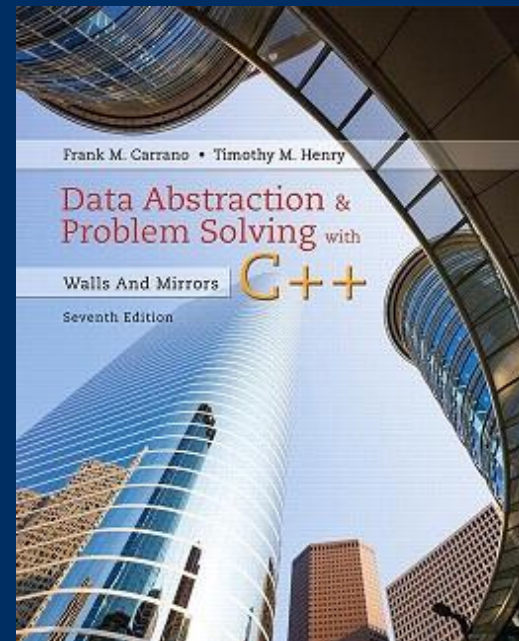


Chapter 14: Queue and Priority Queue Implementations

CS 302 - Data Structures

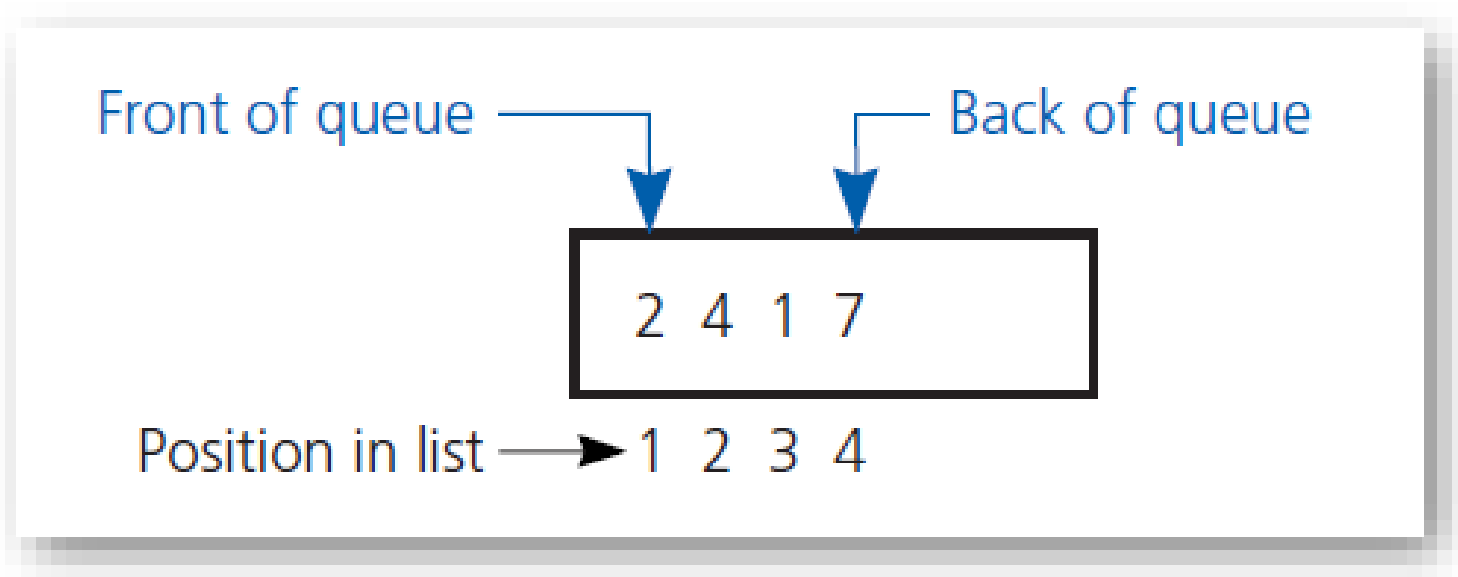
M. Abdullah Canbaz





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 of the ADT queue that stores its entries in a list

```
1  /** ADT queue: ADT list implementation.
2   * @file ListQueue.h */
3
4  #ifndef LIST_QUEUE_
5  #define LIST_QUEUE_
6
7  #include "QueueInterface.h"
8  #include "LinkedList.h"
9  #include "PrecondViolatedExcept.h"
10 #include <memory>
11
12 template<class ItemType>
13 class ListQueue : public QueueInterface<ItemType>
14 {
15 private:
```

- The header file for the class **ListQueue**

```
16     std::unique_ptr<LinkedList<ItemType>> listPtr; // Pointer to list of queue items
17
18 public:
19     ListQueue();
20     ListQueue(const ListQueue& aQueue);
21     ~ListQueue();
22     bool isEmpty() const;
23     bool enqueue(const ItemType& newEntry);
24     bool dequeue();
25
26     /** @throw PrecondViolatedExcept if this queue is empty. */
27     ItemType peekFront() const throw(PrecondViolatedExcept);
28 }; // end ListQueue
29 #include "ListQueue.cpp"
30 #endif
```

