CHAPTER 13

LISTS, STACKS, AND QUEUES

CHAPTER GOALS

To become familiar with the list, stack, and queue data types

To understand the implementation of linked lists

To understand the efficiency of vector and list operations

CHAPTER CONTENTS

In this chapter, we introduce a new data structure, the *linked list*. A linked list is made up of nodes, each of which is connected to the neighboring nodes. You will learn how to use lists and the related stack and queue types. You will study the implementation of linked lists and analyze when linked lists are more efficient than arrays or vectors.

13.1 Using Linked Lists

A *linked list* is a data structure for collecting a sequence of objects, such that addition and removal of elements in the middle of the sequence is efficient.

To understand the need for such a data structure, imagine a program that maintains a vector of employee records, sorted by the last name of the employees. When a new employee is hired, an object needs to be inserted into the vector. Unless the company happens to hire employees in dictionary order, it is likely that a new employee object needs to be inserted into the middle of the vector. In that case, many other objects must be moved toward the end. Conversely, if an employee leaves the company, the hole in the sequence needs to be closed by moving all objects that came after it. Moving a large number of objects can involve a substantial amount of computer time. We would like to structure the data in a way that minimizes this cost.

A linked list consists of a number of nodes, each of which has a pointer to the neighboring nodes.

Rather than storing the data in a single block of memory, a *linked list* uses a different strategy. Each value is stored in its own memory block, together with the locations of the neighboring blocks in the sequence (see Figure 1).

It is now an easy matter to add another value to the sequence (see Figure 2), or to remove a value from the sequence (Figure 3), without moving the others.

Adding and removing elements in the middle of a linked list is efficient.

What's the catch? Linked lists allow speedy insertion and removal, but element access can be slow. For example, suppose you want to locate the fifth element. You must first traverse the first four. This is a problem if you need to access the elements in arbitrary order. The term *random access* is used in computer science to describe an access pattern in which elements are accessed in arbitrary (not necessarily random) order. In contrast, sequential access visits the elements in sequence. For example, a binary search requires random access, whereas a linear search requires only sequential access.

Visiting the elements of a linked list in sequential order is efficient, but random access is not.

Of course, if you mostly visit all elements in sequence (for example, to display or print the elements), the inefficiency of random access is not a problem. You use linked lists when you are concerned about the efficiency of inserting or removing elements and you rarely need element access in random order.

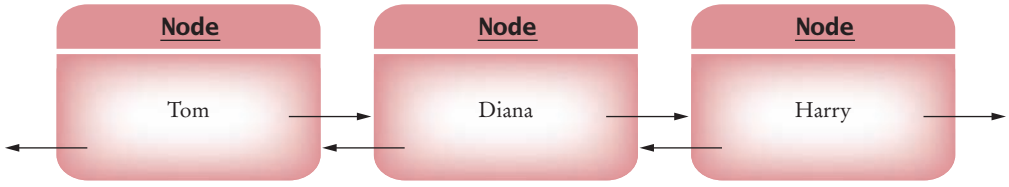


Figure 1 A Linked List

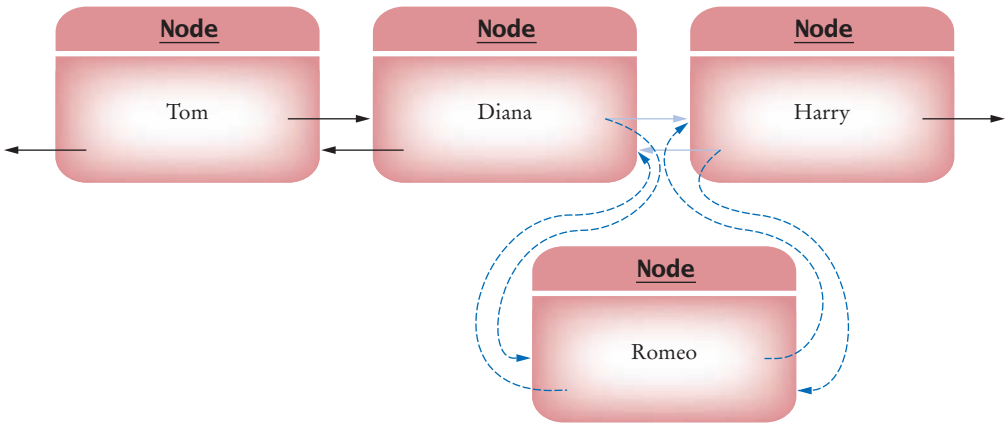


Figure 2 Adding a Node to a Linked List

The standard C++ library has an implementation of the linked list container structure. In this section, you will learn how to use the standard linked list structure. Later you will look "under the hood" and find out how to implement linked lists. (The linked list of the standard C++ library has links going in both directions. Such a list is often called a *doubly-linked* list. A *singly-linked* list lacks the links to the predecessor elements.)

Just like `vector`, the standard `list` is a *template:* You can declare lists for different types. For example, to make a list of strings, define an object of type `list<string>`. Then you can use the `push_back` function to add strings to the end of the list. The following code segment defines a list of strings, `names`, and adds three strings to it:

```
list<string> names;

names.push_back("Tom");
names.push_back("Diana");
names.push_back("Harry");
```

This code looks exactly like the code that you would use to build a vector of strings. There is, however, one major difference. Suppose you want to access the last element in the list. You cannot directly refer to `names[2]`. Because the values are not stored in one contiguous block in memory, there is no immediate way to access the third element. Instead, you must visit each element in turn, starting at the beginning of the list and then proceeding to the next element.

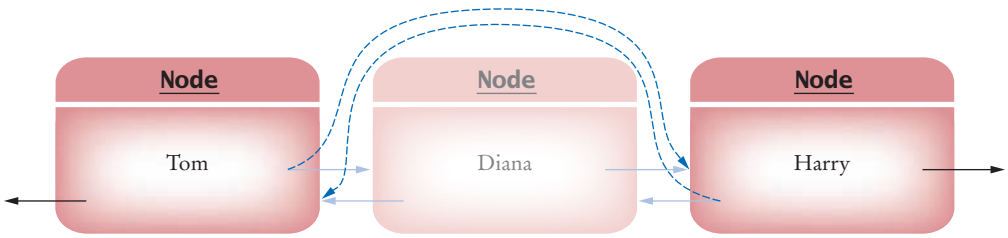


Figure 3 Removing a Node from a Linked List

You can inspect and edit a linked list with an iterator. An iterator points to a node in a linked list.

To visit an element, you use a *list iterator*. An iterator marks a *position* in the list. To get an iterator that marks the beginning position in the list, you define an iterator variable, then call the `begin` function of the `list` class to get the beginning position:

```
list<string>::iterator pos;
pos = names.begin();
```

To move the iterator to the next position, use the `++` operator:

```
pos++;
```

You can also move the iterator backward with the `--` operator:

```
pos--;
```

You use the `*` operator to find the value that is stored in the position marked by the iterator:

```
string value = *pos;
```

You have to be careful to distinguish between the iterator `pos`, which represents a position in the list, and the value `*pos`, which represents the value that is stored in the list. For example, if you change `*pos`, then you update the contents in the list:

```
*pos = "Romeo";
   // The list value at the position is changed
```

If you change `pos`, then you merely change the current position.

```
pos = names.begin();
   // The position is again at the beginning of the list
```

To insert another string before the iterator position, use the `insert` function:

```
names.insert(pos, "Romeo");
```

The `insert` function inserts the new element *before* the iterator position, rather than after it. This convention makes it easy to insert a new element before the first value of the list:

```
pos = names.begin();
names.insert(pos, "Romeo");
```

That raises the question of how you insert a value after the end of the list. Each list has an *end position* that does not correspond to any value in the list but that points past the list's end. The `end` function returns that position:

```
pos = names.end(); // Points past the end of the list
names.insert(pos, "Juliet");
  // Insert past the end of the list
```

It is an error to compute

```
string value = *names.end(); // Error
```

The `end` position does not point to any value, so you cannot look up the value at that position. This error is equivalent to the error of accessing `v[10]` in a vector with 10 elements.

