


# 第8章 数据库索引



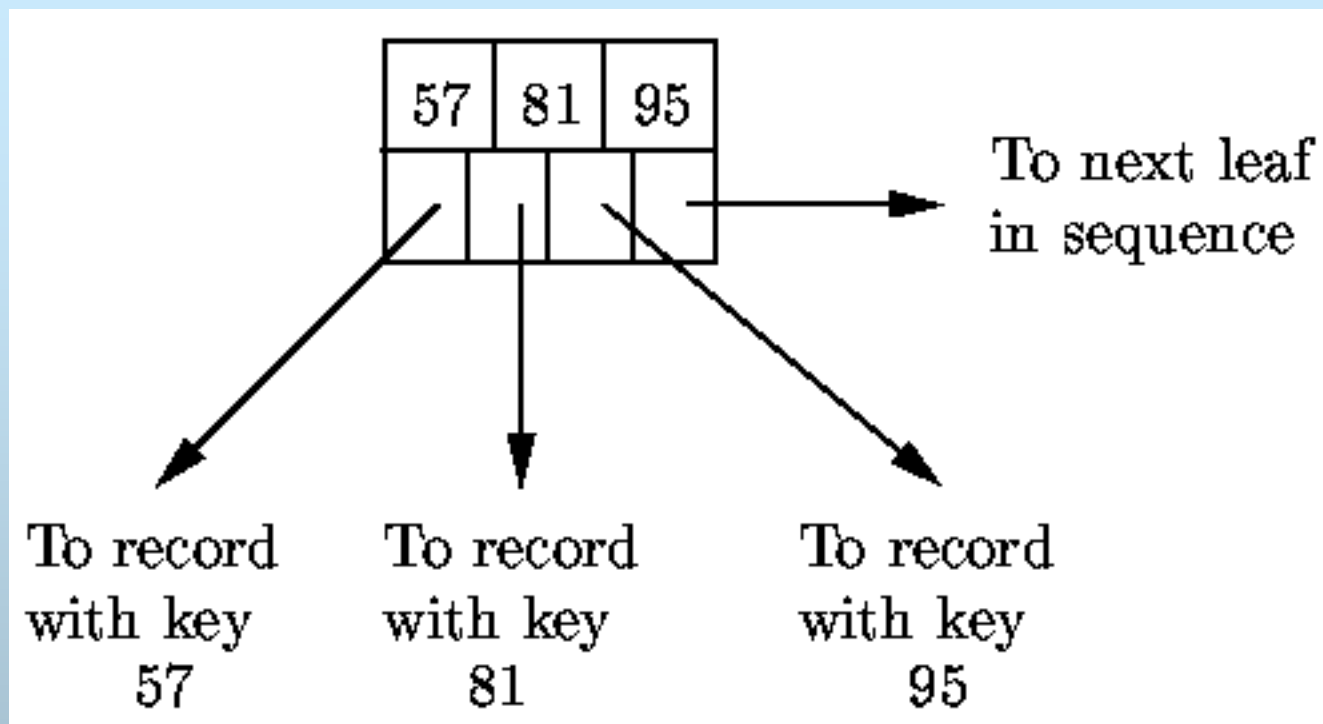
# 主要内容

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

## 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

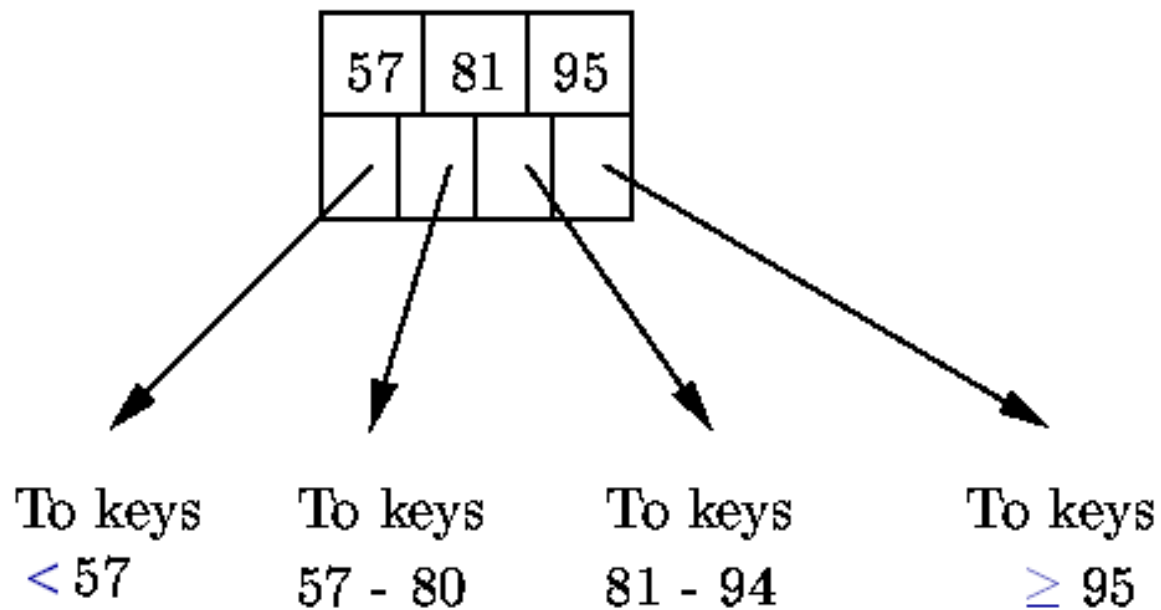
# 1、叶结点



- 1个指向相邻叶结点的指针
- $n$ 对键—指针对

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

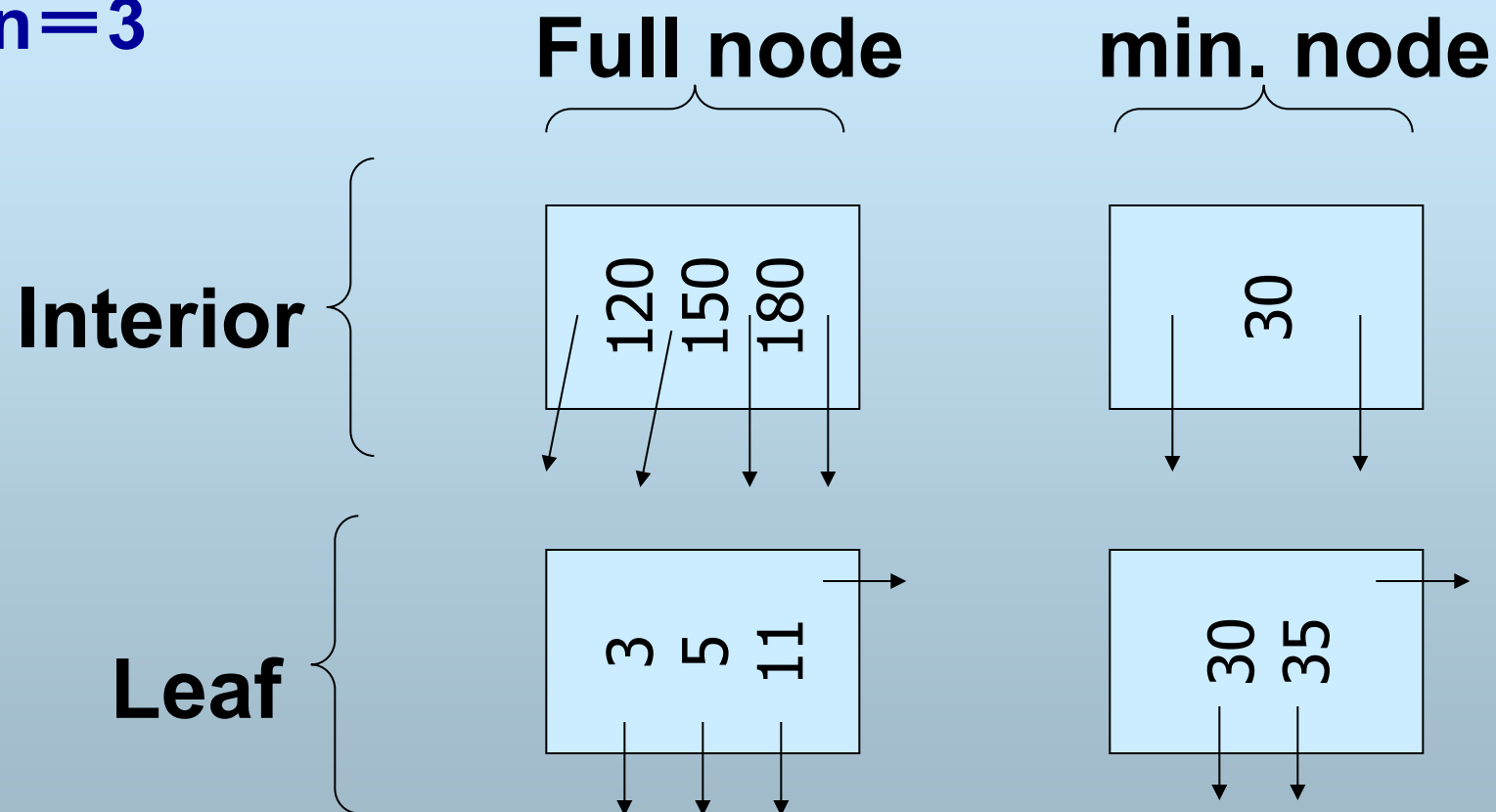
## 2、中间结点



- $n$ 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

$n=3$



# 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

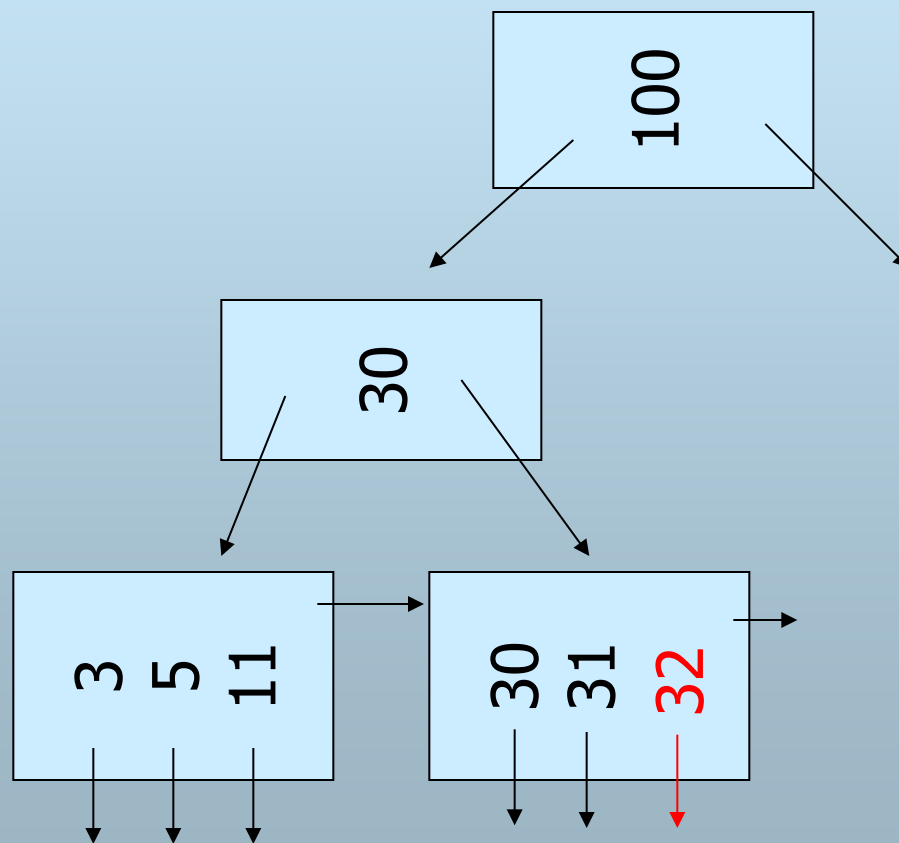
- 查找插入叶结点
- 若叶结点中有空闲位置（键），则插入
- 若没有空间，则分裂叶结点
  - 叶结点的分裂可视作是父结点中插入一个子结点
  - 递归向上分裂
  - 分裂过程中需要对父结点中的键加以调整
  - **例外**：若根结点分裂，则需要创建一个新的根结点



