The most glaring weakness of the heap implementation, aside from the inability to perform *Finds*, is that combining two heaps into one is a hard operation. This extra operation 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.

5.6. 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 pointers. In practice, we can expect that this will make all the other operations slower; pointer manipulation is generally more time-consuming than multiplication and division by 2.

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. 不能用数组实现.

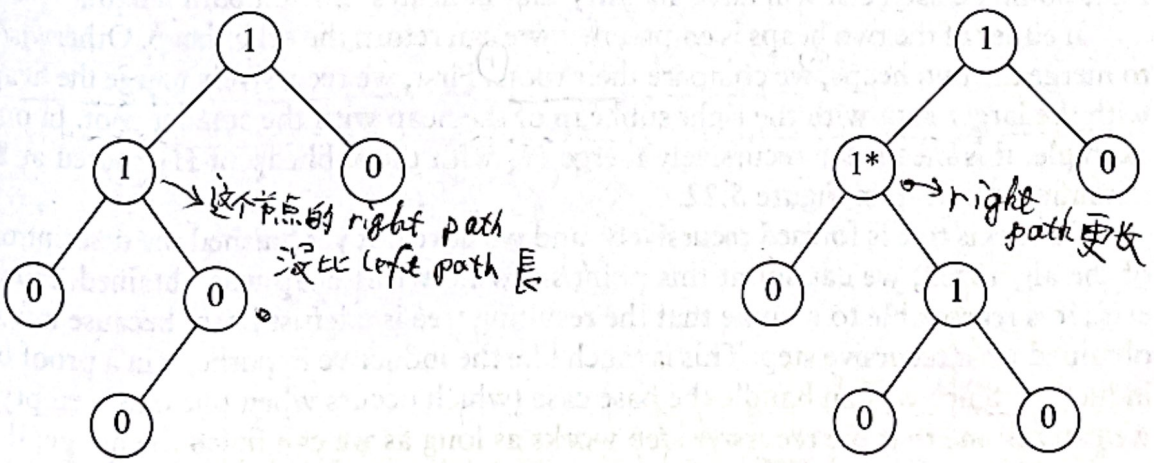
5.6.1. Leftist Heap Property

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 5.20, 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

Figure 5.20 Null path lengths for two trees; only the left tree is leftist



satisfied by only one of the trees in Figure 5.20, namely, the tree on the left. This property actually goes 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 5.2.

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 *Inserts* and *Merges* on the right path could destroy the leftist heap property. It turns out to be extremely easy to restore the property.

5.6.2. 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 be done nonrecursively. Our input is the two leftist heaps, H_1 and H_2, in Figure 5.21. 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 pointers, each cell 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 5.22.

