CHAPTER 20

Linked Lists, Queues, and Priority Queues

Objectives

- To create nodes to store elements in a linked list (§20.2).
- To access the nodes in a linked list via pointers (§20.3).
- To define a `LinkedList` class for storing and processing data in a list (§20.4).
- To add an element to the head of a list (§20.4.1).
- To add an element to the end of a list (§20.4.2).
- To insert an element into a list (§20.4.3).
- To remove the first element from a list (§20.4.4).
- To remove the last element from a list (§20.4.5).
- To remove an element at a specified position in a list (§20.4.6).
- To implement iterators for traversing the elements in various types of containers (§20.5).
- To traverse the elements in a container using the foreach loop (§20.6).
- To explore the variations of linked lists (§20.7).
- To implement the `Queue` class using a linked list (§20.8).
- To implement the `PriorityQueue` class using a heap (§20.9).

20.1 Introduction

Key Point: *This chapter focuses on designing and implementing custom data structures.*

Section 12.4, "Class Templates," introduced a generic **Stack** class. The elements in the stack are stored in an array. The array size is fixed. If the array is too small, the elements cannot be stored in the stack; if it is too large, a lot of space will be wasted. A possible solution was proposed in Section 12.5, "Improving the **Stack** Class." Initially, the stack uses a small array. When there is no room to add a new element, the stack creates a new array that doubles the size of the old array, copies the contents from the old array to this new one, and discards the old array. It is time consuming to copy the array.

This chapter introduces a new data structure, called *linked list*. A linked list is efficient for storing and managing a varying number of elements. This chapter also discusses how to implement queues using linked lists.

20.2 Nodes

Key Point: *In a linked list, each element is contained in a structure called the node.*

When a new element is added, a node is created to contain it. All the nodes are chained through pointers, as shown in Figure 20.1.

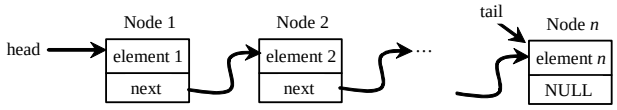


Figure 20.1

A linked list consists of any number of nodes chained together.

Nodes can be defined using a class. The class definition for a node can be as follows:

```
1  template<typename T>
2  class Node
```

```
 3  {
 4  public:
 5    T element;  // Element contained in the node
 6    Node* next; // Pointer to the next node
 7
 8    Node() // No-arg constructor
 9    {
10      next = NULL;
11    }
12
13    Node(T element) // Constructor
14    {
15      this->element = element;
16      next = NULL;
17    }
18  };
```

Node is defined as a template class with a type parameter **T** for specifying the element type.

By convention, pointer variables named **head** and **tail** are used to point to the first and the last node in the list. If the list is empty, both **head** and **tail** should be **NULL**. Recall that **NULL** is a C++ constant for **0**, which indicates that a pointer does not point to any node. The definition of **NULL** is in a number of standard libraries including **<iostream>** and **<cstddef>**. Here is an example that creates a linked list to hold three nodes. Each node stores a string element.

Step 1: Declare **head** and **tail**:

```
Node<string>* head = NULL;
Node<string>* tail = NULL;
```

The list is empty now.

head is NULL and tail is NULL

head and **tail** are both **NULL**. The list is empty.

Step 2: Create the first node and insert it to the list:

```
head = new Node<string>("Chicago");
tail = head;
```

After the first node is inserted

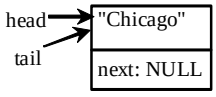


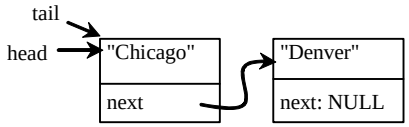
Figure 20.2

Append the first node to the list.

After the first node is inserted in the list, **`head`** and **`tail`** point to this node, as shown in Figure 20.2.

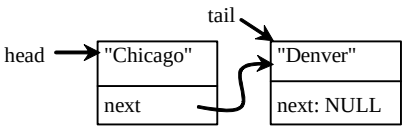
Step 3: Create the second node and append it to the list:

```
tail->next
  = new Node<string>("Denver");
```



(a)

```
tail = tail->next;
```



(b)

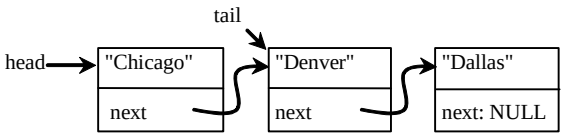
Figure 20.3

Append the second node to the list.

To append the second node to the list, link it with the first node, as shown in Figure 20.3a. The new node is now the tail node. So you should move tail to point to this new node, as shown in Figure 20.3b.

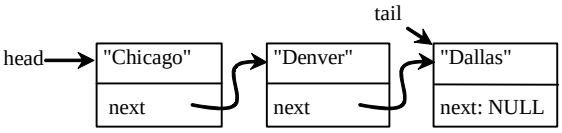
Step 4: Create the third node and append it to the list:

```
tail->next =
  new Node<string>("Dallas");
```



(a)

```
tail = tail->next;
```



(b)

Figure 20.4

Append the third node to the list.

To append the new node to the list, link it with the last node, as shown in Figure 20.4a. The new node is now the tail node. So you should move tail to point to this new node, as shown in Figure 20.4b.

Each node contains the element and a pointer that points to the next element. If the node is the last in the list, its pointer data field **next** contains the value **NULL**. You can use this property to detect the last node. For example, you may write the following loop to traverse all the nodes in the list.

```
1  Node<string>* current = head;
2  while (current != NULL)
3  {
4    cout << current->element << endl;
5    current = current->next;
6  }
```

The **current** pointer points to the first node in the list initially (line 1). In the loop, the element of the current node is retrieved (line 4), and then **current** points to the next node (line 5). The loop continues until the current node is **NULL**.

Pedagogical NOTE

Follow the link

www.cs.armstrong.edu/liang/animation/LinkedListAnimation.html to see how a linked list works, as shown in Figure 20.5.

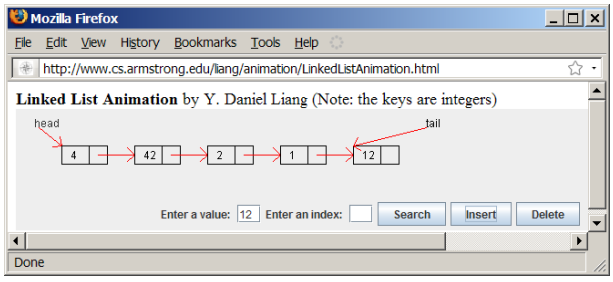


Figure 20.5

The animation tool enables you to see how a linked list works visually.

Check point

20.1

Are the following class declarations correct?

```
class A
{
public:
  A()
  {
  }

private:
  A *a;
  int i;
};
```

(a)

```
class A
{
public:
  A()
  {
  }

private:
  A a;
  int i;
};
```

(b)

20.2

What is `NULL` for?

20.3

When a node is created using the `Node` class, is the `next` pointer of this new node `NULL`?

20.3 The `LinkedList` Class

Linked list is a popular data structure for storing data in sequential order. For example, a list of students, a list of available rooms, a list of cities, and a list of books all can be stored using lists. The operations listed here are typical of most lists:

Retrieve an element from a list.

Insert a new element to a list.

Delete an element from a list.

Find how many elements are in a list.

Find whether an element is in a list.

Find whether a list is empty.

Figure 20.6 gives the class diagram for **LinkedList**. **LinkedList** is a class template with type parameter **T** that represents the type of the elements stored in the list.

