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

Copyright © 2009-2012 by Clifford A. Shaffer.
This document is made freely available in PDF form for educational and other non-commercial use. You may make copies of this file and redistribute in electronic form without charge. You may extract portions of this document provided that the front page, including the title, author, and this notice are included. Any commercial use of this document requires the written consent of the author. The author can be reached at shaffer@cs.vt.edu.
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/.

Example 4.3 The **TOH** function shown in Figure 2.2 makes two recursive calls: one to move $n-1$ rings off the bottom ring, and another to move these $n-1$ rings back to the goal pole. We can eliminate the recursion by using a stack to store a representation of the three operations that **TOH** must perform: two recursive calls and a move operation. To do so, we must first come up with a representation of the various operations, implemented as a class whose objects will be stored on the stack.

Figure 4.22 shows such a class. We first define an enumerated type called **TOHop**, with two values MOVE and TOH, to indicate calls to the **move** function and recursive calls to **TOH**, respectively. Class **TOHobj** stores five values: an operation field (indicating either a move or a new TOH operation), the number of rings, and the three poles. Note that the move operation actually needs only to store information about two poles. Thus, there are two constructors: one to store the state when imitating a recursive call, and one to store the state for a move operation.

An array-based stack is used because we know that the stack will need to store exactly $2n+1$ elements. The new version of **TOH** begins by placing on the stack a description of the initial problem for n rings. The rest of the function is simply a **while** loop that pops the stack and executes the appropriate operation. In the case of a **TOH** operation (for $n > 0$), we store on the stack representations for the three operations executed by the recursive version. However, these operations must be placed on the stack in reverse order, so that they will be popped off in the correct order.

Recursive algorithms lend themselves to efficient implementation with a stack when the amount of information needed to describe a sub-problem is small. For example, Section 7.5 discusses a stack-based implementation for Quicksort.

4.3 Queues

Like the stack, the **queue** is a list-like structure that provides restricted access to its elements. Queue elements may only be inserted at the back (called an **enqueue** operation) and removed from the front (called a **dequeue** operation). Queues operate like standing in line at a movie theater ticket counter.[1] If nobody cheats, then newcomers go to the back of the line. The person at the front of the line is the next to be served. Thus, queues release their elements in order of arrival. Accountants have used queues since long before the existence of computers. They call a queue a "FIFO" list, which stands for "First-In, First-Out." Figure 4.23 shows a sample

[1] In Britain, a line of people is called a "queue," and getting into line to wait for service is called "queuing up."

```
// Operation choices: DOMOVE will move a disk
// DOTOH corresponds to a recursive call
enum TOHop { DOMOVE, DOTOH };
class TOHobj { // An operation object
public:
  TOHop op;              // This operation type
  int num;               // How many disks
  Pole start, goal, tmp; // Define pole order

  // DOTOH operation constructor
  TOHobj(int n, Pole s, Pole g, Pole t) {
    op = DOTOH; num = n;
    start = s; goal = g; tmp = t;
  }

  // DOMOVE operation constructor
  TOHobj(Pole s, Pole g)
    { op = DOMOVE; start = s; goal = g; }
};

void TOH(int n, Pole start, Pole goal, Pole tmp,
         Stack<TOHobj*>& S) {
  S.push(new TOHobj(n, start, goal, tmp)); // Initial
  TOHobj* t;
  while (S.length() > 0) {       // Grab next task
    t = S.pop();
    if (t->op == DOMOVE)   // Do a move
      move(t->start, t->goal);
    else if (t->num > 0) {
      // Store (in reverse) 3 recursive statements
      int num = t->num;
      Pole tmp = t->tmp;  Pole goal = t->goal;
      Pole start = t->start;
      S.push(new TOHobj(num-1, tmp, goal, start));
      S.push(new TOHobj(start, goal));
      S.push(new TOHobj(num-1, start, tmp, goal));
    }
    delete t; // Must delete the TOHobj we made
  }
}
```

Figure 4.22 Stack-based implementation for Towers of Hanoi.

queue ADT. This section presents two implementations for queues: the array-based queue and the linked queue.

4.3.1 Array-Based Queues

The array-based queue is somewhat tricky to implement effectively. A simple conversion of the array-based list implementation is not efficient.

