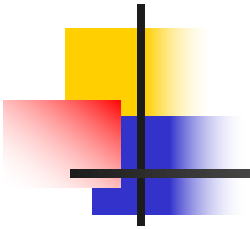
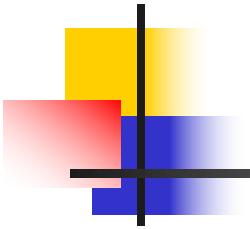


第五章 数据库的存储结构

5.1 数据库存储介质的特点

数据库是大量、持久数据的集合，在现阶段用内存作为数据库的存储介质是不合适的。

- 采用多级存储器，用的最多的辅存是磁盘。
- 光盘由于速度和价格上的原因，近期无法取代硬盘。
- 磁带是顺序存取存储器，通常用作后备存储器。

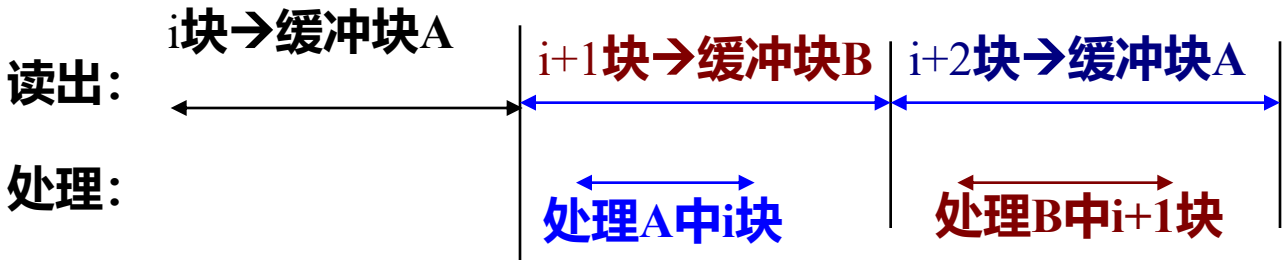


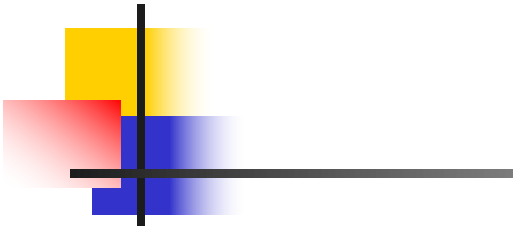
- 活动头磁盘的存取时间由三部分组成：寻道时间、等待时间以及传输时间。
- 磁盘上的数据划分为大小相等的物理块。磁盘与内存间的数据交换以物理块为单位。

以物理块为交换单位的优点：
1).减少I/O的次数，从而减少寻道和等待的时间。
2).减少间隙的数目，提高磁盘空间利用率。
物理快的大小由OS决定。

- 一般，在磁盘和内存之间设立缓冲区以解决二者的速度不匹配问题。

  由于有多个缓冲块可供申请使用，磁盘的读写操作和读写数据的处理可以重叠进行。



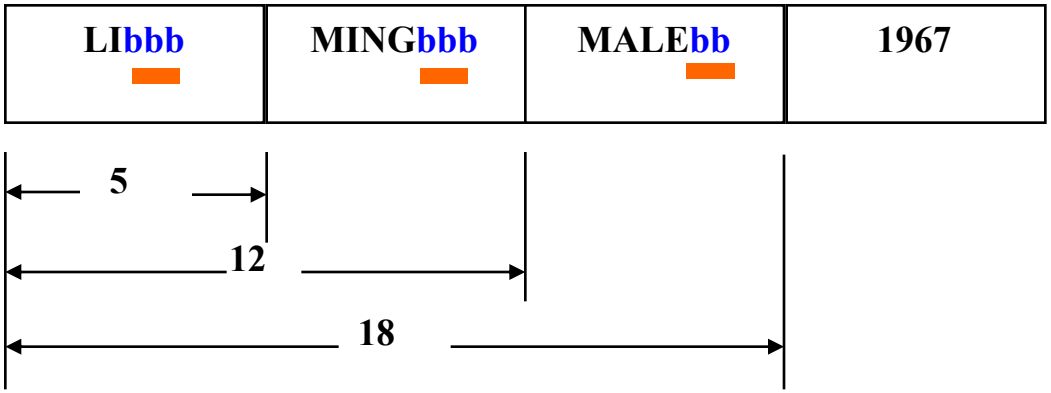


- OS与DBMS都有各自的缓冲区。
- 不少DBMS采用延迟写与提前读技术，减少I/O，改善性能。

5.2 记录的存储结构

- 记录是目前商用数据库的基本数据单元，有定长和变长之分。
- 记录的存储结构

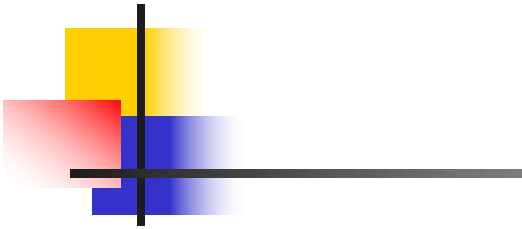
1.定位法——每个字段按其最大可能长度分配定长的位置



2.相对法——每个字段没有固定的长度，而是用特殊的字符分隔开

LI?	MING?	MALE?	1967#

问题：字段中也需要用到这些分隔符时，如何进行表示?



3.计数法——每个字段的开始加上表示该字段长度的字段

02LI04MING04MALE041967

问题：计数法对字段的实际长度有什么要求?

