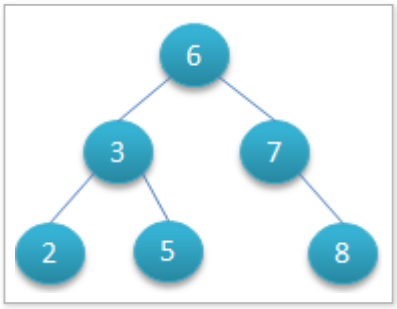
BTree 和 B+Tree 详解

B + 树索引是 B + 树在 数据库 中的一种实现，是最常见也是数据库中使用最为频繁的一种索引。B + 树中的 B 代表平衡（balance），而不是二叉（binary），因为 B + 树是从最早的平衡二叉树演化而来的。在讲 B + 树之前必须先了解二叉查找树、平衡二叉树（AVLTree）和平衡多路查找树（B-Tree），B + 树即由这些树逐步优化而来。

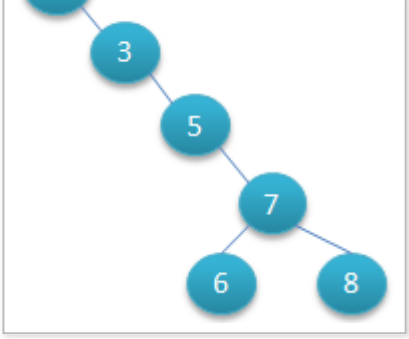
二叉查找树

二叉树具有以下性质：左子树的键值小于根的键值，右子树的键值大于根的键值。
如下图所示就是一棵二叉查找树，



对该二叉树的节点进行查找发现深度为 1 的节点的查找次数为 1，深度为 2 的查找次数为 2，深度为 n 的节点的查找次数为 n，因此其平均查找次数为 (1+2+2+3+3+3) / 6 = 2.3 次

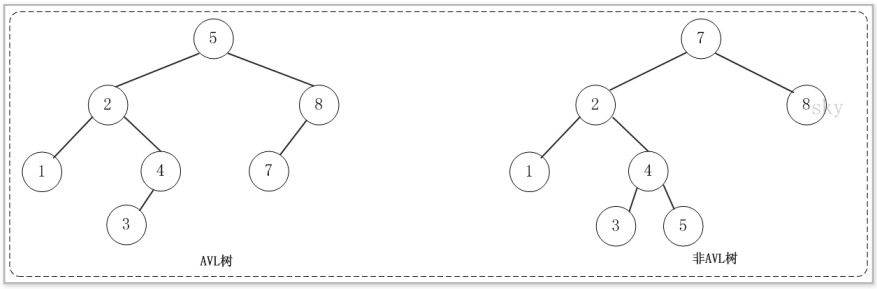
二叉查找树可以任意地构造，同样是 2,3,5,6,7,8 这六个数字，也可以按照下图的方式来构造：



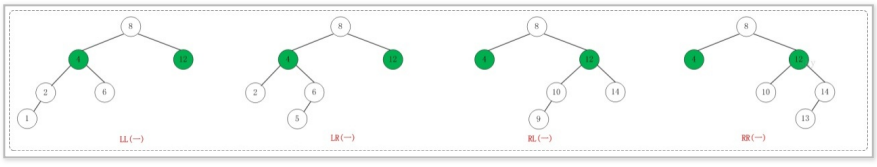
但是这棵二叉树的查询效率就低了。因此若想二叉树的查询效率尽可能高，需要这棵二叉树是平衡的，从而引出新的定义——平衡二叉树，或称 AVL 树。

平衡二叉树（AVL Tree）

平衡二叉树（AVL 树）在符合二叉查找树的条件下，还满足任何节点的两个子树的高度最大差为 1。下面的两张图片，左边是 AVL 树，它的任何节点的两个子树的高度差 <=1；右边的不是 AVL 树，其根节点的左子树高度为 3，而右子树高度为 1；



如果在 AVL 树中进行插入或删除节点，可能导致 AVL 树失去平衡，这种失去平衡的二叉树可以概括为四种姿态：LL（左左）、RR（右右）、LR（左右）、RL（右左）。它们的示意图如下：



