**UNIT-II**

**Pageno 192-209 and 219-227 in text book by mark allen weiss**

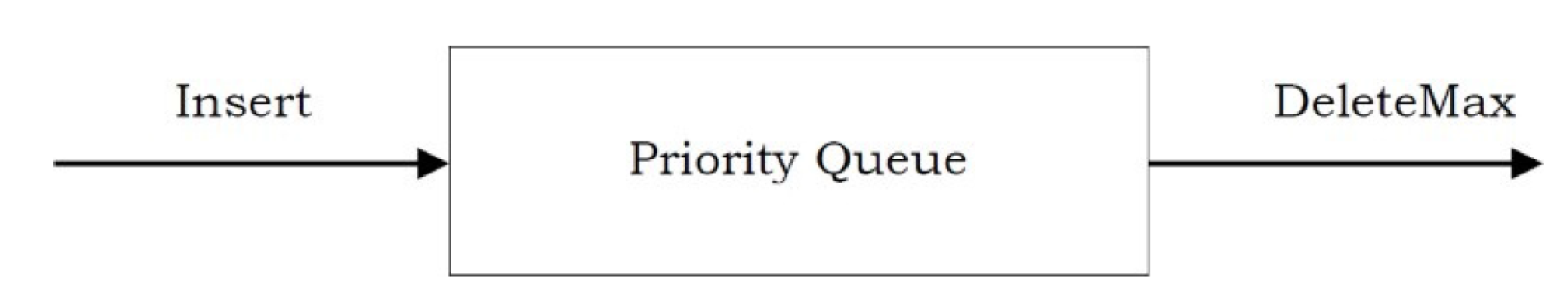
**7.1 What is a Priority Queue?**

In some situations we may need to find the minimum/maximum element among a collection of elements. We can do this with the help of Priority Queue ADT. A priority queue ADT is a data

structure that supports the operations *Insert* and *DeleteMin* (which returns and removes the minimum element) or *DeleteMax* (which returns and removes the maximum element).

These operations are equivalent to *EnQueue* and *DeQueue o*perations of a queue. The difference is that, in priority queues, the order in which the elements enter the queue may not be the same in which they were processed. An example application of a priority queue is job scheduling, which is prioritized instead of serving in first come first serve.

A priority queue is called an *ascending* – *priority* queue, if the item with the smallest key has the highest priority (that means, delete the smallest element always). Similarly, a priority queue is said to be a *descending* –*priority* queue if the item with the largest key has the highest priority (delete the maximum element always). Since these two types are symmetric we will be concentrating on one of them: ascending-priority queue.



**Figure 6.1 Basic model of a priority queue**

As with most data structures, it is sometimes possible to add other perations,

but these are extensions and not part of the basic model depicted in Figure 6.1.

Priority queues have many applications besides operating systems. In Chapter 7, we will see how priority queues are used for external sorting. Priority queues are also important in the implementation of greedy algorithms, which operate by repeatedly finding a minimum; Although jobs sent to a line printer are generally placed on a queue, this might not always be the best thing to do. For instance, one job might be particularly important, so that it might be desirable to allow that job to be run as soon as the printer is available. Conversely, if, when the printer becomes available,there are several one-page jobs and one hundred-page job, it might be reasonable to make the long job go last, even if it is not the last job submitted. (Unfortunately, most systems do not do this, which can be particularly annoying at times.)

Similarly, in a multiuser environment, the operating system scheduler must decide which of several processes to run. Generally a process is only allowed to run for a fixed period of time. One algorithm uses a queue. Jobs are initially placed at the end of the queue. The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up, and place it at the end of the queue if it does not finish. This strategy is generally not appropriate, because very short jobs will seem to take a long time because of the wait involved to run. Generally, it is important that short jobs finish as fast as possible, so these jobs should have preference over jobs that have already been running. Furthermore, some jobs that are not short are still very important and should also have preference. This particular application seems to require a special kind of queue, known as a priority queue.

6.1. Model

A priority queue is a data structure that allows at least the following two

operations: insert, which does the obvious thing, and delete\_min, which finds,returns and removes the minimum element in the heap. The insert operation is the equivalent of enqueue, and delete\_min is the priority queue equivalent of the queue's dequeue operation. The delete\_min function also alters its input. Current thinking in the software engineering community suggests that this is no longer a good idea. However, we will continue to use this function because of historical reasons--many programmers expect delete\_min to operate this way.

6.2. Simple Implementations

There are several obvious ways to implement a priority queue. We could use a

simple linked list, performing insertions at the front in O(1) and traversing the list, which requires O(n) time, to delete the minimum. Alternatively, we could insist that the list be always kept sorted; this makes insertions expensive (O (n)) and delete\_mins cheap (O(1)). The former is probably the better idea of the two, based on the fact that there are never more delete\_mins than insertions.

Another way of implementing priority queues would be to use a binary search tree.This gives an O(log n) average running time for both operations. This is true in spite of the fact that although the insertions are random, the deletions are not. Recall that the only element we ever delete is the minimum. Repeatedly removing a node that is in the left subtree would seem to hurt the balance of the tree by making the right subtree heavy. However, the right subtree is random. In the worst case, where the delete\_mins have depleted the left subtree, the right subtree would have at most twice as many elements as it should. This adds only a small constant to its expected depth. Notice that the bound can be made into a worst-case bound by using a balanced tree; this protects one against bad insertion sequences. Using a search tree could be overkill because it supports a host of operations that are not required. The basic data structure we will use will not require pointers and will support both operations in O(log n) worst-case time. Insertion will actually take constant time on 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.

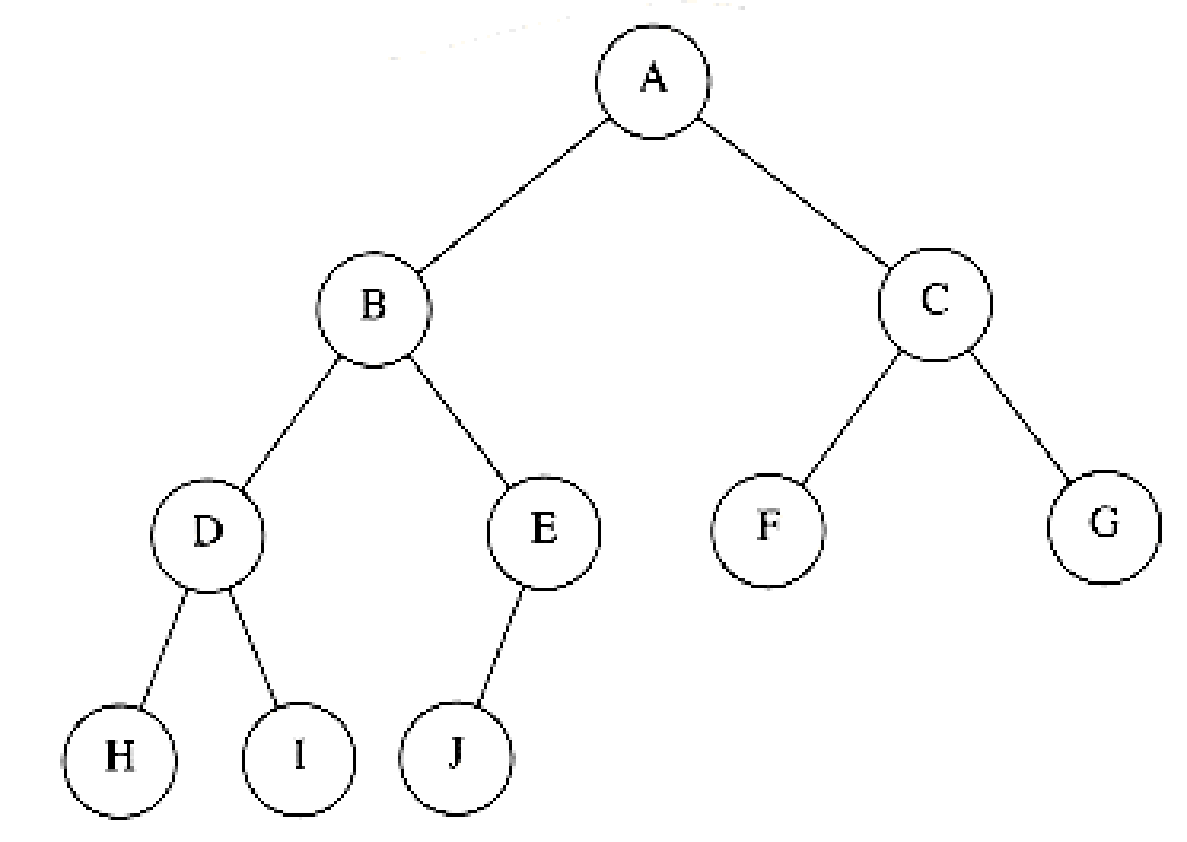
6.3.1. Structure Property

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 2h and

2h+1 - 1 nodes. This implies that the height of a complete binary tree is log

n , 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.

****

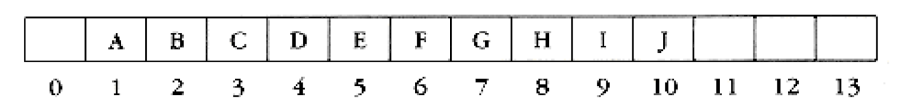
****

Figure 6.3 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 *i*/2 . 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.

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;

