15.3 Comparison of the Two Methods

Both the array and linked-list versions run in constant time per operation. Thus they are so fast that they are unlikely to be the bottleneck of any algorithm. As such, it will rarely matter which version is used.

The array versions of these data structures are likely to be faster than their linked-list counterparts, especially if an accurate estimation of the capacity is available. If an additional constructor is provided to specify the initial capacity (see Exercise 15.3) and the estimate is correct, no doubling will be performed. Also, the sequential access provided by an array is typically faster than the potential nonsequential access offered by dynamic memory allocation.

The array implementation does have two drawbacks, however. First, for queues, the array implementation is arguably more complex than the linked-list implementation due to the combined code for wraparound and array doubling. Our implementation of array doubling was not as efficient as possible (see Exercise 15.8), thus a faster implementation of the queue would require a few additional lines of code. Even the array implementation of the stack uses a few more lines of code than its linked-list counterpart.

The second drawback affects other languages, but not Java. When doubling, we require, temporarily, three times as much space as the number of data items would suggest. This is because when the array is doubled, we need to have memory to store both the old and the new (double-sized) array. Further, at the queue's peak size, the array is between 50 percent and 100 percent full: On average, it is 75 percent full, meaning that for every three items in the array, one spot is empty. The wasted space is thus 33 percent on average and 100 percent when the table is only half full. In Java, this is not a problem because each element in the array is simply a reference. In other languages, such as C++, objects are stored directly, rather than referenced. In these languages, the wasted space could be significant when compared to the linked-list-based version that uses only an extra reference per item.

15.4 Double-ended Queues

A *double-ended queue (deque)* allows access at both ends. Much of its functionality can be derived from the queue class.

Implementation of the deque is simple when inheritance is used.

This chapter closes with a discussion of the use of inheritance to derive a new data structure. A *double-ended queue* (*deque*) is like a queue, except that access is allowed at both ends. Exercise 14.11 describes an application of the deque. Rather than the terms `enqueue` and `dequeue`, the terms used are `addFront`, `addRear`, `removeFront`, and `removeRear`. Figure 15.26 shows the derived class `Deque`. We derive from `QueueAr` because deletion from the back of the list is not efficiently supported.

The default constructor is a call to `super`; this is acceptable. `enqueue` and `dequeue` still work; we cannot remove methods. `getFront` is unchanged from the queue class and is thus inherited unmodified. `addBack` and `removeFront` call the existing `enqueue` and `dequeue` routines. The only routines that need to