- The header file for the class **ListQueue**

```
1  /** ADT queue: ADT list implementation.
2   @file ListQueue.cpp */
3  #include "ListQueue.h" // Header file
4  #include <memory>
5
6  template<class ItemType>
7  ListQueue<ItemType>::ListQueue()
8      : listPtr(std::make_unique<LinkedList<ItemType>>())
9  {
10 } // end default constructor
11
12 template<class ItemType>
13 ListQueue<ItemType>::ListQueue(const ListQueue& aQueue)
14     : listPtr(aQueue.listPtr)
15 {
16 } // end copy constructor
17
```

- The implementation file for the class **ListQueue**

```
17
18  template<class ItemType>
19  ListQueue<ItemType>::~~ListQueue()
20  {
21  } // end destructor
22
23  template<class ItemType>
24  bool ListQueue<ItemType>::isEmpty() const
25  {
26      return listPtr->isEmpty();
27  } // end isEmpty
28
29  template<class ItemType>
30  bool ListQueue<ItemType>::enqueue(const ItemType& newEntry)
31  {
32      return listPtr->insert(listPtr->getLength() + 1, newEntry);
33  } // end enqueue
```

- The implementation file for the class **ListQueue**

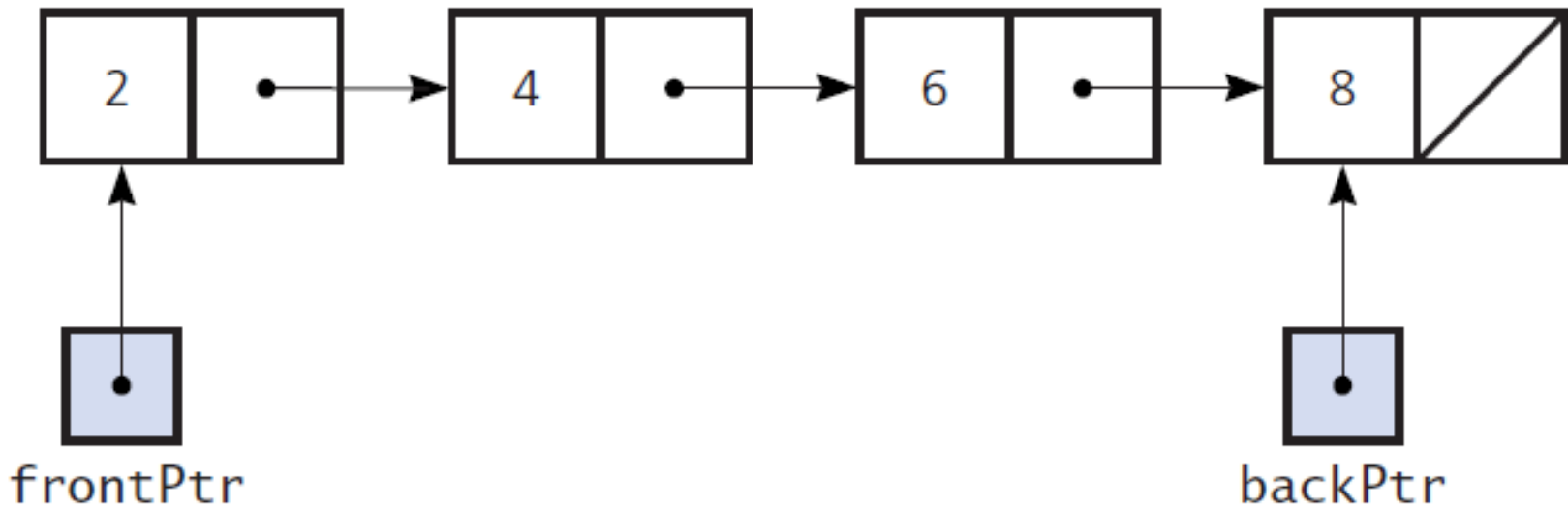
```
35  template<class ItemType>
36  bool ListQueue<ItemType>::dequeue()
37  {
38      return listPtr->remove(1);
39  } // end dequeue
40
41  template<class ItemType>
42  ItemType ListQueue<ItemType>::peekFront() const throw(PrecondViolatedExcept)
43  {
44      if (isEmpty())
45          throw PrecondViolatedExcept("peekFront() called with empty queue.");
46
47      // Queue is not empty; return front
48      return listPtr->getEntry(1);
49  } // end peekFront
50  // end of implementation file
```

- 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 chain of linked nodes with head and tail pointers

