CHAPTER

17 Linked Lists

TOPICS

17.1 Introduction to the Linked List ADT

CONCEPT: Dynamically allocated data structures may be linked together in memory to form a chain.

A linked list is a series of connected *nodes*, where each node is a data structure. A linked list can grow or shrink in size as the program runs. This is possible because the nodes in a linked list are dynamically allocated. If new data need to be added to a linked list, the program simply allocates another node and inserts it into the series. If a particular piece of data needs to be removed from the linked list, the program deletes the node containing that data.

Advantages of Linked Lists over Arrays and `vectors`

Although linked lists are more complex to code and manage than arrays, they have some distinct advantages. First, a linked list can easily grow or shrink in size. In fact, the programmer doesn't need to know how many nodes will be in the list. They are simply created in memory as they are needed.

One might argue that linked lists are not superior to `vectors` (found in the Standard Template Library), because `vectors`, too, can expand or shrink. The advantage that linked lists have over `vectors`, however, is the speed at which a node may be inserted into or deleted from the list. To insert a value into the middle of a `vector` requires all the elements below the insertion point to be moved one position toward the `vector`'s end, thus making room for the new value. Likewise, removing a value from a `vector` requires all the elements below the removal point to be moved one position toward the `vector`'s beginning. When a node is inserted into or deleted from a linked list, none of the other nodes have to be moved.

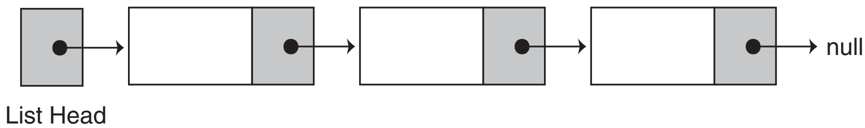
The Composition of a Linked List

Each node in a linked list contains one or more members that represent data. (Perhaps the nodes hold inventory records, or customer names, addresses, and telephone numbers.) In addition to the data, each node contains a pointer, which can point to another node. The makeup of a node is illustrated in Figure 17-1.

Figure 17-1

A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list. This creates a chain where the first node points to the second node, the second node points to the third node, and so on. This is illustrated in Figure 17-2.

Figure 17-2



The list depicted in Figure 17-2 has three nodes, plus a pointer known as the *list head*. The head simply points to the first node in the list. Each node, in turn, points to the next node in the list. The first node points to the second node, which points to the third node. Because the third node is the last one in the list, it points to the null address (address 0). This is usually how the end of a linked list is signified—by letting the last node point to null.

> **NOTE:** Figure 17-2 depicts the nodes in the linked list as being very close to each other, neatly arranged in a row. In reality, the nodes may be scattered around various parts of memory.

Declarations

So how is a linked list created in C++? First you must declare a data structure that will be used for the nodes. For example, the following `struct` could be used to create a list where each node holds a `double`:

```
struct ListNode
{
   double value;
   ListNode *next;
};
```

The first member of the `ListNode` structure is a `double` named `value`. It will be used to hold the node's data. The second member is a pointer named `next`. The pointer can hold the address of any object that is a `ListNode` structure. This allows each `ListNode` structure to point to the next `ListNode` structure in the list.

Because the `ListNode` structure contains a pointer to an object of the same type as that being declared, it is known as a *self-referential data structure*. This structure makes it possible to create nodes that point to other nodes of the same type.

The next step is to define a pointer to serve as the list head, as shown here.

```
ListNode *head;
```

Before you use the `head` pointer in any linked list operations, you must be sure it is initialized to `nullptr` because that marks the end of the list. Once you have declared a node data structure and have created a null `head` pointer, you have an empty linked list. The next step is to implement operations with the list.

Checkpoint

17.1 Describe the two parts of a node.

17.2 What is a list head?

17.3 What signifies the end of a linked list?

17.4 What is a self-referential data structure?

17.2 Linked List Operations

CONCEPT: The basic linked list operations are appending a node, traversing the list, inserting a node, deleting a node, and destroying the list.

In this section we will develop an abstract data type that performs basic linked list operations using the `ListNode` structure and `head` pointer defined in the previous section. We will use the following class declaration, which is stored in `NumberList.h`.

Contents of `NumberList.h`

```
// Specification file for the NumberList class
#ifndef NUMBERLIST_H
#define NUMBERLIST_H

class NumberList
{
private:
   // Declare a structure for the list
   struct ListNode
   {
      double value;           // The value in this node
      struct ListNode *next;  // To point to the next node
   };

   ListNode *head;            // List head pointer

```

```
public:
   // Constructor
   NumberList()
      { head = nullptr; }

   // Destructor
   ~NumberList();

   // Linked list operations
   void appendNode(double);
   void insertNode(double);
   void deleteNode(double);
   void displayList() const;
};
#endif
```

Notice that the constructor initializes the `head` pointer to `nullptr`. This establishes an empty linked list. The class has member functions for appending, inserting, and deleting nodes, as well as a `displayList` function that displays all the values stored in the list. The destructor destroys the list by deleting all its nodes. These functions are defined in `NumberList.cpp`. We will examine the member functions individually.

Appending a Node to the List

VideoNote
Appending a Node to a Linked List

To append a node to a linked list means to add the node to the end of the list. The `appendNode` member function accepts a `double` argument, `num`. The function will allocate a new `ListNode` structure, store the value in `num` in the node's `value` member, and append the node to the end of the list. Here is a pseudocode representation of the general algorithm:

```
Create a new node.
Store data in the new node.
If there are no nodes in the list
   Make the new node the first node.
Else
   Traverse the list to find the last node.
   Add the new node to the end of the list.
End If.
```

Here is the actual C++ code for the function:

```
void NumberList::appendNode(double num)
{
   ListNode *newNode; // To point to a new node
   ListNode *nodePtr; // To move through the list

   // Allocate a new node and store num there.
   newNode = new ListNode;
   newNode->value = num;
   newNode->next = nullptr;

   // If there are no nodes in the list
   // make newNode the first node.
```

```
23      if (!head)
24         head = newNode;
25      else // Otherwise, insert newNode at end.
26      {
27         // Initialize nodePtr to head of list.
28         nodePtr = head;
29
30         // Find the last node in the list.
31         while (nodePtr->next)
32            nodePtr = nodePtr->next;
33
34         // Insert newNode as the last node.
35         nodePtr->next = newNode;
36      }
37   }
```

Let's examine the statements in detail. In lines 13 and 14 the function defines the following local variables:

```
ListNode *newNode;  // To point to a new node
ListNode *nodePtr;  // To move through the list
```

The `newNode` pointer will be used to allocate and point to the new node. The `nodePtr` pointer will be used to travel down the linked list, in search of the last node.

The following statements, in lines 17 through 19, create a new node and store `num` in its `value` member:

```
newNode = new ListNode;
newNode->value = num;
newNode->next = nullptr;
```

The statement in line 19 is important. Because this node will become the last node in the list, its `next` pointer must be a null pointer.

In line 23, we test the `head` pointer to determine whether there are any nodes already in the list. If `head` points to `nullptr`, we make the new node the first in the list. Making `head` point to the new node does this. Here is the code:

```
if (!head)
   head = newNode;
```

If `head` does not point to `nullptr`, however, there are nodes in the list. The `else` part of the `if` statement must contain code to find the end of the list and insert the new node. The code, in lines 25 through 36, is shown here:

```
else
{
   // Initialize nodePtr to head of list.
   nodePtr = head;

   // Find the last node in the list.
   while (nodePtr->next)
      nodePtr = nodePtr->next;

   // Insert newNode as the last node.
   nodePtr->next = newNode;
}
```

The code uses `nodePtr` to travel down the list. It does this by first assigning `nodePtr` to `head` in line 28:

```
nodePtr = head;
```

The `while` loop in lines 31 and 32 is then used to *traverse* (or travel through) the list searching for the last node. The last node will be the one whose `next` member points to `nullptr`:

```
while (nodePtr->next)
   nodePtr = nodePtr->next;
```

When `nodePtr` points to the last node in the list, we make its `next` member point to `newNode` in line 35 with the following statement.

```
nodePtr->next = newNode;
```

This inserts `newNode` at the end of the list. (Remember, `newNode->next` already points to `nullptr`.)