The `end` position has another useful purpose: it is the stopping point for traversing the list. The following code iterates over all elements of the list and prints them out:

```
pos = names.begin();
while (pos != names.end())
{
   cout << *pos << endl;
   pos++;
```

```
}
```

The traversal can be described more concisely with a `for` loop:

```
for (pos = names.begin(); pos != names.end(); pos++)
{
   cout << *pos << endl;
}
```

Of course, this looks very similar to the typical `for` loop for traversing an array:

```
for (i = 0; i < size; i++)
{
   cout << a[i] << endl;
}
```

Finally, to remove an element from a list, you move an iterator to the position that you want to remove, then call the `erase` function. The `erase` function returns an iterator that points to the element after the one that has been erased.

The following code erases the second element of the list:

```
pos = names.begin();
pos++;
pos = names.erase(pos);
```

Now `pos` points to the element that was previously the third element and is now the second element.

Here is a short example program that adds elements to a list, inserts and erases list elements, and finally traverses the resulting list:

ch13/list1.cpp

```
1  #include <string>
2  #include <list>
3  #include <iostream>
4
5  using namespace std;
6
7  int main()
8  {
9     list<string> names;
10
11    names.push_back("Tom");
12    names.push_back("Diana");
13    names.push_back("Harry");
14    names.push_back("Juliet");
15
16    // Add a value in fourth place
17
18    list<string>::iterator pos = names.begin();
19    pos++;
20    pos++;
21    pos++;
22
23    names.insert(pos, "Romeo");
24
25    // Remove the value in second place
26
27    pos = names.begin();
28    pos++;
29
30    names.erase(pos);
```

```
31
32    // Print all values
33
34    for (pos = names.begin(); pos != names.end(); pos++)
35    {
36       cout << *pos << endl;
37    }
38
39    return 0;
40 }
```

Program Run

```
Tom
Harry
Romeo
Juliet
```

1. Do linked lists take more storage space than arrays of the same size?
2. Why don't we need iterators with arrays?
3. Make a linked list of integers containing the numbers 1 through 10.
4. How do you erase the first element of the linked list `names`?
5. How do you erase the last element of the linked list `names`?
6. How do you add `"Buffy"` as the second element in the list `names`?

Practice It Now you can try these exercises at the end of the chapter: R13.4, R13.5, P13.4.

13.2 Implementing Linked Lists

The previous section showed you how to put linked lists to use. However, because the implementation of the `list` class is hidden from you, you had to take it on faith that the list values are really stored in separate memory blocks. We will now walk through an implementation of the list, node, and iterator classes.

For simplicity, we will implement linked lists of strings. To implement the linked list class in C++ that can hold values of arbitrary types, you need to know how to program with templates (see Horstmann and Budd, *Big C++, 2nd ed.*, Chapter 16). To implement iterators that behave exactly like the ones in the C++ library, you also need to know about operator overloading and nested classes (see *Big C++, 2nd ed.*, Chapters 17 and 18).

13.2.1 The Classes for Lists, Nodes, and Iterators

The `list` class of the standard library defines many useful member functions. For simplicity, we will only study the implementation of the most useful ones: `push_back`, `insert`, `erase`, and the iterator operations. We call our class `List`, with an uppercase `L`, to differentiate it from the standard `list` class template.

A linked list stores each value in a separate object, called a *node*. A node object holds a value, together with pointers to the previous and next nodes:

When implementing a linked list, we need to define list, node, and iterator classes.

```
class Node
{
public:
   Node(string s);
private:
   string data;
   Node* previous;
   Node* next;
friend class List;
friend class Iterator;
};
```

A list node contains pointers to the next and previous nodes.

Note the `friend` declarations. They indicate that the `List` and `Iterator` member functions are allowed to inspect and modify the data members of the `Node` class, which we will write presently.

A class should not grant friendship to another class lightly, because it breaks the privacy protection. In this case, it makes sense, though, because the list and iterator functions do all the necessary work and the node class is just an artifact of the implementation that is invisible to the users of the list class. Note that no code other than the member functions of the list and iterator classes can access the data members of the node class, so the data integrity is still guaranteed.

A list object contains pointers to the first and last nodes.

A list object holds the locations of the first and last nodes in the list:

```
class List
{
public:
   List();
   void push_back(string data);
   void insert(Iterator pos, string s);
   Iterator erase(Iterator pos);
   Iterator begin();
   Iterator end();
private:
   Node* first;
   Node* last;
friend class Iterator;
};
```

If the list is empty, then the `first` and `last` pointers are `NULL`. Note that a list object stores no data; it just knows where to find the node objects that store the list contents.

An iterator contains a pointer to the current node, and to the list that contains it.

Finally, an *iterator* denotes a position in the list. It holds a pointer to the node that denotes its current position, and a pointer to the list that created it. Our iterator class uses member functions `get`, `next`, `previous`, and `equals` instead of operators `*`, `++`, `--`, and `==`. For example, we will call `pos.next()` instead of `pos++`.

```
class Iterator
{
public:
   Iterator();
   string get() const;
   void next();
   void previous();
   bool equals(Iterator b) const;
private:
   Node* position;
   List* container;
friend class List;
};
```

If the iterator points past the end of the list, then the `position` pointer is `NULL`. In that case, the `previous` member function uses the `container` pointer to move the iterator back from the past-the-end position to the last element of the list. (This is only one possible choice for implementing the past-the-end position. Another choice would be to store an actual dummy node at the end of the list. Some implementations of the standard `list` class do just that.)

13.2.2 Implementing Iterators

Iterators are created by the `begin` and `end` member functions of the `List` class. The `begin` function creates an iterator whose `position` pointer points to the first node in the list. The `end` function creates an iterator whose `position` pointer is `NULL`.

```
Iterator List::begin()
{
   Iterator iter;
   iter.position = first;
   iter.container = this;
   return iter;
}

Iterator List::end()
{
   Iterator iter;
   iter.position = NULL;
   iter.container = this;
   return iter;
}
```

The `next` function (which is the equivalent of the `++` operator) advances the iterator to the next position. This is a very typical operation in a linked list; let us study it in detail. The `position` pointer points to the current node in the list. That node has a data member `next`. Because `position` is a node pointer, you use the `->` operator to access the data member `next`:

```
position->next
```

That `next` data member is itself a pointer, pointing to the next node in the linked list (see Figure 4). To make `position` point to that next node, write

```
position = position->next;
```

Thus, the `next` function is simply:

```
void Iterator::next()
{
   position = position->next;
}
```

Note that you can evaluate `position->next` only if `position` is not `NULL`, because it is an error to dereference a `NULL` pointer. That is, it is illegal to advance the iterator once it is in the past-the-end position. Our implementation does not check for this error; neither does the implementation of the standard C++ library.

The `previous` function (which is the equivalent of the `--` operator) is a bit more complex. In the ordinary case, you move the position backward with the instruction

```
position = position->previous;
```

However, if the iterator is currently past the end, then you must make it point to the last element in the list.

```
void Iterator::previous()
{
   if (position == NULL)
   {
      position = container->last;
   }
   else
   {
      position = position->previous;
   }
}
```

The `get` function (which is the equivalent of the `*` operator) simply returns the `data` value of the node to which `position` points—that is, `position->data`. It is illegal to call `get` if the iterator points past the end of the list:

```
string Iterator::get() const
{
   return position->data;
}
```

Finally, the `equals` function (which is the equivalent of the `==` operator) compares two `position` pointers:

```
bool Iterator::equals(Iterator b) const
{
  return position == b.position;
}
```

