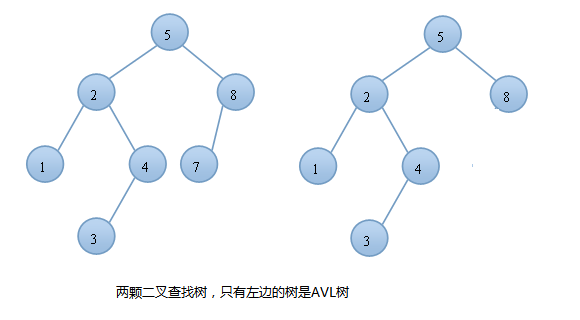
# 树结构（AVL树,2-3树,红黑树,B树,B+树）

## AVL树（平衡二叉树）

AVL树本质上是一颗二叉查找树，但是它又具有以下特点：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。在AVL树中任何节点的两个子树的高度最大差别为一，所以它也被称为平衡二叉树。下面是平衡二叉树和非平衡二叉树对比的例图：

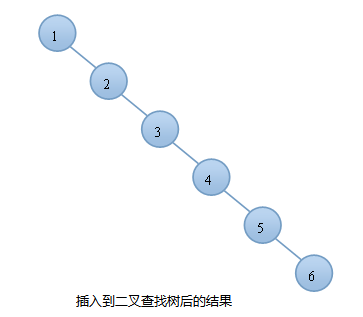
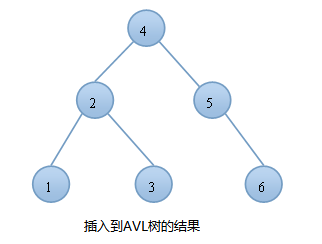


平衡因子(bf)：结点的左子树的深度减去右子树的深度，那么显然-1<=bf<=1;

## AVL树的作用：

我们知道，对于一般的二叉搜索树（Binary Search Tree），其期望高度（即为一棵平衡树时）为log2n，其各操作的时间复杂度（O(log2n)）同时也由此而决定。但是，在某些极端的情况下（如在插入的序列是有序的时），二叉搜索树将退化成近似链或链，此时，其操作的时间复杂度将退化成线性的，即O(n)。我们可以通过随机化建立二叉搜索树来尽量的避免这种情况，但是在进行了多次的操作之后，由于在删除时，我们总是选择将待删除节点的后继代替它本身，这样就会造成总是右边的节点数目减少，以至于树向左偏沉。这同时也会造成树的平衡性受到破坏，提高它的操作的时间复杂度。

　　例如：我们按顺序将一组数据1,2,3,4,5,6分别插入到一颗空二叉查找树和AVL树中，插入的结果如下图：

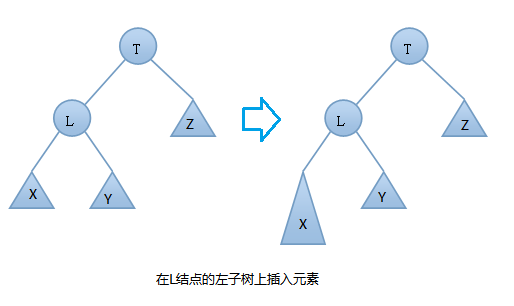
　　由上图可知，同样的结点，由于插入方式不同导致树的高度也有所不同。特别是在带插入结点个数很多且正序的情况下，会导致二叉树的高度是O(N)，而AVL树就不会出现这种情况，树的高度始终是O(lgN).高度越小，对树的一些基本操作的时间复杂度就会越小。这也就是我们引入AVL树的原因

## AVL树的基本操作：

AVL树的操作基本和二叉查找树一样，这里我们关注的是两个变化很大的操作：插入和删除！

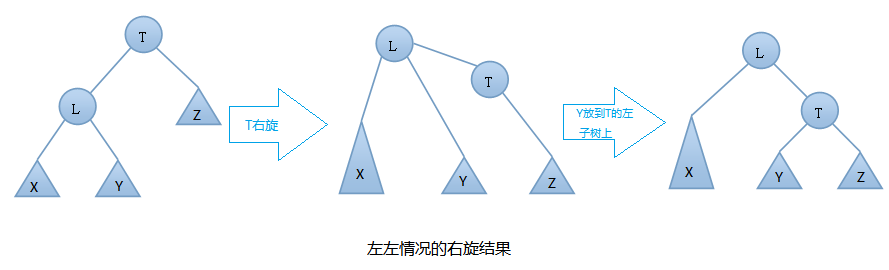
　　我们知道，AVL树不仅是一颗二叉查找树，它还有其他的性质。如果我们按照一般的二叉查找树的插入方式可能会破坏AVL树的平衡性。同理，在删除的时候也有可能会破坏树的平衡性，所以我们要做一些特殊的处理，包括：单旋转和双旋转！

### ****AVL树的插入，单旋转的第一种情况---右旋：****

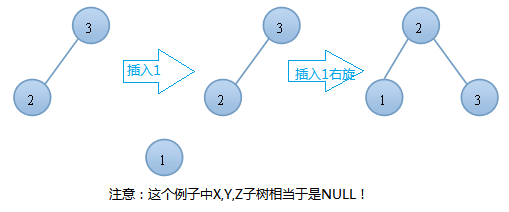


　　由上图可知：在插入之前树是一颗AVL树，而插入之后结点T的左右子树高度差的绝对值不再 < 1,此时AVL树的平衡性被破坏，我们要对其进行旋转。由上图可知我们是在结点T的左结点的左子树上做了插入元素的操作，我们称这种情况为左左情况，我们应该进行右旋转(只需旋转一次，故是单旋转)。具体旋转步骤是：

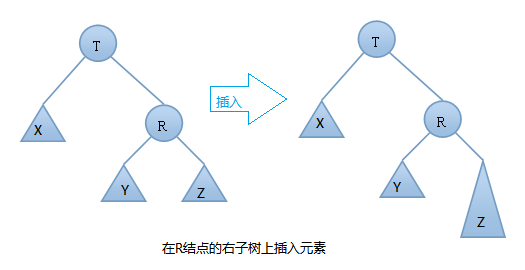
　　T向右旋转成为L的右结点，同时，Y放到T的左孩子上。这样即可得到一颗新的AVL树，旋转过程图如下：



