Queue and Priority Queue Implementations

Chapter 14

Implementations of the ADT Queue

- Like stacks, queues can have
  - Array-based or
  - Link-based implementation.
- Can also use implementation of ADT list
  - Efficient to implement
  - Might not be most time efficient as possible

An Implementation That Uses the ADT List

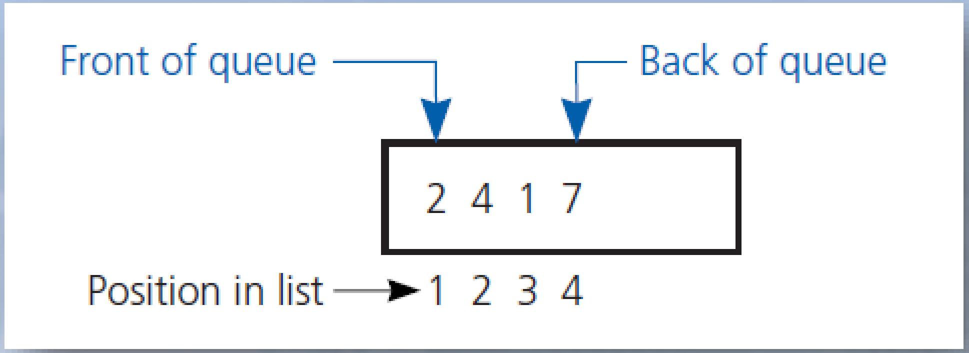
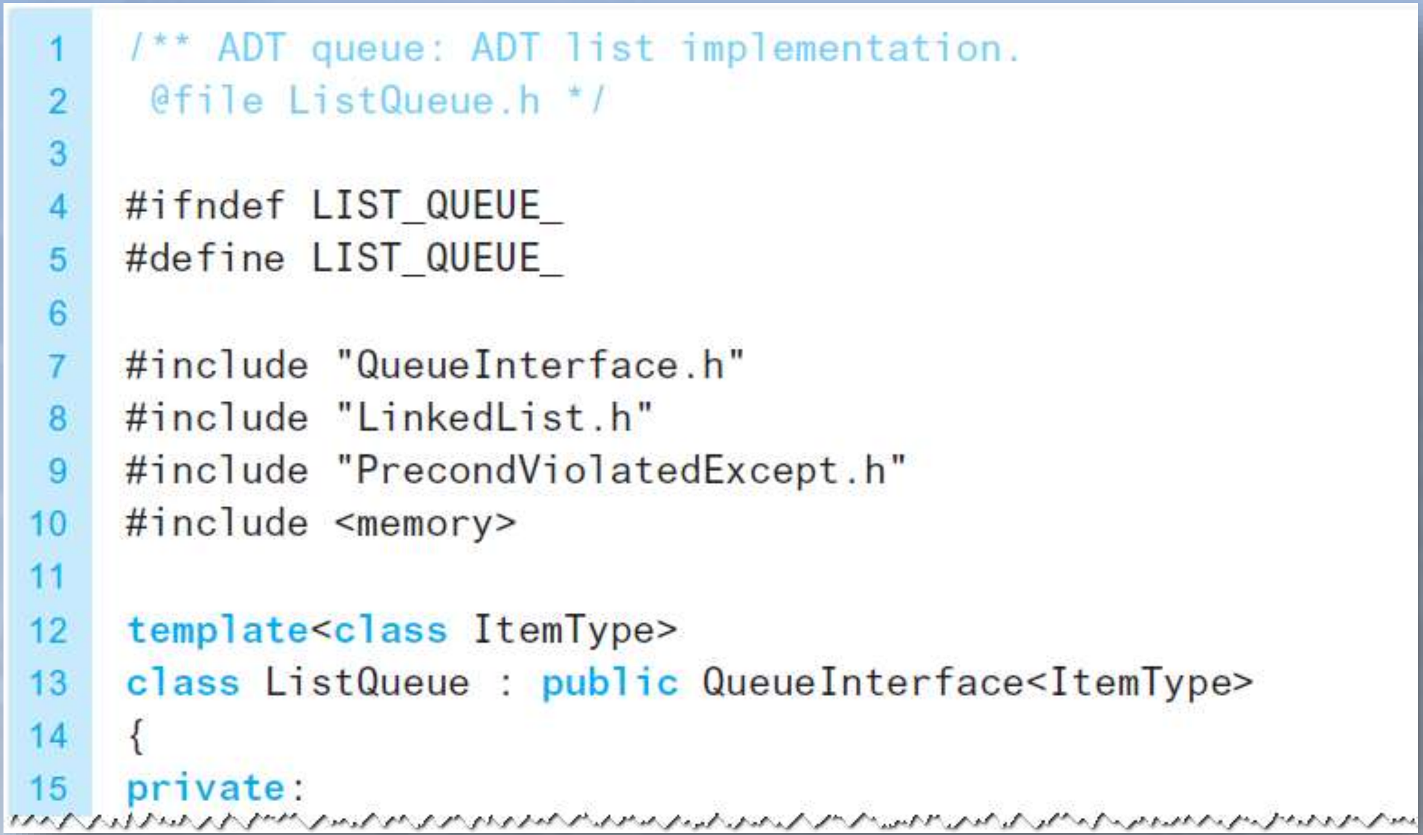


FIGURE 14-1 An implementation of the ADT queue that stores its entries in a list

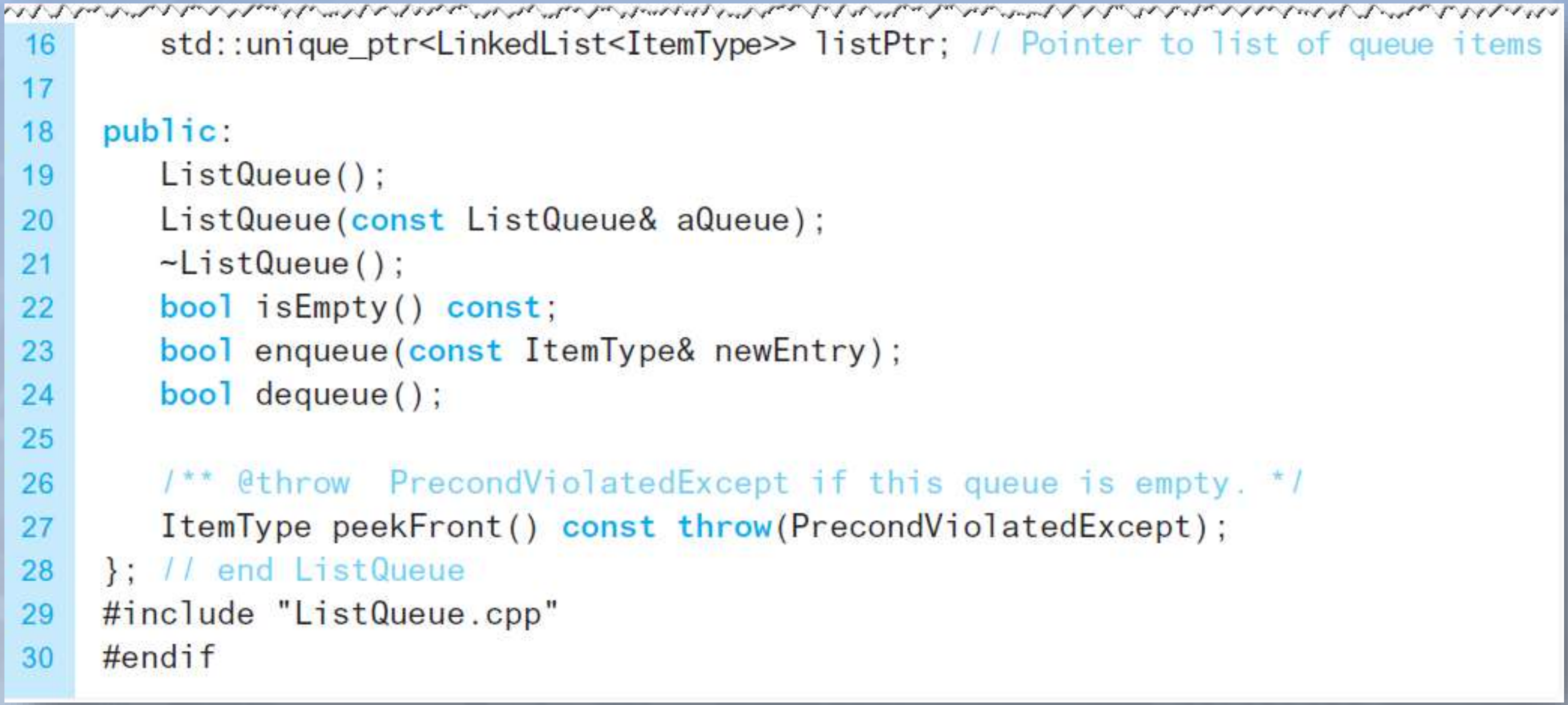
An Implementation That Uses the ADT List



```
/** ADT queue: ADT list implementation.
 @file ListQueue.h */

#ifndef LIST_QUEUE_
#define LIST_QUEUE_

#include "QueueInterface.h"
#include "LinkedList.h"
#include "PrecondViolatedExcept.h"
#include <memory>

template<class ItemType>
class ListQueue : public QueueInterface<ItemType>
{
private:
```

LISTING 14-1 The header file for the class ListQueue

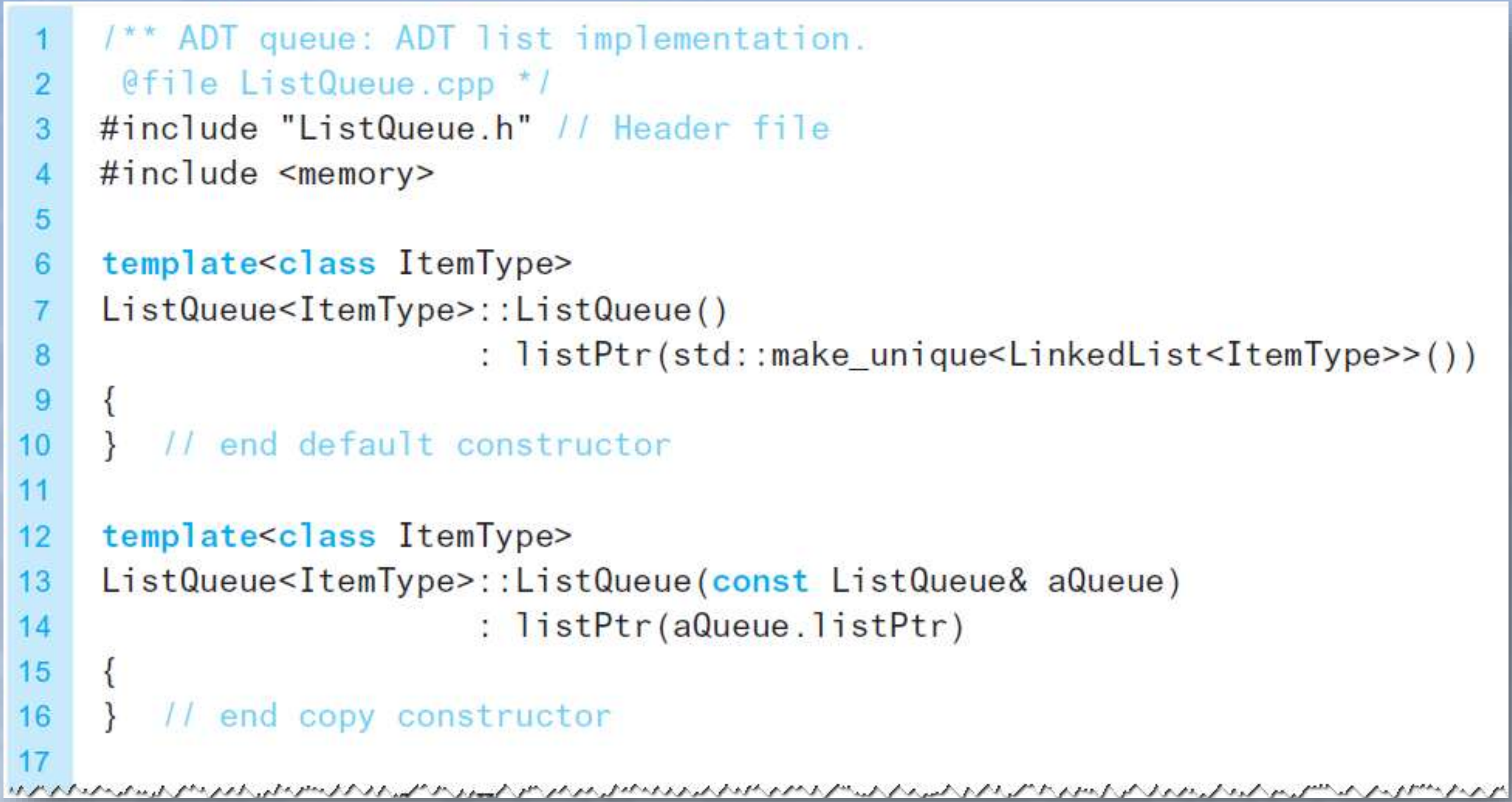
An Implementation That Uses the ADT List



```cpp
   std::unique_ptr<LinkedList<ItemType>> listPtr; // Pointer to list of queue items

public:
   ListQueue();
   ListQueue(const ListQueue& aQueue);
   ~ListQueue();
   bool isEmpty() const;
   bool enqueue(const ItemType& newEntry);
   bool dequeue();

   /** @throw  PrecondViolatedExcept if this queue is empty. */
   ItemType peekFront() const throw(PrecondViolatedExcept);
}; // end ListQueue
#include "ListQueue.cpp"
#endif
```

