Queues

3

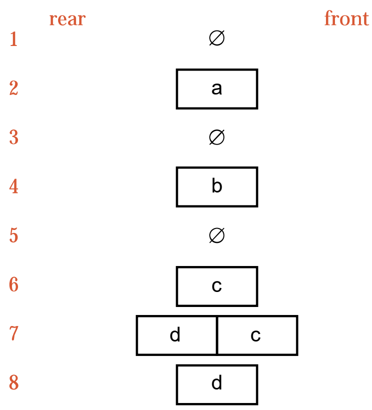
3.1 DEFINITIONS

Exercises 3.1

E1. *Suppose that* q *is a* Queue *that holds characters and that* x *and* y *are character variables. Show the contents of* q *at each step of the following code segments.*

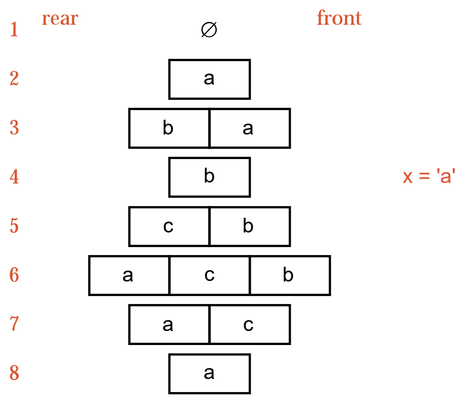
(a)
```
Queue q;
q.append('a');
q.serve();
q.append('b');
q.serve();
q.append('c');
q.append('d');
q.serve();
```

Answer



(b)
```
Queue q;
q.append('a');
q.append('b');
q.retrieve(x);
q.serve();
q.append('c');
q.append(x);
q.serve();
q.serve();
```

Answer



TOC Index Help

(c)
```
Queue q;
q.append('a');
x = 'b';
q.append('x');
q.retrieve(y);
q.serve();
q.append(x);
q.serve();
q.append(y);
```

Answer

	rear		front	
1		∅		
2		a		
3				x = 'b'
4	x		a	
5		x		y = 'a'
6	b		x	
7		b		
8	a		b	

E2. *Suppose that you are a financier and purchase 100 shares of stock in Company X in each of January, April, and September and sell 100 shares in each of June and November. The prices per share in these months were*

accounting

Jan	Apr	Jun	Sep	Nov
\$10	*\$30*	*\$20*	*\$50*	*\$30*

Determine the total amount of your capital gain or loss using **(a)** *FIFO (first-in, first-out) accounting and* **(b)** *LIFO (last-in, first-out) accounting [that is, assuming that you keep your stock certificates in* **(a)** *a queue or* **(b)** *a stack]. The 100 shares you still own at the end of the year do not enter the calculation.*

Answer Purchases and sales:

January:	purchase	$100 \times \$10 = \$1000$
April:	purchase	$100 \times \$30 = \$3000$
June:	sell	$100 \times \$20 = \$2000$
September:	purchase	$100 \times \$50 = \$5000$
November:	sell	$100 \times \$30 = \$3000$

With FIFO accounting, the 100 shares sold in June were those purchased in January, so the profit is \$2000 − \$1000 = \$1000. Similarly, the November sales were the April purchase, both at \$300, so the transaction breaks even. *Total profit* = \$1000. With LIFO accounting, June sales were April purchases giving a loss of \$1000 (\$2000 − \$3000), and November sales were September purchases giving a loss of \$2000 (\$3000 − \$5000). *Total loss* = \$3000.

E3. *Use the methods for stacks and queues developed in the text to write functions that will do each of the following tasks. In writing each function, be sure to check for empty and full structures as appropriate. Your functions may declare other, local structures as needed.*

(a) *Move all the entries from a* Stack *into a* Queue.

Answer The following procedures use both stacks and queues. Where both queues and stacks are involved in the solution, entries will be of type Entry, as opposed to Stack_entry or Queue_entry. This can be accomplished by adding the type declarations **typedef** Entry Stack_entry; and **typedef** Entry Queue_entry; to the implementations. The type of Entry is specified by client code in the usual way.

```
Error_code stack_to_queue(Stack &s, Queue &q)
/* Pre:  The Stack s and the Queue q have the same entry type.
   Post: All entries from s have been moved to q. If there is not enough room in q to hold all
         entries in s return a code of overflow, otherwise return success. */
```

TOC
Index
Help

```
{
   Error_code outcome = success;
   Entry item;
   while (outcome == success && !s.empty()) {
      s.top(item);
      outcome = q.append(item);
      if (outcome == success) s.pop();
   }
   return (outcome);
}
```

(b) *Move all the entries from a* Queue *onto a* Stack.

