Formal Specification, Design and Implementation of the Circular Queue Data Structure

M. Devi Prasad
School of Information Sciences, Manipal University.
devi.prasad@manipal.edu

Abstract

We present a formal specification of the *Queue* data structure. First, we describe the essential properties of the queue abstract data type. We subsequently develop a behavioral specification of the ADT. This step introduces a few design decisions that influence the representation and efficiency of operations on queues. This, in turn, helps in shaping a concrete implementation in the C programming language. This entire exercise demonstrates how formal specification may guide different design choices, and thus lead to different implementations.

1 Introduction

Most textbooks employ a pattern in describing data structures: first informally describe the behavior of a particular data structure, and follow it up with an implementation in a high level programming language or pseudo code. The informal descriptions use diagrams, operation traces and textual narratives. The time and space complexity of algorithms, and implementation efficiency are discussed to some extent in many books. THe depth and breadth of exposition depends on factors such as author's predispositions and the target audience.

We wish to take a reasonably different approach to the way data structures course is taught. We wish to bring a little more engineering discipline to describing and developing data structures: we first bring out a specification, in some detail, of the interfaces of the underlying ADT. We follow it up with the design of the representation and associated behaviors of the data structure. We finally implement at the least two different versions in order to understand the pros and cons of different implementation strategies.

2 Notation and Conventions

We employ sets, sequences and tuples to develop an expressive formal specification of the queue data structure. We use the following notation:

Sequence	$[a_1, ..., a_n]$	*where* $0 \leq n$
n-tuple	$(a_1, a_2, ..., a_n)$	

In the following sections, we will be using sequences to represent the actual data elements held by queue instances. An empty queue content is represented as $[]$ and a non-empty sequence of queue elements is shown as $[a_1, ...a_n]$.

We write $d :: s$ to mean a sequence of at least one element with item 'd' at the head of the sequence; 's' in this case represents the rest of the sequence which may be empty. The expression $s :: d$ represents a sequence ending with an item 'd'.

We use two auxiliary functions, *abs* and *len*; the former computes the absolute value while the latter determines the length of a sequence.

Certain queue operations defined in the next section employ 2-tuples to represent the `Result` of their execution. A 2-tuple of type $\mathbb{R} \times \mathbb{N}$ is used to hold the value ($\mathbb{R}$ component) and the (potential error) status `OK`, `FULL`, and `EMPTY`.

3 The Definition of Queue and Result Interfaces

Queues are 'First In First Out' data structures, in contrast to stacks, which are 'Last In First Out' data structures. In order to ensure that the first element to enter the queue is the first one to exit, we designate two independent positions in a queue - a *head* and a *tail*. New elements are added to the queue at the tail position, and items are removed from the head position.

In order to simplify bookkeeping, and to allow the queue to be filled to its capacity, we maintain a running count of the items in the queue. This simplifies reasoning and improves understandability of the specification, design, and code.

The queue interface consists of a constructor function, two inspector functions, and two procedures that update the state of an instance. We define an auxiliary interface named 'Result' to represent either the answer or the error (status) returned by certain operations.

3.1 The Queue Interface

While instantiating a queue object, we indicate the desired length of the queue.

$$\textbf{new} : \mathbb{N} \rightarrow \text{Queue}$$

The design as well as the implementation may choose to impose a limit on the permitted queue length. The specified length value (passed to textbfnew) will be stored as a part of queue's internal representation.

Queues support two operations: **add** and **remove** which operate on the *tail* and *head* of the queue, respectively. New elements are added to the queue at the tail end.

Elements are removed from the head of the queue.

add : Queue $\times \mathbb{R} \rightarrow$ Queue $\times$ Result

remove : Queue $\times \mathbb{R} \rightarrow$ Queue $\times$ Result

When the queue is full, **add** records the status, FULL, in the 'status' component of the Result tuple. When the queue is empty, **remove** operation records EMPTY in the Result tuple.

