**https://blog.csdn.net/eagle89/article/details/72578148**

B+树索引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+树是从平衡二叉树演化而来的。了解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。

**二分查找法**

二分查找法：将记录按有序化（递增或递减）排列，查找过程中采用跳跃式方式查找；

或者

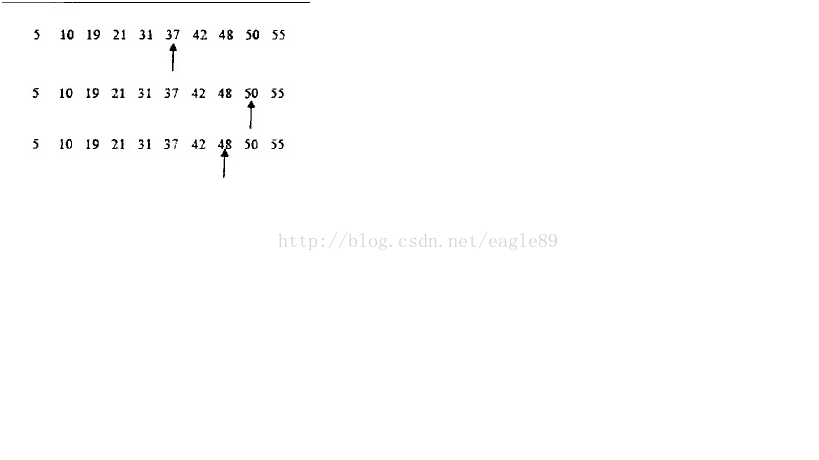
按照二叉树来理解：中间值为二叉树的根，前半部分为左子树，后半部分为右子树。折半查找法的查找次数正好为该值所在的层数。

二分查找（又称折半查找）前提：数组必须是有序的。

优点是比较次数少，查找速度快，平均性能好；

缺点是要求待查表为有序表，且插入删除困难

例如：5、10、19、21、31、37、42、48、50、52这10个数，如图所示：



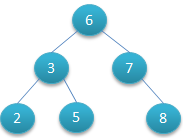
 用了三次查找速度就能找到48。如果是顺序查找的话，则需要8次。对于上面10个数来说，顺序查找的平均查找次数为5.5次，而二分查找法为2.9次，在最坏的情况下，顺序查找的次数为10，而二分查找的次数为4。二分查找在innodb中Page Directory中的槽是按照主键的顺序存放的，对于每一条具体记录的查询时通过对PageDirectory进行二分查找。

