You have already seen how data is organized and processed sequentially using an array called a *sequential list*. You have performed several operations on sequential lists, such as sorting, inserting, deleting, and searching. You also found that if data is not sorted, then searching for an item in the list can be very time consuming especially with large lists. Once the data is sorted, you can use a binary search and improve the search algorithm. However, in this case, insertion and deletion become time consuming especially with large lists, because these operations require data movement. Also, because the array size must be fixed during execution, new items can be added only if there is room. Thus, there are limitations on when you organize data in an array.

This chapter helps you to overcome some of these problems. Chapter 14 showed how memory (variables) can be dynamically allocated and deallocated using pointers. This chapter uses pointers to organize and process data in lists called **linked lists**. Recall that when data is stored in an array, memory for the components of the array is contiguous—that is, the blocks are allocated one after the other. However, as we will see, the components (called nodes) of a linked list need not be contiguous.

Linked Lists

A linked list is a collection of components called **nodes.** Every node (except the last node) contains the address of the next node. Thus, every node in a linked list has two components: one to store the relevant information (that is, data) and one to store the address, called the **link**, of the next node in the list. The address of the first node in the list is stored in a separate location called the **head** or **first**. Figure 18-1 is a pictorial representation of a node.

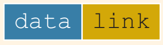


FIGURE 18-1 Structure of a node

Linked list: A list of items, called **nodes**, in which the order of the nodes is determined by the address, called the **link**, stored in each node.

The list in Figure 18-2 is an example of a linked list.

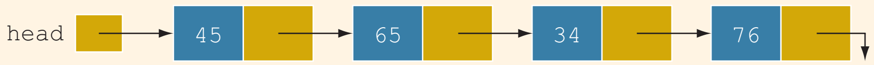


FIGURE 18-2 Linked list

The arrow in each node indicates that the address of the node to which it is pointing is stored in that node. The down arrow in the last node indicates that this link field is `NULL`.

For a better understanding of this notation, suppose that the first node is at memory location 1200, and the second node is at memory location 1575. We thus have Figure 18-3.

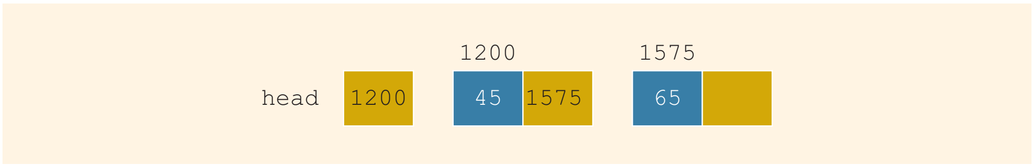


FIGURE 18-3 Linked list and values of the links

The value of the head is 1200, the data part of the first node is 45, and the link component of the first node contains 1575, the address of the second node. If no confusion arises, then we will use the arrow notation whenever we draw the figure of a linked list.

For simplicity and for the ease of understanding and clarity, Figures 18-3 through 18-6 use decimal integers as the values of memory addresses. However, in computer memory, the memory addresses are in binary.

Because each node of a linked list has two components, we need to declare each node as a **class** or **struct**. The data type of each node depends on the specific application—that is, what kind of data is being processed. However, the link component of each node is a pointer. The data type of this pointer variable is the node type itself. For the previous linked list, the definition of the node is as follows. (Suppose that the data type is **int**.)

```
struct nodeType
{
    int info;
    nodeType *link;
};
```

The variable declaration is:

```
nodeType *head;
```

Linked Lists: Some Properties

To help you better understand the concept of a linked list and a node, some important properties of linked lists are described next.

Consider the linked list in Figure 18-4.

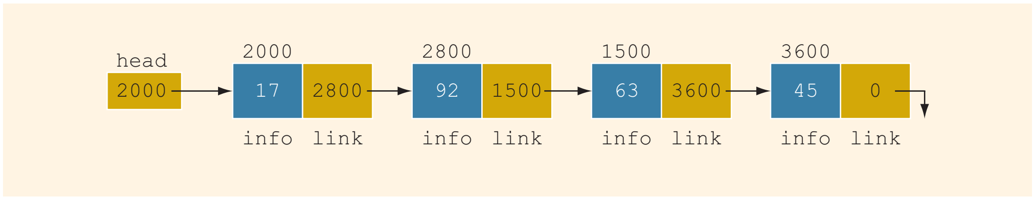


FIGURE 18-4 Linked list with four nodes

This linked list has four nodes. The address of the first node is stored in the pointer `head`. Each node has two components: `info`, to store the info, and `link`, to store the address of the next node. For simplicity, we assume that `info` is of type **`int`**.

Suppose that the first node is at location `2000`, the second node is at location `2800`, the third node is at location `1500`, and the fourth node is at location `3600`. Therefore, the value of `head` is `2000`, the value of the component `link` of the first node is `2800`, the value of the component `link` of the second node is `1500`, and so on. Also, the value `0` in the component link of the last node means that this value is `NULL`, which we indicate by drawing a down arrow. The number at the top of each node is the address of that node. The following table shows the values of `head` and some other nodes in the list shown in Figure 18-4.

	Value	**Explanation**
`head`	`2000`	
`head->info`	`17`	Because `head` is `2000` and the `info` of the node at location `2000` is `17`
`head->link`	`2800`	
`head->link->info`	`92`	Because `head->link` is `2800` and the `info` of the node at location `2800` is `92`

Suppose that `current` is a pointer of the same type as the pointer `head`. Then, the statement:

```
current = head;
```

copies the value of `head` into `current` (see Figure 18-5).

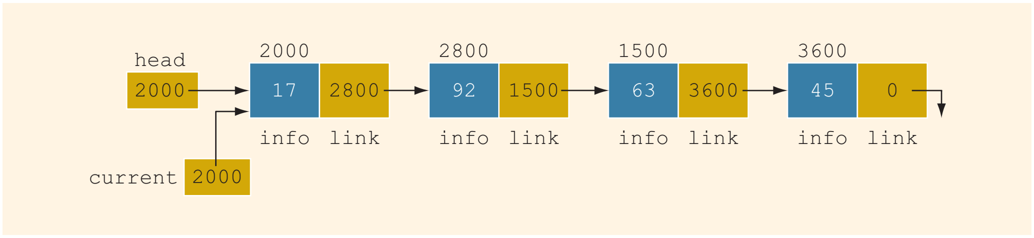


FIGURE 18-5 Linked list after the statement `current = head;` executes

Clearly, in Figure 18-5:

	Value
`current`	`2000`
`current->info`	`17`
`current->link`	`2800`
`current->link->info`	`92`

Now consider the statement:

```
current = current->link;
```

This statement copies the value of `current->link`, which is `2800`, into `current`. Therefore, after this statement executes, `current` points to the second node in the list. (When working with linked lists, we typically use these types of statements to advance a pointer to the next node in the list.) See Figure 18-6.

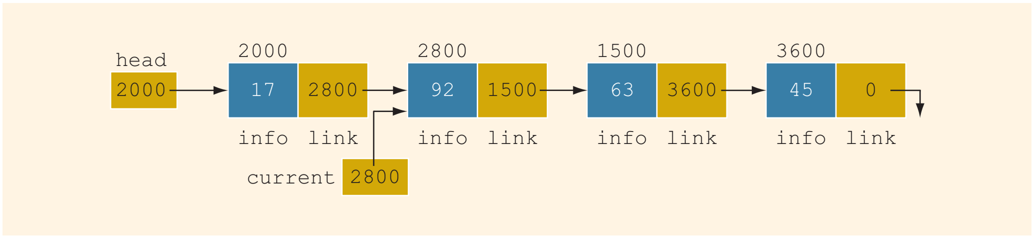


FIGURE 18-6 List after the statement `current = current->link;` executes

In Figure 18-6:

	Value
`current`	`2800`
`current->info`	`92`
`current->link`	`1500`
`current->link->info`	`63`

Finally, note that in Figure 18-6:

	Value
`head->link->link`	`1500`
`head->link->link->info`	`63`
`head->link->link->link`	`3600`
`head->link->link->link->info`	`45`
`current->link->link`	`3600`
`current->link->link->info`	`45`
`current->link->link->link`	`0` (that is, `NULL`)
`current->link->link->link->info`	Does not exist

From now on, when working with linked lists, we will use only the arrow notation.

TRAVERSING A LINKED LIST

The basic operations of a linked list are as follows: search the list to determine whether a particular item is in the list, insert an item in the list, and delete an item from the list.

These operations require the list to be traversed. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

Suppose that the pointer head points to the first node in the list, and the link of the last node is NULL. We cannot use the pointer head to traverse the list because if we use head to traverse the list, we would lose the nodes of the list. This problem occurs because the links are in only one direction. The pointer head contains the address of the first node, the first node contains the address of the second node, the second node contains the address of the third node, and so on. If we move head to the second node, the first node is lost (unless we save a pointer to this node). If we keep advancing head to the next node, we will lose all of the nodes of the list (unless we save a pointer to each node before advancing head, which is impractical because it would require additional computer time and memory space to maintain the list).

Therefore, we always want head to point to the first node. It now follows that we must traverse the list using another pointer of the same type. Suppose that current is a pointer of the same type as head. The following code traverses the list:

```
current = head;

while (current != NULL)
{
    //Process the current node
    current = current->link;
}
```

For example, suppose that head points to a linked list of numbers. The following code outputs the data stored in each node:

```
current = head;

while (current != NULL)
{
    cout << current->info << " ";
    current = current->link;
}
```

ITEM INSERTION AND DELETION

This section discusses how to insert an item into, and delete an item from, a linked list. Consider the following definition of a node. (For simplicity, we assume that the info type is int. The next section, which discusses linked lists as an abstract data type (ADT) using templates, uses the generic definition of a node.)

```
struct nodeType
{
    int info;
    nodeType *link;
};
```

We will use the following variable declaration:

```
nodeType *head, *p, *q, *newNode;
```

INSERTION

Consider the linked list shown in Figure 18-7.

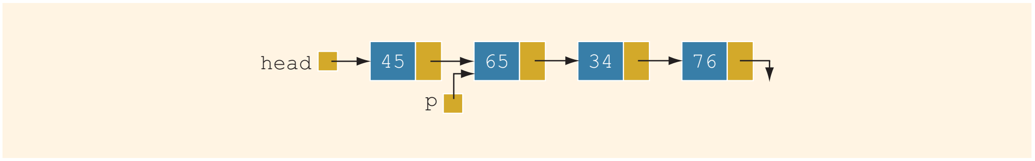


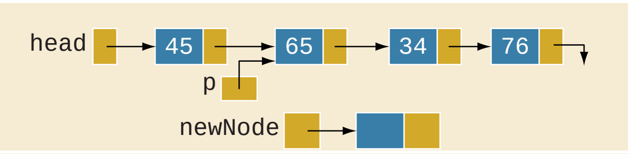
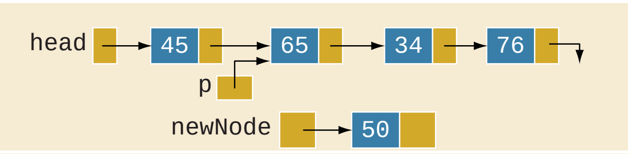
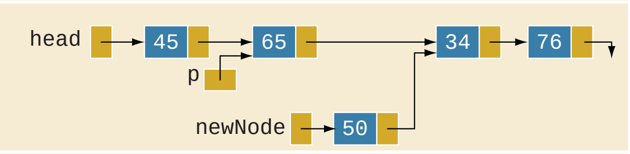
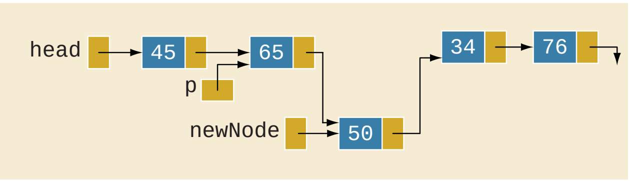
FIGURE 18-7 Linked list before item insertion

Suppose that p points to the node with info 65, and a new node with info 50 is to be created and inserted after p. Consider the following statements:

```
newNode = new nodeType;   //create newNode
newNode->info = 50;        //store 50 in the new node
newNode->link = p->link;
p->link = newNode;
```

Table 18-1 shows the effect of these statements.

TABLE 18-1 Inserting a Node in a Linked List

Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

Note that the sequence of statements to insert the node is very important because to insert `newNode` in the list, we use only one pointer, `p`, to adjust the links of the node of the linked list. Suppose that we reverse the sequence of the statements and execute the statements in the following order:

```
p->link = newNode;
newNode->link = p->link;
```

Figure 18-8 shows the resulting list after these statements execute.

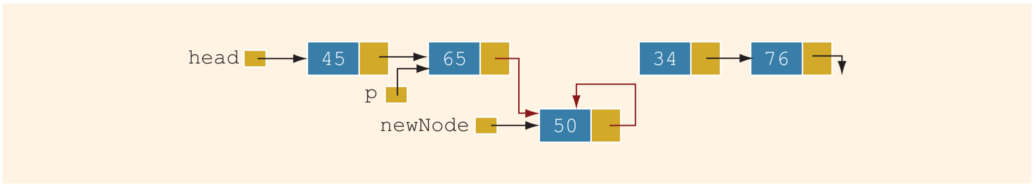


FIGURE 18-8 List after the execution of the statement `p->link = newNode;` followed by the execution of the statement `newNode->link = p->link;`

From Figure 18-8, it is clear that `newNode` points back to itself and the remainder of the list is lost.

Using two pointers, we can simplify the insertion code somewhat. Suppose `q` points to the node with `info 34` (see Figure 18-9).

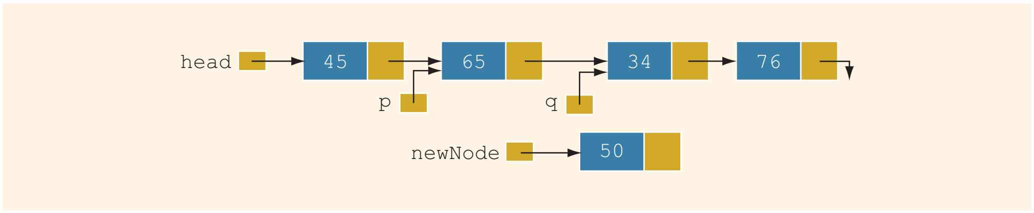


FIGURE 18-9 List with pointers `p` and `q`

The following statements insert `newNode` between `p` and `q`.

