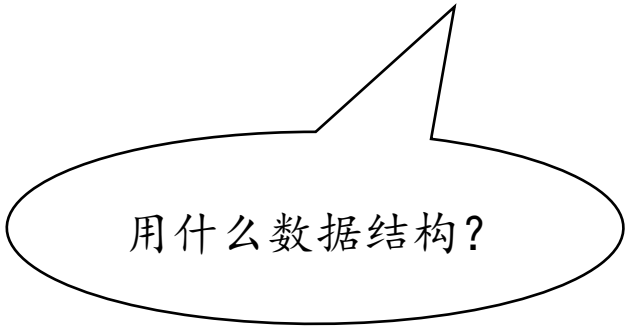
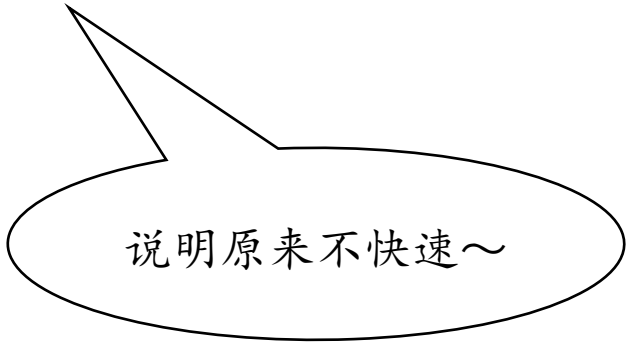
03 索引结构、实现及使用

南京大学软件学院

什么是索引?

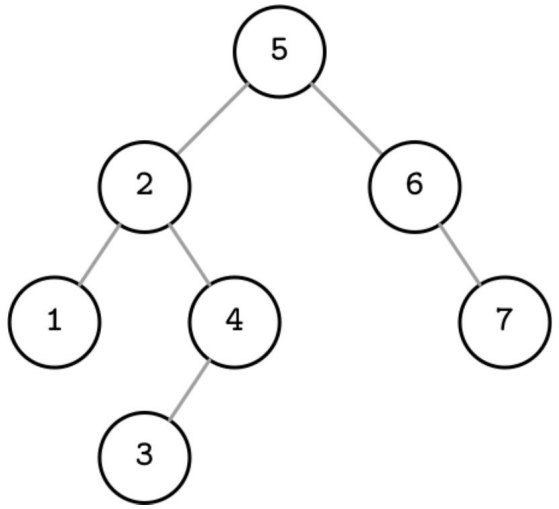
数据库管理系统（DBMS）中一个排序的数据结构，以协助快速查询、更新数据库表中数据。

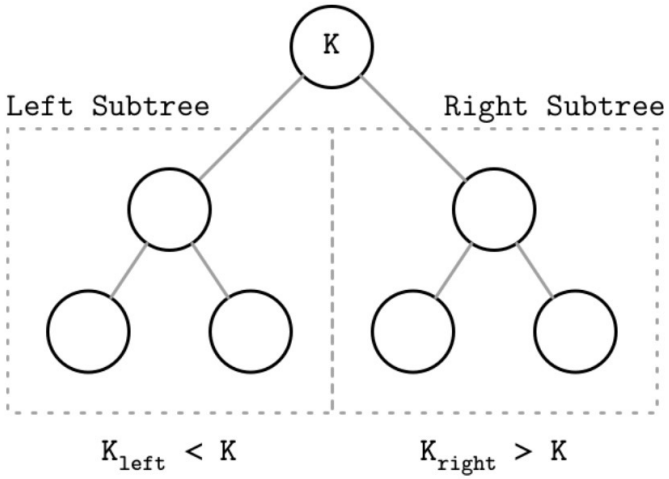




树（Tree）

二分搜索树（BST，Binary Search Trees）





节点：键、值、两个字节点指针（二分）

有且只有一个根节点

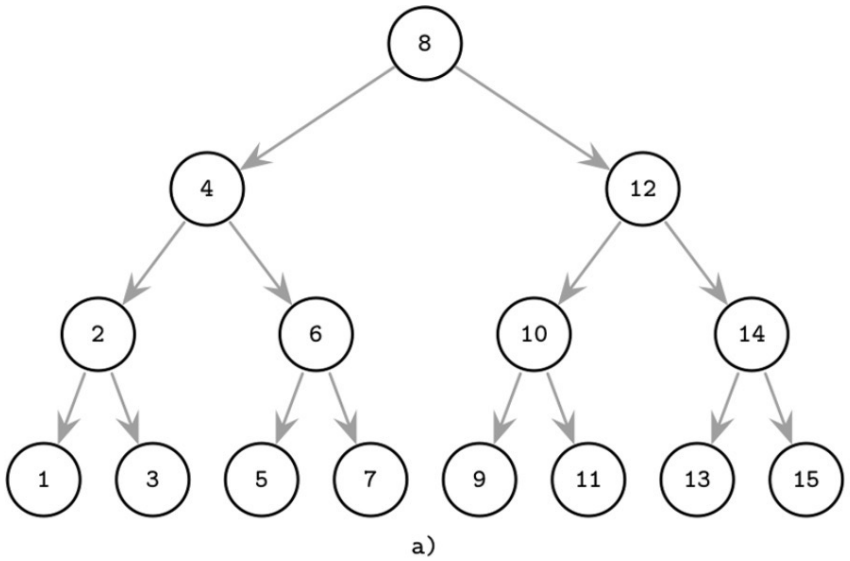
有序的内存结构，高效的键值查找

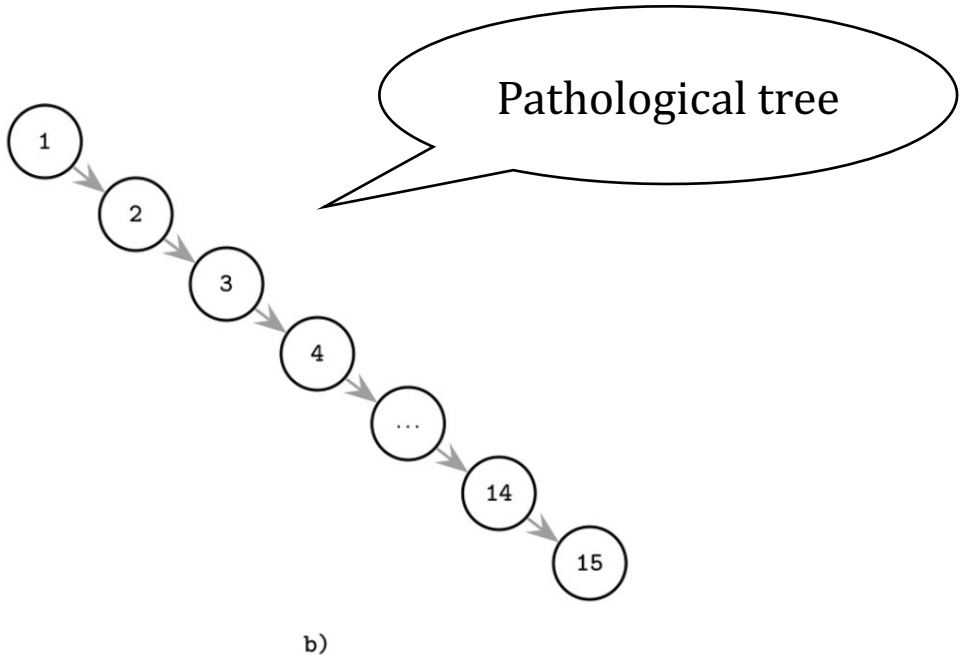
一直沿着左指针，得到最小，反之最大

二分树节点的约束条件

a node key is *greater than* any key stored in its left subtree and *less than* any key stored in its right subtree [SEDGEWICK11]

为什么需要 Balancing



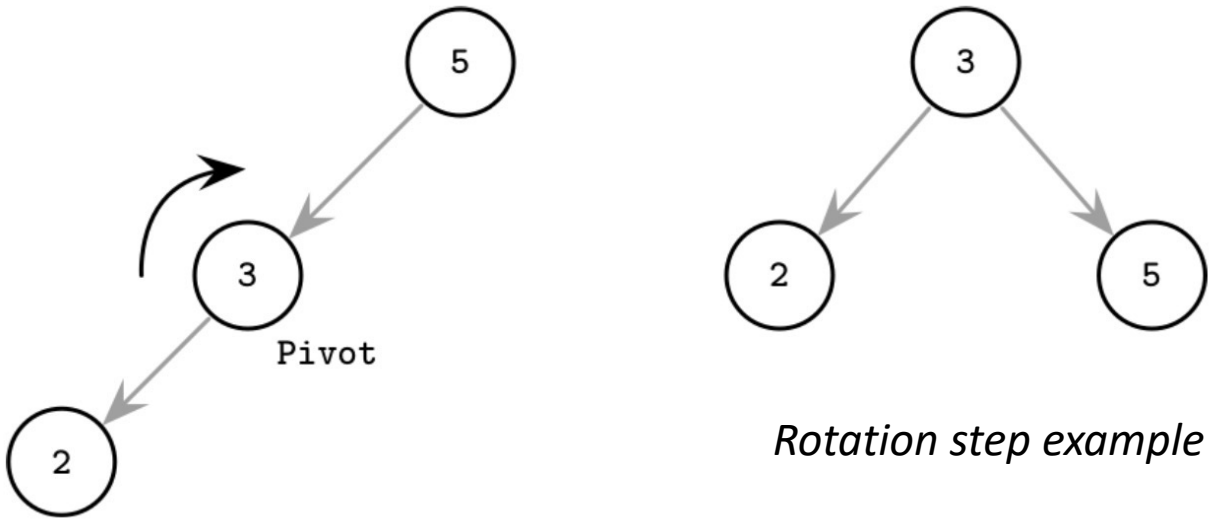


O（$\log_2 N$）

O(N)

平衡的本质是：沿着左指针或右指针移动会将搜索空间平均减少一半

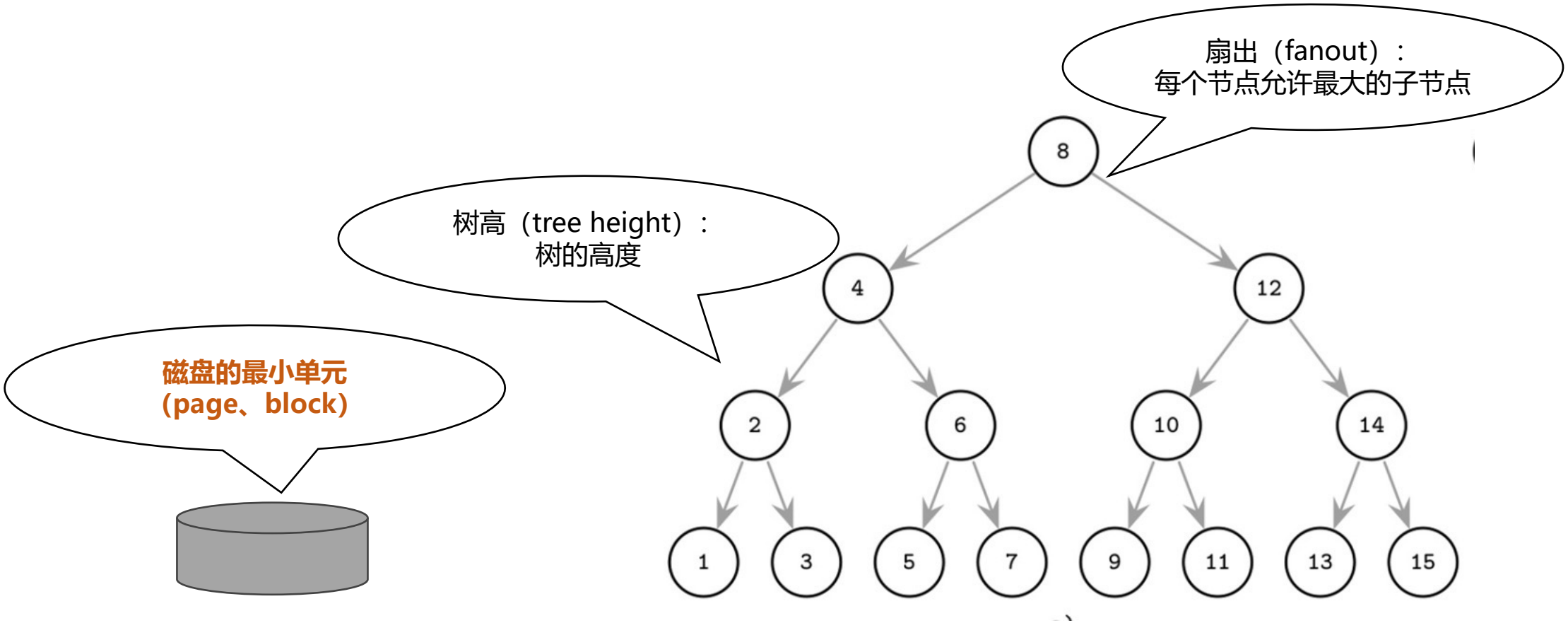
怎么做 Tree Balancing?



Rotation step example

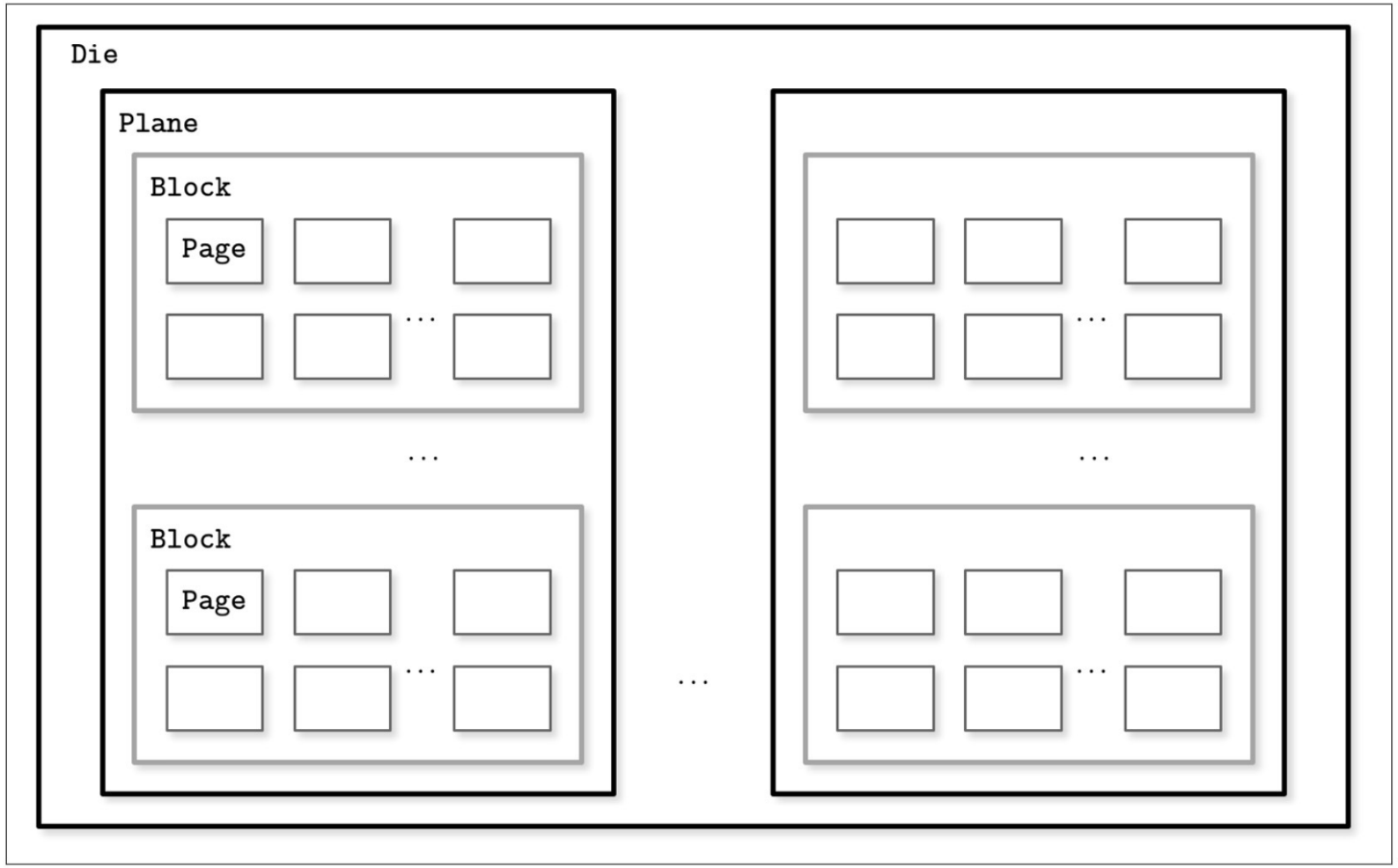
保持树的平衡的方法之一是在添加或删除节点后执行旋转

基于磁盘存储的树



高扇出，以改善临近键的数据局限性；**低高度**，以减少遍历期间的寻道次数

HDD和SSDs的差异



SSD organization schematics

Page：2KB-16KB

Block：64-512 Page

Plane

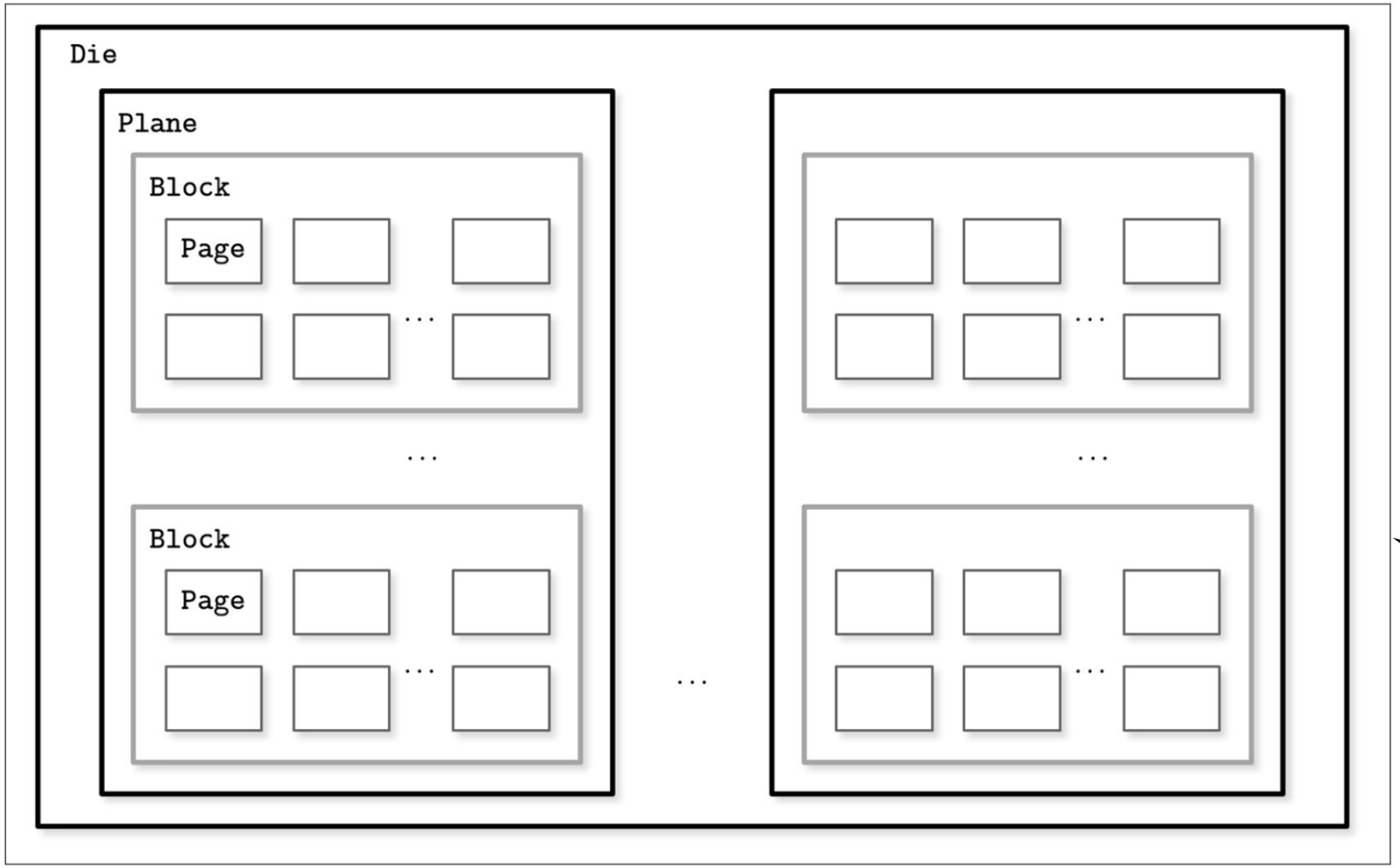
Die

可写的最小单元是Page

最小的擦除单元是Block

SSD不过于像HDD一样非常强调随机I/O和顺序I/O的差别，因为延迟差异不是很大。但是，预取、读取连续页和内部并行等方面，二者差距依旧存在[GOOSSAERT14]

好像还缺了点啥?



