```
    list2.deleteNode(num);                          //Line 25

    cout << "Line 26: After deleting "
         << num << ", list2: " << endl;             //Line 26
    list2.print();                                  //Line 27
    cout << endl;                                   //Line 28

    return 0;                                       //Line 29
}                                                   //Line 30
```

Sample Run: In this sample run, the user input is shaded:

```
Line 7: Enter numbers ending with -999.
23 65 34 72 12 82 36 55 29 -999

Line 15: list1: 12 23 29 34 36 55 65 72 82
Line 19: list2: 12 23 29 34 36 55 65 72 82
Line 22: Enter the number to be deleted: 34

Line 26: After deleting 34, list2:
12 23 29 36 55 65 72 82
```

The preceding output is self-explanatory. The details are left as an exercise for you.

NOTE

Notice that the function `insert` does not check whether the item to be inserted is already in the list, that is, it does not check for duplicates. Programming Exercise 7 at the end of this chapter asks yu to revise the definition of the function `insert` so that before inserting the item it checks whether it is already in the list. If the item to be inserted is already in the list, the function outputs an appropriate error message. In other words, duplicates are not allowed.

Doubly Linked Lists

A doubly linked list is a linked list in which every node has a next pointer and a back pointer. In other words, every node contains the address of the next node (except the last node), and every node contains the address of the previous node (except the first node). (See Figure 5-27.)

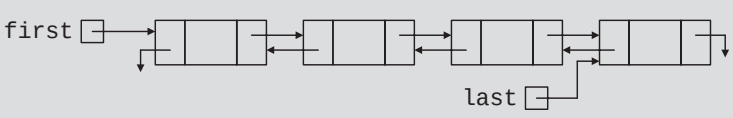

FIGURE 5-27 Doubly linked list

A doubly linked list can be traversed in either direction. That is, we can traverse the list starting at the first node or, if a pointer to the last node is given, we can traverse the list starting at the last node.

As before, the typical operations on a doubly linked list are as follows: Initialize the list, destroy the list, determine whether the list is empty, search the list for a given item, insert an item, delete an item, and so on. The following class defines a doubly linked list as an ADT and specifies the basic operations on a doubly linked list:

```
//*************************************************************
// Author: D.S. Malik
//
// This class specifies the members to implement the basic
// properties of an ordered doubly linked list.
//*************************************************************

  //Definition of the node
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *next;
    nodeType<Type> *back;
};

template <class Type>
class doublyLinkedList
{
public:
    const doublyLinkedList<Type>& operator=
                          (const doublyLinkedList<Type> &);
      //Overload the assignment operator.

    void initializeList();
      //Function to initialize the list to an empty state.
      //Postcondition: first = NULL; last = NULL; count = 0;

    bool isEmptyList() const;
      //Function to determine whether the list is empty.
      //Postcondition: Returns true if the list is empty,
      //               otherwise returns false.

    void destroy();
      //Function to delete all the nodes from the list.
      //Postcondition: first = NULL; last = NULL; count = 0;

    void print() const;
      //Function to output the info contained in each node.

    void reversePrint() const;
      //Function to output the info contained in each node
      //in reverse order.
```

```
    int length() const;
      //Function to return the number of nodes in the list.
      //Postcondition: The value of count is returned.

    Type front() const;
      //Function to return the first element of the list.
      //Precondition: The list must exist and must not be empty.
      //Postcondition: If the list is empty, the program terminates;
      //    otherwise, the first element of the list is returned.

    Type back() const;
      //Function to return the last element of the list.
      //Precondition: The list must exist and must not be empty.
      //Postcondition: If the list is empty, the program terminates;
      //    otherwise, the last element of the list is returned.

    bool search(const Type& searchItem) const;
      //Function to determine whether searchItem is in the list.
      //Postcondition: Returns true if searchItem is found in the
      //    list, otherwise returns false.

    void insert(const Type& insertItem);
      //Function to insert insertItem in the list.
      //Precondition: If the list is nonempty, it must be in order.
      //Postcondition: insertItem is inserted at the proper place
      //    in the list, first points to the first node, last points
      //    to the last node of the new list, and count is
      //    incremented by 1.

    void deleteNode(const Type& deleteItem);
      //Function to delete deleteItem from the list.
      //Postcondition: If found, the node containing deleteItem is
      //    deleted from the list; first points to the first node of
      //    the new list, last points to the last node of the new
      //    list, and count is decremented by 1; otherwise an
      //    appropriate message is printed.

    doublyLinkedList();
      //default constructor
      //Initializes the list to an empty state.
      //Postcondition: first = NULL; last = NULL; count = 0;

    doublyLinkedList(const doublyLinkedList<Type>& otherList);
      //copy constructor
    ~doublyLinkedList();
      //destructor
      //Postcondition: The list object is destroyed.

protected:
    int count;
    nodeType<Type> *first; //pointer to the first node
    nodeType<Type> *last;  //pointer to the last node
```

```
private:
    void copyList(const doublyLinkedList<Type>& otherList);
      //Function to make a copy of otherList.
      //Postcondition: A copy of otherList is created and assigned
      //    to this list.
};
```

We leave the UML class diagram of the `class doublyLinkedList` as an exercise for you, see Exercise 11 at the end of this chapter.

The functions to implement the operations of a doubly linked list are similar to the ones discussed earlier. Here, because every node has two pointers, `back` and `next`, some of the operations require the adjustment of two pointers in each node. For the insert and delete operations, because we can traverse the list in either direction, we use only one pointer to traverse the list. Let us call this pointer `current`. We can set the value of `trailCurrent` by using both the `current` pointer and the `back` pointer of the node pointed to by `current`. We give the definition of each function here, with four exceptions. Definitions of the functions `copyList`, the copy constructor, overloading the assignment operator, and the destructor are left as exercises for you. (See Programming Exercise 10 at the end of this chapter.) Furthermore, the function `copyList` is used only to implement the copy constructor and overload the assignment operator.

