Queues

1 Introduction

A queue is a very intuitive data type, especially among civilized societies. In the United States, people call "lines" what the British call *queues*. In the U.S., people stand "in line" for services such as purchasing a ticket for one thing or another, paying for merchandise, or boarding a train, bus or plane. The British stand in queues to do the same thing. What characterizes these queues is that arriving customers always go to the "end of the line" and the next customer to be served is always taken from the "front" of the line. As customers are served, the other customers steadily get closer to the front of the line, in the order in which they arrived. The idea that people are served in the order in which they arrive, or to put it another way, that *the first one in is the first one out*, is the notion of fairness implicit in queues.

Formally, in computer science terminology, a queue is a list in which all insertions take place at the end of the list and all deletions and accesses take place at the front of the list. The end of the list is called the ***rear***, or sometimes the ***back***, of the queue. The front keeps its name. Because this leads to a first-in, first-out behavior, a queue is known as a ***FIFO list***.

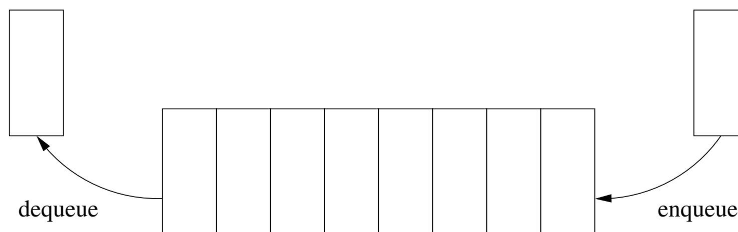

Figure 1: Inserting into and deleting from a queue

2 The Queue ADT

Operations on queues are analogous to operations on stacks. There is a one-to-one correspondence between them. The only difference is that the push operation of the stack appends an item to the front of the stack (which we call the top), whereas the ***enqueue*** operation appends to the rear. Popping a queue is called ***dequeuing*** the queue. Other than its having a different name, dequeuing a queue is the same as popping a stack. The single difference between stacks and queues, namely which end of the list new items are inserted, has a major consequence in terms of how the queue abstract data type behaves. See Figure 1.

The queue methods are:

create() - Create an empty queue.

destroy() - Destroy a queue.

bool empty() - Return true if the queue is empty and false if not.

bool enqueue([in] item) - Append an item to the rear of the queue, returning true if successful, false if not.

bool dequeue([out] item) - Remove the item from the front of the queue, and provide it to the caller, returning true if successful and false otherwise.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

bool dequeue() - Remove the item from the front of the queue, returning true if successful and false otherwise.

item front() - Return the item at the front of the queue to the caller without removing it

size_type size() - Return the number of items in the queue.

Notes.

- There is no access to an item that is not at the front.
- The `dequeue()` operation is analogous to the stack's `pop()` operation, and it is convenient to have two forms of it: one that removes the front item without retrieving it, and another that removes it but also copies it into an out parameter to make it available to the caller.
- `enqueue()` is called `push` in the C++ standard queue template class.
- `dequeue()` is called `pop` in the C++ standard queue template class.
- The `front()` operation returns the item at the front of the queue. This is the way that most people expect it to be. The problem is how to deal with calling `front()` when the queue is empty.

3 Refining the Queue ADT

The above descriptions are informal. We now provide a UML description of the queue abstraction, continuing to use our names for the enqueue and dequeue operations.

```
+empty():boolean  {query}
// returns true if the queue is empty, false if not

+enqueue([in] new_item:QueueItemType): boolean throw queue_exception
// appends new_item to the rear of the queue, returning true if successful
// if it fails, it throws a queue_exception and also returns false

+dequeue(): boolean  throw queue_exception
// removes the front element from the queue, returning true if successful
// if it fails, it throws an exception and returns false

+dequeue([out] front_item:QueueItemType) throw queue_exception
// same as dequeue() above but stores the item removed into front_item,
// throwing an exception if it fails

+front():QueueItemType {query} throw queue_exception
// returns  the item at the front of the queue,
// throwing an exception if it fails.
```

As with stacks, these descriptions are an abstraction; a specific interface may choose, for example, to define `front()` so that it passes the item from the front as a parameter rather than returning it. In this way it could return a boolean to indicate success or failure.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

4 The C++ Queue Template Class Interface

C++ provides a `queue` class template, which can be accessed by including the `<queue>` header file in the application. It makes the following methods available:

```
bool empty() const;
size_type size() const;
value_type& front();
value_type& back();
void push(const value_type& _X);
void pop();
```

Notes.

