B+树用于实现DBMS中表的索引

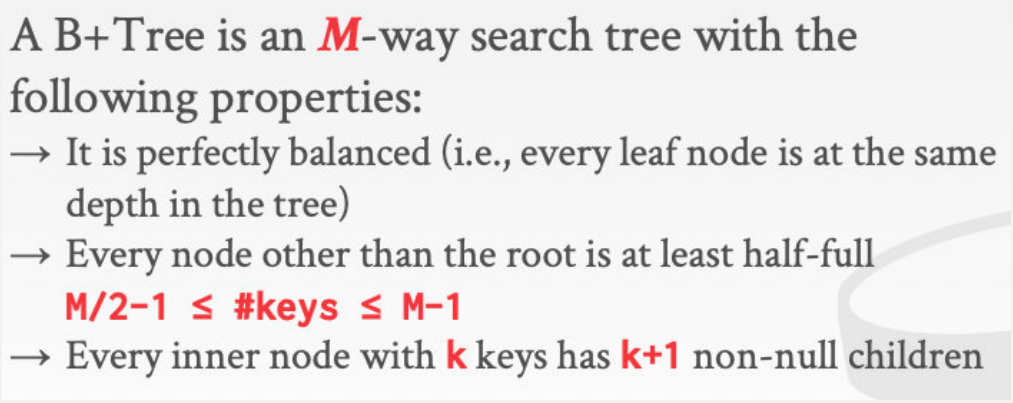
什么是索引呢？索引的本质就是从数据库中的完整的表中抽取出部分字段（attribute）构建成的小表，是大表的一个副本，比如说数据库中有存储公司所有员工信息的一个表，我们想构建员工年龄的索引，只需把id和年龄这两个字段拿出来构建一个小表，也就是一个副本，而且这个副本被构建出来的时候一般都是已经排好序的，我们使用这个副本就可以完成高效的数据访问，DBMS还会保证索引中的数据和数据库中的完整的表的数据是同步的，用户对后者的修改会被同步到前者当中

索引存在的本质就是加速数据库的查询的执行，还是上面的例子，当我们想查询公司里年龄为30的员工有哪些时，我们可以直接去基于年龄构建的索引中查询，而不是去遍历数据库的完整的表，这样做有很大的效率提升

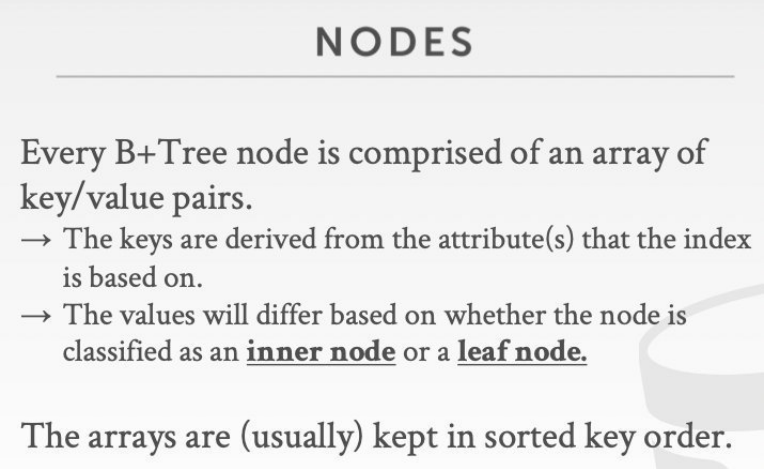
在构建索引时，要有一些权衡考量（trade-off），我们建立的索引越多（比如说给数据库表里年龄/身高/性别这些字段都分别构建索引），数据库查询的执行就会越快，但是也会带来在存储和维护索引（对数据库的完整表的增删改查要同步到索引当中）上的开销

B+ Tree属于B-Tree Family，是一种自平衡的数据结构，搜索，顺序访问（也叫线性遍历，简单的说就是，B+树可以比较好的完成哈希表所不擅长的范围查询），插入，删除的时间复杂度都是O(log n)，也就是说，B+树存储的数据规模增加时，前面的各种操作的开销的增长比数据规模的增长要缓慢的多，此外，B+树对于在磁盘上读写有好几个页大的文件有特殊的优势

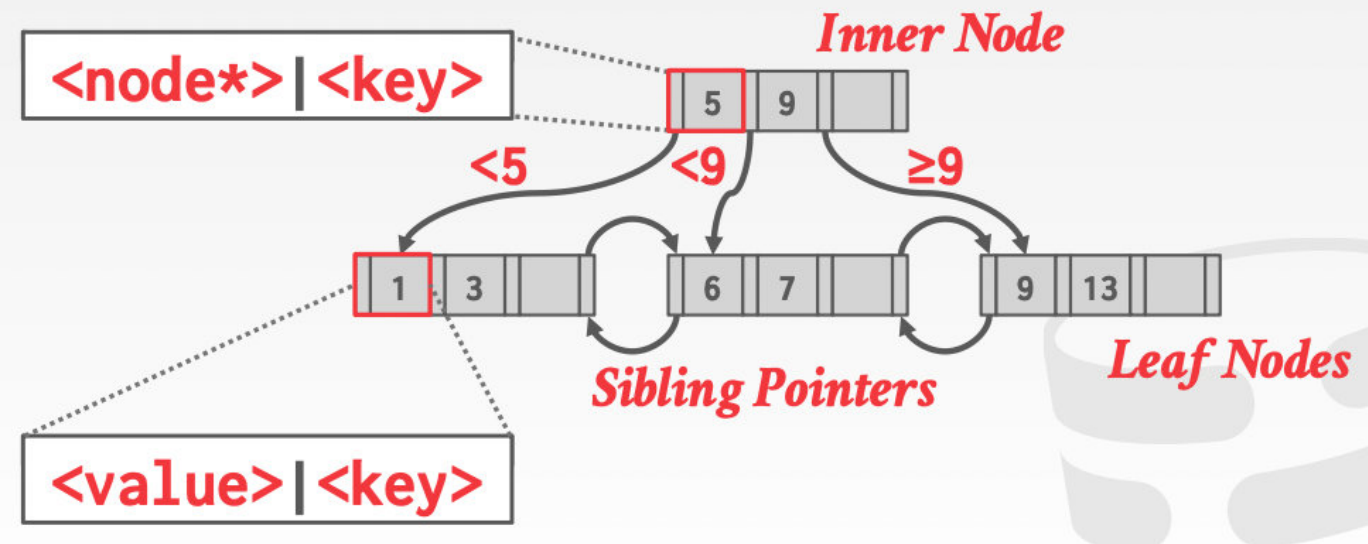
二叉树里一个父节点只有2个子节点，是2-way的，而B+树是M-way的，父节点可以有多个子节点，并且B+树除了根节点以外的每个节点都存有多个数据（更详细的说，是每个节点存储的KV的数目在[M/2-1, M-1]这个区间里），这和每个节点只存储一个数据的大部分树不同