Default Constructor

The default constructor initializes the doubly linked list to an empty state. It sets `first` and `last` to `NULL` and `count` to `0`.

```
template <class Type>
doublyLinkedList<Type>::doublyLinkedList()
{
    first= NULL;
    last = NULL;
    count = 0;
}
```

isEmptyList

This operation returns `true` if the list is empty; otherwise, it returns `false`. The list is empty if the pointer `first` is `NULL`.

```
template <class Type>
bool doublyLinkedList<Type>::isEmptyList() const
{
    return (first == NULL);
}
```

Destroy the List

This operation deletes all the nodes in the list, leaving the list in an empty state. We traverse the list starting at the first node and then delete each node. Furthermore, `count` is set to `0`.

```
template <class Type>
void doublyLinkedList<Type>::destroy()
{
    nodeType<Type> *temp; //pointer to delete the node

    while (first != NULL)
    {
        temp = first;
        first = first->next;
        delete temp;
    }

    last = NULL;
    count = 0;
}
```

Initialize the List

This operation reinitializes the doubly linked list to an empty state. This task can be done by using the operation `destroy`. The definition of the function `initializeList` is as follows:

```
template <class Type>
void doublyLinkedList<Type>::initializeList()
{
    destroy();
}
```

Length of the List

The length of a linked list (that is, how many nodes are in the list) is stored in the variable `count`. Therefore, this function returns the value of this variable.

```
template <class Type>
int doublyLinkedList<Type>::length() const
{
    return count;
}
```

Print the List

The function `print` outputs the `info` contained in each node. We traverse the list starting from the first node.

```
template <class Type>
void doublyLinkedList<Type>::print() const
{
    nodeType<Type> *current; //pointer to traverse the list

    current = first;  //set current to point to the first node

    while (current != NULL)
```

```
    {
        cout << current->info << "  ";  //output info
        current = current->next;
    }//end while
}//end print
```

Reverse Print the List

This function outputs the `info` contained in each node in reverse order. We traverse the list in reverse order starting from the last node. Its definition is as follows:

```
template <class Type>
void doublyLinkedList<Type>::reversePrint() const
{
    nodeType<Type> *current; //pointer to traverse the list

    current = last;  //set current to point to the last node

    while (current != NULL)
    {
        cout << current->info << "  ";
        current = current->back;
    }//end while
}//end reversePrint
```

Search the List

The function `search` returns `true` if `searchItem` is found in the list; otherwise, it returns `false`. The search algorithm is exactly the same as the search algorithm for an ordered linked list.

```
template <class Type>
bool doublyLinkedList<Type>::search(const Type& searchItem) const
{
    bool found = false;
    nodeType<Type> *current; //pointer to traverse the list

    current = first;

    while (current != NULL && !found)
        if (current->info >= searchItem)
            found = true;
        else
            current = current->next;

    if (found)
       found = (current->info == searchItem); //test for equality

    return found;
}//end search
```

First and Last Elements

The function `front` returns the first element of the list and the function `back` returns the last element of the list. If the list is empty, both functions terminate the program. Their definitions are as follows:

```
template <class Type>
Type doublyLinkedList<Type>::front() const
{
    assert(first != NULL);

    return first->info;
}

template <class Type>
Type doublyLinkedList<Type>::back() const
{
    assert(last != NULL);

    return last->info;
}
```

INSERT A NODE

Because we are inserting an item in a doubly linked list, the insertion of a node in the list requires the adjustment of two pointers in certain nodes. As before, we find the place where the new item is supposed to be inserted, create the node, store the new item, and adjust the link fields of the new node and other particular nodes in the list. There are four cases:

Case 1: Insertion in an empty list

Case 2: Insertion at the beginning of a nonempty list

Case 3: Insertion at the end of a nonempty list

Case 4: Insertion somewhere in a nonempty list

Both cases 1 and 2 require us to change the value of the pointer `first`. Cases 3 and 4 are similar. After inserting an item, `count` is incremented by `1`. Next, we show case 4.

Consider the doubly linked list shown in Figure 5-28.

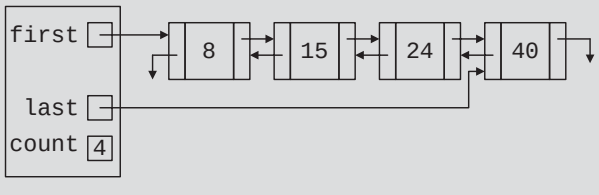

FIGURE 5-28 Doubly linked list before inserting 20

Suppose that 20 is to be inserted in the list. After inserting 20, the resulting list is as shown in Figure 5-29.

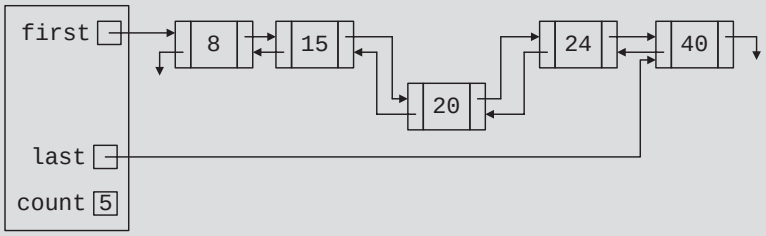

FIGURE 5-29 Doubly linked list after inserting 20

From Figure 5-29, it follows that the `next` pointer of node 15, the `back` pointer of node 24, and both the `next` and `back` pointers of node 20 need to be adjusted.

The definition of the function `insert` is as follows:

```
template <class Type>
void doublyLinkedList<Type>::insert(const Type& insertItem)
{
    nodeType<Type> *current;      //pointer to traverse the list
    nodeType<Type> *trailCurrent; //pointer just before current
    nodeType<Type> *newNode;      //pointer to create a node
    bool found;

    newNode = new nodeType<Type>; //create the node
    newNode->info = insertItem;  //store the new item in the node
    newNode->next = NULL;
    newNode->back = NULL;

    if (first == NULL) //if list is empty, newNode is the only node
    {
        first = newNode;
        last = newNode;
        count++;
    }
    else
    {
        found = false;
        current = first;

        while (current != NULL && !found) //search the list
            if (current->info >= insertItem)
                found = true;
            else
            {
                trailCurrent = current;
                current = current->next;
            }
```

```
        if (current == first) //insert newNode before first
        {
            first->back = newNode;
            newNode->next = first;
            first = newNode;
            count++;
        }
        else
        {
              //insert newNode between trailCurrent and current
            if (current != NULL)
            {
                trailCurrent->next = newNode;
                newNode->back = trailCurrent;
                newNode->next = current;
                current->back = newNode;
            }
            else
            {
                trailCurrent->next = newNode;
                newNode->back = trailCurrent;
                last = newNode;
            }

            count++;
        }//end else
    }//end else
}//end insert
```

DELETE A NODE

This operation deletes a given item (if found) from the doubly linked list. As before, we first search the list to see whether the item to be deleted is in the list. The search algorithm is the same as before. Similar to the `insert` operation, this operation (if the item to be deleted is in the list) requires the adjustment of two pointers in certain nodes. The delete operation has several cases:

Case 1: The list is empty.