SSD organization schematics

闪存转换层

(Flash Translation Layer)

Page id 映射到对应的物理位置，

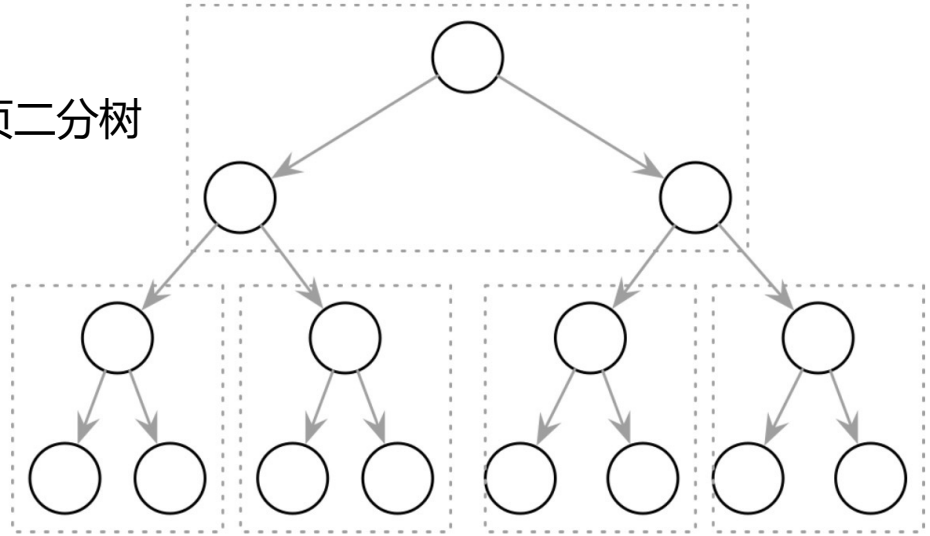
跟踪管理空的、被写过的、被丢弃的页

垃圾收集，查找安全擦除的块，迁移某些页

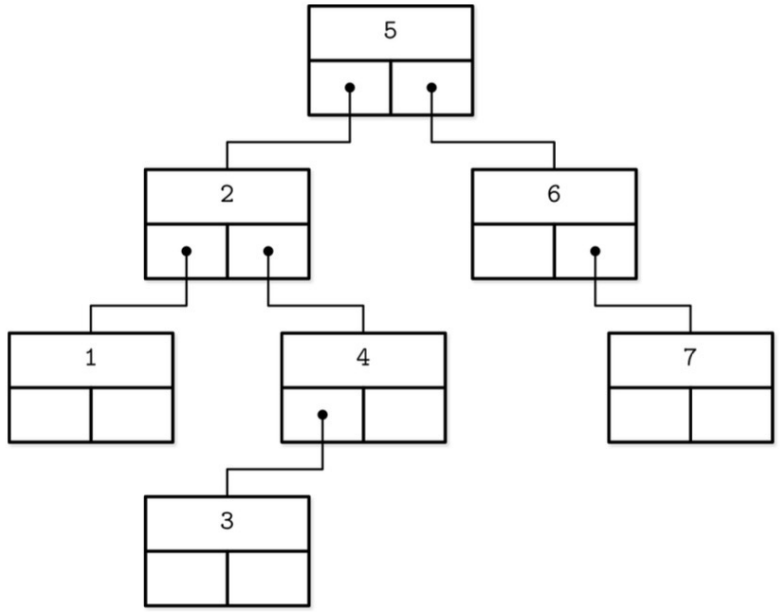
操作系统
块设备（Block Device）
在处理磁盘存储数据的结构
需要考虑块读取的问题

逻辑图和实现结构图的差异

分页二分树



Paged binary trees

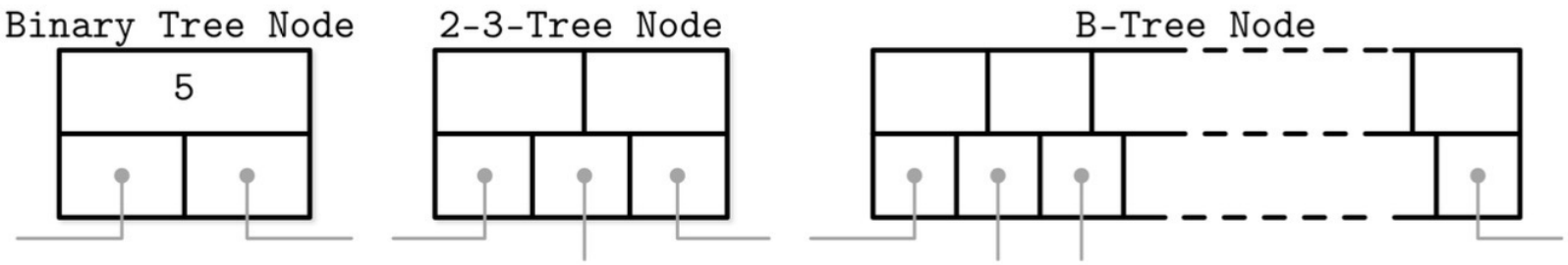


Alternative representation of a binary tree

如何实现这两个目标

高扇出，以改善临近键的数据局限性；**低高度**，以减少遍历期间的寻道次数

B-Tree（B+Tree）结构的目标

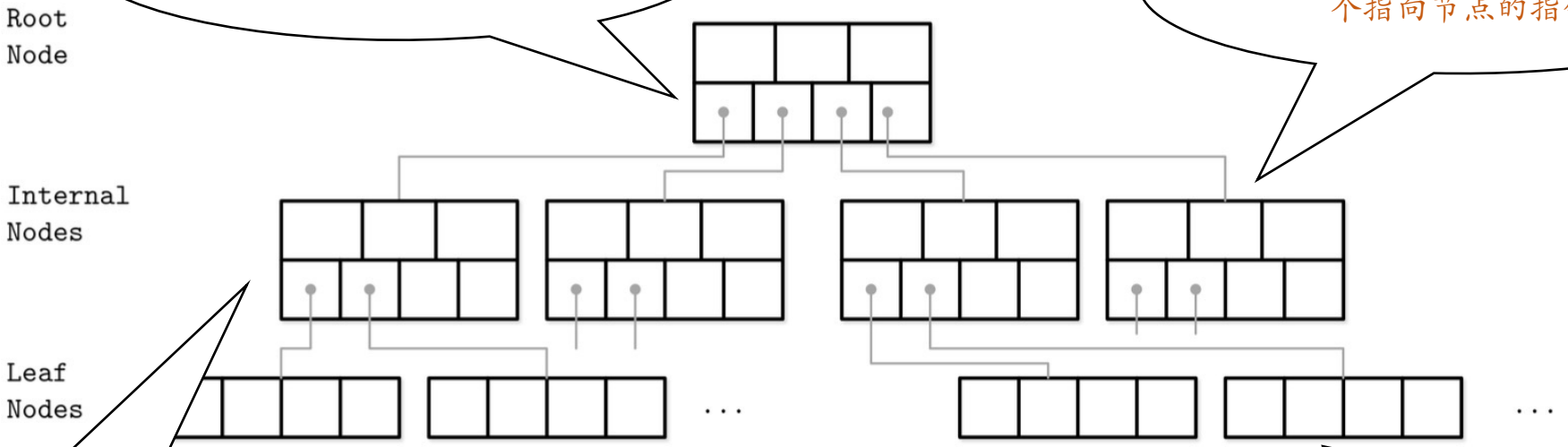


Binary tree, 2-3-Tree, and B-Tree nodes side by side

B树构建了一个快速导航和定位搜索项的层次结构，

达到这个目标需要高扇出，低树高

B-Tree（B+Tree）结构



页组织技术，一个节点就是一个Page

每个节点最多N个键和N+1个指向节点的指针

节点容量和实际持有键的数量之间的关系叫占用率

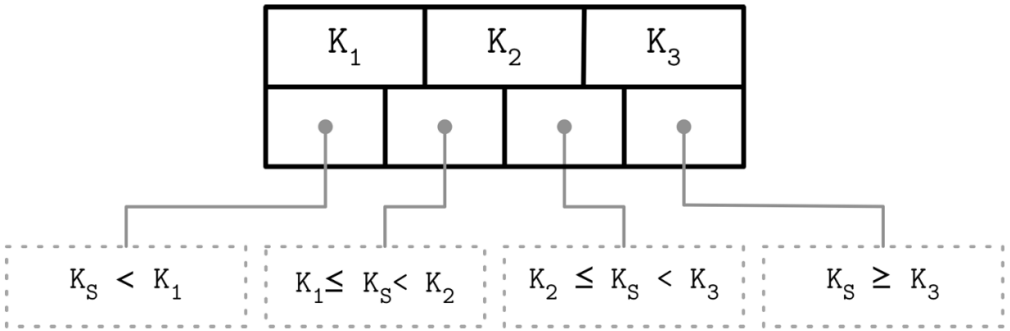
是不是只有叶节点存值是B+树和B树的差异

B-Tree node hierarchy

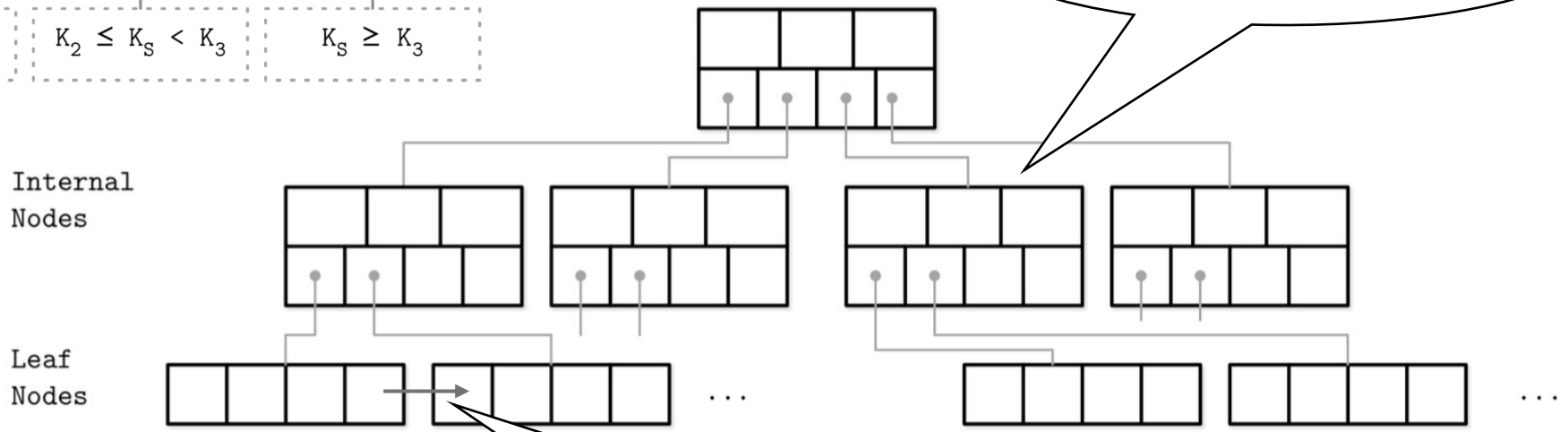
B tree作为一类共享所有或大部分上述属性的数据结构的统称，其实更精确的描述是B+Tree
[KNUTH98]将高扇出的树称之为多向树（multiway tree）

分隔符

键——
索引条目（index entry）
分隔符（separator key）
分割单元（divider cell）



键排好序了，用二分搜索，
通过对键比较从高层次搜索
到低层次，最后搜索到叶



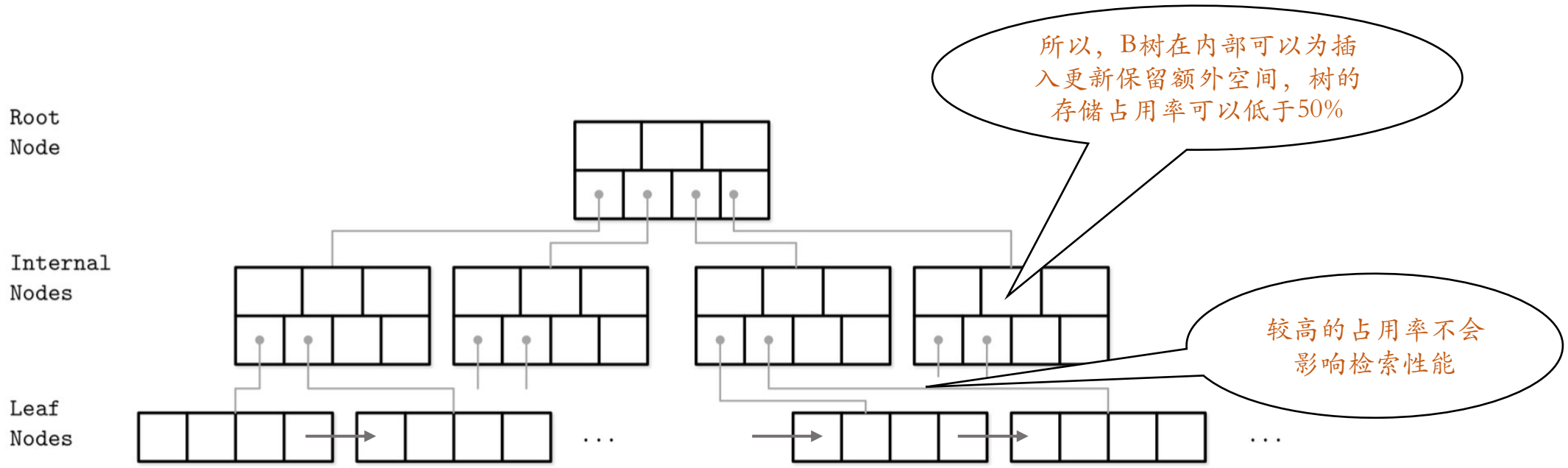
是不是只有叶节点存值是B+
树和B树的差异

叶节点的同级节点指针

B树的本质特点是 ——

它不是自上而下构建的（例如BST），而是采用相反的构建方式——自下而上

随着叶节点数量的增加，内部节点的数量和树的高度也将增加



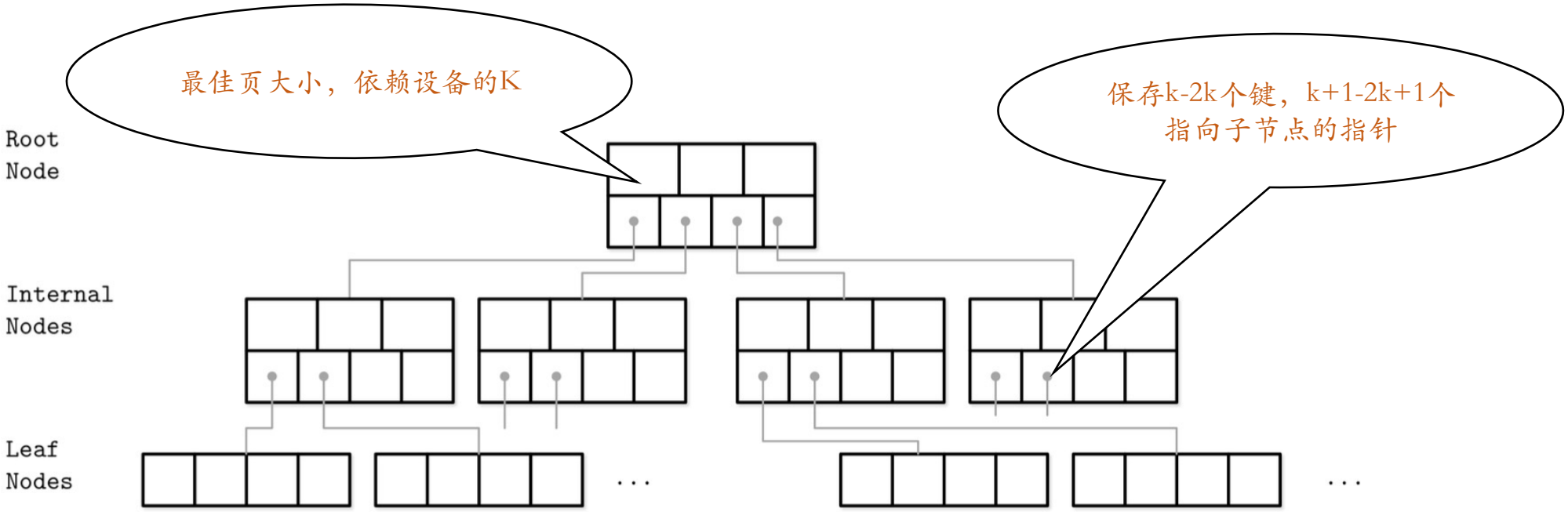
B树查找算法

- 查找，从根节点到叶节点的单向遍历
  - 从根节点上执行二分搜索算法，将要搜索的K，与存储在根节点中的Kn进行比较，直到找到大于K的第一个分隔键，这样定位了一个要搜索的子树，顺着相应指针继续相同的搜索过程，直到目标叶节点，找到数据主文件指针
  - 进行范围扫描时，迭代从找到的最近的键值开始，顺着同级指针继续移动，直到达到范围的末尾
- 复杂度是多少？

