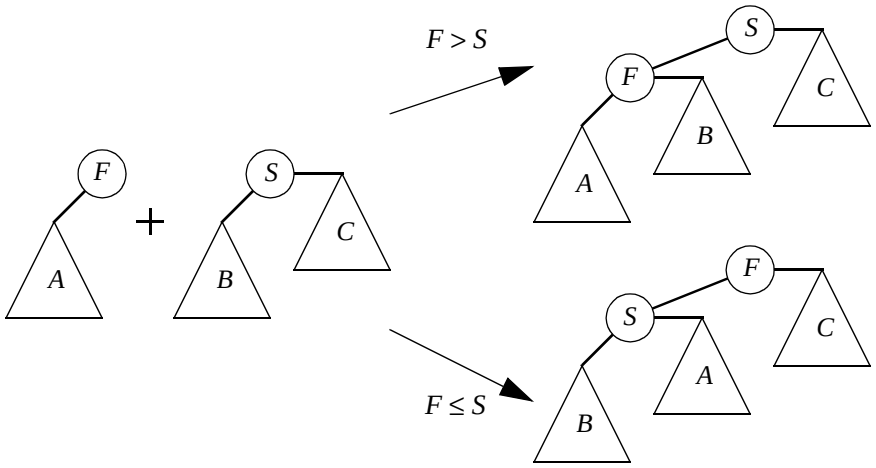

Figure 22.14 `compareAndLink` merges two trees

The two remaining routines are `compareAndLink`, which combines two trees, and `combineSiblings`, which combines all the siblings, when given the first sibling. Figure 22.14 shows how two subheaps are combined. The procedure is generalized to allow the second subheap to have siblings (this is needed for the second pass in the two-pass merge). As mentioned earlier in the chapter, the subheap with the larger root is made a leftmost child of the other subheap. The code is shown in Figure 22.15. Notice that there are several instances in which a reference is tested against `null` before it accesses its `prev` data field. This suggests that perhaps it would be useful to have a `nullNode` sentinel, as was customary in the advanced search tree implementations. This is left as Exercise 22.13.

Finally, Figure 22.16 (page 658) implements `combineSiblings`. We use the array `treeArray` to store the subtrees. In the worst case, we could have $N - 1$ siblings, so the array uses 10,000 as the capacity, for simplicity. We begin by separating the subtrees and storing them in `treeArray`, using the loop at lines 16 to 23. Assuming we have more than one sibling to merge, we make a left-to-right pass at lines 27 to 29. The special case of an odd number of trees is handled at lines 33 to 36. We finish the merging with a right-to-left pass at lines 40 to 42. Once we are done, the result is found in array position 0 and can be returned.

```
1     /**
2      * Internal method that is the basic operation to
3      * maintain order.
4      * Links first and second together to satisfy heap order.
5      * @param first root of tree 1, which may not be null.
6      *    first.nextSibling MUST be null on entry.
7      * @param second root of tree 2, which may be null.
8      * @return result of the tree merge.
9      */
10    private PairNode
11    compareAndLink( PairNode first, PairNode second )
12    {
13        if( second == null )
14            return first;
15
16        if( second.element.lessThan( first.element ) )
17        {
18            // Attach first as leftmost child of second
19            second.prev = first.prev;
20            first.prev = second;
21            first.nextSibling = second.leftChild;
22            if( first.nextSibling != null )
23                first.nextSibling.prev = first;
24            second.leftChild = first;
25            return second;
26        }
27        else
28        {
29            // Attach second as leftmost child of first
30            second.prev = first;
31            first.nextSibling = second.nextSibling;
32            if( first.nextSibling != null )
33                first.nextSibling.prev = first;
34            second.nextSibling = first.leftChild;
35            if( second.nextSibling != null )
36                second.nextSibling.prev = second;
37            first.leftChild = second;
38            return first;
39        }
40    }
```

Figure 22.15 `compareAndLink` routine

As a practical matter, placing an arbitrary limit on the number of children of the root is unjustified. It would be better to use a respectable initial size and expand the array as needed. The online code has a more flexible solution.

```
1     /**
2      * Internal method that implements two-pass merging.
3      * @param firstSibling the root of the conglomerate;
4      *     assumed not null.
5      */
6
7       // Assumes at most 10000 children; online code is better.
8     static PairNode [ ] treeArray = new PairNode[ 10000 ];
9
10    private PairNode combineSiblings( PairNode firstSibling )
11    {
12        if( firstSibling.nextSibling == null )
13            return firstSibling;
14
15            // Store the subtrees in an array
16        int numSiblings = 0;
17        for( ; firstSibling != null; numSiblings++ )
18        {
19            treeArray[ numSiblings ] = firstSibling;
20            firstSibling.prev.nextSibling = null;
21            firstSibling = firstSibling.nextSibling;
22        }
23        treeArray[ numSiblings ] = null;
24
25        // Combine subtrees two at a time, going left to right
26        int i = 0;
27        for( ; i + 1 < numSiblings; i += 2 )
28            treeArray[ i ] = compareAndLink( treeArray[ i ],
29                                             treeArray[ i + 1 ] );
30
31            // j has the result of last compareAndLink.
32            // If an odd number of trees, get the last one.
33        int j = i - 2;
34        if( j == numSiblings - 3 )
35            treeArray[ j ] = compareAndLink( treeArray[ j ],
36                                             treeArray[ j + 2 ] );
37
38            // Now go right to left, merging last tree with
39            // next to last. The result becomes the new last.
40        for( ; j >= 2; j -= 2 )
41             treeArray[ j - 2 ] = compareAndLink(
42                         treeArray[ j - 2 ], treeArray[ j ] );
43
44        return treeArray[ 0 ];
45    }
```

Figure 22.16 The heart of the pairing heap algorithm; implementing a two-pass merge to combine all of the siblings, when the first sibling is given