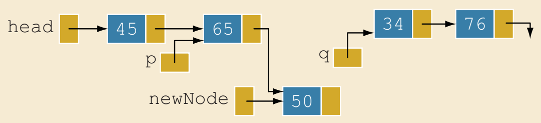
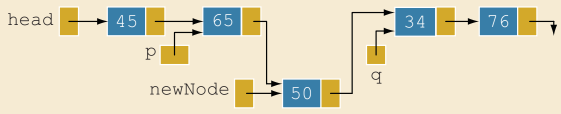
```
newNode->link = q;
p->link = newNode;
```

The order in which these statements execute does not matter. To illustrate this, suppose that we execute the statements in the following order:

```
p->link = newNode;
newNode->link = q;
```

Table 18-2 shows the effect of these statements.

TABLE 18-2 Inserting a Node in a Linked List Using Two Pointers

Statement	Effect
`p->link = newNode;`	
`newNode->link = q;`	

Deletion

Consider the linked list shown in Figure 18-10.

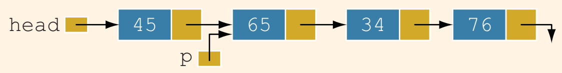


FIGURE 18-10 Node to be deleted is with info 34

Suppose that the node with `info 34` is to be deleted from the list. The following statement removes the node from the list.

```
p->link = p->link->link;
```

Figure 18-11 shows the resulting list after the preceding statement executes.

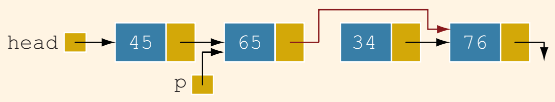


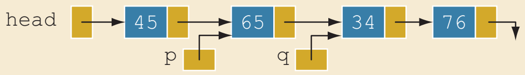
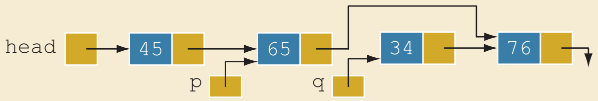
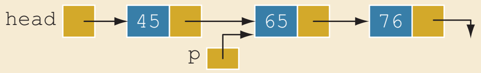
FIGURE 18-11 List after the statement `newNode->link = q;` executes

From Figure 18-11, it is clear that the node with `info 34` is removed from the list. However, the memory is still occupied by this node, and this memory is inaccessible; that is, this node is dangling. To deallocate the memory, we need a pointer to this node. The following statements delete the node from the list and deallocate the memory occupied by this node.

```
q = p->link;
p->link = q->link;
delete q;
```

Table 18-3 shows the effect of these statements.

TABLE 18-3 Deleting a Node from a Linked List

Statement	Effect
`q = p->link;`	
`p->link = q->link;`	
`delete q;`	

Building a Linked List

Now that we know how to insert a node in a linked list, let us see how to build a linked list. First, we consider a linked list in general. If the data we read is unsorted, the linked list will be unsorted. Such a list can be built in two ways: forward and backward. In the forward manner, a new node is always inserted at the end of the linked list. In the backward manner, a new node is always inserted at the beginning of the list. We will consider both cases.

BUILDING A LINKED LIST FORWARD

Suppose that the nodes are in the usual `info-link` form, and `info` is of type `int`. Let us assume that we process the following data:

```
2 15 8 24 34
```

We need three pointers to build the list: one to point to the first node in the list, which cannot be moved; one to point to the last node in the list; and one to create the new node. Consider the following variable declaration:

```
nodeType *first, *last, *newNode;
int num;
```

Suppose that first points to the first node in the list. Initially, the list is empty, so both first and last are NULL. Thus, we must have the statements:

```
first = NULL;
last = NULL;
```

to initialize first and last to NULL.

Next, consider the following statements:

```
1   cin >> num;               //read and store a number in num
2   newNode = new nodeType;   //allocate memory of type nodeType
                              //and store the address of the
                              //allocated memory in newNode
3   newNode->info = num;      //copy the value of num into the
                              //info field of newNode
4   newNode->link = NULL;     //initialize the link field of
                              //newNode to NULL
5   if (first == NULL)        //if first is NULL, the list is empty;
                              //make first and last point to newNode
    {
5a     first = newNode;
5b     last = newNode;
    }
6   else                      //list is not empty
    {
6a     last->link = newNode;  //insert newNode at the end of the list
6b     last = newNode;         //set last so that it points to the
                               //actual last node in the list
    }
```

Let us now execute these statements. Initially, both first and last are NULL. Therefore, we have the list as shown in Figure 18-12.

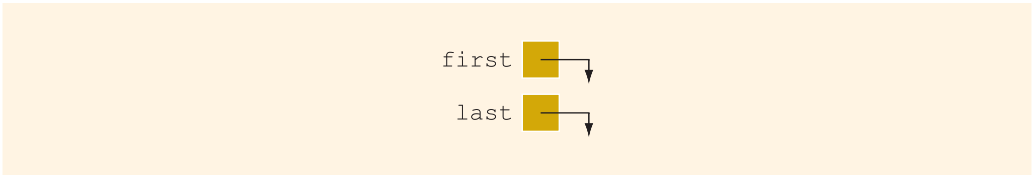


FIGURE 18-12 Empty list

After statement 1 executes, num is 2. Statement 2 creates a node and stores the address of that node in newNode. Statement 3 stores 2 in the info field of newNode, and statement 4 stores NULL in the link field of newNode (see Figure 18-13).

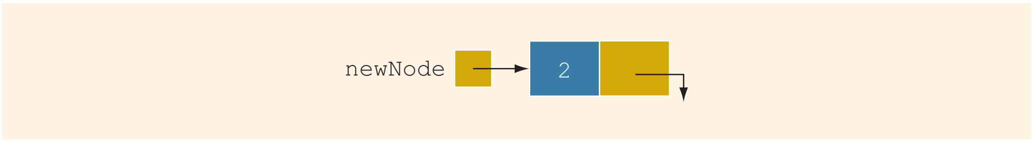


FIGURE 18-13 newNode with info 2

Because `first` is `NULL`, we execute statements 5a and 5b. Figure 18-14 shows the resulting list.

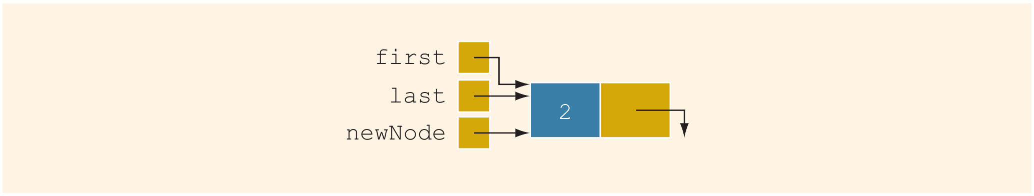


FIGURE 18-14 List after inserting `newNode` in it

We now repeat statements 1 through 6b. After statement 1 executes, `num` is `15`. Statement 2 creates a node and stores the address of this node in `newNode`. Statement 3 stores `15` in the `info` field of `newNode`, and statement 4 stores `NULL` in the link field of `newNode` (see Figure 18-15).

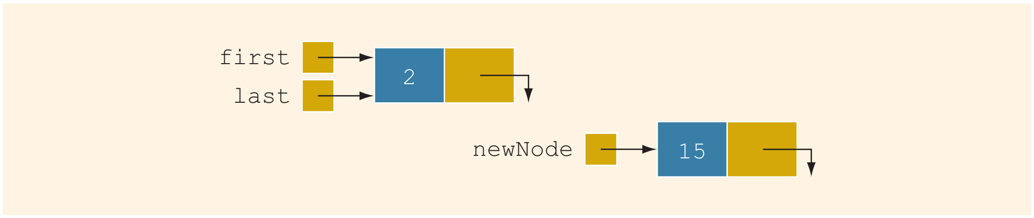


FIGURE 18-15 List and `newNode` with `info` `15`

Because `first` is not `NULL`, we execute statements 6a and 6b. Figure 18-16 shows the resulting list.

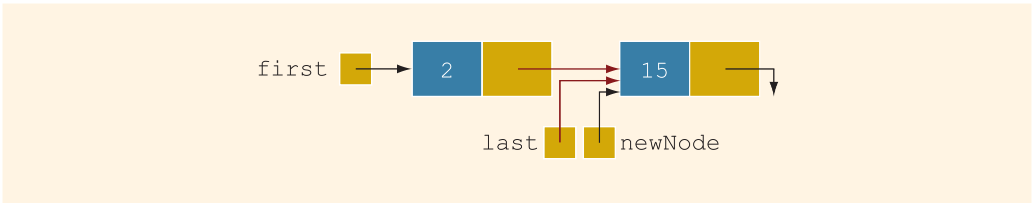


FIGURE 18-16 List after inserting `newNode` at the end

We now repeat statements 1 through 6b three more times. Figure 18-17 shows the resulting list.

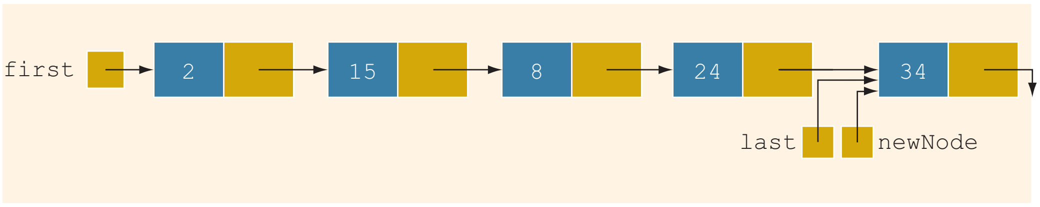


FIGURE 18-17 List after inserting 8, 24, and 34

To build the linked list, we can put the previous statements in a loop and execute the loop until certain conditions are met. We can, in fact, write a C++ function to build a linked list.

Suppose that we read a list of integers ending with -999. The following function, buildListForward, builds a linked list (in a forward manner) and returns the pointer of the built list.

```
nodeType* buildListForward()
{
    nodeType *first, *newNode, *last;
    int num;

    cout << "Enter a list of integers ending with -999."
         << endl;
    cin >> num;
    first = NULL;

    while (num != -999)
    {
        newNode = new nodeType;
        newNode->info = num;
        newNode->link = NULL;

        if (first == NULL)
        {
            first = newNode;
            last = newNode;
        }
        else
        {
            last->link = newNode;
            last = newNode;
        }
        cin >> num;
    } //end while

    return first;
} //end buildListForward
```

BUILDING A LINKED LIST BACKWARD

Now we consider the case of building a linked list backward. For the previously given data—2, 15, 8, 24, and 34—the linked list is as shown in Figure 18-18.

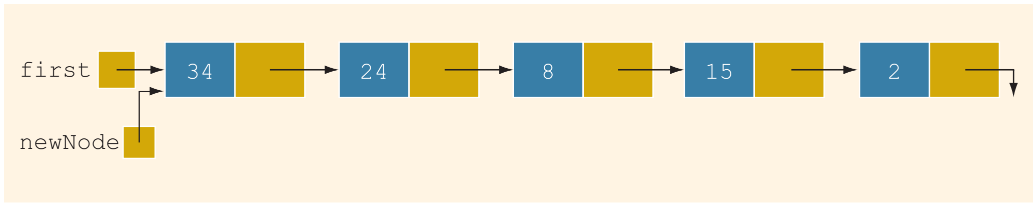


FIGURE 18-18 List after building it backward

Because the new node is always inserted at the beginning of the list, we do not need to know the end of the list, so the pointer `last` is not needed. Also, after inserting the new node at the beginning, the new node becomes the first node in the list. Thus, we need to update the value of the pointer `first` to correctly point to the first node in the list. We see, then, that we need only two pointers to build the linked list: one to point to the list and one to create the new node. Because initially the list is empty, the pointer `first` must be initialized to `NULL`. In pseudocode, the algorithm is:

```
1. Initialize first to NULL.
2. For each item in the list,
    a. Create the new node, newNode.
    b. Store the item in newNode.
    c. Insert newNode before first.
    d. Update the value of the pointer first.
```

The following C++ function builds the linked list backward and returns the pointer of the built list.

```
nodeType* buildListBackward()
{
    nodeType *first, *newNode;
    int num;

    cout << "Enter a list of integers ending with -999."
         << endl;
    cin >> num;
    first = NULL;

    while (num != -999)
    {
        newNode = new nodeType;   //create a node
        newNode->info = num;      //store the data in newNode
        newNode->link = first;    //put newNode at the beginning
```

```
                                        //of the list
        first = newNode;                //update the head pointer of
                                        //the list, that is, first
        cin >> num;                     //read the next number
    }

    return first;
} //end buildListBackward
```

Linked List as an ADT

The previous sections taught you the basic properties of linked lists and how to construct and manipulate them. Because a linked list is a very important data structure, rather than discuss specific lists such as a list of integers or a list of strings, this section discusses linked lists as an abstract data type (ADT). Using templates, this section gives a generic definition of linked lists, which is then used in the next section and later in this book. The programming example at the end of this chapter also uses this generic definition of linked lists.

The basic operations on linked lists are:

1. Initialize the list.
2. Determine whether the list is empty.
3. Print the list.
4. Find the length of the list.
5. Destroy the list.
6. Retrieve the `info` contained in the first node.
7. Retrieve the `info` contained in the last node.
8. Search the list for a given item.
9. Insert an item in the list.
10. Delete an item from the list.
11. Make a copy of the linked list.

In general, there are two types of linked lists—sorted lists, whose elements are arranged according to some criteria, and unsorted lists, whose elements are in no particular order. The algorithms to implement the operations search, insert, and remove slightly differ for sorted and unsorted lists. Therefore, we will define the `class linkedListType` to implement the basic operations on a linked list as an `abstract class`. Using the principle of inheritance, we, in fact, will derive two `class`es—`unorderedLinkedList` and `orderedLinkedList`—from the `class linkedListType`.

