**实现优先队列堆结构知识点**

实现优先队列（PriorityQueue）结构主要是通过**堆完成**，主要有：二叉堆、d堆、左式堆、斜堆、二项堆、斐波那契堆、pairing 堆等。

三大优先队列：二项堆、斐波那契堆、Pairing 堆

<http://dsqiu.iteye.com/blog/1714961>

<http://www.cnblogs.com/wuchanming/p/3809496.html>

**一、二项堆**

<http://www.cnblogs.com/skywang12345/p/3656098.html> （很详细）

**1、定义**

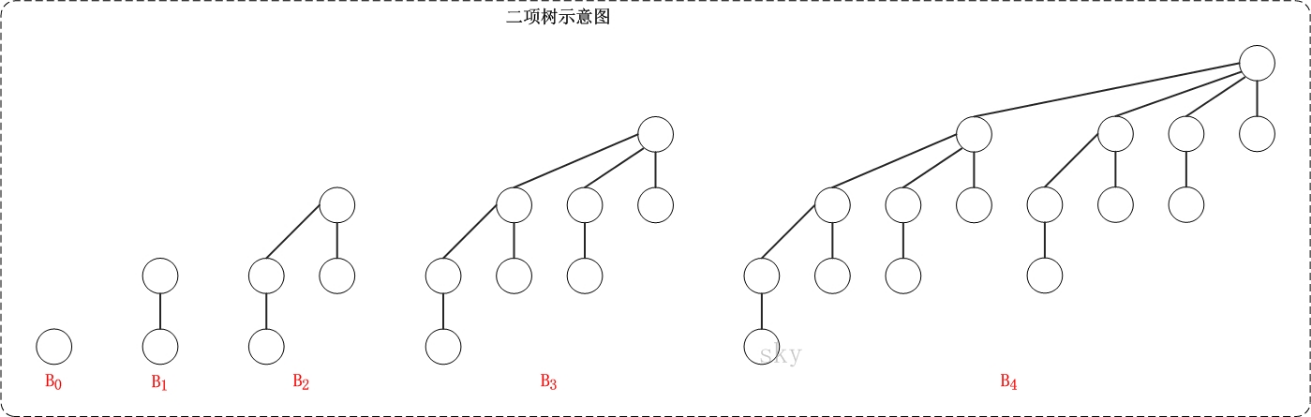
**二项堆是二项树的集合**。在了解二项堆之前，先对二项树进行介绍。

二项树是一种递归定义的有序树。它的递归定义如下：

(01) 二项树B0只有一个结点；

(02) 二项树Bk由**两棵二项树B(k-1)**组成的，其中***一棵树是另一棵树根的最左孩子***。

如下图所示：



上图的B0、B1、B2、B3、B4都是二项树。对比前面提到的二项树的定义：B0只有一个节点，B1由两个B0所组成，B2由两个B1所组成，B3由两个B2所组成，B4由两个B3所组成；而且，当两颗相同的二项树组成另一棵树时，其中一棵树是另一棵树的最左孩子。

**2、性质及介绍**

二项树有以下性质：

* **[性质一]：**Bk共有2k个节点。

如上图所示，B0有20=1节点，B1有21=2个节点，B2有22=4个节点，...

* **[性质二]：**Bk的*高度为k*。

如上图所示，B0的高度为0，B1的高度为1，B2的高度为2，...

* **[性质三]：**Bk在*深度i处恰好有C(k,i)个节点*，其中i=0,1,2,...,k。

C(k,i)是高中数学中阶乘元素，例如，C(10,3)=(10\*9\*8) / (3\*2\*1)=240

B4中深度为0的节点C(4,0)=1

B4中深度为1的节点C(4,1)= 4 / 1 = 4

B4中深度为2的节点C(4,2)= (4\*3) / (2\*1) = 6

B4中深度为3的节点C(4,3)= (4\*3\*2) / (3\*2\*1) = 4

B4中深度为4的节点C(4,4)= (4\*3\*2\*1) / (4\*3\*2\*1) = 1

合计得到B4的节点分布是(1,4,6,4,1)。

* **[性质四]：***根的度数为k，它大于任何其它节点的度数*。

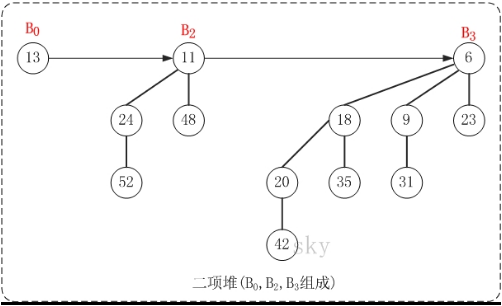
节点的度数是该结点**拥有的子树的数目**。

**注意：***树的高度和深度是相同的*。关于树的高度的概念，《算法导论》中只有一个节点的树的高度是0，而"维基百科"中只有一个节点的树的高度是1。本文使用了《算法导论中》"树的高度和深度"的概念。

**二项堆**和之前所讲的堆(二叉堆、左倾堆、斜堆)一样，也是***用于实现优先队列的***。二项堆是指**满足以下性质的二项树的集合：**

(01) 每棵二项树都*满足最小堆性质*。即，父节点的关键字 <= 它的孩子的关键字。

(02) *不能有两棵或以上的二项树具有相同的度数(包括度数为0)*。换句话说，具有度数k的二项树有0个或1个。



上图就是一棵**二项堆**，它由二项树B0、B2和B3组成。对比二项堆的定义：**(01)二项树B0、B2、B3都是最小堆；(02)二项堆不包含相同度数的二项树**。

二项堆的第(01)个性质保证了*二项堆的最小节点就是某个二项树的根节点*，第(02)个性质则说明结点数为n的二项堆**最多只有log{n} + 1棵二项树**。实际上，将包含n个节点的二项堆，表示成*若干个2的指数和(或者转换成二进制)，则每一个2个指数都对应一棵二项树*。例如，13(二进制是1101)的2个指数和为13=23 + 22+ 20, 因此具有13个节点的二项堆由度数为3, 2, 0的三棵二项树组成。（**即将节点数转换为二进制数，对应的位1的位数即为二项堆中对应的二项树**）。比如某二项堆的节点数为**15 = 23+22+21+20**，则可知该二项堆是由度为3、2、1、0的四棵二项树组成。

**3、基本操作**

二项堆是***可合并堆***，它的合并操作的复杂度是**O(log n)**。

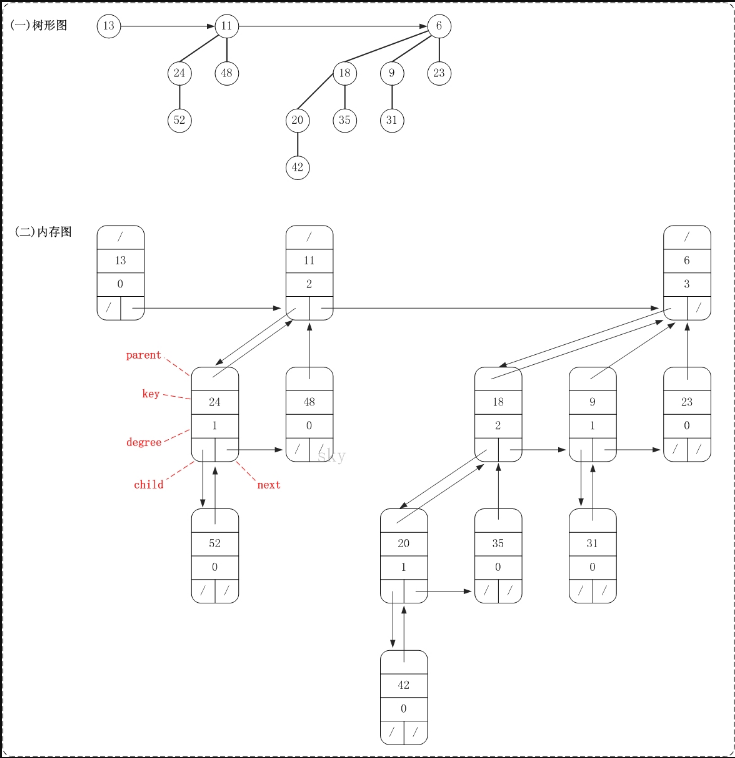
**1）基本定义**



BinomialNode是**二项堆的节点**。它包括了关键字(key)，用于比较节点大小；度数(degree)，用来表示*当前节点的度数*；左孩子(child)、父节点(parent)以及兄弟节点(next)。

BinomialHeap是二项堆对应的类，它包括了二项堆的**根节点mRoot**以及二项堆的基本操作的定义。

下面是一棵二项堆的**树形图和它对应的内存结构关系图**。



**2）合并操作**

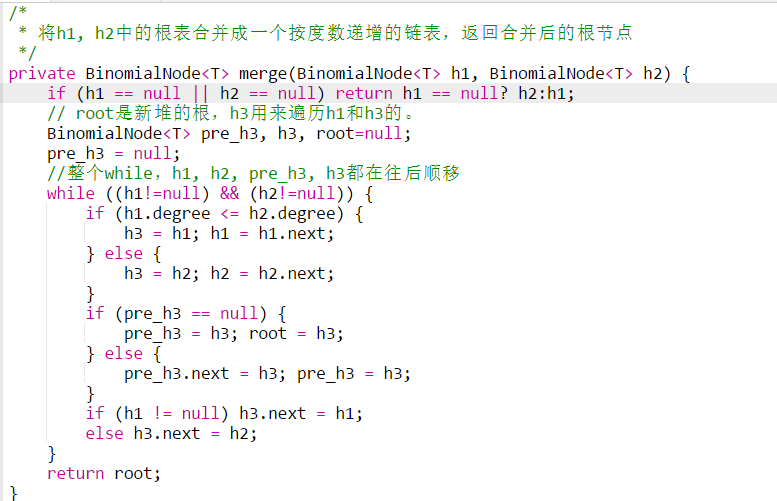
**合并操作是二项堆的重点**，它的添加操作也是基于合并操作来实现的。合并两个二项堆，需要的步骤概括起来如下：

(01) 将两个*二项堆的根链表合并成一个链表，合并后的新链表按照"节点的度数"单调递增排列*。

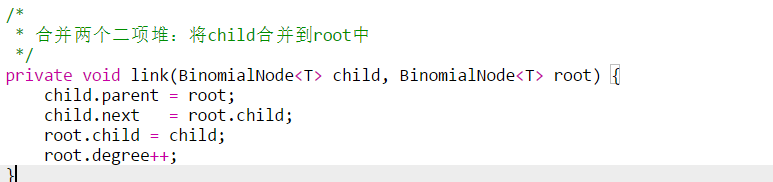
(02) 将**新链表**中"根节点度数相同的二项树"连接起来，直到所有根节点度数都不相同。

下面，先看看合并操作的代码；然后再通过示意图对合并操作进行说明。

**merge()代码：**



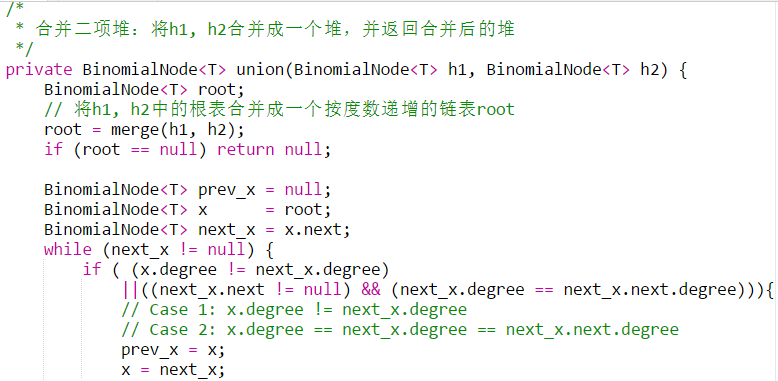
**link()代码：（将根表合并后度相同的二项树合并在一起，二项树的合并操作）**

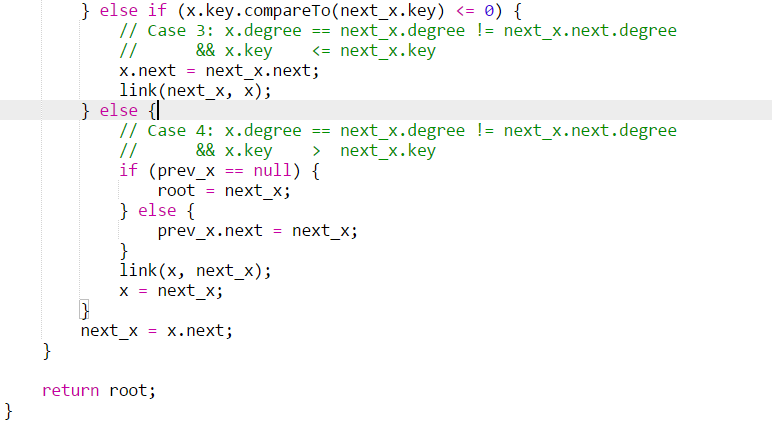


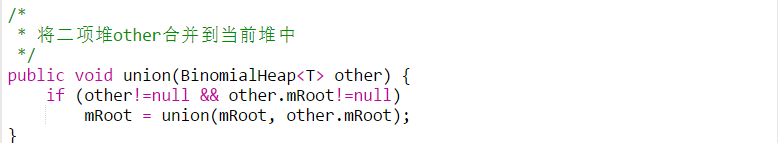
**合并操作代码(Java)：**

**注意：二项堆满足最小堆性质**

**分类讨论各种可能出现的情况**







合并函数**union(h1, h2)**的作用是将h1和h2合并，并返回合并后的二项堆。在union (h1, h2)中，涉及到了两个函数merge(h1, h2)和link(child, root)。

merge(h1, h2)就是我们前面所说的*"两个二项堆的根链表合并成一个链表，合并后的新链表按照'节点的度数'单调递增排序"*。

link(child, root)则是为了合并操作的辅助函数，它的作用是将*"二项堆child的根节点"设为"二项堆root的左孩子"，从而将child整合到root中去*。

在union (h1, h2)中对h1和h2进行合并时；首先通过 merge(h1, h2) 将h1和h2的根链表合并成一个"按节点的度数单调递增"的链表；然后进入**while循环**，对合并得到的新链表进行遍历，*将新链表中"根节点度数相同的二项树"连接起来，直到所有根节点度数都不相同为止*。在将新联表中"根节点度数相同的二项树"连接起来时，可以将被连接的情况概括为4种。

x是根链表的当前节点，next\_x是x的下一个(兄弟)节点。

* **Case 1：**x->degree != next\_x->degree

即，"当前节点的度数"与"下一个节点的度数"相等时。此时，不需要执行任何操作，继续查看后面的节点。

* **Case 2：**x->degree == next\_x->degree == next\_x->next->degree

即，"当前节点的度数"、"下一个节点的度数"和"下下一个节点的度数"都相等时。此时，暂时不执行任何操作，还是继续查看后面的节点。实际上，这里是*将"下一个节点"和"下下一个节点"等到后面再进行整合连接*。

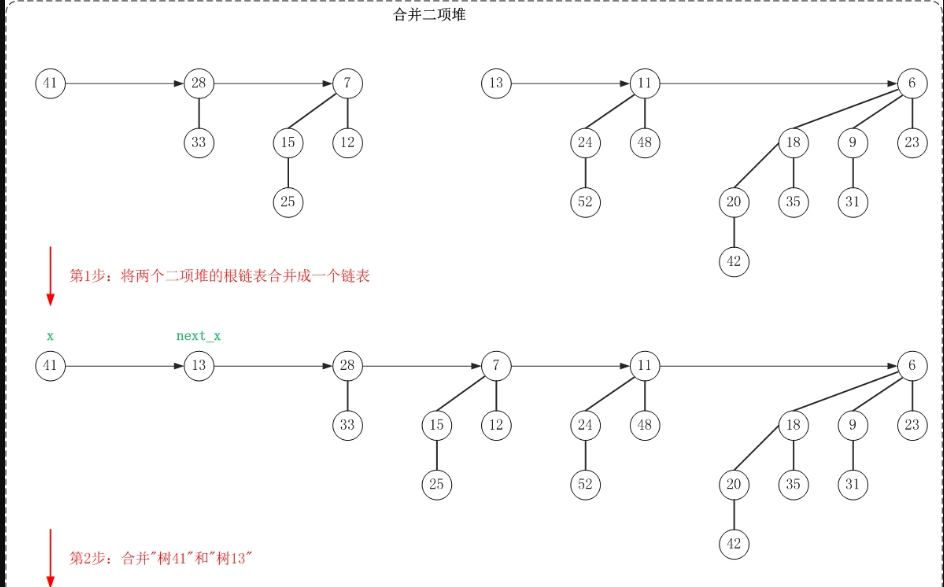
* **Case 3：**x->degree == next\_x->degree != next\_x->next->degree&& x->key <= next\_x->key

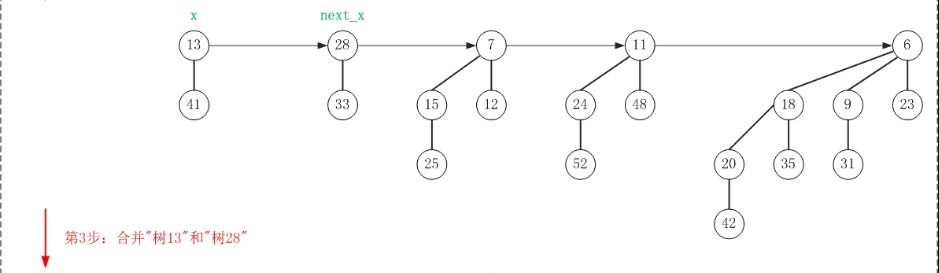
即，"当前节点的度数"与"下一个节点的度数"相等，并且"当前节点的键值"<="下一个节点的度数"。此时，将"下一个节点(对应的二项树)"作为"当前节点(对应的二项树)的左孩子"。