Program 17-1 demonstrates the function.

Program 17-1

```
1  // This program demonstrates a simple append
2  // operation on a linked list.
3  #include <iostream>
4  #include "NumberList.h"
5  using namespace std;
6
7  int main()
8  {
9     // Define a NumberList object.
10    NumberList list;
11
12    // Append some values to the list.
13    list.appendNode(2.5);
14    list.appendNode(7.9);
15    list.appendNode(12.6);
16    return 0;
17 }
```

(This program displays no output.)

Let's step through the program, observing how the `appendNode` function builds a linked list to store the three argument values used.

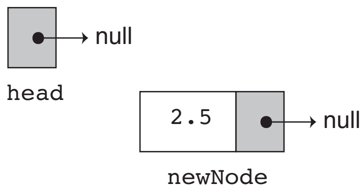
The `head` pointer is declared as a private member variable of the `NumberList` class. `head` is initialized to `nullptr` by the `NumberList` constructor, which indicates that the list is empty.

The first call to `appendNode` in line 13 passes 2.5 as the argument. In the following statements, a new node is allocated in memory, 2.5 is copied into its `value` member, and `nullptr` is assigned to the node's `next` pointer:

```
newNode = new ListNode;
newNode->value = num;
newNode->next = nullptr;
```

Figure 17-3 illustrates the state of the `head` pointer and the new node.

Figure 17-3

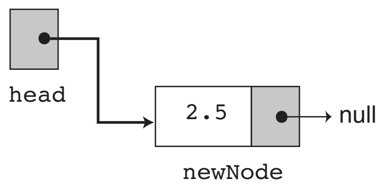


The next statement to execute is the following `if` statement:

```
if (!head)
   head = newNode;
```

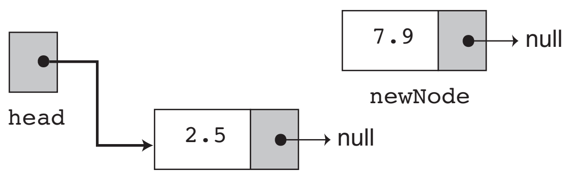
Because `head` is a null pointer, the condition `!head` is true. The statement `head = newNode;` is executed, making `newNode` the first node in the list. This is illustrated in Figure 17-4.

Figure 17-4



There are no more statements to execute, so control returns to function `main`. In the second call to `appendNode`, in line 14, 7.9 is passed as the argument. Once again, the first three statements in the function create a new node, store the argument in the node's `value` member, and assign its `next` pointer to `nullptr`. Figure 17-5 illustrates the current state of the list and the new node.

Figure 17-5

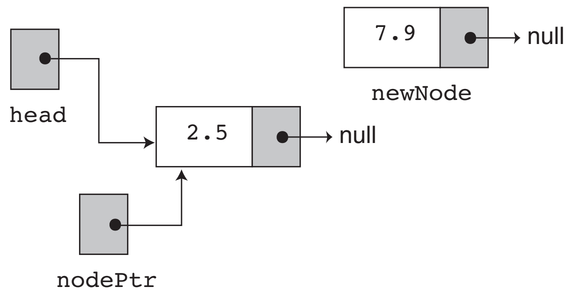


Because `head` no longer points to `nullptr`, the `else` part of the `if` statement executes:

```
else    // Otherwise, insert newNode at end.
{
        // Initialize nodePtr to head of list.
        nodePtr = head;

        // Find the last node in the list.
        while (nodePtr->next)
           nodePtr = nodePtr->next;

        // Insert newNode as the last node.
        nodePtr->next = newNode;
}
```

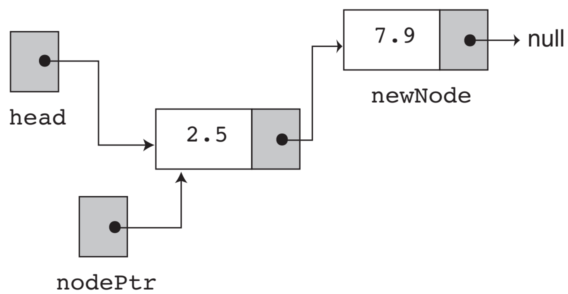
The first statement in the `else` block assigns the value in `head` to `nodePtr`. This causes `nodePtr` to point to the same node that `head` points to. This is illustrated in Figure 17-6.

Figure 17-6



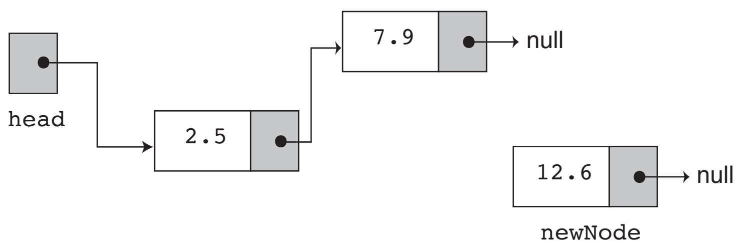
Look at the `next` member of the node that `nodePtr` points to. It is a null pointer, which means that `nodePtr->next` is also a null pointer. `nodePtr` is already at the end of the list, so the `while` loop immediately terminates. The last statement, `nodePtr->next = newNode;` causes `nodePtr->next` to point to the new node. This inserts `newNode` at the end of the list as shown in Figure 17-7.

Figure 17-7



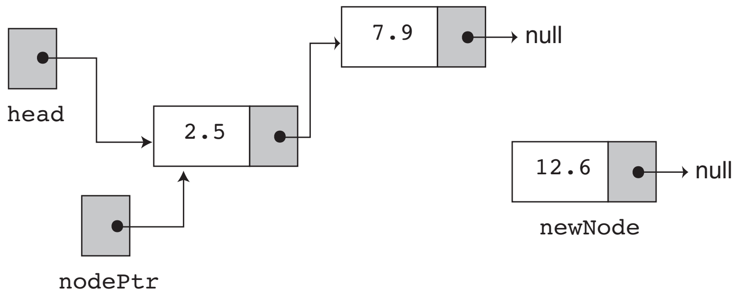
The third time `appendNode` is called, in line 15, 12.6 is passed as the argument. Once again, the first three statements create a node with the argument stored in the `value` member. This is shown in Figure 17-8.

Figure 17-8



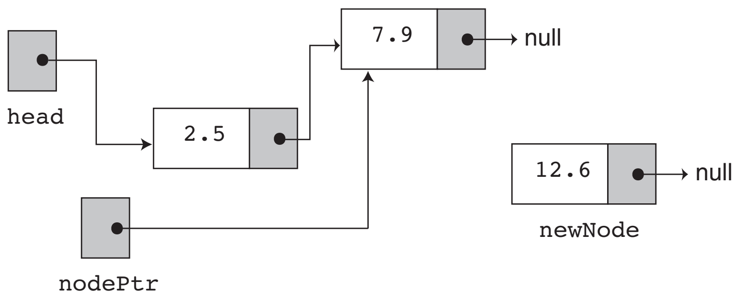
Next, the `else` part of the `if` statement executes. As before, `nodePtr` is made to point to the same node as `head`, as shown in Figure 17-9.