LISTING 14-1 The header file for the class ListQueue

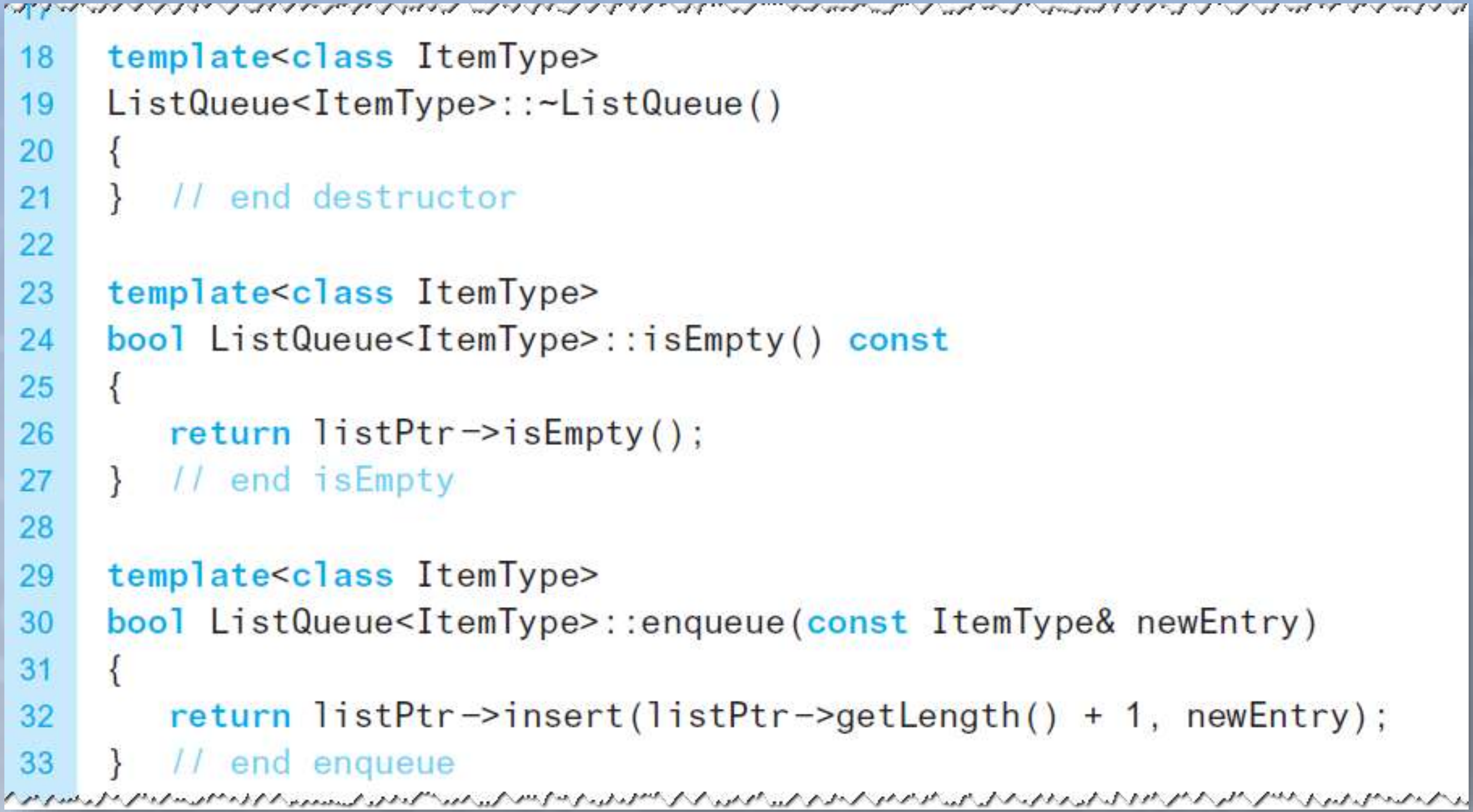
An Implementation That Uses the ADT List



```
/** ADT queue: ADT list implementation.
 @file ListQueue.cpp */
#include "ListQueue.h" // Header file
#include <memory>

template<class ItemType>
ListQueue<ItemType>::ListQueue()
                   : listPtr(std::make_unique<LinkedList<ItemType>>())
{
}  // end default constructor

template<class ItemType>
ListQueue<ItemType>::ListQueue(const ListQueue& aQueue)
                   : listPtr(aQueue.listPtr)
{
}  // end copy constructor

```

LISTING 14-2 The implementation file for the class ListQueue

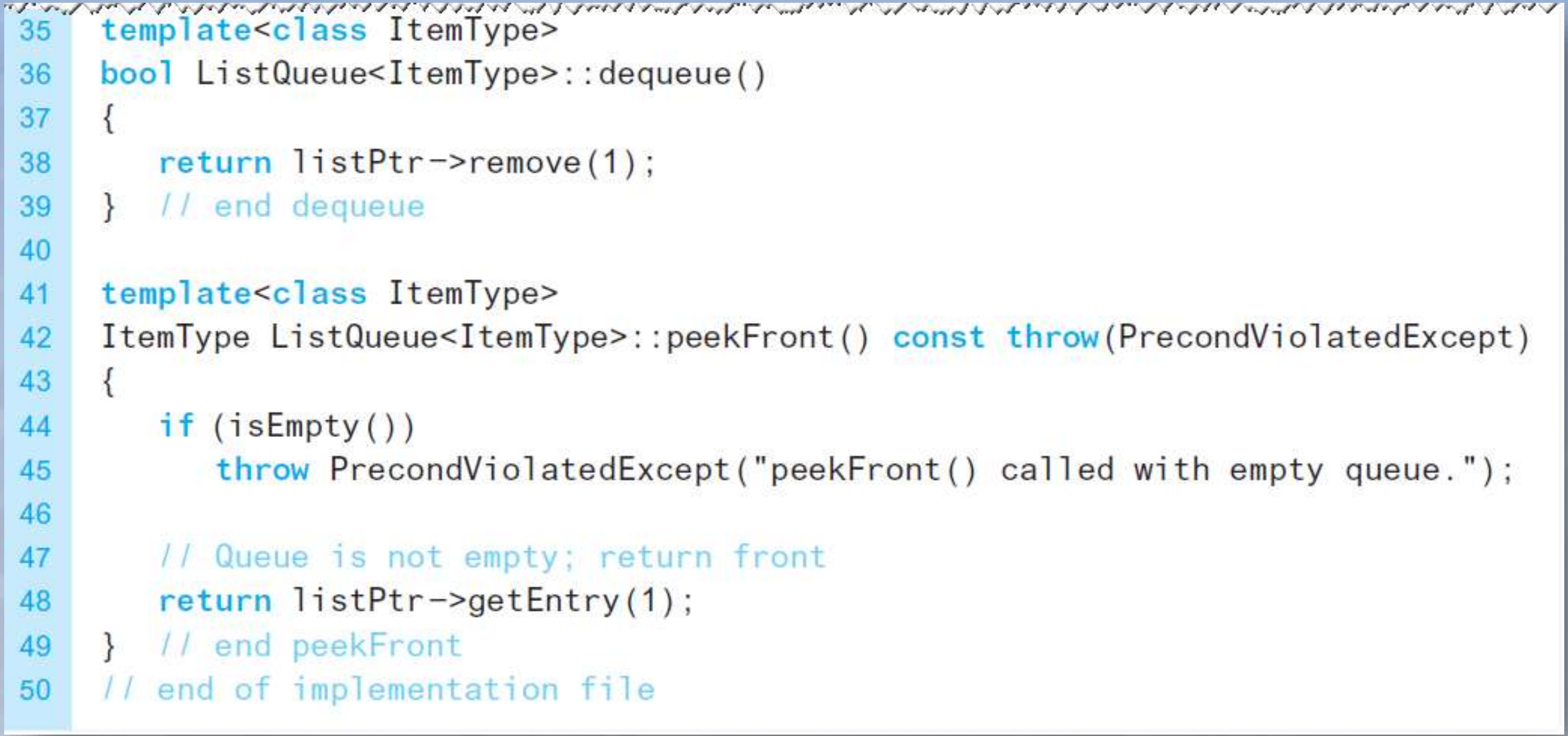
An Implementation That Uses the ADT List



```
template<class ItemType>
ListQueue<ItemType>::~ListQueue()
{
}  // end destructor

template<class ItemType>
bool ListQueue<ItemType>::isEmpty() const
{
   return listPtr->isEmpty();
}  // end isEmpty

template<class ItemType>
bool ListQueue<ItemType>::enqueue(const ItemType& newEntry)
{
   return listPtr->insert(listPtr->getLength() + 1, newEntry);
}  // end enqueue
```

LISTING 14-2 The implementation file for the class ListQueue

An Implementation That Uses the ADT List



```
template<class ItemType>
bool ListQueue<ItemType>::dequeue()
{
   return listPtr->remove(1);
}  // end dequeue

template<class ItemType>
ItemType ListQueue<ItemType>::peekFront() const throw(PrecondViolatedExcept)
{
   if (isEmpty())
      throw PrecondViolatedExcept("peekFront() called with empty queue.");

   // Queue is not empty; return front
   return listPtr->getEntry(1);
}  // end peekFront
// end of implementation file
```

LISTING 14-2 The implementation file for the class ListQueue

A Link-Based Implementation

- Similar to other link-based implementation
- One difference … Must be able to remove entries
  – From front
  – From back
- Requires a pointer to chain’s last node
  – Called the “tail pointer”