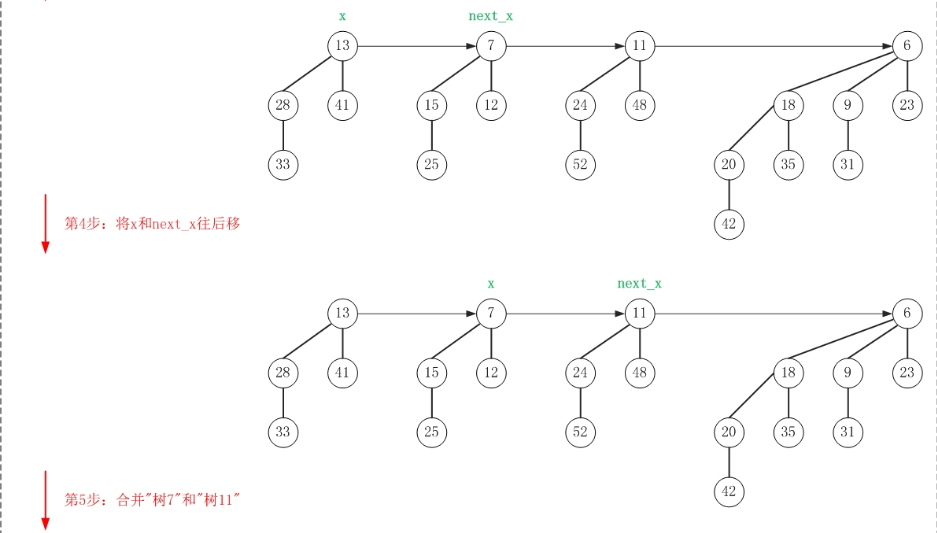
* **Case 4：**x->degree == next\_x->degree != next\_x->next->degree&& x->key > next\_x->key

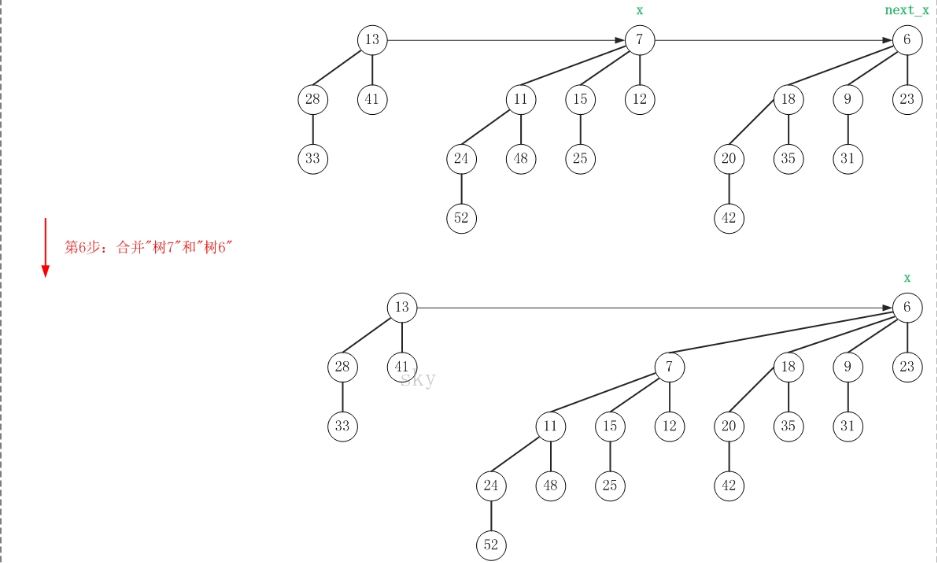
即，"当前节点的度数"与"下一个节点的度数"相等，并且"当前节点的键值">"下一个节点的度数"。此时，将"当前节点(对应的二项树)"作为"下一个节点(对应的二项树)的左孩子"。

下面通过示意图来对合并操作进行说明：









* **第1步：**将两个二项堆的根链表合并成一个链表；

执行完第1步之后，得到的新链表中有许多度数相同的二项树。实际上，此时得到的是对应"Case 4"的情况，"树41"(根节点为41的二项树)和"树13"的度数相同，且"树41"的键值 > "树13"的键值。此时，将"树41"作为"树13"的左孩子。

* **第2步：**合并"树41"和"树13"；

执行完第2步之后，得到的是对应"Case 3"的情况，"树13"和"树28"的度数相同，且"树13"的键值 < "树28"的键值。此时，将"树28"作为"树13"的左孩子。

* **第3步：**合并"树13"和"树28"；

执行完第3步之后，得到的是对应"Case 2"的情况，"树13"、"树28"和"树7"这3棵树的度数都相同。此时，将x设为下一个节点。

* **第4步：**将x和next\_x往后移；

执行完第4步之后，得到的是对应"Case 3"的情况，"树7"和"树11"的度数相同，且"树7"的键值 < "树11"的键值。此时，将"树11"作为"树7"的左孩子。

* **第5步：**合并"树7"和"树11"；

执行完第5步之后，得到的是对应"Case 4"的情况，"树7"和"树6"的度数相同，且"树7"的键值 > "树6"的键值。此时，将"树7"作为"树6"的左孩子

* **第6步：**合并"树7"和"树6"

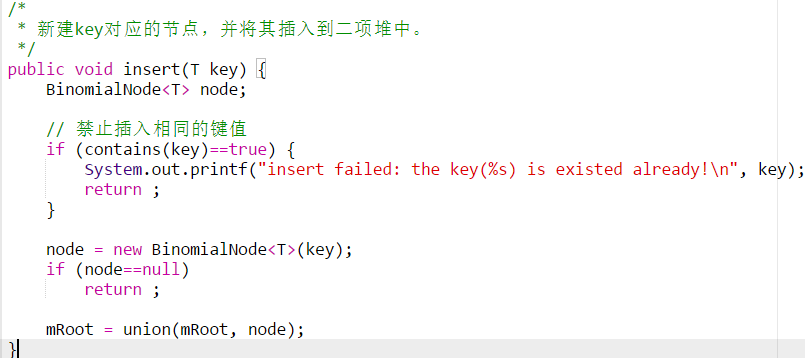
此时，合并操作完成！

**PS. 合并操作的图文解析过程与"测试程序(Main.java)中的testUnion()函数"是对应的！**

**3）插入操作**

理解了"合并"操作之后，插入操作就相当简单了。*插入操作可以看作是将"要插入的节点"和当前已有的堆进行合并*。

插入操作代码：



在插入时，首先*通过contains(key)查找键值为key的节点*。存在的话，则直接返回；不存在的话，则新建BinomialNode对象node，然后将node和heap进行合并。

**注意：**我这里实现的二项堆是**"禁止插入相同节点的"**！若你想允许插入相同键值的节点，则屏蔽掉插入操作中的contains(key)部分代码即可。

**4）删除操作（某个节点）**

***删除二项堆中的某个节点***，需要的步骤概括起来如下：

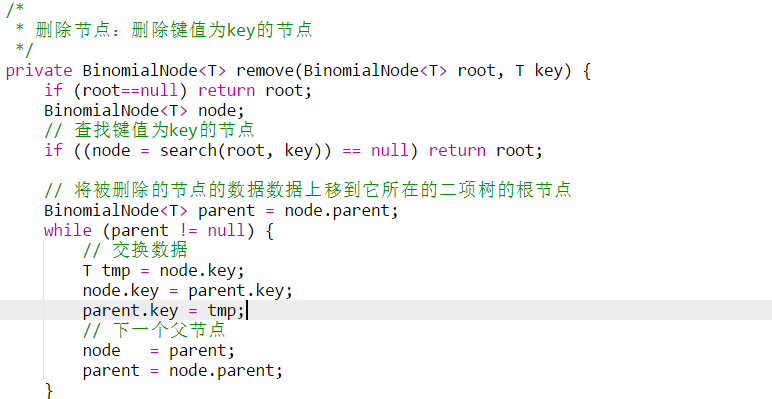
(01) 将"该节点"*交换到"它所在二项树"的根节点位置*。方法是，从"该节点"不断向上(即向树根方向)"遍历，不断交换父节点和子节点的数据，直到被删除的键值到达树根位置。

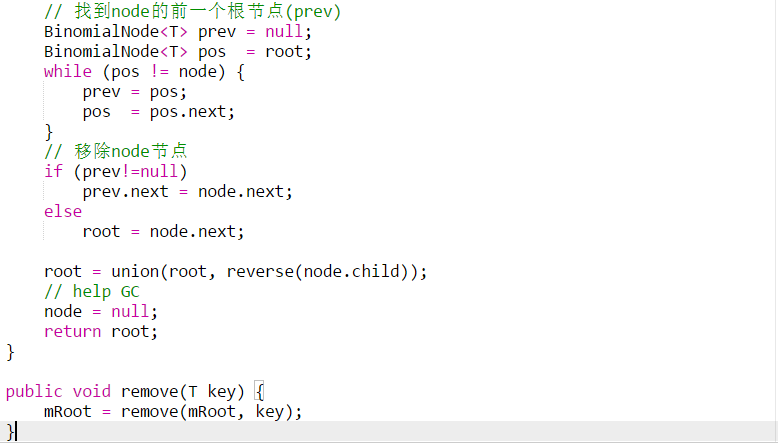
(02) 将"该节点所在的二项树"从二项堆中**移除**；将***该二项堆记为heap***。

(03) 将"该节点所在的二项树"进行**反转**。反转的意思，就是*将根的所有孩子独立出来，并将这些孩子整合成二项堆，将该二项堆记为child*。

(04) 将child和heap进行合并操作。

下面，先看看删除操作的代码；再进行图文说明。

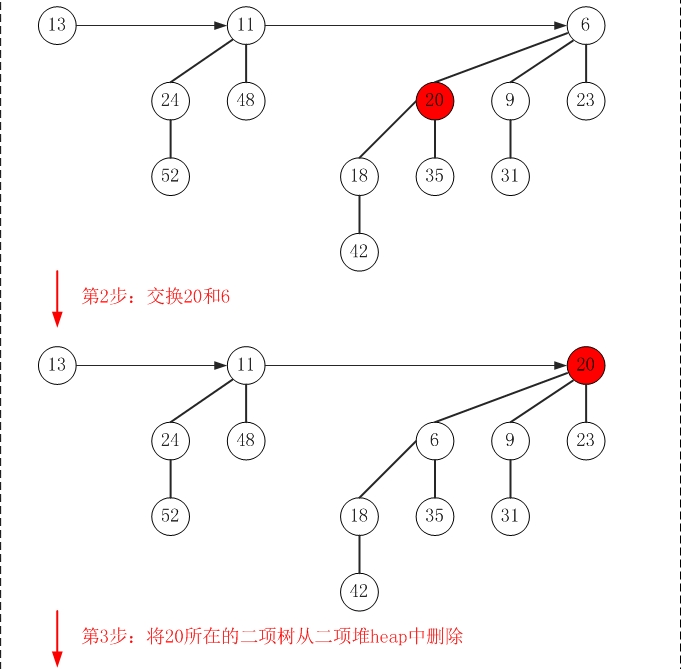


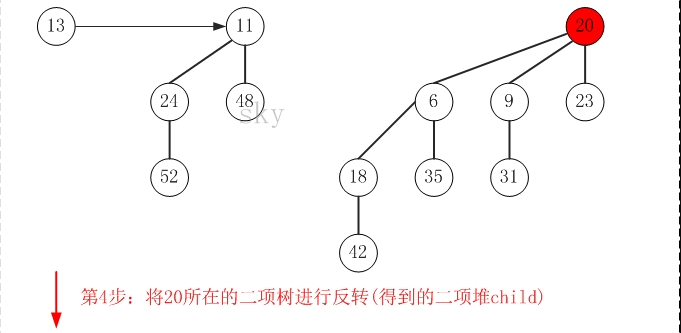


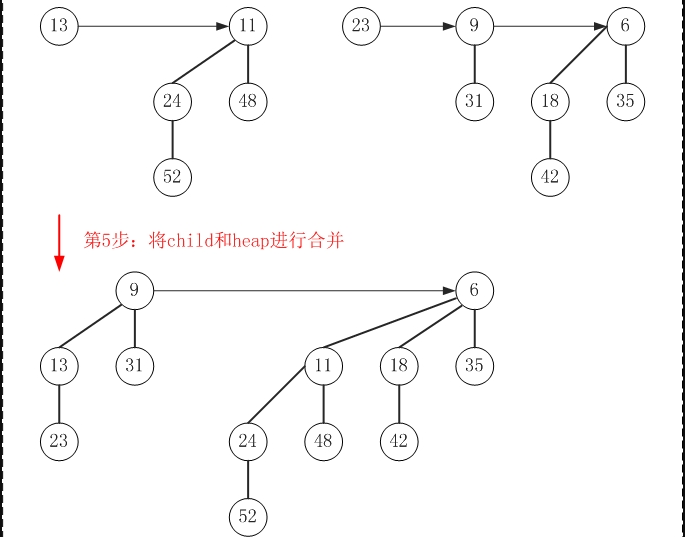
**remove(key)**的作用是***删除二项堆中键值为key的节点，并返回删除节点后的二项堆***。

下面通过示意图来对删除操作进行说明(删除二项堆中的节点20)。









**总的思想**，就是将被"删除节点"从它所在的二项树中孤立出来，然后再对二项树进行相应的处理。

PS. 删除操作的图文解析过程与"测试程序(Main.java)中的testDelete()函数"是对应的！

**5）更新操作（修改节点的值）**

更新二项堆中的某个节点，就是修改节点的值，它包括两部分分：**"减少节点的值" 和 "增加节点的值"** 。

**更新操作代码：**



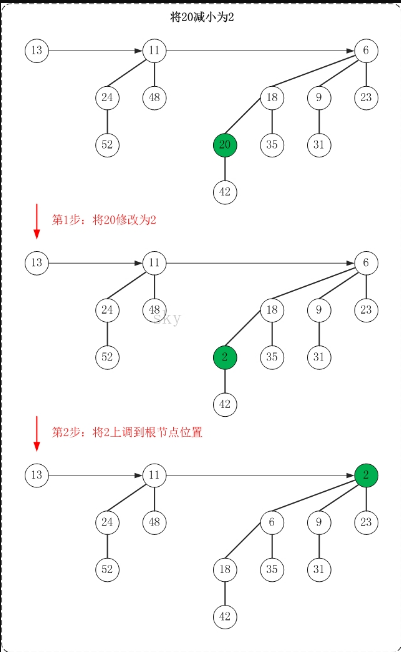
**A. 减少节点的值**

减少节点值的**操作很简单**：该节点*一定位于一棵二项树中，减小"二项树"中某个节点的值后要保证"该二项树仍然是一个最小堆"*；因此，就需要我们**不断的将该节点上调**。

减少操作代码：



下面是减少操作的示意图(20->2)



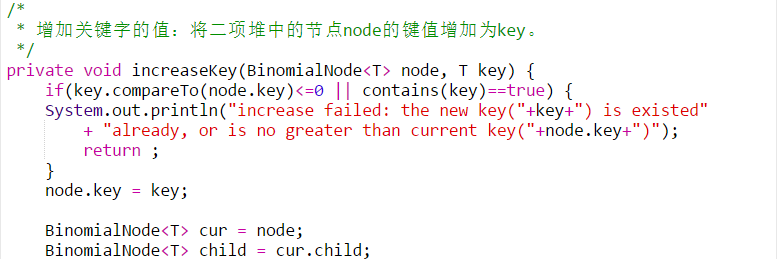
减少操作的思想很简单，就是**"保持被减节点所在二项树的最小堆性质"**。

PS. 减少操作的图文解析过程与"测试程序(Main.java)中的testDecrease()函数"是对应的！

**B. 增加节点的值**

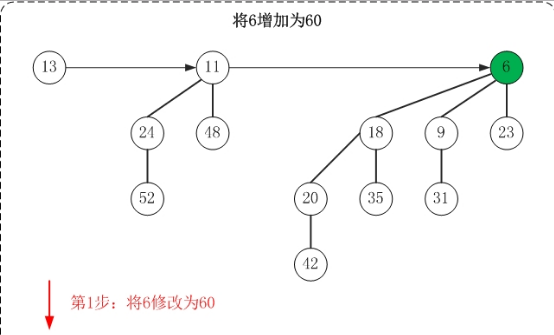
增加节点值的操作也很简单。上面说过减少要将被减少的节点不断上调，从而保证"被减少节点所在的二项树"的最小堆性质；而增加操作则是将被增加节点**不断的下调**，从而***保证"被增加节点所在的二项树"的最小堆性质***。

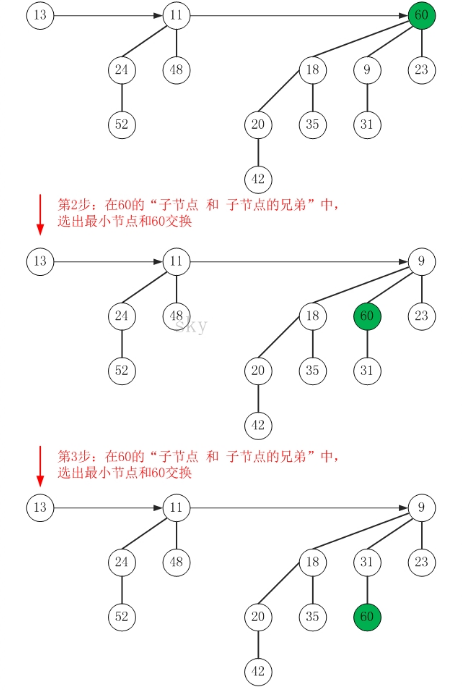
增加操作代码：





下面是增加操作的示意图(6->60)

