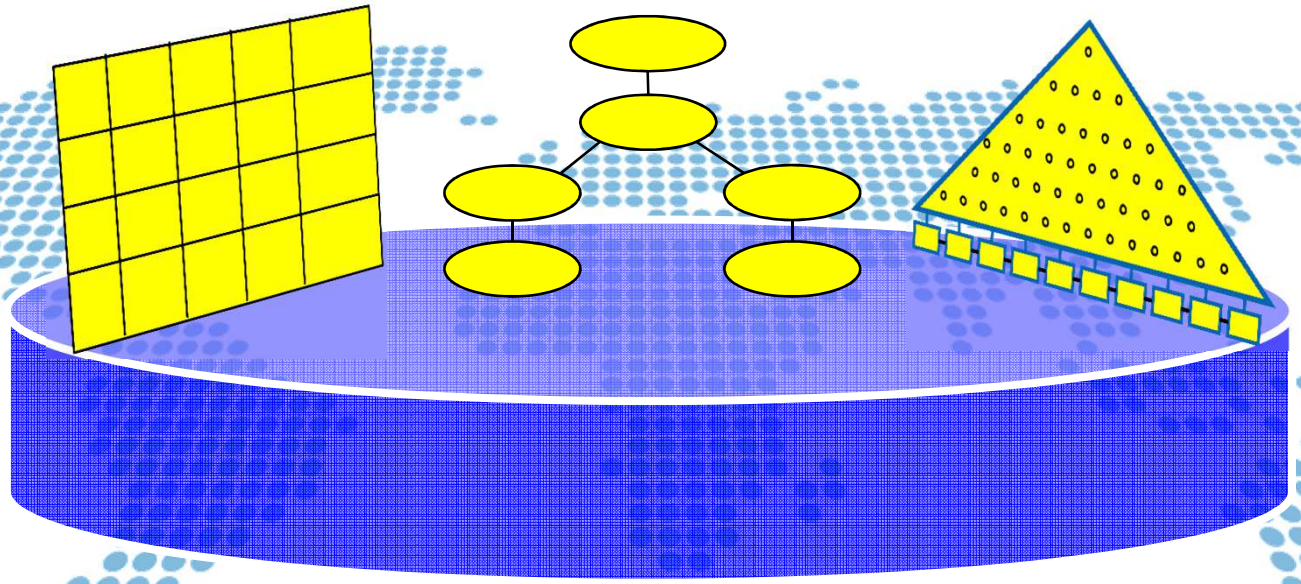
数据库系统

数据存储与访问路径(2)

陈世敏

（中科院计算所）



上节内容

- 数据库系统内部架构概述
- 数据存储与访问路径概述
  - 存储层次
  - 存储介质：磁盘、固态硬盘等
  - 磁盘阵列
  - 操作系统支持
  - 数据与索引
- 磁盘空间管理：工作原理
- 记录文件格式
  - 行式文件页结构
  - 行式记录结构
  - 列式文件结构
  - 顺序读和I/O模型
- 缓冲区管理：工作原理，替换算法

Outline

- 索引的概念
- 树结构索引
- 哈希索引
- 位图索引
- 倒排索引

数据的顺序访问

```
select Name, GPA
from Student
where Major = '计算机';
```

- 顺序读取Student表的每个page
- 对于每个page，顺序访问每个tuple
- 检查条件是否成立
- 对于成立的读取Name和GPA

☞如果有100个专业会怎么样？

数据的顺序访问

```
select Name, GPA
from Student
where Major = '计算机';
```

☞如果有100个专业会怎么样？

有些浪费

如果每个专业的学生人数大致相同

那么会扔掉99%的记录！

Selective Data Access (有选择性的访问)

```
select Name, GPA
from Student
where Major = '计算机';
```

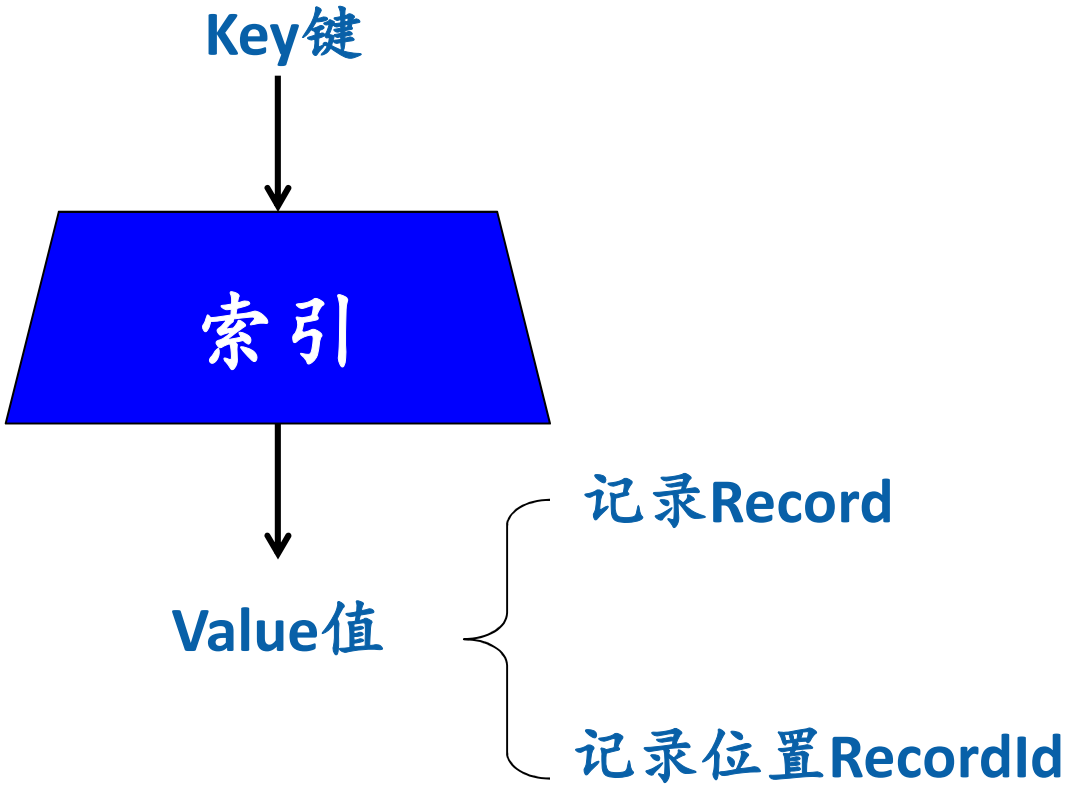
☞如果有100个专业会怎么样？

希望可以直接找到相关的记录

而不需要扫描整个的Table

👍使用索引

索引（Index）概念



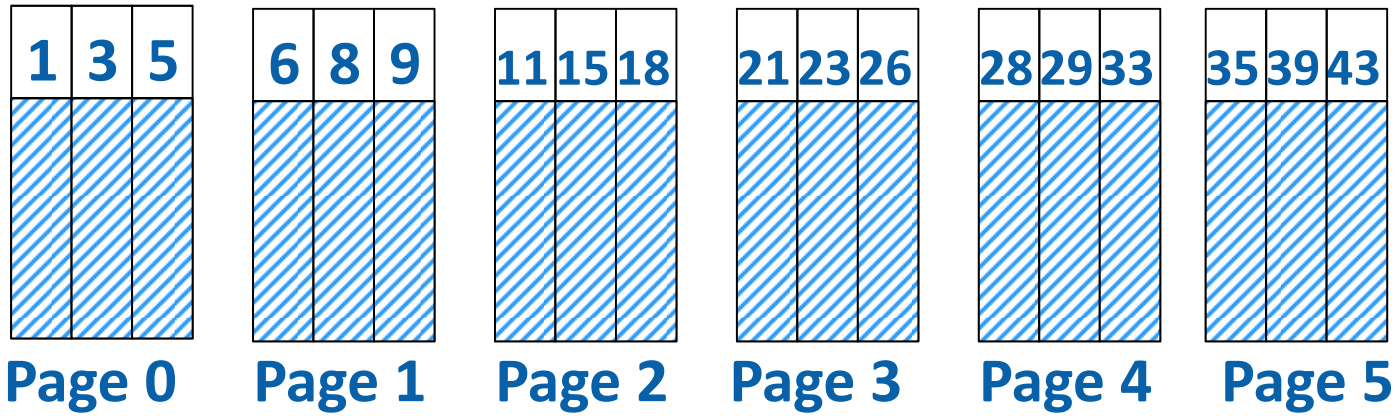
- 给定一个键，找到对应的值
- 在数据库里，值通常对应于记录或记录的位置

Outline

- 索引的概念
- 树结构索引
  - 从顺序文件到静态索引
  - B+树
  - 主索引（聚簇索引）和辅助索引（二级索引）
- 哈希索引
- 位图索引
- 倒排索引
- 多维索引

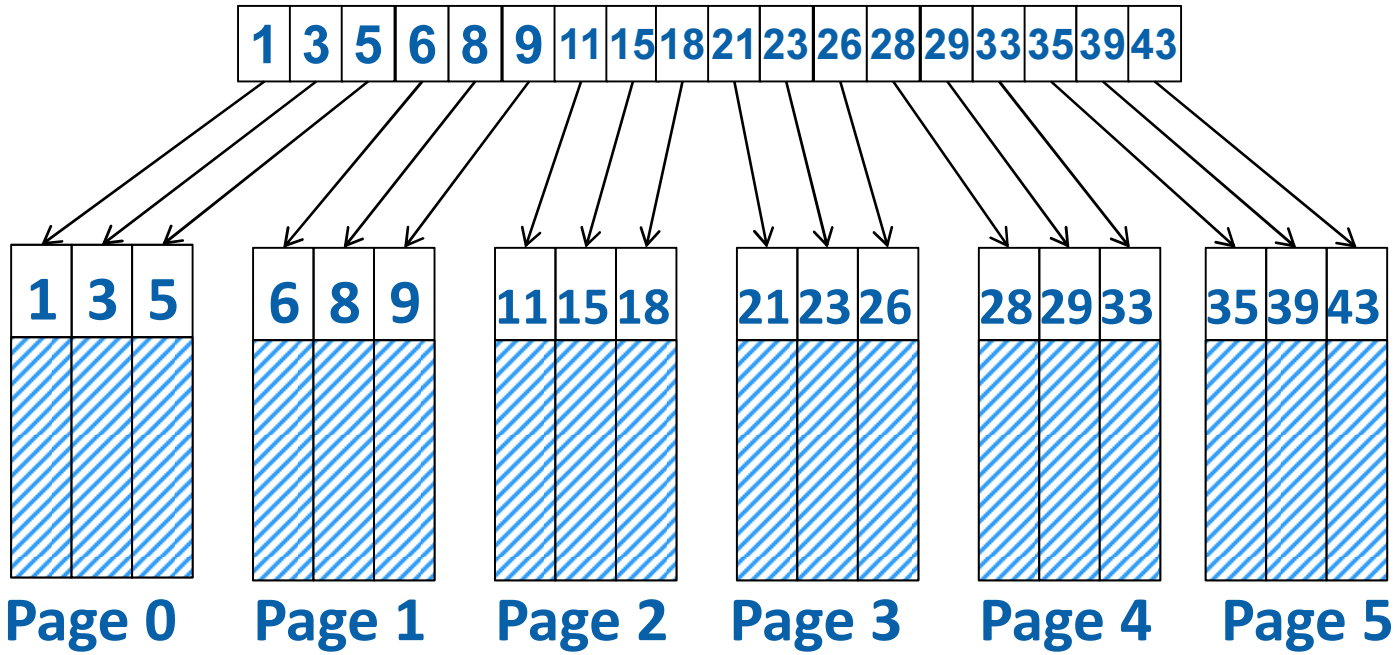
排序文件

- 按照某个列的顺序排序
  - 还记得无序的文件叫什么吗？
- 可以二分查找
  - 有什么问题？
  - 每次需要一次I/O读一个Page



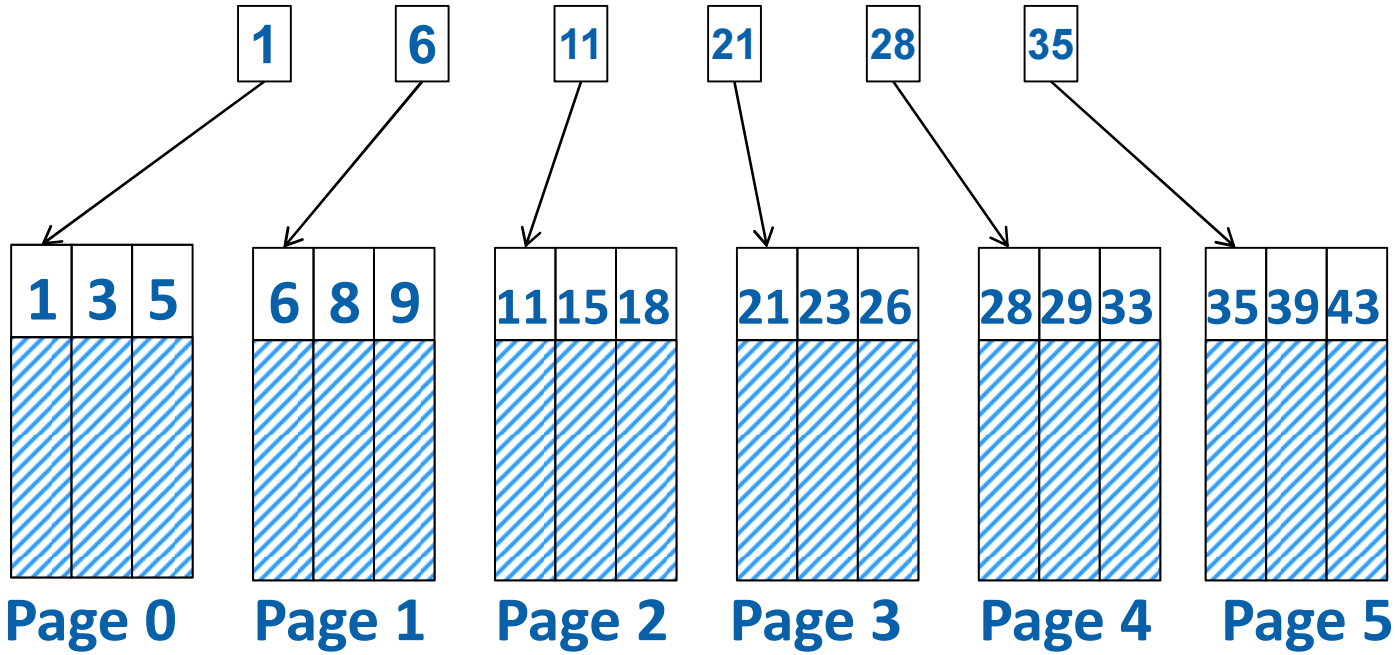
排序文件+稠密索引

- 在排序的文件上提取每个键组成一个稠密索引



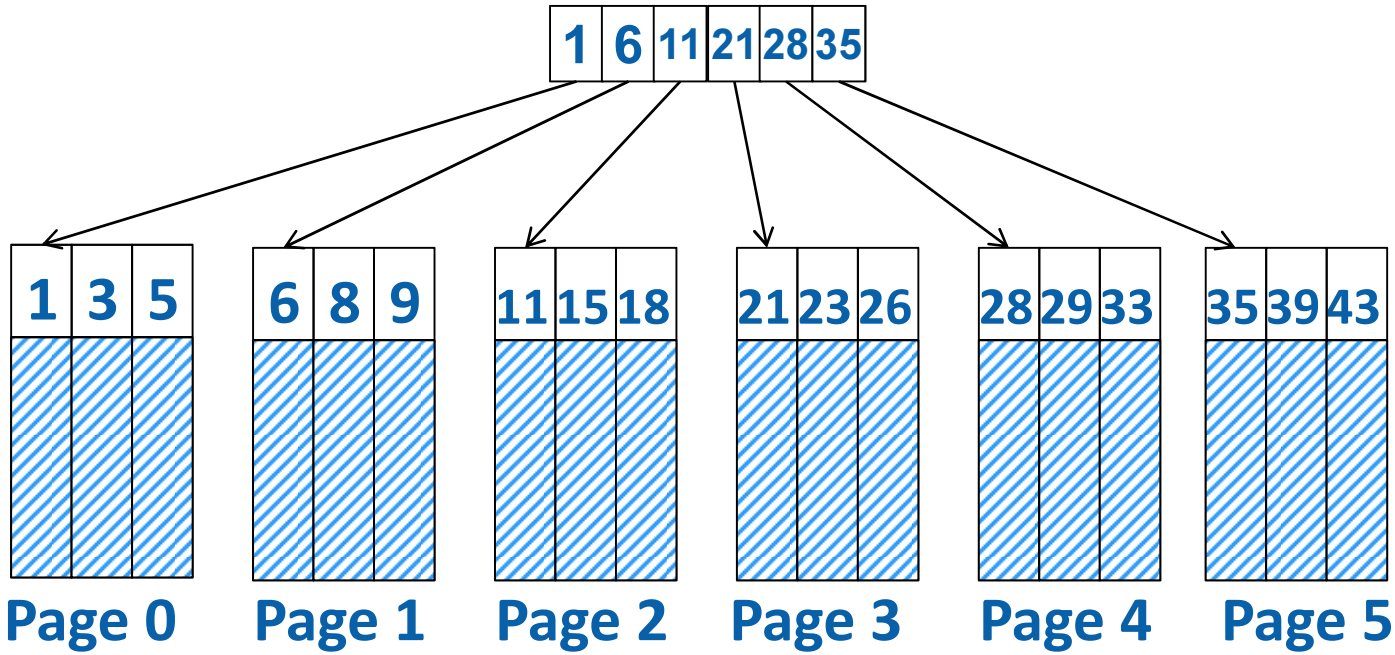
排序文件+稀疏索引

- 既然是聚簇（Clustered）的，可以每个Page只记录一个键
- 这样的索引是稀疏索引



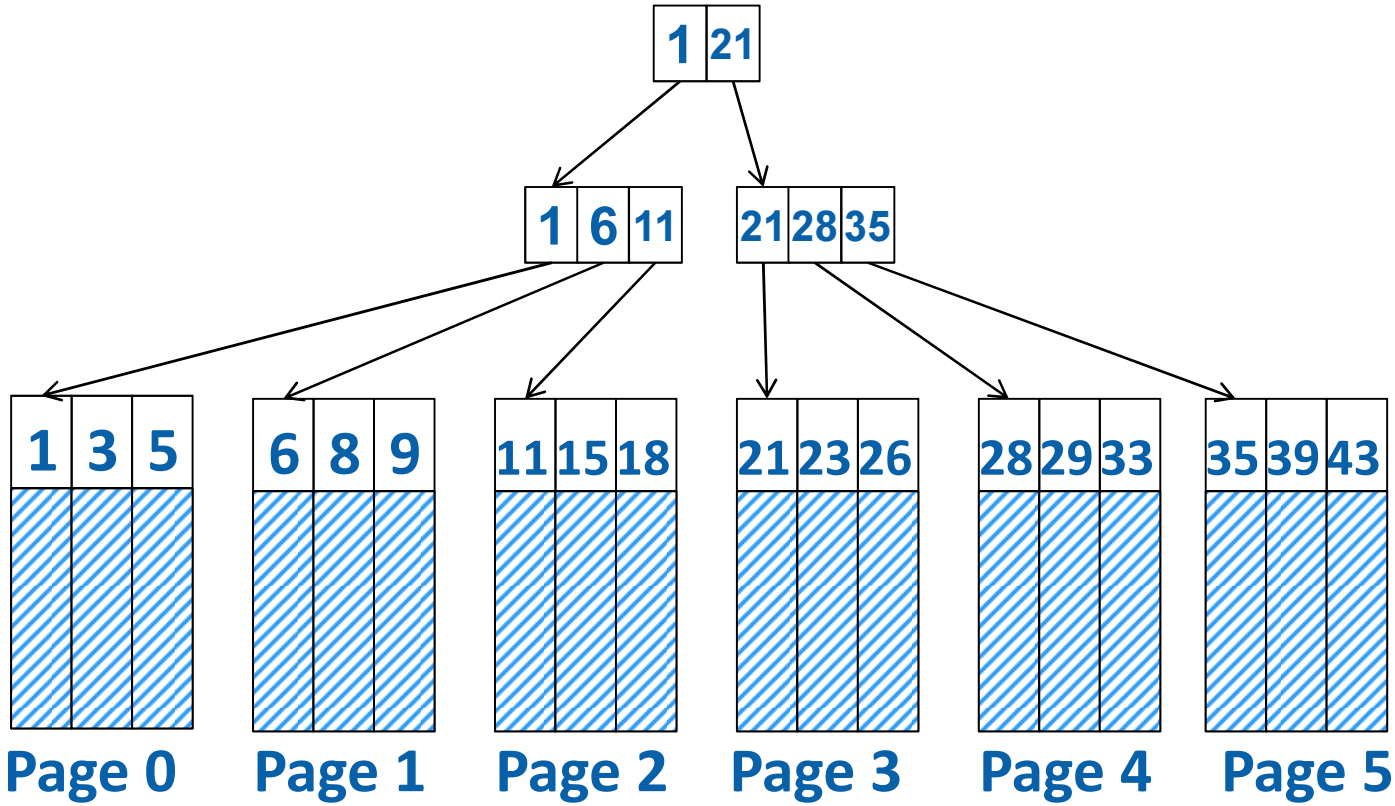
排序文件+稀疏索引

- 既然是聚簇（Clustered）的，可以每个Page只记录一个键
- 这样的索引是稀疏索引



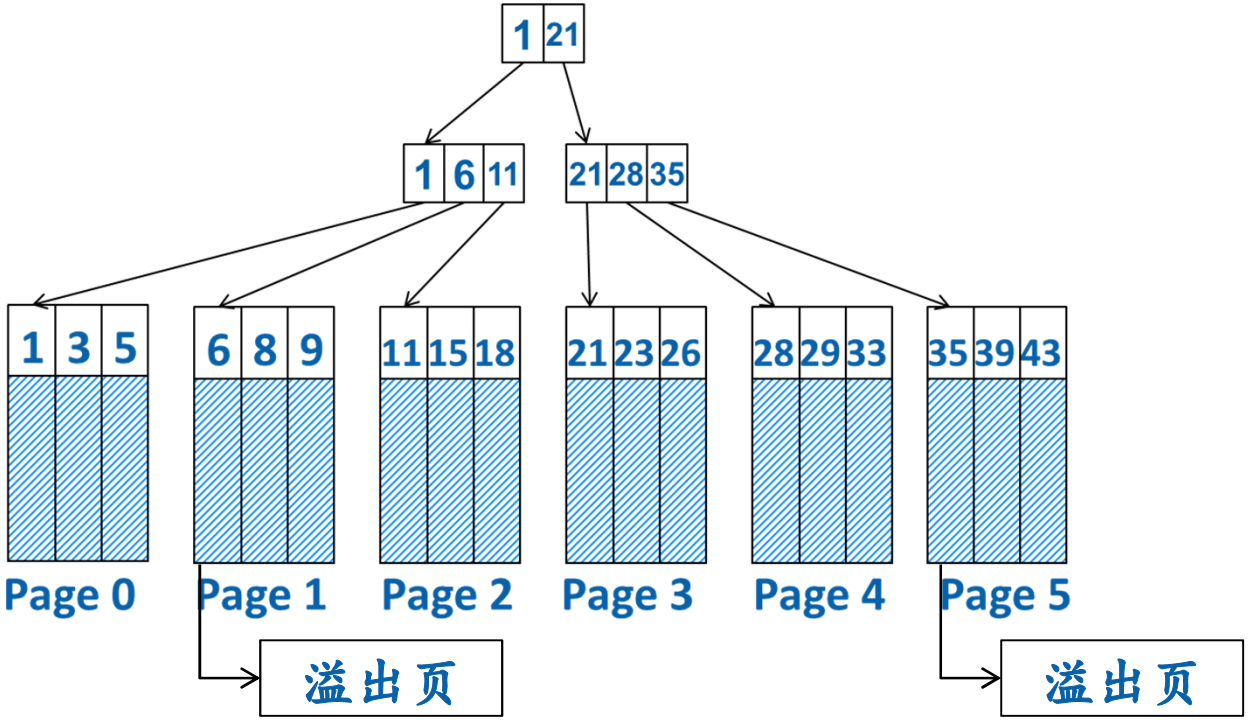
排序文件+多层索引

- 如果索引本身也比较大，那么可以进一步建立多层索引
  - 索引上再加索引



这种索引又称为ISAM (Indexed Sequential Access Method)

