**BST与AVL、Huffman树知识点**

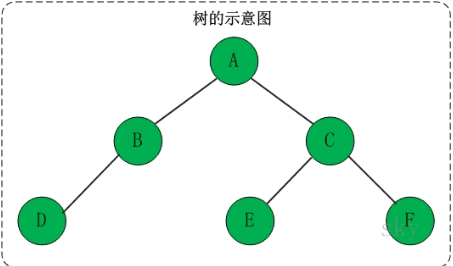
BST：二叉查找树 AVL：平衡二叉查找树

**一、BST**

<http://www.cnblogs.com/skywang12345/p/3576452.html>

**1、定义**

树是一种数据结构，它是由n（n>=1）个有限节点组成一个具有层次关系的集合。



把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

(01) 每个节点有零个或多个子节点；

(02) 没有父节点的节点称为**根节点**；

(03) 每一个非根节点有且只有一个父节点；

(04) 除了根节点外，每个子节点可以分为多个不相交的子树。

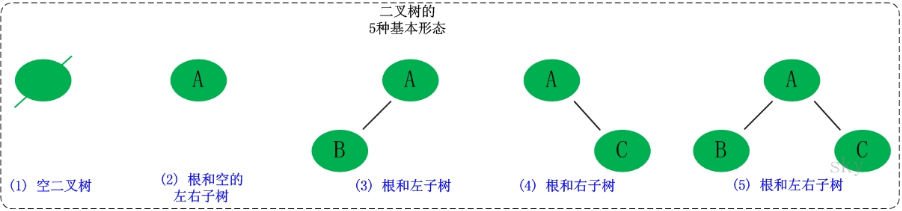
**树的基本术语：**

若一个结点有子树，那么该结点称为子树根的"双亲"，子树的根是该结点的"孩子"。有相同双亲的结点互为"兄弟"。一个结点的所有子树上的任何结点都是该结点的后裔。从根结点到某个结点的路径上的所有结点都是该结点的祖先。

* **结点的度：**结点拥有的子树的数目。
* **叶子：**度为零的结点。
* **分支结点：**度不为零的结点。
* **树的度：**树中结点的最大的度。
* **层次：***根结点的层次为1*，其余结点的层次等于该结点的双亲结点的层次加1。
* 树的高度：树中结点的最大层次。
* **无序树：**如果树中结点的各子树之间的次序是不重要的，可以交换位置。
* **有序树：**如果树中结点的各子树之间的次序是重要的, 不可以交换位置。
* **森林：**0个或多个不相交的树组成。对森林加上一个根，森林即成为树；删去根，树即成为森林。

**2、二叉树介绍**

二叉树是每个节点最多有两个子树的树结构。它有五种基本形态：二叉树可以是空集；根可以有空的左子树或右子树；或者左、右子树皆为空。



二叉树的性质：TODO(上标和下标)

* **性质1：**二叉树第i层上的结点数目最多为 2{i-1} (i≥1)。
* **性质2：**深度为k的二叉树至多有2{k}-1个结点(k≥1)。
* **性质3：**包含n个结点的二叉树的高度至少为**log2 (n+1)**。
* **性质4：**在任意一棵二叉树中，若***终端结点的个数为n0***，度为2的结点数为n2，则**n0=n2+1**。

**性质1：**二叉树第i层上的结点数目最多为 2{i-1} (i≥1)

**证明：**下面用"数学归纳法"进行证明。

(01) 当i=1时，第i层的节点数目为2{i-1}=2{0}=1。因为第1层上只有一个根结点，所以命题成立。

(02) 假设当i>1，第i层的节点数目为2{i-1}。这个是根据(01)推断出来的！下面根据这个假设，推断出"第(i+1)层的节点数目为2{i}"即可。由于二叉树的每个结点至多有两个孩子，故"第(i+1)层上的结点数目" 最多是 "第i层的结点数目的2倍"。即，第(i+1)层上的结点数目最大值=2×2{i-1}=2{i}。

故假设成立，原命题得证！

**性质2：**深度为k的二叉树至多有2{k}-1个结点(k≥1)

**证明：**在具有相同深度的二叉树中，当每一层都含有最大结点数时，其树中结点数最多。利用"性质1"可知，深度为k的二叉树的结点数至多为：20+21+…+2k-1=2k-1

故原命题得证！

**性质3：**包含n个结点的二叉树的高度至少为log2 (n+1)

**证明：**根据"性质2"可知，高度为h的二叉树最多有2{h}–1个结点。反之，对于包含n个节点的二叉树的高度至少为log2(n+1)。

**性质4：**在任意一棵二叉树中，若终端结点的个数为n0，度为2的结点数为n2，则n0=n2+1

**证明：**因为二叉树中所有结点的度数均不大于2，所以**结点总数(记为n)="0度结点数(n0)" + "1度结点数(n1)" + "2度结点数(n2)"**。由此，得到等式一。

**(等式一) n=n0+n1+n2**

另一方面，0度结点没有孩子，1度结点有一个孩子，2度结点有两个孩子，故二叉树中孩子结点总数是：n1+2n2。此外，只有根不是任何结点的孩子。故二叉树中的结点总数又可表示为等式二。

