Chapter 8: Queues

Exercises 8.2

1. 

```
    myFront   myArray   myBack
 0   1   2   3   4
  A   B   C   A   ?
```

```
    myFront   myArray   myBack
 0   1   2   3   4
  X   Y   Z   ?   ?
```

Queue is now empty

3. 

```
    myFront   myArray   myBack
 0   1   2   3   4
  s   r   q   r   r
```

```
    myFront   myArray   myBack
 0   1   2   3   4
  C   A   A   B   B
```

Error occurs when i = 4. After ch = 'A' is inserted in location 2, myBack is 3 and myFront is 4, which means the queue is full, so the next enqueue() operation fails.

5. /*-- DQueue.h ------------------------------------------------------
This header file defines a Queue data type.
Basic operations:
  constructor:      Constructs an empty queue
  copy constructor: Constructs a copy of a queue
  =:                Assignment operator
  destructor:      Destroys a queue
  empty:            Checks if a queue is empty
  enqueue:          Modifies a queue by adding a value at the back
  front:            Accesses the top stack value; leaves queue unchanged
  dequeue:          Modifies queue by removing the value at the front
  display:          Displays all the queue elements
-------------------------------------------------------------------*/

#include <iostream>

#ifndef DQUEUE
#define DQUEUE

typedef int QueueElement;
#endif
class Queue
{
public:
    /** Function members *****/
    /** Constructors *****/
    Queue(int numElements = 128);
    /************************************************************
    Construct a Queue object.
    Precondition: None.
    Postcondition: An empty Queue object has been constructed
    (myFront and myBack are initialized to 0 and myArray
    is an array with numElements (default 128) elements
    of type QueueElement).
    -----------------------------------------------*/

    Queue (const Queue & original);
    /************************************************************
    Copy Constructor
    Precondition: original is the queue to be copied and
    is received as a const reference parameter.
    Postcondition: A copy of original has been constructed.
    -----------------------------------------------*/

    /***** Destructor *****/
~Queue();
    /************************************************************
    Class destructor
    Precondition: None
    Postcondition: The dynamic array in the queue has been
    deallocated.
    -----------------------------------------------*/

    /***** Assignment *****/
const Queue & operator= (const Queue & rightHandSide);
    /************************************************************
    Assignment Operator
    Precondition: original is the queue to be assigned and
    is received as a const reference parameter.
    Postcondition: The current queue becomes a copy of
    original and a const reference to it is returned.
    -----------------------------------------------*/

    bool empty() const;
    /************************************************************
    Check if queue is empty.
    Precondition: None
    Postcondition: Returns true if queue is empty and
    false otherwise.
    -----------------------------------------------*/
void enqueue(const QueueElement & value);
	/**-------------------------------------------------------
	 Add a value to a queue.
	
Precondition:  value is to be added to this queue
Postcondition: value is added at back of queue provided
    there is space; otherwise, a queue-full message is
    displayed and execution is terminated.
--------------------------------------------------------*/

void display(ostream & out) const;
	/**-------------------------------------------------------
	 Display values stored in the queue.
	
Precondition:  ostream out is open.
Postcondition: Queue's contents, from front to back, have
    been output to out.
--------------------------------------------------------*/

QueueElement front() const;
	/**-------------------------------------------------------
	 Retrieve value at front of queue (if any).
	
Precondition: Queue is nonempty
Postcondition: Value at front of queue is returned, unless
    the queue is empty; in that case, an error message is
    displayed and a "garbage value" is returned.
--------------------------------------------------------*/

void dequeue();
	/**-------------------------------------------------------
	 Remove value at front of queue (if any).
	