Figure 17-9



Because `nodePtr->next` is not a null pointer, the `while` loop will execute. After its first iteration, `nodePtr` will point to the second node in the list. This is shown in Figure 17-10.

Figure 17-10



The `while` loop's conditional test will fail after the first iteration because `nodePtr->next` is now a null pointer. The last statement, `nodePtr->next = newNode;` causes `nodePtr->next` to point to the new node. This inserts `newNode` at the end of the list as shown in Figure 17-11.

Figure 17-11

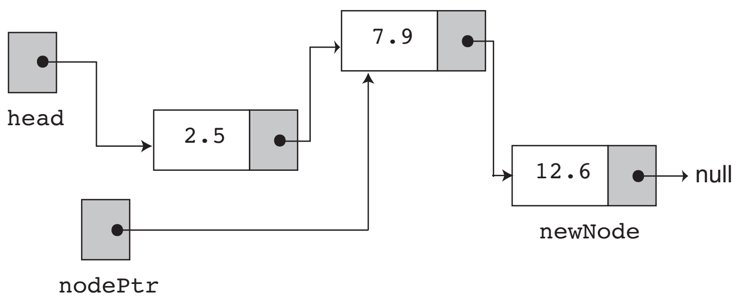


Figure 17-11 depicts the final state of the linked list.

Traversing a Linked List

The `appendNode` function demonstrated in the previous section contains a `while` loop that traverses, or travels through the linked list. In this section we will demonstrate the `displayList` member function that traverses the list, displaying the `value` member of each node. The following pseudocode represents the algorithm.

```
Assign List head to node pointer.
While node pointer is not null
    Display the value member of the node pointed to by node pointer.
    Assign node pointer to its own next member.
End While.
```

The function is shown here:

```cpp
void NumberList::displayList() const
{
    ListNode *nodePtr; // To move through the list

    // Position nodePtr at the head of the list.
    nodePtr = head;

    // While nodePtr points to a node, traverse
    // the list.
    while (nodePtr)
    {
        // Display the value in this node.
        cout << nodePtr->value << endl;

        // Move to the next node.
        nodePtr = nodePtr->next;
    }
}
```

Program 17-2, a modification of Program 17-1, demonstrates the function.

Program 17-2

```cpp
// This program demonstrates the displayList member function.
#include <iostream>
#include "NumberList.h"
using namespace std;

int main()
{
    // Define a NumberList object.
    NumberList list;

    // Append some values to the list.
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);

    // Display the values in the list.
    list.displayList();
    return 0;
}
```

Program Output

```
2.5
7.9
12.6
```

Usually, when an operation is to be performed on some or all the nodes in a linked list, a traversal algorithm is used. You will see variations of this algorithm throughout this chapter.

Inserting a Node

Appending a node is a straightforward procedure. Inserting a node in the middle of a list, however, is more involved. For example, suppose the values in a list are sorted and you wish all new values to be inserted in their proper position. This will preserve the order of the list. Using the `ListNode` structure again, the following pseudocode shows an algorithm for finding a new node's proper position in the list and inserting it there. The algorithm assumes the nodes in the list are already in order.

VideoNote
Inserting a Node in a Linked List

```
Create a new node.
Store data in the new node.
If there are no nodes in the list
     Make the new node the first node.
Else
     Find the first node whose value is greater than or equal to the new
         value, or the end of the list (whichever is first).
     Insert the new node before the found node, or at the end of the list
         if no such node was found.
End If.
```

Notice that the new algorithm finds the first node whose value is greater than or equal to the new value. The new node is then inserted before the found node. This will require the use of two node pointers during the traversal: one to point to the node being inspected and another to point to the previous node. The code for the traversal algorithm is as follows. (As before, `num` holds the value being inserted into the list.)

```
// Position nodePtr at the head of list.
nodePtr = head;

// Initialize previousNode to nullptr.
previousNode = nullptr;

// Skip all nodes whose value is less than num.
while (nodePtr != nullptr && nodePtr->value < num)
{
     previousNode = nodePtr;
     nodePtr = nodePtr->next;
}
```

This code segment uses the `ListNode` pointers `nodePtr` and `previousNode`. `previousNode` always points to the node before the one pointed to by `nodePtr`. The entire `insertNode` function is shown here:

```
void NumberList::insertNode(double num)
{
    ListNode *newNode;                  // A new node
    ListNode *nodePtr;                  // To traverse the list
    ListNode *previousNode = nullptr; // The previous node

    // Allocate a new node and store num there.
    newNode = new ListNode;
    newNode->value = num;

    // If there are no nodes in the list
    // make newNode the first node
    if (!head)
    {
        head = newNode;
        newNode->next = nullptr;
    }
    else // Otherwise, insert newNode
    {
        // Position nodePtr at the head of list.
        nodePtr = head;

        // Initialize previousNode to nullptr.
        previousNode = nullptr;

        // Skip all nodes whose value is less than num.
        while (nodePtr != nullptr && nodePtr->value < num)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }

        // If the new node is to be the 1st in the list,
        // insert it before all other nodes.
        if (previousNode == nullptr)
        {
            head = newNode;
            newNode->next = nodePtr;
        }
        else // Otherwise insert after the previous node.
        {
            previousNode->next = newNode;
            newNode->next = nodePtr;
        }
    }
}
```

Program 17-3 is a modification of Program 17-2. It uses the `insertNode` member function to insert a value in its ordered position in the list.

Program 17-3

```
// This program demonstrates the insertNode member function.
#include <iostream>
#include "NumberList.h"
using namespace std;

int main()
{
    // Define a NumberList object.
    NumberList list;

    // Build the list with some values.
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);

    // Insert a node in the middle of the list.
    list.insertNode(10.5);

    // Display the list
    list.displayList();
    return 0;
}
```

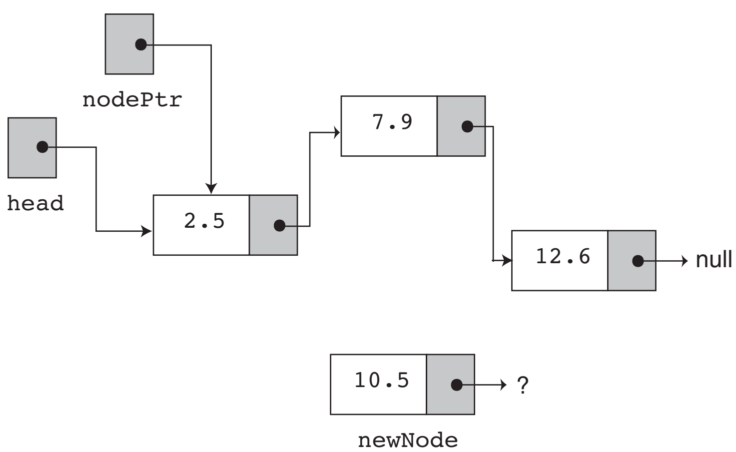
Program Output

```
2.5
7.9
10.5
12.6
```

Like Program 17-2, Program 17-3 calls the `appendNode` function three times to build the list with the values 2.5, 7.9, and 12.6. The `insertNode` function is then called, with the argument 10.5.

