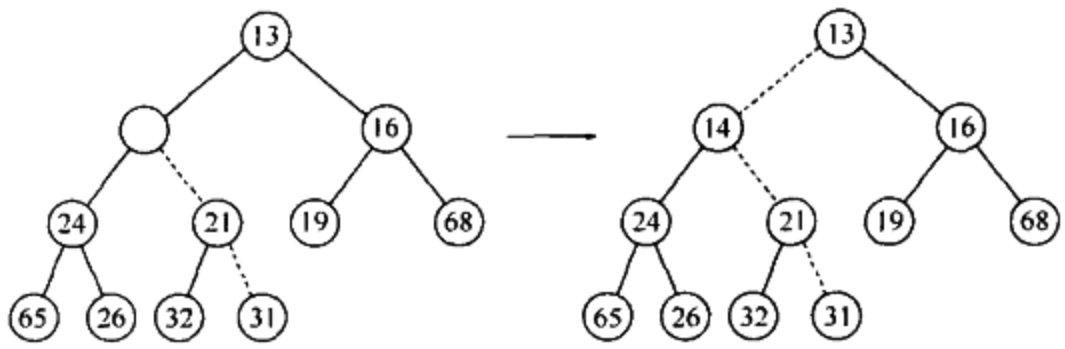

图 6-7 将 14 插入到前面的堆中的其余两步

```
1       /**
2        * Insert into the priority queue, maintaining heap order.
3        * Duplicates are allowed.
4        * @param x the item to insert.
5        */
6       public void insert( AnyType x )
7       {
8           if( currentSize == array.length - 1 )
9               enlargeArray( array.length * 2 + 1 );
10
11              // Percolate up
12          int hole = ++currentSize;
13          for( ; hole > 1 && x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
14              array[ hole ] = array[ hole / 2 ];
15          array[ hole ] = x;
16      }
```

图 6-8 插入到一个二叉堆的过程

其实我们本可以使用 insert 例程通过反复执行交换操作直至建立正确的序来实现上滤过程，可是一次交换需要 3 条赋值语句。如果一个元素上滤 d 层，那么由于交换而执行的赋值次数就达到 $3d$，而我们这里的方法却只用到 $d+1$ 次赋值。

如果要插入的元素是新的最小值，那么它将一直被推向顶端。这样在某一时刻 hole 将是 1，并且需要程序跳出循环。当然我们可以用显式的测试做到这一点，或者把对被插入项的引用放到位置 0 处使循环终止。我们选择显式的方式来完成插入的实现。

如果欲插入的元素是新的最小元从而一直上滤到根处，那么这种插入的时间将长达 $O(\log N)$。平均看来，上滤终止得要早；业已证明，执行一次插入平均需要 2.607 次比较，因此平均 insert 操作上移元素 1.607 层。

deleteMin(删除最小元)

deleteMin 以类似于插入的方式处理。找出最小元是容易的，困难之处是删除它。当删除一个最小元时，要在根节点建立一个空穴。由于现在堆少了一个元素，因此堆中最后一个元素 X 必须移动到该堆的某个地方。如果 X 可以被放到空穴中，那么 deleteMin 完成。不过这一般不太可能，因此我们将空穴的两个儿子中较小者移入空穴，这样就把空穴向下推了一层。重复该步骤直到 X 可以被放入空穴中。因此，我们的做法是将 X 置入沿着从根开始包含最小儿子的一条路径上的一个正确的位置。

图 6-9 中左图显示了 deleteMin 之前的堆。删除 13 后，我们必须试图正确地将 31 放到堆中。31 不能放在空穴中，因为这将破坏堆序性质。于是，我们把较小的儿子 14 置入空穴，同时空穴