B树查找算法的复杂度

- 算法复杂度从两个角度讨论——块传输、键比较
- 块传输：对数基于N（每个节点的键数，也就是容量）。从根节点每往下走一层，节点个数增加K倍，并跟随一个子指针可以将搜索空间减少至N分之一。所以，在查找期间最多寻址$\log_k M$（M是B树中键的总数）个页来查找一个搜索Key。换句话说，块传输的数量等于树的高度H。
- 比较次数：对数基是2，二分搜索完成，每次比较，搜索空间减半，复杂度是$\log_2 M$
- 所以，一般书上都把查找复杂度记为logM

块中键数量的设计



B-Tree node hierarchy

复习：“标准” 的B+树定义

一棵m阶的B+树需要满足下列条件：

1) 每个分支结点最多有m 棵子树(孩子结点)。

2) 非叶根结点至少有两棵子树,其他每个分支结点至少有[m/2]棵子树。

3) 结点的子树个数与关键字个数相等。

4) 所有叶结点包含全部关键字及指向相应记录的指针,叶结点中将关键字按大小顺序排列并且相邻叶结点按大小顺序相互链接起来。

5) 所有分支结点(可视为索引的索引)中仅包含它的各个子结点(即下一级的索引块)中关键字的最大值及指向其子结点的指针。

B+树的插入与删除操作和B树类似。

参考文献：

[1]: 严蔚敏, 吴伟民. 数据结构: C 语言版[M]. 清华大学出版社有限公司, 2002.

[2]: 王道论坛. 2025年数据结构考研复习指导[M].电子工业出版社, 2024.

复习：“标准” 的B+树定义

省流版：

1） m阶B+树非根内部节点关键字个数n范围为⌈m/2⌉≦n ≦m，根节点1≦n ≦m；

2） 在 B+树中,叶结点包含了全部关键字,即在非叶结点中出现的关键字也会出现在叶结点中；

3） 叶节点之间相互链接。

参考文献：

[1]: 严蔚敏, 吴伟民. 数据结构: C 语言版[M]. 清华大学出版社有限公司, 2002.

[2]: 王道论坛. 2025年数据结构考研复习指导[M].电子工业出版社, 2024.

国外“标准” 的B+树定义

省流版：

1) Minimum 50% occupancy (except for root). Each node contains d <= m <= 2d entries. The parameter d is called the order of the tree.

2) A B+Tree is a self-balancing, ordered m-way tree for searches, sequential access, insertions, and deletions in $O(\log_m n)$ where m is the tree fanout. → It is perfectly balanced (i.e., every leaf node is at the same depth in the tree) → Every node other than the root is at least half-full $m/2-1 \leqslant \#keys \leqslant m-1$

参考：

[1]: CS346,2014,https://web.stanford.edu/class/cs346/2014/Blink.pdf

[2]: CMU15445/645,2024, https://15445.courses.cs.cmu.edu/fall2024/slides/08-indexes1.pdf

[3]: visualization BPlusTree, https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

“标准” 的B+树定义

先前所述：

1） m阶B+树非根内部节点关键字个数n范围为

[m/2]≦n ≦m，根节点1≦n ≦m；

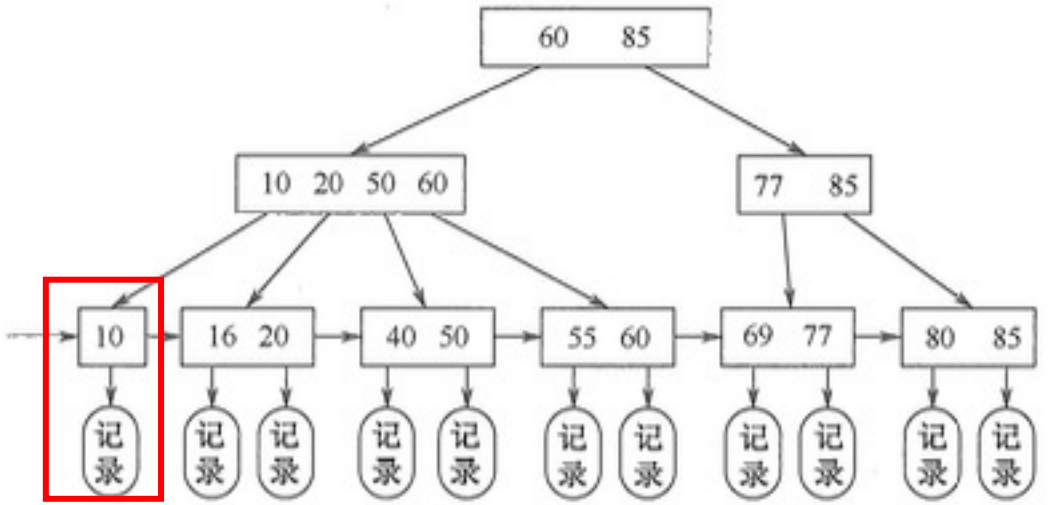


图 7.8 B+树结构示意图

阶数order、扇出fanout、度数degree……这些“标准”令人精神分裂。甚至在同一个标准下还会自相矛盾。为了解决这个问题，下面给出一种最方便理解的解释。

接下来会用到的B+树定义

一棵m阶(order)的B+树有以下特征：

1) B+树是一种自平衡、有序的m路(m-way)树，用于搜索、顺序访问、插入和删除，其中m是树的扇出(fanout);

2) 它是完全平衡的（即，每个叶节点在树中的深度相同）；

3) 除根之外的每个节点至少是半满(half full)的，m/2-1≤ #keys≤m-1，下限不用取整;

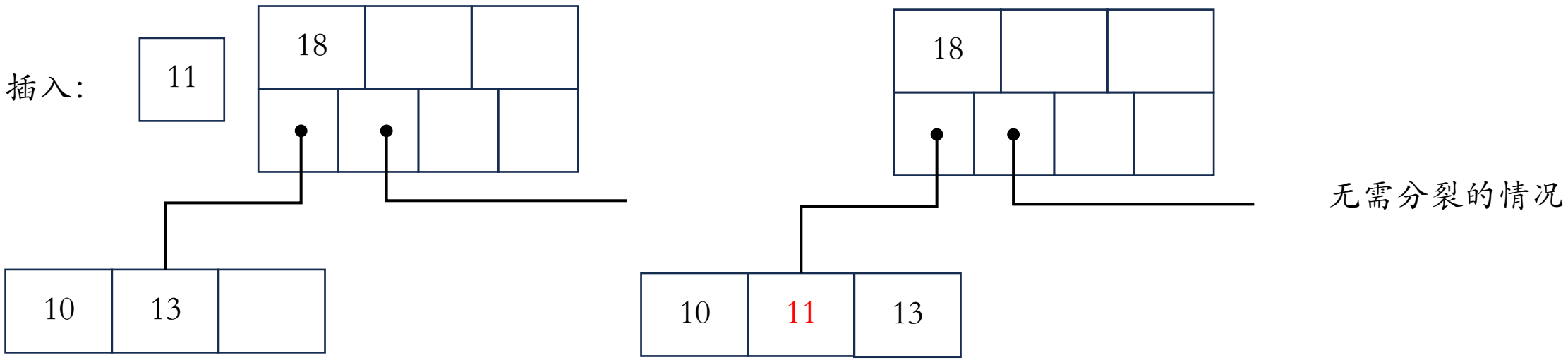
4) 每个具有k个键的内部节点都有k+1个非空子节点;

5) 所有叶结点包含全部关键字及指向相应记录的指针,叶结点中将关键字按大小顺序排列并且相邻叶结点按大小顺序相互链接起来;

6) 所有分支结点(可视为索引的索引)中仅包含它的各个子结点(即下一级的索引块)中关键字的最大值及指向其子结点的指针。

如果不慎遇到其他定义（阶d为扇出的一半等），请以实际情况为准。

B+树的节点插入和分裂（叶节点分裂）



Step 1: 利用B树查找算法，查找算法定位目标叶节点，并将新值关联

Step 2: 有空就插入，流程结束。如果空间不足，就会发生“节点溢出”overflow，必须先分裂再插入

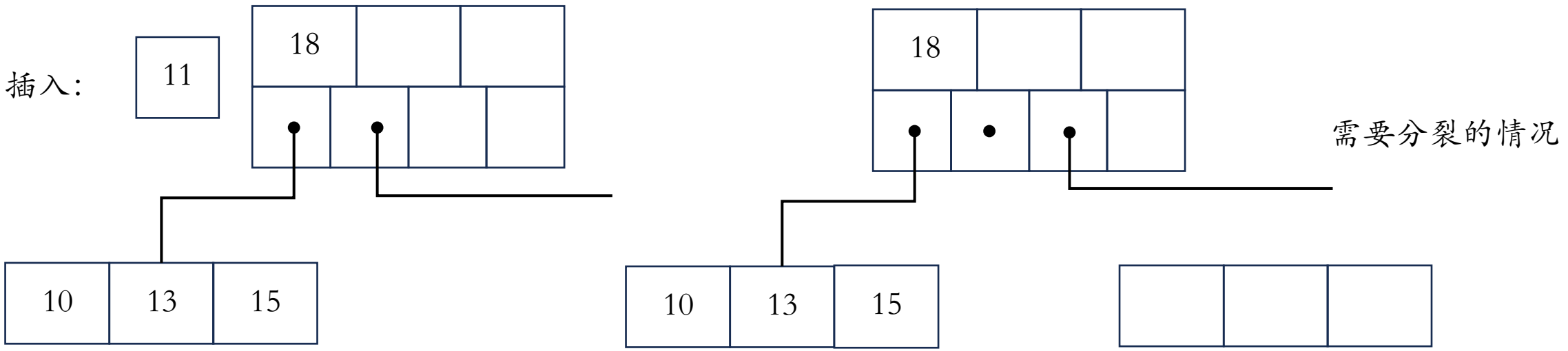
C1:叶节点：超过m-1

C2:非叶节点：指针超过m，

Step 3: 分裂——分配新节点，将一半（⌈(m-1)/2⌉以后）元素从待分裂节点传输给新节点，并添加它的第一个键和指向父节点的指针，将原节点下标为⌈(m-1)/2⌉后的元素复制到非叶节点，这时候，键被提升了（promote/copy up）

执行分裂的数组下标(1开始)称之为分裂点（也叫中点middle key），分裂点之后的所有元素被传输到新创建的兄弟节点

B+树的节点插入和分裂（叶节点分裂）



Step 1: 利用B树查找算法，查找算法定位目标叶节点，并将新值关联

Step 2: 有空就插入，流程结束。如果空间不足，就会发生“节点溢出”overflow，必须先分裂再插入

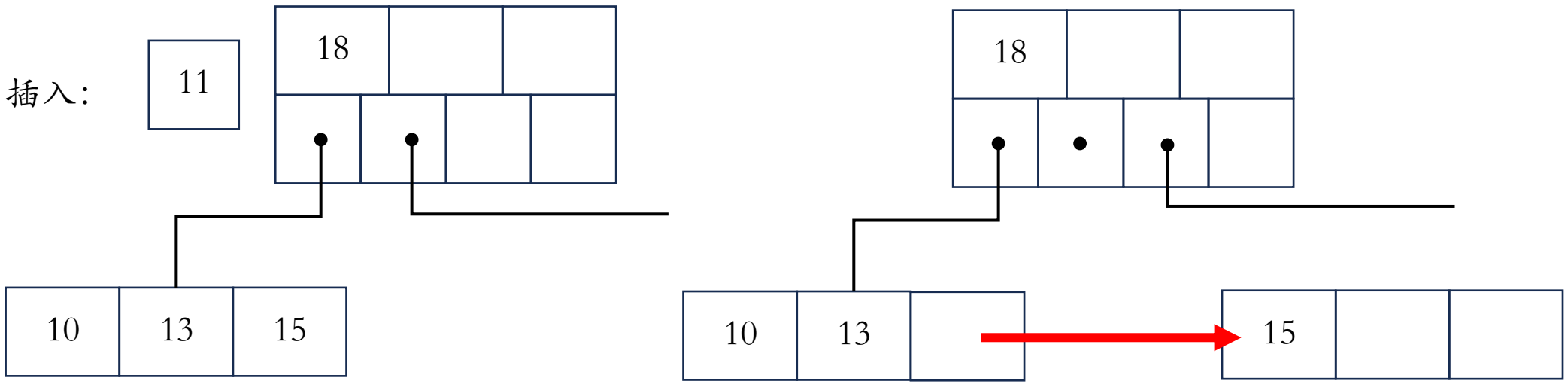
C1:叶节点：超过m-1

C2:非叶节点：指针超过m，

Step 3: 分裂——分配新节点，将一半（⌈(m-1)/2⌉以后）元素从待分裂节点传输给新节点，并添加它的第一个键和指向父节点的指针，将原节点下标为⌈(m-1)/2⌉后的元素复制到非叶节点，这时候，键被提升了（promote/copy up）

执行分裂的数组下标(1开始)称之为分裂点（也叫中点middle key），分裂点之后的所有元素被传输到新创建的兄弟节点

B+树的节点插入和分裂（叶节点分裂）



Step 1: 利用B树查找算法，查找算法定位目标叶节点，并将新值关联

Step 2: 有空就插入，流程结束。如果空间不足，就会发生“节点溢出”overflow，必须先分裂再插入

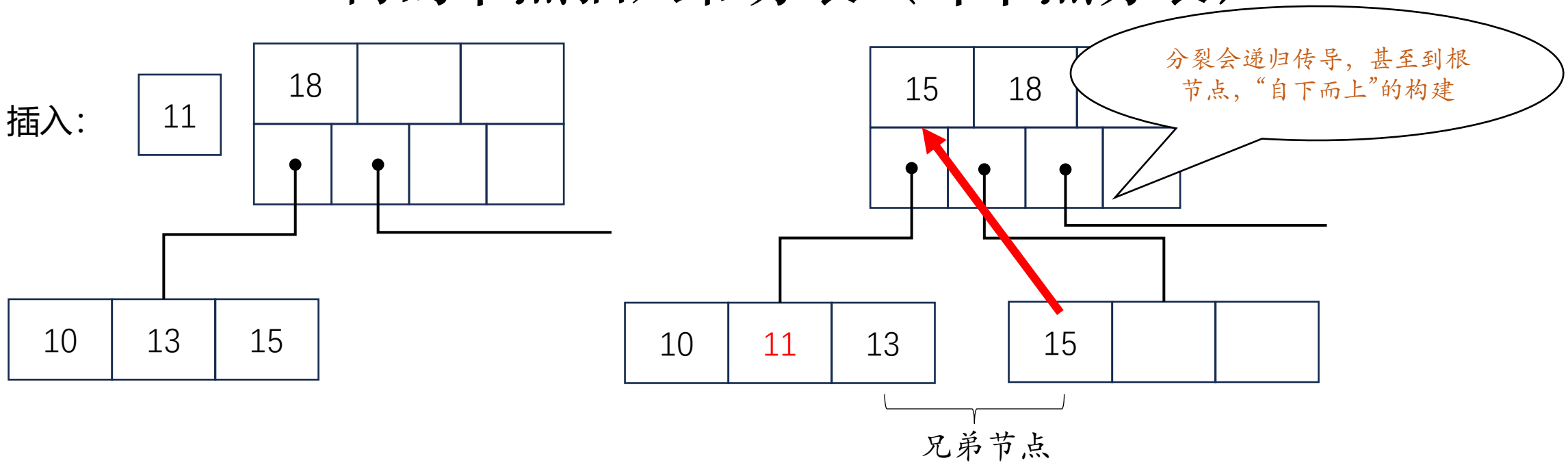
C1:叶节点：超过m-1

C2:非叶节点：指针超过m，

Step 3: 分裂——分配新节点，将一半（⌈(m-1)/2⌉以后）元素从待分裂节点传输给新节点，并添加它的第一个键和指向父节点的指针，将原节点下标为⌈(m-1)/2⌉后的元素复制到非叶节点，这时候，键被提升了（promote/copy up）