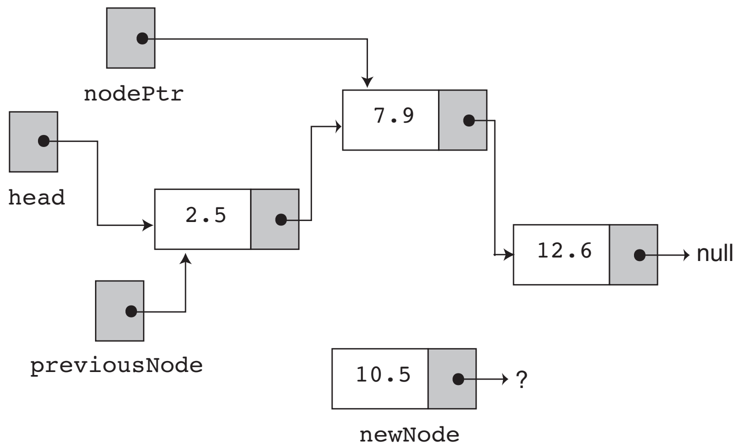
In `insertNode`, a new node is created, and the function argument is copied to its `value` member. Because the list already has nodes stored in it, the `else` part of the `if` statement will execute. It begins by assigning `nodePtr` to `head`. Figure 17-12 illustrates the state of the list at this point.

Figure 17-12



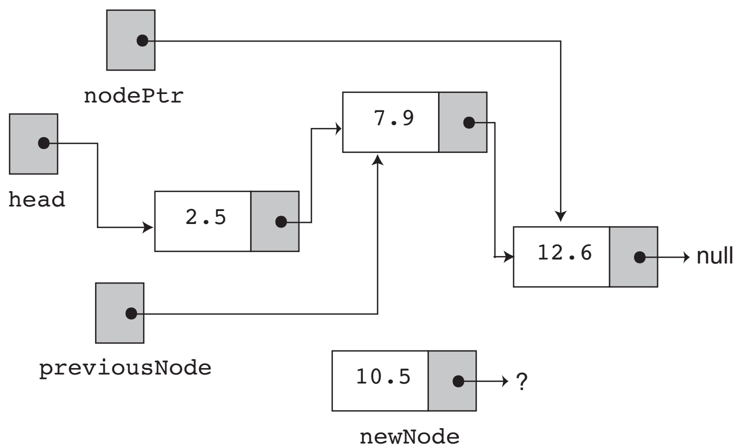
Because `nodePtr` is not a null pointer and `nodePtr->value` is less than `num`, the `while` loop will iterate. During the iteration, `previousNode` will be made to point to the node that `nodePtr` is pointing to. `nodePtr` will then be advanced to point to the next node. This is shown in Figure 17-13.

Figure 17-13



Once again, the loop performs its test. Because `nodePtr` is not a null pointer and `nodePtr->value` is less than `num`, the loop will iterate a second time. During the second iteration, both `previousNode` and `nodePtr` are advanced by one node in the list. This is shown in Figure 17-14.

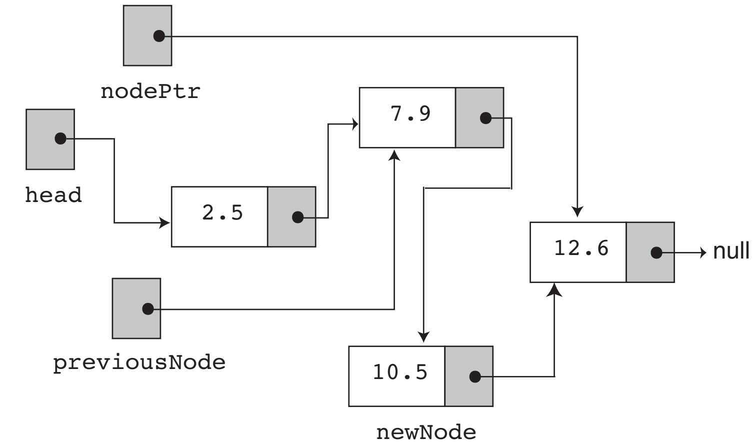
Figure 17-14



This time, the loop's test will fail because `nodePtr->value` is not less than `num`. The statements after the loop will execute, which cause `previousNode->next` to point to `newNode`, and `newNode->next` to point to `nodePtr`. This is illustrated in Figure 17-15.

This leaves the list in its final state. If you follow the links, from the `head` pointer to the null pointer, you will see that the nodes are stored in the order of their `value` members.

Figure 17-15



Checkpoint

17.5 What is the difference between appending a node to a list and inserting a node into a list?

17.6 Which is easier to code: appending or inserting?

17.7 Why does the `insertNode` function shown in this section use a `previousNode` pointer?

Deleting a Node

Deleting a node from a linked list requires two steps:

1. Remove the node from the list without breaking the links created by the `next` pointers.
2. Delete the node from memory.

VideoNote
Deleting a Node from a Linked List

The `deleteNode` member function searches for a node containing a particular value and deletes it from the list. It uses an algorithm similar to the `insertNode` function. Two node pointers, `nodePtr` and `previousNode`, are used to traverse the list. `previousNode` always points to the node whose position is just before the one pointed to by `nodePtr`. When `nodePtr` points to the node that is to be deleted, `previousNode->next` is made to point to `nodePtr->next`. This removes the node pointed to by `nodePtr` from the list. The final step performed by this function is to free the memory used by the node with the `delete` operator. The entire function is shown below.

```
void NumberList::deleteNode(double num)
{
    ListNode *nodePtr;       // To traverse the list
    ListNode *previousNode; // To point to the previous node

    // If the list is empty, do nothing.
    if (!head)
        return;

```

```
    // Determine if the first node is the one.
    if (head->value == num)
    {
        nodePtr = head->next;
        delete head;
        head = nodePtr;
    }
    else
    {
        // Initialize nodePtr to head of list
        nodePtr = head;

        // Skip all nodes whose value member is
        // not equal to num.
        while (nodePtr != nullptr && nodePtr->value != num)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }

        // If nodePtr is not at the end of the list,
        // link the previous node to the node after
        // nodePtr, then delete nodePtr.
        if (nodePtr)
        {
            previousNode->next = nodePtr->next;
            delete nodePtr;
        }
    }
}
```

Program 17-4 demonstrates the function by first building a list of three nodes and then deleting them one by one.

Program 17-4

```
// This program demonstrates the deleteNode member function.
#include <iostream>
#include "NumberList.h"
using namespace std;

int main()
{
    // Define a NumberList object.
    NumberList list;

    // Build the list with some values.
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);

```

```
    // Display the list.
    cout << "Here are the initial values:\n";
    list.displayList();
    cout << endl;

    // Delete the middle node.
    cout << "Now deleting the node in the middle.\n";
    list.deleteNode(7.9);

    // Display the list.
    cout << "Here are the nodes left.\n";
    list.displayList();
    cout << endl;

    // Delete the last node.
    cout << "Now deleting the last node.\n";
    list.deleteNode(12.6);

    // Display the list.
    cout << "Here are the nodes left.\n";
    list.displayList();
    cout << endl;

    // Delete the only node left in the list.
    cout << "Now deleting the only remaining node.\n";
    list.deleteNode(2.5);

    // Display the list.
    cout << "Here are the nodes left.\n";
    list.displayList();
    return 0;
}
```

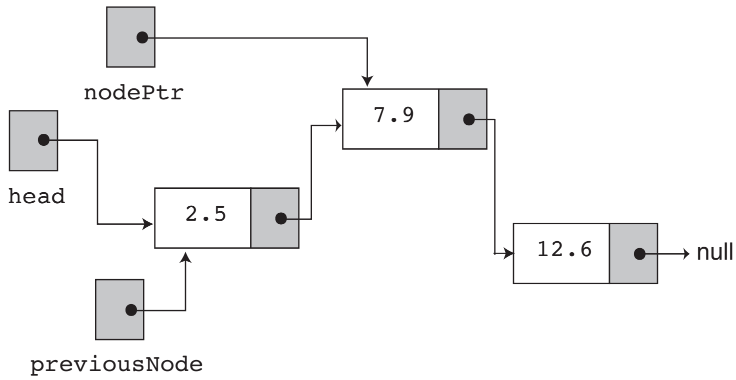
Program Output

```
Here are the initial values:
2.5
7.9
12.6

Now deleting the node in the middle.
Here are the nodes left.
2.5
12.6

Now deleting the last node.
Here are the nodes left.
2.5

Now deleting the only remaining node.
Here are the nodes left.
```

To illustrate how `deleteNode` works, we will step through the first call, which deletes the node containing 7.9 as its value. This node is in the middle of the list.