 左左情况的右旋举例：

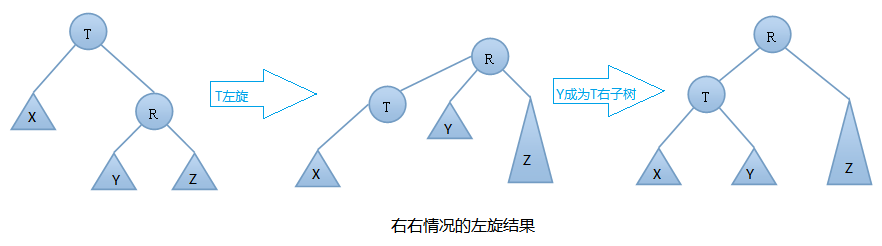


### AVL树的插入，单旋转的第一种情况---左旋：



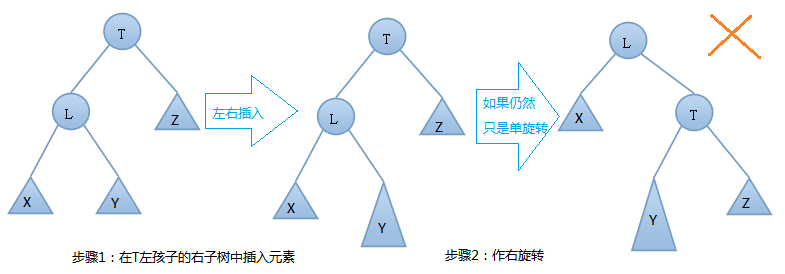
　　　由上图可知：在插入之前树是一颗AVL树，而插入之后结点T的左右子树高度差的绝对值不再 < 1,此时AVL树的平衡性被破坏，我们要对其进行旋转。由上图可知我们是在结点T的右结点的右子树上做了插入元素的操作，我们称这种情况为右右情况，我们应该进行左旋转(只需旋转一次，故事单旋转)。具体旋转步骤是：

　　　T向右旋转成为R的左结点，同时，Y放到T的左孩子上。这样即可得到一颗新的AVL树，旋转过程图如下：

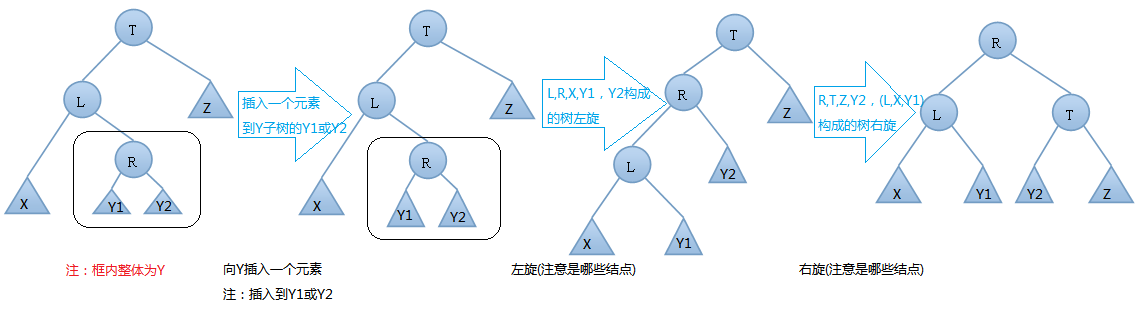


以上就是插入操作时的单旋转情况！我们要注意的是：谁是T谁是L，谁是R还有谁是X,Y,Z!T始终是开始不平衡的左右子树的根节点。显然L是T的左结点，R是T的右节点。X、Y、Y是子树当然也可以为NULL.NULL归NULL，但不能破坏插入时我上面所说的左左情况或者右右情况。

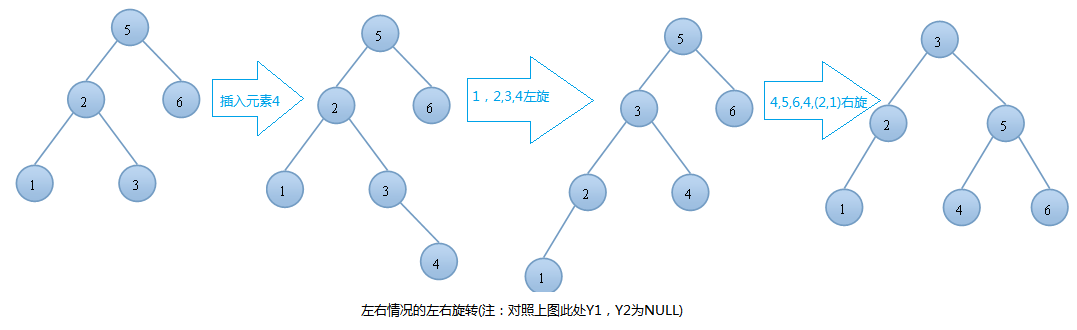
### AVL树的插入，双旋转的第一种情况---左右(先左后右)旋：

****

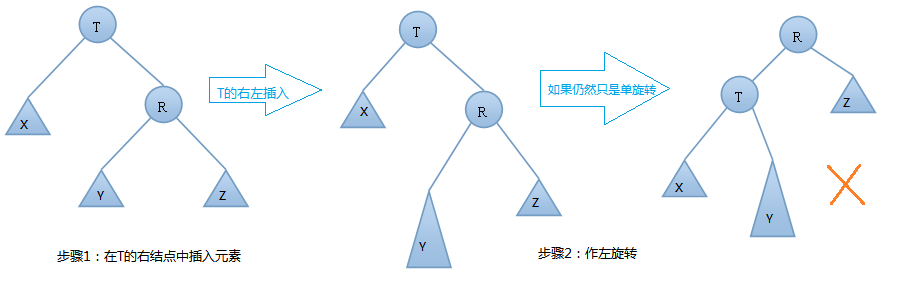
由上图可知，我们在T结点的左结点的右子树上插入一个元素时，会使得根为T的树的左右子树高度差的绝对值不再 < 1，如果只是进行简单的右旋，得到的树仍然是不平衡的。我们应该按照如下图所示进行二次旋转：