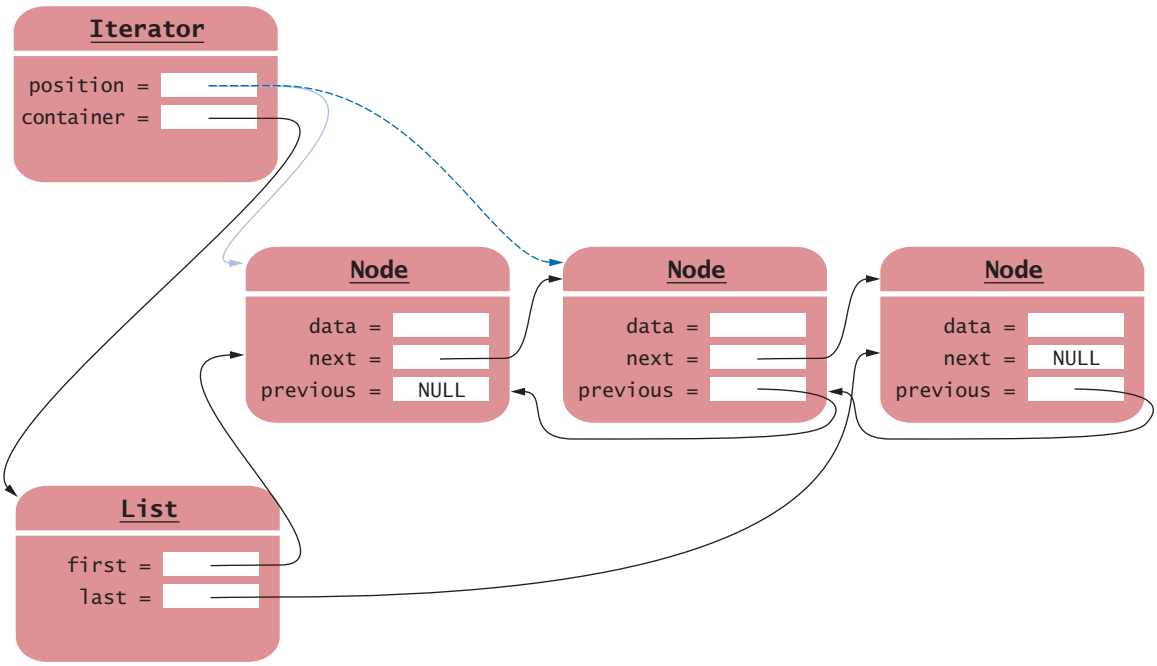


Figure 4 Advancing an Iterator

13.2.3 Implementing Insertion and Removal

In the last section you saw how to implement the iterators that traverse an existing list. Now you will see how to build up lists by adding and removing elements, one step at a time.

First, we will implement the `push_back` function. It appends an element to the end of the list (see Figure 5). Make a new node:

```
Node* new_node = new Node(s);
```

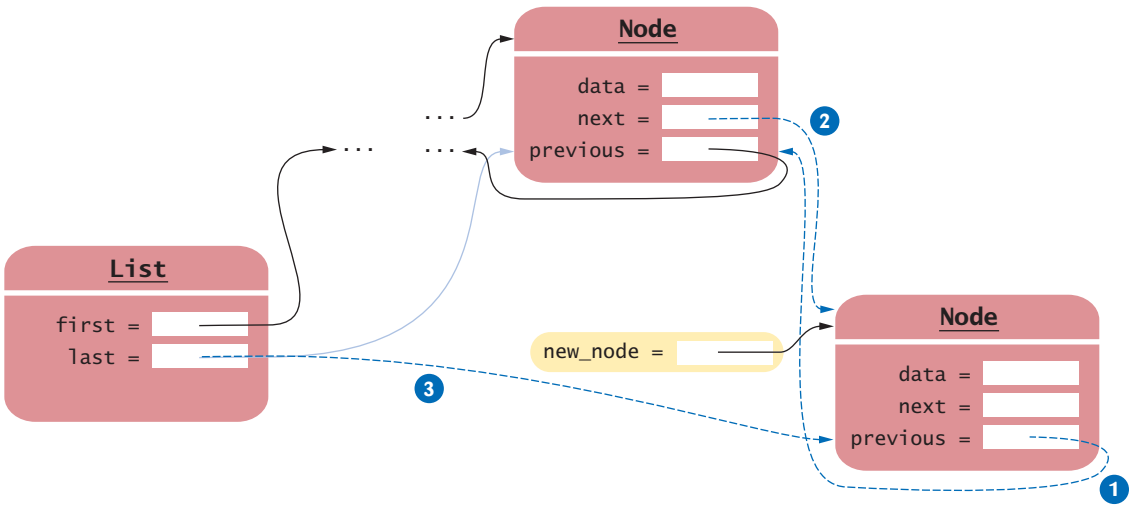


Figure 5 Appending a Node to the End of a Linked List

List nodes are allocated on the heap, using the new operator.

This new node must be integrated into the list after the node to which the `last` pointer points. That is, the data member `next` of the last node (which is currently `NULL`) must be updated to `new_node`. Also, the data member `previous` of the new node must point to what used to be the last node:

```
new_node->previous = last; (1)
last->next = new_node; (2)
```

Finally, you must update the `last` pointer to reflect that the new node is now the last node in the list:

```
last = new_node; (3)
```

However, there is a special case when `last` is `NULL`, which can happen only when the list is empty. After the call to `push_back`, the list has a single node—namely, `new_node`. In that case, both `first` and `last` must be set to `new_node`:

```
void List::push_back(string data)
{
   Node* new_node = new Node(data);
   if (last == NULL) // List is empty
   {
      first = new_node;
      last = new_node;
   }
   else
   {
```

```
      new_node->previous = last;
      last->next = new_node;
      last = new_node;
   }
}
```

Inserting an element in the middle of a linked list is a little more difficult, because the node pointers in the *two* nodes surrounding the new node need to be updated. The function declaration is

```
void List::insert(Iterator iter, string s)
```

That is, a new node containing `s` is inserted before `iter.position` (see Figure 6).

Give names to the surrounding nodes. Let `before` be the node before the insertion location, and let `after` be the node after that. That is,

```
Node* after = iter.position;
Node* before = after->previous;
```

What happens if `after` is `NULL`? After all, it is illegal to apply `->` to a `NULL` pointer. In this situation, you are inserting past the end of the list. Simply call `push_back` to handle that case separately. Otherwise, you need to insert `new_node` between `before` and `after`:

```
new_node->previous = before;  1
new_node->next = after;  2
```

You must also update the nodes from `before` and `after` to point to the new node:

```
after->previous = new_node;  3
before->next = new_node;   // If before != NULL  4
```

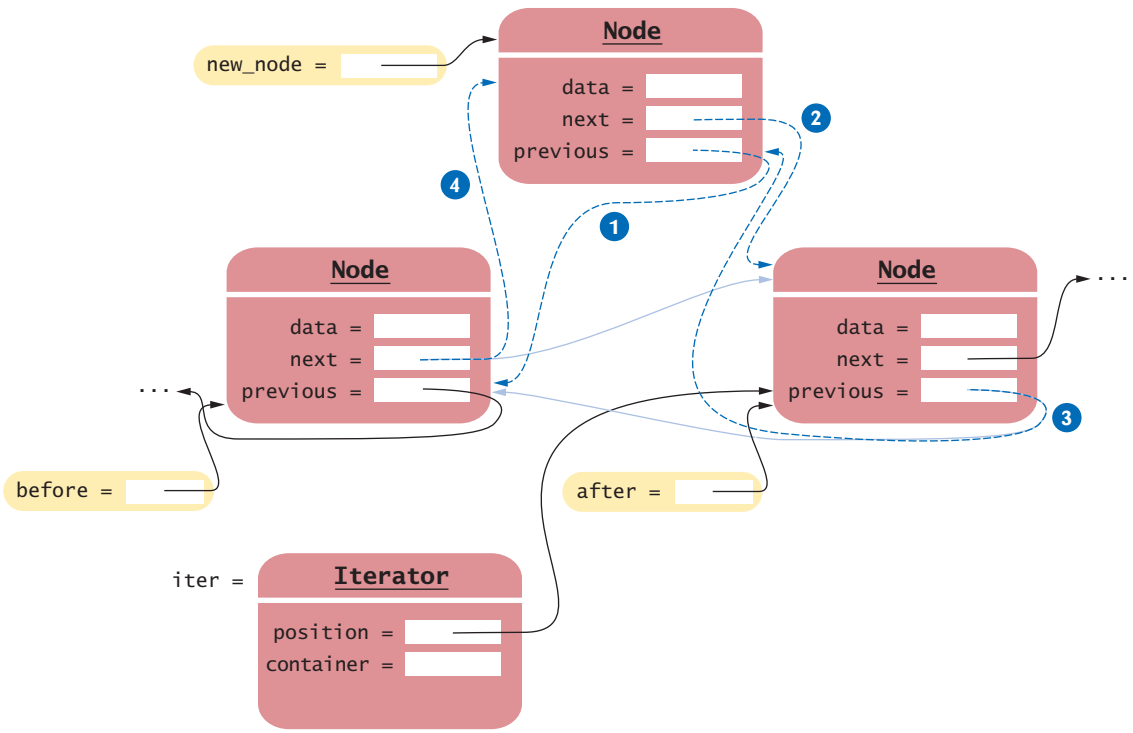


Figure 6 Inserting a Node into a Linked List

However, you must be careful. You know that after is not NULL, but it is possible that before is NULL. In that case, you are inserting at the beginning of the list and need to adjust first:

```
if (before = NULL) // Insert at beginning
{
   first = new_node;
}
else
{
   before->next = new_node;
}
```