**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。

要理解B树，必须从二叉查找树（Binarysearch tree）讲起。

B树是通过二叉查找树，再由平衡二叉查树。

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

二叉查找树是一种查找效率非常高的数据结构，它有三个特点。

    （1）每个节点最多只有两个子树 。

    （2）左子树都为小于父节点的值，右子树都为大于父节点的值。

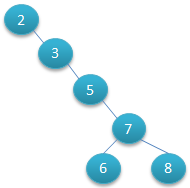
    （3）在n个节点中找到目标值，一般只需要log(n)次比较。

一颗非空的二叉查找树（Binary Search Tree, BST），满足如下特性：

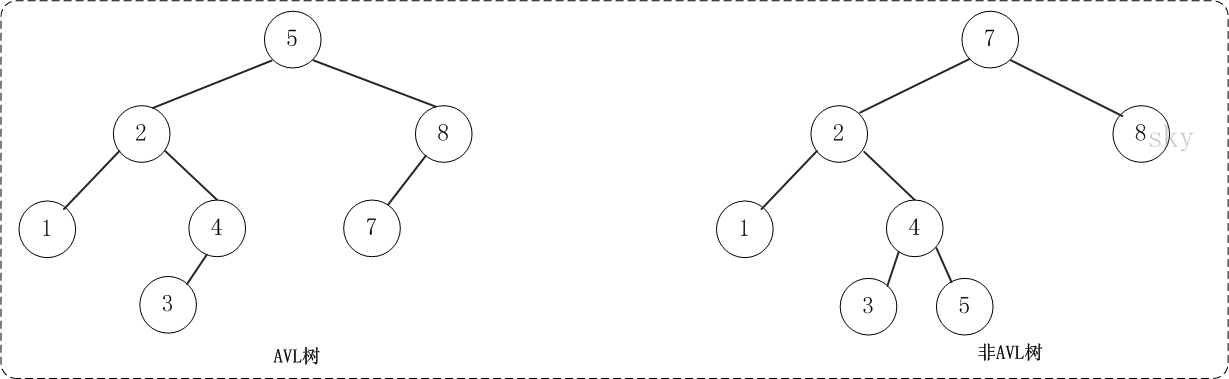
·        若根节点的左子树非空，则左子树所有节点数值小于根节点；

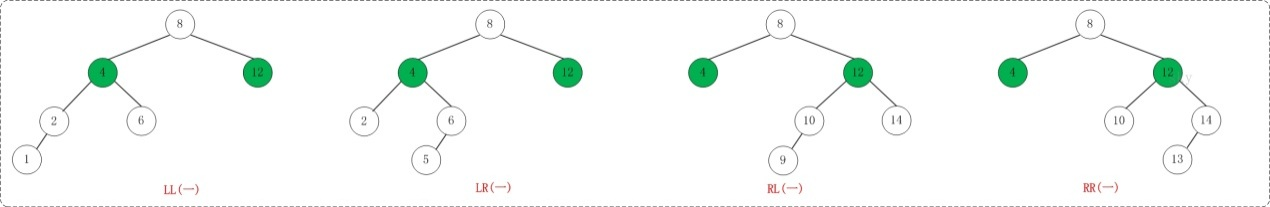
·        若根节点的右子树非空，则右子树所有节点数值大于根节点；

·        左右子树分别为一颗二叉查找树.

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）**

**平衡二叉树基于二叉查找树（BinarySearch Tree）,**平衡二叉树是一颗形态均匀的二叉树，平衡二叉树特点是一颗平衡二叉树的左右子树的深度之差应该小于或等于1，从根节点到所有的叶子节点深度不会相差太多。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。

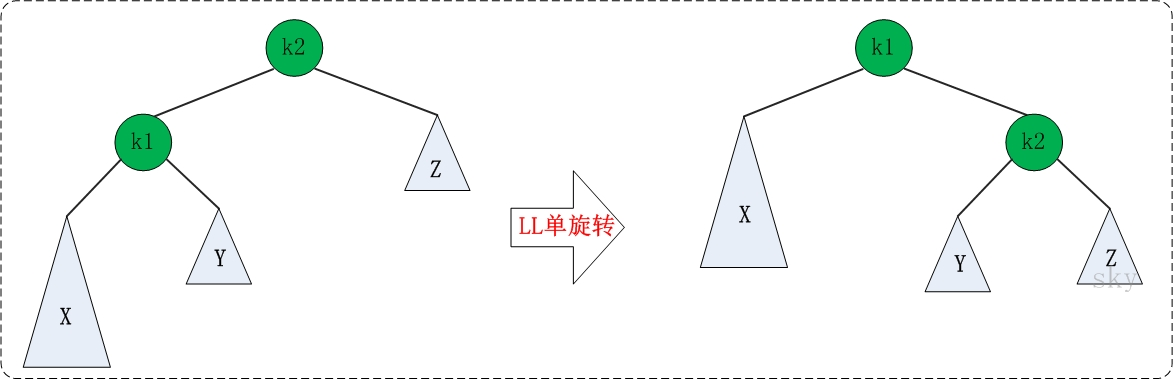
**LR**：LeftRight，也称“左右”。插入或删除一个节点后，根节点的左孩子（Left Child）的右孩子（Right Child）还有非空节点，导致根节点的左子树高度比右子树高度高2，AVL树失去平衡。