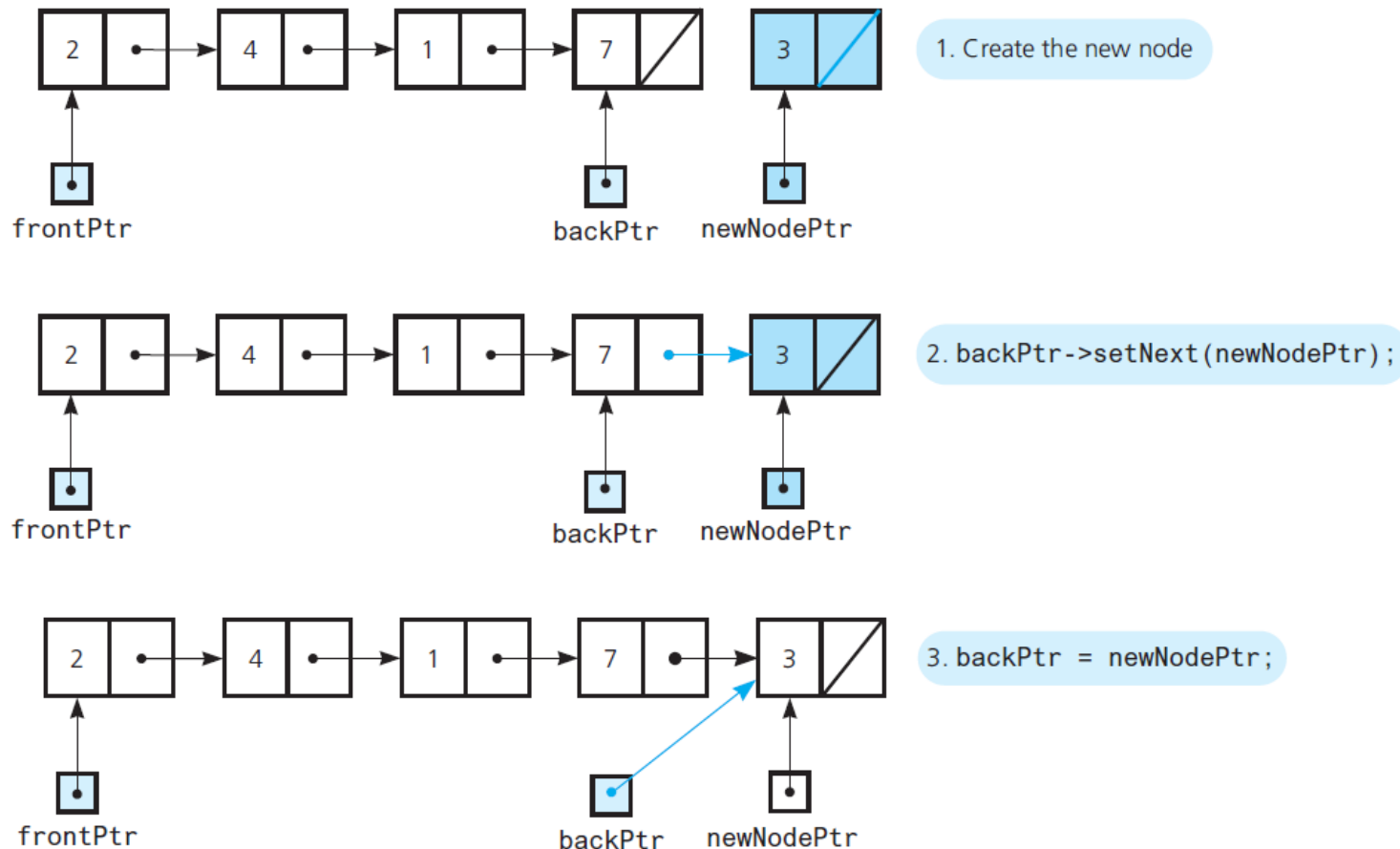
- The header file for the class `LinkedList`

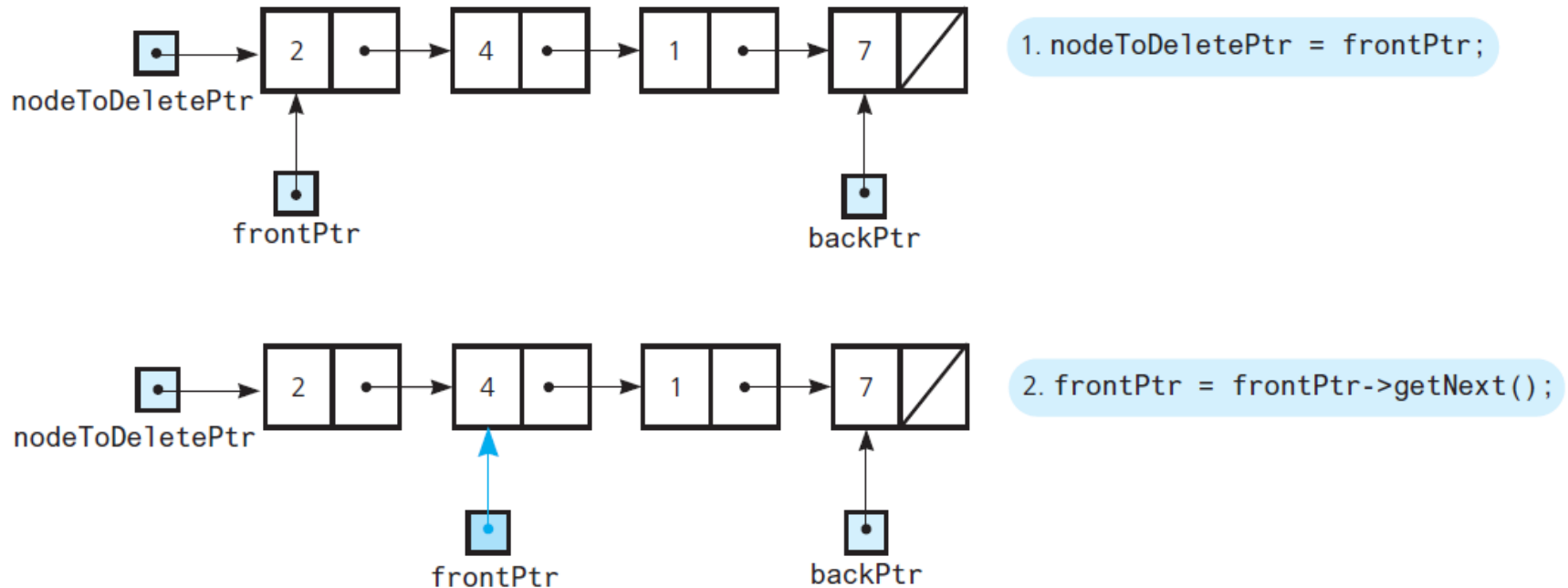
```
1  /** ADT queue: Link-based implementation.
2   * @file LinkedList.h */
3
4  #ifndef LINKED_QUEUE_
5  #define LINKED_QUEUE_
6
7  #include "QueueInterface.h"
8  #include "Node.h"
9  #include "PrecondViolatedExcept.h"
10 #include <memory>
11
12 template<class ItemType>
13 class LinkedList : public QueueInterface<ItemType>
14 {
15 private:
16     // The queue is implemented as a chain of linked nodes that has
17     // two external pointers, a head pointer for the front of the queue
18     // and a tail pointer for the back of the queue.
19     std::shared_ptr<Node<ItemType>> frontPtr;
20     std::shared_ptr<Node<ItemType>> backPtr;
```