Case 2: The item to be deleted is in the first node of the list, which would require us to change the value of the pointer `first`.

Case 3: The item to be deleted is somewhere in the list.

Case 4: The item to be deleted is not in the list.

After deleting a node, `count` is decremented by `1`. Let us demonstrate case 3. Consider the list shown in Figure 5-30.

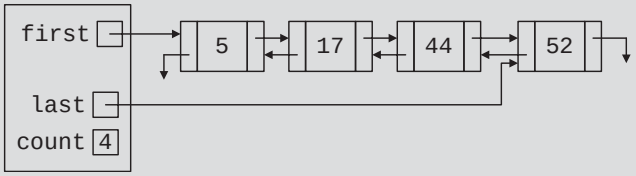

FIGURE 5-30 Doubly linked list before deleting `17`

Suppose that the item to be deleted is `17`. First, we search the list with two pointers and find the node with `info` `17`, and then adjust the link field of the affected nodes. (See Figure 5-31.)

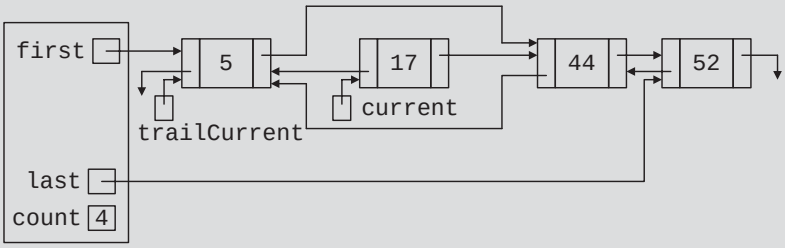

FIGURE 5-31 List after adjusting the links of the nodes before and after the node with `info 17`

Next, we delete the node pointed to by `current`. (See Figure 5-32.)

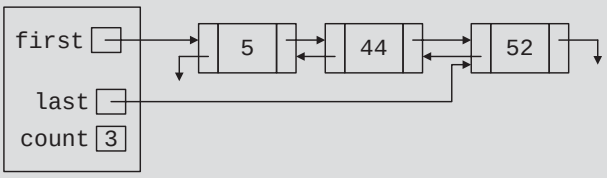

FIGURE 5-32 List after deleting the node with `info 17`

The definition of the function `deleteNode` is as follows:

```
template <class Type>
void doublyLinkedList<Type>::deleteNode(const Type& deleteItem)
{
    nodeType<Type> *current; //pointer to traverse the list
    nodeType<Type> *trailCurrent; //pointer just before current
```

```
    bool found;

    if (first == NULL)
        cout << "Cannot delete from an empty list." << endl;
    else if (first->info == deleteItem) //node to be deleted is
                                        //the first node
    {
        current = first;
        first = first->next;

        if (first != NULL)
            first->back = NULL;
        else
            last = NULL;

        count--;

        delete current;
    }
    else
    {
        found = false;
        current = first;

        while (current != NULL && !found)  //search the list
            if (current->info >= deleteItem)
                found = true;
            else
                current = current->next;

        if (current == NULL)
            cout << "The item to be deleted is not in "
                 << "the list." << endl;
        else if (current->info == deleteItem) //check for equality
        {
            trailCurrent = current->back;
            trailCurrent->next = current->next;

            if (current->next != NULL)
                current->next->back = trailCurrent;

            if (current == last)
                last = trailCurrent;

            count--;
            delete current;
        }
        else
            cout << "The item to be deleted is not in list." endl;
    }//end else
}//end deleteNode
```

STL Sequence Container: list

Chapter 4 listed three types of sequence containers—vector, deque, and list. The sequence containers vector and deque are described in Chapter 4. This section describes the STL sequence container list. List containers are implemented as doubly linked lists. Thus, every element in a list points to its immediate predecessor and to its immediate successor (except the first and last elements). Recall that a linked list is not a random access data structure such as an array. Therefore, to access, for example, the fifth element in the list, we must first traverse the first four elements.

The name of the class containing the definition of the class list is list. The definition of the class list, and the definitions of the functions to implement the various operations on a list, are contained in the header file list. Therefore, to use list in a program, the program must include the following statement:

```
#include <list>
```

Like other container classes, the class list contains several constructors. Thus, a list object can be initialized in several ways when it is declared, as described in Table 5-9.

TABLE 5-9 Various ways to declare a list object

Statement	Description
`list<elemType> listCont;`	Creates the empty list container listCont. (The default constructor is invoked.)
`list<elemType> listCont(otherList);`	Creates the list container listCont and initializes it to the elements of otherList. listCont and otherList are of the same type.
`list<elemType> listCont(size);`	Creates the list container listCont of size size. listCont is initialized using the default constructor.
`list<elemType> listCont(n, elem);`	Creates the list container listCont of size n. listCont is initialized using n copies of the element elem.
`list<elemType> listCont(beg, end);`	Creates the list container listCont. listCont is initialized to the elements in the range [beg, end), that is, all the elements in the range beg...end-1. Both beg and end are iterators.

Table 4-5 describes the operations that are common to all containers, and Table 4-6 describes the operations that are common to all sequence containers. In addition to these common operations, Table 5-10 describes the operations that are specific to a `list` container. The name of the function implementing the operation is shown in bold. (Suppose that `listCont` is a container of type `list`.)

TABLE 5-10 Operations specific to a `list` container

Expression	Description
`listCont.assign(n, elem)`	Assigns `n` copies of `elem`.
`listCont.assign(beg, end)`	Assigns all the elements in the range `beg...end-1`. Both `beg` and `end` are iterators.
`listCont.push_front(elem)`	Inserts `elem` at the beginning of `listCont`.
`listCont.pop_front()`	Removes the first element from `listCont`.
`listCont.front()`	Returns the first element. (Does not check whether the container is empty.)
`listCont.back()`	Returns the last element. (Does not check whether the container is empty.)
`listCont.remove(elem)`	Removes all the elements that are equal to `elem`.
`listCont.remove_if(oper)`	Removes all the elements for which `oper` is `true`.
`listCont.unique()`	If the consecutive elements in `listCont` have the same value, removes the duplicates.
`listCont.unique(oper)`	If the consecutive elements in `listCont` have the same value, removes the duplicates, for which `oper` is `true`.
`listCont1.splice(pos, listCont2)`	All the elements of `listCont2` are moved to `listCont1` before the position specified by the iterator `pos`. After this operation, `listCont2` is empty.