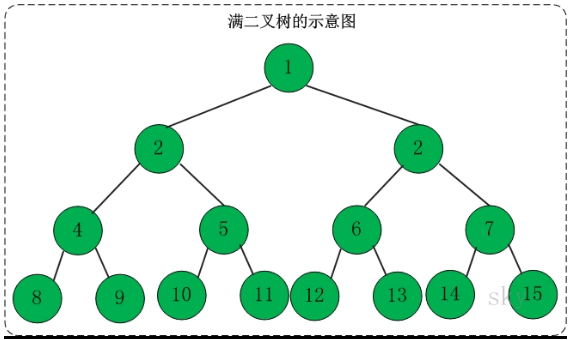
**(等式二) n=n1+2n2+1**

由(等式一)和(等式二)计算得到：**n0=n2+1**。原命题得证！

**满二叉树，完全二叉树和二叉查找树**

**满二叉树：**

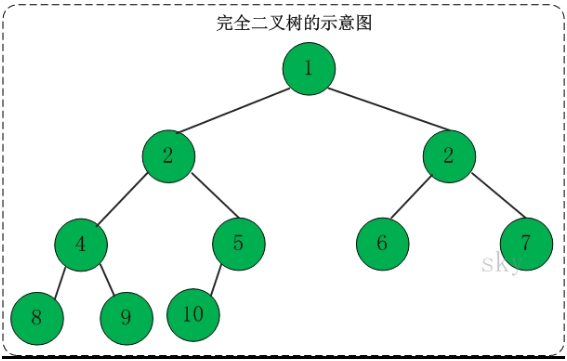
定义：高度为h，并且由2{h} –1个结点的二叉树，被称为满二叉树。



**完全二叉树：**

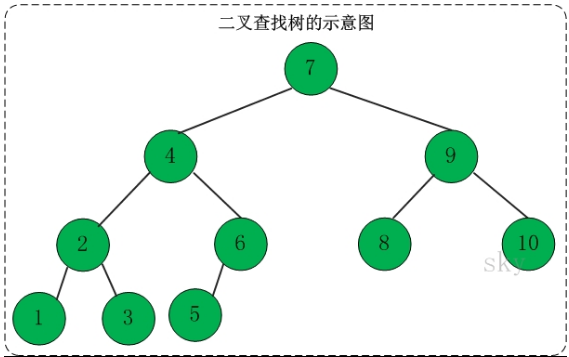
**定义：**一棵二叉树中，只有最下面两层结点的度可以小于2，并且最下一层的叶结点集中在靠左的若干位置上。这样的二叉树称为完全二叉树。

**特点：**叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。显然，一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。



**二叉查找树：**

**定义：**二叉查找树(Binary Search Tree)，又被称为二叉搜索树。设x为二叉查找树中的一个结点，x节点包含关键字key，节点x的key值记为key[x]。如果y是x的左子树中的一个结点，则key[y] <= key[x]；如果y是x的右子树的一个结点，则key[y] >= key[x]。



**在二叉查找树中：**

(01) 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

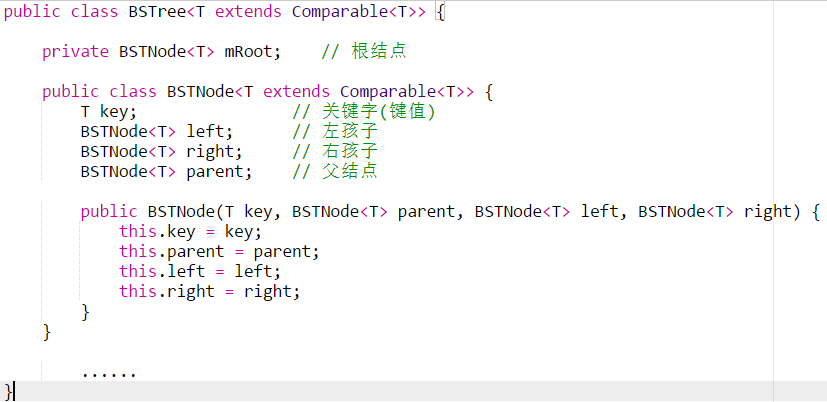
(02) 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

(03) 任意节点的左、右子树也分别为二叉查找树。

(04) 没有键值相等的节点（no duplicate nodes）。

**3、具体操作**

**1）节点定义**



BSTree是二叉树，它保护了二叉树的根节点mRoot；mRoot是BSTNode类型，而BSTNode是二叉查找树的节点，它是BSTree的内部类。BSTNode包含二叉查找树的几个基本信息：