A couple of self-explanatory inspector functions are handy. Some programs may prefer to check the state of the queue before attempting other operations.

empty : Queue $\rightarrow$ Boolean

full : Queue $\rightarrow$ Boolean

3.2 The Result Interface

The **Result** interface has a constructor, and value and status component extractors:

result : $\mathbb{R} \times$ Status $\rightarrow$ Result

val : Result $\rightarrow \mathbb{R}$

status : Result $\rightarrow$ Status

Status is a set of three values representing the state after performing an operations on a queue instance:

Status $= \{OK, EMPTY, FULL\}$

4 The Semantics of Queue Operations

While instantiating a new queue, there is a provision to set the actual length of the queue to a specific value. This value is passed as an argument to the operation. If this value exceeds a limit defined by a particular implementation, the size is set to that limiting value (referred to as MAX_QUEUE_LEN below).

We represent a queue instance by a 5-tuple. The first component represents the number of items this queue instance may hold. The second component represents the running count of items in the queue. The third and fourth components represents the head and tail index positions within the contents sequence. the The fifth component is the sequence representing the actual contents of the queue instance. When a queue object is freshly instantiated, its contents will be empty.

$$\textbf{new}(\texttt{size}) = \begin{cases} (size, 0, 0, 0, [\,]) & \text{if } \ 0 < size < \texttt{MAX\_QUEUE\_LEN}. \\ (\texttt{MAX\_QUEUE\_LEN}, 0, 0, 0, [\,]) & \text{otherwise.} \end{cases}$$

We employ circular queues for efficiency reasons. Both head and tail indices wrap around while removing items from, and adding items to the queue. This slightly complicates bookkeeping logic and the associated reasoning.

The following invariants hold for a queue instance:

$$(count = 0) \implies (head = tail)$$

$$(count = size) \implies (head = tail)$$

$$0 < count < size \implies tail = (head + count)\,\%\,size$$

It is easy to see that with slight modification, the third rule above can be made generic, and make it covers the first two cases. Therefore, we will retain only the following rule as the invariant.

$$0 \leq count \leq size \implies tail = (head + count)\,\%\,size$$

We define two state inspector operations, *empty* and *full*, by pattern-matching over the representation structure of queue instances. It is appropriate, at this point, to note that a freshly created queue instance is empty.

$$\textbf{empty}(\texttt{queue}) = \begin{cases} True & \text{if } \texttt{queue} = (size, 0, h, t, [\,]) \\ & \quad \wedge\ h = t \\ False & \text{otherwise.} \end{cases}$$

A queue instance is full when there is no room to hold a new element.

$$\textbf{full}(\texttt{queue}) = \begin{cases} True & \text{if } \texttt{queue} = (size, n, h, t, s) \\ & \quad \wedge\ n = len(s) \\ & \quad \wedge\ n = size. \\ & \quad \wedge\ h = t. \\ False & \text{otherwise.} \end{cases}$$

Attempt to add a new item to the queue will succeed if the queue is not already full. This is indicated by the OK status component of the answer. In the other case where queue is full, the status FULL is returned.

$$\textbf{add}(\texttt{queue}, \texttt{d}) = \begin{cases} ((size, n', \_, t', s'), (\_, OK)) & \text{if } \texttt{queue} = (size, n, \_, t, s) \\ & \quad \wedge\ n = len(s) \\ & \quad \wedge\ (n+1) \leq size \\ & \quad \wedge\ n' = (n+1) \\ & \quad \wedge\ t' = (t+1)\,\%\,size \\ & \quad \wedge\ s' = s :: d \\ (\texttt{queue},\ (\_, FULL)\ ) & \text{if } \texttt{queue} = (size, n, \_, \_, s) \\ & \quad \wedge\ n = len(s) \\ & \quad \wedge\ n = size. \end{cases}$$

We should convince ourselves that equational reasoning holds good if we substitute the expression for **full** (shown above) on the right-side conditionals of each case of the **push** operation.

The **remove** operation succeeds if the queue in not empty. Again, we should convince that equational reasoning holds if we plug **empty**, from its definition above, on the right side conditionals in the following:

$$\textbf{remove}(\texttt{queue}) = \begin{cases} ((size, n', h', \_, s'), (d, OK)) & \text{if } \texttt{queue} = (size, n, h, \_, s) \\ & \quad \wedge\ n = len(q) \\ & \quad \wedge\ n > 0 \\ & \quad \wedge\ n' = (n-1) \\ & \quad \wedge\ h' = (h+1)\,\%\,size \\ & \quad \wedge\ s = d :: s' \\ (\texttt{queue}, (\_, EMPTY)) & \text{if } \texttt{queue} = (size, 0, \_, \_, []). \end{cases}$$