TABLE 5-10 Operations specific to a list container (continued)

Expression	Description
`listCont1.splice(pos, listCont2, pos2)`	All the elements starting at pos2 of listCont2 are moved to listCont1 before the position specified by the iterator pos.
`listCont1.splice(pos, listCont2, beg, end)`	All the elements in the range beg...end-1 of listCont2 are moved to listCont1 before the position specified by the iterator pos. Both beg and end are iterators.
`listCont.sort()`	The elements of listCont are sorted. The sort criterion is <.
`listCont.sort(oper)`	The elements of listCont are sorted. The sort criterion is specified by oper.
`listCont1.merge(listCont2)`	Suppose that the elements of listCont1 and listCont2 are sorted. This operation moves all the elements of listCont2 into listCont1. After this operation, the elements in listCont1 are sorted. Moreover, after this operation, listCont2 is empty.
`listCont1.merge(listCont2, oper)`	Suppose that the elements of listCont1 and listCont2 are sorted according to the sort criteria oper. This operation moves all the elements of listCont2 into listCont1. After this operation, the elements in listCont1 are sorted according to the sort criteria oper.
`listCont.reverse()`	The elements of listCont are reversed.

Example 5-1 shows how to use various operations on a list container.

EXAMPLE 5-1

```
//*****************************************************************
// Author: D.S. Malik
//
// This program illustrates how to use a list container in a
// program.
//*****************************************************************
```

```
#include <iostream>                                          //Line 1
#include <list>                                              //Line 2
#include <iterator>                                          //Line 3
#include <algorithm>                                         //Line 4

using namespace std;                                         //Line 5

int main()                                                   //Line 6
{                                                            //Line 7
    list<int> intList1, intList2;                            //Line 8

    ostream_iterator<int> screen(cout, " ");                 //Line 9

    intList1.push_back(23);                                  //Line 10
    intList1.push_back(58);                                  //Line 11
    intList1.push_back(58);                                  //Line 12
    intList1.push_back(36);                                  //Line 13
    intList1.push_back(15);                                  //Line 14
    intList1.push_back(98);                                  //Line 15
    intList1.push_back(58);                                  //Line 16

    cout << "Line 17: intList1: ";                           //Line 17
    copy(intList1.begin(), intList1.end(), screen);          //Line 18
    cout << endl;                                            //Line 19

    intList2 = intList1;                                     //Line 20

    cout << "Line 21: intList2: ";                           //Line 21
    copy(intList2.begin(), intList2.end(), screen);          //Line 22
    cout << endl;                                            //Line 23

    intList1.unique();                                       //Line 24

    cout << "Line 25: After removing the consecutive "
         << "duplicates," << endl
         << "         intList1: ";                          //Line 25
    copy(intList1.begin(), intList1.end(), screen);          //Line 26
    cout << endl;                                            //Line 27

    intList2.sort();                                         //Line 28

    cout << "Line 29: After sorting, intList2: ";            //Line 29
    copy(intList2.begin(), intList2.end(), screen);          //Line 30
    cout << endl;                                            //Line 31

    return 0;                                                //Line 32
}                                                            //Line 33
```

Sample Run:

```
Line 17: intList1: 23 58 58 36 15 98 58
Line 21: intList2: 23 58 58 36 15 98 58
Line 25: After removing the consecutive duplicates,
         intList1: 23 58 36 15 98 58
Line 29: After sorting, intList2: 15 23 36 58 58 58 98
```

For the most part, the output of the preceding program is straightforward. The statements in Lines 10 through 16 insert the element numbers `23`, `58`, `58`, `36`, `15`, `98`, and `58` (in that order) into `intList1`. The statement in Line 20 copies the elements of `intList1` into `intList2`. After this statement executes, `intList1` and `intList2` are identical. The statement in Line 24 removes any consecutive occurrences of the same elements. For example, the number `58` appears consecutively two times. The operation unique removes two occurrences of `58`. Note that this operation has no effect on the `58` that appears at the end of `intList1`. The statement in Line 28 sorts `intList2`.

Linked Lists with Header and Trailer Nodes

When inserting and deleting items from a linked list (especially an ordered list), we saw that there are special cases, such as inserting (or deleting) at the beginning (the first node) of the list or in an empty list. These cases needed to be handled separately. As a result, the insertion and deletion algorithms were not as simple and straightforward as we would like. One way to simplify these algorithms is to never insert an item before the first or last item and to never delete the first node. Next we discuss how to accomplish this.

Suppose the nodes of a list are in order; that is, they are arranged with respect to a given key. Suppose it is possible for us to determine what the smallest and largest keys are in the given data set. In this case, we can set up a node, called the **header**, at the beginning of the list containing a value smaller than the smallest value in the data set. Similarly, we can set up a node, called the **trailer**, at the end of the list containing a value larger than the largest value in the data set. These two nodes, header and trailer, serve merely to simplify the insertion and deletion algorithms and are not part of the actual list. The actual list is between these two nodes.

For example, suppose the data are ordered according to the last name. Further, assume that the last name is a string of at most `8` characters. The smallest last name is larger than the string `"A"` and the largest last name is smaller than the string `"zzzzzzzz"`. We can set up the header node with the value `"A"` and the trailer node with the value `"zzzzzzzz"`. Figure 5-33 shows an empty and a nonempty linked list with header and trailer nodes.

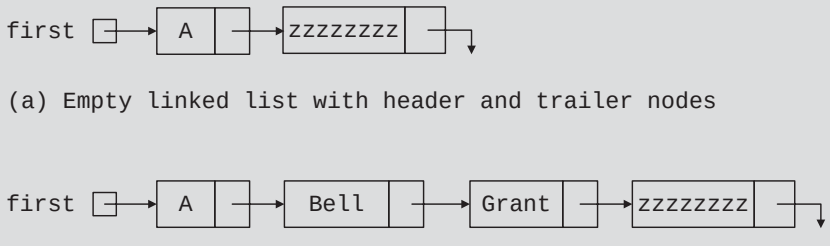

FIGURE 5-33 Linked list with header and trailer nodes

As before, the usual operations on lists with header and trailer nodes are as follows: Initialize the list (to an empty state), destroy the list, print the list, find the length of the list, search the list for a given item, insert an item in the list, delete an item from the list, and copy the list.

We leave it as an exercise for you to design a class to implement a linked list with header and trailer nodes. (See Programming Exercise 12 at the end of this chapter.)

Circular Linked Lists

A linked list in which the last node points to the first node is called a **circular linked list**. Figure 5-34 shows various circular linked lists.