5.2.2 记录在物理块上的分配

- 磁盘上，记录必须分配到物理块中。
  - 记录跨快存储（spanned）
  - 记录不垮块存储（unspanned）

设 B 为物理块的有效空间大小， R 为固定长记录的大小，若 B ＞ R，则每个物理块可容纳的记录数为：

$$p=[B/R]$$

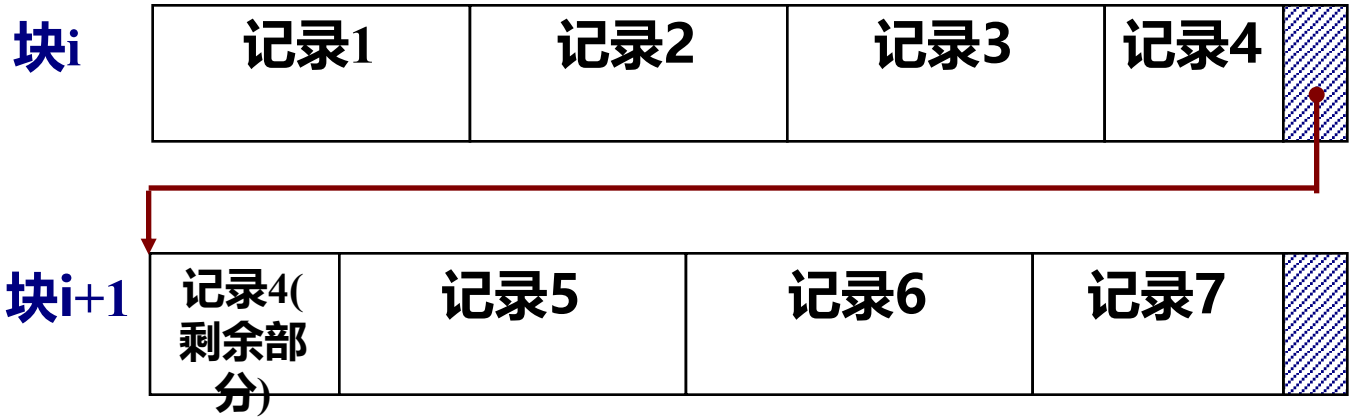
p称为块因子（Blocking Factor）。

记录一般不会刚好填满物理块，会留下不用的零头空间：

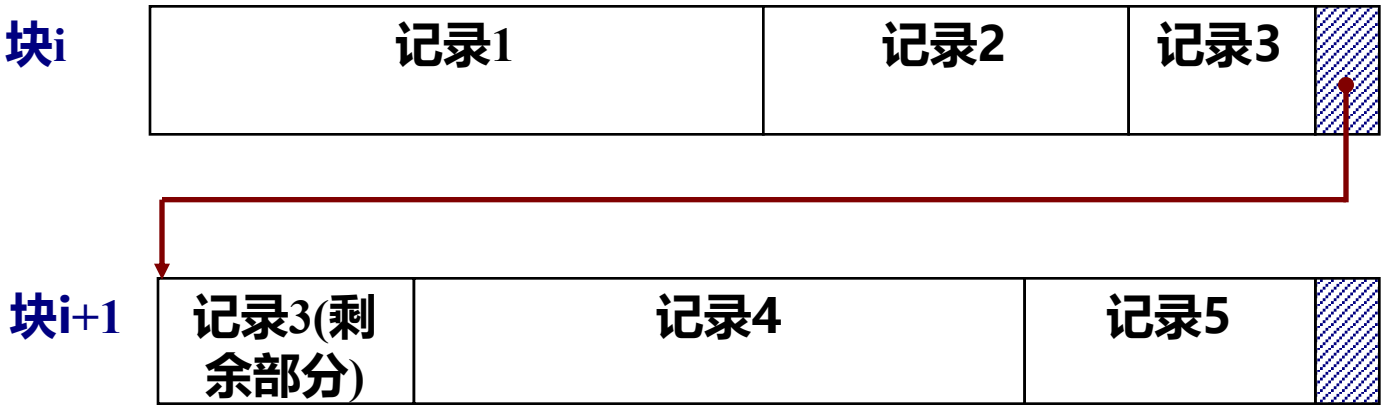
$$B—p\times R<R$$

为了利用这部分空间，可以利用记录的跨块存储组织(spanned organization)。

定长记录（跨块）



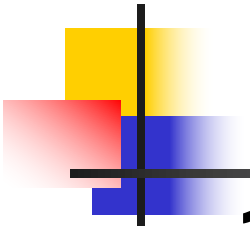
变长记录（跨块）



5.2.3 物理块在磁盘上的分配

早期的DBMS中，通常由操作系统分配数据库所需的物理块，逻辑上相邻的数据可能被分散到磁盘的不同区域。使得访问数据时，性能下降。

现代DBMS中，都改由DBMS初始化时向操作系统一次性的申请所需的存储空间。

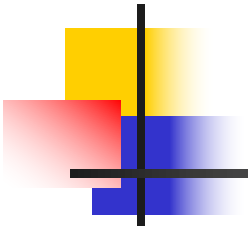


1、连续分配法（contiguous allocation）

将一个文件的块分配在磁盘的连续空间上，块的次序就是其存储的次序，有利于顺序存取多块文件，不利于文件的扩充。

2、链接分配法（linked allocation）

物理块未必分配在磁盘的连续存储空间上，各物理块用指针链接，有利于文件的扩展，但效率较差。

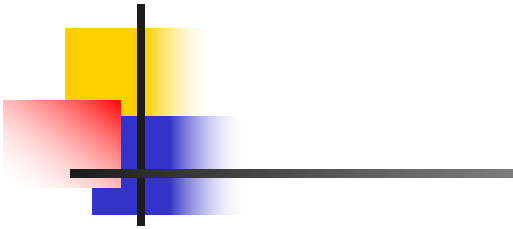


3、簇集分配法（clustered allocation）

上述两种方法的结合。

4、索引分配法（indexed allocation）

每个文件有一个逻辑块号与其物理块地址对照的索引。



- **数据压缩技术**

1.消零或空格符法（null suppression）
例如，bbbbb可以用#5表示；
000000可以用@6表示等。

2.串型代替法（pattern substitution）

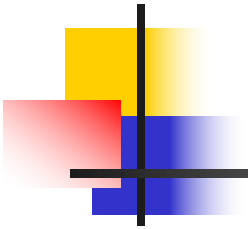
对反复出现的字符串，可以用一个省略符代替。

例如，串型表如右：

IBM PC/XT	@
0000	#

原始数据
IBM PC/XT 00001
IBM PC/XT 00002

压缩数据
@#1
@#2



3.索引法（indexing）

串行代替法的变种，对重复出现的串行，单独存储，在用到这些串行的地方，用指针引用它。

索引法示例：

原始数据

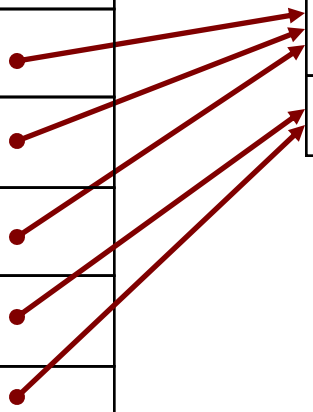
SHOP#	CITY
0001	Nanjing
0002	Nanjing
0003	Nanjing
0004	Shanghai
0005	Shanghai

压缩数据

SHOP#	CITY
0001	
0002	
0003	
0004	
0005	

CITY表

Beijing
Nanjing
Shanghai



问题：索引法对串型的长度有什么要求?

