average, and our implementation will allow building a heap of n items in linear time, if no deletions intervene. We will then discuss how to implement heaps to support efficient merging. This additional operation seems to complicate matters a bit and apparently requires the use of pointers.

6.3. Binary Heap

The implementation we will use is known as a *binary heap*. Its use is so common for priority queue implementations that when the word *heap* is used without a qualifier, it is generally assumed to be referring to this implementation of the data structure. In this section, we will refer to binary heaps as merely *heaps*. Like binary search trees, heaps have two properties, namely, a structure property and a heap order property. As with AVL trees, an operation on a heap can destroy one of the properties, so a heap operation must not terminate until all heap properties are in order. This turns out to be simple to do.

6.3.1. Structure Property

6.3.2. Heap Order Property

6.3.3. Basic Heap Operations

6.3.4. Other Heap Operations

6.3.1. Structure Property

A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a *complete binary tree*. Figure 6.2 shows an example.

It is easy to show that a complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes. This implies that the height of a complete binary tree is $\lfloor \log n \rfloor$, which is clearly $O(\log n)$.

An important observation is that because a complete binary tree is so regular, it can be represented in an array and no pointers are necessary. The array in Figure 6.3 corresponds to the heap in Figure 6.2.

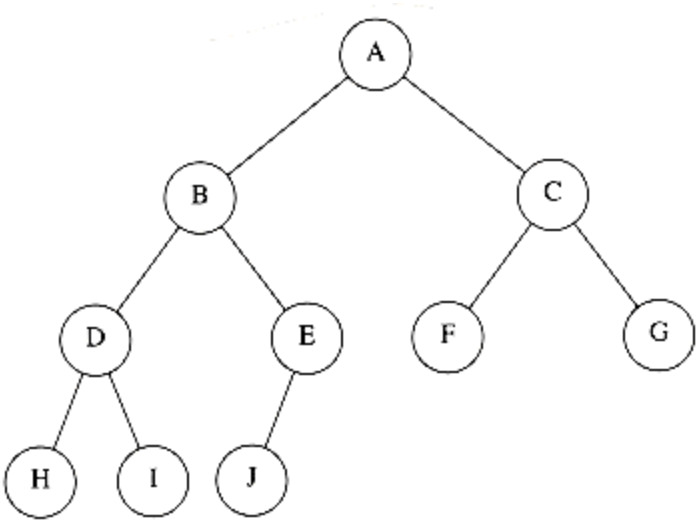

Figure 6.2 A complete binary tree

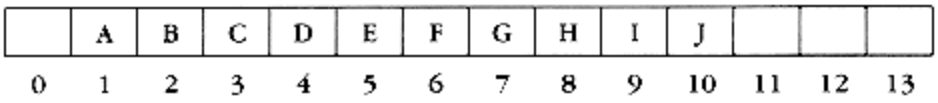

Figure 6.3 Array implementation of complete binary tree

For any element in array position i, the left child is in position $2i$, the right child is in the cell after the left child ($2i + 1$), and the parent is in position $\lfloor i/2 \rfloor$. Thus not only are pointers not required, but the operations required to traverse the tree are extremely simple and likely to be very fast on most computers. The only problem with this implementation is that an estimate of the maximum heap size is required in advance, but typically this is not a problem. In the figure above, the limit on the heap size is 13 elements. The array has a position 0; more on this later.

A heap data structure will, then, consist of an array (of whatever type the key is) and integers representing the maximum 2nd current heap size. Figure 6.4 shows a typical priority queue declaration. Notice the similarity to the stack declaration in Figure 3.47. Figure 6.4a creates an empty heap. Line 11 will be explained later.

Throughout this chapter, we shall draw the heaps as trees, with the implication that an actual implementation will use simple arrays.