执行分裂的数组下标(1开始)称之为分裂点（也叫中点middle key），分裂点之后的所有元素被传输到新创建的兄弟节点

B+树的节点插入和分裂（叶节点分裂）



Step 1: 利用B树查找算法，查找算法定位目标叶节点，并将新值关联

Step 2: 有空就插入，流程结束。如果空间不足，就会发生“节点溢出”overflow，必须先分裂再插入

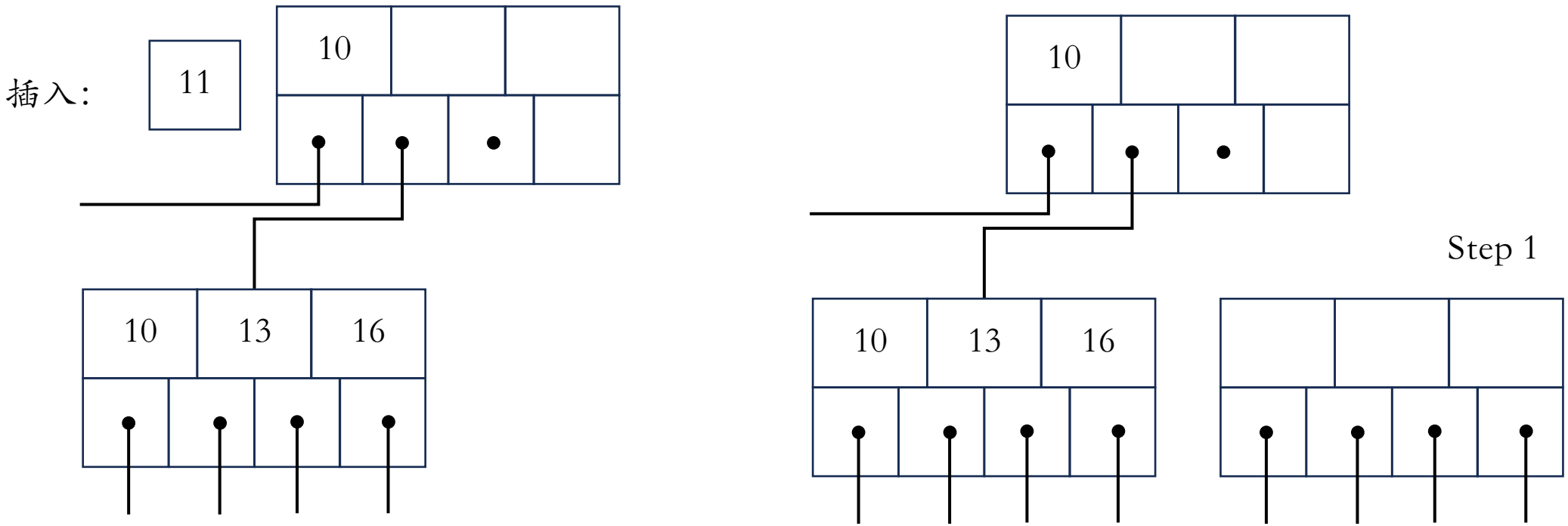
C1:叶节点：超过m-1

C2:非叶节点：指针超过m，

Step 3: 分裂——分配新节点，将一半（⌈(m-1)/2⌉以后）元素从待分裂节点传输给新节点，并添加它的第一个键和指向父节点的指针，将原节点下标为⌈(m-1)/2⌉后的元素复制到非叶节点，这时候，键被提升了（promote/copy up）

