5.1 Heaps

A ***(binary) heap*** data structure is an array object that we can also view as a nearly complete tree as shown in Fig. 6.1. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from left up to a point.

The term "heap" was originally coined in the context of heapsort, but it has since come to refer to "garbage-collected storage" in case of programming languages such as Java or Lisp. In this lecture, when we refer to heap, we are referring to heap data structure.

5.2 The heap data structure

A binary heap can have two forms: max-heap or min-heap. A max-heap is a complete binary tree with each key at a parent node being greater than the keys at its children nodes in values, in case of min-heap, its the other way round.

In case of max-heap, the maximum value of the heap will be at the root. However, the values in level 3 maybe smaller than the values in level 4 in case they are in different branches. A max-heap structure is valid as long as the keys at each parent nodes has larger values than the keys at their children.

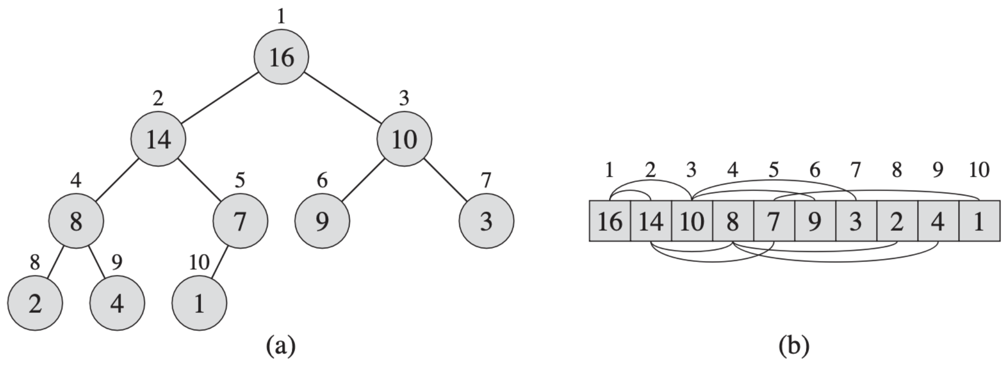

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

5.2.1 Implementation of the heap

We can implement a heap as an array. We populate the array by listing the key value of the nodes from the tree from left to right and from top to bottom. In Figure 6.1, the heap in (a) has been implemented in array shown in (b).

An array A that represents a heap is an object with two attributes: $A.length$, which (as usual) gives the number of elements in the array, and $A.heapsize$, which represents how many elements are actually part of the heap. That is, although $A[1 \mathinner{..} A.heapsize]$, where $0 \leq A.heapsize \leq A.length$, are valid elements of the heap. The root of the tree is $A[1]$, and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

```
PARENT($i$)
1  return $\lfloor i/2 \lfloor$
```

```
LEFT($i$)
1  return $2i$
```

```
RIGHT($i$)
1  return $2i + 1$
```

You can verify these properties using the heap shown in Figure 6.1.

In case of ***max-heap***, the ***max-heap property*** is that for every node i other than the root,

$A[\text{PARENT}(i)] \geq A[i]$,

that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at its root, and the same property is maintained in the subtrees of the heap as well.

In case of ***min-heap***,

$A[\text{PARENT}(i)] \leq A[i]$,

therefore, the smallest element in a min-heap is at the root.

Definition 1. Viewing a heap as a tree, we define the ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root.

5.3 Classwork

Exercise 1. What are the minimum and maximum numbers of elements in a heap of height h?

Exercise 2. Show that an n-element heap has height $\lfloor \lg n \rfloor$.

5.4 Maintaining the heap property

In order to maintain the heap property, we have to ensure that each node has at most value of its children, otherwise, the node is violating the heap property. In case of such violation, we must "heapify" the heap A where the node i must be "floated down" (in case of max-heap) to where it fits.