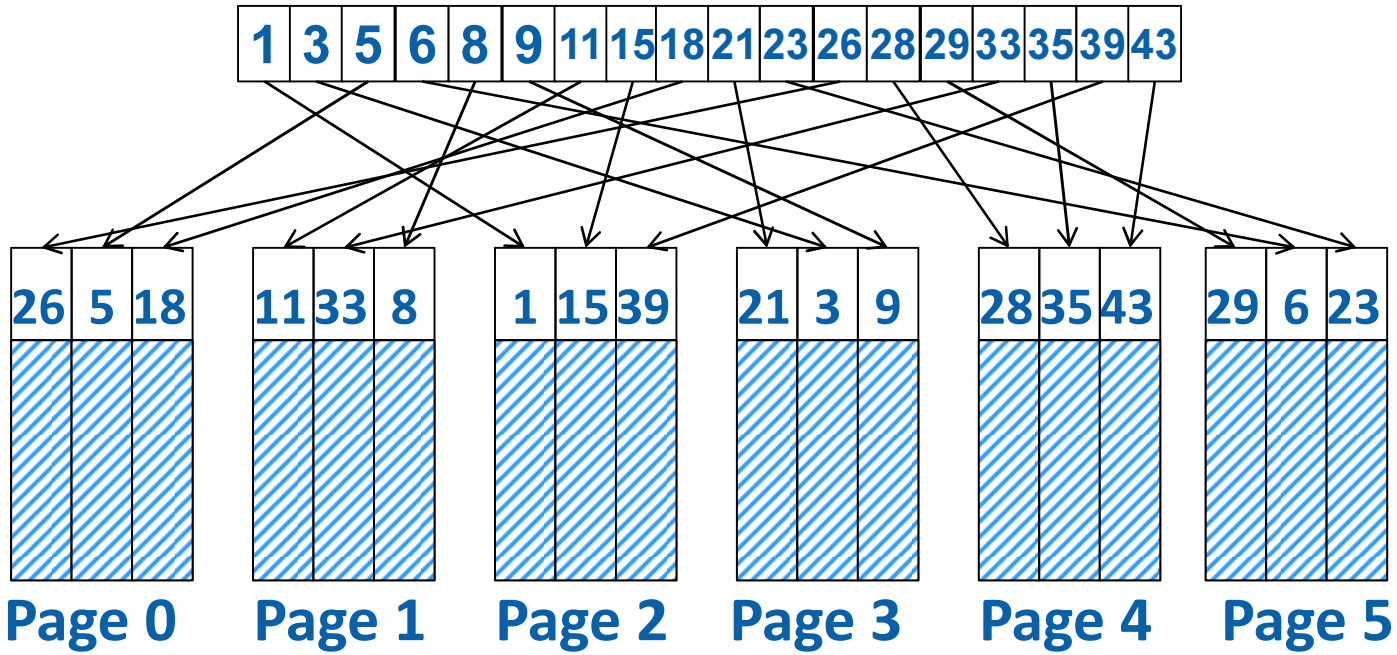
ISAM插入新记录怎么处理？



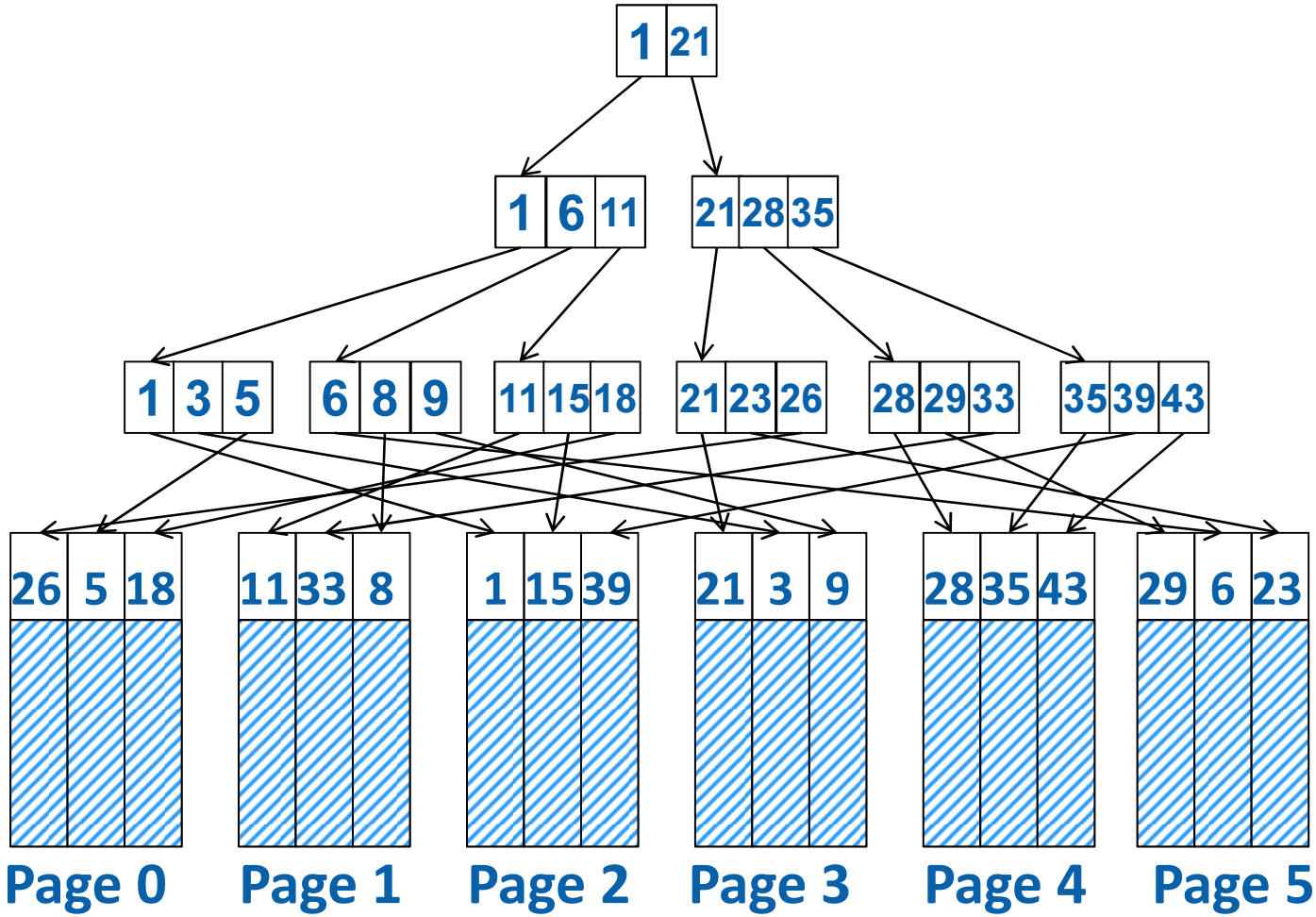
- 上面索引部分不变
- 数据部分增加溢出页
  - 但是溢出页增多，会影响性能

堆文件+稠密索引

- 在无序文件上提取每个键组成一个稠密索引
  - 为什么不可能是稀疏索引？

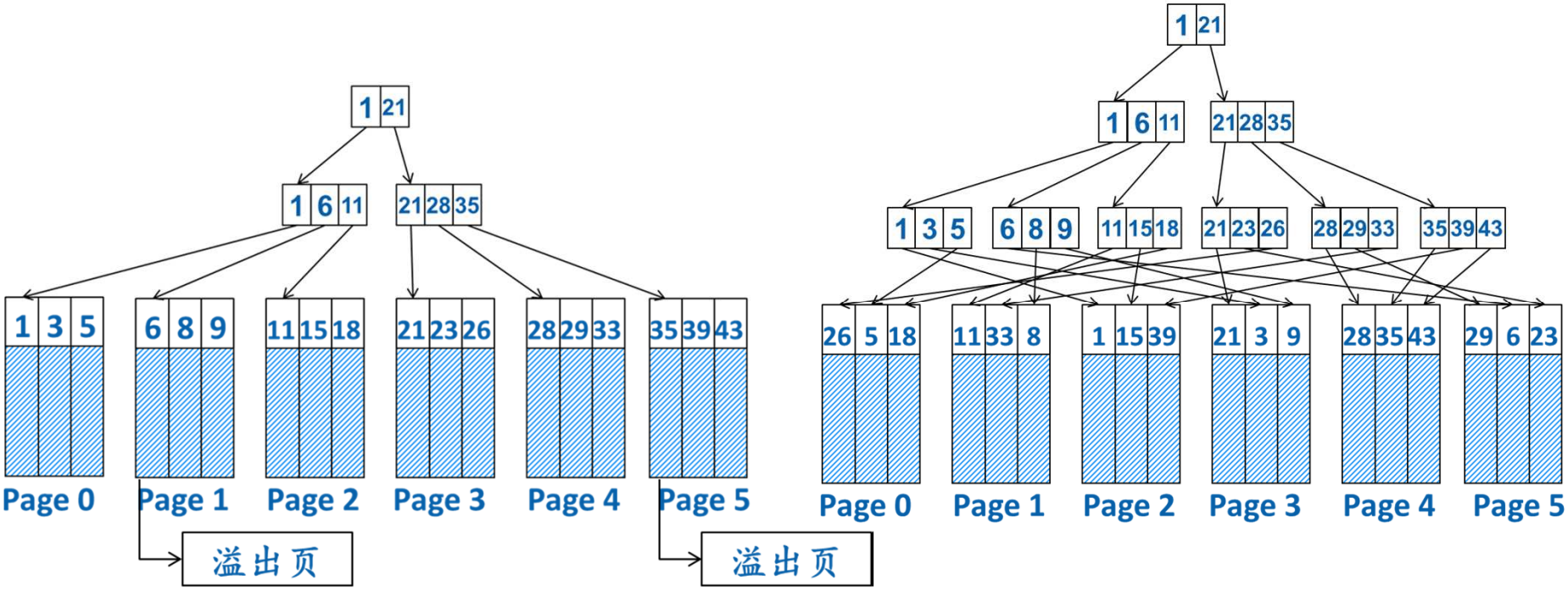


堆文件+多层稠密索引



插入新记录怎么办？

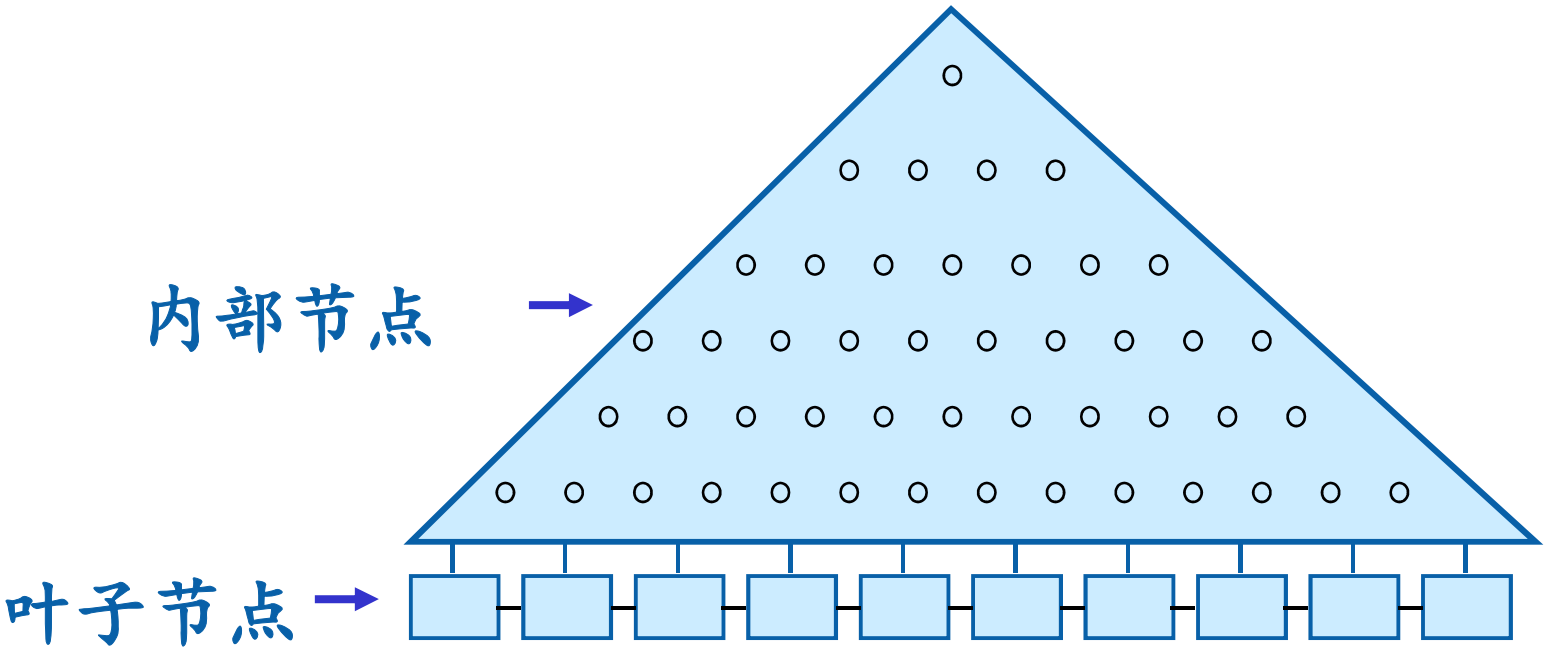
本质是静态索引



- 难以支持频繁的插入、删除操作

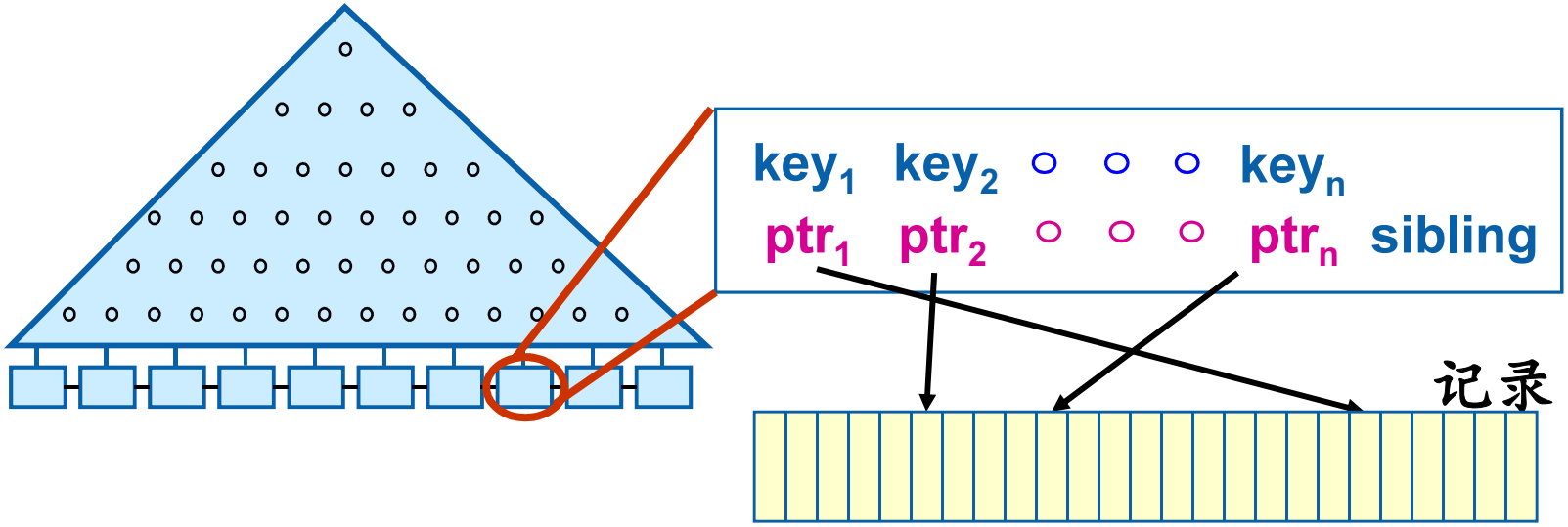
☞ B^+-Tree

B+-Trees



- 每个节点是一个page
- 所有key存储在叶子节点
- 内部节点完全是索引作用

叶子节点

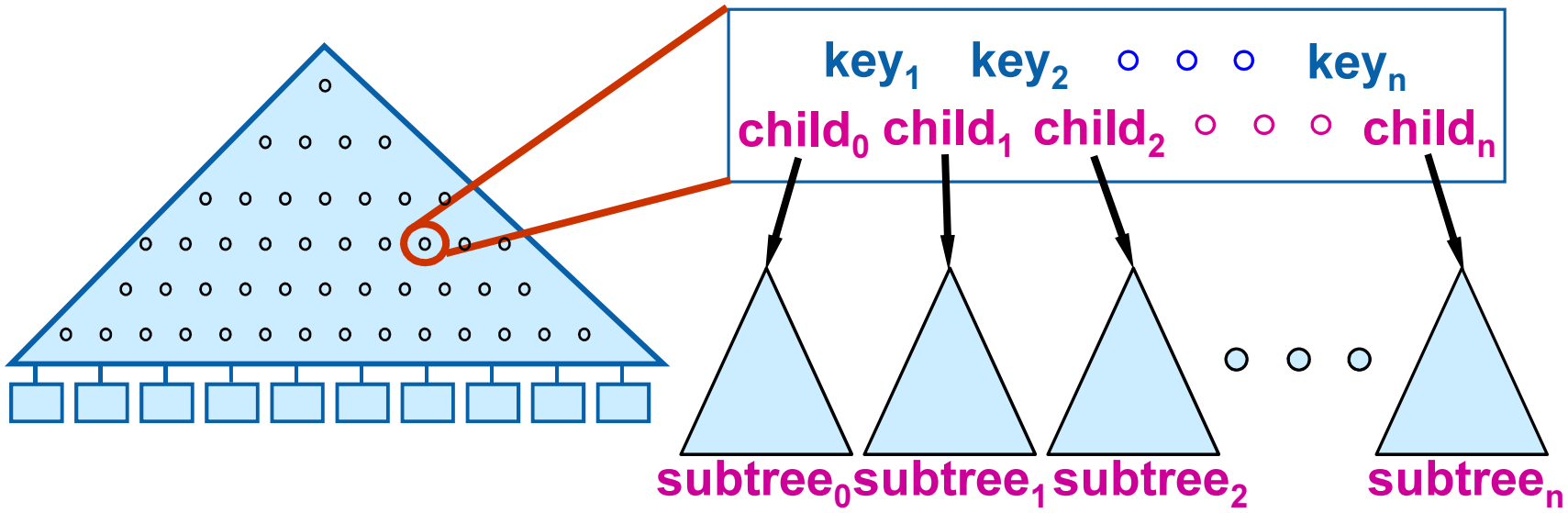


Keys 按照从小到大顺序排列：$key_1 < key_2 < \cdots < key_n$

叶节点自左向右也是从小到大排序，以sibling pointer链起来

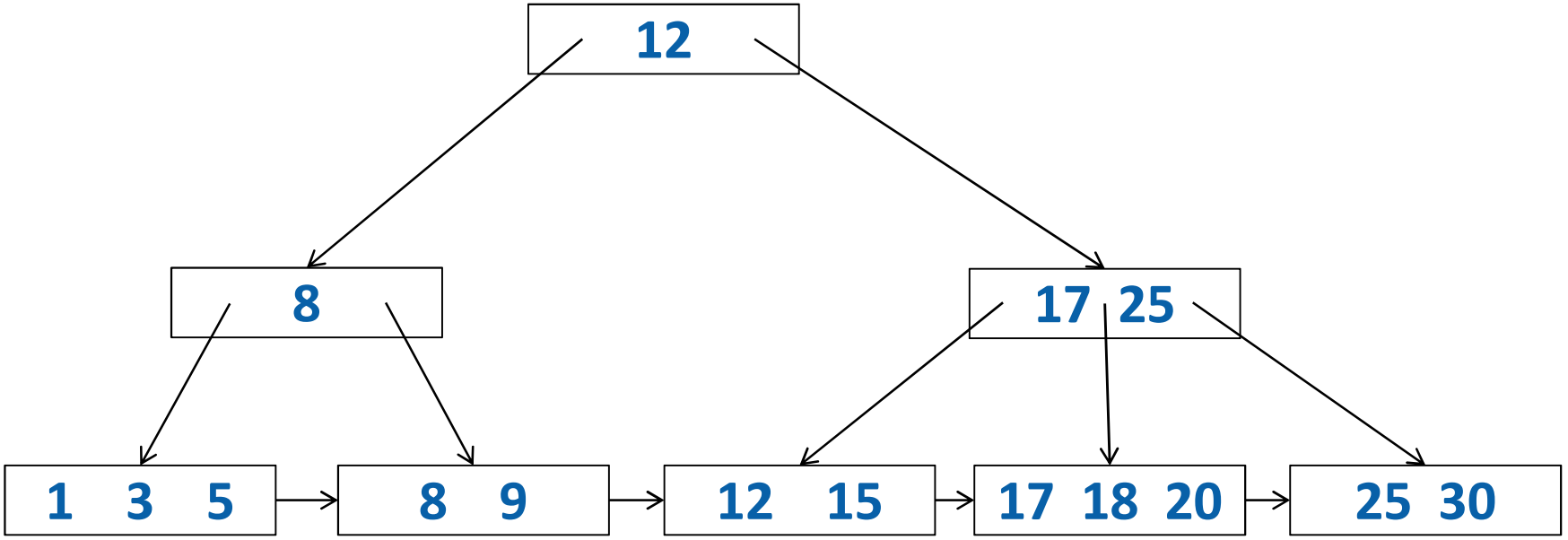
(ptr= record ID; sibling = page ID)