- The header file for the class `LinkedList`

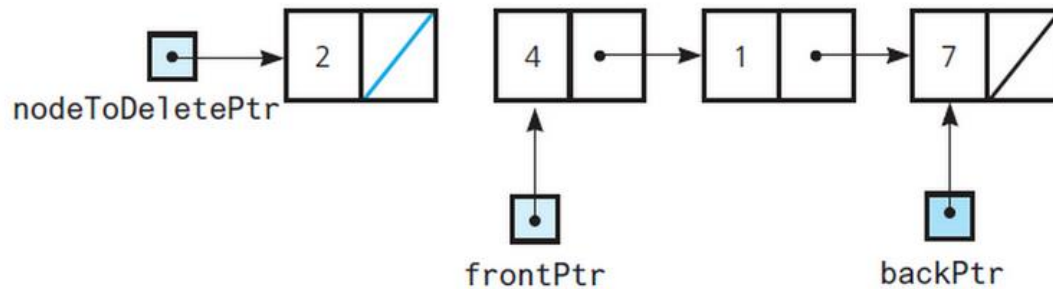
```
20 // ...
21
22 public:
23     LinkedList();
24     LinkedList(const LinkedList& aQueue);
25     ~LinkedList();
26
27     bool isEmpty() const;
28     bool enqueue(const ItemType& newEntry);
29     bool dequeue();
30
31     /** @throw PrecondViolatedExcept if the queue is empty */
32     ItemType peekFront() const throw(PrecondViolatedExcept);
33 }; // end LinkedList
34 #include "LinkedList.cpp"
35 #endif
```