# B+树插入例子

(a) Insert key = 32

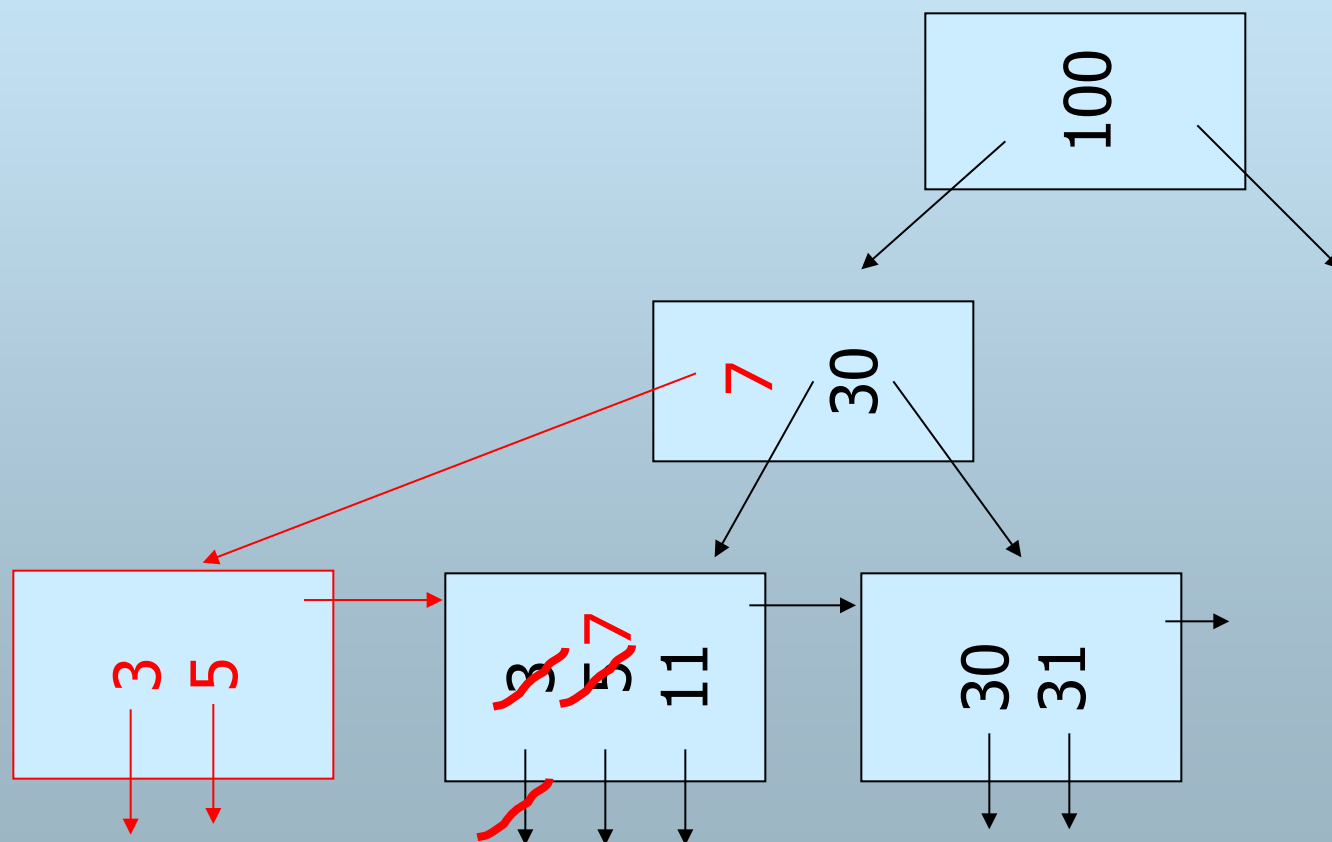
n=3



# B+树插入例子

(b) Insert key = 7

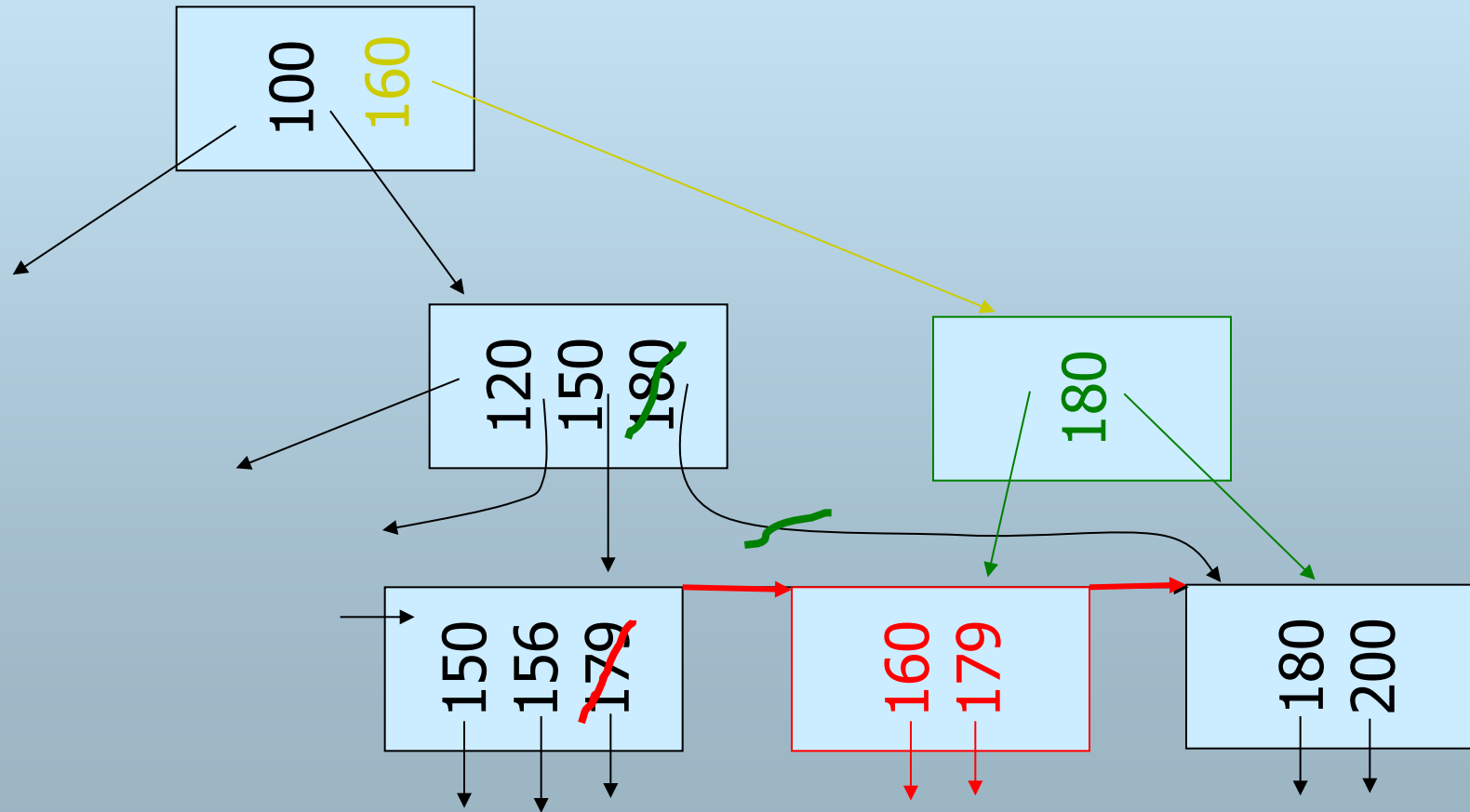
n=3



# B+树插入例子

(c) Insert key = 160

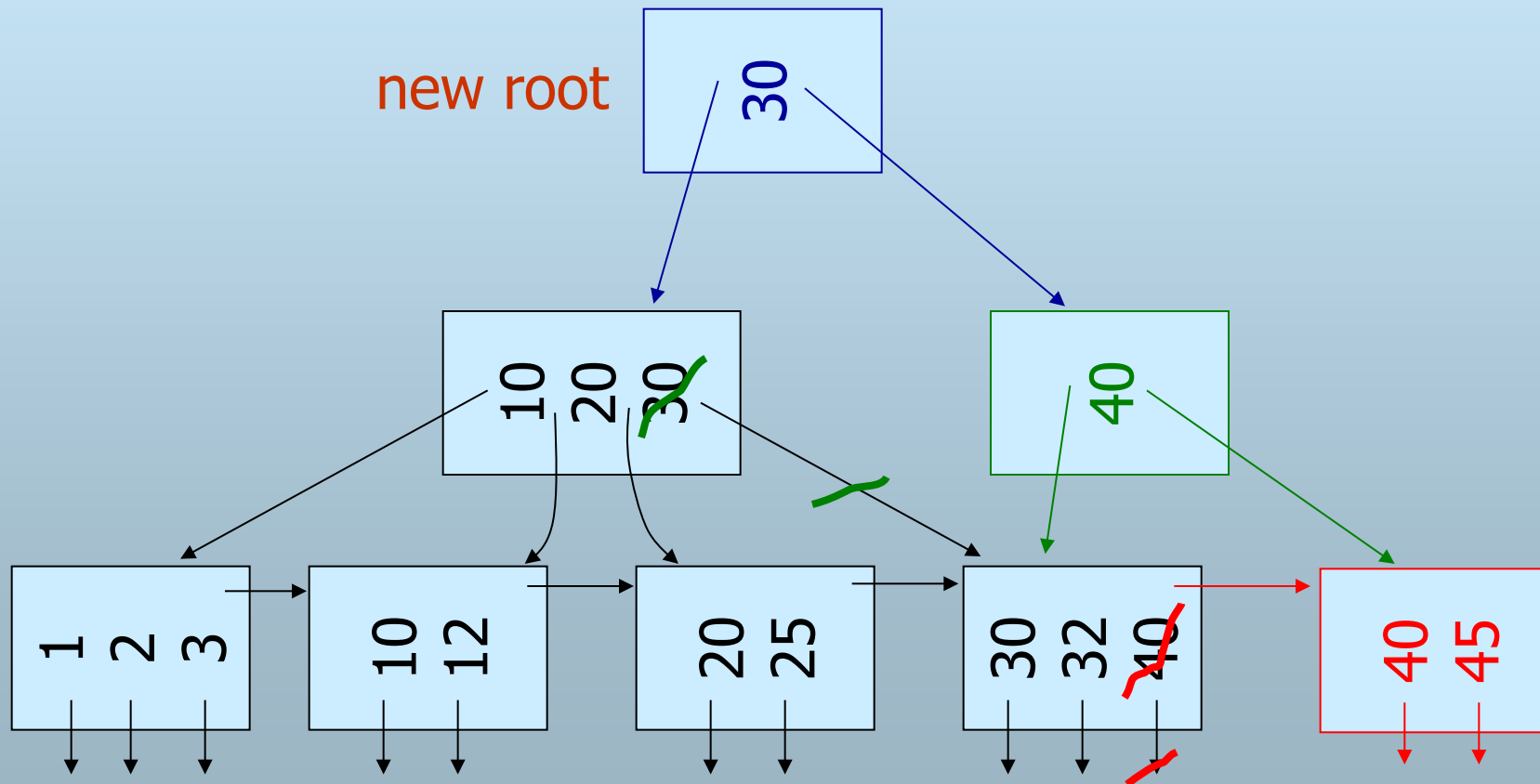
n=3



# B+树插入例子

(d) New root, insert 45

n=3



## 5、B+树删除

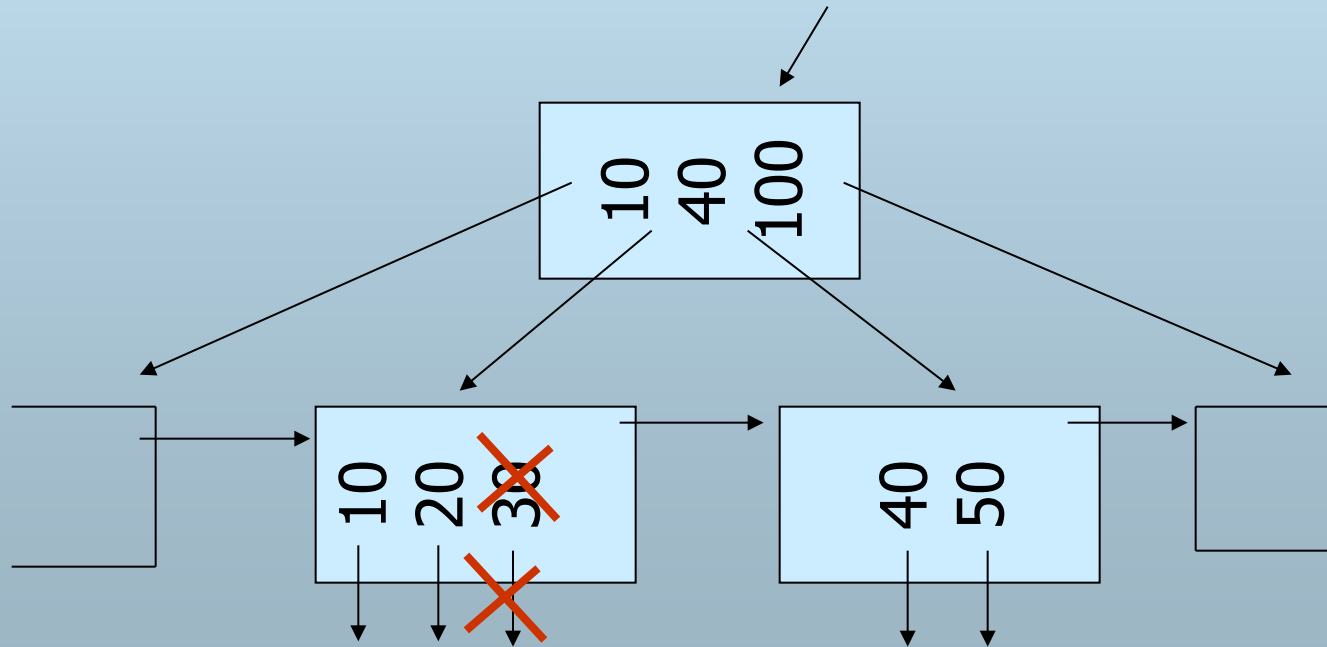
- 查找要删除的键值，并删除之
- 若结点的键值填充低于规定值，则调整
  - 若相邻的叶结点中键填充高于规定值，则将其中一个键值移到该结点中
  - 否则，合并该结点与相邻结点
    - ◆ 合并可视为在父结点中删除一个子结点
  - 递归向上删除
- 若删除的是叶结点中的最小键值，则需对父结点的键值加以调整