6.3.2. Heap Order Property

The property that allows operations to be performed quickly is the *heap order* property. Since we want to be able to find the minimum quickly, it makes sense that the smallest element should be at the root. If we consider that any subtree should also be a heap, then any node should be smaller than all of its descendants.

Applying this logic, we arrive at the heap order property. In a heap, for every node *X*, the key in the parent of *X* is smaller than (or equal to) the key in *X*, with the obvious exception of the root (which has no parent).* In Figure 6.5 the tree on the left is a heap, but the tree on the right is not (the dashed line shows the violation of heap order). As usual, we will assume that the keys are integers, although they could be arbitrarily complex.

*Analogously, we can declare a (*max*) heap, which enables us to efficiently find and remove the maximum element, by changing the heap order property. Thus, a priority queue can be used to find *either* a minimum or a maximum, but this needs to be decided ahead of time.

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

```
struct heap_struct
{
/* Maximum # that can fit in the heap */
unsigned int max_heap_size;
/* Current # of elements in the heap */
unsigned int size;
element_type *elements;
};
typedef struct heap_struct *PRIORITY_QUEUE;
```

Figure 6.4 Declaration for priority queue

```
PRIORITY_QUEUE
create_pq( unsigned int max_elements )
{
PRIORITY_QUEUE H;
/*1*/       if( max_elements < MIN_PQ_SIZE )
/*2*/            error("Priority queue size is too small");
/*3*/       H = (PRIORITY_QUEUE) malloc ( sizeof (struct heap_struct) );
/*4*/       if( H == NULL )
/*5*/            fatal_error("Out of space!!!");
/* Allocate the array + one extra for sentinel */
/*6*/       H->elements = (element_type *) malloc
( ( max_elements+1) * sizeof (element_type) );
/*7*/       if( H->elements == NULL )
/*8*/           fatal_error("Out of space!!!");
/*9*/       H->max_heap_size = max_elements;
/*10*/      H->size = 0;
/*11*/      H->elements[0] = MIN_DATA;
/*12*/      return H;
}
```

Figure 6.4a

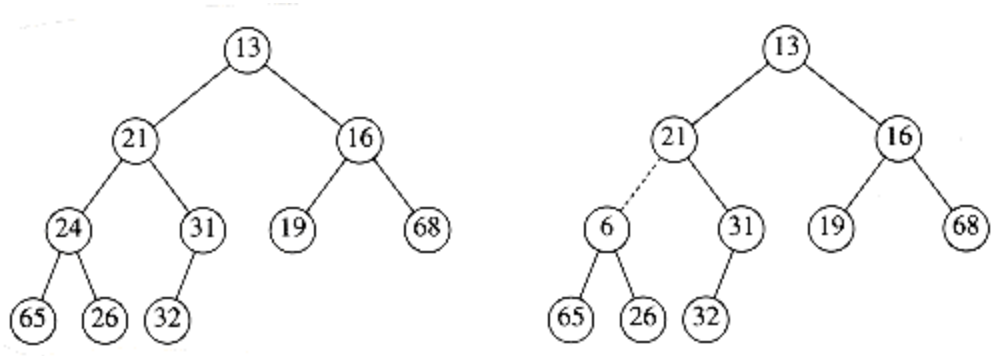

Figure 6.5 Two complete trees (only the left tree is a heap)

6.3.3. 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

Delete_min

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 6.6 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 6.7 until the correct location for 14 is found.

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 6.8.

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 3*d*. Our method uses *d* + 1 assignments.