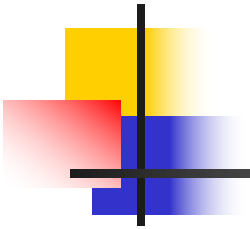
5.3 文件结构和存取路径

5.3.1 访问文件的方式

传统的数据模型都以记录为基础，记录的集合构成文件。文件须按一定的结构组织和存储记录，按一定的存取路径访问有关记录。

对数据库的操作最终要落实到对文件的操作。

文件结构及其所提供的存储路径直接影响数据访问的速度，通常针对不同的数据访问采用不同的文件结构。



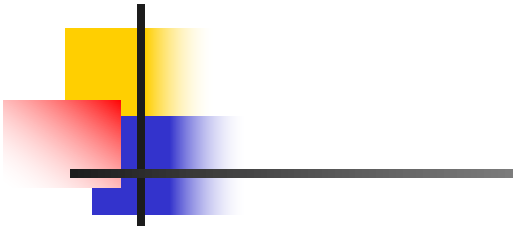
文件访问方式按设计文件结构的观点分5类

1.查询文件的全部或相当多的记录（≥15%）
2.查询某一特定记录
3.查询某些记录
4.范围查询
5.记录的更新

5.3.2 数据库对文件的要求

一些DBMS就以OS的的文件管理系统作为其物理层的基础，更多的DBMS不用OS的文件管理系统，而是独立设计其存储结构。原因如下：

- 传统文件系统不能提供实现DBMS功能所需的附加信息。DBMS为了实现其功能，须在文件目录、文件描述块、物理块等部分附加一些信息。
- 传统文件系统主要面向批处理，数据库系统要求即时访问、动态修改。这就要求文件的结构能适应数据的动态变化，提供快速访问路径。



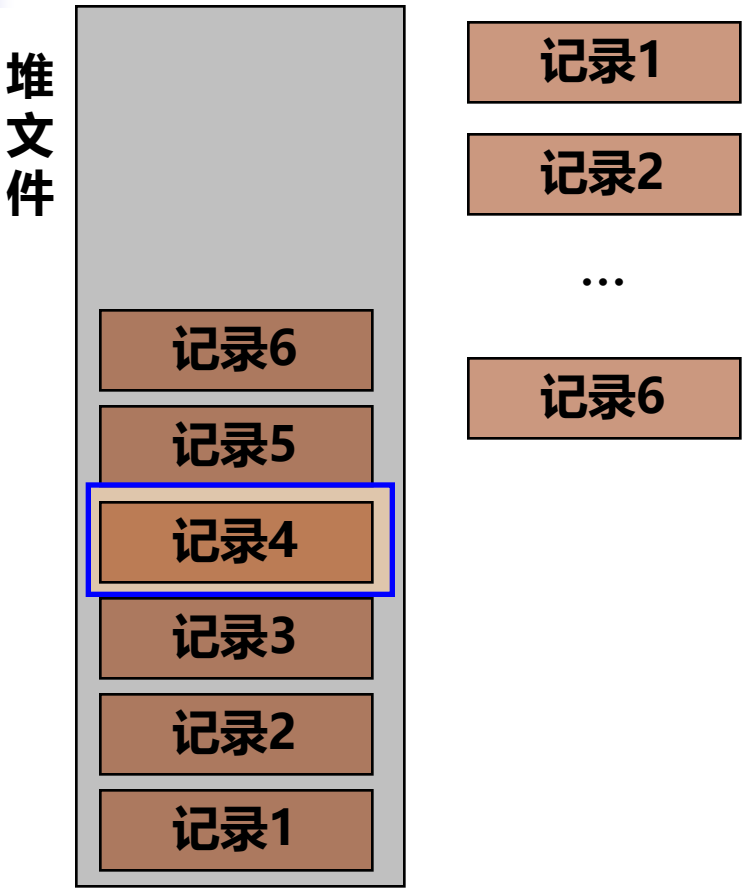
- 传统文件系统服务对象特殊，用途单一，共享度低；数据库中的文件供所有用户共享，有些用途甚至是不可知的。
- 减少DBMS对OS的依赖，提高DBMS的可移植性。
- 传统文件系统一旦建立以后，数据量比较稳定；数据库中文件的数据量变化较大。

5.3.3 文件的基本类型

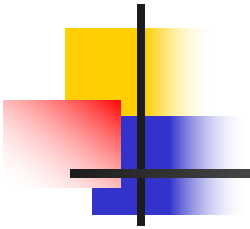
不同类型的访问各有其使用的文件结构和存取路径。DBMS通常提供多种文件结构，供数据库设计者选用。

1.堆文件（heap file)
2.直接文件（direct file)
3.索引文件（indexed file)

1.堆文件



- 最简单、最早使用的文件结构。记录按其插入的先后次序存放（所有记录在物理上不一定相邻接）
- 堆文件插入容易、查找不方便（所提供的唯一存取路径就是顺序搜索）



- 适于访问全部或相当多的记录，对于其它类访问效率较低。（设文件记录数为N，查找某一记录的平均访问次数为N+1/2）
- 若堆文件的记录按检索属性排序，可用二分查找法。（但要以排序为代价）