# B+树删除例子

## (a) Simple delete

- Delete 30

n=4

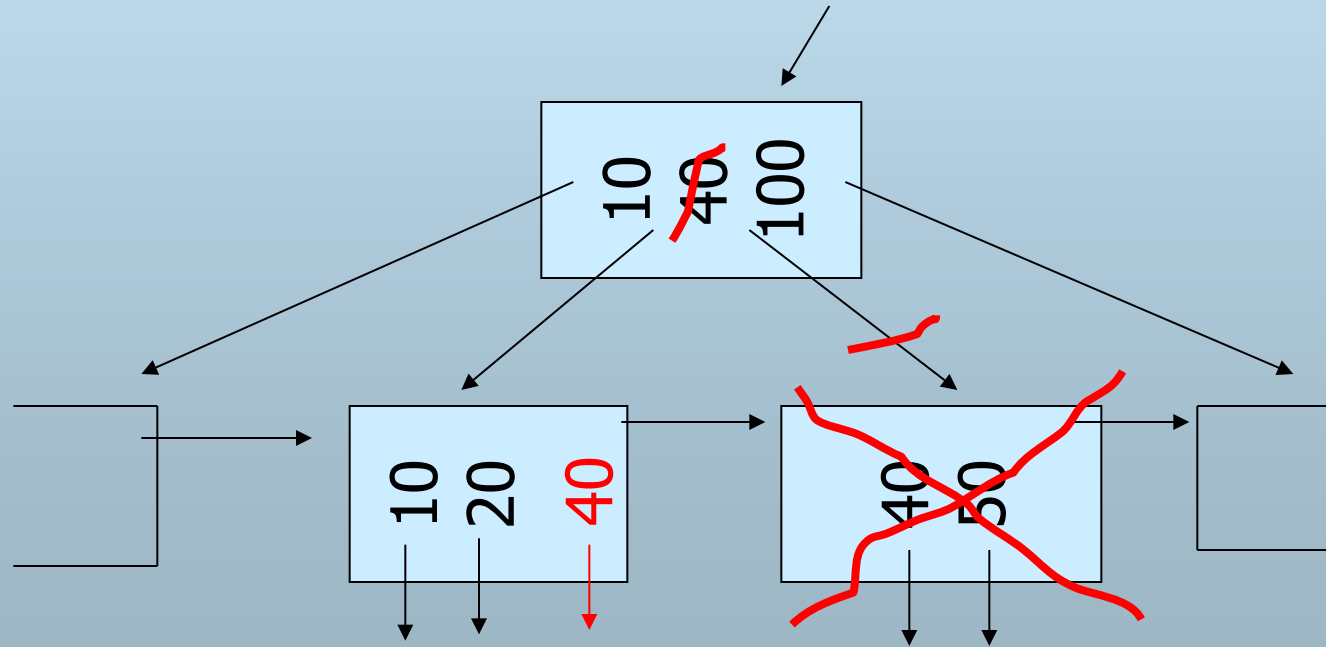


# B+树删除例子

## (b) Coalesce with sibling

- Delete 50

n=4

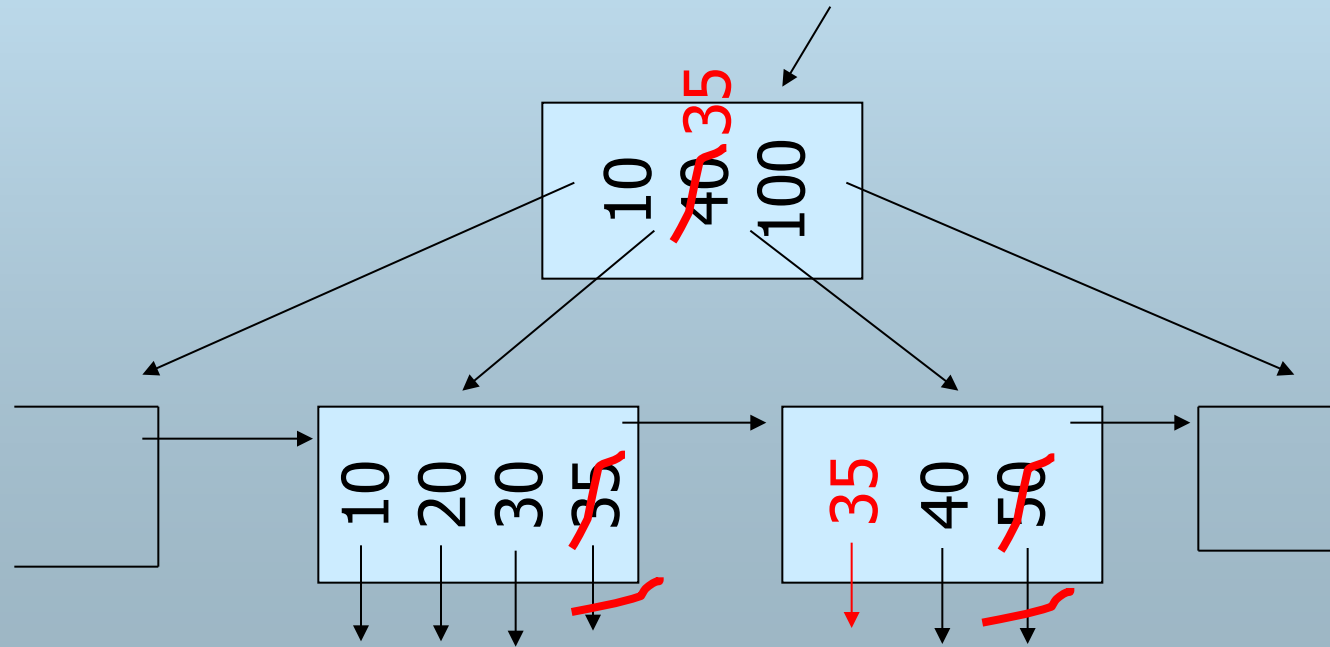


# B+树删除例子

## (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( $2^{33}$ )

# 7、B树 vs. B+树

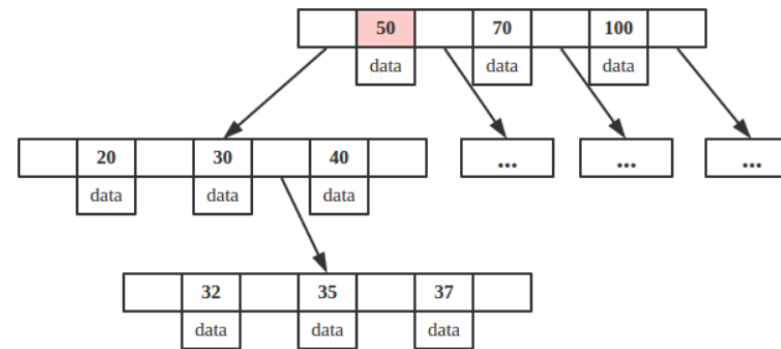
## ■ B-tree

- 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

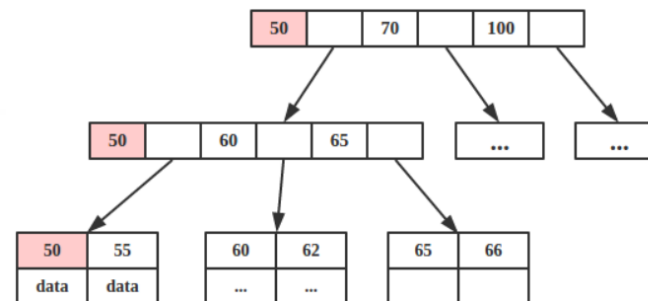
- 节点本身不存储数据

B-树



B-树

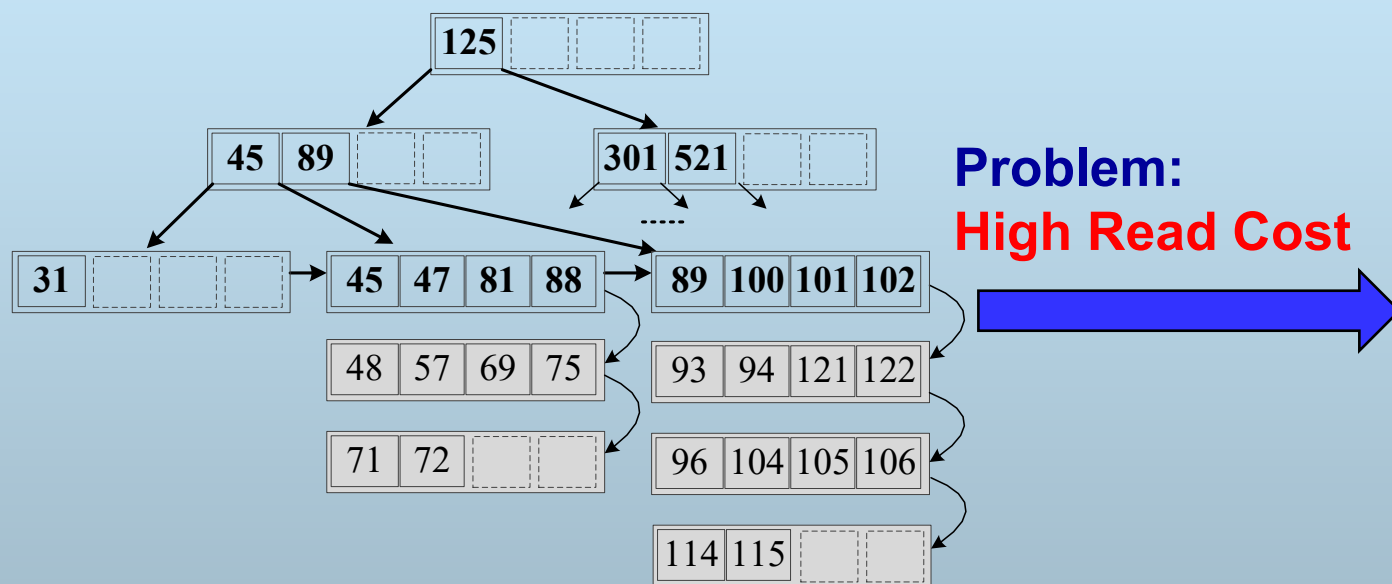
B+树



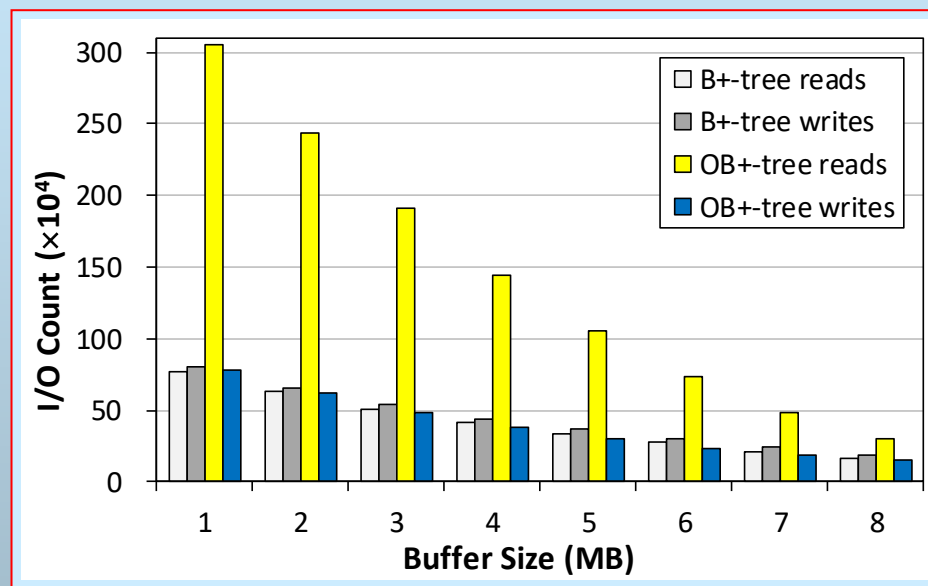
B+树

# 8、B+树的改进

## ■ 如何提高B+树的写性能？



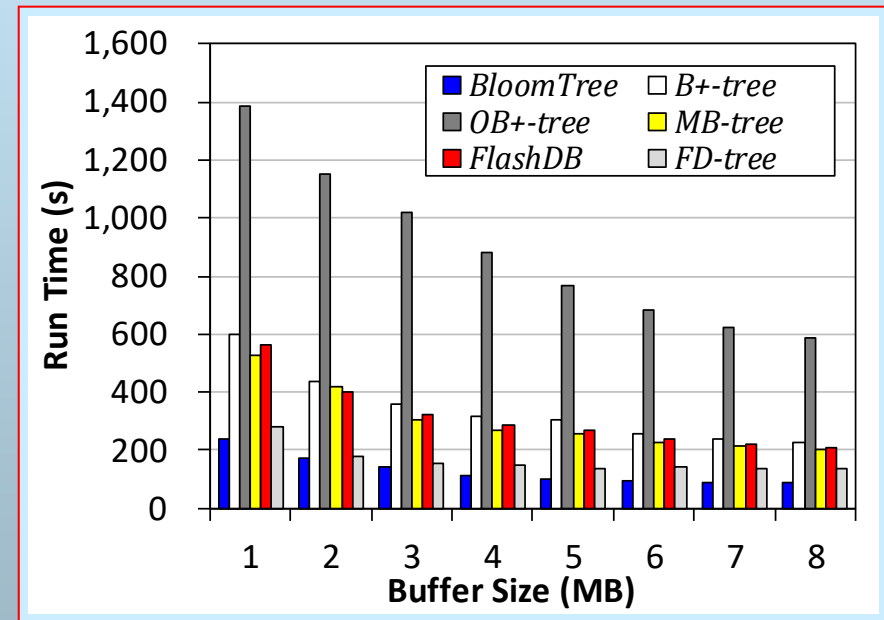
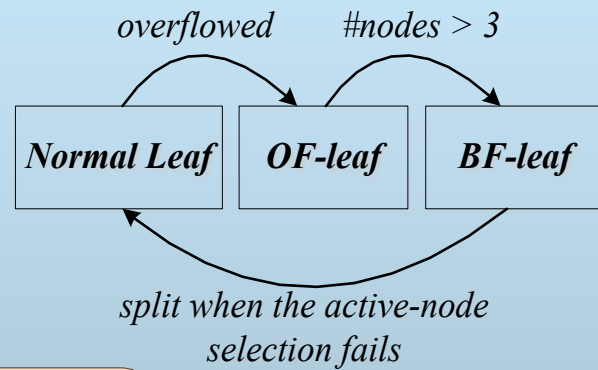
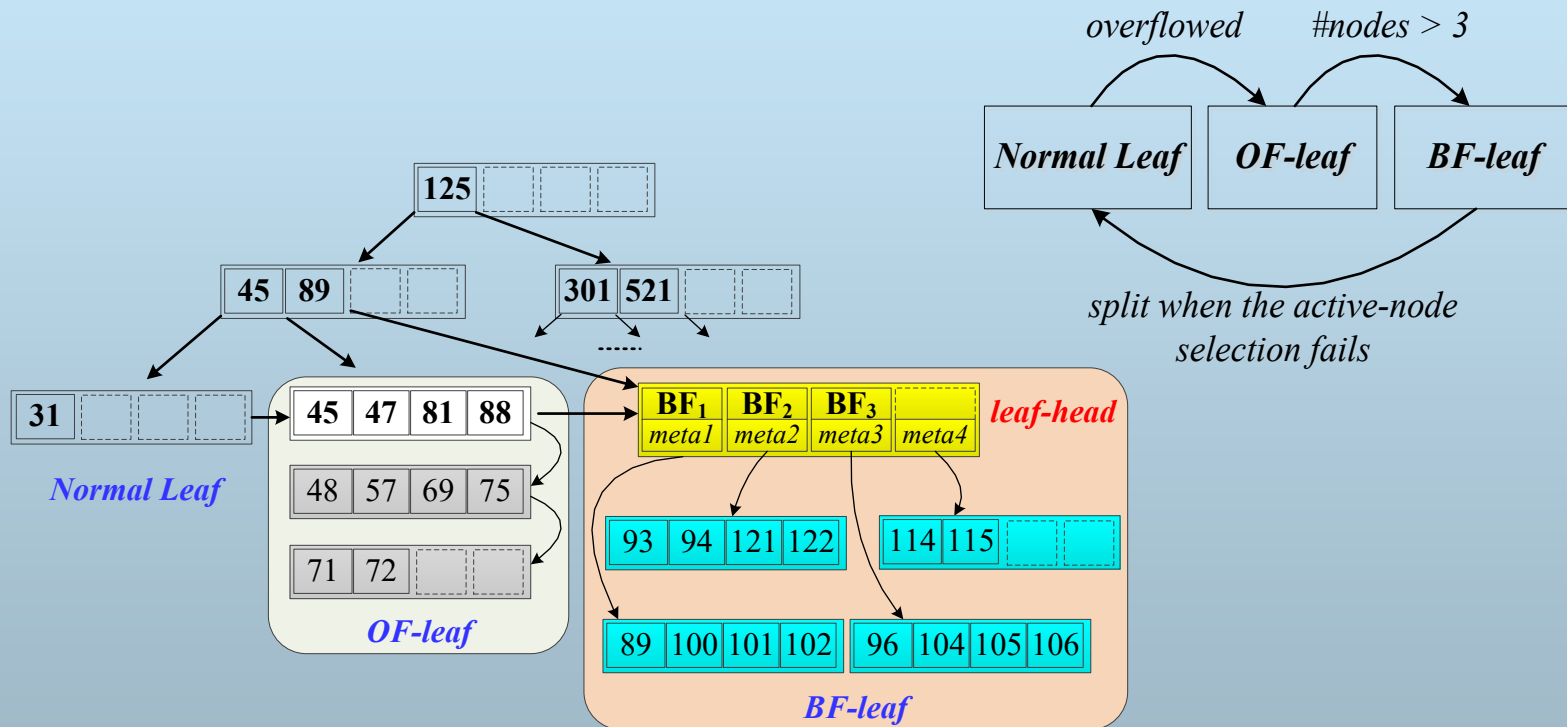
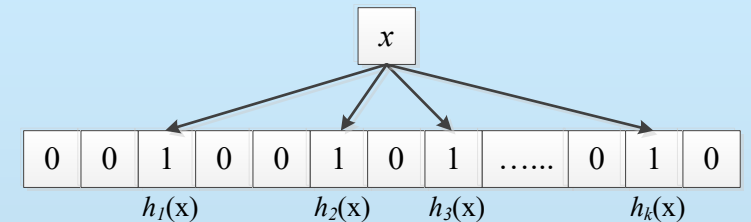
Overflow B+-tree