(01) key -- 它是关键字，是用来对二叉查找树的节点进行排序的。

(02) left -- 它指向当前节点的左孩子。

(03) right -- 它指向当前节点的右孩子。

(04) parent -- 它指向当前节点的父结点。

**2）遍历**

前序遍历、中序遍历、后序遍历3种方式

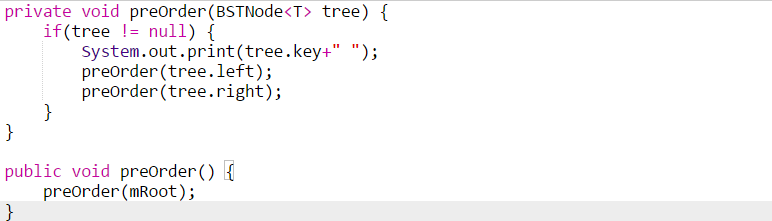
**A. 前序遍历**

若二叉树非空，则执行以下操作：

(01) 访问根结点；

(02) 先序遍历左子树；

(03) 先序遍历右子树。



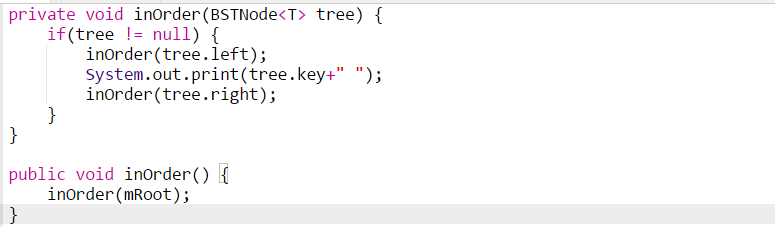
**B. 中序遍历**

若二叉树非空，则执行以下操作：

(01) 中序遍历左子树；

(02) 访问根结点；

(03) 中序遍历右子树。



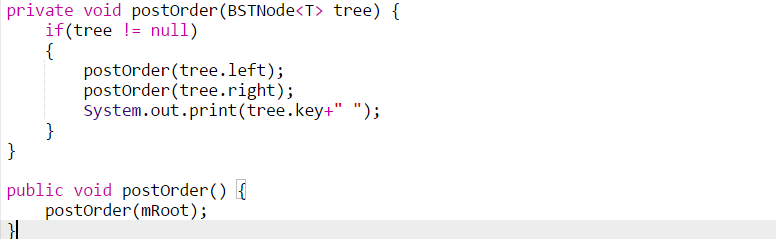
**C. 后序遍历**

若二叉树非空，则执行以下操作：

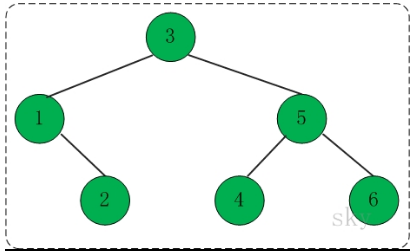
(01) 后序遍历左子树；

(02) 后序遍历右子树；

(03) 访问根结点。



看看下面这颗树的各种遍历方式：



对于上面的二叉树而言，

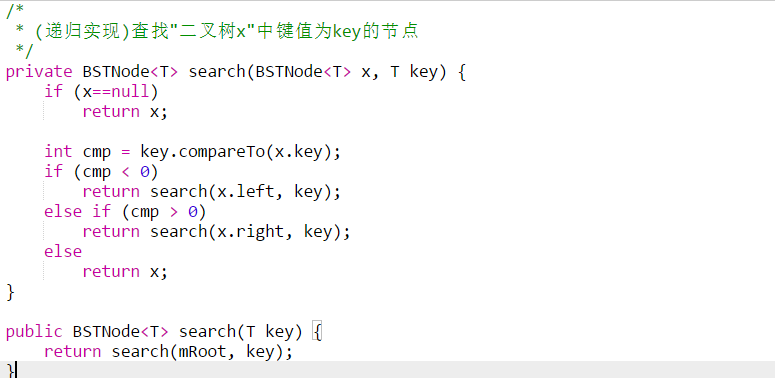
(01) 前序遍历结果： 3 1 2 5 4 6

(02) 中序遍历结果： 1 2 3 4 5 6

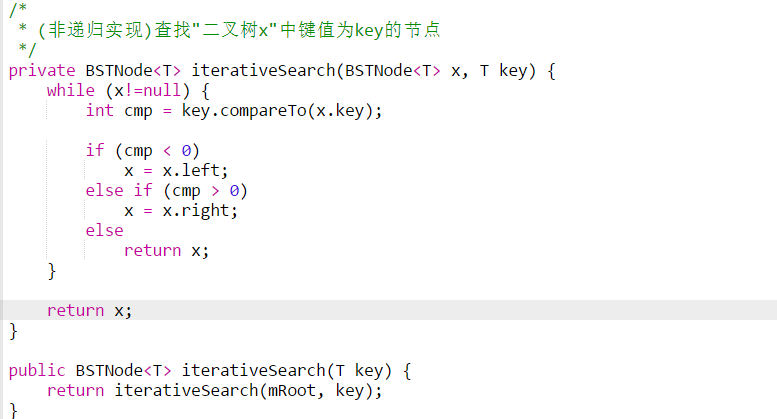
(03) 后序遍历结果： 2 1 4 6 5 3

**3）查找**

递归版本的代码：



非递归版本的代码：

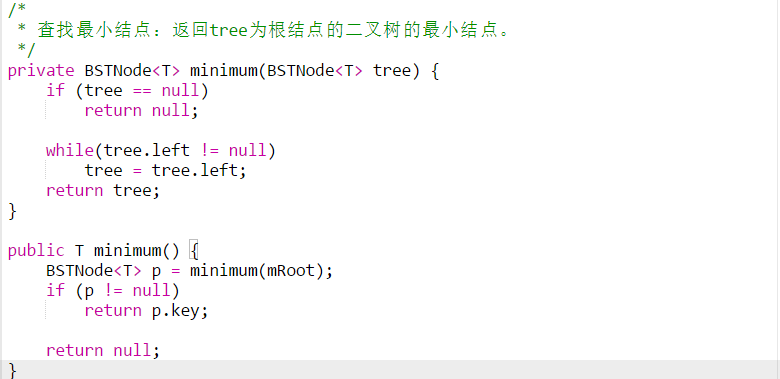


**4）最大值和最小值**

查找最大值的代码：



查找最小值的代码：

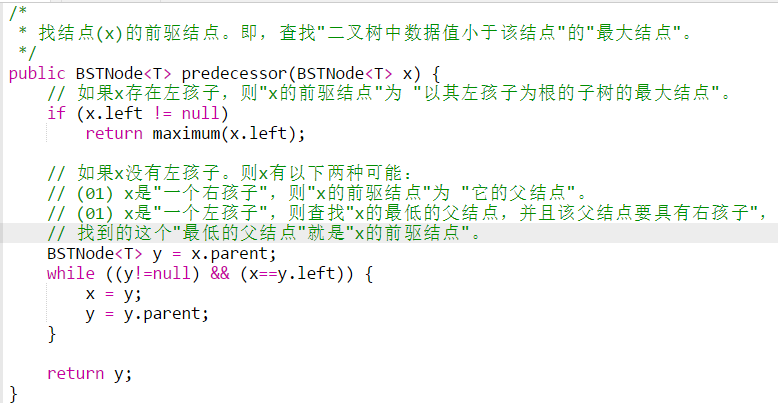


**5）前驱和后继**

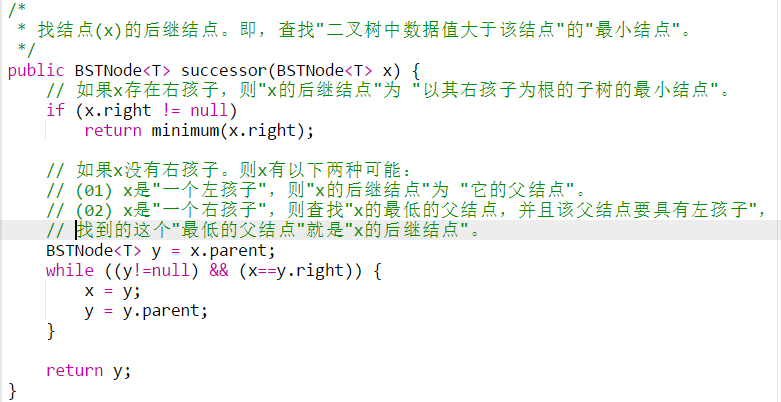
**节点的前驱：**是该节点的左子树中的最大节点。

**节点的后继：**是该节点的右子树中的最小节点。

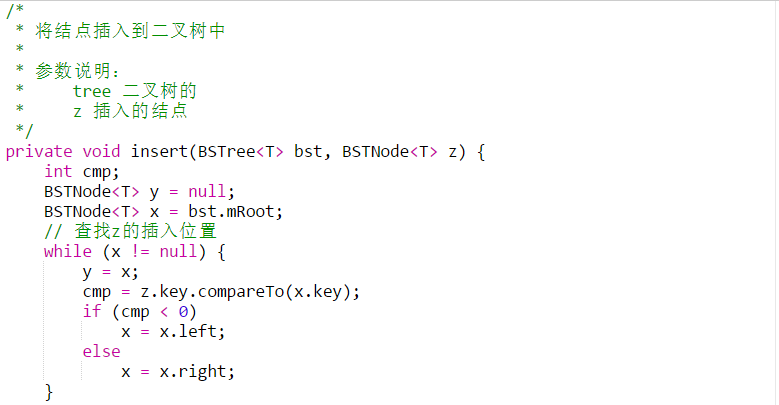
查找前驱节点的代码：



查找后继节点的代码



**6）插入**



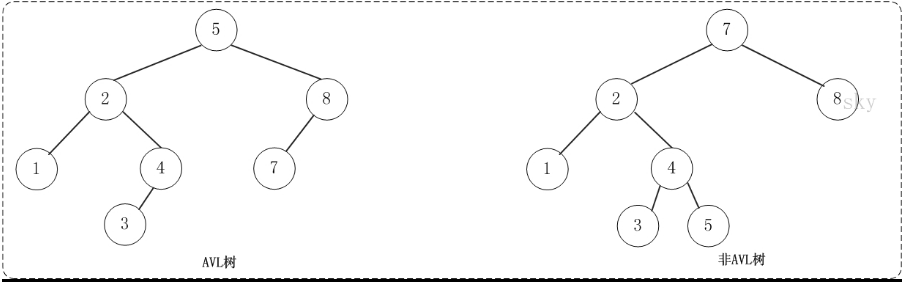


**二、AVL**

<http://www.cnblogs.com/skywang12345/p/3577479.html>

**1、定义**

AVL树是*高度平衡的而二叉树*。它的特点是：AVL树中任何节点的两个子树的高度最大差别为1。

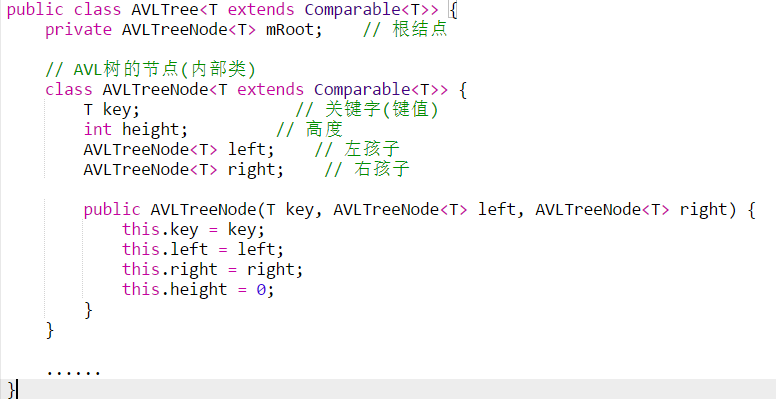


上面的两张图片，左边的是AVL树，它的任何节点的两个子树的高度差别都<=1；而右边的不是AVL树，因为7的两颗子树的高度相差为2(以2为根节点的树的高度是3，而以8为根节点的树的高度是1)。