- Adding an item to a nonempty queue

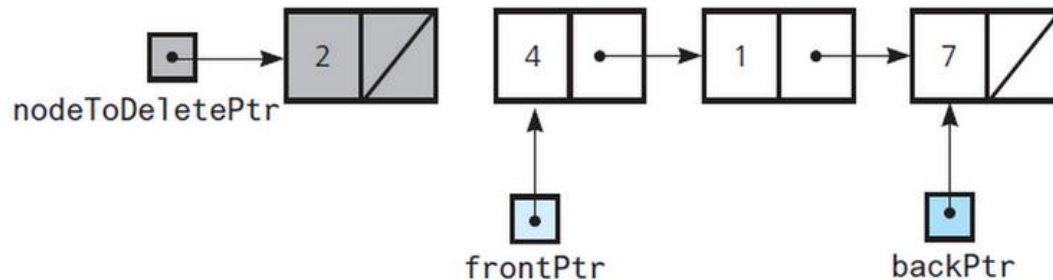




- Removing an item from a queue of more than one item

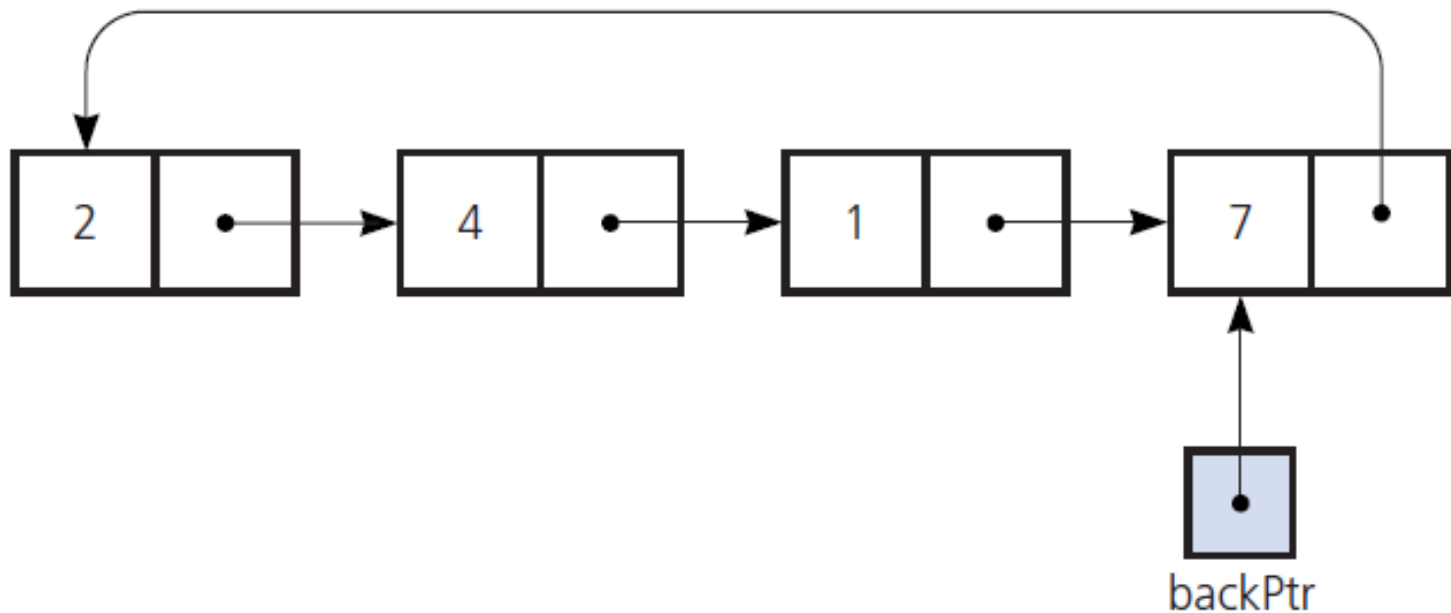


```
3. nodeToDeletePtr->setNext(nullptr);
```

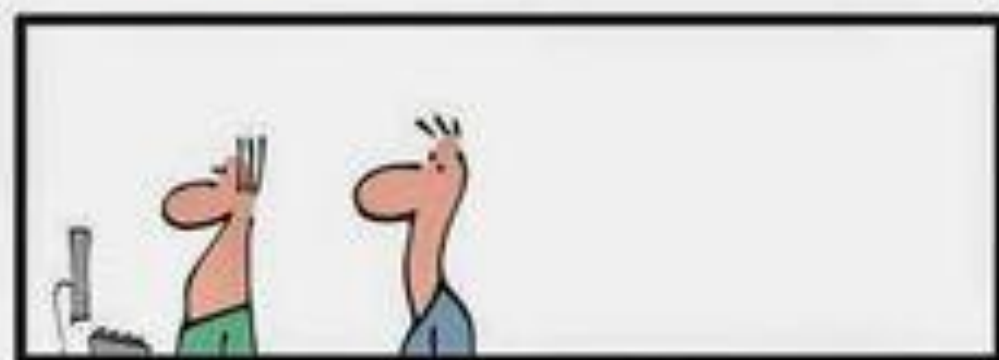


Removed node is deallocated when method ends, if we use smart pointers. Otherwise, we need to delete the node.

- Removing an item from a queue of more than one item



- A circular chain of linked nodes with one external pointer

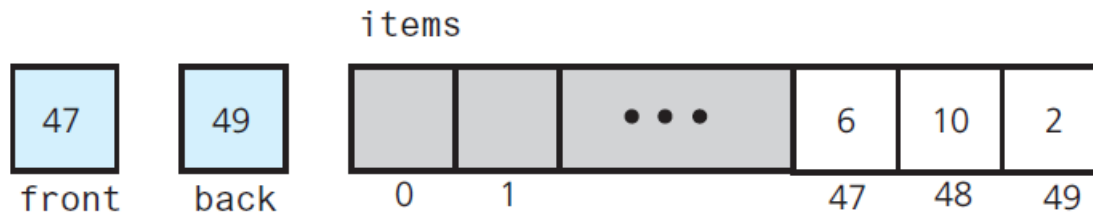


THE ART OF PROGRAMMING - PART 2: KISS

(a) A queue after four enqueue operations



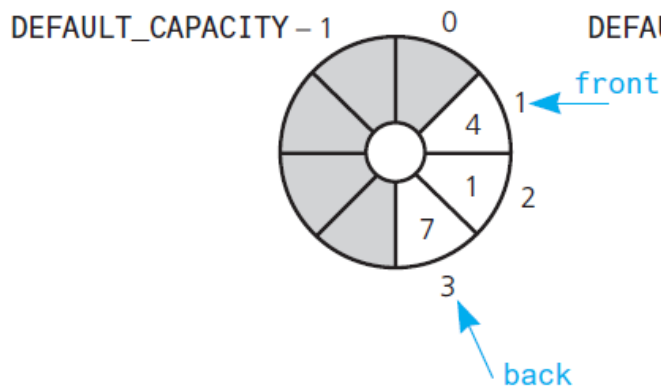
(b) The queue appears full after several enqueue and dequeue operations