执行分裂的数组下标(1开始)称之为分裂点（也叫中点middle key），分裂点之后的所有元素被传输到新创建的兄弟节点

B+树的节点插入和分裂（非叶节点分裂）



Step 1: 创造新节点，如没有空位， 则对父节点继续分裂

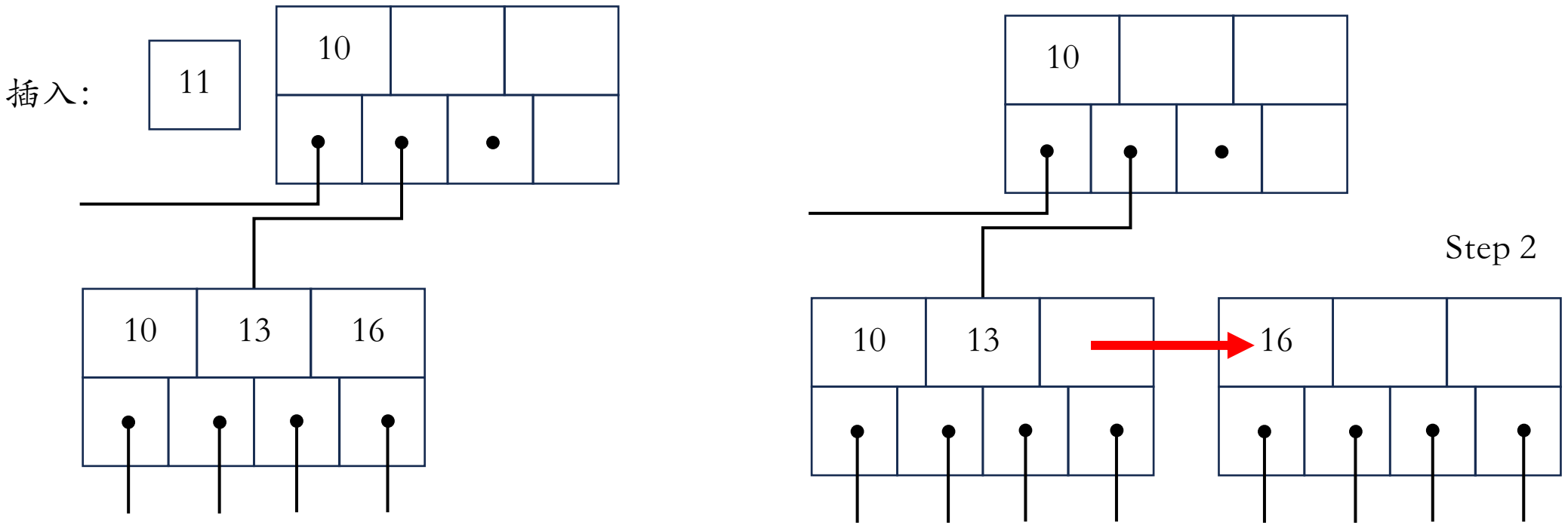
Step 2: 将原节点下标为⌈(m-1)/2⌉以后的元素，移动到新节点

Step 3: 分裂点的键被提升(promote/push up)到父一级

Step 4: 插入要插入的节点

Step 5: 生成额外指针，指向新裂变的节点

B+树的节点插入和分裂（非叶节点分裂）



Step 1: 创造新节点，如没有空位， 则对父节点继续分裂

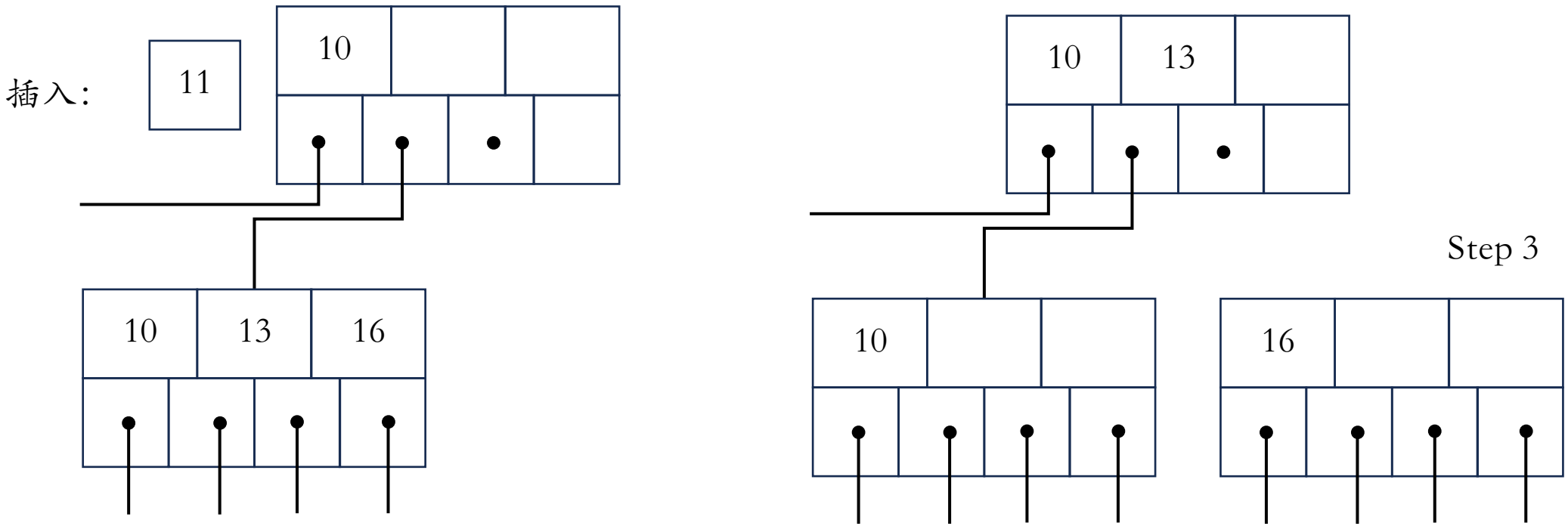
Step 2: 将原节点下标为⌈(m-1)/2⌉以后的元素，移动到新节点

Step 3: 分裂点的键被提升(promote/push up)到父一级

Step 4: 插入要插入的节点

Step 5: 生成额外指针，指向新裂变的节点

B+树的节点插入和分裂（非叶节点分裂）



Step 1: 创造新节点，如没有空位， 则对父节点继续分裂

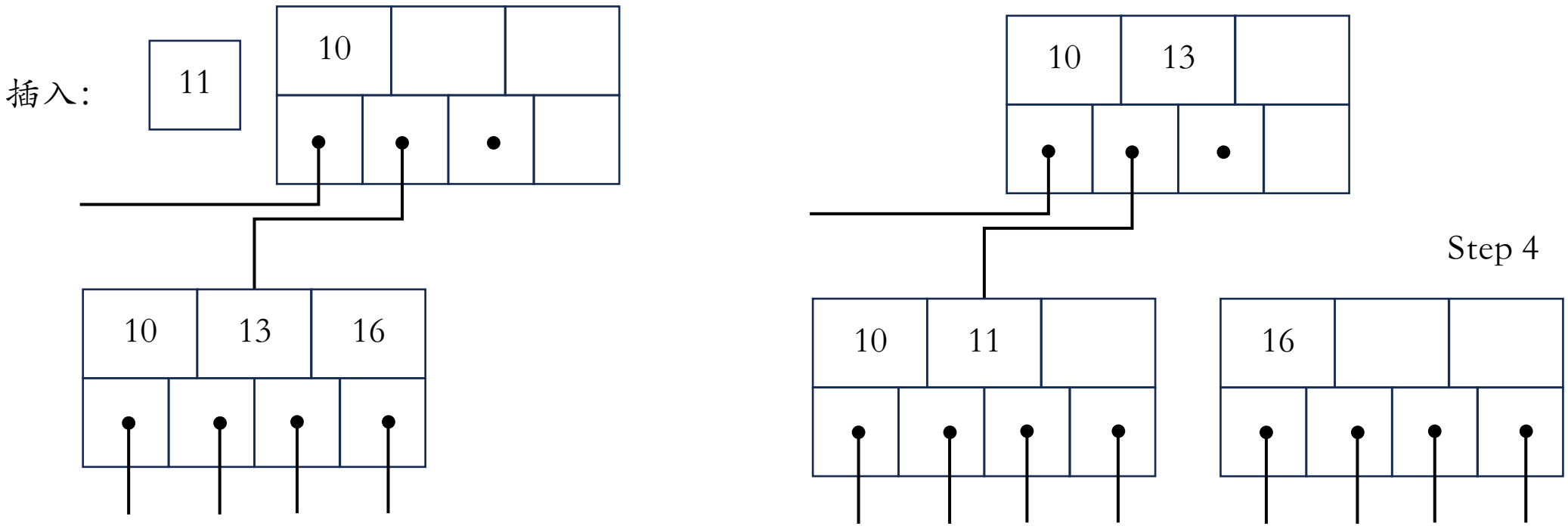
Step 2: 将原节点下标为⌈(m-1)/2⌉以后的元素，移动到新节点

Step 3: 分裂点的键被提升(promote/push up)到父一级

Step 4: 插入要插入的节点

Step 5: 生成额外指针，指向新裂变的节点

B+树的节点插入和分裂（非叶节点分裂）



Step 1: 创造新节点，如没有空位， 则对父节点继续分裂

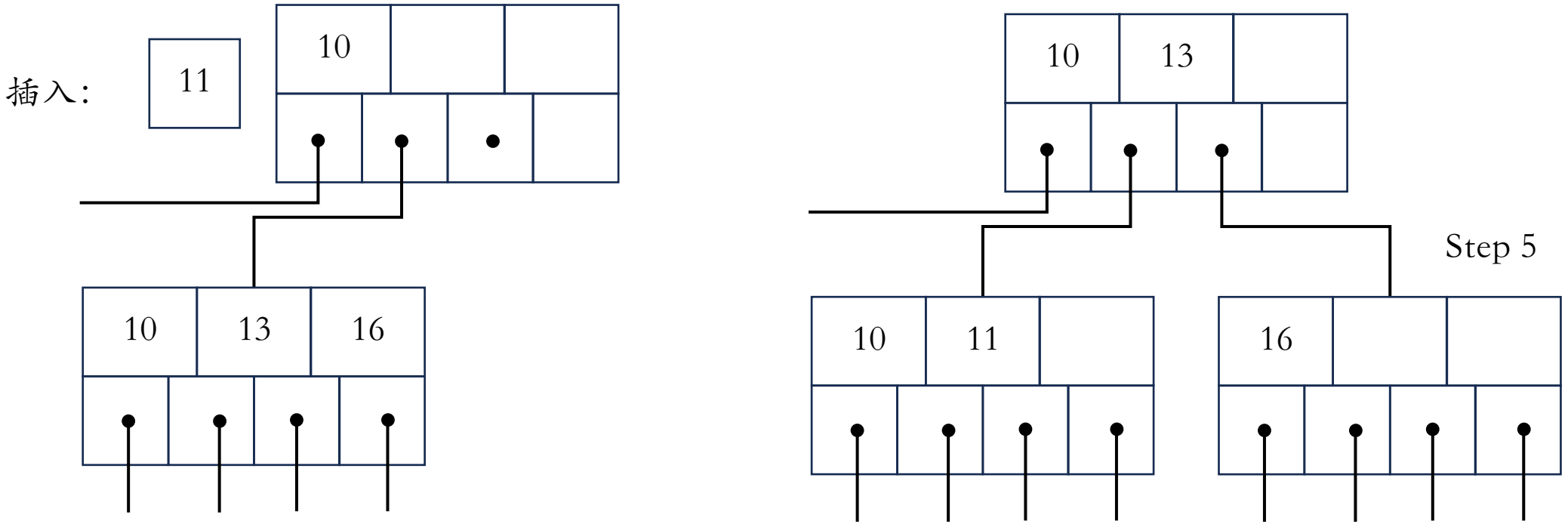
Step 2: 将原节点下标为⌈(m-1)/2⌉以后的元素，移动到新节点

Step 3: 分裂点的键被提升(promote/push up)到父一级

Step 4: 插入要插入的节点

Step 5: 生成额外指针，指向新裂变的节点

B+树的节点插入和分裂（非叶节点分裂）



Step 1: 创造新节点，如没有空位， 则对父节点继续分裂

Step 2: 将原节点下标为⌈(m-1)/2⌉以后的元素，移动到新节点

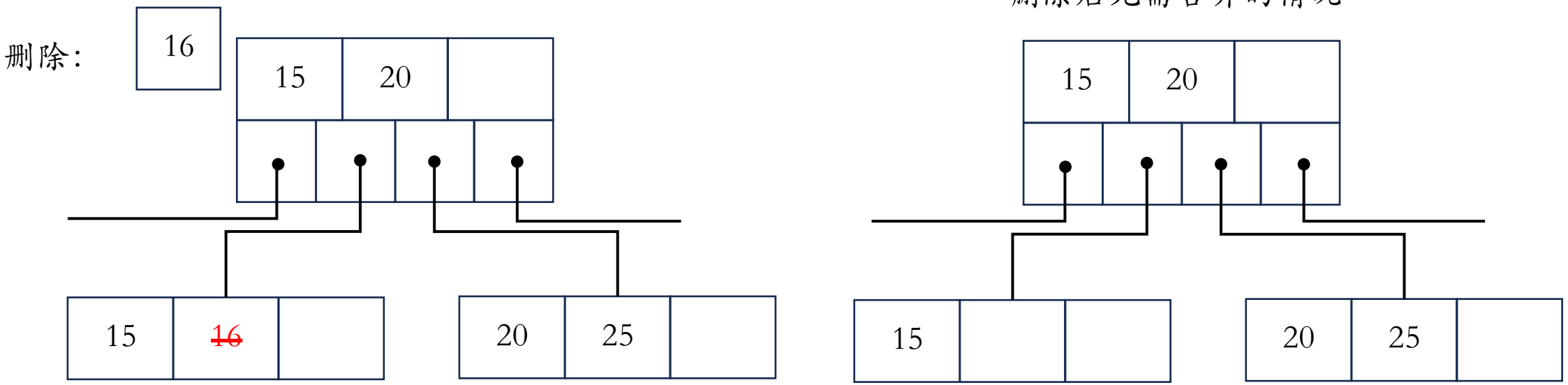
Step 3: 分裂点的键被提升(promote/push up)到父一级

Step 4: 插入要插入的节点

Step 5: 生成额外指针，指向新裂变的节点

B树的节点删除和合并（叶节点合并）

删除后无需合并的情况



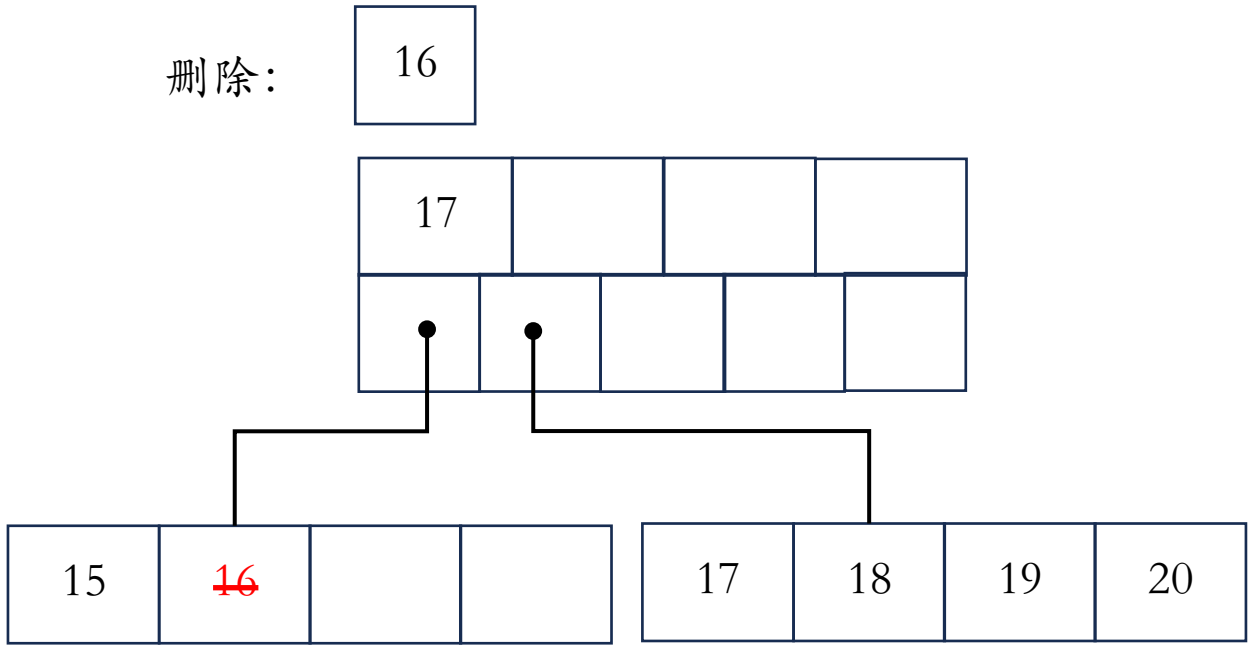
节点合并的判定条件： 节点最大容纳m-1个键值对，m个指针

对于叶节点：两个相邻节点中的键值对数量 小于或等于 m-1

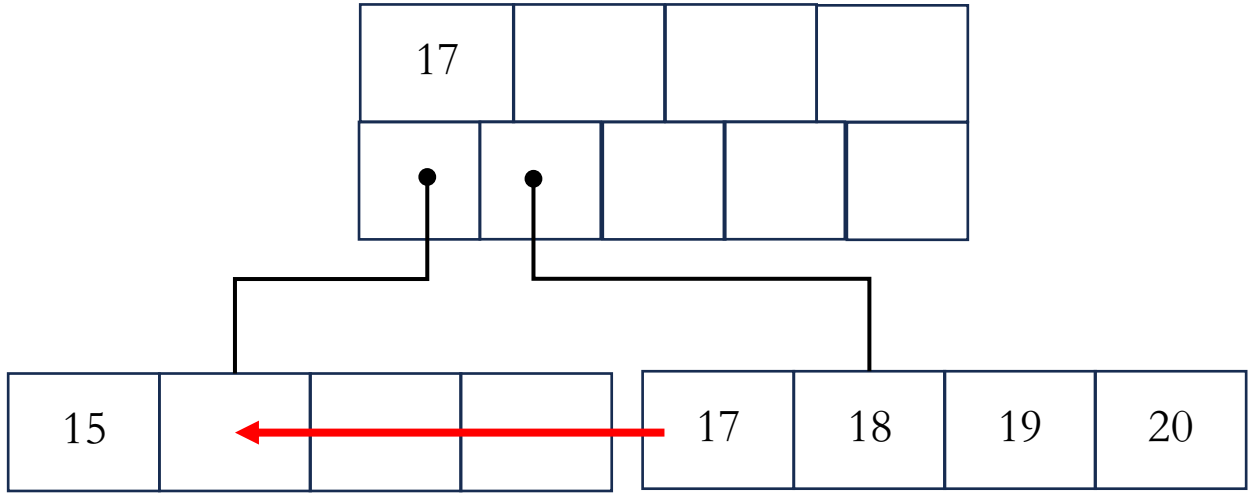
对于非叶节点：两个相邻节点中指针的数量 小于或等于 m

一般50%是树状结构节点占用率的阈值，这里我们就用⌈(m-1)/2⌉了

B树的节点删除和合并（叶节点合并）



删除后需要“借”的情况



节点合并的判定条件： 节点最大容纳m-1个键值对，m个指针

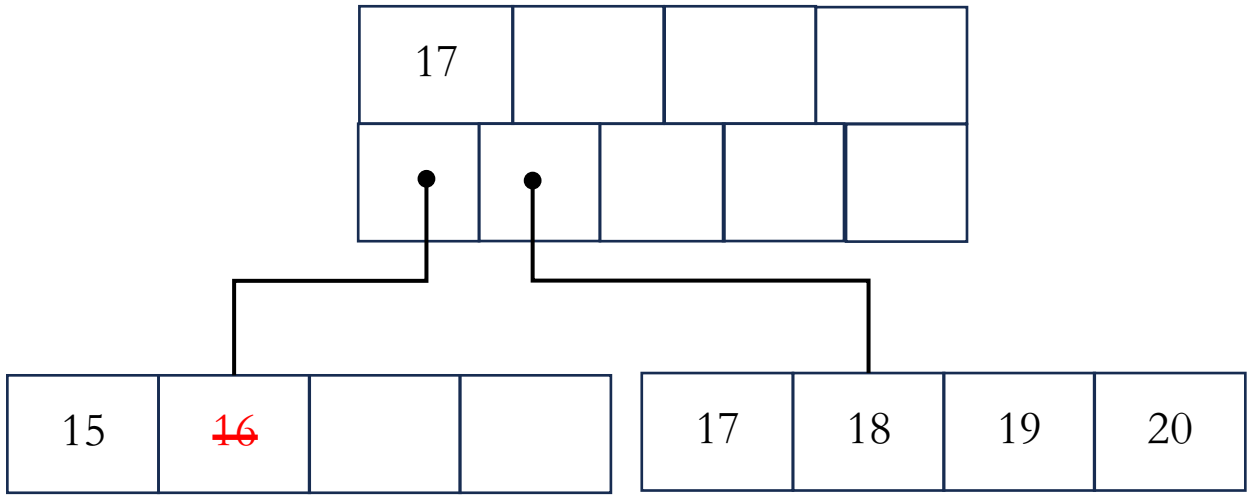
对于叶节点：两个相邻节点中的键值对数量 小于或等于 m-1

对于非叶节点：两个相邻节点中指针的数量 小于或等于 m

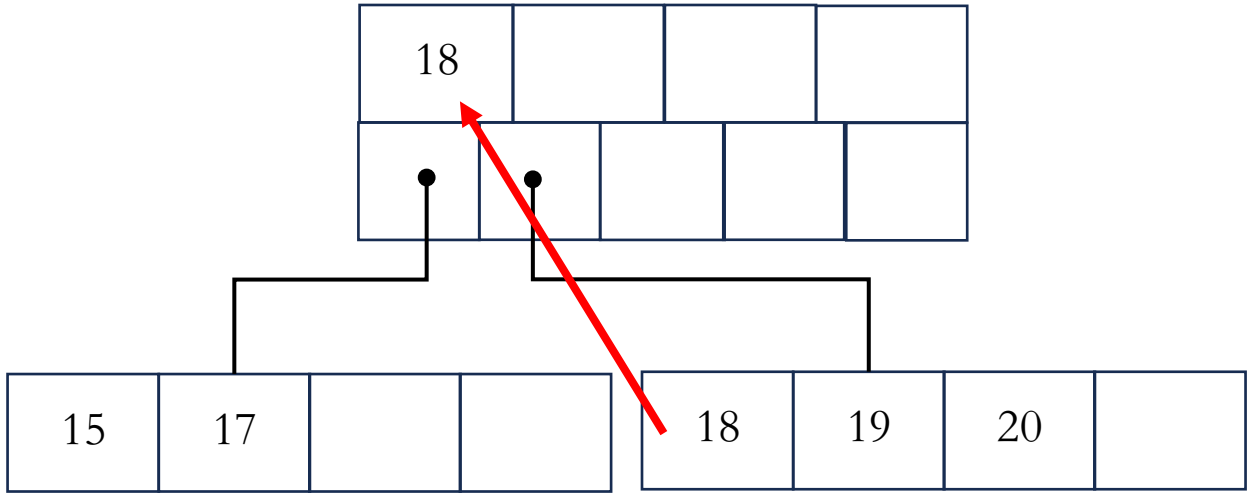
一般50%是树状结构节点占用率的阈值，这里我们就用⌈(m-1)/2⌉了

B树的节点删除和合并（叶节点合并）

删除：



从兄弟节点“借”，保持平衡



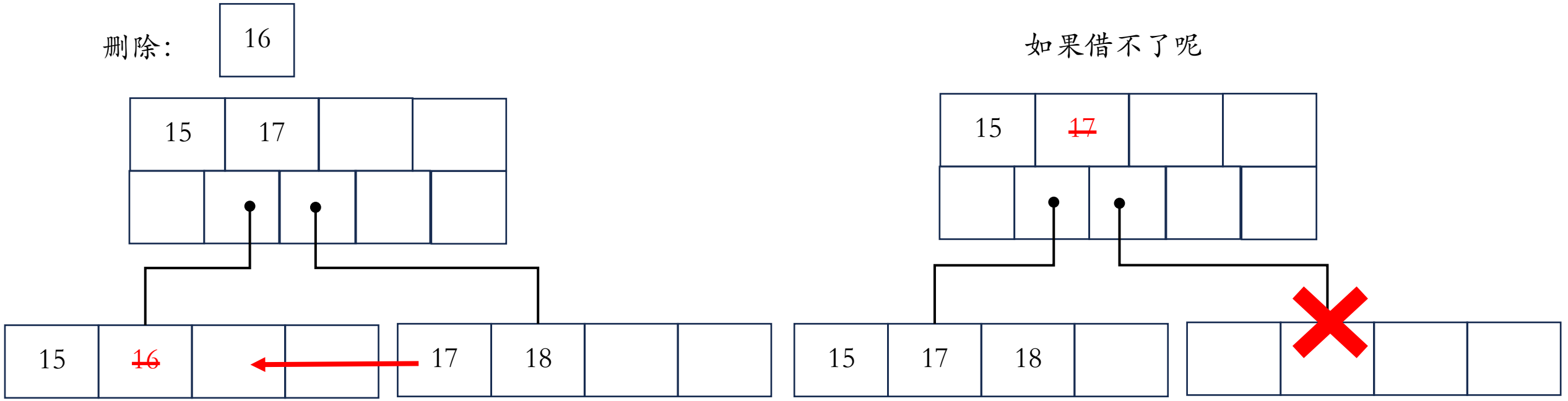
节点合并的判定条件： 节点最大容纳m-1个键值对，m个指针

对于叶节点：两个相邻节点中的键值对数量 小于或等于 m-1

对于非叶节点：两个相邻节点中指针的数量 小于或等于 m

一般50%是树状结构节点占用率的阈值，这里我们就用⌈(m-1)/2⌉了

B树的节点删除和合并（叶节点合并）

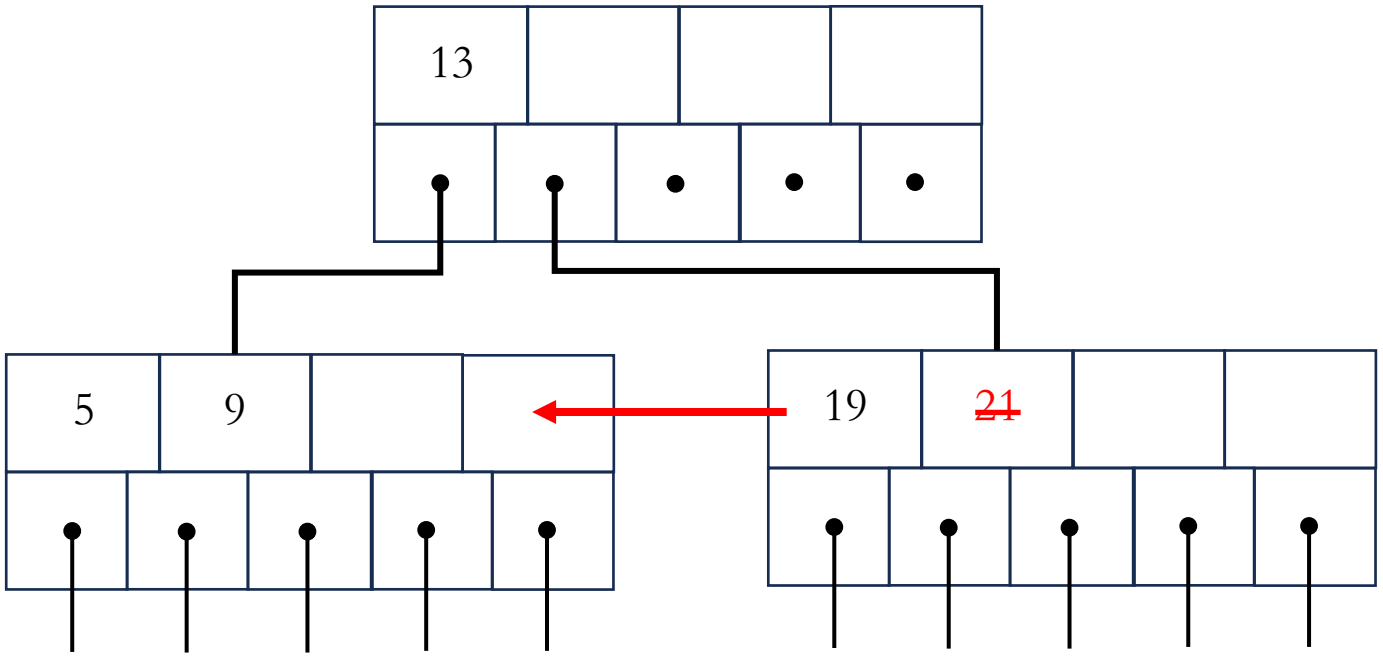


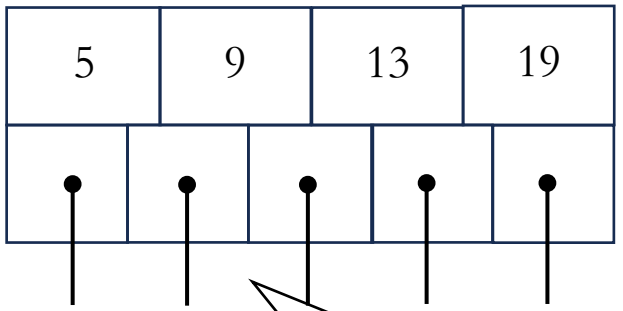
假设元素已经被删除，节点合并的步骤

Step 1:从右节点复制所有元素去左节点，删除右节点

Step 2:从父节点删除右节点指针（非叶节点则降级）

B树的节点合并（非叶节点合并）





为了减少分裂和合并的次数，DBMS可能会使用“再平衡”(redistribute)。

假设元素已经被删除，节点合并的步骤

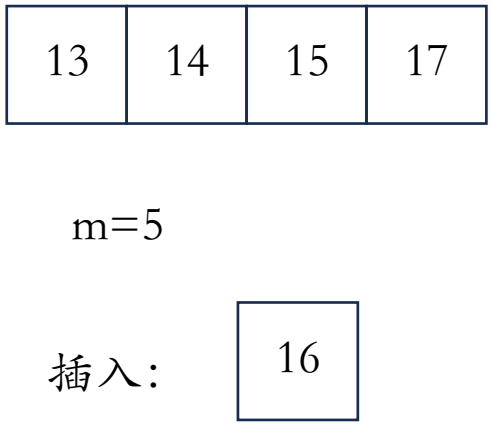
Step 1:从右节点复制所有元素去左节点，删除右节点

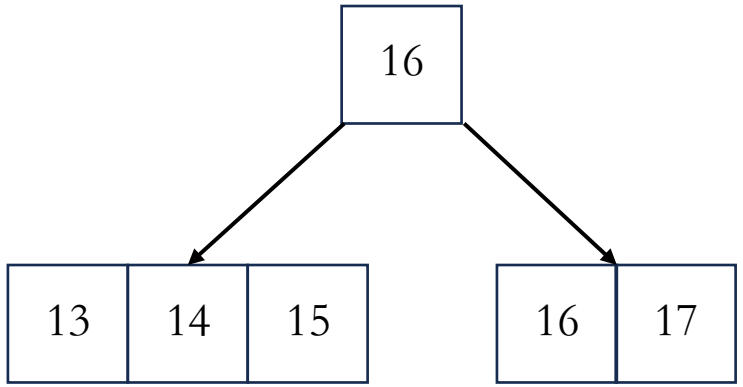
Step 2:从父节点删除右节点指针（非叶节点则降级）

参考：

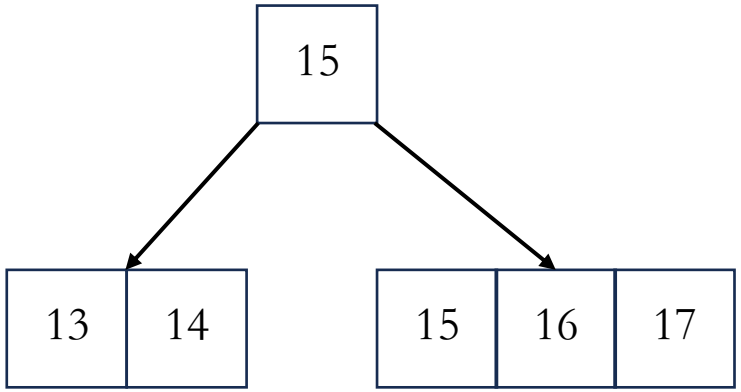
[1]: CMU15445/645,2024, https://15445.courses.cs.cmu.edu/fall2024/slides/08-indexes1.pdf page 34

B+树的插入删除解决了……吗?





①



②

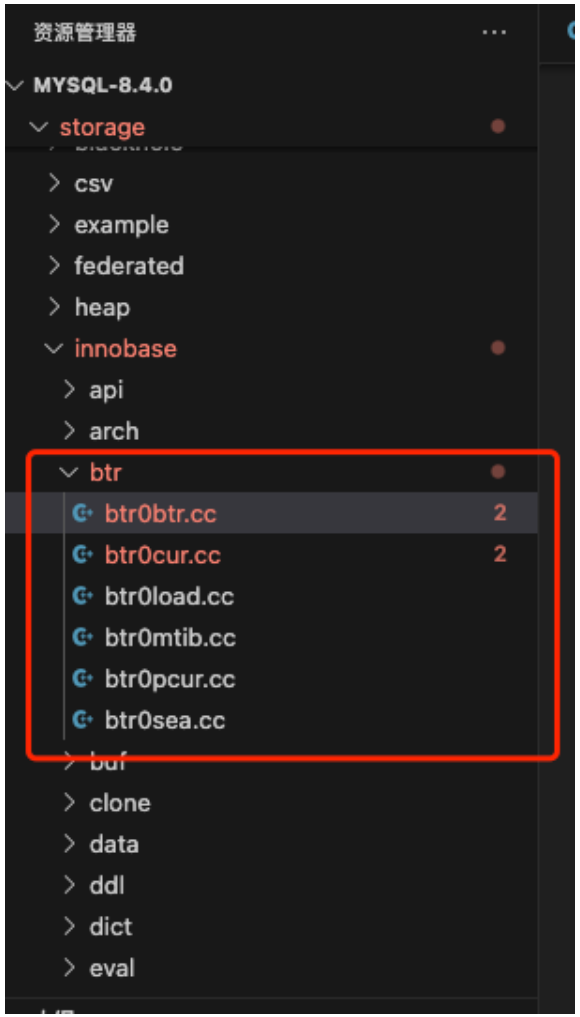
回过头来看，为什么是后者而不是前者，有什么区别吗?