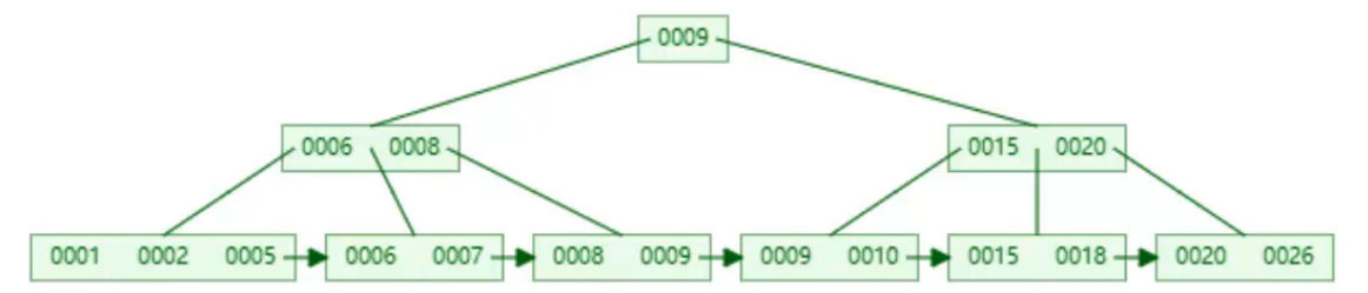
B+树的每个节点都是一个KV数组，这个数组内部是严格按照K的大小顺序组织的，内部节点的V存储的是指向子节点的指针，叶子节点的V是我们最终想要索引得到的信息



B+树的结构如下图所示，从内部节点所延伸出的三个指针分别指向Key值小于5，在5和9之间，大于9的叶子节点，每个叶子节点有多个Key-Value，互为兄弟节点的叶子节点之间有sibling pointer互相指向对方

