7 Queues

7.1 Introduction

Queues are what we usually refer to as lines, as in "please get in line for a free lunch." The essential features of a queue are that it is ordered and that access to it is restricted to its ends: things can enter a queue only at the rear and leave the queue only at the front.

> **Queue**: A dispenser holding a sequence of elements that allows insertions only at one end, called the **back** or **rear**, and deletions and access to elements at the other end, called the **front**.

Queues are also called first-in-first-out, or FIFO, lists. Queues are important in computing because of the many cases where resources provide service on a first-come-first-served basis, such as jobs sent to a printer, or processes waiting for the CPU in a operating system.

7.2 The Queue ADT

Queues are containers and they hold values of some type. We must therefore speak of the ADT *queue of T*, where *T* is the type of the elements held in the queue. The carrier set of this type is the set of all queues holding elements of type *T*. The carrier set thus includes the empty queue, the queues with one element of type *T*, the queues with two elements of type *T*, and so forth. The essential operations of the type are the following, where *q* is a queue of *T* and *e* is a *T* value.

> *enter*(*q,e*)—Return a new queue just like *q* except that *e* has been added at the rear of *q*.
>
> *leave*(*q*)—Remove the front element of *q* and return the resulting (shorter) queue. The precondition of the *leave()* operation is that *q* is not empty.
>
> *empty?*(*q*)—Return the Boolean value true just in case *q* is empty.
>
> *front*(*q*)—Return the front element of *q* without removing it. Like *leave*(), this operation has the precondition that *q* is not empty.

The queue ADT operations thus transform queues into one another. When implemented, a single data structure stores the values in the queue, so there is generally no need to return queues once they have been changed. Hence the operations of a queue type usually have slightly different parameters and return types when implemented, as we will see next.

7.3 The Queue Interface

A Queue interface is a sub-interface of Dispenser, which is a sub-interface of Container, so it already contains an empty?() operation inherited from Container. The Queue interface need only add operations for entering elements, removing elements, and peeking at the front element of the queue. The diagram in Figure 1 shows the Queue interface.

Note that a generic or template is used to generalize the interface for any element type. Note also that preconditions have been added for the operations that need them.

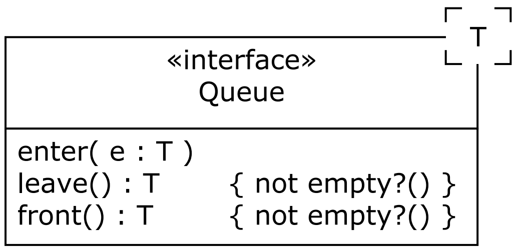

Figure 1: The Queue Interface

What do you want to do?
No matter what you want out of your future career, an employer with a broad range of operations in a load of countries will always be the ticket. Working within the Volvo Group means more than 100,000 friends and colleagues in more than 185 countries all over the world. We offer graduates great career opportunities – check out the Career section at our web site www.volvogroup.com. We look forward to getting to know you!
VOLVO
AB Volvo (publ)
www.volvogroup.com
Volvo Trucks I Renault Trucks I Mack Trucks I Volvo Buses I Volvo Construction Equipment I Volvo Penta I Volvo Aero I Volvo IT
Volvo Financial Services I Volvo 3P I Volvo Powertrain I Volvo Parts I Volvo Technology I Volvo Logistics I Business Area Asia

7.4 Using Queues—An Example

When sending a document to a printer, it may have to be held until the printer finishes whatever job it is working on. Holding jobs until a printer is free is generally the responsibility of a print spooler (a program that manages the input to a printer). Print spoolers hold jobs in a Queue until the printer is free. This provides fair access to the printer and guarantees that no print job will be held forever. The pseudocode in Figure 2 describes the main activities of a print spooler.

```
Queue(Job) queue;

def spool(Document d) {
  queue.enter(Job.new(d))
}

def run {
  while (true) {
   if (printer.isFree && !queue.empty?)
     printer.print(queue.leave)
  }
}
```

Figure 2: Using A Queue to Spool Pages for Printing

The print spooler has a job queue stored in the variable `queue`. A client can ask the spooler to print a document for it using the `spool()` operation and the spooler will add the document to its job queue. Meanwhile, the spooler is continuously checking the printer and its job queue; whenever the printer is free and there is a job in `queue`, it will remove the job from `queue` and send it to the printer.

7.5 Contiguous Implementation of the Queue ADT

There are two approaches to implementing the carrier set for the queue ADT: a contiguous implementation using arrays, and a linked implementation using singly linked lists; we consider each in turn.