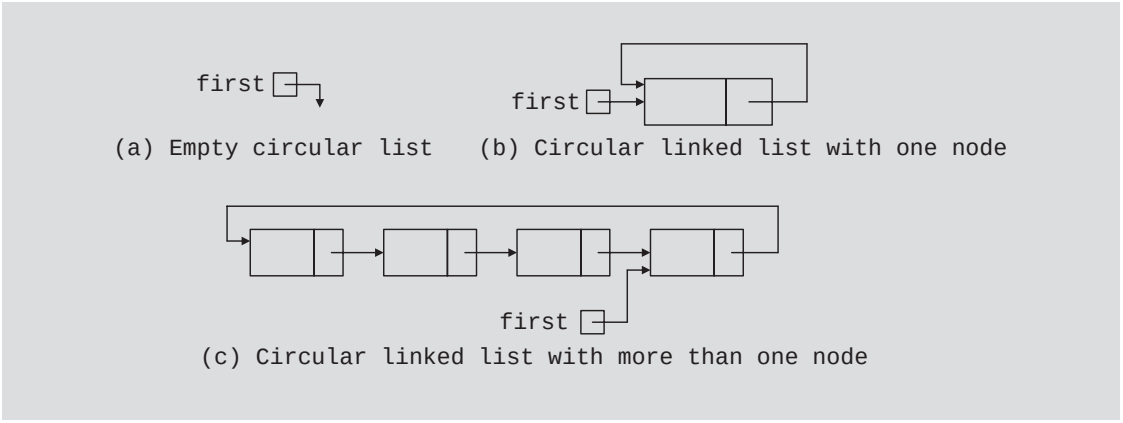

FIGURE 5-34 Circular linked lists

In a circular linked list with more than one node, as in Figure 5-34(c), it is convenient to make the pointer `first` point to the last node of the list. Then by using `first` you can access both the first and the last node of the list. For example, first points to the last node and `first->link` points to the first node.

As before, the usual operations on a circular list are as follows: Initialize the list (to an empty state), determine if the list is empty, destroy the list, print the list, find the length of the list, search the list for a given item, insert an item in the list, delete an item from the list, and copy the list.

We leave it as exercise for you to design a class to implement a sorted circular linked list. (See Programming Exercise 13 at the end of this chapter.)

PROGRAMMING EXAMPLE: Video Store

During holidays or on weekends, a family or an individual typically rents a movie either from a local store or online. Therefore, we write a program that does the following:

1. Rent a video; that is, check out a video.
2. Return, or check in, a video.
3. Create a list of videos owned by the store.
4. Show the details of a particular video.
5. Print a list of all the videos in the store.
6. Check whether a particular video is in the store.
7. Maintain a customer database.
8. Print a list of all the videos rented by each customer.

Let us write a program for the video store. This example further illustrates the object-oriented design methodology and, in particular, inheritance and overloading.

The programming requirement tells us that the video store has two major components: videos and customers. We will describe these two components in detail. We also need to maintain various lists:

- A list of all the videos in the store
- A list of all the store's customers
- Lists of the videos currently rented by each customer

We will develop the program in two parts. In part 1, we design, implement, and test the video component. In part 2, we design and implement the customer component, which is then added to the video component developed in part 1. That is, after completing parts 1 and 2, we can perform all the operations listed previously.

PART 1: VIDEO COMPONENT

Video Object

This is the first stage, wherein we discuss the video component. The common things associated with a video are as follows:

- Name of the movie
- Names of the stars
- Name of the producer
- Name of the director
- Name of the production company
- Number of copies in the store

From this list, we see that some of the operations to be performed on the video object are as follows:

1. Set the video information—that is, the title, stars, production company, and so on.
2. Show the details of a particular video.
3. Check the number of copies in the store.
4. Check out (that is, rent) the video. In other words, if the number of copies is greater than zero, decrement the number of copies by one.
5. Check in (that is, return) the video. To check in a video, first we must check whether the store owns such a video and, if it does, increment the number of copies by one.
6. Check whether a particular video is available—that is, check whether the number of copies currently in the store is greater than zero.

The deletion of a video from the video list requires that the video list be searched for the video to be deleted. Thus, we need to check the title of a video to find out which video is to be deleted from the list. For simplicity, we assume that two videos are the same if they have the same title.

The following class defines the video object as an ADT:

```
//***********************************************************
// Author: D.S. Malik
//
// class videoType
// This class specifies the members to implement a video.
//***********************************************************

#include <iostream>
#include <string>

using namespace std;

class videoType
{
    friend ostream& operator<< (ostream&, const videoType&);

public:
    void setVideoInfo(string title, string star1,
                      string star2, string producer,
                      string director, string productionCo,
                      int setInStock);
      //Function to set the details of a video.
      //The private member variables are set according to the
      //parameters.
```

```
  //Postcondition: videoTitle = title; movieStar1 = star1;
  //     movieStar2 = star2; movieProducer = producer;
  //     movieDirector = director;
  //     movieProductionCo = productionCo;
  //     copiesInStock = setInStock;

int getNoOfCopiesInStock() const;
  //Function to check the number of copies in stock.
  //Postcondition: The value of copiesInStock is returned.

void checkOut();
  //Function to rent a video.
  //Postcondition: The number of copies in stock is
  //    decremented by one.

void checkIn();
  //Function to check in a video.
  //Postcondition: The number of copies in stock is
  //    incremented by one.

void printTitle() const;
  //Function to print the title of a movie.

void printInfo() const;
  //Function to print the details of a video.
  //Postcondition: The title of the movie, stars, director,
  //    and so on are displayed on the screen.

bool checkTitle(string title);
  //Function to check whether the title is the same as the
  //title of the video.
  //Postcondition: Returns the value true if the title is the
  //    same as the title of the video; false otherwise.

void updateInStock(int num);
  //Function to increment the number of copies in stock by
  //adding the value of the parameter num.
  //Postcondition: copiesInStock = copiesInStock + num;

void setCopiesInStock(int num);
  //Function to set the number of copies in stock.
  //Postcondition: copiesInStock = num;

string getTitle() const;
  //Function to return the title of the video.
  //Postcondition: The title of the video is returned.

videoType(string title = "", string star1 = "",
          string star2 = "", string producer = "",
          string director = "", string productionCo = "",
          int setInStock = 0);
```

```
        //constructor
        //The member variables are set according to the
        //incoming parameters. If no values are specified, the
        //default values are assigned.
        //Postcondition: videoTitle = title; movieStar1 = star1;
        //    movieStar2 = star2; movieProducer = producer;
        //    movieDirector = director;
        //    movieProductionCo = productionCo;
        //    copiesInStock = setInStock;

        //Overload the relational operators.
    bool operator==(const videoType&) const;
    bool operator!=(const videoType&) const;

private:
    string videoTitle;  //variable to store the name of the movie
    string movieStar1;  //variable to store the name of the star
    string movieStar2;  //variable to store the name of the star
    string movieProducer; //variable to store the name of the
                          //producer
    string movieDirector; //variable to store the name of the
                          //director
    string movieProductionCo; //variable to store the name
                              //of the production company
    int copiesInStock;  //variable to store the number of
                        //copies in stock
};
```

We leave the UML diagram of the `class videoType` as an exercise for you, see Exercise 15 at the end of this chapter.