In the `deleteNode` function, look at the `else` part of the second `if` statement. This is lines 138 through 159. This is where the function will perform its action since the list is not empty, and the first node does not contain the value 7.9. Just like `insertNode`, this function uses `nodePtr` and `previousNode` to traverse the list. The `while` loop in lines 145 through 149 terminates when the value 7.9 is located. At this point, the list and the other pointers will be in the state depicted in Figure 17-16.

Figure 17-16

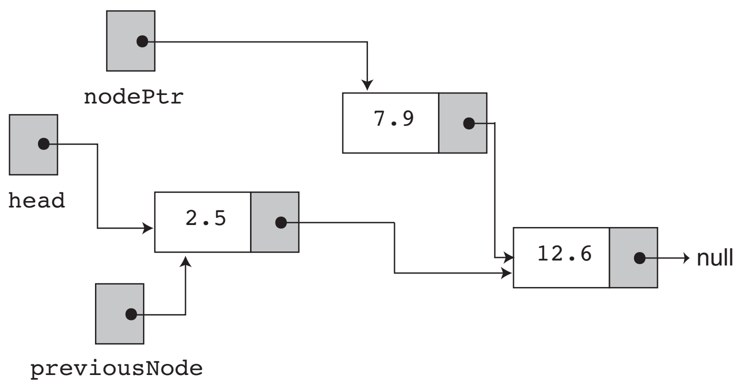


Next, the following statement in line 156 executes:

```
previousNode->next = nodePtr->next;
```

This statement causes the links in the list to bypass the node that `nodePtr` points to. Although the node still exists in memory, this removes it from the list, as illustrated in Figure 17-17.

Figure 17-17



The statement in line 157 uses the `delete` operator to complete the total deletion of the node.

Destroying the List

It's important for the class's destructor to release all the memory used by the list. It does so by stepping through the list, deleting one node at a time. The code is shown here:

```
NumberList::~NumberList()
{
    ListNode *nodePtr;    // To traverse the list
    ListNode *nextNode;   // To point to the next node

    // Position nodePtr at the head of the list.
    nodePtr = head;

    // While nodePtr is not at the end of the list...
    while (nodePtr != nullptr)
    {
        // Save a pointer to the next node.
        nextNode = nodePtr->next;

        // Delete the current node.
        delete nodePtr;

        // Position nodePtr at the next node.
        nodePtr = nextNode;
    }
}
```

Notice the use of `nextNode` instead of `previousNode`. The `nextNode` pointer is used to hold the position of the next node in the list, so that it will be available after the node pointed to by `nodePtr` is deleted.

Checkpoint

17.8 What are the two steps involved in deleting a node from a linked list?

17.9 When deleting a node, why can't you just use the `delete` operator to remove it from memory? Why must you take the steps you listed in response to Question 17.8?

17.10 In a program that uses several linked lists, what might eventually happen if the class destructor does not destroy its linked list?

17.3 A Linked List Template

CONCEPT: A template can be easily created to store linked lists of any type.

The limitation of the `NumberList` class is that it can only hold `double` values. The class can easily be converted to a template that will accept any data type, as shown in the following code. (This file is stored in the Student Source Code Folder `Chapter 17\LinkedList Template Version 1`.)

Contents of `LinkedList.h` (Version 1)

```
// A class template for holding a linked list.
#ifndef LINKEDLIST_H
#define LINKEDLIST_H
#include <iostream>     // For cout
using namespace std;

```

```
template <class T>
class LinkedList
{
private:
    // Declare a structure for the list.
    struct ListNode
    {
        T value;                 // The value in this node
        struct ListNode *next;  // To point to the next node
    };

    ListNode *head;   // List head pointer

public:
    // Constructor
    NumberList()
        { head = nullptr; }

    // Destructor
    ~NumberList();

    // Linked list operations
    void appendNode(T);
    void insertNode(T);
    void deleteNode(T);
    void displayList() const;
};


//**************************************************
// appendNode appends a node containing the value *
// passed into newValue, to the end of the list.  *
//**************************************************

template <class T>
void LinkedList<T>::appendNode(T newValue)
{
    ListNode *newNode;  // To point to a new node
    ListNode *nodePtr;  // To move through the list

    // Allocate a new node and store num there.
    newNode = new ListNode;
    newNode->value = num;
    newNode->next = nullptr;

    // If there are no nodes in the list
    // make newNode the first node.
    if (!head)
        head = newNode;
    else  // Otherwise, insert newNode at end.
    {
        // Initialize nodePtr to head of list.
        nodePtr = head;

```

```
        // Find the last node in the list.
        while (nodePtr->next)
            nodePtr = nodePtr->next;

        // Insert newNode as the last node.
        nodePtr->next = newNode;
    }
}

//**************************************************
// displayList shows the value                     *
// stored in each node of the linked list          *
// pointed to by head.                             *
//**************************************************

template <class T>
void LinkedList<T>::displayList()
{
    ListNode *nodePtr; // To move through the list

    // Position nodePtr at the head of the list.
    nodePtr = head;

    // While nodePtr points to a node, traverse
    // the list.
    while (nodePtr)
    {
        // Display the value in this node.
        cout << nodePtr->value << endl;

        // Move to the next node.
        nodePtr = nodePtr->next;
    }
}

//**************************************************
// The insertNode function inserts a node with     *
// newValue copied to its value member.            *
//**************************************************

template <class T>
void LinkedList<T>::insertNode(T newValue)
{
    ListNode *newNode;                 // A new node
    ListNode *nodePtr;                 // To traverse the list
    ListNode *previousNode = nullptr; // The previous node

    // Allocate a new node and store num there.
    newNode = new ListNode;
    newNode->value = num;

    // If there are no nodes in the list
    // make newNode the first node.
```

```
    if (!head)
    {
        head = newNode;
        newNode->next = nullptr;
    }
    else // Otherwise, insert newNode.
    {
        // Position nodePtr at the head of list.
        nodePtr = head;

        // Initialize previousNode to nullptr.
        previousNode = nullptr;

        // Skip all nodes whose value is less than num.
        while (nodePtr != nullptr && nodePtr->value < num)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }

        // If the new node is to be the 1st in the list,
        // insert it before all other nodes.
        if (previousNode == nullptr)
        {
            head = newNode;
            newNode->next = nodePtr;
        }
        else // Otherwise insert after the previous node.
        {
            previousNode->next = newNode;
            newNode->next = nodePtr;
        }
    }
}

//*****************************************************
// The deleteNode function searches for a node        *
// with searchValue as its value. The node, if found, *
// is deleted from the list and from memory.          *
//*****************************************************

template <class T>
void LinkedList<T>::deleteNode(T searchValue)
{
    ListNode *nodePtr;       // To traverse the list
    ListNode *previousNode;  // To point to the previous node

    // If the list is empty, do nothing.
    if (!head)
        return;

    // Determine if the first node is the one.
    if (head->value == num)
```

```
    {
        nodePtr = head->next;
        delete head;
        head = nodePtr;
    }
    else
    {
        // Initialize nodePtr to head of list.
        nodePtr = head;

        // Skip all nodes whose value member is
        // not equal to num.
        while (nodePtr != nullptr && nodePtr->value != num)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }

        // If nodePtr is not at the end of the list,
        // link the previous node to the node after
        // nodePtr, then delete nodePtr.
        if (nodePtr)
        {
            previousNode->next = nodePtr->next;
            delete nodePtr;
        }
    }
}

//**************************************************
// Destructor                                      *
// This function deletes every node in the list.   *
//**************************************************

template <class T>
LinkedList<T>::~LinkedList()
{
    ListNode *nodePtr;   // To traverse the list
    ListNode *nextNode;  // To point to the next node

    // Position nodePtr at the head of the list.
    nodePtr = head;

    // While nodePtr is not at the end of the list...
    while (nodePtr != nullptr)
    {
        // Save a pointer to the next node.
        nextNode = nodePtr->next;

        // Delete the current node.
        delete nodePtr;

```

```
        // Position nodePtr at the next node.
        nodePtr = nextNode;
    }
}
#endif
```

Note that the template uses the ==, !=, and < relational operators to compare node values, and it uses the << operator with `cout` to display node values. Any type passed to the template must support these operators.