内部节点



$$subtree_0 < key_1 \leq subtree_1 < key_2 \cdots < key_n$$

举例

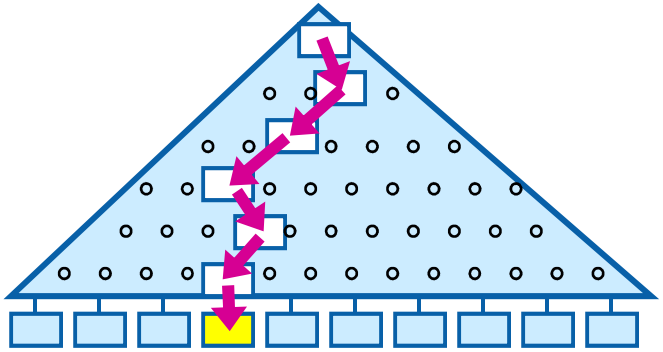


假设每个节点的child/pointer个数为B=3

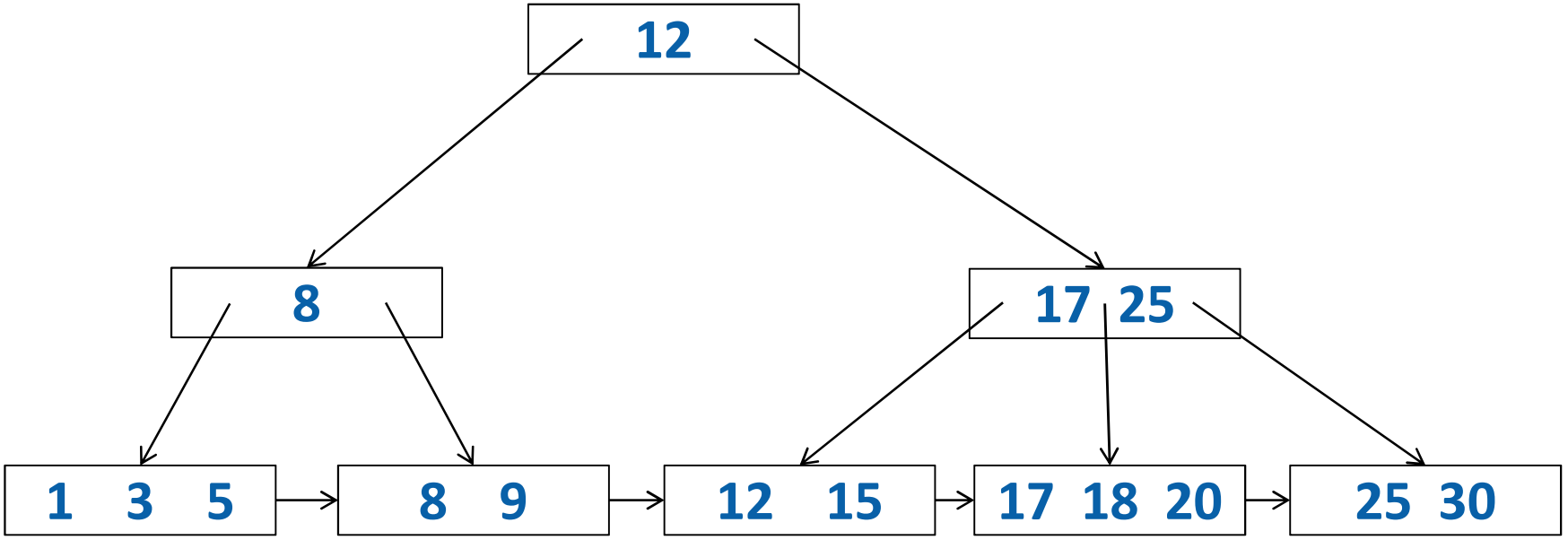
B+-Tree: Search

Search:

- 从根节点到叶节点
- 每个节点中进行二分查找
  - 内部节点：找到包括search key的子树
  - 叶节点：找到匹配

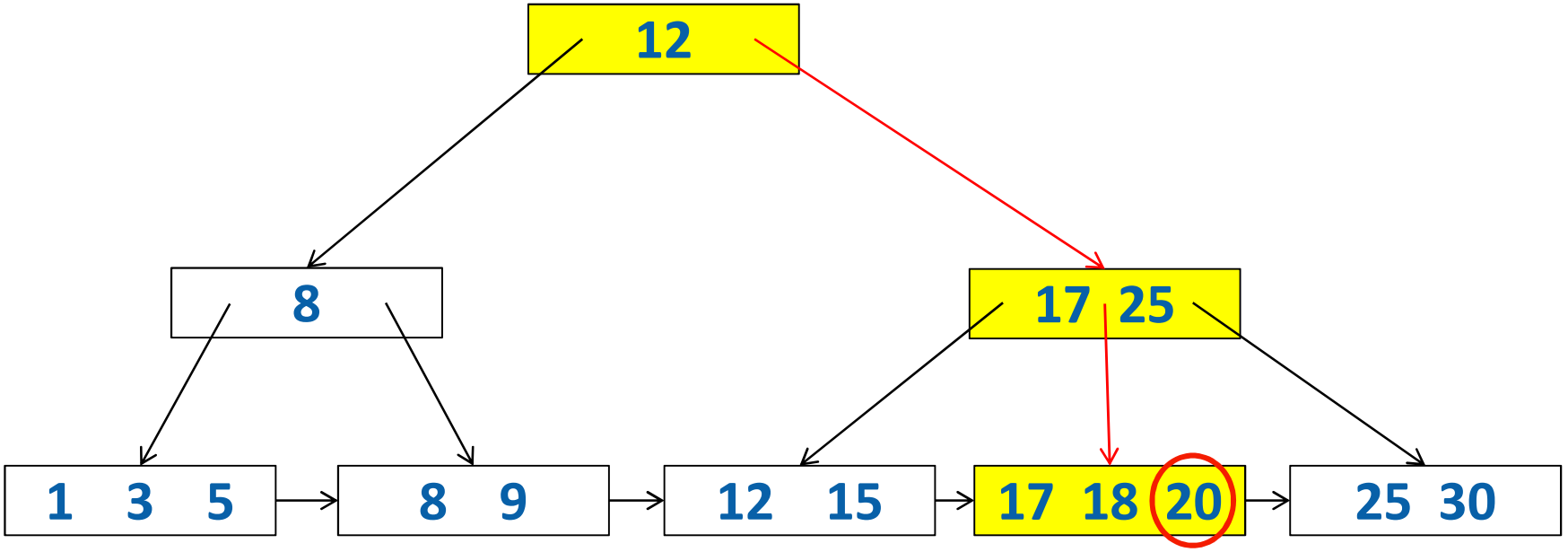


举例 Search(20)



假设每个节点的child/pointer个数为B=3

举例 Search(20)

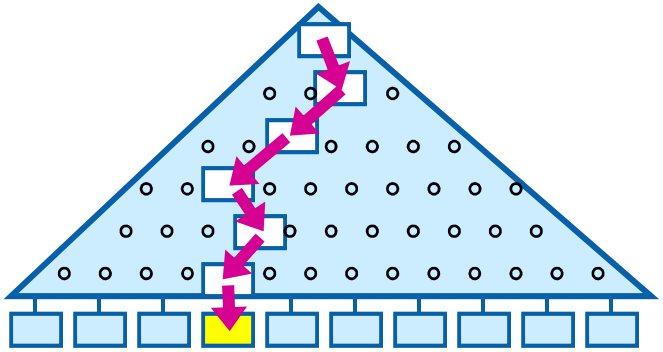


假设每个节点的child/pointer个数为B=3

B+-Tree: Search

Search代价

- 共有N个key
- 每个节点的child/pointer个数为B
- 总I/O次数=树高： $O(\log_B N)$
- 总比较次数
  - 每个节点内部二分查找： $O(\log_2 B)$
  - $O(\log_B N) \times O(\log_2 B) = O(\log_2 N)$

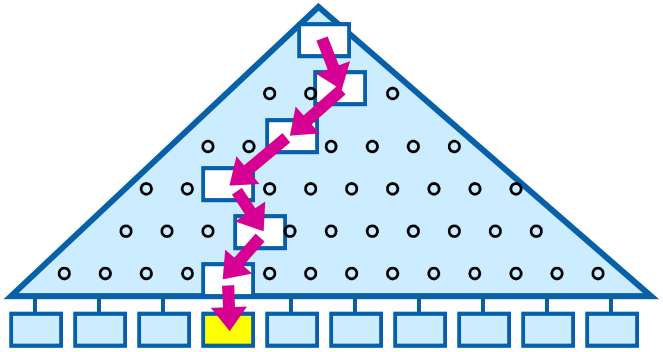


- 注意：为了保证性能，B必须有一个下限
  - 所以，在B+-Tree的定义中要求
    - 根节点至少有2个孩子
    - 其它节点至少是半满

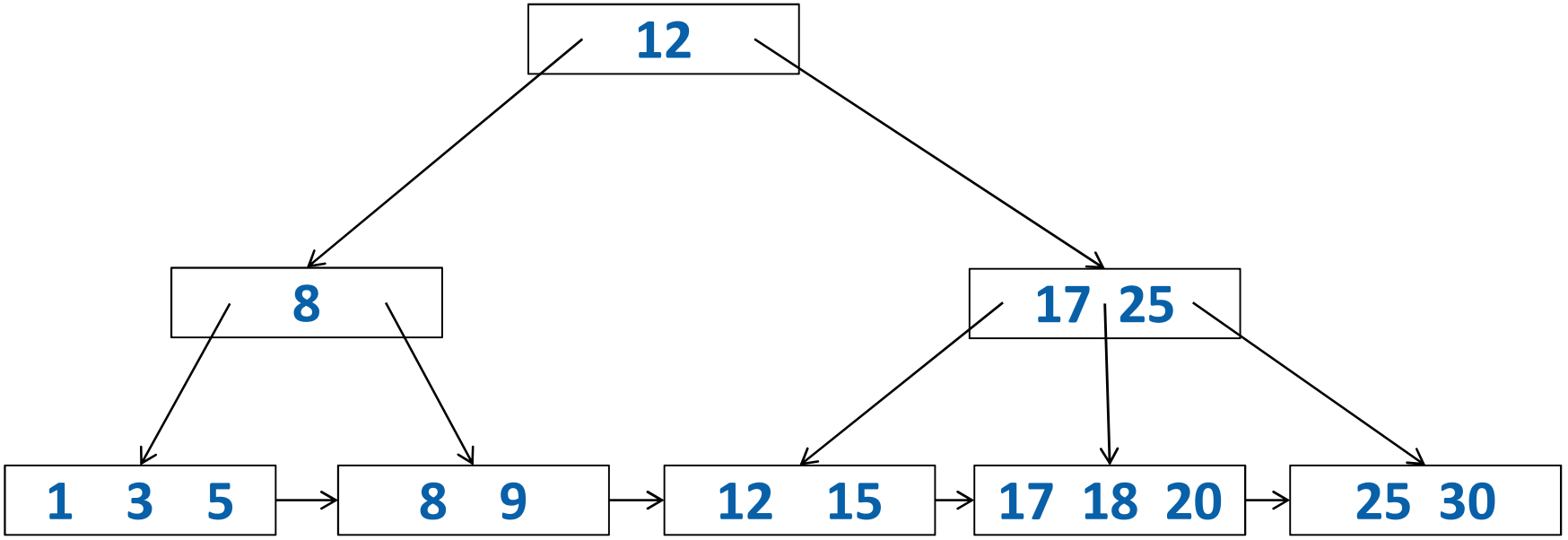
B+-Tree: Insertion

Insertion

- Search 然后在节点中插入
- 叶节点未满，插入叶节点
- 叶节点满了，node split(节点分裂)



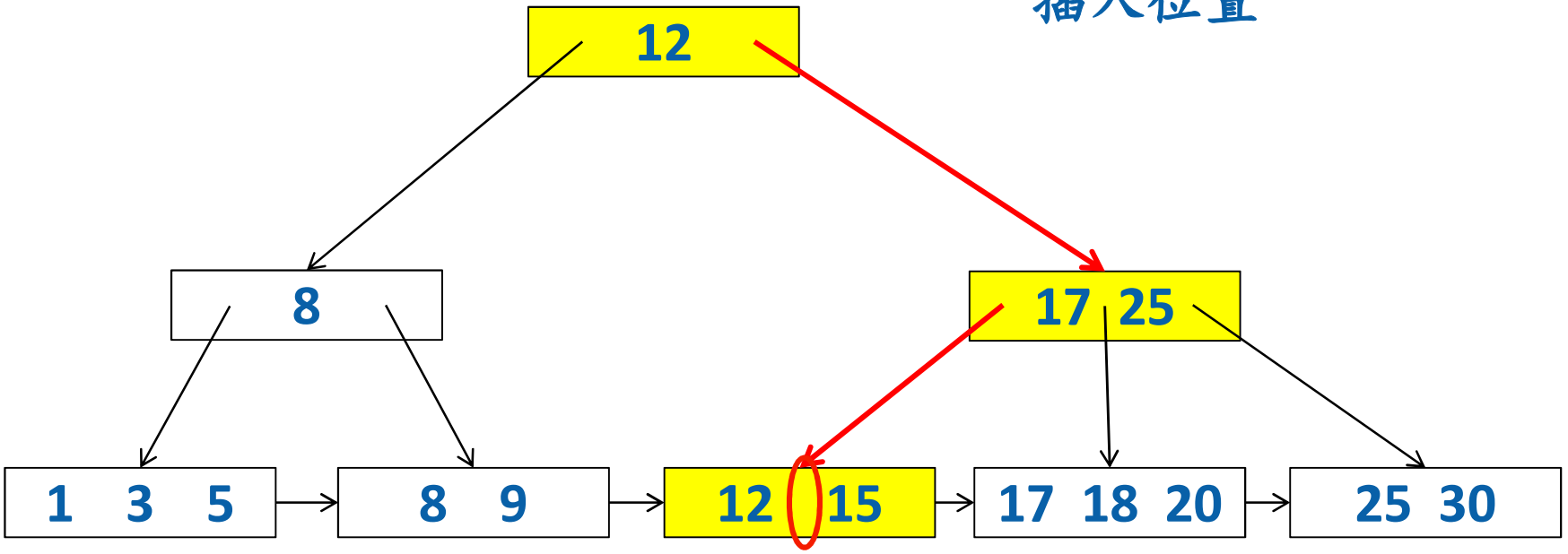
举例 Insert(14)



假设每个节点的child/pointer个数为B=3

举例 Insert(14)

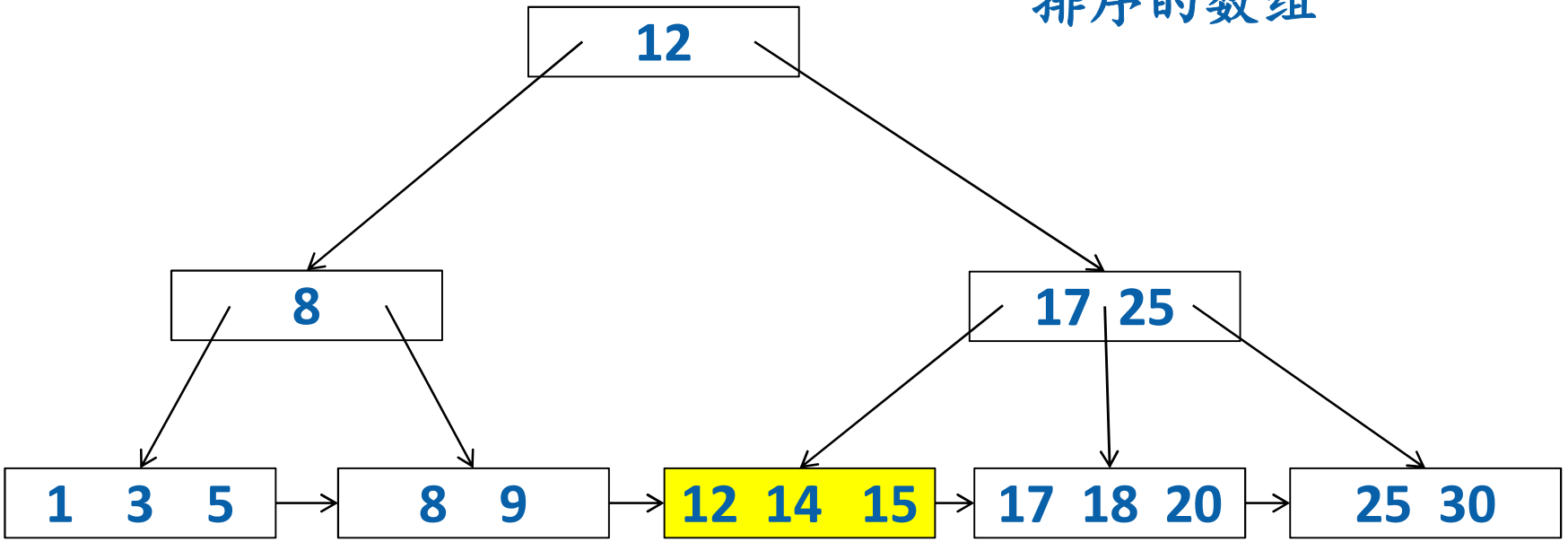
通过search找到
插入位置



假设每个节点的child/pointer个数为B=3

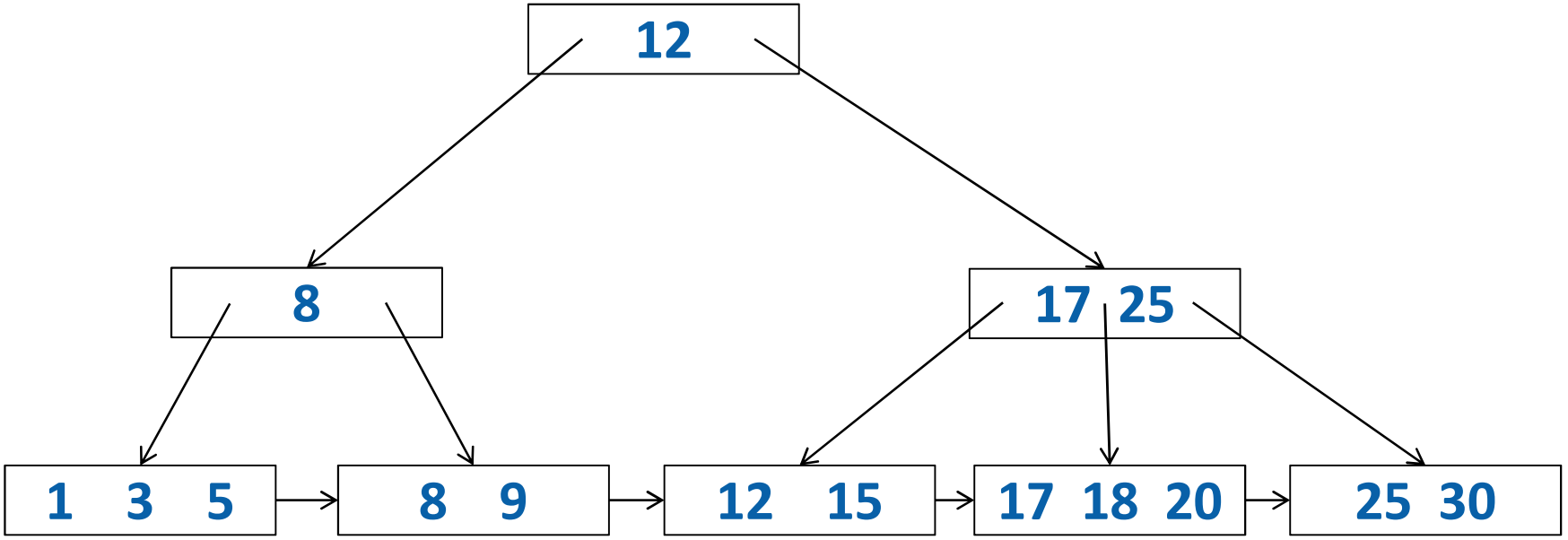
举例 Insert(14)