Here is the complete code for the insert function:

```
void List::insert(Iterator iter, string s)
{
   if (iter.position == NULL)
   {
      push_back(s);
      return;
   }

   Node* after = iter.position;
   Node* before = after->previous;
   Node* new_node = new Node(s);
   new_node->previous = before;
   new_node->next = after;
   after->previous = new_node;
   if (before == NULL) // Insert at beginning
   {
      first = new_node;
   }
   else
   {
      before->next = new_node;
   }
}
```

Finally, look at the implementation of the erase function:

```
Iterator List::erase(Iterator iter)
```

You want to remove the node to which iter.position points. As before, give names to the node to be removed, the node before it, and the node after it:

```
Node* remove = iter.position;
Node* before = remove->previous;
Node* after = remove->next;
```

You need to update the next and previous pointers of the before and after nodes to bypass the node that is to be removed (see Figure 7).

```
before->next = after;  // If before != NULL 1
after->previous = before;  // If after != NULL 2
```

However, as before, you need to cope with the possibility that `before`, `after`, or both are `NULL`. If `before` is `NULL`, you are erasing the first element in the list. It has no predecessor to update, but you must change the `first` pointer of the list. Conversely, if `after` is `NULL`, you are erasing the last element of the list and must update the `last` pointer of the list:

```
if (remove == first)
{
   first = after;
}
else
{
   before->next = after;
}
if (remove == last)
{
   last = before;
}
else
{
   after->previous = before;
}
```

You must adjust the iterator position so it no longer points to the removed element.

```
iter.position = after;
```

3

When a list node is erased, it is recycled to the heap with the `delete` operator.

Finally, you must remember to recycle the removed node:

```
delete remove;
```

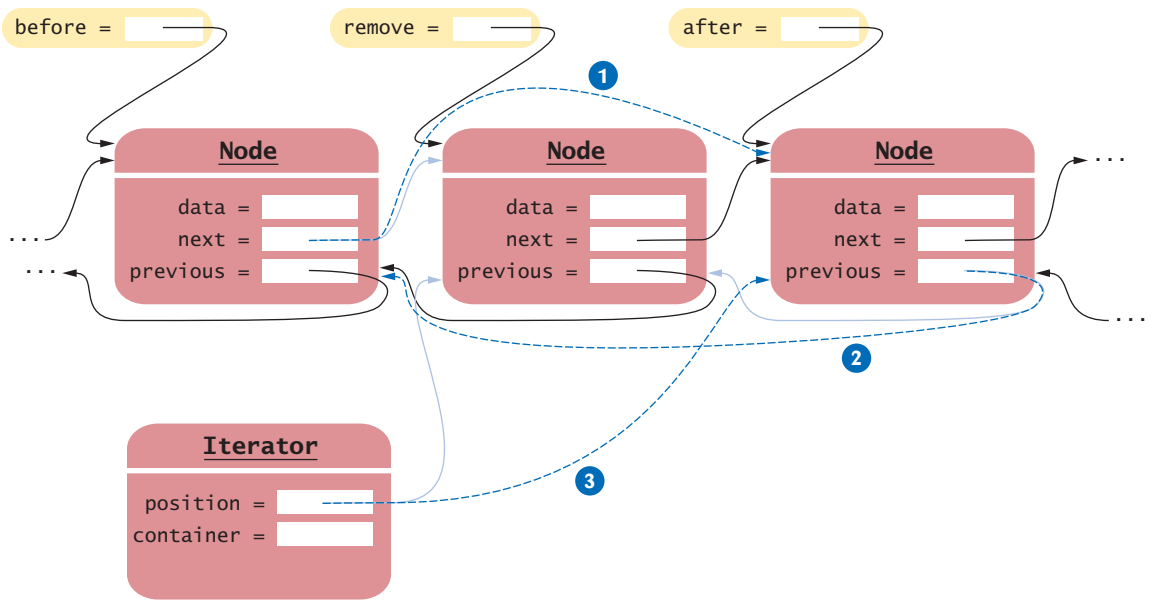


Figure 7 Removing a Node from a Linked List

Here is the complete `erase` function. Note that the function returns an iterator to the element following the erased one:

```
Iterator List::erase(Iterator iter)
{
   Node* remove = iter.position;
   Node* before = remove->previous;
   Node* after = remove->next;
   if (remove == first)
   {
      first = after;
   }
   else
   {
      before->next = after;
   }
   if (remove == last)
   {
      last = before;
   }
   else
   {
      after->previous = before;
   }
   delete remove;
   Iterator r;
   r.position = after;
   r.container = this;
   return r;
}
```

Implementing operations that modify a linked list is challenging—you need to make sure that you update all node pointers correctly.

Implementing these linked list operations is somewhat complex. It is also error-prone. If you make a mistake and misroute some of the pointers, you can get subtle errors. For example, if you make a mistake with a `previous` pointer, you may never notice it until you traverse the list backwards. If a node has been deleted, then that same storage area may later be reallocated for a different purpose, and if you have kept a pointer to it, following that invalid node pointer will lead to disaster. You must exercise special care when implementing any operations that manipulate the node pointers directly.

Here is a program that puts our linked list to use and demonstrates the `insert` and `erase` operations:

ch13/list2.cpp

```
1  #include <string>
2  #include <iostream>
3
4  using namespace std;
5
6  class List;
7  class Iterator;
8
9  class Node
10 {
11 public:
12    /**
13       Constructs a node with a given data value.
14       @param s the data to store in this node
```

```
15     */
16     Node(string s);
17  private:
18     string data;
19     Node* previous;
20     Node* next;
21  friend class List;
22  friend class Iterator;
23  };
24
25  class List
26  {
27  public:
28     /**
29        Constructs an empty list.
30     */
31     List();
32     /**
33        Appends an element to the list.
34        @param data the value to append
35     */
36     void push_back(string data);
37     /**
38        Inserts an element into the list.
39        @param iter the position before which to insert
40        @param s the value to append
41     */
42     void insert(Iterator iter, string s);
43     /**
44        Removes an element from the list.
45        @param iter the position to remove
46        @return an iterator pointing to the element after the
47        erased element
48     */
49     Iterator erase(Iterator iter);
50     /**
51        Gets the beginning position of the list.
52        @return an iterator pointing to the beginning of the list
53     */
54     Iterator begin();
55     /**
56        Gets the past-the-end position of the list.
57        @return an iterator pointing past the end of the list
58     */
59     Iterator end();
60  private:
61     Node* first;
62     Node* last;
63  friend class Iterator;
64  };
65
66  class Iterator
67  {
68  public:
69     /**
70        Constructs an iterator that does not point into any list.
71     */
72     Iterator();
73     /**
74        Looks up the value at a position.
```

```
75          @return the value of the node to which the iterator points
76       */
77       string get() const;
78       /**
79          Advances the iterator to the next node.
80       */
81       void next();
82       /**
83          Moves the iterator to the previous node.
84       */
85       void previous();
86       /**
87          Compares two iterators.
88          @param b the iterator to compare with this iterator
89          @return true if this iterator and b are equal
90       */
91       bool equals(Iterator b) const;
92    private:
93       Node* position;
94       List* container;
95    friend class List;
96    };
97
98    Node::Node(string s)
99    {
100      data = s;
101      previous = NULL;
102      next = NULL;
103   }
104
105   List::List()
106   {
107      first = NULL;
108      last = NULL;
109   }
110
111   void List::push_back(string data)
112   {
113      Node* new_node = new Node(data);
114      if (last == NULL) // List is empty
115      {
116         first = new_node;
117         last = new_node;
118      }
119      else
120      {
121         new_node->previous = last;
122         last->next = new_node;
123         last = new_node;
124      }
125   }
126
127   void List::insert(Iterator iter, string s)
128   {
129      if (iter.position == NULL)
130      {
131         push_back(s);
132         return;
133      }
134
```

```
135      Node* after = iter.position;
136      Node* before = after->previous;
137      Node* new_node = new Node(s);
138      new_node->previous = before;
139      new_node->next = after;
140      after->previous = new_node;
141      if (before == NULL) // Insert at beginning
142      {
143         first = new_node;
144      }
145      else
146      {
147         before->next = new_node;
148      }
149   }
150
151   Iterator List::erase(Iterator iter)
152   {
153      Node* remove = iter.position;
154      Node* before = remove->previous;
155      Node* after = remove->next;
156      if (remove == first)
157      {
158         first = after;
159      }
160      else
161      {
162         before->next = after;
163      }
164      if (remove == last)
165      {
166         last = before;
167      }
168      else
169      {
170         after->previous = before;
171      }
172      delete remove;
173      Iterator r;
174      r.position = after;
175      r.container = this;
176      return r;
177   }
178
179   Iterator List::begin()
180   {
181      Iterator iter;
182      iter.position = first;
183      iter.container = this;
184      return iter;
185   }
186
187   Iterator List::end()
188   {
189      Iterator iter;
190      iter.position = NULL;
191      iter.container = this;
192      return iter;
193   }
194
```

```
195 Iterator::Iterator()
196 {
197    position = NULL;
198    container = NULL;
199 }
200
201 string Iterator::get() const
202 {
203    return position->data;
204 }
205
206 void Iterator::next()
207 {
208    position = position->next;
209 }
210
211 void Iterator::previous()
212 {
213    if (position == NULL)
214    {
215       position = container->last;
216    }
217    else
218    {
219       position = position->previous;
220    }
221 }
222
223 bool Iterator::equals(Iterator b) const
224 {
225    return position == b.position;
226 }
227
228 int main()
229 {
230    List names;
231
232    names.push_back("Tom");
233    names.push_back("Diana");
234    names.push_back("Harry");
235    names.push_back("Juliet");
236
237    // Add a value in fourth place
238
239    Iterator pos = names.begin();
240    pos.next();
241    pos.next();
242    pos.next();
243
244    names.insert(pos, "Romeo");
245
246    // Remove the value in second place
247
248    pos = names.begin();
249    pos.next();
250
251    names.erase(pos);
252
253    // Print all values
254
```

```
255    for (pos = names.begin(); !pos.equals(names.end()); pos.next())
256    {
257       cout << pos.get() << endl;
258    }
259
260    return 0;
261 }
```

