两个 merge 例程(图 6-26)被设计成消除一些特殊情形并保证 H_1 有较小根的驱动程序。实际的合并操作在 mergel 中进行(图 6-27)。公有的 merge 方法将 rhs 合并到控制堆中。rhs 变成了空的。在这个公有方法中的别名测试不接受 h.merge(h)。

执行合并的时间与诸右路径的长的和成正比，因为在递归调用期间对每一个被访问的节点花费的是常数工作量。因此，我们得到合并两个左式堆的时间界为 $O(\log N)$。也可以分两趟来非递归地执行该操作。在第一趟，我们通过合并两个堆的右路径建立一棵新的树。为此，以排序的方式安排 H_1 和 H_2 右路径上的节点，保持它们各自的左儿子不变。在我们的例子中，新的右路径是 3，6，7，8，18，而最后得到的树如图 6-28 所示。第二趟构成堆，儿子的交换工作在左式堆性质被破坏的那些节点上进行。在图 6-28 中，在节点 7 和 3 各有一次交换，并得到与前面相同的树。非递归的做法更容易理解，但编程困难。我们留给读者去证明：递归过程和非递归过程的结果是相同的。

```
1       /**
2        * Merge rhs into the priority queue.
3        * rhs becomes empty. rhs must be different from this.
4        * @param rhs the other leftist heap.
5        */
6       public void merge( LeftistHeap<AnyType> rhs )
7       {
8           if( this == rhs )    // Avoid aliasing problems
9               return;
10
11          root = merge( root, rhs.root );
12          rhs.root = null;
13      }
14
15      /**
16       * Internal method to merge two roots.
17       * Deals with deviant cases and calls recursive mergel.
18       */
19      private Node<AnyType> merge( Node<AnyType> h1, Node<AnyType> h2 )
20      {
21          if( h1 == null )
22              return h2;
23          if( h2 == null )
24              return h1;
25          if( h1.element.compareTo( h2.element ) < 0 )
26              return mergel( h1, h2 );
27          else
28              return mergel( h2, h1 );
29      }
```

图 6-26　合并左式堆的驱动例程

上面提到，我们可以通过把被插入项看成单节点堆并执行一次 merge 来完成插入。为了执行 deleteMin，我们只要除掉根而得到两个堆，然后再将这两个堆合并即可。因此，执行一次 deleteMin 的时间为 $O(\log N)$。这两个例程在图 6-29 和图 6-30 中给出。