叶子未满，插入到已排序的数组



假设每个节点的child/pointer个数为B=3

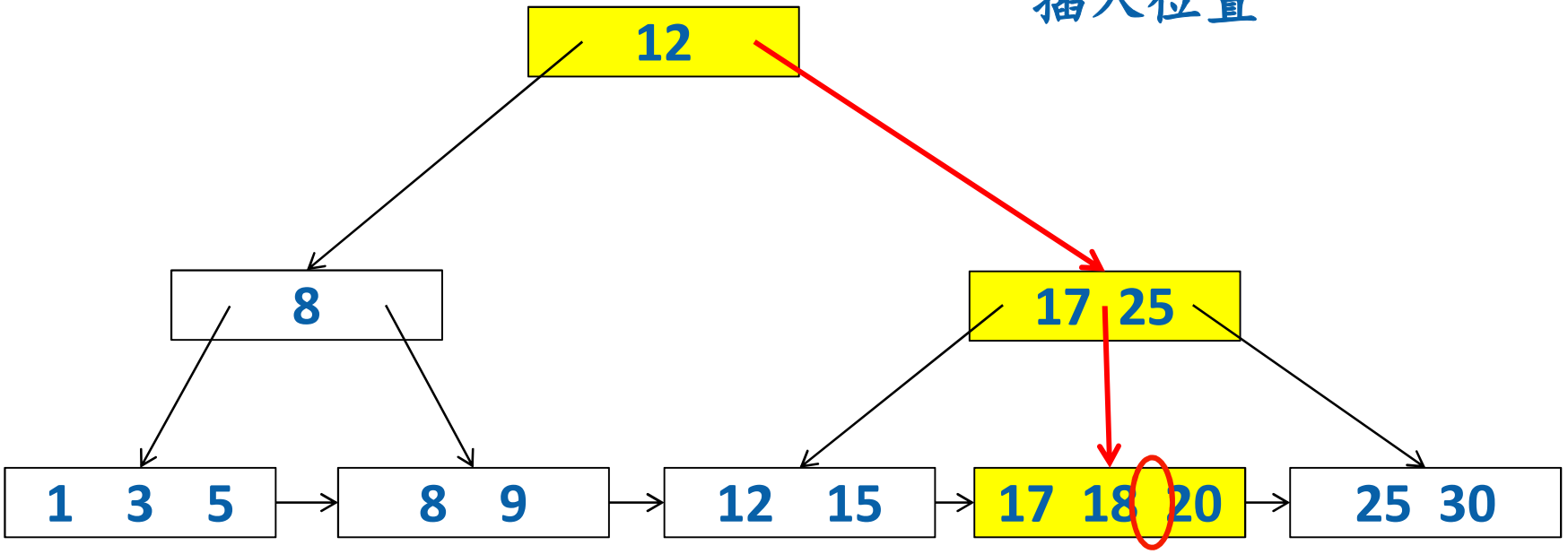
举例 Insert(19)



假设每个节点的child/pointer个数为B=3

举例 Insert(19)

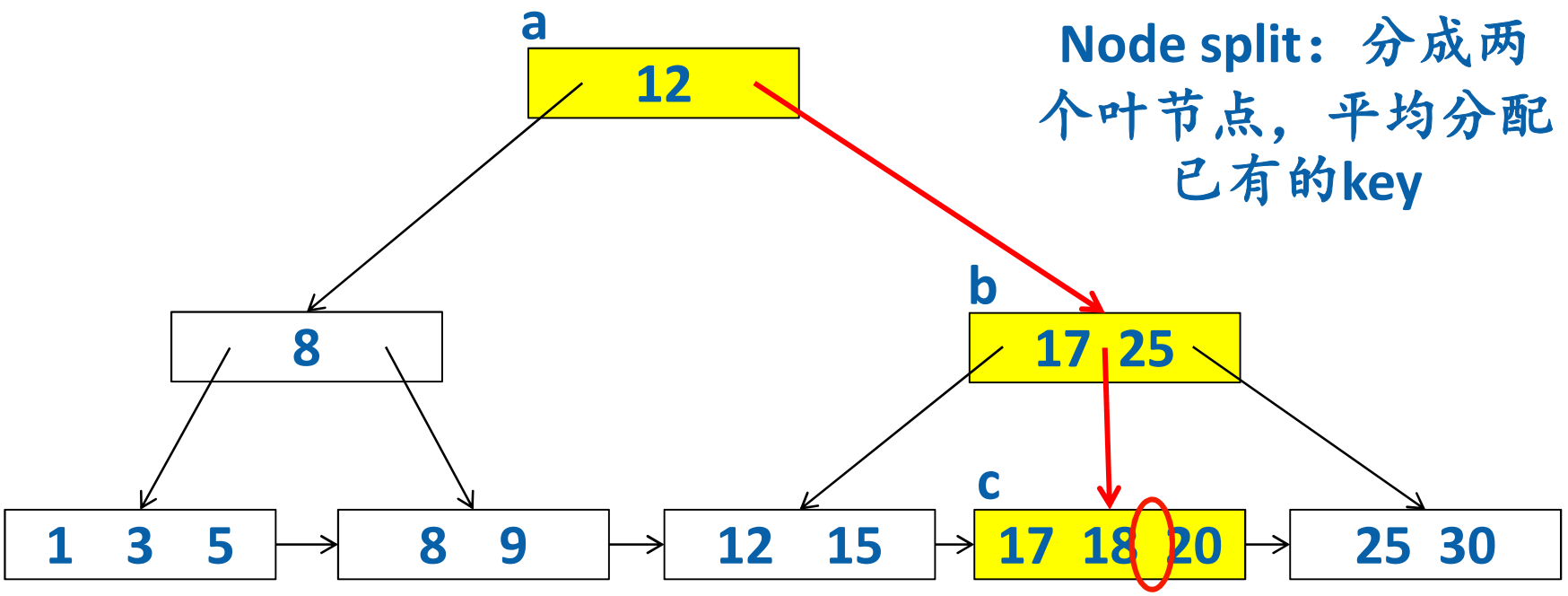
通过search找到
插入位置



假设每个节点的child/pointer个数为B=3

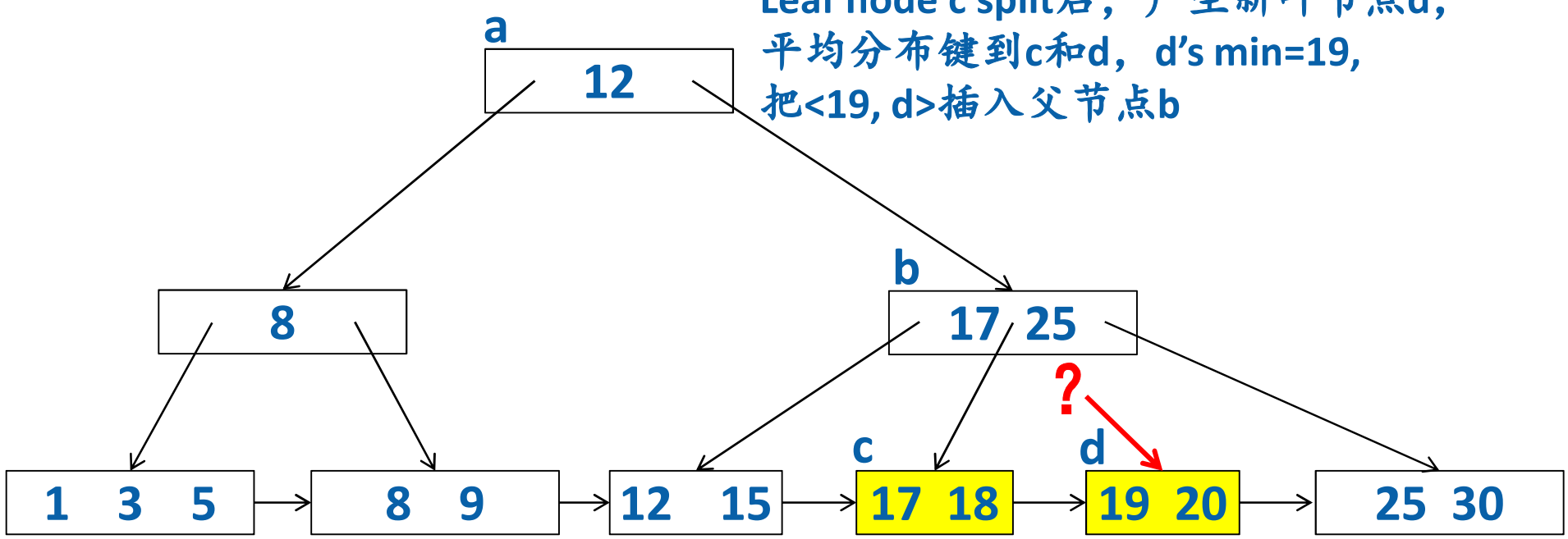
举例 Insert(19)

叶节点满了！
Node split：分成两个叶节点，平均分配已有的key



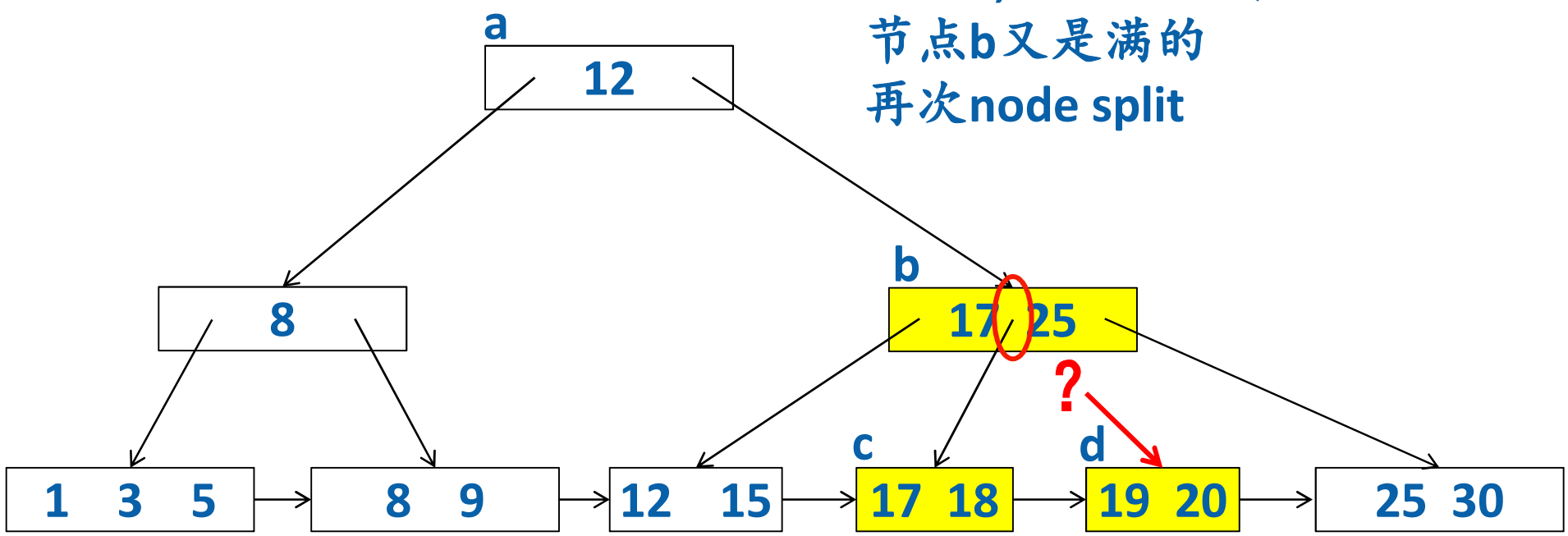
假设每个节点的child/pointer个数为B=3

举例 Insert(19)



假设每个节点的child/pointer个数为B=3

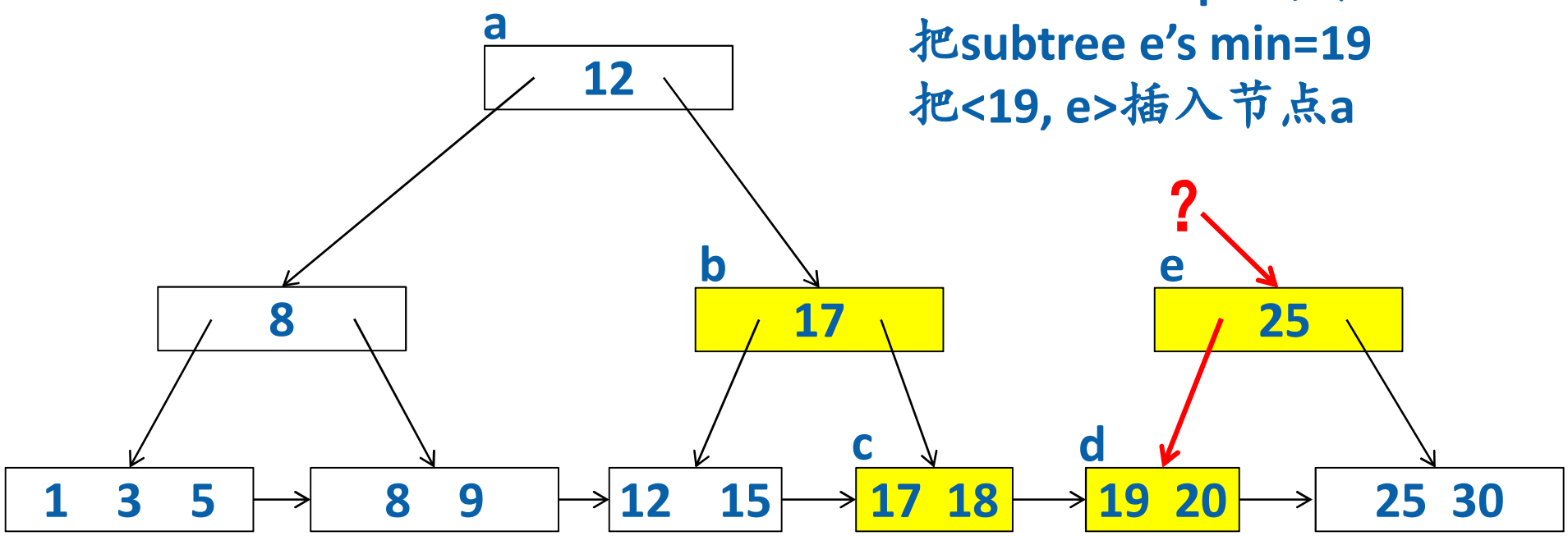
举例 Insert(19)



假设每个节点的child/pointer个数为B=3

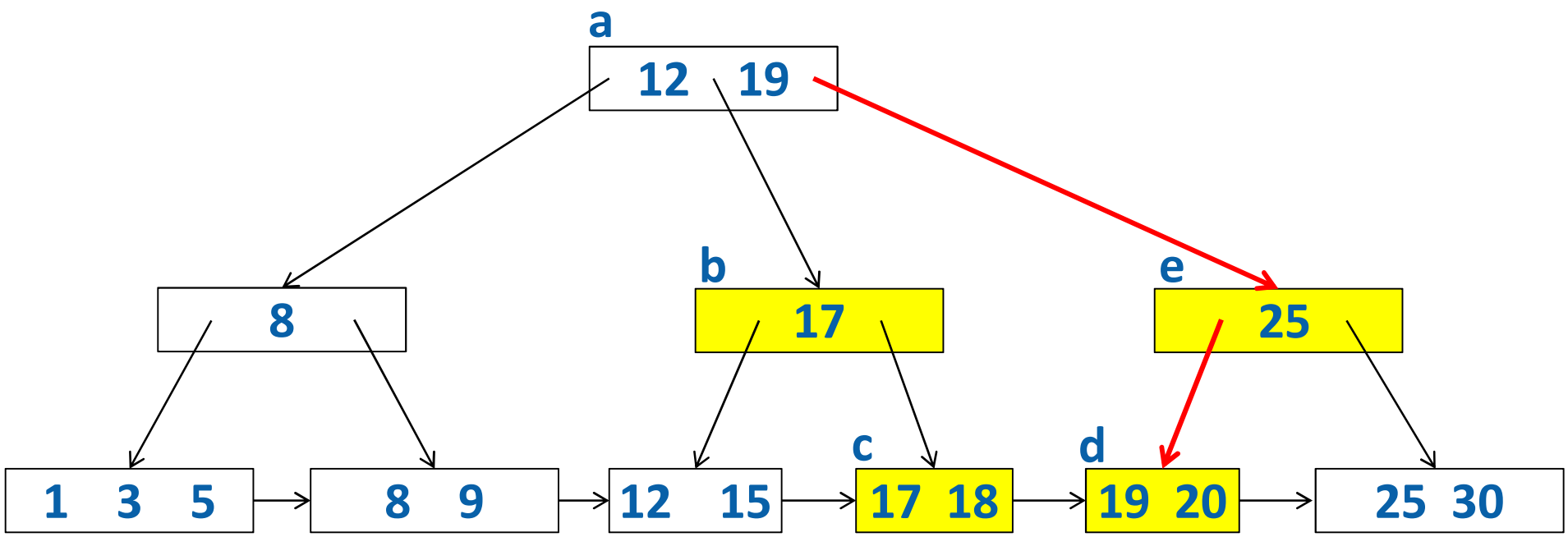
举例 Insert(19)

b 经过node split分裂出e
把subtree e's min=19
把<19, e>插入节点a



假设每个节点的child/pointer个数为B=3

举例 Insert(19)