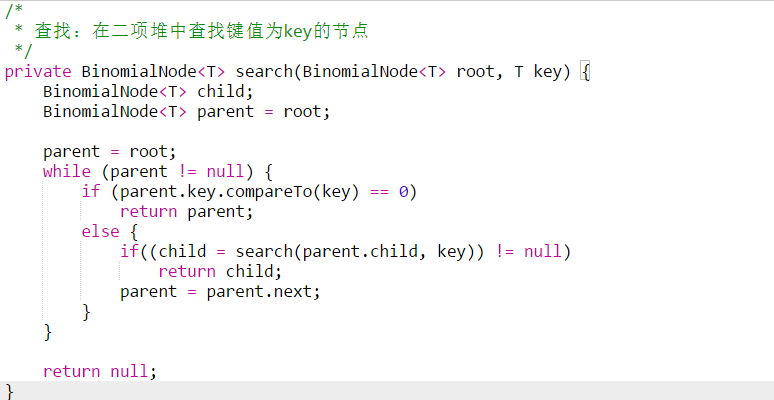




增加操作的思想很简单，"保持被增加点所在二项树的最小堆性质"。

**PS. 增加操作的图文解析过程与"测试程序(Main.java)中的testIncrease()函数"是对应的！**

**6）搜索节点**

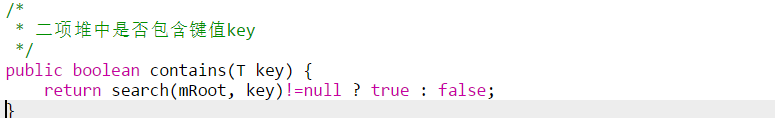


**7）reverse反转操作**

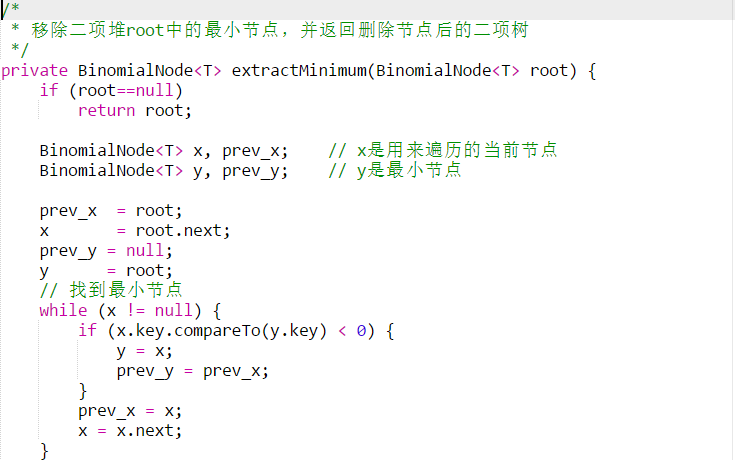
反转示意图可参考删除操作部分的示意分析图

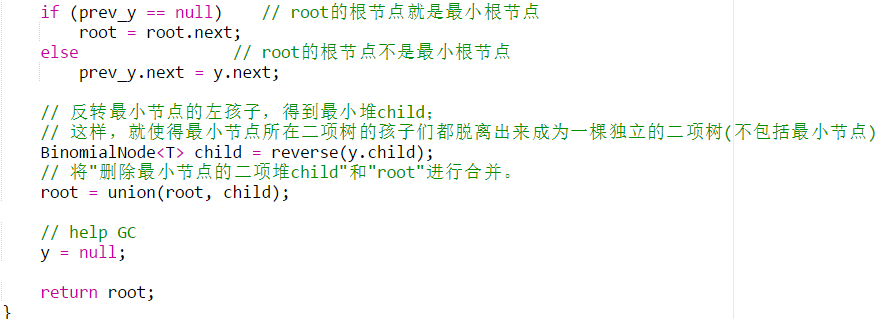


**8）contains包含操作**



**9）移除最小节点**





**二、斐波那契堆**

<http://www.cnblogs.com/skywang12345/p/3659122.html>

<http://www.cnblogs.com/lengyue365/p/5099608.html>

<http://www.cse.yorku.ca/~aaw/Jason/FibonacciHeapAnimation.html>

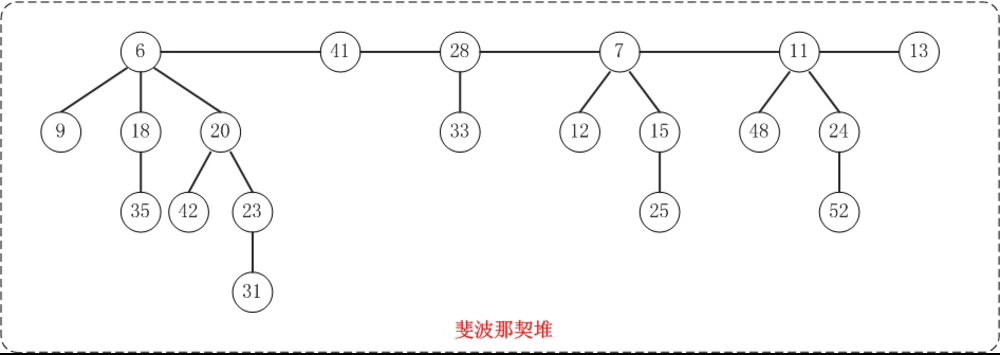
<http://gdeepak.com/IADSA/L22binomialfibonacciheaps.pdf>

**1、介绍**

斐波那契堆(Fibonacci heap)是一种**可合并堆（基于一个循环双向链表实现的）**，可用于*实现合并优先队列*。它比二项堆具有更好的平摊分析性能，它的**合并操作的时间复杂度是O(1)**。与二项堆一样，它也是由*一组堆最小有序树组成，并且是一种可合并堆*。

斐波那契堆是可合并堆在不涉及删除的操作（除去EXTRACT和DELETE）中，操作仅需O(1)的**平摊运行时间**。当EXTRACT和DELETE的操作数目较小时，斐波那契堆能得到较好的运行效率。斐波那契堆不能有效地支持SEARCH操作，可用于解决诸如*最小生成树和寻找单源最短路径等问题的快速算法*都要用到**斐波那契堆**。

与二项堆不同的是，**斐波那契堆中的树不一定是二项树**；而且**二项堆中的树是有序排列的，但是斐波那契堆中的树都是有根而无序的**。



在计算机科学中，*斐波那契堆是由树的集合所组成的堆数据结构*。它比二项堆的平摊运行时间更好。斐波那契堆的名字来自于**斐波那契数列**，这些数列被用来做运行时间分析。

求最小值(find-mininum)、插入(insert)、降低元素值(decrease-key)和合并(merge/union)可以在**常数平摊时间**内完成。删除(delete)和删除最小值(delete minimun)可以在O(log n)平摊时间内完成。

在**优先队列(priority queues)中使用斐波那契堆**可以提升重要算法的渐进运行时间，例如**Dijkstra算法**，该算法用来计算一个图中两个结点的最短的距离。

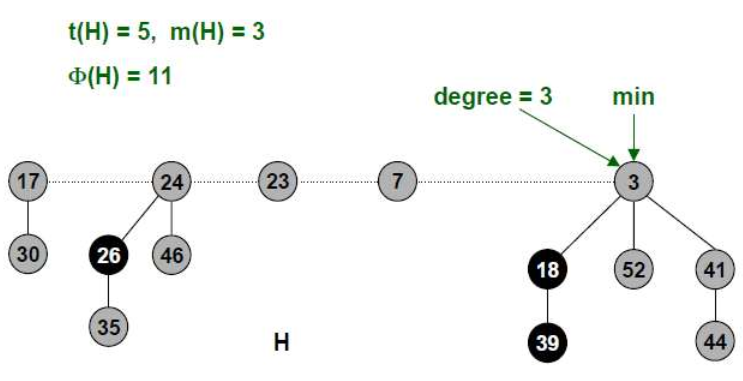
斐波那契堆是由一些树的集合所组成，其中*每一棵树都满足最小堆(minimum-heap)的属性，也就是说树中每个子结点的值都大于或者等于其父结点的值，而最小值则处在根结点上*。

与二项堆不同，斐波那契堆中的树更加灵活，没有规定的形状，在极端情况下，堆中每个元素都是一棵单独的树。这种灵活性使得一些操作可以以**“偷懒”**的方式来执行，而“剩下”的工作*将推迟到后面的操作中来完成*。比如**堆的合并仅仅将由树所组成的链表链接起来**，而*降低元素值(decrease key)有时直接从父结点中剪断而形成一棵新树*。

下面给出斐波那契堆中**关键的量值：**

* **degree[x]:** 表示结点x的子结点个数
* **mark[x]:** 一个结点是否**被marked了**（当执行decrease key操作时会用到）
* **t(H):** 表示堆中树的个数
* **m(H):** 表示被marked的结点数量
* **Φ(H)=t(H)+2m(H):** 表示势函数

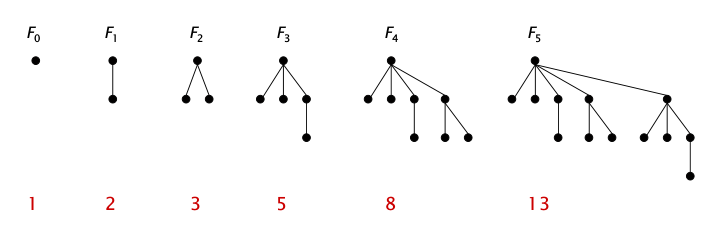
如下图所示：从图中可以看出，一共有5棵树，即t(H)=5，**最小值的指针**指向元素值为3的根结点，在这棵包含最小值的树中，根结点有3个子结点，所以其degree等于3，整个斐波那契堆用 H 来表示。其中黑色表示被marked了，灰色表示没有被marked，所以m(H)=3，势函数**Φ(H)=t(H)+2m(H)=5+2\*3=11**



下面来看看“斐波那契堆”的名字是怎么来的，在 **decrease key 的操作**中，假设一个结点P的元素值为A，现在要将其降低为B，其父结点的值为C，则遵循这样的规则（降低节点值）：

* 若B>=C, 则直接将A替换为B即可；
* 若B<C, 结点P的父结点是**根结点**，则直接将结点P从根结点处剪断，形成一棵新树；
* 若B<C, 结点P的父结点**不是根结点，**则将结点P从**父结点**处剪断，形成一棵新树，然后将父结点设置为**marked**；
* 若B<C, 并且父结点已经被marked了，则***不仅要将结点P从父结点处剪断***，而且依次往上递归*将被marked的父结点也从其父结点处剪断，直到碰到一个父结点没有被marked，或者碰到根结点为止*。

由上面的规则可以得到，只能将父结点下面的一个子结点剪断并形成一颗新树，此时父结点已经被marked了，如果再执行一次 decrease key 操作，并且需要从父结点处剪断，那么此时父结点本身也将被剪断，同时父结点的父结点的 degree 将变为 degree -1 ，所以在*保证父结点 degree 不发生改变的情况下，子结点从父结点处剪断的操作只能执行一次*，这样对于一个给定的结点，在其 degree 不发生改变的情况下，**子结点的个数**（加上根结点）有一个最小值，这些**最小值满足斐波那契数列**，如下图所示：



从图中可以看出，第一棵树的 degree 为0，结点数目为1，第二棵树的 degree 为1，结点数目为2，只有一个子结点，此时如果执行 decrease key 操作，并且要将子结点从父结点处剪断，那么这棵树的根结点就没有子结点了，也就是说 degree 变成0了，此时就与第一棵的情况相同了，第三棵树同理可以得出。而 degree 从1开始一直往下，树的结点个数就形成了斐波那契数列，多么美妙啊，其中 **degree=k 的结点个数为 Fk+2**, 表示斐波那契数列中的第k+2项。（波那契数列**Fk+2=Fk+1 + Fk**）

常见堆的常见操作时间复杂度比较，其中N代表结点个数



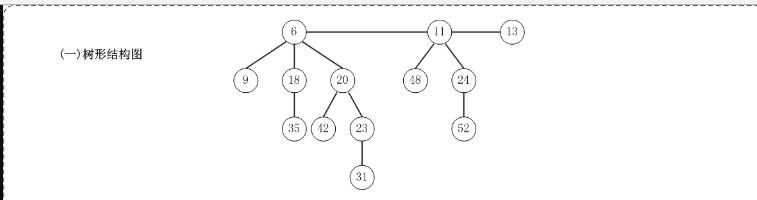
**2、基本操作**

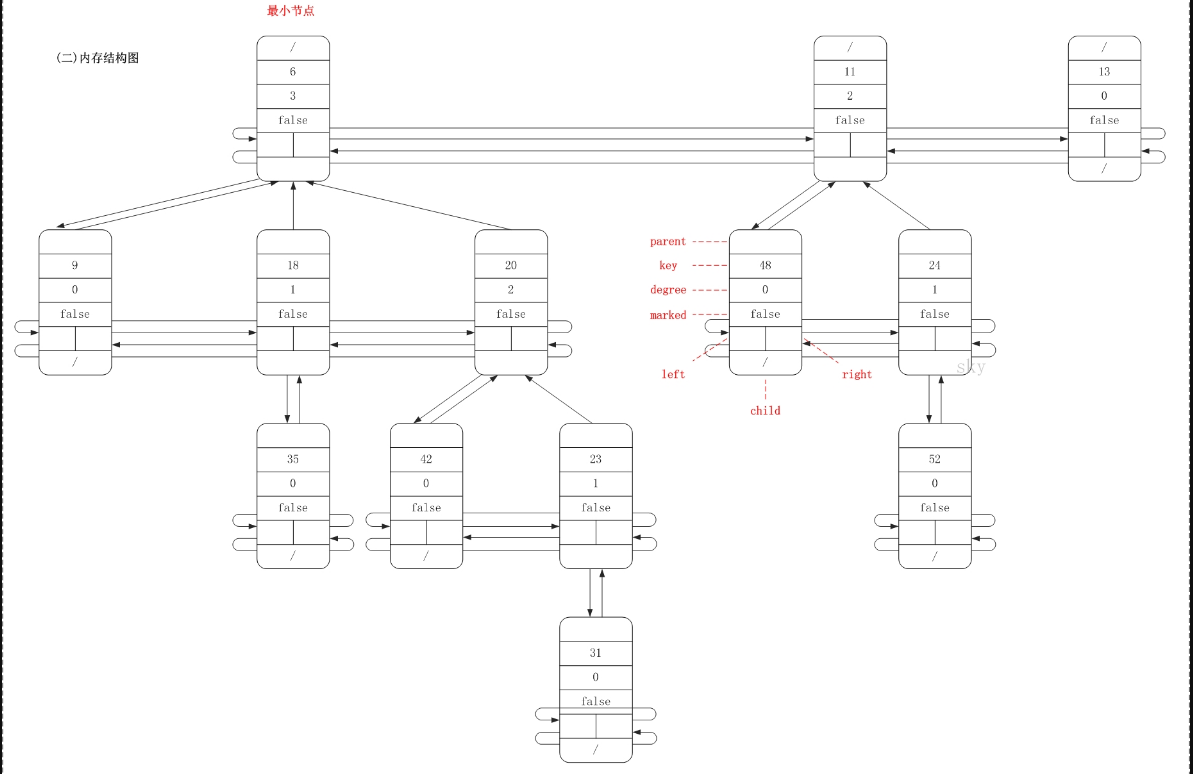
**1）基本定义（循环双链表结构）**



FibNode是斐波那契堆的节点类，它包含的信息较多。***key是用于比较节点大小的，degree是记录节点的度，left和right分别是指向节点的左右兄弟，child是节点的第一个孩子，parent是节点的父节点，marked是记录该节点是否被删除第1个孩子(marked在删除节点时有用)***。

FibHeap是斐波那契堆对应的类。min是保存当前堆的**最小节点**，keyNum用于记录**堆中节点的总数**，maxDegree用于记录堆中最大度，而cons在删除节点时来暂时保存堆数据的临时空间。



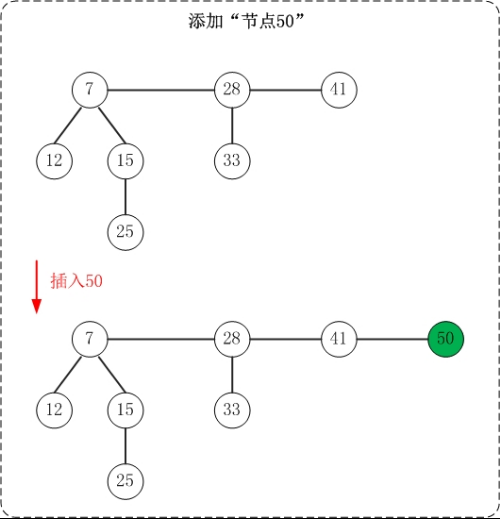


上面是斐波那契堆的两种不同结构图的对比。从中可以看出，斐波那契堆是由一组最小堆组成，这些**最小堆的根节点组成了循环双向链表(后文称为"根链表")**；**斐波那契堆中的最小节点就是"根链表中的最小节点"！**

PS. 上面这幅图的结构和测试代码中的"基本信息"测试函数的结果是一致的；你可以通过测试程序来亲自验证！

**2）插入操作**

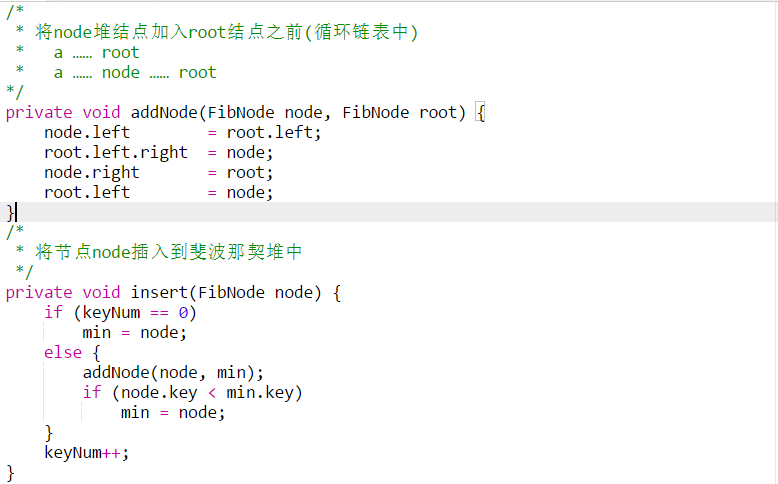
插入操作非常简单：插入一个节点到堆中，直接将该节点插入到**"根链表的min节点"**之前即可；若被插入节点比"min节点"小，则更新"min节点"为被插入节点。

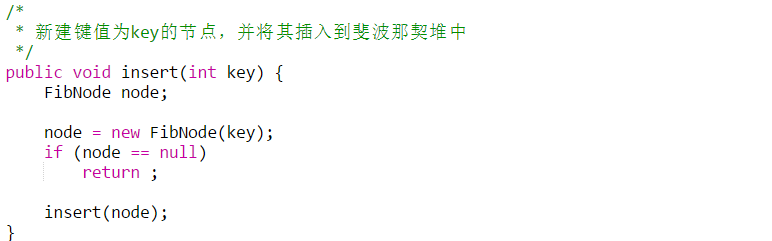


上面是插入操作的示意图。**波那契堆的根链表是"循环双向链表"**，这里*将min节点看作双向联表的表头(后文也是如此)*。在插入节点时，每次都是"将节点插入到min节点之前(即插入到双链表**末尾**)"。此外，对于根链表中最小堆都只有一个节点的情况，插入操作就很演化成双向链表的插入操作。

此外，插入操作示意图与测试程序中的"插入操作"相对应，感兴趣的可以亲自验证。

插入操作代码：（加入到同一子链中，即插入到根节点之前）





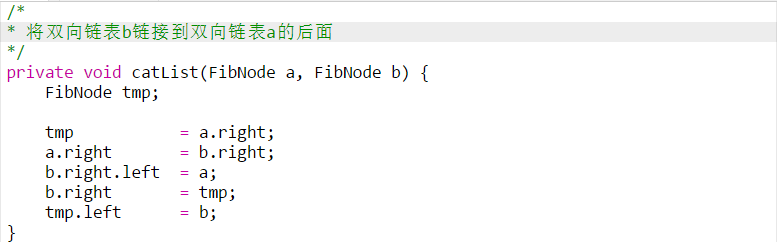
**3）合并操作**

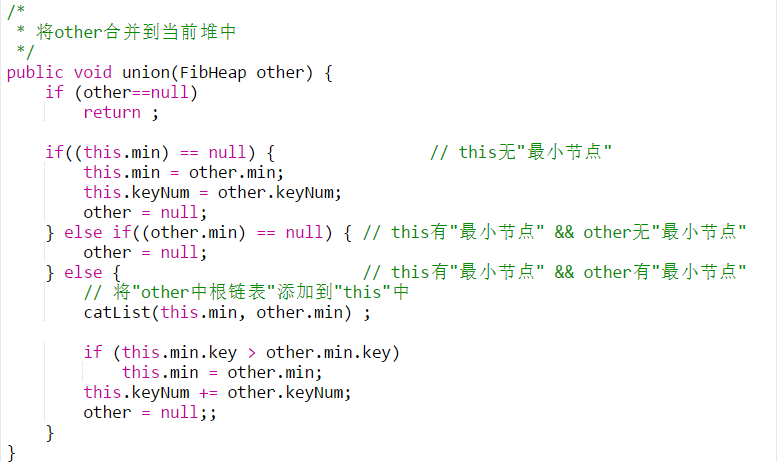
合并操作和插入操作的原理**非常类似**：将一个堆的根链表插入到另一个堆的根链表上即可。简单来说，就是***将两个双链表拼接成一个双向链表***。