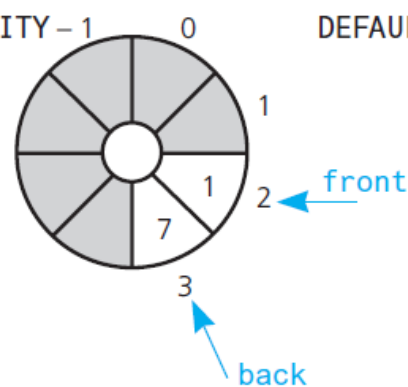
- A naive array-based implementation of a queue for which rightward drift can cause the queue to appear full

An Array-Based Implementation

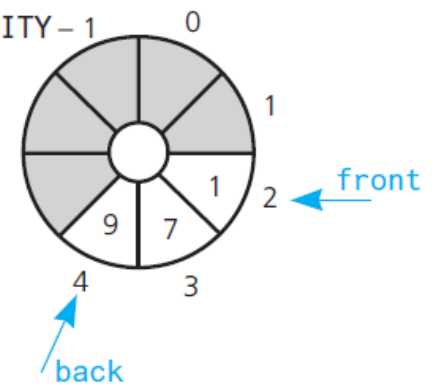
dequeue();



dequeue();



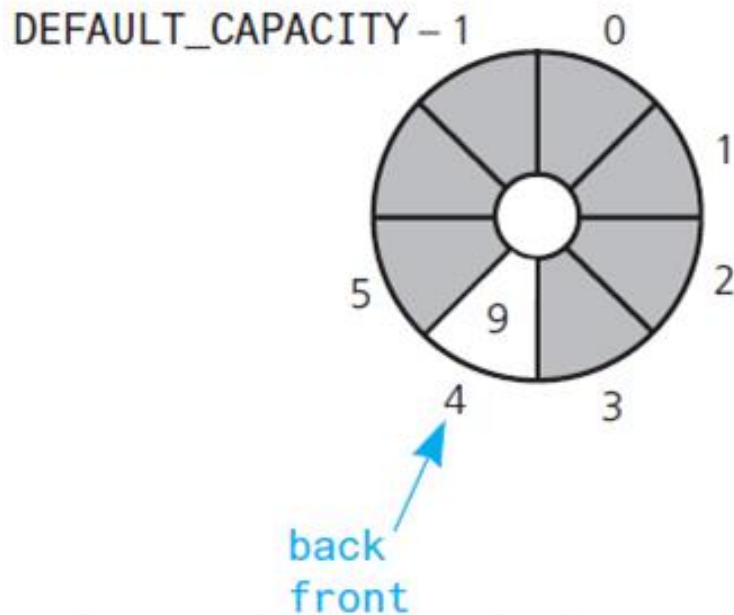
enqueue(9);



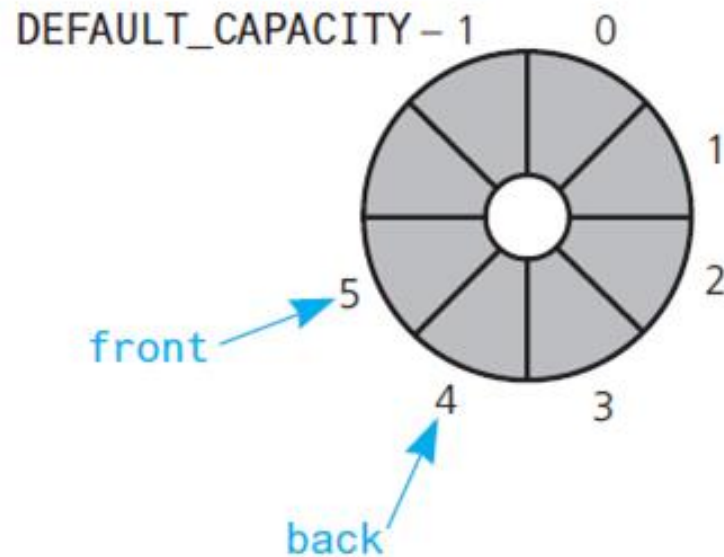
- The effect of three consecutive operations on the queue