# 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)

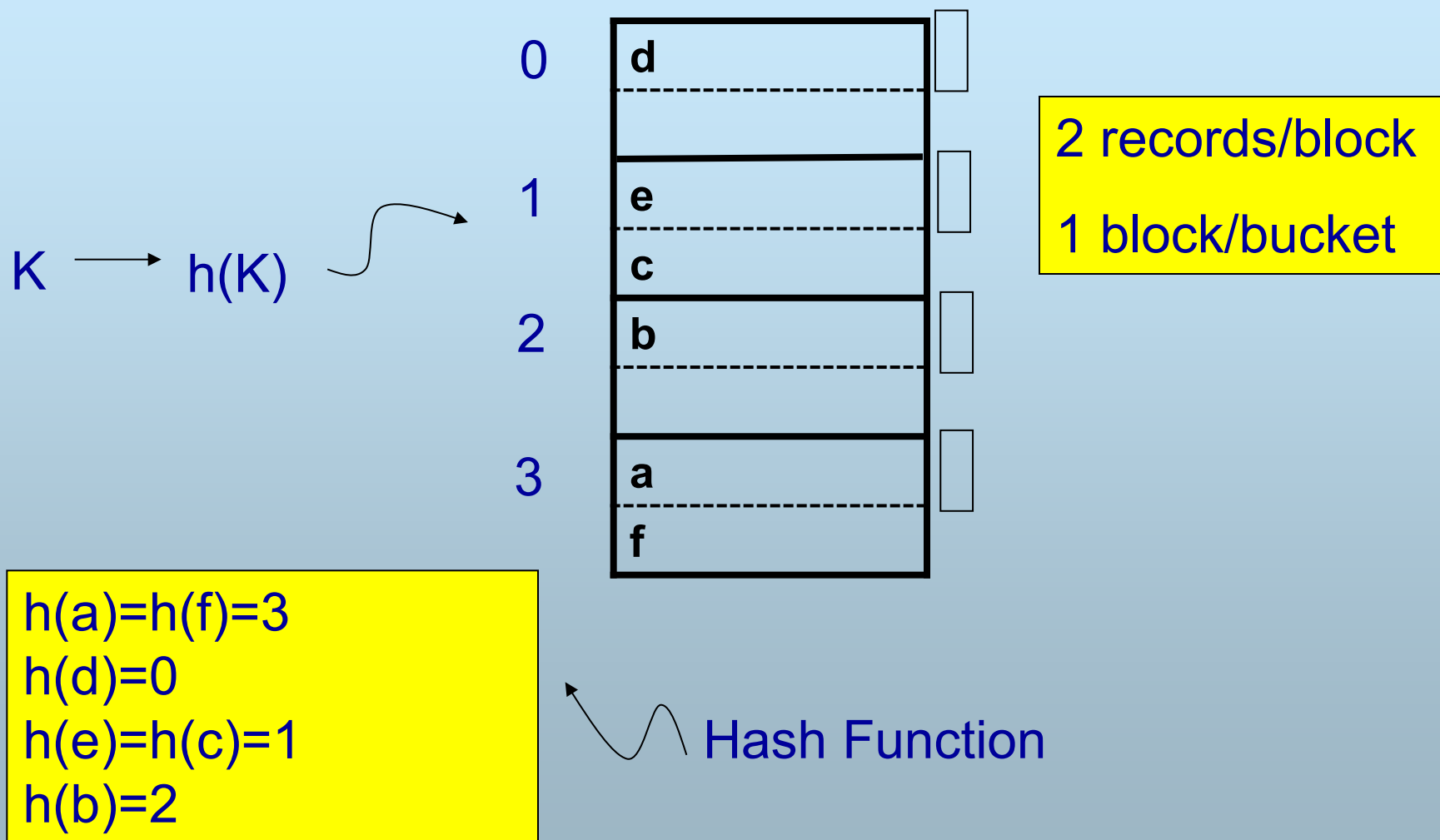
## ■ 散列函数(Hash Functions)