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

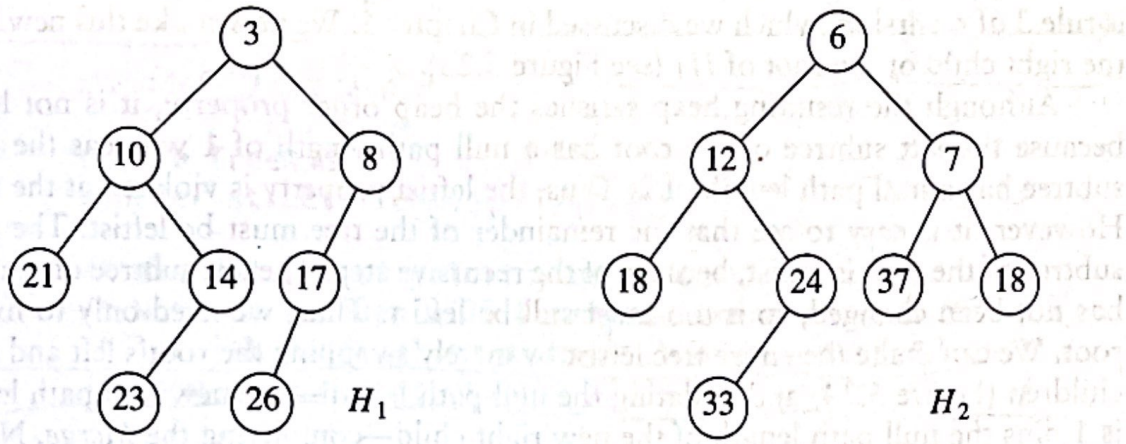


Figure 5.21 Two leftist heaps H_1 and H_2

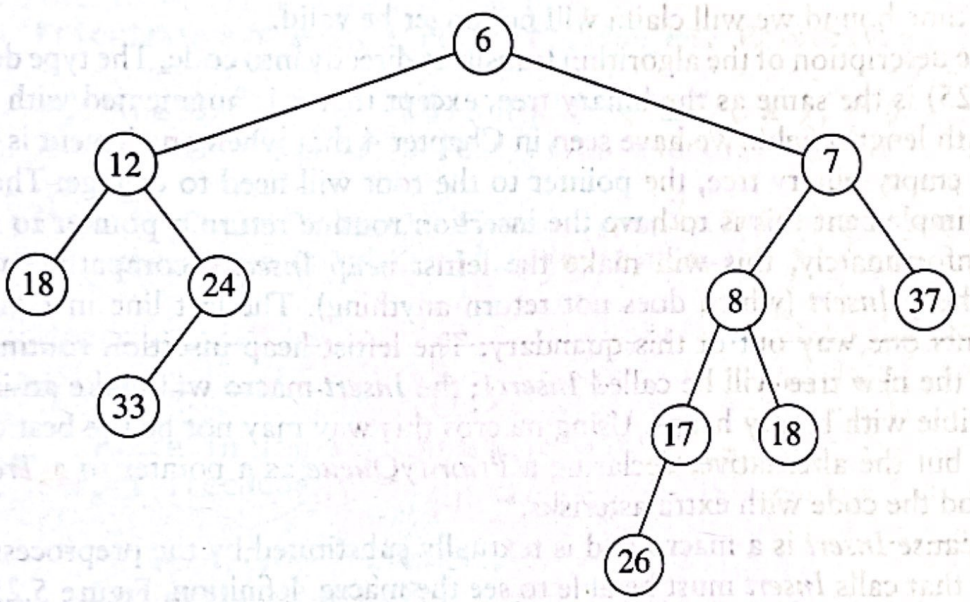


Figure 5.22 Result of merging H_2 with H_1's right subheap

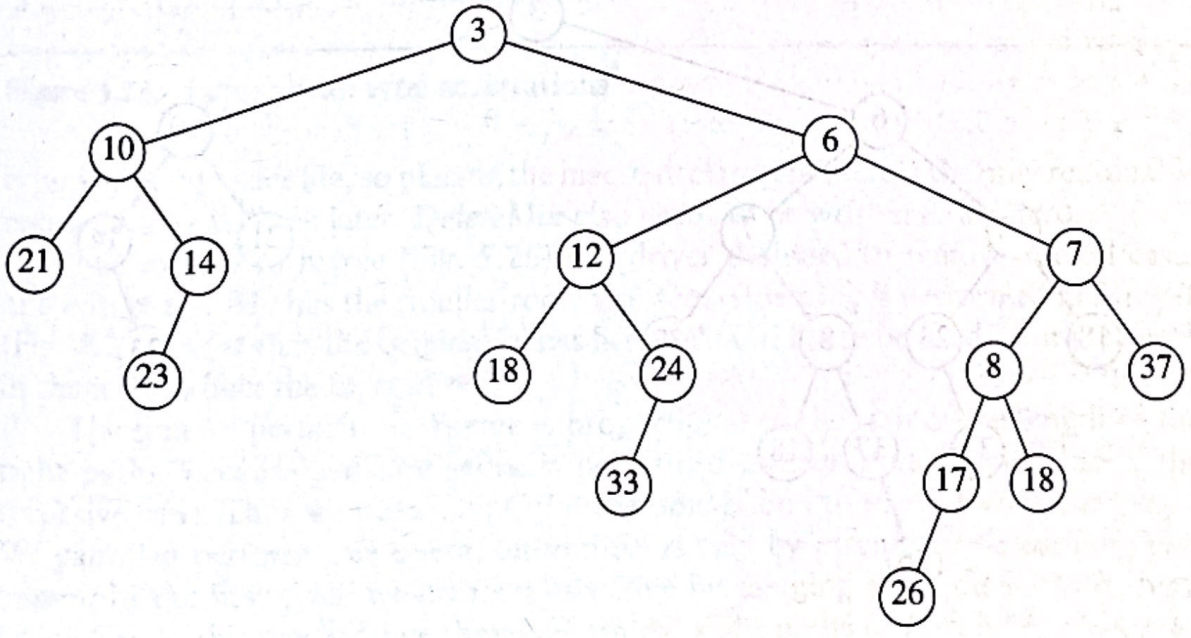


Figure 5.23 Result of attaching leftist heap of previous figure as H_1's right child

