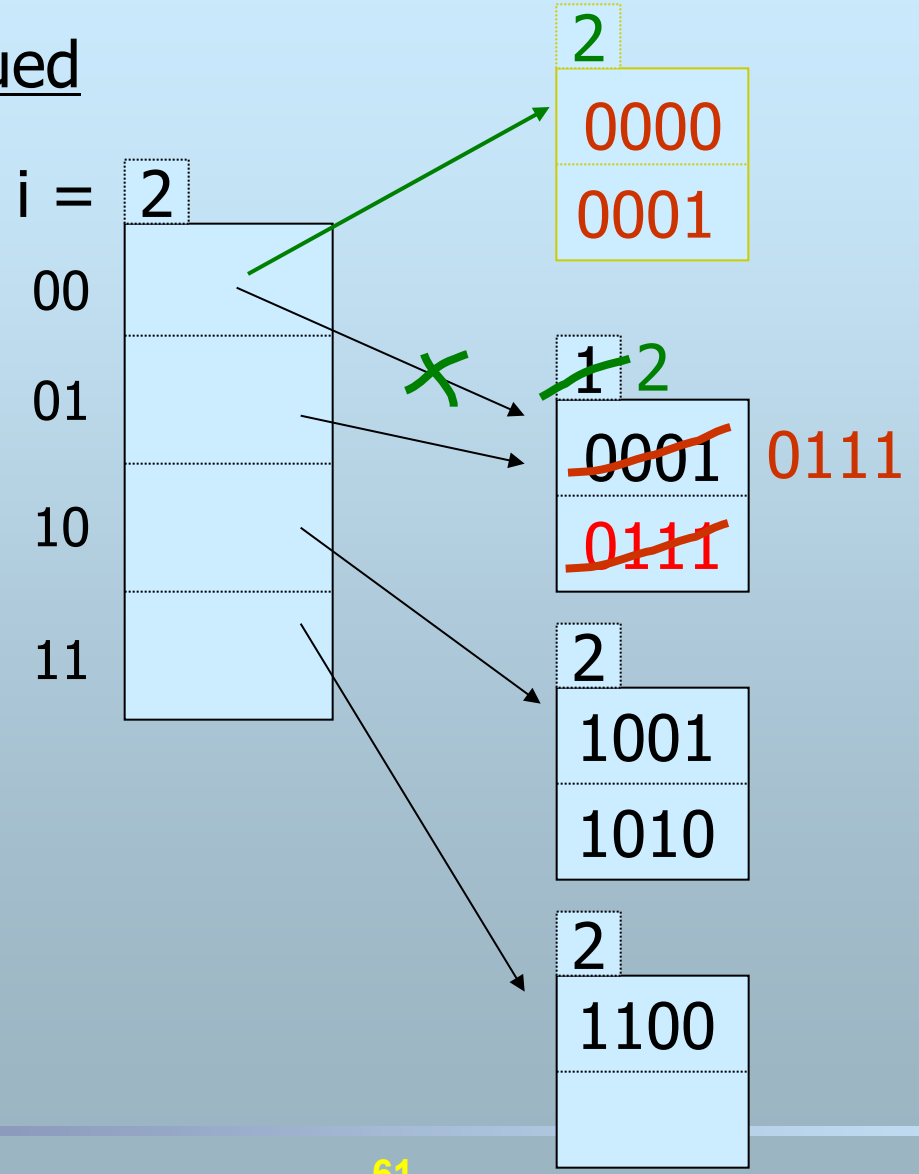
6、可扩展散列表

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



6、可扩展散列表

Example continued

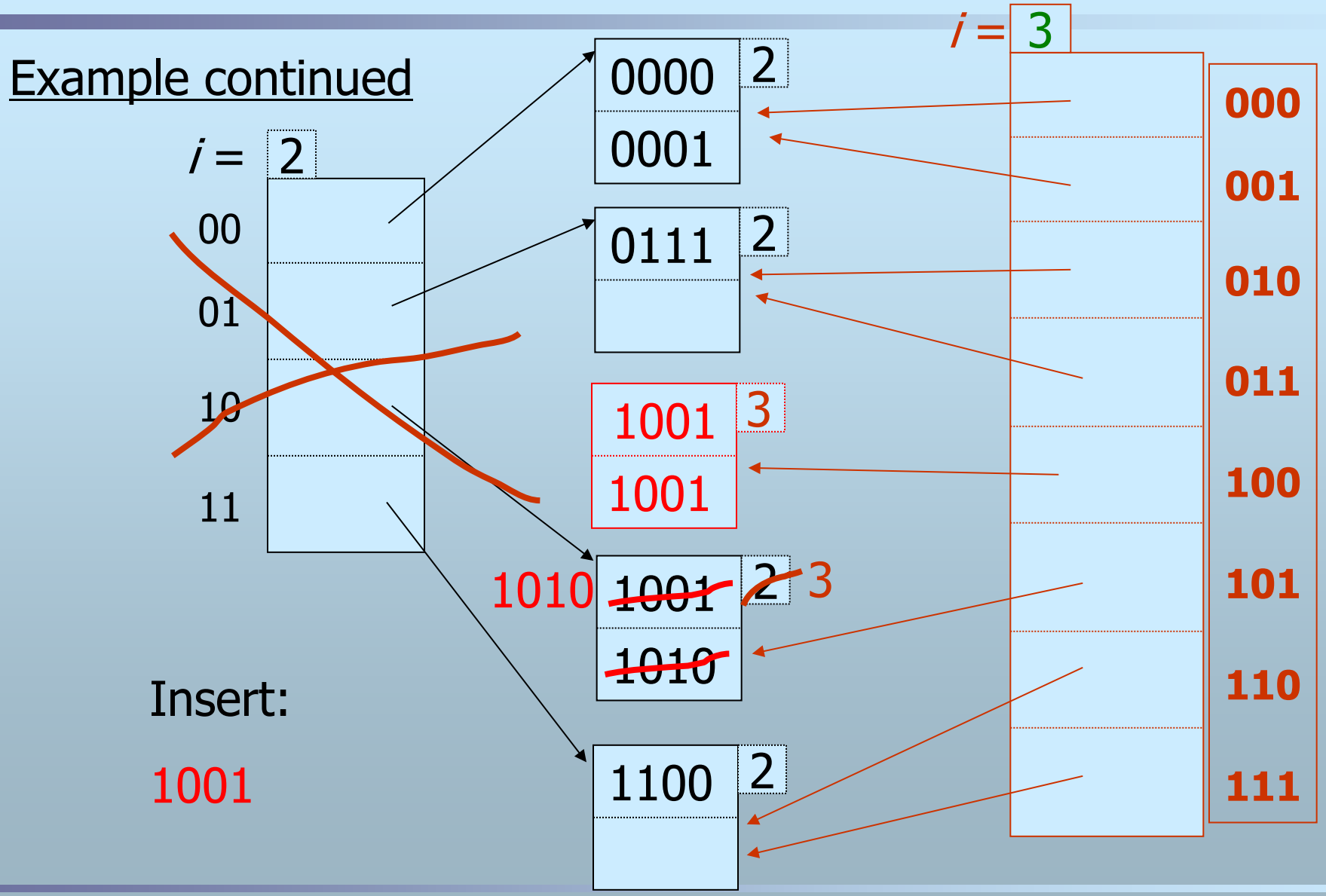


Insert:

0111

0000

6、可扩展散列表


Example continued
i = 2
00
01
10
11
0000
0001
2
0111
2
1001
1001
3
1010