}

Structures, Algorithm Analysis: CHAPTER 6: PRIORITY QUEUES (HEAPS) 页码，5/46

mk:@

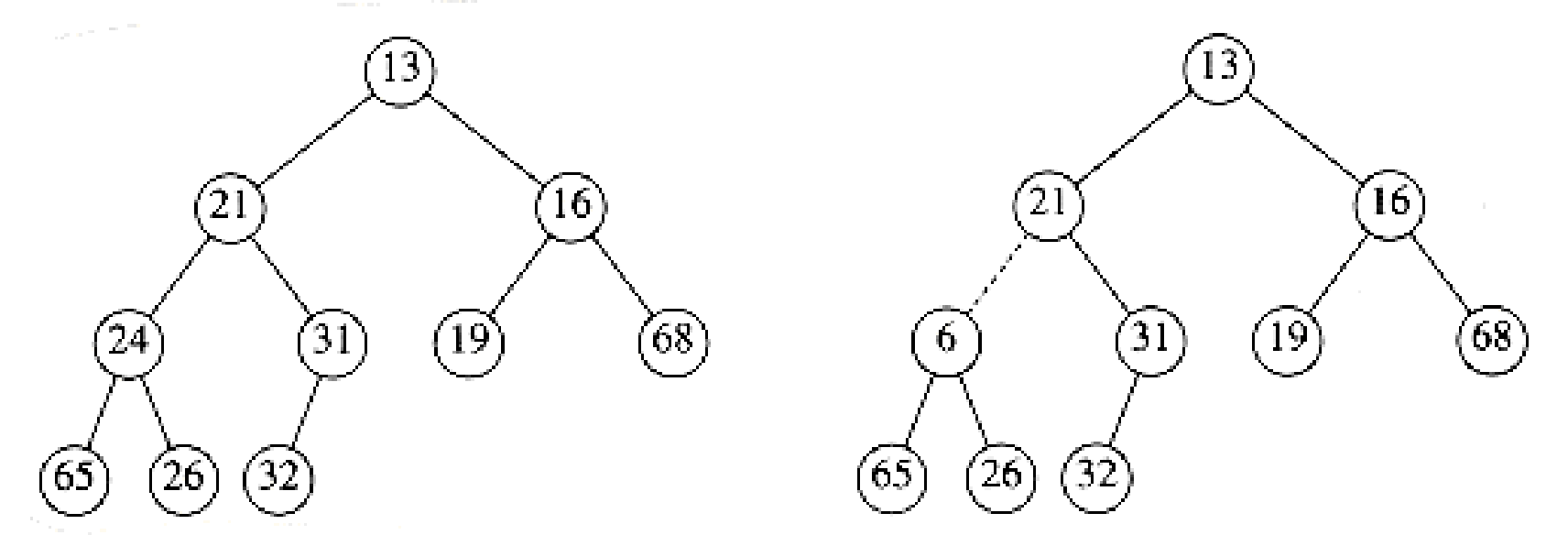
****

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

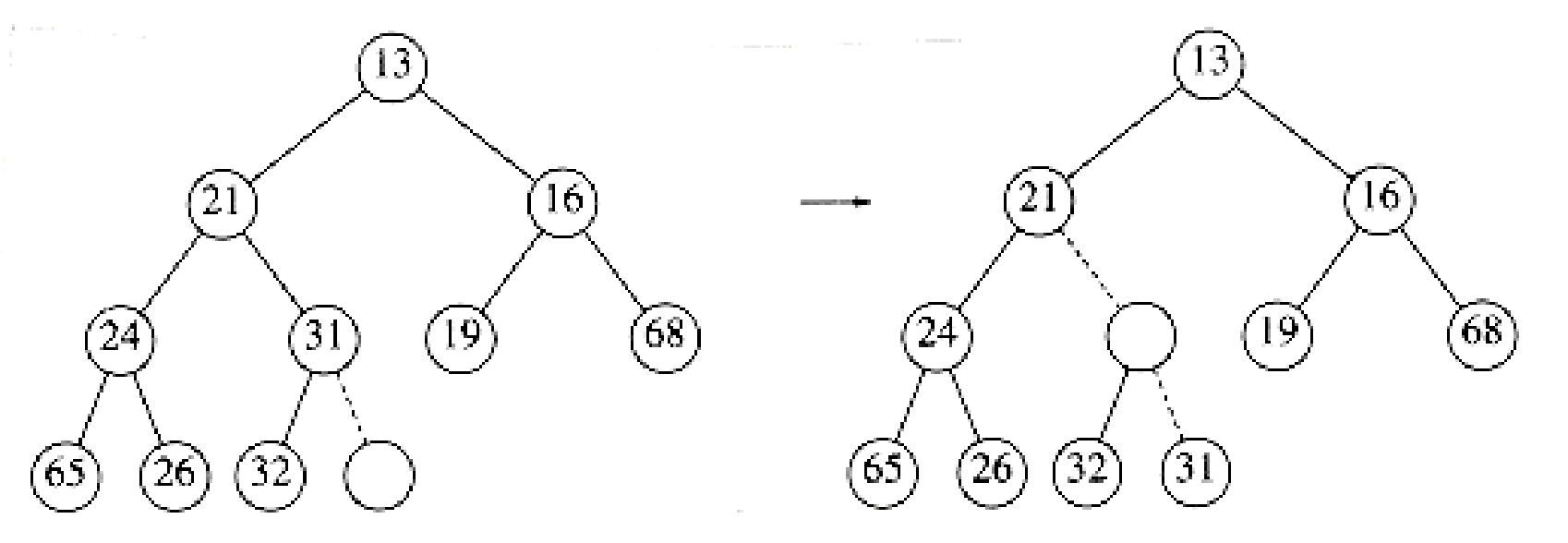
To insert an element x into the heap, we create a hole in the next available

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 3d. Our method uses d + 1 assignments.

****

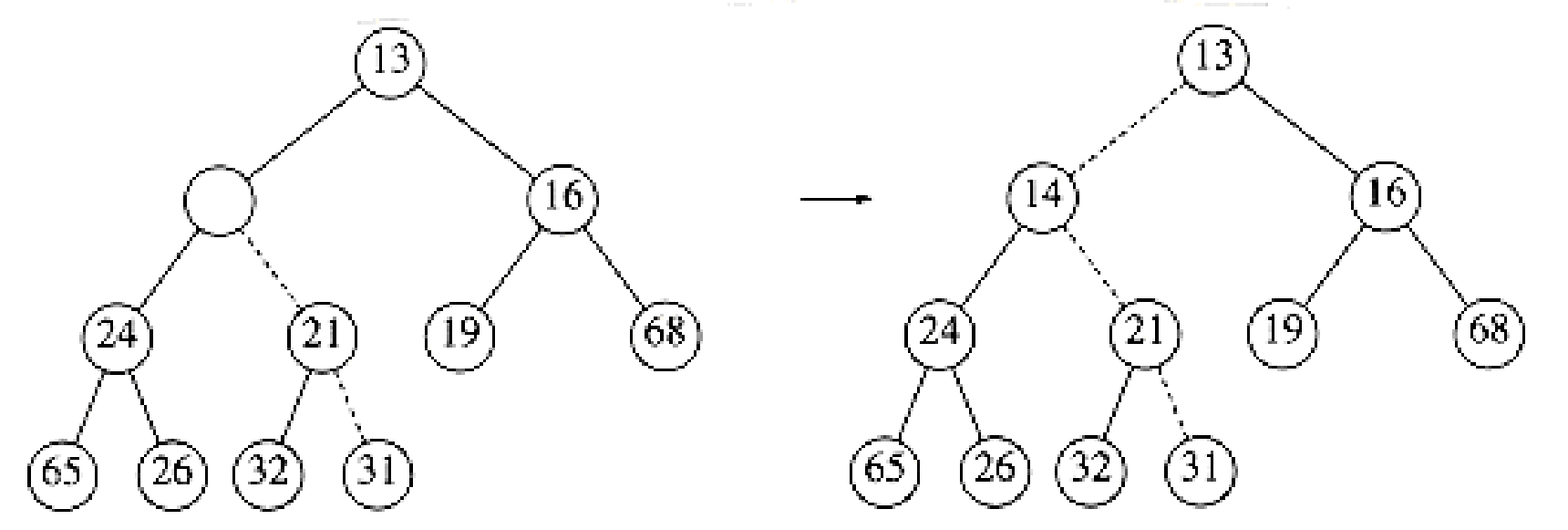
****

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.

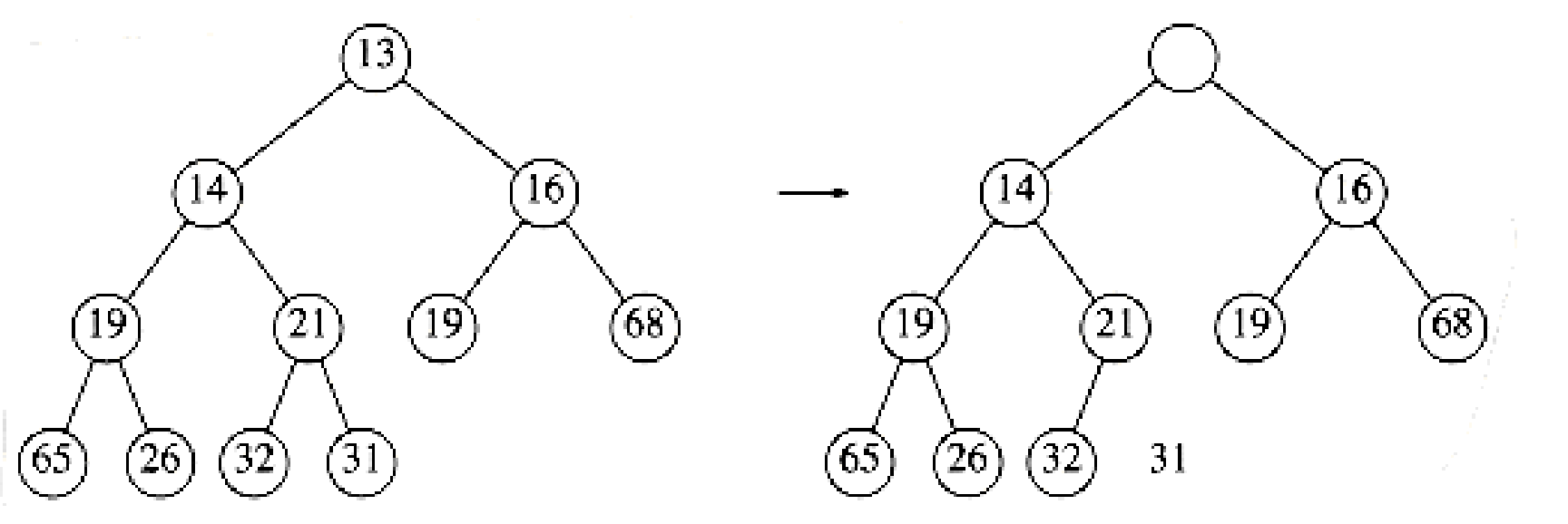
The time to do the insertion could be as much as O (log n), if the element to be inserted is the new minimum and is percolated all the way to the root. On

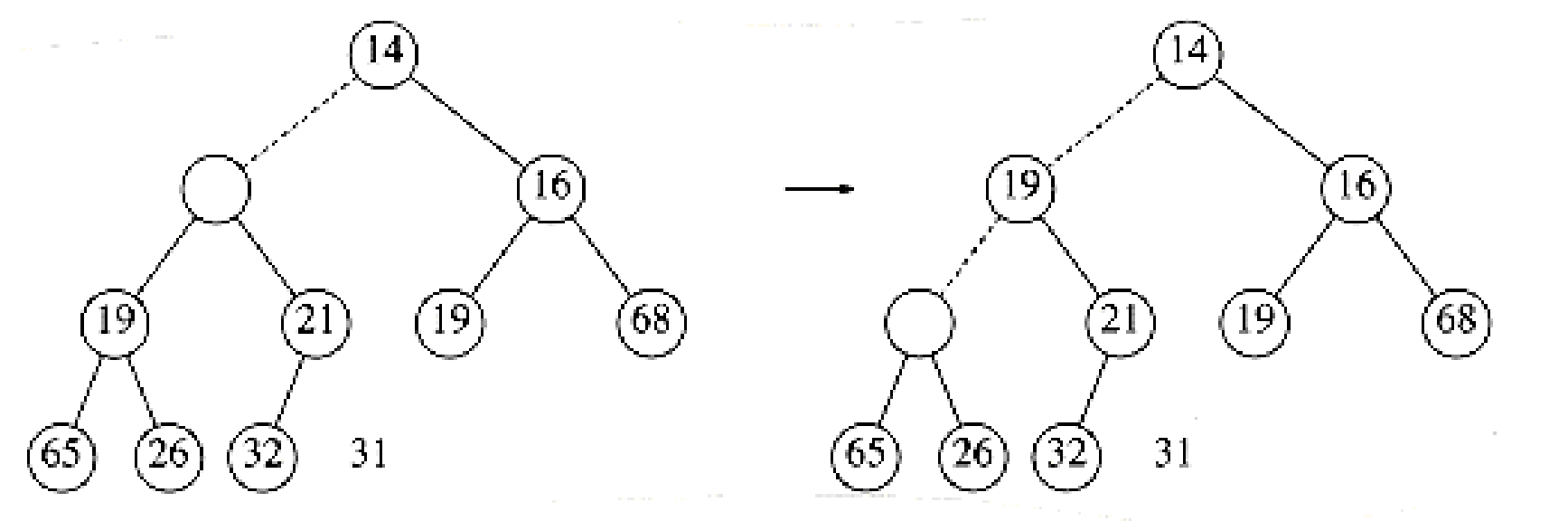
average, the percolation terminates early; it has been shown that 2.607 comparisons are required on average to perform an insert, so the average insert moves an element up 1.607 levels.

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.

****

****

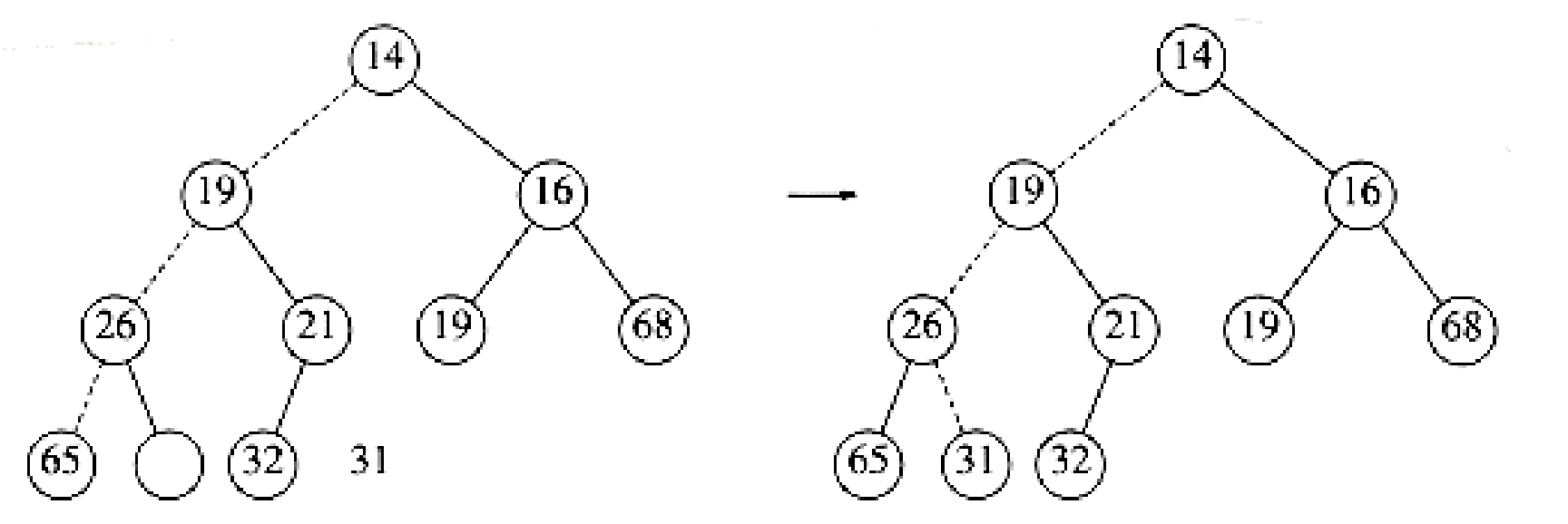
****

Figure 6.11 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 ecause 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, , H) operation lowers the value of the key at position x

by a positive amount . 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, , H) operation increases the value of the key at position

x by a positive amount . 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.