Objects of the `class unorderedLinkedList` would arrange list elements in no particular order, that is, these lists may not be sorted. On the other hand, objects of the `class orderedLinkedList` would arrange elements according to some comparison criteria, usually less than or equal to. That is, these lists will be in ascending order. Moreover, after inserting an element into or removing an element from an ordered list, the resulting list will be ordered.

If a linked list is unordered, we can insert a new item at either the end or the beginning. Furthermore, you can build such a list in either a forward manner or a backward manner. The function `buildListForward` inserts the new item at the end, whereas the function `buildListBackward` inserts the new item at the beginning. To accommodate both operations, we will write two functions: `insertFirst` to insert the new item at the beginning of the list and `insertLast` to insert the new item at the end of the list. Also, to make the algorithms more efficient, we will use two pointers in the list: `first`, which points to the first node in the list, and `last`, which points to the last node in the list.

Structure of Linked List Nodes

Recall that each node of a linked list must store the data as well as the address for the next node in the list (except the last node of the list). Therefore, the node has two member variables. To simplify operations such as insert and delete, we define the class to implement the node of a linked list as a **`struct`**. The definition of the **`struct`** `nodeType` is:

```
//Definition of the node

template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};
```

NOTE The class to implement the node of a linked list is declared as a **`struct`**. Programming Exercise 9, at the end of this chapter, asks you to redefine the class to implement the nodes of a linked list so that the member variables of the **`class`** `nodeType` are **`private`**.

Member Variables of the class linkedListType

To maintain a linked list, we use two pointers: `first` and `last`. The pointer `first` points to the first node in the list, and `last` points to the last node in the list. We also keep a count of the number of nodes in the list. Therefore, the **`class`** `linkedListType` has three member variables, as follows:

```
protected:
    int count;    //variable to store the number of
                  //elements in the list
    nodeType<Type> *first; //pointer to the first node
                           //of the list
    nodeType<Type> *last;  //pointer to the last node
                           //of the list
```

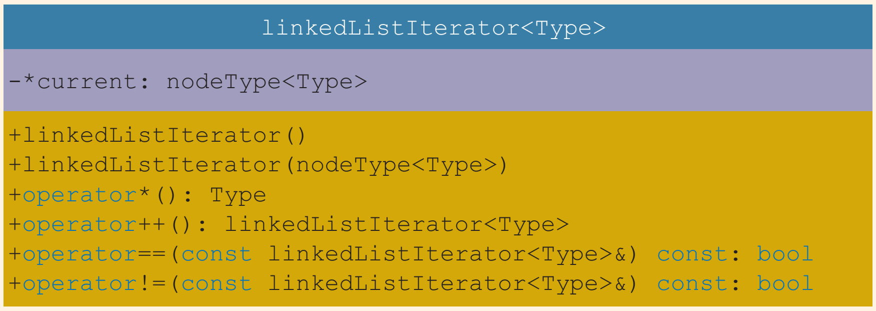
Linked List Iterators

One of the basic operations performed on a list is to process each node of the list. This requires the list to be traversed, starting at the first node. Moreover, a specific application requires each node to be processed in a very specific way. A common technique to accomplish this is to provide an iterator. So what is an iterator? An **iterator** is an object that produces each element of a container, such as a linked list, one element at a time. The two most common operations on iterators are ++ (the increment operator) and * (the dereferenceing operator). The increment operator advances the iterator to the next node in the list, and the dereferencing operator returns the info of the current node.

Note that an iterator is an object. So we need to define a class, which we will call `linkedListIterator`, to create iterators to objects of the `class linkedListType`. The iterator class would have one member variable pointing to (the current) node.

```
template <class Type>
class linkedListIterator
{
public:
    linkedListIterator();
      //Default constructor.
      //Postcondition: current = NULL;

    linkedListIterator(nodeType<Type> *ptr);
      //Constructor with a parameter.
      //Postcondition: current = ptr;

    Type operator*();
      //Function to overload the dereferencing operator *.
      //Postcondition: Returns the info contained in the node.

    linkedListIterator<Type> operator++();
      //Overload the pre-increment operator.
      //Postcondition: The iterator is advanced to the next
      //               node.

    bool operator==(const linkedListIterator<Type>& right) const;
      //Overload the equality operator.
      //Postcondition: Returns true if this iterator is equal to
      //               the iterator specified by right,
      //               otherwise it returns false.

    bool operator!=(const linkedListIterator<Type>& right) const;
      //Overload the not equal to operator.
      //Postcondition: Returns true if this iterator is not equal
      //               to the iterator specified by right,
      //               otherwise it returns false.

private:
    nodeType<Type> *current; //pointer to point to the current
                             //node in the linked list
};
```

Figure 18-19 shows the UML class diagram of the **class** `linkedListIterator`.



```
linkedListIterator<Type>

-*current: nodeType<Type>

+linkedListIterator()
+linkedListIterator(nodeType<Type>)
+operator*(): Type
+operator++(): linkedListIterator<Type>
+operator==(const linkedListIterator<Type>&) const: bool
+operator!=(const linkedListIterator<Type>&) const: bool
```

FIGURE 18-19 UML class diagram of the **class** `linkedListIterator`

The definitions of the functions of the **class** `linkedListIterator` are:

```
template <class Type>
linkedListIterator<Type>::linkedListIterator()
{
    current = NULL;
}

template <class Type>
linkedListIterator<Type>::
                 linkedListIterator(nodeType<Type> *ptr)
{
    current = ptr;
}

template <class Type>
Type linkedListIterator<Type>::operator*()
{
    return current->info;
}

template <class Type>
linkedListIterator<Type> linkedListIterator<Type>::operator++()
{
    current = current->link;

    return *this;
}

template <class Type>
bool linkedListIterator<Type>::operator==
                (const linkedListIterator<Type>& right) const
{
    return (current == right.current);
}
```

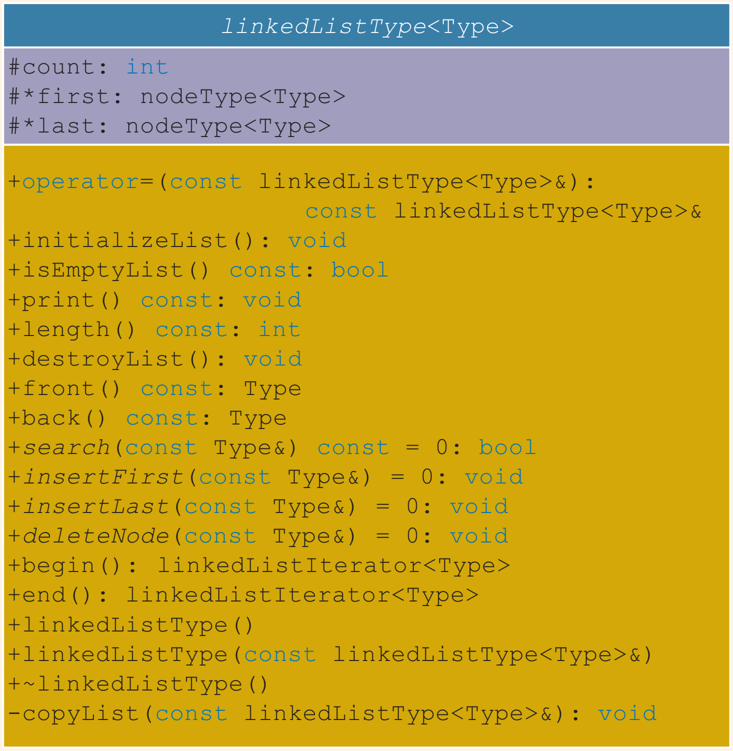
```
template <class Type>
bool linkedListIterator<Type>::operator!=
                  (const linkedListIterator<Type>& right) const
{
    return (current != right.current);
}
```

Now that we have defined the classes to implement the node of a linked list and an iterator to a linked list, next we describe the `class linkedListType` to implement the basic properties of a linked list.

The following abstract class defines the basic properties of a linked list as an ADT.

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

    void initializeList();
      //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 it returns false.

    void print() const;
      //Function to output the data contained in each node.
      //Postcondition: none

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

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

    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.
    virtual bool search(const Type& searchItem) const = 0;
      //Function to determine whether searchItem is in the list.
      //Postcondition: Returns true if searchItem is in the
      //               list, otherwise the value false is
      //               returned.

    virtual void insertFirst(const Type& newItem) = 0;
      //Function to insert newItem at the beginning of the list.
      //Postcondition: first points to the new list, newItem is
      //               inserted at the beginning of the list,
      //               last points to the last node in the list,
      //               and count is incremented by 1.

    virtual void insertLast(const Type& newItem) = 0;
      //Function to insert newItem at the end of the list.
      //Postcondition: first points to the new list, newItem
      //               is inserted at the end of the list,
      //               last points to the last node in the list,
      //               and count is incremented by 1.

    virtual void deleteNode(const Type& deleteItem) = 0;
      //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, last
      //               points to the last node of the updated
      //               list, and count is decremented by 1.

    linkedListIterator<Type> begin();
      //Function to return an iterator at the begining of the
      //linked list.
      //Postcondition: Returns an iterator such that current is
      //               set to first.

    linkedListIterator<Type> end();
      //Function to return an iterator one element past the
      //last element of the linked list.
      //Postcondition: Returns an iterator such that current is
      //               set to NULL.

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

    linkedListType(const linkedListType<Type>& otherList);
      //copy constructor

    ~linkedListType();
      //destructor
      //Deletes all the nodes from the list.
      //Postcondition: The list object is destroyed.

protected:
    int count;   //variable to store the number of
                 //elements in the list
    nodeType<Type> *first; //pointer to the first node of the list
    nodeType<Type> *last;  //pointer to the last node of the list

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

Figure 18-20 shows the UML class diagram of the **class** linkedListType.



```
linkedListType<Type>
---------------------------------------------
#count: int
#*first: nodeType<Type>
#*last: nodeType<Type>
---------------------------------------------
+operator=(const linkedListType<Type>&):
                    const linkedListType<Type>&
+initializeList(): void
+isEmptyList() const: bool
+print() const: void
+length() const: int
+destroyList(): void
+front() const: Type
+back() const: Type
+search(const Type&) const = 0: bool
+insertFirst(const Type&) = 0: void
+insertLast(const Type&) = 0: void
+deleteNode(const Type&) = 0: void
+begin(): linkedListIterator<Type>
+end(): linkedListIterator<Type>
+linkedListType()
+linkedListType(const linkedListType<Type>&)
+~linkedListType()
-copyList(const linkedListType<Type>&): void
```

FIGURE 18-20 UML class diagram of the **class** linkedListType

Note that typically, in the UML diagram, the name of an abstract class and abstract function is shown in italics.

The instance variables first and last, as defined earlier, of the **class** linkedListType are **protected**, not **private**, because as noted previously, we will derive the **class**es unorderedLinkedList and orderedLinkedList from the **class** linkedListType. Because each of the **class**es unorderedLinkedList and orderedLinkedList will provide separate definitions of the functions search, insertFirst, insertLast, and deleteNode and because these functions would access the instance variable, to provide direct access to the instance variables, the instance variables are declared as **protected**.

The definition of the **class** linkedListType includes a member function to overload the assignment operator. For classes that include pointer data members, the assignment operator must be explicitly overloaded (see Chapters 14 and 15). For the same reason, the definition of the class also includes a copy constructor.

Notice that the definition of the **class** linkedListType contains the member function copyList, which is declared as a **private** member. This is due to the fact that this function is used only to implement the copy constructor and overload the assignment operator.

Next, we write the definitions of the nonabstract functions of the **class** LinkedListClass.

The list is empty if first is NULL. Therefore, the definition of the function isEmptyList to implement this operation is as follows:

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

DEFAULT CONSTRUCTOR

The default constructor, linkedListType, is quite straightforward. It simply initializes the list to an empty state. Recall that when an object of the linkedListType type is declared and no value is passed, the default constructor is executed automatically.

```
template <class Type>
linkedListType<Type>::linkedListType() //default constructor
{
    first = NULL;
    last = NULL;
    count = 0;
}
```

DESTROY THE LIST

The function destroyList deallocates the memory occupied by each node. We traverse the list starting from the first node and deallocate the memory by calling the operator **delete**. We need a temporary pointer to deallocate the memory. Once the entire list is destroyed, we must set the pointers first and last to NULL and count to 0.

```
template <class Type>
void linkedListType<Type>::destroyList()
{
    nodeType<Type> *temp;   //pointer to deallocate the memory
                            //occupied by the node
    while (first != NULL)   //while there are nodes in the list
    {
        temp = first;         //set temp to the current node
        first = first->link; //advance first to the next node
        delete temp;   //deallocate the memory occupied by temp
    }

    last = NULL; //initialize last to NULL; first has already
                 //been set to NULL by the while loop
    count = 0;
}
```

INITIALIZE THE LIST

The function initializeList initializes the list to an empty state. Note that the default constructor or the copy constructor has already initialized the list when the list object was declared. This operation, in fact, reinitializes the list to an empty state, so it must delete the nodes (if any) from the list. This task can be accomplished by using the destroyList operation, which also resets the pointers first and last to NULL and sets count to 0.

```
template <class Type>
void linkedListType<Type>::initializeList()
{
    destroyList(); //if the list has any nodes, delete them
}
```

Print the List

The member function print prints the data contained in each node. To do so, we must traverse the list, starting at the first node. Because the pointer first always points to the first node in the list, we need another pointer to traverse the list. (If we use first to traverse the list, the entire list will be lost.)

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

    current = first;    //set current so that it points to
                        //the first node
    while (current != NULL) //while more data to print
    {
        cout << current->info << " ";
        current = current->link;
    }
}//end print
```

Length of a 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 linkedListType<Type>::length() const
{
    return count;
}
```

Retrieve the Data of the First Node

The function `front` returns the `info` contained in the first node, and its definition is straightforward:

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

    return first->info; //return the info of the first node
}//end front
```

Notice that if the list is empty, the `assert` statement terminates the program. Therefore, before calling this function, check to see whether the list is nonempty.

Retrieve the Data of the Last Node

The function `back` returns the `info` contained in the last node, and its definition is straightforward:

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

    return last->info; //return the info of the last node
}//end back
```

Notice that if the list is empty, the `assert` statement terminates the program. Therefore, before calling this function, check to see whether the list is nonempty.