Precondition: Queue is nonempty.
Postcondition: Value at front of queue has been removed,
    unless the queue is empty; in that case, an error
    message is displayed and execution allowed to proceed.
--------------------------------------------------------*/

private:
	/*** Data members ****/
    int myFront,  // front
    myBack;   //   and back of queue
    int myCapacity;  // capacity of queue
    QueueElement * myArray;  // dynamic array to store elements;
        // empty slot used to distinguish
        // between empty and full
};  // end of class declaration

#endif

/*-- DQueue.cpp----------------------------------------------------------
 This file implements Stack member functions.
 Empty slot used to distinguish between empty and full
-----------------------------------------------------------------------*/

/*-- DQueue.cpp----------------------------------------------------------
 This file implements Stack member functions.
 Empty slot used to distinguish between empty and full
-----------------------------------------------------------------------*/
```cpp
#include <iostream>
#include <cassert>
#include <new>
using namespace std;

#include "DQueue.h"

//--- Definition of Queue constructor
Queue::Queue(int numElements)
{
    assert (numElements > 0);  // check precondition
    myCapacity = numElements;  // set queue capacity
    // allocate array of this capacity
    myArray = new(nothrow) QueueElement[myCapacity];
    if (myArray != 0)          // memory available
        myFront = myBack = 0;
    else
    {
        cerr << "Inadequate memory to allocate queue \n"
             "-- terminating execution\n";
        exit(1);
    }                          // or assert(myArray != 0);
}

//--- Definition of Queue copy constructor
Queue::Queue(const Queue & original)
: myCapacity(original.myCapacity),
  myFront(original.myFront), myBack(original.myBack)
{
    //--- Get new array for copy
    myArray = new(nothrow) QueueElement[myCapacity];
    if (myArray != 0)          // check if memory available
        // copy original's array member into this new array
        for (int i = myFront; i!= myBack; i = (i + 1)%myCapacity)
            myArray[i] = original.myArray[i];
    else
    {
        cerr << "*Inadequate memory to allocate queue ***\n"
             exit(1);
    }
}

//--- Definition of Queue destructor
Queue::~Queue()
{
    delete [] myArray;
}

//--- Definition of assignment operator
const Queue & Queue::operator=(const Queue & rightHandSide)
{
    if (this != &rightHandSide)                // check that not st = st
    {
        //-- Allocate a new array if necessary
        if (myCapacity != rightHandSide.myCapacity)
        {
            delete[] myArray;                     // destroy previous array
            myCapacity = rightHandSide.myCapacity; // copy myCapacity
```
myArray = new QueueElement[myCapacity];
if (myArray == 0) // check if memory available
{
    cerr << "*** Inadequate memory ***\n";
    exit(1);
}

myFront = rightHandSide.myFront; // copy myFront member
myBack = rightHandSide.myBack; // copy myBack member
// copy queue elements
for (int i = myFront; i!= myBack; i= (i + 1)%myCapacity)
    myArray[i] = rightHandSide.myArray[i];

return *this;

//-- Definition of empty()
inline bool Queue::empty() const
{
    return myFront == myBack;
}

//-- Definition of enqueue()
void Queue::enqueue(const QueueElement & item)
{
    if ((myBack +1)% myCapacity == myFront)
        cerr << "Queue is full: cannot add to queue.  Error!! " << endl;
    else
    {
        myArray[myBack] = item;
        myBack = (myBack+ 1) % myCapacity;
    }
}

//-- Definition of front()
QueueElement Queue::front() const
{
    if (myFront == myBack)
    {
        cerr <<"Queue is empty: error!  Returning garbage value\n";
        QueueElement garbage;
        return garbage;
    }else
        return myArray[myFront];
}

//-- Definition of dequeue()
void Queue::dequeue()
{
    if (myFront == myBack)
        cerr <<"Queue is empty: cannot remove from queue: error!\n";
    else
        myFront= (myFront + 1) % myCapacity;
void Queue::display(ostream & out) const
{
    for (int i = myFront; i != myBack; i = (i + 1) % myCapacity)
        cout << myArray[i] << " ";
    cout << endl;
}

6. // Prototype:
bool full() const;
/*---------------------------------------------------------------------
Check if queue is full.
Precondition: None
Postcondition: Returns true if queue is full and false otherwise.
--------------------------------------------------------------------*/

// Definition:
bool Queue::full()
{
    return myFront == (myBack + 1)% QUEUE_CAPACITY;
}