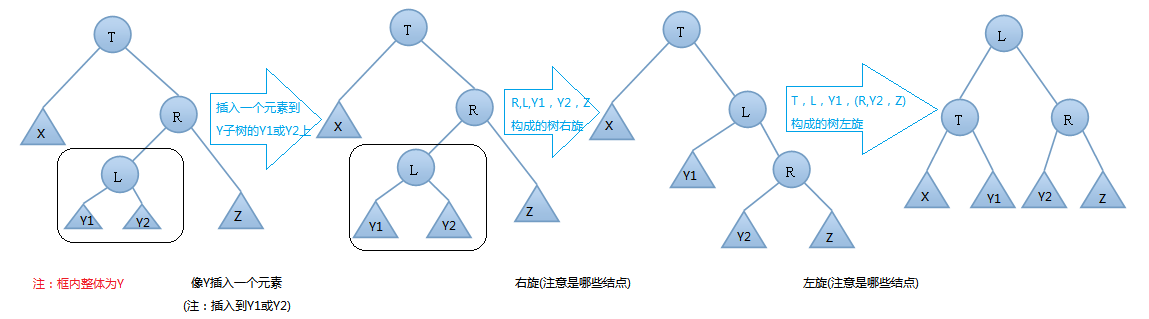
　　左右情况的左右旋转实例：



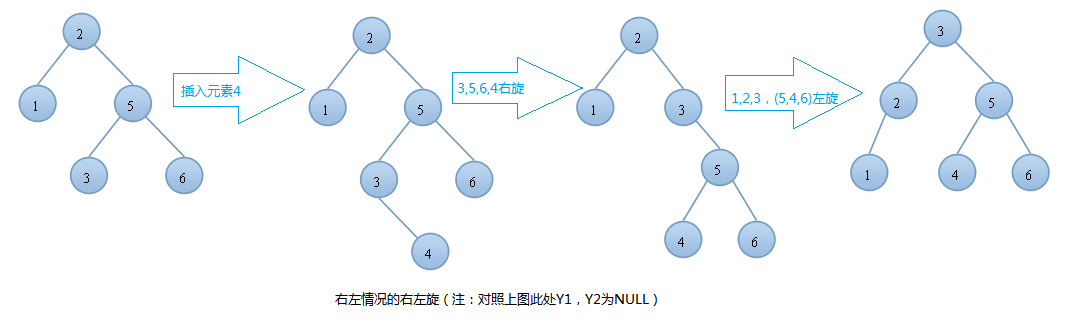
### AVL树的插入，双旋转的第二种情况---右左(先右后左)旋：

****

　　由上图可知，我们在T结点的右结点的左子树上插入一个元素时，会使得根为T的树的左右子树高度差的绝对值不再 < 1，如果只是进行简单的左旋，得到的树仍然是不平衡的。我们应该按照如下图所示进行二次旋转：



　　右左情况的右左旋转实例：

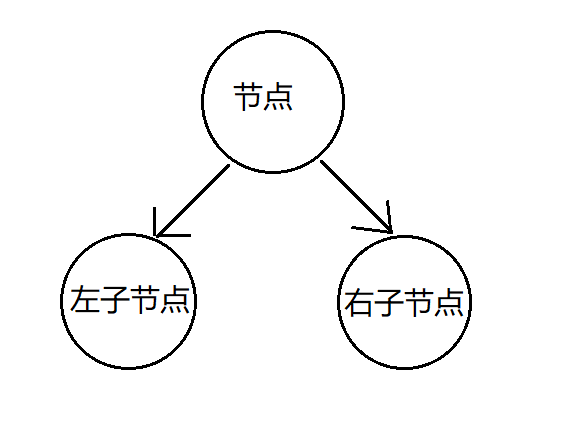


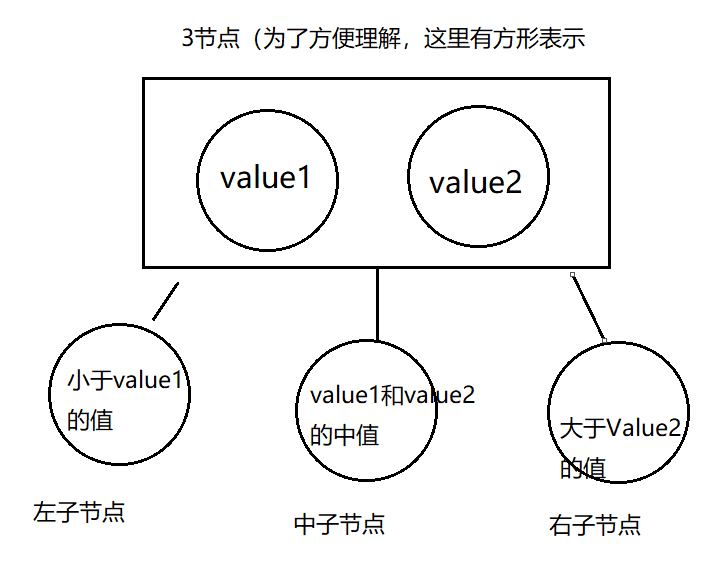
## 2-3树

数据结构是为了减少查询、删除的时间复杂度和空间复杂度。

链表的一个节点是由：节点值、节点的下一个节点（字节点）的地址  
二叉查找树一个节点：节点值、节点的左子节点的地址、节点的右子节点的地址  
2-3树：由2节点、3节点组成