Begin and End

The function `begin` returns an iterator to the first node in the linked list, and the function `end` returns an iterator to one past the last node in the linked list. Their definitions are:

```
template <class Type>
linkedListIterator<Type> linkedListType<Type>::begin()
{
    linkedListIterator<Type> temp(first);

    return temp;
}
```

```
template <class Type>
linkedListIterator<Type> linkedListType<Type>::end()
{
    linkedListIterator<Type> temp(NULL);

    return temp;
}
```

Copy the List

The function copyList makes an identical copy of a linked list. Therefore, we traverse the list to be copied, starting at the first node. Corresponding to each node in the original list, we:

a. Create a node, and call it newNode.

b. Copy the info of the node (in the original list) into newNode.

c. Insert newNode at the end of the list being created.

The definition of the function copyList is:

```
template <class Type>
void linkedListType<Type>::copyList
                   (const linkedListType<Type>& otherList)
{
    nodeType<Type> *newNode; //pointer to create a node
    nodeType<Type> *current; //pointer to traverse the list

    if (first != NULL) //if the list is nonempty, make it empty
        destroyList();

    if (otherList.first == NULL) //otherList is empty
    {
        first = NULL;
        last = NULL;
        count = 0;
    }
    else
    {
        current = otherList.first; //current points to the
                                   //list to be copied
        count = otherList.count;

            //copy the first node
        first = new nodeType<Type>;  //create the node
        first->info = current->info; //copy the info
        first->link = NULL;        //set the link field of
                                   //the node to NULL
        last = first;              //make last point to the
                                   //first node
        current = current->link;     //make current point to
                                     //the next node
```

```
           //copy the remaining list
        while (current != NULL)
        {
            newNode = new nodeType<Type>;  //create a node
            newNode->info = current->info; //copy the info
            newNode->link = NULL;       //set the link of
                                        //newNode to NULL
            last->link = newNode;  //attach newNode after last
            last = newNode;        //make last point to
                                   //the actual last node
            current = current->link;   //make current point
                                       //to the next node
        }//end while
    }//end else
}//end copyList
```

Destructor

The destructor deallocates the memory occupied by the nodes of a list when the class object goes out of scope. Because memory is allocated dynamically, resetting the pointers `first` and `last` does not deallocate the memory occupied by the nodes in the list. We must traverse the list, starting at the first node, and delete each node in the list. The list can be destroyed by calling the function `destroyList`. Therefore, the definition of the destructor is:

```
template <class Type>
linkedListType<Type>::~linkedListType() //destructor
{
    destroyList();
}
```

Copy Constructor

Because the **class** `linkedListType` contains pointer data members, the definition of this class contains the copy constructor. Recall that if a formal parameter is a value parameter, the copy constructor provides the formal parameter with its own copy of the data. The copy constructor also executes when an object is declared and initialized using another object. (For more information, see Chapter 14.)

The copy constructor makes an identical copy of the linked list. This can be done by calling the function `copyList`. Because the function `copyList` checks whether the original is empty by checking the value of `first`, we must first initialize the pointer `first` to `NULL` before calling the function `copyList`.

The definition of the copy constructor is:

```
template <class Type>
linkedListType<Type>::linkedListType
                     (const linkedListType<Type>& otherList)
{
    first = NULL;
    copyList(otherList);
}//end copy constructor
```

Overloading the Assignment Operator

The definition of the function to overload the assignment operator for the class linkedListType is similar to the definition of the copy constructor. We give its definition for the sake of completeness.

```
         //overload the assignment operator
template <class Type>
const linkedListType<Type>& linkedListType<Type>::operator=
                      (const linkedListType<Type>& otherList)
{
    if (this != &otherList) //avoid self-copy
    {
        copyList(otherList);
    }//end else

     return *this;
}
```

Unordered Linked Lists

As described in the preceding section, we derive the class unorderedLinkedList from the abstract class linkedListType and implement the operations search, insertFirst, insertLast, and deleteNode.

The following class defines an unordered linked list as an ADT.

```
template <class Type>
class unorderedLinkedList: public linkedListType<Type>
{
public:
    bool search(const Type& searchItem) const;
      //Function to determine whether searchItem is in the list.
      //Postcondition: Returns true if searchItem is in the
      //               list, otherwise the value false is
      //               returned.

    void insertFirst(const Type& newItem);
      //Function to insert newItem at the beginning of the list.
      //Postcondition: first points to the new list, newItem is
      //               inserted at the beginning of the list,
      //               last points to the last node in the
      //               list, and count is incremented by 1.

    void insertLast(const Type& newItem);
      //Function to insert newItem at the end of the list.
      //Postcondition: first points to the new list, newItem
      //               is inserted at the end of the list,
      //               last points to the last node in the
      //               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, last
      //                points to the last node of the updated
      //                list, and count is decremented by 1.
};
```

Figure 18-21 shows a UML class diagram of the class unorderedLinkedList and the inheritance hierarchy.

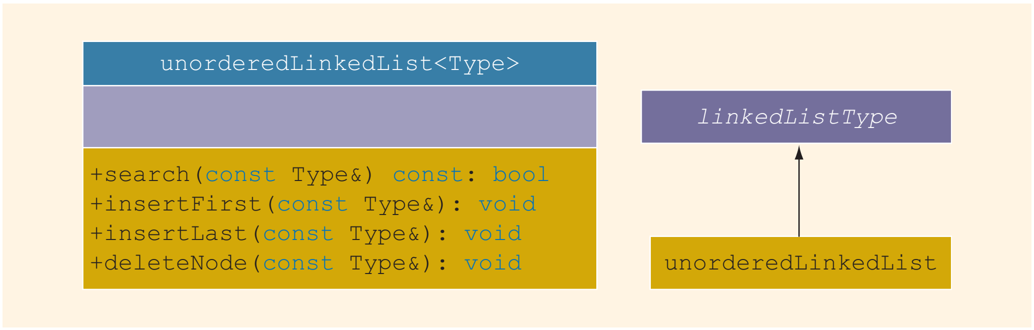

unorderedLinkedList<Type>
+search(const Type&) const: bool
+insertFirst(const Type&): void
+insertLast(const Type&): void
+deleteNode(const Type&): void
linkedListType
unorderedLinkedList

FIGURE 18-21 UML class diagram of the class unorderedLinkedList and inheritance hierarchy

Next, we give the definitions of the member functions of the class unorderedLinkedList.

Search the List

The member function search searches the list for a given item. If the item is found, it returns true; otherwise, it returns false. Because a linked list is not a random-access data structure, we must sequentially search the list, starting from the first node.

This function has the following steps:

1. Compare the search item with the current node in the list. If the info of the current node is the same as the search item, stop the search; otherwise, make the next node the current node.
2. Repeat Step 1 until either the item is found or no more data is left in the list to compare with the search item.

```
template <class Type>
bool unorderedLinkedList<Type>::
                   search(const Type& searchItem) const
{
    nodeType<Type> *current; //pointer to traverse the list
    bool found = false;
    current = first; //set current to point to the first
                     //node in the list
    while (current != NULL && !found)     //search the list
        if (current->info == searchItem) //searchItem is found
            found = true;
        else
            current = current->link; //make current point to
                                     //the next node

    return found;
}//end search
```

NOTE The function search can also be written as:

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

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

    while (current != NULL)                 //search the list
        if (current->info == searchItem)  //searchItem is found
            return true;
        else
            current = current->link; //make current point to
                                     //the next node

    return false; //searchItem is not in the list, return false
}//end search
```

Insert the First Node

The function `insertFirst` inserts the new item at the beginning of the list—that is, before the node pointed to by `first`. The steps needed to implement this function are as follows:

1. Create a new node.
2. Store the new item in the new node.
3. Insert the node before `first`.
4. Increment `count` by `1`.

```
template <class Type>
void unorderedLinkedList<Type>::insertFirst(const Type& newItem)
{
    nodeType<Type> *newNode; //pointer to create the new node
    newNode = new nodeType<Type>; //create the new node
    newNode->info = newItem;    //store the new item in the node
    newNode->link = first;      //insert newNode before first
    first = newNode;            //make first point to the
                                //actual first node
    count++;                    //increment count

    if (last == NULL)    //if the list was empty, newNode is also
                       //the last node in the list
        last = newNode;
}//end insertFirst
```

Insert the Last Node

The definition of the member function `insertLast` is similar to the definition of the member function `insertFirst`. Here, we insert the new node after `last`. Essentially, the function `insertLast` is:

```
template <class Type>
void unorderedLinkedList<Type>::insertLast(const Type& newItem)
{
    nodeType<Type> *newNode; //pointer to create the new node

    newNode = new nodeType<Type>; //create the new node
    newNode->info = newItem;    //store the new item in the node
    newNode->link = NULL;    //set the link field of newNode
                             //to NULL

    if (first == NULL)  //if the list is empty, newNode is
                        //both the first and last node
    {
        first = newNode;
        last = newNode;
        count++;        //increment count
    }
    else    //the list is not empty, insert newNode after last
    {
        last->link = newNode; //insert newNode after last
        last = newNode; //make last point to the actual
                        //last node in the list
        count++;        //increment count
    }
}//end insertLast
```

DELETE A NODE

Next, we discuss the implementation of the member function `deleteNode`, which deletes a node from the list with a given `info`. We need to consider several cases:

Case 1: The list is empty.

Case 2: The first node is the node with the given `info`. In this case, we need to adjust the pointer `first`.

Case 3: The node with the given info is somewhere in the list. If the node to be deleted is the last node, then we must adjust the pointer last.

Case 4: The list does not contain the node with the given info.

If list is empty, we can simply print a message indicating that the list is empty. If list is not empty, we search the list for the node with the given info and, if such a node is found, we delete this node. After deleting the node, count is decremented by 1. In pseudocode, the algorithm is:

```
if list is empty
   Output(cannot delete from an empty list);
else
{
   if the first node is the node with the given info
       adjust the head pointer, that is, first, and deallocate
       the memory;
   else
   {
       search the list for the node with the given info
       if such a node is found, delete it and adjust the
       values of last (if necessary) and count.
   }
}
```

Case 1: The list is empty.

If the list is empty, output an error message as shown in the pseudocode.

Case 2: The list is not empty. The node to be deleted is the first node.

This case has two scenarios: list has only one node, and list has more than one node. Consider the list with one node, as shown in Figure 18-22.

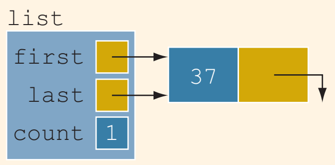


FIGURE 18-22 list with one node

Suppose that we want to delete 37. After deletion, the list becomes empty. Therefore, after deletion, both first and last are set to NULL, and count is set to 0.

Now consider the list of more than one node, as shown in Figure 18-23.

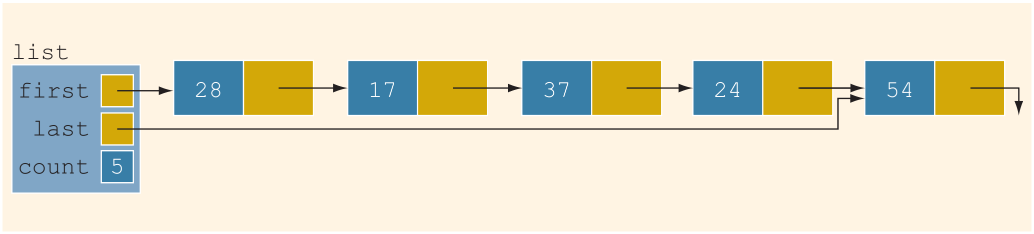


FIGURE 18-23 `list` with more than one node

Suppose that the node to be deleted is `28`. After deleting this node, the second node becomes the first node. Therefore, after deleting this node, the value of the pointer `first` changes; that is, after deletion, `first` contains the address of the node with `info` `17`, and `count` is decremented by `1`. Figure 18-24 shows the list after deleting `28`.

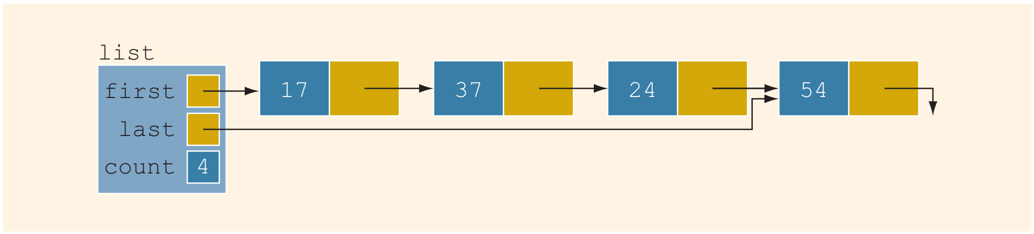


FIGURE 18-24 `list` after deleting node with `info` `28`

Case 3: The node to be deleted is not the first node but is somewhere in the list.

This case has two subcases: (a) the node to be deleted is not the last node, and (b) the node to be deleted is the last node. Let us illustrate both cases.

Case 3a: The node to be deleted is not the last node.

Consider the list shown in Figure 18-25.

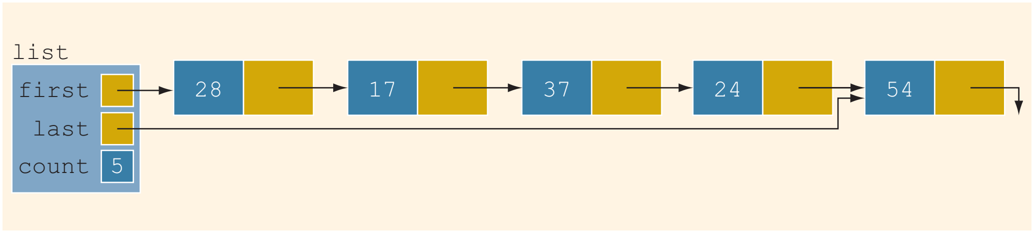


FIGURE 18-25 `list` before deleting `37`

Suppose that the node to be deleted is `37`. After deleting this node, the resulting list is as shown in Figure 18-26. (Notice that the deletion of `37` does not require us to change the values of `first` and `last`. The link field of the previous node—that is, `17`—changes. After deletion, the node with `info` `17` contains the address of the node with `24`.)

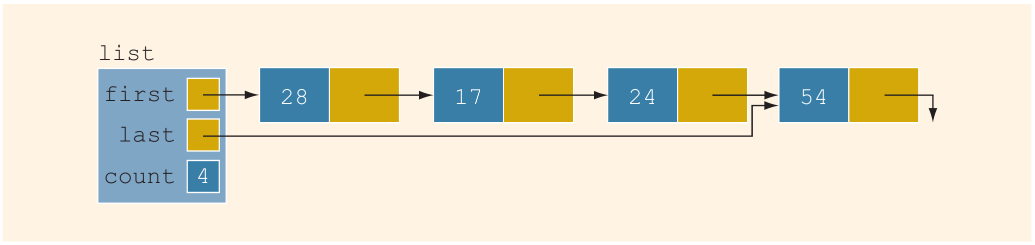


FIGURE 18-26 `list` after deleting `37`

Case 3b: The node to be deleted is the last node.

Consider the list shown in Figure 18-27. Suppose that the node to be deleted is `54`.

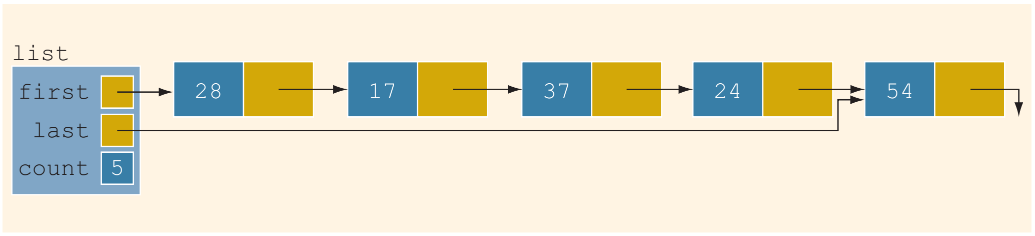


FIGURE 18-27 `list` before deleting `54`

After deleting `54`, the node with `info 24` becomes the last node. Therefore, the deletion of `54` requires us to change the value of the pointer `last`. After deleting `54`, `last` contains the address of the node with `info 24`. Also, `count` is decremented by `1`. Figure 18-28 shows the resulting list.

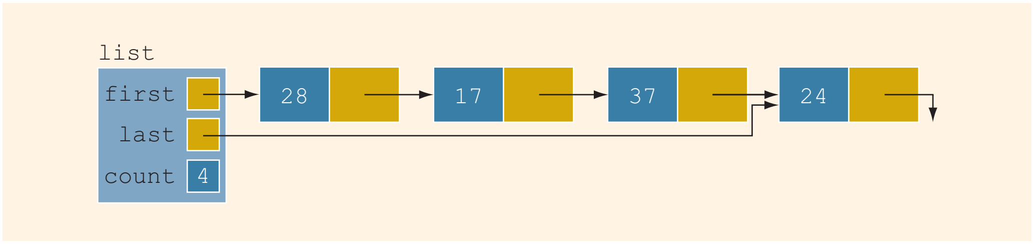


FIGURE 18-28 `list` after deleting `54`

Case 4: The node to be deleted is not in the list. In this case, the list requires no adjustment. We simply output an error message, indicating that the item to be deleted is not in the list.

From Cases 2, 3, and 4, it follows that the deletion of a node requires us to traverse the list. Because a linked list is not a random-access data structure, we must sequentially search the list. We handle Case 1 separately, because it does not require us to traverse the list. We sequentially search the list, starting at the second node. If the node to be deleted is in the middle of the list, we need to adjust the link field of the node just before the node to be deleted. Thus, we need a pointer to the previous node. When we search the list for the given `info`, we use two pointers: one to check the `info` of the current node and one to keep track of the node just before the current node. If the node to be deleted is the last node, we must adjust the pointer `last`.

The definition of the function `deleteNode` is:

```
template <class Type>
void unorderedLinkedList<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)    //Case 1; the list is empty
        cout << "Cannot delete from an empty list."
             << endl;
    else
    {
        if (first->info == deleteItem) //Case 2
        {
            current = first;
            first = first->link;
            count--;

            if (first == NULL)    //the list has only one node
                last = NULL;

            delete current;
        }
        else //search the list for the node with the given info
        {
            found = false;
            trailCurrent = first;  //set trailCurrent to point
                                   //to the first node
            current = first->link; //set current to point to
                                   //the second node

            while (current != NULL && !found)
            {
                if (current->info != deleteItem)
                {
                    trailCurrent = current;
                    current = current-> link;
                }
                else
                    found = true;
            }//end while
            if (found) //Case 3; if found, delete the node
            {
                trailCurrent->link = current->link;
                count--;

                if (last == current)   //node to be deleted
                                       //was the last node
                    last = trailCurrent; //update the value
                                         //of last
                delete current;  //delete the node from the list
            }
            else
                cout << "The item to be deleted is not in "
                     << "the list." << endl;
        }//end else
    }//end else
}//end deleteNode
```

Header File of the Unordered Linked List

For the sake of completeness, we will show how to create the header file that defines the class unorderedListType and the operations on such lists. (We assume that the definition of the class linkedListType and the definitions of the functions to implement the operations are in the header file linkedlist.h.)

```
#ifndef H_UnorderedLinkedList
#define H_UnorderedLinkedList

#include "linkedList.h"

using namespace std;

template <class Type>
class unorderedLinkedList: public linkedListType<Type>
{
public:
    bool search(const Type& searchItem) const;
      //Function to determine whether searchItem is in the list.
      //Postcondition: Returns true if searchItem is in the
      //               list, otherwise the value false is
      //               returned.

    void insertFirst(const Type& newItem);
      //Function to insert newItem at the beginning of the list.
      //Postcondition: first points to the new list, newItem is
      //               inserted at the beginning of the list,
      //               last points to the last node in the
      //               list, and count is incremented by 1.

    void insertLast(const Type& newItem);
      //Function to insert newItem at the end of the list.
      //Postcondition: first points to the new list, newItem
      //               is inserted at the end of the list,
      //               last points to the last node in the
      //               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, last
      //               points to the last node of the updated
      //               list, and count is decremented by 1.
};

//Place the definitions of the functions search,
//insertFirst, insertLast, and deleteNode here.
.
.
.
#endif
```

NOTE The Web site accompanying this book contains several programs illustrating how to use the `class unorderedLinkedList`.

Ordered Linked Lists

The preceding section described the operations on an unordered linked list. This section deals with ordered linked lists. As noted earlier, we derive the `class orderedLinkedList` from the `class linkedListType` and provide the definitions of the abstract functions `insertFirst`, `insertLast`, `search`, and `deleteNode` to take advantage of the fact that the elements of an ordered linked list are arranged using some ordering criteria. For simplicity, we assume that elements of an ordered linked list are arranged in ascending order.

Because the elements of an ordered linked list are in order, we include the function `insert` to insert an element in an ordered list at the proper place.

The following class defines an ordered linked list as an ADT:

```
template <class Type>
class orderedLinkedList: public linkedListType<Type>
{
public:
    bool search(const Type& searchItem) const;
      //Function to determine whether searchItem is in the list.
      //Postcondition: Returns true if searchItem is in the list,
      //               otherwise the value false is returned.

    void insert(const Type& newItem);
      //Function to insert newItem in the list.
      //Postcondition: first points to the new list, newItem
      //               is inserted at the proper place in the
      //               list, and count is incremented by 1.

    void insertFirst(const Type& newItem);
      //Function to insert newItem at the beginning of the list.
      //Postcondition: first points to the new list, newItem is
      //                 inserted at the proper place in the list,
      //                 last points to the last node in the
      //                 list, and count is incremented by 1.

    void insertLast(const Type& newItem);
      //Function to insert newItem at the end of the list.
      //Postcondition: first points to the new list, newItem is
      //                 inserted at the proper place in the list,
      //                 last points to the last node in the
      //                 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, and count is decremented by 1.
      //                 If deleteItem is not in the list, an
      //                 appropriate message is printed.
};
```

Figure 18-29 shows a UML class diagram of the `class orderedLinkedList` and the inheritance hierarchy.

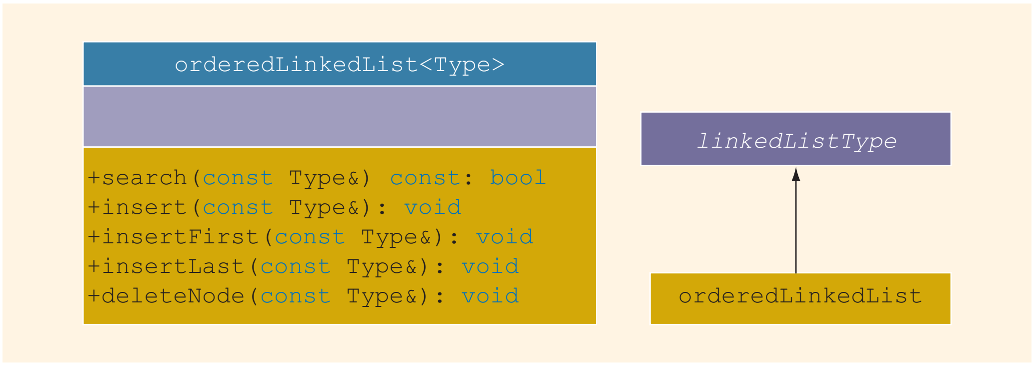

orderedLinkedList<Type>
+search(const Type&) const: bool
+insert(const Type&): void
+insertFirst(const Type&): void
+insertLast(const Type&): void
+deleteNode(const Type&): void
linkedListType
orderedLinkedList

FIGURE 18-29 UML class diagram of the `class orderedLinkedList` and the inheritance hierarchy

Next, we give the definitions of the member functions of the `class orderedLinkedList`.

Search the List

First, we discuss the search operation. The algorithm to implement the search operation is similar to the search algorithm for general lists discussed earlier. Here, because the list is sorted, we can improve the search algorithm somewhat. As before, we start the search at the first node in the list. We stop the search as soon as we find a node in the list with `info` greater than or equal to the search item or when we have searched the entire list.

The following steps describe this algorithm:

1. Compare the search item with the current node in the list. If the `info` of the current node is greater than or equal to the search item, stop the search; otherwise, make the next node the current node.
2. Repeat Step 1 until either an item in the list that is greater than or equal to the search item is found or no more data is left in the list to compare with the search item.

Note that the loop does not explicitly check whether the search item is equal to an item in the list. Thus, after the loop executes, we must check whether the search item is equal to the item in the list.

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

    current = first;  //start the search at the first node

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

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

    return found;
}//end search
```

Insert a Node

To insert an item in an ordered linked list, we first find the place where the new item is supposed to go, and then we insert the item in the list. To find the place for the new item, as before, we search the list. Here, we use two pointers, `current` and `trailCurrent`, to search the list. The pointer `current` points to the node whose `info` is being compared with the item to be inserted, and `trailCurrent` points to the node just before `current`. Because the list is in order, the search algorithm is the same as before. The following cases arise:

Case 1: The list is initially empty. The node containing the new item is the only node and thus the first node in the list.

Case 2: The new item is smaller than the smallest item in the list. The new item goes at the beginning of the list. In this case, we need to adjust the list's head pointer—that is, `first`. Also, `count` is incremented by `1`.

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

3a: The new item is larger than all of the items in the list. In this case, the new item is inserted at the end of the list. Thus, the value of `current` is `NULL`, and the new item is inserted after `trailCurrent`. Also, `count` is incremented by `1`.

3b: The new item is to be inserted somewhere in the middle of the list. In this case, the new item is inserted between `trailCurrent` and `current`. Also, `count` is incremented by `1`.

The following statements can accomplish both Cases 3a and 3b. Assume `newNode` points to the new node.

```
trailCurrent->link = newNode;
newNode->link = current;
```

Let us next illustrate these cases.

Case 1: The list is empty.

Consider the list shown in Figure 18-30(a).

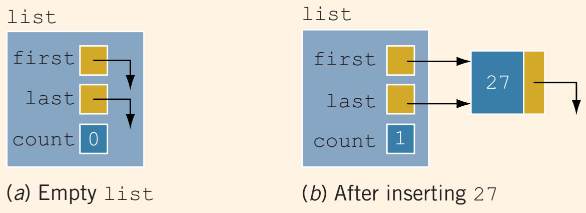


FIGURE 18-30 `list`

Suppose that we want to insert `27` in the list. To accomplish this task, we create a node, copy `27` into the node, set the link of the node to `NULL`, and make `first` point to the node. Figure 18-30(b) shows the resulting list. Notice that, after inserting `27`, the values of both `first` and `count` change.

Case 2: The list is not empty, and the item to be inserted is smaller than the smallest item in the list. Consider the list shown in Figure 18-31.

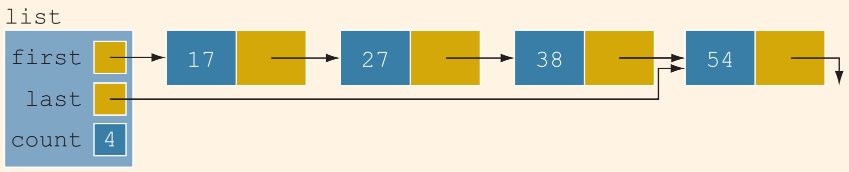


FIGURE 18-31 Nonempty `list` before inserting `10`

Suppose that 10 is to be inserted. After inserting 10 in the list, the node with info 10 becomes the first node of list. This requires us to change the value of first. Also, count is incremented by 1. Figure 18-32 shows the resulting list.