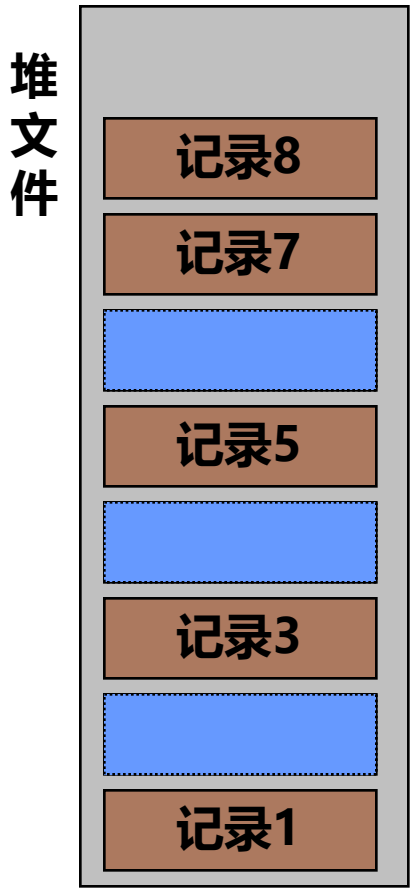
- 删除记录较麻烦

假设，要删除右图堆文件中的第2，4，6条数据记录。

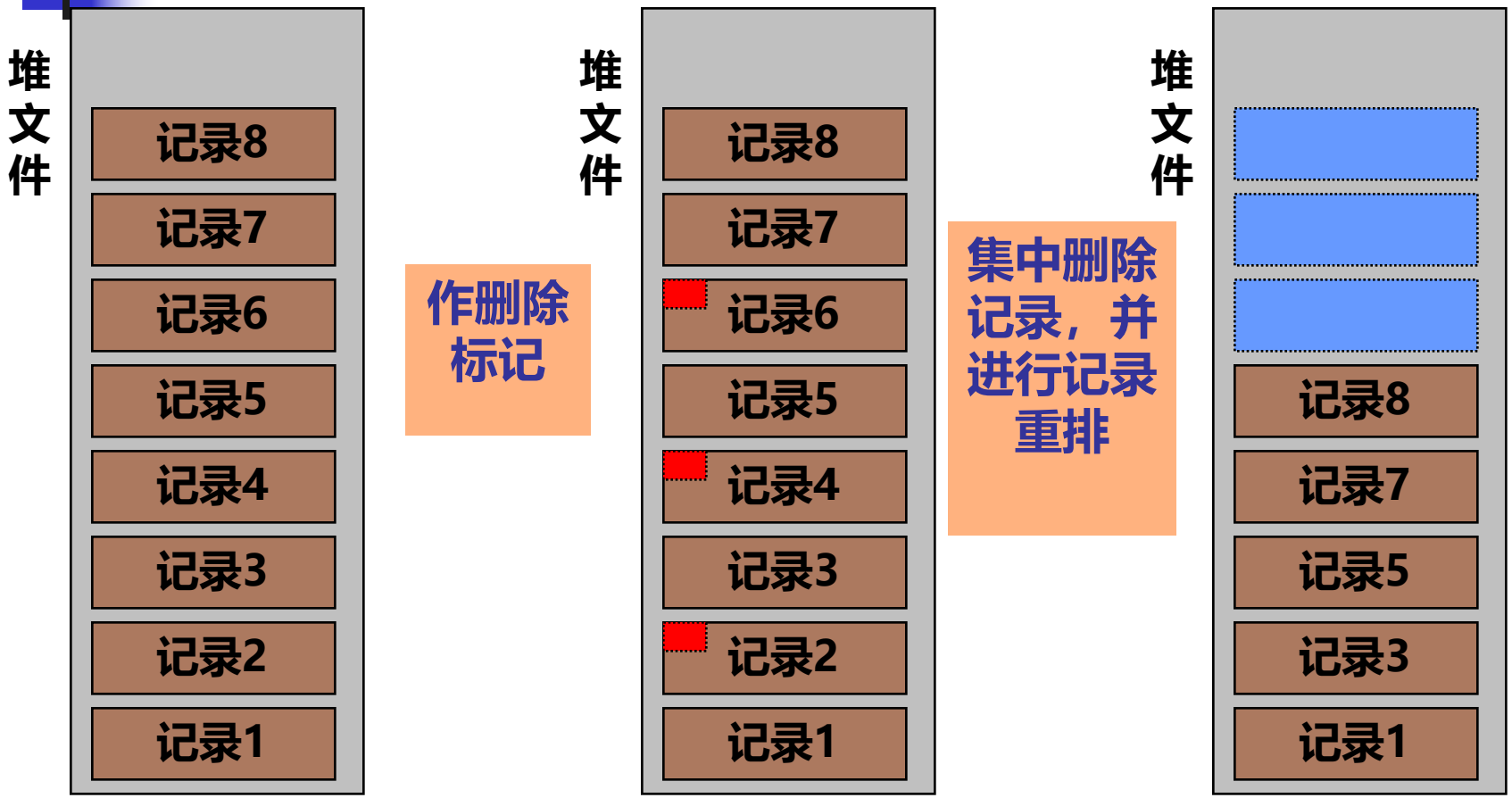
直接删除这三条记录的情况如右图所示。

带来什么问题?

空间回收问题，后续记录需搬移，影响对文件的操作效率。



解决方法

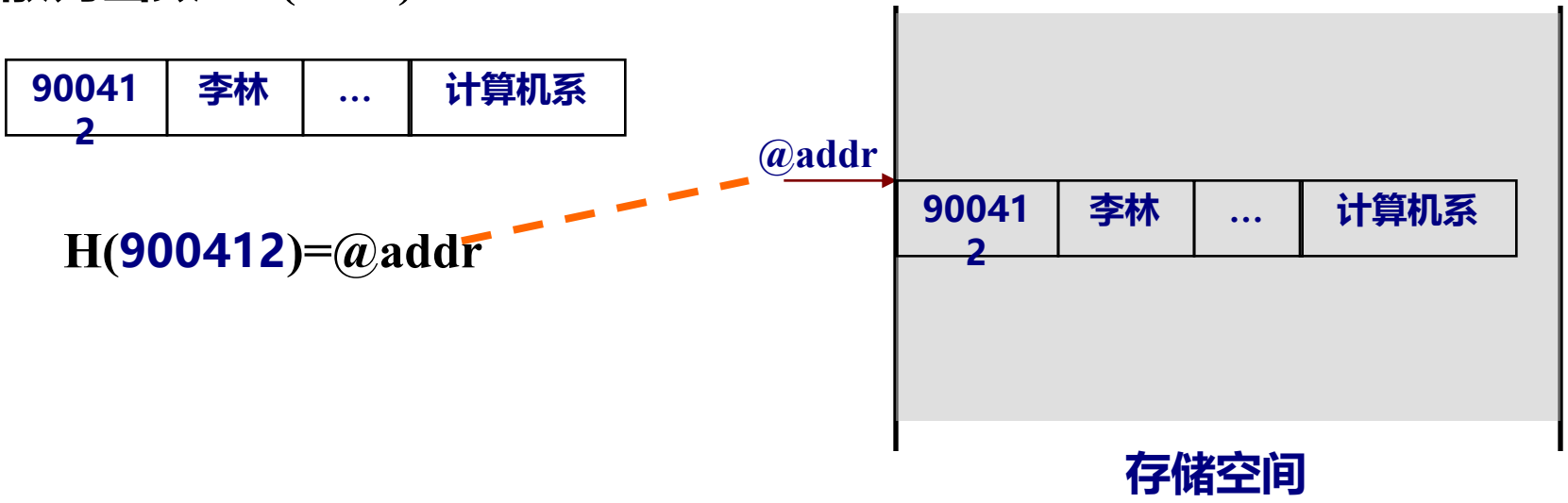

堆文件
记录8
记录7
记录6
记录5
记录4
记录3
记录2
记录1
作删除标记
堆文件
记录8
记录7
记录6
记录5
记录4
记录3
记录2
记录1
集中删除记录，并进行记录重排
堆文件
记录8
记录7
记录5
记录3
记录1

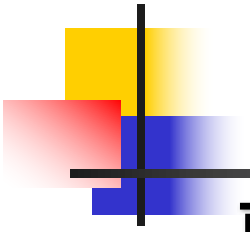
2.直接文件

直接文件中，将记录的某一属性用散列函数直接映射成记录的地址，被散列的属性称为散列键。

Student(SNO,SNAME,SEX,BIRTHDATE, DEPT)

散列函数　H(SNO)=Address





直接文件存在的问题：

- 键所映射的地址范围固定（地址范围设的太大或太小都不好,为什么？）。
- 地址重叠问题（处理地址重叠增加了开销）。
- 直接文件只对散列键的访问有效。
- 不便于处理变长记录。
- 对于通用的DBMS很难找到通用的散列函数。

上述原因导致直接文件目前在数据库系统中使用不多。

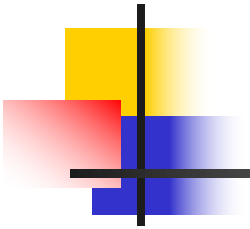
3.索引文件

索引相当于一个映射机构，把关系中相应记录的某个（些）属性的键值转换为该记录的存储地址（或地址集）。

存储空间

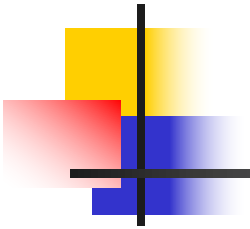
SNO	Address
900412	@addr2
900417	@addr1
900418	@addr3
…	…

学生表的索引文件



- 索引与散列的区别——索引文件有记录才占用存储空间，使用散列空文件也占用全部文件空间。
- 索引本省占用空间，但索引一般较记录小得多

针对非散列键和非索引属性的访问，都不能有效发挥直接文件或索引文件的优势。散列或索引失效时，两者谁的访问代价更大?