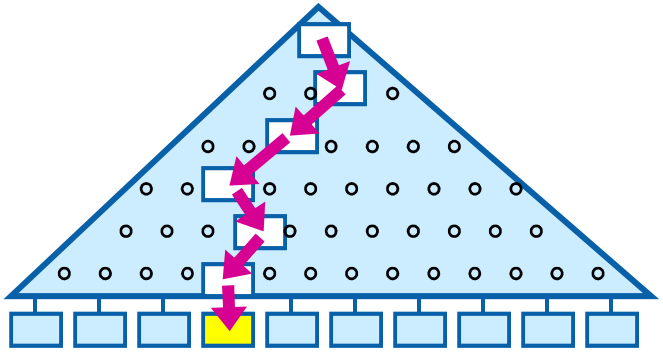


假设每个节点的child/pointer个数为B=3

B+-Tree: Deletion

Deletion

- Search 然后在节点中删除
- node merge?
  - 理论设计：当节点中key个数小于一半，需要merge，保证B的下限

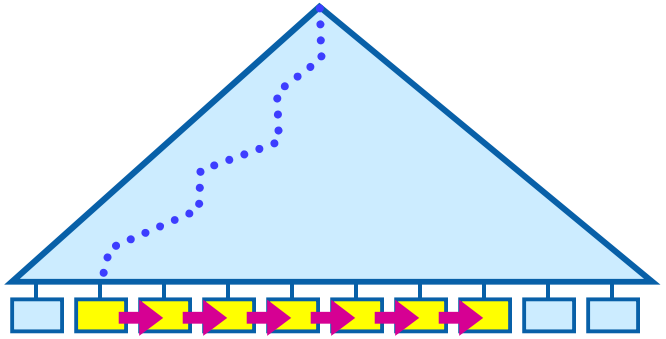


- 实际实现：数据总趋势是增长的
  - 只有节点为空时才node merge
  - 或者完全不进行node merge

B+-Tree： Range Scan

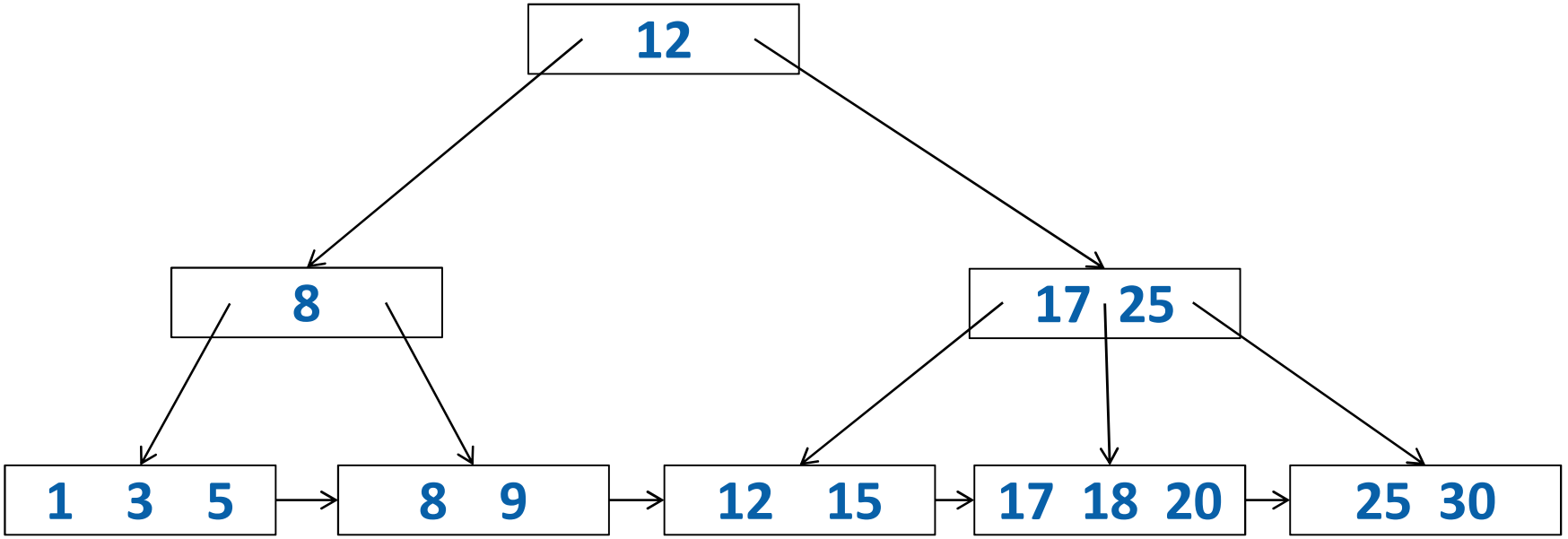
Range Scan

- 找到起始叶结点，包括范围起始值
- 沿着叶的链接读下一个叶结点
- 直至遇到范围终止值



举例

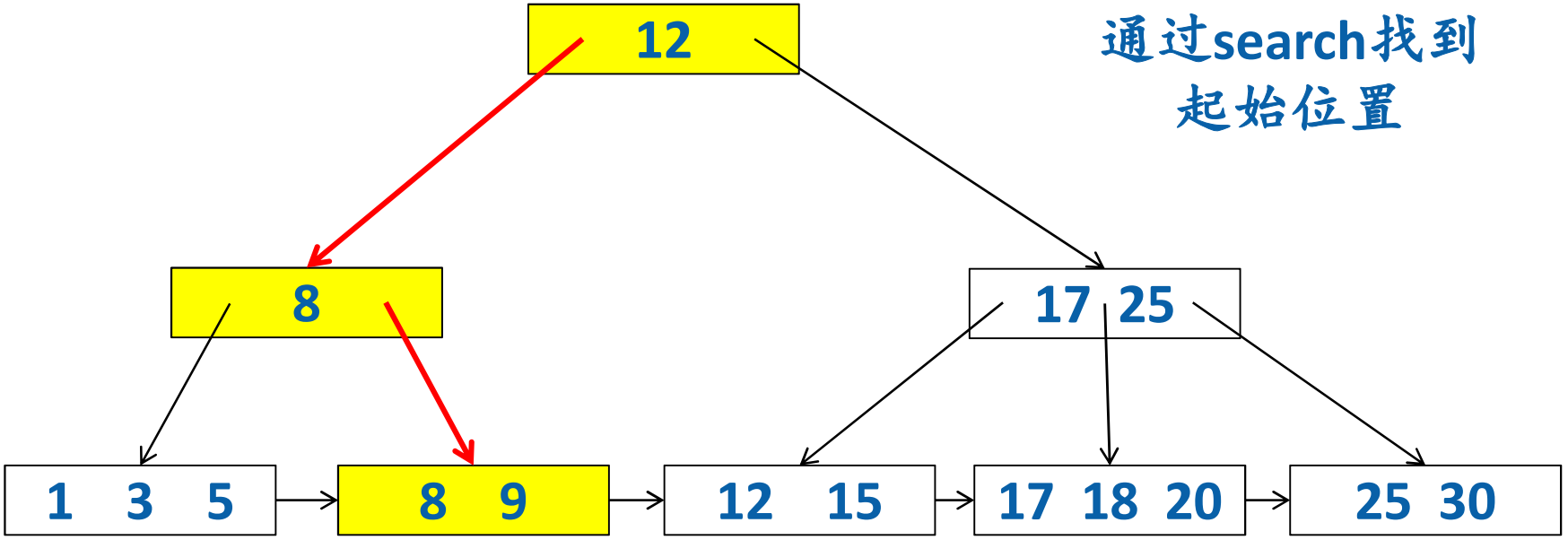
Range scan (9, 20):
获取[9, 20]区间的index entry



假设每个节点的child/pointer个数为B=3

举例

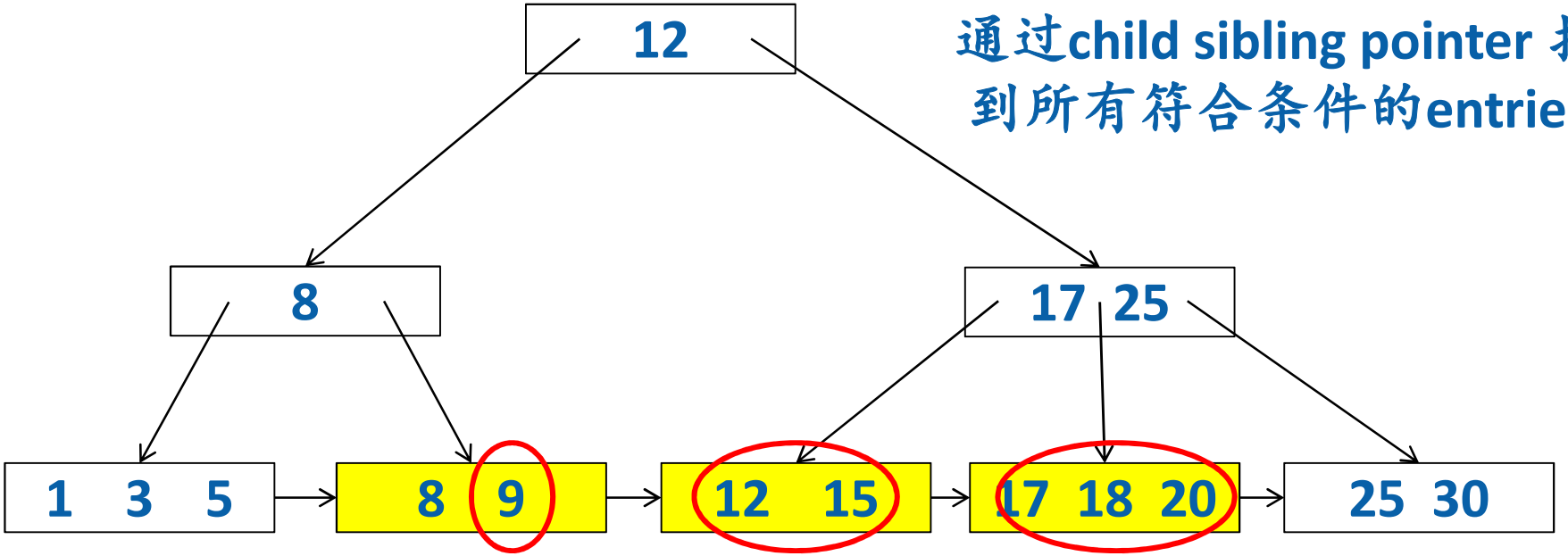
Range scan (9, 20):
获取[9, 20]区间的index entry



假设每个节点的child/pointer个数为B=3

举例

Range scan (9, 20):
获取[9, 20]区间的index entries



通过child sibling pointer 找到所有符合条件的entries

假设每个节点的child/pointer个数为B=3

索引种类

- Clustered index(聚簇索引，主索引)
  - 记录顺序就是index顺序
  - 可以认为记录就存在index中
  - 例如：primary key index
- Secondary index(辅助索引，二级索引)
  - 记录顺序不是index顺序
  - index中存储RecordId
  - 例如：在其它属性列上的索引
- 注意：一个表上只可能有一个主索引

索引数据访问

```
select Name, GPA
from Student
where Major = '计算机';
```

假设已经建立了以Major为key的二级索引

- 在二级索引中搜索 *Major* = *'计算机'*
- 对于每个匹配项，访问相应的tuple
- 读取Name和GPA

比较顺序访问与二级索引访问

- 顺序访问
  - 需要处理每一个记录
  - 顺序读每一个page
- 二级索引访问
  - 有选择地处理记录
  - 随机读相关的page

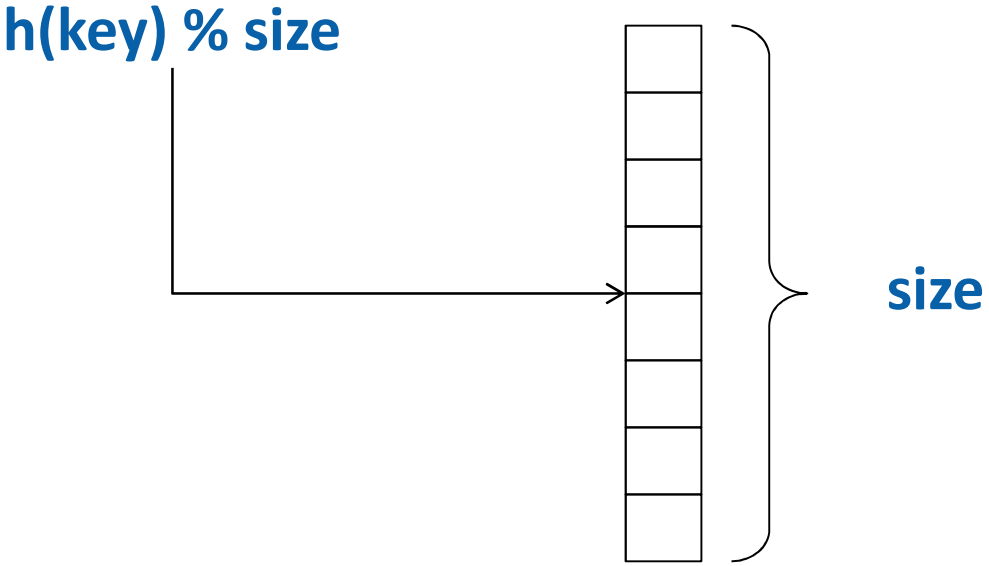
到底应该采用哪种方式呢？

- 由最终选中了多大比例的记录决定：selectivity
- 可以根据预测的selectivity、硬盘顺序读和随机读的性能，估算两种方式的执行时间
- 选择时间小的方案
- 这就是query optimizer的一个任务

Outline

- 索引的概念
- 树结构索引
- 哈希索引（散列表）
  - Hash function
  - Chained hashing
  - Extendible hashing
  - Linear hashing
- 位图索引
- 倒排索引

哈希表（Hash Table）



- 思路：索引查找➔地址/下标运算
- 哈希函数h( )?
- 冲突解决？

哈希函数h( )

- 目的：键key➔近乎随机的整数
- 通常对key的字节串进行运算

乘积型哈希函数h( )

```
uint32_t multhash(const char *key, int len) {
    uint32_t hash = INIT_VAL;
    for (uint32_t i = 0; i < len; ++i)
        hash = M * hash + key[i];
    return hash;
}
```

(例如: Kernighan and Ritchie's function, INIT_VAL=0, M=31)

复杂哈希函数举例

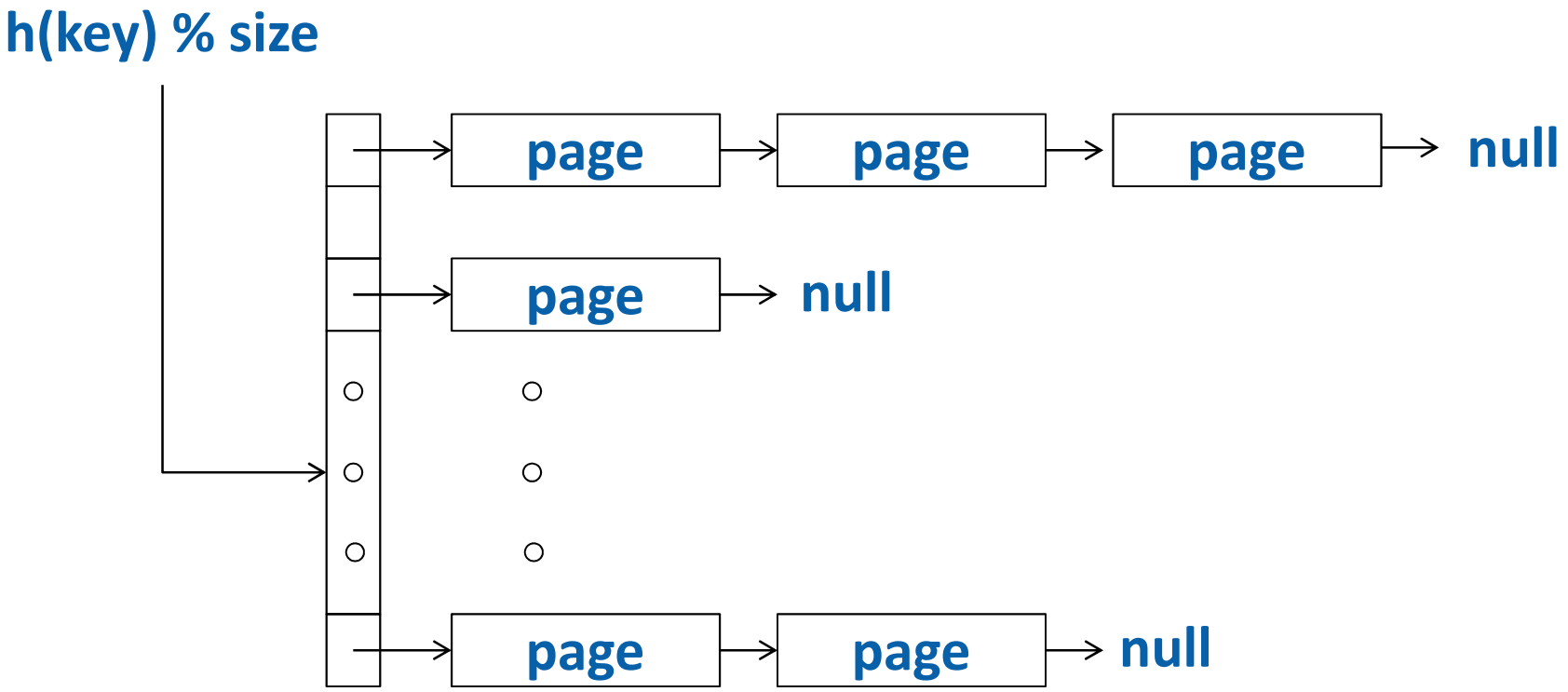
```
uint32_t jenkins_one_at_a_time_hash(char *key, size_t len)
{
    uint32_t hash, i;
    for(hash = i = 0; i < len; ++i)
    {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash;
}
```

冲突解决方法？

- 数据结构课上
  - 线性散列等

- 链表/溢出Page的方法

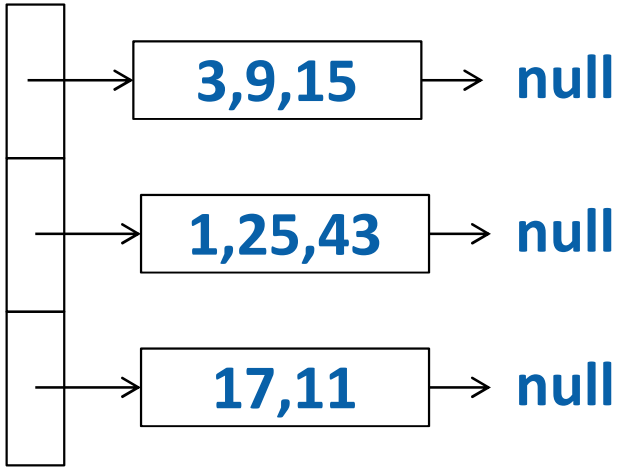
Chained Hash Table



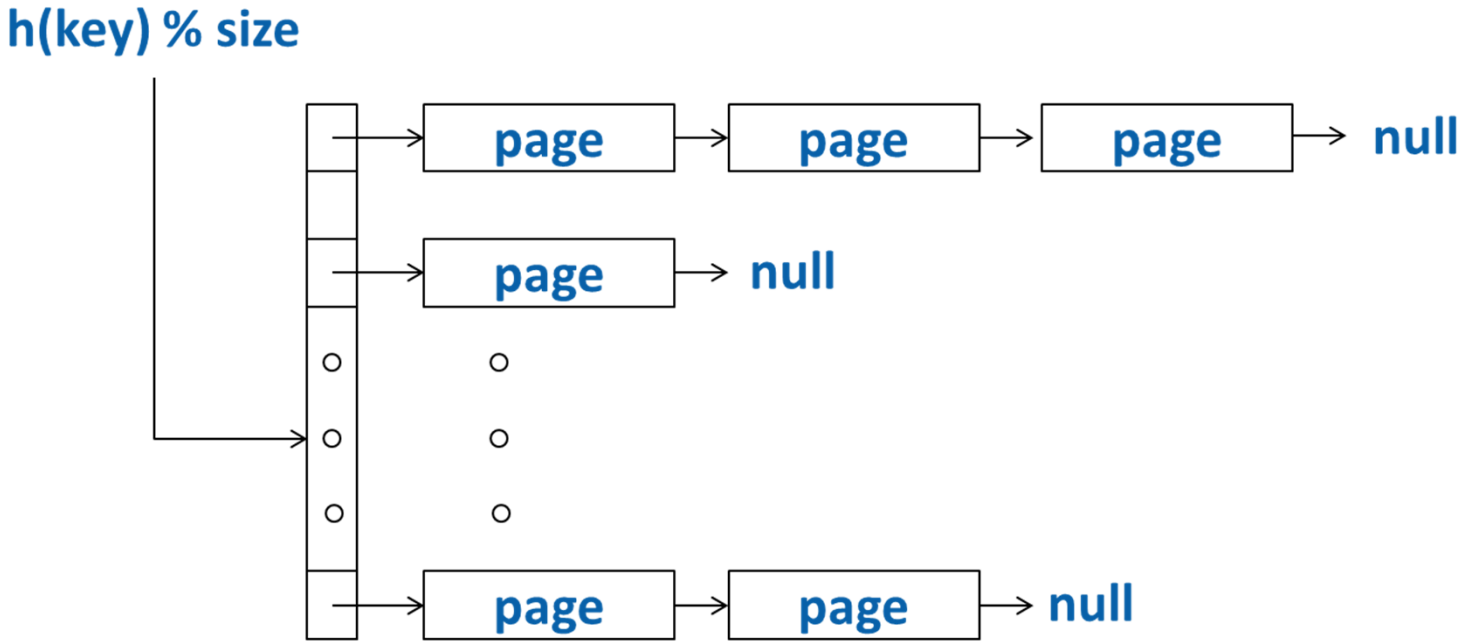
- 每个bucket由1到多个Page组成

举例

- Key: 1, 3, 25, 17, 9, 11, 43, 15
- h(key) = key
- Size = 3



Rehashing



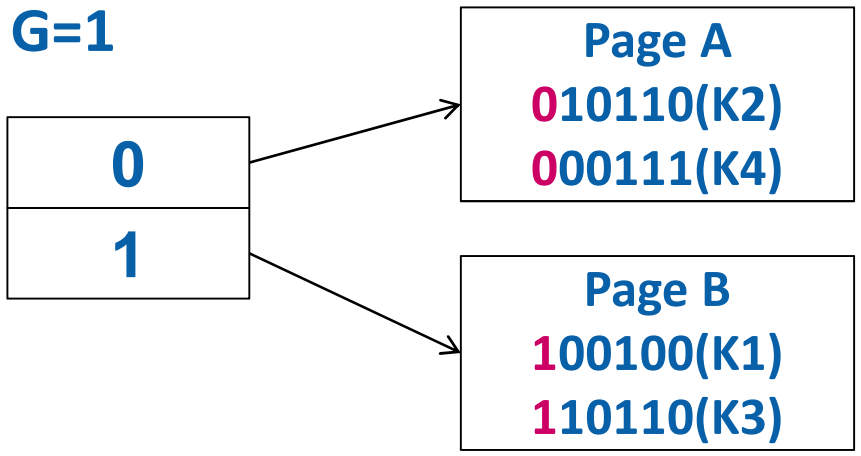
- 当一个bucket有太多的page时，性能变差！
- 所以，需要Rehashing
  - 把Table_Size变大： Table_size *= 2;
  - 重新建一遍Hash Table
- 代价昂贵！

