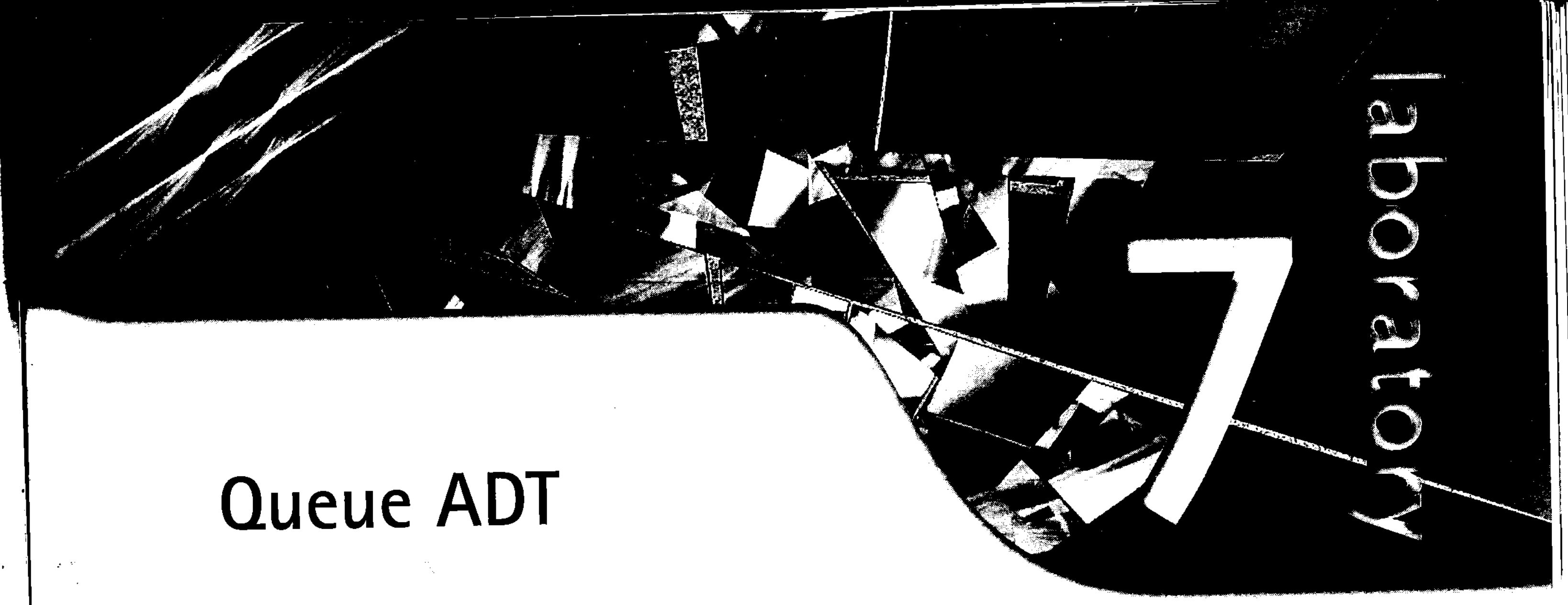

Queue ADT

objectives

In this laboratory you

- 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.
- utilize a queue in a simple simulation.
- learn about pseudo random numbers.
- analyze the memory requirements of your array and linked list queue representations.

ADT 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 here.

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.

C++ Concepts Overview

The Queue uses the same C++ constructs as the stack. See Laboratory 6 for details if needed.

Queue ADT

Data items:

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

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 = MAX_QUEUE_SIZE )`

Requirements:
None

Results:
Constructor. Creates an empty queue. Allocates enough memory for the queue containing `maxNumber` data items (if necessary).[1]

`Queue ( const Queue& other )`

Requirements:
None

Results:
Copy constructor. Initializes the queue to be equivalent to the `other` Queue object parameter.[1]

`Queue& operator= ( const Queue& other )`

Requirements:
None

Results:
Overloaded assignment operator. Sets the queue to be equivalent to the `other` Queue object parameter and returns a reference to the modified queue.[2]

`~Queue ()`

Requirements:
None

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

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

Requirements:
Queue is not full.