A Link-Based Implementation

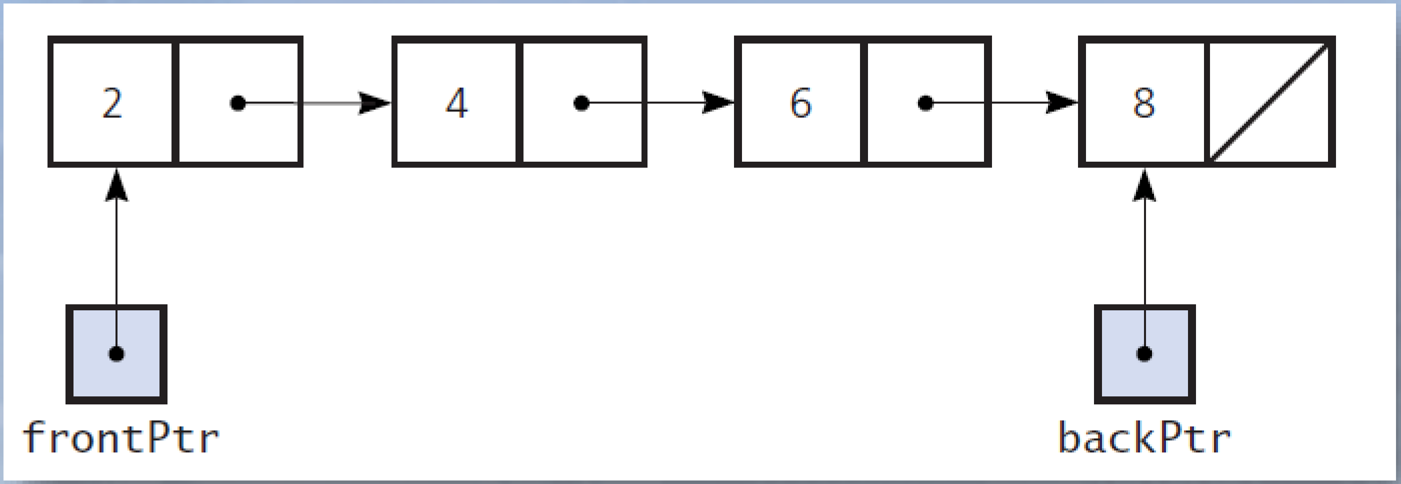
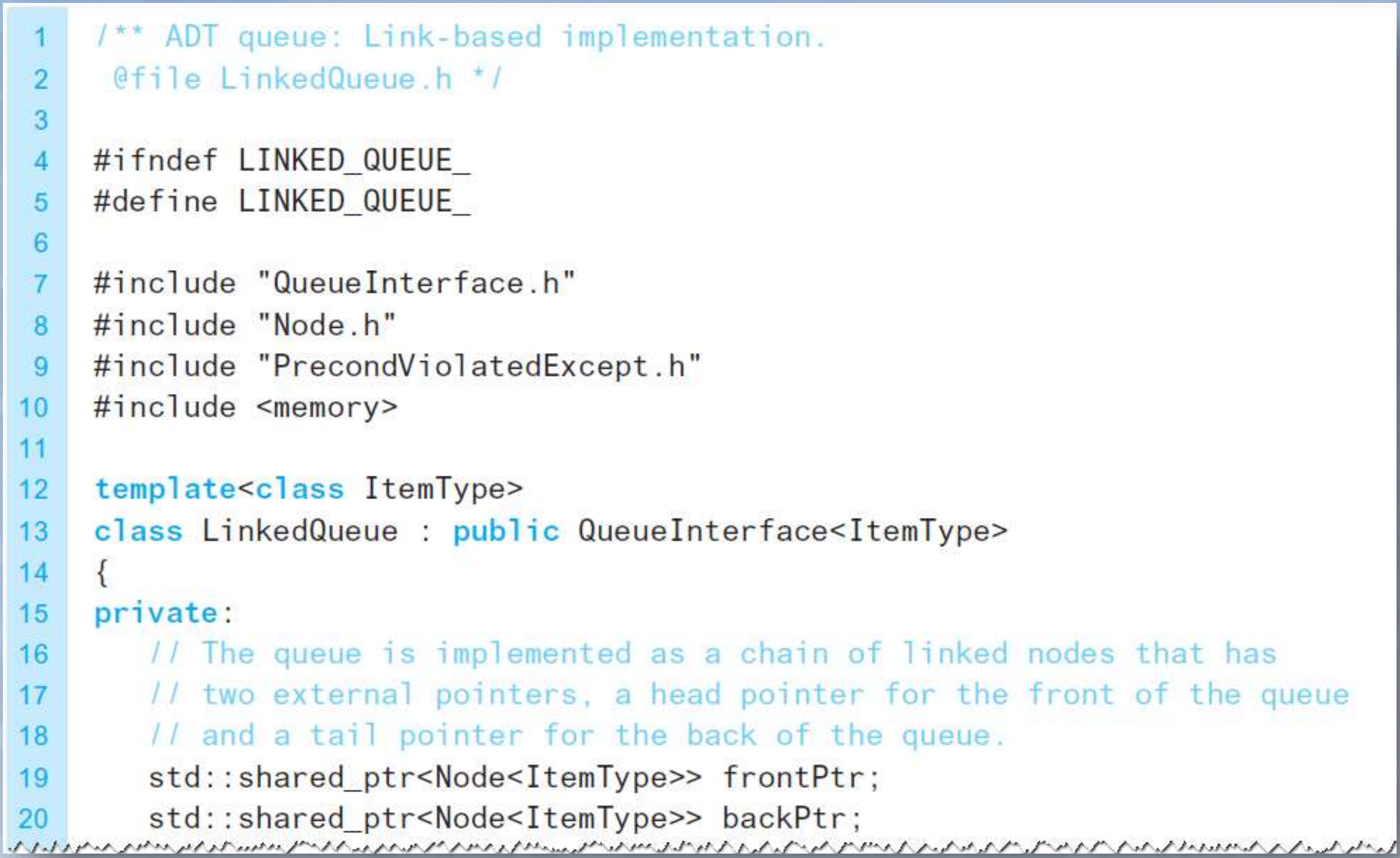


FIGURE 14-2 A chain of linked nodes with head and tail pointers

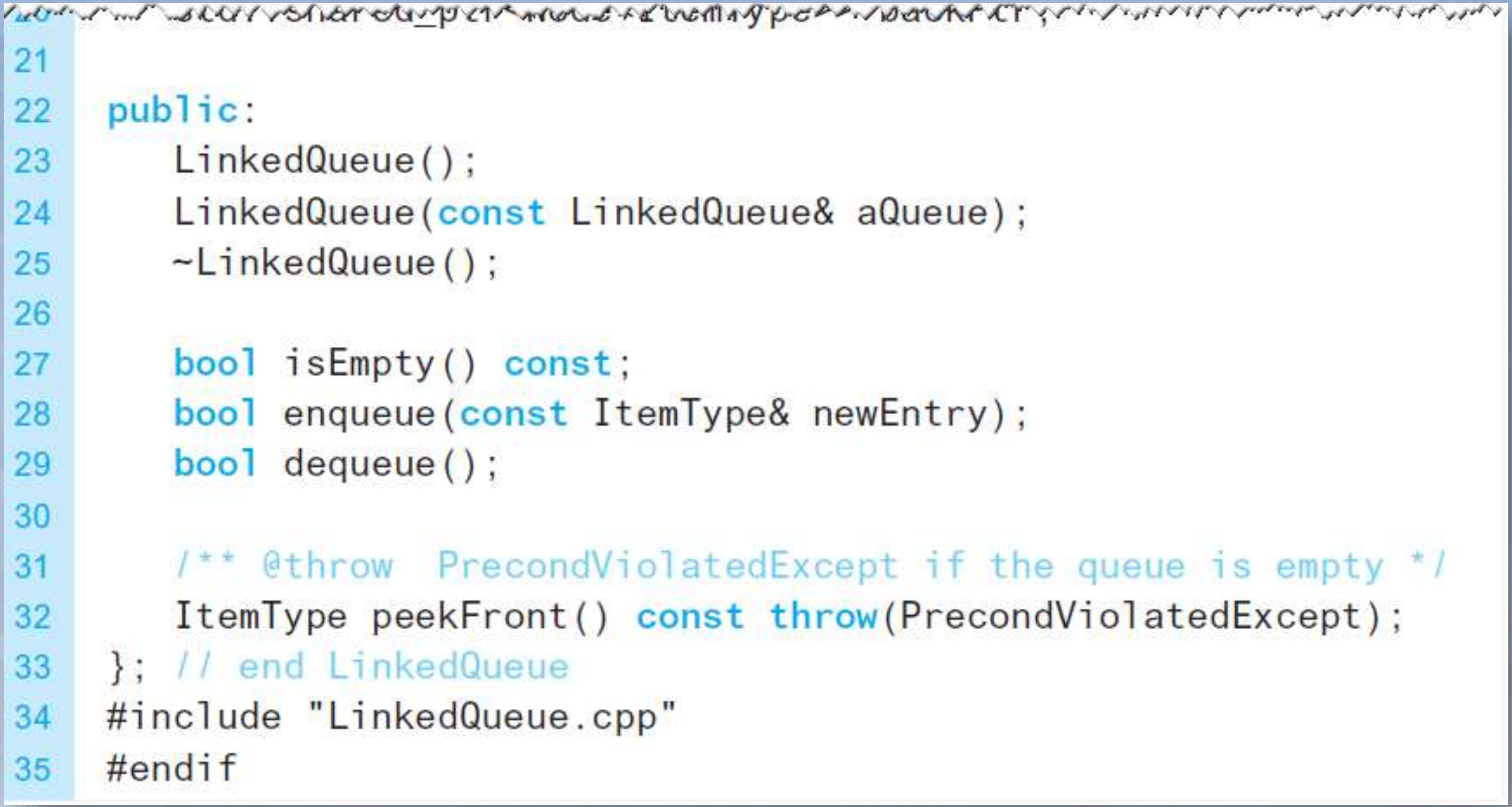
A Link-Based Implementation



```
/** ADT queue: Link-based implementation.
 @file LinkedQueue.h */

#ifndef LINKED_QUEUE_
#define LINKED_QUEUE_

#include "QueueInterface.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"
#include <memory>

template<class ItemType>
class LinkedQueue : public QueueInterface<ItemType>
{
private:
   // The queue is implemented as a chain of linked nodes that has
   // two external pointers, a head pointer for the front of the queue
   // and a tail pointer for the back of the queue.
   std::shared_ptr<Node<ItemType>> frontPtr;
   std::shared_ptr<Node<ItemType>> backPtr;
```

LISTING 14-3 The header file for the class LinkedQueue

A Link-Based Implementation



```
public:
   LinkedQueue();
   LinkedQueue(const LinkedQueue& aQueue);
   ~LinkedQueue();

   bool isEmpty() const;
   bool enqueue(const ItemType& newEntry);
   bool dequeue();

   /** @throw  PrecondViolatedExcept if the queue is empty */
   ItemType peekFront() const throw(PrecondViolatedExcept);
}; // end LinkedQueue
#include "LinkedQueue.cpp"
#endif
```

