laboratory 6

Queue ADT

Objectives

In this laboratory you will

- Create two implementations of the Queue ADT—one based on an array representation of a queue, the other based on a singly linked list representation
- Create a program that simulates the flow of customers through a line
- Create an array implementation of a dequeue
- Analyze the memory requirements of your array and linked list queue representations

Overview

This laboratory focuses on another constrained linear data structure, the **queue**. The data items in a queue are ordered from least recently added (the **front**) to most recently added (the **rear**). Insertions are performed at the rear of the queue and deletions are performed at the front. You use the **enqueue** operation to insert data items and the **dequeue** operation to remove data items. A sequence of enqueues and dequeues is shown below.

Enqueue a	*Enqueue b*	*Enqueue c*	*Dequeue*	*Dequeue*
`a`	`a b`	`a b c`	`b c`	`c`
←front	←front	←front	←front	←front

The movement of data items through a queue reflects the "first in, first out" (FIFO) behavior that is characteristic of the flow of customers in a line or the transmission of information across a data channel. Queues are routinely used to regulate the flow of physical objects, information, and requests for resources (or services) through a system. Operating systems, for example, use queues to control access to system resources such as printers, files, and communications lines. Queues also are widely used in simulations to model the flow of objects or information through a system.

Queue ADT

Data Items

The data items in a queue are of generic type DT.

Structure

The queue data items are linearly ordered from least recently added (the front) to most recently added (the rear). Data items are inserted at the rear of the queue (enqueued) and are removed from the front of the queue (dequeued).

Operations

```
Queue ( int maxNumber = defMaxQueueSize ) throw ( bad_alloc )
```

Requirements:
None

Results:
Constructor. Creates an empty queue. Allocates enough memory for a queue containing `maxNumber` data items (if necessary).

```
~Queue ()
```

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used to store a queue.

```
void enqueue ( const DT &newDataItem ) throw ( logic_error )
```

Requirements:
Queue is not full.

Results:
Inserts `newDataItem` at the rear of a queue.

```
DT dequeue () throw ( logic_error )
```

Requirements:
Queue is not empty.

Results:
Removes the least recently added (front) data item from a queue and returns it.

```
void clear ()
```

Requirements:
None

Results:
Removes all the data items in a queue.

```
bool isEmpty () const
```

Requirements:
None

Results:
Returns `true` if a queue is empty. Otherwise, returns `false`.

```
bool isFull () const
```

Requirements:
None

Results:
Returns `true` if a queue is full. Otherwise, returns `false`.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs the data items in a queue. If the queue is empty, outputs "Empty queue". Note that this operation is intended for testing/debugging purposes only. It only supports queue data items that are one of C++'s predefined data types (`int`, `char`, and so forth).

Laboratory 6: Cover Sheet

Name _____ Date _____

Section _____

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Prelab Exercise		
Bridge Exercise		
In-lab Exercise 1		
In-lab Exercise 2		
In-lab Exercise 3		
Postlab Exercise 1		
Postlab Exercise 2		
Total		

Laboratory 6: Prelab Exercise

Name _____ Date _____

Section _____

In this laboratory you will create two implementations of the Queue ADT. One of these implementations is based on an array; the other is based on a singly linked list. Following the example introduced in Lab 5, the generic data type will be named `DT` for Data Type.