Implementing queues of elements of type *T* using arrays requires a *T* array to hold the contents of the queue and some way to keep track of the front and the rear of the queue. We might, for example, decide that the front element of the queue would always be stored at location 0, and record the size of the queue, implying that the rear element would be at location size-1. This approach requires that the data be moved forward in the array every time an element leaves, which is not very efficient.

A clever solution to this problem is to allow the data in the array to "float" upwards as elements enter and leave, and then wrap around to the start of the array when necessary. It is as if the locations in the array are in a circular rather than a linear arrangement. Figure 3 illustrates this solution. Queue elements are held in the store array. The variable frontIndex keeps track of the array location holding the element at the front of the queue, and count holds the number of elements in the queue. The capacity is the size of the array and hence the number of elements that can be stored in the queue.

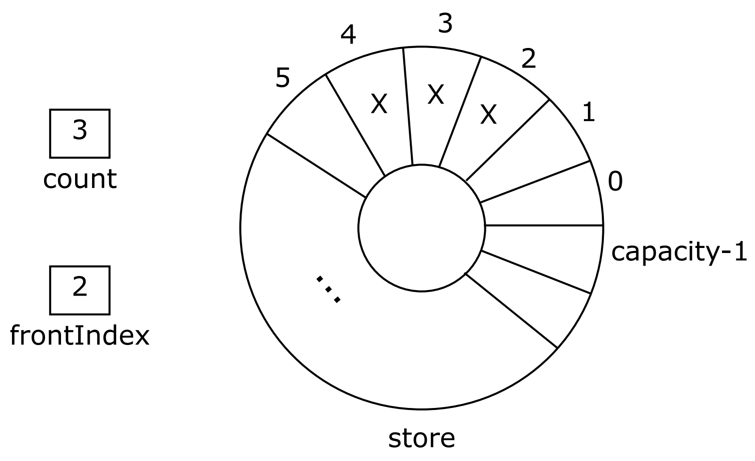

Figure 3: Implementing a Queue With a Circular Array

In Figure 3, data occupies the regions with an X in them: there are three elements in the queue, with the front element at store[2] (as indicated by the frontIndex variable) and the rear at store[4] (because frontIndex+count-1 is 4). The next element entering the queue would be placed at store[5]; in general, elements enter the queue at

store[(frontIndex+count) % capacity]

The modular division is what makes the queue values wrap around the end of the array to its beginning. This trick of using a circular array is the standard approach to implementing queues in contiguous locations.

If a static array is used, then the queue can become full; if a dynamic array is used, then the queue is essentially unbounded. As we have mentioned before, resizing an array is an expensive operation because new space must be allocated, the contents of the array copied, and the old space deallocated, so this flexibility is acquired at a cost. Care must also be taken to move elements properly to the expanded array—remember that the front of the queue may be somewhere in the middle of the full array, with elements wrapping around to the front.

Implementing the operations of the queue ADT using this data structure is quite straightforward. For example, to implement the leave() operation, a check is first made that the precondition of the operation (that the queue is not empty) is not violated by testing whether count equals 0. If not, then the value at store[frontIndex] is saved in a temporary variable, frontIndex is set to (frontIndex+1) % capacity, count is decremented, and the value stored in the temporary variable is returned.

A class realizing this implementation might be called ArrayQueue(T). It would implement the Queue(T) interface and have the store array and the frontIndex and count variables as private attributes. The queue operations would be public methods. Its constructor would create an empty queue. The value of capacity could be a constant defined at compile time, or a constructor parameter (if dynamic arrays are available).

7.6 Linked Implementation of the Queue ADT

A linked implementation of a queue ADT uses a linked data structure to represent values of the ADT carrier set. A singly linked list is all that is required, so list nodes need only contain a value of type *T* and a link to the next node. We could keep a reference to only the head of the list, but this would require moving down the list from its head to its tail whenever an operation required manipulation of the other end of the queue, so it is more efficient to keep a reference to each end of the list. We will thus use both frontNode and rearNode references to keep track of both ends of the list.

If rearNode refers to the head of the list and frontNode to its tail, it will be impossible to remove elements from the queue without walking down the list from its head; in other words, we will have gained nothing by using an extra reference. Thus we must have frontNode refer to the head of the list, and rearNode to its tail. Figure 4 illustrates this data structure. Each node has an item field (for values of type *T*) and a next field (for the links). The figure shows a queue with three elements. The queue is empty when the frontNode and rearNode references are both nil; the queue never becomes full (unless memory is exhausted).