We sketch out the procedure called Max-Heapify, which takes the heap A, and node i to fix the violation mentioned above. If there is no violation, the procedure does not modify the heap. **When it is called, Max-Heapify assumes that the binary trees rooted at Left(i) and Right(i) are max-heaps, but that $A[i]$ might be smaller than its children, thus, violating the max-heap property.**

```
Max-Heapify($A, i$)
 1  $l$ = Left($i$)
 2  $r$ = Right($i$)
 3  if $l \leq A.heapsize$ and $A[l] > A[i]$
 4        $largest = l$
 5  else $largest = i$
 6  if $r \leq A.heapsize$ and $A[r] > A[largest]$
 7        $largest = r$
 8  if $largest \neq i$
 9        exchange $A[i]$ with $A[largest]$
10        Max-Heapify($A, largest$)
```

We first find the left node and right node to the current node, i, provided in line 1 and 2. Next, we check if the left node is within the heapsize and whether $A[l] > A[i]$, which indicate the heap property violation. If $A[l] > A[i]$ is true within the heapsize, we copy current node to $largest = i$. If the conditional check in line 3 fails, we keep i as the largest node. Next, we repeat the same check with the right node. If $largest \neq i$, either $largest = l$ or $largest = r$. Therefore, we need to repeat the same procedure in the subtree rooted at $largest$, which can be either l or r, thus, calling Max-Heapify($A, largest$).

Max-Heapify runs in the time $O(h)$, where h is the height of the node.

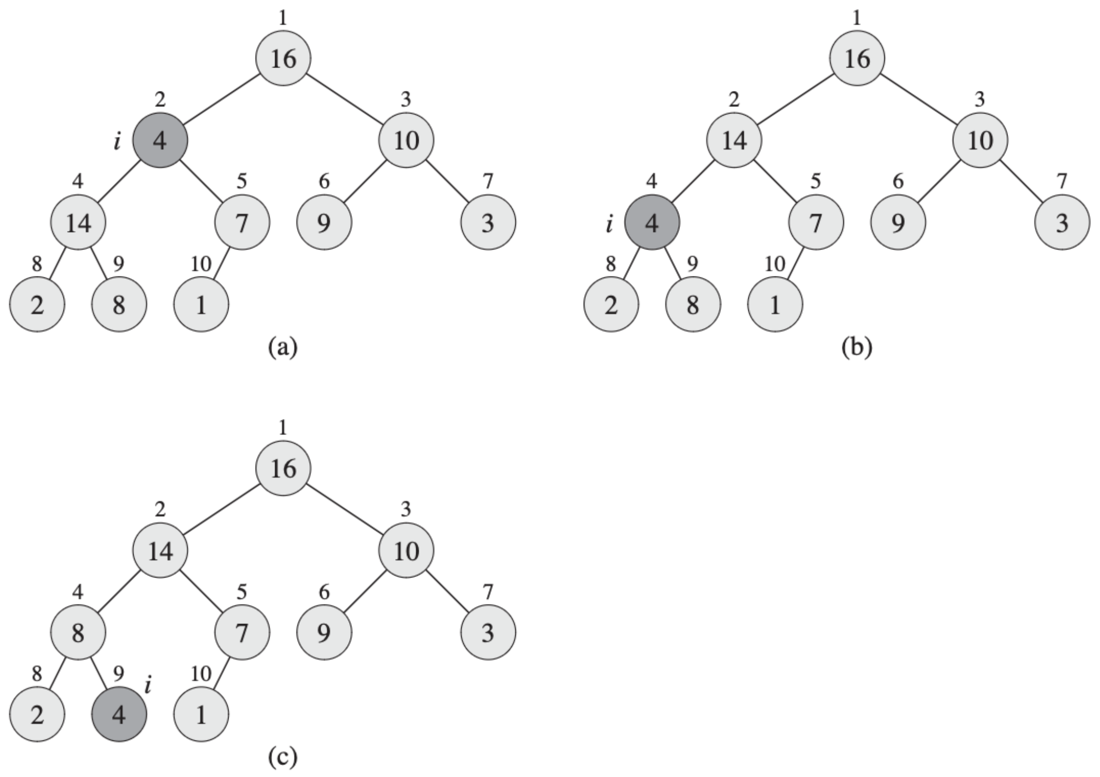

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

Exercise 3. Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, ..., n.

5.5 Building a heap

From Exercise 3, we know that the elements in the subarray $A[(\lfloor n/2 \rfloor) \mathinner{.\,.} n]$ are all leaves of the tree, and so each is a 1-element heap to begin with. Therefore, the procedure BUILD-MAX-HEAP(A) goes through the remaining nodes of the tree and runs the MAX-HEAPIFY on each one.