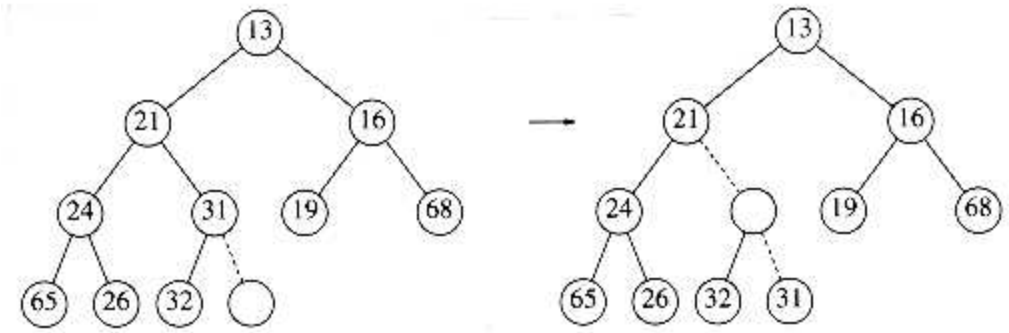

Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

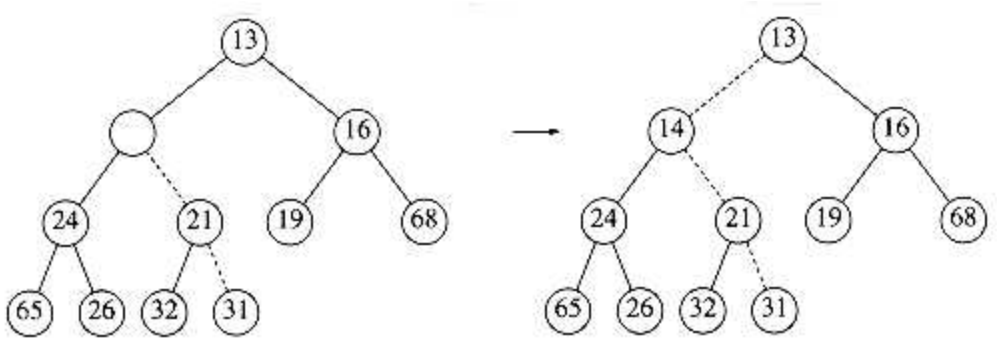

Figure 6.7 The remaining two steps to insert 14 in previous heap

```
/* H->element[0] is a sentinel */
void
insert( element_type x, PRIORITY_QUEUE H )
{
unsigned int i;
/*1*/       if( is_full( H ) )
/*2*/            error("Priority queue is full");
else
{
/*3*/            i = ++H->size;
/*4*/            while( H->elements[i/2] > x )
{
/*5*/                 H->elements[i] = H->elements[i/2];
/*6*/                 i /= 2;
}
/*7*/            H->elements[i] = x;
}
}
```

Figure 6.8 Procedure to insert into a binary heap

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.

Delete_min

Delete_mins 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, then we are done. This is unlikely, so we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat this step until x can be placed in the hole. Thus, our action is to place x in its correct spot along a path from the root containing *minimum* children.

In Figure 6.9 the left figure shows a heap prior to the *delete_min*. After 13 is removed, we must now try to place 31 in the heap. 31 cannot be placed in the hole, because this would violate heap order. Thus, we place the smaller child (14) in the hole, sliding the hole down one level (see Fig. 6.10). We repeat this again, placing 19 into the hole and creating a new hole one level deeper. We then place 26 in the hole and create a new hole on the bottom level. Finally, we are able to place 31 in the hole (Fig. 6.11). This general strategy is known as a *percolate down*. We use the same technique as in the *insert* routine to avoid the use of swaps in this routine.