这四种失去平衡的姿态都有各自的定义：
LL：LeftLeft，也称 “左左”。插入或删除一个节点后，根节点的左孩子（Left Child）的左孩子（Left Child）还有非空节点，导致根节点的左子树高度比右子树高度高 2，AVL 树失去平衡。

RR：RightRight，也称 “右右”。插入或删除一个节点后，根节点

的右孩子（Right Child）的右孩子（Right Child）还有非空节点，导致根节点的右子树高度比左子树高度高 2，AVL 树失去平衡。

LR：LeftRight，也称 “左右”。插入或删除一个节点后，根节点的左孩子（Left Child）的右孩子（Right Child）还有非空节点，导致根节点的左子树高度比右子树高度高 2，AVL 树失去平衡。

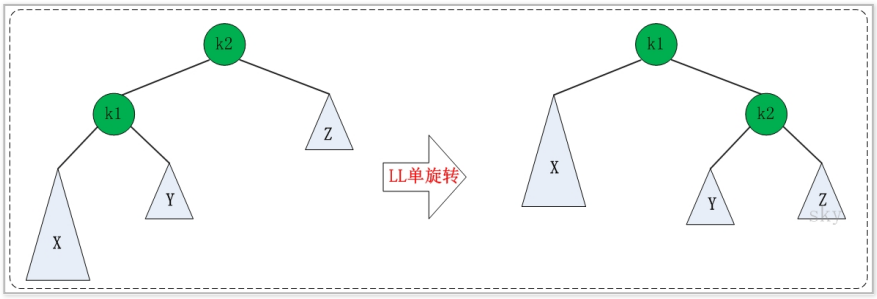
RL：RightLeft，也称 “右左”。插入或删除一个节点后，根节点的右孩子（Right Child）的左孩子（Left Child）还有非空节点，导致根节点的右子树高度比左子树高度高 2，AVL 树失去平衡。