LISTING 14-3 The header file for the class LinkedQueue

A Link-Based Implementation

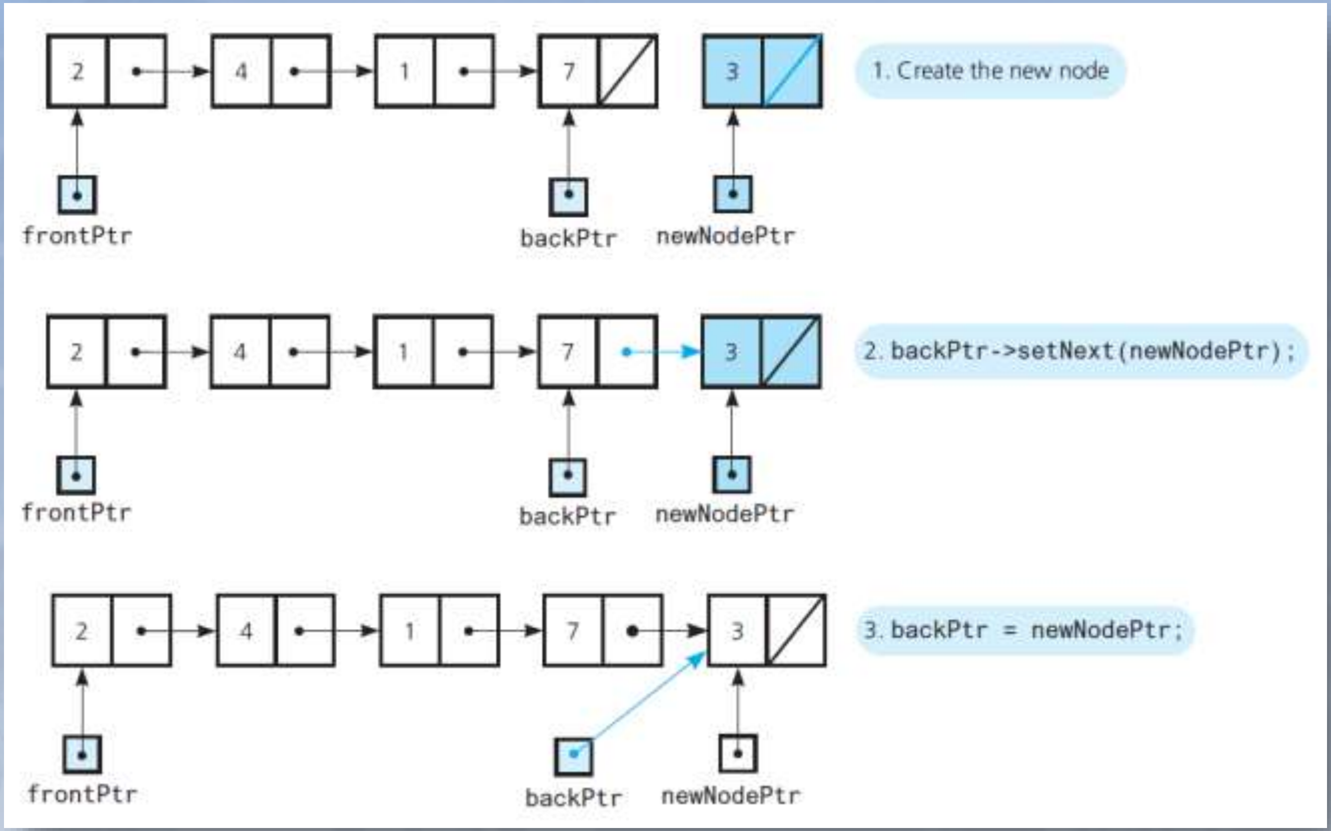


FIGURE 14-3 Adding an item to a nonempty queue

A Link-Based Implementation

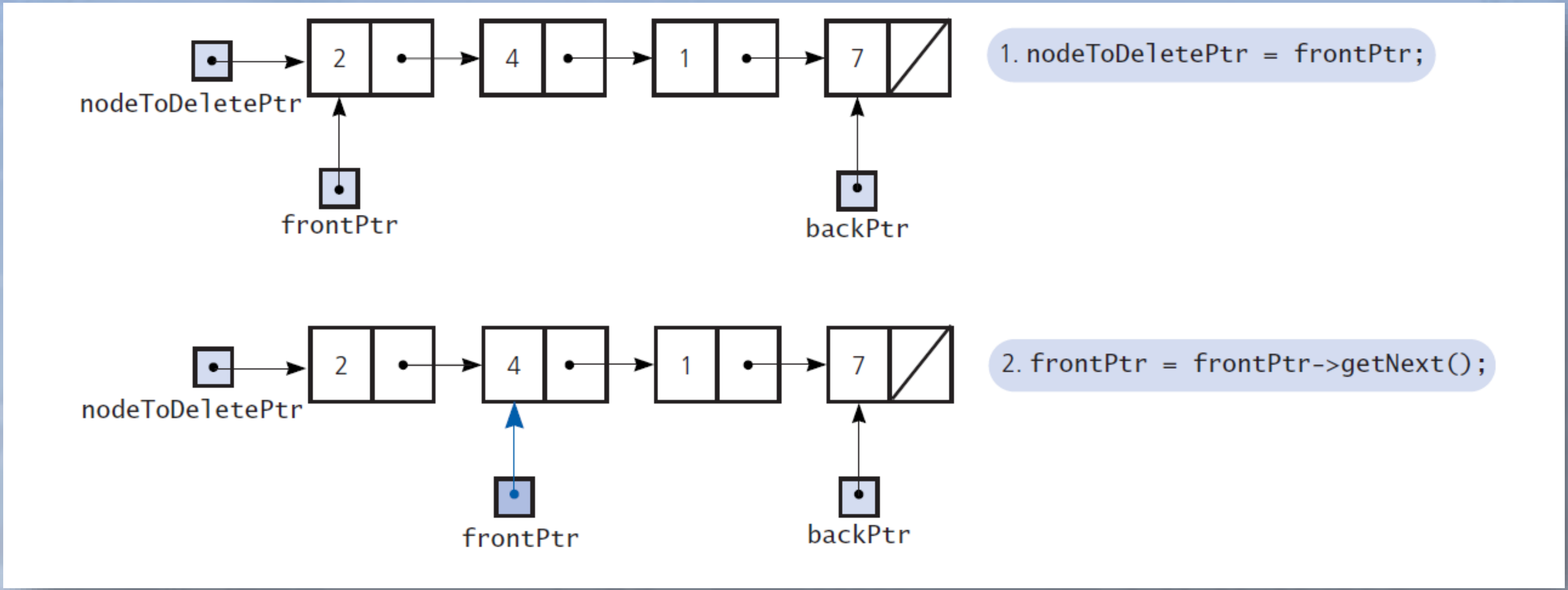


FIGURE 14-5 Removing an item from a queue of more than one item

A Link-Based Implementation

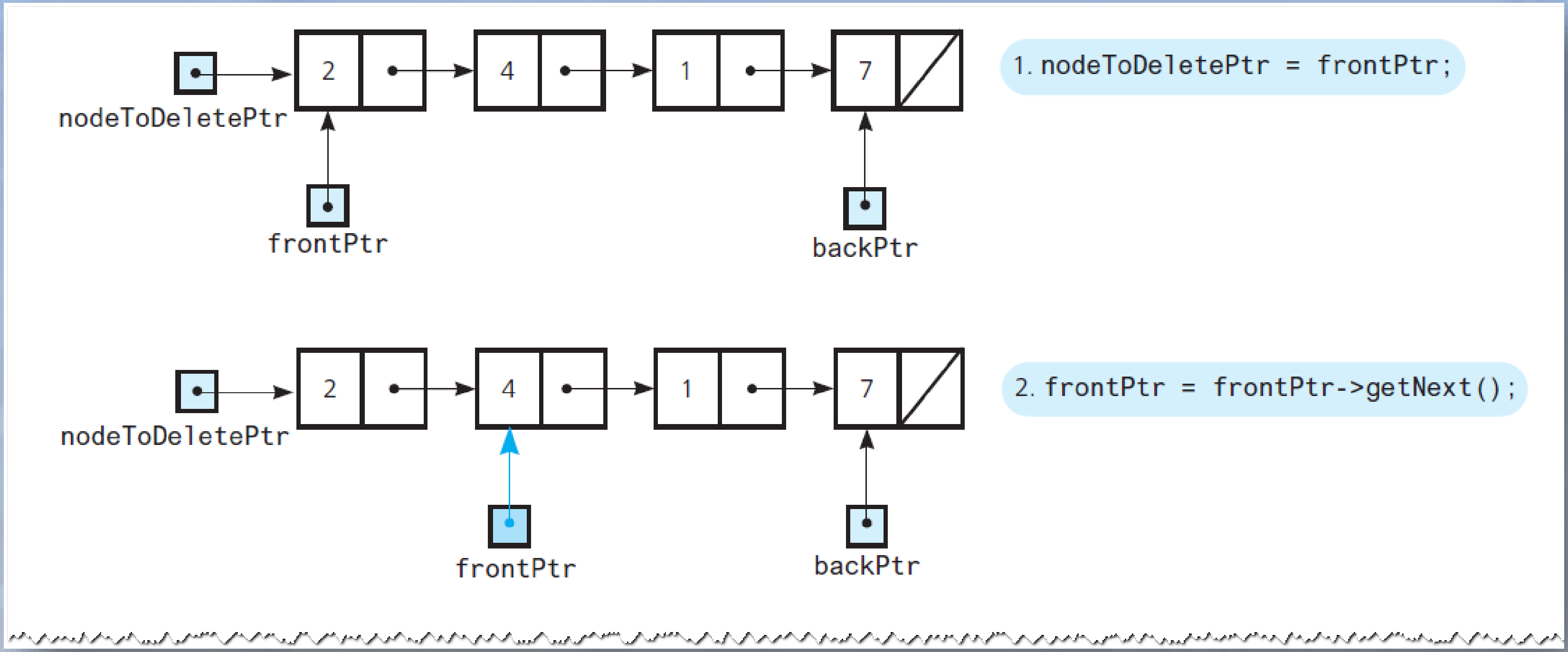


FIGURE 14-5 Removing an item from a queue of more than one item

A Link-Based Implementation

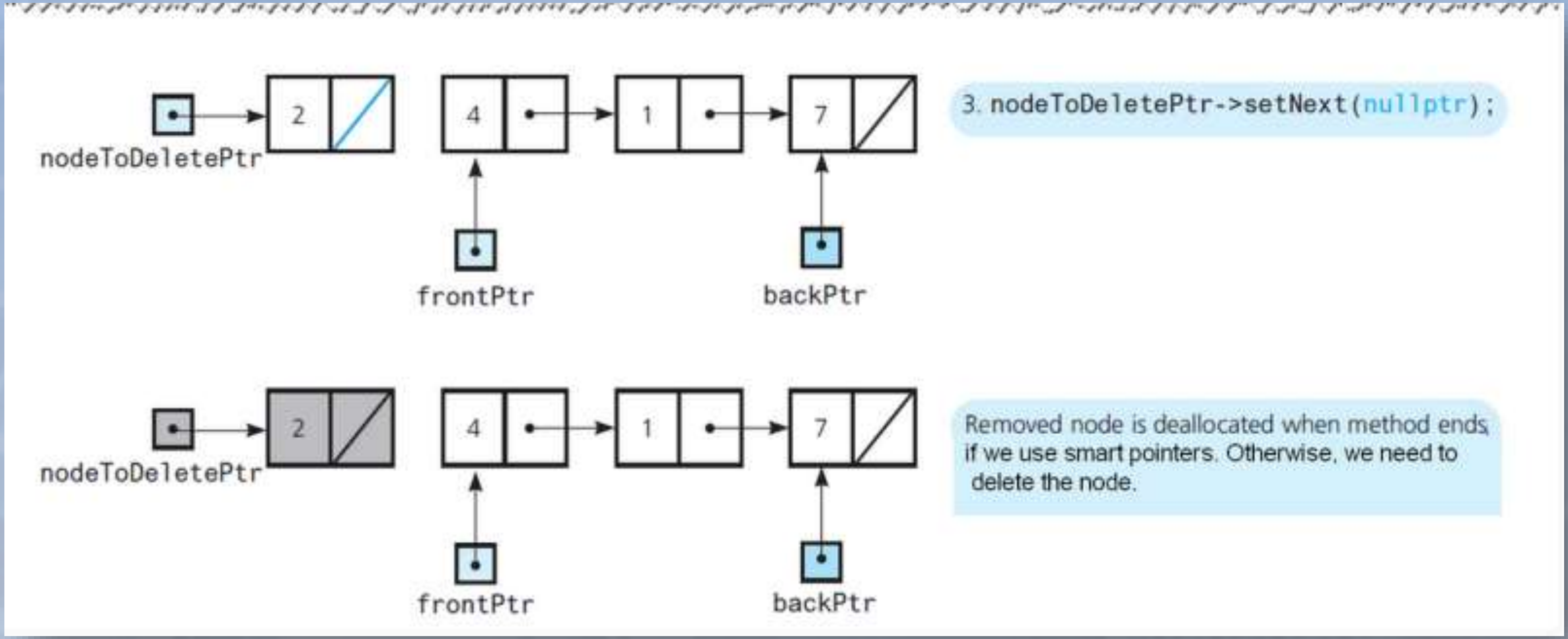

nodeToDeletePtr
2
4
1
7
frontPtr
backPtr
3. nodeToDeletePtr->setNext(nullptr);
nodeToDeletePtr
2
4
1
7
frontPtr
backPtr
Removed node is deallocated when method ends, if we use smart pointers. Otherwise, we need to delete the node.