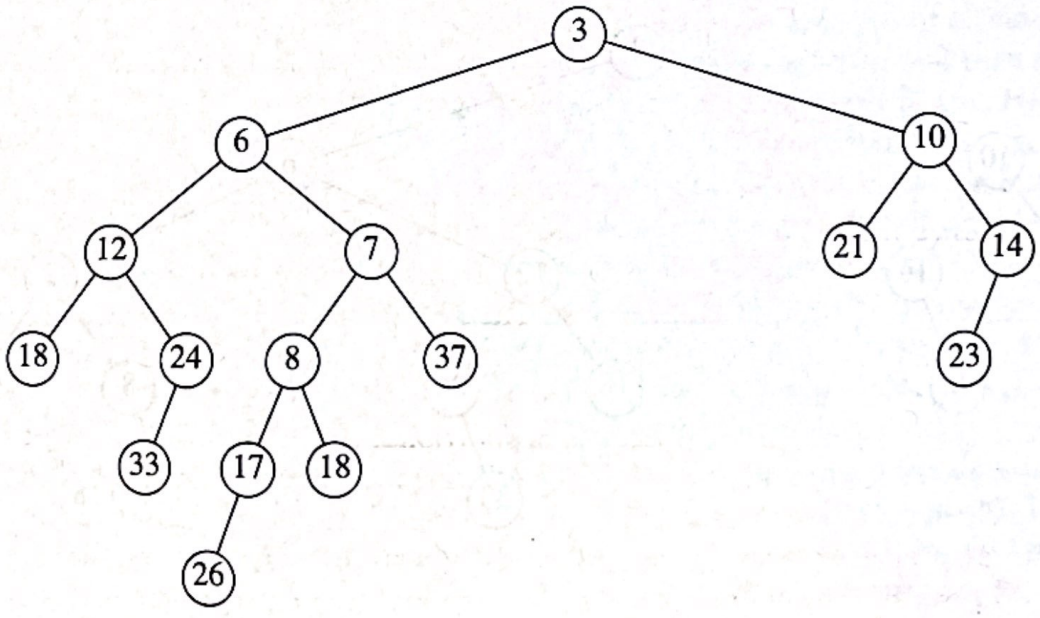
is rule 3 of recursion, which we discussed in Chapter 1. We now make this new heap the right child of the root of H_1 (see Figure 5.23).

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 5.24) 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 type definition (Fig. 5.25) is the same as the binary tree, except that it is augmented with the *Npl* (null path length) field. We have seen in Chapter 4 that when an element is inserted into an empty binary tree, the pointer to the root will need to change. The easiest way to implement this is to have the insertion routine return a pointer to the new tree. Unfortunately, this will make the leftist heap *Insert* incompatible with the binary heap *Insert* (which does not return anything). The last line in Figure 5.25 represents one way out of this quandary. The leftist heap insertion routine which returns the new tree will be called *Insert1*; the *Insert* macro will make an insertion compatible with binary heaps. Using macros this way may not be the best or safest course, but the alternative, declaring a *PriorityQueue* as a pointer to a *TreeNode*, will flood the code with extra asterisks.*

Because *Insert* is a macro and is textually substituted by the preprocessor, any routine that calls *Insert* must be able to see the macro definition. Figure 5.25 would

Figure 5.24 Result of swapping children of H_1's root



*Another possibility is to accept the incompatible interfaces as a necessary evil.

```
#ifndef _LeftHeap_H

struct TreeNode;
typedef struct TreeNode *PriorityQueue;

/* Minimal set of priority queue operations */
/* Note that nodes will be shared among several */
/* leftist heaps after a merge; the user must */
/* make sure to not use the old leftist heaps */

PriorityQueue Initialize( void );
ElementType FindMin( PriorityQueue H );
int IsEmpty( PriorityQueue H );
PriorityQueue Merge( PriorityQueue H1, PriorityQueue H2 );

#define Insert( X, H ) ( H = Insert1( ( X ), H ) )
/* DeleteMin macro is left as an exercise */

PriorityQueue Insert1( ElementType X, PriorityQueue H );
PriorityQueue DeleteMin1( PriorityQueue H );

#endif


/* Place in implementation file */
struct TreeNode
{
    ElementType   Element;
    PriorityQueue Left;
    PriorityQueue Right;
    int           Npl;
};
```

Figure 5.25 Leftist heap type declarations

typically be a header file, so placing the macro declaration there is the only reasonable course. As we will see later, *DeleteMin* also needs to be written as a macro.

The routine to merge (Fig. 5.26) is a driver designed to remove special cases and ensure that H_1 has the smaller root. The actual merging is performed in *Merge1* (Fig. 5.27). Note that the original leftist heaps should never be used again; changes in them will affect the merged result.

