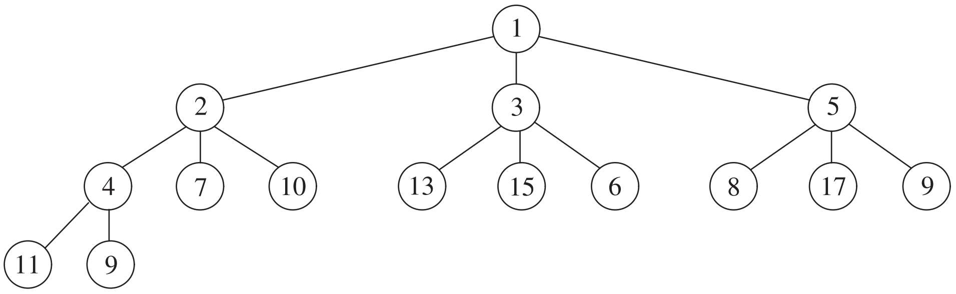

Figure 6.19 A *d*-heap

is known as a `merge`. There are quite a few ways of implementing heaps so that the running time of a `merge` is $O(\log N)$. We will now discuss three data structures, of various complexity, that support the `merge` operation efficiently. We will defer any complicated analysis until Chapter 11.

6.6 Leftist Heaps

It seems difficult to design a data structure that efficiently supports merging (that is, processes a `merge` in $o(N)$ time) and uses only an array, as in a binary heap. The reason for this is that merging would seem to require copying one array into another, which would take $\Theta(N)$ time for equal-sized heaps. For this reason, all the advanced data structures that support efficient merging require the use of a linked data structure. In practice, we can expect that this will make all the other operations slower.

Like a binary heap, a **leftist heap** has both a structural property and an ordering property. Indeed, a leftist heap, like virtually all heaps used, has the same heap-order property we have already seen. Furthermore, a leftist heap is also a binary tree. The only difference between a leftist heap and a binary heap is that leftist heaps are not perfectly balanced but actually attempt to be very unbalanced.

6.6.1 Leftist Heap Property

We define the **null path length**, *npl(X)*, of any node *X* to be the length of the shortest path from *X* to a node without two children. Thus, the *npl* of a node with zero or one child is 0, while *npl*(`null`) $= -1$. In the tree in Figure 6.20, the null path lengths are indicated inside the tree nodes.

Notice that the null path length of any node is 1 more than the minimum of the null path lengths of its children. This applies to nodes with less than two children because the null path length of `null` is -1.

The leftist heap property is that for every node *X* in the heap, the null path length of the left child is at least as large as that of the right child. This property is satisfied by only one of the trees in Figure 6.20, namely, the tree on the left. This property actually goes

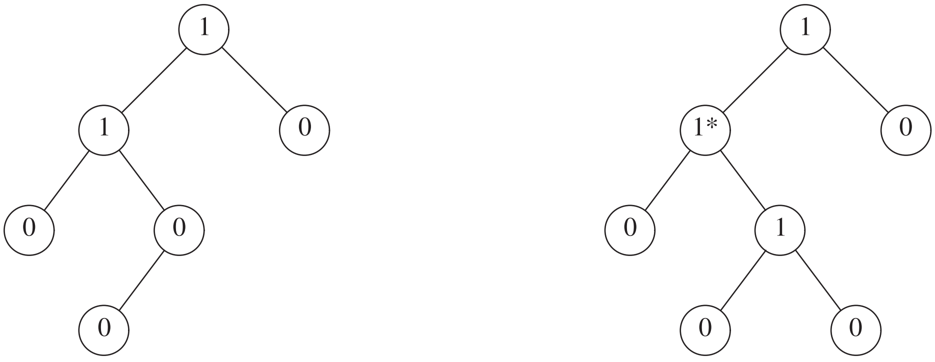

Figure 6.20 Null path lengths for two trees; only the left tree is leftist

out of its way to ensure that the tree is unbalanced, because it clearly biases the tree to get deep toward the left. Indeed, a tree consisting of a long path of left nodes is possible (and actually preferable to facilitate merging)—hence the name *leftist heap*.