- As noted above, there is no enqueue or dequeue operation; these are named `push` and `pop` respectively. Do not confuse them with the stack's operations. They are functionally the same as enqueue and dequeue. C++ implements queues and stacks as classes that encapsulate a more basic container class template and which restrict access to the embedded container. Classes whose implementations are built on broader, more general classes but which provide restricted methods are called ***adaptor containers***.
- Notice that this interface includes a `back()` method, which accesses the item in the rear of the queue. This is not consistent with the definition of a queue as an abstract data type. You will often encounter queue implementations that provide methods other than those that define the queue as a data abstraction. For example, you maye also find operations that can remove elements from the interior of a queue, which is a gross violation of what makes it a queue.
- Notice too that the `front()` and `back()` methods return a reference to the item at the respective locations. This implies that the caller can modify these elements within the queue. The interface de defined above returns by value, not by reference.

5 Queue Applications

Queues have many applications in computer science, partly because they act as first-in-first-out ***buffers***. A buffer can be thought of as a storage area for data that is in a state of transit. There are many circumstances in which one process generates data for another process, but the two processes run independently and at possibly different rates of speed. The process that generates the data puts it into a buffer, and the one that uses that data removes it from the buffer in the order in which it was placed. For example, when you burn a music CD with data from a hard drive, one process delivers that data to the CD burner, and the CD burner burns it to the CD. If the process reading from the hard drive and sending the data sends it too quickly and the CD burner is busy burning data, the new data would be lost if there were no buffer for it. On the other hand, if the data is delivered too slowly and there is no buffered data, the CD burner may reach the next track with no data to burn and will create gaps inadvertently. The buffer allows the two processes to have intermittent pauses or bursts without resulting in missing or duplicated final data. If the buffer is not large enough, there can still be overruns or underruns that could lead to failure in the burning process, depending on how the two processes are designed.

Almost all services within the machine use a queue for their waiting tasks. The queue acts as a buffer of jobs that have yet to be completed. In particular,

- the printer spooler stores print jobs in a queue;
- the various network servers maintain queues of pending service requests; and

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

- the operating system maintains a complex of queues for a variety of services, such as access to the CPU, allocation of memory or disk space, and even the processing of keyboard characters delivered to the machine from a keyboard device.

Queues are also used in applications outside of computer science, especially in simulations of all kinds. One can use a queue to model air traffic at an airport, vehicular traffic at a toll plaza, or customers waiting for available cashiers at a retail store, for example.

5.1 Example Application

An application lets a user enter three commands on the keyboard interactively: `e` to enter data items, `p` to print the data items entered so far, 4 per line, and `q`, to quit. The data items are positive integers. There is no limit to the number of items they can enter at a time; they stop entering by typing `-1`. When the print command is issued, all items entered but not yet displayed are printed. Because this is a buffering problem, a queue can be used.