// Definition:
bool Queue::full()
{
    return myFront == (myBack + 1)% myCapacity;
}

7. // Prototype:
int size() const;
/*----------------------------------------------------------------------
Find number of elements in the queue.
Precondition: None
Postcondition: Number of queue elements is returned.
----------------------------------------------------------------------*/

// Definition:
int Queue::size() const
{
    if (myFront == myBack)
        return 0;
    else if (myFront > myBack)
        return myBack - myFront + QUEUE_CAPACITY;
    else
        return myBack - myFront;
}

8. // Prototype
int size(Queue q);
/*----------------------------------------------------------------------
Find number of elements in a queue received as a value parameter.
Precondition: None
Postcondition: Number of queue elements is returned.
----------------------------------------------------------------------*/
// Definition
int size(Queue q)
{
    int count = 0;
    while (!q.empty())
    {
        q.removeQ();
        count++;
    }
    return count;
}

/* Here is a version that preserves the parameter q. */
int size(Queue q)
{
    Queue temp;
    int count = 0;
    while (!q.Empty())
    {
        temp.addQ(q.front());
        q.removeQ();
        count++;
    }
    while (!temp.empty())
    {
        q.addQ(temp.front());
        temp.removeQ();
    }
    return count;
}

9. // Prototype:
QueueElement back() const;
/*----------------------------------------------------------------------
Retrieve the back element of this queue.
Precondition: None
Postcondition: Back element of the queue is returned, unless there
was none, in which case a queue-empty message is displayed.
---------------------------------------------------------------------*/

// Definition:
QueueElement Queue::back() const
{
    if (myFront == myBack)
    {
        cerr << "Error: queue is empty -- returning garbage value\n";
        QueueElement garbage;
        return garbage;
    }
    //else
    if (myBack == 0)
        return myArray[QUEUE_CAPACITY - 1];
    //else
    return myArray[myBack - 1];
}
10.

// Prototype:
QueueElement back();

/*----------------------------------------------------------------------
Retrieve the back element of a queue received as a value parameter.
Precondition: None
Postcondition: Back element of the queue is returned, unless there
was none, in which case a queue-empty message is displayed.
--------------------------------------------------------------------*/

// Definition:
QueueElement back(Queue q)
{
  if (q.empty())
  {
    cerr << "Error: queue is empty -- returning garbage value\n";
    QueueElement garbage;
    return garbage;
  }
  else
  
  QueueElement last;
  while (!q.empty())
  {
    last = q.front();
    q.dequeue();
  }
  return last;
}

//-- Non-destructive version (preserves parameter q)
QueueElement back(Queue q) const
{
  if (q.empty())
  {
    cerr << "Error: queue is empty -- returning garbage value\n";
    QueueElement garbage;
    return garbage;
  }
  else
  
  Queue temp;
  QueueElement last;
  while (!q.empty())
  {
    last = q.front();
    temp.addQ(last);
    q.removeQ();
  }
  while (!temp.empty())
  {
    q.addQ(temp.front());
    temp.removeQ();
  }
  return last;
}
11.

// Prototype:
QueueElement nthElement(int n);
/*-------------------------------
   Retrieve the n-th element of a queue.
   Precondition:  1 <= n <= number of queue elements
   Postcondition: n-th element of the queue is returned, unless queue
                  has fewer than n elements, in which case an error message is
                  displayed. Also, the elements preceding the n-th element are
                  removed from the queue.
   */

// Definition:
QueueElement Queue::nthElement(int n)
{
    QueueElement elem;
    while( n > 0 && !empty())
    {
        elem = front();
        removeQ();
        n--;
    }
    if (n > 0)
    {
        cerr << "Error: insufficient number of elements in the queue\n";
        "-- returning garbage value\n";
        QueueElement garbage;
        return garbage;
    } //else
    return elem;
}

12.