Because leftist heaps tend to have deep left paths, it follows that the right path ought to be short. Indeed, the right path down a leftist heap is as short as any in the heap. Otherwise, there would be a path that goes through some node X and takes the left child. Then X would violate the leftist property.

Theorem 6.2.
A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

Proof.
The proof is by induction. If $r = 1$, there must be at least one tree node. Otherwise, suppose that the theorem is true for $1, 2, \ldots, r$. Consider a leftist tree with $r+1$ nodes on the right path. Then the root has a right subtree with r nodes on the right path, and a left subtree with at least r nodes on the right path (otherwise it would not be leftist). Applying the inductive hypothesis to these subtrees yields a minimum of $2^r - 1$ nodes in each subtree. This plus the root gives at least $2^{r+1} - 1$ nodes in the tree, proving the theorem.

From this theorem, it follows immediately that a leftist tree of N nodes has a right path containing at most $\lfloor \log(N + 1) \rfloor$ nodes. The general idea for the leftist heap operations is to perform all the work on the right path, which is guaranteed to be short. The only tricky part is that performing `insert`s and `merge`s on the right path could destroy the leftist heap property. It turns out to be extremely easy to restore the property.

6.6.2 Leftist Heap Operations

The fundamental operation on leftist heaps is merging. Notice that insertion is merely a special case of merging, since we may view an insertion as a `merge` of a one-node heap with a larger heap. We will first give a simple recursive solution and then show how this might

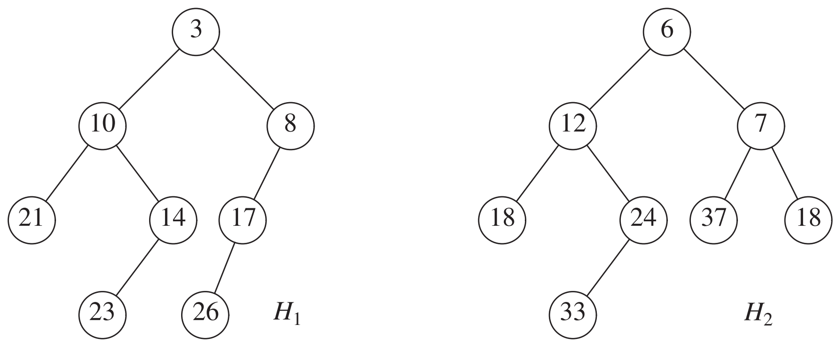

Figure 6.21 Two leftist heaps H_1 and H_2

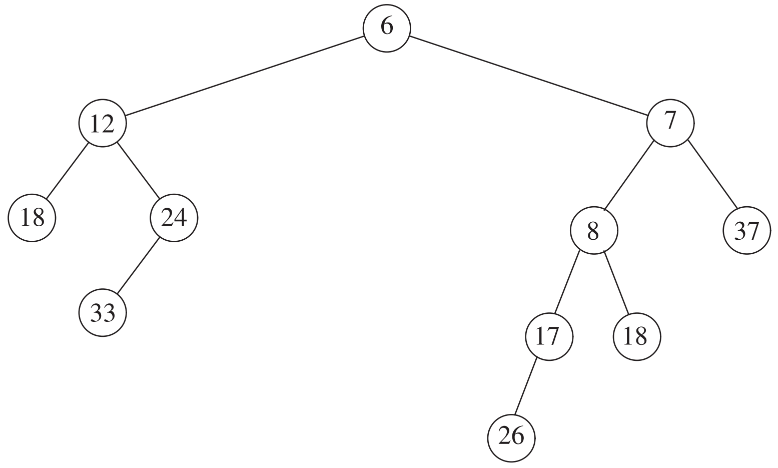

Figure 6.22 Result of merging H_2 with H_1's right subheap

be done nonrecursively. Our input is the two leftist heaps, H_1 and H_2, in Figure 6.21. You should check that these heaps really are leftist. Notice that the smallest elements are at the roots. In addition to space for the data and left and right references, each node will have an entry that indicates the null path length.