FIGURE 14-5 Removing an item from a queue of more than one item

A Link-Based Implementation

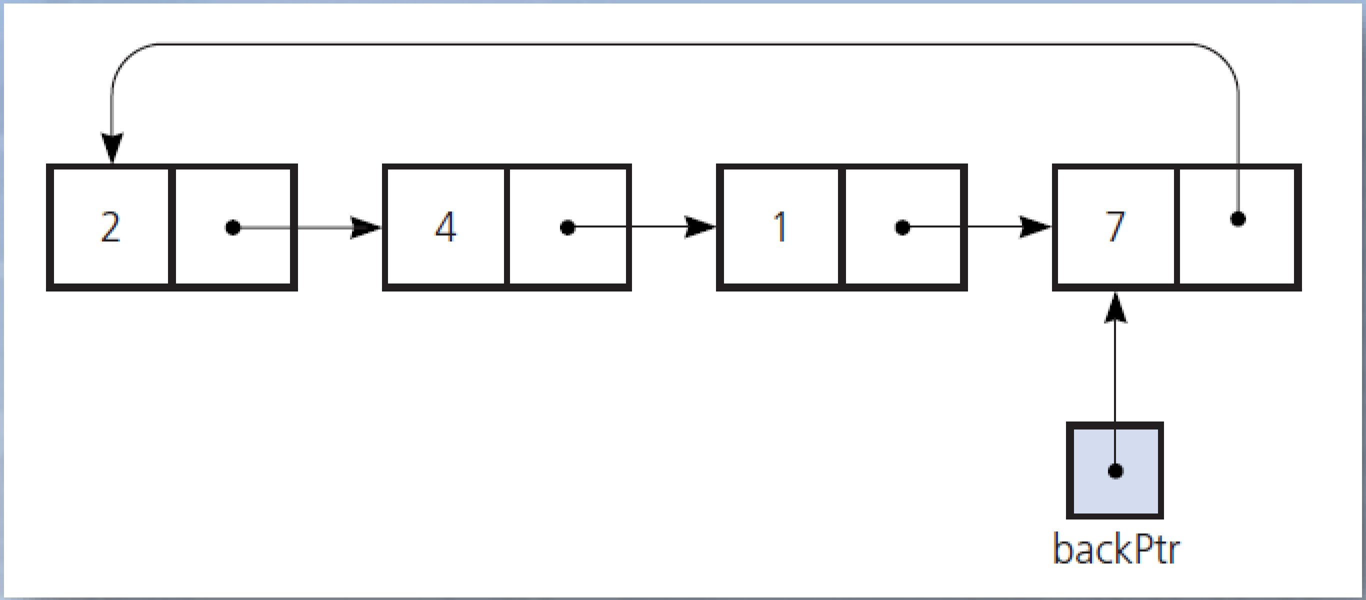


FIGURE 14-6 A circular chain of linked nodes with one external pointer

A Link-Based Implementation

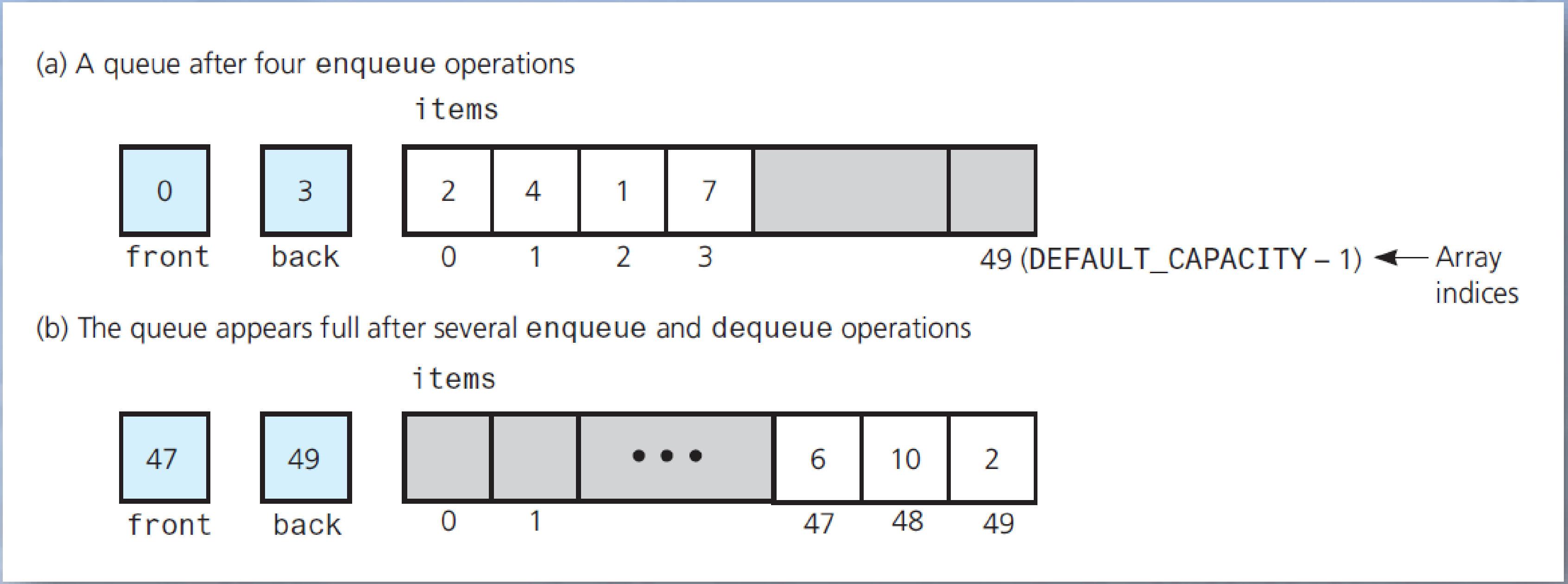

(a) A queue after four enqueue operations
items
0
3
front
back
2
4
1
7
0
1
2
3
49 (DEFAULT_CAPACITY – 1)
Array indices
(b) The queue appears full after several enqueue and dequeue operations
items
47
49
front
back
6
10
2
0
1
47
48
49

Figure 14-7 A naive array-based implementation of a queue for which rightward drift can cause the queue to appear full

A Link-Based Implementation

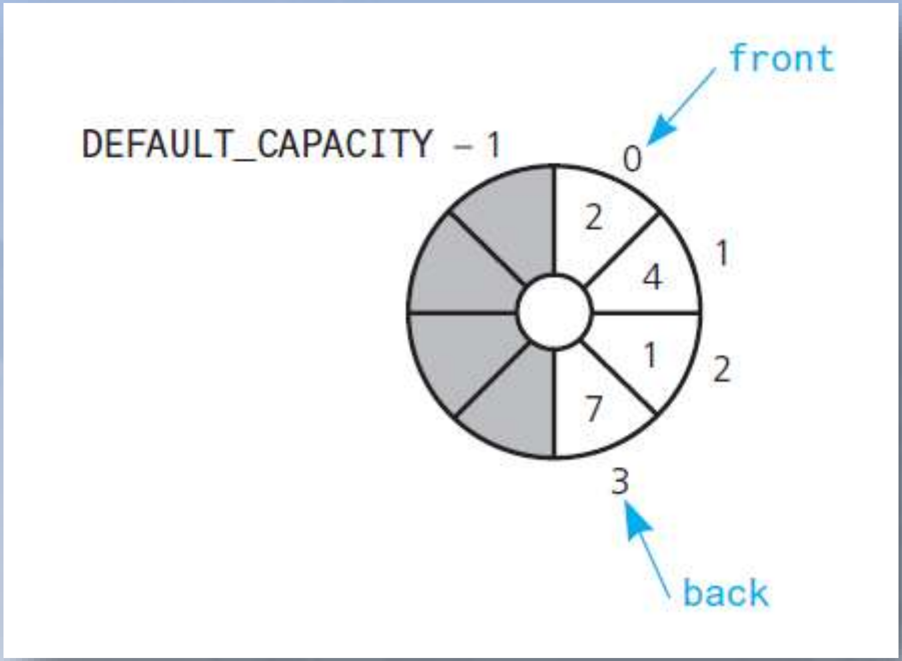


Figure 14-8 A circular array as an implementation of a queue

An Array-Based Implementation

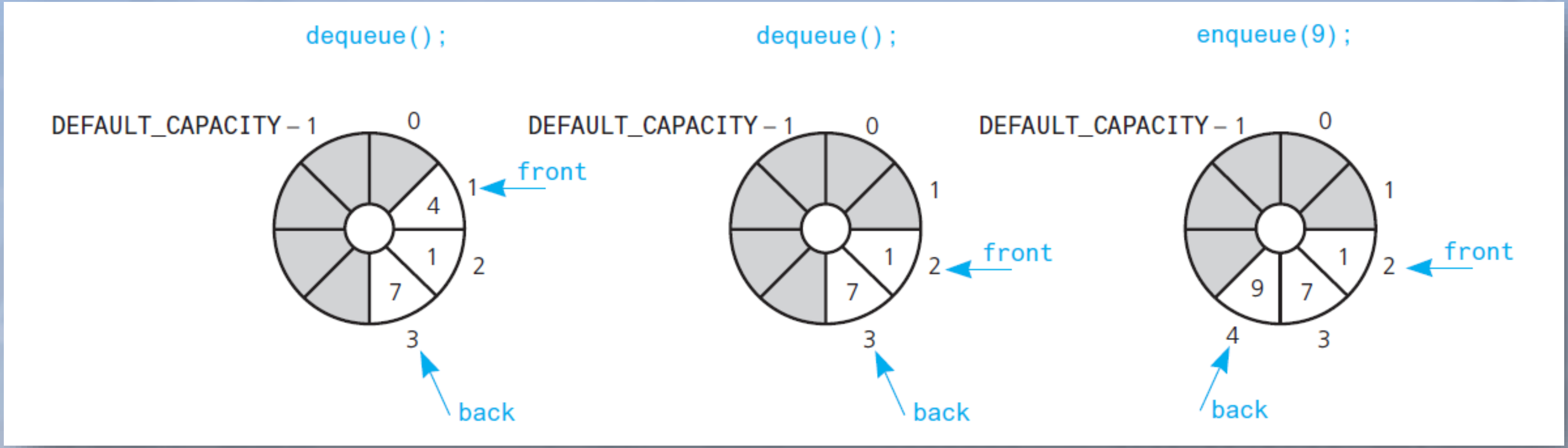


FIGURE 14-9 The effect of three consecutive operations on the queue in Figure 14-8