上面是合并操作的示意图。该操作示意图与测试程序中的"合并操作"相对应！

合并操作代码：**合并后注意更新新链表的最小节点，以及堆中的节点数**



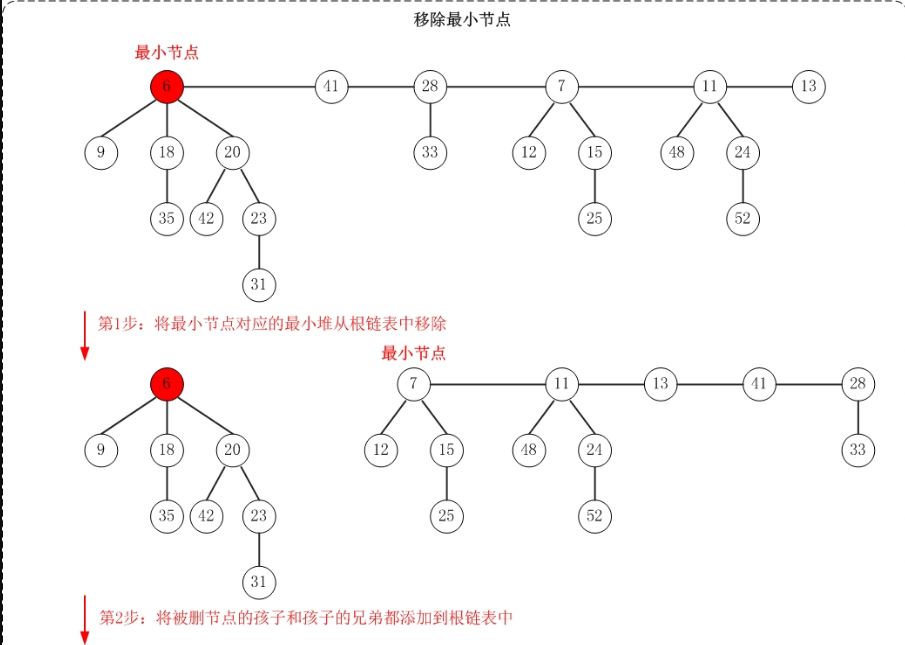


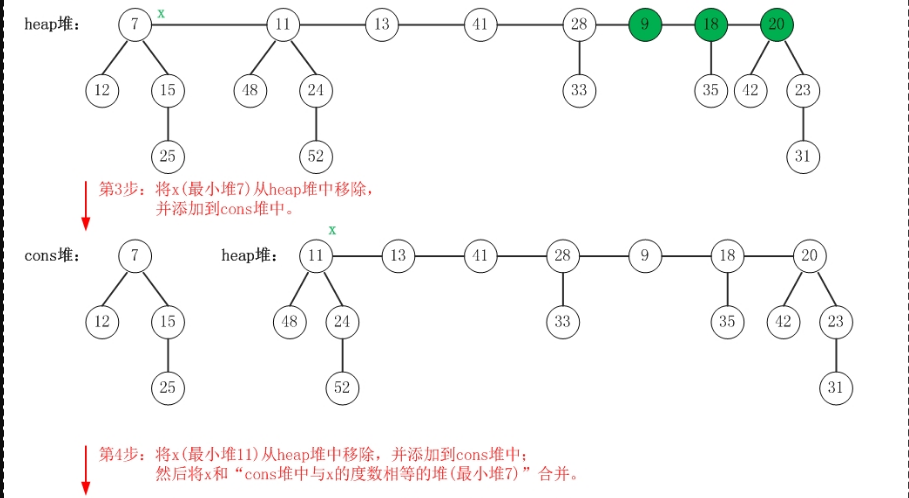
**4）取出最小节点**

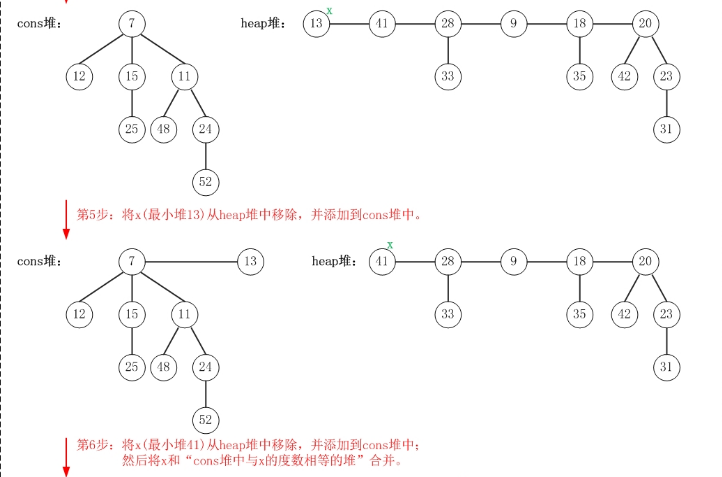
抽取最小结点的操作是斐波那契堆中较复杂的操作。

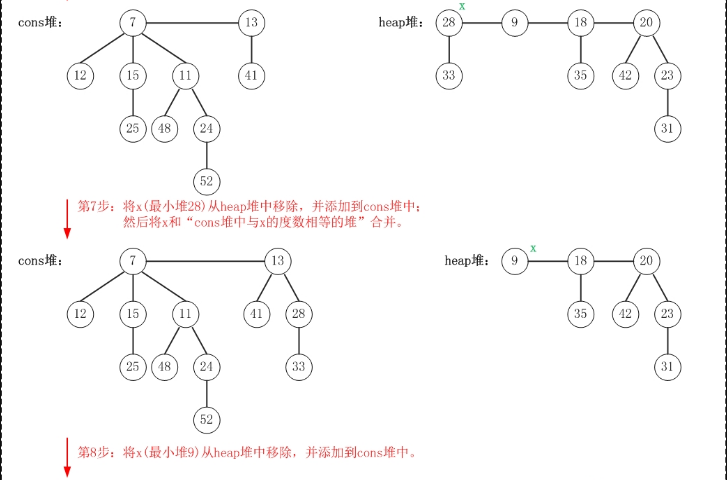
(1）将要*抽取****最小结点的子树****都直接串联在根表中*；

(2）*合并所有degree相等的树，直到没有相等的degree的树*。



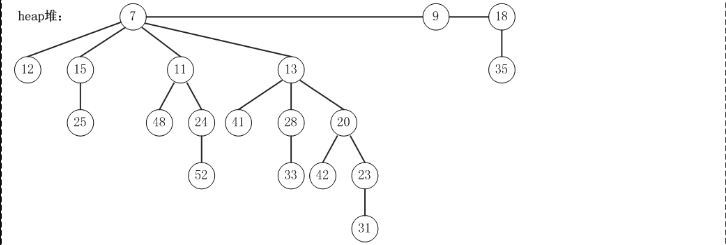








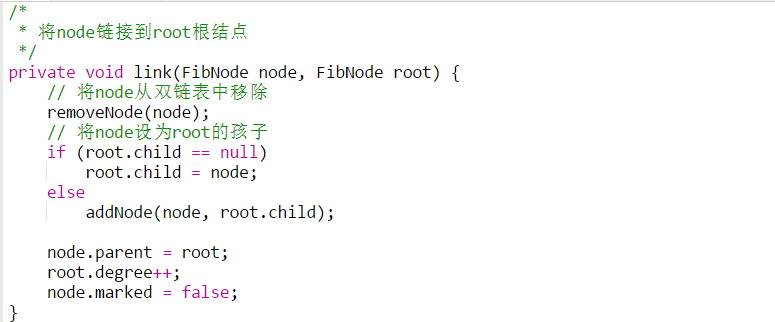


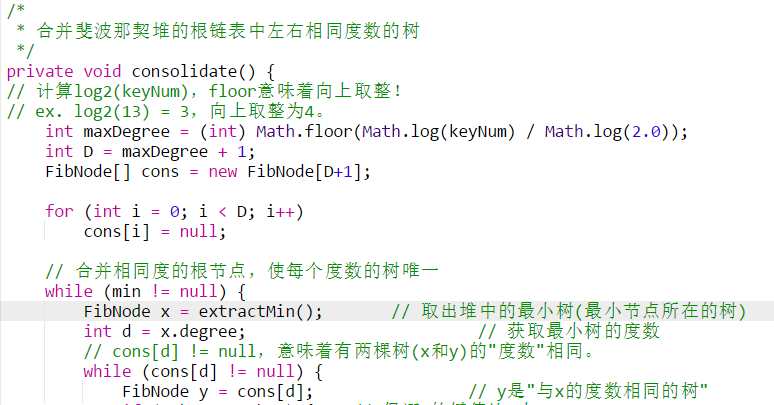


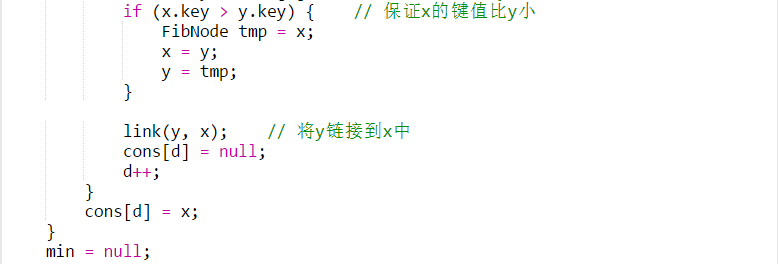
上面是取出最小节点的示意图。图中应该写的非常明白了，若有疑问，看代码。

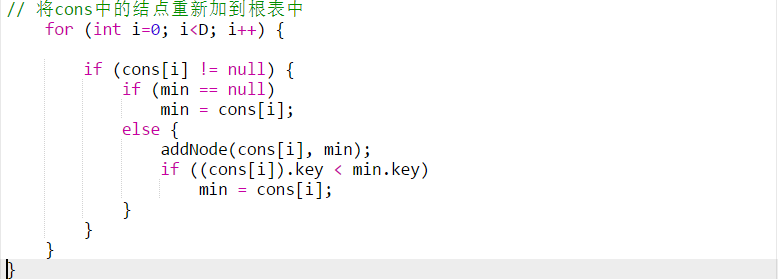
此外，该操作示意图与测试程序中的"删除最小节点"相对应！有兴趣的可以亲自验证。

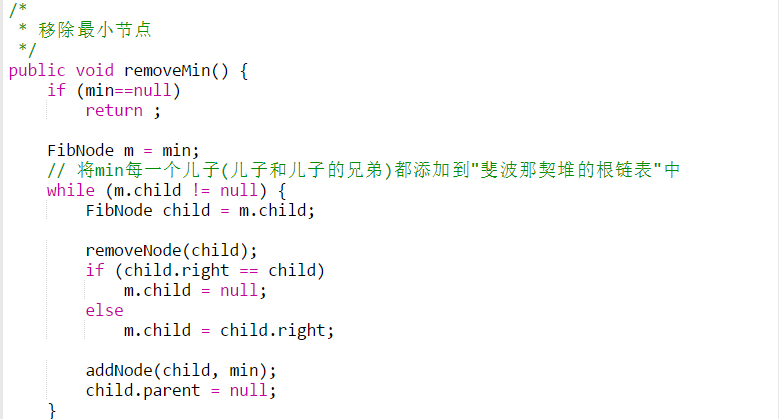
**取出最小节点代码：**

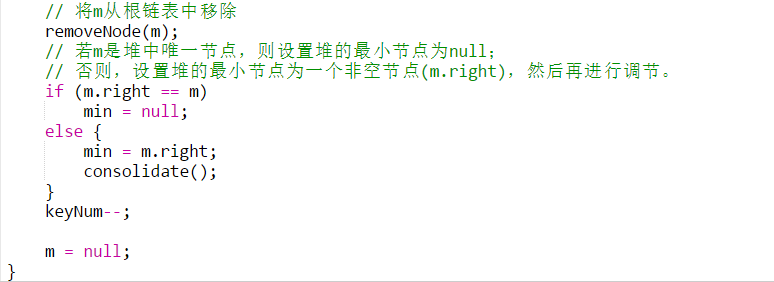












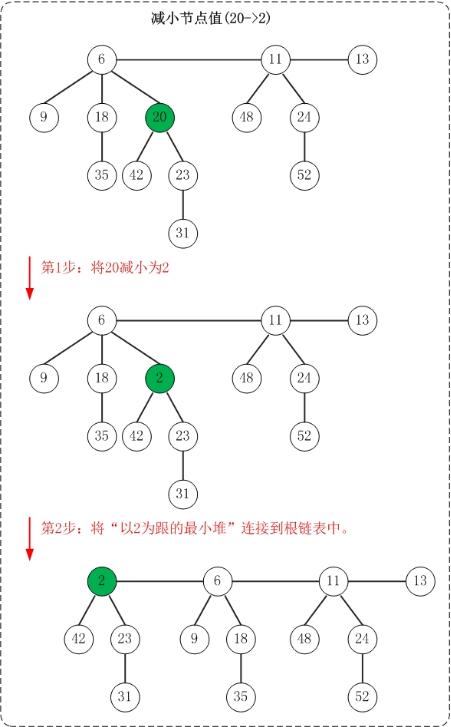
**5）减小节点值**

减少斐波那契堆中的节点的键值，这个操作的难点是：**如果减少节点后破坏了"最小堆"性质，如何去维护呢？**下面对一般性情况进行分析。

* (1)首先，将"被减小节点"从"它所在的最小堆"**剥离出来**；然后将"该节点"关联到"根链表"中。 倘若被减小的节点不是单独一个节点，而是**包含子树的树根**。则是将以"被减小节点"为根的子树从"最小堆"中剥离出来，然后将该树**关联到根链表**中。
* (2) 接着，对"被减少节点"的原父节点进行**"级联剪切"**。所谓"级联剪切"，就是在被减小节点破坏了最小堆性质，并被切下来之后；再从"它的父节点"进行递归级联剪切操作。

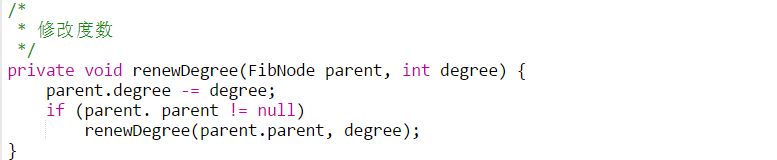
而**级联操作的具体动作**则是：若父节点(被减小节点的父节点)的**marked标记为false**，则将其设为true，然后退出。否则，将父节点从最小堆中切下来(方式和"切被减小节点的方式"一样)；然后*递归对祖父节点进行"级联剪切"*。**marked标记的作用就是用来标记"该节点的子节点是否有被删除过"**，它的作用是来实现级联剪切。而**级联剪切的真正目的是为了防止"最小堆"由二叉树演化成链表**。