Program Run

```
Tom
Harry
Romeo
Juliet
```

7. Trace through the `push_back` method when adding an element to an empty list.
8. If the iterator `pos` has been set to `names.end()`, trace through the call `names.insert(pos, "Fred")`.
9. If the iterator `pos` has been set to `names.begin()`, trace through the call `names.insert(pos, "Fred")`. Assume the list is not empty.
10. Why does the insert method have three separate cases?
11. What happens when you try to move an iterator past the end of a list?

Practice It Now you can try these exercises at the end of the chapter: P13.6, P13.7, P13.10.

13.3 The Efficiency of List, Array, and Vector Operations

In this section, we will formally analyze how efficient the fundamental operations on linked lists, arrays, and vectors are. We will consider these operations:

- Getting the *k*th element
- Adding and removing an element at a given position (an iterator or index)
- Adding and removing an element at the end

Locating the *k*th element is an $O(k)$ operation for linked lists.

To get the *k*th element of a linked list, you start at the beginning of the list and advance the iterator *k* times. Suppose it takes an amount of time *T* to advance the iterator once. This quantity is independent of the iterator position—advancing an iterator does some checking and then it follows the `next` pointer. Therefore, advancing the iterator to the *k*th element consumes *kT* time. Therefore, locating the *k*th element is an $O(k)$ operation.

Locating an element is an $O(1)$ operation for arrays and vectors.

To get the *k*th element of an array, we use an expression such as `a[k]`. This is executed in a constant amount of time that is independent of *k*. We say that accessing an array element takes $O(1)$ time.

To analyze the situation for vectors, we need to peek under the hood and see how the `vector` class is implemented.

A vector maintains a pointer to an array of elements termed the *buffer*. An integer data member, called the *capacity*, is the maximum number of elements that can be

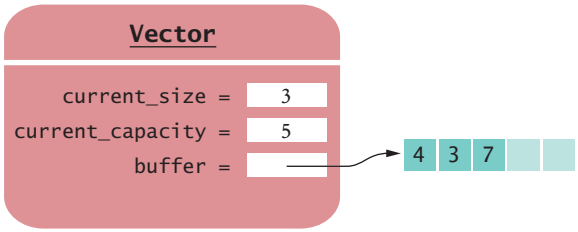


Figure 8
Internal Data Members Maintained by Vector

stored in the current buffer. The buffer is usually larger than is necessary to hold the current elements in the container. The *size* is the number of elements actually being held by the container. Because vectors use zero-based indexing, the size can also be interpreted as the first free location in the array. Figure 8 shows vector internals.

The *k*th element is accessed through the expression `buffer[k]`, which is done in constant or $O(1)$ time.

Next, consider the task of adding an element in the middle of a list, array, or vector. For a linked list, we assume that we already have an iterator to the insertion location. It might have taken some time to get there, but we are now concerned with the cost of insertion after the position has been established.

Adding an element in a linked list is an $O(1)$ operation.

As shown in Figure 6, you add an element by modifying the `previous` and `next` pointers of the new node and the surrounding nodes. This operation takes a constant number of steps, independent of the position. The same holds for removing an element. We conclude that list insertion and removal are $O(1)$ operations.

Adding an element in the middle of an array or vector of size n is an $O(n)$ operation.

For arrays and vectors, the situation is less rosy. To insert an element at position k, the elements with higher index values need to move (see Figure 9). How many elements are affected? For simplicity, we will assume that insertions happen at random locations. On average, each insertion moves $n / 2$ elements, where n is the size of the array or vector.

The same argument holds for removing an element. On average, $n / 2$ elements need to be moved. Therefore, we say that array and vector insertion and removal are $O(n)$ operations.

There is one situation where adding an element to an array or vector isn't so costly: when the insertion happens at the end. The `push_back` member function carries out that operation.

Adding an element to the end of an array is an $O(1)$ operation.

If the size of the vector is less than the capacity, the new element is simply moved into place and the size is incremented, as shown in Figure 10. This is an $O(1)$ operation.

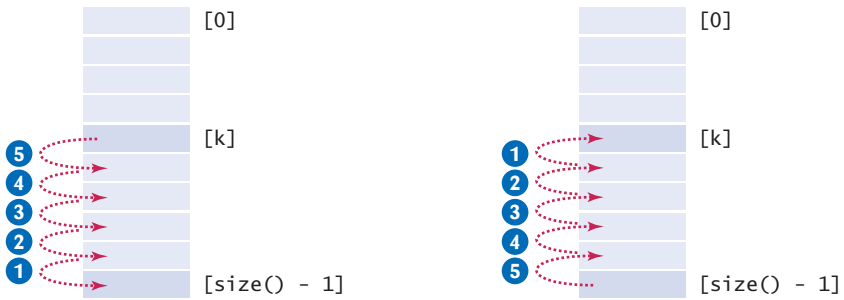
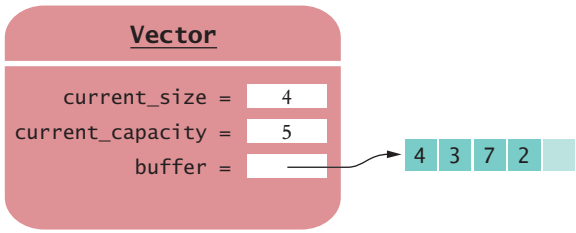


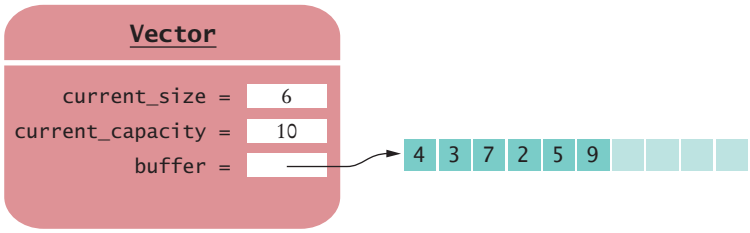
Figure 9 Inserting and Removing Vector Elements

Figure 10
Vector After push_back



If, however, the size is equal to the capacity, it means that no more space is available. With an array, there is nothing to be done—the element cannot be inserted. Vectors, on the other hand, can grow. In order to make new space, a new and larger buffer is allocated. This new buffer is typically twice the size of the current buffer (see Figure 11). The existing elements are then copied into the new buffer, the old buffer is deleted, and insertion takes place as before. Reallocation is an $O(n)$ operation because all elements need to be copied to the new buffer.