(a) front passes back when the queue becomes empty

Queue with single item



dequeue ()—queue becomes empty

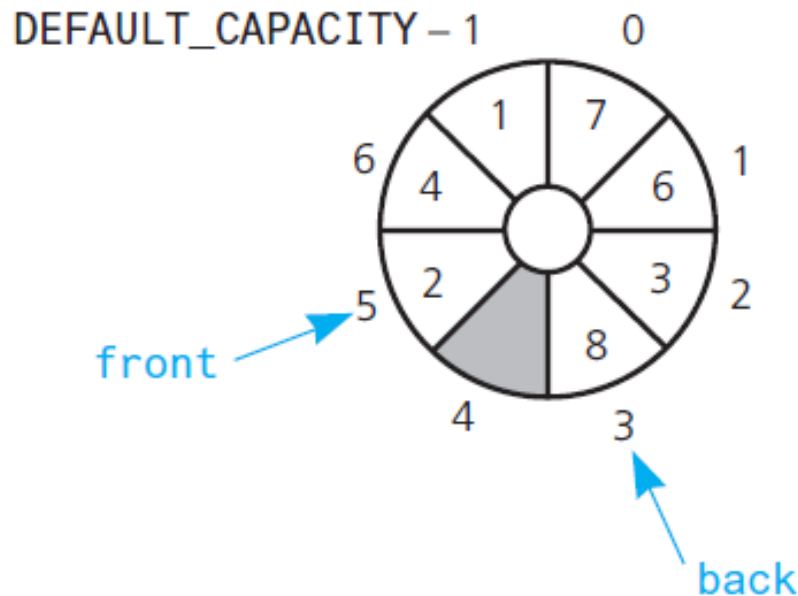


- **front** and **back** as the queue becomes empty and as it becomes full

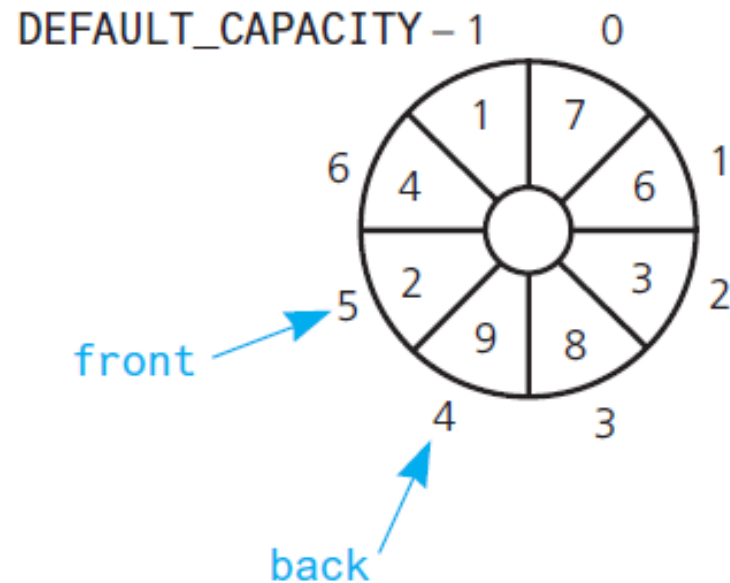
An Array-Based Implementation

(b) back catches up to front when the queue becomes full

Queue with single empty slot



enqueue (9)—queue becomes full



- **front** and **back** as the queue becomes empty and as it becomes full

```
1  /** ADT queue: Circular array-based implementation.
2   * @file ArrayQueue.h */
3  #ifndef ARRAY_QUEUE_
4  #define ARRAY_QUEUE_
5  #include "QueueInterface.h"
6  #include "PrecondViolatedExcept.h"
7
8  template<class ItemType>
9  class ArrayQueue : public QueueInterface<ItemType>
10 {
11 private:
12     static const int DEFAULT_CAPACITY = 50;
13     ItemType items[DEFAULT_CAPACITY]; // Array of queue items
14     int front;                        // Index to front of queue
15     int back;                         // Index to back of queue
16     int count;                       // Number of items currently in the queue
17 }
```

- The header file for the class **ArrayQueue**

```
1  /** ADT queue: Circular array-based implementation.
2   @file ArrayQueue.cpp */
3  #include "ArrayQueue.h" // Header file
4
5  template<class ItemType>
6  ArrayQueue<ItemType>::ArrayQueue()
7      : front(0), back(DEFAULT_CAPACITY - 1), count(0)
8  {
9  } // end default constructor
10
11 template<class ItemType>
12 bool ArrayQueue<ItemType>::isEmpty() const
13 {
14     return count == 0;
15 } // end isEmpty
16
```

- The implementation file for the class **ArrayQueue**


```
16
17 template<class ItemType>
18 bool ArrayQueue<ItemType>::enqueue(const ItemType& newEntry)
19 {
20     bool result = false;
21     if (count < DEFAULT_CAPACITY)
22     {
23         // Queue has room for another item
24         back = (back + 1) % DEFAULT_CAPACITY;
25         items[back] = newEntry;
26         count++;
27         result = true;
28     } // end if
29
30     return result;
31 } // end enqueue
32
```