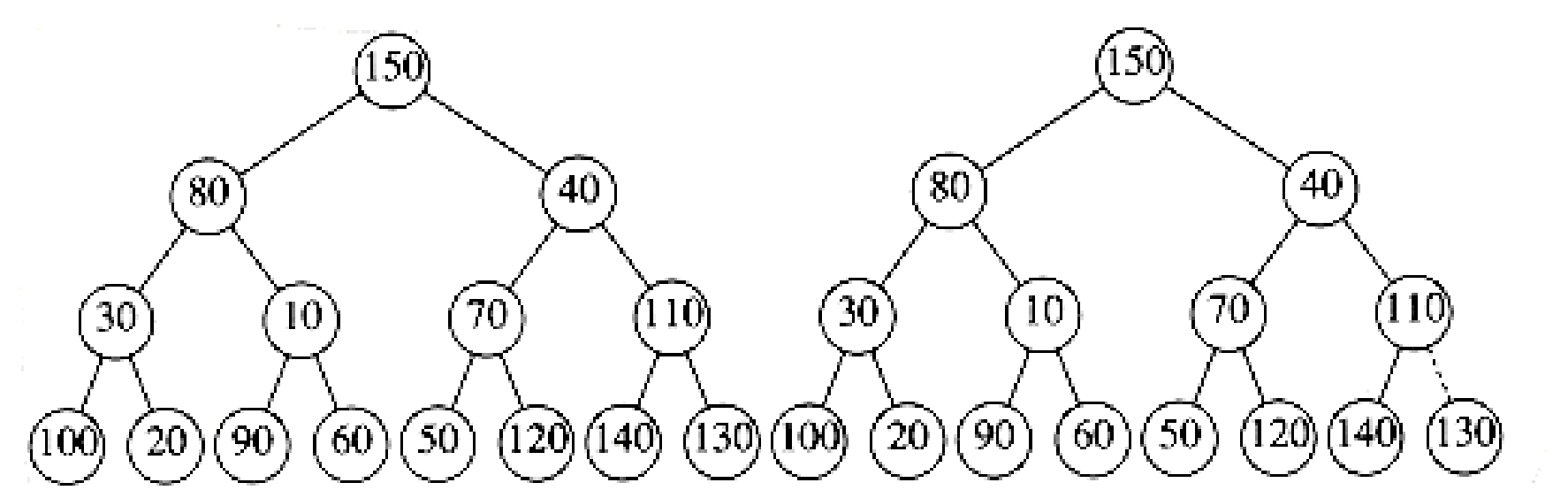
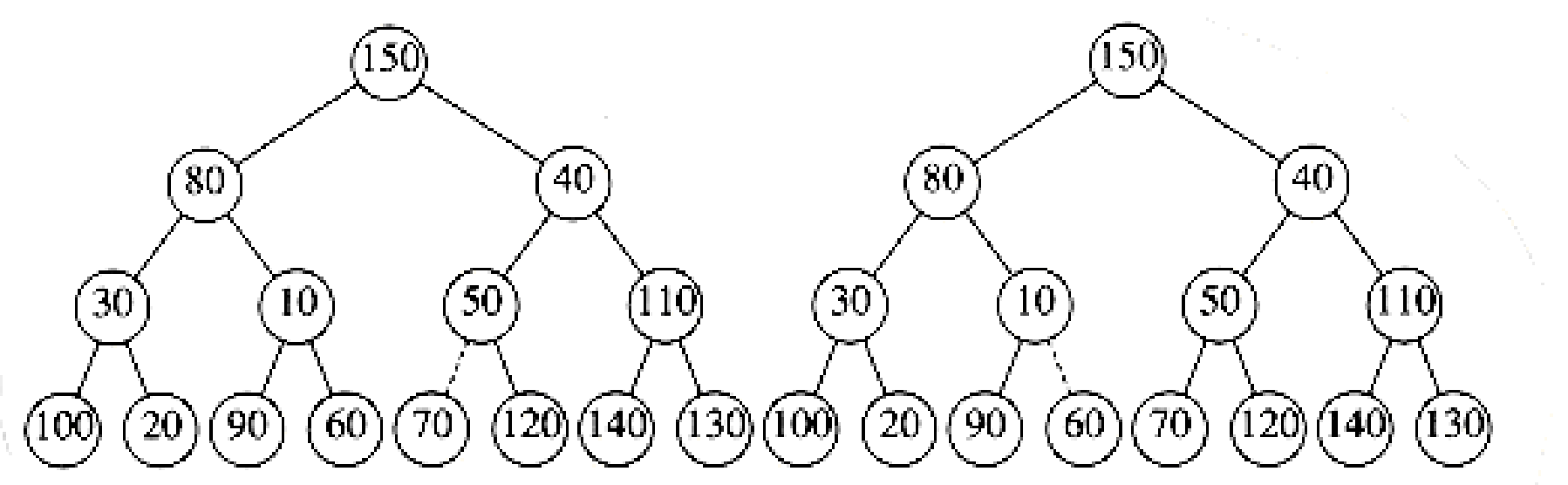
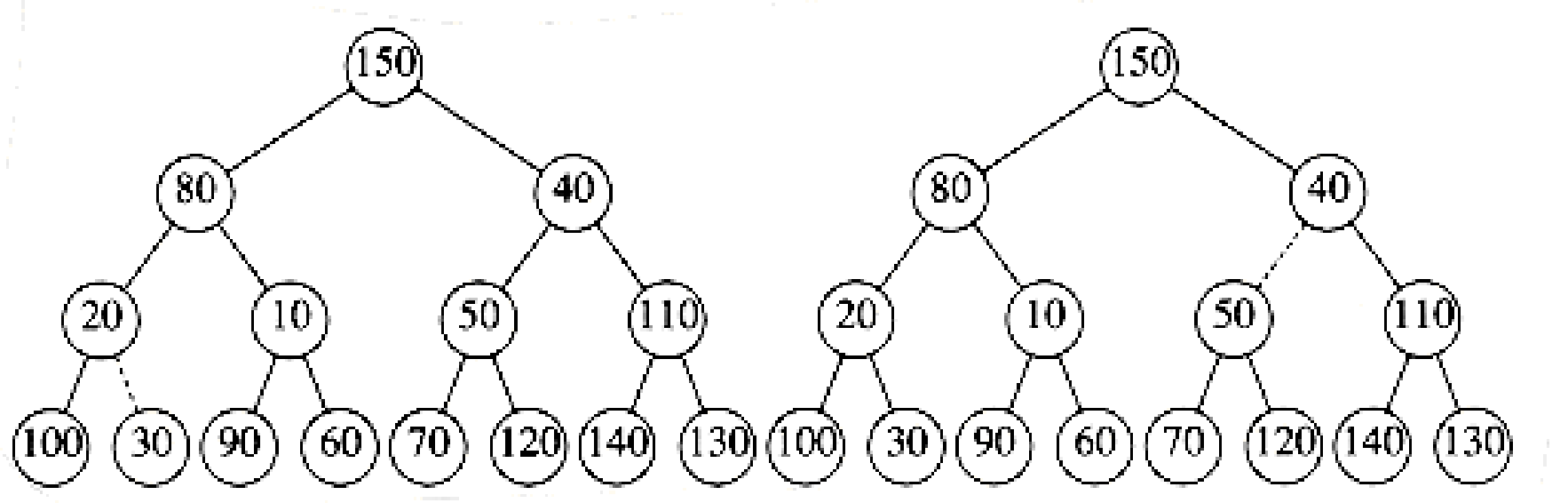
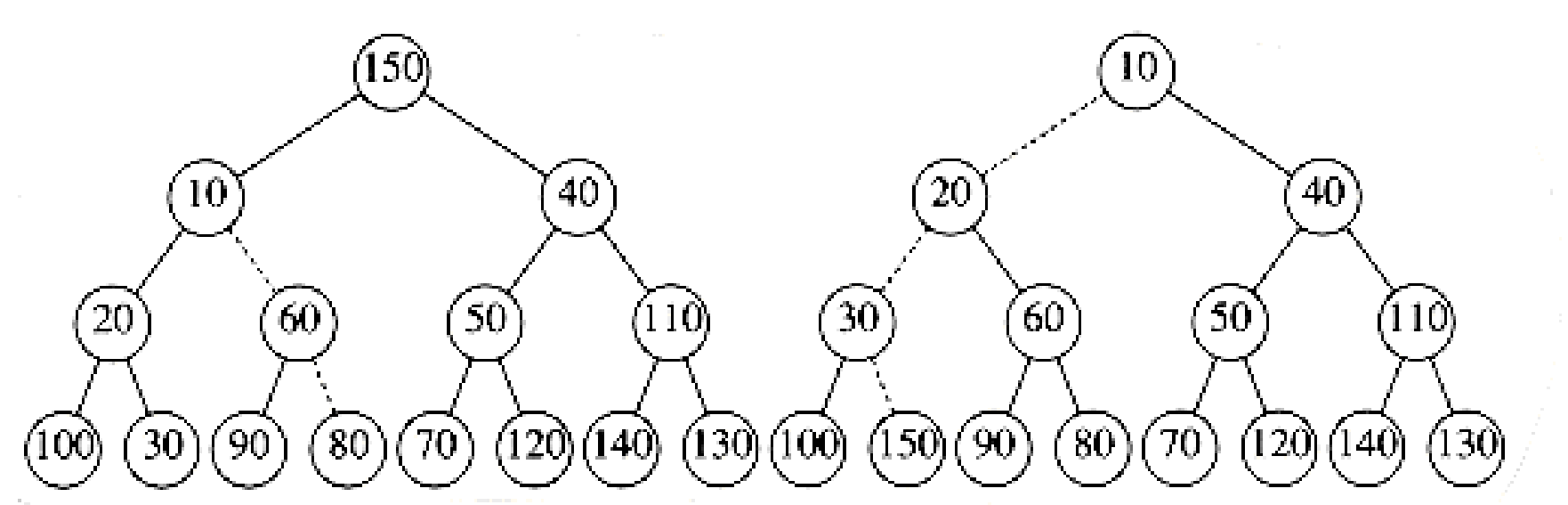
The general algorithm is to place the n keys into the tree in any order,

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 6.18 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.

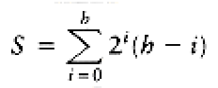
For the perfect binary tree of height h containing 2h+1 - 1 nodes, the sum of the heights of the nodes is 2h+1 - 1 - (h + 1).

PROOF:

It is easy to see that this tree consists of 1 node at height h, 2 nodes at

height h - 1, 22 nodes at height h - 2, and in general 2i nodes at height h - i.

The sum of the heights of all the nodes is then

****

= h +2(h - 1) + 4(h - 2) + 8(h - 3) + 16(h - 4) +. . .+ 2h-1(1)

(6.1)

Multiplying by 2 gives the equation

2S = 2h + 4(h - 1) + 8(h - 2) + 16(h - 3) + . . . + 2h(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), 2h, 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 + . . . + 2h-1 + 2h = (2h+1 - 1) - (h + 1)

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 2h and 2h+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 = 2h 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.

**Priority Queue ADT**

The following operations make priority queues an ADT.

**Main Priority Queues Operations**

A priority queue is a container of elements, each having an associated key.

• Insert (key, data): Inserts data with *key* to the priority queue. Elements are ordered based on key.

• DeleteMin/DeleteMax: Remove and return the element with the smallest/largest key.

• GetMinimum/GetMaximum: Return the element with the smallest/largest key without deleting it.

**Auxiliary Priority Queues Operations**

• *kth -* Smallest/*kth* – Largest: Returns the *kth* -Smallest/*kth –*Largest key in priority queue.

• Size: Returns number of elements in priority queue.

• Heap Sort: Sorts the elements in the priority queue based on priority (key).

**Priority Queue Applications**

Priority queues have many applications - a few of them are listed below:

• Data compression: Huffman Coding algorithm

• Shortest path algorithms: Dijkstra’s algorithm

• Minimum spanning tree algorithms: Prim’s algorithm

• Event-driven simulation: customers in a line

• Selection problem: Finding *kth-* smallest element

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 Algorit hm, 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(n2). 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 k) (why?). If k = *n*/2

, then both algorithms are O(n2). Notice that for any k, we can solve the

symmetric problem of finding the (n - k + 1)th smallest element, so k = *n*/2

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 = *n*/2 , 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 = *n*/2 , then the running time is (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.

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 Sk. Notice that Sk is the smallest element in S. If the new element is larger, then it replaces Sk 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 Sk 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 (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.

The problem with this simulation strategy is that its running time does not

depend on the number of customers or events (there are two events per customer), but instead depends on the number of ticks, which is not really part of the input. To see why this is important, suppose we changed the clock units to milliticks and multiplied all the times in the input by 1,000. The result would be that the simulation would take 1,000 times longer!