为了减少Rehashing的代价

- 介绍两种经典的动态方案
- Extendible Hashing（可扩展哈希）
- Linear Hashing（线性哈希）

Extendible Hashing

h(k1)=100100
h(k2)=010110
h(k3)=110110
h(k4)=000111



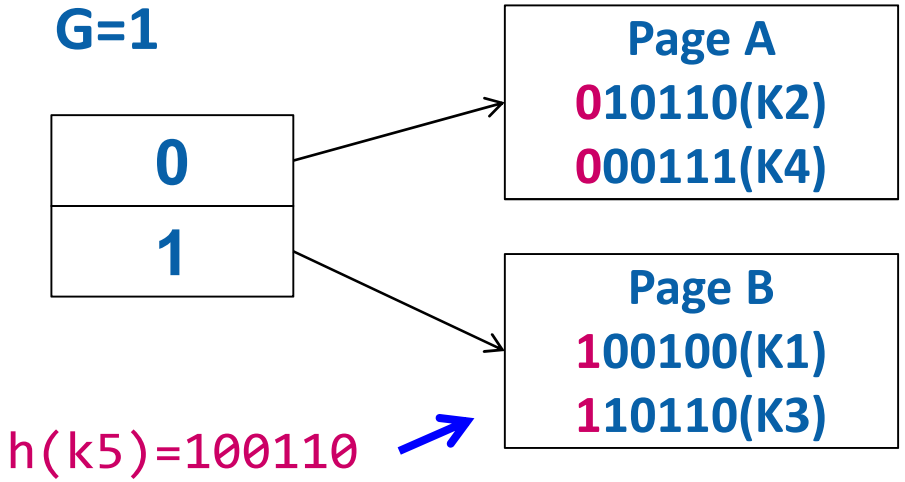
假设每个Page只能放2个项

- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G

Extendible Hashing

h(k1)=100100
h(k2)=010110
h(k3)=110110
h(k4)=000111

h(k5)=100110
怎么办？



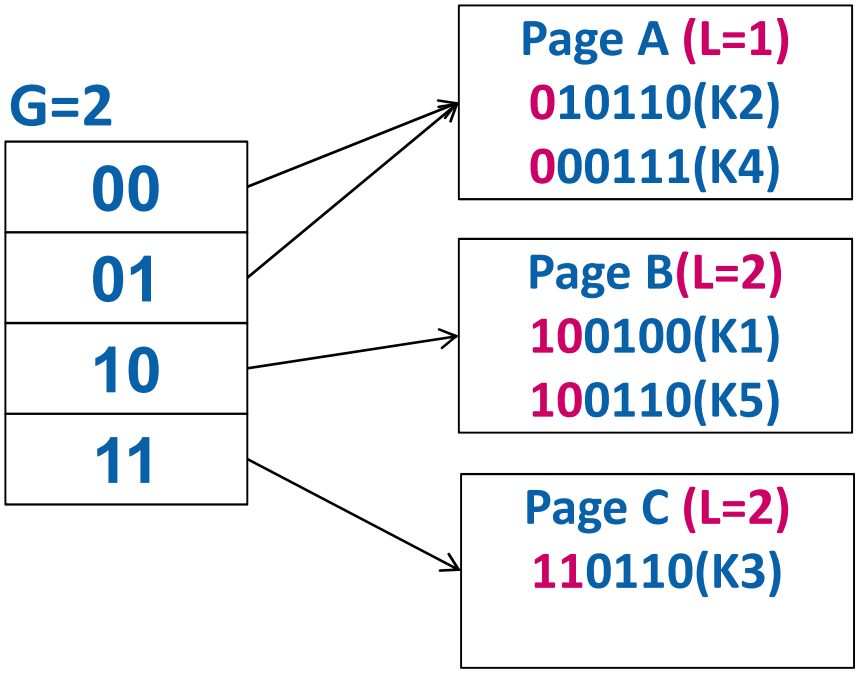
假设每个Page只能放2个项

- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G

Extendible Hashing

h(k1)=100100
h(k2)=010110
h(k3)=110110
h(k4)=000111

h(k5)=100110
怎么办？

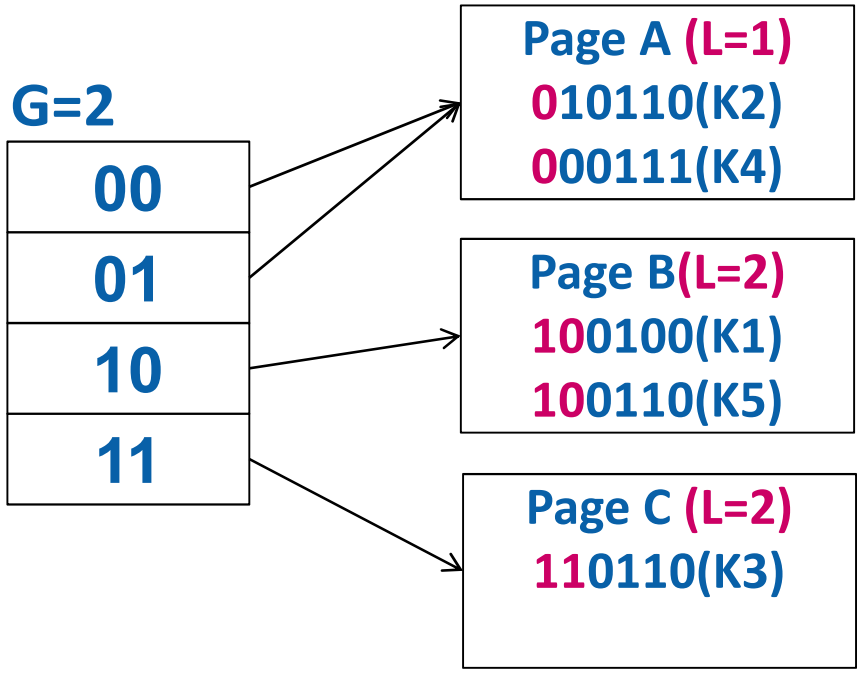


假设每个Page只能放2个项

- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G
- 变化
  - G++，Table Size变为2倍
  - Page B分裂为B和C
  - 多个桶指向同一个Page
- 每个Page中所有的项都有相同的前缀，前L位相同
  - L: Local depth
  - 共有2^{G-L}个桶指向这个Page

Extendible Hashing

h(k1)=100100
h(k2)=010110
h(k3)=110110
h(k4)=000111
h(k5)=100110
h(k6)=000000

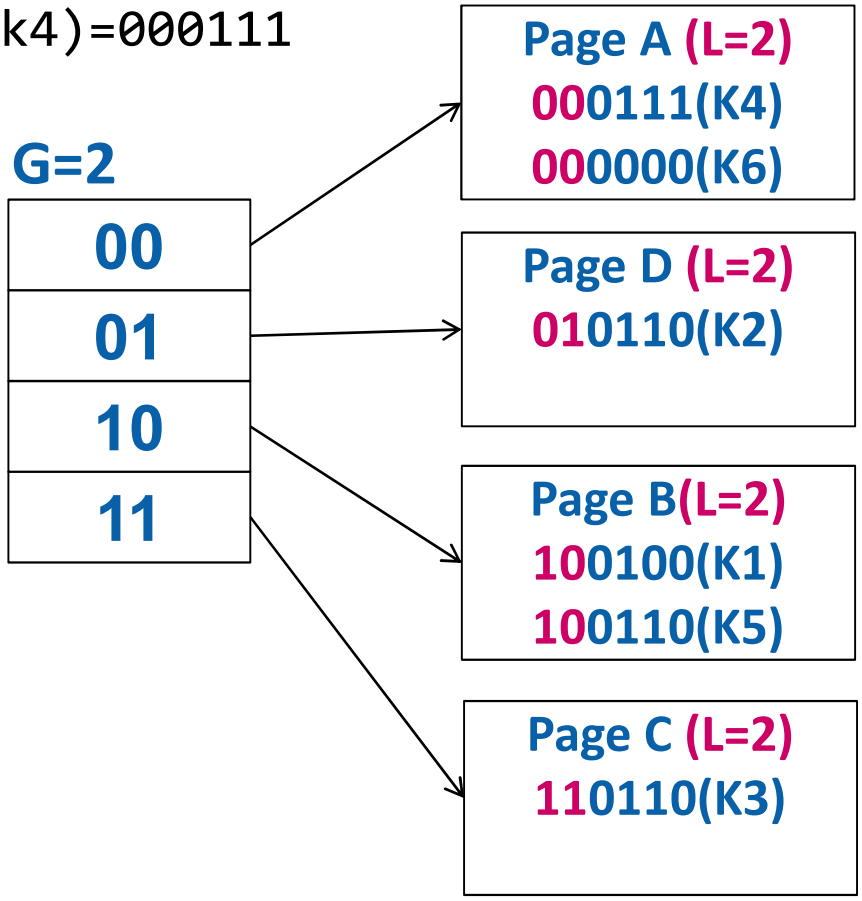


假设每个Page只能放2个项

- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G
- 每个Page中所有的项都有相同的前缀，前L位相同
  - L: Local depth
  - 共有2^{G-L}个桶指向这个Page

Extendible Hashing

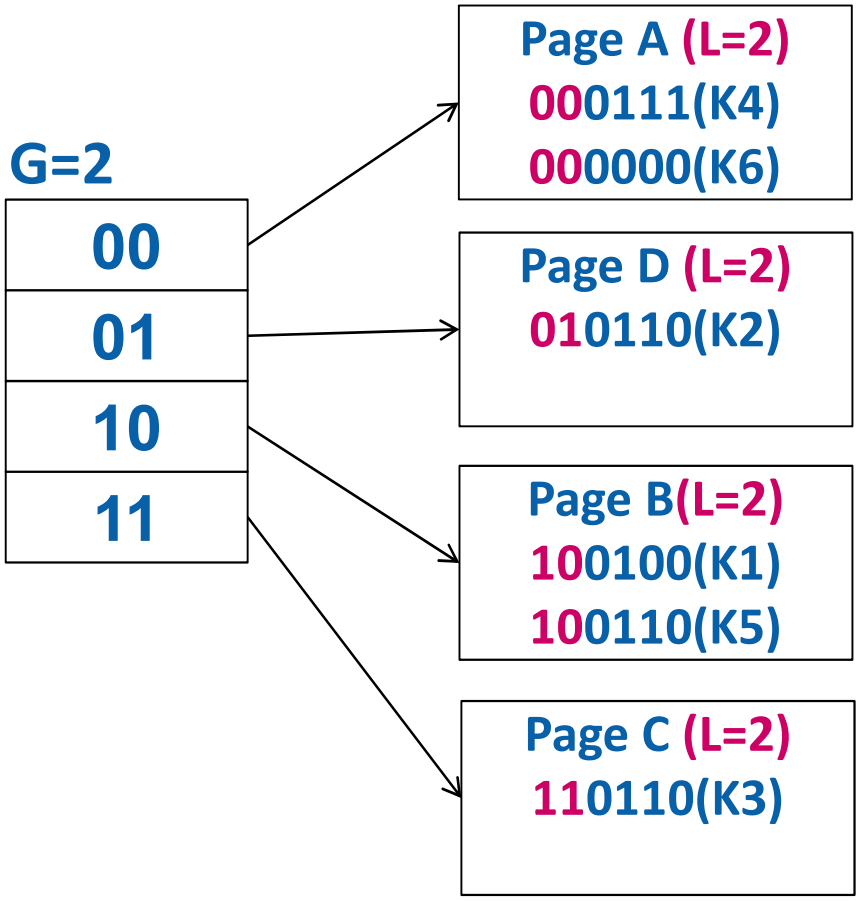
h(k1)=100100
h(k2)=010110
h(k3)=110110
h(k4)=000111
h(k5)=100110
h(k6)=000000



- bucket下标=hash后的整数的前G位
  - G: Global depth
  - Table size = 2^G
- 每个Page中所有的项都有相同的前缀，前L位相同
  - L: Local depth
  - 共有2^{G-L}个桶指向这个Page
- 当L<G时，分裂桶不需要改变Table Size
  - 只需要把相同前缀的项放在同一页，L++

Extendible Hashing

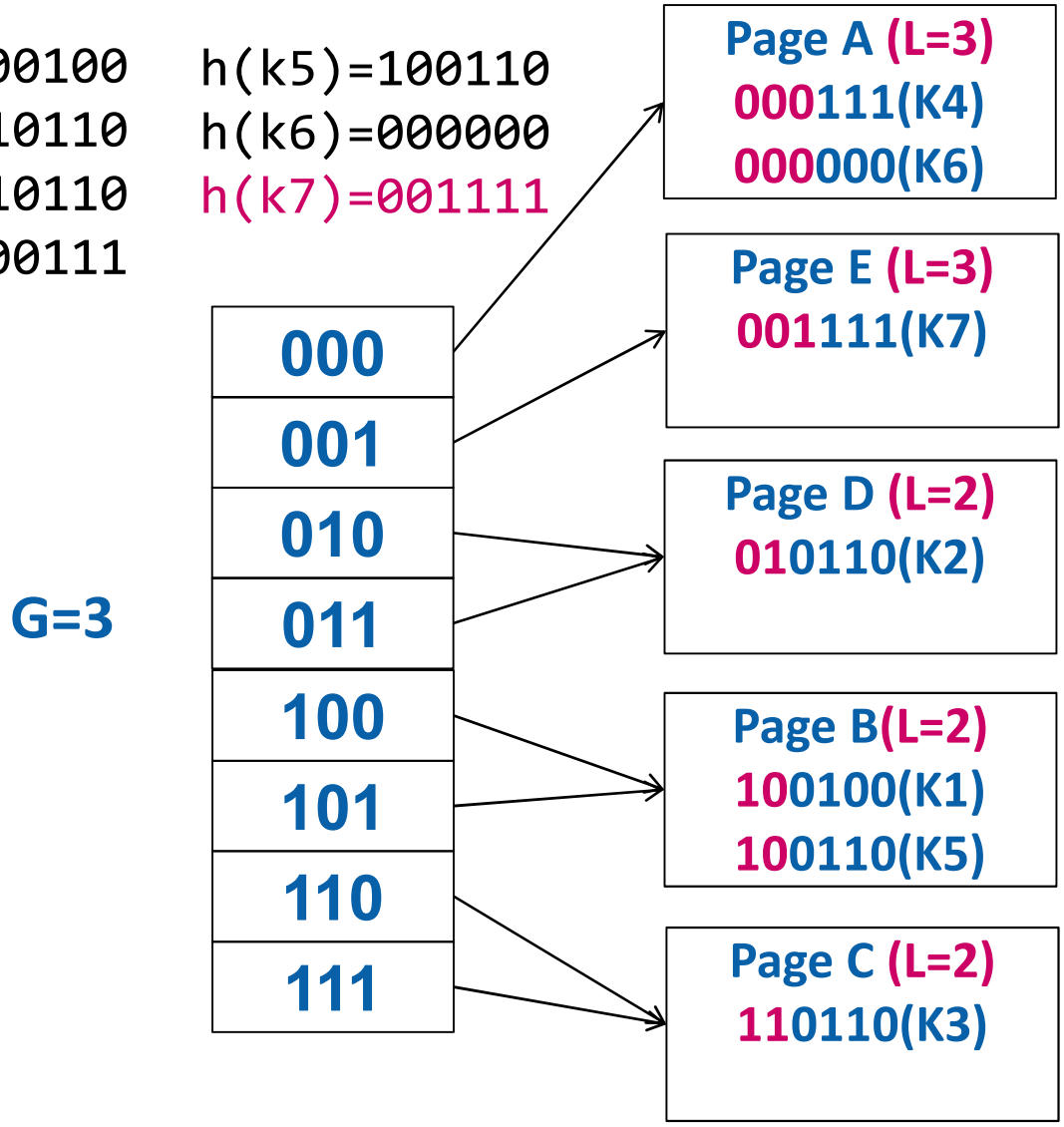
h(k1)=100100
h(k2)=010110
h(k3)=110110
h(k4)=000111
h(k5)=100110
h(k6)=000000
h(k7)=001111



Extendible Hashing

h(k1)=100100
h(k2)=010110
h(k3)=110110
h(k4)=000111

h(k5)=100110
h(k6)=000000
h(k7)=001111

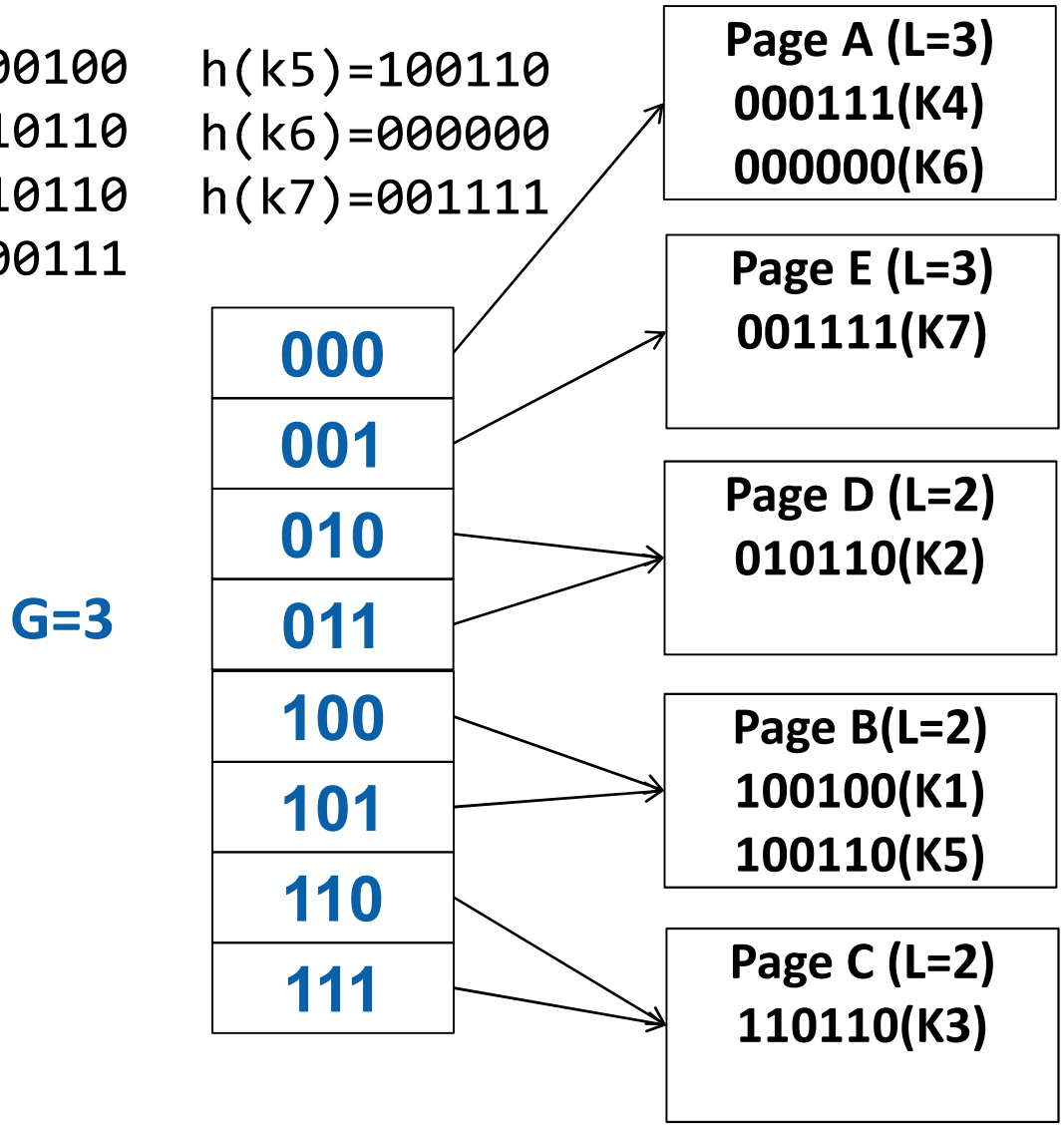


Extendible Hashing： Insert(key)

```
hash= h(key);
page= table[prefix(hash, G)];
if (page不满) {
  把 key , hash 插入Page;
}
else {
  if (page.L == G) {
    G++, 把Table[]变为原来的两倍
    Table’[i<<1|0] = Table’[i<<1|1] = Table[i];
  }
  page.L ++;
  分配一个新的page, 根据prefix(*, page.L)分配各项;
  修改相应的Table[]内容;
}
```

Extendible Hashing

h(k1)=100100
h(k2)=010110
h(k3)=110110
h(k4)=000111
h(k5)=100110
h(k6)=000000
h(k7)=001111