1.主索引——以主键为索引键(primary index)。
2.次索引——以非主键为索引键(secendary index)，建立次索引可以提高查询的效率，但增加了索引维护的开销。
3.倒排文件——主索引+次索引的极端情况（文件的所有属性上都建立索引），有利于多属性值的查找，但数据更新时开销很大。

为了便于检索，索引项总是按索引键排序。

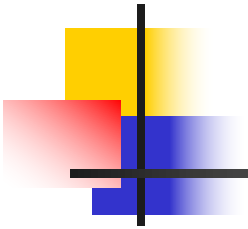
文件中的记录按索引键排序吗?

注意：索引项≠记录,并不是文件中的记录按索引键排序

受按门牌号找住户的启示（住户在“物理”上按门牌号码排序），提出非稠密索引。

- **非稠密索引与稠密索引**

1.不为每个键值设立索引项的索引——非稠密索引
2.可以节省索引的存储空间，代价是文件要按索引键排序
3.对一个文件，只能为一个索引键（一般为主键）建立非稠密索引 ***（为什么？）***

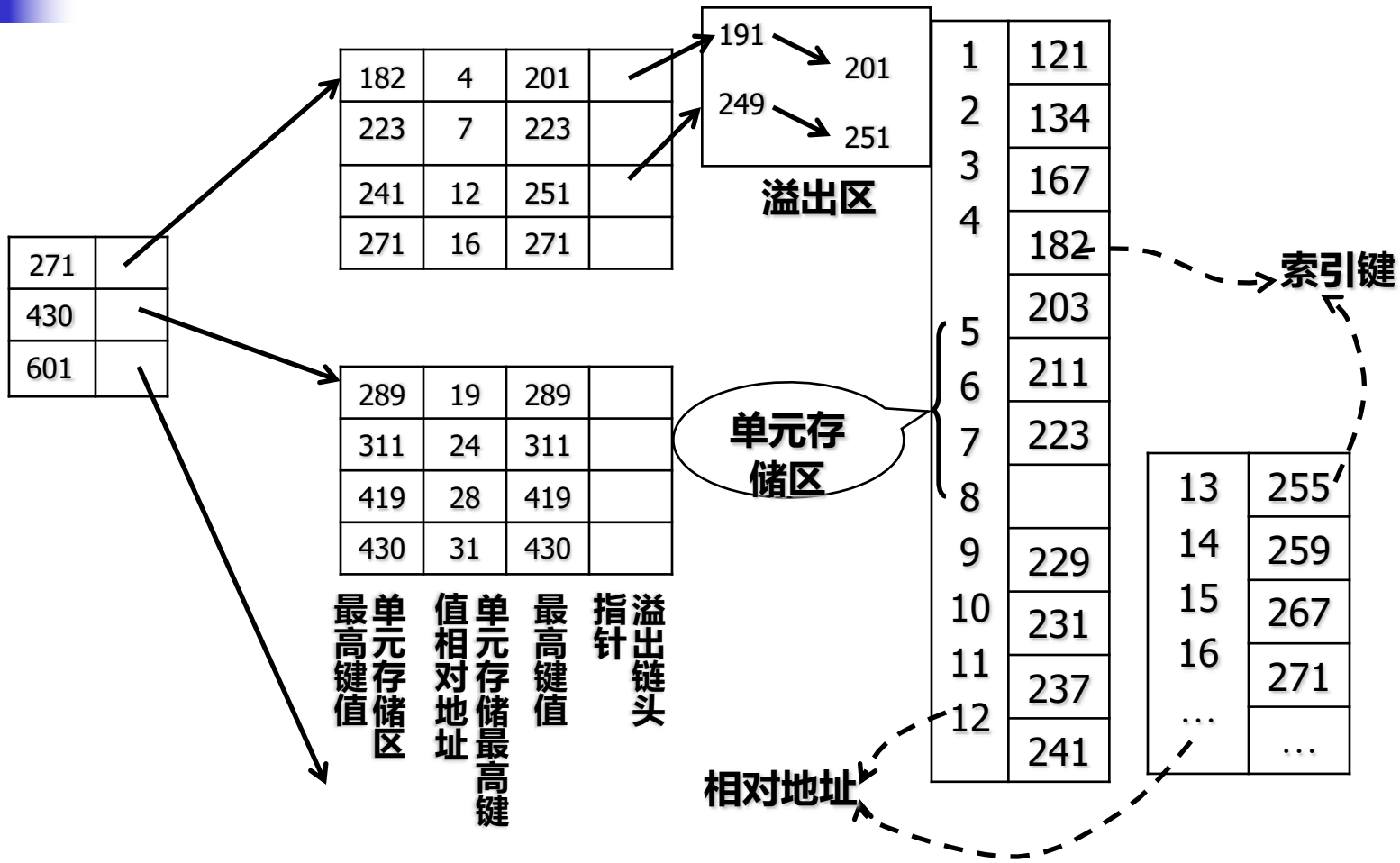


4.非稠密索引中，若干个记录组成一个单元存储区，单元存储区中的记录按索引键排序。

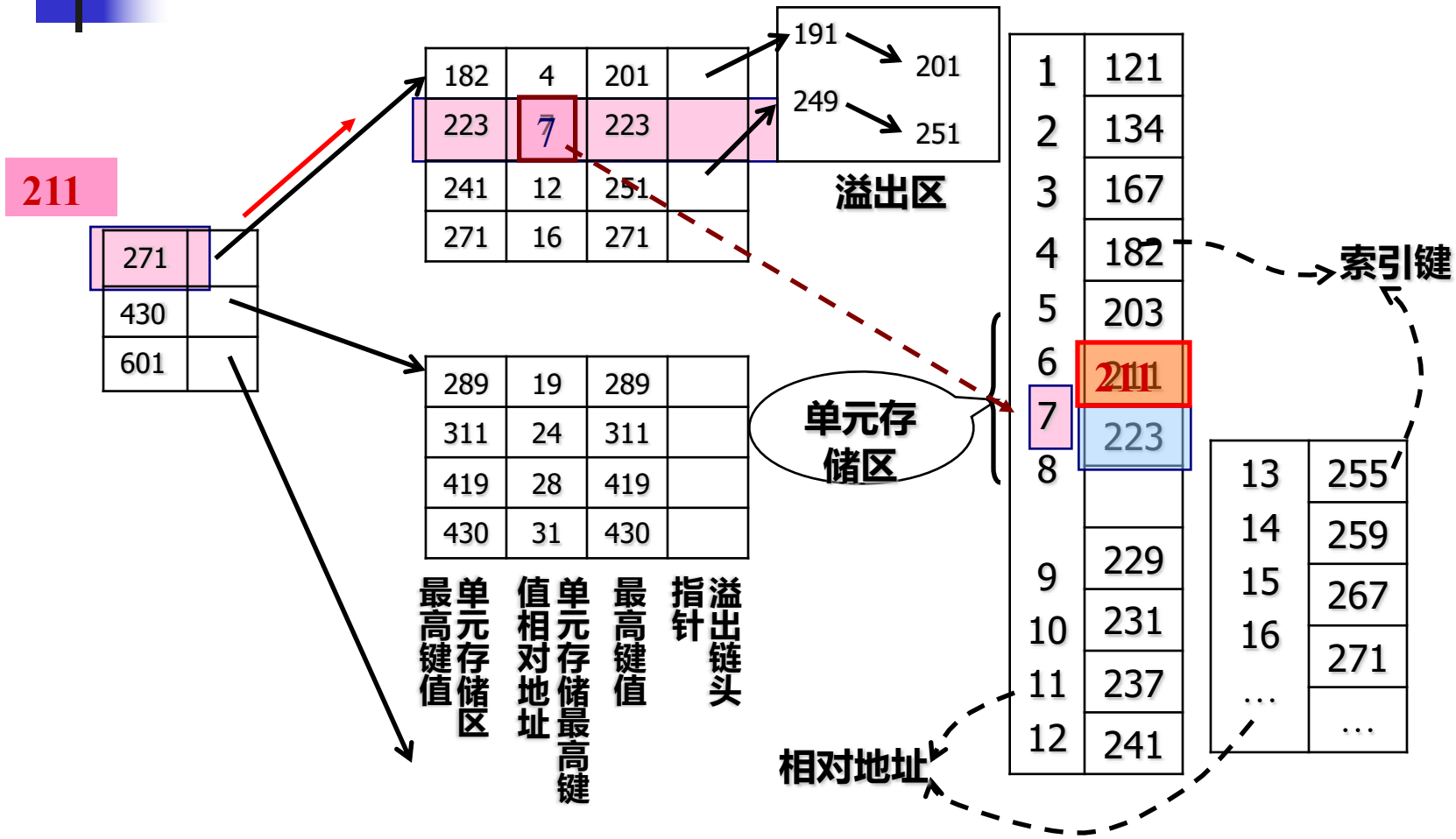
5.单元存储区装满后，可向溢出区溢出（用指针指向溢出区），但溢出太多时，指针链接次数增加，将导致数据库性能下降。

6.可以建立多级非稠密索引（通常最高一级索引应尽量保证可以常驻内存）。

示例


271
430
601
182 4 201
223 7 223
241 12 251
271 16 271
191 201
249 251
溢出区
289 19 289
311 24 311
419 28 419
430 31 430
单元存储区
最高键值
单元存储区相对地址
单元存储最高键值
溢出链头指针
单元存储区