If either of the two heaps is empty, then we can return the other heap. Otherwise, to merge the two heaps, we compare their roots. First, we recursively merge the heap with the larger root with the right subheap of the heap with the smaller root. In our example, this means we recursively merge H_2 with the subheap of H_1 rooted at 8, obtaining the heap in Figure 6.22.

Since this tree is formed recursively, and we have not yet finished the description of the algorithm, we cannot at this point show how this heap was obtained. However, it is reasonable to assume that the resulting tree is a leftist heap, because it was obtained via a recursive step. This is much like the inductive hypothesis in a proof by induction. Since we can handle the base case (which occurs when one tree is empty), we can assume that the recursive step works as long as we can finish the merge; this is rule 3 of recursion, which we discussed in Chapter 1. We now make this new heap the right child of the root of H_1 (see Figure 6.23).

Although the resulting heap satisfies the heap-order property, it is not leftist because the left subtree of the root has a null path length of 1 whereas the right subtree has a null path length of 2. Thus, the leftist property is violated at the root. However, it is easy to see that the remainder of the tree must be leftist. The right subtree of the root is leftist, because of the recursive step. The left subtree of the root has not been changed, so it too must still be leftist. Thus, we need only to fix the root. We can make the entire tree leftist by merely swapping the root's left and right children (Figure 6.24) and updating the null path length—the new null path length is 1 plus the null path length of the new right child—completing the `merge`. Notice that if the null path length is not updated, then all null path lengths will be 0, and the heap will not be leftist but merely random. In this case, the algorithm will work, but the time bound we will claim will no longer be valid.

The description of the algorithm translates directly into code. The node class (Figure 6.25) is the same as the binary tree, except that it is augmented with the `npl` (null path length) field. The leftist heap stores a reference to the root as its data member. We have seen in Chapter 4 that when an element is inserted into an empty binary tree, the node referenced by the root will need to change. We use the usual technique of implementing `private` recursive methods to do the merging. The class skeleton is also shown in Figure 6.25.

The two `merge` routines (Figure 6.26) are drivers designed to remove special cases and ensure that H_1 has the smaller root. The actual merging is performed in `merge1` (Figure 6.27). The public `merge` method merges `rhs` into the controlling heap. `rhs` becomes empty. The alias test in the public method disallows `h.merge(h)`.

The time to perform the merge is proportional to the sum of the length of the right paths, because constant work is performed at each node visited during the recursive calls. Thus we obtain an $O(\log N)$ time bound to merge two leftist heaps. We can also perform this operation nonrecursively by essentially performing two passes. In the first pass, we create a new tree by merging the right paths of both heaps. To do this, we arrange the nodes on the right paths of H_1 and H_2 in sorted order, keeping their respective left children. In our example, the new right path is 3, 6, 7, 8, 18 and the resulting tree is shown in Figure 6.28. A second pass is made up the heap, and child swaps are performed at nodes that violate the leftist heap property. In Figure 6.28, there is a swap at nodes 7 and 3, and the same tree as before is obtained. The nonrecursive version is simpler to visualize but harder to code. We leave it to the reader to show that the recursive and nonrecursive procedures do the same thing.

As mentioned above, we can carry out insertions by making the item to be inserted a one-node heap and performing a `merge`. To perform a `deleteMin`, we merely destroy the root, creating two heaps, which can then be merged. Thus, the time to perform a `deleteMin` is $O(\log N)$. These two routines are coded in Figure 6.29 and Figure 6.30.