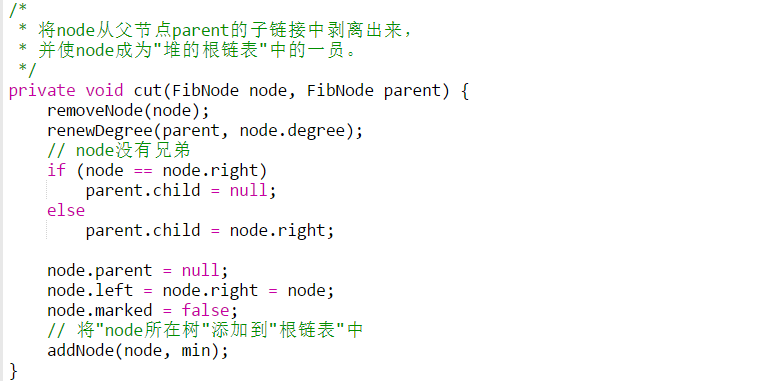
* (3) 最后，别忘了对根链表的最小节点进行更新。

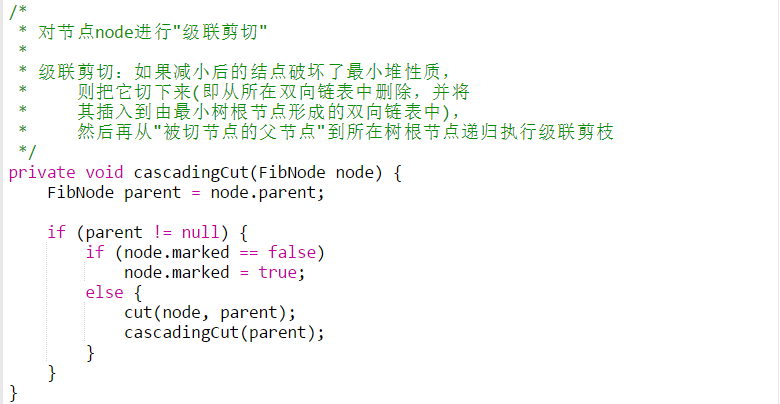


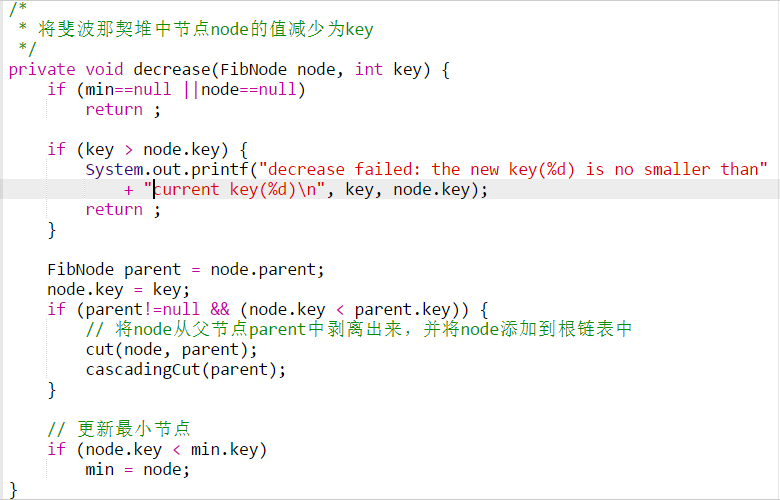
上面是减小节点值的示意图。该操作示意图与测试程序中的"减小节点"相对应！

**减小节点值的代码：（注意删除节点或节点树，要及时更新整个最小堆树对应的根节点的度）**







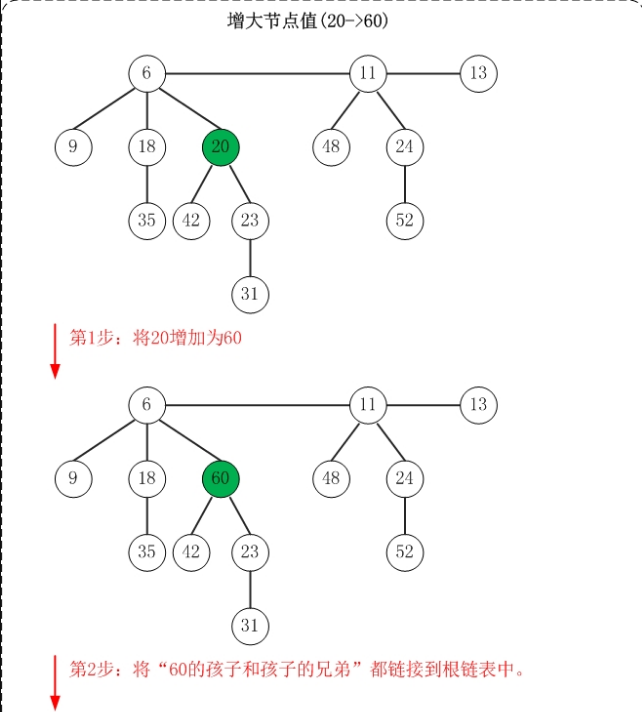


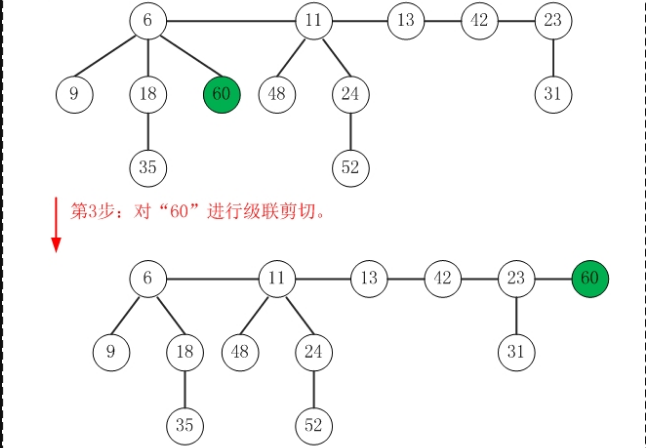
**6）增加节点值**

增加节点值和减少节点值类似，这个操作的难点也是如何维护"最小堆"性质。思路如下：

(1) 将*"被增加节点"的"左孩子和左孩子的所有兄弟"都链接到根链表中*。

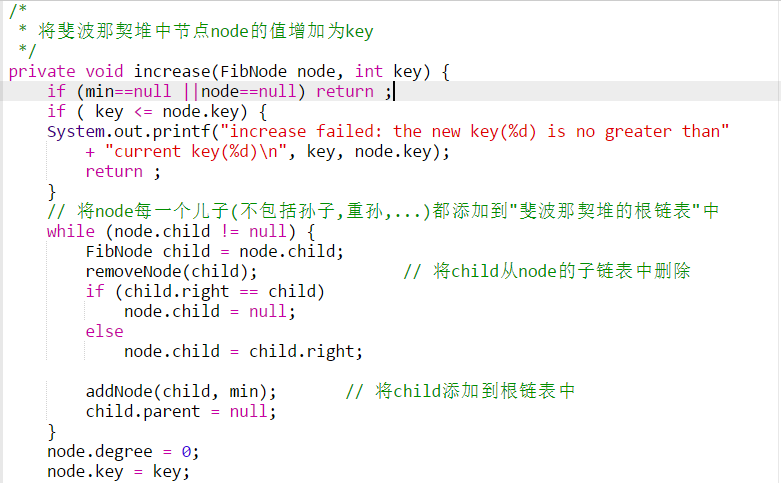
(2) 接下来，把"被增加节点"添加到根链表；但是别忘了对其进行**级联剪切**。

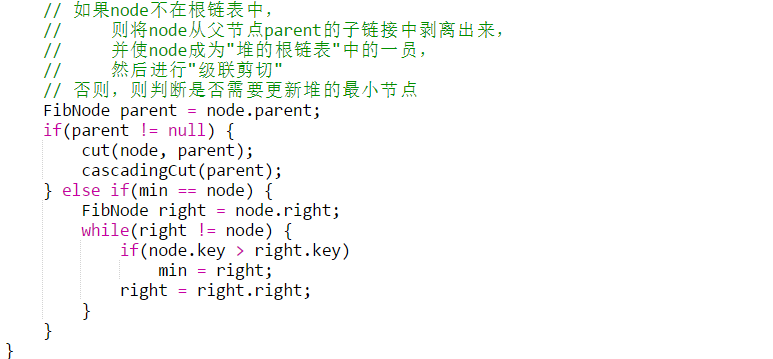




上面是增加节点值的示意图。该操作示意图与测试程序中的"增大节点"相对应！

**增加节点值的代码：**



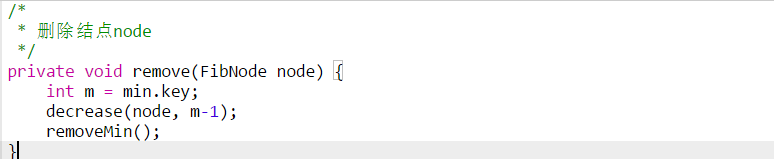


**7）删除节点**

删除节点，本文采用了操作是：**"取出最小节点"和"减小节点值"的组合**。

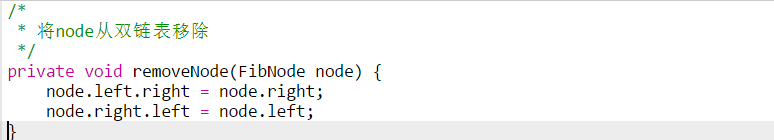
(1) 先*将被删除节点的键值减少。减少后的值要比"原最小节点的值"小即可*。

(2) 接着，*取出最小节点*即可。



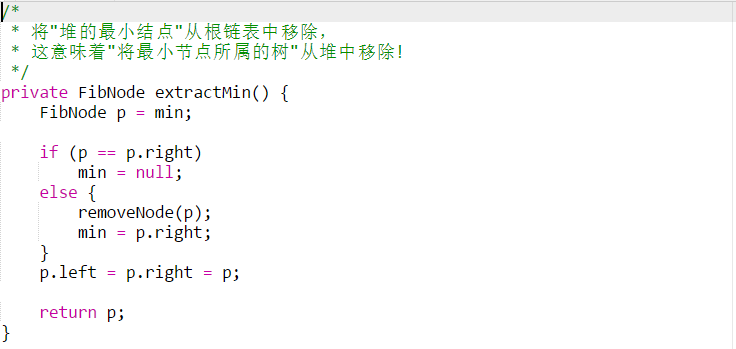
**8）removeNode**

从链表中移除节点，即将该节点设置为孤立的节点



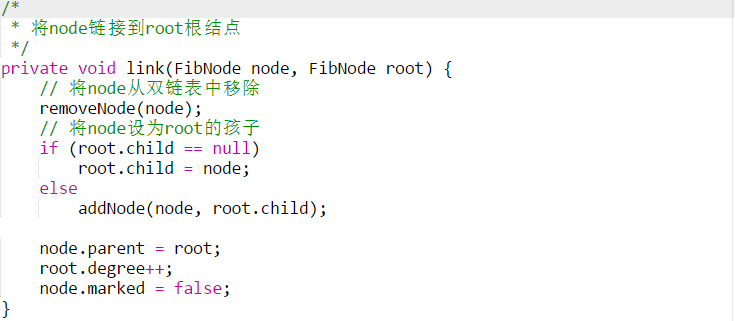
**9）extractMin**

将"堆的最小结点"从根链表中移除



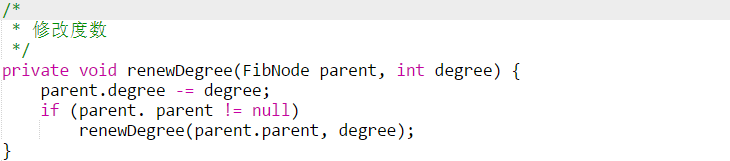
**10）link**

将该节点链接到对应的root节点，即将该节点设置为root的child节点



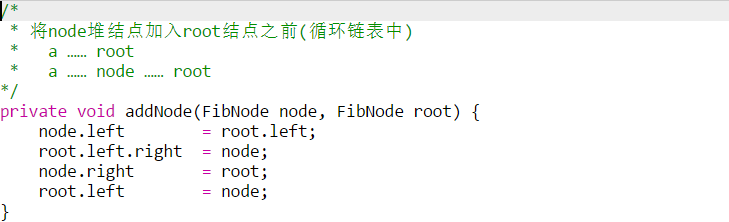
**12）renewDegree**

修改节点的度数，即当某一节点的child节点发生变化时，则需要修改对应父节点的度数**（级联修改）**



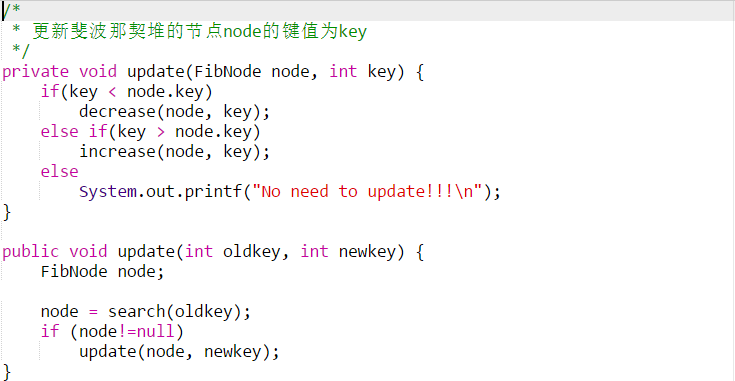
**13）addNode**

将节点加入到对应树根节点的子链表中

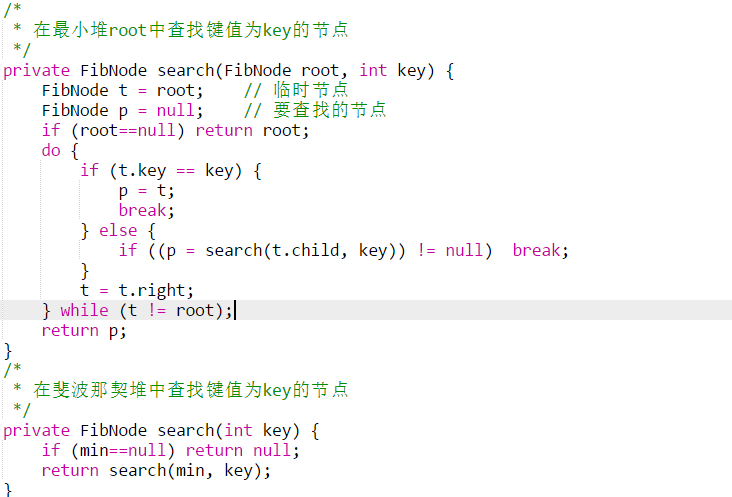


**14）更新节点值**

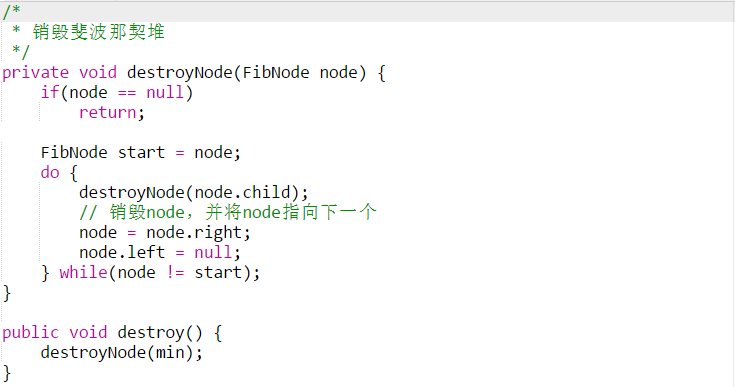
更新某一节点的key值，同理根据更新后节点值与原来值的大小关系，分类讨论分析：即增加或减小节点值



**15）搜索节点值**



**16）销毁堆**



**3、实现**

参考：<http://www.cnblogs.com/skywang12345/p/3659122.html> 非常详细

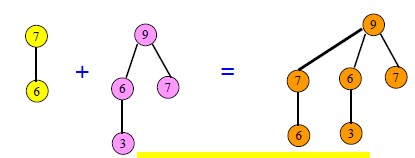
**三、Pairing 堆（配对堆）**

配对堆(Pairing Heap)是一个简单实用的min-heap结构（当然也可以做成max-heap）。它是一颗**多路树(multiway tree)**，类似于Leftist Heap和Skew Heap，但是与Binomial Tree和Fibonacci Heap不一样。它的基本操作是*两个多路树的连接(link)*，所以取名叫Pairing Heap。连接操作(参考以下实现中的方法**linkPair**)类似于Binomial Tree和Fibonacci Heap中的link操作，即*将root key值最大的树作为key值最小的树的孩子（一般作为最左边的孩子，特别是Binomial Heap必须这样做）*，其复杂度是常数级。因为Pairing Heap只有一棵树，所以它的merge操作(类似于Fibonacci Heap中的union)也很简单，只需要link两棵树就可以了，平摊复杂度与Fibonacci Heap类似，都是常数级操作，而在Binomial Heap中需要union两个root lists，所以复杂度为O(logn)。在算法分析中，往往有很多数据结构实现起来比较简单，但是分析起来很复杂，例如**快速排序(Quicksort)**，配对堆也是一个典型例子。配对堆的merge，insert和findMin的平摊复杂度都是O(1)，extract-min的平摊复杂度是O(logn)，这与Fibonacci Heap中的相应操作的复杂度相当。但是，decrease-key的平摊复杂度比Fibonacci Heap大，后者的decrease-key的平摊复杂度是O(1)。关于配对堆的decrease-key操作的平摊复杂度结果可以参考：http://en.wikipedia.org/wiki/Pairing\_heap。

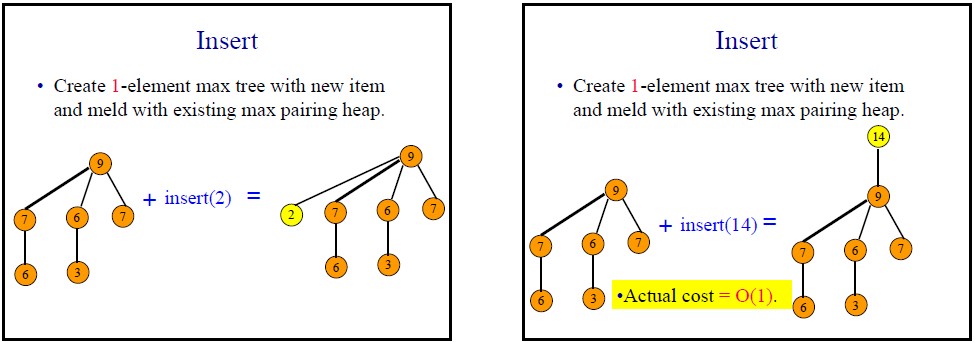
在以下实现中，Pairing Heap采用**“leftmost child, right sibling”**（左孩子，右兄弟）方式表示，而且每一个结点还有一个left属性：对于*第一个孩子，left属性表示该孩子的父结点*；对于其他结点，left属性表示该结点的左兄弟。Extract-Min操作比较有意思，首先采用类似Binomial Heap和Fibonacci Heap中做法，即先删除root结点，然后得到root的孩子结点双向链表，链表中每一个结点对应一个子堆(subheap)；接下来考虑如何将子堆合并到原来的堆中，在这里可以比较一下二项堆，Fibonacci堆和配对堆的合并做法：在Binomial Heap中将孩子结点倒排，生成按degree从小到大顺序的单向链表，然后将该单链表跟原来剩余的堆结点root list链表作union操作。在Fibonacci Heap中的做法是，将孩子结点依次添加到root list中（不用考虑先后次序），然后通过consolidate生成degree唯一的**双向循环链表**。二者都是在Extract-min时让每个堆结构变得更加紧凑，恢复成理想的状态，同时Extract-min的操作成本也相对比较高。在Pairing Heap中做法类似：如果没有Extract-min操作，其他的操作（比如insert,merge，decrease-key）势必使得root结点的孩子链表变得很长，通过Extract-Min两两合并，让Pairing Heap变得更加有序。Extract-Min两两合并做法是：先从左到右将相邻的孩子结点两两link，生成一个缩减的双向链表，然后对该新的双向链表从右到左link（上一次合并的结果作为下一次link中的右兄弟结点）。