The program will use the C++ `queue` class, instantiating it to hold items of type `data`, which is `int` in this case.

```
int main()
{
    typedef  int  data;
    queue<data>   buffer;
    data          item;
    int           count;
    char       c;
    bool          done = false;
    while ( !done ) {
        cout << “Enter command:”;
        cin  >> c;
        switch ( c ) {
        case ‘e’:
            cout << "Enter positive numbers;
                 << terminate with -1:\n";
            while ( cin >> item ) {
                if ( item != -1 )
                    buffer.push(item);
                else
                    break;
            }
            break;
        case ‘p’:
            count = 4;
            while (!buffer.empty() ) {
                if (count == 4) {
                    count = 0;
                    cout << endl;
                }
                cout << buffer.front() << “ “;
                buffer.pop();
                count++;
            }
            cout << endl;
            break;
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

```
        case 'q':
            done = true;
            break;
        }
    }
}
```

6 Queue Implementations

This time we relied on an existing implementation of a queue to solve a problem, namely the one from the C++ library. Once again, the power of data abstraction is demonstrated. But you should know how to implement a queue, because it is an important data structure. To start, we define the interface we will implement and a class of queue exceptions that can be thrown as needed. If you are not familiar with exceptions, refer to the notes on exception handling.

The following exception class will be used by all queue implementations.

```
#include <stdexcept>
#include <string>
using namespace std;
class queue_exception: public logic_error
{
public:
    queue_exception(const string & message="")
         : logic_error(message.c_str())    {}
};
```

The `logic_error` exception is defined in `<stdexcept>`. Its constructor takes a string, which can be printed using the `what()` method of the class. The next logical step would be to derive separate exception types such as `queue_overflow`, or `queue_underflow`, as illustrated below.

```
class queue_overflow: public queue_exception
{
public:
    queue_overflow(const string & message=""): queue_exception(message) {}
};

class queue_underflow: public queue_exception
{
public:
    queue_underflow(const string & message=""): queue_exception(message) {}
};
```

The following is the interface we will implement:

```
typedef data_item QueueItemType;
class Queue
{
public:
    // constructors and destructor:
    Queue();                // default constructor
    Queue(const Queue &);   // copy constructor
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

```
        ~Queue();              // destructor

        // queue operations:
        bool empty() const;
        // Determines whether a queue is empty.
        // Precondition: None.
        // Postcondition: Returns true if the queue is empty;
        // otherwise returns false.

        int size() const;
        // returns the size of the queue.
        // Precondition: None.
        // Postcondition: None

        void enqueue(const QueueItemType& new_item) throw(queue_exception);
        // Adds an item to the rear of a queue.
        // Precondition: new_item is the item to be added.
        // Postcondition: If the insertion is successful, new_item
        // is at the rear of the queue.
        // Exception: Throws queue_exception if the item cannot
        // be placed in the queue.

        void dequeue() throw(queue_exception);
        // Removes the front of a queue.
        // Precondition: None.
        // Postcondition: If the queue is not empty, the item
        // that was enqueued least recently is removed. However, if
        // the queue is empty, deletion is impossible.
        // Exception: Throws queue_exception if the queue is empty.

        void dequeue(QueueItemType& front_item) throw(queue_exception);
        // Retrieves and removes the front of a queue.
        // Precondition: None.
        // Postcondition: If the queue is not empty, front_item
        // contains the item that was enqueued least recently and the
        // item is removed. However, if the queue is empty,
        // deletion is impossible and front_item is unchanged.
        // Exception: Throws queue_exception if the queue is empty.

        QueueItemType front() const  throw(queue_exception);
        // Retrieves the front of a queue.
        // Precondition: None.
        // Postcondition:
        // Returns: If the queue is not empty, front_item
        //          otherwise, the return value is defined and a
        //          queue_exception is thrown.

    private:
       ...
    };
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

Notes.

- There is very little difference between a stack and a queue with respect to the potential implementations, and the advantages and disadvantages are the same.
- Since a queue is just a special type of list, a queue can be implemented with a list. In this case, you can make a list a private member of the class and implement the queue operations by calling the list methods on the private list.
- It is more efficient to implement a queue directly, avoiding the unnecessary function calls. Direct methods include using an array, or using a linked representation, such as a singly-linked list.
- A queue can be implemented as an array, but not in the manner you might first envision. The trick is to implement a circular array and put the queue within it. This will make the `enqueue()` and `dequeue()` operations efficient, but still the array size might be exceeded. This can be handled by a resizing operation when this happens.
- A queue implemented as a linked list overcomes the problem of using a fixed size data structure. It is also relatively fast, since the `enqueue()` and `dequeue()` operations do not require many pointer manipulations. The storage is greater because the links take up four bytes each for every queue item.

6.1 Linked Implementation

We begin with the linked implementation. LIke that of the stack, a linked list implementation of a queue does not pre-allocate storage; it allocates its nodes as needed. Each node will have a data item and a pointer to the next node. Because the linked list implementation hides the fact that it is using linked nodes, the node structure is declared within the private part of the class. The private data members of the `Queue` class include a pointer to the node at the front of the queue and a pointer to the node at the rear of the queue. If the queue is empty, the front pointer and rear pointer are both `NULL`. The private part of the `Queue` class would therefore be:

```
private:
   struct QueueNode
   {
      QueueItemType item;
      QueueNode    *next;
   };
   QueueNode *front_p;     // pointer to front of queue
   QueueNode *rear_p;      // pointer to rear of queue
   int       num_items;    // to keep track of the size
};
```

Remarks.

- The constructor sets `front_p` and `rear_p` to `NULL` and `num_items` to 0; the test for emptiness can check whether either is `NULL` or num_items is zero.
- The copy constructor is similar to that of the stack; it has to traverse the linked list of the queue passed to it. It copies each node from the passed queue to the queue being constructed. It allocates a node, fills it, and attaches it to the preceding node. We could implement the copy constructor by repeated calls to `enqueue()`, but it is faster to implement it directly.
- Because the queue allocates its storage dynamically, it must have a user-defined, deep destructor. The destructor repeatedly calls `dequeue()` to empty the list.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

- The `enqueue()` operation inserts a node at the rear of the queue. There is a special case when the list is empty because the `front_p` pointer has to be set in this case.
- The two `dequeue()` operations remove the node at the front of the queue, throwing an exception if the queue is empty. One provides this item in the parameter. For simplicity, the latter is implemented by calling the former.
- `front()` simply returns the item at the front of the queue, throwing an exception if the queue is empty.
- In theory it is possible that the operating system will fail to provide additional memory to the object when it calls `new()` in both the copy constructor and the `enqueue()` method. This implementation shows how to handle this within a `try` block.

```
// constructor
Queue::Queue() : front_p(NULL), rear_p(NULL), num_items(0) { }

// copy constructor
Queue::Queue(const Queue& aQueue)
{
    if (aQueue.num_items == 0) {
        front_p = NULL;
        rear_p  = NULL;
        num_items = 0;
    }
    else {
        // set num_items
        num_items = aQueue.num_items;
        // copy first node
        front_p       = new QueueNode;
        front_p->item = aQueue.front_p->item;
        rear_p        = front_p;

        // copy rest of queue
        QueueNode *newPtr = front_p;   // new list pointer
        QueueNode *origPtr = aQueue.front_p; // start at front of list

        while ( origPtr != aQueue.rear_p ) {
            origPtr      = origPtr->next;  // move to next node
            newPtr->next = new QueueNode;  // create new node in new queue
            newPtr       = newPtr->next;   // advance in new queue
            newPtr->item = origPtr->item;  // fill with item
        }
        rear_p       = newPtr;  // set rear to last node inserted
        rear_p->next = NULL;    // set rear node's next to NULL
   }
}