**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。

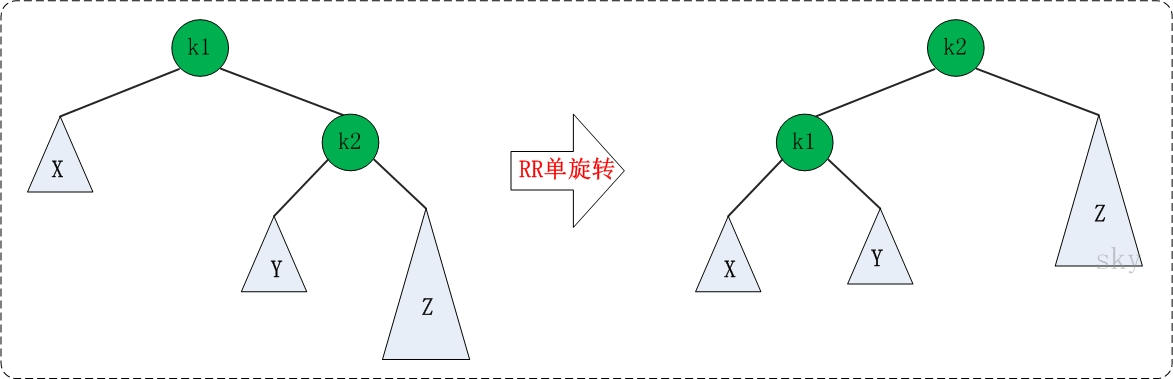
**LL的旋转**。LL失去平衡的情况下，可以通过一次旋转让AVL树恢复平衡。步骤如下：