- The implementation file for the class **ArrayQueue**

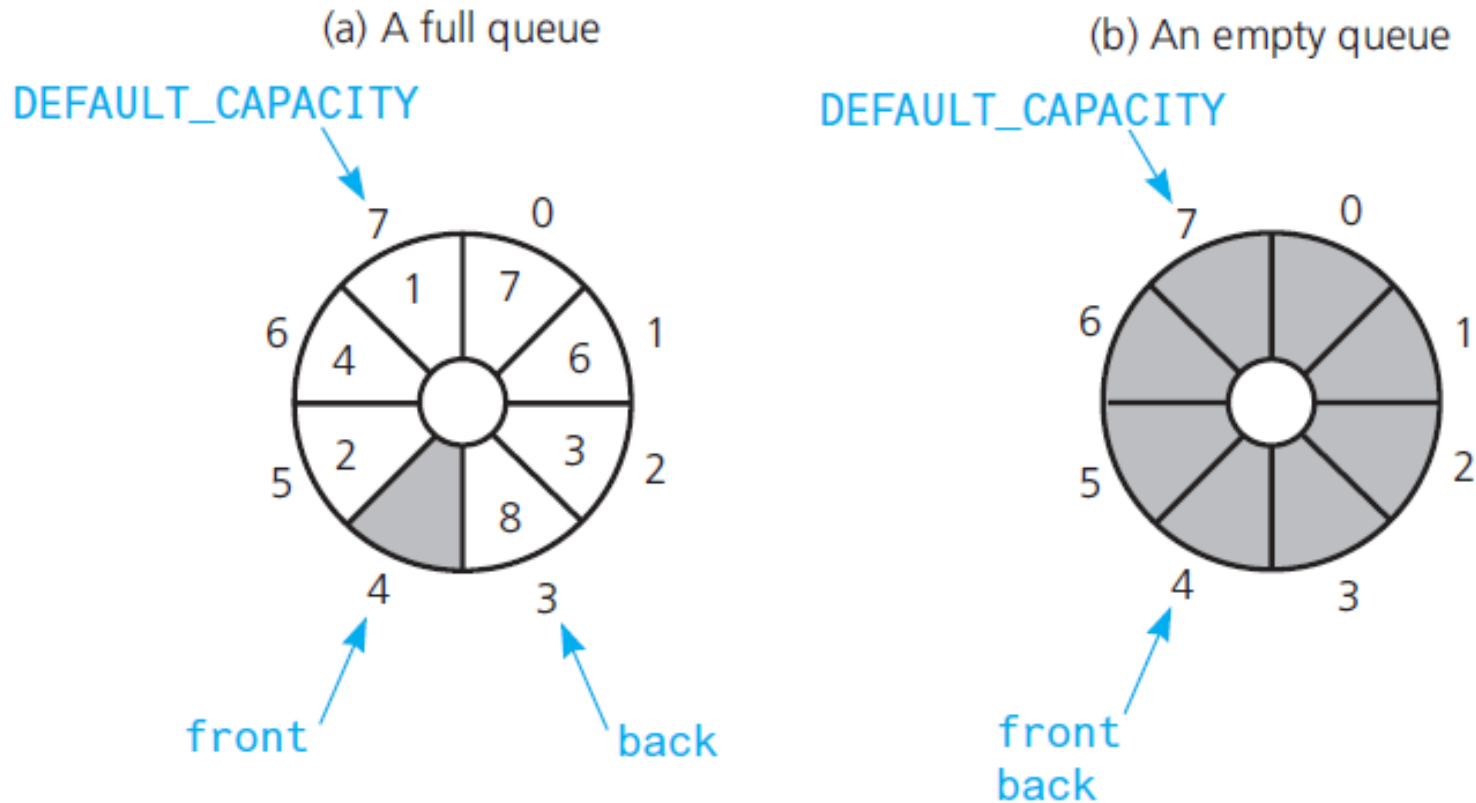
```
33  template<class ItemType>
34  bool ArrayQueue<ItemType>::dequeue()
35  {
36      bool result = false;
37      if (!isEmpty())
38      {
39          front = (front + 1) % DEFAULT_CAPACITY;
40          count--;
41          result = true;
42      } // end if
43
44      return result;
45  } // end dequeue
46
```

- The implementation file for the class `ArrayQueue`

```
46
47 template<class ItemType>
48 ItemType ArrayQueue<ItemType>::peekFront() const throw(PrecondViolatedExcept)
49 {
50     // Enforce precondition
51     if (isEmpty())
52         throw PrecondViolatedExcept("peekFront() called with empty queue");
53
54     // Queue is not empty; return front
55     return items[front];
56 } // end peekFront
57 // End of implementation file.
```

- The implementation file for the class **ArrayQueue**

An Array-Based Implementation



- 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)

- A header file for the class `SL_PriorityQueue`.

```
1  /** ADT priority queue: ADT sorted list implementation.
2   @file SL_PriorityQueue.h */
3  #ifndef PRIORITY_QUEUE_
4  #define PRIORITY_QUEUE_
5
6  #include "PriorityQueueInterface.h"
7  #include "LinkedSortedList.h"
8  #include "PrecondViolatedExcept.h"
9  #include <memory>
10
11  template<class ItemType>
12  class SL_PriorityQueue : public PriorityQueueInterface<ItemType>
13  {
14  private:
15      std::unique_ptr<LinkedSortedList<ItemType>> slistPtr; // Ptr to sorted list
16                                                         // of items
17
```

- A header file for the class `SL_PriorityQueue`.

```
16
17
18 public:
19     SL_PriorityQueue();
20     SL_PriorityQueue(const SL_PriorityQueue& pq);
21     ~SL_PriorityQueue();
22
23     bool isEmpty() const;
24     bool enqueue(const ItemType& newEntry);
25     bool dequeue();
26
27     /** @throw PrecondViolatedExcept if priority queue is empty. */
28     ItemType peekFront() const throw (PrecondViolatedExcept);
29 }; // end SL_PriorityQueue
30 #include "SL_PriorityQueue.cpp"
31 #endif
```


I HATE PROGRAMMING

I HATE PROGRAMMING

I HATE PROGRAMMING



IT WORKS!

I LOVE PROGRAMMING