1001
1010
2
3
1100
2
i = 3
000
001
010
011
100
101
110
111
Insert:
1001

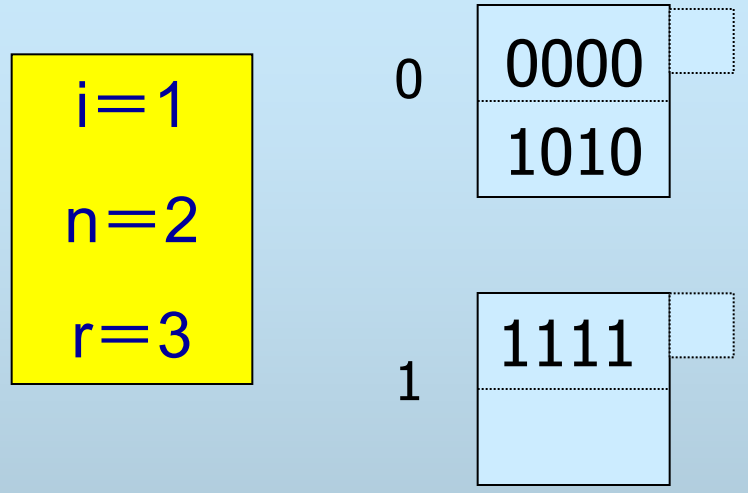
6、可扩展散列表

- 优点
  - 大部分情况下不存在着溢出块，因此当查找记录时，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$n \le m < 2^i$,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