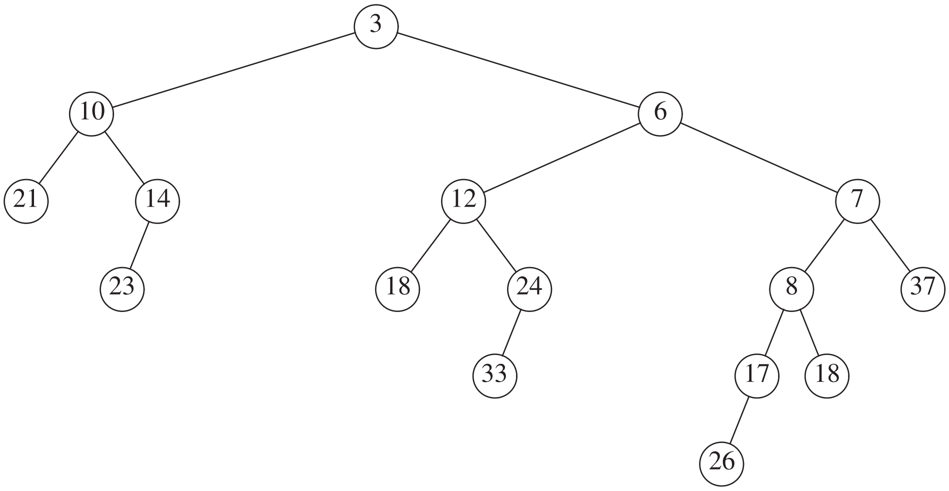

Figure 6.23 Result of attaching leftist heap of previous figure as H_1's right child

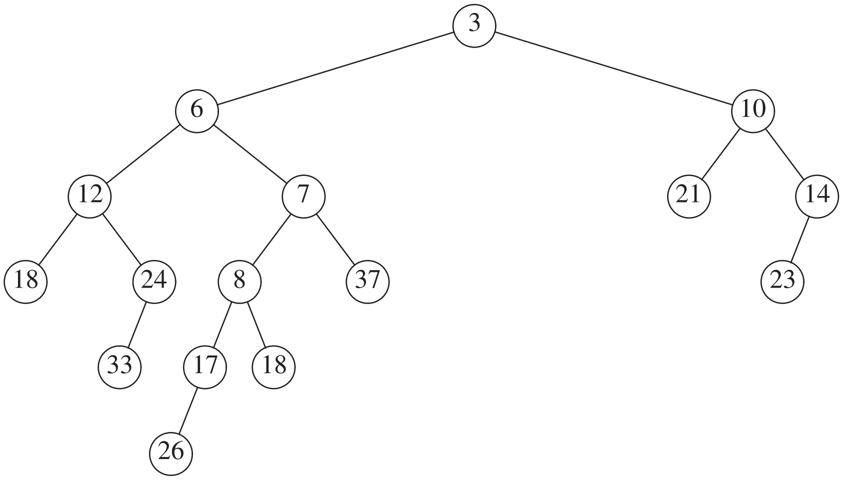

Figure 6.24 Result of swapping children of H_1's root

```
public class LeftistHeap<AnyType extends Comparable<? super AnyType>>
{
    public LeftistHeap( )
      { root = null; }

    public void merge( LeftistHeap<AnyType> rhs )
      { /* Figure 6.26 */ }
    public void insert( AnyType x )
      { /* Figure 6.29 */ }
    public AnyType findMin( )
      { /* See online code */ }
    public AnyType deleteMin( )
      { /* Figure 6.30 */ }

    public boolean isEmpty( )
      { return root == null; }
    public void makeEmpty( )
      { root = null; }

    private static class Node<AnyType>
    {
            // Constructors
        Node( AnyType theElement )
          { this( theElement, null, null ); }

        Node( AnyType theElement, Node<AnyType> lt, Node<AnyType> rt )
          { element = theElement; left = lt; right = rt; npl = 0; }

        AnyType       element;      // The data in the node
        Node<AnyType> left;         // Left child
        Node<AnyType> right;        // Right child
        int           npl;          // null path length
    }

    private Node<AnyType> root;    // root

    private Node<AnyType> merge( Node<AnyType> h1, Node<AnyType> h2 )
      { /* Figure 6.26 */ }
    private Node<AnyType> merge1( Node<AnyType> h1, Node<AnyType> h2 )
      { /* Figure 6.27 */ }
    private void swapChildren( Node<AnyType> t )
      { /* See online code */ }
}
```