The key to avoiding this problem is to advance the clock to the next event time at each stage . This is conceptually easy to do. At any point, the next event that can occur is either (a) the next customer in the input file arrives, or (b) one of the customers at a teller leaves. Since all the times when the events will happen are available, we just need to find the event that happens nearest in the future and process that event.

If the event is a departure, processing includes gathering statistics for the

departing customer and checking the line (queue) to see whether there is another customer waiting. If so, we add that customer, process whatever statistics are required, compute the time when that customer will leave, and add that departure to the set of events waiting to happen.

If the event is an arrival, we check for an available teller. If there is none,

we place the arrival on the line (queue); otherwise we give the customer a

teller, compute the customer's departure time, and add the departure to the set of events waiting to happen.

The waiting line for customers can be implemented as a queue. Since we need to

find the event nearest in the future, it is appropriate that the set of

departures waiting to happen be organized in a priority queue. The next event is thus the next arrival or next departure (whichever is sooner); both are easily available.

It is then straightforward, although possibly time-consuming, to write the

simulation routines. If there are C customers (and thus 2C events) and k tellers, then the running time of the simulation would be O(C log(k + 1))\* because computing and processing each event takes O(logH), where H = k + 1 is the size of the heap.

\* We use O(C log( k + 1)) instead of O(C log k) to avoid confusion for the k = 1 case.

**Priority Queue Implementations**

Before discussing the actual implementation, let us enumerate the possible options.

**Unordered Array Implementation**

Elements are inserted into the array without bothering about the order. Deletions (DeleteMax) are

performed by searching the key and then deleting.

Insertions complexity: O(1). DeleteMin complexity: O(*n*).

**Unordered List Implementation**

It is very similar to array implementation, but instead of using arrays, linked lists are used.

Insertions complexity: O(1). DeleteMin complexity: O(*n*).

**Ordered Array Implementation**

Elements are inserted into the array in sorted order based on key field. Deletions are performed at

only one end.

Insertions complexity: O(*n*). DeleteMin complexity: O(1).

**Ordered List Implementation**

Elements are inserted into the list in sorted order based on key field. Deletions are performed at

only one end, hence preserving the status of the priority queue. All other functionalities associated

with a linked list ADT are performed without modification.

Insertions complexity: O(*n*). DeleteMin complexity: O(1).

**Binary Search Trees Implementation**

Both insertions and deletions take O(*logn*) on average if insertions are random (refer to *Trees*

chapter).

**Balanced Binary Search Trees Implementation**

Both insertions and deletion take O(*logn*) in the worst case (refer to *Trees* chapter).

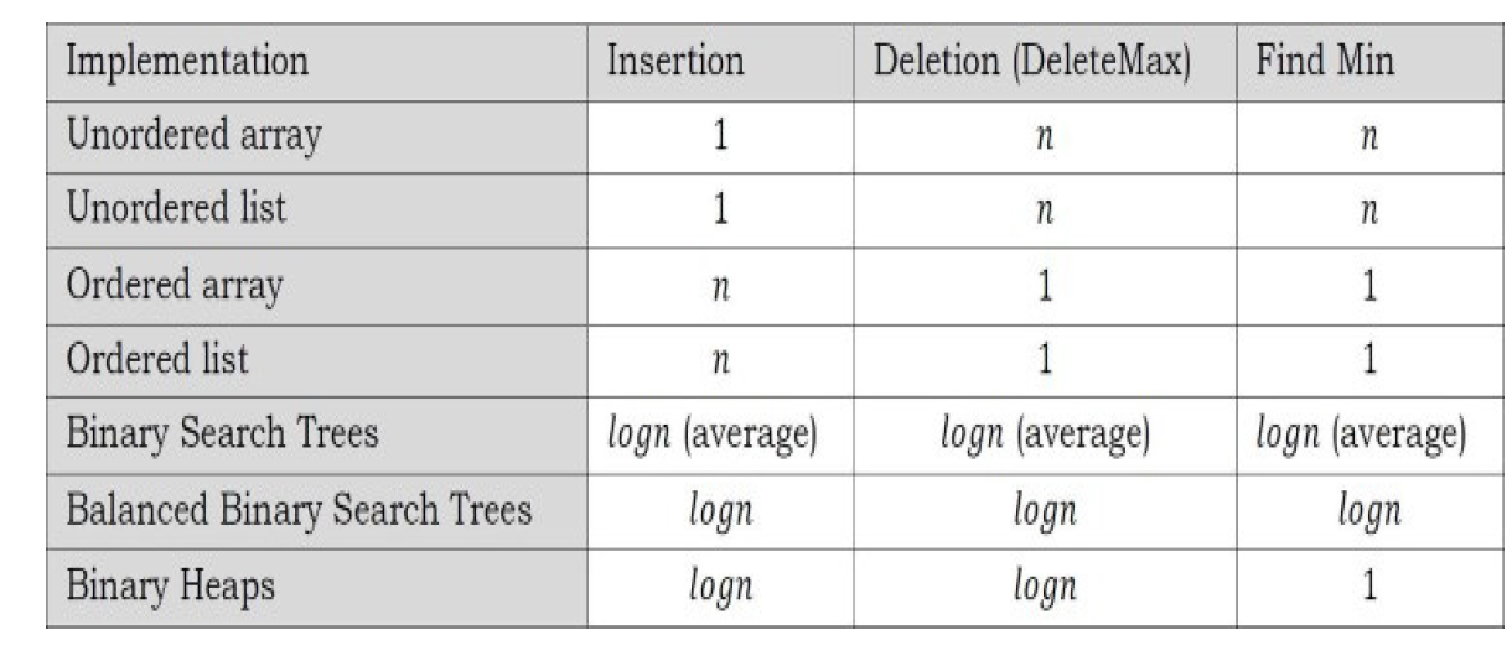
**Binary Heap Implementation**

In subsequent sections we will discuss this in full detail. For now, assume that binary heap

implementation gives O(*logn*) complexity for search, insertions and deletions and O(1) for

finding the maximum or minimum element.