Figure 11
Vector After a Buffer Reallocation



If we carefully analyze the total cost of a sequence of push_back operations, it turns out that these reallocations are not as expensive as they first appear. The key observation is that reallocation does not happen very often. Suppose we start with a vector of capacity 10 and double the size with each reallocation. We must reallocate when the buffer reaches sizes 10, 20, 40, 80, 160, 320, 640, 1280, and so on.

Let us assume that one insertion without reallocation takes time T_1 and that reallocation of k elements takes time kT_2. What is the cost of 1,280 push_back operations? Of course, we pay $1280 \cdot T_1$ for the insertions. The reallocation cost is

$$\begin{aligned} 10T_2 + 20T_2 + 40T_2 + \cdots + 1280T_2 &= (1 + 2 + 4 + \cdots + 128) \cdot 10 \cdot T_2 \\ &= 255 \cdot 10 \cdot T_2 \\ &< 256 \cdot 10 \cdot T_2 \\ &= 1280 \cdot 2 \cdot T_2 \end{aligned}$$

Therefore, the total cost is a bit less than

$$1280 \cdot (T_1 + 2T_2)$$

In general, the total cost of n push_back operations is less than $n \cdot (T_1 + 2T_2)$. Because the second factor is a constant, we conclude that n push_back operations take $O(n)$ time.

We know that it isn't quite true that an individual push_back operation takes $O(1)$ time. After all, occasionally a push_back is unlucky and must reallocate the buffer. But if the cost of that reallocation is distributed over the preceding push_back operations, then the surcharge for each of them is still a constant amount.

Table 1 Execution Times for Container Operations

Operation	Array/Vector	Linked List
Add/remove element at end	$O(1)+$	$O(1)$
Add/remove element in the middle	$O(n)$	$O(1)$
Get kth element	$O(1)$	$O(k)$

An element can be added to the end of a vector in amortized $O(1)$ time.

We say that `push_back` takes *amortized* $O(1)$ time, which is written as $O(1)+$. (Accountants say that a cost is amortized when it is distributed over multiple periods.)

Finally, we note that the `push_back` operation for a linked list takes $O(1)$ time, provided that the linked list implementation maintains a pointer to the last element of the list. Table 1 summarizes the execution times that we discussed in this section.

12. What is the big-Oh efficiency for removing the middle element of a linked list?

13. What is the big-Oh efficiency for removing the middle element of an array?

14. Why doesn't it make sense to use a binary search algorithm on a sorted list?

Practice It Now you can try these exercises at the end of the chapter: R13.11, R13.15.

13.4 Stacks and Queues

In this section, you will consider two common data types that allow insertion and removal of items at the ends only, not in the middle.

A stack is a container of items with "last in, first out" retrieval.

A *stack* lets you insert and remove elements at one end only, traditionally called the *top* of the stack. To visualize a stack, think of a stack of books (see Figure 12).

New items can be added (or pushed) to the top of the stack. Items are removed (or popped) from the top of the stack as well. Therefore, they are removed in the order that is opposite from the order in which they have been added, also called *last in, first out* or *LIFO* order. For example, if you push strings `"Tom"`, `"Diana"`, and `"Harry"` into a stack, and then pop them one by one, then you will first see `"Harry"`, then `"Diana"`, and finally `"Tom"`.

Figure 12
A Stack of Books

To obtain a stack in the standard C++ library, you use the `stack` template:

```
stack<string> s;
s.push("Tom");
s.push("Diana");
s.push("Harry");
while (s.size() > 0)
{
   cout << s.top() << endl;
   s.pop();
}
```

The `pop` member function removes the top of the stack without returning a value. If you want to obtain the value before popping it, first call `top`, then `pop`.

A queue is a container of items with "first in, first out" retrieval.

A *queue* lets you add items to one end of the queue (the *back*) and remove them from the other end of the queue (the *front*). To visualize a queue, simply think of people lining up (see Figure 13). People join the back of the queue and wait until they have reached the front of the queue. Queues store items in a *first in, first out* or *FIFO* fashion. Items are removed in the same order in which they have been added.

The standard `queue` template implements a queue in C++. As with stacks, the addition and removal operations are called `push` and `pop`. The `front` member function yields the first element of the queue (that is, the next one to be removed). The `back` member function yields the element that was most recently added. You cannot access any other elements of the queue. Here is an example of using a queue:

```
queue<string> q;
q.push("Tom");
q.push("Diana");
q.push("Harry");
while (q.size() > 0)
{
   cout << q.front() << endl;
   q.pop();
}
```

In the standard C++ library, the `push` and `pop` functions of the `stack` and `queue` classes have $O(1)$ efficiency.

Figure 13 A Queue

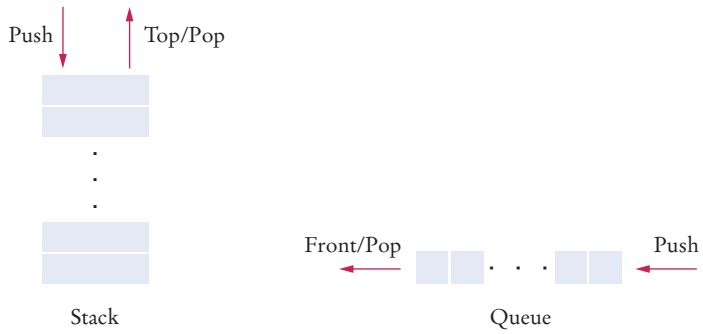


Figure 14 Stack and Queue Behavior

Figure 14 contrasts the behaviors of the stack and queue data types.

There are many uses of stacks and queues in computer science. For example, consider an algorithm that attempts to find a path through a maze. When the algorithm encounters an intersection, it pushes the location on the stack, and then it explores the first branch. If that branch is a dead end, it returns to the location at the top of the stack and explores the next untried branch. If all branches are dead ends, it pops the location off the stack, revealing a previously encountered intersection. Another important example is the run-time stack that a processor keeps to organize the variables of nested functions. Whenever a new function is called, its parameters and local variables are pushed onto a stack. When the method exits, they are popped off again. This stack makes recursive function calls possible.

As an example for the use of a queue, consider a printer that receives requests to print documents from multiple applications. If each of the applications sends printing data to the printer at the same time, then the printouts will be garbled. Instead, each application places all data to be sent to the printer into a file and inserts that file into the *print queue*. When the printer is done printing one file, it retrieves the next one from the queue. Therefore, print jobs are printed using the first in, first out rule, which is a fair arrangement for users of the shared printer.

The following sample program demonstrates the first-in, first-out order of a queue and the last-in, last-out order of a stack.

ch13/fifolifo.cpp

```
1  #include <iostream>
2  #include <string>
3  #include <queue>
4  #include <stack>
5
6  using namespace std;
7
8  int main()
9  {
10    cout << "FIFO order:" << endl;
11
12    queue<string> q;
13    q.push("Tom");
```

```
14    q.push("Diana");
15    q.push("Harry");
16
17    stack<string> s;
18    while (q.size() > 0)
19    {
20       string name = q.front();
21       q.pop();
22       cout << name << endl;
23       s.push(name);
24    }
25
26    cout << "LIFO order:" << endl;
27
28    while (s.size() > 0)
29    {
30       cout << s.top() << endl;
31       s.pop();
32    }
33
34    return 0;
35 }
```

Program Run

```
FIFO order:
Tom
Diana
Harry
LIFO order:
Harry
Diana
Tom
```

15. Why wouldn't you want to use a stack to manage print jobs?
16. What does this code print?

```
queue<int> q;
for (int i = 1; i <= 10; i++) { q.push(i); }
for (int i = 1; i <= 5; i++) { q.pop(); }
cout << q.front() << endl;
```

17. What does this code print?

```
stack<int> s;
for (int i = 1; i <= 10; i++) { s.push(i); }
for (int i = 1; i <= 5; i++) { s.pop(); }
cout << s.top() << endl;
```

18. Describe how a stack can be used to check whether the parentheses in an arithmetic expression are balanced correctly. For example, 3 + (4 / (5 – 6)) is correct, but 3 + (4 / 5)) – (6 is not.
19. Why would it not be a good idea to use a vector for a queue?