// Prototype:
QueueElement nthElement(int n) const;
/*-------------------------------
   Retrieve the n-th element of a queue.
   Precondition:  1 <= n <= number of queue elements
   Postcondition: n-th element of the queue is returned, unless queue
                  has fewer than n elements, in which case an error message is
                  displayed.
   */

// Definition:
QueueElement Queue::nthElement(int n) const
{
    if (myFront < myBack && myBack - myFront < n
        || myFront > myBack && QUEUE_CAPACITY - (myFront - myBack) < n)
    {
        cerr << "Error: insufficient number of elements in the queue\n";
        "-- returning garbage value\n";
        QueueElement garbage;
        return garbage;
    }
//else
int index_n = (myFront + n - 1) % QUEUE_CAPACITY;
return myArray[index_n];
}

13. The algorithm is as follows:
   1. Create a stack.
   2. While the queue is not empty, do the following:
      a. Remove an item from the queue.
      b. Push this item onto the stack.
   3. While the stack is not empty, do the following:
      a. Pop an item from the stack.
      b. Add this item to the queue.

14. (a) For n = 3:
    Possible Permutations
    123 132 213 231 312
    Impossible Permutations
    321
    ====================================================
    (b) For n = 4:
    Possible Permutations
    1234 1324 1342 1423
    2134 2143 2314 2341 2413
    3124 3142 3412
    4123
    Impossible Permutations
    1243
    2431
    3214 3241 3421
    4132 4312 4321 4213 4231
    ====================================================
    (c) For n = 5:
    Possible Permutations
    12345 12354 12435 12453 12534
    13245 13254 13425 13452 13524
    14235 14253 14523
    15234
    Impossible Permutations
    12543
    13542
    14325 14352 14532
    15243 15324 15342 15423 15432
    21345 21354 21435 21453 21534
    23145 23154 23415 23451 23514
    24135 24153 24513
    25134
    31245 31254 31425 31452 31524
    32145 32154 32415 32451 32514 32541
    34125 34152 34512
    35124
    35142 35214 35241 35412 35421

    – 80 –
41235 41253 41523 41325 41352 41532
42135 42153 42315 42351 42513 42531
43125 43152 43215 43251 43512 43521
45123 45132 45213 45231 45312 45321
51234 51243 51324 51342 51423 51432
52134 52143 52314 52341 52413 52431
53124 53142 53214 53241 53412 53421
54123 54132 54213 54231 54312 54321

(d) The rule is: for each digit $d$ in the number, the digits to the right of $d$ that are less than $d$ MUST be in ascending order.

15.
/* Implementation of Queue class.
   Count of elements used to distinguish between empty and full
   Add a data member:  int myCount; to the private section of
   the Queue class declaration.
*/
#include <iostream>
using namespace std;

Queue::Queue(): myFront(0), myBack(0), myCount(0)
{}

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

void Queue::enqueue(const QueueElement & value)
{
    if (myCount < QUEUE_CAPACITY)
    {
        myArray[myBack] = value;
        myBack = (myBack + 1) % QUEUE_CAPACITY;
        myCount++;
    } else
    {
        cerr << "*** Queue full -- can't add new value ***\n"
            << "Must increase value of QUEUE_CAPACITY in Queue.h\n"
            << exit(1);
    }
}

QueueElement Queue::front()
{
    if (myCount > 0)
        return myArray[myFront];
    else
    {
cerr << "*** Queue is empty -- returning garbage value ***\n";
QueueElement garbage;
return garbage;
}

void Queue::dequeue()
{
    if (myCount > 0)
    {
        myFront = (myFront + 1) % QUEUE_CAPACITY;
        myCount--;
    }
    else
        cerr << "*** Queue is empty -- can’t remove a value ***\n";
}

16. /* Implementation of Queue class. 
Count of elements used to distinguish between empty and full. 
No data member myBack is used. 
Add a data member: int myCount; to the private section of 
the Queue class declaration and remove: int myBack; */

#include <iostream>
using namespace std;