An Array-Based Implementation

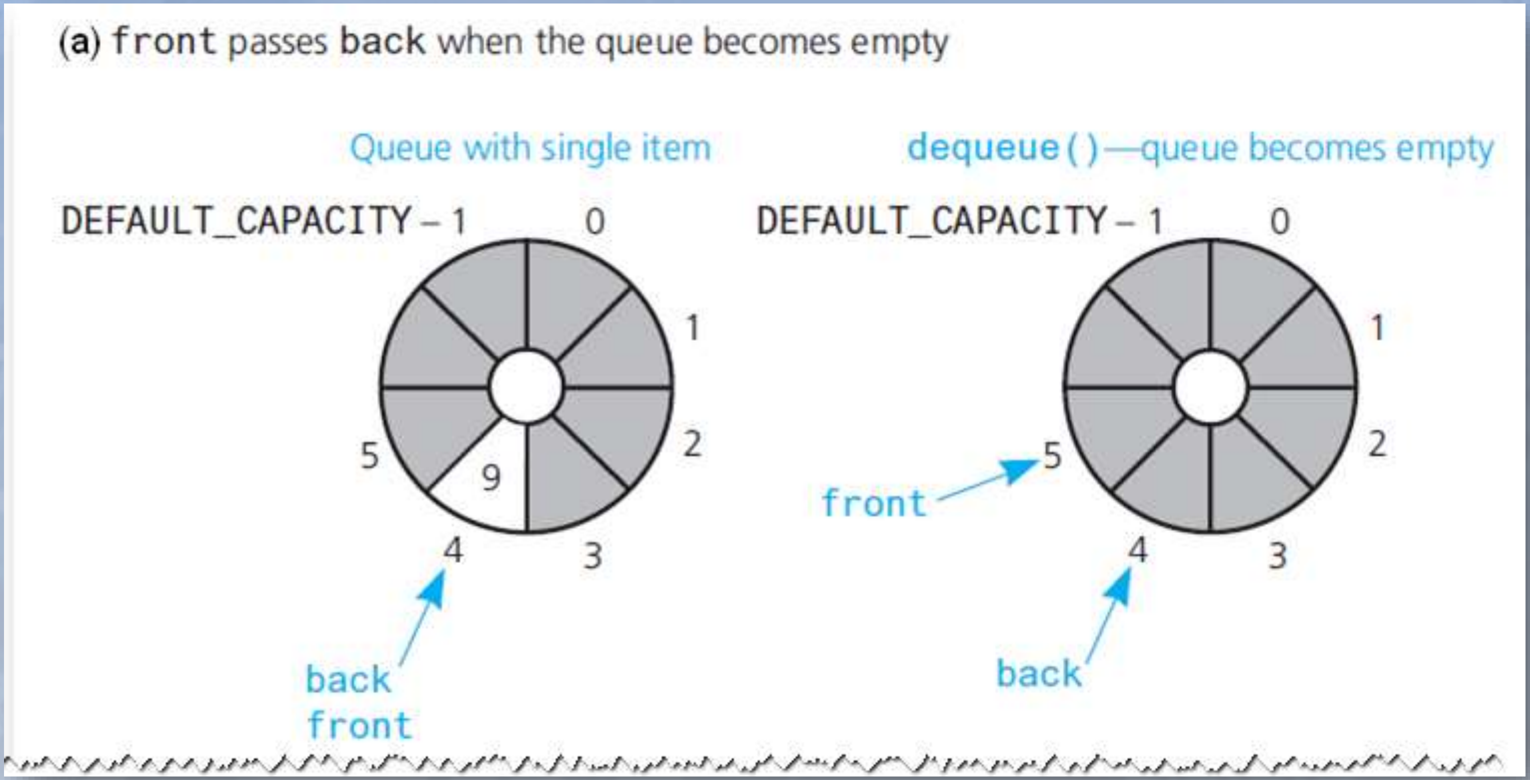


Figure1 4-10 front and back as the queue becomes empty and as it becomes full

An Array-Based Implementation

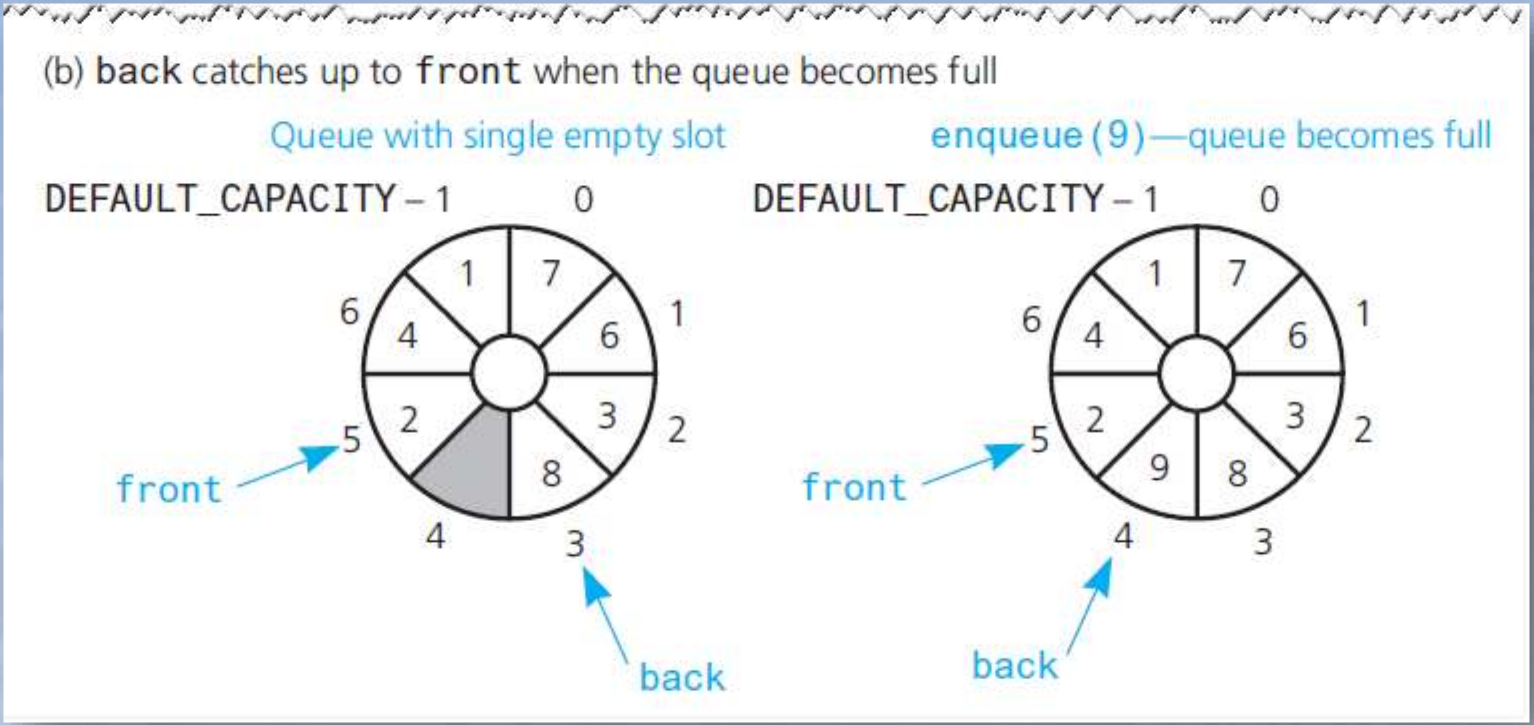
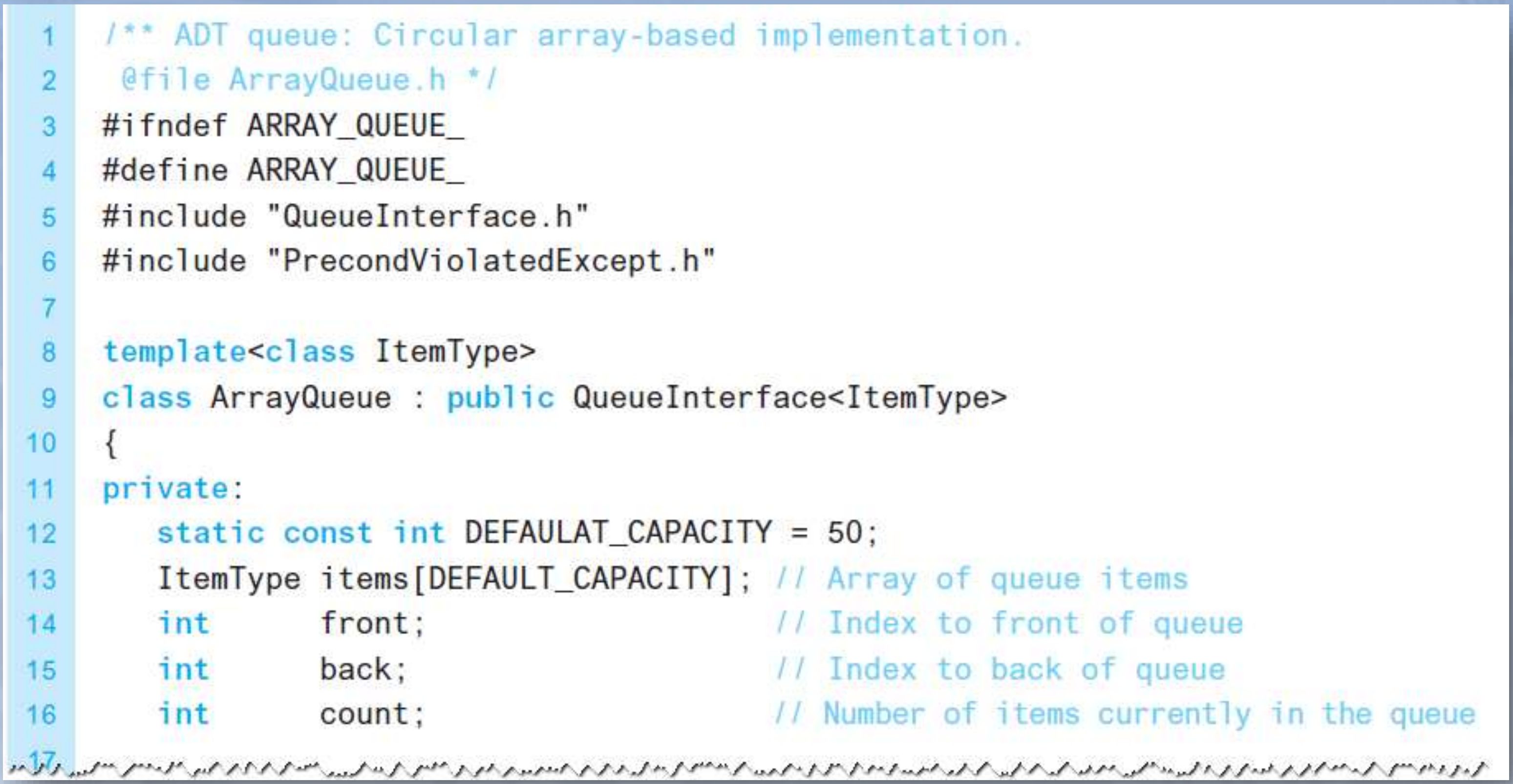

(b) back catches up to front when the queue becomes full
Queue with single empty slot
enqueue(9)—queue becomes full
DEFAULT_CAPACITY – 1
0
1
2
3
4
5
6
1
7
6
3
8
2
4
9
front
back

Figure1 4-10 front and back as the queue becomes empty and as it becomes full

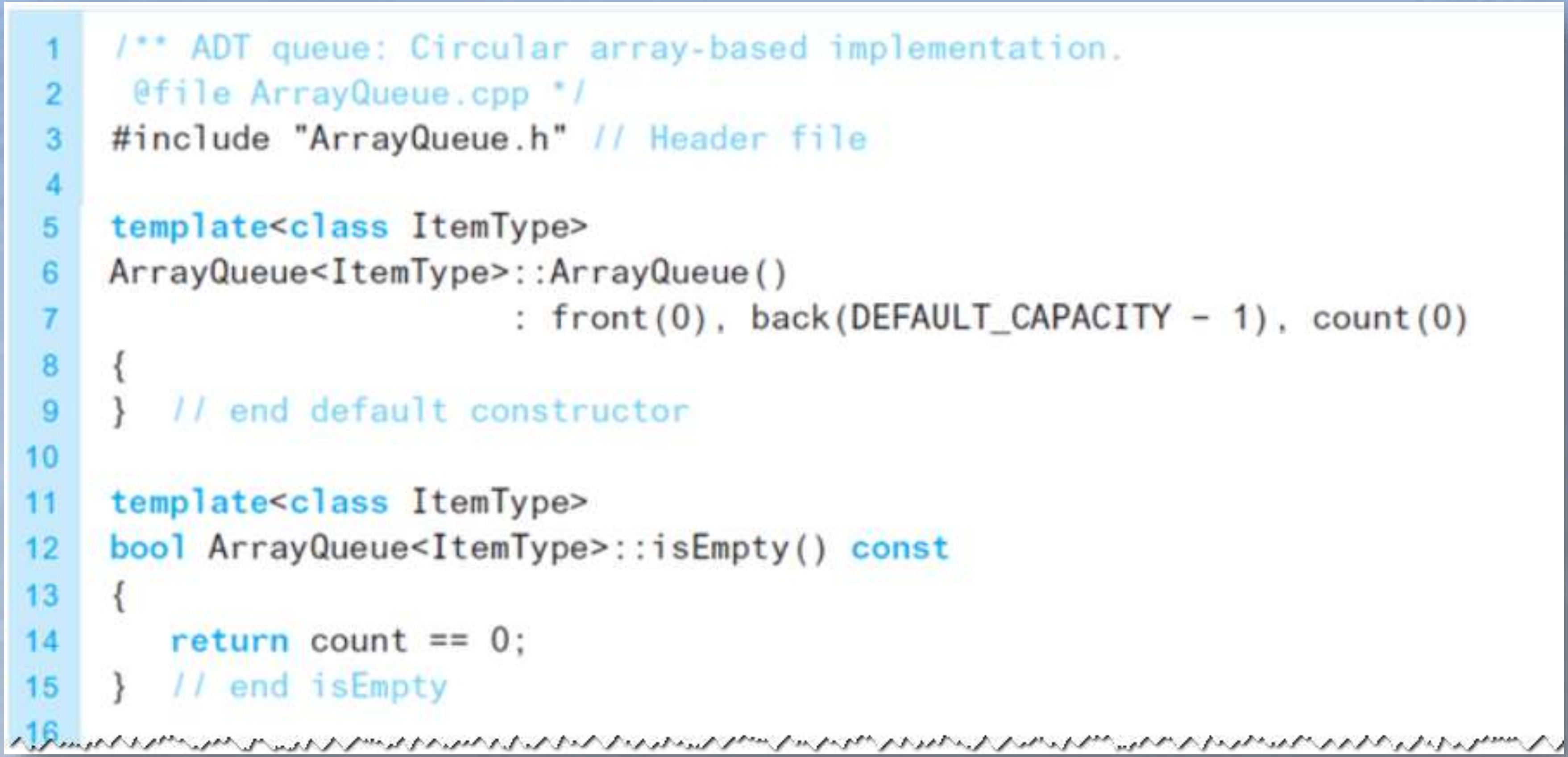
An Array-Based Implementation



```
/** ADT queue: Circular array-based implementation.
 @file ArrayQueue.h */
#ifndef ARRAY_QUEUE_
#define ARRAY_QUEUE_
#include "QueueInterface.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class ArrayQueue : public QueueInterface<ItemType>
{
private:
   static const int DEFAULAT_CAPACITY = 50;
   ItemType items[DEFAULT_CAPACITY]; // Array of queue items
   int      front;                   // Index to front of queue
   int      back;                    // Index to back of queue
   int      count;                   // Number of items currently in the queue
```