**2、实现**

**1）节点**

**A. 节点的定义**



AVLTree是AVL树对应的类，而AVLTreeNode是AVL树节点，它是AVLTree的内部类。AVLTree包含了AVL树的根节点，AVL树的基本操作也定义在AVL树中。AVLTreeNode包括的几个组成对象：

(01) key -- 是关键字，是用来对AVL树的节点进行排序的。

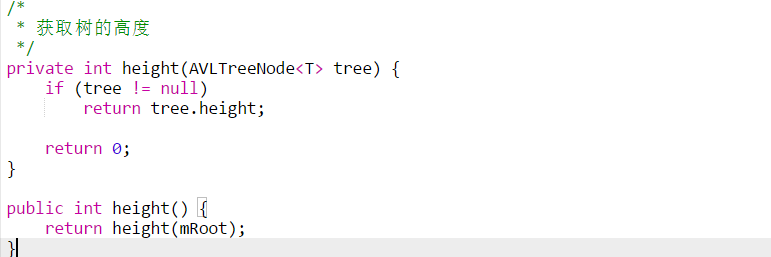
(02) left -- 是左孩子。

(03) right -- 是右孩子。

(04) height -- 是高度。

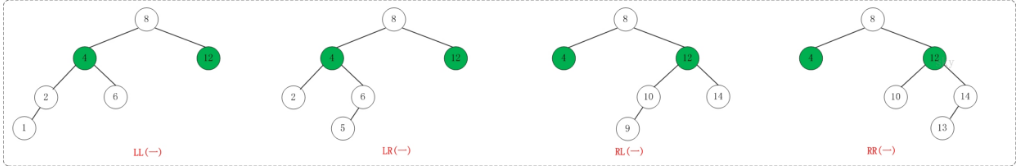
**B. 树的高度**

关于高度，有的地方将"空二叉树的高度是-1"，而本文采用维基百科上的定义：树的高度为最大层次。即空的二叉树的高度是0，非空树的高度等于它的最大层次(根的层次为1，根的子节点为第2层，依次类推)。

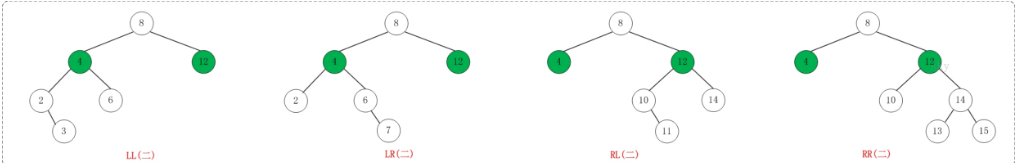


**2）旋转**

如果在AVL树中进行**插入或删除节点**后，可能*导致AVL树失去平衡*。这种失去平衡的可以概括为4种姿态：**LL(左左)，LR(左右)，RR(右右)和RL(右左)**。下面给出它们的示意图：



上图中的4棵树都是"失去平衡的AVL树"，从左往右的情况依次是：LL、LR、RL、RR。除了上面的情况之外，还有其它的失去平衡的AVL树，如下图：



上面的两张图都是为了便于理解，而列举的关于"失去平衡的AVL树"的例子。总的来说，AVL树失去平衡时的情况一定是LL、LR、RL、RR这4种之一，它们都由各自的定义：

* **LL：**LeftLeft，也称为"左左"。插入或删除一个节点后，*根节点的****左子树的左子树还有非空子节点****，导致"根的左子树的高度"比"根的右子树的高度"大2*，导致AVL树失去了平衡。

例如，在上面LL情况中，由于"根节点(8)的左子树(4)的左子树(2)还有非空子节点"，而"根节点(8)的右子树(12)没有子节点"；导致"根节点(8)的左子树(4)高度"比"根节点(8)的右子树(12)"高2。

* **LR：**LeftRight，也称为"左右"。插入或删除一个节点后，*根节点的****左子树的右子树还有非空子节点****，导致"根的左子树的高度"比"根的右子树的高度"大2*，导致AVL树失去了平衡。

例如，在上面LR情况中，由于"根节点(8)的左子树(4)的右子树(6)还有非空子节点"，而"根节点(8)的右子树(12)没有子节点"；导致"根节点(8)的左子树(4)高度"比"根节点(8)的右子树(12)"高2。

* **RL：**RightLeft，称为"右左"。插入或删除一个节点后，*根节点的右子树的左子树还有非空子节点，导致"根的右子树的高度"比"根的左子树的高度"大2*，导致AVL树失去了平衡。

例如，在上面RL情况中，由于"根节点(8)的右子树(12)的左子树(10)还有非空子节点"，而"根节点(8)的左子树(4)没有子节点"；导致"根节点(8)的右子树(12)高度"比"根节点(8)的左子树(4)"高2。

