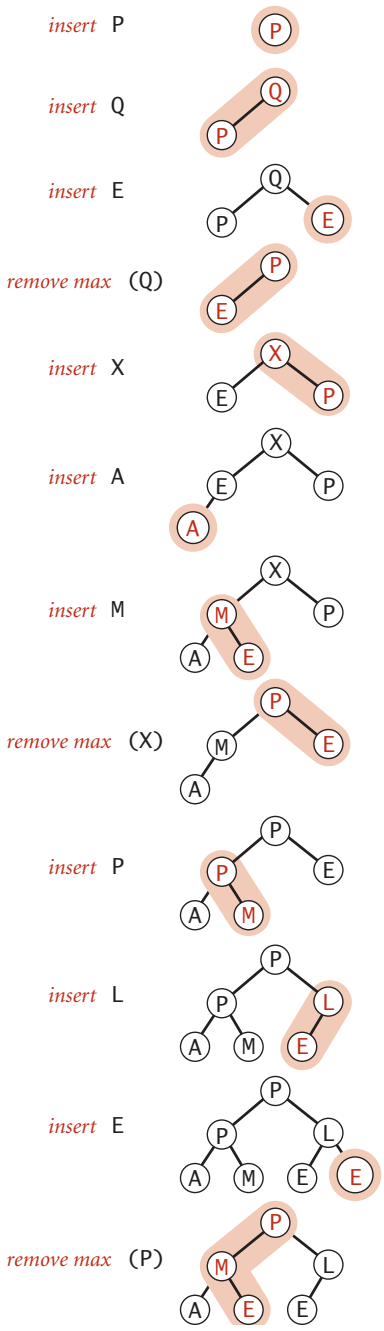

Proposition Q. In an N-key priority queue, the heap algorithms require no more than lg N compares for *insert* and no more than 2 lg N compares for *remove the maximum*.

Proof: Both operations involve moving along a path between the root and the bottom of the heap whose length is no more than lg N by PROPOSITION P. The *remove the maximum* operation requires two compares for each node: one to find the child with the larger key, the other to decide whether that child needs to be promoted.

For typical applications that require a large number of intermixed insert and remove the maximum operations in a large priority queue, PROPOSITION Q represents an important performance breakthrough, summarized in the table at the top of the next page. Where elementary implementations using an ordered array or an unordered array require linear time for one of the operations, a heap-based implementation provides a guarantee that both operations complete in logarithmic time. This improvement can make the difference between solving a problem and not being able to address it at all.

WE CONCLUDE our study of the heap priority queue API implementation with a few practical considerations.

Dynamic array resizing. Adding a no-argument constructor, code for array doubling in `insert()`, and code for array halving in `delMax()` is easily accomplished, just as we did for stacks and queues in CHAPTER 1. 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 2.4.23).

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