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

INSERT(S, x, k) inserts the element x with key k into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S 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 large as x's current key value.

Among their other applications, you can use max-priority queues to schedule jobs on a computer shared among multiple users. 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. A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program calls EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, the simulator inserts them into the min-priority queue by calling INSERT. We'll see other uses for min-priority queues, highlighting the DECREASE-KEY operation, in Chapters 21 and 22.

When you use a heap to implement a priority queue within a given application, elements of the priority queue correspond to objects in the application. Each object contains a key. If the priority queue is implemented by a heap, you need to determine which application object corresponds to a given heap element, and vice versa. Because the heap elements are stored in an array, you need a way to map application objects to and from array indices.

One way to map between application objects and heap elements uses ***handles***, which are additional information stored in the objects and heap elements that give enough information to perform the mapping. Handles are often implemented to be opaque to the surrounding code, thereby maintaining an abstraction barrier between the application and the priority queue. For example, the handle within an application object might contain the corresponding index into the heap array. But since only the code for the priority queue accesses this index, the index is entirely hidden from the application code. Because heap elements change locations within

the array during heap operations, an actual implementation of the priority queue, upon relocating a heap element, must also update the array indices in the corresponding handles. Conversely, each element in the heap might contain a pointer to the corresponding application object, but the heap element knows this pointer as only an opaque handle and the application maps this handle to an application object. Typically, the worst-case overhead for maintaining handles is $O(1)$ per access.

As an alternative to incorporating handles in application objects, you can store within the priority queue a mapping from application objects to array indices in the heap. The advantage of doing so is that the mapping is contained entirely within the priority queue, so that the application objects need no further embellishment. The disadvantage lies in the additional cost of establishing and maintaining the mapping. One option for the mapping is a hash table (see Chapter 11).[1] The added expected time for a hash table to map an object to an array index is just $O(1)$, though the worst-case time can be as bad as $\Theta(n)$.

Let's see how to implement the operations of a max-priority queue using a max-heap. In the previous sections, we treated the array elements as the keys to be sorted, implicitly assuming that any satellite data moved with the corresponding keys. When a heap implements a priority queue, we instead treat each array element as a pointer to an object in the priority queue, so that the object is analogous to the satellite data when sorting. We further assume that each such object has an attribute *key*, which determines where in the heap the object belongs. For a heap implemented by an array A, we refer to $A[i].key$.

The procedure MAX-HEAP-MAXIMUM on the facing page implements the MAXIMUM operation in $\Theta(1)$ time, and MAX-HEAP-EXTRACT-MAX implements the operation EXTRACT-MAX. MAX-HEAP-EXTRACT-MAX is similar to the **for** loop body (lines 3–5) of the HEAPSORT procedure. We implicitly assume that MAX-HEAPIFY compares priority-queue objects based on their *key* attributes. We also assume that when MAX-HEAPIFY exchanges elements in the array, it is exchanging pointers and also that it updates the mapping between objects and array indices. The running time of MAX-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, plus whatever overhead is incurred within MAX-HEAPIFY for mapping priority-queue objects to array indices.

The procedure MAX-HEAP-INCREASE-KEY on page 176 implements the INCREASE-KEY operation. It first verifies that the new key k will not cause the key in the object x to decrease, and if there is no problem, it gives x the new key value. The procedure then finds the index i in the array corresponding to object x,

---

[1] In Python, dictionaries are implemented with hash tables.

```
MAX-HEAP-MAXIMUM(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  return A[1]

MAX-HEAP-EXTRACT-MAX(A)
1  max = MAX-HEAP-MAXIMUM(A)
2  A[1] = A[A.heap-size]
3  A.heap-size = A.heap-size − 1
4  MAX-HEAPIFY(A, 1)
5  return max
```

so that $A[i]$ is x. Because increasing the key of $A[i]$ might violate the max-heap property, the procedure then, in a manner reminiscent of the insertion loop (lines 5–7) of INSERTION-SORT on page 19, traverses a simple path from this node toward the root to find a proper place for the newly increased key. As MAX-HEAP-INCREASE-KEY traverses this path, it repeatedly compares an element's key to that of its parent, exchanging pointers 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. (See Exercise 6.5-7 for a precise loop invariant.) Like MAX-HEAPIFY when used in a priority queue, MAX-HEAP-INCREASE-KEY updates the information that maps objects to array indices when array elements are exchanged. Figure 6.5 shows an example of a MAX-HEAP-INCREASE-KEY operation. In addition to the overhead for mapping priority queue objects to array indices, the running time of MAX-HEAP-INCREASE-KEY on an n-element heap is $O(\lg n)$, since the path traced from the node updated in line 3 to the root has length $O(\lg n)$.

The procedure MAX-HEAP-INSERT on the next page implements the INSERT operation. It takes as inputs the array A implementing the max-heap, the new object x to be inserted into the max-heap, and the size n of array A. The procedure first verifies that the array has room for the new element. It then expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls MAX-HEAP-INCREASE-KEY to set the key of this new element to its correct value and maintain the max-heap property. The running time of MAX-HEAP-INSERT on an n-element heap is $O(\lg n)$ plus the overhead for mapping priority queue objects to indices.

In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time, plus the overhead for mapping priority queue objects to array indices.

```
MAX-HEAP-INCREASE-KEY(A, x, k)
1  if k < x.key
2       error "new key is smaller than current key"
3  x.key = k
4  find the index i in array A where object x occurs
5  while i > 1 and A[PARENT(i)].key < A[i].key
6       exchange A[i] with A[PARENT(i)], updating the information that maps
            priority queue objects to array indices
7       i = PARENT(i)

MAX-HEAP-INSERT(A, x, n)
1  if A.heap-size == n
2       error "heap overflow"
3  A.heap-size = A.heap-size + 1
4  k = x.key
5  x.key = −∞
6  A[A.heap-size] = x
7  map x to index heap-size in the array
8  MAX-HEAP-INCREASE-KEY(A, x, k)
```

Exercises

6.5-1

Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-2

Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-INSERT$(A, 10)$ on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-3

Write pseudocode to implement a min-priority queue with a min-heap by writing the procedures MIN-HEAP-MINIMUM, MIN-HEAP-EXTRACT-MIN, MIN-HEAP-DECREASE-KEY, and MIN-HEAP-INSERT.

6.5-4

Write pseudocode for the procedure MAX-HEAP-DECREASE-KEY(A, x, k) in a max-heap. What is the running time of your procedure?