很遗憾，目前没有答案

现实世界的B+树的插入删除——以MySQL Innodb为例

你是一位刚入职的数据库天命打工人

当你打开源代码，你会发现B+树操作比课上所演示的更加复杂。



参考：

[1]: MySQL,5.7.44, 8.4.0,, https://dev.mysql.com/downloads/mysql/, storage/innobase/btr

眼看喜 - 乐观插入操作

数据行插入的入口函数是row_ins_index_entry

悲观插入和乐观插入是根据row_ins_index_entry_low的第一个参数来判断的

传入参数为BTR_MODIFY_LEAF，表示只修改叶子节点，然后先执行乐观插入btr_cur_optimistic_insert 。

如果失败了则再次调用入口函数并传入参数为BTR_MODIFY_TREE，表示需要修改B-TREE，这时候会选择调用函数：btr_cur_pessimistic_insert

眼看喜 - 乐观插入操作

在进行插入时，系统会优先进行乐观插入。

btr_cur_optimistic_insert:

•检查页面剩余空间。如果页面中有足够的空闲空间，返回为DB_SUCCESS，则直接在指定位置插入，无需进行分裂操作

•如果可用空间不足，若返回为DB_FAIL，则可能进入btr_cur_pessimistic_insert。

•如果页面中只有一条记录，则必然成功，防止对单条记录进行分页

耳听怒 - 悲观插入与页面分裂

当乐观插入失败时，系统会进入悲观插入过程。悲观插入会尝试修改整个B+树的结构。

btr_cur_pessimistic_insert：

1.判断是否需要分裂：如果定位的cursor在当前页的尾部，先试图向右兄弟页插入，插入失败则进行分裂操作。

2.计算分裂点：通过判断页面记录的数量和位置，确定split_rec的位置。

3.分裂与插入：创建一个新页面并将部分记录移到新页面，然后在合适的位置插入新记录。

页面分裂流程：

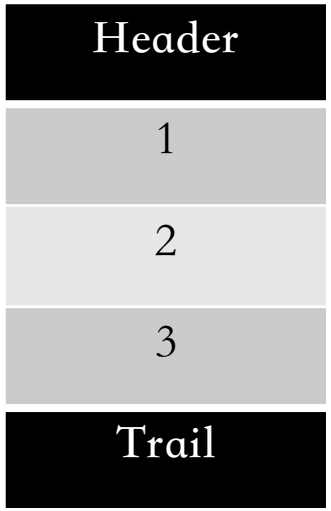
•根据insert_left的值，判断新记录应插入到分裂后的左页还是右页。

•如果插入失败，则对页面进行重组并重试插入。

耳听怒 - 悲观插入与页面分裂

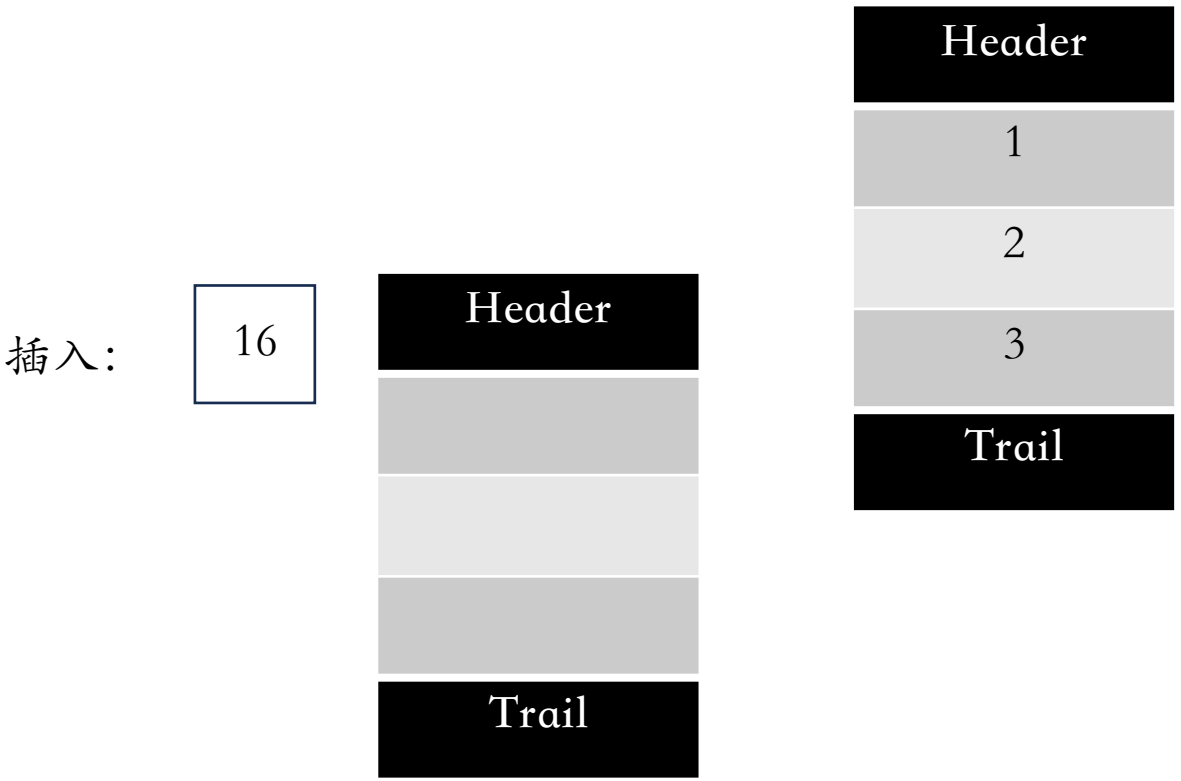
数据复制的过程不是剪切，而是先全量复制再删除，保证始终存有可供恢复的数据。特别是root页的分裂和提升，必须成功，否则无法回退。

插入：



耳听怒 - 悲观插入与页面分裂

数据复制的过程不是剪切，而是先全量复制再删除，保证始终存有可供恢复的数据。特别是root页的分裂和提升，必须成功，否则无法回退。

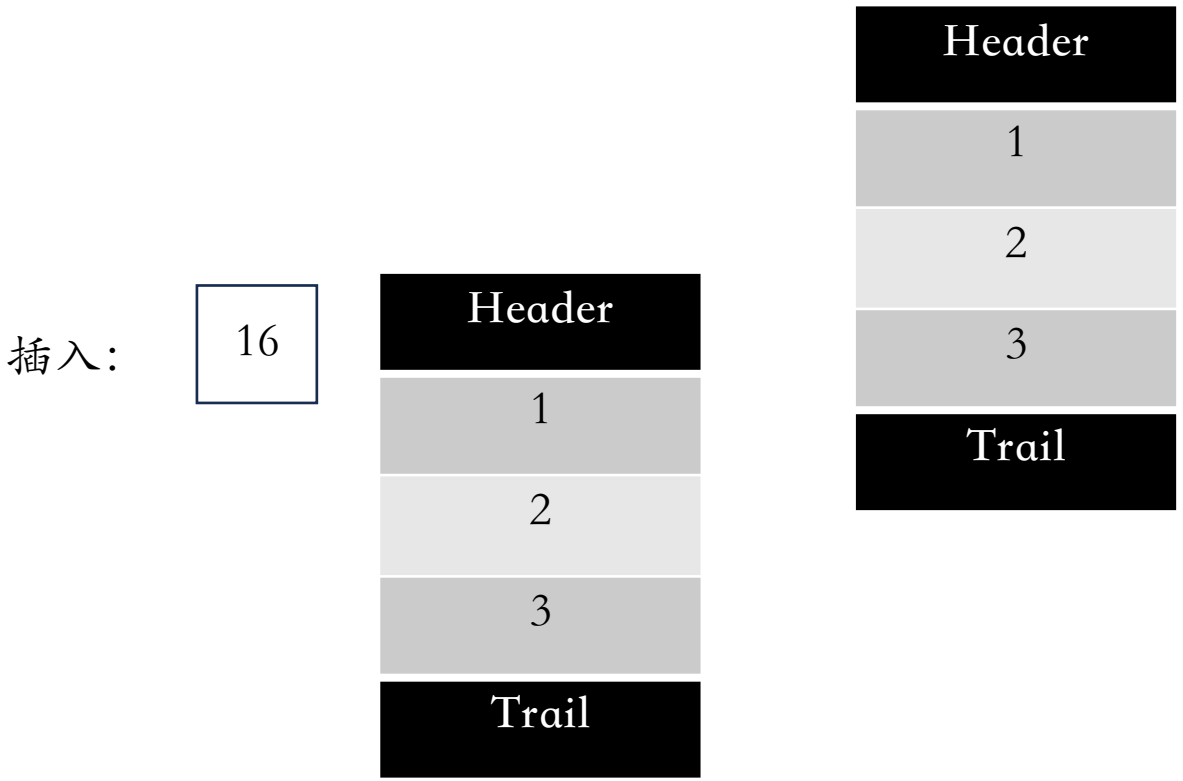


btr_page_alloc

btr_page_create

耳听怒 - 悲观插入与页面分裂

数据复制的过程不是剪切，而是先全量复制再删除，保证始终存有可供恢复的数据。特别是root页的分裂和提升，必须成功，否则无法回退。



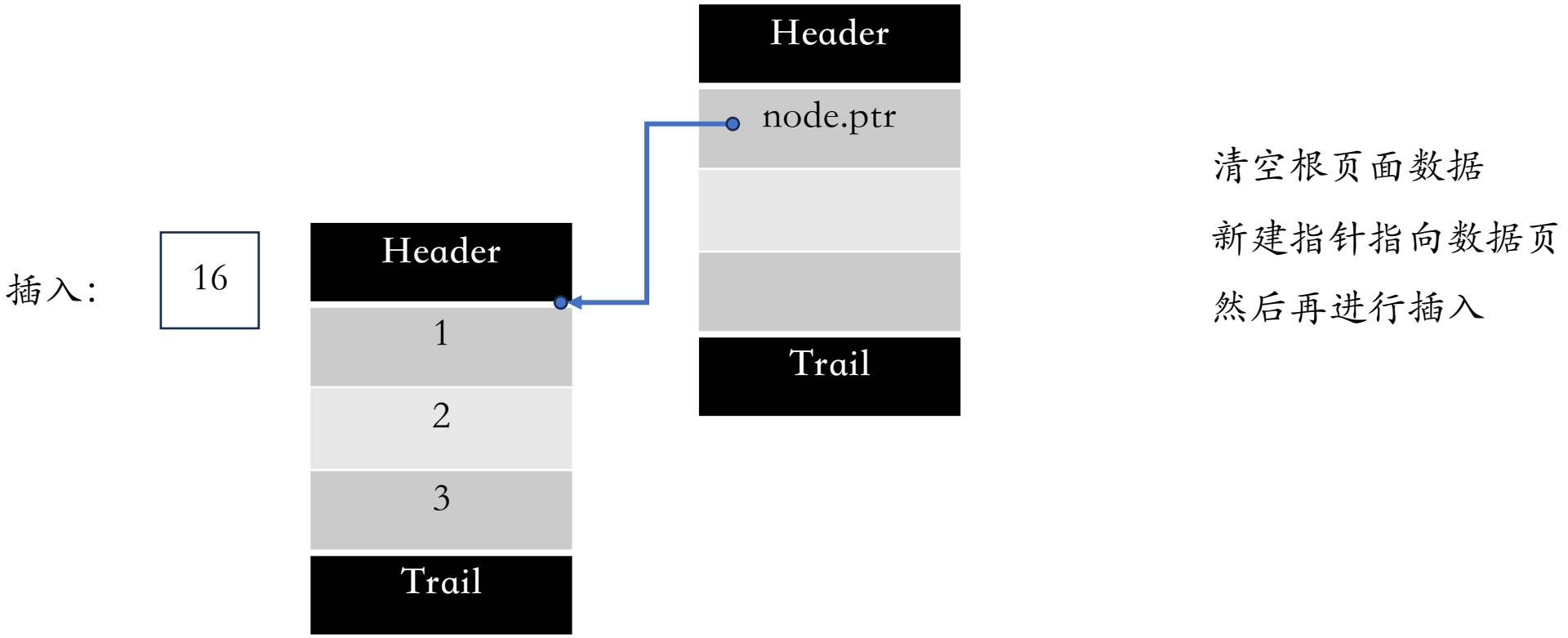
buf_block_get_frame

buf_block_get_page_zip

复制页面数据

耳听怒 - 悲观插入与页面分裂

数据复制的过程不是剪切，而是先全量复制再删除，保证始终存有可供恢复的数据。特别是root页的分裂和提升，必须成功，否则无法回退。



清空根页面数据

新建指针指向数据页

然后再进行插入

鼻嗅爱 - 页面分裂的优化策略

选择最佳的分裂点：

•页面分裂时，会根据页面中记录的数量和大小来选择合适的分裂点，通常是页面中间的记录（split_rec）。将页面记录尽量均匀地分布在左右两个页面中，减少未来操作中树的高度增加。

判断左右节点的分配：

•插入的新记录通过判断其与分裂点的位置关系，决定是分配到左页面还是右页面。

•btr_page_get_split_rec_to_left 判断新记录是否应该插入到左节点。如果新记录位置靠前，它会被分配左节点。

•btr_page_get_split_rec_to_right 判断新记录是否应该插入到右节点。如果新记录位置靠后，它会被分配到右节点。

顺序插入优化：

•当记录按顺序插入时，系统可以通过提前判断顺序来优化分裂，减少实际分裂发生的次数。顺序插入时，B+树在页面分裂时会尝试将更多记录移动到同一侧页面，从而减少不必要的分裂。

舌尝思 - 空闲空间管理

在进行插入时，系统会检查是否需要为未来的插入操作预留足够的空间。

btr_cur_optimistic_insert:

- 如果可用空间不足，或需要为将来的插入预留空间，则可能需要进行页面重组或分裂。

页面重组:

- 当页面中的碎片过多时，系统会尝试重组页面，以腾出更多的连续空间来插入新记录。这种重组操作通常用于延缓页面分裂，从而提高B+树的效率和稳定性。

空闲空间策略:

- 系统会根据BTR_CUR_PAGE_REORGANIZE_LIMIT（通常为页面大小的1/32）来判断是否进行页面重组，以确保未来的插入操作能够顺利进行。

身本忧 - B+树删除操作

删除操作可能导致页面合并，影响B+树的平衡和性能。

btr_cur_optimistic_delete：

执行乐观删除操作，确定删除不会使页面变的太空(too empty)且不需要压缩页面（no_compress_needed 为 true）就直接执行删除。否则还需检查页面的压缩状态，如果记录包含外部存储或需要页面压缩，导致乐观删除操作无法完成。此时预取页面的兄弟节点，以便进行更复杂的悲观删除操作。