Step 1: Implement the operations in the Queue ADT using an array to store the queue data items. Queues change in size; therefore, you need to store the maximum number of data items the queue can hold (`maxSize`) and the array index of the data items at the front and rear of the queue (`front` and `rear`), along with the queue data items themselves (`dataItems`). Base your implementation on the following declarations from the file *queuearr.h*. An implementation of the `showStructure` operation is given in the file *show6.cpp*.

```
const int defMaxQueueSize = 10;   // Default maximum queue size

template < class DT >
class Queue
{
  public:

    // Constructor
    Queue ( int maxNumber = defMaxQueueSize ) throw ( bad_alloc );

    // Destructor
    ~Queue ();

    // Queue manipulation operations
    void enqueue ( const DT &newData )        // Enqueue data item
        throw ( logic_error );
    DT dequeue ()                             // Dequeue data item
        throw ( logic_error );
    void clear ();                            // Clear queue

    // Queue status operations
    bool isEmpty () const;                    // Queue is empty
    bool isFull () const;                     // Queue is full

    // Output the queue structure — used in testing/debugging
    void showStructure () const;
```

```
  private:

    // Data members
    int maxSize,    // Maximum number of data data items in the queue
        front,      // Index of the front data data item
        rear;       // Index of the rear data data item
    DT *dataItems;  // Array containing the queue data items
};
```

Step 2: Save your array implementation of the Queue ADT in the file *queuearr.cpp*. Be sure to document your code.

Step 3: Implement the operations in the Queue ADT using a singly linked list to store the queue data items. Each node in the linked list should contain a queue data item (`dataItem`) and a pointer to the node containing the next data item in the queue (`next`). Your implementation also should maintain pointers to the nodes containing the front and rear data items in the queue (`front` and `rear`). Base your implementation on the following declarations from the file *queuelnk.h*. An implementation of the `showStructure` operation is given in the file *show6.cpp*.

```
template < class DT >        // Forward declaration of the Queue class
class Queue;

template < class DT >
class QueueNode              // Facilitator class for the Queue class
{
  private:

    // Constructor
    QueueNode ( const DT &nodeData, QueueNode *nextPtr );

    // Data members
    DT dataItem;         // Queue data item
    QueueNode *next;     // Pointer to the next data item

  friend class Queue<DT>;
};

template < class DT >
class Queue
{
  public:

    // Constructor
    Queue ( int ignored = 0 );

    // Destructor
    ~Queue ();

    // Queue manipulation operations
    void enqueue ( const DT &newData )        // Enqueue data data item
        throw ( logic_error );
```

```
    DT dequeue ()                              // Dequeue data data item
        throw ( logic_error );

    void clear ();                             // Clear queue

    // Queue status operations
    bool isEmpty () const;                     // Queue is empty
    bool isFull () const;                      // Queue is full

    // Output the queue structure - used in testing/debugging
    void showStructure () const;

  private:

    // Data members
    QueueNode<DT> *front,   // Pointer to the front node
                  *rear;    // Pointer to the rear node
};
```

Step 4: Save your linked list implementation of the Queue ADT in the file *queuelnk.cpp*. Be sure to document your code.

Laboratory 6: Bridge Exercise

Name _____ Date _____

Section _____

Check with your instructor whether you are to complete this exercise prior to your lab period or during lab.

The test program in the file *test6.cpp* allows you to interactively test your implementations of the Queue ADT using the following commands.

Command	Action
`+x`	Enqueue data item `x`.
`-`	Dequeue a data item and output it.
`E`	Report whether the queue is empty.
`F`	Report whether the queue is full.
`C`	Clear the queue.
`Q`	Exit the test program.

Step 1: Compile and link the test program. Note that compiling this program will compile your array implementation of the Queue ADT (in the file *queuearr.cpp*) to produce an array implementation for a queue of characters.

Step 2: Complete the following test plan by adding test cases in which you

- Enqueue a data item onto a queue that has been emptied by a series of dequeues
- Combine enqueues and dequeues so that you "go around the end" of the array (array implementation)
- Dequeue a data item from a full queue (array implementation)
- Clear the queue

Step 3: Execute your test plan. If you discover mistakes in your array implementation of the Queue ADT, correct them and execute your test plan again.

Step 4: Modify the test program so that your linked list implementation of the Queue ADT in the file *queuelnk.cpp* is included in place of your array implementation.