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

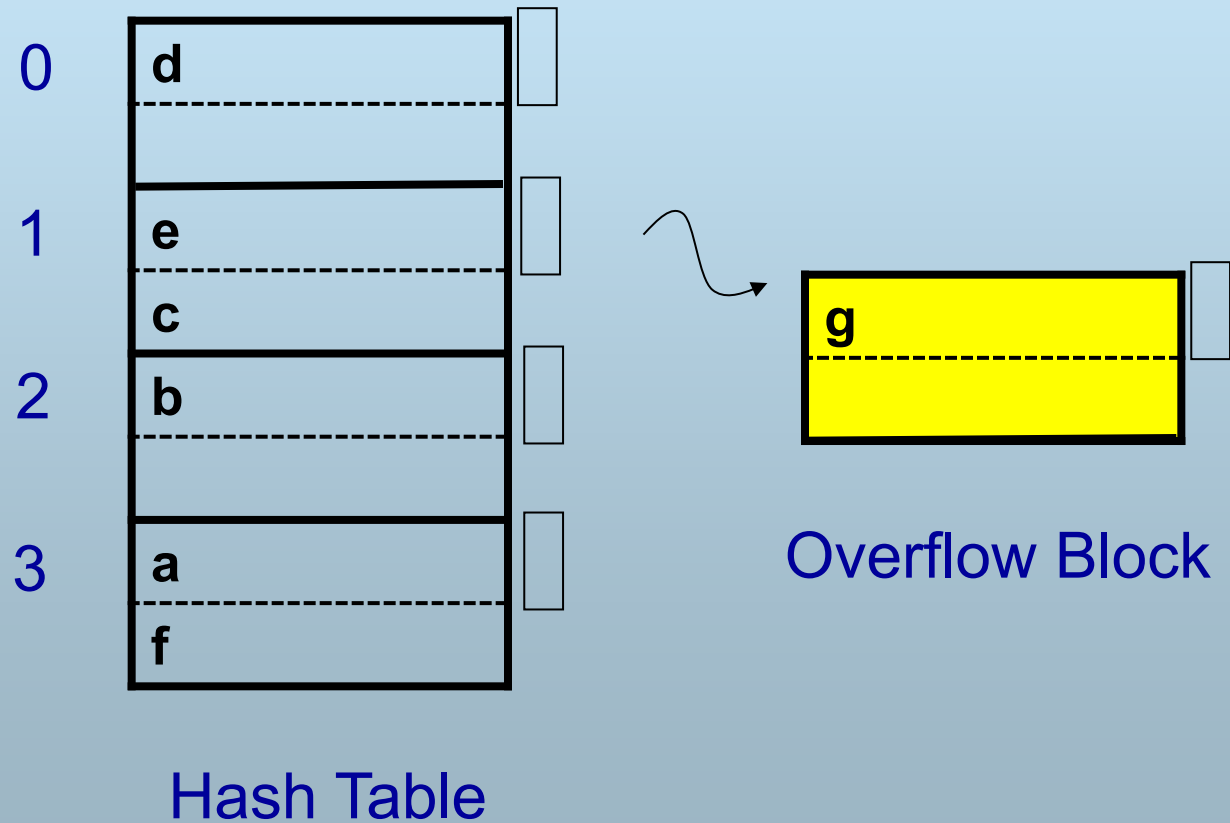


### 3、散列表插入

- 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

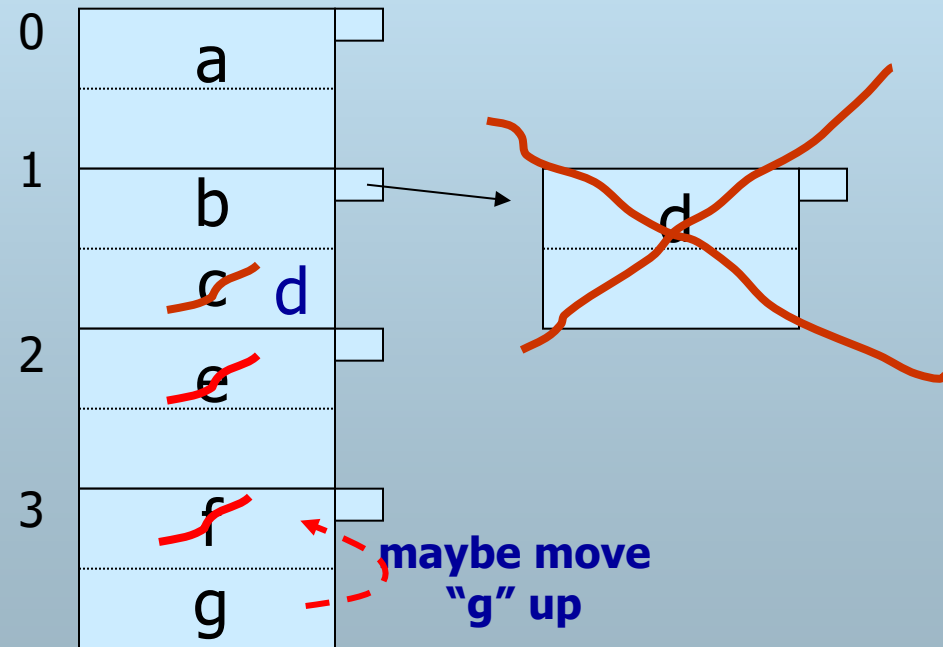
## EXAMPLE: deletion

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%：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）