Pairing Heap 的操作：

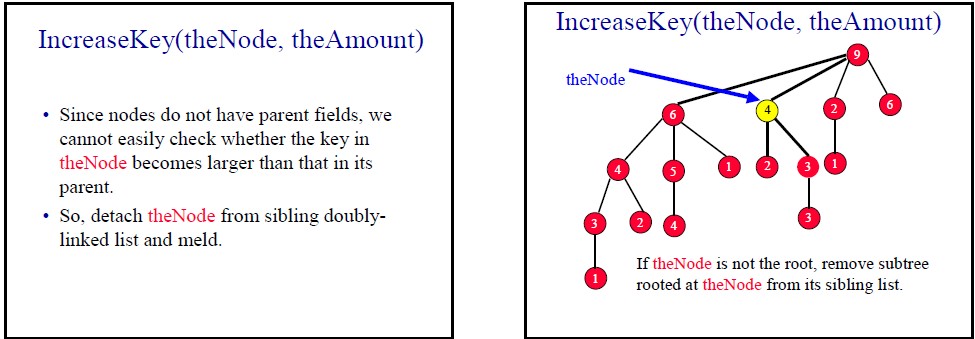
1、合并两个子堆（最大堆）

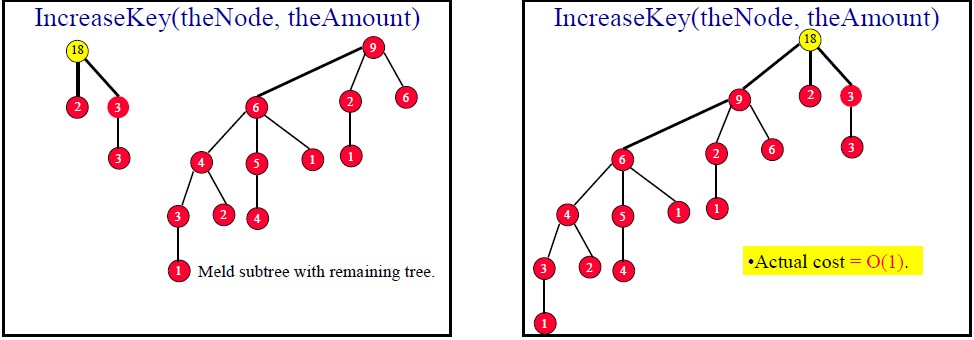


2、插入一个结点

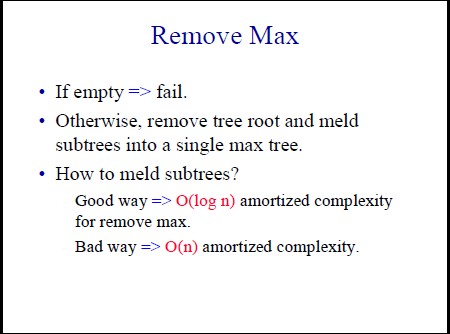


3、增加（减小）一个关键字





4、删除最小结点



**四、二叉堆**

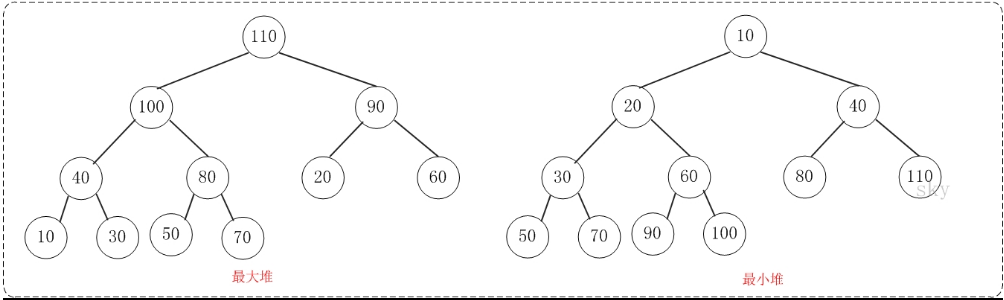
<http://www.cnblogs.com/skywang12345/p/3610390.html>

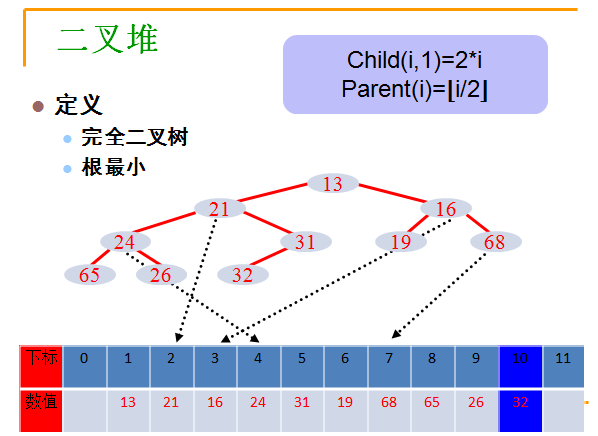
**1、介绍**

二叉堆是*完全二元树或者是近似完全二元树*，按照数据的排列方式可以分为两种：最大堆和最小堆。

* 最大堆：父结点的键值总是大于或等于任何一个子节点的键值；
* 最小堆：父结点的键值总是小于或等于任何一个子节点的键值。

二叉堆一般都通过**"数组"**来实现，下面是数组实现的最大堆和最小堆的示意图：



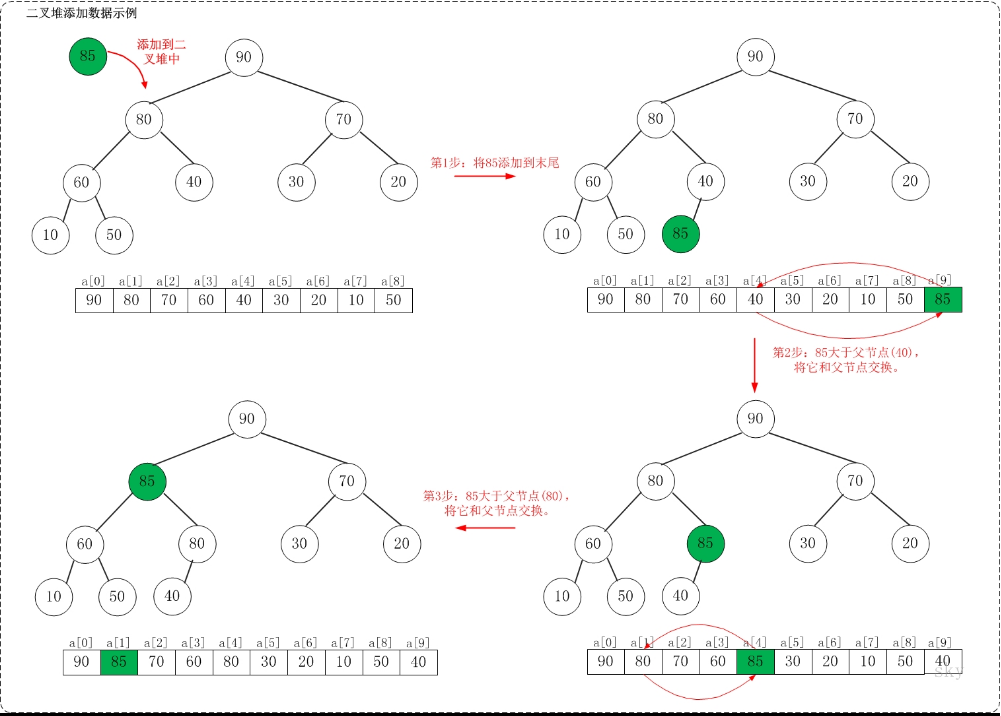


**2、基本操作**

图文解析是以"最大堆"来进行介绍的。最大堆的核心内容是**"添加"和"删除"**，理解这两个算法，二叉堆也就基本掌握了。下面对它们进行介绍，其它内容请参考后面的完整源码。

**1）添加**

假设在最大堆[90,80,70,60,40,30,20,10,50]种添加85，需要执行的步骤如下：

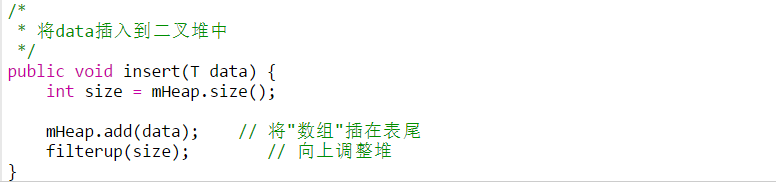


如上图所示，当向最大堆中添加数据时：先将数据加入到最大堆的最后，然后尽可能把这个元素往上挪，直到挪不动为止！将85添加到[90,80,70,60,40,30,20,10,50]中后，最大堆变成了[90,85,70,60,80,30,20,10,50,40]。

**插入操作代码：**

其中，private List<T> mHeap; // 队列(实际上是**动态数组ArrayList的实例**)



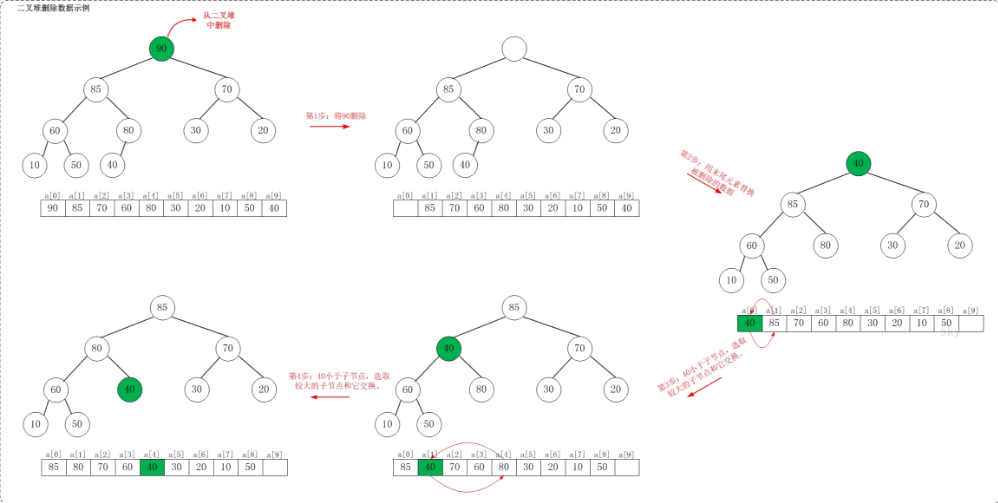


insert(data)的作用：将数据data添加到最大堆中。mHeap是动态数组ArrayList对象。

当堆已满的时候，添加失败；否则data添加到最大堆的末尾。然后通过**上调算法**重新调整数组，使之重新成为最大堆。

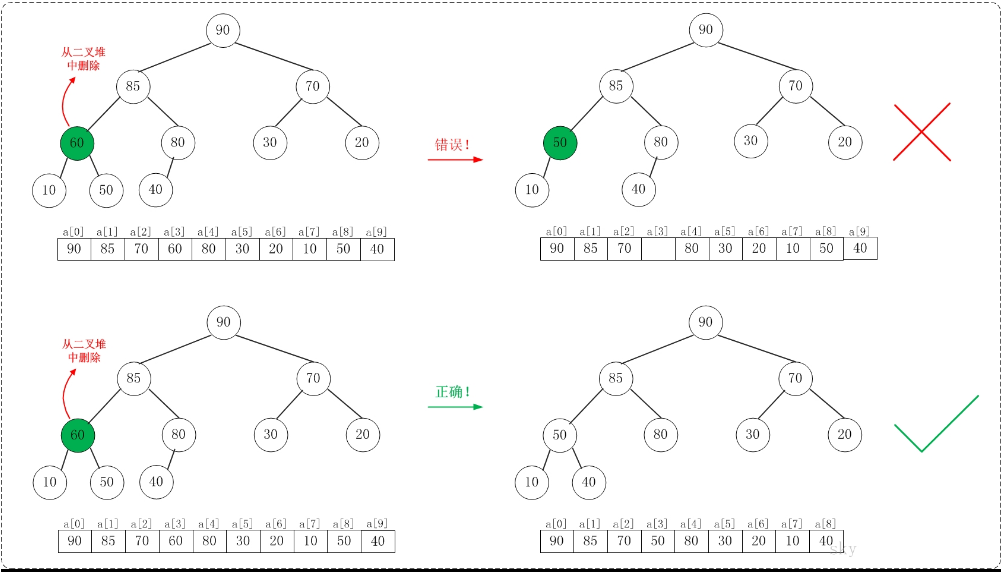
**2）删除**

假设从最大堆[90,85,70,60,80,30,20,10,50,40]中删除90，需要执行的步骤如下：

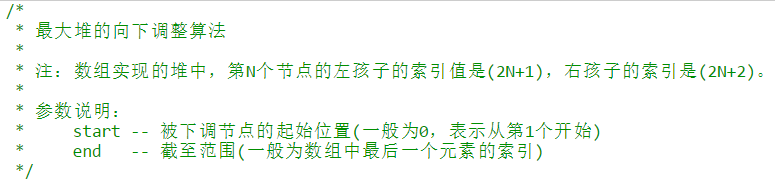


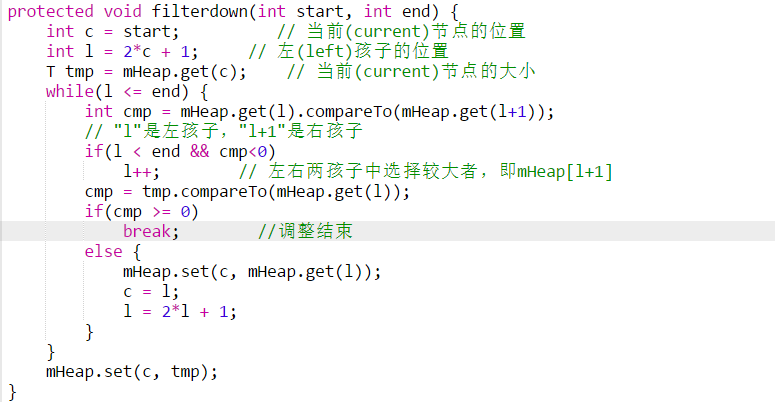
如上图所示，当从最大堆中删除数据时：*先删除该数据，然后用最大堆中最后一个的元素插入这个空位；*接着，把这个“空位”尽量往上挪，直到剩余的数据变成一个最大堆。从[90,85,70,60,80,30,20,10,50,40]删除90之后，最大堆变成了[85,80,70,60,40,30,20,10,50]。

注意：考虑从最大堆[90,85,70,60,80,30,20,10,50,40]中删除60，执行的步骤不能单纯的用它的字节点来替换；而必须考虑到"替换后的树仍然要是最大堆"！



删除代码：

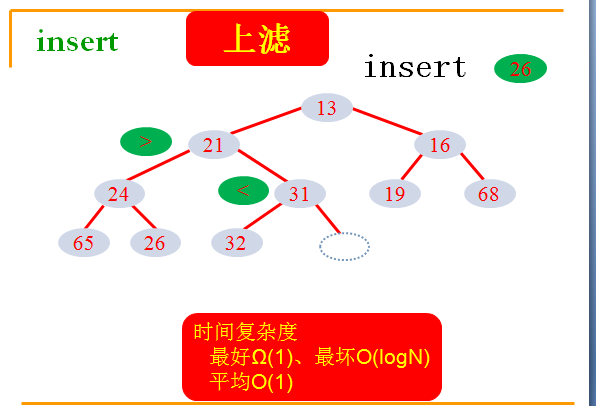




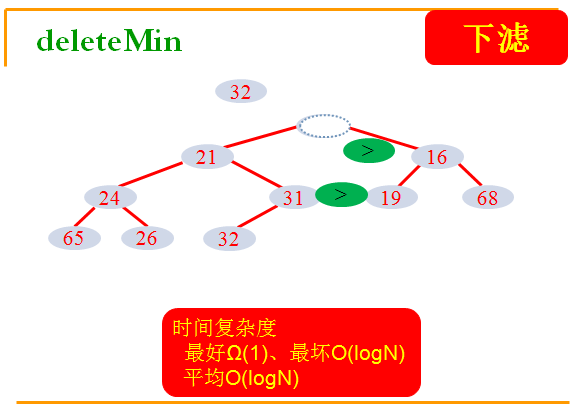


下面均是基于最小堆分析的

**insert（上滤）：**插入末尾 26，不断向上比较，大于26则交换位置，小于则停止。



**deleteMin（下滤）：**提取末尾元素，放在堆顶，不断下滤：

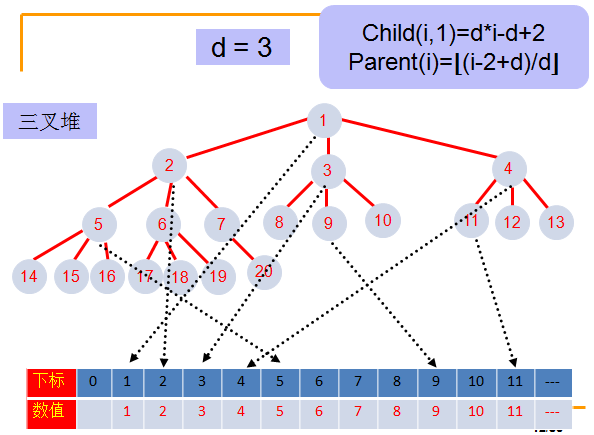


都是基于insert（上滤）与deleteMin（下滤）的操作。

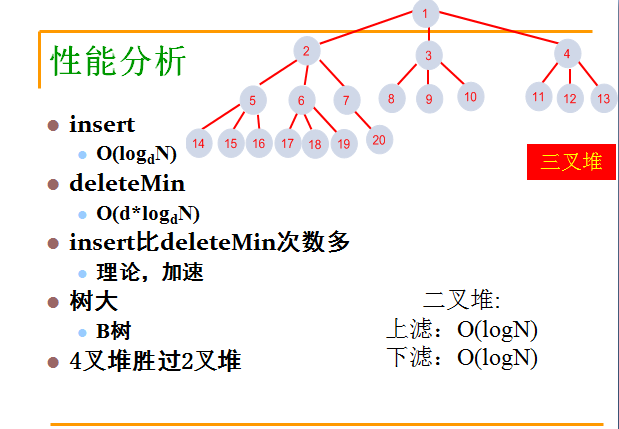
* **减小元素：**减小节点的值，上滤调整堆。
* **增大元素：**增加节点的值，下滤调整堆。
* **删除非顶点节点：**直接删除会出问题。方法：减小元素的值到无穷小，上滤后删除。
* Merge：insert one by one

**五、d 堆**

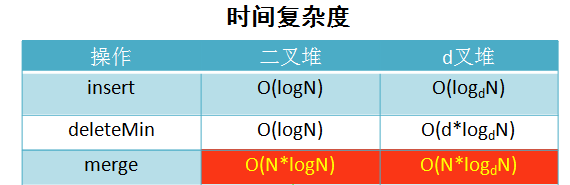
完全d叉树，根最小。存储时使用层序。



**操作：**操作跟*二叉堆基本一致*：insert,deleteMin、增大元素、减小元素、删除非顶元素、merge。



二叉堆与d叉堆的对比：



**六、左式堆（左倾堆）**

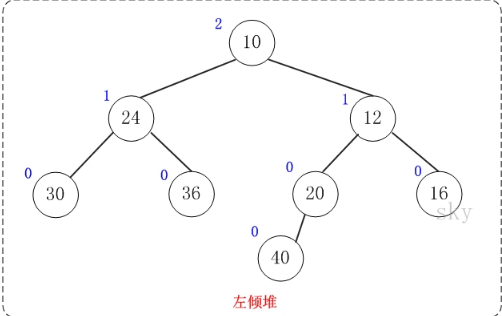
<http://www.cnblogs.com/skywang12345/p/3638384.html>

**1、介绍及定义**

左倾堆(leftist tree 或 leftist heap)，又被成为**左偏树、左偏堆，最左堆**等。

它和二叉堆一样，都是**优先队列**实现方式。当优先队列中涉及到"对两个优先队列进行合并"的问题时，二叉堆的效率就无法令人满意了，而本文介绍的左倾堆，则可以很好地解决这类问题。

**左倾堆的定义：**



上图是一颗**左倾树**，它的节点除了和二叉树的节点一样具有**左右子树指针**外，还有两个属性：**键值和零距离**。

(01) 键值的作用是来比较节点的大小，从而对节点进行排序。

(02) **零距离**(英文名NPL，即Null Path Length)则是*从一个节点到一个"最近的不满节点"的路径长度*。**不满节点**是指*该节点的左右孩子至少有有一个为NULL*。**叶节点的NPL为0，NULL节点的NPL为-1。**