**Comparing Implementations**

**7.5**

**Heaps and Binary Heaps**

**What is a Heap?**

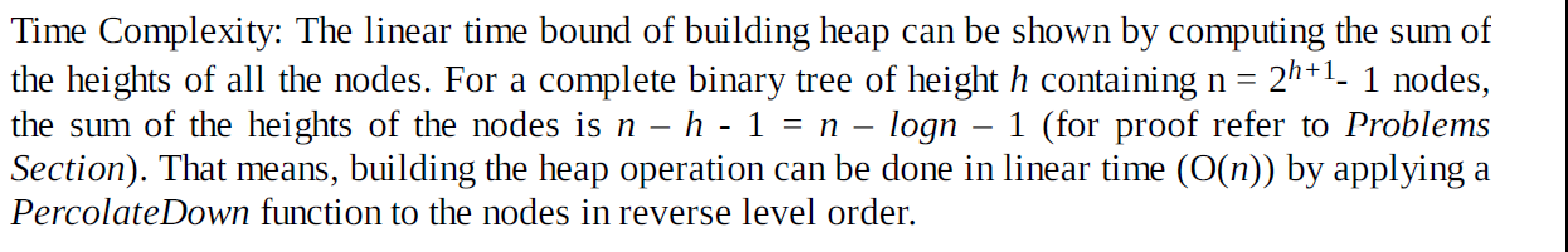
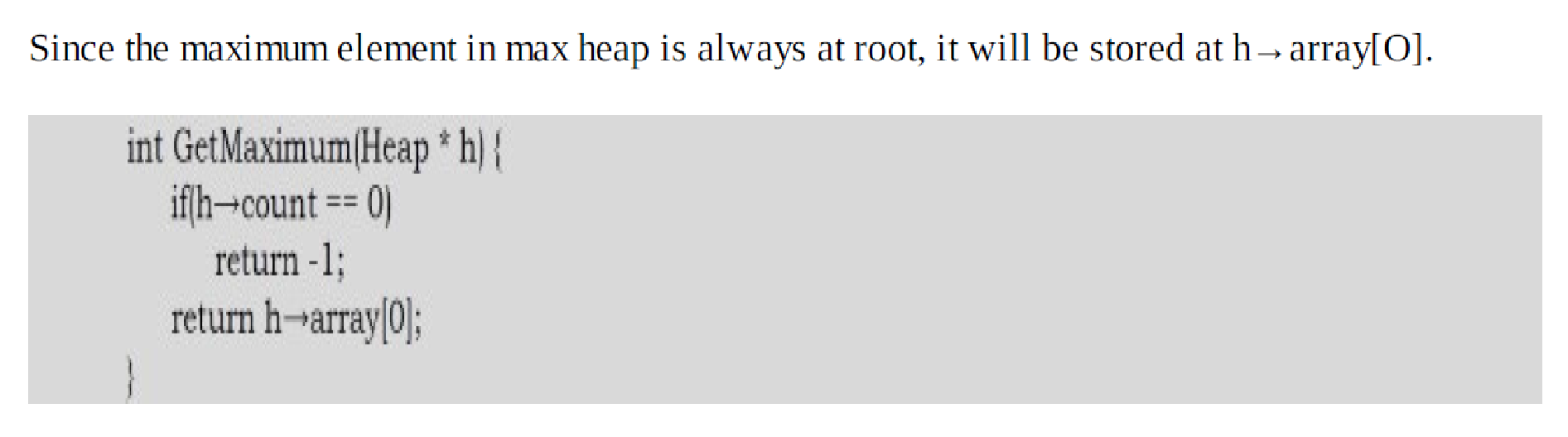
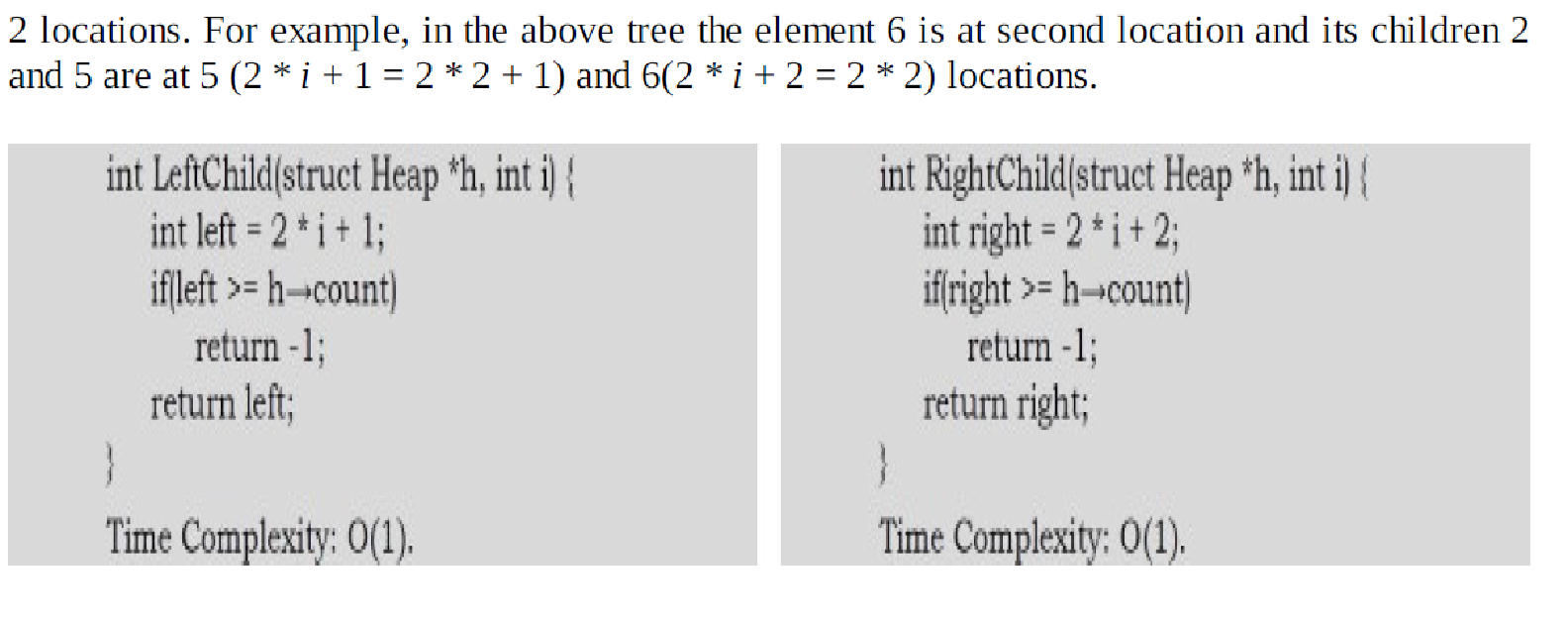
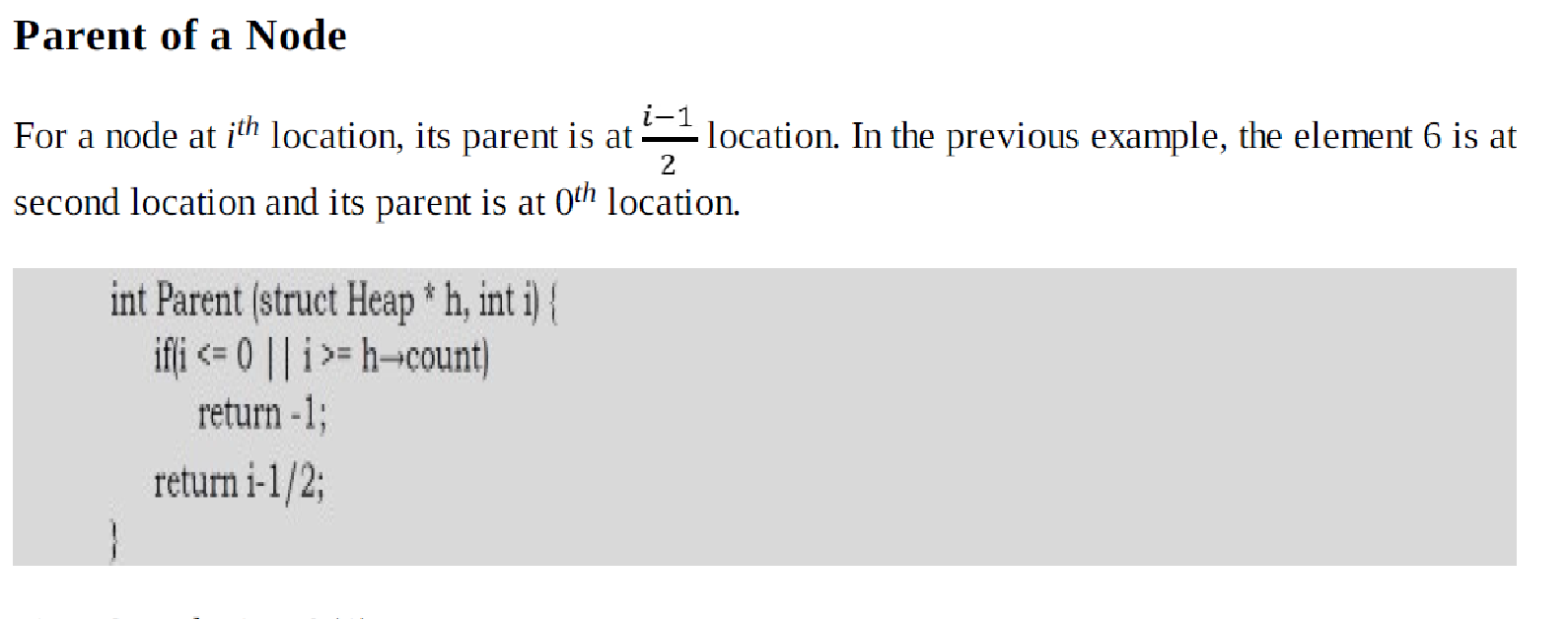
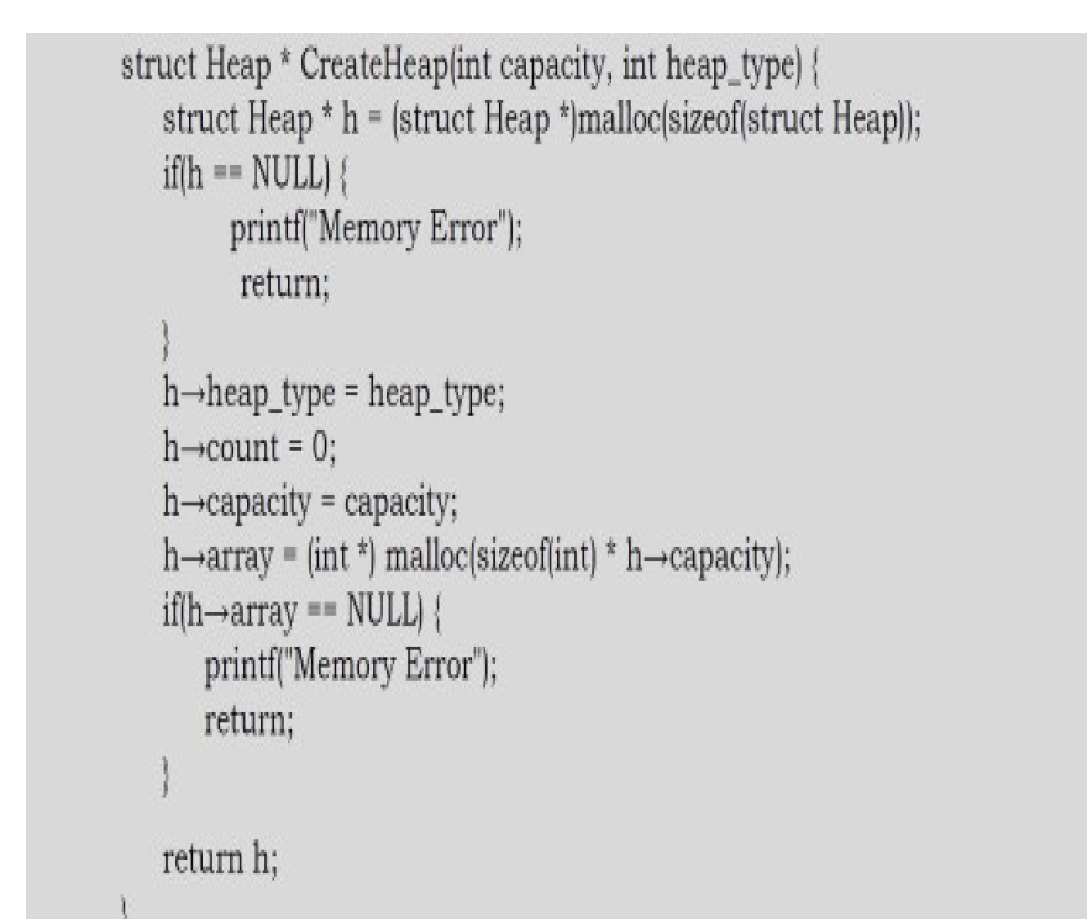
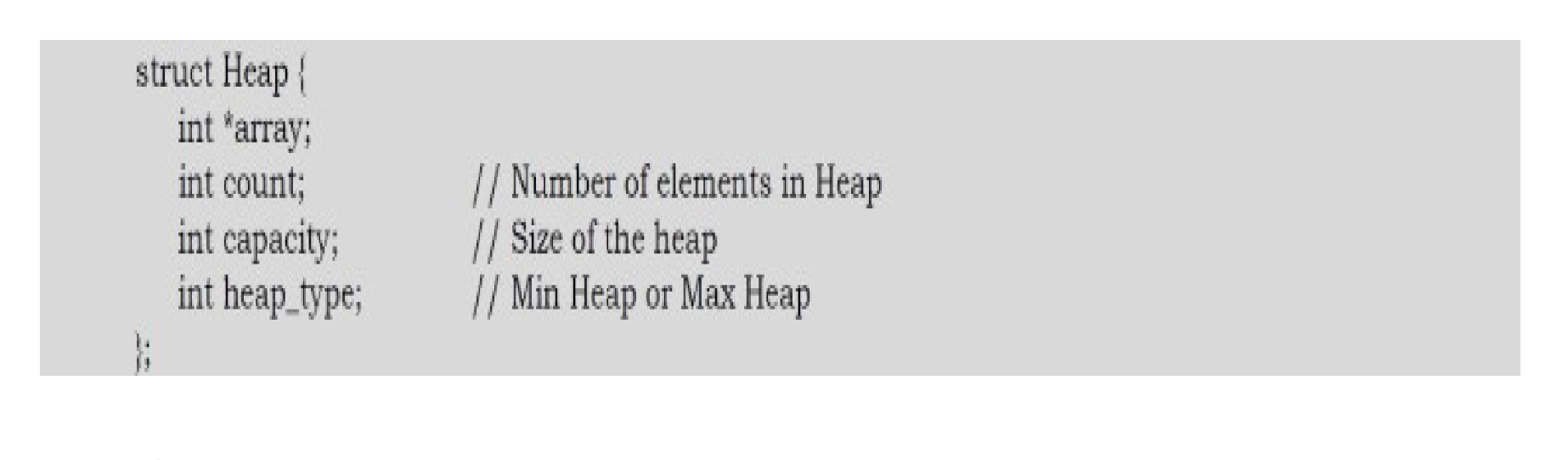
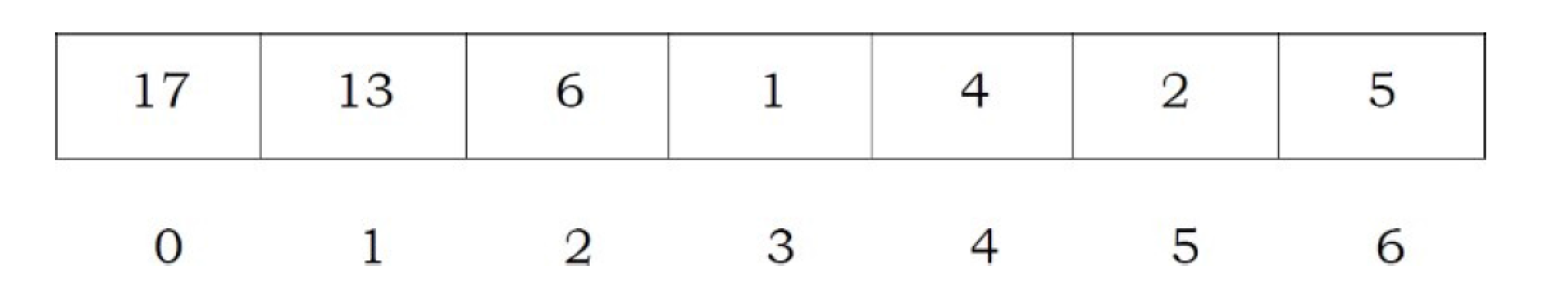
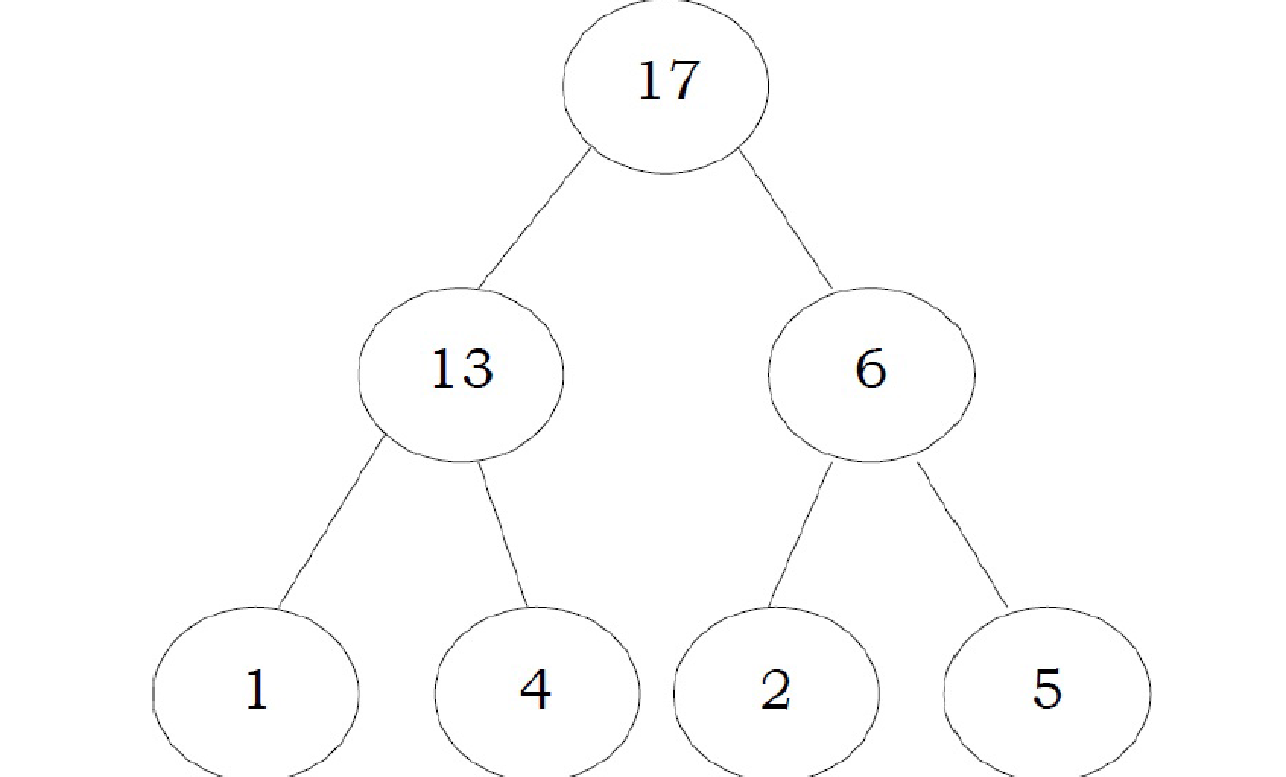
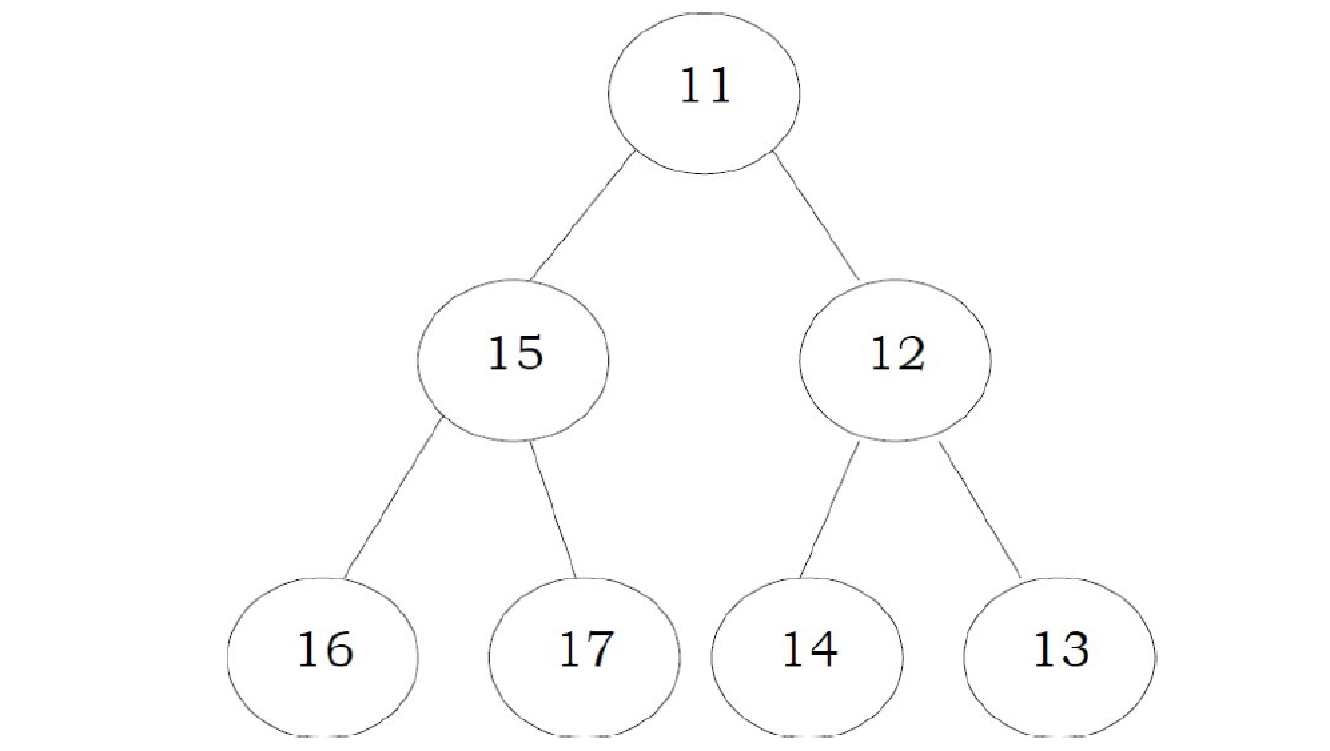
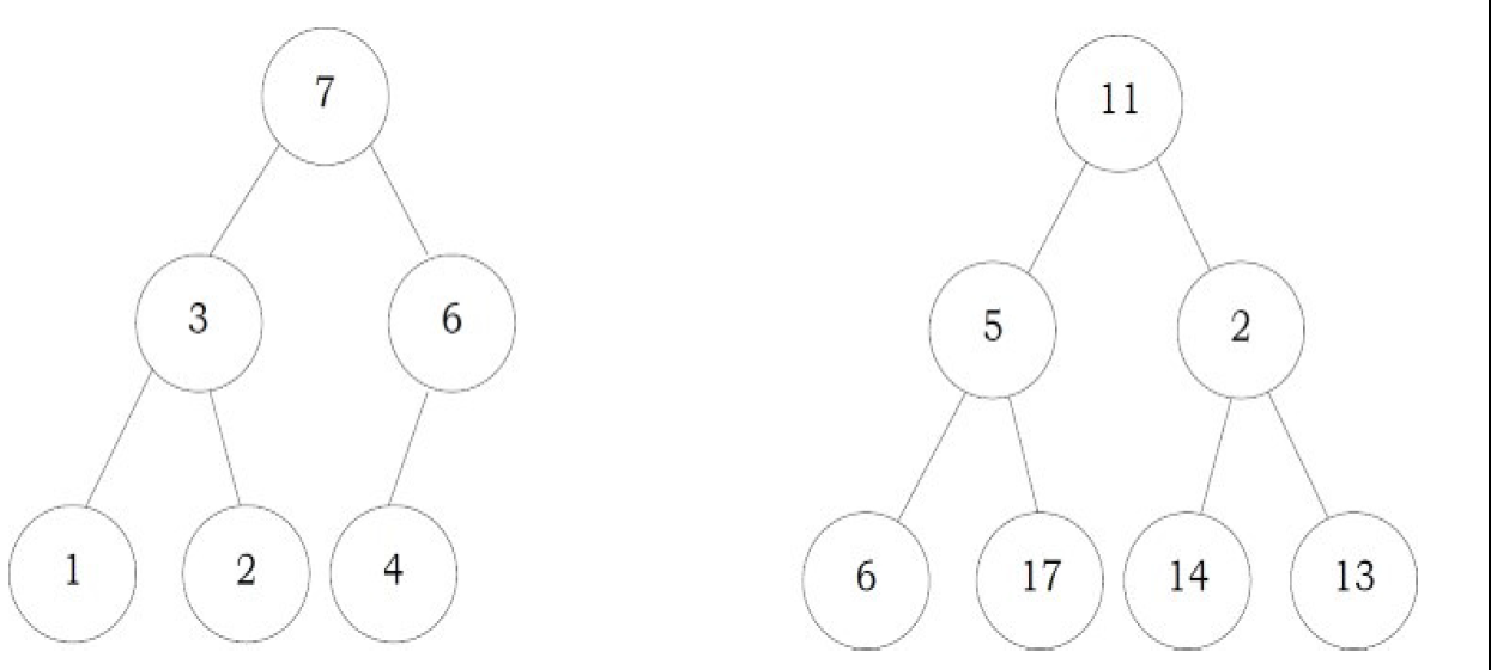
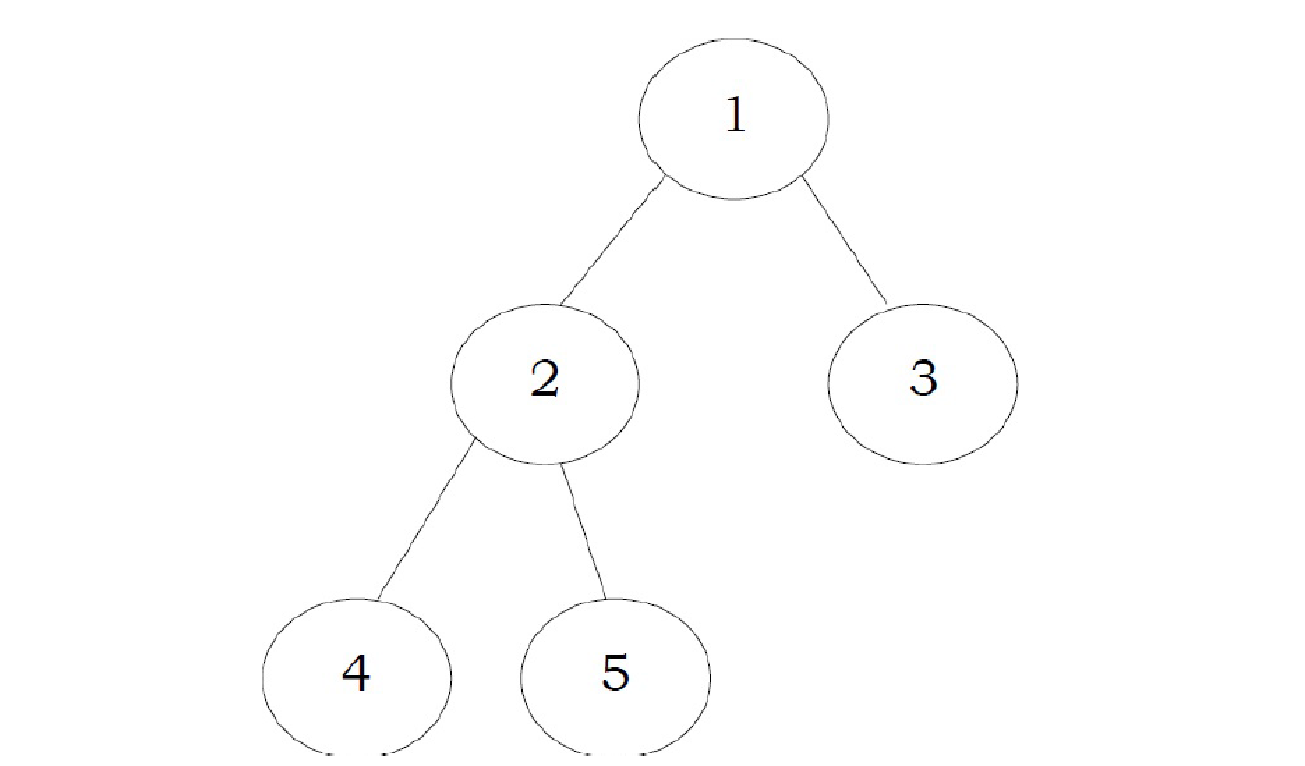
A heap is a tree with some special properties. The basic requirement of a heap is that the value of