Queue::Queue()
: myFront(0), myCount(0)
}

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

void Queue::enqueue(const QueueElement & value)
{
    if (myCount < QUEUE_CAPACITY)
    {
        int back = (myFront + myCount) % QUEUE_CAPACITY;
        myArray[back] = value;
        myCount++;
    }
    else
    {
        cerr << "*** Queue full -- can’t add new value ***\n"
            "Must increase value of QUEUE_CAPACITY in Queue.h\n";
        exit(1);
    }
}

QueueElement Queue::front()
{
    if (myCount > 0)
        return myArray[myFront];

else
{
    cerr << "*** Queue is empty -- returning garbage value ***\n";
    QueueElement garbage;
    return garbage;
}
}

void Queue::dequeue()
{
    if (myCount > 0)
    {
        myFront = (myFront + 1) % QUEUE_CAPACITY;
        myCount--;
    }
    else
    cerr << "*** Queue is empty -- can't remove a value ***\n";
}

17.
/* Implementation of Queue class.
Full data member used to distinguish between empty and full
Add a data member:   bool iAmFull;  to the private section of
the Queue class declaration.
*/
#include <iostream>
using namespace std;

Queue::Queue()
: myFront(0), myCount(0), iAmFull(false)
{}

bool Queue::empty() const
{
    return (myBack == myFront && !iAmFull);
}

void Queue::enqueue(const QueueElement & item)
{
    if (!iAmFull)
    {
        myArray[myBack] = item;
        myBack = (myBack + 1) % QUEUE_CAPACITY;
        iAmFull = (myBack == myFront);
    }
    else
    {
        cerr << "*** Queue full -- can't add new value ***\n"
            "Must increase value of QUEUE_CAPACITY in Queue.h\n"
        exit(1);
    }
}
QueueElement Queue::front()
{
    if (!empty())
    {
        return myArray[myFront];
    }
    else
    {
        cerr << "*** Queue is empty -- returning garbage value ***\n";
        QueueElement garbage;
        return garbage;
    }
}

void Queue::dequeue()
{
    if (!empty())
    {
        myFront = (myFront + 1) % QUEUE_CAPACITY;
        iAmFull = false;
    }
    else
        cerr << "Queue is empty: cannot remove from queue. Error!!" " \n" << endl;
}

18. This is similar to the use of one buffer for two stacks (Exer. 12 in §7.2): If two queues were to be stored in one array with the front of each being at the ends of the array, then the queues could grow until the backs met in the middle. Then, one of the queues would have to be shifted back to its end. If each queue size is fixed, wraparound within each queue could be employed to avoid shifting elements.

**Exercises 8.3**

1. 

2. 

3. 

---
4.

```
myFront

myBack
```

5. // Prototype:
QueueElement back() const;
/*---------------------------------------------
Retrieve the back element of this queue.
Precondition: None
Postcondition: Back element of the queue is returned, unless there
was none, in which case a queue-empty message is displayed.
---------------------------------------------*/

// Definition:
QueueElement Queue::back() const
{
    if (myBack != 0)
        return * myArray;
    //else
    cerr <<"Error: queue is empty -- returning garbage value\n";
    QueueElement garbage;
    return garbage;
}

6. // Prototype:
QueueElement nthElement(int n);
/*---------------------------------------------
Retrieve the n-th element of a queue.
Precondition: 1 <= n <= number of queue elements
Postcondition: n-th element of the queue is returned, unless queue
has fewer than n elements, in which case an error message is
displayed. Also, the elements preceding the n-th element are
removed from the queue.
---------------------------------------------*/

// Definition:
QueueElement Queue::nthElement(int n)
{
    QueueElement elem;
    while( n > 0 && !empty())
    {
        elem = front();
        removeQ();
        n--;
    }
    if (n > 0)
    {
        cerr << "Error: insufficient number of elements in the queue\n";
        "-- returning garbage value\n";
        QueueElement garbage;
        return garbage;
    }
//else 
return elem;
}

7. // Prototype: 
QueueElement nthElement(int n) const; 
/*----------------------------------------------------------------------
Retrieve the n-th element of a queue.
Precondition:  1 <= n <= number of queue elements
Postcondition: n-th element of the queue is returned, unless queue
has fewer than n elements, in which case an error message is
displayed.. 
----------------------------------------------------------------------*/