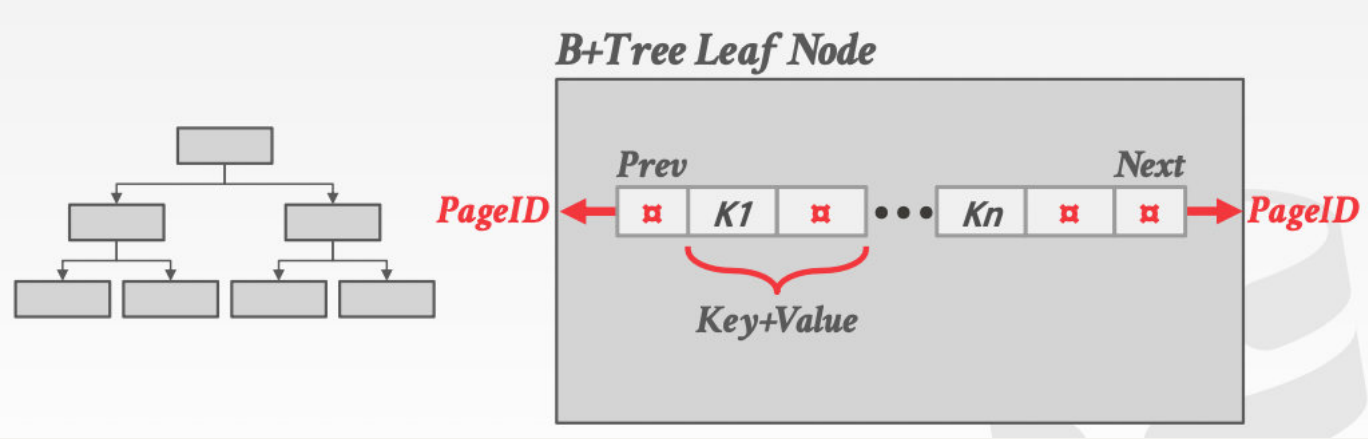


数据结构可视化网站上面可以看到B+树的全貌，不过这里的叶子节点没有value

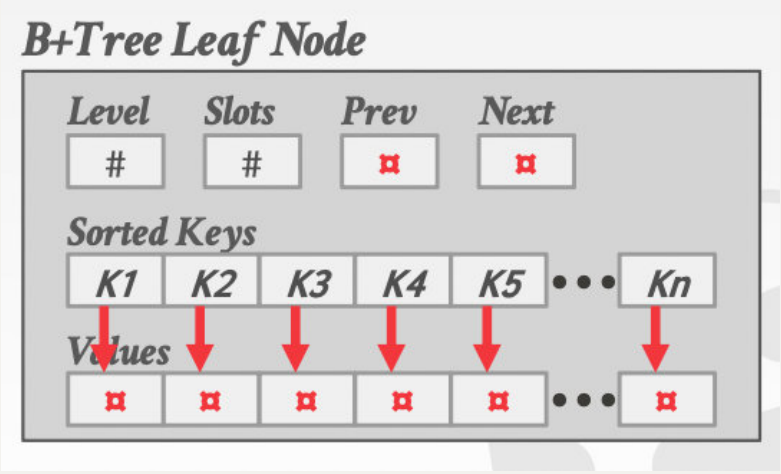


可以看到，最底下的叶子节点那一层像一个链表，往上面的倒数第二层像这个链表的索引，倒数第三层可以看作倒数第二层所构成的链表的索引

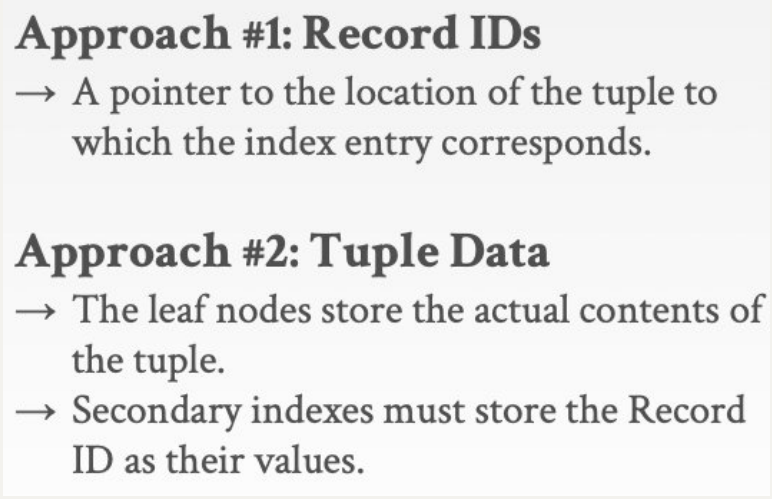
哈希表并不是逻辑上分成一块一块的，把它存入磁盘时可能要强行分块存储，与之相反，B+树如果设计的合理，可以使得一个节点的很多KV数据加起来的大小就是一页，也就是数据库文件里的一页对应B+树的一个节点，从而可以以B+树节点为单位读/写文件，当然也有些数据库的设计中是一个B+树节点由好几个页构成，如果每个叶子节点都是正好占一页的话，那么叶子节点中就可以拿page ID作为指向兄弟节点的指针



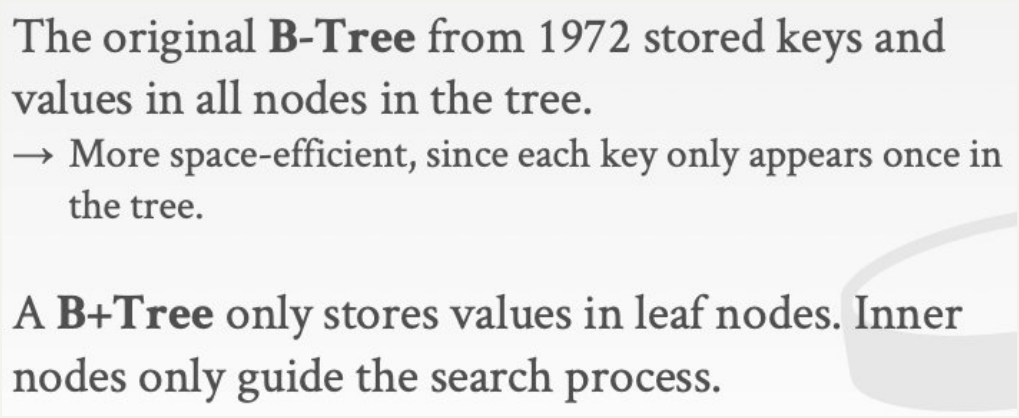
叶子节点实际的实现方式，即内部的layout也可以与上图不同，如下图，K和V分别存放在两个数组里，叶子节点里面还可以存一些元数据，比如说描述该节点位于B+树第几层的`Level`，用于索引叶子节点所存储的KV的`Slots`



叶子节点所存储的KV中的V可以是对应tuple的ID，也可以是tuple本身，正如下图的Approach 2所说描述的那样，这种场景的确有可能发生，很多数据库的完整的表本身就是使用B+树的结构来存储的，K是主键，V是表里面整整一行的数据

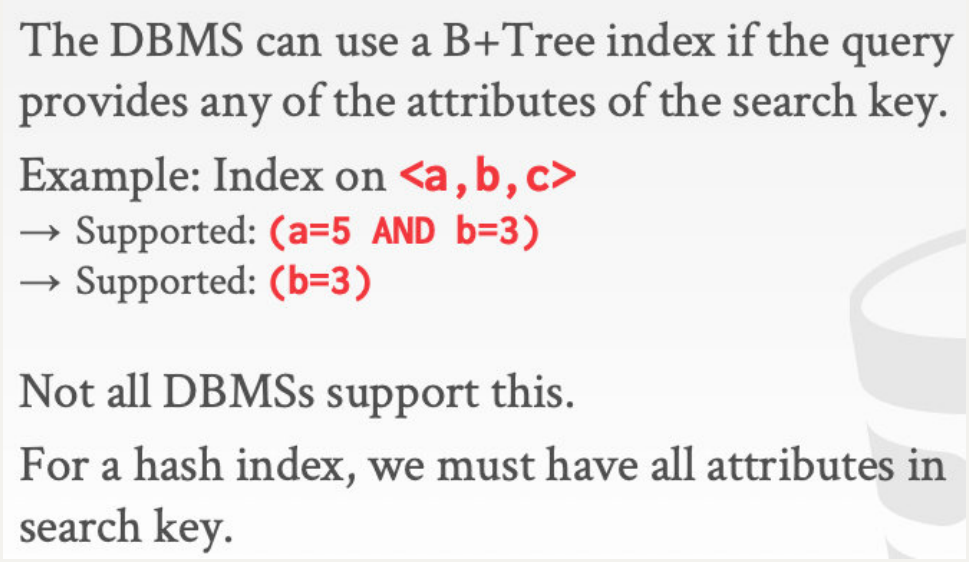


B+树和B树的区别在于，B+树的内部节点只用于向叶子节点索引，而不用于存储数据，B树的内部节点既用于索引，也用于存储数据，因此B树可以占用更少的存储空间来存储同样规模的数据