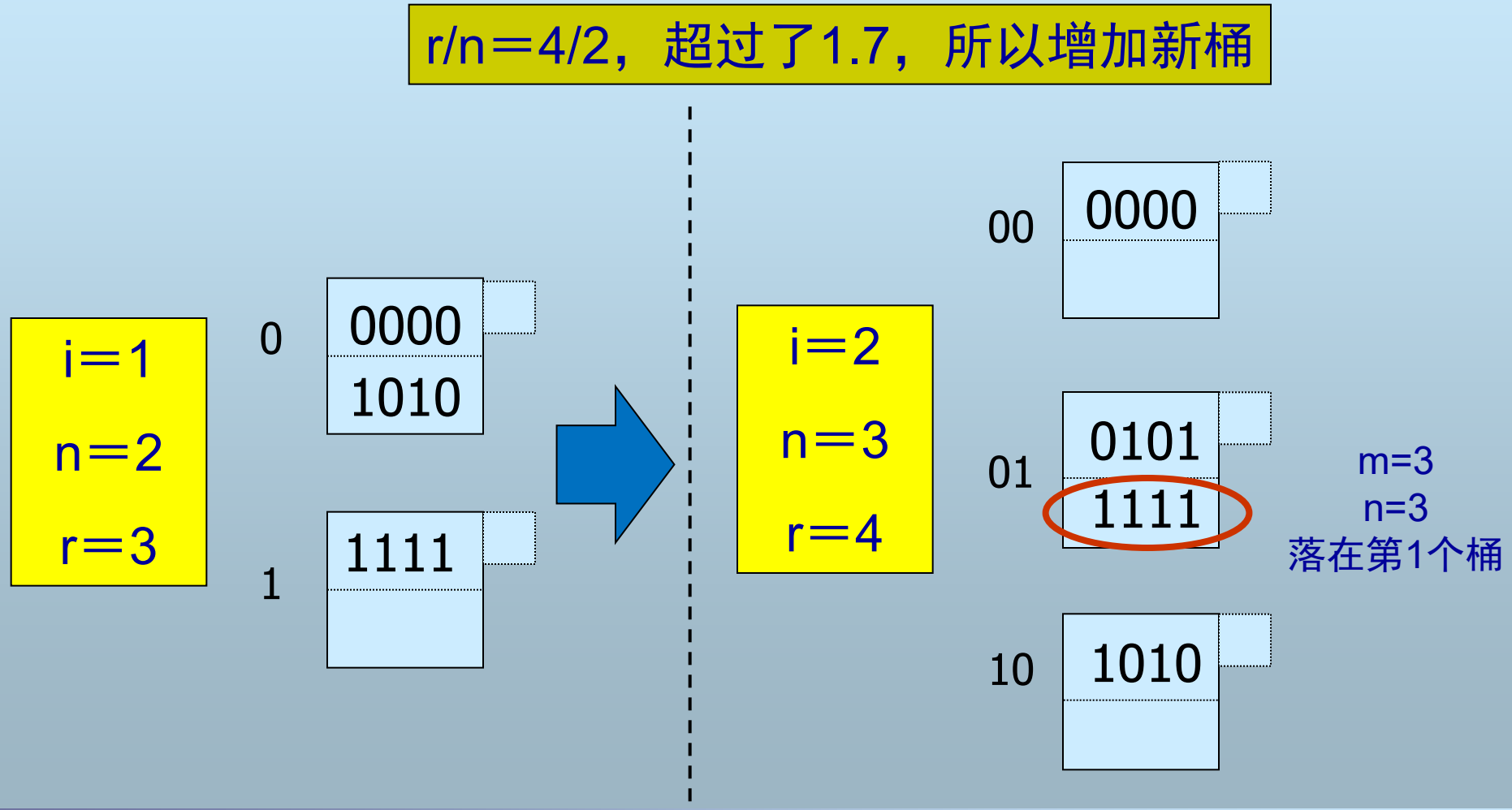
i：当前被使用的散列函数值的位数，从低位开始

n：当前的桶数

r：当前散列表中的记录总数　r/n ＜ 1.7

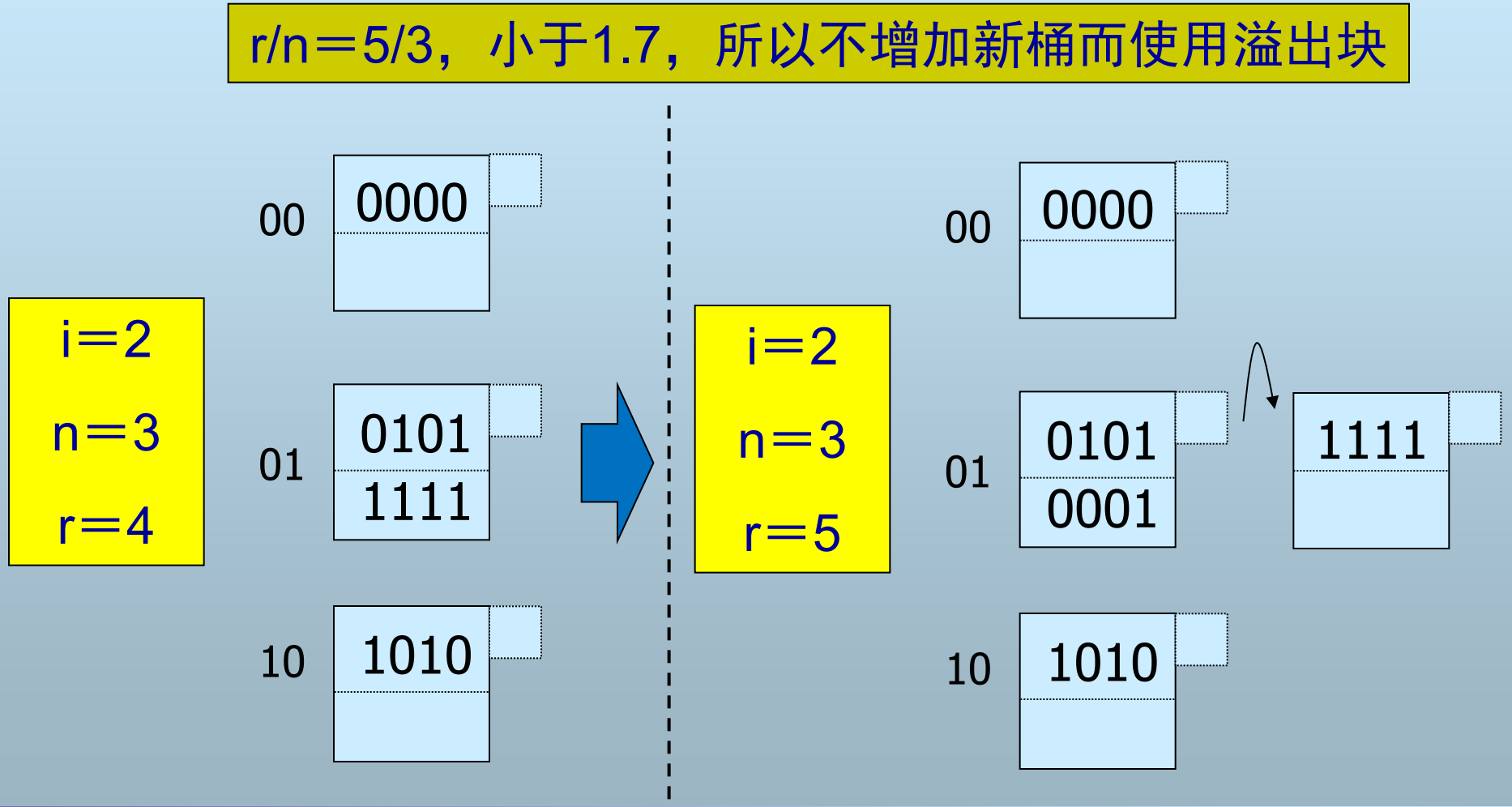