Now let's see how the template can be used to create a list of objects. Recall the `FeetInches` class that was introduced in Chapter 14. That class overloaded numerous operators, including ==, <, and <<. In the `Chapter 17\LinkedList Template Version 1` folder we have included a modified version of the `FeetInches` class that also overloads the `!=` operator. Program 17-5 is stored in that same folder. This program uses the `LinkedList` template to create a linked list of `FeetInches` objects.

Program 17-5

```
// This program demonstrates the linked list template.
#include <iostream>
#include "LinkedList.h"
#include "FeetInches.h"
using namespace std;

int main()
{
    // Define a LinkedList object.
    LinkedList<FeetInches> list;

    // Define some FeetInches objects.
    FeetInches distance1(5, 4); // 5 feet 4 inches
    FeetInches distance2(6, 8); // 6 feet 8 inches
    FeetInches distance3(8, 9); // 8 feet 9 inches

    // Store the FeetInches objects in the list.
    list.appendNode(distance1); // 5 feet 4 inches
    list.appendNode(distance2); // 6 feet 8 inches
    list.appendNode(distance3); // 8 feet 9 inches

    // Display the values in the list.
    cout << "Here are the initial values:\n";
    list.displayList();
    cout << endl;

    // Insert another FeetInches object.
    cout << "Now inserting the value 7 feet 2 inches.\n";
    FeetInches distance4(7, 2);
    list.insertNode(distance4);

    // Display the values in the list.
    cout << "Here are the nodes now.\n";
    list.displayList();
    cout << endl;
```

```

    // Delete the last node.
    cout << "Now deleting the last node.\n";
    FeetInches distance5(8, 9);
    list.deleteNode(distance5);

    // Display the values in the list.
    cout << "Here are the nodes left.\n";
    list.displayList();
    return 0;
}
```

Program Output

```
Here are the initial values:
5 feet, 4 inches
6 feet, 8 inches
8 feet, 9 inches

Now inserting the value 7 feet 2 inches.
Here are the nodes now.
5 feet, 4 inches
6 feet, 8 inches
7 feet, 2 inches
8 feet, 9 inches

Now deleting the last node.
Here are the nodes left.
5 feet, 4 inches
6 feet, 8 inches
7 feet, 2 inches
```

Using a Class Node Type

In the `LinkedList` class template, the following structure was used to create a data type for the linked list node.

```
struct ListNode
{
    T value;
    struct ListNode *next;
};
```

Another approach is to use a separate class template to create a data type for the node. Then, the class constructor can be used to store an item in the `value` member and set the `next` pointer to `nullptr`. Here is an example:

```
template <class T>
class ListNode
{
public:
    T value;              // Node value
    ListNode<T> *next;    // Pointer to the next node
```

```
        // Constructor
        ListNode (T nodeValue)
            { value = nodeValue;
              next = nullptr; }
};
```

The `LinkedList` class template can then be written as the following:

```
template <class T>
class LinkedList
{
private:
    ListNode<T> *head;  // List head pointer
public:
    // Constructor
    LinkedList()
        { head = nullptr; }

    // Destructor
    ~LinkedList();

    // Linked list operations
    void appendNode(T);
    void insertNode(T);
    void deleteNode(T);
    void displayList() const;
};
```

Because the `ListNode` class constructor assigns a value to the `value` member and sets the `next` pointer to `nullptr`, some of the code in the `LinkedList` class can be simplified. For example, the following code appears in the previous version of the `LinkedList` class template's `appendNode` function:

```
newNode = new ListNode;
newNode->value = newValue;
newNode->next = nullptr;
```

By using the `ListNode` class template with its constructor, these three lines of code can be reduced to one:

```
newNode = new ListNode<T>(newValue);
```

(This file is stored in the Student Source Code Folder `Chapter 17\LinkedList Template Version 2`.)

Contents of `LinkedList.h` (Version 2)

```
// A class template for holding a linked list.
// The node type is also a class template.
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

//*********************************************
// The ListNode class creates a type used to *
// store a node of the linked list.          *
//*********************************************

template <class T>
```

```
class ListNode
{
public:
    T value;           // Node value
    ListNode<T> *next; // Pointer to the next node

    // Constructor
    ListNode (T nodeValue)
        { value = nodeValue;
          next = nullptr; }
};

//*********************************************
// LinkedList class                           *
//*********************************************

template <class T>
class LinkedList
{
private:
    ListNode<T> *head;  // List head pointer

public:
    // Constructor
    LinkedList()
        { head = nullptr; }

    // Destructor
    ~LinkedList();

    // Linked list operations
    void appendNode(T);
    void insertNode(T);
    void deleteNode(T);
    void displayList() const;
};


//**************************************************
// appendNode appends a node containing the value *
// passed into newValue, to the end of the list.  *
//**************************************************

template <class T>
void LinkedList<T>::appendNode(T newValue)
{
    ListNode<T> *newNode;  // To point to a new node
    ListNode<T> *nodePtr;  // To move through the list

    // Allocate a new node and store newValue there.
    newNode = new ListNode<T>(newValue);

    // If there are no nodes in the list
    // make newNode the first node.
```

```
    if (!head)
        head = newNode;
    else  // Otherwise, insert newNode at end.
    {
        // Initialize nodePtr to head of list.
        nodePtr = head;

        // Find the last node in the list.
        while (nodePtr->next)
            nodePtr = nodePtr->next;

        // Insert newNode as the last node.
        nodePtr->next = newNode;
    }
}

//**************************************************
// displayList shows the value stored in each node *
// of the linked list pointed to by head.          *
//**************************************************

template <class T>
void LinkedList<T>::displayList() const
{
    ListNode<T> *nodePtr; // To move through the list

    // Position nodePtr at the head of the list.
    nodePtr = head;

    // While nodePtr points to a node, traverse
    // the list.
    while (nodePtr)
    {
        // Display the value in this node.
        cout << nodePtr->value << endl;

        // Move to the next node.
        nodePtr = nodePtr->next;
    }
}

//*************************************************
// The insertNode function inserts a node with    *
// newValue copied to its value member.           *
//*************************************************

template <class T>
void LinkedList<T>::insertNode(T newValue)
{
    ListNode<T> *newNode;               // A new node
    ListNode<T> *nodePtr;               // To traverse the list
    ListNode<T> *previousNode = nullptr; // The previous node