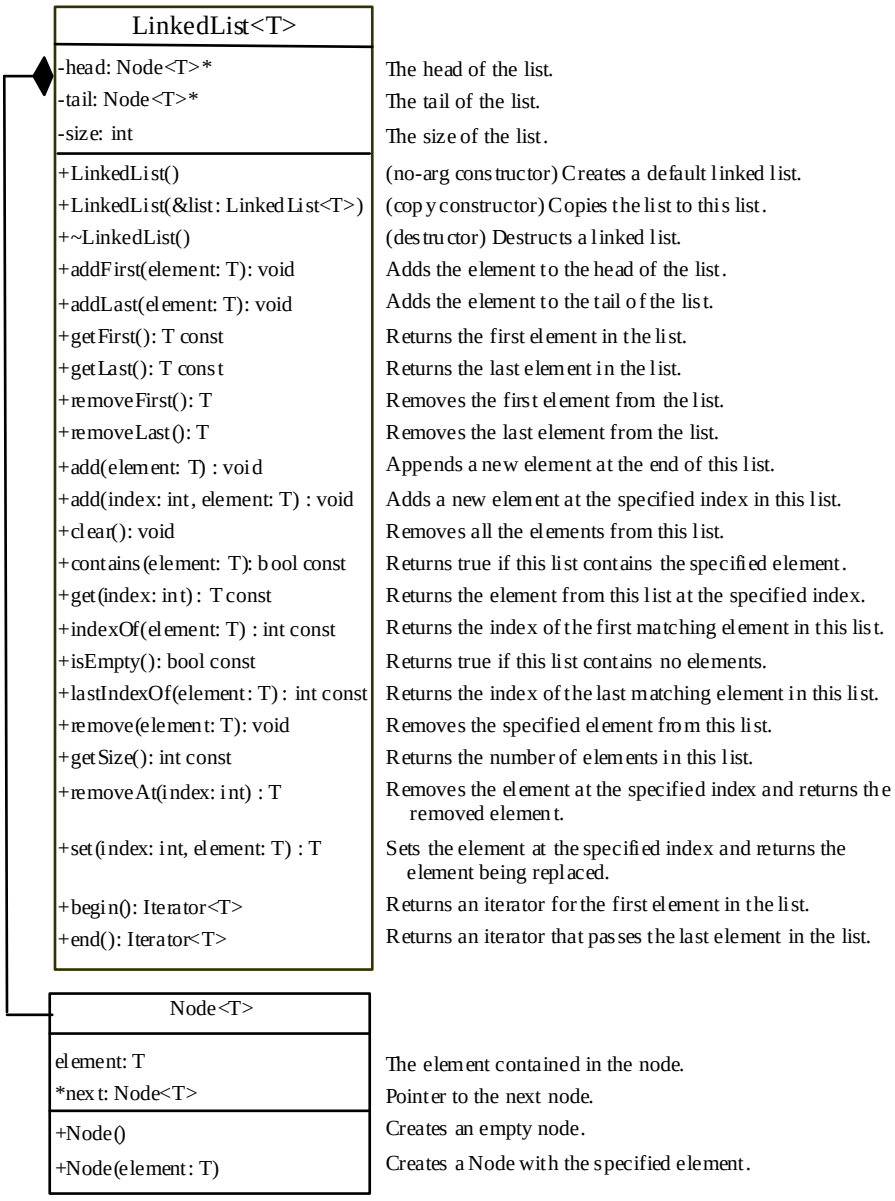


Figure 20.6

`LinkedList` *implements a list using a linked list of nodes.*

You can get an element from the list using **get(int index)**. The index is 0-based; i.e., the node at the head of the list has index **0**. Assume that the **LinkedList** class is available in the header file LinkedList.h. Let us begin by writing a test program that uses the **LinkedList** class, as shown in Listing 20.1. The program

creates a list using **LinkedList** (line 18). It uses the **add** function to add strings to the list and the **remove** function to remove strings.

Listing 20.1 TestLinkedList.cpp

```
 1  #include <iostream>
 2  #include <string>
 3  #include "LinkedList.h"
 4  using namespace std;
 5
 6  void printList(const LinkedList<string>& list)
 7  {
 8    for (int i = 0; i < list.getSize(); i++)
 9    {
10      cout << list.get(i) << " ";
11    }
12    cout << endl;
13  }
14
15  int main()
16  {
17    // Create a list for strings
18    LinkedList<string> list;
19
20    // Add elements to the list
21    list.add("America"); // Add it to the list
22    cout << "(1) ";
23    printList(list);
24
25    list.add(0, "Canada"); // Add it to the beginning of the list
26    cout << "(2) ";
27    printList(list);
28
29    list.add("Russia"); // Add it to the end of the list
30    cout << "(3) ";
31    printList(list);
32
33    list.add("France"); // Add it to the end of the list
34    cout << "(4) ";
35    printList(list);
36
37    list.add(2, "Germany"); // Add it to the list at index 2
38    cout << "(5) ";
39    printList(list);
40
41    list.add(5, "Norway"); // Add it to the list at index 5
42    cout << "(6) ";
43    printList(list);
44
45    list.add(0, "Netherlands"); // Same as
list.addFirst("Netherlands")
46    cout << "(7) ";
47    printList(list);
48
```

```
 49    // Remove elements from the list
 50    list.removeAt(0); // Same as list.remove("Netherlands ") in this
case
 51    cout << "(8) ";
 52    printList(list);
 53
 54    list.removeAt(2); // Remove the element at index 2
 55    cout << "(9) ";
 56    printList(list);
 57
 58    list.removeAt(list.getSize() - 1); // Remove the last element
 59    cout << "(10) ";
 60    printList(list);
 61
 62    return 0;
 63  }
```

Sample output

```
(1) America
(2) Canada America
(3) Canada America Russia
(4) Canada America Russia France
(5) Canada America Germany Russia France
(6) Canada America Germany Russia France Norway
(7) Netherlands Canada America Germany Russia France Norway
(8) Canada America Germany Russia France Norway
(9) Canada America Russia France Norway
(10) Canada America Russia France
```

Check point

20.4

Which of the following statements are used to insert a string `s` to the head of the list? Which ones are used to append a string `s` to the end of the list?

```
list.addFirst(s);
list.add(s);
list.add(0, s);
list.add(1, s);
```

20.5

Which of the following statements are used to remove the first element from the list? Which ones are used to remove the last element from the list?

```
list.removeFirst(s);
list.removeLast(s);
list.removeFirst();
list.removeLast();
list.remove(0);
list.removeAt(0);
```

```
list.removeAt(list.getSize() - 1);
list.removeAt(list.getSize());
```

20.6

Suppose the **removeAt** function is renamed as **remove** so that there are two overloaded functions, **remove(T element)** and **remove(int index)**. This is incorrect. Explain the reason.

20.4 Implementing LinkedList

Key Point: *A linked list is implemented using a linked structure.*

Now let us turn our attention to implementing the **LinkedList** class. Some functions are easy to implement. For example, the **isEmpty()** function simply returns **head == NULL**, and the **clear()** function simply destroys all nodes in the list and sets **head** and **tail** to **NULL**. The **addLast(T element)** function is same as the **add(T element)** function. The reason for defining both is convenience.

20.4.1 Implementing addFirst(T element)

The **addFirst(T element)** function can be implemented as follows:

```
1  template<typename T>
2  void LinkedList<T>::addFirst(T element)
3  {
4    Node<T>* newNode = new Node<T>(element);
5    newNode->next = head;
6    head = newNode;
7    size++;
8
9    if (tail == NULL)
10     tail = head;
11 }
```

The **addFirst(T element)** function creates a new node (line 4) to store the element and insert the node to the beginning of the list (line 5), as shown in Figure 20.7a. After the insertion, **head** should point to this new element node (line 6), as shown in Figure 20.7b.

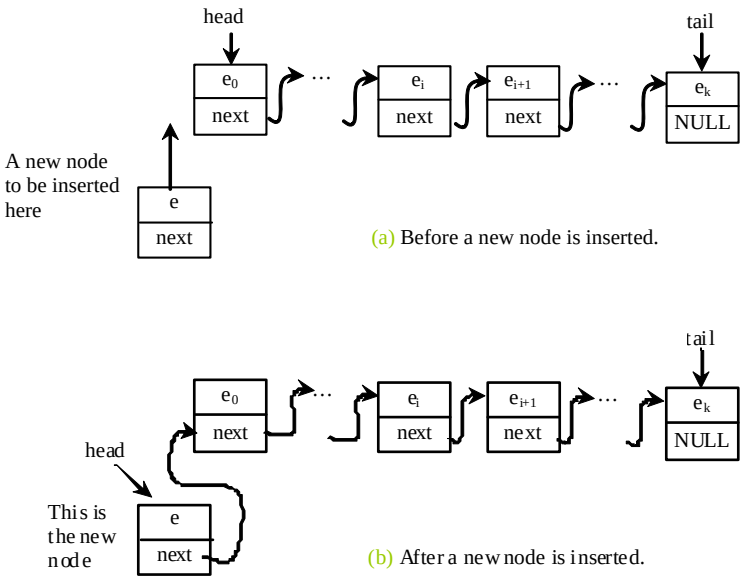


Figure 20.7

A new element is inserted to the beginning of the list.

If the list is empty (line 9), both **head** and **tail** will point to this new node (line 10). After the node is created, the size should be increased by **1** (line 7).