Answer

```
Error_code queue_to_stack(Stack &s, Queue &q)
/* Pre:   The Stack s and the Queue q have the same entry type.
   Post:  All entries from q have been moved to s. If there is not enough room in s to hold all
          entries in q return a code of overflow, otherwise return success. */
{
   Error_code outcome = success;
   Entry item;
   while (outcome == success && !q.empty()) {
      q.retrieve(item);
      outcome = s.push(item);
      if (outcome == success) q.serve();
   }
   return (outcome);
}
```

(c) *Empty one* Stack *onto the top of another* Stack *in such a way that the entries that were in the first* Stack *keep the same relative order.*

Answer

```
Error_code move_stack(Stack &s, Stack &t)
/* Pre:   None.
   Post:  All entries from s have been moved in order onto the top of t. If there is not enough
          room in t to hold these entries return a code of overflow, otherwise return success. */
{
   Error_code outcome = success;
   Entry item;
   Stack temp;
   while (outcome == success && !s.empty()) {
      s.top(item);
      outcome = temp.push(item);
      if (outcome == success) s.pop();
   }
   while (outcome == success && !temp.empty()) {
      temp.top(item);
      outcome = t.push(item);
      if (outcome == success) temp.pop();
   }
   while (!temp.empty()) {                    //  replace any entries to s that can not fit on t
      temp.top(item);
      s.push(item);
   }
   return (outcome);
}
```

TOC
Index
Help

(d) *Empty one* Stack *onto the top of another* Stack *in such a way that the entries that were in the first* Stack *are in the reverse of their original order.*

Answer

```
Error_code reverse_move_stack(Stack &s, Stack &t)
/* Pre:  None.
   Post: All entries from s have been moved in reverse order onto the top of t. If there is not
         enough room in t to hold these entries return a code of overflow, otherwise return
         success. */
{
   Error_code outcome = success;
   Entry item;
   while (outcome == success && !s.empty()) {
      s.top(item);
      outcome = t.push(item);
      if (outcome == success) s.pop();
   }
   return (outcome);
}
```

(e) *Use a local* Stack *to reverse the order of all the entries in a* Queue.

Answer

```
Error_code reverse_queue(Queue &q)
/* Pre:  None.
   Post: All entries from q have been reversed. */
{
   Error_code outcome = success;
   Entry item;
   Stack temp;
   while (outcome == success && !q.empty()) {
      q.retrieve(item);
      outcome = temp.push(item);
      if (outcome == success) q.serve();
   }
   while (!temp.empty()) {
      temp.top(item);
      q.append(item);
      temp.pop();
   }
   return (outcome);
}
```

(f) *Use a local* Queue *to reverse the order of all the entries in a* Stack.

Answer

```
Error_code reverse_stack(Stack &s)
/* Pre:  None.
   Post: All entries from s have been reversed. */
{
   Error_code outcome = success;
   Entry item;
   Queue temp;
   while (outcome == success && !s.empty()) {
      s.top(item);
      outcome = temp.append(item);
      if (outcome == success) s.pop();
   }
```

TOC
Index
Help

```
   while (!temp.empty()) {
      temp.retrieve(item);
      s.push(item);
      temp.serve();
   }
   return (outcome);
}
```

3.3 CIRCULAR IMPLEMENTATION OF QUEUES IN C++

Exercises 3.3

E1. *Write the remaining methods for queues as implemented in this section:*

(a) empty

Answer

```
bool Queue :: empty( ) const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return count == 0;
}
```

(b) retrieve

Answer

```
Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue is empty
         return an Error_code of underflow. */
{
   if (count <= 0) return underflow;
   item = entry[front];
   return success;
}
```

E2. *Write the remaining methods for extended queues as implemented in this section:*

(a) full

Answer

```
bool Extended_queue :: full( ) const
/* Post: Return true if the Extended_queue is full; return false otherwise. */
{
   return count == maxqueue;
}
```

(b) clear

Answer

```
void Extended_queue :: clear( )
/* Post: All entries in the Extended_queue have been deleted; the Extended_queue is empty. */
{
   count = 0;
   front = 0;
   rear = maxqueue − 1;
}
```