// Definition: 
QueueElement Queue::nthElement(int n) const
{
    int count = 0;
    Queue::NodePonter ptr = myFront;

    for (int count = 0; count < n && ptr != 0; count++)
        ptr = ptr->next;

    if (ptr != 0)
        return *ptr;
//else
    cerr << "Error: insufficient number of elements in the queue\n"; 
    "-- returning garbage value\n";
    QueueElement garbage;
    return garbage;
}

8. /*-- CLQueue.h ------------------------------------------------------
This header file defines a Queue data type.
Basic operations:
    constructor: Constructs an empty queue
    copy constructor: Constructs a copy of a queue
    =: Assignment operator
    destructor: Destroys a queue
    empty: Checks if a queue is empty
    enqueue: Modifies a queue by adding a value at the back
    front: Accesses the top stack value; leaves queue unchanged
    dequeue: Modifies queue by removing the value at the front
    display: Displays all the queue elements

A circular linked list is used to store the queue elements.
---------------------------------------------------------------------*/
#ifndef CLQUEUE
#define CLQUEUE

#include <iostream>

typedef int QueueElement;

class Queue
{
    private:
        class Node
        {
            public:
                //------ DATA MEMBERS OF Node
                QueueElement data;
                Node * next;

                //------ Node OPERATIONS

                /* --- The Node default class constructor initializes a Node's next member.
                 
                Precondition: None
                Postcondition: The next member has been set to 0.
                ---------------------------------------------------------------*/
                Node()
                : next(0)
                {}

                /* --- The Node class constructor initializes a Node's data members.
                 
                Precondition: None
                Postcondition: The data and next members have been set to dataValue and 0, respectively.
                ---------------------------------------------------------------*/
                Node(QueueElement dataValue)
                : data(dataValue), next(0)
                {}
            } //--- end of Node class

typedef Node * NodePointer;

public:
    /***** Function members *****
    /***** Constructors *****/

    Queue();
    /*---------------------------------
       Construct a Queue object.
       
       Precondition: None.
       Postcondition: An empty Queue object has been constructed (myBack is initialized to 0).
       ---------------------------------*/
Queue (const Queue & original);
/*----------------------------------------------------------
Copy Constructor
Precondition: original is the queue to be copied and
is received as a const reference parameter.
Postcondition: A copy of original has been constructed.
----------------------------------------------------------*/

~Queue();
/*----------------------------------------------------------
Class destructor
Precondition: None
Postcondition: The linked list in the queue has been
destroyed.
----------------------------------------------------------*/

const Queue & operator=(const Queue & rightHandSide);
/*----------------------------------------------------------
Assignment Operator
Precondition: original is the queue to be assigned and
is received as a const reference parameter.
Postcondition: The current queue becomes a copy of
original and a const reference to it is returned.
----------------------------------------------------------*/

bool empty() const;
/*----------------------------------------------------------
Check if queue is empty.
Precondition: None
Postcondition: Returns true if queue is empty and
false otherwise.
----------------------------------------------------------*/

void enqueue(const QueueElement & value);
/*----------------------------------------------------------
Add a value to a queue.
Precondition: value is to be added to this queue
Postcondition: value is added at back of queue provided
memory is available otherwise, a memory-error message
is displayed and execution is terminated.
----------------------------------------------------------*/

QueueElement front() const;
/*----------------------------------------------------------
Retrieve value at front of queue (if any).
Precondition: Queue is nonempty
Postcondition: Value at front of queue is returned, unless
the queue is empty; in that case, an error message is
displayed and a "garbage value" is returned.
----------------------------------------------------------*/
void dequeue();
/*-----------------------------------------------------------------------------
   Remove value at front of queue (if any).
   
   Precondition:  Queue is nonempty.
   Postcondition: Value at front of queue has been removed,
   unless the queue is empty; in that case, an error
   message is displayed and execution allowed to proceed.
-----------------------------------------------------------------------------*/

void display(ostream & out) const;
/*-----------------------------------------------------------------------------
   Display values stored in the queue.
   