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

```
        PriorityQueue
        Merge( PriorityQueue H1, PriorityQueue H2 )
        {
/* 1*/      if( H1 == NULL )
/* 2*/          return H2;
/* 3*/      if( H2 == NULL )
/* 4*/          return H1;
/* 5*/      if( H1->Element < H2->Element )
/* 6*/          return Merge1( H1, H2 );
            else
/* 7*/          return Merge1( H2, H1 );
        }
```

Figure 5.26 Driving routine for merging leftist heaps

```
        static PriorityQueue
        Merge1( PriorityQueue H1, PriorityQueue H2 )
        {
/* 1*/      if( H1->Left == NULL )  /* Single node */
/* 2*/          H1->Left = H2;      /* H1->Right is already NULL,
                                       H1->Npl is already 0 */
            else
            {
/* 3*/          H1->Right = Merge( H1->Right, H2 );
/* 4*/          if( H1->Left->Npl < H1->Right->Npl )
/* 5*/              SwapChildren( H1 );

/* 6*/          H1->Npl = H1->Right->Npl + 1;
            }
/* 7*/      return H1;
        }
```

Figure 5.27 Actual routine to merge leftist heaps

3, 6, 7, 8, 18 and the resulting tree is shown in Figure 5.28. A second pass is made up the heap, and child swaps are performed at nodes that violate the leftist heap property. In Figure 5.28, 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 5.29 and Figure 5.30. *DeleteMin* can be written as a macro that calls *DeleteMin1* and *FindMin*. This is left as an exercise to the reader.

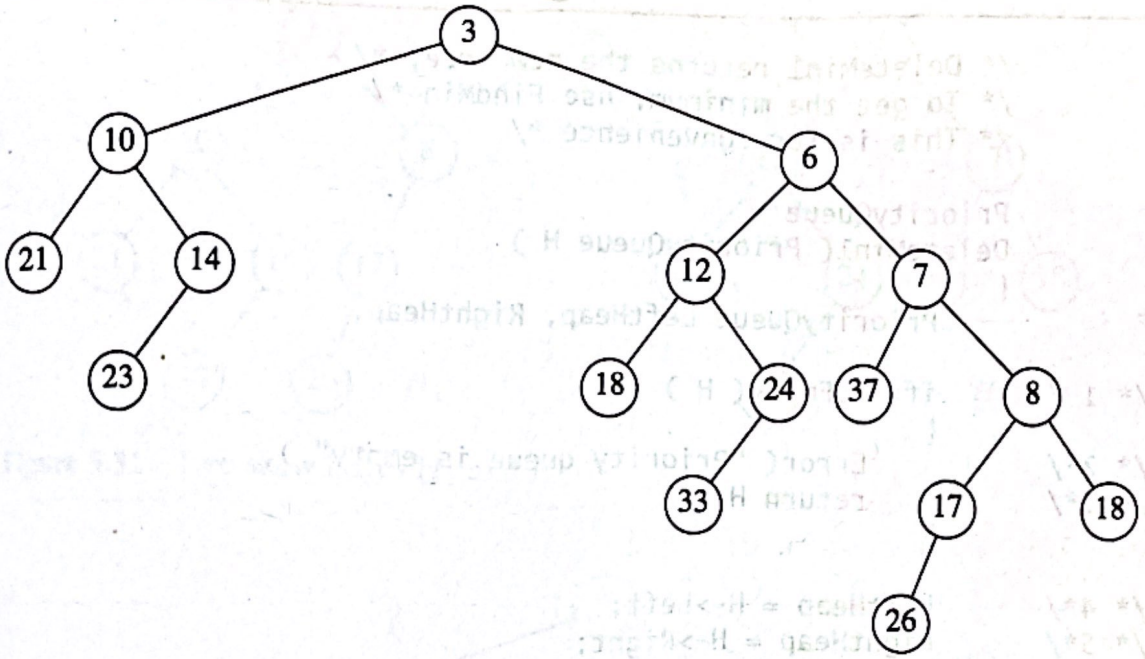


Figure 5.28 Result of merging right paths of H_1 and H_2

```
        PriorityQueue
        Insert1( ElementType X, PriorityQueue H )
        {
            PriorityQueue SingleNode;

/* 1*/      SingleNode = malloc( sizeof( struct TreeNode ) );
/* 2*/      if( SingleNode == NULL )
/* 3*/          FatalError( "Out of space!!!" );
            else
            {
/* 4*/          SingleNode->Element = X; SingleNode->Npl = 0;
/* 5*/          SingleNode->Left = SingleNode->Right = NULL;
/* 6*/          H = Merge( SingleNode, H );
            }
/* 7*/      return H;
        }
```