a node must be ≥ (or ≤) than the values of its children. This is called *heap property*. A heap also

has the additional property that all leaves should be at *h* or *h* – 1 levels (where *h* is the height of

the tree) for some *h >* 0 (*complete binary trees*). That means heap should form a *complete binary*

*tree* (as shown below).



6.8. Binomial Queues

Although both leftist and skew heaps support merging, insertion, and delete\_min 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.

< P>

6.8.1. 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 are of a constrained form known as a binomial tree (the

There is at most one binomial tree of every height. A binomial tree of height 0 is a one-node tree; a binomial tree, Bk, of height k is formed by attaching a binomial tree, Bk-1, to the root of another binomial tree, Bk-1. Figure 6.34 shows binomial trees B0, B1, B2, B3, and B4.

It is probably obvious from the diagram that a binomial tree, Bk consists of a root with children

B0, B1, . . ., Bk-1. Binomial trees of height k have exactly 2k nodes, and the number of nodes at

depth d is the binomial coefficient . If we impose heap order on the binomial trees and allow

at most one binomial tree of any height, we can uniquely represent a priority queue of any size

by a collection of binomial trees. For instance, a priority queue of size 13 could be represented

by the forest B3, B2, B0. We might write this representation as 1101, which not only represents

13 in binary but also represents the fact that B3, B2 and B0 are present in the representation

and B1 is not As an example, a priority queue of six elements could be represented as in Figure 6.35.