Figure 6.25 Leftist heap type declarations

```
    /**
     * Merge rhs into the priority queue.
     * rhs becomes empty. rhs must be different from this.
     * @param rhs the other leftist heap.
     */
    public void merge( LeftistHeap<AnyType> rhs )
    {
        if( this == rhs )    // Avoid aliasing problems
            return;

        root = merge( root, rhs.root );
        rhs.root = null;
    }

    /**
     * Internal method to merge two roots.
     * Deals with deviant cases and calls recursive merge1.
     */
    private Node<AnyType> merge( Node<AnyType> h1, Node<AnyType> h2 )
    {
        if( h1 == null )
            return h2;
        if( h2 == null )
            return h1;
        if( h1.element.compareTo( h2.element ) < 0 )
            return merge1( h1, h2 );
        else
            return merge1( h2, h1 );
    }
```

Figure 6.26 Driving routines for merging leftist heaps

Finally, we can build a leftist heap in $O(N)$ time by building a binary heap (obviously using a linked implementation). Although a binary heap is clearly leftist, this is not necessarily the best solution, because the heap we obtain is the worst possible leftist heap. Furthermore, traversing the tree in reverse-level order is not as easy with links. The `buildHeap` effect can be obtained by recursively building the left and right subtrees and then percolating the root down. The exercises contain an alternative solution.

```
    /**
     * Internal method to merge two roots.
     * Assumes trees are not empty, and h1's root contains smallest item.
     */
    private Node<AnyType> merge1( Node<AnyType> h1, Node<AnyType> h2 )
    {
        if( h1.left == null )   // Single node
            h1.left = h2;       // Other fields in h1 already accurate
        else
        {
            h1.right = merge( h1.right, h2 );
            if( h1.left.npl < h1.right.npl )
                swapChildren( h1 );
            h1.npl = h1.right.npl + 1;
        }
        return h1;
    }
```

Figure 6.27 Actual routine to merge leftist heaps

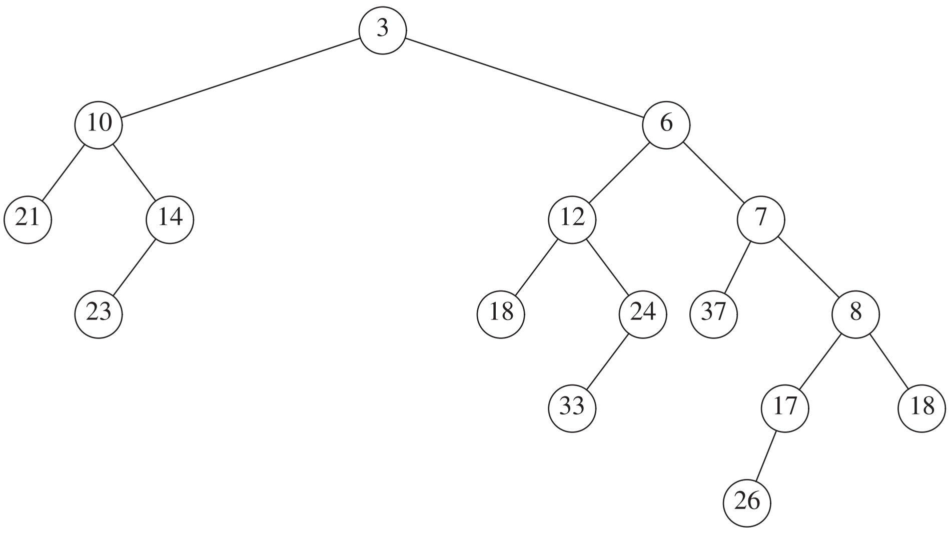

Figure 6.28 Result of merging right paths of H_1 and H_2

```
    /**
     * Insert into the priority queue, maintaining heap order.
     * @param x the item to insert.
     */
    public void insert( AnyType x )
    {
        root = merge( new Node<>( x ), root );
    }
```