Figure 5.29 Insertion routine for leftist heaps

Finally, we can build a leftist heap in $O(N)$ time by building a binary heap (obviously using a pointer 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 pointers. 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.

```
        /* DeleteMin1 returns the new tree; */
        /* To get the minimum, use FindMin */
        /* This is for convenience */

        PriorityQueue
        DeleteMin1( PriorityQueue H )
        {
            PriorityQueue LeftHeap, RightHeap;

/* 1*/      if( IsEmpty( H ) )
            {
/* 2*/          Error( "Priority queue is empty" );
/* 3*/          return H;
            }

/* 4*/      LeftHeap = H->Left;
/* 5*/      RightHeap = H->Right;
/* 6*/      free( H );
/* 7*/      return Merge( LeftHeap, RightHeap );
        }
```

Figure 5.30 *DeleteMin* routine for leftist heaps

5.7. 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 order 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,

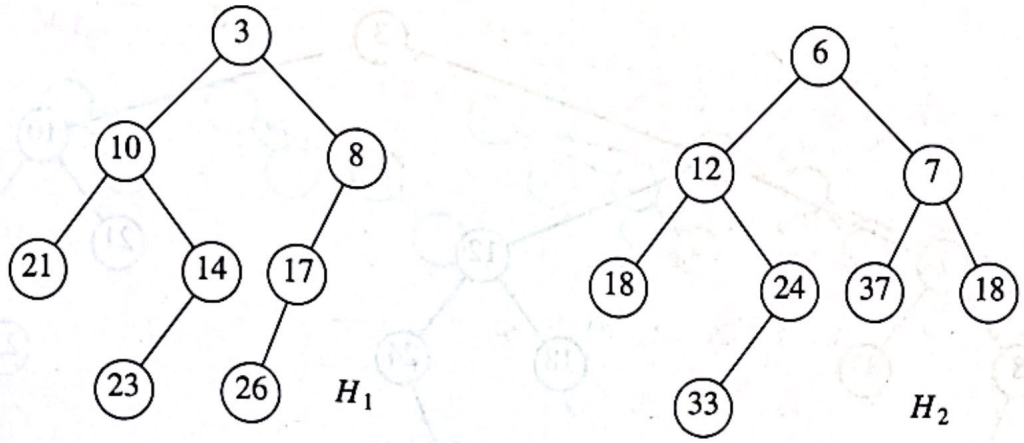


Figure 5.31 Two skew heaps H_1 and H_2

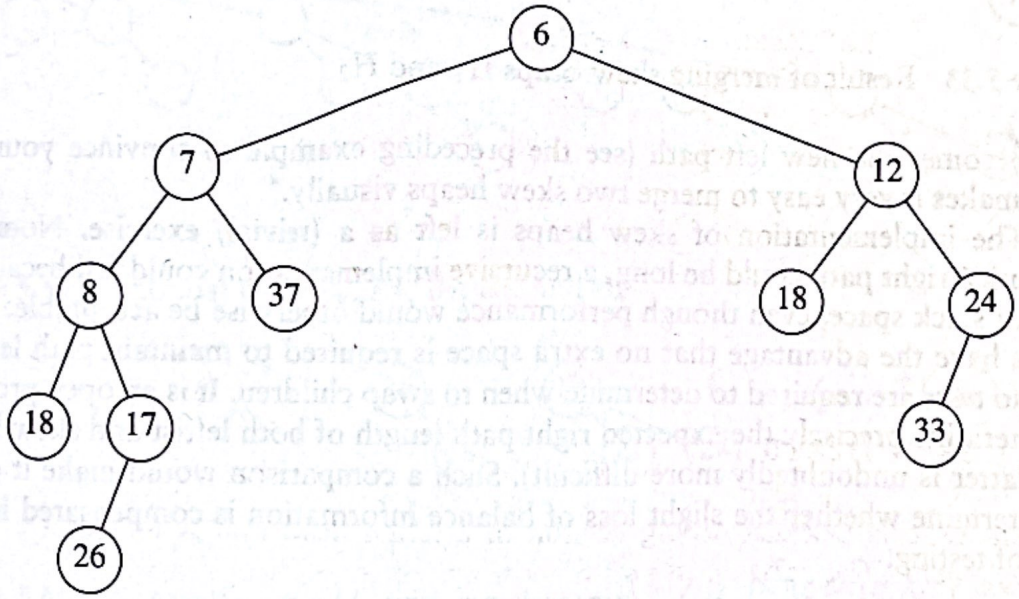


Figure 5.32 Result of merging H_2 with H_1's right subheap

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 5.31.