For easy output, we will overload the output stream insertion operator, `<<`, for the `class videoType`.

Next, we write the definitions of each function in the **`class videoType`**. The definitions of these functions, as shown here, are quite straightforward and easy to follow:

```
void videoType::setVideoInfo(string title, string star1,
                             string star2, string producer,
                             string director,
                             string productionCo,
                             int setInStock)
{
    videoTitle = title;
    movieStar1 = star1;
    movieStar2 = star2;
    movieProducer = producer;
    movieDirector = director;
    movieProductionCo = productionCo;
    copiesInStock = setInStock;
}
```

```
void videoType::checkOut()
{
    if (getNoOfCopiesInStock() > 0)
        copiesInStock--;
    else
        cout << "Currently out of stock" << endl;
}

void videoType::checkIn()
{
    copiesInStock++;
}

int videoType::getNoOfCopiesInStock() const
{
    return copiesInStock;
}

void videoType::printTitle() const
{
    cout << "Video Title: " << videoTitle << endl;
}

void videoType::printInfo() const
{
    cout << "Video Title: " << videoTitle << endl;
    cout << "Stars: " << movieStar1 << " and "
         << movieStar2 << endl;
    cout << "Producer: " << movieProducer << endl;
    cout << "Director: " << movieDirector << endl;
    cout << "Production Company: " << movieProductionCo << endl;
    cout << "Copies in stock: " << copiesInStock << endl;
}

bool videoType::checkTitle(string title)
{
    return (videoTitle == title);
}

void videoType::updateInStock(int num)
{
    copiesInStock += num;
}

void videoType::setCopiesInStock(int num)
{
    copiesInStock = num;
}
```

```
string videoType::getTitle() const
{
    return videoTitle;
}

videoType::videoType(string title, string star1,
                     string star2, string producer,
                     string director,
                     string productionCo, int setInStock)
{
    setVideoInfo(title, star1, star2, producer, director,
                 productionCo, setInStock);
}

bool videoType::operator==(const videoType& other) const
{
    return (videoTitle == other.videoTitle);
}

bool videoType::operator!=(const videoType& other) const
{
    return (videoTitle != other.videoTitle);
}

ostream& operator<< (ostream& osObject, const videoType& video)
{
    osObject << endl;
    osObject << "Video Title: " << video.videoTitle << endl;
    osObject << "Stars: " << video.movieStar1 << " and "
             << video.movieStar2 << endl;
    osObject << "Producer: " << video.movieProducer << endl;
    osObject << "Director: " << video.movieDirector << endl;
    osObject << "Production Company: "
             << video.movieProductionCo << endl;
    osObject << "Copies in stock: " << video.copiesInStock
             << endl;
    osObject << "_____" << endl;

    return osObject;
}
```

Video List

This program requires us to maintain a list of all the videos in the store, and we should be able to add a new video to our list. In general, we would not know how many videos are in the store, and adding or deleting a video from the store would change the number of videos in the store. Therefore, we will use a linked list to create a list of videos.

Earlier in this chapter, we defined the `class unorderedLinkedList` to create a linked list of objects. We also defined the basic operations such as insertion and deletion

of a video in the list. However, some operations are very specific to the video list, such as check out a video, check in a video, set the number of copies of a video, and so on. These operations are not available in the `class unorderedLinkedList`. We, therefore, derive a `class videoListType` from the `class unorderedLinkedList` and add these operations.

The definition of the `class videoListType` is as follows:

```
//*****************************************************************
// Author: D.S. Malik
//
// class videoListType
// This class specifies the members to implement a list of videos.
//*****************************************************************

#include <string>
#include "unorderedLinkedList.h"
#include "videoType.h"

using namespace std;

class videoListType:public unorderedLinkedList<videoType>
{
public:
    bool videoSearch(string title) const;
      //Function to search the list to see whether a
      //particular title, specified by the parameter title,
      //is in the store.
      //Postcondition: Returns true if the title is found, and
      //    false otherwise.

    bool isVideoAvailable(string title) const;
      //Function to determine whether a copy of a particular
      //video is in the store.
      //Postcondition: Returns true if at least one copy of the
      //    video specified by title is in the store, and false
      //    otherwise.

    void videoCheckOut(string title);
      //Function to check out a video, that is, rent a video.
      //Postcondition: copiesInStock is decremented by one.

    void videoCheckIn(string title);
      //Function to check in a video returned by a customer.
      //Postcondition: copiesInStock is incremented by one.

    bool videoCheckTitle(string title) const;
      //Function to determine whether a particular video is in
      //the store.
      //Postcondition: Returns true if the video's title is the
      //    same as title, and false otherwise.
```

```
    void videoUpdateInStock(string title, int num);
      //Function to update the number of copies of a video
      //by adding the value of the parameter num. The
      //parameter title specifies the name of the video for
      //which the number of copies is to be updated.
      //Postcondition: copiesInStock = copiesInStock + num;

    void videoSetCopiesInStock(string title, int num);
      //Function to reset the number of copies of a video.
      //The parameter title specifies the name of the video
      //for which the number of copies is to be reset, and the
      //parameter num specifies the number of copies.
      //Postcondition: copiesInStock = num;

    void videoPrintTitle() const;
      //Function to print the titles of all the videos in the store.

 private:
    void searchVideoList(string title, bool& found,
                         nodeType<videoType>* &current) const;
      //This function searches the video list for a particular
      //video, specified by the parameter title.
      //Postcondition: If the video is found, the parameter found is
      //    set to true, otherwise it is set to false. The parameter
      //    current points to the node containing the video.
};
```

Note that the `class videoListType` is derived from the `class unorderedLinkedList` via a `public` inheritance. Furthermore, `unorderedLinkedList` is a class template and we have passed the `class videoType` as a parameter to this class. That is, the `class videoListType` is not a template. Because we are now dealing with a very specific data type, the `class videoListType` is no longer needed to be a template. Thus, the `info` type of each node in the linked list is now `videoType`. Through the member functions of the `class videoType`, certain members—such as `videoTitle` and `copiesInStock` of an object of type `videoType`—can now be accessed.

