By the heap order property, the minimum element can always be found at the root. Thus, we get the extra operation, *FindMin,* in constant time.

5.3.3. *Basic Heap Operations*

It is easy (both conceptually and practically) to perform the two required operations. All the work involves ensuring that the heap order property is maintained.

Insert

To insert an element X into the heap, we create a hole in the next available location, since otherwise the tree will not be complete. If X can be placed in the hole without violating heap order, then we do so and are done. Otherwise we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. We continue this process until X can be placed in the hole. Figure 5.6 shows that to insert 14, we create a hole in the next available heap location. Inserting 14 in the hole would violate the heap order property, so 31 is slid down into the hole. This strategy is continued in Figure 5.7 until the correct location for 14 is found.

Figure 5.6 Attempt to insert 14: creating the hole, and bubbling the hole up

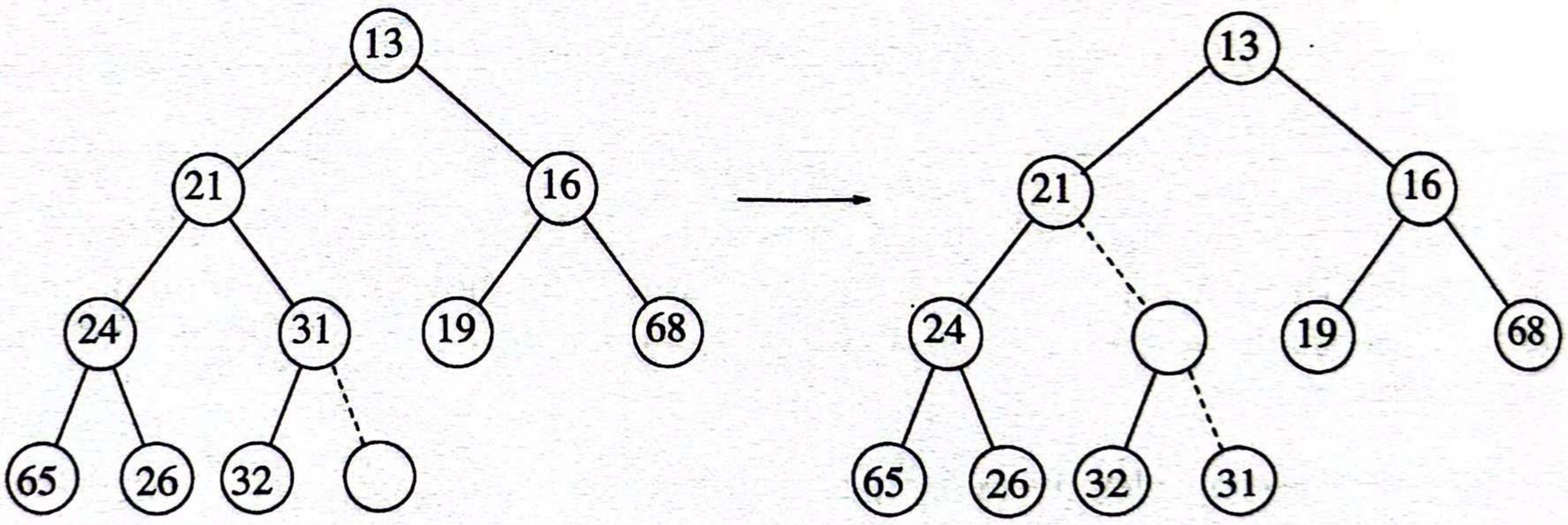
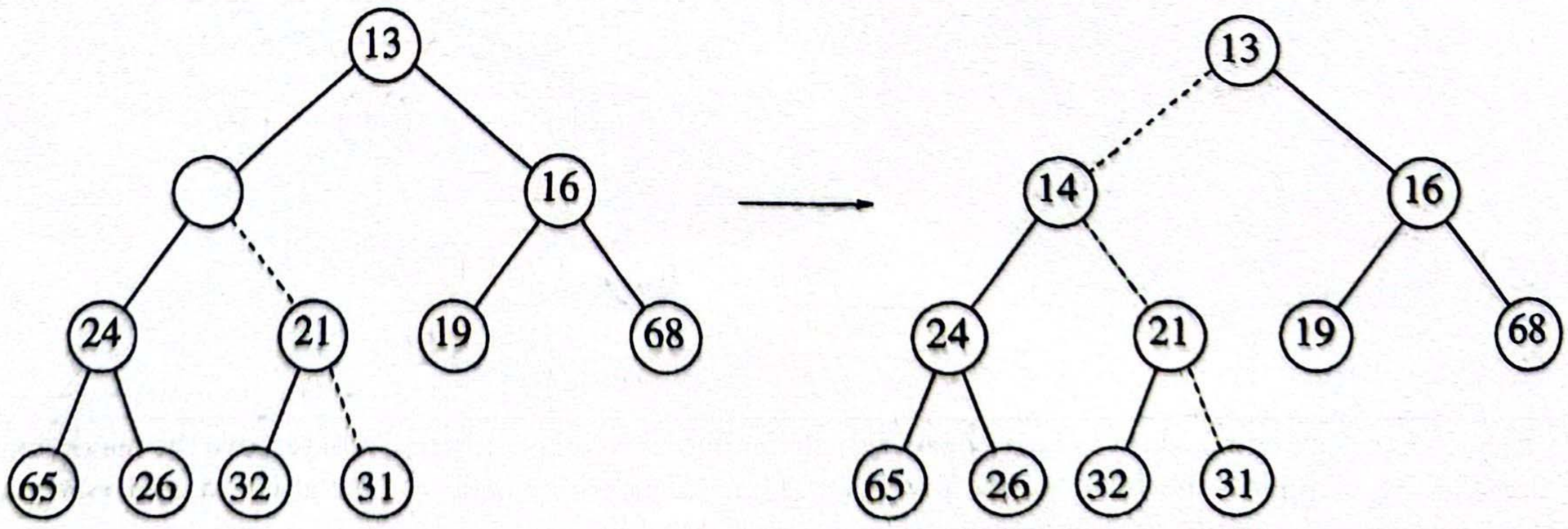