2节点：



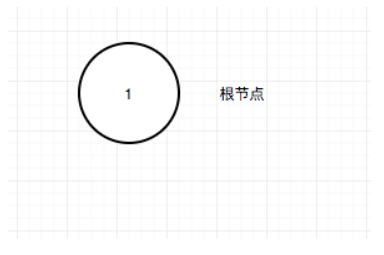
3节点：  


### 插入

2-3树的插入有以下几种情况：

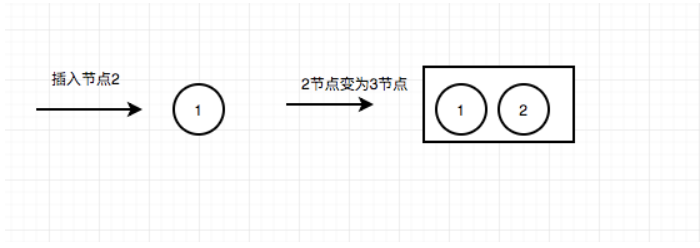
#### 树为空

插入的节点作为根节点并且为2节点。



#### 2节点中插入

插入的节点为2节点，2节点转化为3节点。

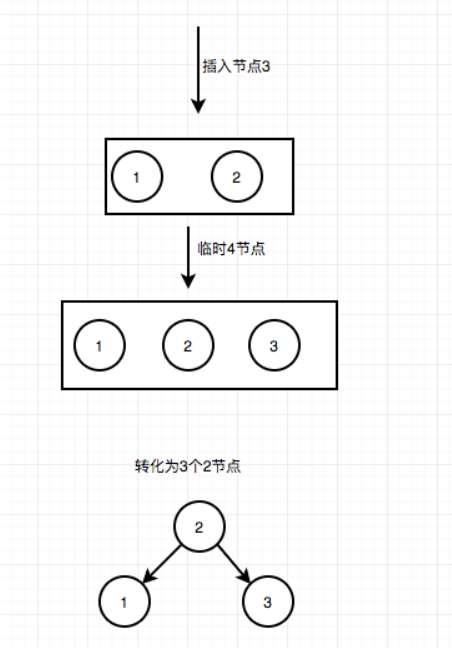


#### 3节点中插入

插入的节点为3节点，则临时调整3节点为4节点（三个值），4节点中的中值变为左右值的父。

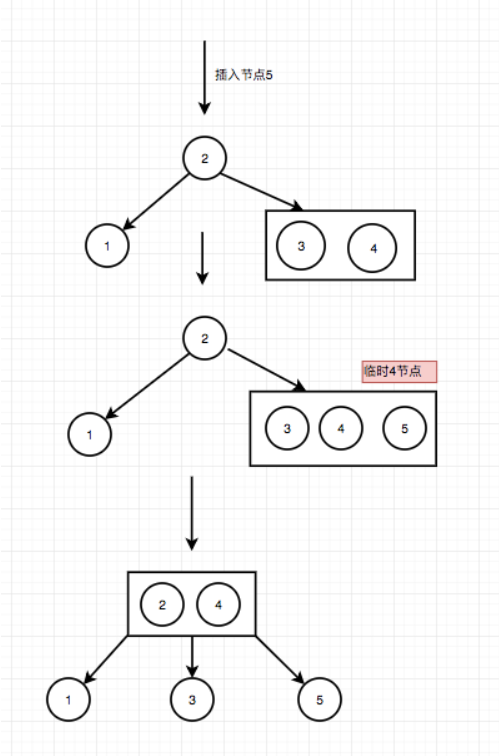
##### 父节点为null

则4节点，转化为3个2节点。



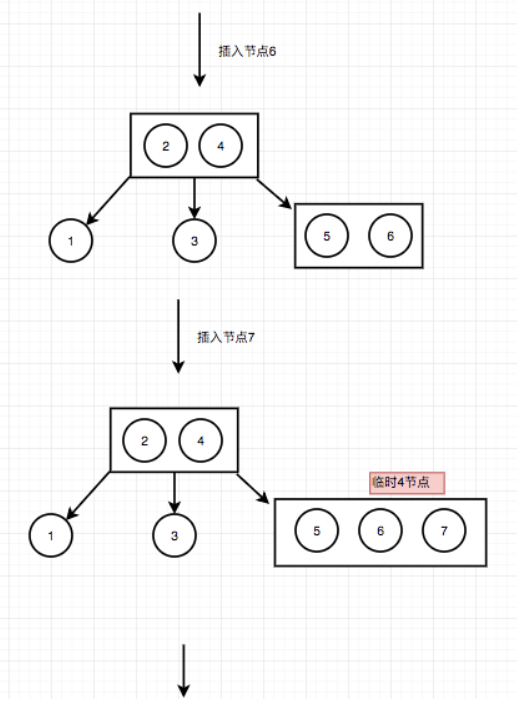
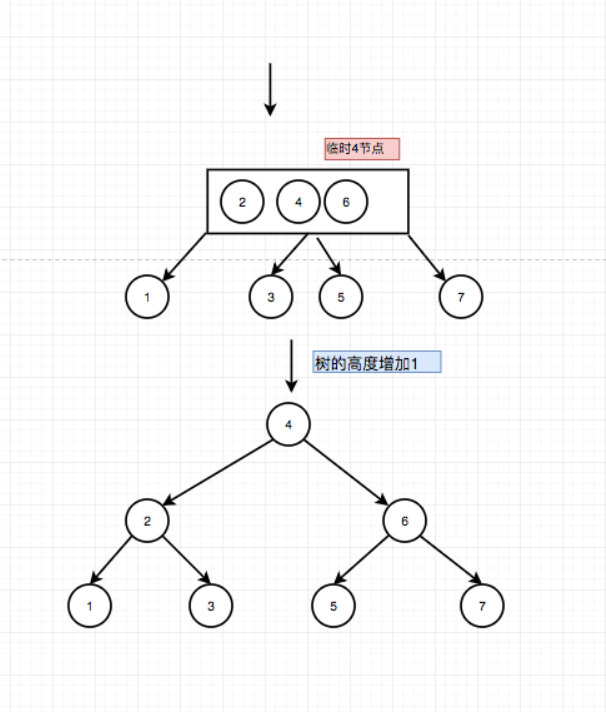
##### 父为2节点