If we recursively merge H_2 with the subheap of H_1 rooted at 8, we will get the heap in Figure 5.32.

Again, this is done recursively, so by the third rule of recursion (Section 1.2) we need not worry about how it was obtained. This heap happens to be leftist, but there is no 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 Fig. 5.33).

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

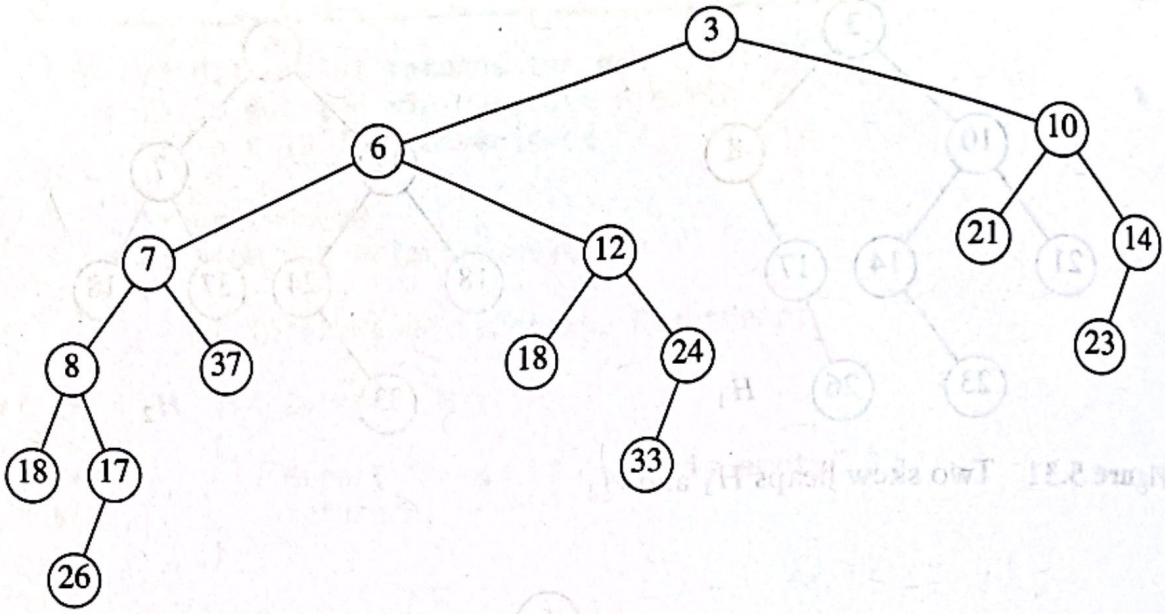


Figure 5.33 Result of merging skew heaps H_1 and H_2

this becomes the new left path (see the preceding example to convince yourself). This makes it very easy to merge two skew heaps visually.*

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.

5.8. Binomial Queues

Although both leftist and skew heaps support merging, insertion, and *DeleteMin* all effectively in $O(\log N)$ time per operation, there is room for improvement because we know that binary heaps support insertion in *constant average* time per operation. Binomial queues support all three operations in $O(\log N)$ worst-case time per operation, but insertions take constant time on average.

5.8.1. Binomial Queue Structure

Binomial queues differ from all the priority queue implementations that we have seen in that a binomial queue is not a heap-ordered tree but rather a *collection* of heap-ordered trees, known as a *forest*. Each of the heap-ordered trees is of a

*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.

THEOREM 11.1.

The amortized running times of Insert, DeleteMin, and Merge are $O(1)$, $O(\log N)$, *and* $O(\log N)$, *respectively, for binomial queues.*

PROOF:

The potential function is the number of trees. The initial potential is 0, and the potential is always nonnegative, so the amortized time is an upper bound on the actual time. The analysis for *Insert* follows from the argument above. For *Merge*, assume the two trees have N_1 and N_2 nodes with T_1 and T_2 trees, respectively. Let $N = N_1 + N_2$. The actual time to perform the merge is $O(\log(N_1) + \log(N_2)) = O(\log N)$. After the merge, there can be at most $\log N$ trees, so the potential can increase by at most $O(\log N)$. This gives an amortized bound of $O(\log N)$. The *DeleteMin* bound follows in a similar manner.