Figure 5.7 The remaining two steps to insert 14 in previous heap



```
/* H->Element[ 0 ] is a sentinel */

void
Insert( ElementType X, PriorityQueue H )
{
    int i;

    if( IsFull( H ) )
    {
        Error( "Priority queue is full" );
        return;
    }

    for( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
        H->Elements[ i ] = H->Elements[ i / 2 ];
    H->Elements[ i ] = X;
}
```

Figure 5.8 Procedure to insert into a binary key

This general strategy is known as a *percolate up;* the new element is percolated up the heap until the correct location is found. Insertion is easily implemented with the code shown in Figure 5.8.

We could have implemented the percolation in the *Insert* routine by performing repeated swaps until the correct order was established, but a swap requires three assignment statements. If an element is percolated up d levels, the number of assignments performed by the swaps would be $3d$. Our method uses $d + 1$ assignments.

If the element to be inserted is the new minimum, it will be pushed all the way to the top. At some point, i will be 1 and we will want to break out of the *while* loop. We could do this with an explicit test, but we have chosen to put a very small value in position 0 in order to make the *while* loop terminate. This value must be guaranteed to be smaller than (or equal to) any element in the heap; it is known as a *sentinel*. This idea is similar to the use of header nodes in linked lists. By adding a dummy piece of information, we avoid a test that is executed once per loop iteration, thus saving some time. ◁

The time to do the insertion could be as much as $O(\log N)$, if the element to be inserted is the new minimum and is percolated all the way to the root. On average, the percolation terminates early; it has been shown that 2.607 comparisons are required on average to perform an insert, so the average *Insert* moves an element up 1.607 levels.

DeleteMin

*DeleteMin*s are handled in a similar manner as insertions. Finding the minimum is easy; the hard part is removing it. When the minimum is removed, a hole is created at the root. Since the heap now becomes one smaller, it follows that the last element X in the heap must move somewhere in the heap. If X can be placed in the hole,