7、线性散列表

- 插入0101



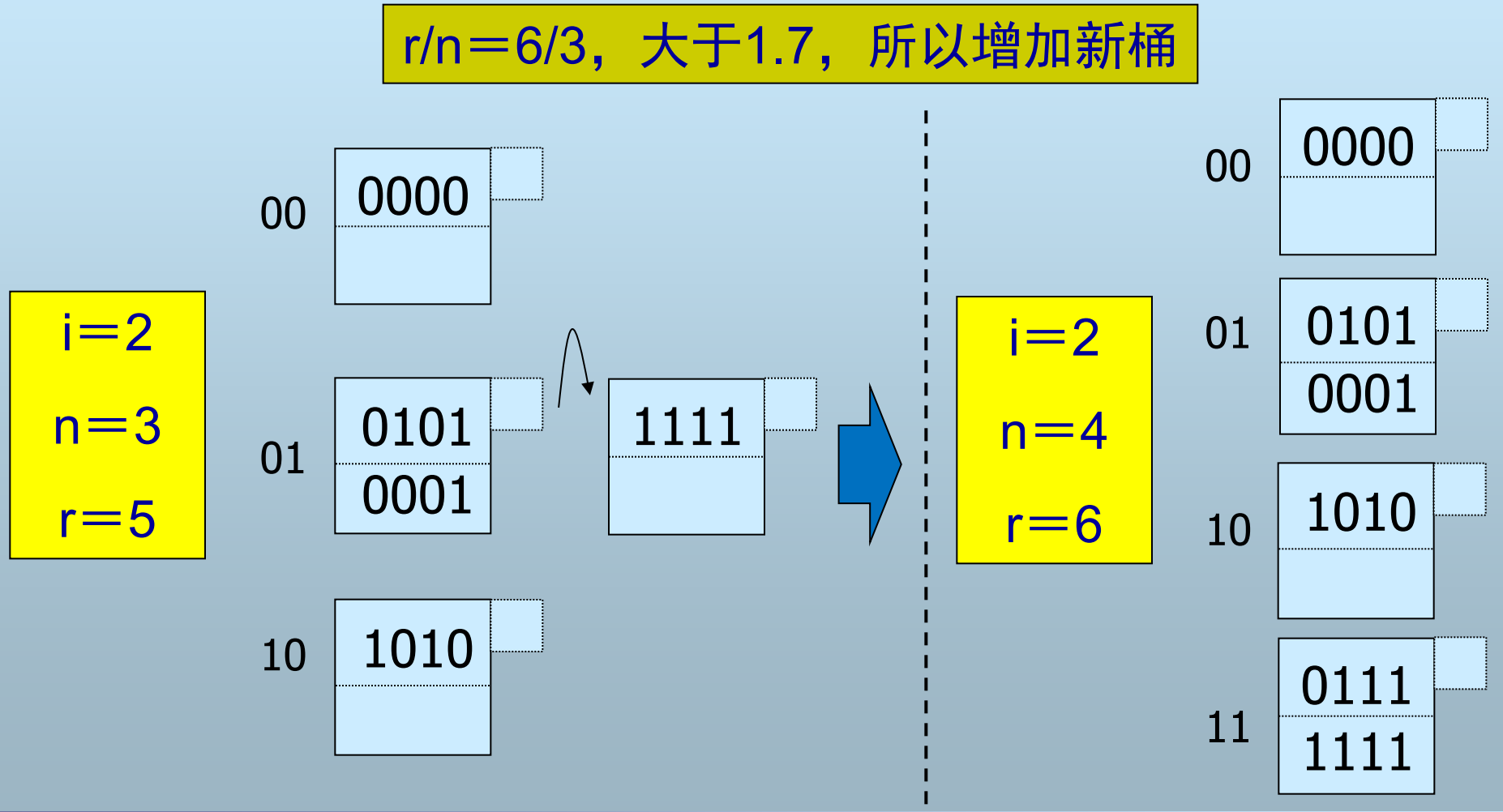
7、线性散列表

- 插入0001



7、线性散列表

- 插入0111



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

本章小结

- 顺序文件上的索引
- 辅助索引
- 一维索引
  - B+树
  - 散列表 (Hash Tables)