## B树与B+树

首先注意：B树就是B-树，"-"是个连字符号，不是减号。   
B-树是一种**平衡**的多路**查找(又称排序)**树，在文件系统中有所应用。主要用作文件的索引。**其中的B就表示平衡(Balance)   
  
B+树有一个最大的好处，方便扫库，B树必须用中序遍历的方法按序扫库**，而B+树直接从叶子结点挨个扫一遍就完了。   
  
**B+树支持range-query(区间查询)非常方便，而B树不支持**。这是数据库选用B+树的最主要原因。   
  
比如要查 5-10之间的，B+树一把到5这个标记，再一把到10，然后串起来就行了，B树就非常麻烦。B树的好处，就是成功查询特别有利，因为树的高度总体要比B+树矮。不成功的情况下，B树也比B+树稍稍占一点点便宜。   
  
B树的优势是当你要查找的值恰好处在一个非叶子节点时，查找到该节点就会成功并结束查询， 而B+树由于非叶节点只是索引部分，这些节点中只含有其子树中的最大(或最小)关键字，当非终端节点上的关键字等于给点值时，查找并不终止，而是继续向下直到叶子节点。 因此在B+树中，无论查找成功与否，都是走了一条从根到叶子节点的路径 。

有很多基于频率的搜索是选用B树，越频繁query的结点越往根上走，前提是需要对query做统计，而且要对key做一些变化。   
另外B树也好B+树也好，根或者上面几层因为被反复query，所以这几块基本都在内存中，不会出现读磁盘IO，一般已启动的时候，就会主动换入内存。 mysql底层存储是用B+树实现的，因为内存中B+树是没有优势的，但是一到磁盘，B+树的威力就出来了。   
B+树的优点：

       （1) B+树的磁盘读写代价更低

              B+的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部             结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的           关键字也就越多。相对来说IO读写次数也就降低了。

       （2）B+树的数据信息遍历更加方便

                B+树只要遍历叶子节点就可以实现整棵树的遍历，而B树不支持这样的操作（或者说效率太低），而且            在数据库中基于范围的查询是非常频繁的，所以数据库索引基本采用B+树

       （3) B+树的查询效率更加稳定

              由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查             找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

B\*树   
**是B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针**；B\*树定义了非叶子结点关键字个数至少为(2/3)\*M，即块的最低使用率为2/3（代替B+树的1/2）；   
B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针；   
B\*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针；   
**所以，B\*树分配新结点的概率比B+树要低，空间使用率更高**

## 二叉树与红黑树

## 2.1 二叉查找树

由于红黑树本质上就是一棵二叉查找树，所以在了解红黑树之前，咱们先来看下二叉查找树。

二叉查找树（Binary Search Tree），也称有序二叉树（ordered binary tree）,排序二叉树（sorted binary tree），是指一棵空树或者具有下列性质的二叉树：

* 若任意结点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
* 若任意结点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
* 任意结点的左、右子树也分别为二叉查找树。
* 没有键值相等的结点（no duplicate nodes）。

因为，一棵由n个结点，随机构造的二叉查找树的高度为lgn，所以顺理成章，一般操作的执行时间为O（lgn）.（至于n个结点的二叉树高度为lgn的证明，可参考算法导论 第12章 二叉查找树 第12.4节）。

但二叉树若退化成了一棵具有n个结点的线性链后，则此些操作最坏情况运行时间为O（n）。后面我们会看到一种基于二叉查找树-红黑树，它通过一些性质使得树相对平衡，使得最终查找、插入、删除的时间复杂度最坏情况下依然为O（lgn）。

## 2.2 红黑树

红黑树是牺牲了严格的高度平衡的优越条件为代价，它只要求部分地达到平衡要求，降低了对旋转的要求，从而提高了性能。红黑树能够以O(log2 n)的时间复杂度进行搜索、插入、删除操

前面我们已经说过，红黑树，本质上来说就是一棵二叉查找树，但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为O(log n)。黑树是一种自平衡的二叉查找树。

但它是如何保证一棵n个结点的红黑树的高度始终保持在h = logn的呢？这就引出了红黑树的5条性质：

1）每个结点要么是红的，要么是黑的。   
2）根结点是黑的。   
3）每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。   
4）如果一个结点是红的，那么它的俩个儿子都是黑的。   
5）对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。

正是红黑树的这5条性质，使得一棵n个结点是红黑树始终保持了logn的高度，从而也就解释了上面我们所说的“红黑树的查找、插入、删除的时间复杂度最坏为O(log n)”这一结论的原因。

什么时候会触发左旋，右旋

**红黑树从根到叶子的最长路径不会超过最短路径的2倍。当插入或删除结点的时候，红黑树的规则有可能被打破，所以需要做一些调整，这样才能继续维持平衡**

在什么情况下需要变色，在什么情况下需要旋转？  
在红黑二叉树中插入节点或删除节点后，如果破坏了红黑树的规则（也就是上述的特性），则需要对修改后的红黑树进行调整，使其重新符合红黑树的规则。首先是变色（往往需要多次变色，一次改变一个节点的颜色），当变色无能为力时，就使用旋转，旋转一次之后，然后再继续多次变色，如此反复循环，直到修改后的红黑树重新符合规则。

调整的方法有两种：变色和旋转，旋转又包括左旋转和右旋转

变色：为了重新符合红黑树的规则，尝试把红色结点变为黑色，或者把黑色结点变为红色。

旋转:

左旋转：

逆时针旋转红黑树的两个结点，使得父结点被自己的右孩子取代，而自己成为自己的左孩子

右旋转：

顺时针旋转红黑树的两个结点，使得父结点被自己的左孩子取代，而自己成为自己的右孩子

**树的旋转知识**

当我们在对红黑树进行插入和删除等操作时，对树做了修改，那么可能会违背红黑树的性质。

为了继续保持红黑树的性质，我们可以通过对结点进行重新着色，以及对树进行相关的旋转操作，即修改树中某些结点的颜色及指针结构，来达到对红黑树进行插入或删除结点等操作后，继续保持它的性质或平衡。

树的旋转，分为左旋和右旋。

左旋/右旋：

当在某个结点pivot上，做左旋操作时，我们假设它的右孩子y不是NIL[T]，pivot可以为任何不是NIL[T]的左孩子结点。

左旋以pivot到y之间的链为“支轴”进行，它使y成为该孩子树新的根，而y的左孩子b则成为pivot的右孩子。

对于树的旋转，能保持不变的只有原树的搜索性质，而原树的红黑性质则不能保持，在红黑树的数据插入和删除后可利用旋转和颜色重涂来恢复树的红黑性质。

**红黑树相对于哈希表，在选择使用的时候有什么依据？**

权衡三个因素: 查找速度, 数据量, 内存使用，可扩展性。