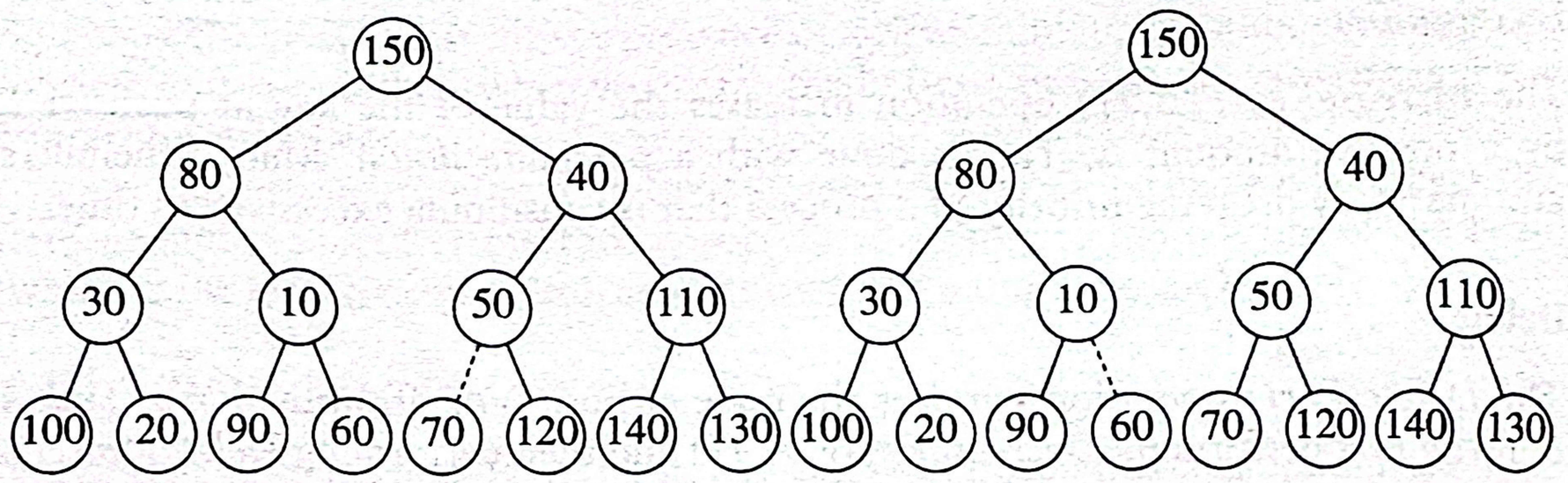


Figure 5.16 Left: after *PercolateDown*(6); right: after *PercolateDown*(5)

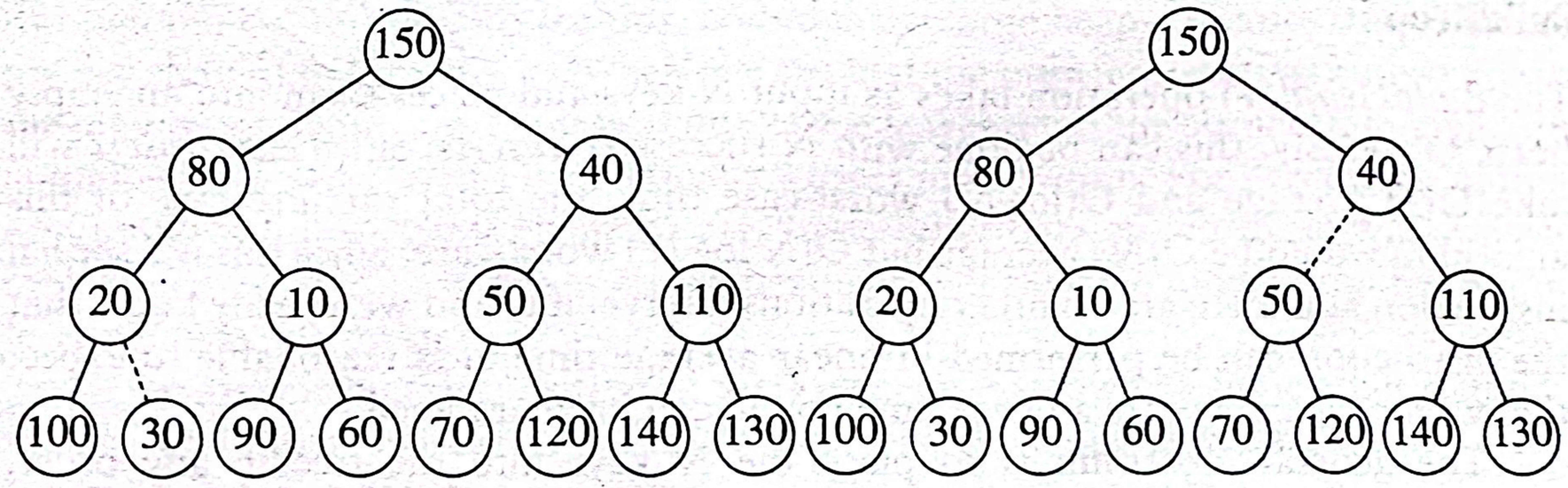


Figure 5.17 Left: after *PercolateDown*(4); right: after *PercolateDown*(3)