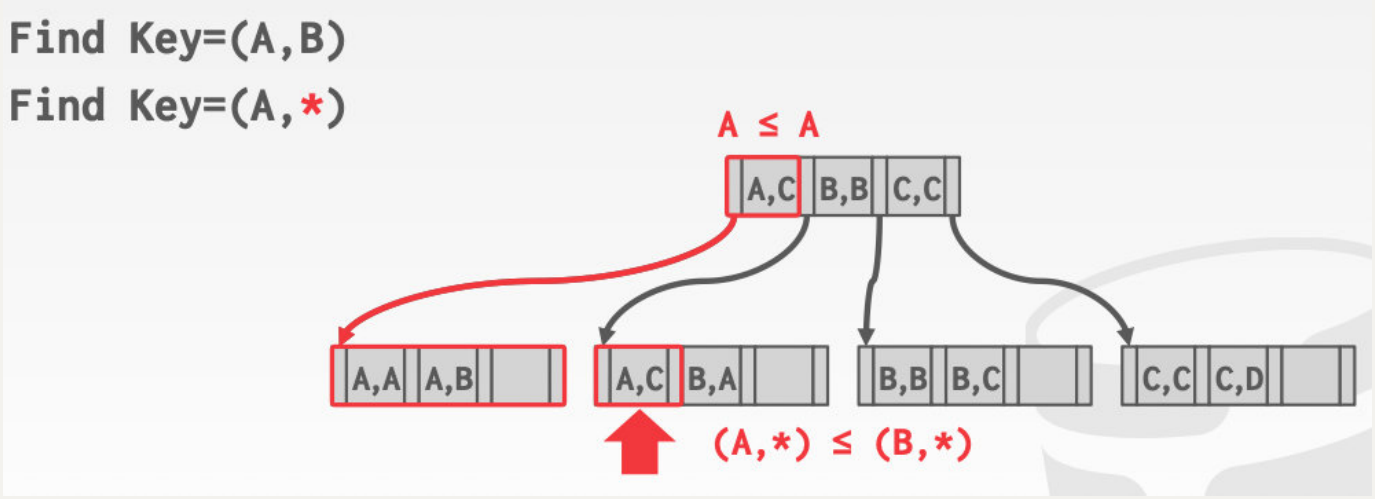


B+树的搜索

B+树适用于联合索引，我们用于索引的Key可以有多个字段/属性，我们可以只拿其中的一个属性的值去做索引，也可以拿部分或全部的属性的值去做索引，这种就叫联合索引



slides中以联合索引为例，给出了(A, B)和(A, *)的搜索过程，

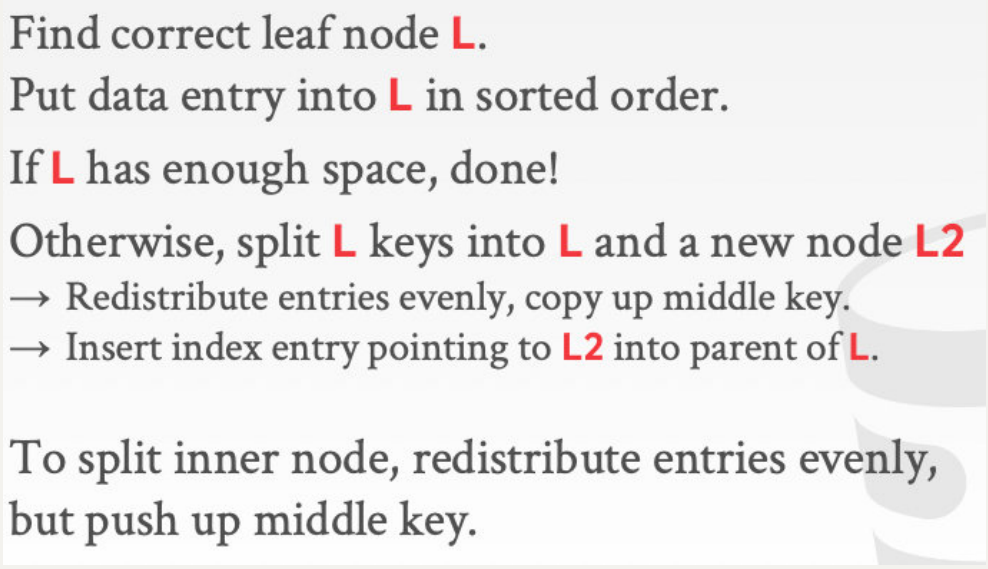


但如果想搜索(*, B)，那么可以分成(A, B)，(B, B)，(C, B)三次搜索，这种方式被称为skip scan，跳跃搜索，这种搜索也并不是所有数据库都会支持的，因为效率并不高

（B+树的插入与删除的具体步骤较为复杂，slides中也是一笔带过，具体步骤可以参考这篇文章）

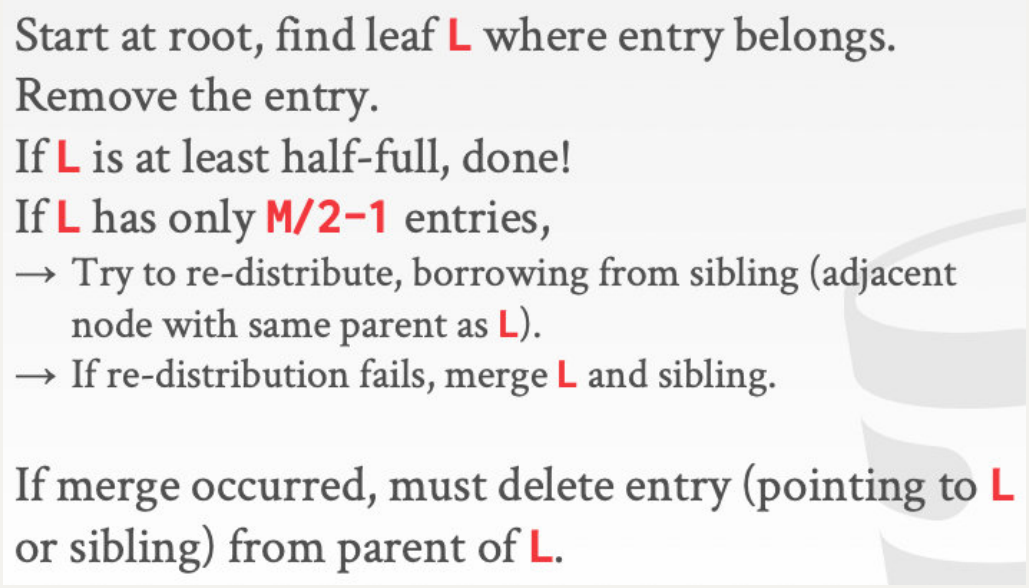
B+树的插入

向B+树中插入数据时要先找到需要插入到的叶子节点，然后在叶子节点里依照之前维护好的顺序将新的数据插入到正确的位置，如果这个叶子节点已经满了的话，就把它分裂成两个叶子节点，



B+树的删除

删除B+树中的某条数据，要先找到这条数据在哪，之后进行删除，如果删除之后该节点所含有的数据太少的话，还需要和周围的节点合并，但很多的DBMS会推迟这个合并的操作，因为在很多情况下，从B+树中删掉某个数据之后还会再插入新的数据，这样刚刚合并好的节点就有可能要再进行分裂，造成很大的开销



回到最开始举的例子，如果对所有人的年龄做索引，但有些人的年龄有重复的，那么应该怎么处理？接下来介绍B+树处理Duplicate Keys的方法

Approach #1: Append Record ID

→ Add the tuple's unique Record ID as part of the key to ensure that all keys are unique.