   Precondition:  ostream out is open.
   Postcondition: Queue's contents, from front to back, have
   been output to out.
-----------------------------------------------------------------------------*/

private:
   /***** Data member *****/
   NodePointer myBack;
}; //--- end of Queue class
#endif

/*-- CLQueue.cpp----------------------------------------------------------
   This file implements Stack member functions.
   A circular linked list with pointer to last node is used to
   store the queue elements.
-----------------------------------------------------------------------*/

#include <iostream>
using namespace std;

#include "CLQueue.h"

// Definition of constructor
Queue::Queue() :
   myBack(0)
{
}

// Definition of empty()
bool Queue::empty() const
{
   return myBack == 0; }

// Definition of enqueue()
void Queue::enqueue(const QueueElement & dataVal)
{
   Queue::NodePointer newPtr = new(nothrow) Node(dataVal);
   if (newPtr == 0)
   { cerr << "Out of memory\n"; exit(1); }

   if (myBack == 0)
      newPtr->next = newPtr;
else
{
    newPtr->next = myBack->next;
    myBack->next = newPtr;
}
myBack = newPtr;

// Definition of front()
QueueElement Queue::front() const
{
    if (myBack == 0)
    {
        cerr <<"Queue is empty: error! Returning garbage value\n";
        QueueElement garbage;
        return garbage;
    }
    // else
    return myBack->next->data;
}

// Definition of dequeue()
void Queue::dequeue()
{
    if (myBack == 0)
        cerr <<"Queue is empty: cannot remove from queue: error!\n";
    else
    {
        Queue::NodePointer ptr = myBack->next;
        if (ptr->next == ptr)    // one-element queue becomes empty
            myBack = 0;
        else
            myBack->next = ptr->next;
        delete ptr;
    }
}

// Definition of the destructor
Queue::~Queue()
{
    if (myBack != 0)
    {
        Queue::NodePointer ptr,
            prev = myBack->next;
        while (prev != myBack)
        {
            ptr = prev->next;
            delete prev;
            prev = ptr;
        }
        delete myBack;
    }
}
// Definition of the copy constructor
Queue::Queue(const Queue & original)
{
    myBack = 0;
    if (!original.empty())
    {
        Queue::NodePointer origPtr = original.myBack->next,
            frontPtr, lastPtr;

        frontPtr = new Node(origPtr->data);
        if (frontPtr == 0)
            { cerr << "Out of memory\n"; exit(1); }

        lastPtr = frontPtr;
        while (origPtr != original.myBack)
        {
            origPtr = origPtr->next;
            lastPtr->next = new Node(origPtr->data);
            if (lastPtr == 0)
                { cerr << "Out of memory\n"; exit(1); }

            lastPtr = lastPtr->next;
        }
        lastPtr->next = frontPtr;
        myBack = lastPtr;
    }
}

// Definition of the assignment operator
const Queue &  Queue::operator=(const Queue & original)
{
    myBack = 0;
    if (this != &original)
    {
        delete myBack;
        Queue::NodePointer origPtr = original.myBack->next,
            frontPtr, lastPtr;

        frontPtr = new Node(origPtr->data);
        if (frontPtr == 0)
            { cerr << "Out of memory\n"; exit(1); }

        lastPtr = frontPtr;
        while (origPtr != original.myBack)
        {
            origPtr = origPtr->next;
            lastPtr->next = new Node(origPtr->data);
            if (lastPtr == 0)
                { cerr << "Out of memory\n"; exit(1); }

            lastPtr = lastPtr->next;
        }
        lastPtr->next = frontPtr;
        myBack = lastPtr;
    }
    return *this;
}
// Definition of the output operators
void Queue::display(ostream & out) const
{
    if (empty()) return;

    Queue::NodePointer ptr = myBack;
    do
    {
        ptr = ptr->next;
        out << ptr->data << " ";
    }
    while (ptr != myBack);
}

inline ostream & operator<<(ostream & out, const Queue & aQueue)
{
    aQueue.display(out);
    return out;
}

// See Programming Problem 18 for a driver

Exercises 8.4

1.