20.4.2 Implementing ***addLast(T element)***

The **addLast(T element)** function creates a node to hold an element and appends the node at the end of the list. It can be implemented as follows:

```
 1  template<typename T>
 2  void LinkedList<T>::addLast(T element)
 3  {
 4    if (tail == NULL)
 5    {
 6      head = tail = new Node<T>(element);
 7    }
 8    else
 9    {
10      tail->next = new Node<T>(element);
11      tail = tail->next;
12    }
13
14    size++;
15  }
```

Consider two cases:

(1) if the list is empty (line 4), both **head** and **tail** will point to this new node (line 6);

(2) otherwise, insert the node at the end of the list (line 10). After the insertion, **tail** should refer to this new element node (line 11), as shown in Figure 20.8. In any case, after the node is created, the size should be increased by **1** (line 14).

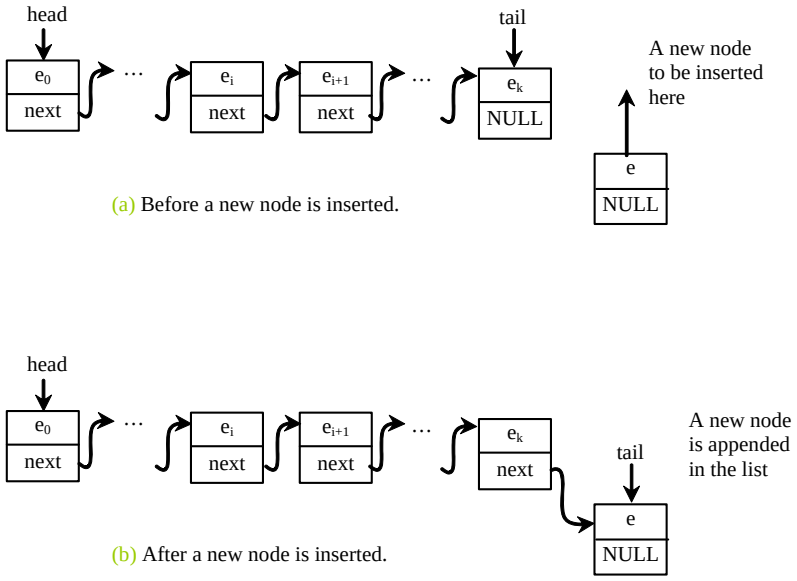


Figure 20.8

A new element is added at the end of the list.

20.4.3 Implementing ***`add(int index, T element)`***

The **add(int index, T element)** function adds an element to the list at the specified index. It can be implemented as follows:

```
1  template<typename T>
2  void LinkedList<T>::add(int index, T element)
3  {
4    if (index == 0)
5      addFirst(element);
6    else if (index >= size)
7      addLast(element);
```

```
 8    else
 9    {
10      Node<T>* current = head;
11      for (int i = 1; i < index; i++)
12        current = current->next;
13      Node<T>* temp = current->next;
14      current->next = new Node<T>(element);
15      (current->next)->next = temp;
16      size++;
17    }
18  }
```

Consider three cases:

(1) If **index** is **0**, invoke **addFirst(element)** (line 5) to insert the element at the beginning of the list;

(2) If **index** is greater than or equal to **size**, invoke **addLast(element)** (line 7) to insert the element at the end of the list;

(3) Otherwise, create a new node to store the new element and locate where to insert it. As shown in Figure 20.9a, the new node is to be inserted between the nodes **current** and **temp**. The function assigns the new node to **current->next** and assigns **temp** to the new node's **next**, as shown in Figure 20.9b. The size is now increased by **1** (line 16).

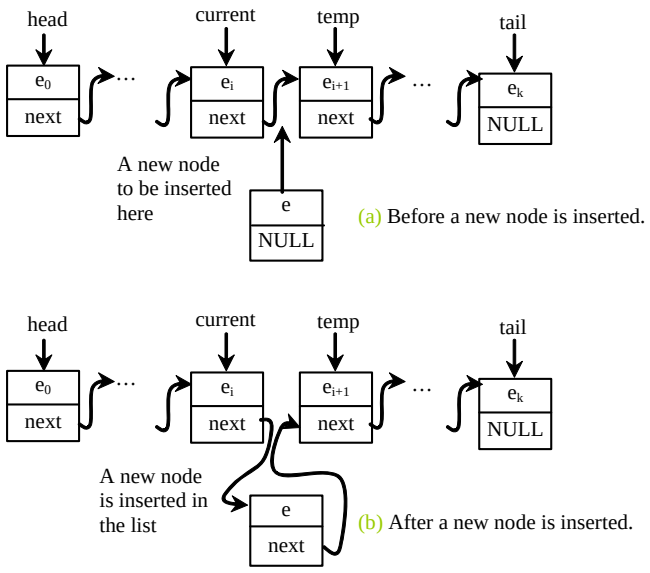


Figure 20.9

A new element is inserted in the middle of the list.

20.4.4 Implementing ***`removeFirst()`***

The **removeFirst()** function can be implemented as follows:

```
1  template<typename T>
2  T LinkedList<T>::removeFirst() throw (runtime_error)
3  {
4    if (size == 0)
5      throw runtime_error("No elements in the list");
6    else
7    {
8      Node<T>* temp = head;
9      head = head->next;
10     size--;
11     if (head == NULL) tail = NULL;
12     T element = temp->element;
13     delete temp;
14     return element;
15   }
16 }
```

Consider three cases:

(1) If the list is empty, an exception is thrown (line 5);

(2) Otherwise, remove the first node from the list by pointing **head** to the second node, as shown in Figure 20.10. The size is reduced by **1** after the deletion (line 10);

(3) If the list has just one node, then after removing it, **tail** should be set to **NULL** (line 11).

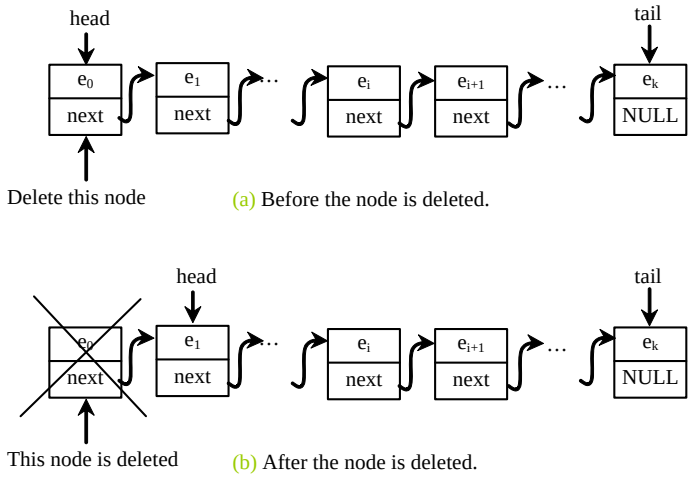


Figure 20.10

The first node is deleted from the list.

20.4.5 Implementing `removeLast()`

The **removeLast()** function can be implemented as follows:

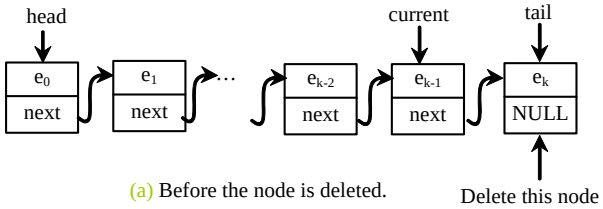
```
1  template<typename T>
2  T LinkedList<T>::removeLast() throw (runtime_error)
3  {
4    if (size == 0)
5      throw runtime_error("No elements in the list");
6    else if (size == 1)
7    {
8      Node<T>* temp = head;
9      head = tail = NULL;
10     size = 0;
11     T element = temp->element;
12     delete temp;
13     return element;
14   }
15   else
16   {
17     Node<T>* current = head;
18
19     for (int i = 0; i < size - 2; i++)
20       current = current->next;
21
22     Node<T>* temp = tail;
23     tail = current;
24     tail->next = NULL;
25     size--;
26     T element = temp->element;
27     delete temp;
28     return element;
29   }
30 }
```

Consider three cases:

(1) If the list is empty, an exception is thrown (line 5);

(2) If the list contains only one node, this node is destroyed; **head** and **tail** both become **NULL** (line 9);

(3) Otherwise, the last node is destroyed and the **tail** is repositioned to point to the second-to-last node, as shown in Figure 20.11. For the last two cases, the size is reduced by **1** after the deletion (lines 10, 25), and the element value of the deleted node is returned (lines 13, 28).



(a) Before the node is deleted.