1. 将根节点的左孩子作为新根节点。
2. 将新根节点的右孩子作为原根节点的左孩子。
3. 将原根节点作为新根节点的右孩子。

LL旋转示意图如下：   


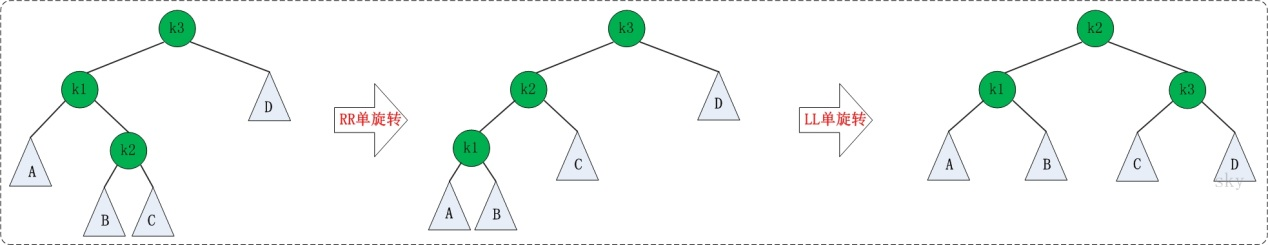
**RR的旋转**：RR失去平衡的情况下，旋转方法与LL旋转对称，步骤如下：

1. 将根节点的右孩子作为新根节点。
2. 将新根节点的左孩子作为原根节点的右孩子。
3. 将原根节点作为新根节点的左孩子。

RR旋转示意图如下：   


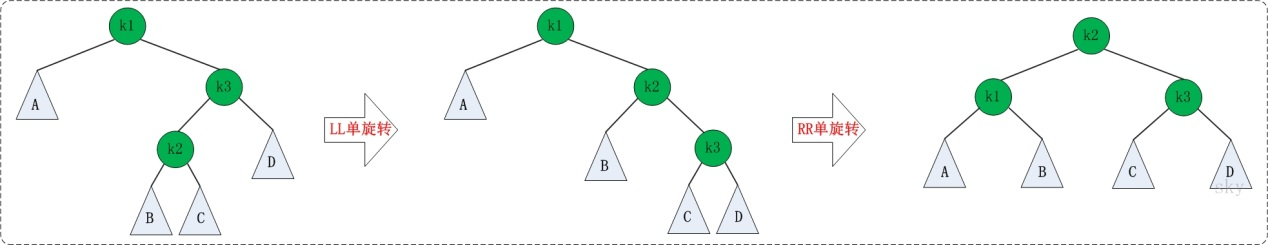
**LR的旋转**：LR失去平衡的情况下，需要进行两次旋转，步骤如下：

1. 围绕根节点的左孩子进行RR旋转。
2. 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：

1. 围绕根节点的右孩子进行LL旋转。
2. 围绕根节点进行RR旋转。

RL的旋转示意图如下：   


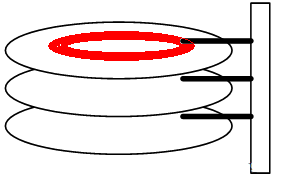
**B-Tree之前先了解下磁盘的相关知识**

     索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，**评价一个数据结构作为索引的优劣最重要的指标索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数**。

内存读取:内存是由一系列的存储单元组成的，每个存储单元存储固定大小的数据，且有一个唯一地址。当需要读内存时，将地址信号放到地址总线上传给内存，内存解析信号并定位到存储单元，然后把该存储单元上的数据放到数据总线上，回传。

写内存时:系统将要写入的数据和单元地址分别放到数据总线和地址总线上，内存读取两个总线的内容，做相应的写操作。

**内存存取效率，跟次数有关，先读取A数据还是后读取A数据不会影响存取效率。**

而磁盘存取就不一样了，磁盘I/O涉及机械操作。磁盘是由大小相同且同轴的圆形盘片组成，磁盘可以转动(各个磁盘须同时转动)。磁盘的一侧有磁头支架，磁头支架固定了一组磁头，每个磁头负责存取一个磁盘的内容。磁头不动，磁盘转动，但磁臂可以前后动，用于读取不同磁道上的数据。磁道就是以盘片为中心划分出来的一系列同心环(如图标红那圈)。磁道又划分为一个个小段，叫扇区，是磁盘的最小存储单元。  
  
磁盘读取时，系统将数据逻辑地址传给磁盘，磁盘的控制电路会解析出物理地址，即哪个磁道哪个扇区。于是磁头需要前后移动到对应的磁道，消耗的时间叫寻道时间，然后磁盘旋转将对应的扇区转到磁头下，消耗的时间叫旋转时间。所以，适当的操作顺序和数据存放可以减少寻道时间和旋转时间。  
为了尽量减少I/O操作，磁盘读取每次都会预读，大小通常为页的整数倍。即使只需要读取一个字节，磁盘也会读取一页的数据(通常为4K)放入内存，内存与磁盘以页为单位交换数据。因为局部性原理认为，通常一个数据被用到，其附近的数据也会立马被用到。

B-Tree：如果一次检索需要访问4个节点，数据库系统设计者利用磁盘预读原理，把节点的大小设计为一个页，那读取一个节点只需要一次I/O操作，完成这次检索操作，最多需要3次I/O(根节点常驻内存)。数据记录越小，每个节点存放的数据就越多，树的高度也就越小，I/O操作就少了，检索效率也就上去了。

B+Tree：非叶子节点只存key，大大滴减少了非叶子节点的大小，那么每个节点就可以存放更多的记录，树更矮了，I/O操作更少了。所以B+Tree拥有更好的性能。

注： **尽量减少随机IO**，需要一种合理高效的外存数据结构，就是下面所要重点阐述的B-tree结构，以及相关的变种结构：B+-tree结构和B\*-tree结构。

**平衡多路查找树（B-Tree）**

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**](http://lib.csdn.net/base/mysql)中可通过如下命令查看页的大小：

mysql>  show variables like 'innodb\_page\_size';

+------------------+-------+

| Variable\_name    | Value |

+------------------+-------+

| innodb\_page\_size | 16384 |

+------------------+-------+

1 row in set (0.00 sec)

* 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：   


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：

1. 根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
2. 比较关键字29在区间（17,35），找到磁盘块1的指针P2。
3. 根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
4. 比较关键字29在区间（26,30），找到磁盘块3的指针P2。
5. 根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
6. 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。

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：

1. 非叶子节点只存储键值信息。
2. 所有叶子节点之间都有一个链指针。
3. 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。