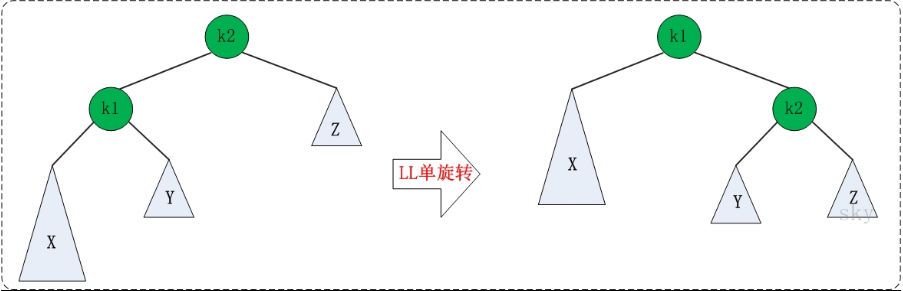
* **RR：**RightRight，称为"右右"。插入或删除一个节点后，*根节点的右子树的右子树还有非空子节点，导致"根的右子树的高度"比"根的左子树的高度"大2*，导致AVL树失去了平衡。

例如，在上面RR情况中，由于"根节点(8)的右子树(12)的右子树(14)还有非空子节点"，而"根节点(8)的左子树(4)没有子节点"；导致"根节点(8)的右子树(12)高度"比"根节点(8)的左子树(4)"高2。

如果在AVL树中进行插入或删除节点后，可能导致AVL树失去平衡。AVL失去平衡之后，可以通过**旋转**使其恢复平衡，下面分别介绍"LL(左左)，LR(左右)，RR(右右)和RL(右左)"这4种情况对应的旋转方法。

**A. LL的旋转**

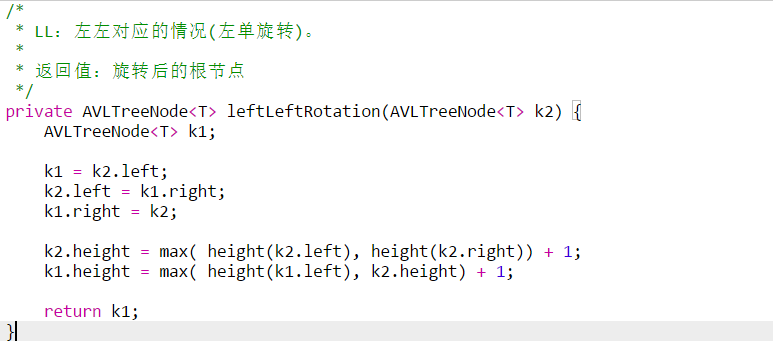
LL失去平衡的情况，可以通过一次旋转让AVL树恢复平衡。如下图：**（根节点k2的右旋）**



图中左边是旋转之前的树，右边是旋转之后的树。从中可以发现，旋转之后的树又变成了AVL树，而且该旋转只需要一次即可完成。

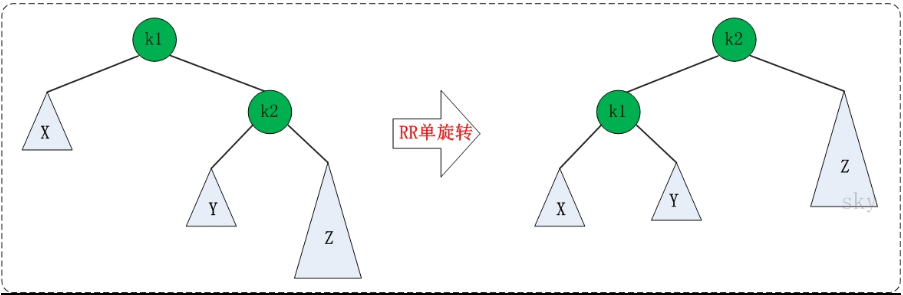
对于LL旋转，你可以这样理解为：**LL旋转是围绕"失去平衡的AVL根节点"进行的**，也就是节点k2；而且由于是LL情况，即左左情况，就用手抓着"左孩子，即k1"使劲摇。将k1变成根节点，k2变成k1的右子树，"k1的右子树"变成"k2的左子树"。