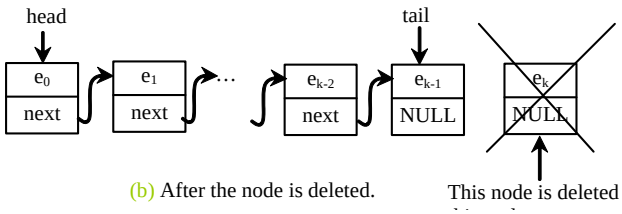


Figure 20.11

The last node is deleted from the list.

20.4.6 Implementing ***`removeAt(int index)`***

The **removeAt(int index)** function finds the node at the specified index and then removes it. It can be implemented as follows:

```
1  template<typename T>
2  T LinkedList<T>::removeAt(int index) throw (runtime_error)
3  {
4    if (index < 0 || index >= size)
5      throw runtime_error("Index out of range");
6    else if (index == 0)
7      return removeFirst();
8    else if (index == size - 1)
9      return removeLast();
10   else
11   {
12     Node<T>* previous = head;
13
14     for (int i = 1; i < index; i++)
15     {
16       previous = previous->next;
17     }
18
19     Node<T>* current = previous->next;
20     previous->next = current->next;
21     size--;
22     T element = current->element;
23     delete current;
24     return element;
25   }
26 }
```

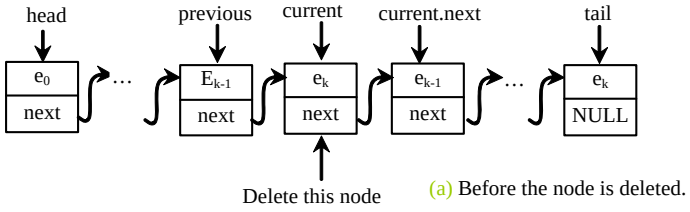
Consider four cases:

(1) If **index** is beyond the range of the list (i.e., **index < 0 || index >= size**), throw an exception (line 5);

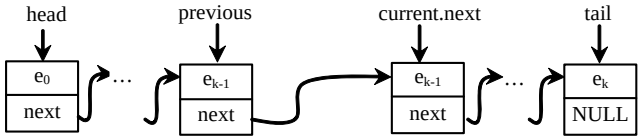
(2) If **index** is **0**, invoke **removeFirst()** to remove the first node (line 7);

(3) If **index** is **size - 1**, invoke **removeLast()** to remove the last node (line 9);

(4) Otherwise, locate the node at the specified **index**. Let **current** denote this node and **previous** denote the node before it, as shown in Figure 20.12a. Assign **current->next** to **previous->next** to eliminate the current node, as shown in Figure 20.12b.



(a) Before the node is deleted.



(b) After the node is deleted.

Figure 20.12

A node is deleted from the list.

20.4.7 The source code for ***LinkedList***

Listing 20.2 gives the implementation of **LinkedList**.

Listing 20.2 LinkedList.h

```
 1  #ifndef LINKEDLIST_H
 2  #define LINKEDLIST_H
 3  #include <stdexcept>
 4  using namespace std;
 5
 6  template<typename T>
 7  class Node
 8  {
 9  public:
10    T element;  // Element contained in the node
11    Node<T>* next; // Pointer to the next node
12
13    Node() // No-arg constructor
14    {
15      next = NULL;
```

```
16    }
17
18    Node(T element) // Constructor
19    {
20      this->element = element;
21      next = NULL;
22    }
23  };
24
25  template<typename T>
26  class Iterator: public std::iterator<std::forward_iterator_tag, T>
27  {
28  public:
29    Iterator(Node<T>* p)
30    {
31      current = p;
32    };
33
34    Iterator operator++() // Prefix ++
35    {
36      current = current->next;
37      return *this;
38    }
39
40    Iterator operator++(int dummy) // Postfix ++
41    {
42      Iterator temp(current);
43      current = current->next;
44      return temp;
45    }
46
47    T& operator*()
48    {
49      return current->element;
50    }
51
52    bool operator==(const Iterator<T>& iterator)
53    {
54      return current == iterator.current;
55    }
56
57    bool operator!=(const Iterator<T>& iterator)
58    {
59      return current != iterator.current;
60    }
61
62  private:
63    Node<T>* current;
64  };
65
66  template<typename T>
67  class LinkedList
68  {
69  public:
70    LinkedList();
71    LinkedList(const LinkedList<T>& list);
72    virtual ~LinkedList();
73    void addFirst(T element);
```

```
 74    void addLast(T element);
 75    T getFirst() const;
 76    T getLast() const;
 77    T removeFirst() throw (runtime_error);
 78    T removeLast();
 79    void add(T element);
 80    void add(int index, T element);
 81    void clear();
 82    bool contains(T element) const;
 83    T get(int index) const;
 84    int indexOf(T element) const;
 85    bool isEmpty() const;
 86    int lastIndexOf(T element) const;
 87    void remove(T element);
 88    int getSize() const;
 89    T removeAt(int index);
 90    T set(int index, T element);
 91
 92    Iterator<T> begin()
 93    {
 94      return Iterator<T>(head);
 95    };
 96
 97    Iterator<T> end()
 98    {
 99      return Iterator<T>(tail->next);
100    };
101
102  private:
103    Node<T>* head;
104    Node<T>* tail;
105    int size;
106  };
107
108  template<typename T>
109  LinkedList<T>::LinkedList()
110  {
111    head = tail = NULL;
112    size = 0;
113  }
114
115  template<typename T>
116  LinkedList<T>::LinkedList(const LinkedList<T>& list)
117  {
118    head = tail = NULL;
119    size = 0;
120
121    Node<T>* current = list.head;
122    while (current != NULL)
123    {
124      this->add(current->element);
125      current = current->next;
126    }
127  }
128
129  template<typename T>
130  LinkedList<T>::~LinkedList()
131  {
```

```
132   clear();
133 }
134
135 template<typename T>
136 void LinkedList<T>::addFirst(T element)
137 {
138   Node<T>* newNode = new Node<T>(element);
139   newNode->next = head;
140   head = newNode;
141   size++;
142
143   if (tail == NULL)
144     tail = head;
145 }
146
147 template<typename T>
148 void LinkedList<T>::addLast(T element)
149 {
150   if (tail == NULL)
151   {
152     head = tail = new Node<T>(element);
153   }
154   else
155   {
156     tail->next = new Node<T>(element);
157     tail = tail->next;
158   }
159
160   size++;
161 }
162
163 template<typename T>
164 T LinkedList<T>::getFirst() const
165 {
166   if (size == 0)
167     throw runtime_error("Index out of range");
168   else
169     return head->element;
170 }
171
172 template<typename T>
173 T LinkedList<T>::getLast() const
174 {
175   if (size == 0)
176     throw runtime_error("Index out of range");
177   else
178     return tail->element;
179 }
180
181 template<typename T>
182 T LinkedList<T>::removeFirst() throw (runtime_error)
183 {
184   if (size == 0)
185    throw runtime_error("No elements in the list");
186   else
187   {
188     Node<T>* temp = head;
189     head = head->next;
```

```
190       if (head == NULL) tail = NULL;
191       size--;
192       T element = temp->element;
193       delete temp;
194       return element;
195     }
196   }
197
198   template<typename T>
199   T LinkedList<T>::removeLast()
200   {
201     if (size == 0)
202       throw runtime_error("No elements in the list");
203     else if (size == 1)
204     {
205       Node<T>* temp = head;
206       head = tail = NULL;
207       size = 0;
208       T element = temp->element;
209       delete temp;
210       return element;
211     }
212     else
213     {
214       Node<T>* current = head;
215
216       for (int i = 0; i < size - 2; i++)
217         current = current->next;
218
219       Node<T>* temp = tail;
220       tail = current;
221       tail->next = NULL;
222       size--;
223       T element = temp->element;
224       delete temp;
225       return element;
226     }
227   }
228
229   template<typename T>
230   void LinkedList<T>::add(T element)
231   {
232     addLast(element);
233   }
234
235   template<typename T>
236   void LinkedList<T>::add(int index, T element)
237   {
238     if (index == 0)
239       addFirst(element);
240     else if (index >= size)
241       addLast(element);
242     else
243     {
244       Node<T>* current = head;
245       for (int i = 1; i < index; i++)
246         current = current->next;
247       Node<T>* temp = current->next;
```

```
248       current->next = new Node<T>(element);
249       (current->next)->next = temp;
250       size++;
251     }
252   }
253
254   template<typename T>
255   void LinkedList<T>::clear()
256   {
257     while (head != NULL)
258     {
259       Node<T>* temp = head;
260       head = head->next;
261       delete temp;
262     }
263
264     tail = NULL;
265     size = 0;
266   }
267
268   template<typename T>
269   T LinkedList<T>::get(int index) const
270   {
271     if (index < 0 || index > size - 1)
272       throw runtime_error("Index out of range");
273
274     Node<T>* current = head;
275     for (int i = 0; i < index; i++)
276       current = current->next;
277
278     return current->element;
279   }
280
281   template<typename T>
282   int LinkedList<T>::indexOf(T element) const
283   {
284     // Implement it in this exercise
285     Node<T>* current = head;
286     for (int i = 0; i < size; i++)
287     {
288       if (current->element == element)
289         return i;
290       current = current->next;
291     }
292
293     return -1;
294   }
295
296   template<typename T>
297   bool LinkedList<T>::isEmpty() const
298   {
299     return head == NULL;
300   }
301
302   template<typename T>
303   int LinkedList<T>::getSize() const
304   {
305     return size;
```

```
306  }
307
308  template<typename T>
309  T LinkedList<T>::removeAt(int index)
310  {
311    if (index < 0 || index >= size)
312      throw runtime_error("Index out of range");
313    else if (index == 0)
314      return removeFirst();
315    else if (index == size - 1)
316      return removeLast();
317    else
318    {
319      Node<T>* previous = head;
320
321      for (int i = 1; i < index; i++)
322      {
323        previous = previous->next;
324      }
325
326      Node<T>* current = previous->next;
327      previous->next = current->next;
328      size--;
329      T element = current->element;
330      delete current;
331      return element;
332    }
333  }
334
335    // The functions remove(T element), lastIndexOf(T element),
336    // contains(T element), and set(int index, T element) are
337    // left as an exercise
338
339  #endif
```