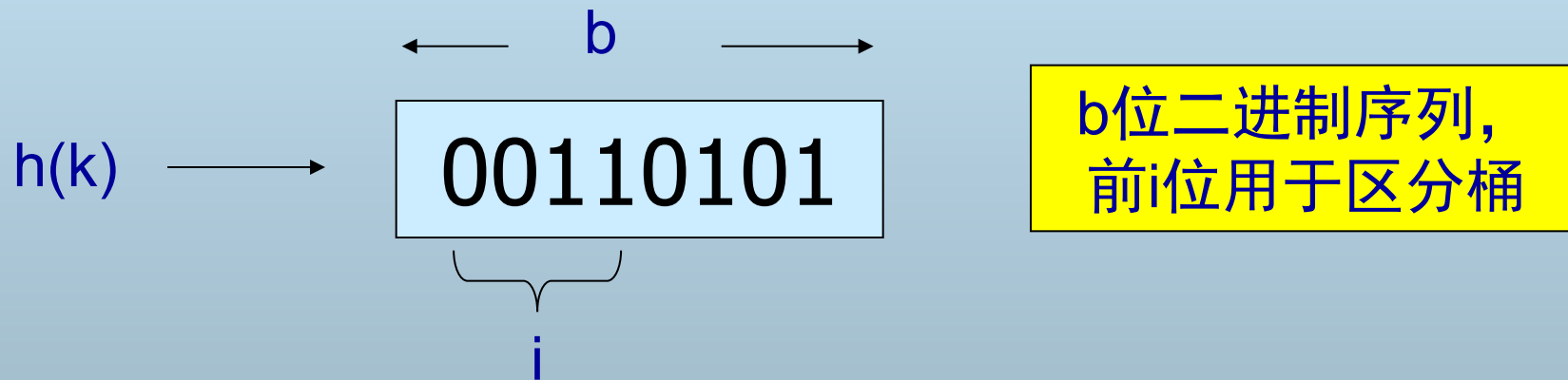
##### ● 成倍增加桶数目

#### ◆ 线性散列表（Linear Hash Tables）

##### ● 线性增加

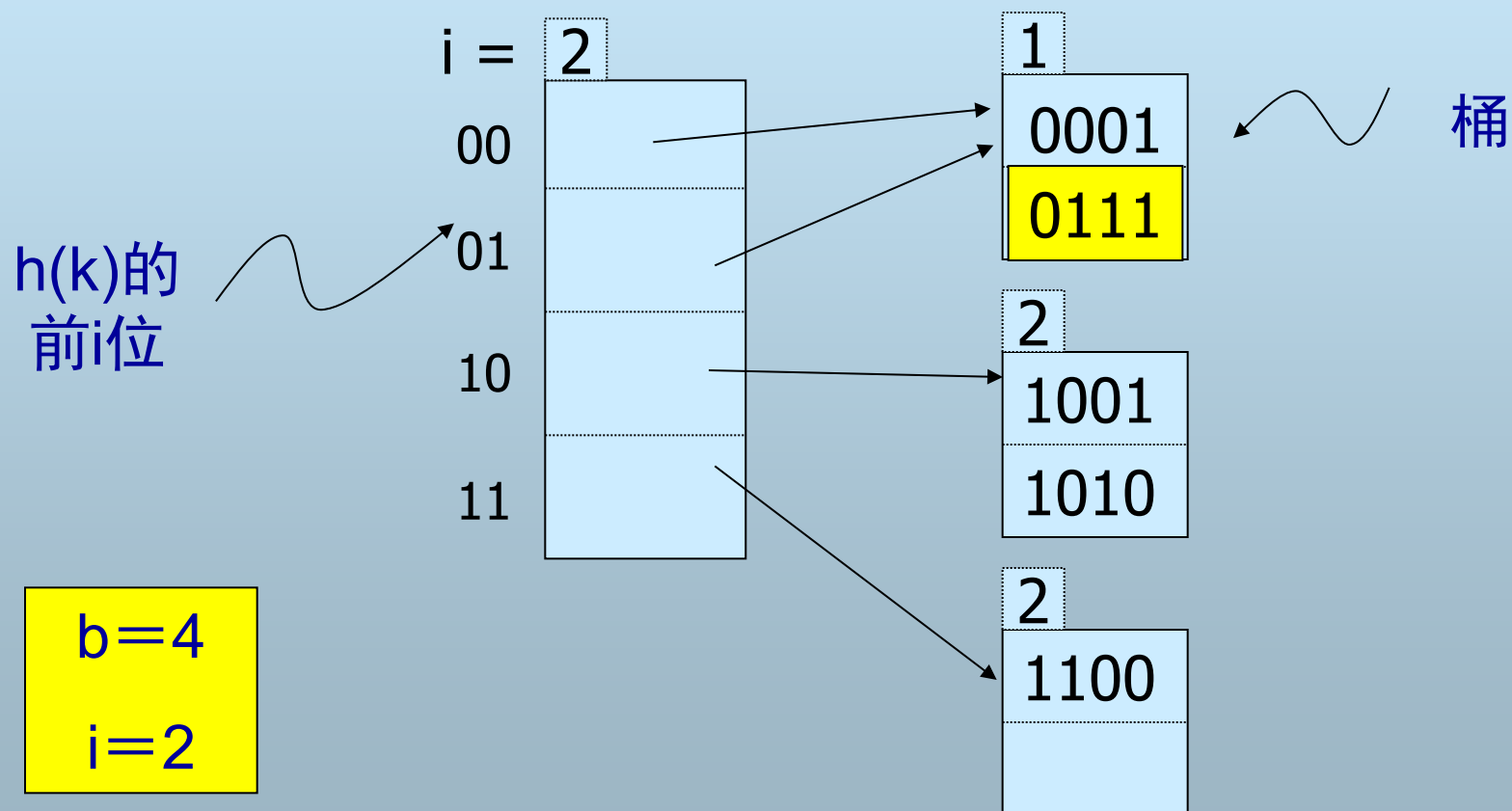
## 6、可扩展散列表

- 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



# 6、可扩展散列表

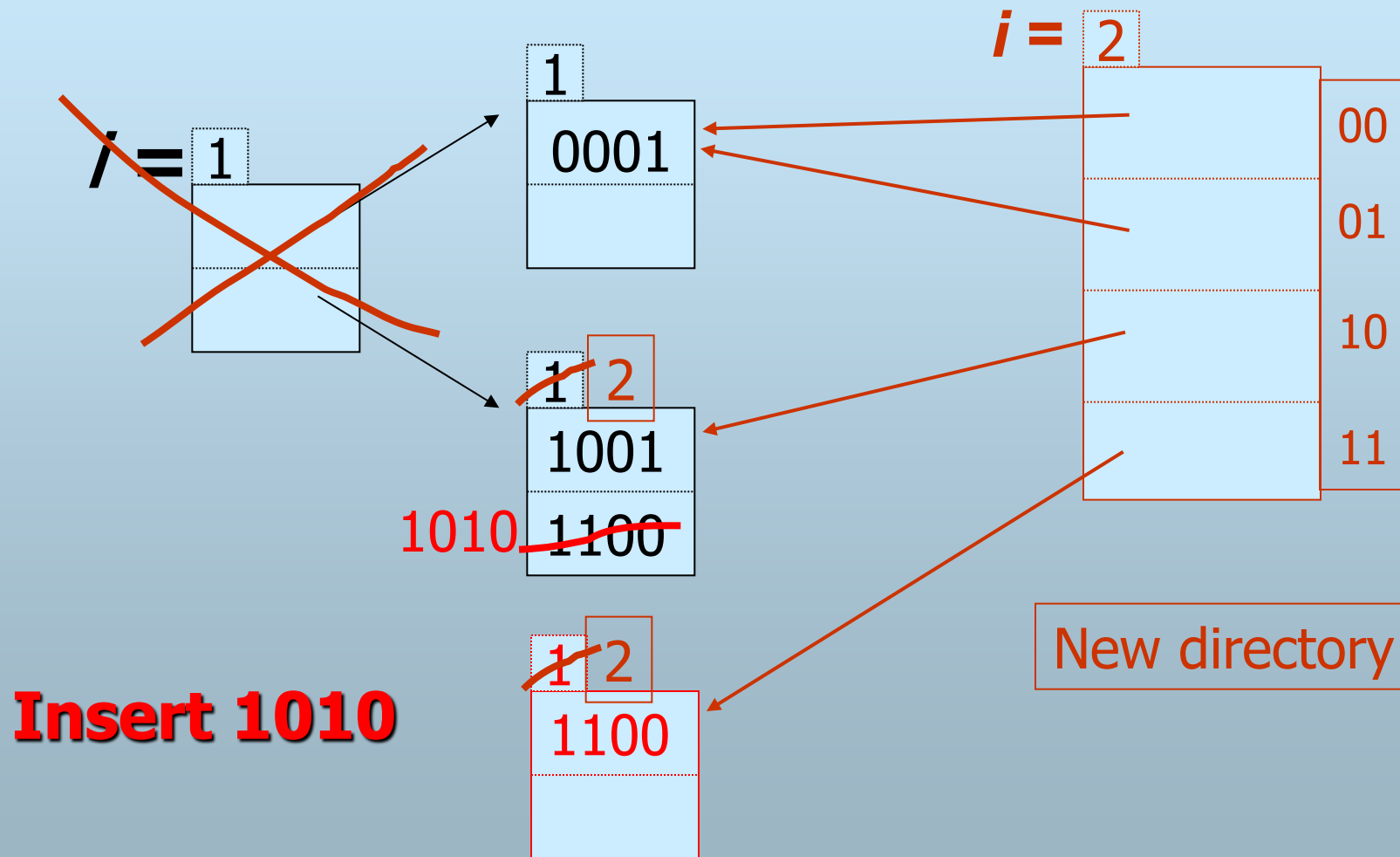
## ■ 前*i*位构成一个桶数组





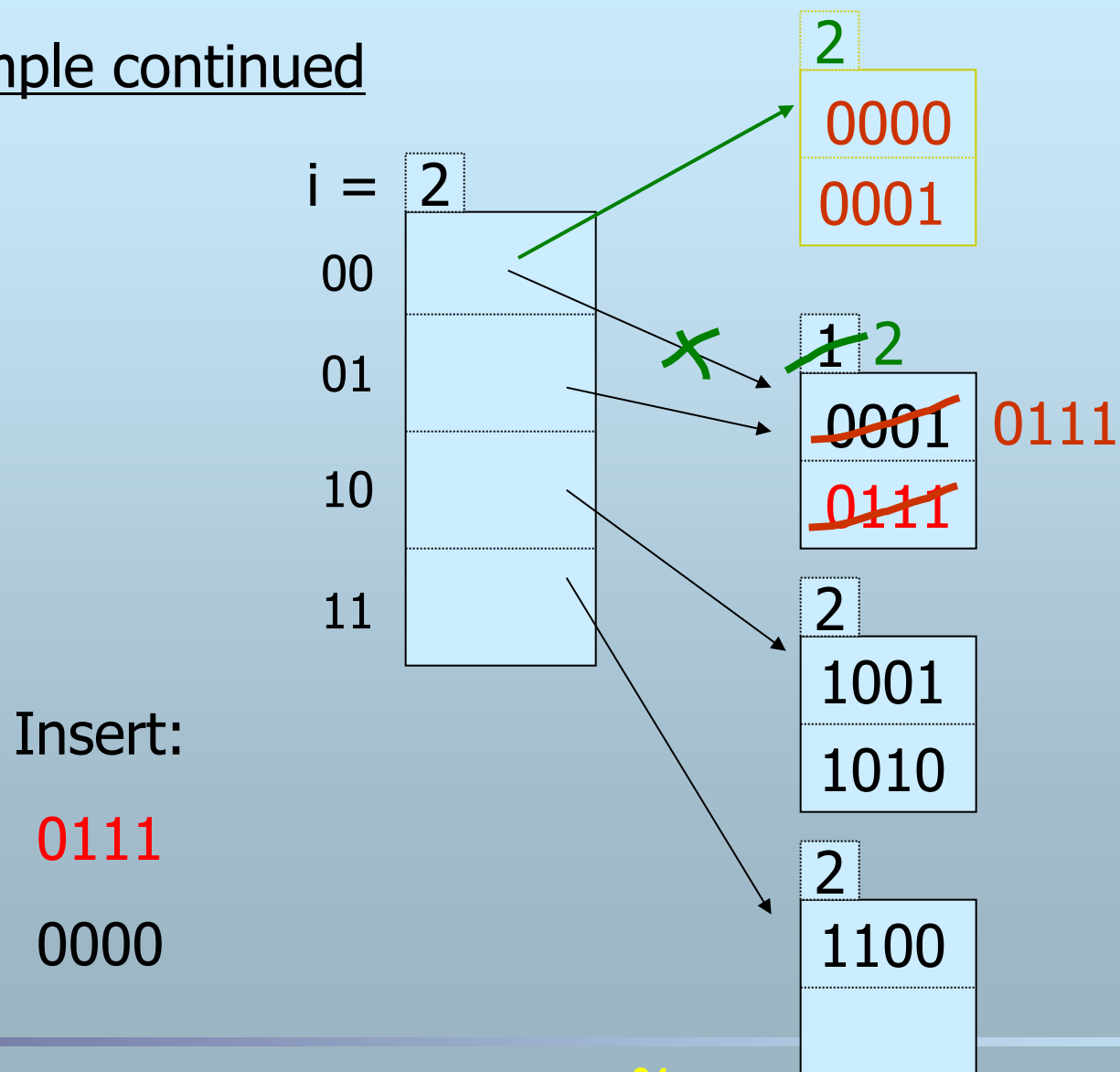
## 6、可扩展散列表

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