代码：



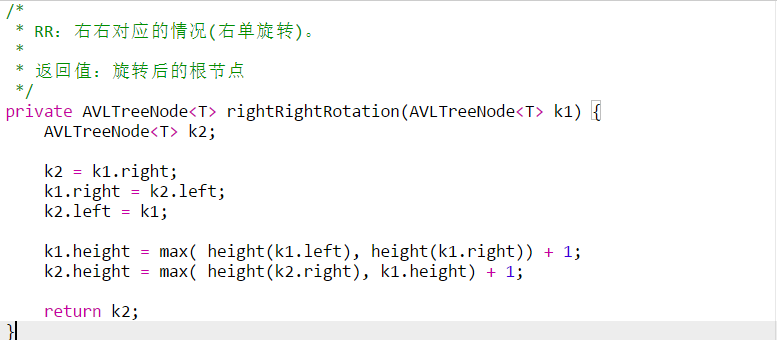
**B. RR的旋转**

理解了LL之后，RR就相当容易理解了。RR是与LL对称的情况！RR恢复平衡的旋转方法如下：**（根节点k1的左旋）**



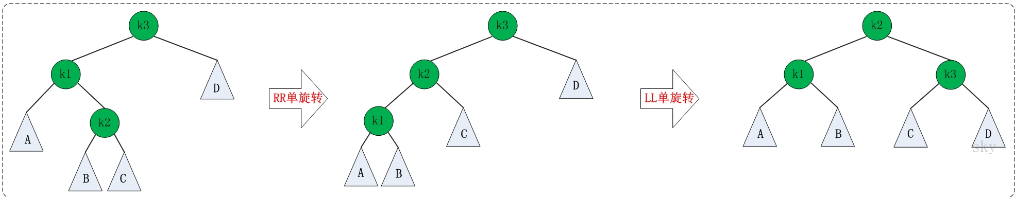
图中左边是旋转之前的树，右边是旋转之后的树。RR旋转也只需要一次即可完成。

**代码：**

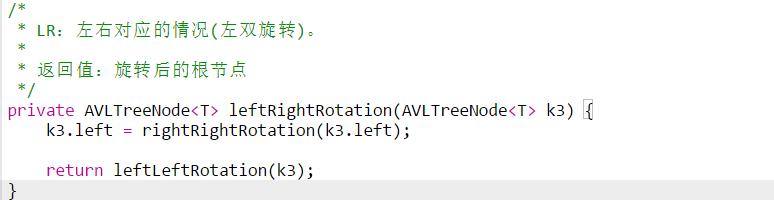


**C. LR的旋转**

LR失去平衡的情况，需要经过两次旋转才能让AVL树恢复平衡。如下图：**先左旋后右旋**

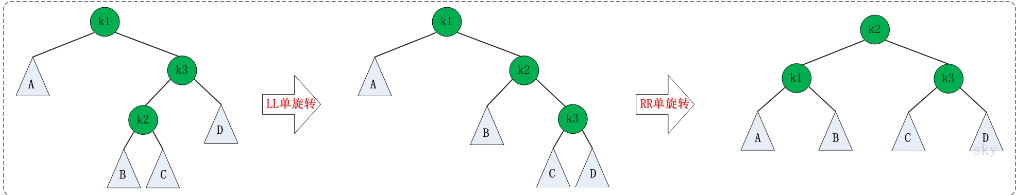


第一次旋转是**围绕"k1"进行的"RR旋转"**，第二次是**围绕"k3"进行的"LL旋转"**。

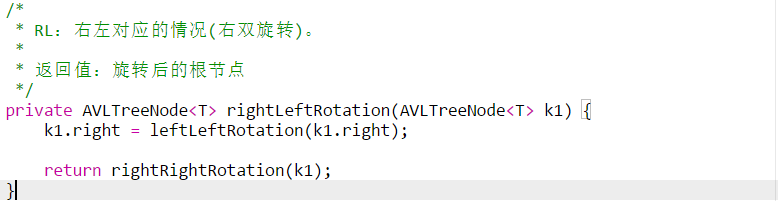


**D. RL的旋转**

RL是与LR的对称情况！RL恢复平衡的旋转方法如下：

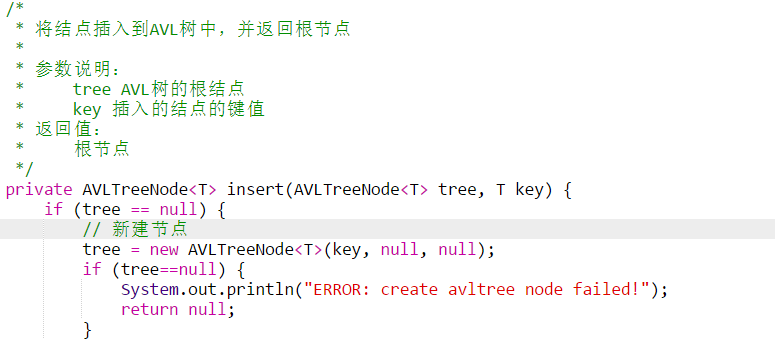


第一次旋转是围绕"k3"进行的"LL旋转"，第二次是围绕"k1"进行的"RR旋转"。

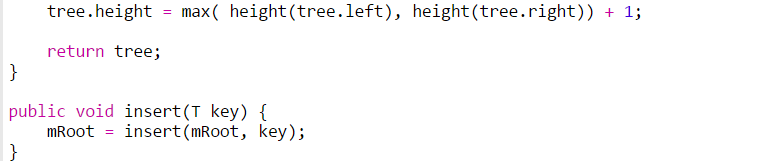


**2）插入**

插入节点的代码：



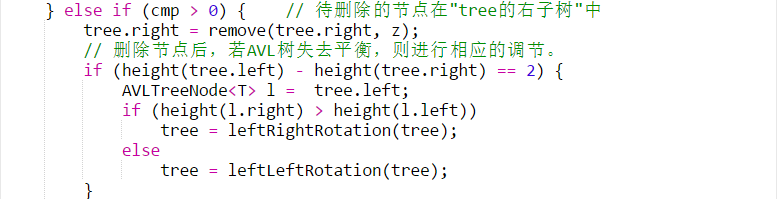


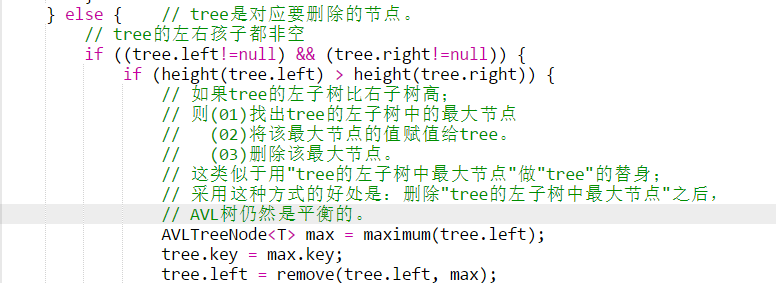


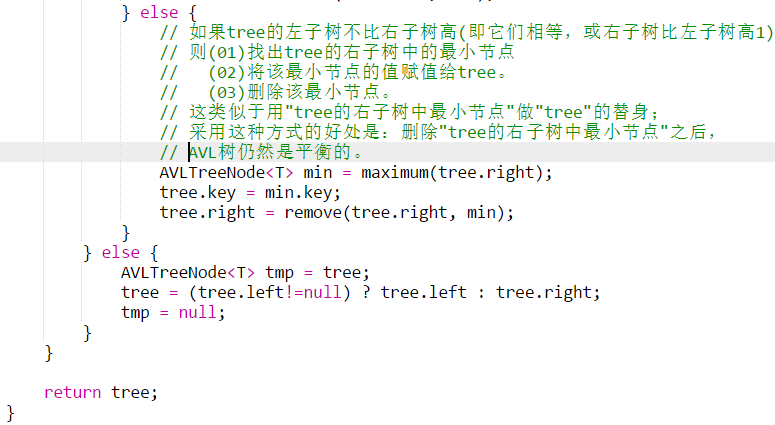
**3）删除**

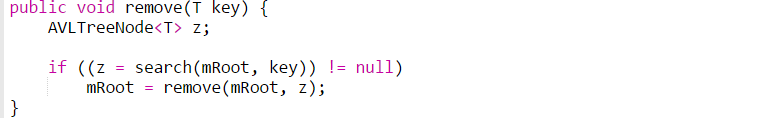
删除节点的代码分析











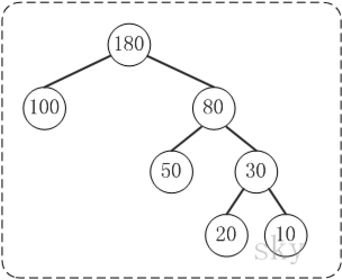
**三、Huffman**

<http://www.cnblogs.com/skywang12345/p/3706833.html>

**1、介绍**

Huffman Tree，中文名是哈夫曼树或霍夫曼树，它是最优二叉树。

**定义：**给定*n个权值作为n个叶子结点，构造一棵二叉树，若树的带权路径长度达到最小，则这棵树被称为哈夫曼树*。 这个定义里面涉及到了几个陌生的概念，下面就是一颗哈夫曼树，我们来看图解答。



* **路径和路径长度**

**定义：**在一棵树中，从一个结点往下可以达到的孩子或孙子结点之间的通路，称为路径。通路中分支的数目称为路径长度。若规定根结点的层数为1，则从根结点到第L层结点的路径长度为L-1。

**例子：**100和80的路径长度是1，50和30的路径长度是2，20和10的路径长度是3。

* **结点的权及带权路径长度**

**定义：**若将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。结点的带权路径长度为：*从根结点到该结点之间的路径长度与该结点的权的乘积*。

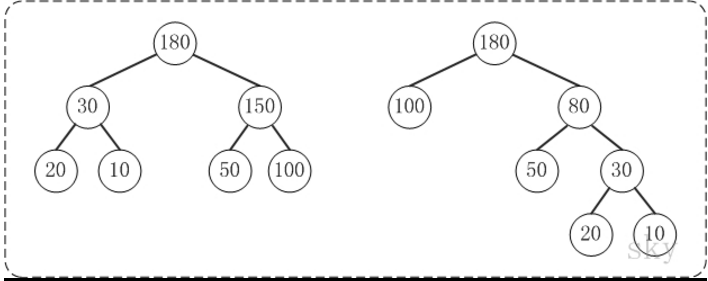