AVL 树失去平衡之后，可以通过旋转使其恢复平衡。下面分别介绍四种失去平衡的情况下对应的旋转方法。

LL 的旋转。LL 失去平衡的情况下，可以通过一次旋转让 AVL 树恢复平衡。步骤如下：

- 将根节点的左孩子作为新根节点。
- 将新根节点的右孩子作为原根节点的左孩子。
- 将原根节点作为新根节点的右孩子。

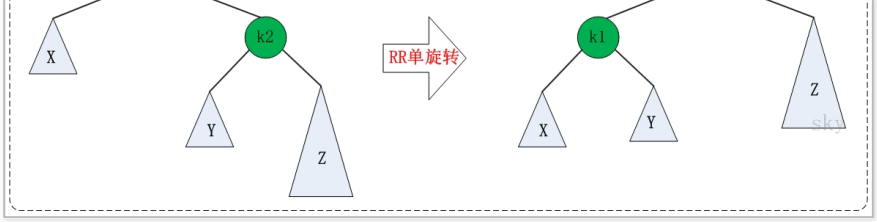
LL 旋转示意图如下：



RR 的旋转：RR 失去平衡的情况下，旋转方法与 LL 旋转对称，步骤如下：

- 将根节点的右孩子作为新根节点。
- 将新根节点的左孩子作为原根节点的右孩子。
- 将原根节点作为新根节点的左孩子。

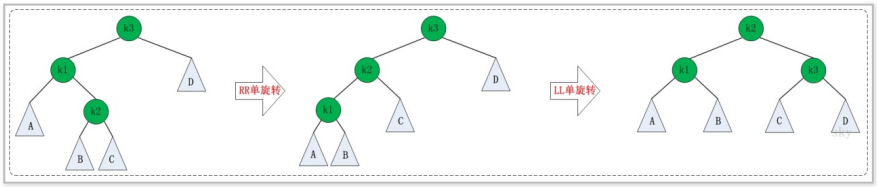
RR 旋转示意图如下：



LR 的旋转：LR 失去平衡的情况下，需要进行两次旋转，步骤如下：

- 围绕根节点的左孩子进行 RR 旋转。
- 围绕根节点进行 LL 旋转。

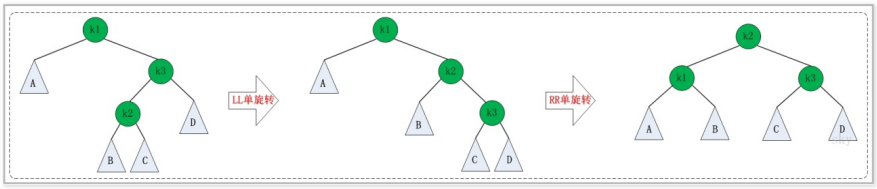
LR 的旋转示意图如下：



RL 的旋转：RL 失去平衡的情况下也需要进行两次旋转，旋转方法与 LR 旋转对称，步骤如下：

- 围绕根节点的右孩子进行 LL 旋转。
- 围绕根节点进行 RR 旋转。

RL 的旋转示意图如下：



平衡多路查找树（B-Tree）

B-Tree 是为磁盘等外存储设备设计的一种平衡查找树。因此在讲 B-Tree 之前先了解下磁盘的相关知识。

系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。

InnoDB 存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB 存储引擎中默认每个页的大小为 16KB，可通过参数

innodb_page_size 将页的大小设置为 4K、8K、16K，在 MySQL 中可通过如下命令查看页的大小：

```
mysql> show variables like 'innodb_page_size';
```

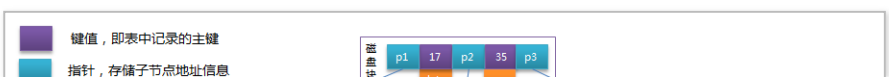
- 1
- 1

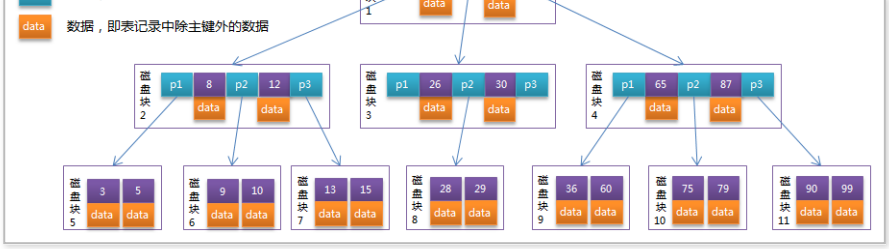
而系统一个磁盘块的存储空间往往没有这么大，因此 InnoDB 每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小 16KB。InnoDB 在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘 I/O 次数，提高查询效率。

B-Tree 结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述 B-Tree，首先定义一条记录为一个二元组 [key, data] ，key 为记录的键值，对应表中的主键值，data 为一行记录中除主键外的数据。对于不同的记录，key 值互不相同。

一棵 m 阶的 B-Tree 有如下特性：
1. 每个节点最多有 m 个孩子。
2. 除了根节点和叶子节点外，其它每个节点至少有 Ceil(m/2) 个孩子。
3. 若根节点不是叶子节点，则至少有 2 个孩子
4. 所有叶子节点都在同一层，且不包含其它关键字信息
5. 每个非终端节点包含 n 个关键字信息（P0,P1,...Pn, k1,...kn）
6. 关键字的个数 n 满足：ceil(m/2)-1 <= n <= m-1
7. ki(i=1,...n) 为关键字，且关键字升序排序。
8. Pi(i=1,...n) 为指向子树根节点的指针。P(i-1) 指向的子树的所有节点关键字均小于 ki，但都大于 k(i-1)

B-Tree 中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个 3 阶的 B-Tree：





每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为 17 和 35，P1 指针指向的子树的数据范围为小于 17，P2 指针指向的子树的数据范围为 17~35，P3 指针指向的子树的数据范围为大于 35。

模拟查找关键字 29 的过程：

- 根据根节点找到磁盘块 1，读入内存。【磁盘 I/O 操作第 1 次】
- 比较关键字 29 在区间（17,35），找到磁盘块 1 的指针 P2。
- 根据 P2 指针找到磁盘块 3，读入内存。【磁盘 I/O 操作第 2 次】
- 比较关键字 29 在区间（26,30），找到磁盘块 3 的指针 P2。
- 根据 P2 指针找到磁盘块 8，读入内存。【磁盘 I/O 操作第 3 次】
- 在磁盘块 8 中的关键字列表中找到关键字 29。