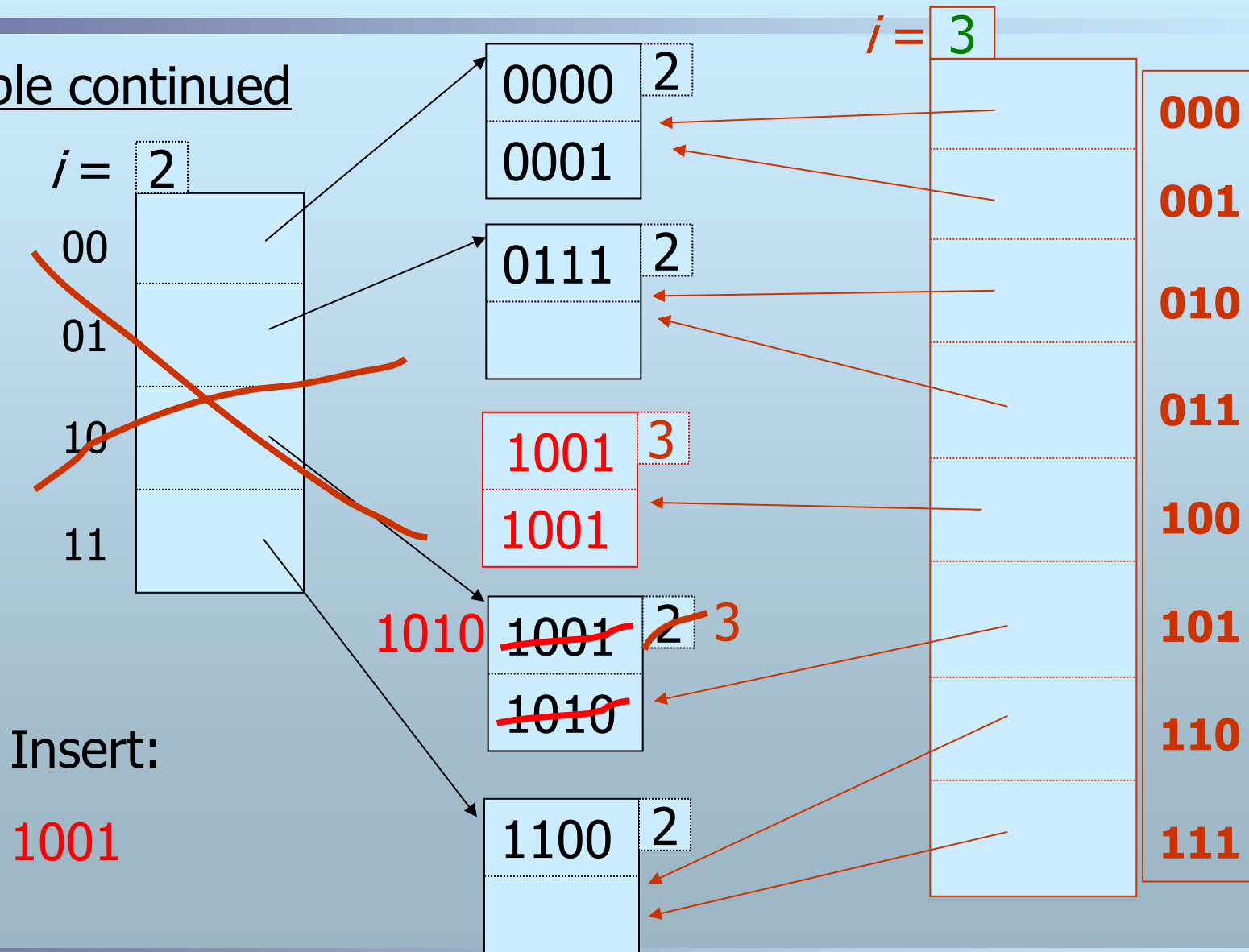
## 6、可扩展散列表

## Example continued



# 6、可扩展散列表

Example continued



## 6、可扩展散列表

### ■ 优点

- 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。

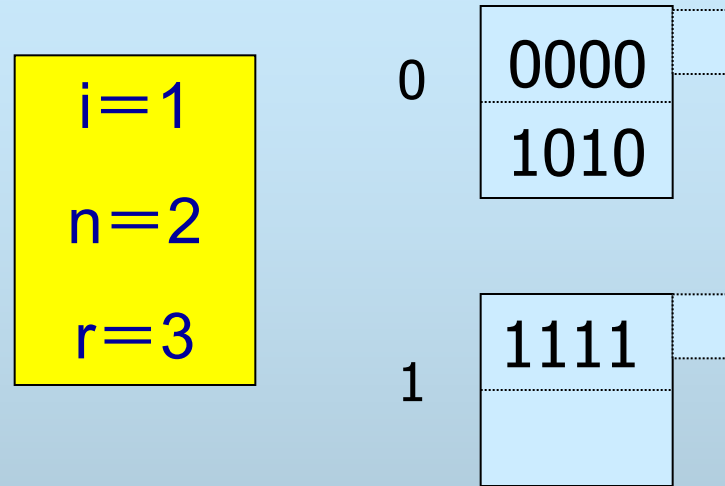
### ■ 缺点

- 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 $n \leq m < 2^i$ ，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