/*----- Deque.h -------------------------------------------------------
A deque (double-ended queue) is similar to a queue but additions
and deletions may be performed on either end. Each store/retrieve
operation must specify at which end the operation is to be performed.

Basic operations:
    Constructor: Constructs an empty deque
    empty: Checks if a deque is empty
    add: Modifies a deque by adding a value at one end
    retrieve: Retrieve the value at one end; leaves deque unchanged
    remove: Modifies a deque by removing the value at one end
    display: Displays the deque elements

Class Invariant:
    1. The deque elements (if any) are stored in consecutive positions
        in myArray, beginning at position myFront.
    2. 0 <= myFront, myBack < DEQUE_CAPACITY
    3. Deque’s size < DEQUE_CAPACITY

---------------------------------------------------------------------*/

#include <iostream>

#ifndef DEQUE
#define DEQUE

const int DEQUE_CAPACITY = 128;
typedef int DequeElement;
enum End {FRONT, BACK};
class Deque
{
    /***** Function Members *****/
public:
    Deque();
    /*---------------------------------------
Construct a Deque object.

Precondition:  None.
Postcondition: An empty Deque object has been constructed; myFront
    and myBack are initialized to -1 and myArray is an array with
    DEQUE_CAPACITY elements of type DequeElement.
    ---------------------------------------*/

    bool empty() const;
    /*---------------------------------------
    Check if deque is empty.

Precondition:  None.
Postcondition: True is returned if the deque is empty and false is
    returned otherwise.
    ---------------------------------------*/

    void add(const DequeElement & value, End where);
    /*---------------------------------------
    Add a value to a deque.

Precondition:  where is FRONT (0) or BACK (1).
Postcondition: value is added at end of deque specified by where,
    provided there is space; otherwise, a deque-full message is
    displayed and execution is terminated.
    ---------------------------------------*/

    DequeElement retrieve(End where) const;
    /*---------------------------------------
    Retrieve value at one end of deque (if any).

Precondition:  Deque is nonempty; where is FRONT (0) or BACK (1).
Postcondition: Value at at end of deque specified by where is
    returned, unless deque is empty; in that case, an error message
    is displayed and a "garbage value" is returned.
    ---------------------------------------*/

    void remove(End where);
    /*---------------------------------------
    Remove value at one end of deque (if any).

Precondition:  Deque is nonempty; where is FRONT (0) or BACK (1).
Postcondition: Value at at end of deque specified by where is
    removed, unless deque is empty; in that case, an error message
    is displayed.
    ---------------------------------------*/
}
// --- display
void display(ostream & out) const;
/*-----------------------------------------------
 Output the values stored in the deque.
 Precondition:  ostream out is open.
 Postcondition: Deque's contents have been output to out.
 -----------------------------------------------*/

/***** Data Members *****/
private:
    DequeElement myArray[DEQUE_CAPACITY];
    int myFront,
        myBack;
}; // end of class declaration
#endif

//----- Deque.cpp -----
#include <iostream>
#include <cassert>
using namespace std;
#include "Deque.h"

//-- Definition of constructor
Deque:: Deque ()
    : myFront(0), myBack(0)
{}

//-- Definition of empty()
bool Deque::empty() const
{
    return myFront == myBack;
}

//-- Definition of add()
void Deque::add(const DequeElement & value, End where)
{
    assert (where == FRONT || where == BACK);
    int newBack = (myBack + 1) % DEQUE_CAPACITY;
    if (newBack == myFront)
    {
        cerr << "Deque is full: cannot add to deque. Error!! " << endl;
        exit(1);
    }
    //else
    if (where == BACK)
    {
        myArray[myBack] = value;
        myBack = newBack;
    }
2. Implementing a scroll is an easy restriction of the deque class in Exercise 1 — simply restrict the add operation to the front and remove to the back.