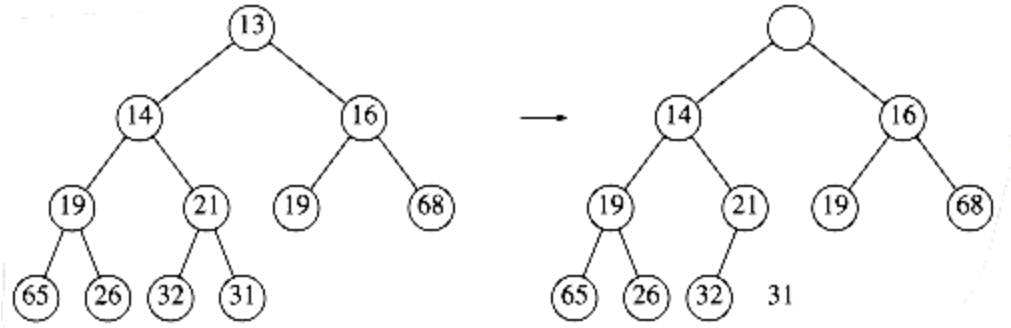

Figure 6.9 Creation of the hole at the root

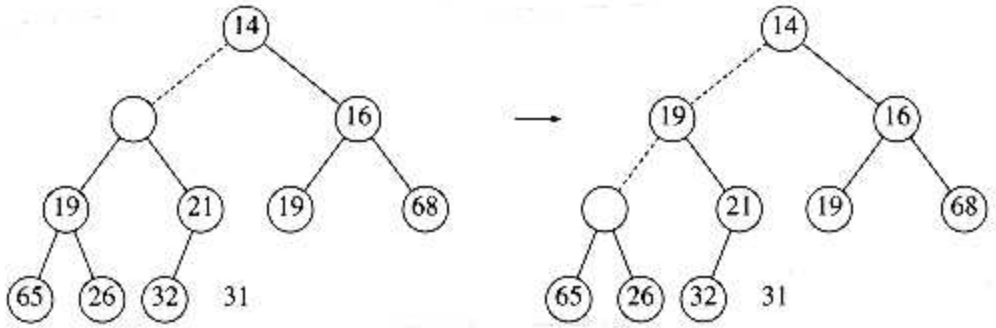

Figure 6.10 Next two steps in delete_min

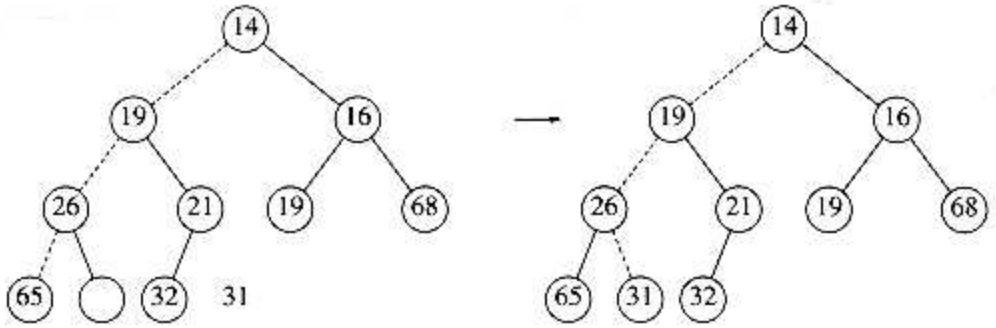

Figure 6.11 Last two steps in delete_min

A frequent implementation error in heaps occurs when there are an even number of elements in the heap, and the one node that has only one child is encountered. You must make sure not to assume that there are always two children, so this usually involves an extra test. In the code, depicted in Figure 6.12, we've done this test at line 8. One extremely tricky solution is always to ensure that your algorithm *thinks* every node has two children. Do this by placing a sentinel, of value higher than any in the heap, at the spot after the heap ends, at the start of each *percolate down* when the heap size is even. You should think very carefully before attempting this, and you must put in a prominent comment if you do use this technique.

```
element_type
delete_min( PRIORITY_QUEUE H )
{
unsigned int i, child;
element_type min_element, last_element;
/*1*/       if( is_empty( H ) )
{
/*2*/            error("Priority queue is empty");
/*3*/            return H->elements[0];
}
/*4*/       min_element = H->elements[1];
/*5*/       last_element = H->elements[H->size--];
/*6*/       for( i=1; i*2 <= H->size; i=child )
{
```

```
/* find smaller child */
/*7*/             child = i*2;
/*8*/             if( ( child != H->size ) &&
( H->elements[child+1] < H->elements [child] ) )
/*9*/                 child++;
/* percolate one level */
/*10*/            if( last_element > H->elements[child] )
/*11*/                H->elements[i] = H->elements[child];
else
/*12*/                break;
}
/*13*/      H->elements[i] = last_element;
/*14*/      return min_element;
}
```

Figure 6.12 Function to perform delete_min in a binary heap

Although this eliminates the need to test for the presence of a right child, you cannot eliminate the requirement that you test when you reach the bottom because this would require a sentinel for every leaf.