1 121
2 134
3 167
4 182
203
5
6 211
7 223
8
9 229
10 231
11 237
12 241
相对地址
索引键
13 255
14 259
15 267
16 271
…

查找索引键为211的记录的存储地址

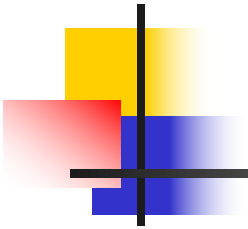


稠密索引（dense index）

- 每个键值对应一个索引项——稠密索引。
- 稠密索引的预查找功能（用记录的地址代替记录参与集合运算，减少I/O次数）。
- 索引溢出的问题

稠密索引中，每增加一个键值，就要增加一个索引项。索引也会有溢出的可能。

解决方法：1.在每个索引块中预留发展的空间
2.建立索引溢出区

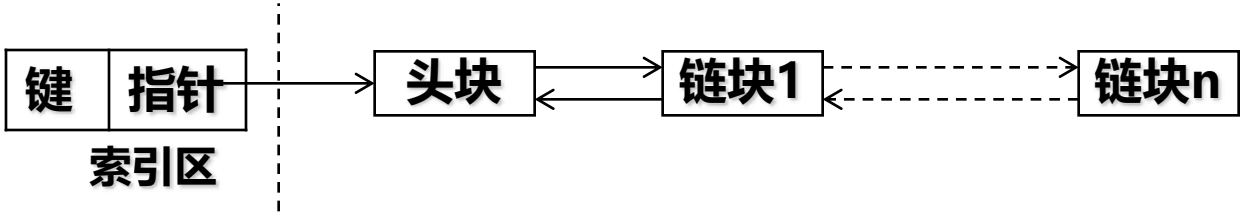


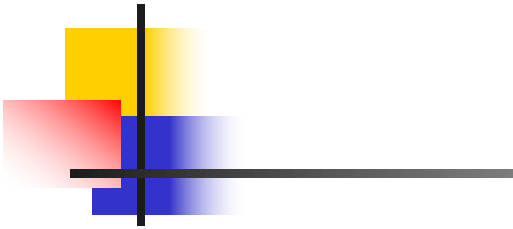
1.稠密索引中，若键值不唯一，则在最低级索引中，每个键值对应的可能是一个地址集（对应多个记录）。如果这些记录分散在不同的物理块内，稠密索引的优点并不能体现出来（并不一定能减少I/O）。

例如：某个键值对应100个记录，且这100个记录分散在100个物理块中，虽然通过稠密索引可以一次获得这100个地址，但仍然要访问100个物理块。

2.解决方法——簇集索引（clustering index）

采用簇集索引：把键值相同的记录在物理地址上集中存放。





3.缺点：建立簇集索引的开销较大，整个文件要重新组织。

常用索引总结

- 索引
  - 主索引
    - 按主键排序——非稠密索引
    - 不按主键排序——稠密索引
  - 次索引
    - 簇集索引——按索引键排序并簇集，稠密索引
    - 非簇集索引——不按索引键排序，稠密索引