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

`DataType dequeue () throw ( logic_error )`

Requirements:
Queue is not empty.

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

[1]Because of the way ABCs work, constructors are actually declared and implemented in the derived classes, not in the base class. This relates to the requirement that constructor names match the class names; it is impossible for the base class and the derived class to have the same name.

[2]Like ABC constructors, `operator=` is declared and implemented in the derived classes with corresponding name changes.

`void clear ()`

Requirements:
None

Results:
Removes all the data items in the queue.

`bool isEmpty () const`

Requirements:
None

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

`bool isFull () const`

Requirements:
None

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

`void showStructure () const`

Requirements:
None

Results:
Outputs the data items in the 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) or other data structures with an overridden ostream `operator<<`.

Implementation Notes

Just like you did with the stack laboratory (Lab 6), you are to 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.

Array-Based Implementation

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 declarations from the file *QueueArray.h*. An array-based implementation of the `showStructure` operation is given in the file *show7.cpp*.

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

Compilation Directions

Compile *test7.cpp*. The value of LAB7_TEST1 in *config.h* determines whether the array-based implementation or the linked-list implementation is tested. If the value is 0 (the default), the array implementation is tested. If the value is 1, then the linked implementation is tested.

Testing

Test your implementation of the array-based Stack ADT using the program in the file *test7.cpp*. The test program allows you to interactively test your ADT implementation using the commands in the following table.

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: Download the online test plans for Lab 7.

Step 2: Complete Test Plan 7-1 by filling in the expected results for each given operation. Add 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 only),
- dequeue a data item from a full queue (array implementation only), and
- clear the queue.

Step 3: Execute Test Plan 7-1. If you discover mistakes in your implementation of the Queue ADT, correct them and execute the test plan again.

Linked-List Implementation

Step 1: 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 should also 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 *QueueLinked.h*. A linked-list implementation of the `showStructure` operation is given in the file *show7.cpp*.

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

Compilation Directions

Edit *config.h* and change the value of LAB7_TEST1 to 1. (If the value is 0, then the array-based implementation is tested instead.) Recompile *test7.cpp*.

Testing

Test your implementation of the linked list Queue ADT using the program in the file *test7.cpp*.

Step 1: Re-execute the pertinent portions of Test Plan 7-1. If you discover mistakes in your linked-list implementation of the Queue ADT, correct them and execute your test plan again.

Programming Exercise 1

In this exercise, you use a queue to simulate the flow of customers through a check-out line in a store. In order to create this simulation, you must model both the passage of time and the flow of customers through the line. You can model time using a loop in which each pass corresponds to a set time interval—1 minute, for example. You can model the flow of customers using a queue in which each data item corresponds to a customer in the line.

In order to complete the simulation, you need to know the rate at which customers join the line, as well as the rate at which they are served and leave the line. Suppose the check-out line has the following properties.

- One customer is served and leaves the line every minute (assuming there is at least one customer waiting to be served during that minute).
- Between zero and two customers join the line every minute, where there is a 50 percent chance that no customers arrive, a 25 percent chance that one customer arrives, and a 25 percent chance that two customers arrive.

You can simulate the flow of customers through the line during a time period *n* minutes long using the following algorithm.

```
Initialize the queue to empty.
While the simulation is not done
    Increment simulated time by one minute
    If the queue is not empty, then remove the customer at the front of the queue.
    Compute a random number k between 0 and 3.
    If k is 1, then add one customer to the line.
    If k is 2, then add two customers to the line.
    Otherwise (if k is 0 or 3), do not add any customers to the line.
    Update queue statistics.
```

Calling the `rand` function is a simple way to generate pseudo-random numbers. It should be available through the `<cstdlib>` function set. Generating random numbers does vary from platform to platform because of compiler and operating system differences. You may need to get help from your lab instructor on how to generate random numbers in your particular context.