Assume that there are n elements in the queue. By analogy to the array-based list implementation, we could require that all elements of the queue be stored in the first n positions of the array. If we choose the rear element of the queue to be in

```
// Abstract queue class
template <typename E> class Queue {
private:
  void operator =(const Queue&) {}     // Protect assignment
  Queue(const Queue&) {}         // Protect copy constructor

public:
  Queue() {}          // Default
  virtual ~Queue() {} // Base destructor

  // Reinitialize the queue.  The user is responsible for
  // reclaiming the storage used by the queue elements.
  virtual void clear() = 0;

  // Place an element at the rear of the queue.
  // it: The element being enqueued.
  virtual void enqueue(const E&) = 0;

  // Remove and return element at the front of the queue.
  // Return: The element at the front of the queue.
  virtual E dequeue() = 0;

  // Return: A copy of the front element.
  virtual const E& frontValue() const = 0;

  // Return: The number of elements in the queue.
  virtual int length() const = 0;
};
```

Figure 4.23 The **C**++ ADT for a queue.

position 0, then **dequeue** operations require only $\Theta(1)$ time because the front element of the queue (the one being removed) is the last element in the array. However, **enqueue** operations will require $\Theta(n)$ time, because the n elements currently in the queue must each be shifted one position in the array. If instead we chose the rear element of the queue to be in position $n-1$, then an **enqueue** operation is equivalent to an **append** operation on a list. This requires only $\Theta(1)$ time. But now, a **dequeue** operation requires $\Theta(n)$ time, because all of the elements must be shifted down by one position to retain the property that the remaining $n-1$ queue elements reside in the first $n-1$ positions of the array.

A far more efficient implementation can be obtained by relaxing the requirement that all elements of the queue must be in the first n positions of the array. We will still require that the queue be stored be in contiguous array positions, but the contents of the queue will be permitted to drift within the array, as illustrated by Figure 4.24. Now, both the **enqueue** and the **dequeue** operations can be performed in $\Theta(1)$ time because no other elements in the queue need be moved.

This implementation raises a new problem. Assume that the front element of the queue is initially at position 0, and that elements are added to successively

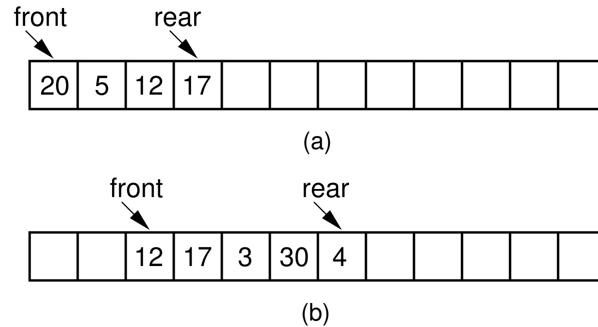

Figure 4.24 After repeated use, elements in the array-based queue will drift to the back of the array. (a) The queue after the initial four numbers 20, 5, 12, and 17 have been inserted. (b) The queue after elements 20 and 5 are deleted, following which 3, 30, and 4 are inserted.

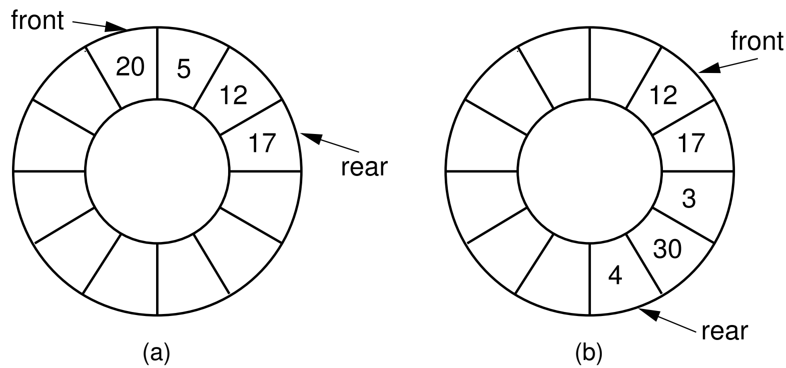

Figure 4.25 The circular queue with array positions increasing in the clockwise direction. (a) The queue after the initial four numbers 20, 5, 12, and 17 have been inserted. (b) The queue after elements 20 and 5 are deleted, following which 3, 30, and 4 are inserted.

higher-numbered positions in the array. When elements are removed from the queue, the front index increases. Over time, the entire queue will drift toward the higher-numbered positions in the array. Once an element is inserted into the highest-numbered position in the array, the queue has run out of space. This happens despite the fact that there might be free positions at the low end of the array where elements have previously been removed from the queue.