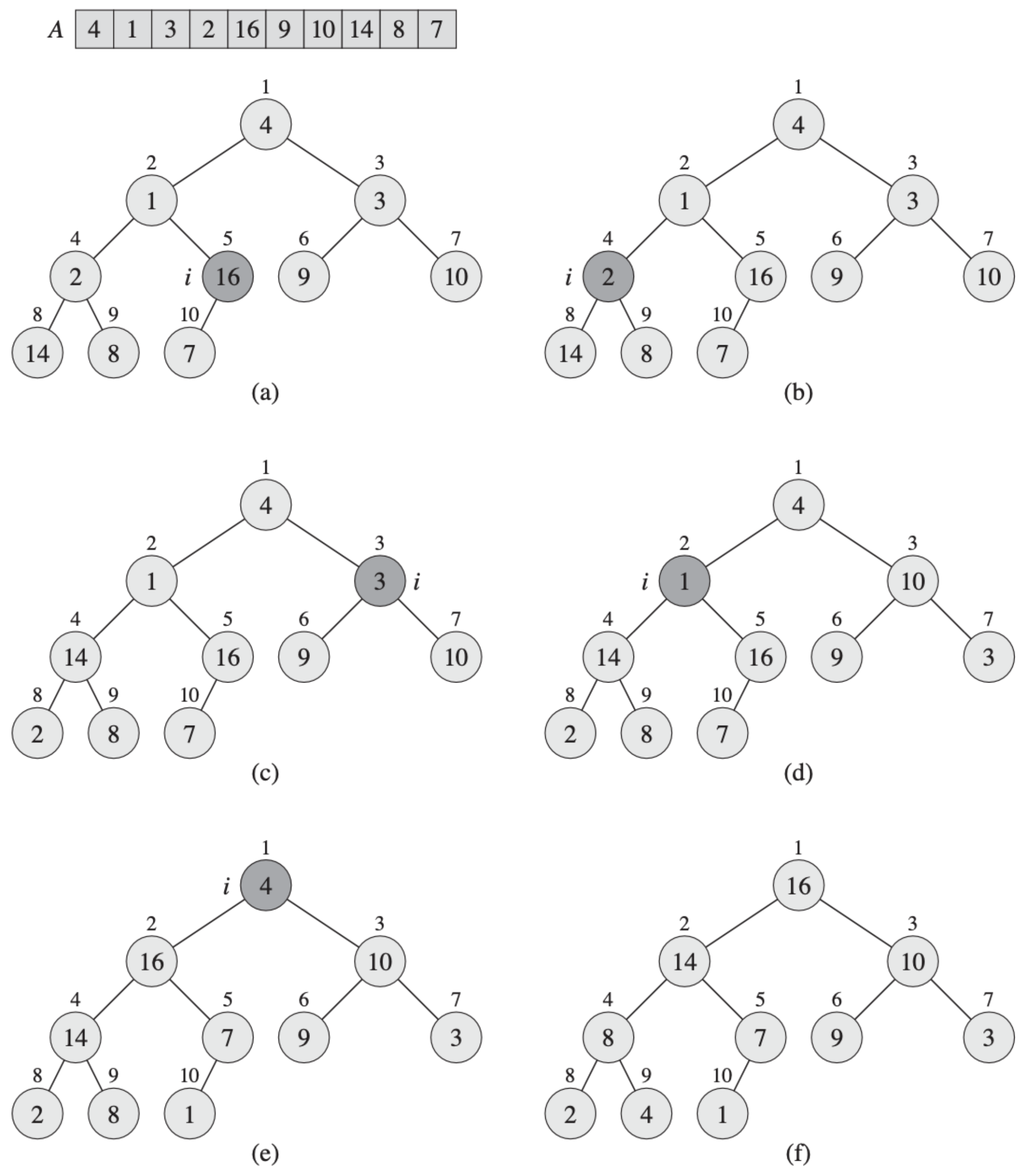

Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

```
BUILD-MAX-HEAP(A)
1  A.heapsize = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

5.5.1 Correctness

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

At the start of each iteration of the **for** loop of lines 2-3, each node $i + 1$, $i + 2$, ..., n is the root of a max-heap.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, ..., n is a leaf and is thus the root of a trivial max-heap.
Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1$, $i + 2$, ..., n are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination $i = 0$. By the loop invariant, each node 1, 2, ..., n is the root of a max-heap. In particular, node 1 is.

In Figure 6.3, when the MAX-HEAPIFY$(A, 5)$ is called, prior to this call, node 6, 7, 8, 9, & 10 are trivially roots to 1-element max-heaps. This is essentially the base case (initialization) step in the proof above.

For the maintenance step, MAX-HEAPIFY(A, i) is called on node 5, 4, 3, 2, & 1. With each call, MAX-HEAPIFY(A, i) re-establishes the loop invariant for each succeeding nodes. When MAX-HEAPIFY$(A, 5)$ is done, node 5 is a root to a subtree which becomes a max-heap rooted at node 5, ready for the next iteration for MAX-HEAPIFY$(A, 4)$. The loop invariant is maintained for 5, 6, 7, 8, 9 & 10 are roots to max-heaps.

This continues till $i = 0$, when the loop terminates. As the loop terminates 1, 2, 3, ..., 10 are all roots to max-heaps.

5.5.2 Runtime

Simpler upper bonud: We can compute a simpler upper bound on the running time of BUILD-MAX-HEAP as follows. Each call to MAX-HEAPIFY costs $O(\lg n)$ time and BUILD-MAX-HEAP makes $O(n)$ such calls. Thus, the running time is $O(n \lg n)$. Although this is a correct upper bound, it's not a tight bound for this problem.