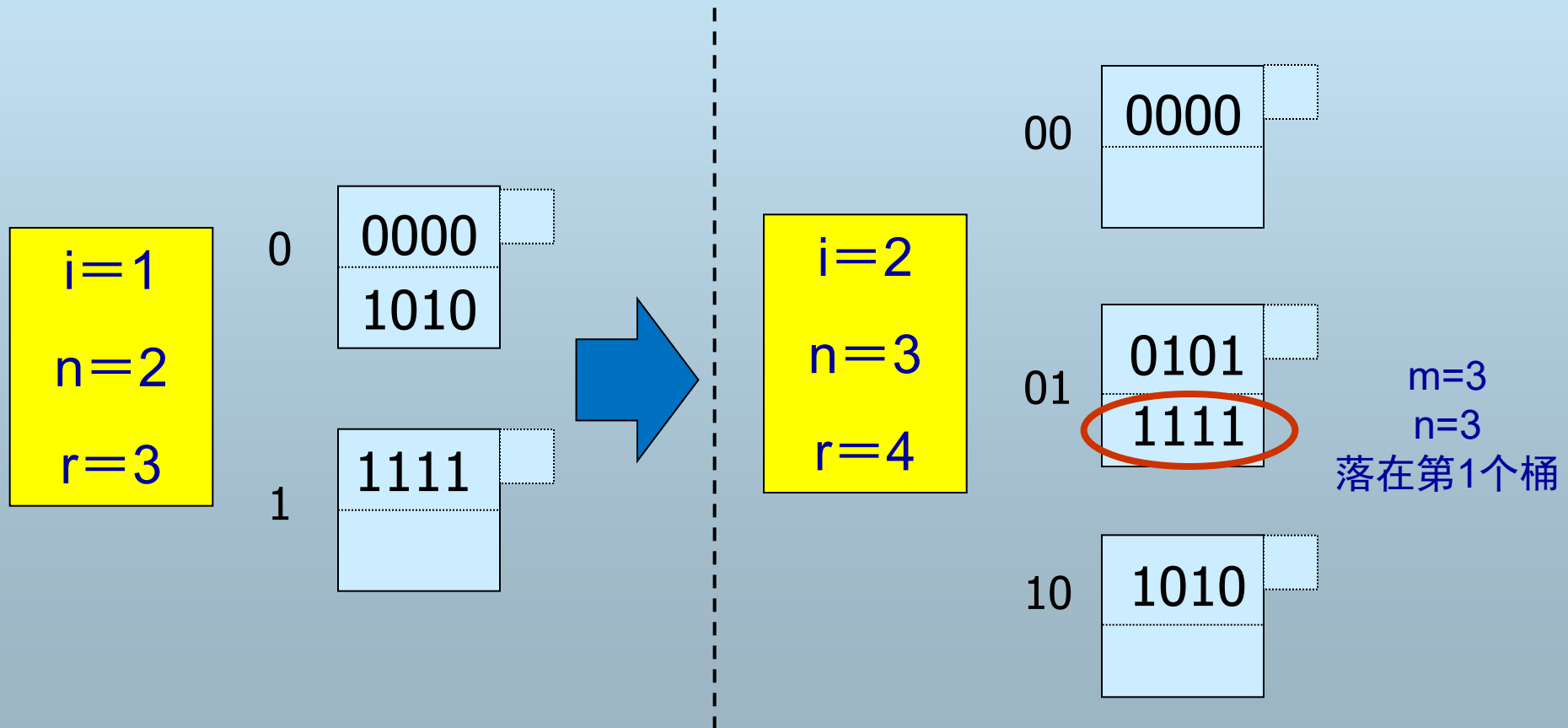


$i$ : 当前被使用的散列函数值的位数，从低位开始  
 $n$ : 当前的桶数  
 $r$ : 当前散列表中的记录总数  $r/n < 1.7$

# 7、线性散列表

## ■ 插入0101

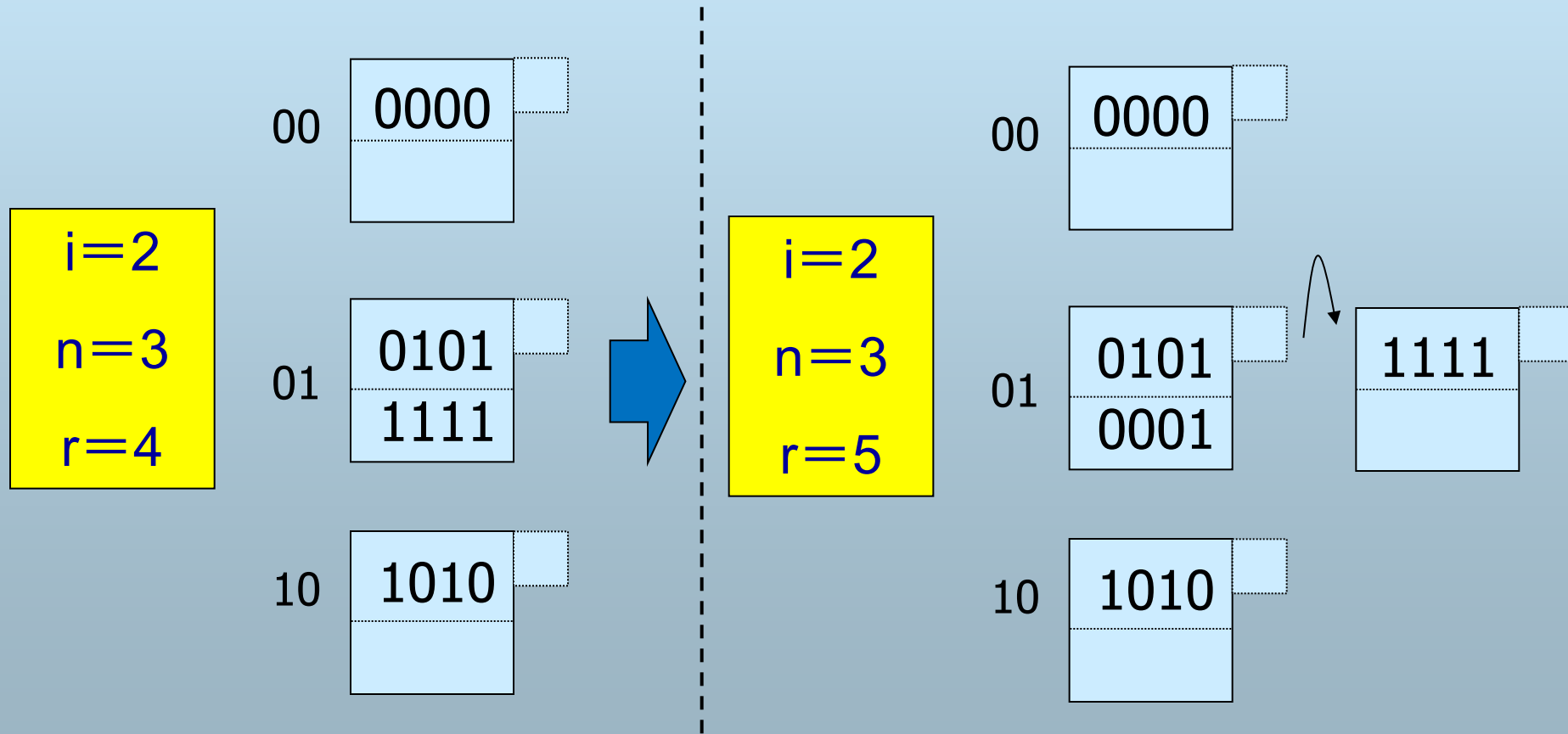
$r/n=4/2$ ，超过了1.7，所以增加新桶



# 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

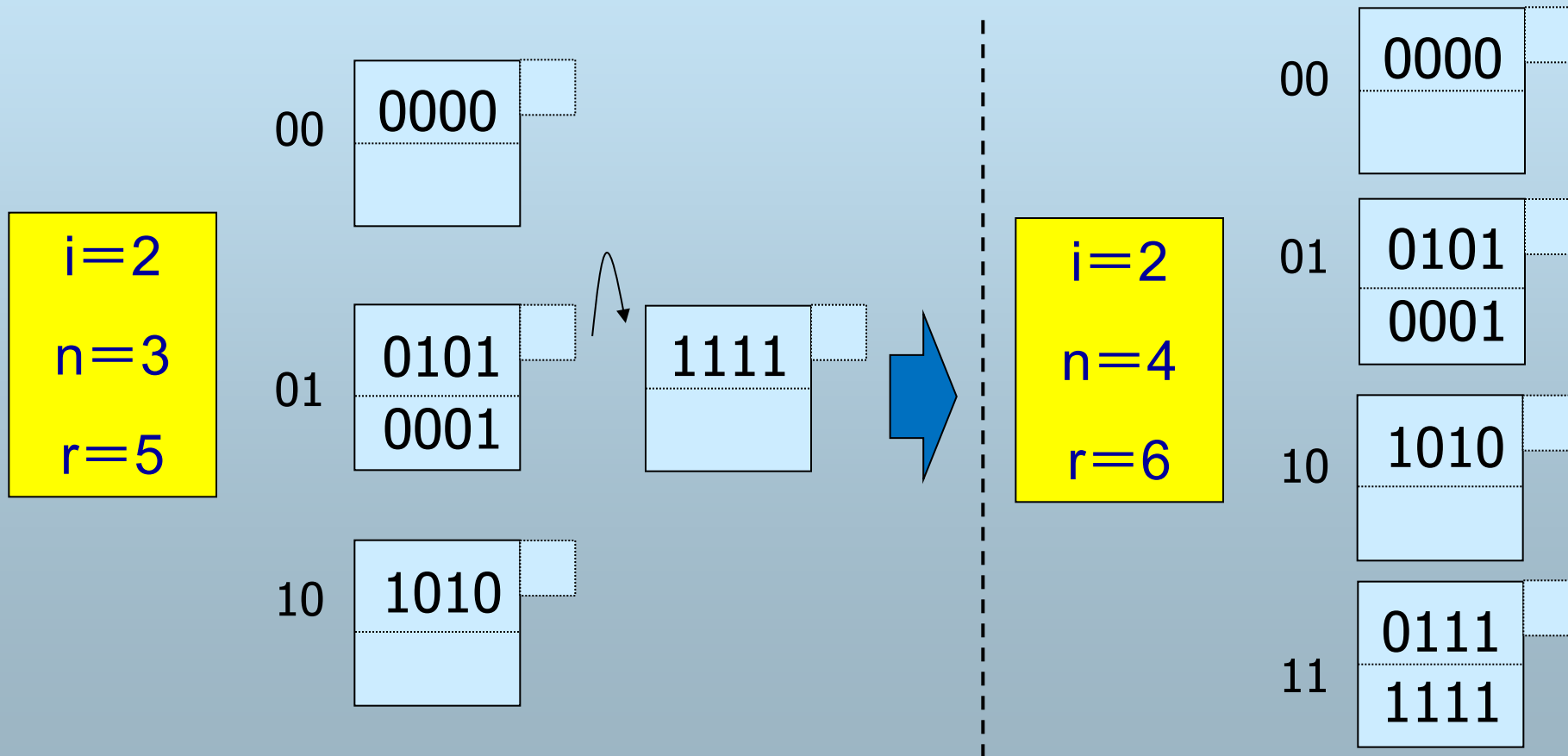




# 7、线性散列表

## ■ 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**