btr_cur_pessimistic_delete：

•执行悲观删除操作，首先检查是否包含外部存储字段（LOB，如大字段）

•其次检查要删除的记录所在的页面，确定是否需要进行页面合并或者废弃页面

•删除操作还可能需要保留一些额外的磁盘空间来确保树结构的修改不会失败，这些空间会在删除操作完成后释放。

意见欲 - 页面合并与压缩

删除后页面合并与压缩操作可以提高B+树的效率，减少树的高度。

btr_compress：

•删除操作可能导致页面的稀疏，系统会在合适的条件下对页面进行压缩。

•如果左兄弟页面有足够的空间，系统会尝试将当前页面的记录合并到左兄弟页面中。

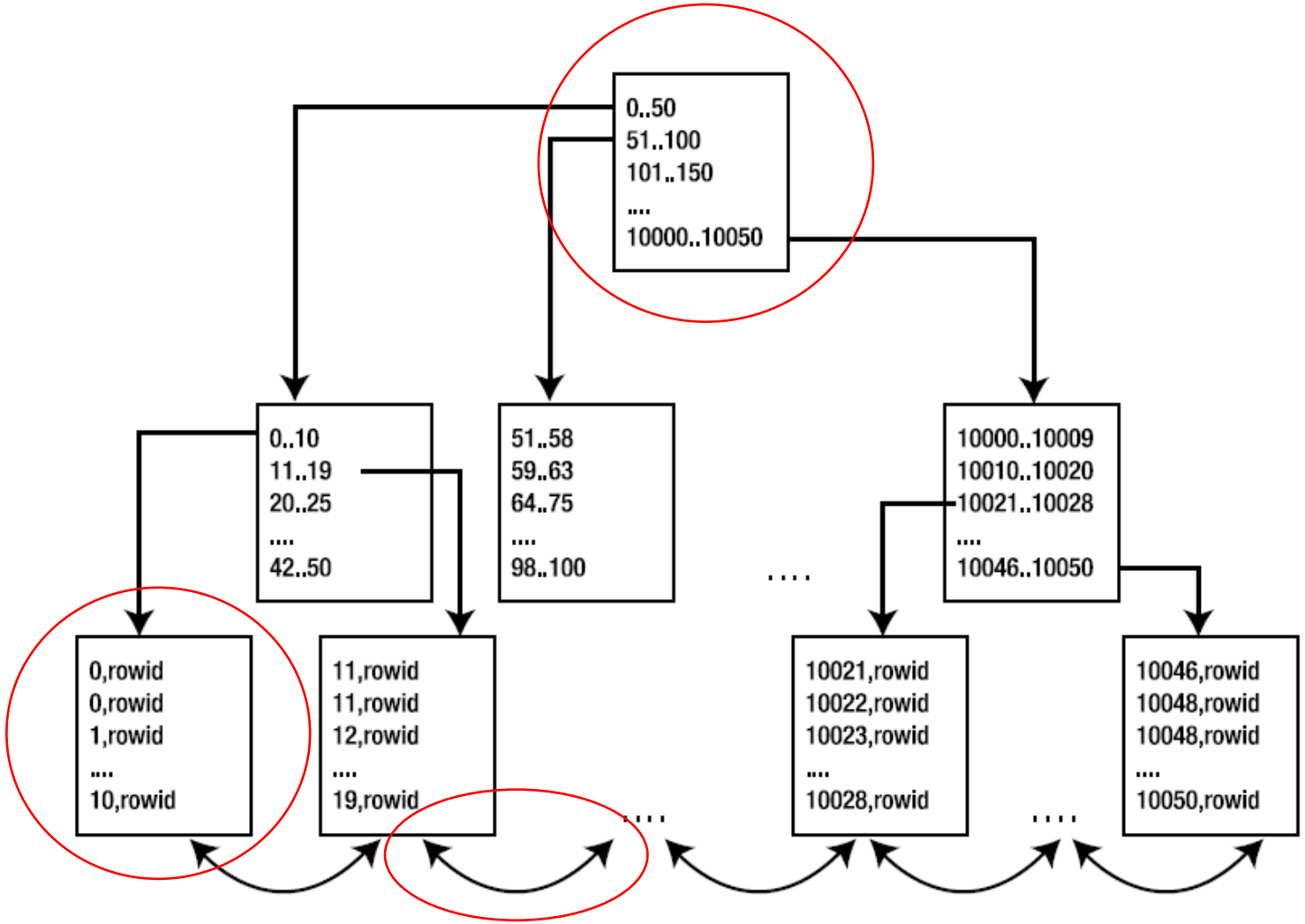
btr_cur_compress_if_useful：

- 删除后，系统会评估页面的使用率。如果页面的记录过少，则会尝试进行页面合并或压缩操作，以保持B+树的平衡。

页面合并条件：

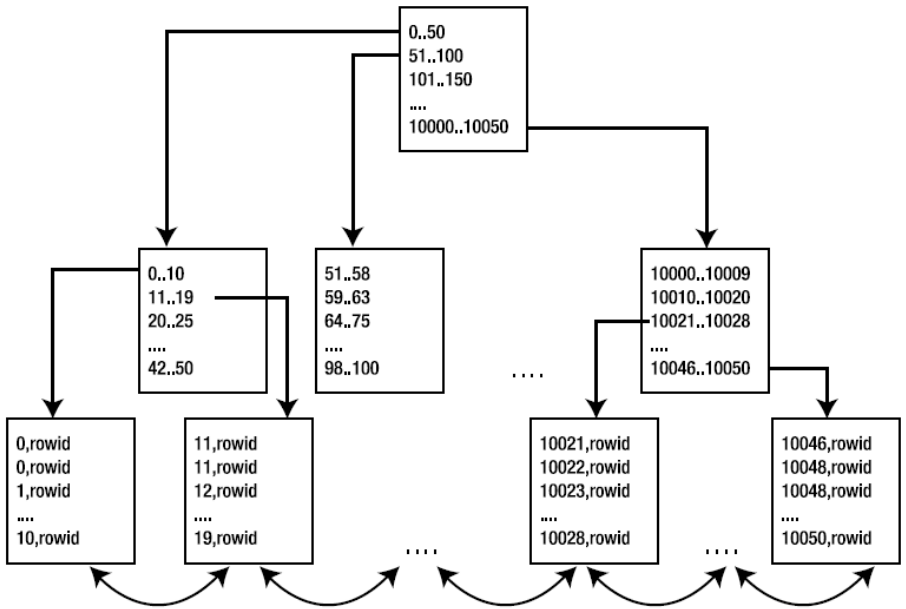
- 空间索引检查：如果当前索引是空间索引，则不会进行合并操作。
- 锁定状态检查：如果页面被锁定，则系统会推迟合并操作，直到合适的时机。
- 页面压缩：如果页面记录过少，系统会尝试压缩页面，确保空间利用效率。(btr_cur_compress_recommendation)

B树索引的使用

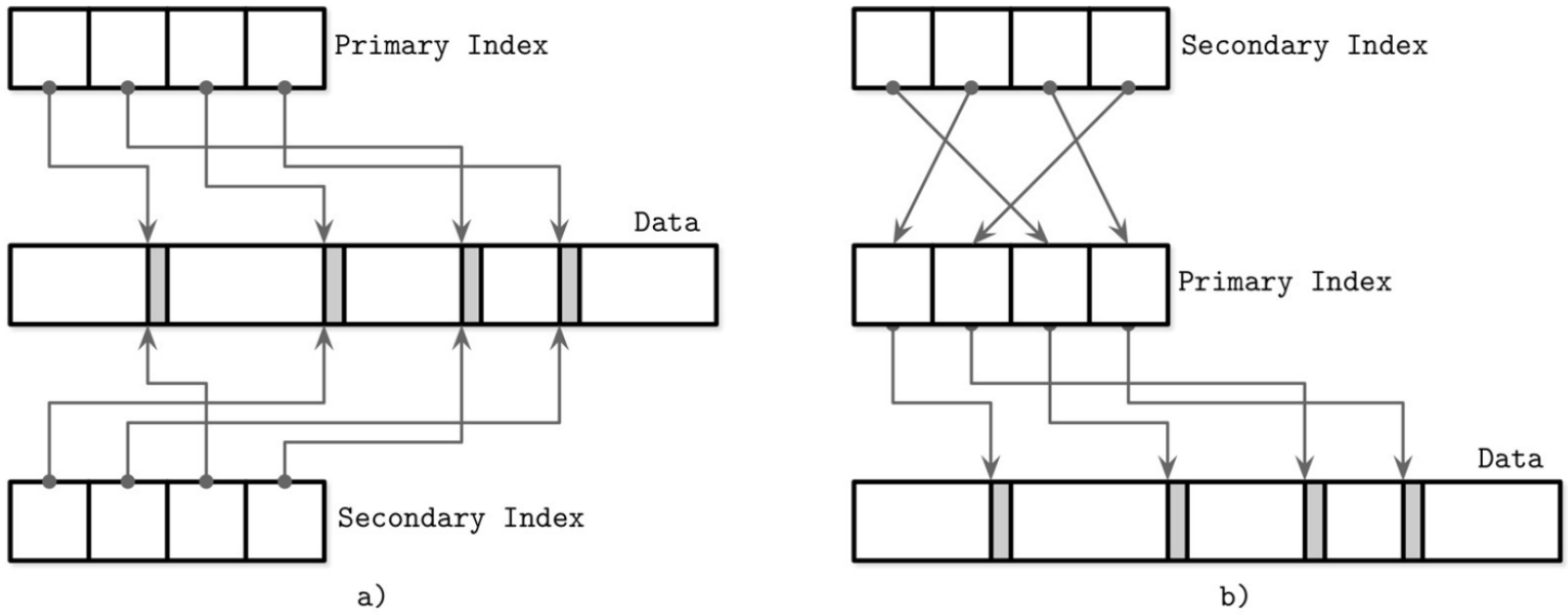

0..50
51..100
101..150
....
10000..10050
0..10
11..19
20..25
....
42..50
51..58
59..63
64..75
....
98..100
10000..10009
10010..10020
10021..10028
....
10046..10050
0,rowid
0,rowid
1,rowid
....
10,rowid
11,rowid
11,rowid
12,rowid
....
19,rowid
10021,rowid
10022,rowid
10023,rowid
....
10028,rowid
10046,rowid
10048,rowid
10048,rowid
....
10050,rowid

B+树索引能做什么？

- 充分理解B+树索引的结构，你就能充分B+树能做什么不能做什么
- 能做的
  - 全键值　　Where x=123
  - 键值范围　　Where 45<x<123
  - 键前缀查找　　where x LIKE ‘J%’
- 根据结构，请思考B+树索引不能做的有哪些？



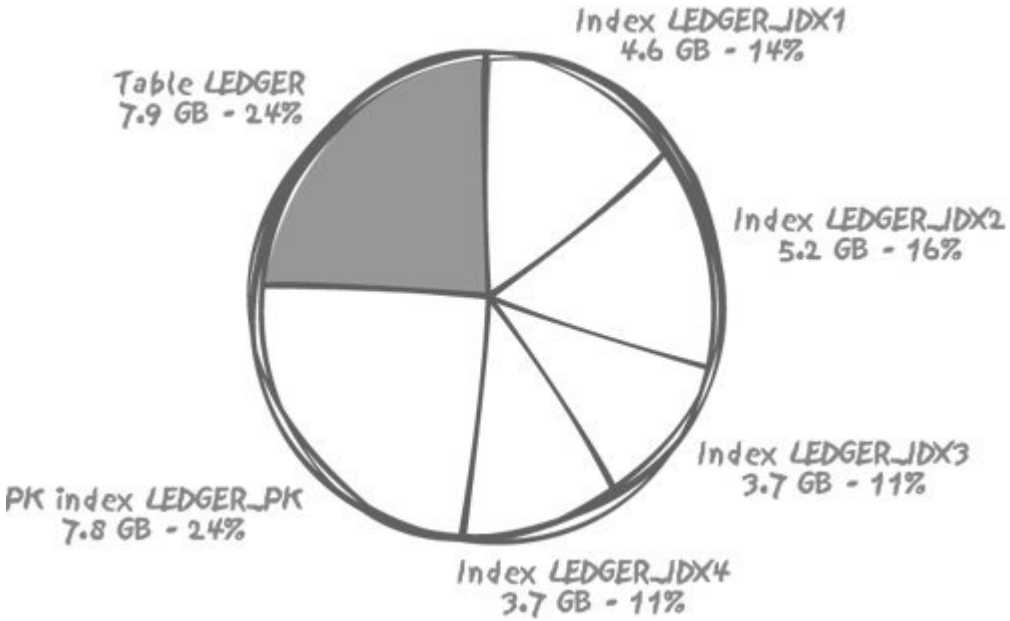
索引对数据的访问只是第一步



Referencing data tuples directly (a) versus using a primary index as indirection (b)

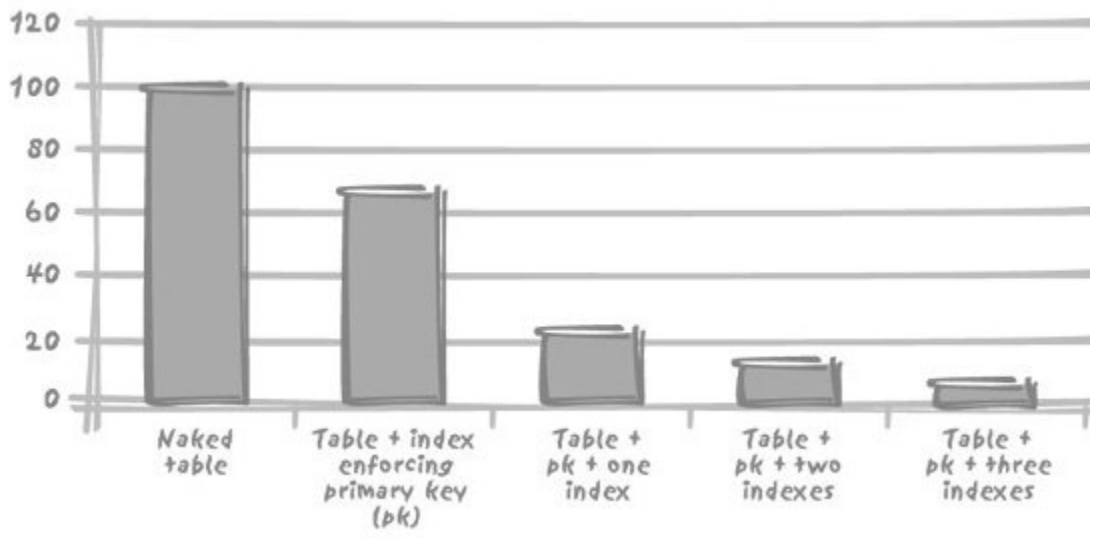
索引的另一面（问题）

- 磁盘空间的开销

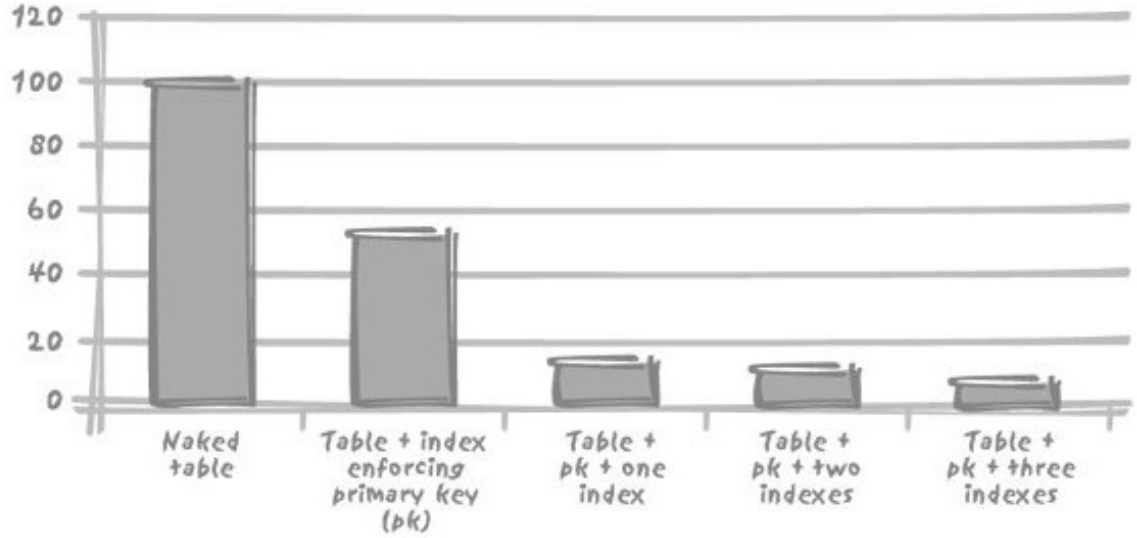


索引的另一面（问题）

- 磁盘空间的开销、处理的开销



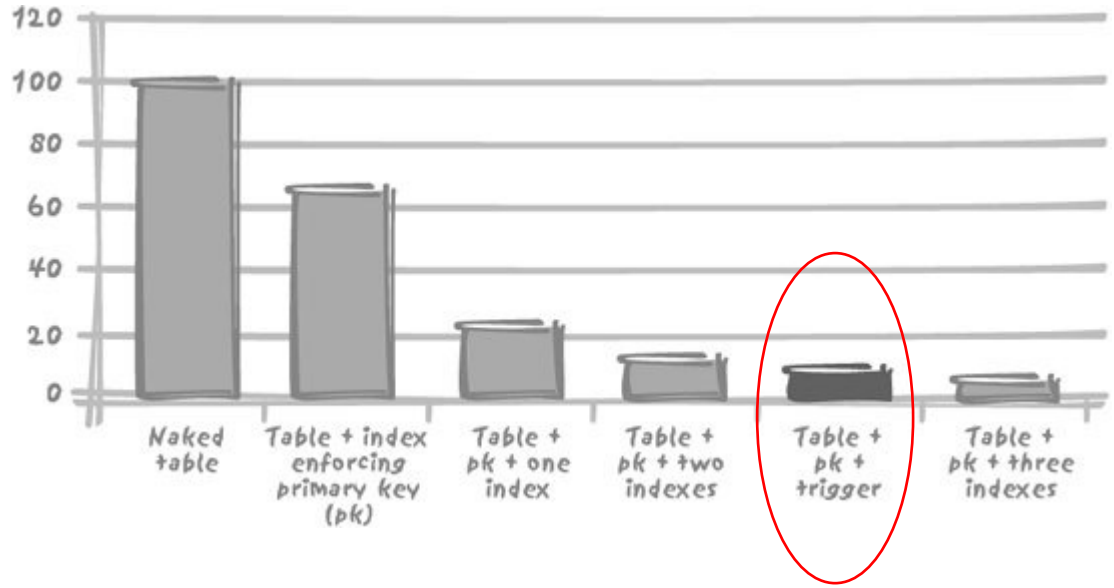
Oracle



MySQL

索引的另一面（问题）

- 数据库系统处理的开销



索引的另一面（问题）

- 那么不管怎么样，但，它至少能够提升查询效率不是吗？
- 思考题
  - 对于B+树索引，不少数据库都有自己的处理方式，比如，MySQL中不同的存储引擎使用了不同的方式把索引保存到磁盘上，他们会影响性能。
    - MyISAM使用前缀压缩以减少索引，而InnoDB不会压缩索引，（有啥差别？）
    - MyISAM索引按照行存储的物理位置引用被索引的行，但是InnDB按照主键值引用行，（有啥差别?）

索引有可能降低查询效率嘛?