左倾堆有以下几个基本性质：

**[性质1]** 节点的键值小于或等于它的左右子节点的键值（最小堆）。

**[性质2]** 节点的左孩子的NPL >= 右孩子的NPL。

**[性质3]** 节点的NPL = 它的**右孩子**的NPL + 1。

**2、图文解析**

**合并操作**是左倾堆的重点。合并两个左倾堆的基本思想如下：

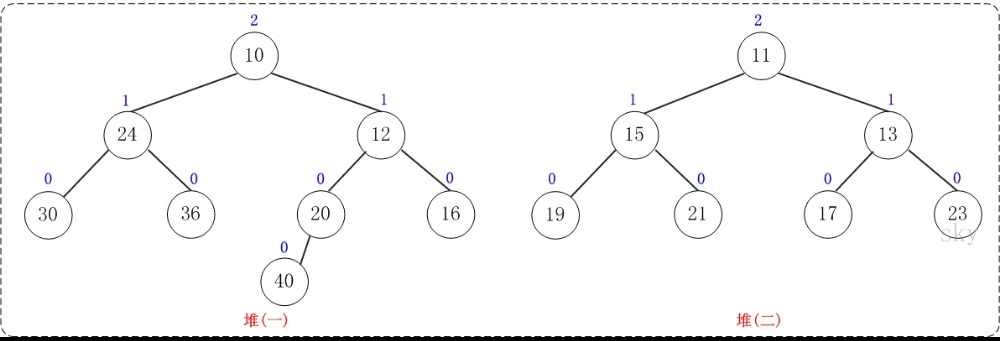
(01) 如果一个空左倾堆与一个非空左倾堆合并，返回非空左倾堆。

(02) 如果两个左倾堆都非空，那么*比较两个根节点*，取较小堆的根节点为新的根节点。将"较小堆的根节点的*右孩子"和"较大堆"进行合并*。

(03) 如果**新堆的右孩子**的NPL > 左孩子的NPL，则**交换**左右孩子。

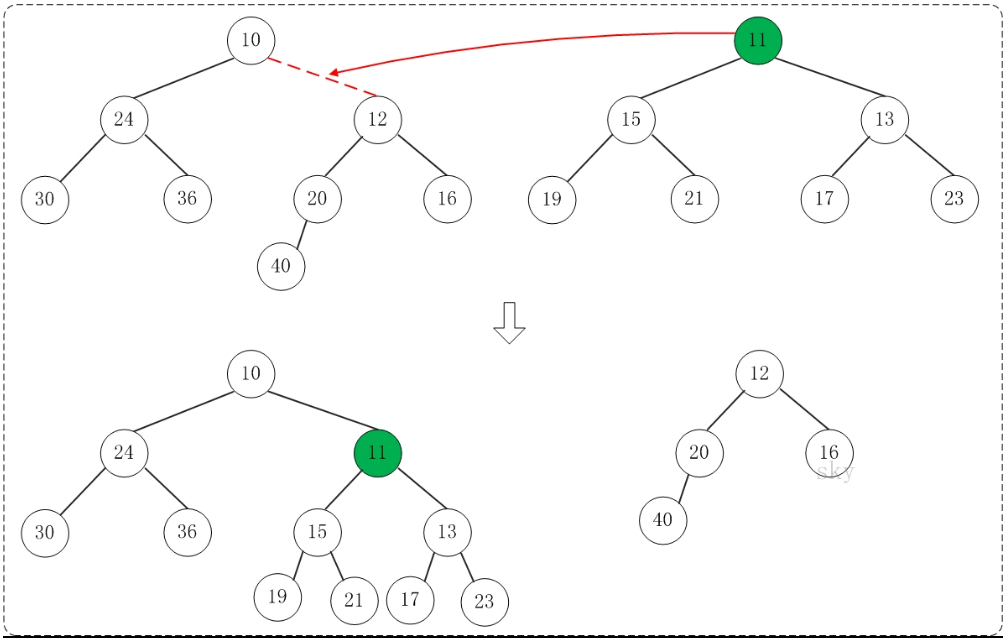
(04) 设置**新堆的根节点**的NPL = 右子堆NPL + 1

下面通过图文演示合并以下两个堆的过程：

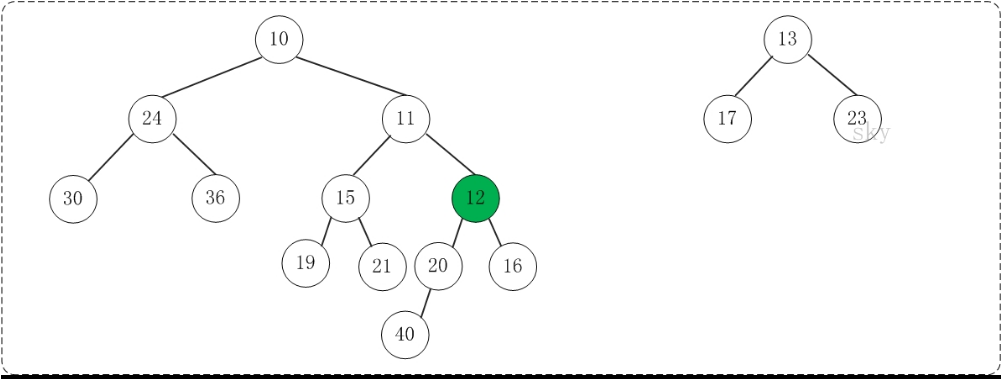


提示：这两个堆的合并过程和测试程序相对应！

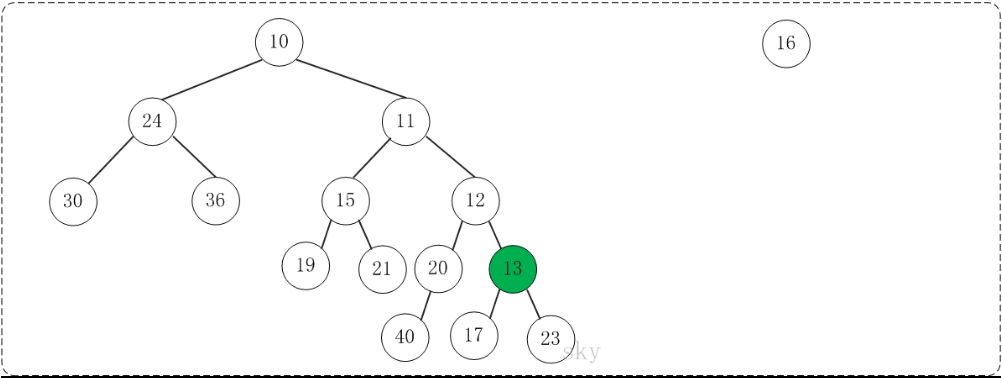
**第1步：**将"较小堆(根为10)的**右孩子"和"较大堆(根为11)"进行合并**。合并的结果，相当于将"较大堆"设置"较小堆"的**右孩子**，如下图所示：



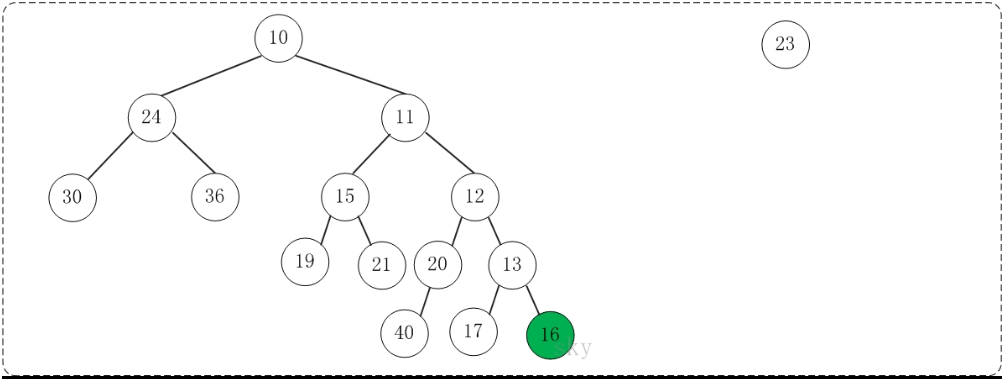
**第2步：**将上一步得到的"根11的右子树"和"根为12的树"进行合并，得到的结果如下：



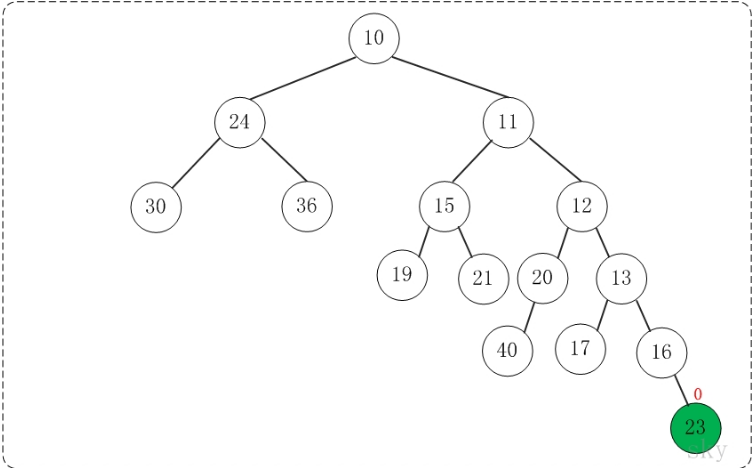
**第3步：**将上一步得到的"根12的右子树"和"根为13的树"进行合并，得到的结果如下：



**第4步：**将上一步得到的"根13的右子树"和"根为16的树"进行合并，得到的结果如下：

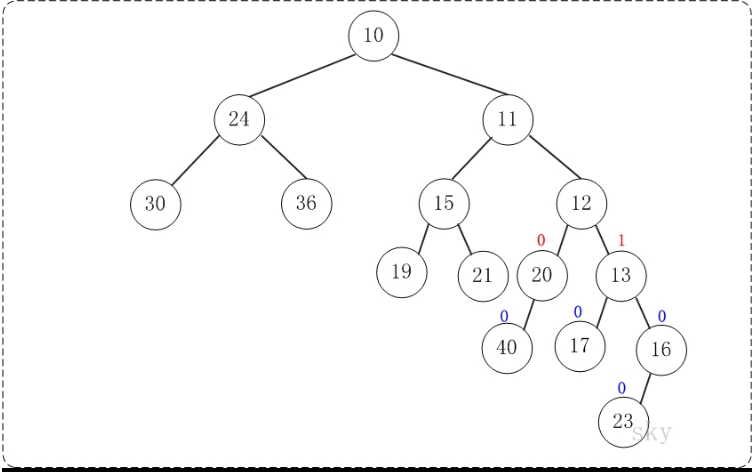


**第5步：**将上一步得到的"根16的右子树"和"根为23的树"进行合并，得到的结果如下：

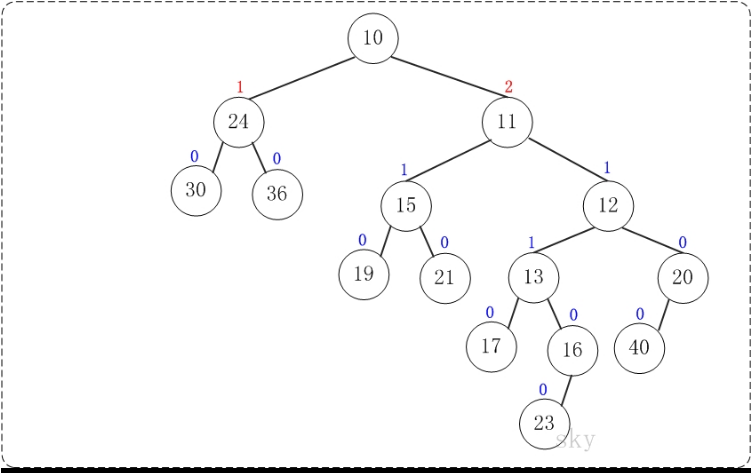


至此，已经成功的将两棵树合并成为一棵树了。接下来，对新**生成的树进行调节**。

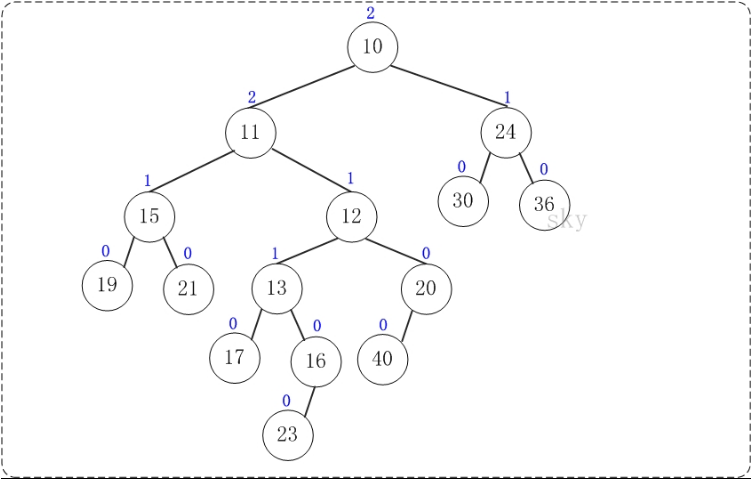
**第6步：**上一步得到的**"树16的右孩子的NPL > 左孩子的NPL"**，因此交换左右孩子。得到的结果如下：



**第7步：**上一步得到的"树12的右孩子的NPL > 左孩子的NPL"，因此交换左右孩子。得到的结果如下：



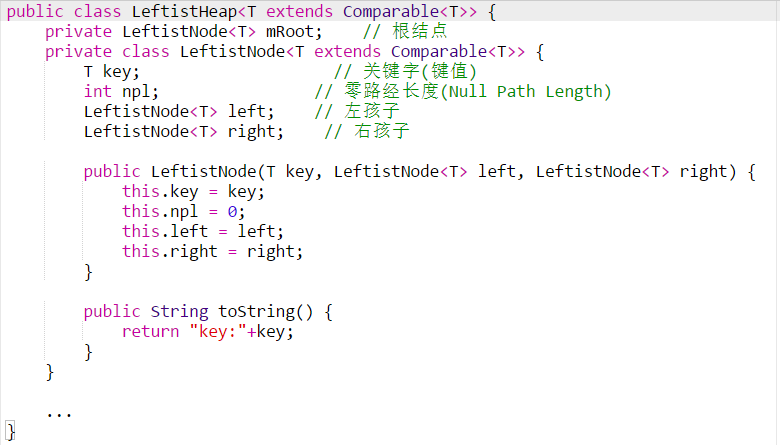
**第8步：**上一步得到的"树10的右孩子的NPL > 左孩子的NPL"，因此交换左右孩子。得到的结果如下：



至此，合并完毕。上面就是合并得到的左倾堆！

**3、基本操作**

**1）基本定义**

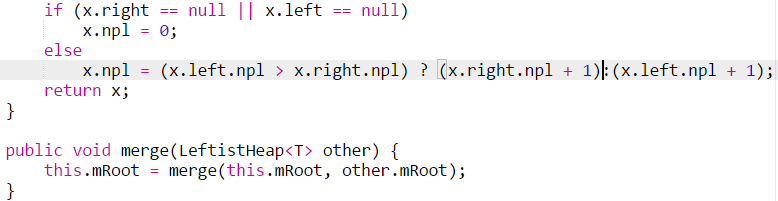


LeftistNode是左倾堆对应的节点类。

LeftistHeap是左倾堆类，它包含了左倾堆的根节点，以及左倾堆的操作

**2）合并**

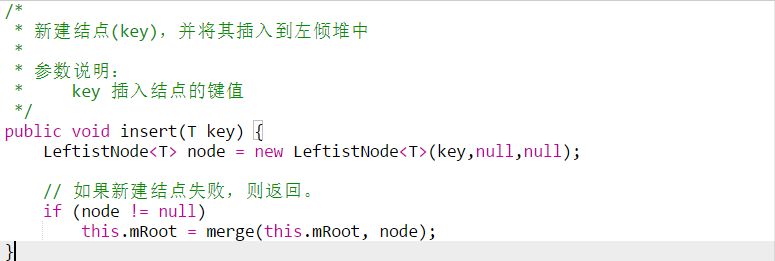




merge(x, y)是内部接口，作用是合并x和y这两个左倾堆，并返回得到的**新堆的根节点**。

merge(other)是外部接口，作用是将other合并到当前堆中。

**3）添加**

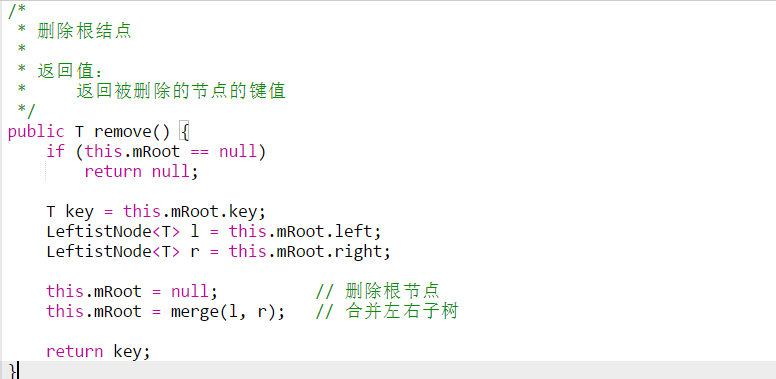


insert(key)的作用是新建键值为key的节点，并将其加入到当前左倾堆中。

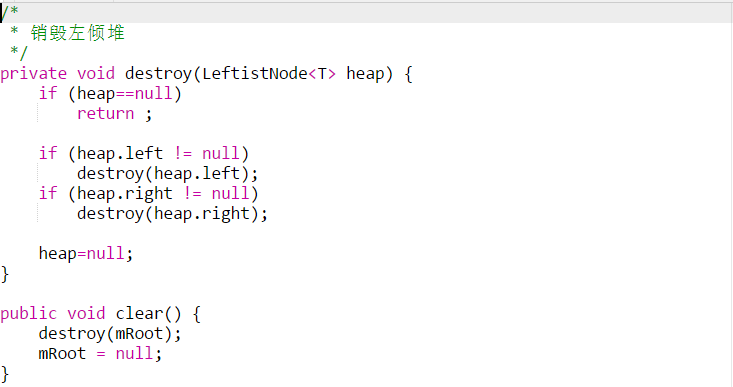
**4）删除根节点**

remove()的作用是删除左倾堆的最小节点。

注意：关于左倾堆的"前序遍历"、"中序遍历"、"后序遍历"、"打印"、"销毁"等接口就不再单独介绍了。



**5）销毁左倾堆**



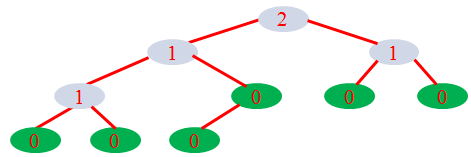
**4、补充**

**零路径长度：**到没有两个儿子的节点最短距离

左式堆：

* 一棵二叉树
* 零路径长：左儿子≧右儿子，父节点= min{儿子} +1（这条性质导致了左式堆的严重左偏）

**零路径长度NPL：**



**七、斜堆**

<http://www.cnblogs.com/skywang12345/p/3638552.html>

**1、介绍**

**斜堆(Skew heap)也叫自适应堆(self-adjusting heap)**，它是左倾堆的一个变种。和左倾堆一样，它通常也用于实现优先队列；作为一种自适应的左倾堆，它的合并操作的时间复杂度也是O(lg n)。

它与左倾堆的**差别是**：

(01) 斜堆的节点没有"零距离"这个属性，而左倾堆则有。

(02) 斜堆的合并操作和左倾堆的合并操作算法不同。

**斜堆的合并操作：**

(01) 如果一个空斜堆与一个非空斜堆合并，返回非空斜堆。

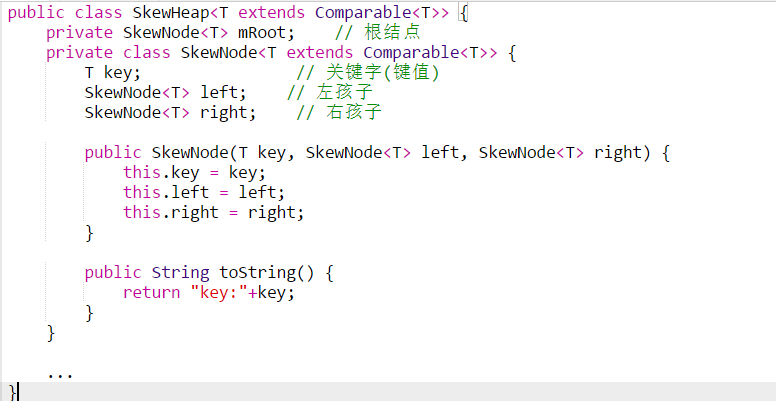
(02) 如果两个斜堆都非空，那么比较**两个根节点**，取较小堆的根节点为新的根节点。将"较小堆的根节点的**右孩子"和"较大堆"进行合并**。