The definitions of the functions to implement the operations of the `class videoListType` are given next.

The primary operations on the video list are to check in a video and to check out a video. Both operations require the list to be searched and the location of the video being checked in or checked out to be found in the video list. Other operations such as seeing whether a particular video is in the store, updating the number of copies of a video, and so on also require the video list to be searched. To simplify the search process, we will write a function that searches the video list for a particular video. If the video is found, it sets a parameter `found` to `true` and returns a pointer to the video so that check-in, check-out, and other operations on the video object can be performed. Note that the function `searchVideoList` is a

private data member of the class videoListType because it is used only for internal manipulation. First, we describe the search procedure.

The following function definition performs the desired search:

```
void videoListType::searchVideoList(string title, bool& found,
                          nodeType<videoType>* &current) const
{
    found = false;   //set found to false

    current = first; //set current to point to the first node

    while (current != NULL && !found)     //search the list
        if (current->info.checkTitle(title)) //the item is found
            found = true;
        else
            current = current->link; //advance current to
                                     //the next node
}//end searchVideoList
```

If the search is successful, the parameter found is set to true and the parameter current points to the node containing the video info. If it is unsuccessful, found is set to false and current will be NULL.

The definitions of the other functions of the class videoListType follow:

```
bool videoListType::isVideoAvailable(string title) const
{
    bool found;
    nodeType<videoType> *location;

    searchVideoList(title, found, location);

    if (found)
        found = (location->info.getNoOfCopiesInStock() > 0);
    else
        found = false;

    return found;
}

void videoListType::videoCheckIn(string title)
{
    bool found = false;
    nodeType<videoType> *location;

    searchVideoList(title, found, location); //search the list

    if (found)
        location->info.checkIn();
```

```
    else
        cout << "The store does not carry " << title
             << endl;
}

void videoListType::videoCheckOut(string title)
{
    bool found = false;
    nodeType<videoType> *location;

    searchVideoList(title, found, location); //search the list

    if (found)
        location->info.checkOut();
    else
        cout << "The store does not carry " << title
             << endl;
}

bool videoListType::videoCheckTitle(string title) const
{
    bool found = false;
    nodeType<videoType> *location;

    searchVideoList(title, found, location); //search the list

    return found;
}

void videoListType::videoUpdateInStock(string title, int num)
{
    bool found = false;
    nodeType<videoType> *location;

    searchVideoList(title, found, location); //search the list

    if (found)
        location->info.updateInStock(num);
    else
        cout << "The store does not carry " << title
             << endl;
}

void videoListType::videoSetCopiesInStock(string title, int num)
{
    bool found = false;
    nodeType<videoType> *location;

    searchVideoList(title, found, location);

    if (found)
        location->info.setCopiesInStock(num);
```

```
    else
        cout << "The store does not carry " << title
             << endl;
}

bool videoListType::videoSearch(string title) const
{
    bool found = false;
    nodeType<videoType> *location;

    searchVideoList(title, found, location);

    return found;
}

void videoListType::videoPrintTitle() const
{
    nodeType<videoType>* current;

    current = first;
    while (current != NULL)
    {
        current->info.printTitle();
        current = current->link;
    }
}
```


PART 2: CUSTOMER COMPONENT

Customer Object

The customer object stores information about a customer, such as the first name, last name, account number, and a list of videos rented by the customer.

Every customer is a person. We have already designed the `class personType` in Example 1-12 (Chapter 1) and described the necessary operations on the name of a person. Therefore, we can derive the `class customerType` from the `class personType` and add the additional members that we need. First, however, we must redefine the `class personType` to take advantage of the new features of object-oriented design that you have learned, such as operator overloading, and then derive the `class customerType`.

The basic operations on an object of type `customerType` are as follows:

1. Print the name, the account number, and the list of rented videos.
2. Set the name and the account number.
3. Rent a video; that is, add the rented video to the list.
4. Return a video; that is, delete the rented video from the list.
5. Show the account number.

The details of implementing the customer component are left as an exercise for you. (See Programming Exercise 14 at the end of this chapter.)

MAIN PROGRAM

We will now write the main program to test the video object. We assume that the necessary data for the videos are stored in a file. We will open the file and create the list of videos owned by the video store. The data in the input file is in the following form:

```
video title (that is, the name of the movie)
movie star1
movie star2
movie producer
movie director
movie production co.
number of copies
.
.
.
```

We will write a function, `createVideoList`, to read the data from the input file and create the list of videos. We will also write a function, `displayMenu`, to show the different choices—such as check in a movie or check out a movie—that the user can make. The algorithm of the function `main` is as follows:

1. Open the input file.

   If the input file does not exist, exit the program.
2. Create the list of videos (`createVideoList`).
3. Show the menu (`displayMenu`).
4. While not done

Perform various operations.

Opening the input file is straightforward. Let us describe Steps 2 and 3, which are accomplished by writing two separate functions: `createVideoList` and `displayMenu`.

`createVideoList`

This function reads the data from the input file and creates a linked list of videos. Because the data will be read from a file and the input file was opened in the function `main`, we pass the input file pointer to this function. We also pass the video list pointer, declared in the function `main`, to this function. Both parameters are reference parameters. Next, we read the data for each video and then insert the video in the list. The general algorithm is as follows:

1. Read the data and store it in a video object.
2. Insert the video in the list.
3. Repeat Steps a and b for each video's data in the file.

displayMenu This function informs the user what to do. It contains the following output statements:

Select one of the following:

1: To check whether the store carries a particular video

2: To check out a video

3: To check in a video

4: To check whether a particular video is in stock

5: To print only the titles of all the videos

6: To print a list of all the videos

9: To exit

PROGRAM LISTING

```
//*****************************************************************
// Author: D.S. Malik
//
// This program illustrates how to use the classes videoType and
// videListType to create and process a list of videos.
//*****************************************************************

#include <iostream>
#include <fstream>
#include <string>
#include "videoType.h"
#include "videoListType.h"

using namespace std;

void createVideoList(ifstream& infile,
                     videoListType& videoList);
void displayMenu();

int main()
{
    videoListType videoList;
    int choice;
    char ch;
    string title;

    ifstream infile;

           //open the input file
    infile.open("videoDat.txt");
    if (!infile)
```

```
{
    cout << "The input file does not exist. "
         << "The program terminates!!!" << endl;
    return 1;
}

    //create the video list
createVideoList(infile, videoList);
infile.close();

    //show the menu
displayMenu();
cout << "Enter your choice: ";
cin >> choice;    //get the request
cin.get(ch);
cout << endl;

    //process the requests
while (choice != 9)
{
    switch (choice)
    {
    case 1:
        cout << "Enter the title: ";
        getline(cin, title);
        cout << endl;

        if (videoList.videoSearch(title))
            cout << "The store carries " << title
                 << endl;
        else
            cout << "The store does not carry "
                 << title << endl;
        break;

    case 2:
        cout << "Enter the title: ";
        getline(cin, title);
        cout << endl;

        if (videoList.videoSearch(title))
        {
            if (videoList.isVideoAvailable(title))
            {
                videoList.videoCheckOut(title);
                cout << "Enjoy your movie: "
                     << title << endl;
            }
            else
                cout << "Currently " << title
                     << " is out of stock." << endl;
        }
```

```
        else
            cout << "The store does not carry "
                 << title << endl;
        break;

    case 3:
        cout << "Enter the title: ";
        getline(cin, title);
        cout << endl;

        if (videoList.videoSearch(title))
        {
            videoList.videoCheckIn(title);
            cout << "Thanks for returning "
                 << title << endl;
        }
        else
            cout << "The store does not carry "
                 << title << endl;
        break;

    case 4:
        cout << "Enter the title: ";
        getline(cin, title);
        cout << endl;

        if (videoList.videoSearch(title))
        {
            if (videoList.isVideoAvailable(title))
                cout << title << " is currently in "
                     << "stock." << endl;
            else
                cout << title << " is currently out "
                     << "of stock." << endl;
        }
        else
            cout << "The store does not carry "
                 << title << endl;
        break;

    case 5:
        videoList.videoPrintTitle();
        break;

    case 6:
        videoList.print();
        break;

    default:
        cout << "Invalid selection." << endl;
    }//end switch
```

```
        displayMenu();     //display menu

        cout << "Enter your choice: ";
        cin >> choice;     //get the next request
        cin.get(ch);
        cout << endl;
    }//end while

    return 0;
}

void createVideoList(ifstream& infile,
                     videoListType& videoList)
{
    string title;
    string star1;
    string star2;
    string producer;
    string director;
    string productionCo;
    char ch;
    int inStock;

    videoType newVideo;

    getline(infile, title);

    while (infile)
    {
        getline(infile, star1);
        getline(infile, star2);
        getline(infile, producer);
        getline(infile, director);
        getline(infile, productionCo);
        infile >> inStock;
        infile.get(ch);
        newVideo.setVideoInfo(title, star1, star2, producer,
                              director, productionCo, inStock);
        videoList.insertFirst(newVideo);

        getline(infile, title);
    }//end while
}//end createVideoList

void displayMenu()
{
    cout << "Select one of the following:" << endl;
    cout << "1: To check whether the store carries a "
         << "particular video." << endl;
    cout << "2: To check out a video." << endl;
    cout << "3: To check in a video." << endl;
```

```
    cout << "4: To check whether a particular video is "
         << "in stock." << endl;
    cout << "5: To print only the titles of all the videos."
         << endl;
    cout << "6: To print a list of all the videos." << endl;
    cout << "9: To exit" << endl;
} //end createVideoList
```

QUICK REVIEW

1. A linked list is a list of items, called nodes, in which the order of the nodes is determined by the address, called a link, stored in each node.
2. The pointer to a linked list—that is, the pointer to the first node in the list—is stored in a separate location, called the head or first.
3. A linked list is a dynamic data structure.
4. The length of a linked list is the number of nodes in the list.
5. Item insertion and deletion from a linked list does not require data movement; only the pointers are adjusted.
6. A (single) linked list is traversed in only one direction.
7. The search on a linked list is sequential.
8. The first (or head) pointer of a linked list is always fixed, pointing to the first node in the list.
9. To traverse a linked list, the program must use a pointer different than the head pointer of the list, initialized to the first node in the list.
10. In a doubly linked list, every node has two links: one points to the next node, and one points to the previous node.
11. A doubly linked list can be traversed in either direction.
12. In a doubly linked list, item insertion and deletion requires the adjustment of two pointers in a node.
13. The name of the class containing the definition of the `class list` is `list`.
14. In addition to the operations that are common to sequence containers (see Chapter 4), the other operations that can be used to manipulate the elements in a list container are `assign`, `push_front`, `pop_front`, `front`, `back`, `remove`, `remove_if`, `unique`, `splice`, `sort`, `merge`, and `reverse`.
15. A linked list with header and trailer nodes simplifies the insertion and deletion operations.
16. The header and trailer nodes are not part of the actual list. The actual list elements are between the header and trailer nodes.

17. A linked list with header and trailer nodes is empty if the only nodes in the list are the header and the trailer.
18. A circular linked list is a list in which, if the list is nonempty, the last node points to the first node.

EXERCISES

1. Mark the following statements as true or false.
   a. In a linked list, the order of the elements is determined by the order in which the nodes were created to store the elements.
   b. In a linked list, memory allocated for the nodes is sequential.
   c. A single linked list can be traversed in either direction.
   d. In a linked list, nodes are always inserted either at the beginning or the end because a linked link is not a random access data structure.
   e. The head pointer of a linked list cannot be used to traverse the list.

   Consider the linked list shown in Figure 5-35. Assume that the nodes are in the usual `info-link` form. Use this list to answer Exercises 2 through 7. If necessary, declare additional variables. (Assume that `list`, `p`, `s`, `A`, and `B` are pointers of type `nodeType`.)

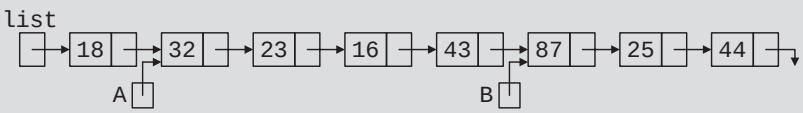

FIGURE 5-35 Linked list for Exercises 2–7

2. What is the output of each of the following C++ statements?
   a. `cout << list->info;`
   b. `cout << A->info;`
   c. `cout << B->link->info;`
   d. `cout << list->link->link->info`
3. What is the value of each of the following relational expressions?
   a. `list->info >= 18`
   b. `list->link == A`
   c. `A->link->info == 16`
   d. `B->link == NULL`
   e. `list->info == 18`