Tigher upper bound:

Lemma 1. An n-element heap has a height of $\lfloor \lg n \rfloor$.

Lemma 2. An n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes for any height h. (Exercise 6.3-3 from CLRS)

Therefore the runtime becomes:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil O(h) \\ &= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) \\ &\le O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(2n) \\ &= O(n) \end{aligned}$$

Proof of summation formula: The sum of infinite geometric series is $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ when $|x| < 1$. Differentiating on the both sides with respect x, and multiplying with x on both sides, we get:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Replacing $k = h$ and $x = 1/2$:

$$\sum_{h=0}^{\infty} h(1/2)^h = \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

5.6 Heapsort

Once we build a max-heap using, BUILD-MAX-HEAP,

1. we can repeatedly pick the largest element from the heap
2. exchange $A[1]$ with the element at the tail of the unsorted subarray $A[1, ...i]$
3. $A.heapsize = A.heapsize - 1$, and adjust the subarray length $i- = 1$
4. The exchange $A[1]$ and $A[i]$ might violate the max-heap, therefore, we max-heapify the root using MAX-HEAPIFY$(A, 1)$

For the steps above will repeatedly till we swap the largest element in the decreasing heap with $A[2]$, we end up with a sorted array.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for $i = A.length$ downto 2
3      exchange $A[1]$ with $A[i]$
4      $A.heapsize = A.heapsize - 1$
5      MAX-HEAPIFY$(A, 1)$
```

Here line 1 takes $O(n)$, the loop runs n times out of which the body of the for loop runs $n-1$. MAX-HEAPIFY$(A, 1)$ in line 5 takes $O(\lg n)$, because the height of the heap is $O(\lg n)$. Putting it together, the runtime of heapsort is $O(n \lg n)$.