Step 5: Recompile and relink the test program. Note that recompiling this program will compile your linked list implementation of the Queue ADT (in the file *queuelnk.cpp*) to produce a linked list implementation for a queue of characters.

Step 6: Use your test plan to check your linked list implementation of the Queue ADT. If you discover mistakes in your implementation, correct them and execute your test plan again.

Test Plan for the Operations in the Queue ADT

Test Case	*Commands*	*Expected Result*	*Checked*
Series of enqueues	`+a +b +c +d`	**`a`** `b c d`	
Series of dequeues	`- - -`	`d`	
More enqueues	`+e +f`	`d e f`	
More dequeues	`- -`	`f`	
Empty? Full?	`E F`	`False False`	
Empty the queue	`-`	Empty queue	
Empty? Full?	`E F`	`True False`	

Note: The front data item is shown in **bold.**

Laboratory 6: In-lab Exercise 2

Name _____ Date _____

Section _____

A **deque** (or double-ended queue) is a linear data structure that allows data items to be inserted and removed at both ends. Adding the operations described below will transform your Queue ADT into a Deque ADT.

```
void putFront ( const DT &newDataItem ) throw ( logic_error )
```

Requirements:
Queue is not full.

Results:
Inserts `newDataItem` at the front of a queue. The order of the preexisting data items is left unchanged.

```
DT getRear () throw ( logic_error )
```

Requirements:
Queue is not empty.

Results:
Removes the most recently added (rear) data item from a queue and returns it. The remainder of the queue is left unchanged.

Step 1: Implement these operations using the array representation of a queue and add them to the file *queuearr.cpp*. Prototypes for these operations are included in the declaration of the Queue class in the file *queuearr.h*.

Step 2: Activate the '>' (put in front) and '=' (get from rear) commands in the test program *test6.cpp* by removing the comment delimiter (and the character '>' or '=') from the lines that begin with "//>" and "//=".

Step 3: Complete the following test plan by adding test cases in which you

- Insert a data item at the front of a newly emptied queue
- Remove a data item from the rear of a queue containing only one data item
- "Go around the end" of the array using each of these operations
- Mix `putFront` and `getRear` with enqueue and dequeue

Step 4: Execute your test plan. If you discover mistakes in your implementation of these operations, correct them and execute the test plan again.

Test Plan for the `putFront` and `getRear` operations

Test Case	*Commands*	*Expected Result*	*Checked*
Series of calls to putFront	`>a >b >c >d`	`d c b a`	
Series of calls to getRear	`= = =`	`d`	
More calls to putFront	`>e >f`	`f e d`	
More calls to getRear	`= =`	`f`	

Note: The front data item is shown in **bold.**

Laboratory 6: In-lab Exercise 3

Name _____ Date _____

Section _____

When a queue is used as part of a model or simulation, the modeler is often very interested in how many data items are on the queue at various points in time. This statistic is produced by the following operation.

```
int getLength () const
```

Requirements:
None

Results:
Returns the number of data items in a queue.

Step 1: Create an implementation of this operation using the array representation of a queue and add it to the file *queuearr.cpp*. A prototype for this operation is included in the declaration of the Queue class in the file *queuearr.h*.

Step 2: Activate the '#' (length) command in the test program *test6.cpp* by removing the comment delimiter (and the character '#') from the lines that begin with "//#".

Step 3: Complete the following test plan by adding test cases in which you check the length of empty queues and queues that "go around the end" of the array.

Step 4: Execute your test plan. If you discover mistakes in your implementation of the length operation, correct them and execute the test plan again.

Test Plan for the `length` Operation

Test Case	*Commands*	*Expected Result*	*Checked*
Series of enqueues	`+a +b +c +d`	**`a b c d`**	
Length	`#`	`4`	
Series of dequeues	`- - -`	**`d`**	
Length	`#`	`1`	
More enqueues	`+e +f`	**`d`** `e f`	
Length	`#`	`3`	

Note: The front data item is shown in **bold**.