The worst-case running time for this operation is $O(\log n)$. On average, the element that is placed at the root is percolated almost to the bottom of the heap (which is the level it came from), so the average running time is $O(\log n)$.

6.3.4. Other Heap Operations

Notice that although finding the minimum can be performed in constant time, a heap designed to find the minimum element (also known as a (*min*) heap) is of no help whatsoever in finding the maximum element. In fact, a heap has very little ordering information, so there is no way to find any particular key without a linear scan through the entire heap. To see this, consider the large heap structure (the elements are not shown) in Figure 6.13, where we see that the only information known about the maximum element is that it is at one of the leaves. Half the elements, though, are contained in leaves, so this is practically useless information. For this reason, if it is important to know where elements are, some other data structure, such as a hash table, must be used in addition to the heap. (Recall that the model does not allow looking inside the heap.)

If we assume that the position of every element is known by some other method, then several other operations become cheap. The three operations below all run in logarithmic worst-case time.

Decrease_key

Increase_key

Delete

Build_heap

Decrease_key

The *decrease_key*(*x*, Δ, *H*) operation lowers the value of the key at position *x* by a positive amount Δ. Since this might violate the heap order, it must be fixed by a *percolate up*. This operation could be useful to system administrators: they can make their programs run with highest priority

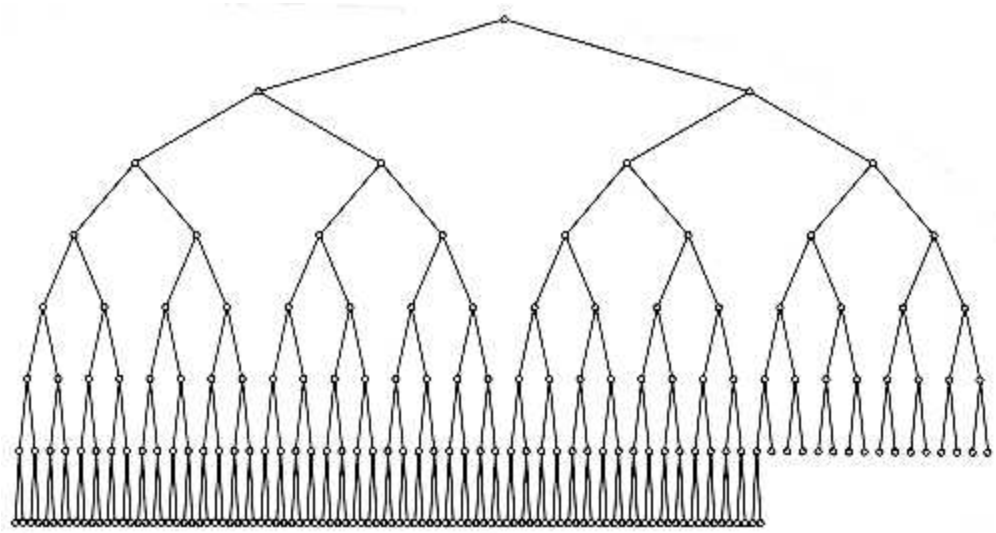

Figure 6.13 A very large complete binary tree

Increase_key

The *increase_key*(*x*, Δ, *H*) operation increases the value of the key at position *x* by a positive amount Δ. This is done with a *percolate down*. Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.