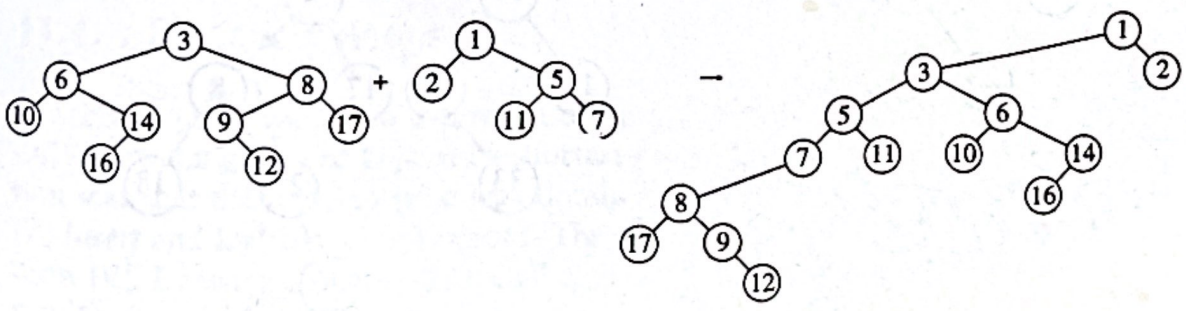
11.3. Skew Heaps

The analysis of binomial queues is a fairly easy example of an amortized analysis. We now look at skew heaps. As is common with many of our examples, once the right potential function is found, the analysis is easy. The difficult part is choosing a meaningful potential function.

Recall that for skew heaps, the key operation is merging. To merge two skew heaps, we merge their right paths and make this the new left path. For each node on the new path, except the last, the old left subtree is attached as the right subtree. The last node on the new left path is known to not have a right subtree, so it is silly to give it one. The bound does not depend on this exception, and if the routine is coded recursively, this is what will happen naturally. Figure 11.6 shows the result of merging two skew heaps.

Suppose we have two heaps, H_1 and H_2, and there are r_1 and r_2 nodes on their respective right paths. Then the actual time to perform the merge is proportional to $r_1 + r_2$, so we will drop the Big-Oh notation and charge one unit of time for each node on the paths. Since the heaps have no structure, it is possible that all the nodes in both heaps lie on the right path, and this would give a $\Theta(N)$ worst-case bound to merge the heaps (Exercise 11.3 asks you to construct an example). We will show that the amortized time to merge two skew heaps is $O(\log N)$.

Figure 11.6 Merging of two skew heaps



What is needed is some sort of a potential function that captures the effect of skew heap operations. Recall that the effect of a *Merge* is that every node on the right path is moved to the left path, and its old left child becomes the new right child. One idea might be to classify each node as a right node or left node, depending on whether or not it is a right child, and use the number of right nodes as a potential function. Although the potential is initially 0 and always nonnegative, the problem is that the potential does not decrease after a merge and thus does not adequately reflect the savings in the data structure. The result is that this potential function cannot be used to prove the desired bound.

A similar idea is to classify nodes as either heavy or light, depending on whether or not the right subtree of any node has more nodes than the left subtree.

DEFINITION: A node p is *heavy* if the number of descendants of p's right subtree is at least half of the number of descendants of p, and *light* otherwise. Note that the number of descendants of a node includes the node itself.

As an example, Figure 11.7 shows a skew heap. The nodes with keys 15, 3, 6, 12, and 7 are heavy, and all other nodes are light.

The potential function we will use is the number of heavy nodes in the (collection) of heaps. This seems like a good choice, because a long right path will contain an inordinate number of heavy nodes. Because nodes on this path have their children swapped, these nodes will be converted to light nodes as a result of the merge.

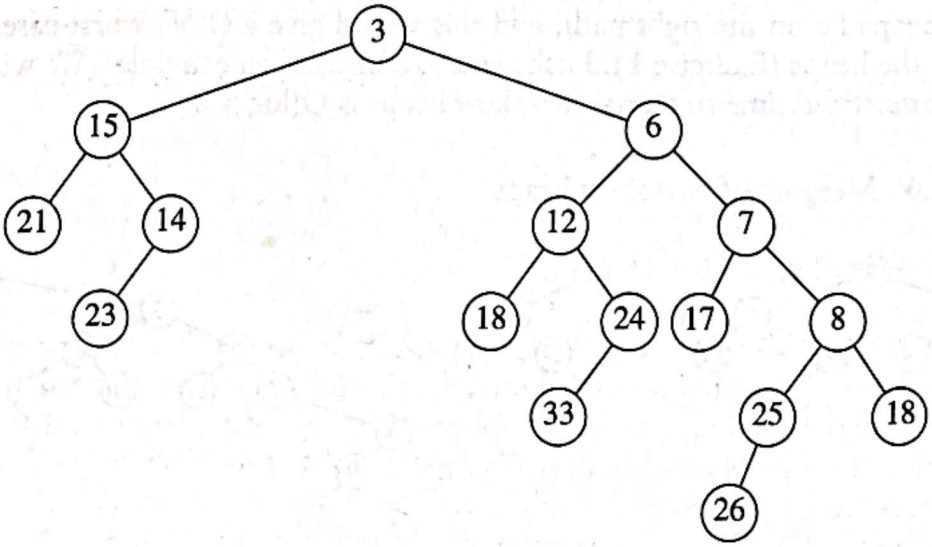
THEOREM 11.2.