分析上面过程，发现需要 3 次磁盘 I/O 操作，和 3 次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而 3 次磁盘 I/O 操作是影响整个 B-Tree 查找效率的决定因素。B-Tree 相对于 AVLTree 缩减了节点个数，使每次磁盘 I/O 取到内存的数据都发挥了作用，从而提高了查询效率。

B+Tree

B+Tree 是在 B-Tree 基础上的一种优化，使其更适合实现外存储索引结构，InnoDB 存储引擎就是用 B+Tree 实现其索引结构。

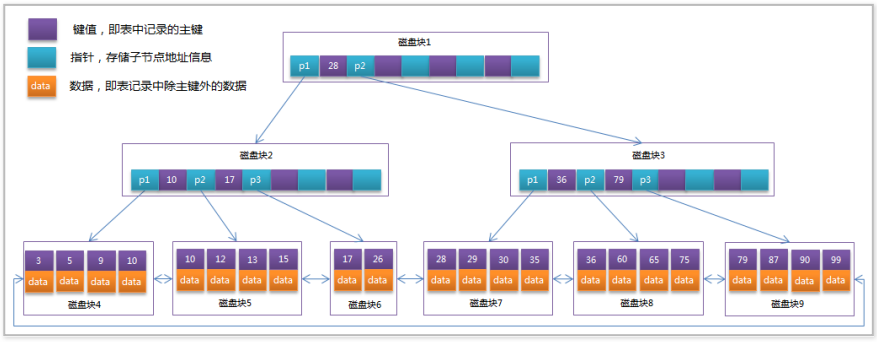
引结构，InnoDB 存储引擎就是用 B+Tree 实现其索引结构。

从上一节中的 B-Tree 结构图中可以看到每个节点中不仅包含数据的 key 值，还有 data 值。而每一个页的存储空间是有限的，如果 data 数据较大时将会导致每个节点（即一个页）能存储的 key 的数量很小，当存储的数据量很大时同样会导致 B-Tree 的深度较大，增大查询时的磁盘 I/O 次数，进而影响查询效率。在 B+Tree 中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储 key 值信息，这样可以大大加大每个节点存储的 key 值数量，降低 B+Tree 的高度。

B+Tree 相对于 B-Tree 有几点不同：

- 非叶子节点只存储键值信息。
- 所有叶子节点之间都有一个链指针。
- 数据记录都存放在叶子节点中。

将上一节中的 B-Tree 优化，由于 B+Tree 的非叶子节点只存储键值信息，假设每个磁盘块能存储 4 个键值及指针信息，则变成 B+Tree 后其结构如下图所示：



通常在 B+Tree 上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对 B+Tree 进行两种查找运算：一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

可能上面例子中只有 22 条数据记录，看不出 B+Tree 的优点，下面做一个推算：

InnoDB 存储引擎中页的大小为 16KB，一般表的主键类型为 INT（占用 4 个字节）或 BIGINT（占用 8 个字节），指针类型也一般为 4 或 8 个字节，也就是说一个页（B+Tree 中的一个节点）中大概存储 16KB/(8B+8B)=1K 个键值（因为是估值，为方便计算，

这里的 K 取值为『10』^3）。也就是说一个深度为 3 的 B+Tree 索引可以维护 10^3 * 10^3 * 10^3 = 10 亿 条记录。

实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree 的高度一般都在 2~4 层。 mysql 的 InnoDB 存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要 1~3 次磁盘 I/O 操作。

数据库中的 B+Tree 索引可以分为聚集索引（clustered index）和辅助索引（secondary index）。上面的 B+Tree 示例图在数据库中的实现即为聚集索引，聚集索引的 B+Tree 中的叶子节点存放的是整张表的行记录数据。辅助索引与聚集索引的区别在于辅助索引的叶子节点并不包含行记录的全部数据，而是存储相应行数据的聚集索引键，即主键。当通过辅助索引来查询数据时，InnoDB 存储引擎会遍历辅助索引找到主键，然后再通过主键在聚集索引中找到完整的行记录数据。

全文完