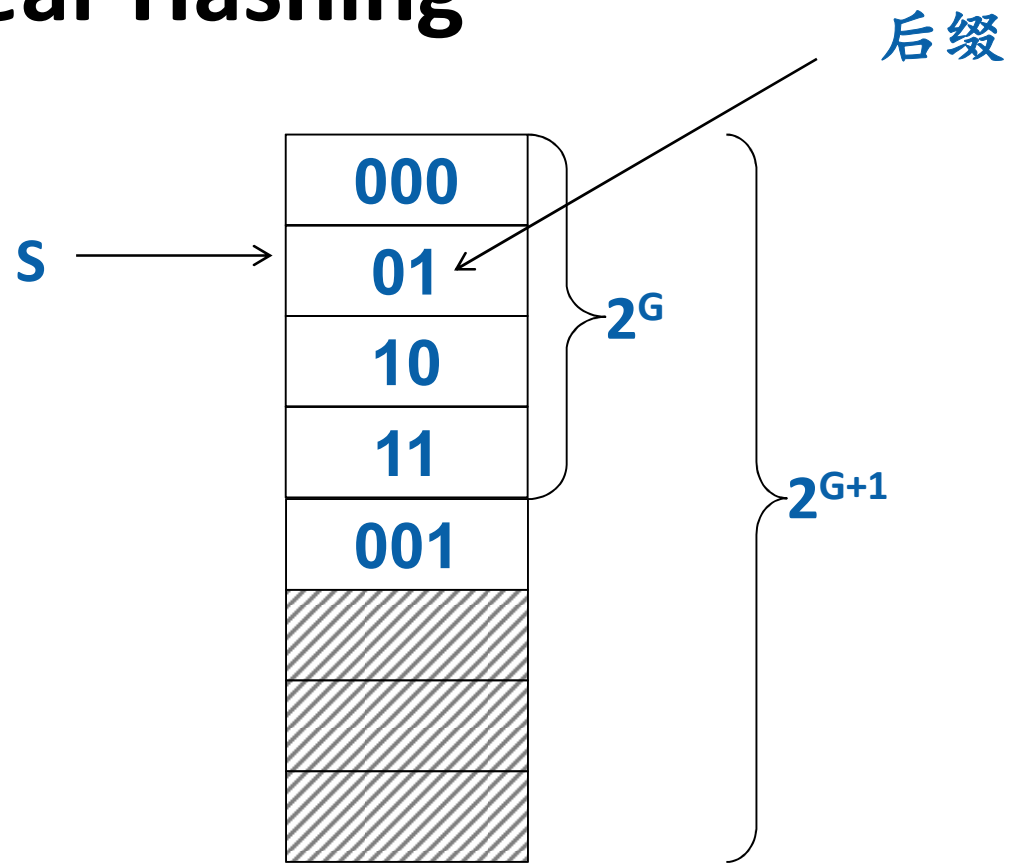


存在的问题：
可能有大量的header，可能造成空间的浪费和性能问题

Linear Hashing

- 希望每次扩展一个header，而不是把Table Size翻番
- 怎么做？

Linear Hashing



- 从2^G扩展到2^{G+1}，不是一步完成，而是渐进地完成
- S指出已经扩展的位置
  - [0, S-1]的bucket已经扩展完毕
  - >=S的bucket还未扩展

什么时候需要扩展？

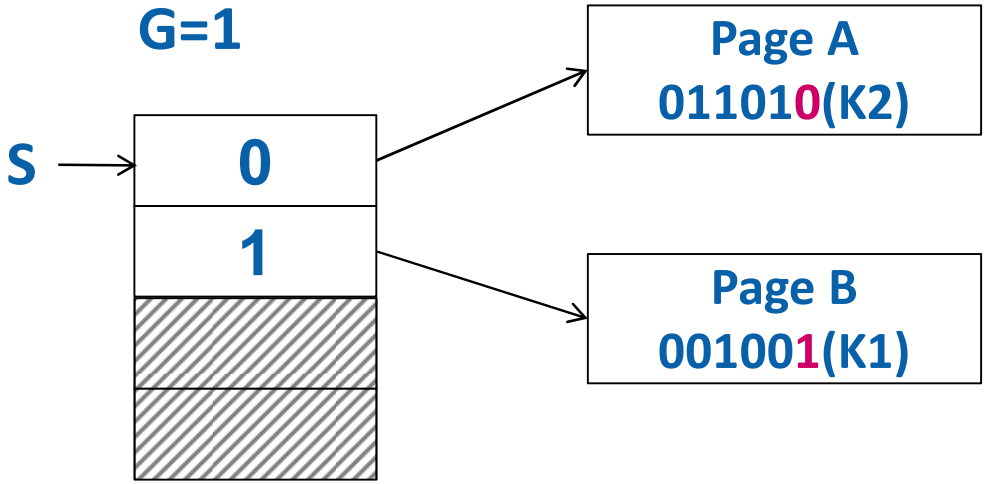
- 定义一个阈值：r
- 当Page数/Bucket数>r时，就触发扩展

Linear Hashing

扩展条件：Page数/Bucket数>1.5

当前：Page数/Bucket数=2/2=1.0 ☺

h(k1)=001001
h(k2)=011010
h(k3)=011011



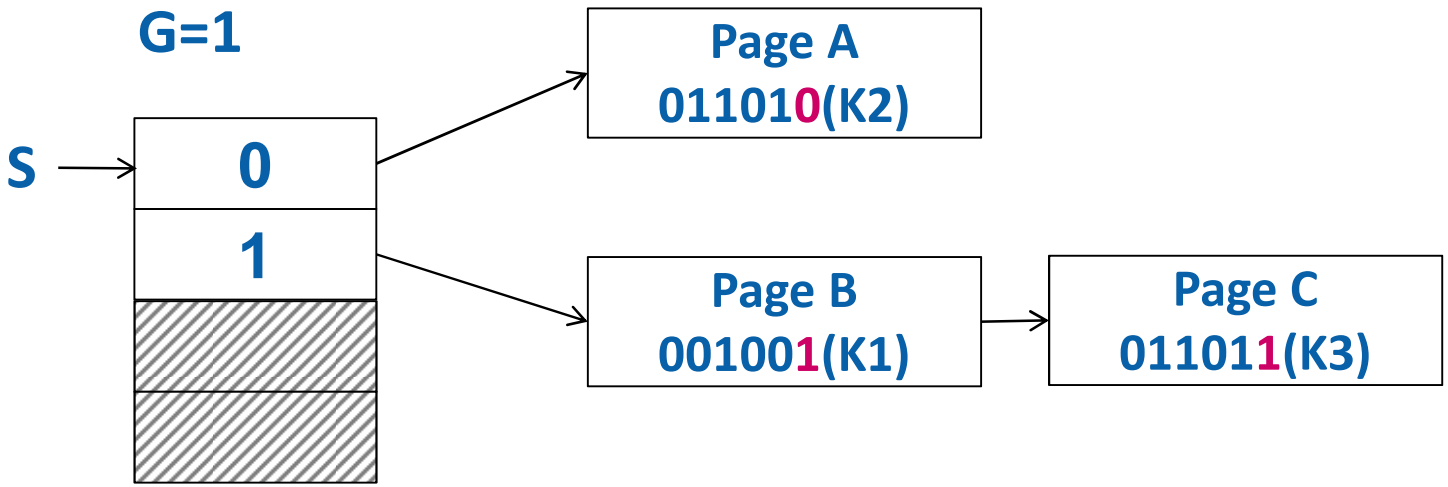
假设每个Page只能放1个项

Linear Hashing

扩展条件：Page数/Bucket数>1.5

当前：Page数/Bucket数=3/2=1.5 ☺

h(k1)=001001
h(k2)=011010
h(k3)=011011
h(k4)=111000



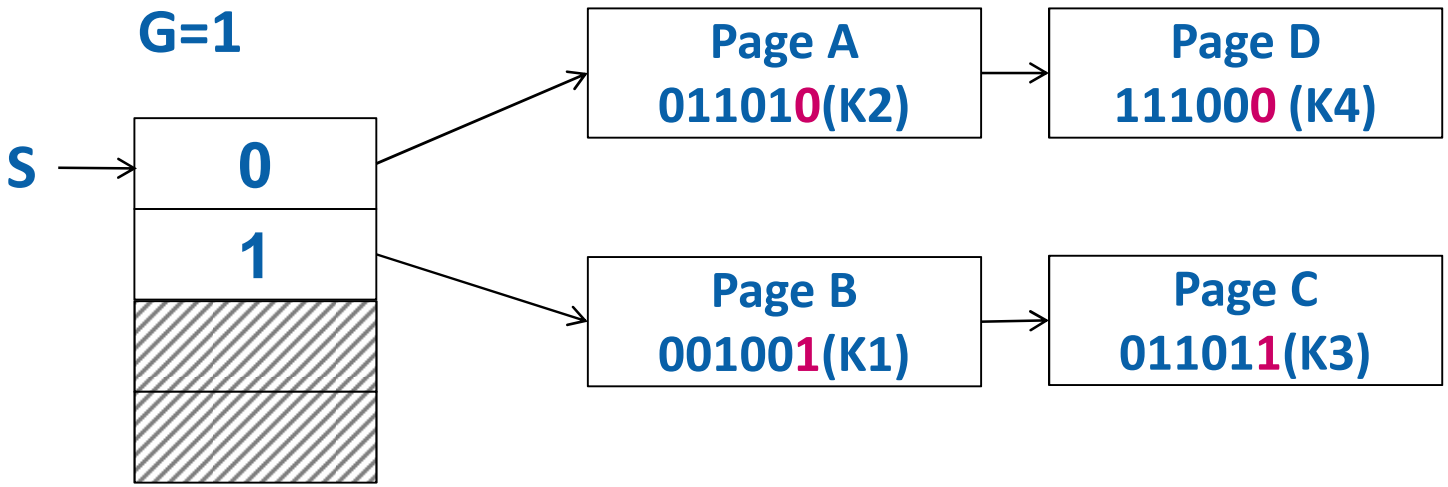
假设每个Page只能放1个项

Linear Hashing

扩展条件：Page数/Bucket数>1.5

当前：Page数/Bucket数=4/2=2.0 ☹

h(k1)=001001
h(k2)=011010
h(k3)=011011
h(k4)=111000



假设每个Page只能放1个项

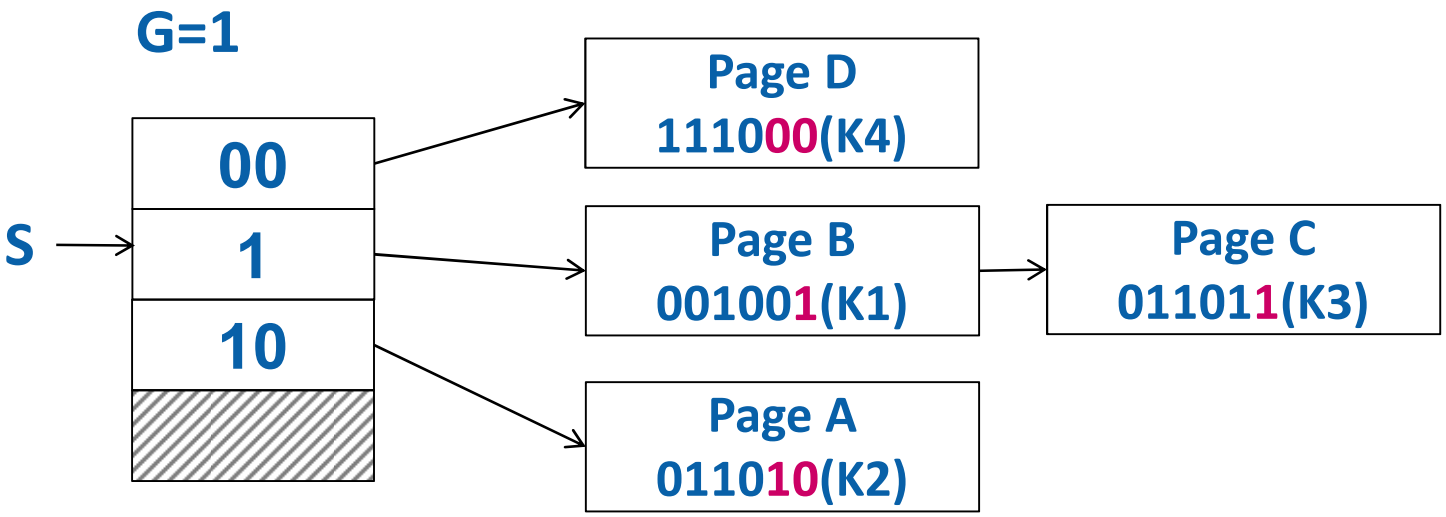
Linear Hashing

扩展条件：Page数/Bucket数>1.5

```
h(k1)=001001
h(k2)=011010
h(k3)=011011
h(k4)=111000
```

进行扩展

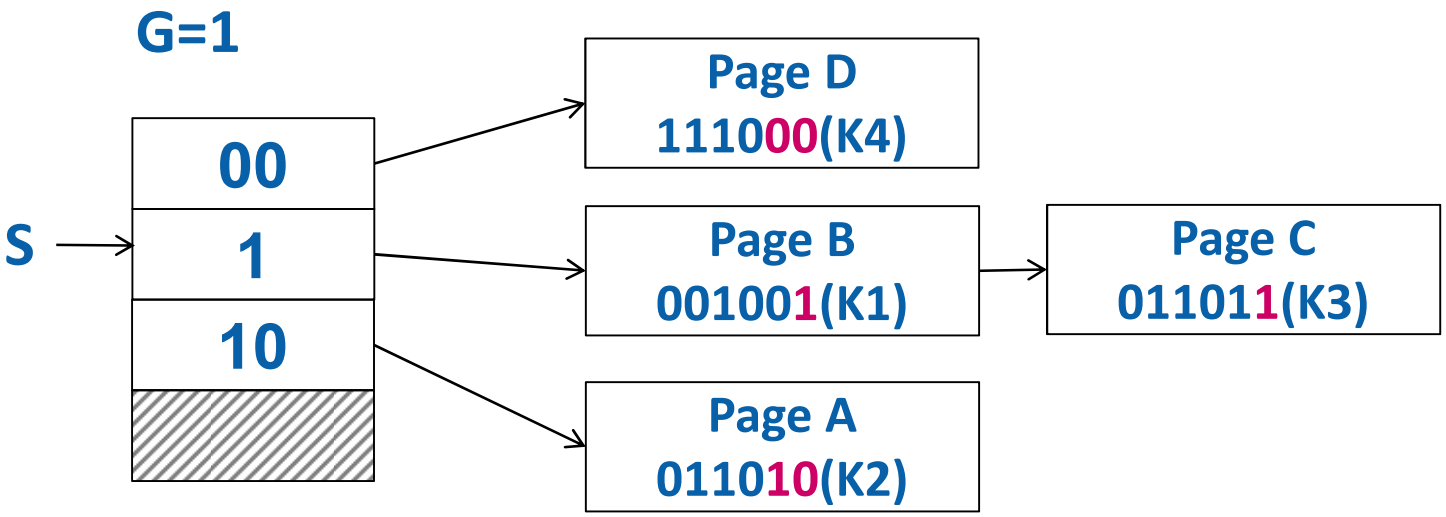
当前：Page数/Bucket数=4/3=1.33 ☺



假设每个Page只能放1个项

Linear Hashing 扩展条件：Page数/Bucket数>1.5

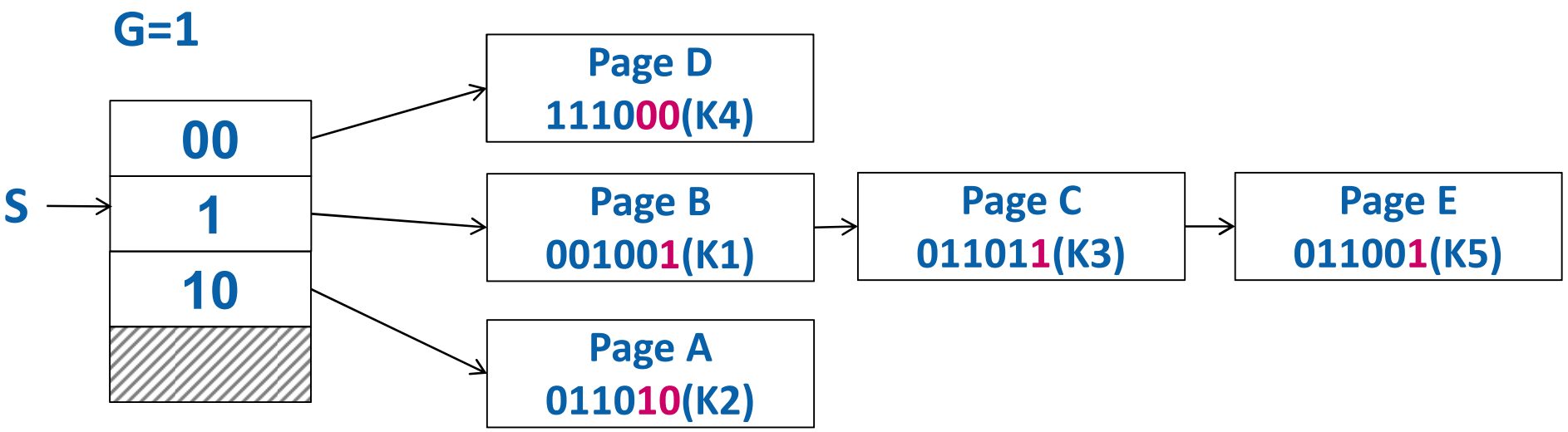
h(k1)=100100 h(k5)=011001
h(k2)=010110
h(k3)=110110
h(k4)=000111



假设每个Page只能放1个项

Linear Hashing 扩展条件：Page数/Bucket数>1.5

h(k1)=100100 h(k5)=011001 当前：Page数/Bucket数=5/3=1.67 ☹
h(k2)=010110
h(k3)=110110
h(k4)=000111



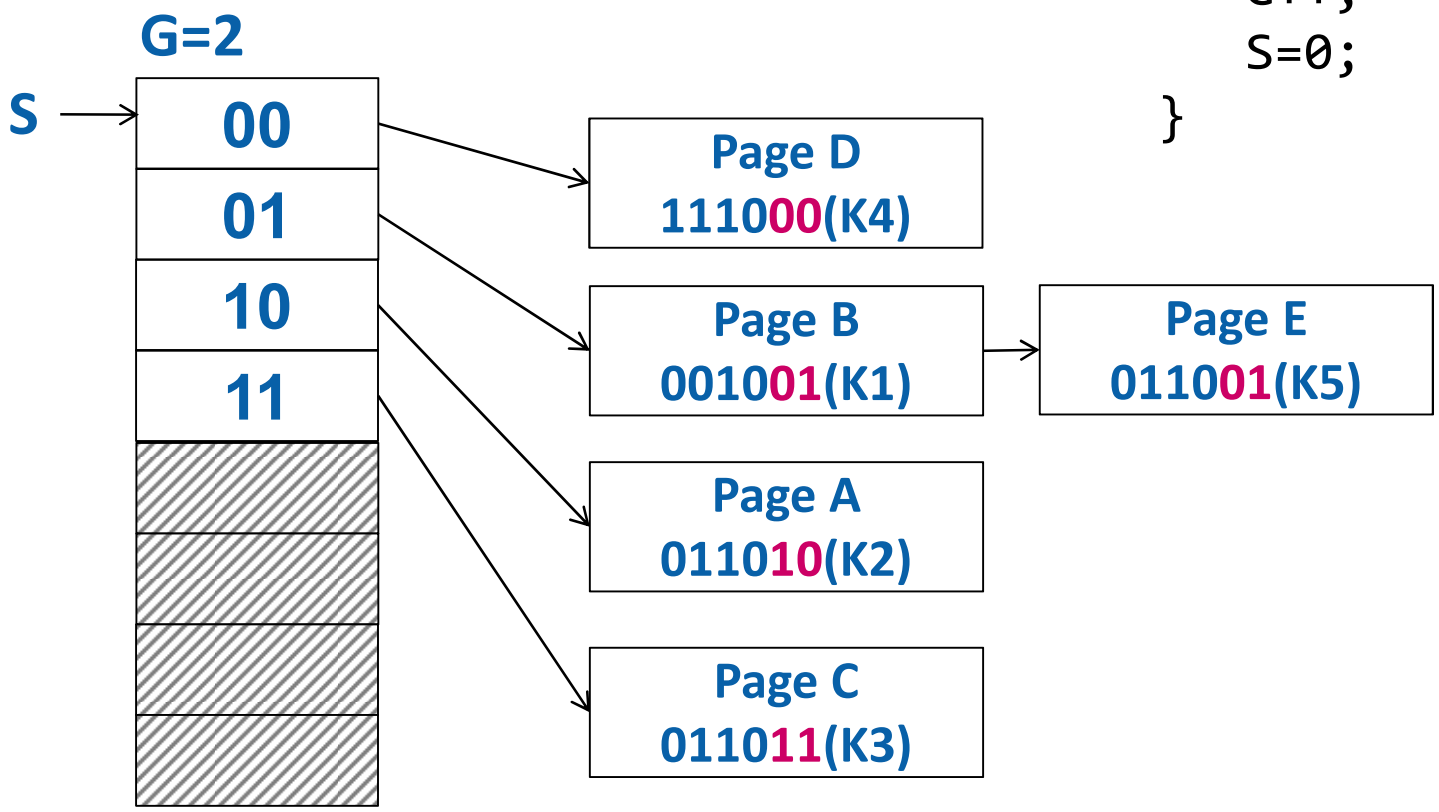
假设每个Page只能放1个项

Linear Hashing 扩展条件：Page数/Bucket数>1.5

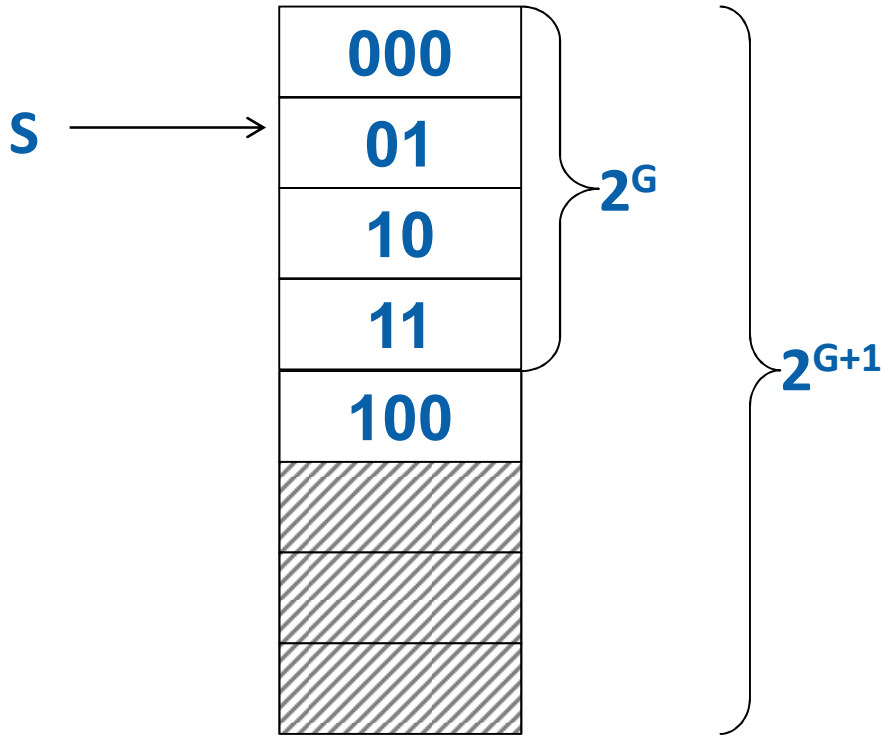
h(k1)=100100 h(k5)=011001
h(k2)=010110
h(k3)=110110
h(k4)=000111