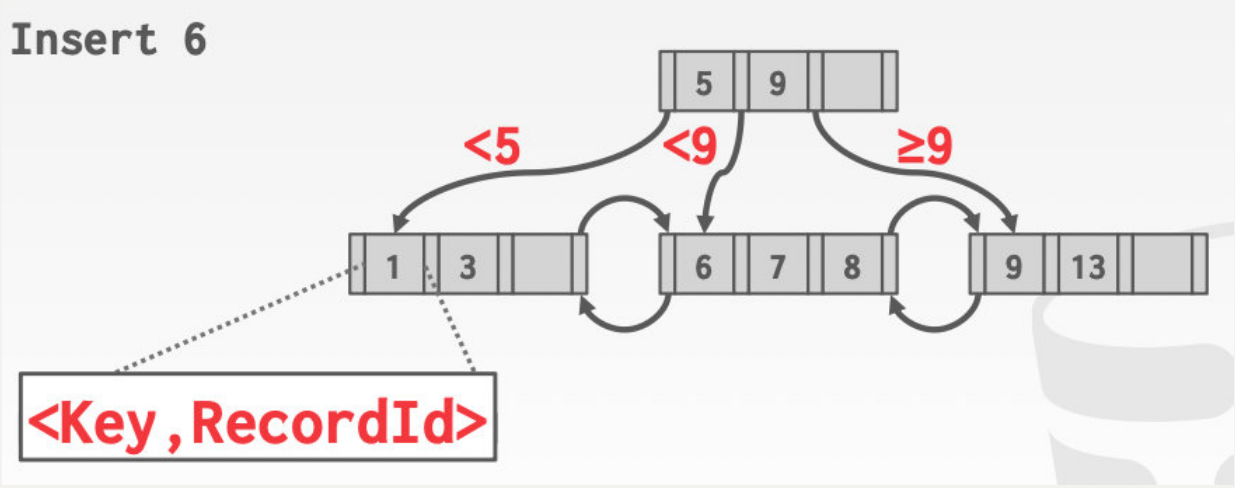
→ The DBMS can still use partial keys to find tuples.

Approach #2: Overflow Leaf Nodes

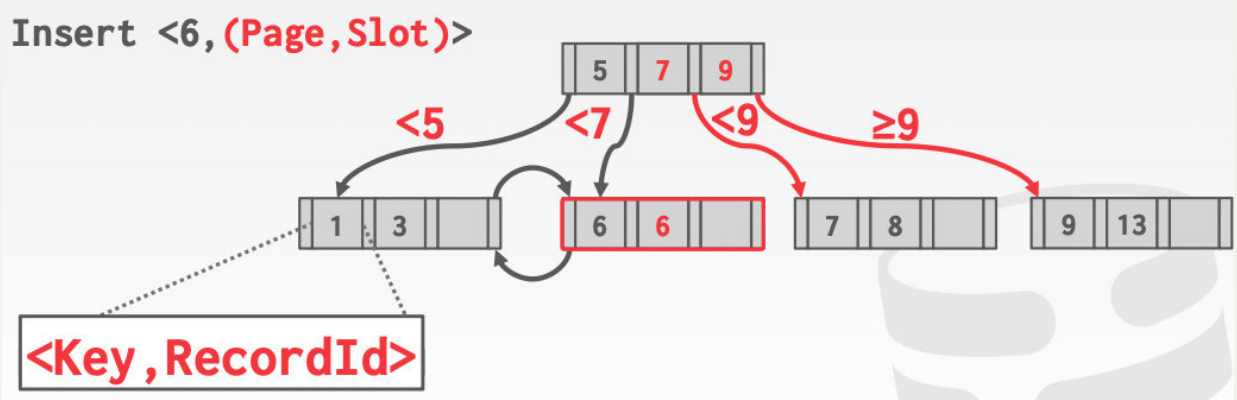
→ Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.

→ This is more complex to maintain and modify.

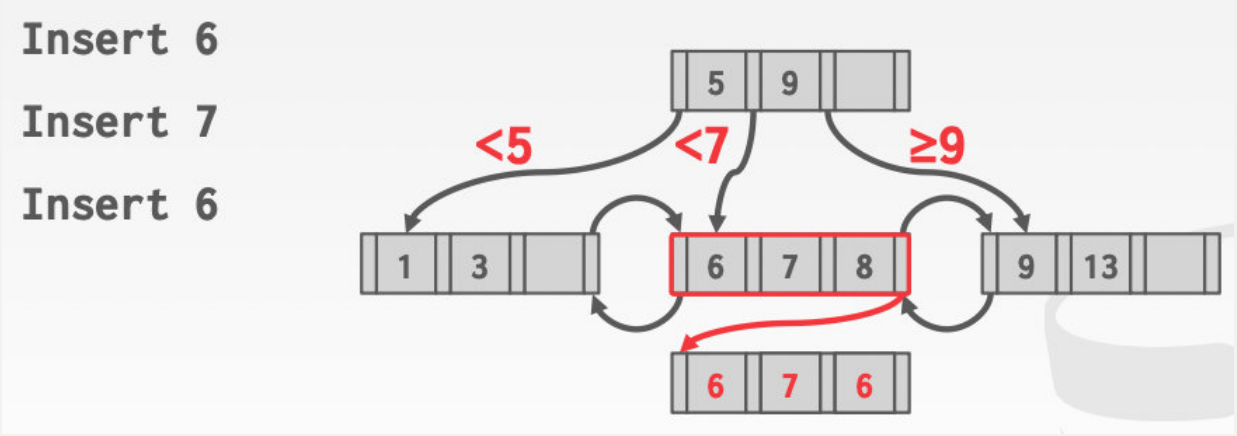
- 在Key后面再加上一个id



-> after a few steps ->



- 叶子节点溢出，在原有的叶子节点后面外接一个溢出节点



Clustered Index，聚簇索引

如果数据库存储的数据是按照主键索引去组织的（就是前面说过的B+树所存储的KV中的V是完整的tuple那种情况），就称为聚簇索引，索引本身将数据库整个表的数据组织了起来

不使用聚簇索引的话，主键索引的KV中的V一般会是对应tuple的页号与slot号

The table is stored in the sort order specified by the primary key.
→ Can be either heap- or index-organized storage.