Figure 6.29 Insertion routine for leftist heaps

```
    /**
     * Remove the smallest item from the priority queue.
     * @return the smallest item, or throw UnderflowException if empty.
     */
    public AnyType deleteMin( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( );

        AnyType minItem = root.element;
        root = merge( root.left, root.right );

        return minItem;
    }
```

Figure 6.30 `deleteMin` routine for leftist heaps

6.7 Skew Heaps

A **skew heap** is a self-adjusting version of a leftist heap that is incredibly simple to implement. The relationship of skew heaps to leftist heaps is analogous to the relation between splay trees and AVL trees. Skew heaps are binary trees with heap order, but there is no structural constraint on these trees. Unlike leftist heaps, no information is maintained about the null path length of any node. The right path of a skew heap can be arbitrarily long at any time, so the worst-case running time of all operations is $O(N)$. However, as with splay trees, it can be shown (see Chapter 11) that for any M consecutive operations, the total worst-case running time is $O(M \log N)$. Thus, skew heaps have $O(\log N)$ amortized cost per operation.

As with leftist heaps, the fundamental operation on skew heaps is merging. The `merge` routine is once again recursive, and we perform the exact same operations as before, with one exception. The difference is that for leftist heaps, we check to see whether the left and right children satisfy the leftist heap structure property and swap them if they do not. For skew heaps, the swap is unconditional; we *always* do it, with the one exception that the largest of all the nodes on the right paths does not have its children swapped. This one exception is what happens in the natural recursive implementation, so it is not really a special case at all. Furthermore, it is not necessary to prove the bounds, but since this node is guaranteed not to have a right child, it would be silly to perform the swap and give it one. (In our example, there are no children of this node, so we do not worry about it.) Again, suppose our input is the same two heaps as before, Figure 6.31.

If we recursively merge H_2 with the subheap of H_1 rooted at 8, we will get the heap in Figure 6.32.

Again, this is done recursively, so by the third rule of recursion (Section 1.3) we need not worry about how it was obtained. This heap happens to be leftist, but there is no