// destructor, which empties the list by calling dequeue() repeatedly
Queue::~Queue()
{
    // dequeue until queue is empty
    while ( num_items > 0 )
        dequeue();
}
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

```
bool Queue::empty() const
{
    return front_p == NULL;
}

bool Queue::size() const
{
    return num_items;
}

void Queue::enqueue(const QueueItemType& newItem) throw(queue_exception)
{
    try {
        // create a new node
        QueueNode *newPtr = new QueueNode;

        // set data portion  of new node
        newPtr->item = newItem;
        newPtr->next = NULL;

        if ( 0 == num_items )
            front_p = newPtr;
        else
            // insert the new node at the rear of the queue
           rear_p->next = newPtr;

        // in either case set rear to point to new node and increment num_items
        rear_p = newPtr;
        num_items++;
    }
    catch (bad_alloc error)
    {
        throw queue_exception(“queue_exception: cannot alloc mem”);
    }
}

void Queue::dequeue() throw(queue_exception)
{
    if ( 0 == num_items )
        throw queue_exception("queue_exception: empty queue");
    else {
        // queue is not empty;
        QueueNode *temp = front_p;
        if ( front_p == rear_p ) {
            front_p = NULL;
            rear_p  = NULL;
        }
        else
            front_p = front_p->next;
        temp->next = NULL;
        delete temp;
        num_items--;
    }
}
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

```
void Queue::dequeue(QueueItemType& front_item) throw(queue_exception)
{
    if ( 0 == num_items )
        throw queue_exception("queue_exception: empty queue");

    else {
        // queue is not empty; retrieve front
        front_item = front_p->item;
        dequeue();
   }
}

void Queue::front(QueueItemType& front_item) const throw(queue_exception)
{
   if ( 0 == num_items )
      throw queue_exception("queue_exception: empty queue");
   else       // queue is not empty; retrieve front
      front_item = front_p->item;
}
```

6.2 Array Implementation

The first implementation that comes to mind, which we will call the ***naive implementation***, is very inefficient. In this implementation the idea would be to set the front of the queue to be index 0 and the rear to the highest index in use. Then enqueuing an item would increment `rear` and insert the item into that position in the array. But what about dequeuing an item?

We could increment `front` to dequeue an item, setting it to 1 instead of 0, then to 2 the second time, and so on. As we continue to dequeue the queue, the front will increase in value, which is called ***rightward drift***. Since enqueuing does not alter the value of `front`, it will steadily increase, eventually exceeding the maximum sixe of the array. This is therefore not a solution. In order for it to stay at index 0, we would have to shift all array items downward with each `dequeue()` operation, and change the value of the `rear` index as well. This means that the computational cost of a `dequeue()` operation is proportional to the number of items in the queue. This is not an acceptable solution.

You should see that an array implementation of a queue is not as simple as that of a stack. Do not be discouraged, however, because one small change will convert an array into a viable implementation of a queue. We will turn the array into a circular array. In a circular array, we think of the last array item as preceding the first item. To illustrate, suppose that the array is of size 8:

Imagine that we bend the array so that it forms a circle, with 0 followed by 7 in this case. In other words, after we reach the item with index 7, the next item after that will be the item with index 0. This is accomplished by using modulo arithmetic when indexing through the array; if `current_index` is an index that sequences through the array, then we would use the following to compute its next index:

```
current_index = (current_index +1) % array_size
```

or equivalently,

```
current_index = (current_index == array_size-1)? 0: current_index+1;
```

Let us assume that the array is named `items` and is of size `MAX_QUEUE`. Now imagine that we maintain two variables called `front_idx` and `rear`_idx, and that `rear_idx` is the index of the last item in the queue, and

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

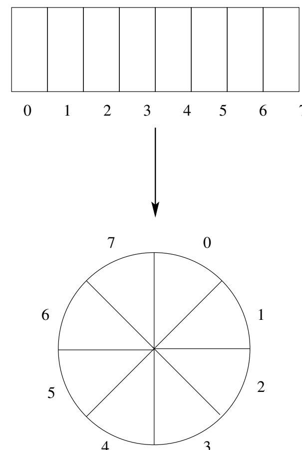

Figure 2: Circular array of size 8

`front_idx` is the index of the first item in the queue. Henceforth let us just say that `front_idx` "points to" the first item and `rear_idx` "points to" the last item. Then to enqueue an item, we would use the instructions