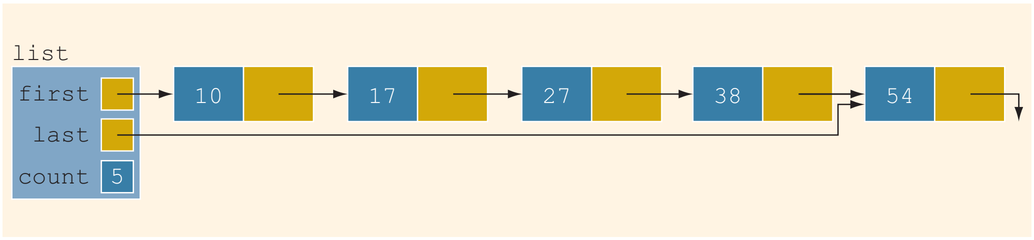


FIGURE 18-32 list after inserting 10

Case 3: The list is not empty, and the item to be inserted is larger than the first item in the list. As indicated previously, this case has two scenarios.

Case 3a: The item to be inserted is larger than the largest item in the list; that is, it goes at the end of the list. Consider the list shown in Figure 18-33.

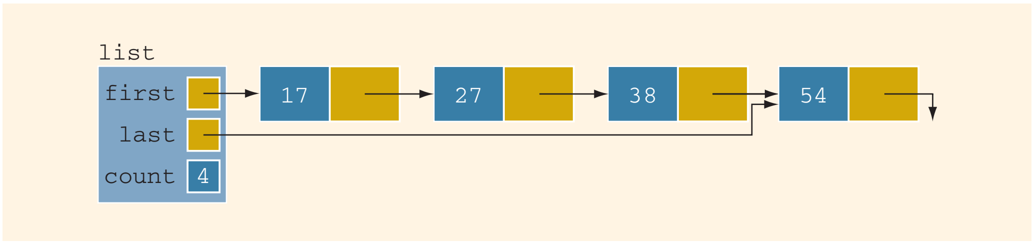


FIGURE 18-33 list before inserting 65

Suppose that we want to insert 65 in the list. After inserting 65, the resulting list is as shown in Figure 18-34.

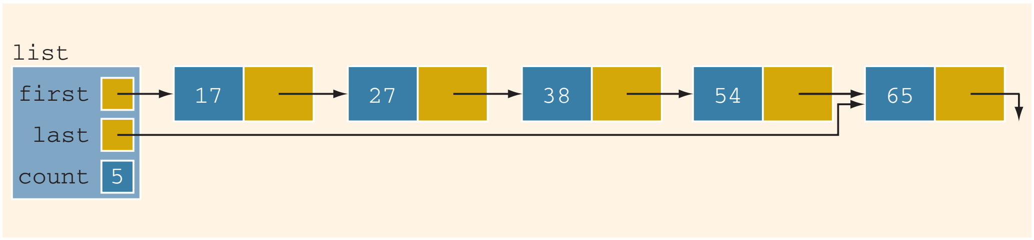


FIGURE 18-34 list after inserting 65

Case 3b: The item to be inserted goes somewhere in the middle of the list. Consider the list shown in Figure 18-35.

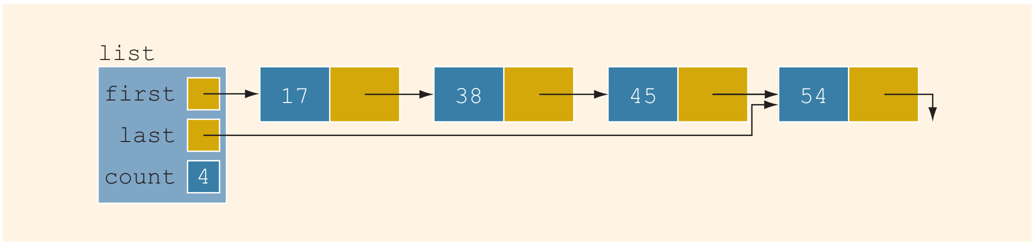


FIGURE 18-35 `list` before inserting `27`

Suppose that we want to insert `27` in this list. Clearly, `27` goes between `17` and `38`, which would require the link of the node with `info` `17` to be changed. After inserting `27`, the resulting list is as shown in Figure 18-36.

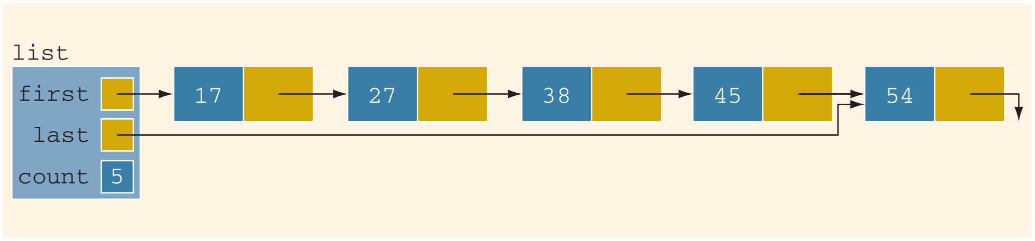


FIGURE 18-36 `list` after inserting `27`

From Case 3, it follows that we must first traverse the list to find the place where the new item is to be inserted. It also follows that we should traverse the list with two pointers—say, `current` and `trailCurrent`. The pointer `current` is used to traverse the list and compare the `info` of the node in the list with the item to be inserted. The pointer `trailCurrent` points to the node just before `current`. For example, in Case 3b, when the search stops, `trailCurrent` points to node `17` and `current` points to node `38`. The item is inserted after `trailCurrent`. In Case 3a, after searching the list to find the place for `65`, `trailCurrent` points to node `54` and `current` is `NULL`.

Essentially, the function `insert` is as follows:

```
template <class Type>
void orderedLinkedList<Type>::insert(const Type& newItem)
{
    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 = newItem;   //store newItem in the node
    newNode->link = NULL;      //set the link field of the node
                               //to NULL

    if (first == NULL)  //Case 1
    {
        first = newNode;
        last = newNode;
        count++;
    }
    else
    {
        current = first;
        found = false;

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

        if (current == first)      //Case 2
        {
            newNode->link = first;
            first = newNode;
            count++;
        }
        else                       //Case 3
        {
            trailCurrent->link = newNode;
            newNode->link = current;

            if (current == NULL)
                last = newNode;

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

Insert First and Insert Last

The function `insertFirst` inserts the new item at the beginning of the list. However, because the resulting list must be sorted, the new item must be inserted at the proper place. Similarly, the function `insertLast` must insert the new item at the proper place.

Therefore, we use the function `insertNode` to insert the new item at its proper place. The definitions of these functions are:

```
template <class Type>
void orderedLinkedList<Type>::insertFirst(const Type& newItem)
{
    insert(newItem);
}//end insertFirst

template <class Type>
void orderedLinkedList<Type>::insertLast(const Type& newItem)
{
    insert(newItem);
}//end insertLast
```

Note that in reality, the functions `insertFirst` and `insertLast` do not apply to ordered linked lists because the new item must be inserted at the proper place in the list. However, you must provide its definition as these functions are declared as abstract in the parent class.

Delete a Node

To delete a given item from an ordered linked list, first we search the list to see whether the item to be deleted is in the list. The function to implement this operation is the same as the delete operation on general linked lists. Here, because the list is sorted, we can somewhat improve the algorithm for ordered linked lists.

As in the case of `insertNode`, we search the list with two pointers, `current` and `trailCurrent`. Similar to the operation `insertNode`, several cases arise:

Case 1: The list is initially empty. We have an error. We cannot delete from an empty list.

Case 2: The item to be deleted is contained in the first node of the list. We must adjust the head pointer of the list—that is, `first`.

Case 3: The item to be deleted is somewhere in the list. In this case, `current` points to the node containing the item to be deleted, and `trailCurrent` points to the node just before the node pointed to by `current`.

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

After deleting a node, `count` is decremented by `1`. The definition of the function `deleteNode` is:

```
template <class Type>
void orderedLinkedList<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) //Case 1
        cout << "Cannot delete from an empty list." << endl;
    else
    {
        current = first;
        found = false;

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

        if (current == NULL)   //Case 4
            cout << "The item to be deleted is not in the "
                 << "list." << endl;
        else
            if (current->info == deleteItem) //the item to be
                                   //deleted is in the list
            {
                if (first == current)       //Case 2
                {
                    first = first->link;

                    if (first == NULL)
                        last = NULL;

                    delete current;
                }
                else                         //Case 3
                {
                    trailCurrent->link = current->link;

                    if (current == last)
                        last = trailCurrent;

                    delete current;
                }
                count--;
            }
            else                            //Case 4
                cout << "The item to be deleted is not in the "
                     << "list." << endl;
    }
}//end deleteNode
```

Header File of the Ordered Linked List

For the sake of completeness, we will show how to create the header file that defines the `class orderedListType`, as well as the operations on such lists. (We assume that the definition of the `class` `linkedListType` and the definitions of the functions to implement the operations are in the header file `linkedlist.h`.)

```
#ifndef H_orderedListType
#define H_orderedListType

#include "linkedList.h"

using namespace std;

template <class Type>
class orderedLinkedList: public linkedListType<Type>
{
public:
    bool search(const Type& searchItem) const;
      //Function to determine whether searchItem is in the list.
      //Postcondition: Returns true if searchItem is in the list,
      //               otherwise the value false is returned.

    void insert(const Type& newItem);
      //Function to insert newItem in the list.
      //Postcondition: first points to the new list, newItem
      //               is inserted at the proper place in the
      //               list, and count is incremented by 1.

    void insertFirst(const Type& newItem);
      //Function to insert newItem at the beginning of the list.
      //Postcondition: first points to the new list, newItem is
      //               inserted at the proper place in the list,
      //               last points to the last node in the
      //               list, and count is incremented by 1.

    void insertLast(const Type& newItem);
      //Function to insert newItem at the end of the list.
      //Postcondition: first points to the new list, newItem is
      //               inserted at the proper place in the list,
      //               last points to the last node in the
      //               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, and count is decremented by 1.
      //               If deleteItem is not in the list, an
      //               appropriate message is printed.
};

//Place the definitions of the functions search, insert,
//insertFirst, insertLast, and deleteNode here.
.
.
.
#endif
```

The following program tests various operations on an ordered linked list.

```
//Program to test the various operations on an ordered linked list

#include <iostream>
#include "orderedLinkedList.h"

using namespace std;

int main()
{
    orderedLinkedList<int> list1, list2;          //Line 1
    int num;                                      //Line 2

    cout << "Line 3: Enter numbers ending "
         << "with -999." << endl;                 //Line 3
    cin >> num;                                   //Line 4
    while (num != -999)                           //Line 5
    {
        list1.insert(num);                        //Line 6
        cin >> num;                               //Line 7
    }

    cout << endl;                                 //Line 8

    cout << "Line 9: list1: ";                    //Line 9
    list1.print();                                //Line 10
    cout << endl;                                 //Line 11

    list2 = list1; //test the assignment operator Line 12

    cout << "Line 13: list2: ";                   //Line 13
    list2.print();                                //Line 14
    cout << endl;                                 //Line 15

    cout << "Line 16: Enter the number to be "
         << "deleted: ";                          //Line 16
    cin >> num;                                   //Line 17
    cout << endl;                                 //Line 18

    list2.deleteNode(num);                        //Line 19

    cout << "Line 20: After deleting "
         << num << ", list2: " << endl;           //Line 20
    list2.print();                                //Line 21
    cout<<endl;                                   //Line 22

    return 0;
}
```

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

```
Line 3: Enter numbers ending with -999.
23 65 34 72 12 82 36 55 29 -999

Line 9: list1: 12 23 29 34 36 55 65 72 82
Line 13: list2: 12 23 29 34 36 55 65 72 82
Line 16: Enter the number to be deleted: 34

Line 20: 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 8 at the end of this chapter asks you 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.

Print a Linked List in Reverse Order (Recursion Revisited)

The nodes of an ordered list (as constructed previously) are in ascending order. Certain applications, however, might require the data to be printed in descending order, which means that we must print the list backward. We now discuss the function `reversePrint`. Given a pointer to a list, this function prints the elements of the list in reverse order.

Consider the linked list shown in Figure 18-37.

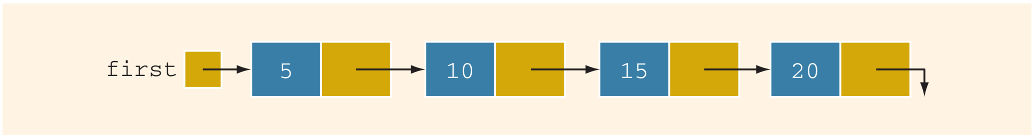


FIGURE 18-37 Linked list

For the list in Figure 18-37, the output should be in the following form:

```
20 15 10 5
```

Because the links are in only one direction, we cannot traverse the list backward starting from the last node. Let us see how we can effectively use recursion to print the list in reverse order.

Let us think in terms of recursion. We cannot print the info of the first node until we have printed the remainder of the list (that is, the tail of the first node). Similarly, we cannot print the info of the second node until we have printed the tail of the second node, and so on. Every time we consider the tail of a node, we reduce the size of the list by 1. Eventually, the size of the list will be reduced to zero, in which case the recursion will stop. Let us first write the algorithm in pseudocode. (Suppose that current is a pointer to a linked list.)

```
if (current != NULL)
{
    reversePrint(current->link);    //print the tail
    cout << current->info << endl; //print the node
}
```

Here, we do not see the base case; it is hidden. The list is printed only if the pointer to the list is not NULL. Also, in the body of the **if** statement, the recursive call is on the tail of the list. Because eventually the tail of the list will be empty, the **if** statement in the next call will fail, and the recursion will stop. Also, note that statements (for example, printing the info of the node) appear after the recursive call; thus, when the transfer comes back to the calling function, we must execute the remaining statements. Recall that the function exits only after the last statement executes. (By the "last statement," we do not mean the physical last statement, but rather the logical last statement.)

Let us write the previous function in C++ and then apply it to a list.

```
template <class Type>
void linkedListType<Type>::reversePrint
                         (nodeType<Type> *current) const
{
    if (current != NULL)
    {
        reversePrint(current->link);  //print the tail
        cout << current->info << " "; //print the node
    }
}
```

Consider the statement:

```
reversePrint(first);
```

in which first is a pointer of type nodeType<Type>.

Let us trace the execution of this statement, which is a function call, for the list shown in Figure 18-37. Because the formal parameter is a value parameter, the value of the actual parameter is passed to the formal parameter. See Figure 18-38.