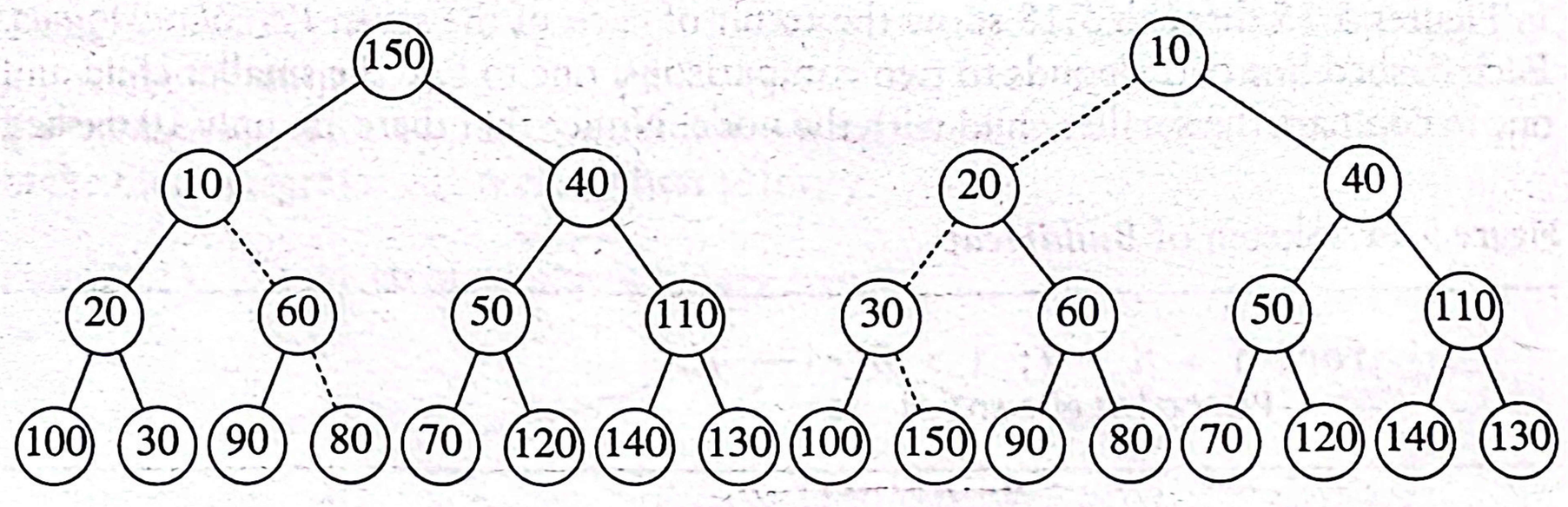


Figure 5.18 Left: after *PercolateDown*(2); right: after *PercolateDown*(1)

lines in the entire algorithm (there could have been an 11th—where?) corresponding to 20 comparisons.

To bound the running time of *BuildHeap*, we must bound the number of dashed lines. This can be done by computing the sum of the heights of all the nodes in the heap, which is the maximum number of dashed lines. What we would like to show is that this sum is $O(N)$.

THEOREM 5.1.

For the perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h + 1)$.

PROOF:
It is easy to see that this tree consists of 1 node at height h, 2 nodes at height $h-1$, 2^2 nodes at height $h-2$, and in general 2^i nodes at height $h-i$. The sum of the heights of all the nodes is then

$$S = \sum_{i=0}^{h} 2^i (h-i)$$

$$= h+2(h-1)+4(h-2)+8(h-3)+16(h-4)+\cdots+2^{h-1}(1) \quad (5.1)$$

Multiplying by 2 gives the equation

$$2S = 2h+4(h-1)+8(h-2)+16(h-3)+\cdots+2^h(1) \quad (5.2)$$

We subtract these two equations and obtain Equation (5.3). We find that certain terms almost cancel. For instance, we have $2h-2(h-1)=2$, $4(h-1)-4(h-2)=4$, and so on. The last term in Equation (5.2), 2^h, does not appear in Equation (5.1); thus, it appears in Equation (5.3). The first term in Equation (5.1), h, does not appear in Equation (5.2); thus, $-h$ appears in Equation (5.3). We obtain

$$S = -h+2+4+8+\cdots+2^{h-1}+2^h = (2^{h+1}-1)-(h+1) \quad (5.3)$$

which proves the theorem.

A complete tree is not a perfect binary tree, but the result we have obtained is an upper bound on the the sum of the heights of the nodes in a complete tree. Since a complete tree has between 2^h and 2^{h+1} nodes, this theorem implies that this sum is $O(N)$, where N is the number of nodes.

Although the result we have obtained is sufficient to show that *BuildHeap* is linear, the bound on the sum of the heights is not as strong as possible. For a complete tree with $N = 2^h$ nodes, the bound we have obtained is roughly $2N$. The sum of the heights can be shown by induction to be $N - b(N)$, where $b(N)$ is the number of 1s in the binary representation of N.

5.4. Applications of Priority Queues

We have already mentioned how priority queues are used in operating systems design. In Chapter 9, we will see how priority queues are used to implement several graph algorithms efficiently. Here we will show how to use priority queues to obtain solutions to two problems.

5.4.1. The Selection Problem

The first problem we will examine is the *selection problem* from Chapter 1. Recall that the input is a list of N elements, which can be totally ordered, and an integer k. The selection problem is to find the kth largest element.