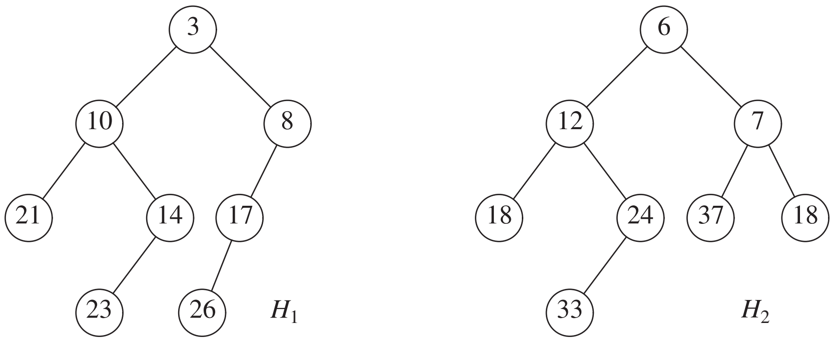

Figure 6.31 Two skew heaps H_1 and H_2

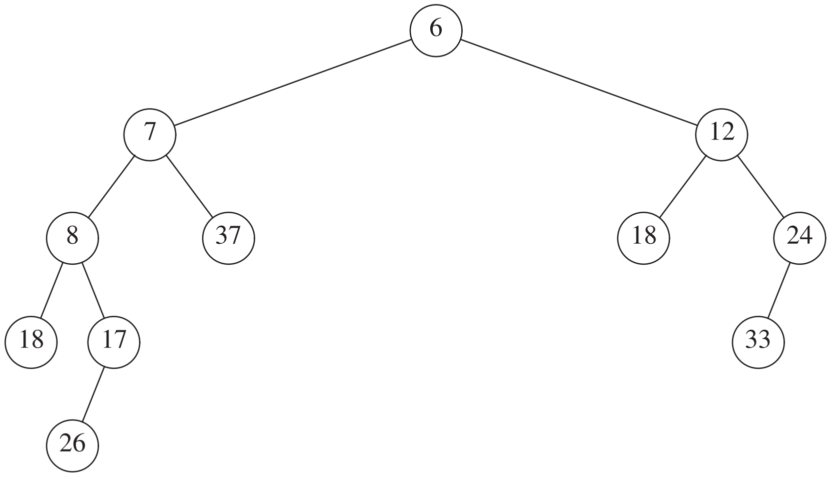

Figure 6.32 Result of merging H_2 with H_1's right subheap

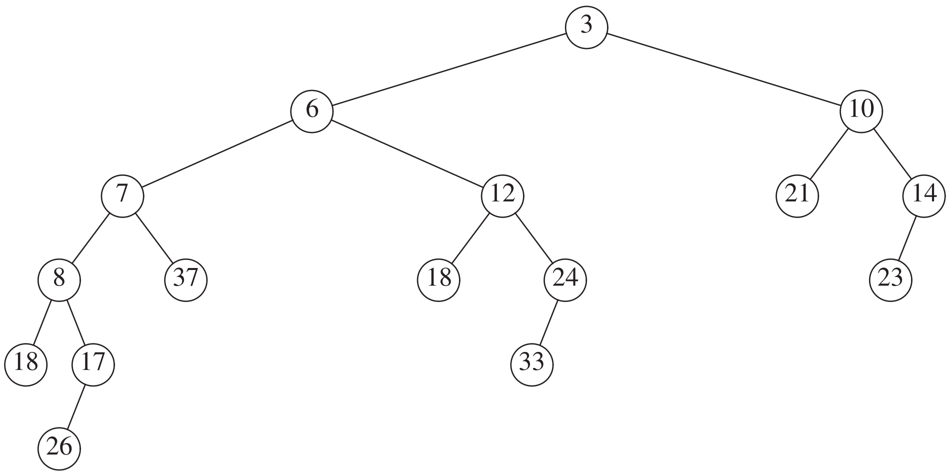

Figure 6.33 Result of merging skew heaps H_1 and H_2

guarantee that this is always the case. We make this heap the new left child of H_1, and the old left child of H_1 becomes the new right child (see Figure 6.33).

The entire tree is leftist, but it is easy to see that that is not always true: Inserting 15 into this new heap would destroy the leftist property.

We can perform all operations nonrecursively, as with leftist heaps, by merging the right paths and swapping left and right children for every node on the right path, with the exception of the last. After a few examples, it becomes clear that since all but the last node on the right path have their children swapped, the net effect is that this becomes the new left path (see the preceding example to convince yourself). This makes it very easy to merge two skew heaps visually.[3]

The implementation of skew heaps is left as a (trivial) exercise. Note that because a right path could be long, a recursive implementation could fail because of lack of stack space, even though performance would otherwise be acceptable. Skew heaps have the advantage that no extra space is required to maintain path lengths and no tests are required to determine when to swap children. It is an open problem to determine precisely the expected right path length of both leftist and skew heaps (the latter is undoubtedly more difficult). Such a comparison would make it easier to determine whether the slight loss of balance information is compensated by the lack of testing.

[3] This is not exactly the same as the recursive implementation (but yields the same time bounds). If we only swap children for nodes on the right path that are above the point where the merging of right paths terminated due to exhaustion of one heap's right path, we get the same result as the recursive version.