A linked list contains nodes defined in the **Node** class (lines 6–23). You can obtain iterators for traversing the elements in a linked list. The **Iterator** class (lines 25–64) will be discussed in Section 20.5.

The header of the **LinkedList** class is defined in lines 66–106. The no-arg constructor (lines 108–1113) constructs an empty linked list with **head** and **tail NULL** and **size 0**.

The copy constructor (lines 115–127) creates a new linked list by copying the contents from an existing list. This is done by inserting the elements from the existing linked list to the new one (lines 121–126).

The destructor (lines 129–133) removes all nodes from the linked list by invoking the **clear** function (lines 255–266), which deletes all the nodes from the list (line 261).

The implementation for functions **addFirst(T element)** (lines 135–145), **addLast(T element)** (lines 147–161), **removeFirst()** (lines 163–170), **removeLast()** (lines 198–227), **add(T element)** (lines 229–233), **add(int index, T element)** (lines 235–252), and **removeAt(int index)** (lines 308–333) is discussed in Sections 20.4.1–20.4.6.

The functions **getFirst()** and **getLast()** (lines 163–179) return the first and last elements in the list, respectively.

The **get(int index)** function returns the element at the specified index (lines 268–279).

The implementation of **lastIndexOf(T element)**, **remove(T element)**, **contains(T element)**, and **set(int index, Object o)** (lines 335–337) is omitted and left as an exercise.

20.4.8 The time complexity of `LinkedList`

Table 20.1 summarizes the complexities of the functions in **LinkedList**.

Table 20.1

Time Complexities for functions in `LinkedList`

Funcitons	LinkList
`add(e: T)`	$O(1)$
`add(index: int, e: T)`	$O(n)$
`clear()`	$O(n)$
`contains(e: T)`	$O(n)$
`get(index: int)`	$O(n)$
`indexOf(e: T)`	$O(n)$
`isEmpty()`	$O(1)$
`lastIndexOf(e: T)`	$O(n)$
`remove(e: T)`	$O(n)$
`size()`	$O(1)$
`remove(index: int)`	$O(n)$
`set(index: int, e: T)`	$O(n)$
`addFirst(e: T)`	$O(1)$
`removeFirst()`	$O(1)$

You can use an array, a vector, or a linked list to store elements. If you don't know the number of elements in advance, it is more efficient to use a vector or a linked list, because these can grow and shrink dynamically. If your application requires frequent insertion and deletion anywhere, it is more efficient to store elements using a linked list, because inserting an element into an array or a vector would require all the elements in the array after the insertion point to be moved. If the number of elements in an application is fixed and the application does not require random insertion and deletion, it is simple and efficient to use an array to store the elements.

Check point

20.7

If a linked list does not contain any nodes, what are the values in **`head`** and **`tail`**?

20.8

If a linked list has only one node, is **`head == tail`** true? List all cases in which **`head == tail`** is true.

20.9

When a new node is inserted to the head of a linked list, will the `head` pointer and the `tail` pointer be changed?

20.10

When a new node is inserted to the end of a linked list, will the `head` pointer and the `tail` pointer be changed?

20.11

When a node is removed from a linked list, what will happen if you don't explicitly use the `delete` operator to release the node?

20.12

Under what circumstances would the functions `removeFirst`, `removeLast`, and `removeAt` throw an exception?

20.13

Discuss the pros and cons of using arrays and linked lists.

20.14

If the number of elements in the program is fixed, what data structure should you use? If the number of elements in the program changes, what data structure should you use?

20.15

If you have to add or delete the elements anywhere in a list, should you use an array or a linked list?

20.16

If you change the function signature for `printList` in line 6 in Listing 20.1 to

```
void printList(const LinkedList<string> list)
```

will the program work? So, what is the difference between the two signatures?

20.17

What will happen when you run the following code?

```
#include <iostream>
#include <string>
#include "LinkedList.h"
using namespace std;

int main()
{
  LinkedList<string> list;
  list.add("abc");
  cout << list.removeLast() << endl;
  cout << list.removeLast() << endl;

  return 0;
}
```

20.18

Show the output of the following code:

```
#include <iostream>
#include <string>
#include "LinkedList.h"
using namespace std;

int main()
{
  LinkedList<string> list;
  list.add("abc");

  try
  {
    cout << list.removeLast() << endl;
    cout << list.removeLast() << endl;
  }
  catch (runtime_error& ex)
  {
    cout << "The list size is " <<  list.getSize() << endl;
  }

  return 0;
```

```
}
```

20.5 Iterators

Key Point: *Iterators are objects that provide a uniform way for traversing elements in various types of containers.*

Iterator is an important notion in C++. The Standard Template Library (STL) uses iterators to access the elements in the containers. The STL will be introduced in Chapters 22 and 23. The present section defines an iterartor class and creates an iterator object for traversing the elements in a linked list. The objectives here are twofold: (1) to look at an example of how to define an iterator class; (2) to become familiar with iterators and how to use them to traverse the elements in a container.

Iterators can be viewed as encapsulated pointers. In a linked list, you can use pointers to traverse the list. But iterators have more functions than pointers. Iterators are objects. Iterators contain functions for accessing and manipulating elements. An iterator typically contains the overloaded operators, as shown in Table 20.2.

Table 20.2

Typical Overloaded Operators in an Iterator

Operator	Description
++	Advances the iterator to the next element.
*	Accesses the element pointed by the iterator.
==	Tests whether two iterators point to the same element.
!=	Tests whether two iterators point to different elements.

The **Iterator** class for traversing the elements in a linked list is defined in Figure 20.13.

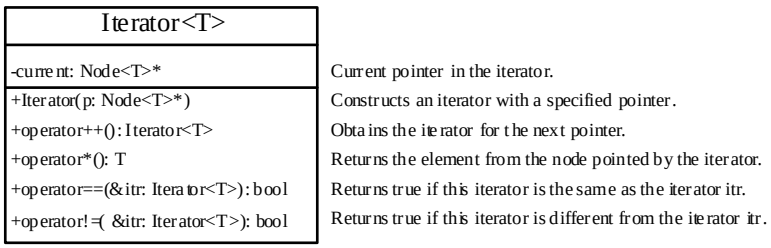


Figure 20.13

Iterator *encapsulates pointers with additional functions.*

This class is implemented in lines 25–64 in Listing 20.2. Since constructors and functions are short, they are implemented as inline functions. The **Iterator** class uses the data field **current** to point to the node being traversed (line 56). The constructor (lines 29–32) creates an iterator that points to a specified node. Both the prefix and postfix forms of the increment operators are implemented (lines 34-45) to move **current** to point to the next node in the list (lines 36, 43). Note that the postfix increment operator increments the iterator (line 43), but returns the original iterator (lines 42, 44). The * operator returns the element pointed at the iterator (line 49). The == operator tests whether the current iterator is the same as another iterator (line 54).