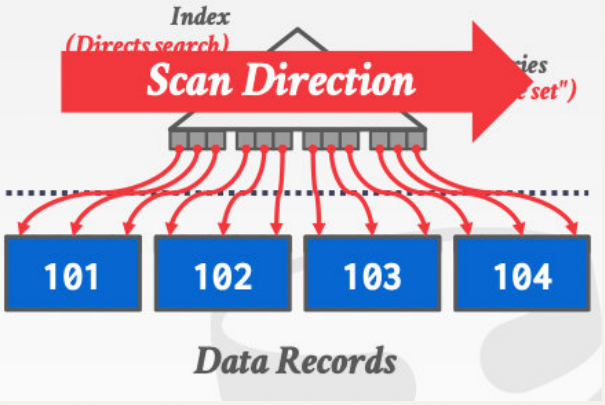
Some DBMSs always use a clustered index.
→ If a table does not contain a primary key, the DBMS will automatically make a hidden primary key.

Other DBMSs cannot use them at all.

使用聚簇存储的话，主键的B+树的叶子节点的结构和实际上磁盘中存储的数据库文件页是一样的，就可以通过遍历B+树的叶子节点从而实现按主键大小有序地遍历数据库文件页里的所有tuple，这个过程中很多时候都是连续的读取磁盘里同一文件页的数据，效率很高

Traverse to the left-most leaf page and then retrieve tuples from all leaf pages.

This will always be better than external sorting.

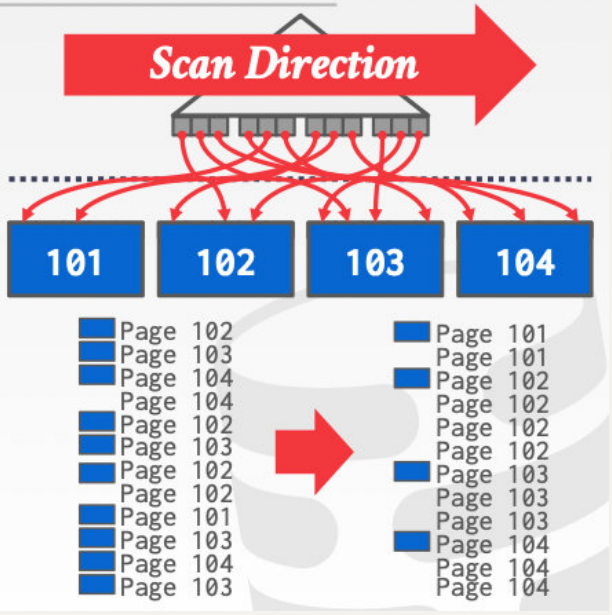


相比之下，如果是非聚簇索引，在遍历叶子节点时，需要通过叶子节点的V提供的tuple地址间接的访问对应tuple，因此不再像前面那样连续读取相同的数据库文件页的内容，而是会导致随机访问不同的文件页中的数据，效率下降，因此DBMS也有相应的优化策略：先缓存下来被遍历到的叶子节点中的各个V都指向第几号数据库文件页，然后等遍历完了全部的叶子节点，一次性地读取需要读的页，如下所示

INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index can be very inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their Page ID.



接下来介绍B+树的一些设计策略

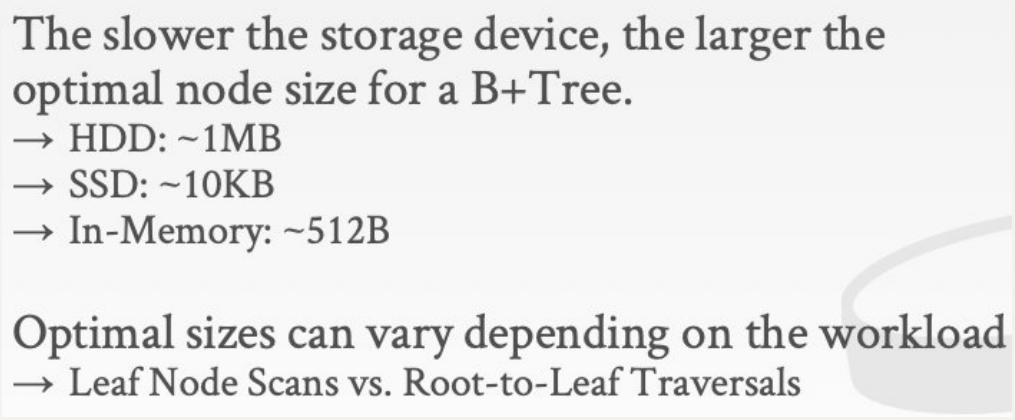
- Node Size

  B+树节点的大小最好和文件页的大小一致或者是其倍数，以便于管理

  磁盘越低速，B+树节点占用的存储空间应该越大，这样的话一次对单个节点的磁盘I/O就可以读取更多的数据，与此同时，越快速的设备越要求B+树节点占用更小的存储空间，因为B+树节点占用的空间越大，所存储的数据越过，读取该节点时所读到的冗余数据也多，如果存储设备足够快速，我们可以让单个节点少存储一些

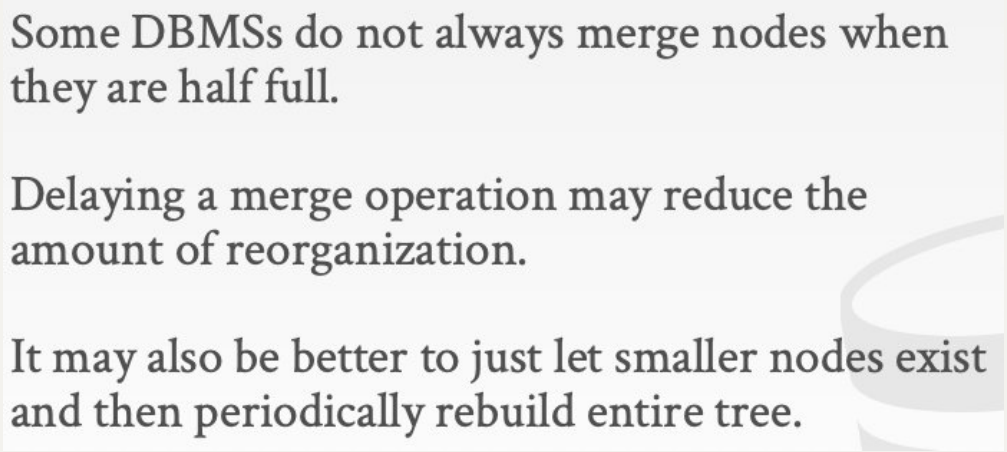
数据，多几次I/O，这样读到的冗余数据就会少很多

B+树的节点大小的选择也和DBMS所应对的负载类型有关，如果是应对OLAP类型的负载，常常会全表扫描，会遍历B+树的叶子节点，因此不妨让每个节点大一点，这样单次I/O就能扫描更多数据，对于OLTP这种事务型的经常进行点查询的工作负载，会经常从B+树的根节点遍历到叶子节点，我们不妨就让B+树的节点小一点，这样的话在查询过程中，在节点之间跳跃的开销就会变小