LISTING 14-4 The header file for the class ArrayQueue

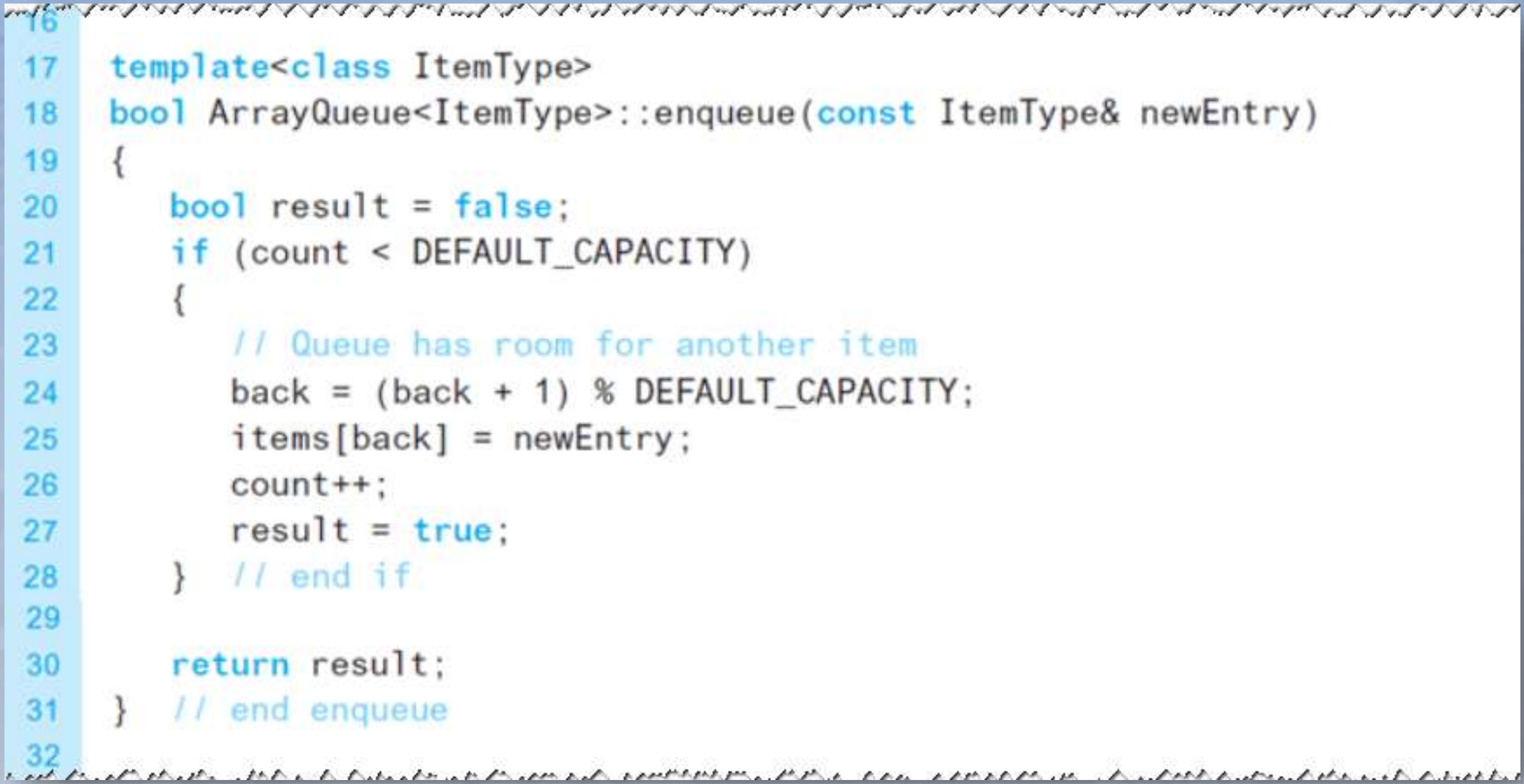
An Array-Based Implementation



```
/** ADT queue: Circular array-based implementation.
 @file ArrayQueue.cpp */
#include "ArrayQueue.h" // Header file

template<class ItemType>
ArrayQueue<ItemType>::ArrayQueue()
                    : front(0), back(DEFAULT_CAPACITY - 1), count(0)
{
}  // end default constructor

template<class ItemType>
bool ArrayQueue<ItemType>::isEmpty() const
{
   return count == 0;
}  // end isEmpty
```

Listing 14-5 The implementation file for the class ArrayQueue

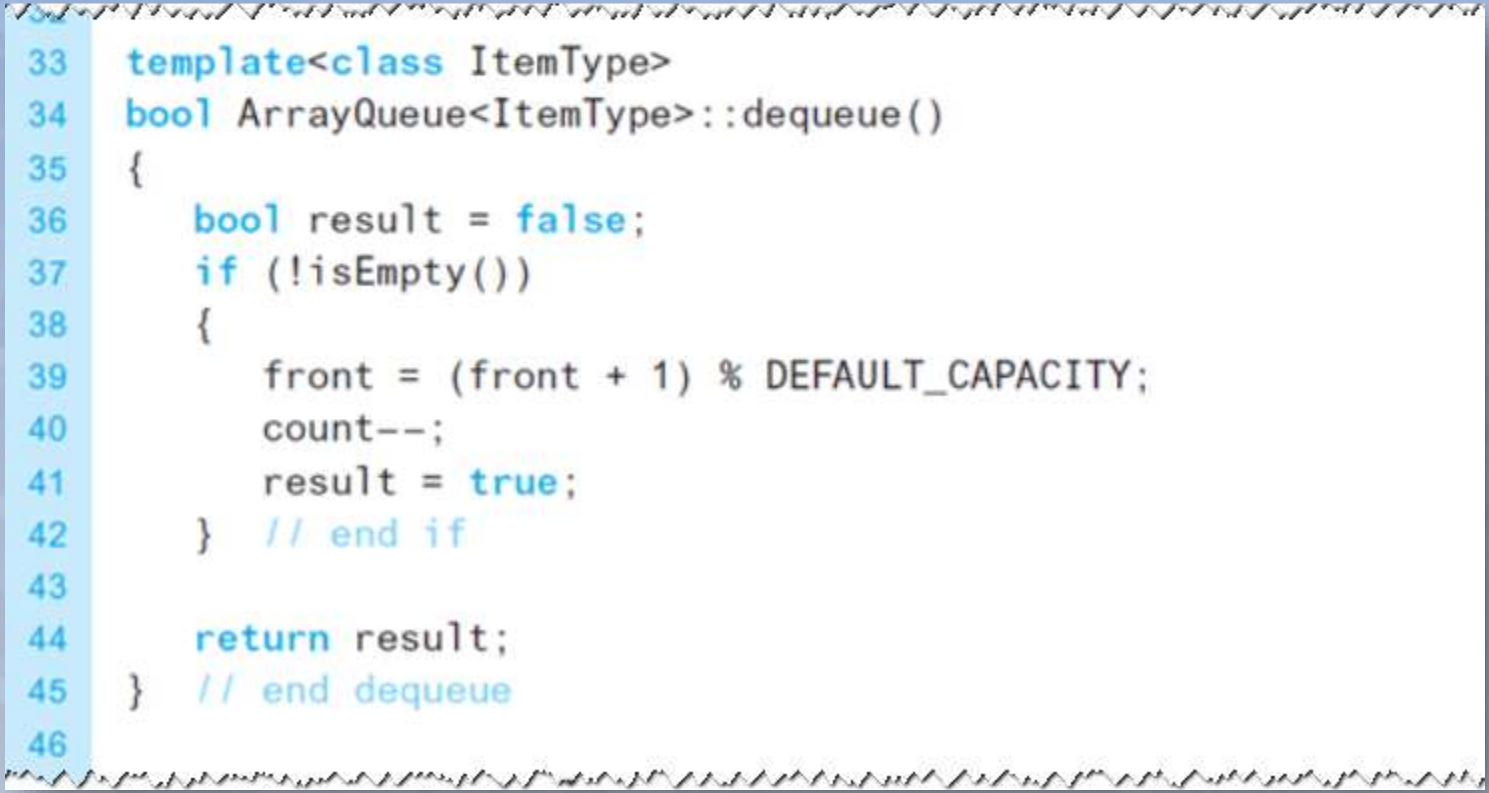
An Array-Based Implementation



```
template<class ItemType>
bool ArrayQueue<ItemType>::enqueue(const ItemType& newEntry)
{
   bool result = false;
   if (count < DEFAULT_CAPACITY)
   {
      // Queue has room for another item
      back = (back + 1) % DEFAULT_CAPACITY;
      items[back] = newEntry;
      count++;
      result = true;
   }  // end if

   return result;
}  // end enqueue
```

Listing 14-5 The implementation file for the class ArrayQueue

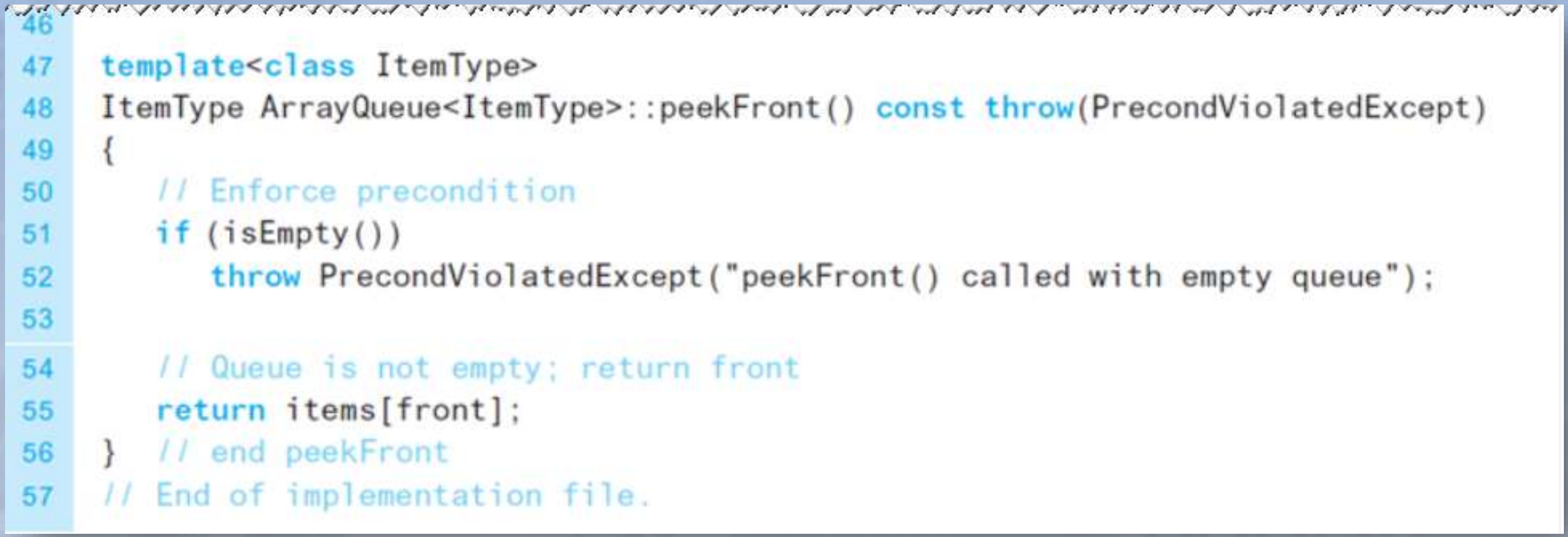
An Array-Based Implementation



```
template<class ItemType>
bool ArrayQueue<ItemType>::dequeue()
{
   bool result = false;
   if (!isEmpty())
   {
      front = (front + 1) % DEFAULT_CAPACITY;
      count--;
      result = true;
   }  // end if

   return result;
}  // end dequeue

```

Listing 14-5 The implementation file for the class ArrayQueue

An Array-Based Implementation



```
template<class ItemType>
ItemType ArrayQueue<ItemType>::peekFront() const throw(PrecondViolatedExcept)
{
   // Enforce precondition
   if (isEmpty())
      throw PrecondViolatedExcept("peekFront() called with empty queue");

   // Queue is not empty; return front
   return items[front];
}  // end peekFront
// End of implementation file.
```

