Data Structures and Algorithm Analysis

Edition 3.2 (C++ Version)

Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

March 28, 2013
Update 3.2.0.10
For a list of changes, see
http://people.cs.vt.edu/~shaffer/Book/errata.html

If you wish to have a printed version of this document, print copies are published by Dover Publications
(see http://store.doverpublications.com/048648582x.html).
Further information about this text is available at
http://people.cs.vt.edu/~shaffer/Book/.

can be distinguished from full queues. Member **maxSize** is used to control the circular motion of the queue (it is the base for the modulus operator). Member **rear** is set to the position of the current rear element, while **front** is the position of the current front element.

In this implementation, the front of the queue is defined to be toward the lower numbered positions in the array (in the counter-clockwise direction in Figure 4.25), and the rear is defined to be toward the higher-numbered positions. Thus, **enqueue** increments the rear pointer (modulus **size**), and **dequeue** increments the front pointer. Implementation of all member functions is straightforward.

4.3.2 Linked Queues

The linked queue implementation is a straightforward adaptation of the linked list. Figure 4.27 shows the linked queue class declaration. Methods **front** and **rear** are pointers to the front and rear queue elements, respectively. We will use a header link node, which allows for a simpler implementation of the enqueue operation by avoiding any special cases when the queue is empty. On initialization, the **front** and **rear** pointers will point to the header node, and front will always point to the header node while rear points to the true last link node in the queue. Method **enqueue** places the new element in a link node at the end of the linked list (i.e., the node that **rear** points to) and then advances **rear** to point to the new link node. Method **dequeue** removes and returns the first element of the list.

4.3.3 Comparison of Array-Based and Linked Queues

All member functions for both the array-based and linked queue implementations require constant time. The space comparison issues are the same as for the equivalent stack implementations. Unlike the array-based stack implementation, there is no convenient way to store two queues in the same array, unless items are always transferred directly from one queue to the other.

4.4 Dictionaries

The most common objective of computer programs is to store and retrieve data. Much of this book is about efficient ways to organize collections of data records so that they can be stored and retrieved quickly. In this section we describe a simple interface for such a collection, called a **dictionary**. The dictionary ADT provides operations for storing records, finding records, and removing records from the collection. This ADT gives us a standard basis for comparing various data structures.

Before we can discuss the interface for a dictionary, we must first define the concepts of a **key** and **comparable** objects. If we want to search for a given record