Array resizing. We can add a no-argument constructor, code for array doubling in `insert()`, and code for array halving in `delMax()`, just as we did for stacks in SECTION 1.3. Thus, clients need not be concerned about arbitrary size restrictions. The logarithmic time bounds implied by PROPOSITION Q are *amortized* when the size of the priority queue is arbitrary and the arrays are resized (see EXERCISE 2.4.22).

Immutability of keys. The priority queue contains objects that are created by clients but assumes that client code does not change the keys (which might invalidate the heap-order invariant). It is possible to develop mechanisms to enforce this assumption, but programmers typically do not do so because they complicate the code and are likely to degrade performance.

Index priority queue. In many applications, it makes sense to allow clients to refer to items that are already on the priority queue. One easy way to do so is to associate a unique integer *index* with each item. Moreover, it is often the case that clients have a universe of items of a known size N and perhaps are using (parallel) arrays to store information about the items, so other unrelated client code might already be using an integer index to refer to items. These considerations lead us to the following API:

`public class IndexMinPQ<Item extends Comparable<Item>>`

	`IndexMinPQ(int maxN)`	*create a priority queue of capacity* `maxN` *with possible indices between* `0` *and* `maxN-1`
`void`	`insert(int k, Item item)`	*insert* `item`*; associate it with* `k`
`void`	`change(int k, Item item)`	*change the item associated with* `k` *to* `item`
`boolean`	`contains(int k)`	*is* `k` *associated with some item?*
`void`	`delete(int k)`	*remove* `k` *and its associated item*
`Item`	`min()`	*return a minimal item*
`int`	`minIndex()`	*return a minimal item's index*
`int`	`delMin()`	*remove a minimal item and return its index*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`int`	`size()`	*number of items in the priority queue*

API for a generic priority queue with associated indices

A useful way of thinking of this data type is as implementing an array, but with fast access to the smallest entry in the array. Actually it does even better—it gives fast access to the minimum entry in a *specified subset* of an array's entries (the ones that have been inserted. In other words, you can think of an `IndexMinPQ` named `pq` as representing a subset of an array `pq[0..N-1]` of items. Think of the call `pq.insert(k, item)` as adding `k` to the subset and setting `pq[k] = item` and the call `pq.change(k, item)` as setting `pq[k] = item`, both also maintaining data structures needed to support the other operations, most importantly `delMin()` (remove and return the index of the minimum key) and `change()` (change the item associated with an index that is already in the data structure—just as in `pq[i] = item`). These operations are important in many applications and are enabled by our ability to refer to the key (with the index). EXERCISE 2.4.33 describes how to extend ALGORITHM 2.6 to implement index priority queues with remarkable efficiency and with remarkably little code. Intuitively, when an item in the heap changes, we can restore the heap invariant with a sink operation (if the key increases) and a swim operation (if the key decreases). To perform the operations, we use the index to find the item in the heap. The ability to locate an item in the heap also allows us to add the `delete()` operation to the API.

operation	order of growth of number of compares
`insert()`	$\log N$
`change()`	$\log N$
`contains()`	1
`delete()`	$\log N$
`min()`	1
`minIndex()`	1
`delMin()`	$\log N$

Worst-case costs for an *N*-item heap-based indexed priority queue

Proposition Q (continued). In an index priority queue of size N, the number of compares required is proportional to at most $\log N$ for *insert, change priority, delete,* and *remove the minimum.*

Proof: Immediate from inspection of the code and the fact that all paths in a heap are of length at most ~lg N.

This discussion is for a minimum-oriented queue; as usual, we also implement on the booksite a maximum-oriented version `IndexMaxPQ`.

Index priority-queue client. The `IndexMinPQ` client `Multiway` on page 322 solves the *multiway merge* problem: it merges together several sorted input streams into one sorted output stream. This problem arises in many applications: the streams might be the output of scientific instruments (sorted by time), lists of information from the web such as music or movies (sorted by title or artist name), commercial transactions (sorted by account number or time), or whatever. If you have the space, you might just read them all into an array and sort them, but with a priority queue, you can read input streams and put them in sorted order on the output *no matter how long they are.*

Multiway merge priority-queue client

```
public class Multiway
{
   public static void merge(In[] streams)
   {
      int N = streams.length;
      IndexMinPQ<String> pq = new IndexMinPQ<String>(N);

      for (int i = 0; i < N; i++)
         if (!streams[i].isEmpty())
             pq.insert(i, streams[i].readString());

      while (!pq.isEmpty())
      {
         StdOut.println(pq.min());
         int i = pq.delMin();
         if (!streams[i].isEmpty())
             pq.insert(i, streams[i].readString());
      }
   }

   public static void main(String[] args)
   {
      int N = args.length;
      In[] streams = new In[N];
      for (int i = 0; i < N; i++)
          streams[i] = new In(args[i]);
      merge(streams);
   }
}
```

This `IndexMinPQ` client merges together the sorted input stream given as command-line arguments into a single sorted output stream on standard output (see text). Each stream index is associated with a key (the next string in the stream). After initialization, it enters a loop that prints the smallest string in the queue and removes the corresponding entry, then adds a new entry for the next string in that stream. For economy, the output is shown on one line below—the actual output is one string per line.

```
% more m1.txt
A B C F G I I Z
% more m2.txt
B D H P Q Q
% more m3.txt
A B E F J N
```

```
% java Multiway m1.txt m2.txt m3.txt
A A B B B C D E F F G H I I J N P Q Q Z
```