Practice It Now you can try these exercises at the end of the chapter: R13.16, R13.18, P13.18.

Random Fact 13.1 Reverse Polish Notation

When you write arithmetic expressions, you are used to operators with different levels of precedence that appear between the operands, except when parentheses are used to specify a different ordering. That is, an expression such as 3 + (4 − 2) × 7 is evaluated by first subtracting the 2 from the 4, then multiplying the result by 7, and finally adding the 3. Notice how the sequence of operations jumps around instead of being analyzed in a strict left to right or right to left order.

In the 1920s a Polish mathematician, Jan Łukasiewicz, noticed that if you wrote the operators first, before the operands, the need for both parentheses and precedence was eliminated and expressions could be read easily from left to right. In Łukasiewicz's notation, the expression would be written as + 3 × − 4 2 . Table 2 shows some other examples.

Evaluating an expression in Łukasiewicz's form is a simple recursive algorithm. Examine the next term; if it is a constant, then that is your result; if it is a binary operator, then recursively examine the following two expressions and produce their result. The scheme was termed Polish Notation in Łukasiewicz's honor (although one can argue it should be called Łukasiewicz Notation). Of course, an entrenched notation is not easily displaced, even when it has distinct disadvantages, and Łukasiewicz's discovery did not cause much of a stir for about 50 years.

In the 1950s, Australian computer scientist Charles Hamblin noted that an even better scheme would be to have the operators *follow* the operands. This was termed *Reverse Polish Notation*, or RPN. The expression given would be written as 3 4 2 − 7 × + in RPN. As you have seen, the evaluation of RPN is relatively simple if you have a stack. Each operand is pushed on the stack. Each operator pops the appropriate number of values from the stack, performs the operation, and pushes the result back onto the stack.

In 1972, Hewlett-Packard introduced the HP 35 calculator that used RPN. For example, to compute 3 + 4 * 5, you enter 3 4 5 * +. RPN calculators have no keys labeled with parentheses or an equals symbol. There is only a key labeled ENTER to push a number onto a stack. For that reason, Hewlett-Packard's marketing department used to refer to their product as "the calculators that have no equal". Indeed, the Hewlett-Packard calculators were a great advance over competing models that were unable to handle algebraic notation and left users with no other choice but to write intermediate results on paper.

Over time, developers of high quality calculators have adapted to the standard algebraic notation rather than forcing users to learn a new notation. However, those users who have made the effort of learning RPN tend to be fanatic proponents, and some Hewlett-Packard calculator models still support it.

Table 2 Polish Notation Examples

Standard Notation	Łukasiewicz Notation	RPN
3 + 4	+ 3 4	3 4 +
3 + 4 × 5	+ 3 * 4 5	3 4 5 * +
3 × (4 + 5)	* 3 + 4 5	3 4 5 + *
(3 + 4) × 5	* + 3 4 5	3 4 + 5 *
3 + 4 + 5	+ + 3 4 5	3 4 + 5 +

CHAPTER SUMMARY

Describe the linked list data structure and the use of list iterators.

- A linked list consists of a number of nodes, each of which has a pointer to the neighboring nodes.
- Adding and removing elements in the middle of a linked list is efficient.
- Visiting the elements of a linked list in sequential order is efficient, but random access is not.
- You can inspect and edit a linked list with an iterator. An iterator points to a node in a linked list.

Explain how linked lists are implemented.

- When implementing a linked list, we need to define list, node, and iterator classes.
- A list object contains pointers to the first and last nodes.
- An iterator contains a pointer to the current node, and to the list that contains it.
- List nodes are allocated on the heap, using the `new` operator.
- When a list node is erased, it is recycled to the heap with the `delete` operator.
- Implementing operations that modify a linked list is challenging—you need to make sure that you update all node pointers correctly.

Know the efficiencies of the fundamental operations on lists, arrays, and vectors.

- Locating the kth element is an $O(k)$ operation for linked lists.
- Locating an element is an $O(1)$ operation for arrays and vectors.
- Adding an element in a linked list is an $O(1)$ operation.
- Adding an element in the middle of an array or vector of size n is an $O(n)$ operation.
- Adding an element to the end of an array is an $O(1)$ operation.
- An element can be added to the end of a vector in amortized $O(1)$ time.

Describe the stack and queue data structures.

- A stack is a container of items with "last in, first out" retrieval.
- A queue is a container of items with "first in, first out" retrieval.

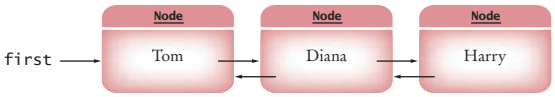
REVIEW EXERCISES

R13.1 If a list has n elements, how many legal positions are there for inserting a new element? For erasing an element?

R13.2 What happens if you keep advancing an iterator past the end of the list? Before the beginning of the list? What happens if you look up the value at an iterator that is past the end? If you erase the past-the-end position? All these are illegal operations, of course. What does the list implementation of your compiler do in these cases?

R13.3 Write a function that prints all values in a linked list, starting from the end of the list.

R13.4 The following code edits a linked list consisting of three nodes.



Draw a diagram showing how they are linked together after the following code is executed.

```
Node* p1 = first->next;
Node* p2 = first;
while (p2->next != NULL) { p2 = p2->next; }
first->next = p2;
p2->next = p1;
p1->next = NULL;
p2->previous = first;
p1->previous = p2;
last = p1;
```

R13.5 Explain what the following code prints.

```
list<string> names;
list<string>::iterator p = names.begin();
names.insert(p, "Tom");
p = names.begin();
names.insert(p, "Diana");
p++;
names.insert(p, "Harry");
for (p = names.begin(); p != names.end(); p++)
   { cout << *p << endl; }
```

R13.6 The `insert` procedure of Section 13.2.3 inserts a new element before the iterator position. To understand the updating of the nodes, draw before/after node diagrams for the following four scenarios.

a. The list is completely empty.

b. The list is not empty, and the iterator is at the beginning of the list.

c. The list is not empty, and the iterator is at the end of the list.

d. The list is not empty, and the iterator is in the middle of the list.

R13.7 What advantages do lists have over vectors? What disadvantages do they have?

R13.8 Suppose you need to organize a collection of telephone numbers for a company division. There are currently about 6,000 employees, and you know that the phone switch can handle at most 10,000 phone numbers. You expect several hundred look-ups against the collection every day. Would you use a vector or a linked list to store the information?

R13.9 Suppose you need to keep a collection of appointments. Would you use a linked list or a vector of `Appointment` objects?

R13.10 Suppose you write a program that models a card deck. Cards are taken from the top of the deck and given out to players. As cards are returned to the deck, they are placed on the bottom of the deck. Would you store the cards in a stack or a queue?

R13.11 Consider the efficiency of locating the kth element in a linked list of length n. If $k > n / 2$, it is more efficient to start at the end of the list and move the iterator to the previous element. Why doesn't this increase in efficiency improve the big-Oh estimate of random access in a linked list?

R13.12 Explain why inserting an element into the middle of a list is faster than inserting an element into the middle of a vector.

R13.13 Explain why the `push_back` operation with a vector is usually constant time, but occasionally much slower.

R13.14 Suppose a vector implementation were to add 10 elements at each reallocation instead of doubling the capacity. Show that the `push_back` operation no longer has amortized constant time.

R13.15 What is the big-Oh efficiency of selection sort when it is applied to a linked list?

R13.16 Suppose the strings `"A"` through `"Z"` are pushed onto a stack. Then they are popped off the stack and pushed onto a second stack. Finally, they are popped off the second stack and printed. In which order are the strings printed?

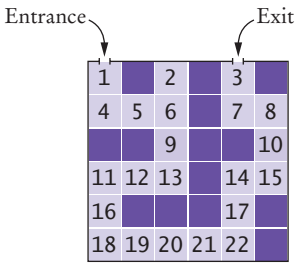
R13.17 What are the efficiencies of the `push` and `pop` operations of a stack when it is implemented using a linked list? Explain your answer.

R13.18 What are the efficiencies of the `push` and `pop` operations of a stack when it is implemented using a vector? Explain your answer.