5.4 动态索引

从数据结构角度看：静态索引是个多分树；动态索引是一种平衡多分树（即B-树），常用B^+树。

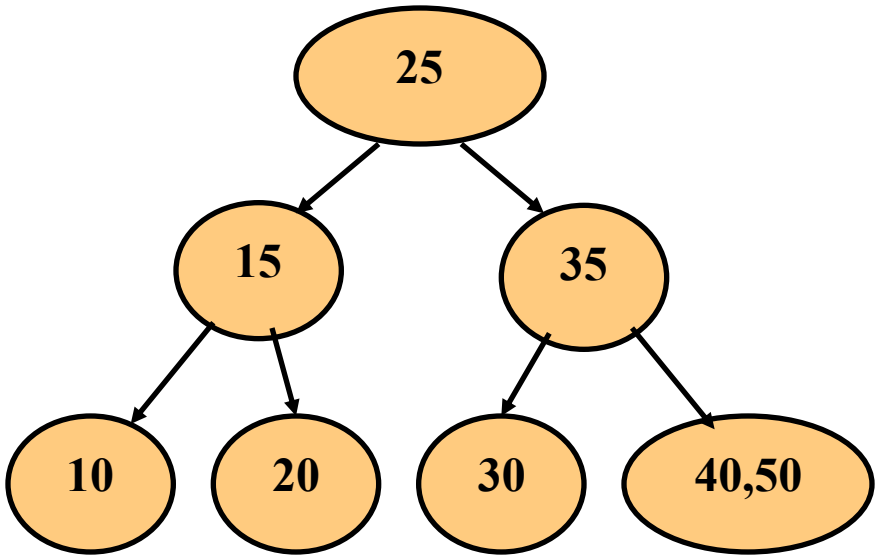
B^+树的限制和规定：

- 每个节点最多有2k个键值，k称为B^+树的秩(Order);
- 根节点至少有一个键值，其它节点至少有k个键值，节点内，键值有序存放;
- 除叶节点无子女外，其它节点若有J个键值，则有J+1个子女;
- 所有叶节点都处于树的同一层，即树始终保持平衡。

插入算法：

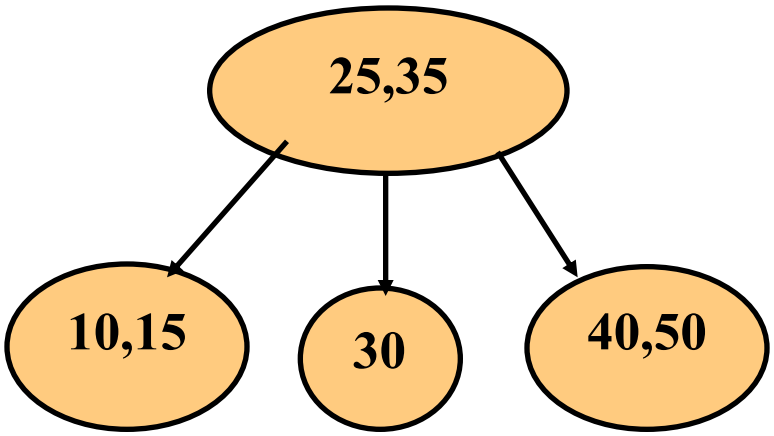
从根向叶搜索，直至相应叶节点，若该叶节点不满，则将键值插入叶节点中；如叶节点已满（即已经有2K个键值），则将此叶节点分裂为二，叶节点分裂后，其双亲节点也必须增加一个键值，若双亲节点不满，插入过程结束；否则双亲节点继续分裂为两个节点，如此继续直到插入过程中止。

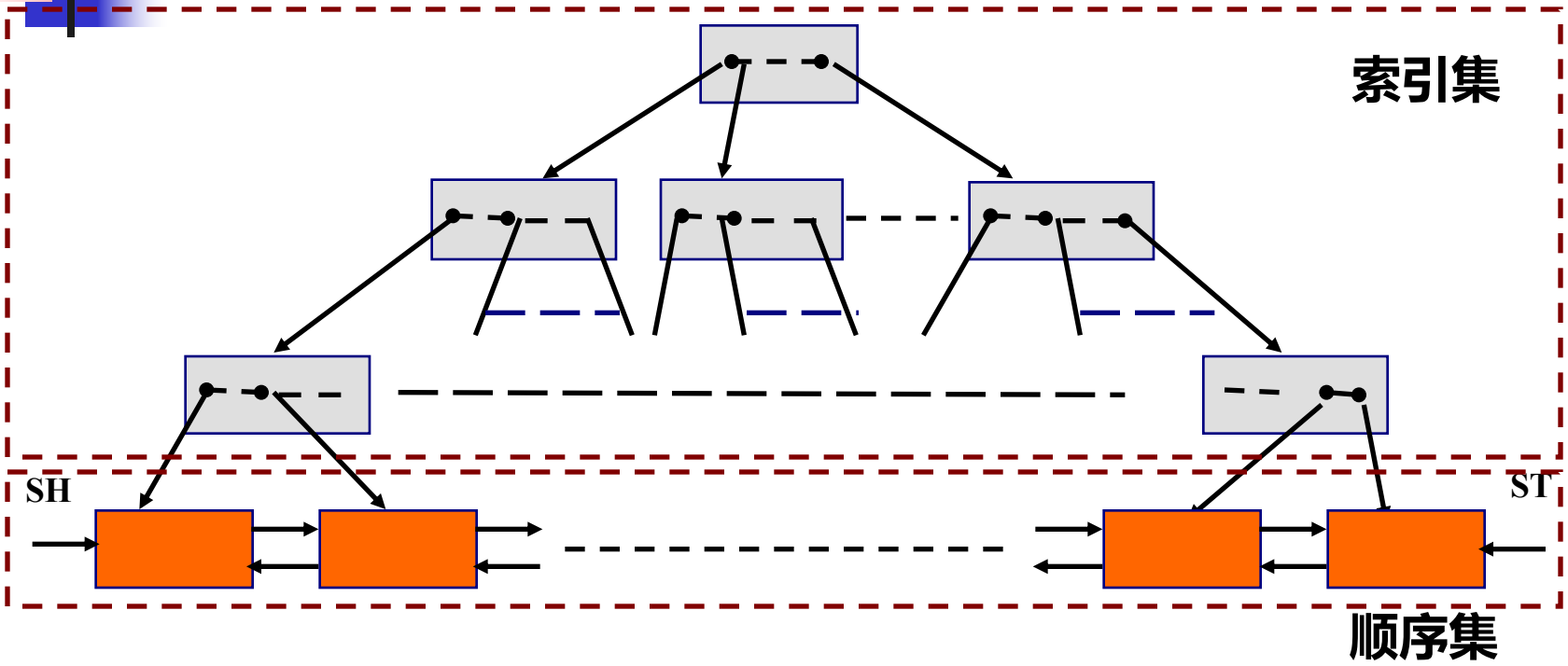
(K=1): 10，15，20，25，30，35，40，50



删除算法：

先从根节点出发，找到待删除键值所在叶节点；若删除该键值后，叶节点中键值数减为K-1个，则向其左右兄弟叶节点借一个键值，以保持每个叶节点存放键值不少于K个；若其左右叶节点都只有K个键值，则可将该叶节点与其左（或右）叶节点合并成包含2K-1个键值的叶节点，合并后，其双亲节点要减少一个键值，有可能导致双亲节点的合并。