```

```
    // Allocate a new node and store newValue there.
    newNode = new ListNode<T>(newValue);

    // If there are no nodes in the list
    // make newNode the first node.
    if (!head)
    {
        head = newNode;
        newNode->next = nullptr;
    }
    else  // Otherwise, insert newNode.
    {
        // Position nodePtr at the head of list.
        nodePtr = head;

        // Initialize previousNode to nullptr.
        previousNode = nullptr;

        // Skip all nodes whose value is less than newValue.
        while (nodePtr != nullptr && nodePtr->value < newValue)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }

        // If the new node is to be the 1st in the list,
        // insert it before all other nodes.
        if (previousNode == nullptr)
        {
            head = newNode;
            newNode->next = nodePtr;
        }
        else // Otherwise insert after the previous node.
        {
            previousNode->next = newNode;
            newNode->next = nodePtr;
        }
    }
}

//*****************************************************
// The deleteNode function searches for a node        *
// with searchValue as its value. The node, if found, *
// is deleted from the list and from memory.          *
//*****************************************************

template <class T>
void LinkedList<T>::deleteNode(T searchValue)
{
    ListNode<T> *nodePtr;      // To traverse the list
    ListNode<T> *previousNode; // To point to the previous node

```

```
    // If the list is empty, do nothing.
    if (!head)
        return;

    // Determine if the first node is the one.
    if (head->value == searchValue)
    {
        nodePtr = head->next;
        delete head;
        head = nodePtr;
    }
    else
    {
        // Initialize nodePtr to head of list
        nodePtr = head;

        // Skip all nodes whose value member is
        // not equal to num.
        while (nodePtr != nullptr && nodePtr->value != searchValue)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }

        // If nodePtr is not at the end of the list,
        // link the previous node to the node after
        // nodePtr, then delete nodePtr.
        if (nodePtr)
        {
            previousNode->next = nodePtr->next;
            delete nodePtr;
        }
    }
}

//**************************************************
// Destructor                                      *
// This function deletes every node in the list.   *
//**************************************************

template <class T>
LinkedList<T>::~LinkedList()
{
    ListNode<T> *nodePtr;  // To traverse the list
    ListNode<T> *nextNode; // To point to the next node

    // Position nodePtr at the head of the list.
    nodePtr = head;

    // While nodePtr is not at the end of the list...
    while (nodePtr != nullptr)
    {
        // Save a pointer to the next node.
        nextNode = nodePtr->next;
```

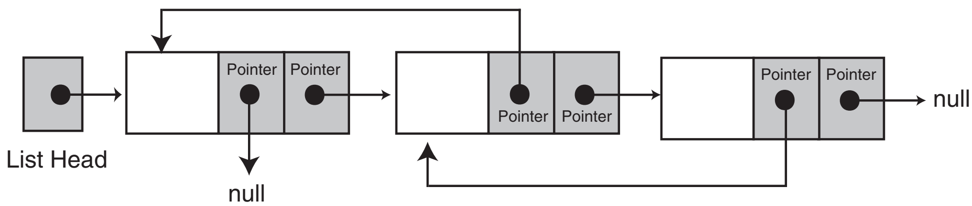
```
             // Delete the current node.
             delete nodePtr;

             // Position nodePtr at the next node.
             nodePtr = nextNode;
        }
}
#endif
```

17.4 Variations of the Linked List

CONCEPT: There are many ways to link dynamically allocated data structures together. Two variations of the linked list are the doubly linked list and the circular linked list.

The linked list examples that we have discussed are considered *singly linked lists*. Each node is linked to a single other node. A variation of this is the *doubly linked list*. In this type of list, each node points not only to the next node, but also to the previous one. This is illustrated in Figure 17-18.

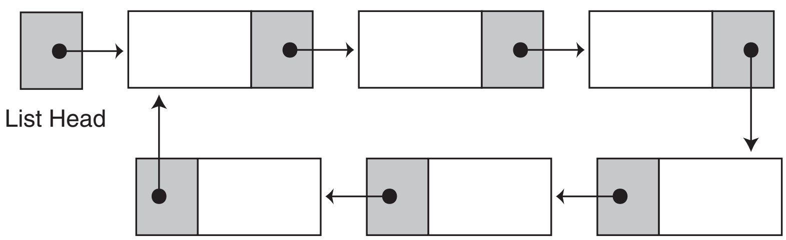
Figure 17-18



In Figure 17-18, the last node and the first node in the list have pointers to the null address. When the program traverses the list it knows when it has reached either end.

Another variation is the *circularly linked list*. The last node in this type of list points to the first, as shown in Figure 17-19.

Figure 17-19



17.5 The STL `list` Container

CONCEPT: The Standard Template Library provides a linked list container.

The `list` container, found in the Standard Template Library, is a template version of a doubly linked list. STL `lists` can insert elements or add elements to their front quicker than `vectors` can because `lists` do not have to shift the other elements. `lists` are also efficient at adding elements at their back because they have a built-in pointer to the last element in the `list` (no traversal required).

Table 17-1 describes some of the `list` member functions.

Table 17-1

Member Function	Examples and Description
`back`	`cout << list.back() << endl;` The `back` member function returns a reference to the last element in the list.
`empty`	`if (list.empty())` The `empty` member function returns `true` if the `list` is empty. If the `list` has elements, it returns `false`.
`end`	`iter = list.end();` `end` returns a bidirectional iterator to the end of the list.

`erase`	`list.erase(iter);`
	`list.erase(firstIter, lastIter)`
	The first example causes the `list` element pointed to by the iterator `iter` to be removed. The second example causes all of the `list` elements from `firstIter` to `lastIter` to be removed.

`front`	`cout << list.front() << endl;` `front` returns a reference to the first element of the `list`.
`insert`	`list.insert(iter, x)` The `insert` member function inserts an element into the `list`. This example inserts an element with the value `x`, just before the element pointed to by `iter`.
`merge`	`list1.merge(list2);` `merge` inserts all the items in `list2` into `list1`. `list1` is expanded to accommodate the new elements plus any elements already stored in `list1`. `merge` expects both lists to be sorted. When `list2` is inserted into `list1`, the elements are inserted into their correct position, so the resulting list is also sorted.
`pop_back`	`list.pop_back();` `pop_back` removes the last element of the list.
`pop_front`	`list.pop_front();` `pop_front` removes the first element of the list.
`push_back`	`list.push_back(x);` `push_back` inserts an element with value `x` at the end of the list.
`push_front`	`list.push_front(x);` `push_front` inserts an element with value `x` at the beginning of the list.

Table 17-1 *(continued)*

Member Function	Examples and Description
`reverse`	`list.reverse();` `reverse` reverses the order in which the elements appear in the list.
`size`	Returns the number of elements in the `list`.
`swap`	`list1.swap(list2)` The swap member function swaps the elements stored in two `lists`. For example, assuming `list1` and `list2` are `lists`, this statement will exchange the values in the two lists.
`unique`	`list.unique();` `unique` removes any element that has the same value as the element before it.

Program 17-6 demonstrates some simple operations with the list container.

Program 17-6

```
// This program demonstrates the STL list container.
#include <iostream>
#include <list>       // Include the list header.
using namespace std;

int main()
{
     // Define a list object.
     list<int> myList;

     // Define an iterator for the list.
     list<int>::iterator iter;

     // Add values to the list.
     for (int x = 0; x < 100; x += 10)
          myList.push_back(x);

     // Display the values.
     for (iter = myList.begin(); iter != myList.end(); iter++)
          cout << *iter << " ";
     cout << endl;

     // Now reverse the order of the elements.
     myList.reverse();

     // Display the values again.
     for (iter = myList.begin(); iter != myList.end(); iter++)
          cout << *iter << " ";
     cout << endl;
     return 0;
}
```

Program Output

```
0 10 20 30 40 50 60 70 80 90
90 80 70 60 50 40 30 20 10 0
```

Review Questions and Exercises

Short Answer

1. What are some of the advantages that linked lists have over arrays?
2. What advantage does a linked list have over the STL `vector`?
3. What is a list head?
4. What is a self-referential data structure?
5. How is the end of a linked list usually signified?
6. Name five basic linked list operations.
7. What is the difference between appending a node and inserting a node?
8. What does "traversing the list" mean?
9. What are the two steps required to delete a node from a linked list?
10. What is the advantage of using a template to implement a linked list?
11. What is a singly linked list? What is a doubly linked list? What is a circularly linked list?
12. What type of linked list is the STL `list` container?