**例子：**节点20的路径长度是3，它的带权路径长度= 路径长度 \* 权 = 3 \* 20 = 60。

* **树的带权路径长度**

**定义：**树的带权路径长度规定为*所有叶子结点的带权路径长度之和*，记为WPL。

**例子：**示例中，树的WPL= 1\*100 + 2\*50 + 3\*20 + 3\*10 = 100 + 100 + 60 + 30 = 290。

比较下面两棵树



上面的两棵树都是以{10, 20, 50, 100}为叶子节点的树。

左边的树WPL=2\*10 + 2\*20 + 2\*50 + 2\*100 = 360

右边的树WPL=290

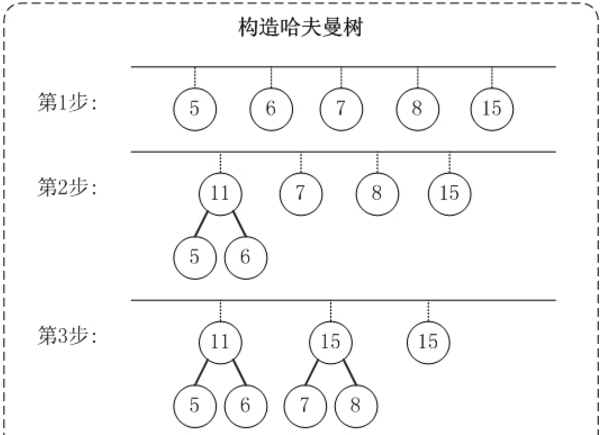
左边的树WPL > 右边的树的WPL。你也可以计算除上面两种示例之外的情况，但实际上右边的树就是{10,20,50,100}对应的哈夫曼树。至此，应该堆哈夫曼树的概念有了一定的了解了，下面看看如何去构造一棵哈夫曼树。

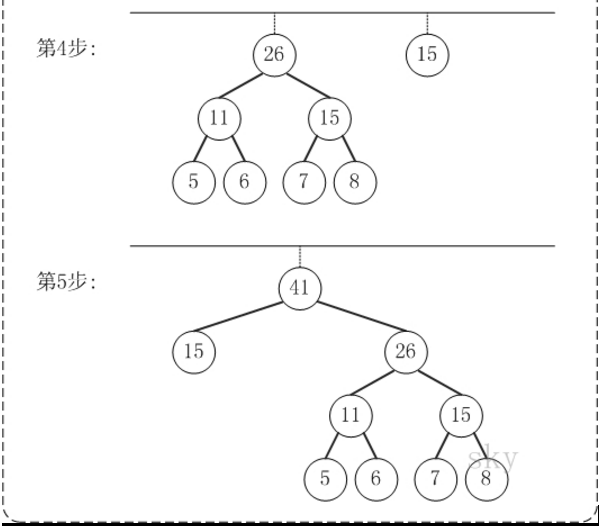
**2、定义**

假设有n个权值，则构造出的哈夫曼树有n个叶子结点。 n个权值分别设为 w1、w2、…、wn，哈夫曼树的构造规则为：

* 将w1、w2、…，wn看成是有n 棵树的森林(每棵树仅有一个结点)；
* 在森林中选出根结点的权值最小的两棵树进行合并，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；
* 从森林中删除选取的两棵树，并将新树加入森林；
* 重复(02)、(03)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。

以{5,6,7,8,15}为例，来构造一棵哈夫曼树。





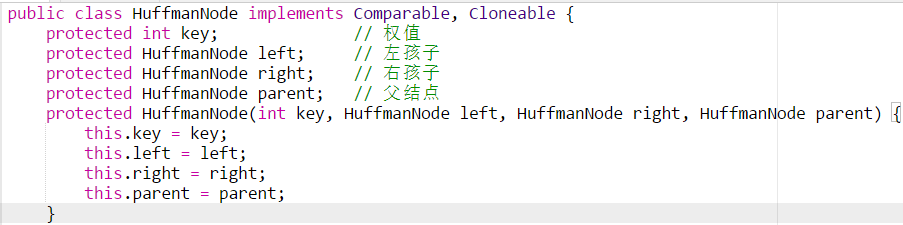
* **第1步：**创建森林，森林包括5棵树，这5棵树的权值分别是5,6,7,8,15。
* **第2步：**在森林中，选择根节点权值最小的两棵树(5和6)来进行合并，将它们作为一颗新树的左右孩子(谁左谁右无关紧要，这里，我们选择较小的作为左孩子)，并且新树的权值是左右孩子的权值之和。即，新树的权值是11。 然后，将"树5"和"树6"从森林中删除，并将新的树(树11)添加到森林中。
* **第3步：**在森林中，选择根节点权值最小的两棵树(7和8)来进行合并。得到的新树的权值是15。 然后，将"树7"和"树8"从森林中删除，并将新的树(树15)添加到森林中。
* **第4步：**在森林中，选择根节点权值最小的两棵树(11和15)来进行合并。得到的新树的权值是26。 然后，将"树11"和"树15"从森林中删除，并将新的树(树26)添加到森林中。
* **第5步：**在森林中，选择根节点权值最小的两棵树(15和26)来进行合并。得到的新树的权值是41。 然后，将"树15"和"树26"从森林中删除，并将新的树(树41)添加到森林中。