Delete

The *delete*(*x*, *H*) operation removes the node at position *x* from the heap. This is done by first performing *decrease_key*(*x*, ∞, H) and then performing *delete_min* (*H*). When a process is terminated by a user (instead of finishing normally), it must be removed from the priority queue.

Build_heap

The *build_heap*(*H*) operation takes as input *n* keys and places them into an empty heap. Obviously, this can be done with *n* successive *inserts*. Since each *insert* will take $O(1)$ average and $O(\log n)$ worst-case time, the total running time of this algorithm would be $O(n)$ average but $O(n \log n)$ worst-case. Since this is a special instruction and there are no

other operations intervening, and we already know that the instruction can be performed in linear average time, it is reasonable to expect that with reasonable care a linear time bound can be guaranteed.

The general algorithm is to place the *n* keys into the tree in any order, maintaining the structure property. Then, if *percolate_down*(*i*) percolates down from node *i*, perform the algorithm in Figure 6.14 to create a heap-ordered tree.

The first tree in Figure 6.15 is the unordered tree. The seven remaining trees in Figures 6.15 through 6.18 show the result of each of the seven *percolate downs*. Each dashed line corresponds to two comparisons: one to find the smaller child and one to compare the smaller child with the node. Notice that there are only 10 dashed lines in the entire algorithm (there could have been an 11th -- where?) corresponding to 20 comparisons.

```
for(i=n/2; i>0; i-- )
percolate_down( i );
```

Figure 6.14 Sketch of build_heap

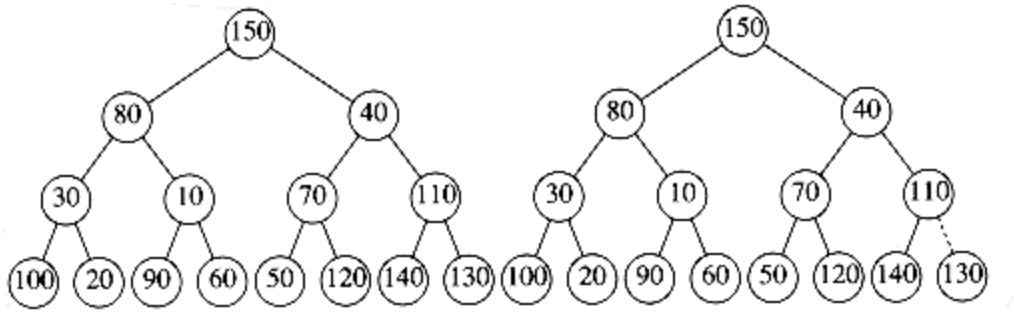

Figure 6.15 Left: initial heap; right: after percolate_down(7)

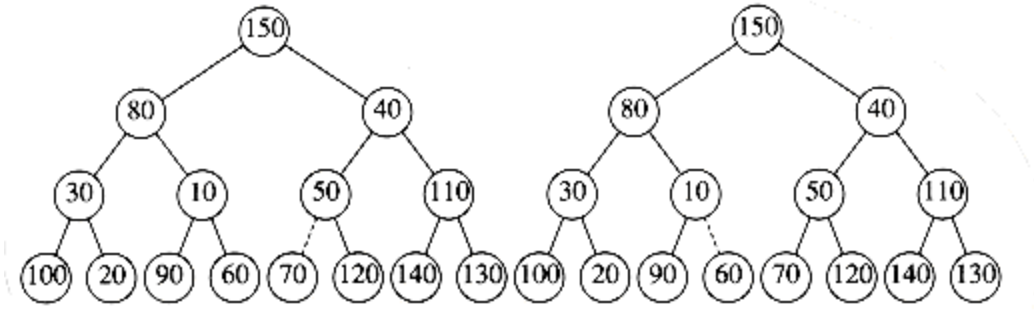

Figure 6.16 Left: after percolate_down(6); right: after percolate_down(5)