则4节点的中子转化为父节点的值，父节点转化为3节点

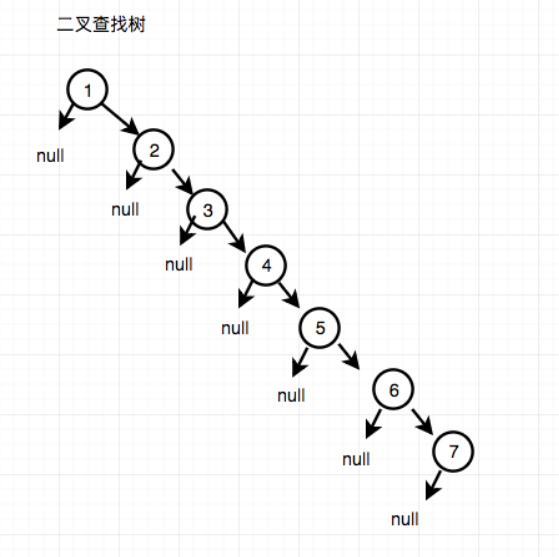


##### 父为3节点

则3节点的中子节点转为父节点的值，父节点临时转化为4节点，此时就变成了插入的节点为3节点的情况上面的两种情况。

 ------》 

## 对比二叉树和2-3树，从1递增插入10，对比树的形状



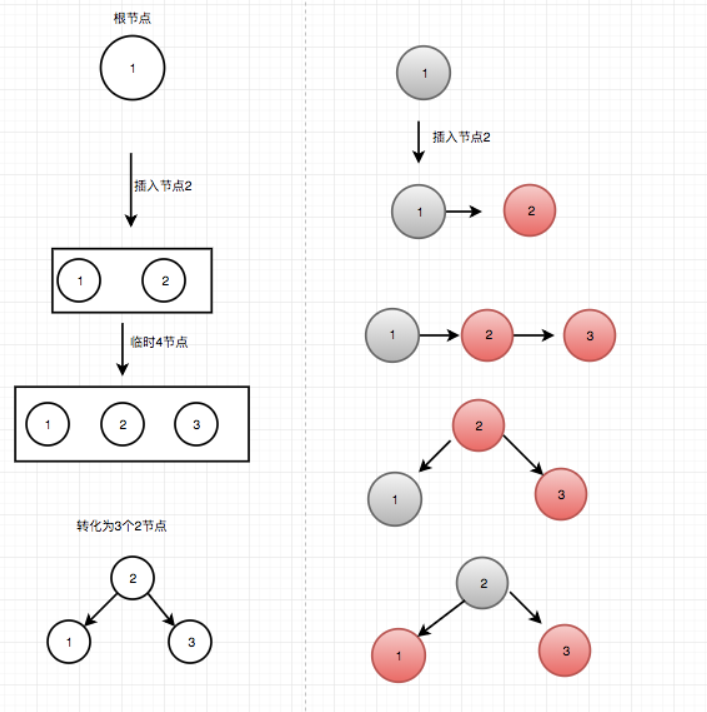
通过对比，2-3和二叉树的上图的插入，可以看到2-3最终是会自平衡的，而二叉查找树最坏会变成链表的形式(长的像链表).

## 红黑树

红黑树其实是2-3树的一种只含2节点的表现形式。还是二叉树节点大于左子节点，小于右子节点。

我们把2-3树中的2节点用黑色表示，3节点用红色表示（3节点的左节点为黑色、右节点为红色）

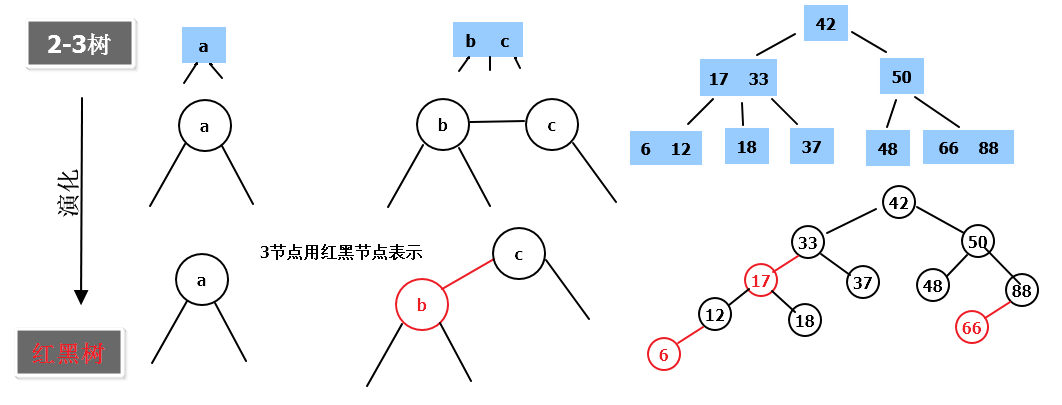
将红色链接画平就是2-3树。



### 红黑树的性质

1. 每个节点要么是红色，要么是黑色（2-3树节点要么是2节点要么是3节点）
2. 根节点必须是黑色
3. 红色节点不能连续（红色节点的子和父不能为红）（不能有4节点。注意2-3的4节点是临时的）
4. 对于每个节点，从该点至null（树的尾端）的任何路径，都含有相同个数的黑色节点。

### 2-3树演化成红黑树



### 为什么有了AVL还需要有红黑树？

红黑树并没有像AVL追求平衡，他不像AVL要求每个节点的平衡因子绝对值必须小于等于1。这样相对于AVL来说红黑树的旋转操作会更少，例如删除，插入节点操作，AVL是要从删除，增加节点到根节点的所有节点进行平衡旋转(O(logn))，而红黑树最多只需要3次就可以实现平衡O(1)(虽然通过上文实现的红黑树并不能做到，但有实现是可以的)，所以红黑树更适合增删多的场景。

所以，在增删多的场景适合红黑树，查找多的场景适合AVL

## B树

B树和2-3树的原理相同，B树也可以是2-4树，2-M树 关于B树有如下的定义，如果一棵B树有M阶（层）：

* 根节点至少有两个孩子节点
* 每个节点有M-1个关键字key（节点中的每一个元素叫关键字），并且以升序排列
* 除去叶子节点和根节点其它节点至少有M/2个孩子节点