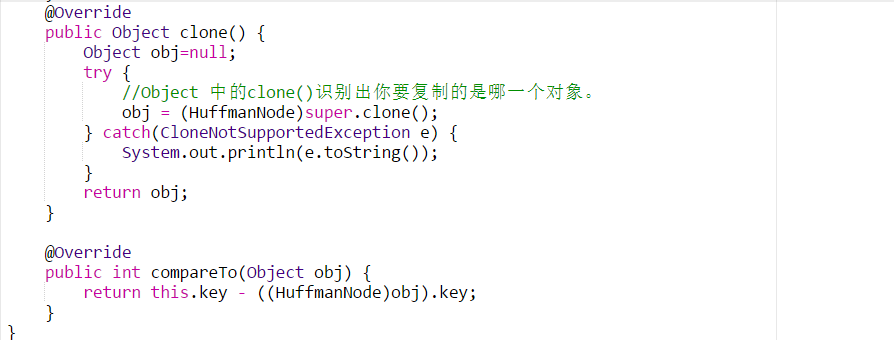
此时，森林中只有一棵树(树41)。这棵树就是我们需要的哈夫曼树！

**3、基本操作**

哈夫曼树的重点是**如何构造哈夫曼树**。本文构造哈夫曼时，用到了以前介绍过的**"(二叉堆)最小堆"**。下面对哈夫曼树进行讲解。

**1）定义**





HuffmanNode是哈夫曼树的节点类。



Huffman是哈夫曼树对应的类，它包含了哈夫曼树的根节点和哈夫曼树的相关操作。

**2）构造哈夫曼树**



首先创建最小堆，然后进入for循环。

每次循环时：

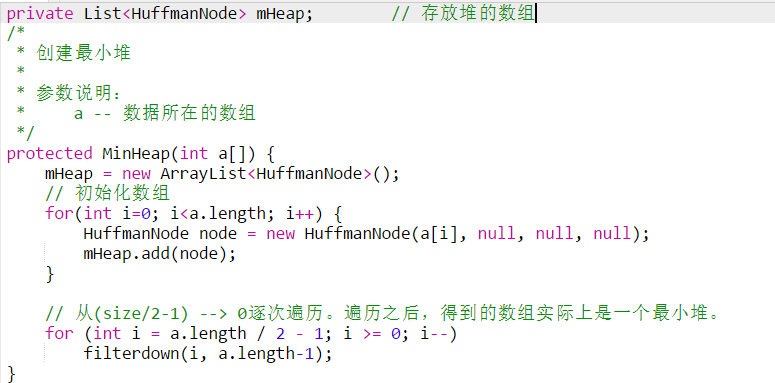
**(01)** 首先，将最小堆中的最小节点拷贝一份并赋值给left，然后重塑最小堆(将最小节点和后面的节点交换位置，接着将"交换位置后的最小节点"之前的全部元素重新构造成最小堆)；

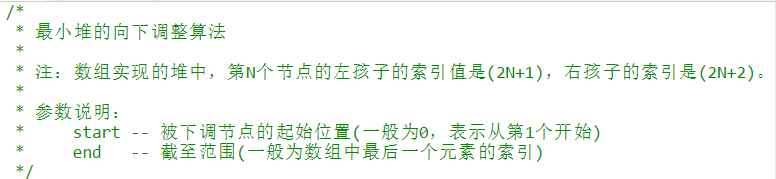
**(02)** 接着，再将最小堆中的最小节点拷贝一份并将其赋值right，然后再次重塑最小堆；

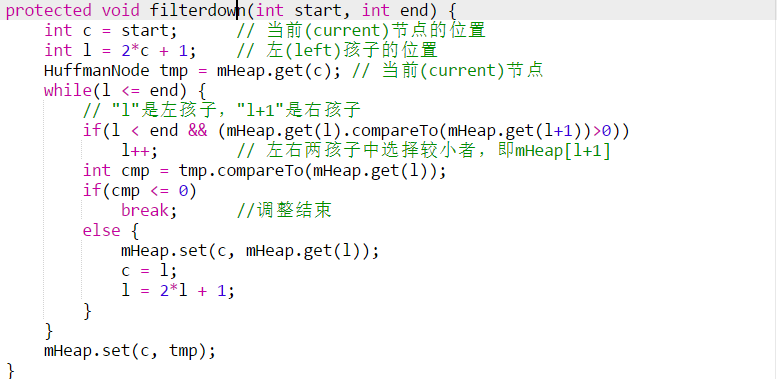
**(03)** 然后，新建节点parent，并将它作为left和right的父节点；

**(04)** 接着，将parent的数据复制给最小堆中的指定节点。

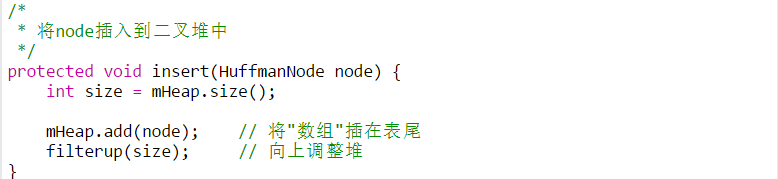
**3）构造最小堆minHeap**

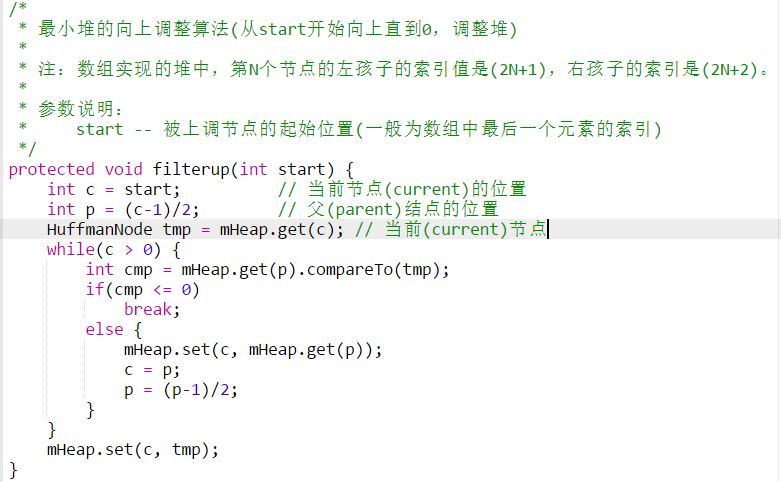






**4）插入insert**





**5）dumpFromMinimum**