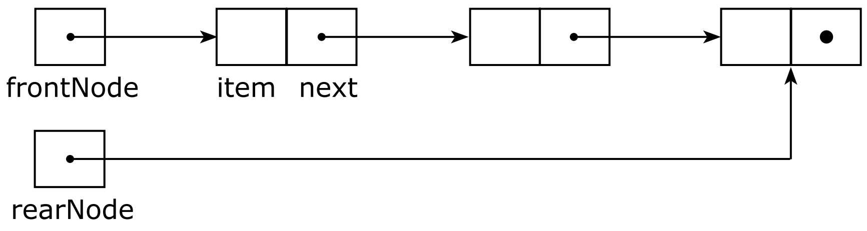

Figure 4: Implementing a Queue With a Linked List

Implementing the operations of the Queue ADT using a linked data structure is quite simple, though some care is needed to keep frontNode and rearNode synchronized. For example, to implement the leave() operation, a check is first made that the queue is not empty. If not, then the item field of the front node is assigned to a temporary variable. The frontNode variable is then assigned the next field of the first node, which removes the first node from the list. If frontNode is nil, then the list has become empty, so rearNode must also be assigned nil. Finally, the value saved in the temporary variable is returned.

A class realizing this implementation might be called LinkedQueue(T). It would implement the Queue(T) interface and have frontNode and rearNode as private attributes. It might also have a private inner Node class for node instances. Its constructor would create an empty queue. As new nodes are needed when values enter the queue, new Node instances would be allocated and used in the list.

7.7 Summary and Conclusion

Both queue implementations are simple and efficient, but the contiguous implementation either places a size restriction on the queue or uses an expensive reallocation technique if a queue grows too large. If contiguously implemented queues are made extra large to make sure that they don't overflow, then space may be wasted.

A linked implementation is essentially unbounded, so the queue never becomes full. It is also very efficient in its use of space because it only allocates enough nodes to store the values actually kept in the queue. Overall, the linked implementation of queues is preferable to the contiguous implementation.

7.8 Review Questions

1. Which operations of the queue ADT have preconditions? Do these preconditions translate to the Queue interface?
2. Why should storage be thought of as a circular rather than a linear arrangement of storage locations when implementing a queue using contiguous memory locations?
3. Why is there a reference to both ends of the linked list used to store the elements of a queue?

7.9 Exercises

1. In the contiguous storage implementation of a queue, is it possible to keep track of only the location of the front element (using a variable `frontIndex`) and the rear element (using a variable `rearIndex`), with no `count` variable? If so, explain how this would work.
2. Suppose that an `ArrayQueue` is implemented so that the array is reallocated when a client attempts to `enter()` an element when the array is full. Assume that the reallocated array is twice as large as the full array, and write Ruby code for the `enter()` operation that includes arranging the data where it needs to be in the newly allocated array.
3. Write a class invariant for a `LinkedQueue` class whose attributes are `frontNode`, `rearNode`, and `count`.
4. Write a version of the `LinkedQueue` class that does not have a `count` attribute.
5. A **circular singly linked list** is a singly linked list in which the last node in the list holds a references to the first element rather than nil. It is possible to implement a `LinkedQueue` efficiently using only a single reference into a circular singly linked list rather than two references into a (non-circular) singly linked list as we did in the text. Explain how this works and write Ruby code for the `enter()` and `leave()` operations to illustrate this technique.
6. A `LinkedQueue` could be implemented using a doubly-linked list. What are the advantages or disadvantages of this approach?
7. Implement the `Queue` interface and the `ArrayQueue` class in Ruby.
8. Implement the `LinkedQueue` class in Ruby.

7.10 Review Question Answers

1. The *leave*() and *front*() operations both have as their precondition that the queue *q* not be empty. This translates directly into the precondition of the `leave()` and `front()` operations of the `Queue` interface that the queue not be empty.
2. If storage is linear, then the data in a queue will "bump up" against the end of the array as the queue grows, even if there is space at the beginning of the array for queue elements. This problem can be solved by copying queue elements downwards in the array (inefficient), or by allowing the queue elements to wrap around to the beginning of the array, which effectively treats the array as a circular rather than a linear arrangement of memory locations.
3. In a queue, alterations are made to both ends of the container. It is not efficient to walk down the entire linked list from its beginning to get to the far end when an alteration must be made there. Keeping a reference to the far end of the list obviates this inefficiency and makes all queue operations very fast.