我们都知道二叉查找树的查找的时间复杂度是Ｏ(log N)，其查找效率已经足够高了，那为什么还有Ｂ树和Ｂ＋树的出现呢？难道它两的时间复杂度比二叉查找树还小吗？

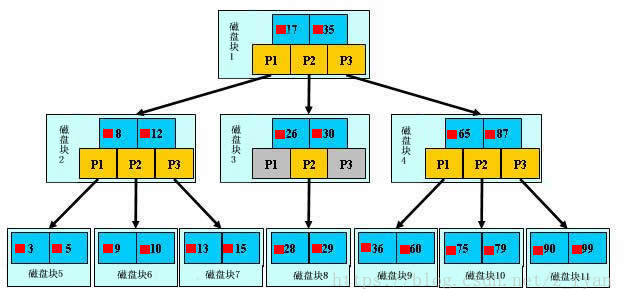
　　答案当然不是，Ｂ树和Ｂ＋树的出现是因为另外一个问题，那就是磁盘ＩＯ；众所周知，ＩＯ操作的效率很低，那么，当在大量数据存储中，查询时我们不能一下子将所有数据加载到内存中，只能逐一加载磁盘页，每个磁盘页对应树的节点。造成大量磁盘ＩＯ操作（最坏情况下为树的高度）。平衡二叉树由于树深度过大而造成磁盘IO读写过于频繁，进而导致效率低下。

　　所以，我们为了减少磁盘ＩＯ的次数，就你必须降低树的深度，将“瘦高”的树变得“矮胖”。一个基本的想法就是：

　　（1）、每个节点存储多个元素

　　（2）、摒弃二叉树结构，采用多叉树

这样就引出来了一个新的查找树结构 ——多路查找树。 根据AVL给我们的启发，一颗平衡多路查找树(B~树)自然可以使得数据的查找效率保证在O(logN)这样的对数级别上



### 查询

　　以上图为例：若查询的数值为５：

　　第一次磁盘ＩＯ：在内存中定位（与17、35比较），比17小，左子树；

　　第二次磁盘ＩＯ：在内存中定位（与８、12比较），比８小，左子树；

　　第三次磁盘ＩＯ：在内存中定位（与3、5比较），找到5，终止。

整个过程中，我们可以看出：比较的次数并不比二叉查找树少，尤其适当某一节点中的数据很多时，但是磁盘IO的次数却是大大减少。比较是在内存中进行的，相比于磁盘IO的速度，比较的耗时几乎可以忽略。所以当树的高度足够低的话，就可以极大的提高效率。相比之下，节点中的元素多点也没关系，仅仅是多了几次内存交互而已，只要不超过磁盘页的大小即可。

### 插入

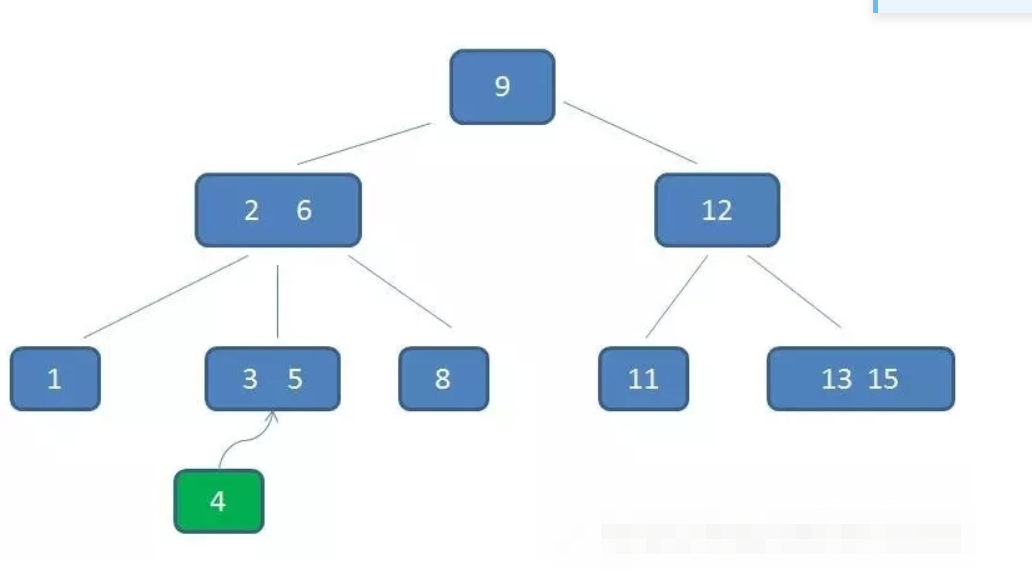
　　对高度为ｋ的m阶B树，新结点一般是插在叶子层。通过检索可以确定关键码应插入的结点位置。然后分两种情况讨论：

　　1、 若该结点中关键码个数小于m-1，则直接插入即可。

　　2、 若该结点中关键码个数等于m-1，则将引起结点的分裂。以中间关键码为界将结点一分为二，产生一个新结点，并把中间关键码插入到父结点(ｋ-1层)中

　　重复上述工作，最坏情况一直分裂到根结点，建立一个新的根结点，整个B树增加一层。

例如：在下面的B树中插入key：4 （高度为3的3阶B树）



第一步：检索key插入的节点位置如上图所示，在3,5之间；

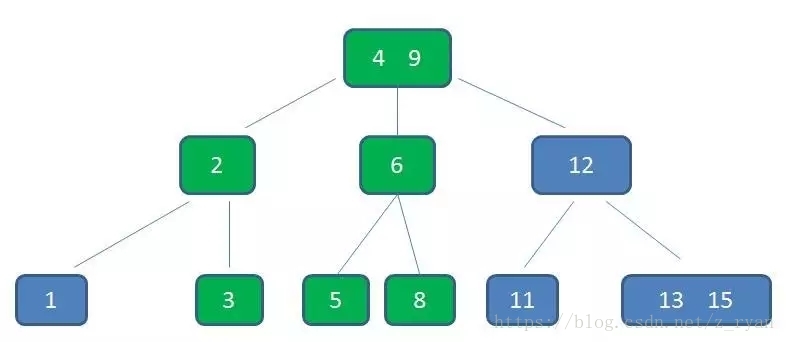
第二步：判断节点中的关键码个数：

　　节点3，5已经是两元素节点，无法再增加。父亲节点 2， 6 也是两元素节点，也无法再增加。根节点9是单元素节点，可以升级为两元素节点。；

第三步：结点分裂：

　　拆分节点3，5与节点2，6，让根节点9升级为两元素节点4，9。节点6独立为根节点的第二个孩子。

最终结果如下图：虽然插入比较麻烦，但是这也能确保Ｂ树是一个自平衡的树

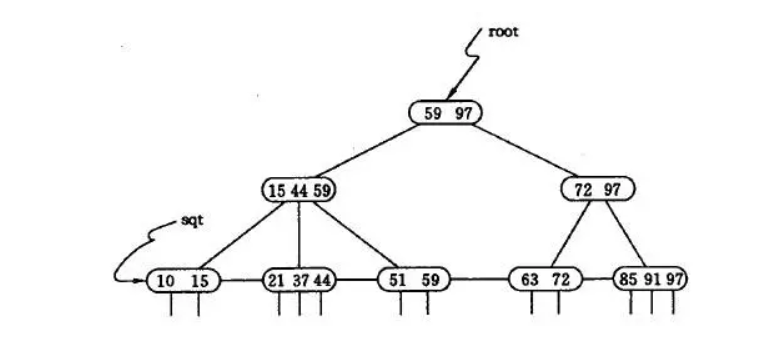
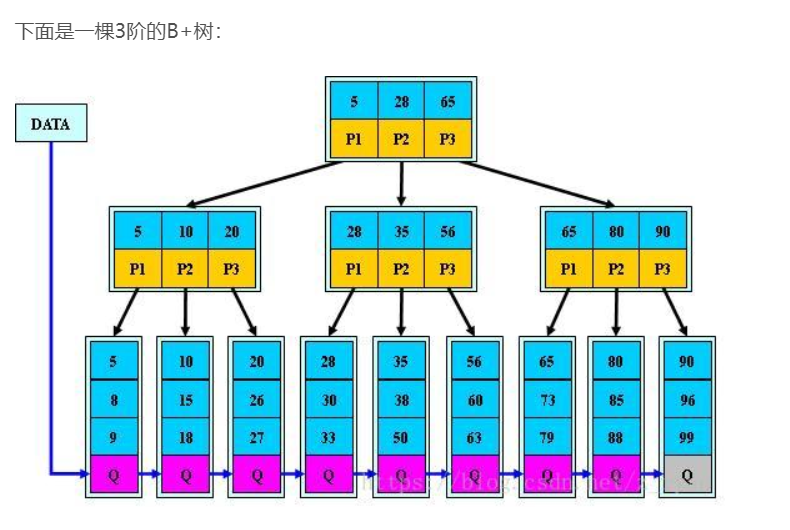


## B+树

B+树和B树类似，但多了几条规则

* 非叶子结点的子树指针个数与关键字（节点中的元素个数）个数相同
* 非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）
* 所有叶子结点有一个链指针
* 所有关键字都在叶子结点出现
* 只有叶子节点有Data域

如图



可以看到最主要的区别就是在叶子节点，叶子节点通过一个sqt将所有叶子节点链成一个链表，并且只有叶子节点有data域（data域就是索引指向的磁盘地址）。

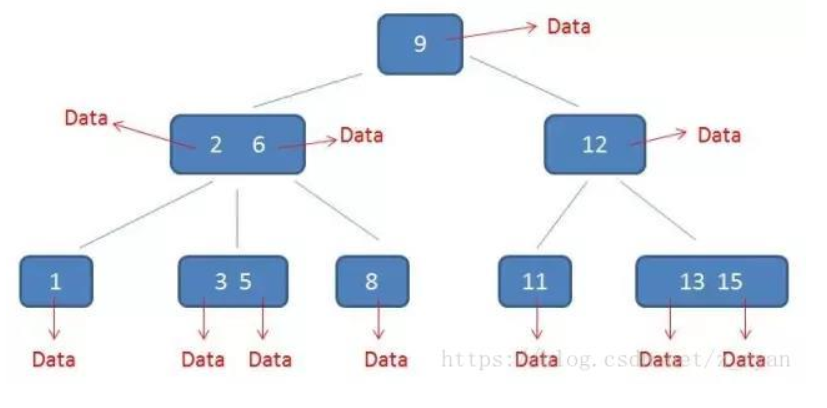
### 查找

　　B+树的优势在于查找效率上，下面我们做一具体说明：

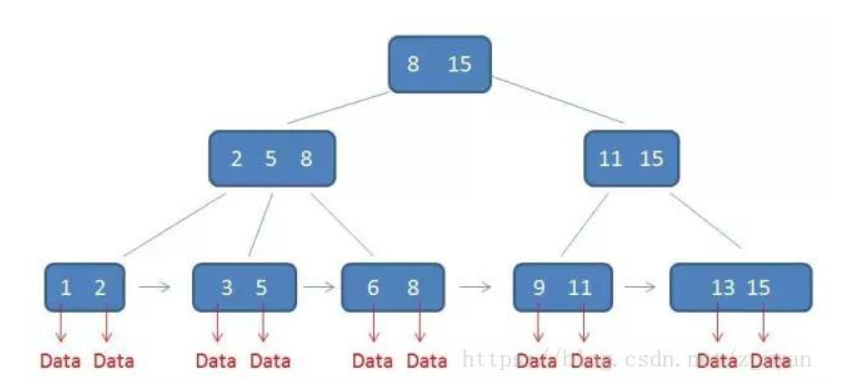
　　首先，Ｂ＋树的查找和Ｂ树一样，类似于二叉查找树。起始于根节点，自顶向下遍历树，选择其分离值在要查找值的任意一边的子指针。在节点内部典型的使用是二分查找来确定这个位置。

　　（1）、不同的是，Ｂ＋树中间节点没有卫星数据（索引元素所指向的数据记录），只有索引，而Ｂ树每个结点中的每个关键字都有卫星数据；这就意味着同样的大小的磁盘页可以容纳更多节点元素，在相同的数据量下，Ｂ＋树更加“矮胖”，ＩＯ操作更少

#### B树的卫星数据：



#### B+树的卫星数据：



需要补充的是，在数据库的聚集索引（Clustered Index）中，叶子节点直接包含卫星数据。在非聚集索引（NonClustered Index）中，叶子节点带有指向卫星数据的指针。

　　（2）其次，因为卫星数据的不同，导致查询过程也不同；Ｂ树的查找只需找到匹配元素即可，最好情况下查找到根节点，最坏情况下查找到叶子结点，所说性能很不稳定，而Ｂ＋树每次必须查找到叶子结点，性能稳定