An iterator is used in conjunction with a container object that actually stores the elements. The container class should provide the **begin()** and **end()** functions for returning iterators, as shown in Table 20.3.

Table 20.3

Common Functions for Returning Iterators

Function	Description
`begin()`	Returns an iterator that points to the first element in the container.
`end()`	Returns an iterator that represents a position past the last element in the container. This iterator can be used to test whether all elements in the container have been traversed.

To obtain iterators from a **LinkedList**, the following two functions are defined and implemented in lines 85–93 in Listing 20.2.

```
Iterator<T> begin() const;
Iterator<T> end() const;
```

The **begin()** function returns the iterator for the first element in the list, and the **end()** function returns the iterator that represents a position past the last element in the list.

Listing 20.3 gives an example that uses iterators to traverse the elements in a linked list and displays the strings in uppercase. The program creates a **LinkedList** for strings in line 17, adds four strings to the list (lines 20–23), and traverses all the elements in the list using iterators and displays them in uppercase (lines 26–30).

Listing 20.3 TestIterator.cpp

```
 1  #include <iostream>
 2  #include <string>
 3  #include "LinkedList.h"
 4  using namespace std;
 5
 6  string toUpperCase(string& s)
 7  {
 8    for (int i = 0; i < s.length(); i++)
 9      s[i] = toupper(s[i]);
10
11    return s;
12  }
13
14  int main()
15  {
16    // Create a list for strings
17    LinkedList<string> list;
18
19    // Add elements to the list
20    list.add("America");
21    list.add("Canada");
22    list.add("Russia");
23    list.add("France");
24
25    // Traverse a list using iterators
26    for (Iterator<string> iterator = list.begin();
27      iterator != list.end(); iterator++)
28    {
29      cout << toUpperCase(*iterator) << " ";
30    }
31
32    return 0;
33  }
```

Sample output

```
AMERICA CANADA RUSSIA FRANCE
```

NOTE

An iterator functions like a pointer. It may be implemented using pointers, array indexes, or other data structures. The abstraction of iterators spares you the details of

implementation. Chapter 22, "STL Containers," will introduce iterators in STL. STL Iterators provide a uniform interface for accessing the elements in a container, so you can use iterators to access the elements in a vector or a set just as in a linked list.

Iterator can be used to traverse in a container efficiently. Note that the `printList` function takes $O(n^2)$ time since the `get(index)` function takes O(n) time to obtain an element in the ith index in the list. So, you should rewrite the printList function using iterators to improve its efficiency.

```
void printList(const LinkedList<string>& list)
{
  Iterator<string> current = list.begin();

  while (current != list.end())
  {
    cout << *current << " ";
    ++current;
  }

  cout << endl;
}
```

Note that the `Iterator` class is derived from `std::iterator<std::forward_iterator_tag, T>`. It is not necessary to make `Iterator` a child class of `std::iterator<std::forward_iterator_tag, T>`, but doing so enables you to invoke the C++ STL library functions for `LinkedList` elements. These functions use iterators to traverse elements in the container. Listing 20.4 gives an example that uses the C++ STL library functions `max_element` and `min_element` to return the maximum and minimum elements in a linked list.

Listing 20.4 TestSTLAlgorithm.cpp

```
 1  #include <iostream>
 2  #include <algorithm>
 3  #include <string>
 4  #include "LinkedList.h"
 5  using namespace std;
 6
 7  int main()
 8  {
 9    // Create a list for strings
10    LinkedList<string> list;
11
```

```
12    // Add elements to the list
13    list.add("America");
14    list.add("Canada");
15    list.add("Russia");
16    list.add("France");
17
18    cout << "The max element in the list is: " <<
19      *max_element(list.begin(), list.end()) << endl;
20
21    cout << "The min element in array1: " <<
22      *min_element(list.begin(), list.end()) << endl;
23
24    return 0;
25  }
```

Sample output

```
The max element in the list is: Russia
The min element in array1: America
```

The **max_element** and **min_element** functions were introduced in Section 11.7, "Useful Array Functions." The functions take points in the parameters. Iterators are like pointers. You can call an iterator just as a pointer for convenience. The **max_element(iterator1, iterator2)** returns the iterator for the maximum element between **iterator1** and **iterator2 - 1**.

20.6 C+11 Foreach Loop

Key Point: *You can use a foreach loop to traverse the elements in a collection.*

The foreach loop is a common computer language feature for traversing elements in a collection. This feature is supported in C++11 to traverse the elements in a collection sequentially. The collection may be an array or any container object with an iterator. For example, the following code displays all the elements in the array **myList**:

```
double myList[] = {3.3, 4.5, 1};
for (double& e: myList)
{
  cout << e << endl;
}
```

You can read the code as "for each element **e** in **myList**, do the following." Note that the variable, **e**, must be declared as the same type as the elements in **myList**.

In general, the syntax for a foreach loop is

```
for (elementType& element: collection)
{
  // Process the element
}
```

In C++, a vector is a collection with an iterator. So, you can traverse all the elements in a vector using a foreach loop. For example, the following code traverses all the elements in a vector of strings.

```
#include <vector>
#include <string>
vector<string> names;
names.push_back("Atlanta");
names.push_back("New York");
names.push_back("Kansas");
for (string& s: names)
{
  cout << s << endl;
}
```

Since list is a collection with an iterator, you can rewrite Listing 20.3 TestIterator.cpp using a foreach loop as shown in Listing 20.5.

Listing 20.5 TestForeachLoop.cpp

```
 1  #include <iostream>
 2  #include <string>
 3  #include "LinkedList.h"
 4  using namespace std;
 5
 6  string toUpperCase(strin& s)
 7  {
 8    for (int i = 0; i < s.length(); i++)
 9      s[i] = toupper(s[i]);
10
11    return s;
12  }
13
14  int main()
15  {
16    // Create a list for strings
17    LinkedList<string> list;
18
19    // Add elements to the list
```

```
20    list.add("America");
21    list.add("Canada");
22    list.add("Russia");
23    list.add("France");
24
25    // Traverse a list using a foreach loop
26    for (string& s: list)
27    {
28      cout << toUpperCase(s) << " ";
29    }
30
31    return 0;
32  }
```

The program creates a list (line 17) and adds four strings into the list (lines 20-23). A foreach loop is used to traverse the strings in the list (lines 26-29).

Note that the foreach loop is supported in the new compilers such as Visual C++ 2012. Old C++ compilers may not support the foreach loop.

Check point

20.19 Show the printout of the following code:

```
#include <iostream>
using namespace std;

int main()
{
  int list[] = {5, 9};

  for (int i: list)
    i++;

  for (int i: list)
    cout << i << endl;

  return 0;
}
```

20.20 Show the printout of the following code:

```
#include <iostream>
using namespace std;

int main()
{
  int list[] = {5, 9};

  for (int& i: list)
      i++;

  for (int i: list)
    cout << i << endl;
```

```
  return 0;
}
```
<end check point>

20.7 Variations of Linked Lists

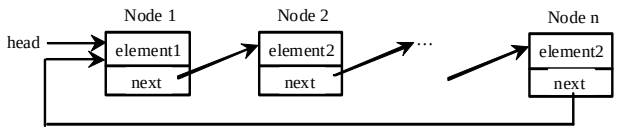
Key Point: Various types of linked lists can be used to organize data for certain applications.

The linked list introduced in the preceding section is known as a *singly linked list*. It contains a pointer to the list's first node, and each node contains a pointer to the next node sequentially. Several variations of the linked list are useful in certain applications.

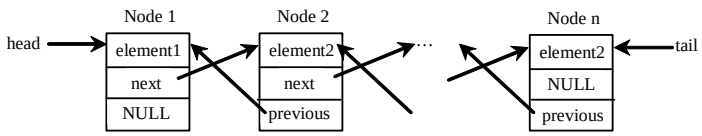
A *circular, singly linked list* differs in that the pointer of the last node points back to the first node, as shown in Figure 20.14a. Note that **tail** is not needed for circular linked lists. A good application of a circular linked list is in the operating system that serves multiple users in a timesharing fashion. The system picks a user from a circular list and grants a small amount of CPU time to the user and then move on to the next user in the list.

A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other to the previous node, as shown in Figure 20.14b. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. So, a doubly linked list can be traversed forward and backward.

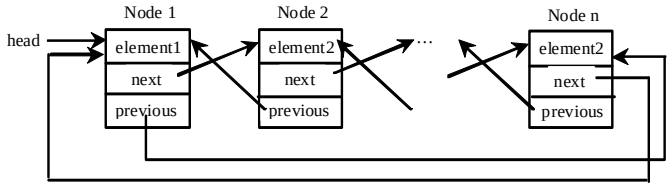
A *circular, doubly linked list* has the property that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node, as shown in Figure 20.14b.