The amortized time to merge two skew heaps is $O(\log N)$.

PROOF:

Let H_1 and H_2 be the two heaps, with N_1 and N_2 nodes respectively. Suppose the right path of H_1 has l_1 light nodes and h_1 heavy nodes, for a total of $l_1 + h_1$.

Figure 11.7 Skew heap—heavy nodes are 3, 6, 7, 12, and 15



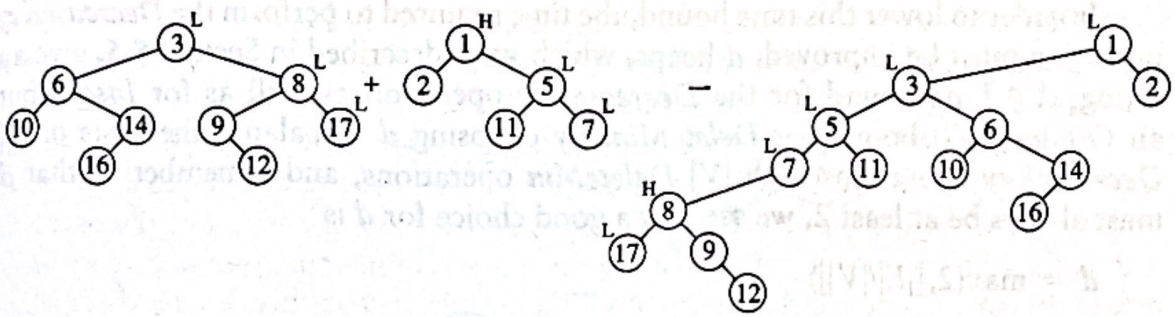


Figure 11.8 Change in heavy/light status after a merge

Likewise, H_2 has l_2 light and h_2 heavy nodes on its right path, for a total of $l_2 + h_2$ nodes.

If we adopt the convention that the cost of merging two skew heaps is the total number of nodes on their right paths, then the actual time to perform the merge is $l_1 + l_2 + h_1 + h_2$. Now the only nodes whose heavy/light status can change are nodes that are initially on the right path (and wind up on the left path), since no other nodes have their subtrees altered. This is shown by the example in Figure 11.8.

If a heavy node is initially on the right path, then after the merge it must become a light node. The other nodes that were on the right path were light and may or may not become heavy, but since we are proving an upper bound, we will have to assume the worst, which is that they become heavy and increase the potential. Then the net change in the number of heavy nodes is at most $l_1 + l_2 - h_1 - h_2$. Adding the actual time and the potential change (Equation (11.2)) gives an amortized bound of $2(l_1 + l_2)$.

Now we must show that $l_1 + l_2 = O(\log N)$. Since l_1 and l_2 are the number of light nodes on the original right paths, and the right subtree of a light node is less than half the size of the tree rooted at the light node, it follows directly that the number of light nodes on the right path is at most $\log N_1 + \log N_2$, which is $O(\log N)$.

The proof is completed by noting that the initial potential is 0 and that the potential is always nonnegative. It is important to verify this, since otherwise the amortized time does not bound the actual time and is meaningless.

Since the *Insert* and *DeleteMin* operations are basically just *Merges*, they also have $O(\log N)$ amortized bounds.

11.4. Fibonacci Heaps

In Section 9.3.2, we showed how to use priority queues to improve on the näive $O(|V|^2)$ running time of Dijkstra's shortest-path algorithm. The important observation was that the running time was dominated by $|E|$ *DecreaseKey* operations and $|V|$ *Insert* and *DeleteMin* operations. These operations take place on a set of size at most $|V|$. By using a binary heap, all these operations take $O(\log |V|)$ time, so the resulting bound for Dijkstra's algorithm can be reduced to $O(|E| \log |V|)$.