Fill-in-the-Blank

13. The __________ points to the first node in a linked list.
14. A data structure that points to an object of the same type as itself is known as a(n) __________ data structure.
15. After creating a linked list's head pointer, you should make sure it points to __________ before using it in any operations.
16. __________ a node means adding it to the end of a list.
17. __________ a node means adding it to a list, but not necessarily to the end.
18. __________ a list means traveling through the list.
19. In a(n) __________ list, the last node has a pointer to the first node.
20. In a(n) __________ list, each node has a pointer to the one before it and the one after it.

Algorithm Workbench

21. Consider the following code:

```
struct ListNode
{
     int value;
     struct ListNode *next;
};

ListNode *head; // List head pointer
```

Assume that a linked list has been created and `head` points to the first node. Write code that traverses the list displaying the contents of each node's `value` member.

22. Write code that destroys the linked list described in Question 21.

23. Write code that defines an STL `list` container for holding `float` values.
24. Write code that stores the values 12.7, 9.65, 8.72, and 4.69 in the `list` container you defined for Question 23.
25. Write code that reverses the order of the items you stored in the `list` container in Question 24.

True or False

26. T F The programmer must know in advance how many nodes will be needed in a linked list.
27. T F It is not necessary for each node in a linked list to have a self-referential pointer.
28. T F In physical memory, the nodes in a linked list may be scattered around.
29. T F When the head pointer points to `nullptr`, it signifies an empty list.
30. T F Linked lists are not superior to STL vectors.
31. T F Deleting a node in a linked list is a simple matter of using the `delete` operator to free the node's memory.
32. T F A class that builds a linked list should destroy the list in the class destructor.

Find the Error

Each of the following member functions has errors in the way it performs a linked list operation. Find as many mistakes as you can.

33.
```
void NumberList::appendNode(double num)
{
     ListNode *newNode, *nodePtr;
     // Allocate a new node & store num
     newNode = new listNode;
     newNode->value = num;

     // If there are no nodes in the list
     // make newNode the first node.
     if (!head)
          head = newNode;
     else          // Otherwise, insert newNode.
     {
          // Find the last node in the list.
          while (nodePtr->next)
               nodePtr = nodePtr->next;
          // Insert newNode as the last node.
          nodePtr->next = newNode;
     }
}
```

34.
```
void NumberList::deleteNode(double num)
{
     ListNode *nodePtr, *previousNode;
     // If the list is empty, do nothing.
     if (!head)
          return;
     // Determine if the first node is the one.
```

```
    if (head->value == num)
        delete head;
    else
    {
        // Initialize nodePtr to head of list.
        nodePtr = head;

        // Skip all nodes whose value member is
        // not equal to num.
        while (nodePtr->value != num)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }
        // Link the previous node to the node after
        // nodePtr, then delete nodePtr.
        previousNode->next = nodePtr->next;
        delete nodePtr;
    }
}
```

35.
```
NumberList::~NumberList()
{
    ListNode *nodePtr, *nextNode;
    nodePtr = head;
    while (nodePtr != nullptr)
    {
        nextNode = nodePtr->next;
        nodePtr->next = nullptr;
        nodePtr = nextNode;
    }
}
```

Programming Challenges

1. **Your Own Linked List**

   Design your own linked list class to hold a series of integers. The class should have member functions for appending, inserting, and deleting nodes. Don't forget to add a destructor that destroys the list. Demonstrate the class with a driver program.

2. **List Print**

   Modify the linked list class you created in Programming Challenge 1 to add a `print` member function. The function should display all the values in the linked list. Test the class by starting with an empty list, adding some elements, and then printing the resulting list out.

3. **List Copy Constructor**

   Modify your linked list class of Programming Challenges 1 and 2 to add a copy constructor. Test your class by making a list, making a copy of the list, and then displaying the values in the copy.

4. **List Reverse**

   Modify the linked list class you created in the previous programming challenges by adding a member function named `reverse` that rearranges the nodes in the list so that their order is reversed. Demonstrate the function in a simple driver program.

5. **List Search**

   Modify the linked list class you created in the previous programming challenges to include a member function named `search` that returns the position of a specific value in the linked list. The first node in the list is at position 0, the second node is at position 1, and so on. If x is not found on the list, the search should return −1. Test the new member function using an appropriate driver program.

6. **Member Insertion by Position**

   VideoNote **Solving the Member Insertion by Position Problem**

   Modify the list class you created in the previous programming challenges by adding a member function for inserting a new item at a specified position. A position of 0 means that the value will become the first item on the list, a position of 1 means that the value will become the second item on the list, and so on. A position equal to or greater than the length of the list means that the value is placed at the end of the list.

7. **Member Removal by Position**

   Modify the list class you created in the previous programming challenges by adding a member function for deleting a node at a specified position. A value of 0 for the position means that the first node in the list (the current head) is deleted. The function does nothing if the specified position is greater than or equal to the length of the list.

8. **List Template**

   Create a list class template based on the list class you created in the previous programming challenges.

9. **Rainfall Statistics Modification**

   Modify Programming Challenge 2 in Chapter 7 (Rainfall Statistics) to let the user decide how many months of data will be entered. Use a linked list instead of an array to hold the monthly data.

10. **Payroll Modification**

    Modify Programming Challenge 10 in Chapter 7 (Payroll) to use three linked lists instead of three arrays to hold the employee IDs, hours worked, and wages. When the program starts, it should ask the user to enter the employee IDs. There should be no limit on the number of IDs the user can enter.

11. **List Search**

    Modify the `LinkedList` template shown in this chapter to include a member function named `search`. The function should search the list for a specified value. If the value is found, it should return a number indicating its position in the list. (The first node is node 1, the second node is node 2, and so forth.) If the value is not found, the function should return 0. Demonstrate the function in a driver program.

12. Double Merge

Modify the `NumberList` class shown in this chapter to include a member function named `mergeArray`. The `mergeArray` function should take an array of `doubles` as its first argument and an integer as its second argument. (The second argument will specify the size of the array being passed into the first argument.)

The function should merge the values in the array into the linked list. The value in each element of the array should be inserted (not appended) into the linked list. When the values are inserted, they should be in numerical order. Demonstrate the function with a driver program. When you are satisfied with the function, incorporate it into the `LinkedList` template.

13. Rainfall Statistics Modification #2

Modify the program you wrote for Programming Challenge 9 so that it saves the data in the linked list to a file. Write a second program that reads the data from the file into a linked list and displays it on the screen.

14. Overloaded `[]` Operator

Modify the linked list class that you created in Programming Challenge 1 (or the `LinkedList` template presented in this chapter) by adding an overloaded `[]` operator function. This will give the linked list the ability to access nodes using a subscript, like an array. The subscript 0 will reference the first node in the list, the subscript 1 will reference the second node in the list, and so forth. The subscript of the last node will be the number of nodes minus 1. If an invalid subscript is used, the function should throw an exception.

15. `pop` and `push` Member Functions

The STL list container has member functions named `pop_back`, `pop_front`, `push_back`, and `push_front`, as described in Table 17-1. Modify the linked list class that you created in Programming Challenge 1 (or the `LinkedList` template presented in this chapter) by adding your own versions of these member functions.