(a) Circular linked list



(b) Doubly linked list



(c) Circular doubly linked list

Figure 20.14

Linked lists may appear in various forms.

The implementation of these linked lists is left to the exercises.

Check point

20.21

What is a circular, singly linked list? What is a doubly linked list? What is a circular, doubly linked list?

20.8 Queues

Key Point: A queue is a first-in and first-out data structure.

A queue represents a waiting list. It can be viewed as a special type of list whose elements are inserted into the end (tail) and are accessed and deleted from the beginning (head), as shown in Figure 20.15.

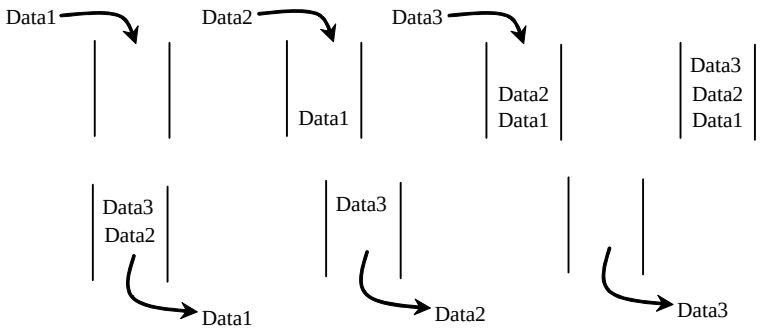


Figure 20.15

A queue holds objects in a first-in, first-out fashion.

Pedagogical NOTE

Follow the link

`www.cs.armstrong.edu/liang/animation/QueueAnimation.html`

to see how a queue works, as shown in Figure 20.16.

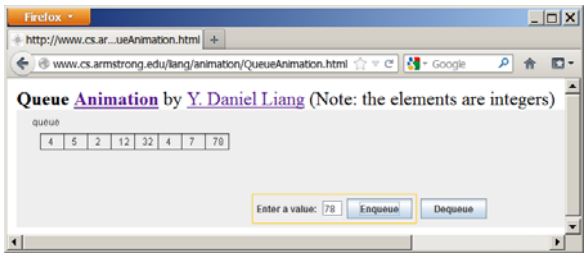


Figure 20.16

The animation tool enables you to see how a queue works visually.

There are two ways to design the queue class:

. Using composition: You can declare a linked list as a data field in the queue class, as shown in Figure 20.17a.

. Using inheritance: You can define a queue class by extending the linked list class, as shown in Figure 20.17b.

(a) Using composition (b) Using inheritance

Figure 20.17

Queue *may be implemented using composition or inheritance.*

Both designs are fine, but using composition is better, because it enables you to define a completely new queue class without inheriting the unnecessary and inappropriate functions from the linked list. Figure 20.18 shows the UML class diagram for the queue. Its implementation is shown in Listing 20.6.

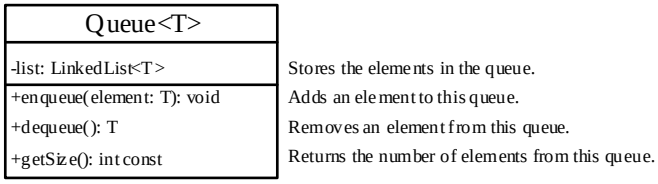


Figure 20.18

Queue *uses a linked list to provide a first-in, first-out data structure.*

Listing 20.6 Queue.h

```
 1  #ifndef QUEUE_H
 2  #define QUEUE_H
 3  #include "LinkedList.h"
 4  #include <stdexcept>
 5  using namespace std;
 6
 7  template<typename T>
 8  class Queue
 9  {
10  public:
11    Queue();
12    void enqueue(T element);
13    T dequeue() throw (runtime_error);
14    int getSize() const;
15
16  private:
17    LinkedList<T> list;
18  };
19
20  template<typename T>
21  Queue<T>::Queue()
22  {
23  }
24
25  template<typename T>
26  void Queue<T>::enqueue(T element)
27  {
28    list.addLast(element);
29  }
30
31  template<typename T>
32  T Queue<T>::dequeue() throw (runtime_error)
33  {
34    return list.removeFirst();
35  }
36
37  template<typename T>
38  int Queue<T>::getSize() const
39  {
40    return list.getSize();
41  }
```

```
42
43  #endif
```

A linked list is created to store the elements in a queue (line 17). The **enqueue(T element)** function (lines 25–29) adds elements into the tail of the queue. The **dequeue()** function (lines 31–35) removes an element from the head of the queue and returns the removed element. The **getSize()** function (lines 37–41) returns the number of elements in the queue.

Listing 20.7 gives an example that creates a queue for **int** values (line 17) and a queue for strings (line 24) using the **Queue** class. It uses the **enqueue** function to add elements to the queues (lines 19, 25–27) and the **dequeue** function to remove int values and strings from the queue.

Listing 20.7 TestQueue.cpp

```
 1  #include <iostream>
 2  #include "Queue.h"
 3  #include <string>
 4  using namespace std;
 5
 6  template<typename T>
 7  void printQueue(Queue<T>& queue)
 8  {
 9    while (queue.getSize() > 0)
10      cout << queue.dequeue() << " ";
11    cout << endl;
12  }
13
14  int main()
15  {
16    // Queue of int values
17    Queue<int> intQueue;
18    for (int i = 0; i < 10; i++)
19      intQueue.enqueue(i);
20
21    printQueue(intQueue);
22
23    // Queue of strings
24    Queue<string> stringQueue;
25    stringQueue.enqueue("New York");
26    stringQueue.enqueue("Boston");
27    stringQueue.enqueue("Denver");
28
29    printQueue(stringQueue);
30
31    return 0;
32  }
```

Sample output

```
0 1 2 3 4 5 6 7 8 9
New York Boston Denver
```

Check point

20.22

You can use inheritance or composition to design the data structures for queues. Discuss the pros and cons of these two approaches.

20.23

Show the output of the following code:

```
#include <iostream>
#include <string>
#include "ImprovedStack.h" // Defined in Listing 12.7
#include "Queue.h"
using namespace std;

int main()
{
  Stack<string> stack;
  Queue<int> queue;

  stack.push("Georgia");
  stack.push("Indiana");
  stack.push("Oklahoma");

  cout << stack.pop() << endl;
  cout << "Stack's size is " << stack.getSize() << endl;

  queue.enqueue(1);
  queue.enqueue(2);
  queue.enqueue(3);

  cout << queue.dequeue() << endl;
  cout << "Queue's size is " << queue.getSize() << endl;

  return 0;
}
```

20.9 Priority Queues

Key Point: Elements in a priority queue are assigned with a priority. The elements with the highest priority is removed first from a priority queue.

A regular queue is a first-in first-out data structure. Elements are appended to the end of the queue and are removed from the beginning. In a priority queue, elements are assigned with priorities. The element with the highest priority is accessed or removed first. A priority queue has a largest-in, first-out behavior. For example, the emergency room in a hospital assigns patients with priority numbers; the patient with the highest priority is treated first.

A priority queue can be implemented using a heap, where the root is the element with the highest priority in the queue. Heap was introduced in Section 19.5, “Heap Sort.” The class diagram for the priority queue is shown in Figure 20.19. Its implementation is given in Listing 20.8.

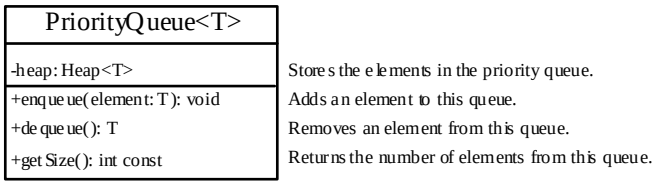


Figure 20.19

PriorityQueue *uses a heap to provide a largest-in, first-out data structure.*

Listing 20.8 PriorityQueue.h

```
 1  #ifndef PRIORITYQUEUE_H
 2  #define PRIORITYQUEUE_H
 3  #include "Heap.h"
 4
 5  template<typename T>
 6  class PriorityQueue
 7  {
 8  public:
 9    PriorityQueue();
10    void enqueue(T element);
11    T dequeue() throw (runtime_error);
12    int getSize() const;
13
14  private:
15    Heap<T> heap;
16  };
17
18  template<typename T>
19  PriorityQueue<T>::PriorityQueue()
20  {
21  }
22
```

```
23  template<typename T>
24  void PriorityQueue<T>::enqueue(T element)
25  {
26    heap.add(element);
27  }
28
29  template<typename T>
30  T PriorityQueue<T>::dequeue() throw (runtime_error)
31  {
32    return heap.remove();
33  }
34
35  template<typename T>
36  int PriorityQueue<T>::getSize() const
37  {
38    return heap.getSize();
39  }
40
41  #endif
```

Listing 20.9 gives an example of using a priority queue for patients. The **Patient** class is defined in lines 5–37. Line 42 creates a priority queue. Four patients with associated priority values are created and enqueued in lines 43–46. Line 50 dequeues a patient from the queue.