- Merge Threshold，节点合并的阈值

  前面有提到，DBMS会延迟B+树删除操作后的节点合并，从而减少重新组织B+树带来的开销



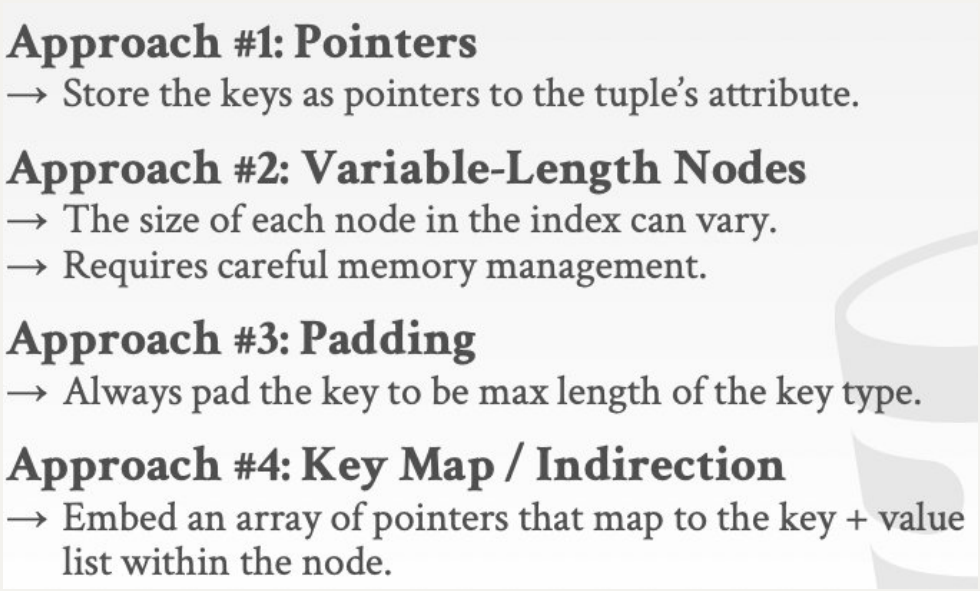
- Variable-Length Keys，可变长度的Key

  如果拿字符串当作索引的Key，那么很有可能它是变长的，对于变长的Key，DBMS有如下的处理方式：

  - 改为存储Key的指针，笔者个人感觉有可能采取ELF文件格式里符号表中的表项里的符号名存储的是该符号名在字符串表中的索引这种策略
  - 让B+树每个节点也都是变长节点，但这种策略并不推荐，因为节点变长的话就不一定对应一个或多个文件页了，管理的难度会变大
  - Padding策略，padding这个词笔者之前在学习OS的引导扇区时见过，OS启动所需的引导扇区的大小是固定的，xv6中引导扇区是位于磁盘的第一个block，但引导扇区中存储的用于启动的代码的长度未必正好有一个

block的大小，因此就需要在程序文件剩下的位置补0，直至总长度达到一个block那么大，然后存入磁盘里对应的扇区，从而制作好引导扇区，这种方式就叫padding，在DBMS里，如果Key是变长的，我们也可以再向Key中补入空的字节，直到让Key的长度达到我们所设计的fixed-size

- 这个跟前面介绍B+树节点内部layout时介绍的使用slot的策略（或者说是存储引擎模块里的tuple在Page里面的Layout通过slot来组织的方式）很像，节点里面有数组形式排布的slot，slot中存储指向对应KV的指针，这使得节点中KV的存储有很大灵活性



除此之外，还有一种方法能够使得在Key变长时，B+树节点能够定长

Mysql的一个tuple不能超过1/2页的大小，这个要求的存在说明节点中存储的数据的数量是不定的

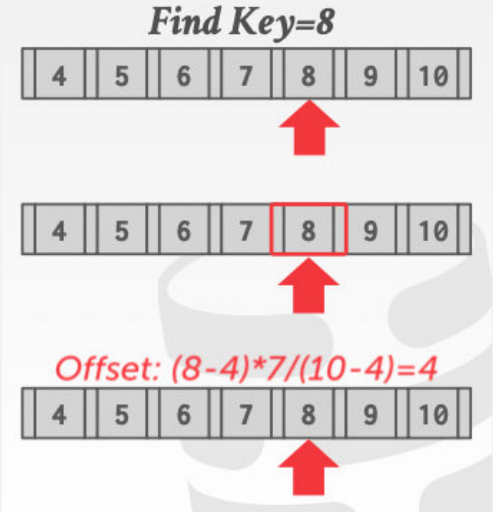
如果B+树要求节点定长，但包含Key在内的数据变长，我们可以让每个节点存储的数据的数量不固定，灵活应变，并且此后以每个节点所存储的数据的量的大小而非数据的多少为节点分裂与合并的阈值，当然这种管理的策略的实际实现也是非常复杂的

- Intra-Node Search，节点内部搜索

在工业界的B+树实现中，一个节点里会存储成百上千个数据，假设我们之前通过索引，已经确定好了想要获取的数据就在某个特定的叶子节点当中，但是叶子节点中仍然有成百上千条数据，如何更快地从中把我们想要的那一条数据找出来仍然是不小的挑战，因此有如下的策略：