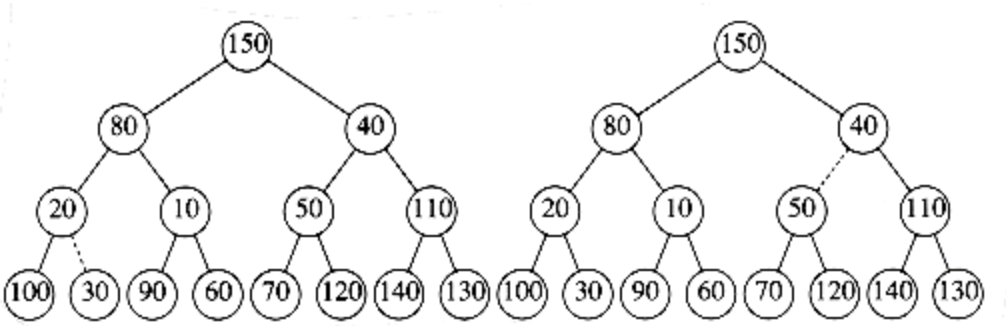

Figure 6.17 Left: after percolate_down(4); right: after percolate_down(3)

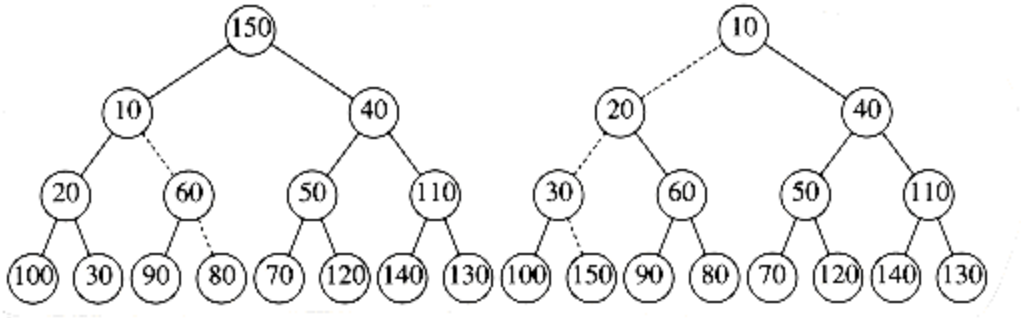

Figure 6.18 Left: after percolate_down(2); right: after percolate_down(1)

To bound the running time of *build_heap*, 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 6.1.

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

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

$= h + 2(h - 1) + 4(h - 2) + 8(h - 3) + 16(h - 4) + \ldots + 2^h - 1$ (1)

(6.1)

Multiplying by 2 gives the equation

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

(6.2)

We subtract these two equations and obtain Equation (6.3). 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 (6.2), 2^h, does not appear in Equation (6.1); thus, it appears in Equation (6.3). The first term in Equation (6.1), h, does not appear in equation (6.2); thus, $-h$ appears in Equation (6.3).

We obtain

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

(6.3)

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 *build_heap* 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.

6.4. 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.

6.4.1. The Selection Problem

6.4.2. Event Simulation

6.4.1. 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.

Two algorithms were given in Chapter 1, but neither is very efficient. The first algorithm, which we shall call Algorithm 1A, is to read the elements into an array and sort them, returning the appropriate element. Assuming a simple sorting algorithm, the running time is $O(n^2)$. The alternative algorithm, 1B, is to read k elements into an array and sort them. The smallest of these is in the kth position. We process the remaining elements one by one. As an element arrives, it is compared with kth element in the array. If it is larger, then the kth element is removed, and the new element is placed in the correct place among the remaining $k - 1$ elements. When the algorithm ends, the element in the kth position is the answer. The running time is $O(n \cdot k)$ (why?). If $k = \lceil n/2 \rceil$, then both algorithms are $O(n^2)$. Notice that for any k, we can solve the symmetric problem of finding the $(n - k + 1)$th smallest element, so $k = \lceil n/2 \rceil$ is really the hardest case for these algorithms. This also happens to be the most interesting case, since this value of k is known as the *median*.