TOC
Index
Help

(c) serve_and_retrieve

Answer

```
Error_code Extended_queue :: serve_and_retrieve(Queue_entry &item)
/* Post: Return underflow if the Extended_queue is empty. Otherwise remove and copy the
         item at the front of the Extended_queue to item. */
{
   if (count == 0) return underflow;
   else {
      count−−;
      item = entry[front];
      front = ((front + 1) == maxqueue) ? 0 : (front + 1);
   }
   return success;
}
```

E3. *Write the methods needed for the implementation of a queue in a linear array when it can be assumed that the queue can be emptied when necessary.*

Answer The class definition for this Queue implementation is as follows.

```
const int maxqueue = 10;                   //  small value for testing
class Queue {
public:
   Queue();
   bool empty() const;
   Error_code serve();
   Error_code append(const Queue_entry &item);
   Error_code retrieve(Queue_entry &item) const;
protected:
   int front, rear;
   Queue_entry entry[maxqueue];
};
```

The method implementations follow.

```
Queue :: Queue()
/* Post: The Queue is initialized to be empty. */
{
   rear = − 1;
   front = 0;
}
bool Queue :: empty() const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return rear < front;
}
Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the Queue is full, then empty the Queue
         before adding item and return an Error_code of overflow. */
{
   Error_code result = success;
   if (rear == maxqueue − 1) {
      result = overflow;
      rear = −1;
      front = 0;
   }
   entry[++rear] = item;
   return result;
}
```

```
Error_code Queue :: serve()
/* Post: The front of the Queue is removed. If the Queue is empty return an Error_code of
         underflow. */
{
   if (rear < front) return underflow;
   front = front + 1;
   return success;
}
Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue is empty
         return an Error_code of underflow. */
{
   if (rear < front) return underflow;
   item = entry[front];
   return success;
}
```

E4. ***Write the methods to implement queues by the simple but slow technique of keeping the front of the queue always in the first position of a linear array.***

Answer The class definition for this Queue implementation is as follows.

```
const int maxqueue = 10;                 //  small value for testing
class Queue {
public:
   Queue();
   bool empty() const;
   Error_code serve();
   Error_code append(const Queue_entry &item);
   Error_code retrieve(Queue_entry &item) const;
protected:
   int rear;                             //  front == 0
   Queue_entry entry[maxqueue];
};
```

The method implementations follow.

```
Queue :: Queue()
/* Post: The Queue is initialized to be empty. */
{
   rear = −1;
}
bool Queue :: empty() const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return rear < 0;
}
Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the Queue is full return an Error_code of
         overflow. */
{
   if (rear == maxqueue − 1) return overflow;
   entry[++rear] = item;
   return success;
}
Error_code Queue :: serve()
/* Post: The front of the Queue is removed. If the Queue is empty return an Error_code of
         underflow. */
```

TOC
Index
Help

```
{
  if (rear < 0) return underflow;
  for (int i = 0; i < rear; i++)
    entry[i] = entry[i + 1];
  rear--;
  return success;
}
Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue is empty
         return an Error_code of underflow. */
{
  if (rear < 0) return underflow;
  item = entry[0];
  return success;
}
```

E5. *Write the methods to implement queues in a linear array with two indices* front *and* rear, *such that, when* rear *reaches the end of the array, all the entries are moved to the front of the array.*

Answer The class definition for this Queue implementation is as follows.

```
const int maxqueue = 10;               //   small value for testing
class Queue {
public:
  Queue();
  bool empty() const;
  Error_code serve();
  Error_code append(const Queue_entry &item);
  Error_code retrieve(Queue_entry &item) const;
protected:
  int front, rear;
  Queue_entry entry[maxqueue];
};
```

The method implementations follow.

```
Queue :: Queue()
/* Post: The Queue is initialized to be empty. */
{
  rear = -1;
  front = 0;
}
bool Queue :: empty() const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
  return rear < front;
}
Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the rear is at or immediately before the end
         of the Queue, move all entries back to the start of the Queue before appending. If the
         Queue is full return an Error_code of overflow. */
{
  if (rear == maxqueue - 1 || rear == maxqueue - 2)
   for (int i = 0; i <= rear - front; i++) {
      entry[i] = entry[i + front];
      rear = rear - front;
      front = 0;
   }
```