  - 线性扫描，可以理解为暴力地去遍历，虽然看起来低效，但很多数据库都是这么实现的，因为相比于把叶子节点从磁盘读入缓存池用的时间，暴力扫描用的时间微不足道

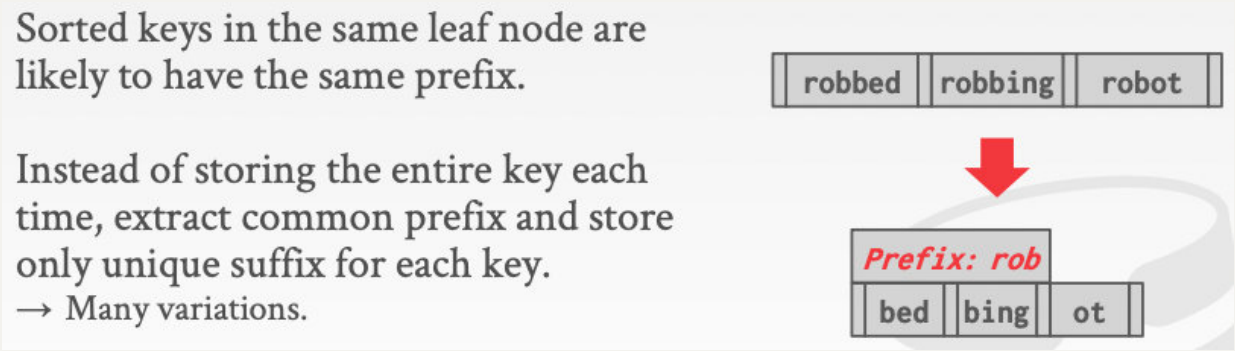
- 二分查找，就是使用简单的二分查找算法，因为叶子节点中的KV都是按照K的大小有序排列的
- 根据节点中诸多KV中的K的排布规律推断出我们想要的那条KV在哪里，如下所示



最后介绍B+树的优化方案

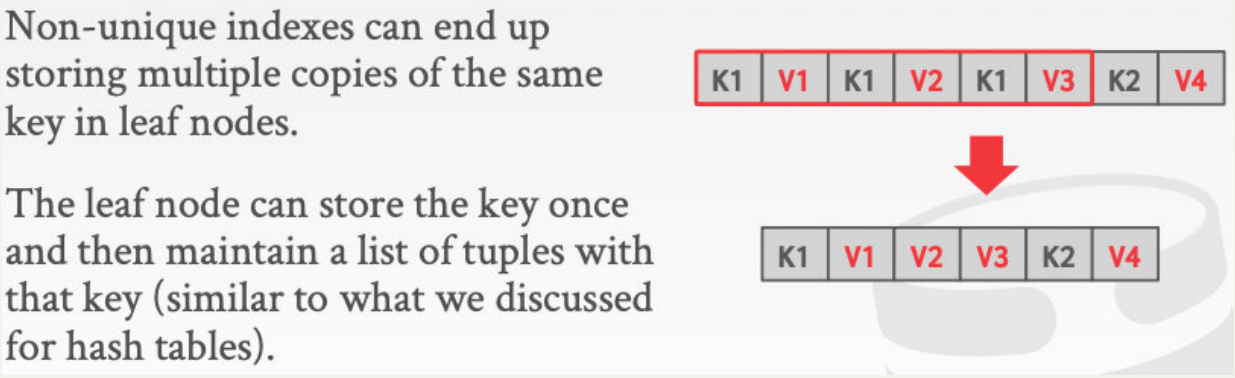
- Prefix Compression，前缀压缩

  在同一个叶子节点里很多Key的前缀是相同的，我们可以在叶子节点的元数据里面找个地方记录一下公共的前缀，实际存储Key的时候只需存储前缀后面不一样的内容即可，如下所示



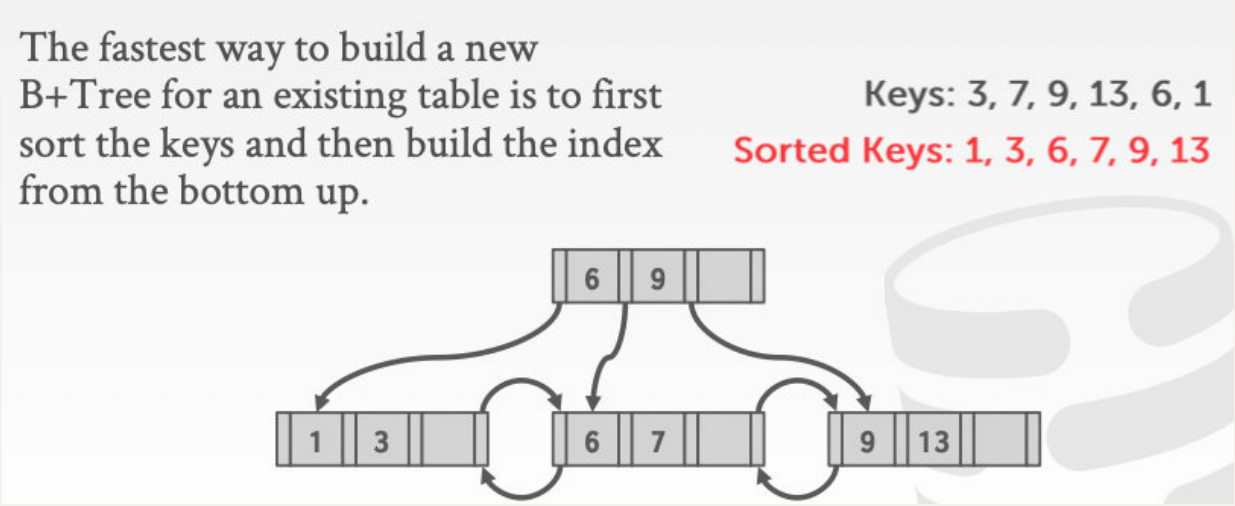
- Deduplication，去重

  前面说过，有些时候K会有冗余的可能，多个KV的K是一样的，我们其实可以通过去掉冗余的K来减少所占用的存储空间，如下所示



- Bulk Insert，按批插入

  向B+树中一个一个地插入KV的效率并不高，因为逐个插入的过程中B+树的对应节点会越来越长，长到一定程度的话，再插入就会导致节点分裂，节点分裂就会导致相应的维护带来的开销，因此我们在创建B+树时无需一个一个插入，而是可以使用高效的如下所示的建立B+树的方法，这种思想和创建堆时可以选择快速建堆策略而非一个一个地插入建堆的思想有点像



  另外，当我们想在原有的B+树中插入n个KV的话，也可以先拿这n个要插入的数据建一个B+树，然后merge到主B+树中，这个效率也是比一个一个插要高的

总结一下，B+树可以说是最适合用来做数据库索引的数据结构，虽然现在有跳表和哈希在数据库索引中的实践，但B+树还是最主流的选择，我们目前介绍的都是单线程下的索引，接下来即将介绍多线程并发进行索引的场景

Reference:

https://www.bilibili.com/video/BV1Bg411w7kW/?spm_id_from=333.788