Step 1: Using the program shell given in the file *storesim.cs* as a basis, create a program in *storesim.cpp* that uses the Queue ADT to implement the model described above. Your program should update the following information during each simulated minute.

- The total number of customers served.
- The combined length of time these customers spent waiting in line.
- The maximum length of time any of these customers spent waiting in line.

The data that is stored in the queue should contain everything that is necessary to 1) represent the customer and 2) compute the previous statistics. To compute how long a customer waited to be served, you need the difference in time from when the customer arrived to when the customer exited the queue. There is no additional information needed in the statistics, nor is there any customer-specific information that is used for our simple simulation. Therefore, we can represent the customer in the queue simply by using the simulated time the customer entered the queue.

Step 2: Use your program to simulate the flow of customers through the line and complete Table 7-2 in the online supplement. Note that the average wait is the combined waiting time divided by the total number of customers served.

Programming Exercise 2

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 DataType& newDataItem ) throw ( logic_error )
```

Requirements:
Queue is not full.

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

```
DataType getRear () throw ( logic_error )
```

Requirements:
Queue is not empty.

Results:
Removes the most recently added (rear) data item from the 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 *QueueArray.cpp*. Prototypes for these operations are included in the declaration of the Queue class in the file *QueueArray.h*.

Step 2: Select the array implementation of Queue by setting the value of LAB7_TEST1 to 0 in *config.h*. Then activate Test 2 by changing the value of LAB7_TEST2 from 0 to 1.

Step 3: Complete Test Plan 7-3 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 (only the array implementation), and
- mix `putFront` and `getRear` with `enqueue` and `dequeue`. The test program uses > to execute the putFront operation and = to execute the getRear operation.

Step 4: Execute Test Plan 7-3. If you discover mistakes in your implementation of these operations, correct them and execute the test plan again.

Step 5: Select the linked implementation of Queue by changing the value of LAB7_TEST1 from 0 to 1 in *config.h*.

Step 6: Re-execute the pertinent portions of Test Plan 7-3 for your linked implementation. If you discover mistakes in your implementation of these operations, correct them and execute the test plan again.

Programming Exercise 3

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

Step 2: Select the array implementation of Queue by setting the value of LAB7_TEST1 to 0 in *config.h*. Then activate Test 3 by changing the value of LAB7_TEST3 from 0 to 1.

Step 3: Complete Test Plan 7-4 by adding test cases in which you test `getLength` with queues of various lengths. The test program uses # to execute the getLength operation.

Step 4: Execute Test Plan 7-4. If you discover mistakes in your implementation of these operations, correct them and execute the test plan again.

Step 5: Select the linked implementation of Queue by changing the value of LAB7_TEST1 from 0 to 1 in *config.h*.

Step 6: Re-execute Test Plan 7-4 for your linked implementation. If you discover mistakes in your implementation of these operations, correct them and execute the test plan again.

Analysis Exercise 1

Part A

Given the following memory requirements

Integer	4 bytes
Address (pointer)	4 bytes

and a queue containing one hundred integers, compare the amount of memory used by your array representation of the queue to the amount of memory used by your singly linked list representation. Assume that the array representation allows a queue to contain a maximum of one hundred data items.

Note: integer and pointers memory requirements vary depending on the operating system and compiler. Integers and addresses range in size from 2 to 8 bytes, or larger. The values above represent a specific platform and were chosen for simplicity of calculation.

Part B

Suppose that you have ten queues of integers. Of these ten queues, four are 50% full, and the remaining six are 10% full. Compare the amount of memory used by your array representation of these queues with the amount of memory used by your singly linked list representation. Assume that the array representation allows a queue to contain a maximum of one hundred data items.

Part C

Suppose that you have a large object that requires 1000 bytes of memory. Repeat the analysis from Part A using a queue of large objects. How does the large object affect the memory efficiency of the two queues?

Analysis Exercise 2

In Programming Exercise 1, you used a queue to simulate the flow of customers through a line. Describe another application where you might use the Queue ADT. What type of information does your application store in each queue data item?