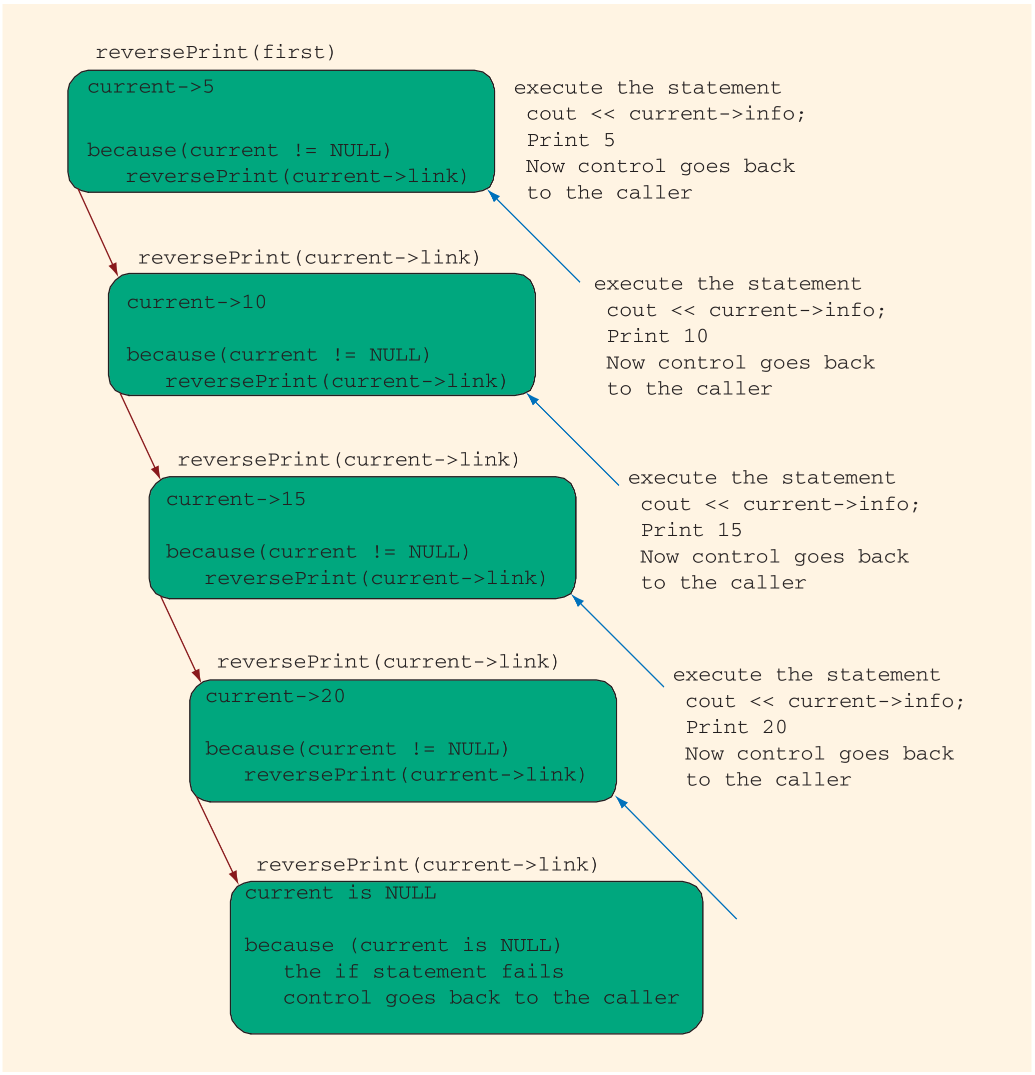


FIGURE 18-38 Execution of the statement `reversePrint(first);`

printListReverse

Now that we have written the function `reversePrint`, we can write the definition of the function `printListReverse`. Its definition is:

```
template <class Type>
void linkedListType<Type>::printListReverse() const
{
    reversePrint(first);
    cout << endl;
}
```

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 18-39).

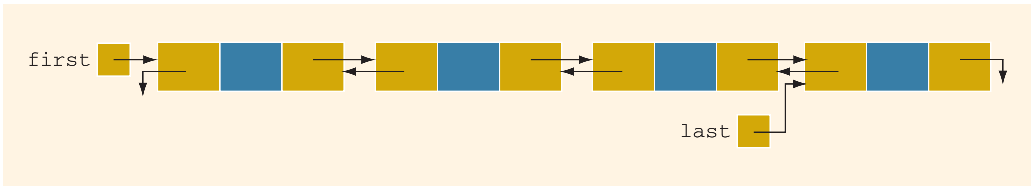


FIGURE 18-39 Doubly linked list

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:

1. Initialize the list.
2. Destroy the list.
3. Determine whether the list is empty.
4. Search the list for a given item.
5. Retrieve the first element of the list.
6. Retrieve the last element of the list.
7. Insert an item in the list.
8. Delete an item from the list.
9. Find the length of the list.
10. Print the list.
11. Make a copy of the doubly linked list.

Next, we describe these operations for an ordered doubly linked list. The following class defines a doubly linked list as an ADT.

```
  //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.

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 11 at the end of this chapter.) Moreover, 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 of 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:

```
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:

```
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:

```
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 18-40.

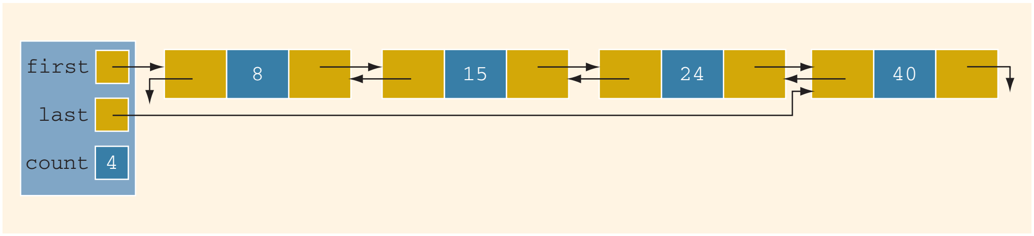


FIGURE 18-40 Doubly linked list before inserting `20`

Suppose that `20` is to be inserted in the list. After inserting `20`, the resulting list is as shown in Figure 18-41.

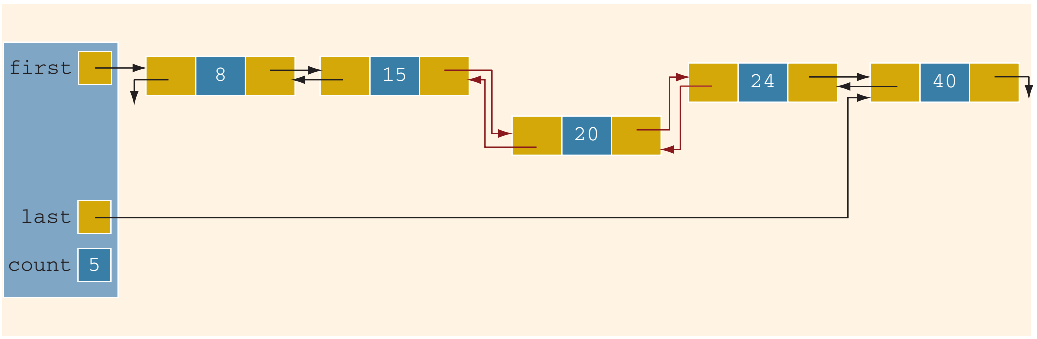


FIGURE 18-41 Doubly linked list after inserting 20

From Figure 18-41, 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:

```
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 the 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 `insertNode` 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 18-42.

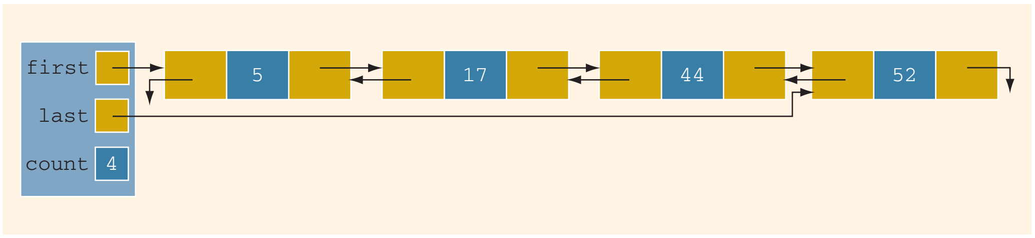


FIGURE 18-42 Doubly linked list before deleting 17

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 18-43).

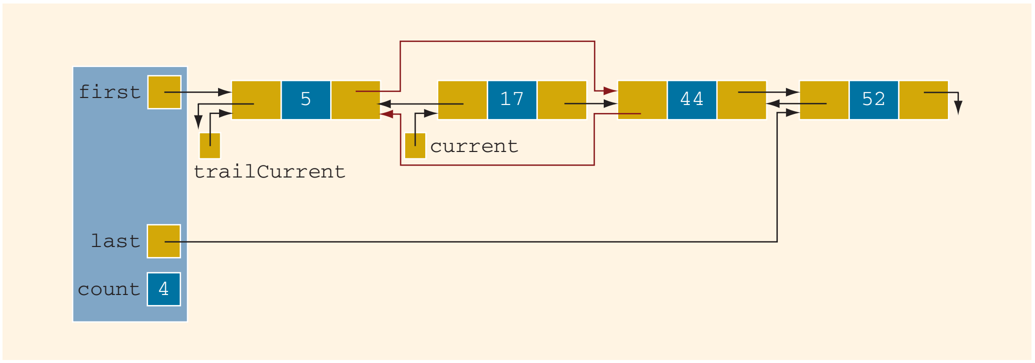


FIGURE 18-43 List after adjusting the links of the nodes before and after the node with info 17

Next, we delete the node pointed to by current (see Figure 18-44).

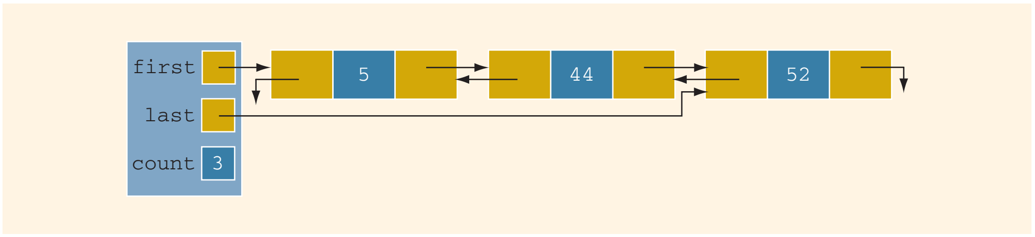


FIGURE 18-44 List after deleting the node with info 17

The definition of the function `deleteNode` is:

```
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
```

Circular Linked Lists

A linked list in which the last node points to the first node is called a **circular linked list**. Figure 18-45 show various circular linked lists.

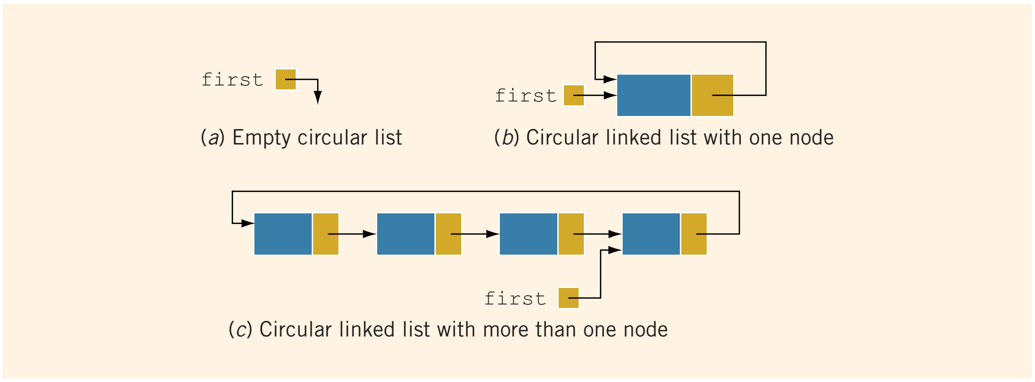

first
(a) Empty circular list
first
(b) Circular linked list with one node
first
(c) Circular linked list with more than one node

FIGURE 18-45 Circular linked lists

In a circular linked list with more than one node, as in Figure 18-45(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 nodes 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:

1. Initialize the list (to an empty state).
2. Determine if the list is empty.
3. Destroy the list.
4. Print the list.
5. Find the length of the list.
6. Search the list for a given item.
7. Insert an item in the list.
8. Delete an item from the list.
9. Copy the list.

We leave it as an 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

For a family or an individual, a favorite place to go on weekends or holidays is to a video store to rent movies. A new video store in your neighborhood is about to open. However, it does not have a program to keep track of its videos and customers. The store managers want someone to write a program for their system so that the video store can function. The program should be able to perform the following operations:

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 of 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 of 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 the following lists:

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

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 of 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:

- 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 a video object are:

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 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 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.

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 given below, 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 of the videos in the store. We also 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 (see Figure 18-46).

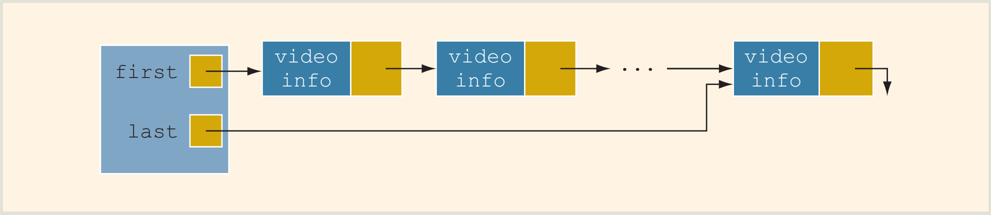


FIGURE 18-46 videoList

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 will, therefore, derive a class videoListType from the class unorderedLinkedList and add these operations.

The definition of the class videoListType is:

```
//***************************************************************
// 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 required 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 determining whether a particular video is in the store, updating the number of copies of a video, and so on, also require the 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.

Consider the node of the video list shown in Figure 18-47.