Listing 20.9 TestPriorityQueue.cpp

```
 1  #include <iostream>
 2  #include "PriorityQueue.h"
 3  #include <string>
 4  using namespace std;
 5
 6  class Patient
 7  {
 8  public:
 9    Patient(const string& name, int priority)
10    {
11      this->name = name;
12      this->priority = priority;
13    }
14
15    bool operator<(const Patient& secondPatient)
16    {
17      return (this->priority < secondPatient.priority);
18    }
19
20    bool operator>(const Patient& secondPatient)
21    {
22      return (this->priority > secondPatient.priority);
23    }
24
25    string getName()
26    {
27      return name;
```

```
28    }
29
30    int getPriority()
31    {
32      return priority;
33    }
34
35  private:
36    string name;
37    int priority;
38  };
39
40  int main()
41  {
42    // Queue of patients
43    PriorityQueue<Patient> patientQueue;
44    patientQueue.enqueue(Patient("John", 2));
45    patientQueue.enqueue(Patient("Jim", 1));
46    patientQueue.enqueue(Patient("Tim", 5));
47    patientQueue.enqueue(Patient("Cindy", 7));
48
49    while (patientQueue.getSize() > 0)
50    {
51      Patient element = patientQueue.dequeue();
52      cout << element.getName() << " (priority: " <<
53        element.getPriority() << ") ";
54    }
55
56    return 0;
57  }
```

Sample output

Cindy(priority: 7) Tim(priority: 5) John(priority: 2) Jim(priority: 1)

The `<` and `>` operators are defined the `Patient` class, so two patients can be compared. You can use any class type for the elements in the heap, provided that the elements can be compared using the `<` and `>` operators.

Check point

20.24

What is a priority queue? How is a priority queue implemented?

Key Terms

- auto type inference xx

- circular doubly linked list 47
- circular singly linked list 47
- dequeue 47
- doubly linked list 47
- enqueue 47
- linked list 47

priority queue

- queue 91
- singly linked list 47

Chapter Summary

1. A linked list grows and shrinks dynamically. Nodes in a linked list are dynamically created using the `new` operator, and they are destroyed using the `delete` operator.
2. A queue represents a waiting list. It can be viewed as a special type of list whose elements are inserted into the end (tail) and are accessed and deleted from the beginning (head).
3. If you don't know the number of elements in advance, it is more efficient to use a linked list, which can grow and shrink dynamically.
4. If your application requires frequent insertion and deletion anywhere, it is more efficient to store elements using a linked list, because inserting an element into an array would require moving all the elements in the array after the insertion point.
5. If the elements need to be processed in a first-in, first-out fashion, use a queue to store the elements.

A priority queue can be implemented using a heap, where the root is the element with the highest priority in the queue.

Quiz

Answer the quiz for this chapter online at www.cs.armstrong.edu/liang/cpp3e/quiz.html.

Programming Exercises

Sections 20.2-20.4

*20.1 (*Implement* **`remove(T element)`**) The implementation of **`remove(T element)`** is omitted in Listing 20.2, LinkedList.h. Implement it.

*20.2 (*Implement* **`lastIndexOf(T element)`**) The implementation of **`lastIndexOf(T element)`** is omitted in Listing 20.2, LinkedList.h. Implement it.

*20.3 (*Implement* **`contains(T element)`**) The implementation of **`contains(T element)`** is omitted in Listing 20.2, LinkedList.h. Implement it.

*20.4 (*Implement* **`set(int index, T element)`**) The implementation of **`contains(T element)`** is omitted in Listing 20.2, LinkedList.h. Implement it.

*20.5 (*Implement* **`reverse()`** function) Add a new member function named **`reverse()`** in the **`LinkedList`** class that reverses the nodes in the list.

*20.6 (*Implement* **`sort()`** function) Add a new member function named **`sort()`** in the **`LinkedList`** class that rearrange the list so that the elements in the nodes are sorted.

20.7 (*Adding set-like operations in* ***`LinkedList`***) Add and implement the following functions in **`LinkedList`**:

```
// Add the elements in otherList to this list.
void addAll(LinkedList<T>& otherList)

// Remove all the elements in otherList from this list
void removeAll(LinkedList<T>& otherList)

// Retain the elements in this list if they are also in otherList
void retainAll(LinkedList<T>& otherList)
```

Add three function operators: **+**, **-**, and **^** for set union, difference, and intersection. Overload the **=** operator to perform a deep copy of a list. Add the **[]** operator for accessing/modifying an element.

Use the following code to test your program:

```
#include <iostream>
#include <string>
#include "LinkedList.h"
using namespace std;

template<typename T>
void printList(const LinkedList<T>& list)
{
  Iterator<T> current = list.begin();

  while (current != list.end())
  {
    cout << *current << " ";
    current++;
  }

  cout << endl;
}

int main()
{
  // Create a list for strings
  LinkedList<string> list;
  list.add("Beijing");
  list.add("Tokyo");
  list.add("New York");
  list.add("London");
  list.add("Paris");

  // Create a list for strings
  LinkedList<string> list2;
  list2.add("Beijing");
  list2.add("Shanghai");
  list2.add("Paris");
  list2.add("Berlin");
  list2.add("Rome");

  LinkedList<string> list1(list);
  cout << "list1: ";
  printList(list1);
  cout << "list2: ";
  printList(list2);

  list1.addAll(list2);
  cout << "list is : ";
  printList(list);
  cout << "After list1.addAll(list2), list1 is ";
  printList(list1);
```

```
  list1 = list;
  cout << "list1: ";
  printList(list1);
  cout << "list2: ";
  printList(list2);
  list1.removeAll(list2);
  cout << "After list1.removeAll(list2), list1 is ";
  printList(list1);

  list1 = list;
  cout << "list1: ";
  printList(list1);
  cout << "list2: ";
  printList(list2);
  list1.retainAll(list2);
  cout << "After list1.retainAll(list2), list1 is ";
  printList(list1);

  list1 = list;
  cout << "list1: ";
  printList(list1);
  cout << "list2: ";
  printList(list2);
  list1 = list1 + list2;
  cout << "After list1 = list1 + list2, list1 is ";
  printList(list1);

  list1 = list;
  cout << "list1: ";
  printList(list1);
  cout << "list2: ";
  printList(list2);
  list1 = list1 - list2;
  cout << "After list1 = list1 - list2, list1 is ";
  printList(list1);

  list1 = list;
  cout << "list1: ";
  printList(list1);
  cout << "list2: ";
  list1 = list1 ^ list2;
  cout << "After list1 = list1 ^ list2, list1 is ";
  printList(list1);

  list1 = list;
  cout << list1[0] << endl;
  list1[0] = "Paris";
  cout << list1[0] << endl;

  return 0;
}
```

Section 20.6

*20.8 (*Create a doubly linked list*) The **`LinkedList`** class in the text is a singly linked list that enables one-way traversal of the list. Modify the **`Node`** class to add the new field name **`previous`** to refer to the previous node in the list, as follows:

```
template<typename T>
class Node
{
public:
  T element;  // Element contained in the node
  Node<T>* previous; // Pointer to the previous node
  Node<T>* next; // Pointer to the next node

  Node() // No-arg constructor
  {
    previous = NULL;
    next = NULL;
  }

  Node(T element) // Constructor
  {
    this->element = element;
    previous = NULL;
    next = NULL;
  }
};
```

Simplify the implementation of the **`add(T element, int index)`** and **`removeAt(int index)`** functions to take advantage of the doubly linked list. Also modify the **`Iterator`** class to implement the decrement operarator **`--`** that move the iterator to point to the previous element.

Sections 20.7-20.8

20.9 (*Implement **`Stack`** using inheritance*) In Listing 12.7, ImprovedStack.h, **`Stack`** is implemented using composition. Create a new stack class that extends **`LinkedList`**.

20.10 (*Implement **`Queue`** using inheritance*) In Listing 20.6 Queue.h, **`Queue`** is implemented using composition. Create a new queue class that extends **`LinkedList`**.