当前：Page数/Bucket数=5/4=1.2 ☺

```
S++;
if (S == 2^G) {
    G++;
    S=0;
}
```



Linear Hashing搜索



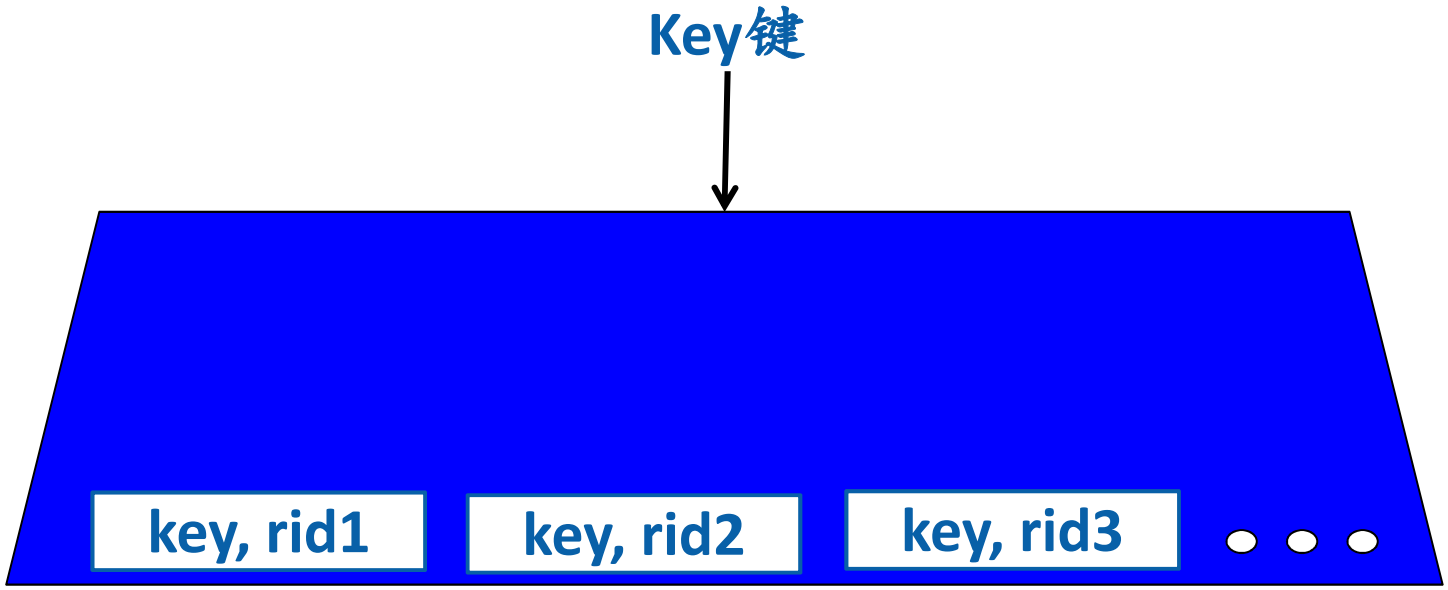

```
idx= hash & ((1<<G)-1);
if (idx < S)
  idx= hash & ((1<<(G+1))-1);
搜索Table[idx]桶
```

Outline

- 索引的概念
- 树结构索引
- 哈希索引
- 位图索引
- 倒排索引

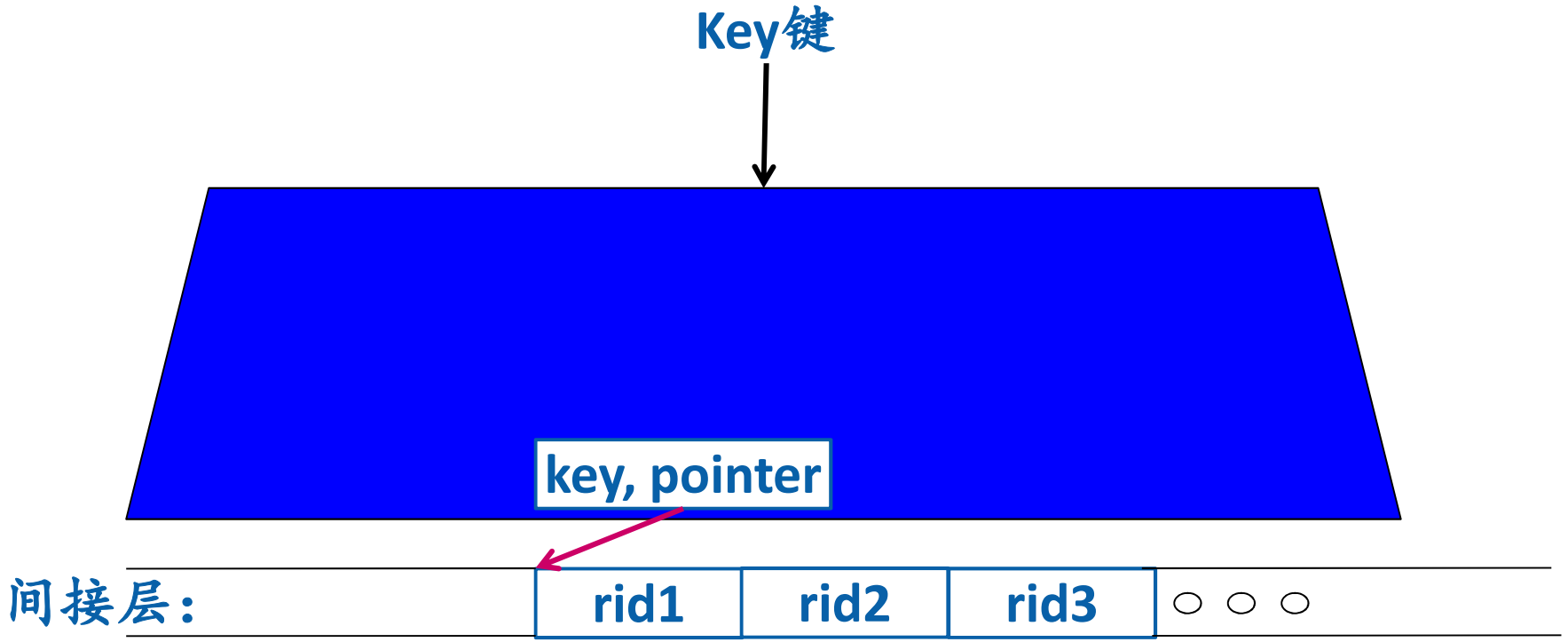
如何处理相同Key的多个记录？

方法1：直接作为不同的项



- 在树结构或哈希表中允许多项具有相同的key
- 问题：当有很多项具有相同的key时，key在索引中存在了多份，占用了额外的空间

方法2：间接层



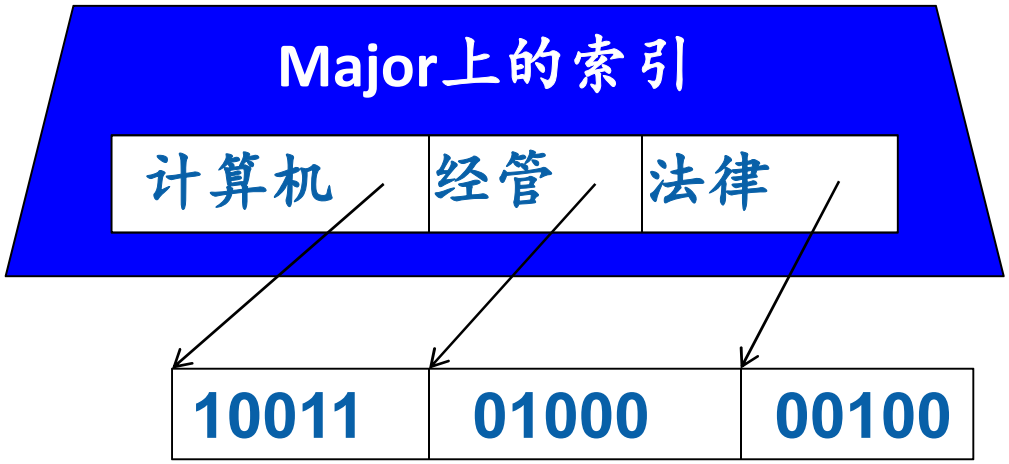
- 在树结构或哈希表中只记录唯一的key
- 多个RecordId记录在间接层中

方法3：Bitmap Index位图索引

- 把间接层记录为多个位图
- 每个key对应一个bitmap
  - bitmap中的每位对应于一条记录
  - 1表示：这条记录在这列上取值是这个key
  - 0表示：这条记录在这列上取值不是这个key

Bitmap Index位图索引

ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85
145678	貂蝉	1996/3/3	女	经管	2014	90
129012	孙权	1994/5/5	男	法律	2012	80
121101	关羽	1994/6/6	男	计算机	2012	90
142233	赵云	1996/7/7	男	计算机	2014	95



位图索引分析

- 位图的大小与key的个数成正比
  - 索引列的取值个数越多，位图索引需要的位图数越多
- 适用场合：当索引列取值个数少的情况
  - 例如：性别、专业、年级等
  - 但是不适合：姓名

位图的优势

- 过滤条件的求值可以通过位运算来实现
- Column = value
  - 从Column对应的索引中得到value所对应的位图
- Column >= value
  - 从Column对应的索引中得到所有大于value所对应的位图
  - 把这些位图OR起来
- 条件1 and 条件2
  - 两个位图的AND
- 条件1 or 条件2
  - 两个位图的OR

位图的压缩

- 当1的个数非常少时，可以进行压缩
- 主要思想
  - 记录两个1之间的距离：表示A个0后跟一个1
  - 采用某种编码，用变长整数表达距离A
    - 关键问题是如何表示长度？
- 在使用时，解压进行处理
  - 位运算是逐位顺序进行的
  - 可以解压一段计算一段

变长整数表示：Unary Code

- 整数A表达为A-1个1和一个0

A	Unary Code
1	0
2	10
3	110
4	1110
5	11110
10	1111111110
50	11111111111111111111111111111111111111111111111111111110

适合于小数字出现频繁的情形

（根据香浓理论，码字长度应该~$\log\frac{1}{P}$）

变长整数表示：Gamma Code

- 假设A的二进制表示中最高为1的位是第k位
- 那么

$$\underbrace{111\ldots10}_{k\text{个}1}A_{k-1}\ldots A_1A_0$$

- 例如：A=101101

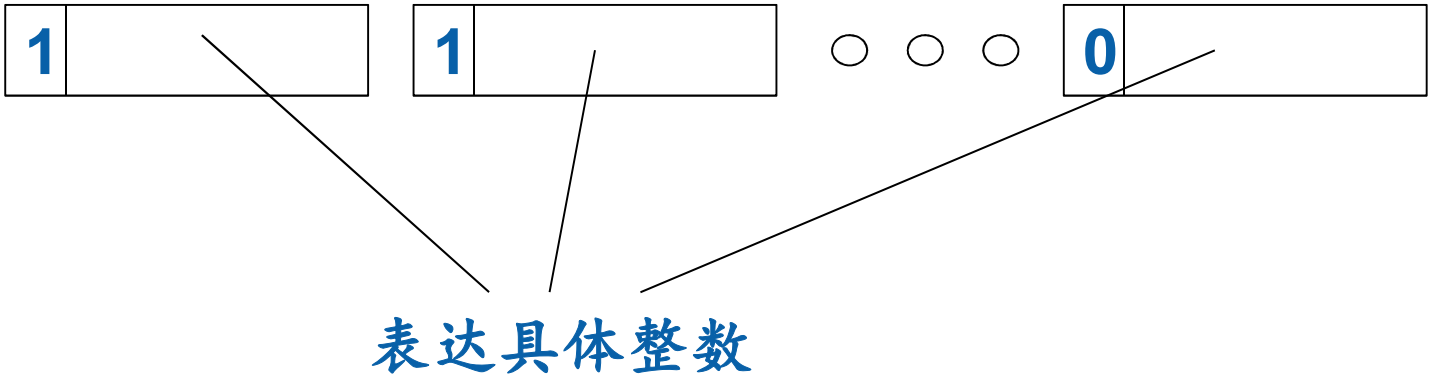
111110 01101

变长整数表示：Gamma Code

A	Gamma Code
1	0
2 $(10)_b$	10 0
3 $(11)_b$	11 1
4 $(100)_b$	110 00
5 $(101)_b$	110 01
10 $(1010)_b$	1110 010
50 $(110010)_b$	111110 10010

变长整数表示：字节对齐码

- Byte-aligned code
- 每个字节用低7位表示整数，最高1位用于标识长度



变长整数表示：字节对齐码

- 考虑如下的可能
  - A=100，$(1100100)_b$，只需要一个字节，可以表示为

    01100100

  - A=1000，$(111\ 1101000)_b$，需要两个字节，可以表示为

    11101000 00000111

- 可以改进？
  - 在使用中最高7位不可能为0（为什么？）
  - 所以，最高7位可以表达1~128，而不是0~127
  - 把最高7位减1后编码

变长整数表示：字节对齐码

- 实际的表达
  - A=100，$(1100100)_b$，只需要一个字节，可以表示为

    01100011　　注：1100100减1
  - A=1000，$(111\ 1101000)_b$，需要两个字节，可以表示为

    11101000　00000110　　注：0000111减1

- 把最高7位减1后编码

编码方式比较

- 我们只介绍了几种简单的编码
  - 有更多的编码方式，例如Golomb code等
- 字节的编码方式便于处理，运算效率高
- 位的编码方式可以获得更高的压缩比
  - 如果追求极致的压缩比，需要具体分析A的分布
  - 根据分布，选择恰当的编码

Outline

- 索引的概念
- 树结构索引
- 哈希索引
- 位图索引
- 倒排索引

Inverted Index（倒排索引）

- 主要用于对文本进行索引
  - 有大量的文档（Document）
  - 每个文档有很多单词（Term）组成

Doc1: {t1, t3, t4, …, t1000}

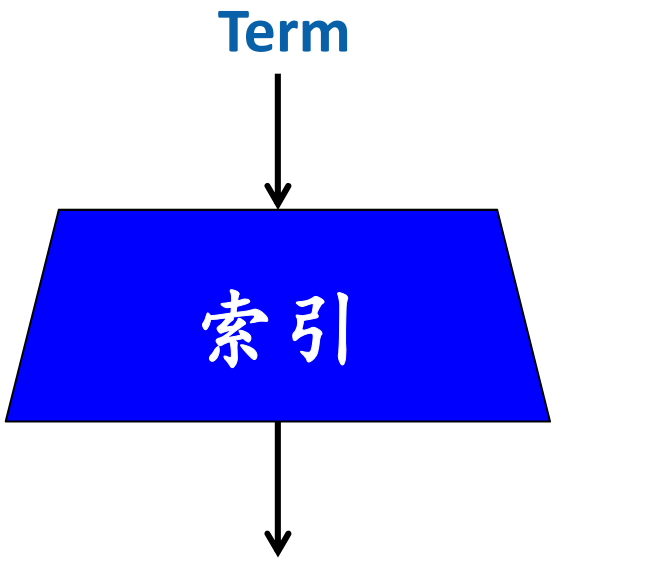
Doc2: {t2, t3, t4, …, t9999}

Doc3: {t5, t3, t9, …, t8888}

……

- 目标
  - 给定关键字term
  - 找到相关的文档，并对结果排序

倒排索引

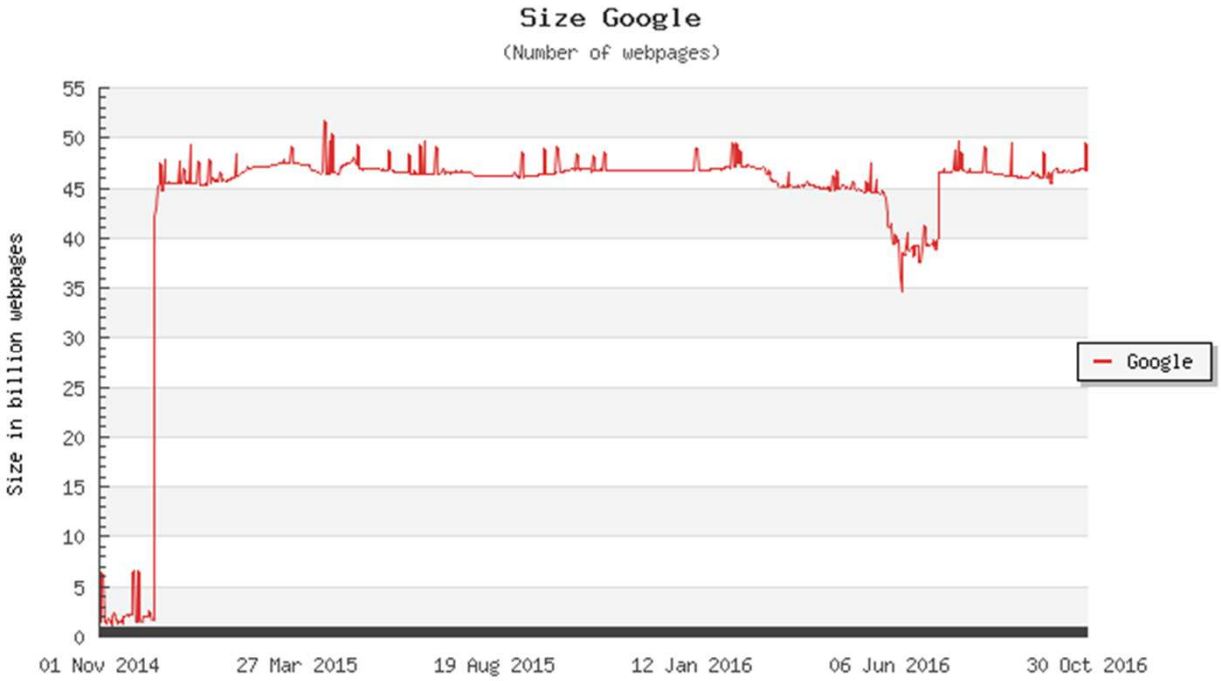


Inverted Index（倒排索引）

- 对于关系型数据库系统
- 可以对文本列产生倒排索引
- 从而加速like等操作

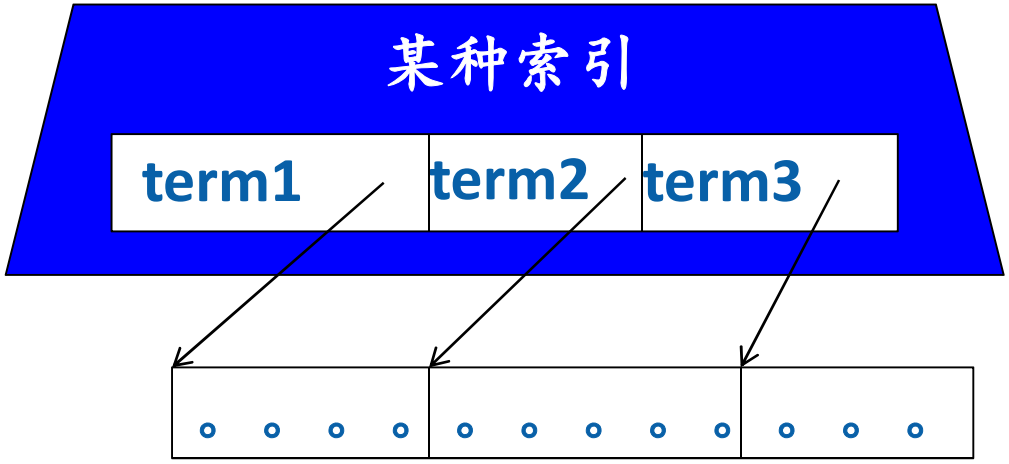
让我们来分析一下需求

- 可能的term(单词，名词等)有多少个?
  - 比较少
  - 常见的词汇：~几万，~几十万?
- 文档有多少个
  - 搜索引擎网页数量
  - 大概450亿
  - 一个term可能出现于大量的文档中



数据来源：http://www.worldwidewebsize.com/

在term上很容易建立索引



- 可以是B+-tree, hash table等
- 主要的难点：

  如何记录每个term对应的大量的DocID和TF？

Inverted List（倒排表）

Term->($DocID_1$,TF_1), ($DocID_2$,TF_2), …, ($DocID_n$,TF_n)

- 这个结构被称为Inverted List，连续存储于外设
  - 可以用顺序读访问
- DocID从小到大排序，求相邻两项DocID的差，只记录这个差值
- 设计的重要问题是如何对DocID和TF进行压缩
  - 采用恰当的变长整数编码
  - 如前所述

小结

- 索引的概念
- 树结构索引
  - 从顺序文件到静态索引
  - B+树
  - 主索引（聚簇索引）和辅助索引（二级索引）
- 哈希索引
  - Hash function
  - Chained hashing
  - Extendible hashing
  - Linear hashing
- 位图索引
- 倒排索引