The "drifting queue" problem can be solved by pretending that the array is circular and so allow the queue to continue directly from the highest-numbered position in the array to the lowest-numbered position. This is easily implemented through use of the modulus operator (denoted by `%` in **C**++). In this way, positions in the array are numbered from 0 through **`size`**-1, and position **`size`**-1 is defined to immediately precede position 0 (which is equivalent to position **`size % size`**). Figure 4.25 illustrates this solution.

There remains one more serious, though subtle, problem to the array-based queue implementation. How can we recognize when the queue is empty or full? Assume that **front** stores the array index for the front element in the queue, and **rear** stores the array index for the rear element. If both **front** and **rear** have the same position, then with this scheme there must be one element in the queue. Thus, an empty queue would be recognized by having **rear** be *one less* than **front** (taking into account the fact that the queue is circular, so position **size**-1 is actually considered to be one less than position 0). But what if the queue is completely full? In other words, what is the situation when a queue with n array positions available contains n elements? In this case, if the front element is in position 0, then the rear element is in position **size**-1. But this means that the value for **rear** is one less than the value for **front** when the circular nature of the queue is taken into account. In other words, the full queue is indistinguishable from the empty queue!

You might think that the problem is in the assumption about **front** and **rear** being defined to store the array indices of the front and rear elements, respectively, and that some modification in this definition will allow a solution. Unfortunately, the problem cannot be remedied by a simple change to the definition for **front** and **rear**, because of the number of conditions or **states** that the queue can be in. Ignoring the actual position of the first element, and ignoring the actual values of the elements stored in the queue, how many different states are there? There can be no elements in the queue, one element, two, and so on. At most there can be n elements in the queue if there are n array positions. This means that there are $n + 1$ different states for the queue (0 through n elements are possible).

If the value of **front** is fixed, then $n+1$ different values for **rear** are needed to distinguish among the $n+1$ states. However, there are only n possible values for **rear** unless we invent a special case for, say, empty queues. This is an example of the Pigeonhole Principle defined in Exercise 2.30. The Pigeonhole Principle states that, given n pigeonholes and $n + 1$ pigeons, when all of the pigeons go into the holes we can be sure that at least one hole contains more than one pigeon. In similar manner, we can be sure that two of the $n + 1$ states are indistinguishable by the n relative values of **front** and **rear**. We must seek some other way to distinguish full from empty queues.

One obvious solution is to keep an explicit count of the number of elements in the queue, or at least a Boolean variable that indicates whether the queue is empty or not. Another solution is to make the array be of size $n + 1$, and only allow n elements to be stored. Which of these solutions to adopt is purely a matter of the implementor's taste in such affairs. My choice is to use an array of size $n + 1$.

Figure 4.26 shows an array-based queue implementation. **listArray** holds the queue elements, and as usual, the queue constructor allows an optional parameter to set the maximum size of the queue. The array as created is actually large enough to hold one element more than the queue will allow, so that empty queues

```
// Array-based queue implementation
template <typename E> class AQueue: public Queue<E> {
private:
  int maxSize;               // Maximum size of queue
  int front;                 // Index of front element
  int rear;                  // Index of rear element
  E *listArray;           // Array holding queue elements

public:
  AQueue(int size =defaultSize) {  // Constructor
    // Make list array one position larger for empty slot
    maxSize = size+1;
    rear = 0;  front = 1;
    listArray = new E[maxSize];
  }

  ~AQueue() { delete [] listArray; } // Destructor

  void clear() { rear = 0; front = 1; } // Reinitialize

  void enqueue(const E& it) {     // Put "it" in queue
    Assert(((rear+2) % maxSize) != front, "Queue is full");
    rear = (rear+1) % maxSize;       // Circular increment
    listArray[rear] = it;
  }

  E dequeue() {           // Take element out
    Assert(length() != 0, "Queue is empty");
    E it = listArray[front];
    front = (front+1) % maxSize;    // Circular increment
    return it;
  }

  const E& frontValue() const {  // Get front value
    Assert(length() != 0, "Queue is empty");
    return listArray[front];
  }

  virtual int length() const         // Return length
   { return ((rear+maxSize) - front + 1) % maxSize; }
};
```

Figure 4.26 An array-based queue implementation.

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