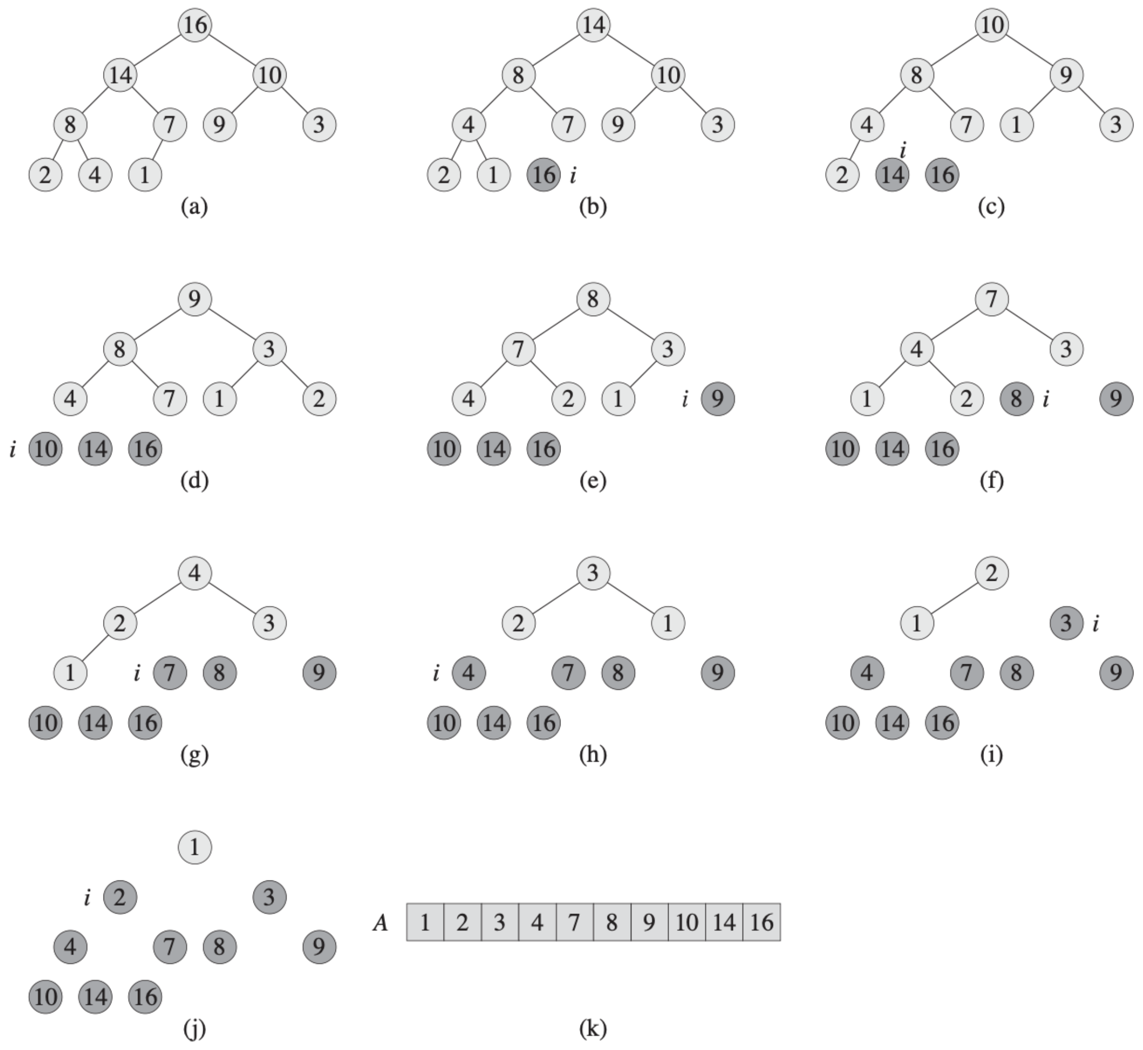

Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

5.7 Priority queues

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called ***key***. A ***max-priority*** queue supports the following operations:

- INSERT(S, x) insert the element x into the set S.
- MAXIMUM(S) returns the element of S with the largest key.
- EXTRACT-MAX(S) removes and returns the element with the largest key.
- INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as x's current key value.

We can use max-priority queues to schedule jobs on a shared computer. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling EXTRACT-MAX. The scheduler can add a new job to the queue at any time by calling INSERT.

Alternatively, a ***min-priority queue*** supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY. Min-priority queues will be used in Dijkstra's algorithm to find the shortest paths in graphs later in this class.

5.7.1 Implementation using heaps

Let's go over the implementation of the operations of a max-priority queue. The procedure HEAP-MAXIMUM implements the MAXIMUM operation in $\Theta(1)$ time.

```
HEAP-MAXIMUM($A$)
1  return $A[1]$
```

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation, which is similar to body of **for** loop in HEAPSORT procedure.

```
HEAP-EXTRACT-MAX($A$)
1  if $A.heapsize < 1$
2      error "heap underflow"
3  $max = A[1]$
4  $A[1] = A[A.heapsize]$
5  $A.heapsize = A.heapsize - 1$
6  MAX-HEAPIFY($A, 1$)
7  return $max$
```

The running time of HEAP-EXTRACT-MAX is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for MAX-HEAPIFY.