FIGURE 18-47 Node of a video list

The component info is of type videoType and contains the necessary information about a video. In fact, the component info of the node has seven members: videoTitle, movieStar1, movieStar2, movieProducer, movieDirector, movieProductionCo, and copiesInStock. (See the definition of the **class** videoType.) Therefore, the node of a video list has the form shown in Figure 18-48.

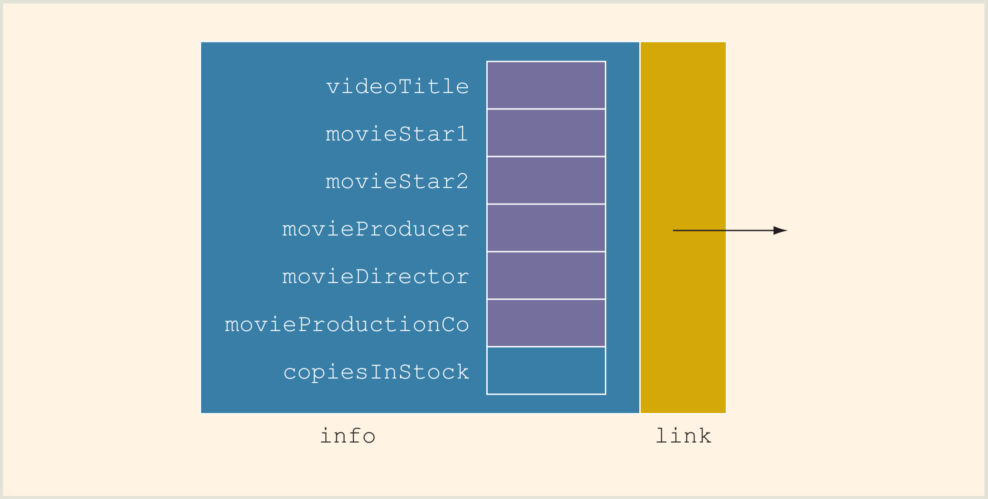


FIGURE 18-48 Video list node showing components of `info`

These member variables are all **`private`** and cannot be accessed directly. The member functions of the **`class`** `videoType` will help us in checking and/or setting the value of a particular component.

Suppose a pointer—say, `current`—points to a node in the video list (see Figure 18-49).

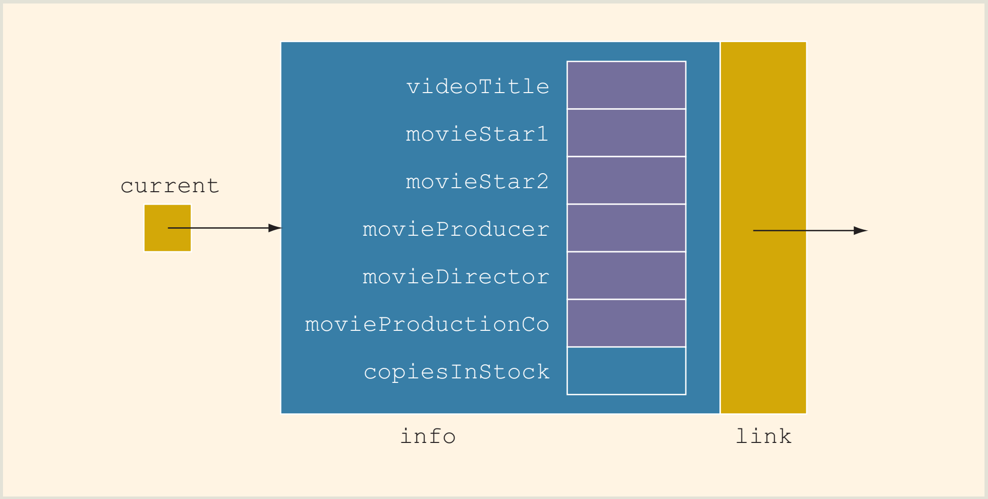


FIGURE 18-49 Pointer `current` and video list node

Now:

```
current->info
```

refers to the info part of the node. Suppose that we want to know whether the title of the video stored in this node is the same as the title specified by the variable title. The expression:

```
current->info.checkTitle(title)
```

is true if the title of the video stored in this node is the same as the title specified by the parameter title, and false otherwise. (Note that the member function checkTitle is a value-returning function. See its declaration in the class videoType.)

As another example, suppose that we want to set copiesInStock of this node to 10. Because copiesInStock is a private member, it cannot be accessed directly. Therefore, the statement:

```
current->info.copiesInStock = 10;  //illegal
```

is incorrect and will generate a compile-time error. We have to use the member function setCopiesInStock as follows:

```
current->info.setCopiesInStock(10);
```

Now that we know how to access a member variable of a video stored in a node, let us describe the algorithm to search the video list.

```
while (not found)
    if the title of the current video is the same as the desired
        title, stop the search
    else
        check the next node
```

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
                     //in the list

    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 12-9 (Chapter 12) 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.

Recall that the basic operations on an object of type personType are:

1. Print the name.
2. Set the name.
3. Show the first name.
4. Show the last name.

Similarly, the basic operations on an object of type customerType are:

1. Print the name, 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:

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:

a. Read the data and store it in a video object.
b. Insert the video in the list.
c. 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

In pseudocode, Step 4 (of the main program) is:

```
a. get choice
b.
   while (choice != 9)
   {
       switch (choice)
       {
       case 1:
           a. get the movie name
           b. search the video list
           c. if found, report success
              else report "failure"
           break;
       case 2:
           a. get the movie name
           b. search the video list
           c. if found, check out the video
              else report "failure"
           break;
       case 3:
           a. get the movie name
           b. search the video list
           c. if found, check in video
              else report "failure"
           break;
       case 4:
           a. get the movie name
           b. search the video list
           c. if found
                 if number of copies > 0
                     report "success"
                 else
                     report "currently out of stock"
              else report "failure"
           break;
       case 5:
           print the titles of the videos
           break;
       case 6:
           print all the videos in the store
           break;
       default: bad selection
       } //end switch
```

```
        displayMenu();
        get choice;
}//end while
```

PROGRAM LISTING

```
/*************************************************************
// Author: D.S. Malik
//
// This program uses the classes videoType and videoListType to
// create a list of videos for a video store. It also performs
// basic operations such as check in and check out 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 displayMenu
```

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 do 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 require the adjustment of two pointers in a node.
13. A linked list in which the last node points to the first node is called a circular linked list.

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.
2. Describe the two typical components of a single linked list node.
3. What is the stored in the link field of the last node of a nonempty single linked list?
4. Suppose that `first` is a pointer to a linked list. What is stored in `first`?
5. Suppose that the fourth node of a linked list is to be deleted, and `p` points to the fourth node? Why do you need a pointer to the third node of the linked list?

Consider the linked list shown in Figure 18-50. Assume that the nodes are in the usual `info-link` form. Use this list to answer Exercises 6 through 12. If necessary, declare additional variables. (Assume that `list`, `p`, `s`, `A`, and `B` are pointers of type `nodeType`.)

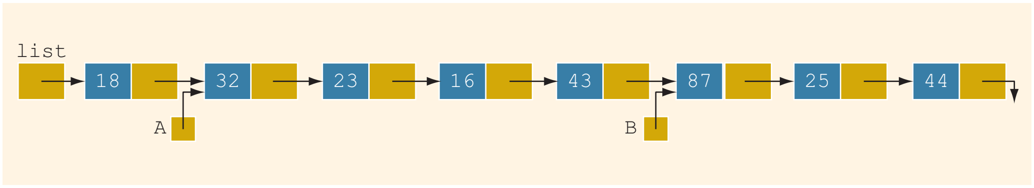


FIGURE 18-50 Linked list for exercises 6 through 12

6. 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;`
7. 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`
8. Mark each of the following statements as valid or invalid. If a statement is invalid, explain why.
   a. `A = B;`
   b. `list->link = A->link;`
   c. `list->link->info = 45;`
   d. `*list = B;`
   e. `*A = *B;`
   f. `B = A->link->info;`
   g. `A->info = B->info;`
   h. `list = B->link->link;`
   i. `B = B->link->link->link;`

9. Write C++ statements to do the following.
   a. Make `A` point to the node containing `info 23`.
   b. Make `list` point to the node containing `16`.
   c. Make `B` point to the last node in the list.
   d. Make `list` point to an empty list.
   e. Set the value of the node containing `25` to `35`.
   f. Create and insert the node with `info 10` after the node pointed to by `A`.
   g. Delete the node with `info 23`. Also, deallocate the memory occupied by this node.
10. What is the output of the following C++ code?

```
p = list;

while (p != NULL)
    cout << p->info << " ";
    p = p->link;
cout << endl;
```

11. If the following C++ code is valid, show the output. If it is invalid, explain why.

```
s = A;
p = B;
s->info = B;
p = p->link;
cout << s->info << " " << p->info << endl;
```

12. If the following C++ code is valid, show the output. If it is invalid, explain why.

```
p = A;
p = p->link;
s = p;
p->link = NULL;
s = s->link;
cout << p->info << " " << s->info << endl;
```

13. Show what is produced by the following C++ code. Assume the node is in the usual `info-link` form with the `info` of type `int`. (`list` and `ptr` are pointers of type `nodeType`.)

```
list = new nodeType;
list->info = 10;
ptr = new nodeType;
ptr->info = 13;
ptr->link = NULL;
list->link = ptr;
ptr = new nodeType;
ptr->info = 18;
ptr->link = list->link;
list->link = ptr;
```

```
cout << list->info << " " << ptr->info << " ";
ptr = ptr->link;
cout << ptr->info << endl;
```

14. Show what is produced by the following C++ code. Assume the node is in the usual `info-link` form with the `info` of type `int`. (list and `ptr` are pointers of type `nodeType`.)

```
list = new nodeType;
list->info = 20;
ptr = new nodeType;
ptr->info = 28;
ptr->link = NULL;
list->link = ptr;
ptr = new nodeType;
ptr->info = 30;
ptr->link = list;
list = ptr;
ptr = new nodeType;
ptr->info = 42;
ptr->link = list->link;
list->link = ptr;
ptr = list;
while (ptr != NULL)
{
   cout << ptr->info << endl;
   ptr = ptr->link;
}
```

15. Assume that the node of a linked list is in the usual `info-link` form with the `info` of type `int`. The following data, as described in parts (a) to (d), is to be inserted into an initially linked list: `72`, `43`, `8`, `12`. Suppose that `head` is a pointer of type `nodeType`. After the linked list is created, `head` should point to the first node of the list. Declare additional variables as you need them. Write the C++ code to create the linked list. After the linked list is created, write a code to print the list. What is the output of your code?

   a. Insert 72 into an empty linked list.

   b. Insert 43 before 72.

   c. Insert 8 at the end of the list.

   d. Insert 12 after 43.

16. Assume that the node of a linked list is in the usual `info-link` form with the `info` of type `int`. (`list` and `ptr` are pointers of type `nodeType`.) The following code creates a linked list.

```
ptr = new nodeType;
ptr->info = 16;
list = new nodeType;
list->info = 25;
list->link = ptr;
ptr = new nodeType;
```

```
ptr->info = 12;
ptr->link = NULL;
list->link->link = ptr;
```

Use the linked list created by this code to answer the following questions. (These questions are independent of each other.) Declare additional pointers if you need them.

a. Which pointer points to the first node of the linked list?

b. Determine the order of the nodes of the linked list.

c. Write a C++ code that creates and inserts a node with `info 45` after the node with `info 16`.

d. Write a C++ code that creates and inserts a node with `info 58` before the node with `info 25`. Does this require you to the change the value of the pointer that was pointing to the first node of the linked list?

e. Write a C++ code that deletes the node with `info 25`. Does this require you to the change the value of the pointer that was pointing to the first node of the linked list?

17. Consider the following C++ statements. (The `class unorderedLinkedList` is as defined in this chapter.)

```
unorderedLinkedList<int> list;

list.insertFirst(15);
list.insertLast(28);
list.insertFirst(30);
list.insertFirst(2);
list.insertLast(45);
list.insertFirst(38);
list.insertLast(25);
list.deleteNode(30);
list.insertFirst(18);
list.deleteNode(28);
list.deleteNode(12);
list.print();
```

What is the output of this program segment?

18. Suppose the input is:

```
18 30 4 32 45 36 78 19 48 75 -999
```

What is the output of the following C++ code? (The `class unorderedLinkedList` is as defined in this chapter.)

```
unorderedLinkedList<int> list;
unorderedLinkedList<int> copyList;
int num;

cin >> num;
while (num != -999)
```

```
{
    if (num % 5 == 0 || num % 5 == 3)
        list.insertFirst(num);
    else
        list.insertLast(num);
    cin >> num;
}

list.print();
cout << endl;

copyList = list;

copyList.deleteNode(78);
copyList.deleteNode(35);

cout << "Copy List = ";
copyList.print();
cout << endl;
```

19. Draw the UML diagram of the **class** `doublyLinkedList` as discussed in this chapter.
20. Draw the UML diagram of the **class** `videoType` of the Video Store programming example.
21. Draw the UML diagram of the **class** `videoListType` of the Video Store programming example.

PROGRAMMING EXERCISES

1. (**Online Address Book revisited**) Programming Exercise 6 in Chapter 13 could handle a maximum of only 500 entries. Using linked lists, redo the program to handle as many entries as required. Add the following operations to your program:
   a. Add or delete a new entry to the address book.
   b. Allow the user to save the data in the address book.
2. Extend the **class** `linkedListType` by adding the following operations:
   a. Find and delete the node with the smallest `info` in the list. (Delete only the first occurrence and traverse the list only once.)
   b. Find and delete all occurrences of a given `info` from the list. (Traverse the list only once.)

   Add these as abstract functions in the **class** `linkedListType` and provide the definitions of these functions in the **class** `unorderedLinkedList`. Also, write a program to test these functions.

3. Extend the `class linkedListType` by adding the following operations:

   a. Write a function that returns the info of the k^th^ element of the linked list. If no such element exists, terminate the program.

   b. Write a function that deletes the k^th^ element of the linked list. If no such element exists, terminate the program.

   Provide the definitions of these functions in the `class linkedListType`. