# 一、二叉排序树

## 1.基本概念

二叉排序树要么是一棵空树，要么具有以下性质：

1. 若根节点的左子树不为空，则左子树上的所有节点的值均小于根节点的值；若根节点的右子树不为空，则右子树上的所有节点的值均大于根节点的值；
2. 左、右子树本身也是一棵二叉排序树

构造一棵二叉排序树的过程就是逐渐插入节点的过程。

## 2.算法分析

1.查找效率最好O(logn)，最坏O(n) 。当它是一棵平衡二叉树是即为O(logn)，当它只有左子树或只有右子树时即为O(n)

2.插入效率和查找效率相同（只插入叶子节点）

3.删除效率最好O(logn)+O(1)->只有左子树或者右子树 最差O(logn)+O(logn)->左子树和右子树同时存在

# 二、平衡二叉树

## 1.基本概念

平衡二叉树要么是一棵空树，要么是一棵具有以下性质的二叉排序树：它的左右子树的深度之差不超过1，并且左、右子树都是平衡二叉树。

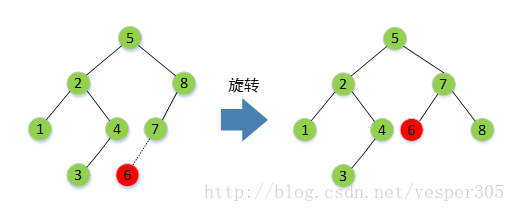
若二叉排序树中存在平衡因子的绝对值大于1的节点，那么该二叉排序树就不是平衡二叉树。

## 2.基本操作

（1）结点插入

1）插入原理

根据二叉平衡树的定义，一定保持左右子树深度绝对值小于1.在平衡二叉树插入工作一定考虑深度差，在AVL树进行插入工作时候，困难在于可能破坏AVL树的平衡属性。例如在下图



上图中插入一个节点6，那么如果不进行后续处理就会破坏树的平衡性。因为8的左子树深度为1，而右子树深度为-1.

针对此类问题，需要根据树的实际结构进行**几种简单的旋转（rotation）操作**就可以让树恢复AVL树的平衡性质

2）旋转问题

对于一个平衡的节点，由于任意节点最多有两个儿子，因此高度不平衡时，此节点的两颗子树的高度差2.容易看出，这种不平衡出现在下面四种情况：



1、6节点的左子树3节点高度比右子树7节点大2，左子树3节点的左子树1节点高度大于右子树4节点，这种情况成为左左。

2、6节点的左子树2节点高度比右子树7节点大2，左子树2节点的左子树1节点高度小于右子树4节点，这种情况成为左右。

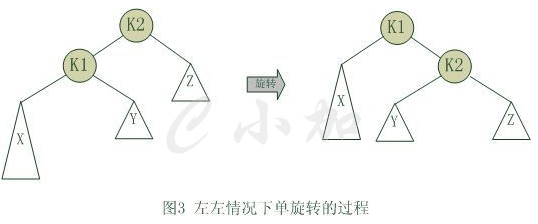
3、2节点的左子树1节点高度比右子树5节点小2，右子树5节点的左子树3节点高度大于右子树6节点，这种情况成为右左。

4、2节点的左子树1节点高度比右子树4节点小2，右子树4节点的左子树3节点高度小于右子树6节点，这种情况成为右右。

从图2中可以可以看出，1和4两种情况是对称的，这两种情况的旋转算法是一致的，只需要经过一次旋转就可以达到目标，我们称之为单旋转。2和3两种情况也是对称的，这两种情况的旋转算法也是一致的，需要进行两次旋转，我们称之为双旋转。

3）旋转操作

**单旋转**是针对于左左和右右这两种情况的解决方案，这两种情况是对称的，只要解决了左左这种情况，右右就很好办了。图3是左左情况的解决方案，节点k2不满足平衡特性，因为它的左子树k1比右子树Z深2层，而且k1子树中，更深的一层的是k1的左子树X子树，所以属于左左情况。



为使树恢复平衡，我们把k2变成这棵树的根节点，因为k2大于k1，把k2置于k1的右子树上，而原本在k1右子树的Y大于k1，小于k2，就把Y置于k2的左子树上，这样既满足了二叉查找树的性质，又满足了平衡二叉树的性质。

这样的操作只需要一部分指针改变，结果我们得到另外一颗二叉查找树，它是一棵AVL树，因为X向上一移动了一层，Y还停留在原来的层面上，Z向下移动了一层。整棵树的新高度和之前没有在左子树上插入的高度相同，插入操作使得X高度长高了。因此，由于这颗子树高度没有变化，所以通往根节点的路径就不需要继续旋转了。

对于左右和右左这两种情况，单旋转不能使它达到一个平衡状态，要经过两次旋转。**双旋转**是针对于这两种情况的解决方案，同样的，这样两种情况也是对称的，只要解决了左右这种情况，右左就很好办了。图4是左右情况的解决方案，节点k3不满足平衡特性，因为它的左子树k1比右子树Z深2层，而且k1子树中，更深的一层的是k1的右子树k2子树，所以属于左右情况。



## 3.算法分析

**(1) 查找代价：**

AVL是严格平衡的BST（平衡因子不超过1）。那么查找过程与BST一样，只是AVL不会出现最差情况的BST(单支树)。因此查找效率最好，最坏情况都是O(logN)数量级的。

**(2) 插入代价：**

AVL必须要保证严格平衡(|bf|<=1)，那么每一次插入数据使得AVL中某些结点的平衡因子超过1就必须进行旋转操作。**事实上，AVL的每一次插入结点操作最多只需要旋转1次(单旋转或双旋转)**。因此，总体上插入操作的代价仍然在O(logN)级别上(插入结点需要首先查找插入的位置)。

**(3) 删除代价：**

AVL删除结点的算法可以参见BST的删除结点，但是删除之后必须检查从删除结点开始到根结点路径上的所有结点的平衡因子。因此删除的代价稍微要大一些。每一次删除操作最多需要O(logN)次旋转。因此，删除操作的时间复杂度为O(logN)+O(logN)=O(2logN)

**AVL 效率总结 :**

**查找**的时间复杂度维持在O(logN)，不会出现最差情况   
AVL树在执行每个**插入**操作时最多需要1次旋转，其时间复杂度在O(logN)左右。   
AVL树在执行**删除**时代价稍大，执行每个删除操作的时间复杂度需要O(2logN)。

# 三、最优二叉树（赫夫曼树）

## 基本概念

路径：从树中一个节点到另一个节点之间的分支构成这两个节点之间的路径

路径长度:路径上的的分支数目

树的路径长度：从树根到每一个节点的路径长度之和

节点的带权路径长度：从节点到树根之间的路径长度乘以节点的权值

树的带权路径长度：树中所有叶子节点的带权路径长度之和

最优二叉树：假设有n个权值{w1,w2,w3,....wn}，构造一棵有n个叶子节点的二叉树，使得每个叶子节点的权值为wi，则其中带权路径长度最小的二叉树为最有二叉树或者赫夫曼树。

特点：赫夫曼树中没有度为1的节点，因此一棵具有n个叶子节点的赫夫曼树共有2n-1个节点。

## 基本构造

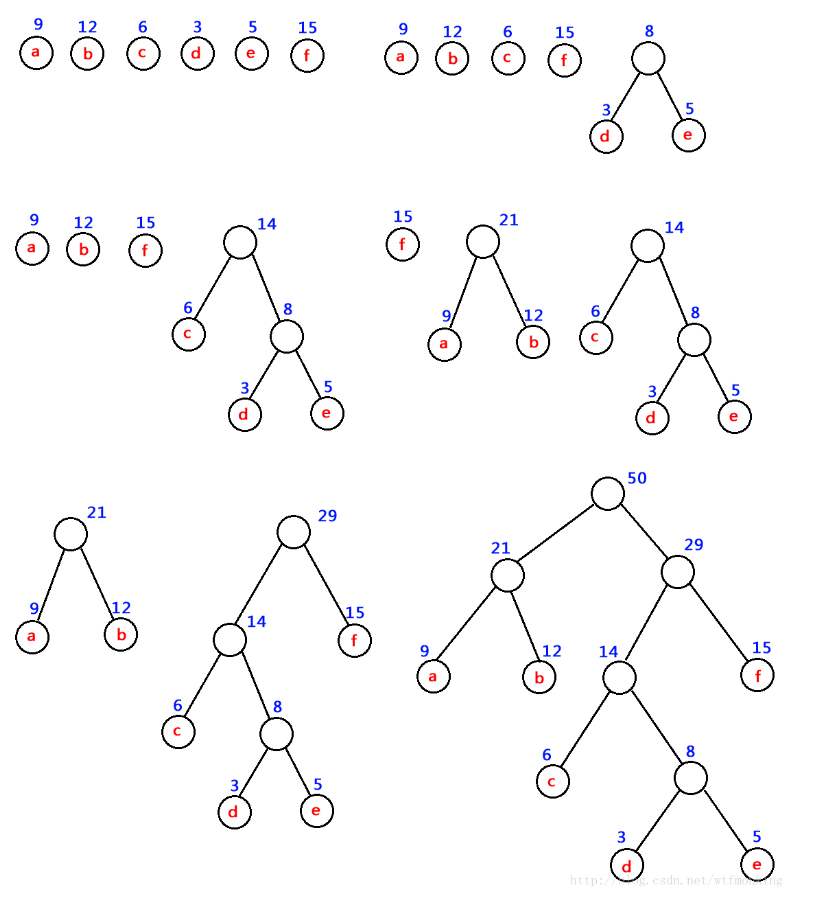
(1) 将w1、w2、…，wn看成是有n 棵树的森林(每棵树仅有一个结点)；

(2) 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；

(3)从森林中删除选取的两棵树，并将新树加入森林；

(4)重复(2)、(3)步，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。

例子：下图中的六个带权叶子结点来构造一棵哈夫曼树，步骤如下：



## 3.应用场景

应用：如何得到电文总长度最短的二进制前缀编码？（以电文中字符出现频率作为权重，构造赫夫曼树，左分支表示0，右分支表示1）

# 四、堆排序

## 1.基本概念

堆通常是一个可以被看做一棵树的数组对象。堆总是满足下列性质：

（1）堆中某个节点的值总是不大于或不小于其父节点的值；

（2）堆总是一棵完全二叉树(完全二叉树：若设二叉树的深度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。）

将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。

若将和此次序列对应的一维数组（即以一维数组作此序列的存储结构）看成是一个完全二叉树，则堆的含义表明，完全二叉树中所有非终端结点的值均不大于（或不小于）其左、右孩子结点的值。由此，若序列{k1,k2,…,kn}是堆，则堆顶元素（或完全二叉树的根）必为序列中n个元素的最小值（或最大值）。

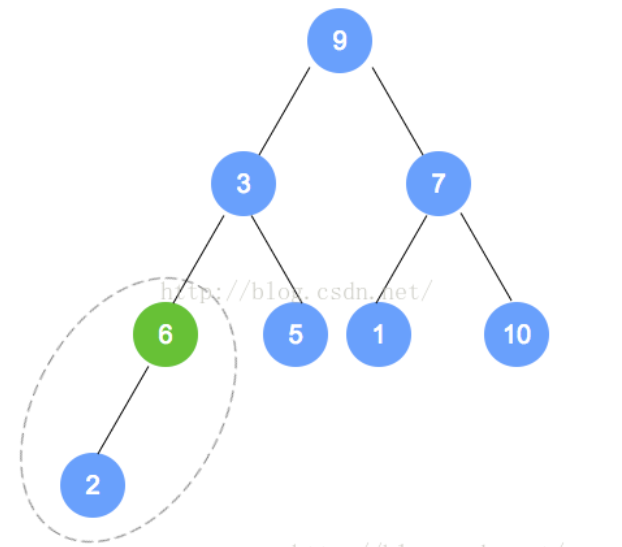
## 2.基本操作

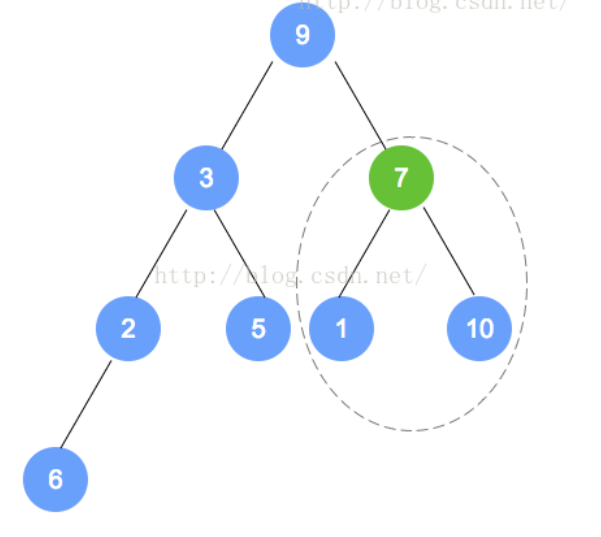
（1）构建

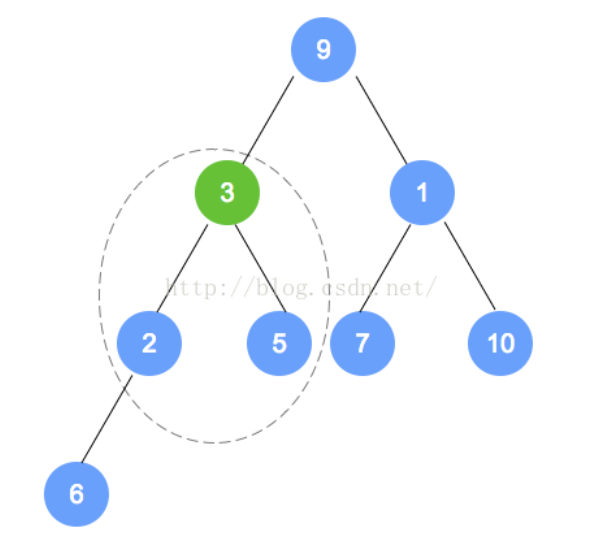
从无序序列建堆的过程就是一个反复调整的过程。若将此序列看成是一个完全二叉树，则最后一个非终端结点是第(n-2)/2个结点，由此调整过程只需从该结点开始，直到堆顶元素。

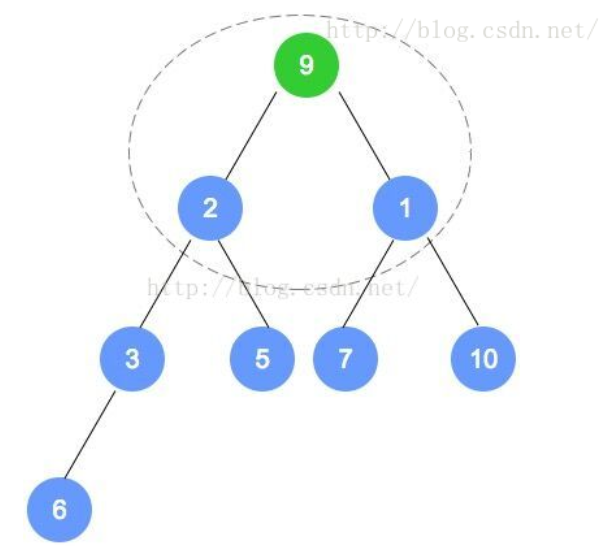
举个例子，初始数组为：9,3,7,6,5,1,10,2，构造小顶堆。按照完全二叉树，将数字依次填入。填入后，找到最后一个结点（本示例为数字2的节点），从它的父节点（本示例为数字6的节点）开始调整。根据性质，小的数字往上移动；至此，第1次调整完成。注意，被调整的节点，还有子节点的情况，需要递归进行调整。第二次调整，是数字6的节点数组下标小1的节点（比数字6的下标小1的节点是数字7的节点），用刚才的规则进行调整。以此类推，直到调整到根节点。

 以下是本示例的图解：

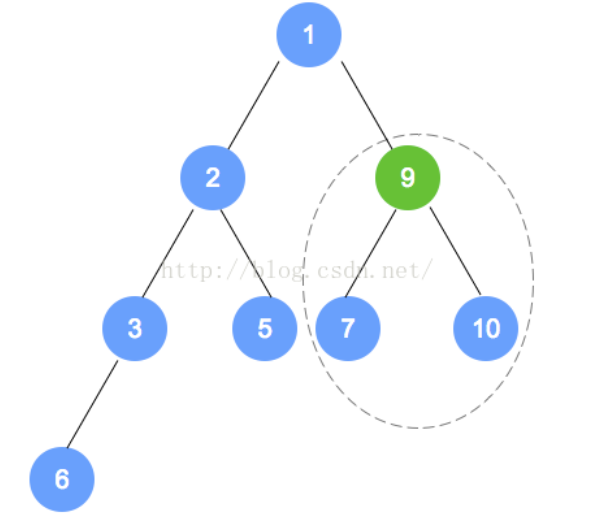


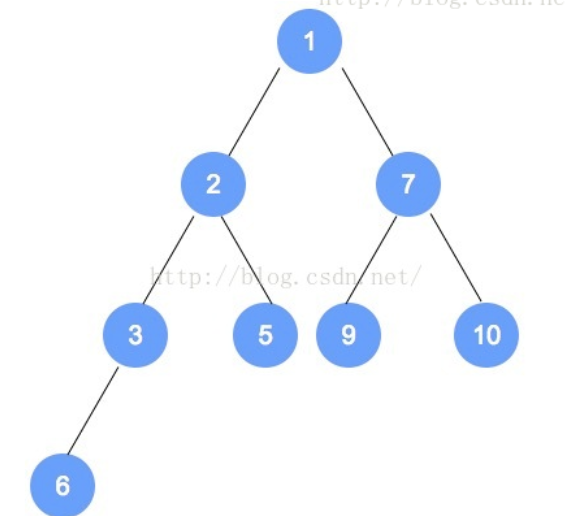






注意：数字9的节点 将和 数字1的节点 发生对调，对调后，需要递归进行调整，请一定注意。



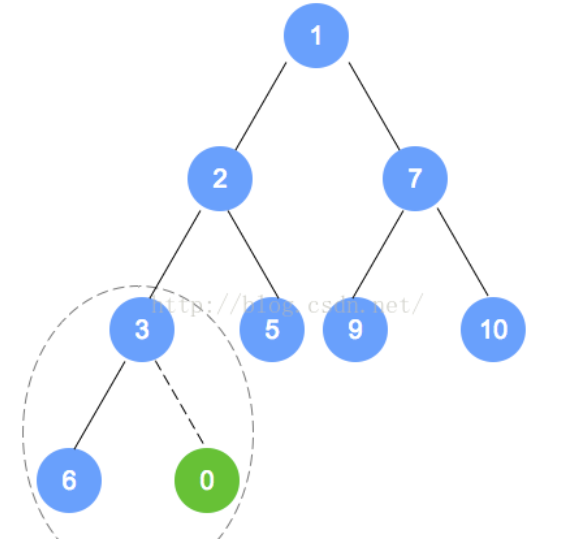


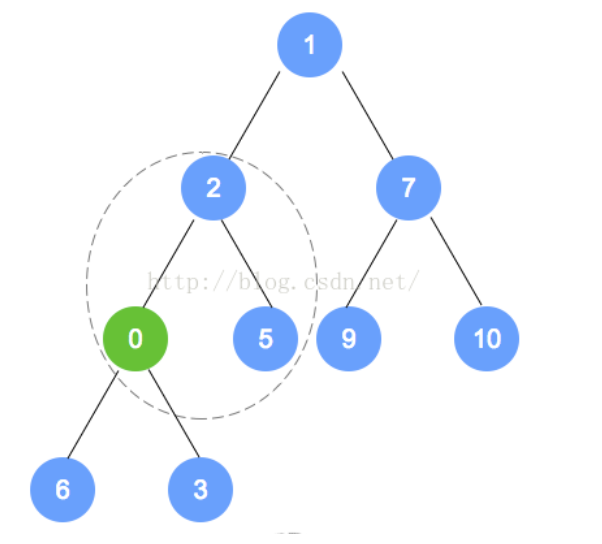
（2）插入

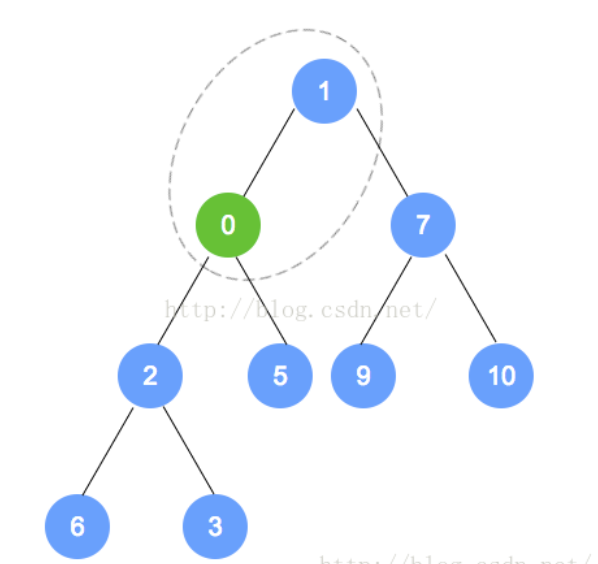
每次插入都是将先将新数据放在数组最后，由于从这个新数据的父结点到根结点必然为一个有序的序列，现在的任务是将这个新数据插入到这个有序序列中——这就类似于直接插入排序中将一个数据并入到有序区间中。

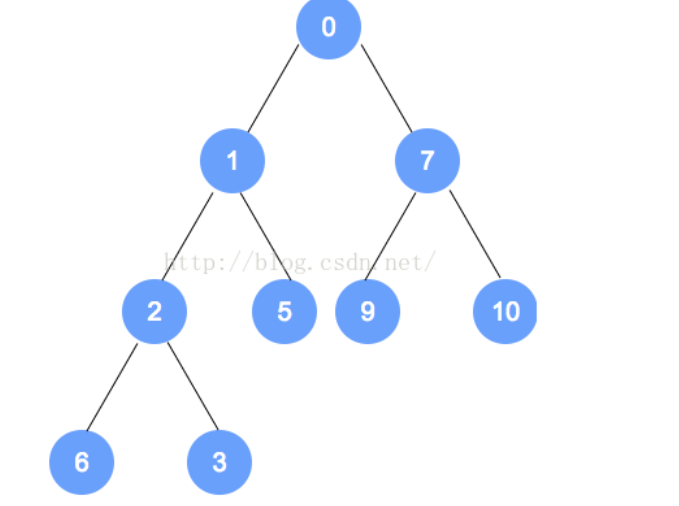
以上个最小堆为例，插入数字0。数字0的节点首先加入到该二叉树最后的一个节点，依据最小堆的定义，自底向上，递归调整。

  以下是插入操作的图解：





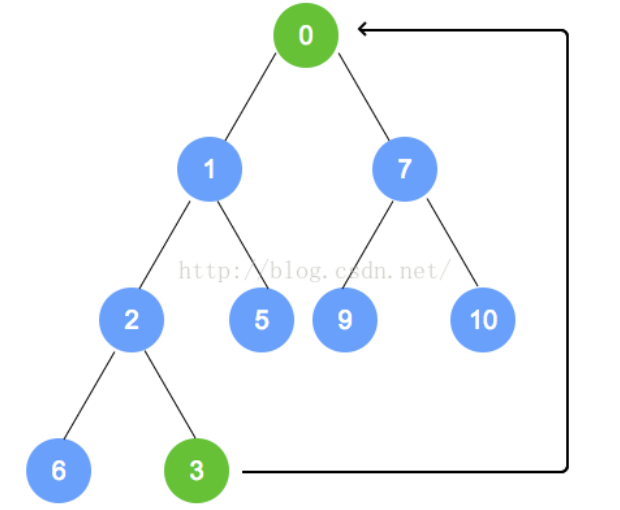


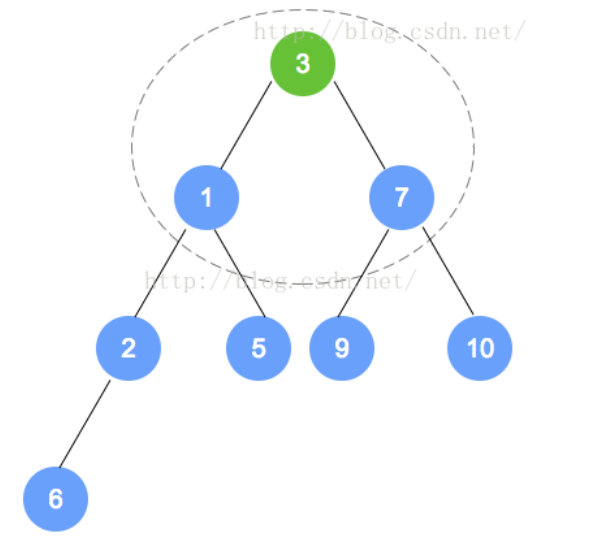


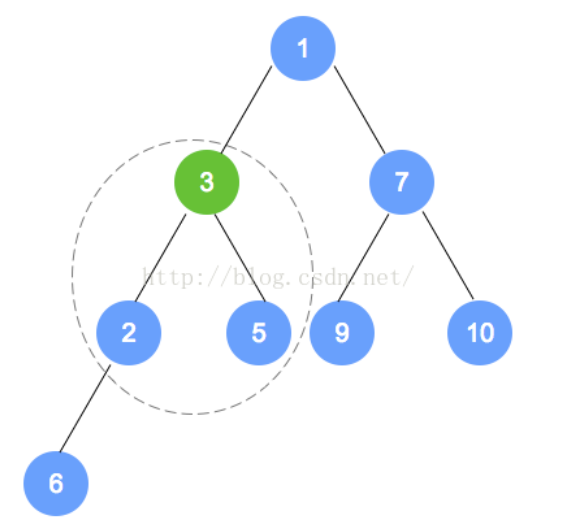
1. 删除

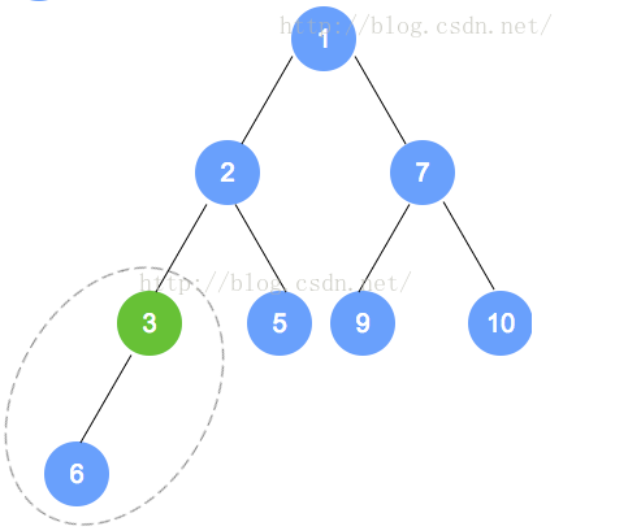
 对于最小堆和最大堆而言，删除是针对于根节点而言。对于删除操作，将二叉树的最后一个节点替换到根节点，然后自顶向下，递归调整。

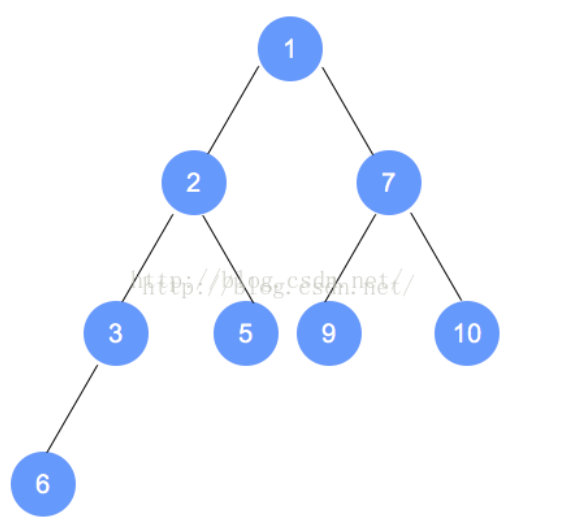
       以下是图解：











## 算法分析

堆排序过程的平均时间复杂度是O(nlgn)，最坏情况下的时间复杂度也是O（nlogn）。因为建堆的时间复杂度是O(n)（调用一次）；调整堆的时间复杂度是lgn，调用了n-1次，所以堆排序的时间复杂度是O(nlgn)。

由于建初始堆所需的比较次数较多，所以堆排序不适宜于记录数较少的文件。

堆排序是就地排序，辅助空间为O(1）.

它是不稳定的排序方法。（排序的稳定性是指如果在排序的序列中，存在前后相同的两个元素的话，排序前 和排序后他们的相对位置不发生变化）

# 五、B树

## 1.基本概念

B树是一种多路平衡查找树，一棵m阶B树(balanced tree of order m)是一棵平衡的m路搜索树。它或者是空树，或者是满足下列性质的树：

（1）根结点至少有两个子女；

（2）每个非根节点所包含的关键字个数j满足m/2-1<=j<=m-1;

（3）除根节点以外的所有节点（不包括叶子节点）的度数正好是关键字总数加1，故内部子树个数k满足：m/2<=k<=m;

（4）所有的叶子节点都位于同一层（叶子节点可以认为不带任何信息，因为倒数第二层是关键字保留了孩子的指针，但是没有具体的值）

## 2.基本操作

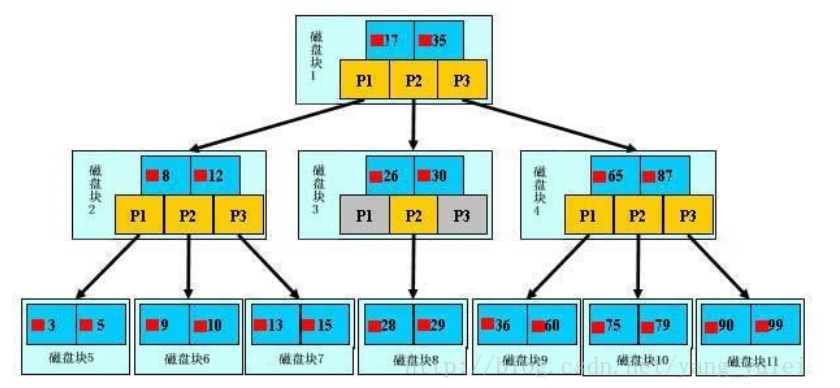
数据结构的基本操作主要有：创建、查找、插入、删除

1. 创建

B树的生成和二叉排序树一样，也是从空树开始，逐个插入关键字得到。由于B树节点中的关键字个数必须大于等于[m/2]-1，因此和平衡二叉树不同，每一次插入一个关键字并不是在树中加一个节点，而是首先在最底层的某个非终端节点中添加一个关键字，若该节点的关键字个数不超过m-1则插入完成，否则产生节点的分裂。

1. 查找

假如每个盘块可以正好存放一个B树的结点（正好存放2个文件名）。那么一个BTNODE结点就代表一个盘块，而子树指针就是存放另外一个盘块的地址。



下面，咱们来模拟下查找文件29的过程：

1.  根据根结点指针找到文件目录的根磁盘块1，将其中的信息导入内存。【磁盘IO操作1次】

2.  此时内存中有两个文件名17、35和三个存储其他磁盘页面地址的数据。根据算法我们发现：17<29<35，因此我们找到指针p2。

3.  根据p2指针，我们定位到磁盘块3，并将其中的信息导入内存。【磁盘IO操作 2次】

4.  此时内存中有两个文件名26，30和三个存储其他磁盘页面地址的数据。根据算法我们发现：26<29<30，因此我们找到指针p2。

5.  根据p2指针，我们定位到磁盘块8，并将其中的信息导入内存。【磁盘IO操作 3次】

6.  此时内存中有两个文件名28，29。根据算法我们查找到文件名29，并定位了该文件内存的磁盘地址。分析上面的过程，发现需要3 3次磁盘IO操作和次磁盘IO操作和3次内存查找 次内存查找操作。关于内存中的文件名查找，由于是一个有序表结构，可以利用折半查找提高效率。至于IO操作是影响整个B树查找效率的决定因素。

1. 插入

对高度为h的m阶B树，新结点一般是插在第h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：

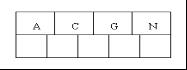
1）若该结点中关键码个数小于m-1，则直接插入即可。

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(h-1层)中

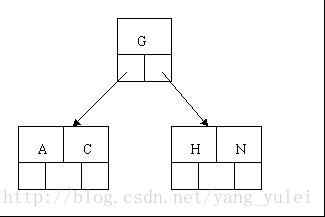
重复上述工作，最坏情况一直分裂到根结点，建立一个新的根结点，整个B树增加一层。

【例】

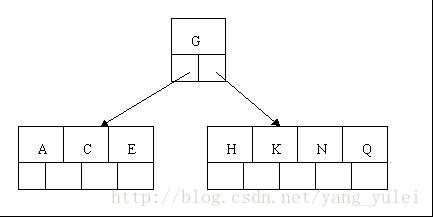
1、下面咱们通过一个实例来逐步讲解下。插入以下字符字母到一棵空的B 树中（非根结点关键字数小了（小于2个）就合并，大了（超过4个）就分裂）：C N G A H E K Q M F W L T Z D P R X Y S，首先，结点空间足够，4个字母插入相同的结点中，如下图：



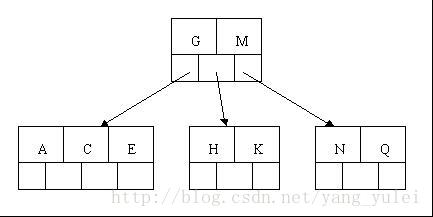
2、当咱们试着插入H时，结点发现空间不够，以致将其分裂成2个结点，移动中间元素G上移到新的根结点中，在实现过程中，咱们把A和C留在当前结点中，而H和N放置新的其右邻居结点中。如下图：



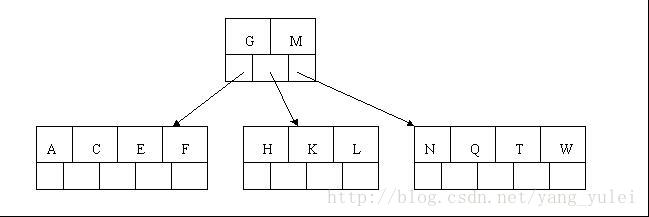
3、当咱们插入E,K,Q时，不需要任何分裂操作



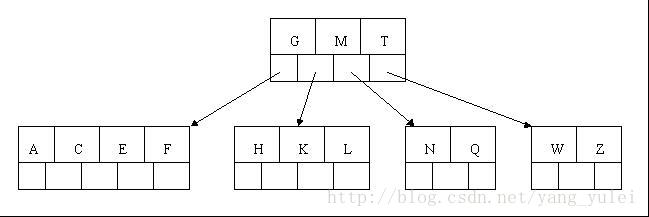
4、插入M需要一次分裂，注意M恰好是中间关键字元素，以致向上移到父节点中



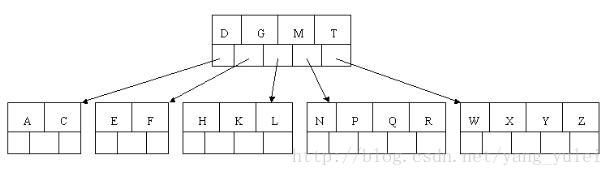
5、插入F,W,L,T不需要任何分裂操作



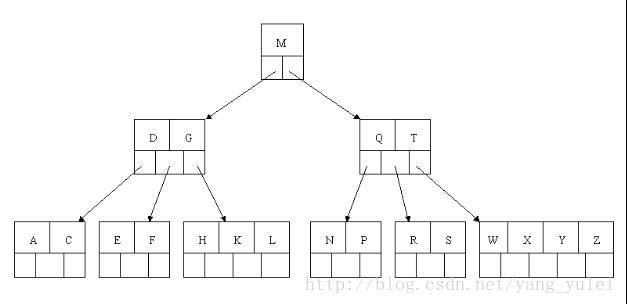
6、插入Z时，最右的叶子结点空间满了，需要进行分裂操作，中间元素T上移到父节点中，注意通过上移中间元素，树最终还是保持平衡，分裂结果的结点存在2个关键字元素。



7、插入D时，导致最左边的叶子结点被分裂，D恰好也是中间元素，上移到父节点中，然后字母P,R,X,Y陆续插入不需要任何分裂操作（别忘了，树中至多5个孩子）。



8、最后，当插入S时，含有N,P,Q,R的结点需要分裂，把中间元素Q上移到父节点中，但是情况来了，父节点中空间已经满了，所以也要进行分裂，将父节点中的中间元素M上移到新形成的根结点中，注意以前在父节点中的第三个指针在修改后包括D和G节点中。这样具体插入操作的完成，下面介绍删除操作，删除操作相对于插入操作要考虑的情况多点。



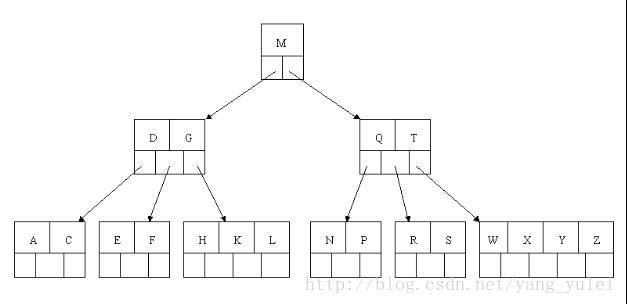
1. 删除

　首先查找B树中需删除的元素,如果该元素在B树中存在，则将该元素在其结点中进行删除，如果删除该元素后，首先判断该元素是否有左右孩子结点，如果有，则上移孩子结点中的某相近元素(“左孩子最右边的节点”或“右孩子最左边的节点”)到父节点中，然后是移动之后的情况；如果没有，直接删除后，移动之后的情况。

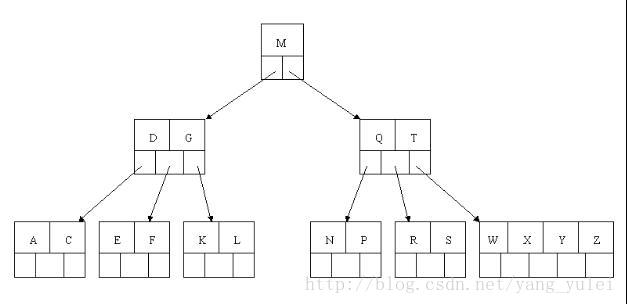
删除元素，移动相应元素之后，如果某结点中元素数目（即关键字数）小于ceil(m/2)-1，则需要看其某相邻兄弟结点是否丰满（结点中元素个数大于ceil(m/2)-1），除根结点之外的结点（包括叶子结点）的关键字的个数n必须满足： （ceil(m / 2)-1）<= n <=m-1。m表示最多含有m个孩子，n表示关键字数。在本小节中举的一颗B树的示例中，关键字数n满足：2<=n<=4），如果丰满，则向父节点借一个元素来满足条件；如果其相邻兄弟都刚脱贫，即借了之后其结点数目小于ceil(m/2)-1，则该结点与其相邻的某一兄弟结点进行“合并”成一个结点，以此来满足条件。那咱们通过下面实例来详细了解吧。

以上述插入操作构造的一棵5阶B树（树中最多含有m（m=5）个孩子，因此关键字数最小为ceil(m/ 2)-1=2。还是这句话，关键字数小了（小于2个）就合并，大了（超过4个）就分裂）为例，依次删除H,T,R,E。

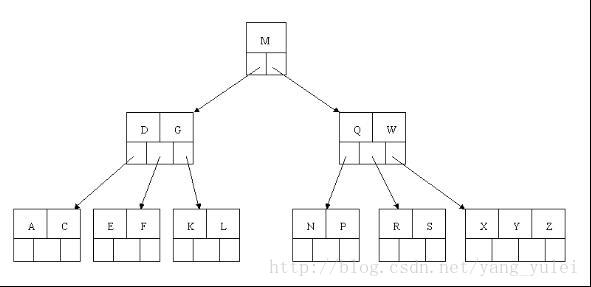
1、首先删除元素H，当然首先查找H，H在一个叶子结点中，且该叶子结点元素数目3大于最小元素数目ceil(m/2)-1=2，则操作很简单，咱们只需要移动K至原来H的位置，移动L至K的位置（也就是结点中删除元素后面的元素向前移动）



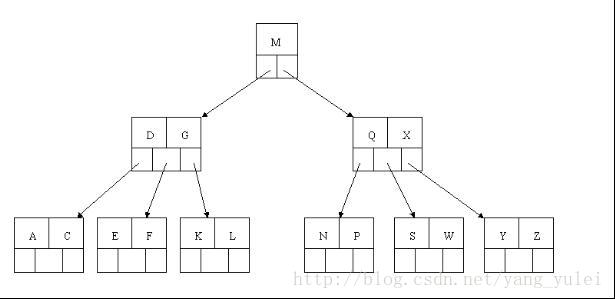
2、下一步，删除T,因为T没有在叶子结点中，而是在中间结点中找到，咱们发现他的继承者W(字母升序的下个元素)，将W上移到T的位置，然后将原包含W的孩子结点中的W进行删除，这里恰好删除W后，该孩子结点中元素个数大于2，无需进行合并操作。



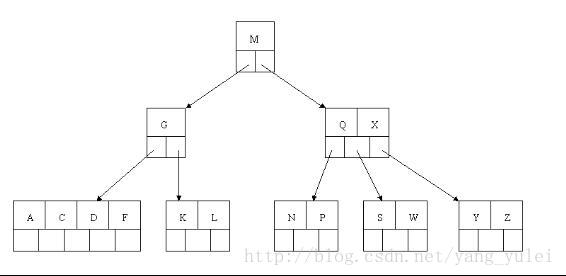
3、下一步删除R，R在叶子结点中,但是该结点中元素数目为2，删除导致只有1个元素，已经小于最小元素数目ceil(5/2)-1=2,而由前面我们已经知道：如果其某个相邻兄弟结点中比较丰满（元素个数大于ceil(5/2)-1=2），则可以向父结点借一个元素，然后将最丰满的相邻兄弟结点中上移最后或最前一个元素到父节点中（有没有看到红黑树中左旋操作的影子?），在这个实例中，右相邻兄弟结点中比较丰满（3个元素大于2），所以先向父节点借一个元素W下移到该叶子结点中，代替原来S的位置，S前移；然后X在相邻右兄弟结点中上移到父结点中，最后在相邻右兄弟结点中删除X，后面元素前移。



4、最后一步删除E， 删除后会导致很多问题，因为E所在的结点数目刚好达标，刚好满足最小元素个数（ceil(5/2)-1=2）,而相邻的兄弟结点也是同样的情况，删除一个元素都不能满足条件，所以需要该节点与某相邻兄弟结点进行合并操作；首先移动父结点中的元素（该元素在两个需要合并的两个结点元素之间）下移到其子结点中，然后将这两个结点进行合并成一个结点。所以在该实例中，咱们首先将父节点中的元素D下移到已经删除E而只有F的结点中，然后将含有D和F的结点和含有A,C的相邻兄弟结点进行合并成一个结点。



5、也许你认为这样删除操作已经结束了，其实不然，在看看上图，对于这种特殊情况，你立即会发现父节点只包含一个元素G，没达标（因为非根节点包括叶子结点的关键字数n必须满足于2=<n<=4，而此处的n=1），这是不能够接受的。如果这个问题结点的相邻兄弟比较丰满，则可以向父结点借一个元素。假设这时右兄弟结点（含有Q,X）有一个以上的元素（Q右边还有元素），然后咱们将M下移到元素很少的子结点中，将Q上移到M的位置，这时，Q的左子树将变成M的右子树，也就是含有N，P结点被依附在M的右指针上。所以在这个实例中，咱们没有办法去借一个元素，只能与兄弟结点进行合并成一个结点，而根结点中的唯一元素M下移到子结点，这样，树的高度减少一层。



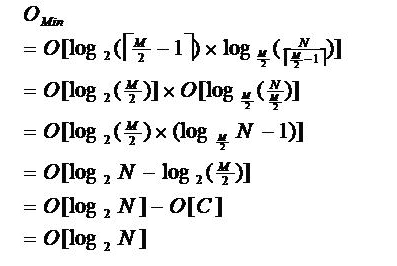
## 3.应用场景

用在磁盘文件组织 数据索引和数据库索引。

## 4.算法分析

B-tree高度与复杂度：

B树的高度是https://images2015.cnblogs.com/blog/524341/201604/524341-20160414100900863-348486693.gif，而不**是其它几种树的H=log2n，**其中T为度数（每个节点包含的元素个数），即所谓的阶数，n为总元素个数或总关键字数。  
  
B树查找的时间复杂度为O(Log2-N)，下面是参考推导过程：



# 六、B+树

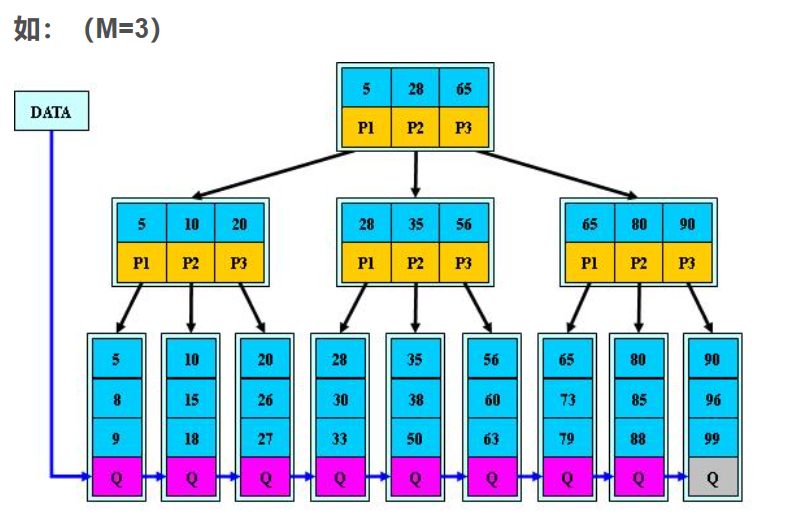
## 1.基本概念

B+树是应文件系统所需而出的一种B-树的变型树。一棵m阶的B+树和m阶的B-树的差异在于：

1.有n棵子树的结点中含有n个关键字，每个关键字不保存数据，只用来索引，所有数据都保存在叶子节点。

2.所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

3.所有的非终端结点可以看成是索引部分，结点中仅含其子树（根结点）中的最大（或最小）关键字。   
通常在B+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。



B+的特性：

1.所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好

是有序的；

2.不可能在非叶子结点命中；

3.非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储

（关键字）数据的数据层；

1. 更适合文件索引系统
2. B树与B+树的区别

以一个m阶树为例。

关键字的数量不同；B+树中分支结点有m个关键字，其叶子结点也有m个，其关键字只是起到了一个索引的作用，但是B树虽然也有m个子结点，但是其只拥有m-1个关键字。

存储的位置不同；B+树中的数据都存储在叶子结点上，也就是其所有叶子结点的数据组合起来就是完整的数据，但是B树的数据存储在每一个结点中，并不仅仅存储在叶子结点上。

分支结点的构造不同；B+树的分支结点仅仅存储着关键字信息和儿子的指针（这里的指针指的是磁盘块的偏移量），也就是说内部结点仅仅包含着索引信息。

查询不同；B树在找到具体的数值以后，则结束，而B+树则需要通过索引找到叶子结点中的数据才结束，也就是说B+树的搜索过程中走了一条从根结点到叶子结点的路径。

## 2.基本操作

B+的搜索与B-树也基本相同，区别是B+树只有达到叶子结点才命中（B-树可以在

非叶子结点命中），其性能也等价于在关键字全集做一次二分查找；

不过需要注意的是，在增加值的时候，如果存在满员的情况，将选择结点中的值作为新的索引，还有在删除值的时候，索引中的关键字并不会删除，也不会存在父亲结点的关键字下沉的情况，因为那只是索引。

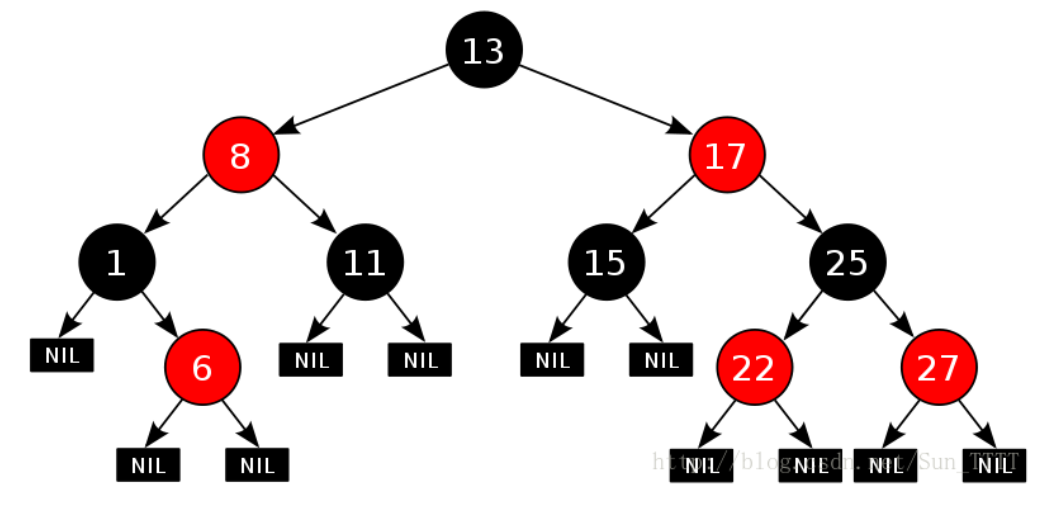
## 3.应用场景

由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引，而B树则常用于文件索引。

# 七、红黑树

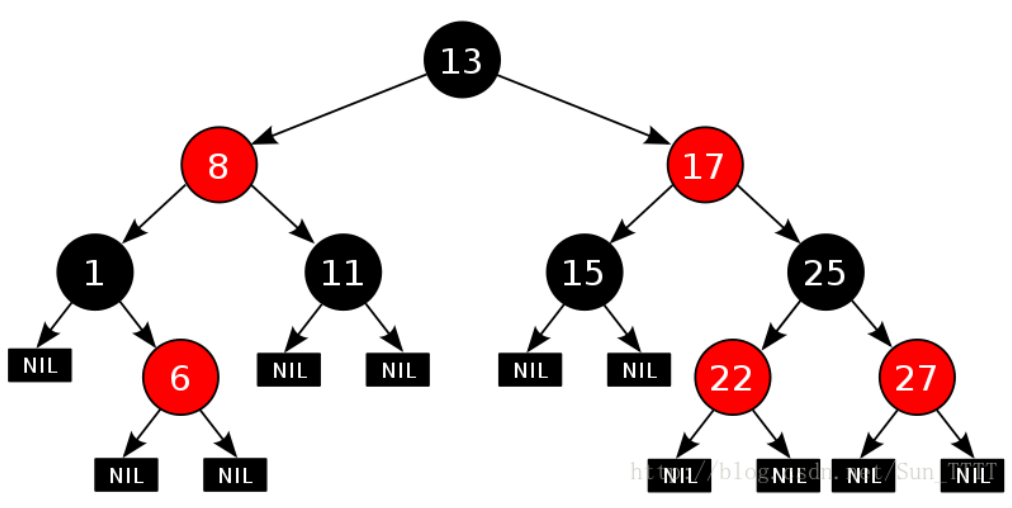
## 1.基本概念

红黑树是一个平衡的二叉树，但不是一个完美的平衡二叉树。虽然我们希望一个所有查找都能在logn次比较内结束，但是这样在动态插入中保持树的完美平衡代价太高，所以，我们稍微放松逛一下限制，希望找到一个能在对数时间内完成查找的数据结构。这个时候，红黑树站了出来。  
 红黑树是在普通二叉树上，对没个节点添加一个颜色属性形成的，同时整个红黑二叉树需要同时满足一下五条性质 ：   
性质一：节点是红色或者是黑色；   
在树里面的节点不是红色的就是黑色的，没有其他颜色。   
性质二：根节点是黑色；   
根节点总是黑色的。它不能为红。   
性质三：每个叶节点（NIL或空节点）是黑色；   
这个可能有点理解困难，可以看图：



这个图片就是一个红黑树，NIL节点是个空节点，并且是黑色的。   
性质四：每个红色节点的两个子节点都是黑色的（也就是说不存在两个连续的红色节点）；   
就是连续的两个节点不能是连续的红色，连续的两个节点的意思就是父节点与子节点不能是连续的红色。

性质五：从任一节点到其每个叶节点的所有路径都包含相同数目的黑色节点；



这五条性质约束了红黑树，可以通过数学证明来证明，满足这五条性质的二叉树可以将查找删除维持在对数时间内。 上述的性质约束了红黑树的关键：从根到叶子的最长可能路径不多于最短可能路径的两倍长。得到这个结论的理由是：

1.红黑树中最短的可能路径是全部为黑色节点的路径

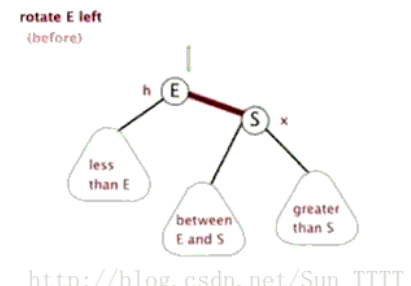
2.红黑树中最长的可能路径是红黑相间的路径

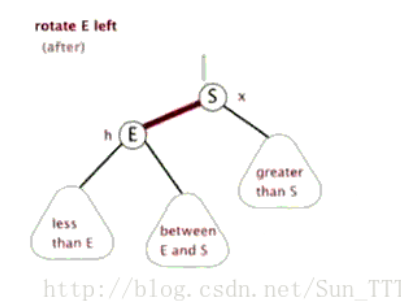
当我们进行插入或者删除操作时所作的一切操作都是为了调整树使之符合这五条性质。

## 2.基本操作

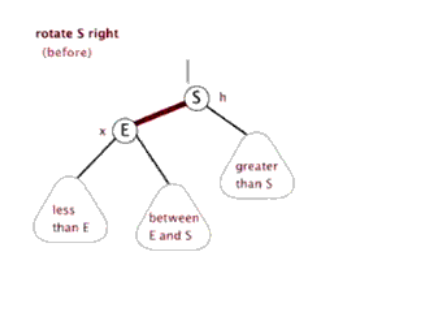
旋转的目的是将节点多的一支出让节点给另一个节点少的一支，旋转操作在插入和删除操作中经常会用到，所以要熟记。

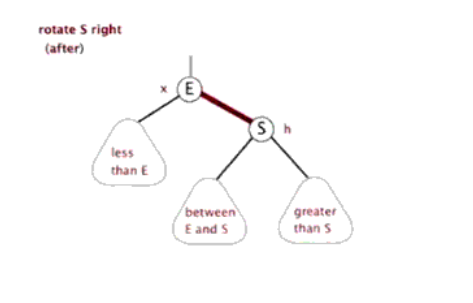
左旋：





右旋：

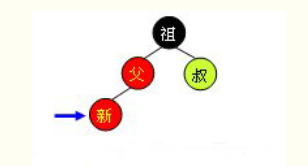


  
在讨论红黑树的插入操作之前必须要明白，任何一个即将插入的新结点的初始颜色都为红色。这一点很容易理解，因为插入黑点会增加某条路径上黑结点的数目，从而导致整棵树黑高度的不平衡。但如果新结点的父结点为红色时（如下图所示），将会违反红黑树的性质：一条路径上不能出现相邻的两个红色结点。这时就需要通过一系列操作来使红黑树保持平衡。

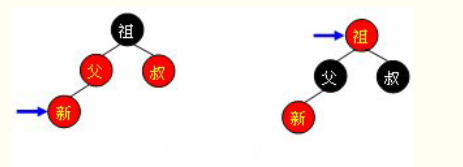
      为了清楚地表示插入操作以下在结点中使用“新”字表示一个新插入的结点；使用“父”字表示新插入点的父结点；使用“叔”字表示“父”结点的兄弟结点；使用“祖”字表示“父”结点的父结点。插入操作分为以下几种情况：  
1、黑父  
     如下图所示，如果新节点的父结点为黑色结点，那么插入一个红点将不会影响红黑树的平衡，此时插入操作完成。红黑树比AVL树优秀的地方之一在于黑父的情况比较常见，从而使红黑树需要旋转的几率相对AVL树来说会少一些。



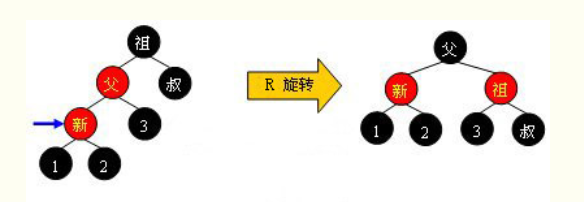
2、红父  
     如果新节点的父结点为红色，这时就需要进行一系列操作以保证整棵树红黑性质。如下图所示，由于父结点为红色，此时可以判定，祖父结点必定为黑色。这时需要根据叔父结点的颜色来决定做什么样的操作。青色结点表示颜色未知。由于有可能需要根结点到新点的路径上进行多次旋转操作，而每次进行不平衡判断的起始点（我们可将其视为新点）都不一样。所以我们在此使用一个蓝色箭头指向这个起始点，并称之为判定点。



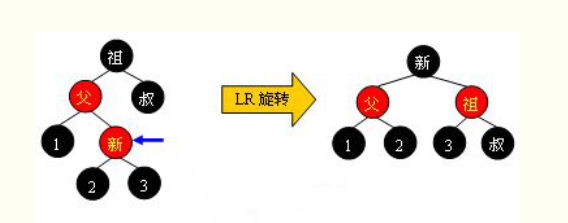
2.1 红叔  
当叔父结点为红色时，如下图所示，无需进行旋转操作，只要将父和叔结点变为黑色，将祖父结点变为红色即可。但由于祖父结点的父结点有可能为红色，从而违反红黑树性质。此时必须将祖父结点作为新的判定点继续向上（迭代）进行平衡操作。



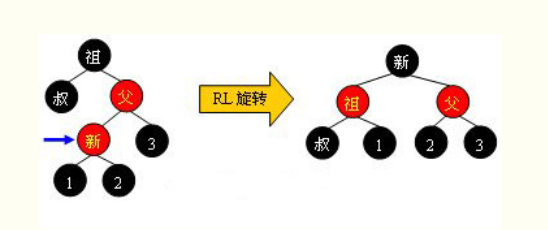
需要注意的是，无论“父节点”在“叔节点”的左边还是右边，无论“新节点”是“父节点”的左孩子还是右孩子，它们的操作都是完全一样的（其实这种情况包括4种，只需调整颜色，不需要旋转树形）。  
2.2 黑叔  
当叔父结点为黑色时，需要进行旋转，以下图示了所有的旋转可能：  
Case 1:



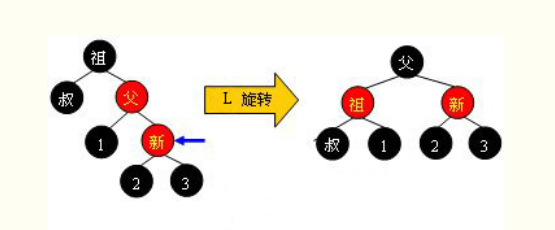
Case 2:



Case 3:



Case 4:



      可以观察到，当旋转完成后，新的旋转根全部为黑色，此时不需要再向上回溯进行平衡操作，插入操作完成。需要注意，上面四张图的“叔”、“1”、“2”、“3”结点有可能为黑哨兵结点。

## 3.应用场景

红黑树引入了“颜色”的概念。引入“颜色”的目的在于使得红黑树的平衡条件得以简化。正如著名的密码学专家Bruce Schneier所说的那样，“Being Partly balanced can be good enough”，红黑树并不追求“完全平衡”——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(log n)的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。当然，还有一些更好的，但实现起来更复杂的数据结构能够做到一步旋转之内达到平衡，但红黑树能够给我们一个比较“便宜”的解决方案。红黑树的算法时间复杂度和AVL相同，但统计性能比AVL树更高。

当然，红黑树并不适应所有应用树的领域。如果数据基本上是静态的，那么让他们待在他们能够插入，并且不影响平衡的地方会具有更好的性能。如果数据完全是静态的，例如，做一个哈希表，性能可能会更好一些。

在实际的系统中

* 著名的linux进程调度[Completely Fair Scheduler](https://link.zhihu.com/?target=https://en.wikipedia.org/wiki/Completely_Fair_Scheduler" \t "https://www.zhihu.com/question/_blank),用红黑树管理进程控制块
* epoll在内核中的实现，用红黑树管理事件块
* nginx中，用红黑树管理timer等
* Java的TreeMap实现

## 4.算法分析

插入和删除操作改变树的平衡性的概率要远远小于AVL（RBT不是高度平衡的）。因此需要的旋转操作的可能性要小，而且一旦需要旋转，插入一个结点最多只需要旋转2次，删除最多只需要旋转3次(小于AVL的删除操作所需要的旋转次数)。虽然变色操作的时间复杂度在O(logN)，但是实际上，这种操作由于简单所需要的代价很小。