Listing 14-5 The implementation file for the class ArrayQueue

An Array-Based Implementation

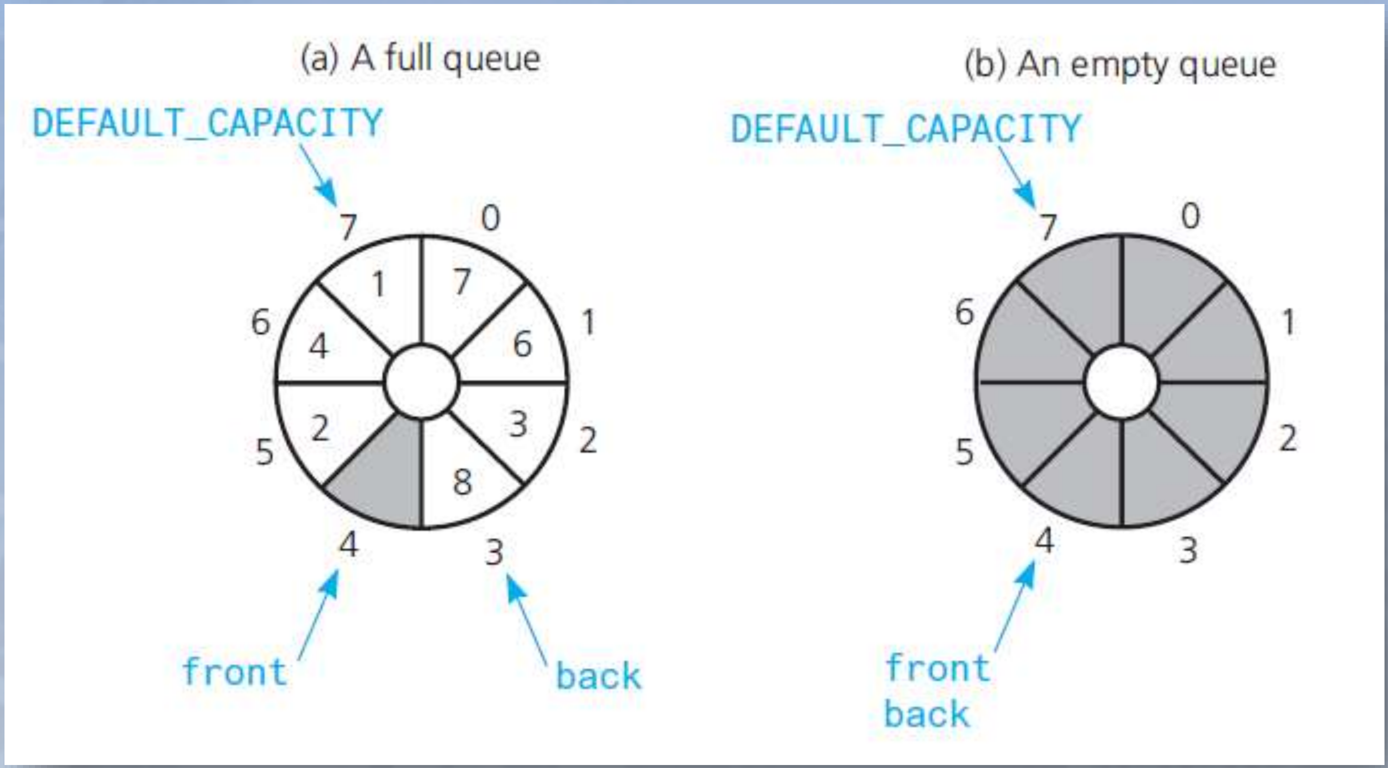
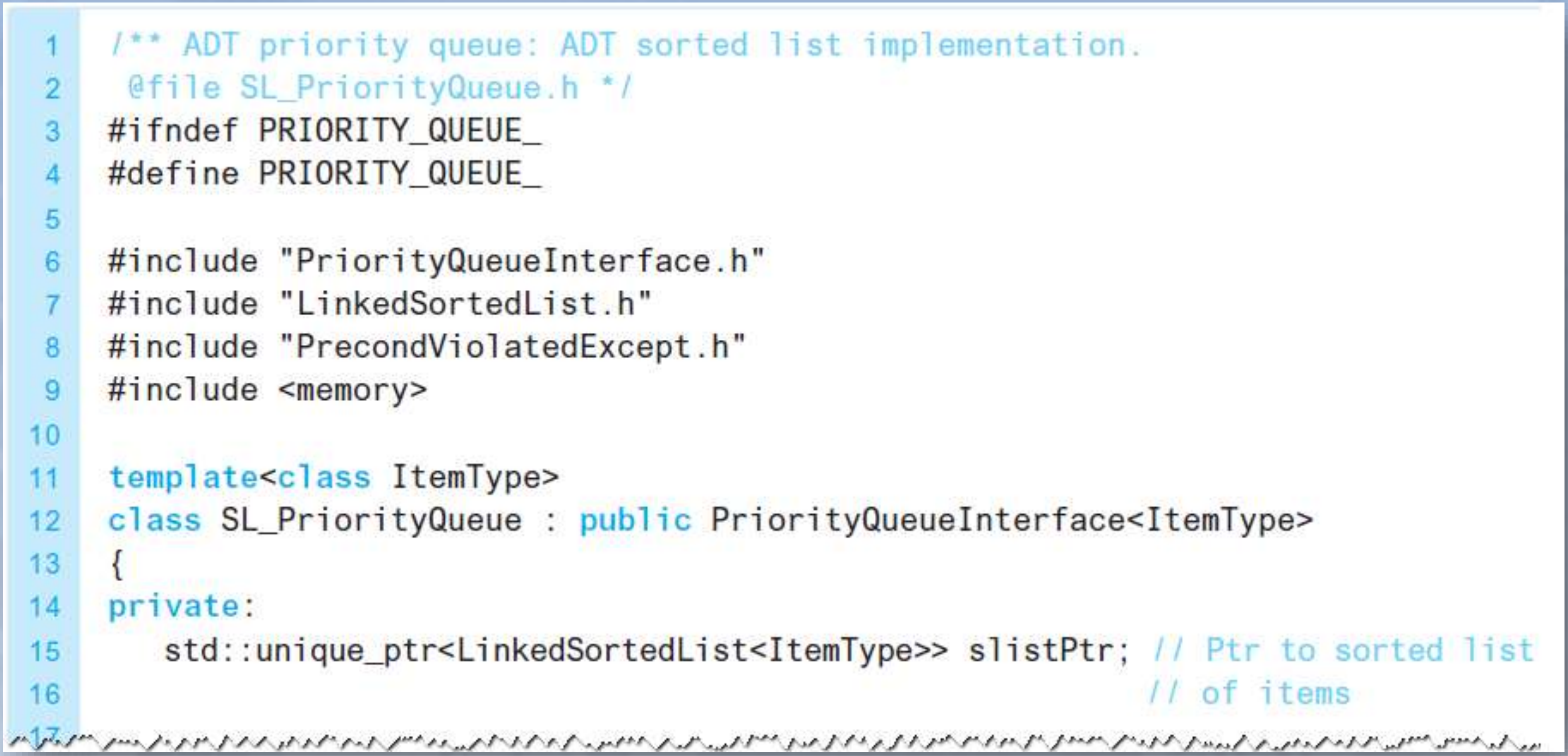


Figure 14-11 A circular array having one unused location as an implementation of a queue

Comparing Implementations

- Issues
  - Fixed size (array-based) versus dynamic size (link-based)
  - Reuse of already implemented class saves time

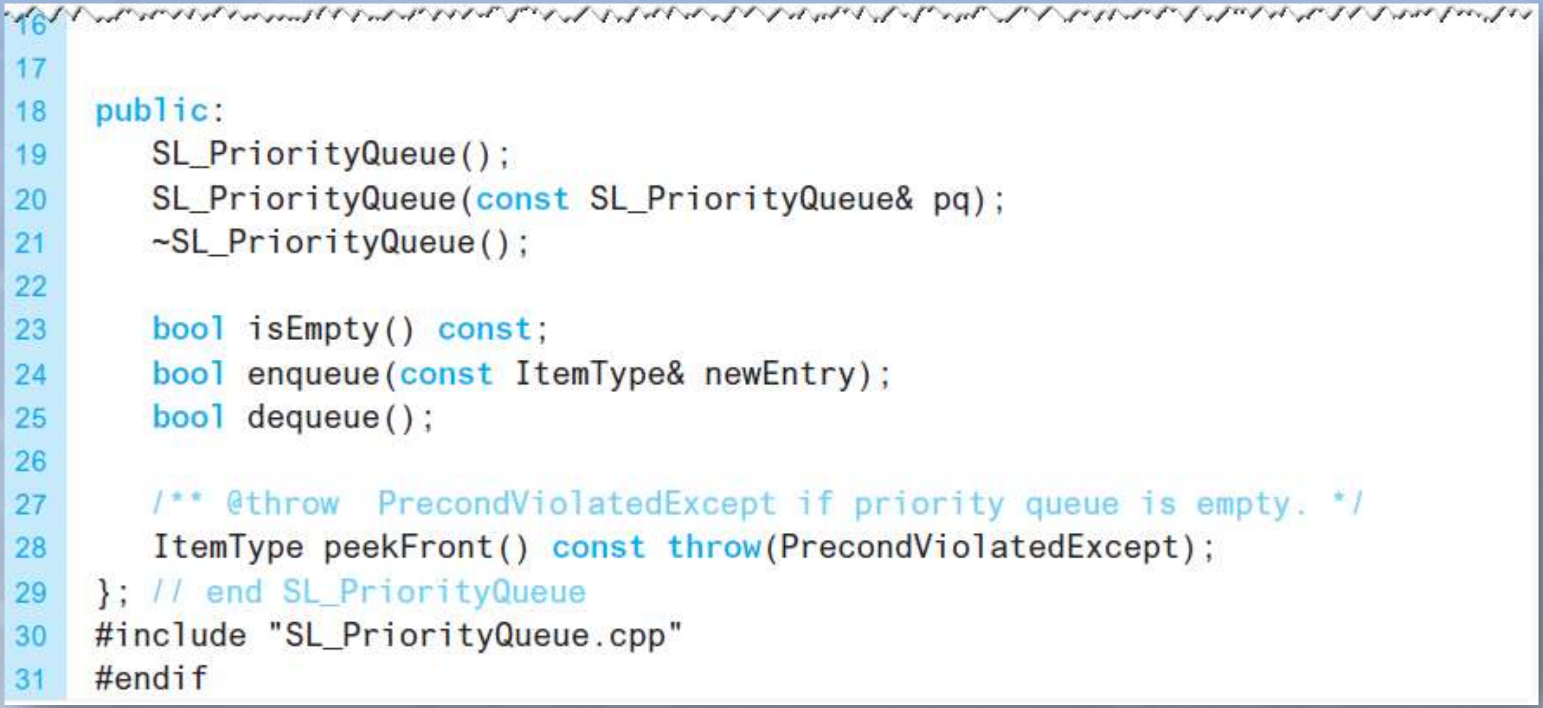
An Implementation of the ADT Priority Queue



```
/** ADT priority queue: ADT sorted list implementation.
 @file SL_PriorityQueue.h */
#ifndef PRIORITY_QUEUE_
#define PRIORITY_QUEUE_

#include "PriorityQueueInterface.h"
#include "LinkedSortedList.h"
#include "PrecondViolatedExcept.h"
#include <memory>

template<class ItemType>
class SL_PriorityQueue : public PriorityQueueInterface<ItemType>
{
private:
   std::unique_ptr<LinkedSortedList<ItemType>> slistPtr; // Ptr to sorted list
                                                         // of items
```

LISTING 14-6 A header file for the class SL_PriorityQueue.

An Implementation of the ADT Priority Queue



```
public:
   SL_PriorityQueue();
   SL_PriorityQueue(const SL_PriorityQueue& pq);
   ~SL_PriorityQueue();

   bool isEmpty() const;
   bool enqueue(const ItemType& newEntry);
   bool dequeue();

   /** @throw  PrecondViolatedExcept if priority queue is empty. */
   ItemType peekFront() const throw(PrecondViolatedExcept);
}; // end SL_PriorityQueue
#include "SL_PriorityQueue.cpp"
#endif
```

LISTING 14-6 A header file for the class SL_PriorityQueue.

End

Chapter 14