We give two algorithms here, both of which run in $O(n \log n)$ in the extreme case of $k = \lceil n/2 \rceil$, which is a distinct improvement.

Algorithm 6A

Algorithm 6B

Algorithm 6A

For simplicity, we assume that we are interested in finding the kth *smallest* element. The algorithm is simple. We read the n elements into an array. We then apply the *build_heap* algorithm to this array. Finally, we'll perform k *delete_min* operations. The last element extracted from the heap is our answer. It should be clear that by changing the heap order property, we could solve the original problem of finding the kth *largest* element.

The correctness of the algorithm should be clear. The worst-case timing is $O(n)$ to construct the heap, if *build_heap* is used, and $O(\log n)$ for each *delete_min*. Since there are k *delete_mins*, we obtain a total running time of $O(n + k \log n)$. If $k = O(n/\log n)$, then the running time is dominated by the *build_heap* operation and is $O(n)$. For larger values of k, the running time is $O(k \log n)$. If $k = \lceil n/2 \rceil$, then the running time is $\Theta(n \log n)$.

Notice that if we run this program for $k = n$ and record the values as they leave the heap, we will have essentially sorted the input file in $O(n \log n)$ time. In Chapter 7, we will refine this idea to obtain a fast sorting algorithm known as *heapsort*.

Algorithm 6B

For the second algorithm, we return to the original problem and find the kth *largest* element. We use the idea from Algorithm 1B. At any point in time we will maintain a set S of the k largest elements. After the first k elements are read, when a new element is read, it is compared with the kth largest element, which we denote by S_k. Notice that S_k is the smallest element in S. If the new element is larger, then it replaces S_k in S. S will then have a new smallest element, which may or may not be the newly added element. At the end of the input, we find the smallest element in S and return it as the answer.

This is essentially the same algorithm described in Chapter 1. Here, however, we will use a heap to implement S. The first k elements are placed into the heap in total time $O(k)$ with a call to *build_heap*. The time to process each of the remaining elements is $O(1)$, to test if the element goes into S, plus $O(\log k)$, to delete S_k and insert the new element if this is necessary. Thus, the total time is $O(k + (n - k) \log k) = O(n \log k)$. This algorithm also gives a bound of $\Theta(n \log n)$ for finding the median.

In Chapter 7, we will see how to solve this problem in $O(n)$ average time. In Chapter 10, we will see an elegant, albeit impractical, algorithm to solve this problem in $O(n)$ worst-case time.

6.4.2. Event Simulation

In Section 3.4.3, we described an important queuing problem. Recall that we have a system, such as a bank, where customers arrive and wait on a line until one of k tellers is available. Customer arrival is governed by a probability distribution function, as is the service time (the amount of time to be served once a teller is available). We are interested in statistics such as how long on average a customer has to wait or how long the line might be.

With certain probability distributions and values of k, these answers can be computed exactly. However, as k gets larger, the analysis becomes considerably more difficult, so it is appealing to use a computer to simulate the operation of the bank. In this way, the bank officers can determine how many tellers are needed to ensure reasonably smooth service.

A simulation consists of processing events. The two events here are (a) a customer arriving and (b) a customer departing, thus freeing up a teller.

We can use the probability functions to generate an input stream consisting of ordered pairs of arrival time and service time for each customer, sorted by arrival time. We do not need to use the exact time of day. Rather, we can use a quantum unit, which we will refer to as a *tick*.

One way to do this simulation is to start a simulation clock at zero ticks. We then advance the clock one tick at a time, checking to see if there is an event. If there is, then we process the event(s) and compile statistics. When there are no customers left in the input stream and all the tellers are free, then the simulation is over.