The procedure Heap-Increase-Key implements Increase-Key operation. An index i into the array identifies the priority queue element whose *key* we wish to increase. The procedure first updates the key of element $A[i]$ to its new value. This update in line 3 can violate the max-heap property, the procedure then, **in a manner reminiscent of the insertion loop of Insertion-Sort, traverses a simple path from this node toward the root to find a proper place for the newly increased key**.

As Heap-Increase-Key traverses this path, it repeatedly compares an element to its parent, exchanging their keys and continuing if the element's key is larger, and terminating if the element's key is smaller, since the max-heap property now holds.

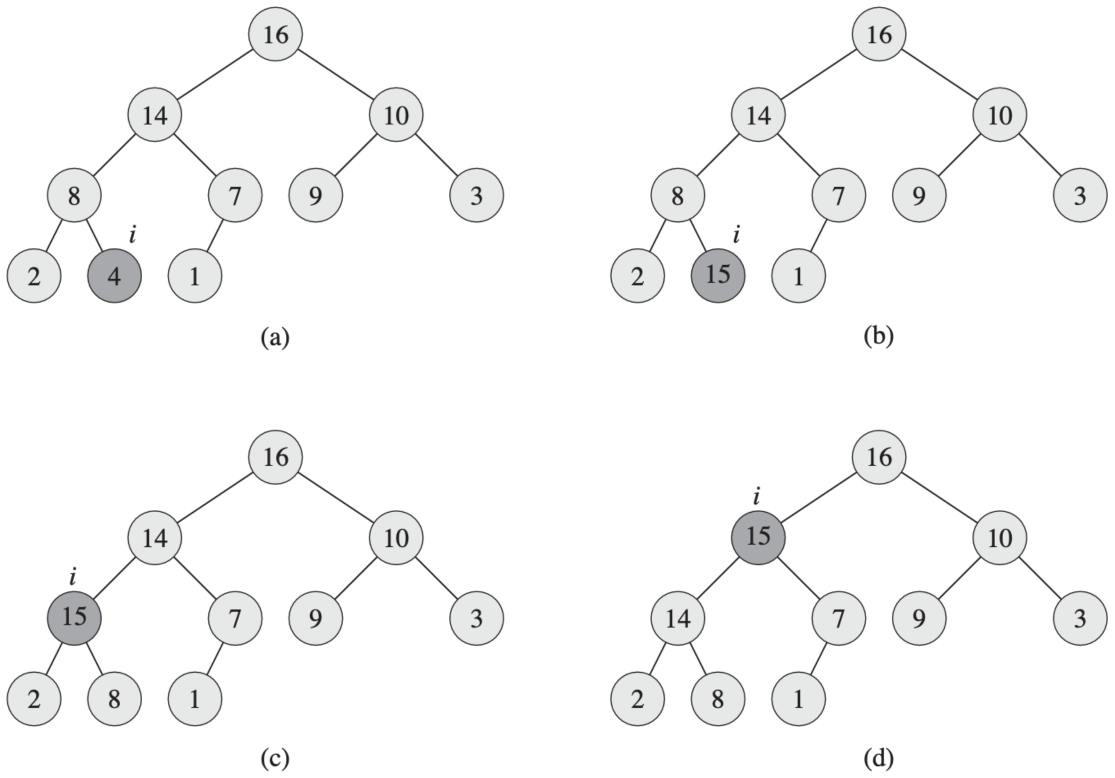

Figure 6.5 The operation of Heap-Increase-Key. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{Parent}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

```
Heap-Increase-Key($A, i, key$)
1  if $key < A[i]$
2       error "new key is smaller than current key"
3  $A[i] = key$
4  while $i > 1$ and $A[\text{Parent}(i)] < A[i]$
5       exchange $A[i]$ with $A[\text{Parent}(i)]$
6       $i = \text{Parent}(i)$
```

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes as an input the key of the new element to be inserted into max-heap A. The procedure first expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.

```
MAX-HEAP-INSERT($A, key$)
1  $A.heapsize = A.heapsize - 1$
2  $A[A.heapsize] = -\infty$
3  HEAP-INCREASE-KEY($A, A.heapsize, key$)
```

The running time of MAX-HEAP-INSERT on an n-element heap in $O(\lg n)$.