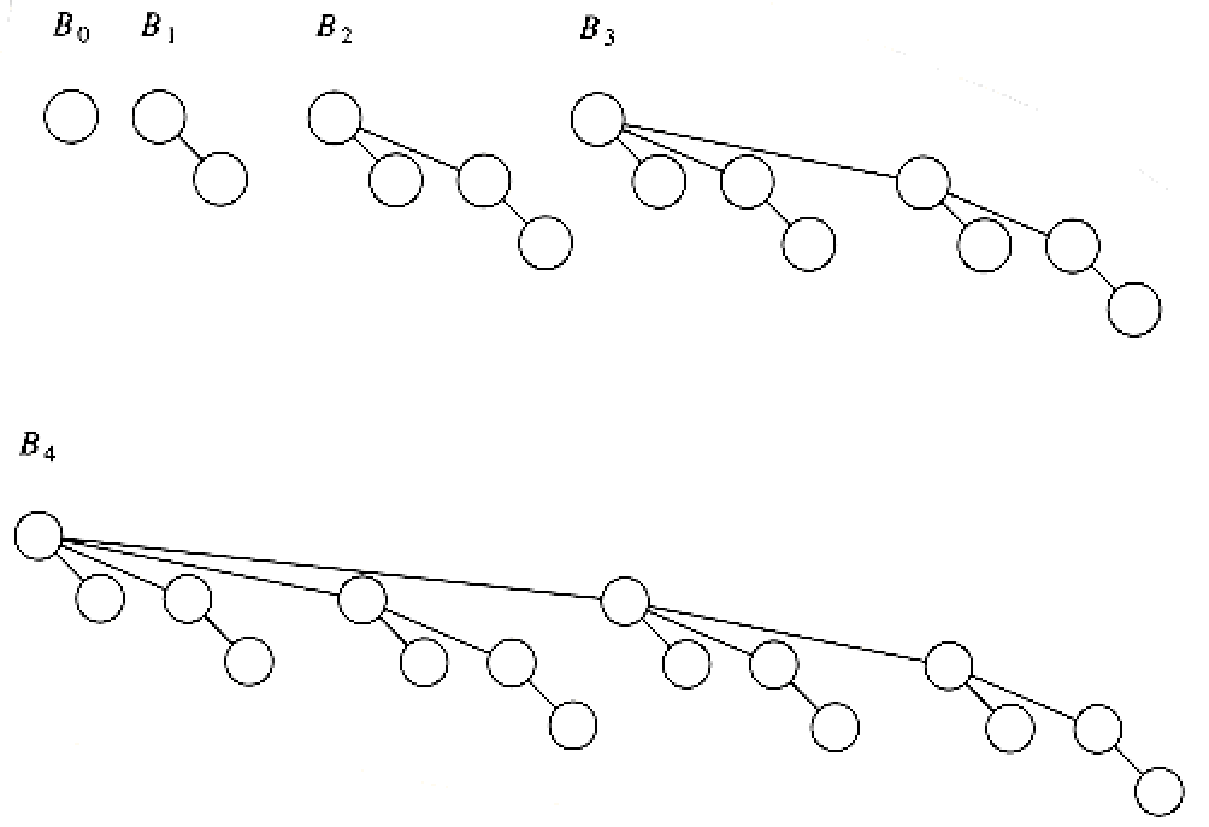
 Figure 6.34 Binomial trees B0**, B1, B2, B3, and B4**

Figure 6.35 Binomial queue H1 **with six elements**

6.8.2. Binomial

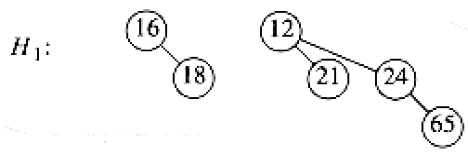


Figure 6.35 Binomial queue H1 **with six elements**

6.8.2. Binomial Queue Operations

The minimum element can then be found by scanning the roots of all the trees. Since there are at

most log n different trees, the minimum can be found in O(log n) time. Alternatively, we can

maintain knowledge of the minimum and perform the operation in O(1) time, if we remember to

update the minimum when it changes during other operations. Merging two binomial queues is a conceptually easy operation, which we will describe by example. Consider the two binomial queues, H1 and H2 with six and seven elements, respectively, pictured in Figure 6.36.

The merge is performed by essentially adding the two queues toge ther. Let H3 be the new binomial queue. Since H1 has no binomial tree of height 0 and H2 does, we can just use the binomial tree of height 0 in H2 as part of H3. Next, we add binomial trees of height 1. Since both H1 and H2 have binomial trees of height 1, we merge them by making the larger root a subtree of the smaller, creating a binomial tree of height 2, shown in Figure 6.37. Thus, H3 will not have a binomial tree of height 1. There are now three binomial trees of height 2, namely, the original

trees of H1 and H2 plus the tree formed by the previous step. We keep one binomial tree of height 2 in H3 and merge the other two, creating a binomial tree of height 3. Since H1 and H2 have no trees of height 3, this tree becomes part of H3 and we are finished. The resulting binomial queue is shown in Figure 6.38.

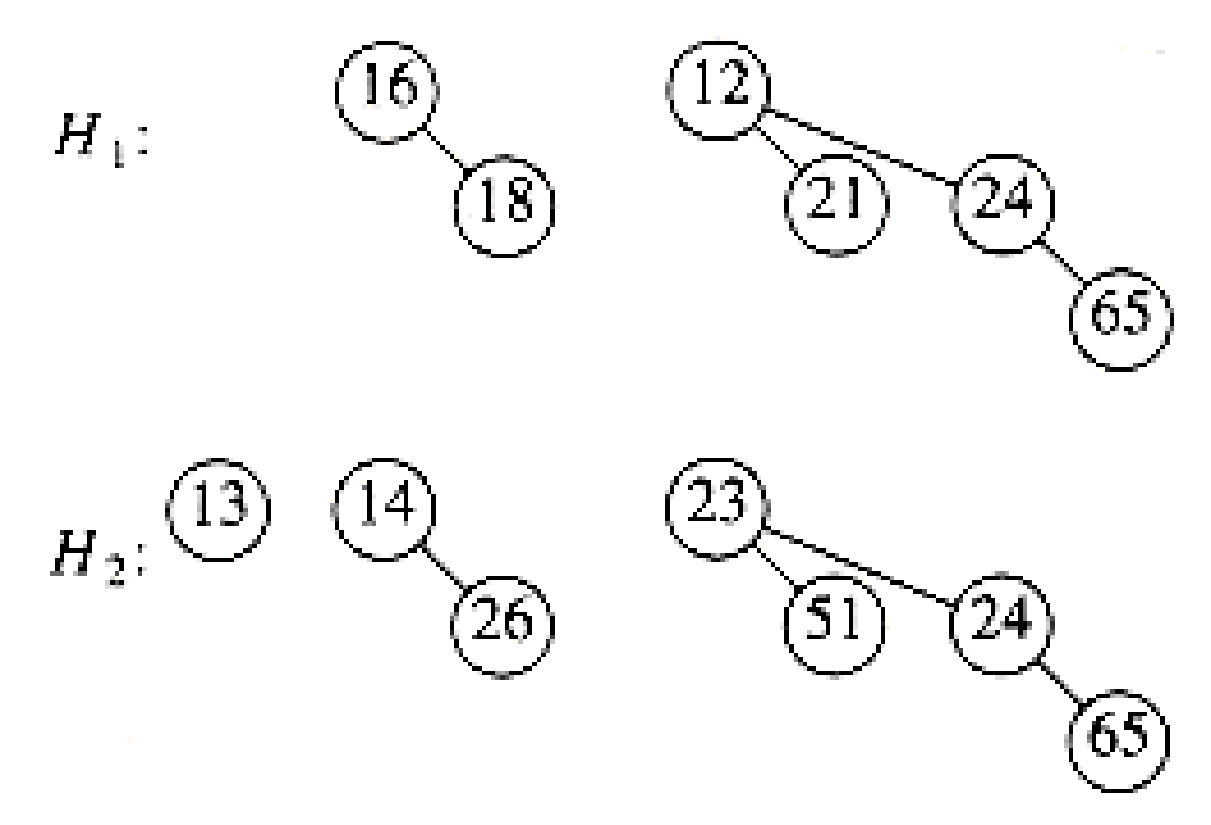


Figure 6.36 Two binomial queues H1 **and H2**

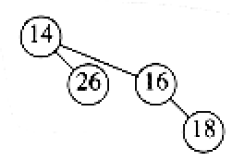
****

Figure 6.37 Merge of the two B1 **trees in H1 and H2**

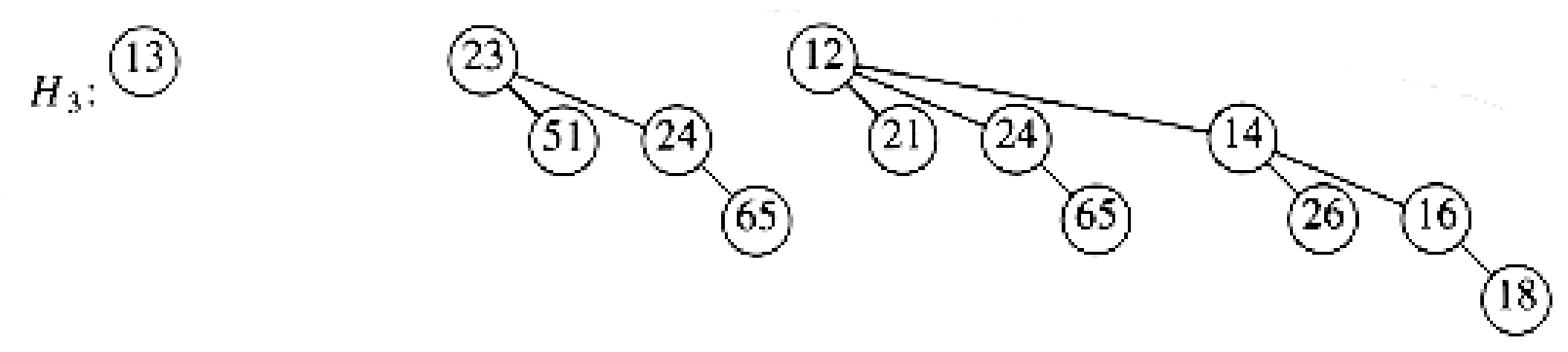
****

Figure 6.38 Binomial queue H3**: the result of merging H1 and H2**

Since merging two binomial trees takes constant time with almost any reasonable implementation, and there are O(log n) binomial trees, the merge takes O(log n) time in the worst case. To make this operation efficient, we need to keep the trees in the binomial queue sorted by height, which is certainly a simple thing to do. Insertion is just a special case of merging, since we merely create a one-node tree and perform a merge. The worst-case time of this operation is likewise O(log n). More precisely, if the priority queue into which the element is being inserted has the property that the smallest nonexistent binomial tree is Bi, the running time is proportional to i + 1. For example, H3 (Fig. 6.38) is missing a binomial tree of height 1, so the insertion will terminate in two steps. Since each tree in a binomial queue is present with probability , it follows that we expect an insertion to terminate in two steps, so the average time is constant. Furthermore, an easy analysis will show that performing n inserts on an initially empty binomial queue will take O(n) worst-case time. Indeed, it is possible to do this operation using only n - 1 comparisons; we leave this as an exercise. As an example, we show in Figures 6.39 through 6.45 the binomial queues that are formed by inserting 1 through 7 in order. Inserting 4 shows off a bad case. We merge 4 with B0, obtaining a new tree of height 1. We then merge this tree with B1, obtaining a tree of height 2, which is the new priority queue. We count this as three steps (two tree merges plus the stopping case). The next insertion after 7 is inserted is another bad case and would require three tree merges. A delete\_min can be performed by first finding the binomial tree with the smallest root. Let this tree be Bk, and let the original priority queue be H. We remove the binomial tree Bk from the forest of trees in H, forming the new binomial queue H'. We also remove the root of Bk, creating binomial trees B0, B1, . . . , Bk - l, which collectively form priority queue H''. We finish the operation by merging H' and H''.

As an example, suppose we perform a delete\_min on H3, which is shown again in

Figure 6.46. The minimum root is 12, so we obtain the two priority queues H' and H'' in Figure 6.47 and Figure 6.48. The binomial queue that results from merging H' and H'' is the final answer and is shown in Figure 6.49. For the analysis, note first that the delete\_min operation breaks the original binomial queue into two. It takes O (log n) time to find the tree containing the minimum element and to create the queues H' and H''. Merging these two queues takes O (log n) time, so the entire delete\_min operation takes O (log n) time.

6.8.3. Implementation of Binomial Queues

The delete\_min operation requires the ability to find all the subtrees of the root quickly, so

the standard representation of general trees is required: The children of each node are kept in a

linked list, and each node has a pointer to its first child (if any). This operation also requires that the children be ordered by the size of their subtrees, in essentially the same way as we have been drawing them. The reason for this is that when a delete\_min is performed, the children will form the binomial queue H''.

We also need to make sure that it is easy to merge two trees. Two binomial trees can be merged

only if they have the same size, so if this is to be done efficiently, the size of the tree must

be stored in the root. Also, when two trees are merged, one of the trees is added as a child to

the other. Since this new tree will be the last child (as it will be the largest subtree), we

must be able to keep track of the last child of each node efficiently. Only then will we be able

to merge two binomial trees, and thus two binomial queues, efficiently. One way to do this is to

use a circular doubly linked list. In this list, the left sibling of the first child will be the

last child. The right sibling of the last child could be defined as the first child, but it might

be easier just to define it as . This makes it easy to test whether the child we are pointing to

is the last. To summarize, then, each node in a binomial tree will contain the data, first child, left and right sibling, and the number of children (which we will call the rank). Since a binomial queu e is just a list of trees, we can use a pointer to the smallest tree as the reference to the data

structure. Figure 6.51 shows how the binomial queue in Figure 6.50 is represented. Figure 6.52 shows the type declarations for a node in the binomial tree.

In order to merge two binomial queues, we need a routine to merge two binomial trees of the same size. Figure 6.53 shows how the pointers change when two binomial trees are merged. First, the root of the new tree gains a child, so we must update its rank. We then need to change several pointers in order to splice one tree into the list of children of the root of the other tree. The code to do this is simple and shown in Figure 6.54.

****

**.** Figure 6.39 After 1 is inserted

Figure 6.40 After 2 is inserted

Figure 6.41

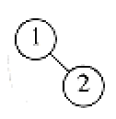
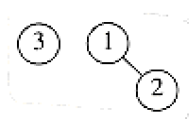
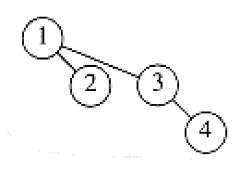
****

Figure 6.40 After 2 is inserted

Figure 6.41

Figure 6.41 After 3 is inserted

****

**** Figure 6.42 After 4 is inserted

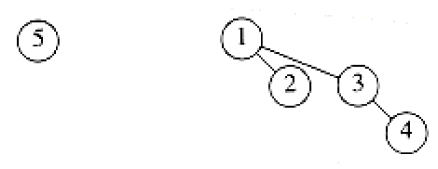
****

Figure 6.43 After 5 is inserted

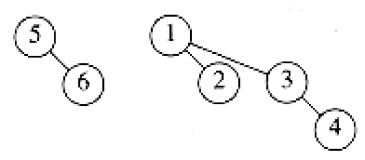
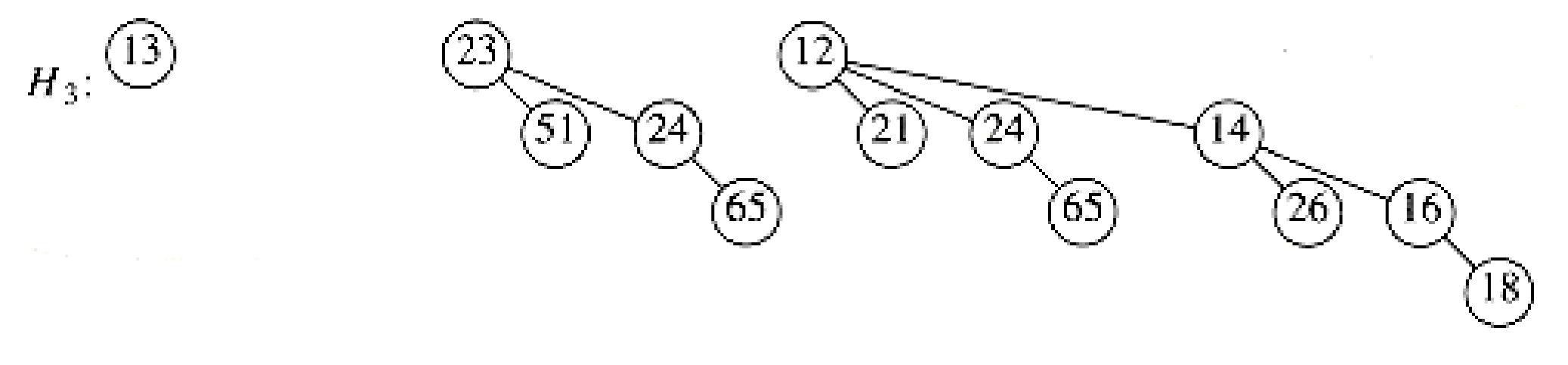
****

Figure 6.44 After 6 is inserted

****

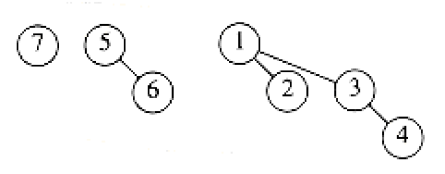


Figure 6.45 After 7 is inserted

****

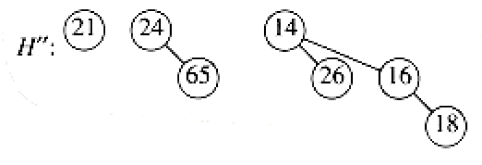
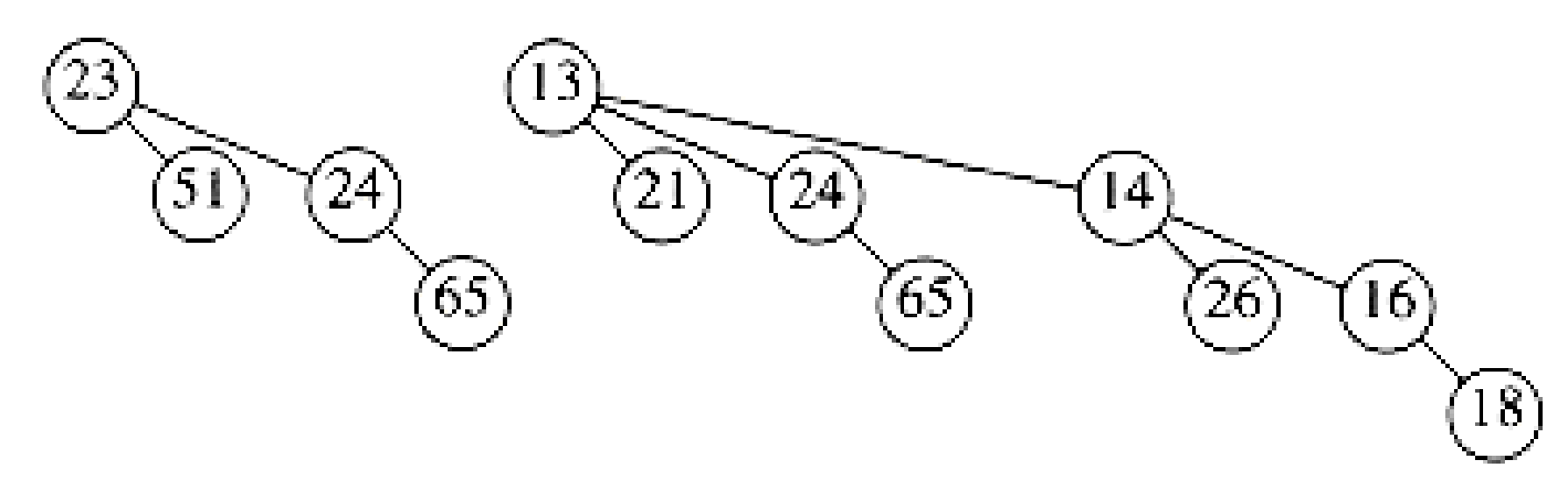
** **

Figure 6.46 Binomial queue H3

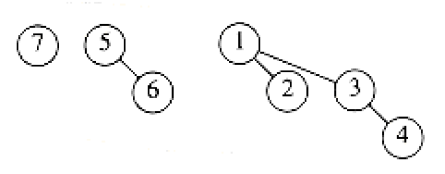
****

Figure 6.47 Binomial queue H', containing all the binomial trees in H3 **except B3**

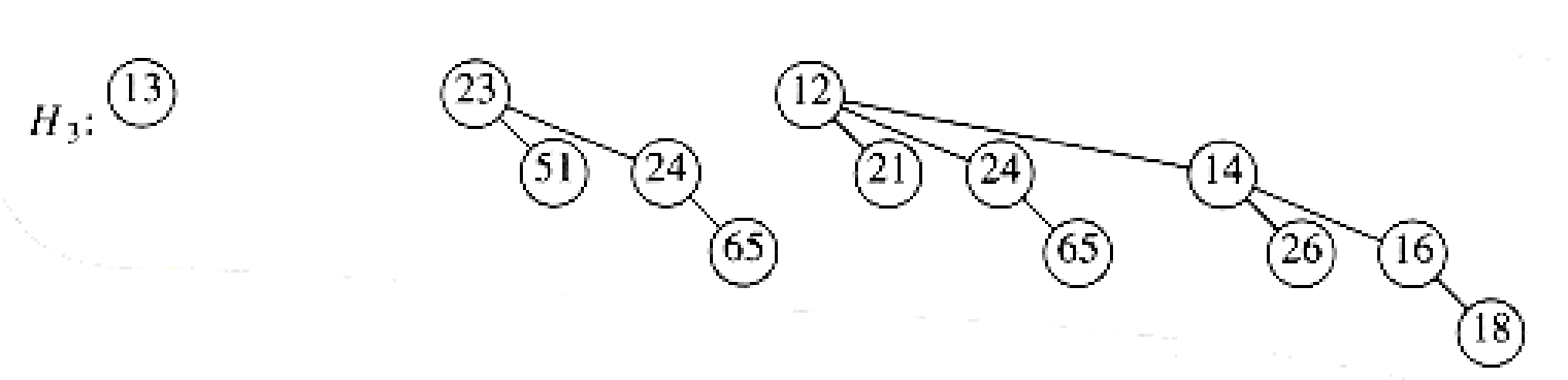
****

Figure 6.48 Binomial queue H'': B3 **with 12 removed**

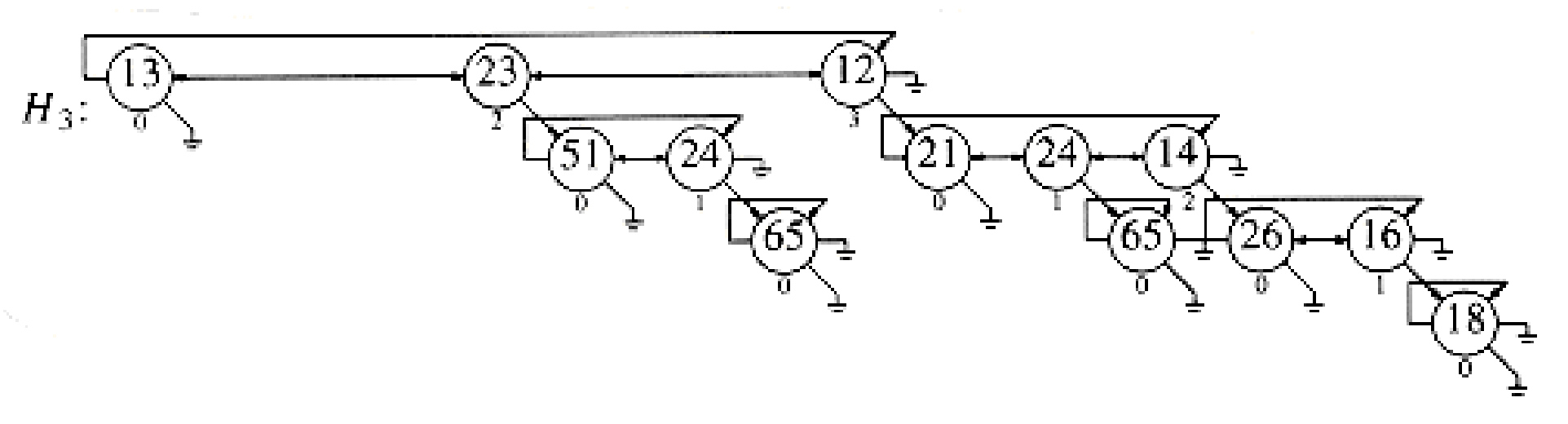
****

Figure 6.49 Result of delete\_min(H3**)**

****

Figure 6.50 Binomial queue H3 **drawn as a forest**

Figure 6.51

typedef struct tree\_node \*tree\_ptr;

struct tree\_node

{

element\_type element;

tree\_ptr l\_sib;

tree\_ptr r\_sib;

tree\_ptr f\_child;

unsigned int rank;

};

typedef tree\_ptr PRIORITY\_QUEUE;

Figure 6.52 Binomial queue type declarations



Figure 6.53 Merging two binomial trees

The routine to merge two binomial queues is relatively simple. We use recursion to keep the code

size small; a nonrecursive procedure will give better performance, and is left as

Exercise 6.32.

We assume the macro extract(T, H), which removes the first tree from the priority queue H,

placing the tree in T. Suppose the smallest binomial tree is contained in H1, but not in H2.

Then, to merge H1, we remove the first tree in H1 and add to it the result of merging the rest of

H1 with H2. If the smallest tree is contained in both Hl and H2, then we remove both trees and

merge them, obtaining a one-tree binomial queue H'. We then merge the remainder of Hl and H2,

and merge this result with H'. This strategy is implemented in Figure 6.55. The other routines

are straightforward implementations, which we leave as exercises.

/\* Merge two equal-sized binomial trees \*/

tree\_ptr

merge\_tree( tree\_ptr T1, tree\_ptr T2 )

{

if( T1->element > T2->element )

return merge\_tree( T2, T1 );

if( T1->rank++ == 0 )

T1->f\_child = T2;

else

{

T2->l\_sib = T1->f\_child->l\_sib;

T2->l\_sib->r\_sib = T2;

T1->f\_child->l\_sib = T2;

}

return T1;

}

Figure 6.54 Routine to merge two equal-sized binomial trees

We can extend binomial queues to support some of the nonstandard operations that binary heaps

allow, such as decrease\_key and delete, when the position of the affected element is known. A

decrease\_key is a percolate up, which can be performed in O(log n) time if we add a field to each

node pointing to its parent. An arbitrary delete can be performed by a combination of

decrease\_key and delete\_min in O(log n) time.

Summary

In this chapter we have seen various implementations and uses of the priority queue ADT. The

standard binary heap implementation is elegant because of its simplicity and speed. It requires

no pointers and only a constant amount of extra space, yet supports the priority queue operations

efficiently.

We considered the additional merge operation and developed three implementations, each of which is unique in its own way. The leftist heap is a wonderful example of the power of recursion. The skew heap represents a remarkable data s