```
rear_idx = (rear_idx + 1) % MAX_QUEUE;
items[rear_idx] = item_to_insert;
```

If `rear_idx` had the value `MAX_QUEUE-1` beforehand, then it would have the value `0` after, since `MAX_QUEUE % MAX_QUEUE = 0`. This is how it behaves circularly. Similarly, to dequeue an item (from the front of the queue) and return it to the caller, we would use

```
front_item = items[front_idx];
front_idx = (front_idx + 1) % MAX_QUEUE;
```

Notice that neither `front_idx` nor `rear_idx` is ever decreased; they just march forward endlessly, but because their world is now round, they do not fall off of it.

The last problem is to decide on the initial values of the front and rear indices. We can arbitrarily set `front_idx` to the first index in the array, 0. It does not matter. What matters is where `rear_idx` is in relation to it. If the invariant assertion about `front_idx` and `rear_idx` are that `front_idx` points to the first item and `rear_idx` points to the last, when `front_idx == rear_idx`, this should mean that the queue has a single item in it. This in turn implies that when the queue is empty, `rear_idx` is the index before `front_idx`, or stated mathematically, `front_idx == (rear_idx + 1) % MAX_QUEUE`. Therefore, we set the initial value of `rear_idx` to `MAX_QUEUE-1`.

We will maintain the size of the queue in the `num_items` member variable as we did with the linked implementation, so the tests for emptiness remains the same. We will also need a test for fullness, because it is possible that the array reaches capacity. The test for fullness will be that `num_items == MAX_QUEUE`.

It is worth pointing out that when the queue is full, every array element contains an item. This implies that `rear_idx` is the index before `front_idx`, or that `front_idx == (rear_idx + 1) % MAX_QUEUE`. But this is exactly the same condition as occurs when the queue is empty. This is why it is especially important that we use the `num_items` variable to test whether the queue is empty or full.

With the preceding discussion in mind, the array-based queue implementation is given below.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

```
class Queue
{
public:
    // same interface as above plus
    bool full() const;

private:
    QueueItemType  items[MAX_QUEUE];   // array of MAX_QUEUE many items
    int            front_idx;          // index of front of queue
    int            rear_idx;           // index of rear of queue
    int            num_items;          // number of items in the queue
};
```

Remarks. The implementation is very straightforward:

- Because the array is a statically allocated structure, the copy constructor and destructor are simple and omitted here.
- The test for emptiness when the counter variable `num_items` is present reduces to `num_items == 0`.
- The remaining operations are all as described above.

The implementation:

```
Queue::Queue(): front_idx(0), rear_idx(MAX_QUEUE-1), num_items(0) { }
 // end default constructor

bool Queue::empty() const
{
    return num_items == 0);
}

bool Queue::full() const
{
    return num_items == MAX_QUEUE);
}

void Queue::enqueue(QueueItemType new_item) throw(queue_exception)
{
   if (num_items == MAX_QUEUE)
       throw queue_exception("queue_exception: queue full on enqueue");
   else {
       // queue is not full; insert item
       rear_idx = (rear_idx+1) % MAX_QUEUE;
       items[rear_idx] = new_item;
       ++num_items;
   }
}

void Queue::dequeue() throw(queue_exception)
{
    if ( 0 == num_items )
        throw queue_exception("queue_exception: empty queue, cannot dequeue");
    else {
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

```
        // queue is not empty; remove front
        front_idx = (front_idx+1) % MAX_QUEUE;
        --num_items;
    }
}

void Queue::dequeue(QueueItemType& queueFront) throw(queue_exception)
{
    if ( 0 == num_items )
        throw queue_exception("queue_exception: empty queue, cannot dequeue");
    else {
        // queue is not empty; retrieve and remove front
        queueFront = items[front_idx];
        front_idx  = (front_idx+1) % MAX_QUEUE;
        --num_items;
    }
}

void Queue::front(QueueItemType& queueFront) const throw(queue_exception)
{
    if ( 0 == num_items )
        throw queue_exception("queue_exception: empty queue, cannot get front");
    else
        // queue is not empty; retrieve front
        queueFront = items[front_idx];
}
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.