(03) 合并后，交换新堆根节点的左孩子和右孩子。

第(03)步是斜堆和左倾堆的合并操作差别的关键所在，如果是**左倾堆**，则*合并后要比较左右孩子的零距离大小，若右孩子的零距离 > 左孩子的零距离，则交换左右孩子*；最后，在设置根的零距离。

**2、基本操作**

**1）基本定义**



SkewNode是斜堆对应的节点类。

SkewHeap是斜堆类，它包含了斜堆的根节点，以及斜堆的操作。

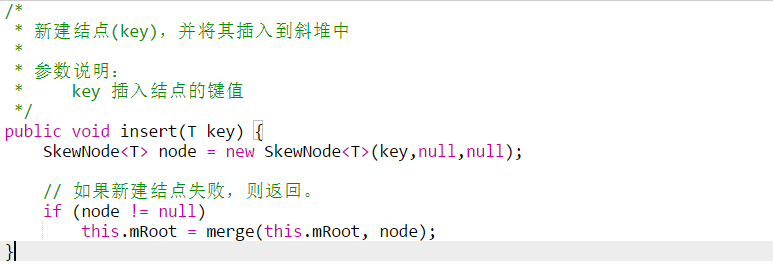
**2）合并**



merge(x, y)是内部接口，作用是合并x和y这两个斜堆，并返回得到的新堆的根节点。

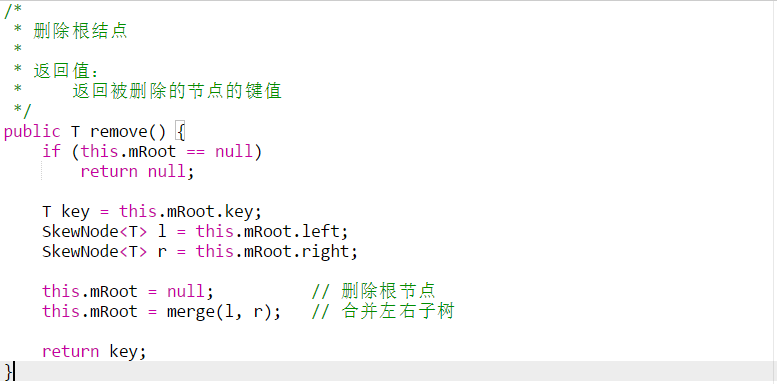
merge(other)是外部接口，作用是将other合并到当前堆中。

**3）添加**

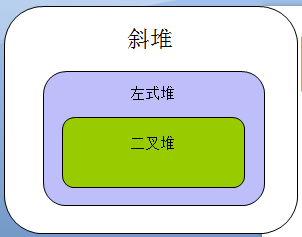


insert(key)的作用是新建键值为key的节点，并将其加入到当前斜堆中。

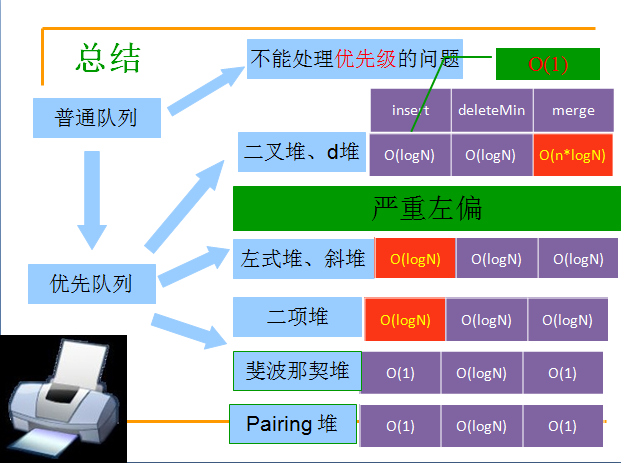
**4）删除**



remove()的作用是删除斜堆的最小节点。







**八、优先队列PriorityQueue**

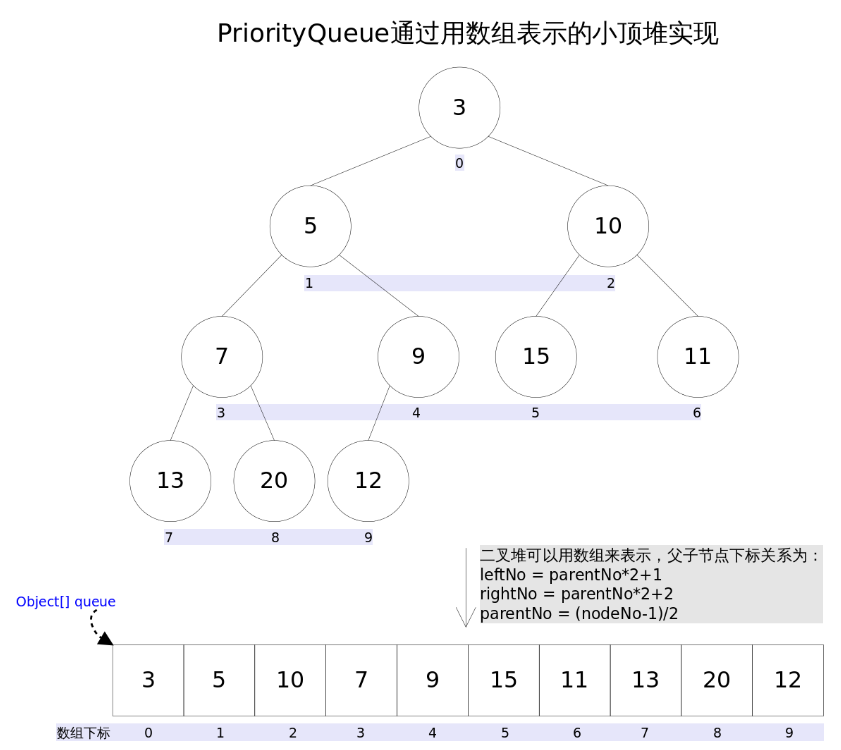
<http://www.cnblogs.com/CarpenterLee/p/5488070.html>

Java中**PriorityQueue**通过二叉小顶堆实现，可以用一棵***完全二叉树***表示。本文从Queue接口函数出发，结合生动的图解，深入浅出地分析PriorityQueue每个操作的具体过程和时间复杂度，将让读者建立对PriorityQueue建立清晰而深入的认识。

**1、总体介绍**

**PriorityQueue优先队列**的作用是能*保证每次取出的元素都是队列中权值最小的*（Java的优先队列每次取最小元素，C++的优先队列每次取最大元素）。这里牵涉到了大小关系，元素大小的评判可以通过元素本身的**自然顺序**（natural ordering），也可以通过构造时传入的比较器（Comparator，类似于C++的仿函数）。

Java中PriorityQueue实现了Queue接口，**不允许放入null元素**；其通过堆实现，具体说是通过完全二叉树（complete binary tree）实现的小顶堆（任意一个非叶子节点的权值，都不大于其左右子节点的权值），也就意味着***可以通过数组来作为PriorityQueue的底层实现。***



上图中我们给每个元素按照层序遍历的方式进行了编号，如果你足够细心，会发现父节点和子节点的编号是有联系的，更确切的说父子节点的编号之间有如下关系：

**leftNo = parentNo\*2+1**

**rightNo = parentNo\*2+2**

**parentNo = (nodeNo-1)/2**

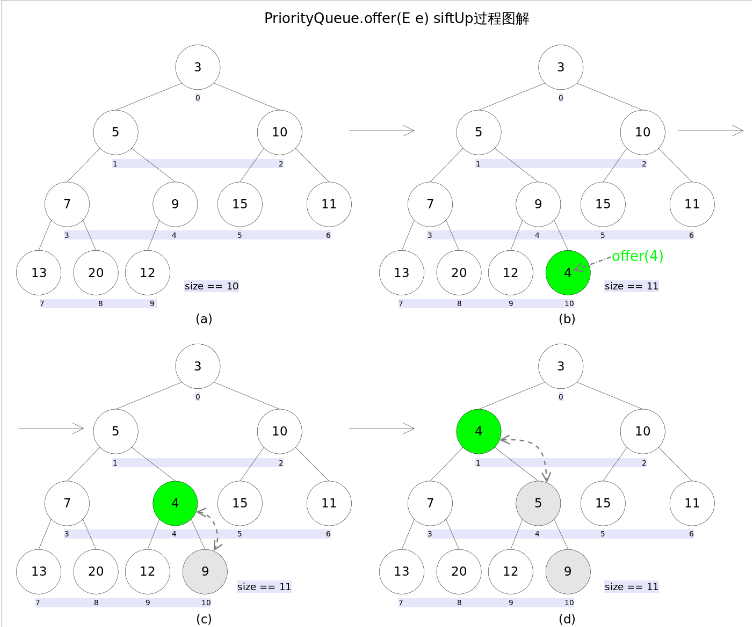
通过上述三个公式，可以轻易计算出某个节点的父节点以及子节点的下标。这也就是为什么可以直接用数组来存储堆的原因。

PriorityQueue的peek()和element操作是常数时间，add(), offer(), 无参数的remove()以及poll()方法的时间复杂度都是**log(N)**。

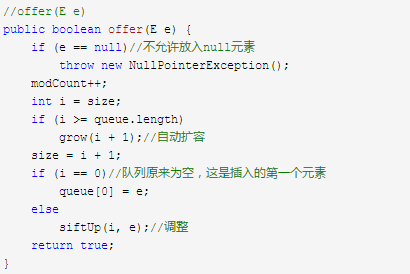
**2、方法解析**

**1）add()和offer()**

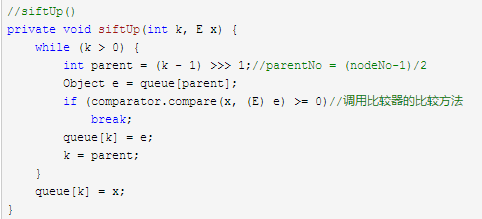
add(E e)和offer(E e)的语义相同，都是向优先队列中插入元素，只是Queue接口规定二者对插入失败时的处理不同，***前者在插入失败时抛出异常，后则则会返回false***。对于PriorityQueue这两个方法其实没什么差别。



新加入的元素可能会破坏小顶堆的性质，因此需要进行必要的调整。



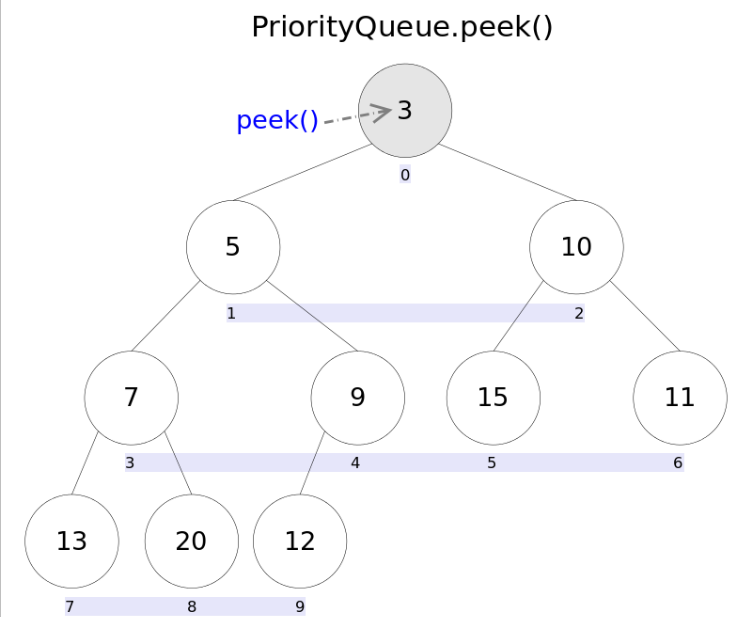
上述代码中，扩容函数grow()类似于ArrayList里的grow()函数，就是再申请一个更大的数组，并将原数组的元素复制过去，这里不再赘述。需要注意的是siftUp(int k, E x)方法，该方法用于插入元素x并维持堆的特性。

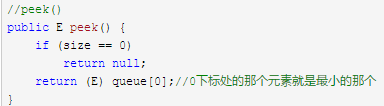


新加入的元素x可能会破坏小顶堆的性质，因此需要进行调整。调整的过程为：从k指定的位置开始，将x逐层与当前点的parent进行比较并交换，直到满足x >= queue[parent]为止。注意这里的比较可以是元素的自然顺序，也可以是依靠比较器的顺序。

**2）element()和peek()**

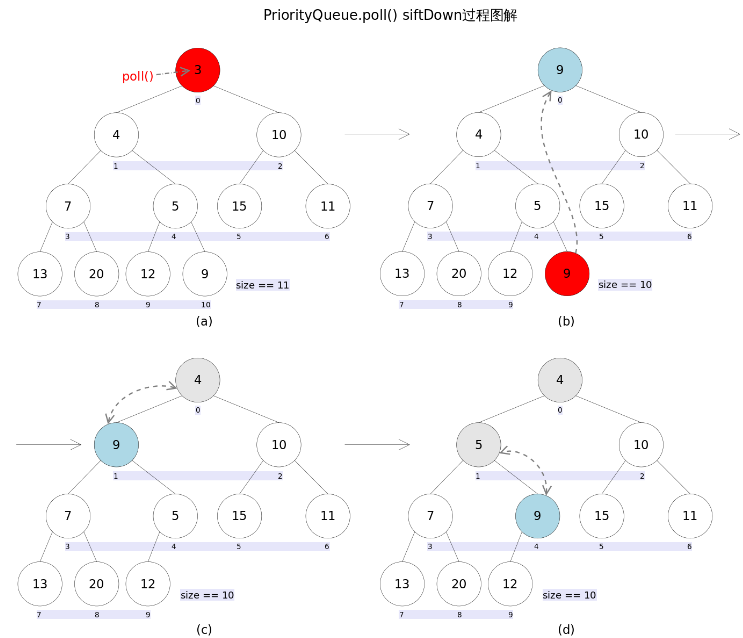
element()和peek()的语义完全相同，都是获取但不删除队首元素，也就是队列中权值最小的那个元素，二者唯一的区别是当方法失败时，**前者抛出异常，后者返回null**。根据小顶堆的性质，堆顶那个元素就是全局最小的那个；由于堆用数组表示，根据下标关系，0下标处的那个元素既是堆顶元素。所以直接返回数组0下标处的那个元素即可。

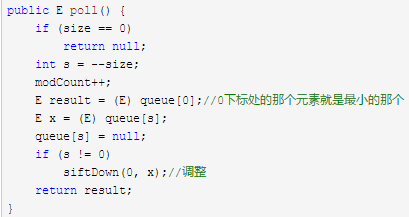




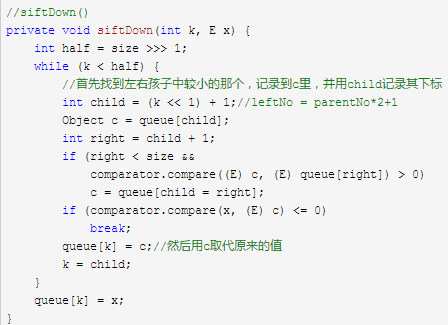
**3）remove()和poll()**

remove()和poll()方法的语义也完全相同，都是**获取并删除队首元素**，区别是当方法失败时前者抛出异常，后者返回null。由于删除操作会改变队列的结构，为维护小顶堆的性质，需要进行必要的调整。





上述代码首先记录0下标处的元素，并用最后一个元素替换0下标位置的元素，之后调用**siftDown()方法对堆进行调整**，最后返回原来0下标处的那个元素（也就是最小的那个元素）。重点是siftDown(int k, E x)方法，该方法的作用是从k指定的位置开始，将x逐层向下与当前点的左右孩子中较小的那个交换，直到x小于或等于左右孩子中的任何一个为止。



**4）remove(Object o)**

remove(Object o)方法用于*删除队列中跟o相等的某一个元素（如果有多个相等，只删除一个）*，该方法不是Queue接口内的方法，而是Collection接口的方法。由于删除操作会改变队列结构，所以要进行调整；又由于删除元素的位置可能是任意的，所以调整过程比其它函数稍加繁琐。具体来说，remove(Object o)可以分为2种情况：1. 删除的是最后一个元素。直接删除即可，不需要调整。2. 删除的不是最后一个元素，从删除点开始以最后一个元素为参照调用一次siftDown()即可。此处不再赘述。