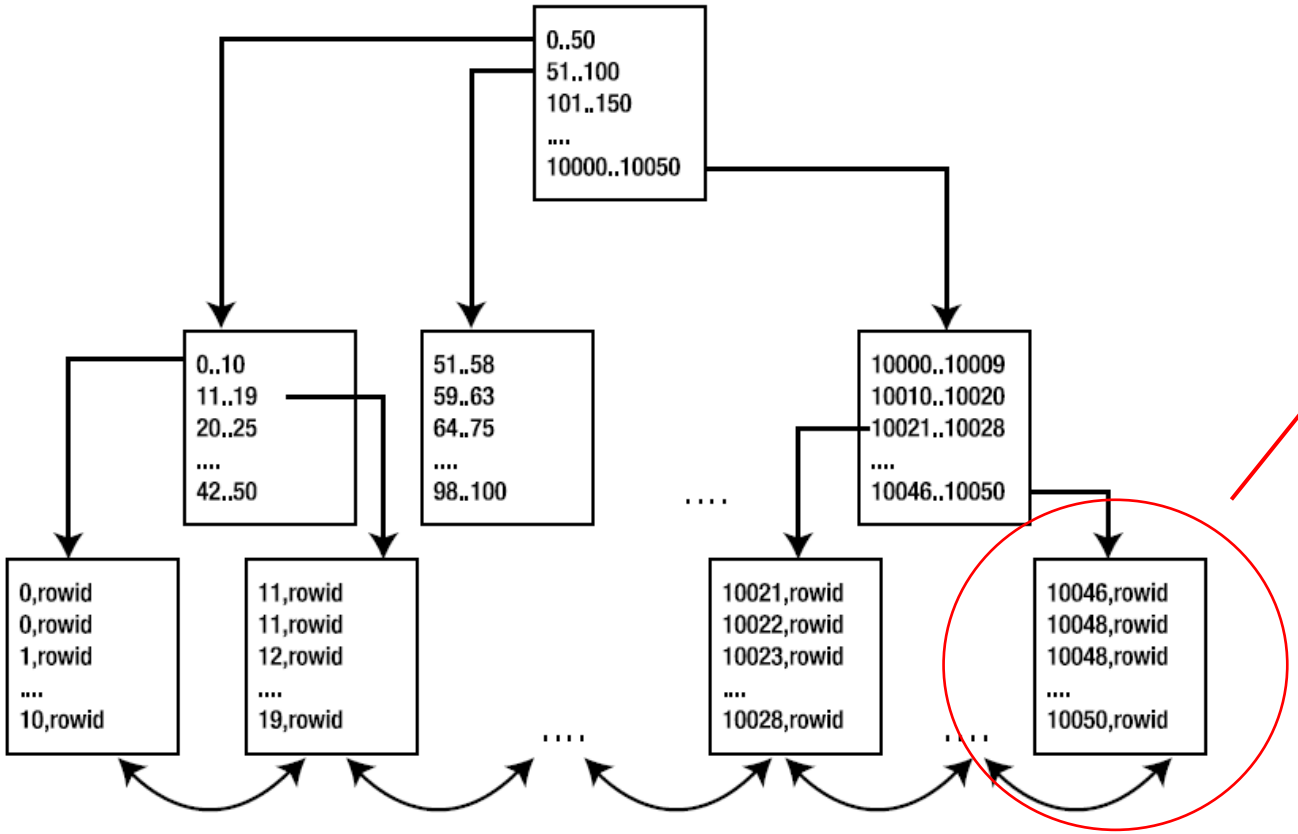
- 有可能……

- 索引和目录
  - 索引和目录是两种完全不同的机制
  - 索引是一种以原子粒度访问数据库的手段而不是为了检索大量数据的

让索引发挥作用

- 索引的使用是否合理，首先取决于它是否有用
- 判断索引适用性的依据是检索比例（retrieval ratios）
- 什么时候应该使用B树索引
  - 仅需要通过索引访问基本表的很少一部分行
  - 如果要处理表中的多行，可以使用索引而不使用表

只使用索引，不使用表



如果是Index（x，y）

10046,2345,rowid
10048,245 ,rowid
10048,290 ,rowid
10048,1356 ,rowed
……

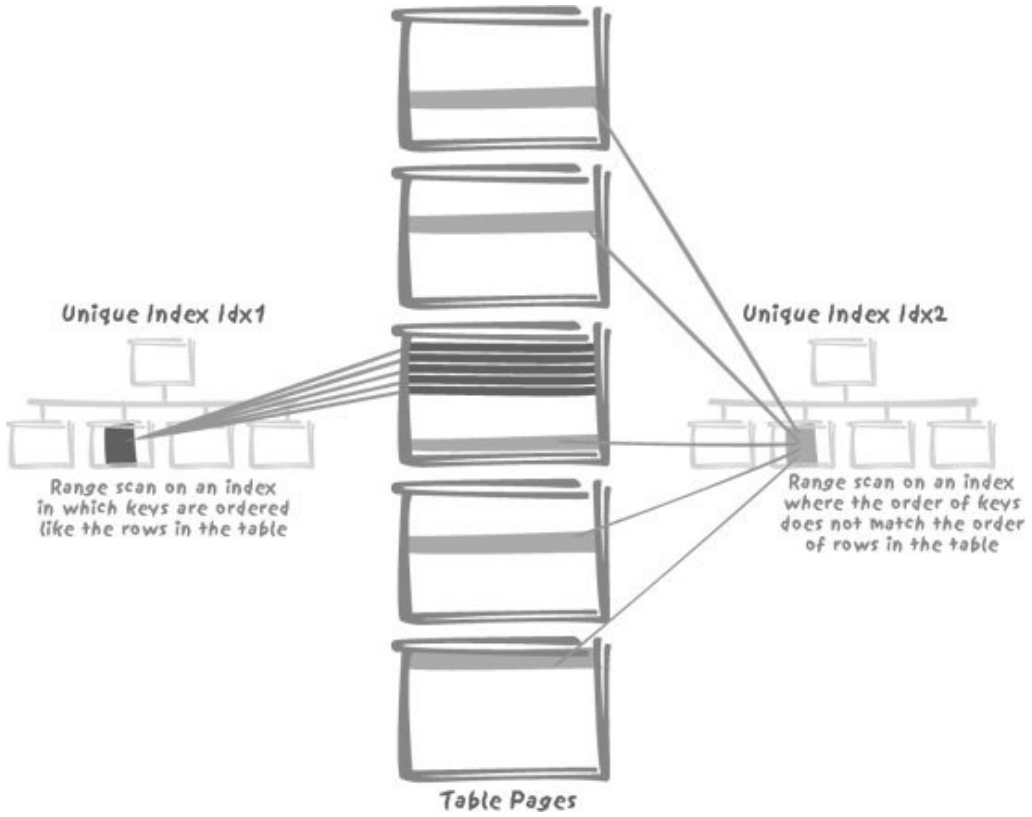
那么

Select x,y
From T
Where x…..

就可以只使用索引不使用表

复合键索引，本质上索引是按照排名第一的字段进行的索引

让索引发挥作用（cont’）



- 索引只是查询工作的第一步
- 读取基本表中的数据才是查询的结束
- 同样的索引，但不同的物理结构，可能会引起查询效率的千差万别
  - 磁盘访问的速率
  - 物理I/O很可能是内存访问
  - 记录存储

索引无论怎样，都是数据库的重要组成部分

- 索引始终是数据库中极重要的组成部分
  - 通用目的或事务处理型数据库系统
  - 决策支持系统
- 事务处理型数据库中“太多索引≈设计不够稳定”

Practice in class 2.1

- 既然，使用复合键索引，在select子句中，如果所有字段都在复合键索引所包括的字段之中的时候，查询可以只使用索引不使用表
- 那么，为什么不可以针对表T（x,y,z）这样的表，直接构建一个索引index（x,y,z），这样所有对这个表的访问就可以直接使用索引不使用表了，这会不会大幅度地提升查询效率呢?

介绍几个其它类型的索引

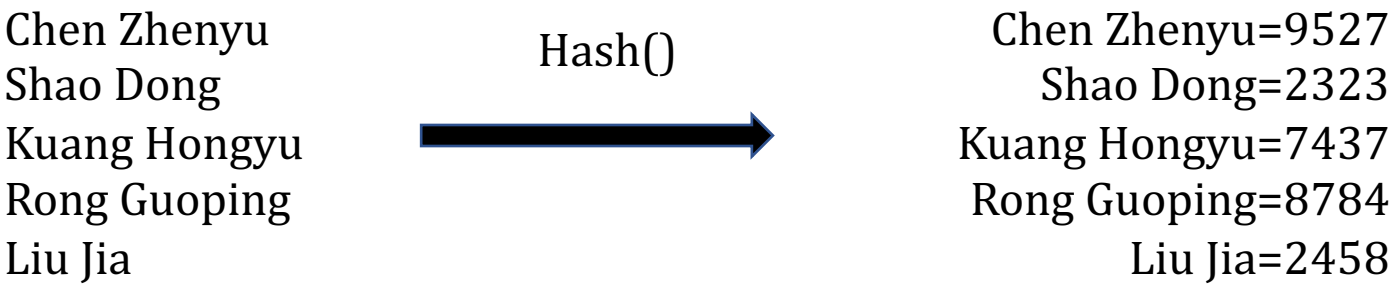
- 哈希索引（Hash Index）： MySQL

- 位图索引（Bitmap Index）： Oracle
- 位图联结索引（Bitmap join index）:Oracle

- 函数索引（function-based index）

哈希索引

- 哈希索引结构：



- 根据结构，你能告诉我哈希索引能做什么不能做什么？
  - 碰撞率的问题、自适应哈希索引
  - 链式哈希表、哈希函数（直接定址、除留余数法、平方取中）
  - 哈希冲突的处理方法——闭散列、开散列（Separate Chaining 、 Open Addressing）

差别是啥？应用在什么地方？

位图索引

- Bitmap index，Oracle7.3引入，位数据库仓库查询环境设计
- 位图索引的结构

Table 11-6. *Representation of How Oracle Would Store the* JOB-IDX *Bitmap Index*

Value/Row	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ANALYST	0	0	0	0	0	0	0	1	0	1	0	0	1	0
CLERK	1	0	0	0	0	0	0	0	0	0	1	1	0	1
MANAGER	0	0	0	1	0	1	1	0	0	0	0	0	0	0
PRESIDENT	0	0	0	0	0	0	0	0	1	0	0	0	0	0
SALESMAN	0	1	1	0	1	0	0	0	0	0	0	0	0	0

Select count(*) from emp where job =’CLERK’ or job = ‘MANAGER’

Select * from emp where job =’CLERK’ or job = ‘MANAGER’

什么时候该使用位图索引

- 相异基数（distinct cardinality）低
- 大量临时查询的聚合

假设你有一个很大的表，T(gender, location, agegroup)
Gender M F
Location 1-50
Agegroup 18 and under 19-25 26-30 31-40 41 and over
而你又必须支持大量临时查询

```
Select count(*)
From t
Where gender = ‘M’  And location in (1, 10,30)
  And agegroup = ’41 and over’
```

```
Select *
From t
Where (  (gender = ‘M’ and location =20)
     Or (gender = ‘F’ and location =22)
  And agegroup = ’18 and under’;
```

```
Select count(*) from t where location in (11,20,30);
Select count(*) from t where agegroup = ’41 and over’ and gender = ‘F’;
```

位图联结索引（bitmap join index）

- 允许使用另外某个表的列对一个给定表建立索引。实际上，这就是允许对一个索引结构（而不是表本身）中的数据进行逆规范化。

Emp(empid,deptno) Dept(Deptno, Dname)

```
Select count(*)
From emp,dept
Where emp.deptno = dept.deptno
And dept.dname = 'SALES'

Select emp.*
From emp,dept
Where emp.deptno = dept.deptno
And dept.dname = 'SALES'
```

```
Create bitmap index emp_bm_idx on emp(d.dname)
From emp e, dept d
Where e.deptno = d.deptno
```

MySQL怎么办?

- MySQL没有位图索引，1）优化替代索引组合；2）低选择性添加特殊索引
- Select * from profiles where sex = ‘M’ order by rating limit 10;
  - 可以添加sex ，rating列上的复合索引。
- select * from profiles where sex = ‘M’ order by rating limit 100000, 10;
  - 依旧很慢，更好的策略是限制用户查看的页数
  - 也可以:

```
Select * from t inner join (
   Select id from t
   Where x.sex = ‘m’ order by rating limit 100000, 10
)AS x USING id;
```

函数索引

- 函数索引，对F(x)的值构建索引，在通过对索引读取x所指向的记录行
  - X索引，和F(x)的索引完全不一样
- 想一想，函数索引能用在哪?
  - 不区分大小写的查询
  - T、F的巨大差异下的索引
  - 有选择的唯一性

Creat index emp_upper_idx on emp(upper (ename))
Select * from emp where upper(name) = 'KING'

Create unique index active_project_must_be_unique on projects(case when status = 'ACTIVE' then name end)

还有很多的其它索引，需要自己学习

- 首先，现看索引的结构，从结构-能做什么-不能做什么-练习，再循环

- 思考题
  - 请尝试，构建一个本课相似的例子（比如本课程的例子、电脑的配置的例子等等）插入大量数据，在MySQL上，尝试用B树索引模拟位图索引的功能。
  - 请再想想，还有什么场景下可以使用函数索引或者哈希索引?

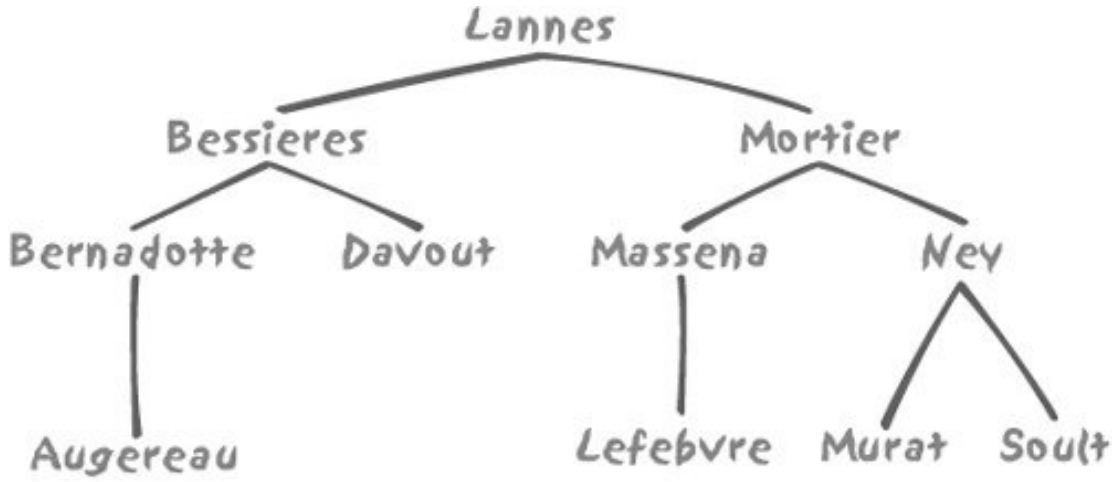
索引使用的典型问题

- 函数和类型转化对索引的影响
- 索引和外键
- 同一个字段，多个索引
- 系统生成键
- 总结，为什么没有使用我的索引?

函数和类型转换对索引的影响

- Where f (indexed_col) = ‘ some value‘
- 这种检索条件会使索引无法发挥作用
  - 日期函数
  - 隐式类型转换

字符串和日期的例子



where name = 'MASSENA'

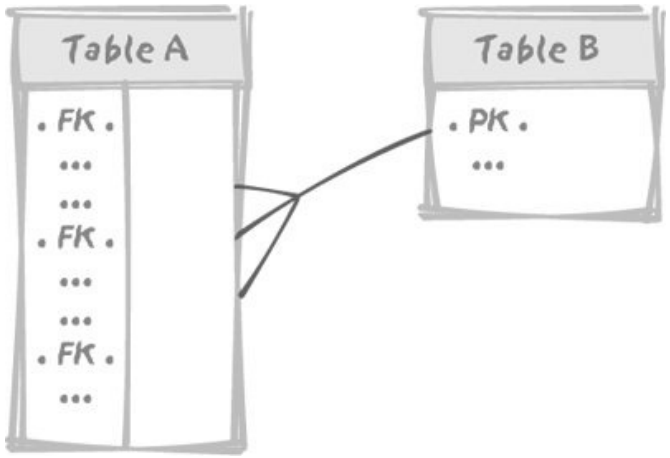
where substr(name, 3, 1) = 'R'

where date_entered = to_date('18-JUN-1815', 'DD-MON-YYYY')

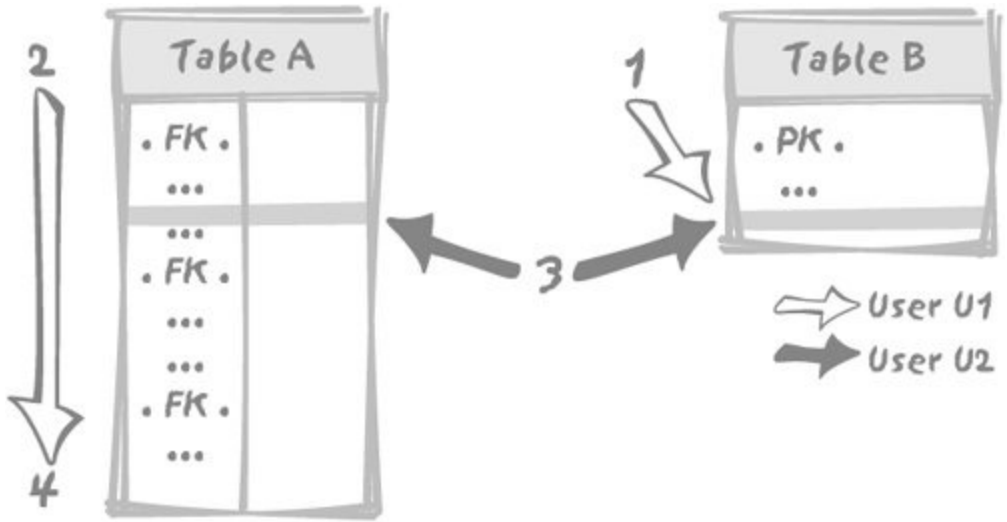
where trunc(date_entered) = to_date('18-JUN-1815', 'DD-MON-YYYY')

where date_entered >= to_date('18-JUN-1815', 'DD-MON-YYYY') and
date_entered < to_date('19-JUN-1815', 'DD-MON-YYYY')

索引与外键



- 系统地对表的外键加上索引的做法非常普遍
  - *但是为什么呢?*
  - *有例外吗?*
- 建立索引必须有理由
  - 无论是对外键，或是其他字段都是如此



同一字段，多个索引

- 如果系统为外键自动增加索引，常常会导致同一字段属于多个索引的情况

Order_Details (Order_id,article_id)

Order_Details (article_id,Order_id,)

当不需要为Article_id构建索引的时候，会引入额外索引

- 为每个外键建立索引，可能会导致多余索引

系统生成键

- 系统生产序列号，远好于
  - 寻找当前最大值并加1
  - 用一个专用表保存”下一个值“且加锁更新
- 但如果插入并发性过高，在主键索引的创建操作上会发生十分严重的资源竞争
- 解决方案
  - 反向键索引或叫逆向索引（reverse index）
  - 哈希索引（hash indexing）

为什么没有使用我的索引?

- 情况1：我们在使用B+树索引，而且谓词中没有使用索引的最前列
  - T，T(X,Y)上有索引，做SELECT * FROM T WHERE Y=5
- 跳跃式索引（仅CBO）

为什么没有使用我的索引？（cont’）

- 情况2：使用SELECT COUNT(*) FROM T，而且T上有索引，但是优化器仍然全表扫描
- 情况3：对于一个有索引的列作出函数查询
  - Select * from t where f(indexed_col) = value
- 情况4：隐形函数查询

为什么没有使用我的索引？（cont’）

- 情况5：此时如果用了索引，实际反而会更慢
- 情况6：没有正确的统计信息，造成CBO无法做出正确的选择
- 总结：归根到底，不使用索引的通常愿意就是“不能使用索引，使用索引会返回不正确的结果”，或者“不该使用索引，如果使用了索引就会变得更慢”

总结：索引访问的不同特点

- “查询使用了索引就万事大吉了”—误解啊～～
- 索引只是访问数据的一种方式
- “通过索引定位记录”只是查询工作的一部分
- 优化器有更多的选择权利
- 总结：索引不是万灵药。充分理解要处理的数据，做出合理的判断，才能获得高效方案

Practice in class 2.2

- 请研究你手上使用的数据库，比如，MySQL or Oracle，请研究数据库管理系统提供的其它索引形式，并阅读相关的文档