B+树实现的主索引

B+树实现的主索引包含如下2部分：索引集和顺序集。

节点类型	块中索引键数	P_0	K_0	P_1	K_1	…	K_{n-1}	P_n

索引集节点

节点类型	块中索引键数	前向指针	后向指针	K_0	tid_0	K_1	tid_1	…	K_n	tid_n

顺序集节点

注：

1.节点一般是一个物理块。

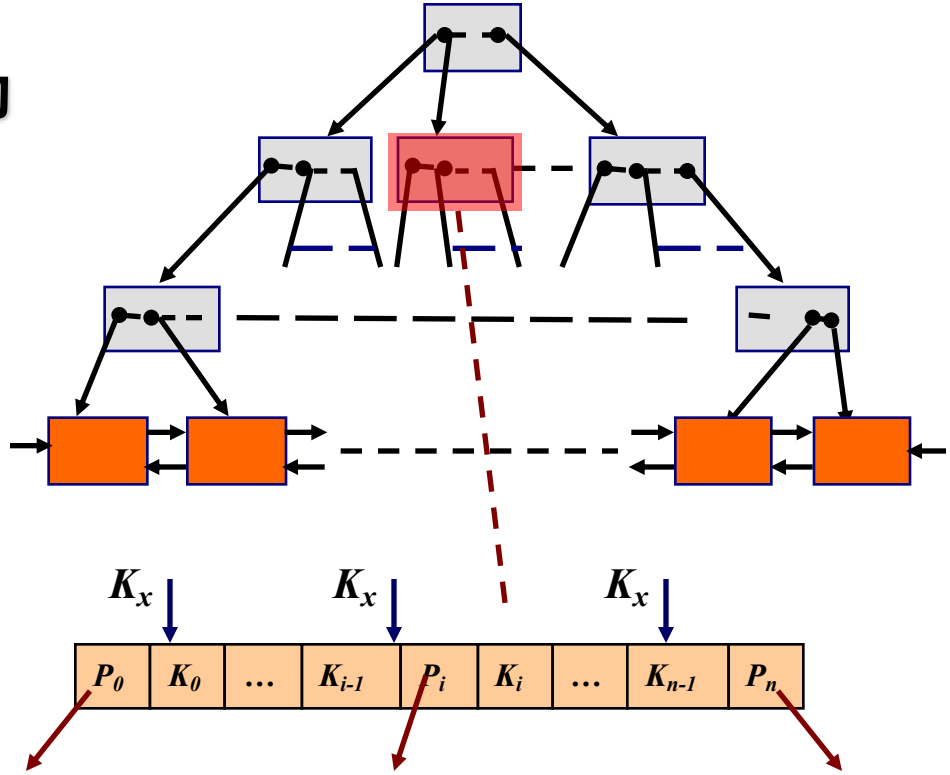
2.元组标识符tid，实际上是记录的地址，由块号和记录在块中的指针号组成（使用记录在块中的指针号表示记录在块中的位置，好于直接用记录在块中的地址，方便记录在块内移动）。

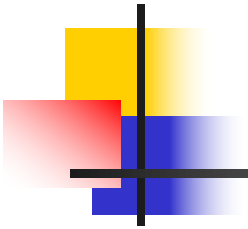
- 要找键值K_X所对应的记录时，从索引树的根开始，按下面的规则自上而下地搜索：

1.若$K_x<K_0$，则沿P_0所指的节点向下搜索；

2.若$K_x>K_{n-1}$，则沿P_n所指的节点向下搜索；

3.若$K_{i-1}<K_x\leq K_i$，则沿P_i所指的节点向下搜索。

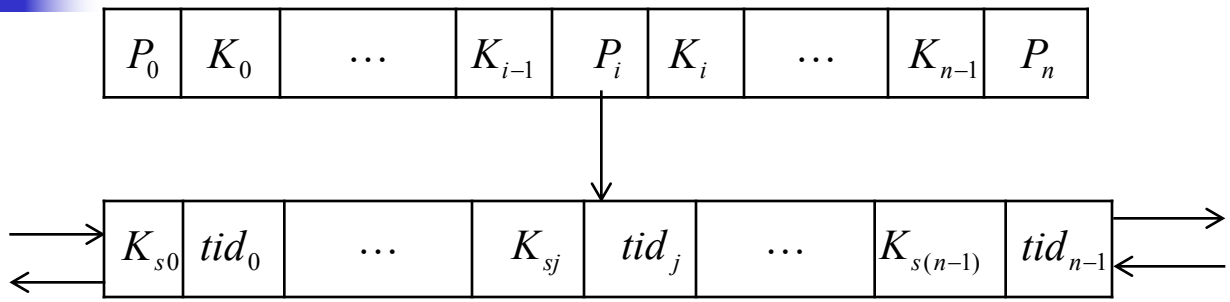




问题：若在某个索引集结点中找到了待查找记录相应的索引键值，是否还要继续遍历B+树，为什么?

注意：索引集节点中的键值不一定是文件中当前存在的键值（仅起“导航路标”的作用）。在搜索过程中，即使发现索引集节点中的键值等于要找的键值，查找仍要按上述规则进行下去。

索引集与顺序集的联系

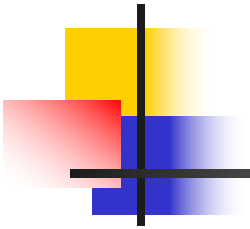


顺序集节点中的键值要满足如下关系：

如果$P_i=P_0$，则$K_{s0}<K_{s1}<\ldots<K_{s(n-1)}\leq K_0$；

如果$P_i=P_n$，则$K_{s(n-1)}>\ldots>K_{s1}>K_{s0}>K_{n-1}$；

否则：$K_{i-1}<K_{s0}<K_{s1}<\ldots<K_{s(n-1)}\leq K_i$。



- B+树实现的主索引稍加修改后也可用于次索引（把顺序集结点的tid换成指针，因为一个键值可能对应多个tid）。
- B+树实现的各种索引都是稠密索引（非稠密索引的概念源于静态索引），提供了顺序搜索的功能，这是它的优点。

- 搜索B+树所需的I/O次数决定于其级数。

设索引属性不同键值的数目为N，若索引键不是候选键，则记录数通常大于N。

B+树的级数决定于N，而不是记录数！

假设B+树索引部分共有L级，其秩为k，则各级的最小结点数依次为：

$1，2，2(k+1)，\ldots，2(k+1)^{L-2}$

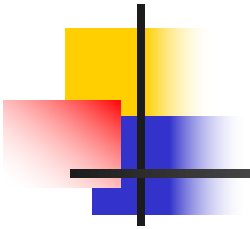
顺序集至少有$2(k+1)^{L-2}$个结点。

得出：

$N \geq 2(k+1)^{L-2} \times k$

$L \leq 2+log_{(k+1)}(N/2k) \approx 1+ log_{(k+1)}(N/2)$

L决定了找到所需顺序集结点所需的I/O次数（访问数据还要额外的I/O），例如k=99，N=2,000,000，有L<5，即至多经过4次I/O就可以找到相应的顺序集结点。



- B^+树提供3种存取路径：

1.通过索引集进行树形搜索

2.通过顺序集进行顺序搜索

3.先通过索引找到入口，再沿顺序集顺序搜索