R13.19 What are the efficiencies of the `push` and `pop` operations of a queue when it is implemented using a linked list? Explain your answer.

R13.20 What are the efficiencies of the `push` and `pop` operations of a queue when it is implemented using a vector? Explain your answer.

R13.21 Consider the following algorithm for traversing a maze such as this one:



Make the cell at the entrance the current cell. Take the following actions, then repeat:

- If the current cell is adjacent to the exit, stop.
- Mark the current cell as visited.
- Add all unvisited neighbors to the north, east, south, and west to a queue.
- Remove the next element from the queue and make it the current cell.

In which order will the cells of the sample maze be visited?

R13.22 Repeat Exercise R13.17, using a stack instead of a queue.

PROGRAMMING EXERCISES

P13.1 Write a function

```
void downsize(list<string>& names)
```

that removes every second value from a linked list.

P13.2 Write a function `maximum` that computes the largest element in a `list<int>`.

P13.3 Write a function `sort` that sorts the elements of a linked list (without copying them into a vector).

P13.4 Write a function `merge` that merges two lists into one, alternating elements from each list until the end of one of the lists has been reached, then appending the remaining elements of the other list. For example, merging the lists containing `A B C` and `D E F G H` should yield the list `A D B E C F G H`.

P13.5 Provide a linked list of integers by modifying the `Node`, `List`, and `Iterator` classes of Section 13.2 to hold integers instead of strings.

P13.6 Write a member function `List::reverse()` that reverses the nodes in a list.

P13.7 Write a member function `List::push_front()` that adds a value to the beginning of a list.

P13.8 Write a member function `List::swap(List& other)` that swaps the elements of this list and `other`. Your method should work in $O(1)$ time.

P13.9 Write a member function `List::get_size()` that computes the number of elements in the list, by counting the elements until the end of the list is reached.

P13.10 Add a `size` data member to the `List` class. Modify the `insert` and `erase` functions to update the data member `size` so that it always contains the correct size. Change the `get_size()` function of Exercise P13.9 to take advantage of this data member.

P13.11 Turn the linked list implementation into a *circular list:* Have the `previous` pointer of the first node point to the last node, and the `next` pointer of the last node point to the first node. Then remove the `last` pointer in the `List` class because the value can now be obtained as `first->previous`. Reimplement the member functions so that they have the same effect as before.

P13.12 Turn the linked list implementation into a *singly-linked list:* Drop the `previous` pointer of the nodes and the `previous` member function of the iterator. Reimplement the other member functions so that they have the same effect as before. *Hint:* In order to remove an element in constant time, iterators should store the predecessor of the current node.

P13.13 Modify the linked list implementation to use a *dummy node* for the past-the-end position whose `data` member is unused. A past-the-end iterator should point to the dummy node. Remove the `container` pointer in the iterator class. Reimplement the member functions so that they have the same effect as before.

P13.14 Write a class `Polynomial` that stores a polynomial such as

$$p(x) = 5x^{10} + 9x^7 - x - 10$$

as a linked list of terms. A term contains the coefficient and the power of x. For example, you would store $p(x)$ as

$$(5,10),(9,7),(-1,1),(-10,0)$$

Supply member functions to add, multiply, and print polynomials. Supply a constructor that makes a polynomial from a single term. For example, the polynomial `p` can be constructed as

```
Polynomial p(Term(-10, 0));
p.add(Polynomial(Term(-1, 1)));
p.add(Polynomial(Term(9, 7)));
p.add(Polynomial(Term(5, 10)));
```

Then compute $p(x) \times p(x)$.

```
Polynomial q = p.multiply(p);
q.print();
```

P13.15 Implement a `Stack` class, using a linked list of strings. Supply operations `size`, `push`, `pop`, and `top`, just like in the standard `stack` template.

P13.16 Implement a `Queue` class, using a linked list of strings. Supply operations `size`, `push`, `pop`, `front`, and `back`, just like in the standard `queue` template.

P13.17 Using a queue of vectors, implement a non-recursive variant of the merge sort algorithm as follows. Start by inserting the entire vector to be sorted. We assume its size is a power of 2. Keep removing vectors from the queue, splitting them into two vectors of equal size, and adding the smaller vectors back into the queue. Once you encounter vectors of size 1, change to the following behavior: Remove pairs of vectors from the queue, merge them into a single vector and add the result back into the queue. Stop when the queue has size 1.

P13.18 Use a stack to enumerate all permutations of a string without using recursion. Suppose you want to find all permutations of the string `meat`. Push the string `+meat` on the stack. Now repeat the following operations until the stack is empty:

- Pop off the top of the stack.
- If that string ends in a `+` (such as `tame+`), remove the `+` and print the string
- Otherwise, remove each letter in turn from the right of the `+`, insert it just before the `+`, and push the resulting string on the stack. For example, after popping `e+mta`, you push `em+ta`, `et+ma`, and `ea+mt`.

P13.19 In a paint program, a "flood fill" fills all empty pixels of a drawing with a given color, stopping when it reaches occupied pixels. In this exercise, you will implement a simple variation of this algorithm, flood-filling a 10×10 array of integers that are initially `0`. Prompt for the starting row and column. Push the (row, column) pair on a stack. (You will need to provide a simple `Pair` class.)

Then repeat the following operations until the stack is empty:

- Pop off the (row, column) pair from the top of the stack.
- If it has not yet been filled, fill it now. (Fill in numbers 1, 2, 3, and so on, to show the order in which the square is filled.)
- Push the coordinates of any unfilled neighbors in the north, east, south, or west direction on the stack.

When you are done, print the entire array.

P13.20 Repeat Exercise P13.19, but use a queue instead.

P13.21 Repeat Exercise P13.18, but use a queue instead.

ANSWERS TO SELF-CHECK QUESTIONS

1. Yes, for two reasons. You need to store the node pointers, and each node is a separate object. (There is a fixed overhead to store each object.)
2. An integer index can be used to access any array location.
3.
```
list<int> numbers;
for (int i = 1; i <= 10; i++) { numbers.push_back(i); }
```
4.
```
names.erase(names.begin());
```
5.
```
list<string>::iterator pos = names.end();
pos--;
names.erase(pos);
```
6.
```
list<string>::iterator pos = names.begin();
pos++;
names.insert(pos, "Buffy");
```
7. A new node is allocated that holds the data and has `next` and `previous` values set to `NULL`. An empty list has `first` and `last` set to `NULL`, so the first branch of the `if` statement is executed. Now `first` and `last` both point to the new node.
8. Tracing through the `end` function, we see that `pos.position` is `NULL`. Hence the `if` statement in the `insert` function is executed, which simply calls `push_back`.
9. In this case, `iter.position` points to the first node, and it is not `NULL`. `after` also points to the first node, and `before` is set to `NULL`. A new node is allocated. Its `previous` pointer is set to `NULL`, and its `next` pointer to the first node, thereby adding it before the first node in the linked list. To complete the linkage, the previously first node has its `previous` pointer updated to point to the new node. Because `before` is `NULL`, the `first` pointer of the list is updated.
10. When inserting at the end of the list, the `last` pointer needs to be updated. When inserting at the beginning of the linked list, the `first` pointer needs to be updated. When inserting in the middle, neither `first` nor `last` are updated.
11. At the end of the list, `iter.position` is `NULL`. When calling the `next` member function, the expression `position->next` is a null pointer error that will likely terminate your program.
12. To reach the middle of the linked list takes $n / 2$ traversal steps. The removal is done in constant time. Thus, the operation is $O(n)$.
13. To remove the middle element, the $n / 2$ elements beyond it must be moved. Thus, the operation is $O(n)$.
14. The first step in the binary search algorithm asks to visit the middle node in the list. That is an $O(n)$ operation, requiring the traversal of half the nodes. Thus, binary search is no longer $O(\log n)$. You might as well inspect the list elements as you traverse them.
15. Stacks use a "last in, first out" discipline. If you are the first one to submit a print job and lots of people add print jobs before the printer has a chance to deal with

your job, they get their printouts first, and you have to wait until all other jobs are completed.

16. 6
17. 5
18. When encountering a `(`, push it on a stack. When encountering a `)`, pop the stack. However, if the stack is empty, report an error. When the end of the expression is reached, the stack should be empty. If not, report an error.
19. Adding an element at the front of a vector is an $O(n)$ operation.