　　（3）、在范围查询方面，B+树的优势更加明显

　　B树的范围查找需要不断依赖中序遍历。首先二分查找到范围下限，在不断通过中序遍历，知道查找到范围的上限即可。整个过程比较耗时。

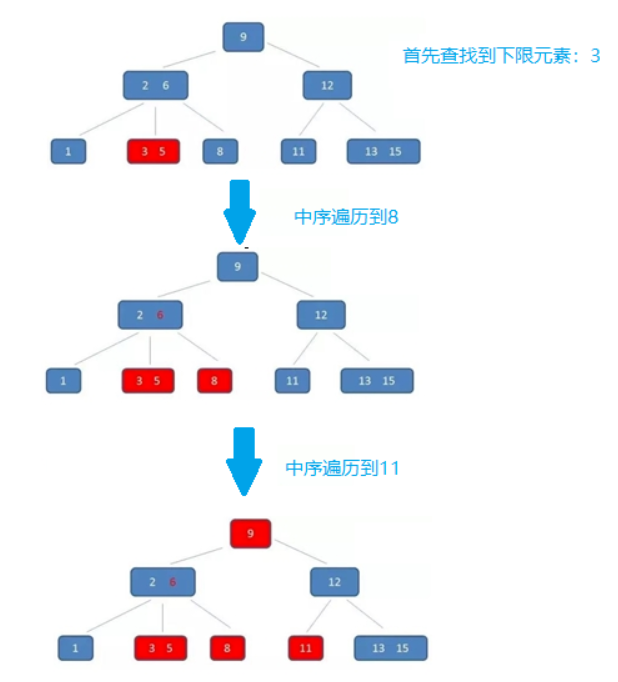
而B+树的范围查找则简单了许多。首先通过二分查找，找到范围下限，然后同过叶子结点的链表顺序遍历，直至找到上限即可，整个过程简单许多，效率也比较高。

B树的范围查找需要不断依赖中序遍历。首先二分查找到范围下限，在不断通过中序遍历，知道查找到范围的上限即可。整个过程比较耗时。

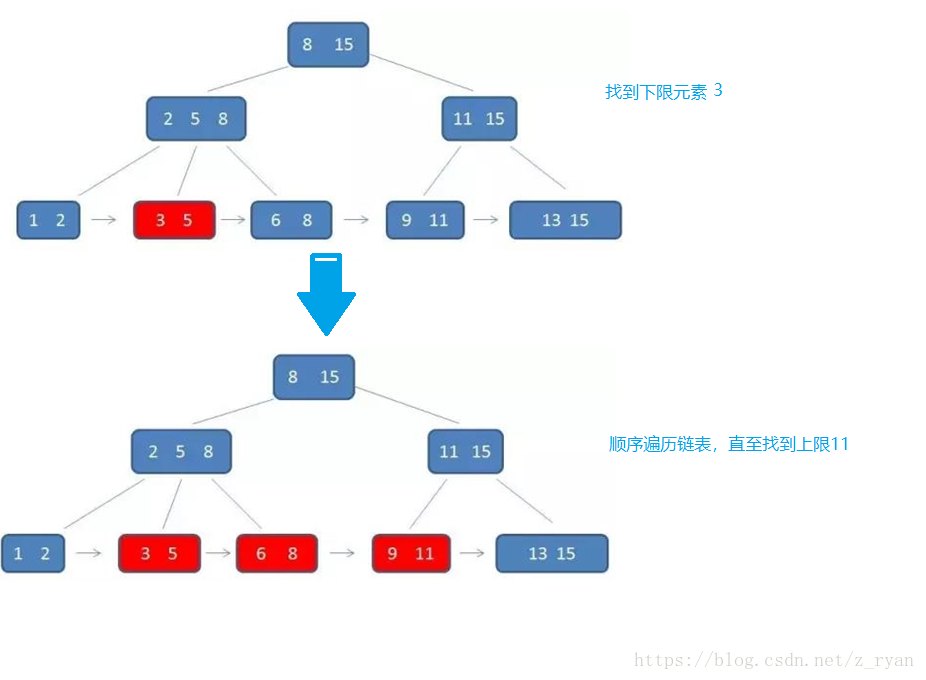
　　而B+树的范围查找则简单了许多。首先通过二分查找，找到范围下限，然后同过叶子结点的链表顺序遍历，直至找到上限即可，整个过程简单许多，效率也比较高。

　　例如：同样查找范围[3-11]，两者的查询过程如下：

　　B树的查找过程：



B+树的查找过程：



### 插入

　　 B+树的插入与B树的插入过程类似。不同的是B+树在叶结点上进行，如果叶结点中的关键码个数超过m，就必须分裂成关键码数目大致相同的两个结点，并保证上层结点中有这两个结点的最大关键码。

### 删除

B+树中的关键码在叶结点层删除后，其在上层的复本可以保留，作为一个”分解关键码”存在，如果因为删除而造成结点中关键码数小于ceil(m/2)，其处理过程与B-树的处理一样。在此，我就不多做介绍了。

### 总结

B+树相比B树的优势：

1.单一节点存储更多的元素，使得查询的IO次数更少；

2.所有查询都要查找到叶子节点，查询性能稳定；

　3.所有叶子节点形成有序链表，便于范围查询。

### mysql的Innodb引擎采用B+树的索引方式

总结了这些树形结构，再对mysql中innodb存储引擎为什么使用B+树作为索引做个总结

#### 为什么不用AVL或红黑树？

我们假设B+树一个节点可以有100个关键字，那么3层的B树可以容纳大概1000000多个关键字（100+101\*100+101\*101\*100）。而红黑树要存储这么多至少要20层。所以使用B树相对于红黑树和AVL可以减少IO操作

#### 为什么不用数组或链表？

链表查询速度慢。数组要开辟连续空间不可能。

#### 为什么不用哈希表？

Innodb能够进行范围查询，哈希表无法实现。例如LIKE很难用哈希表实现

#### 为什么不用B树？

B+树只有叶子节点存放数据，而其他节点只存放索引，而B树每个节点都有Data域。所以相同大小的节点B+树包含的索引比B树的索引更多（因为B树每个节点还有Data域）

还有就是B+树的叶子节点是通过链表连接的，所以找到下限后能很快进行区间查询，比B树中序遍历快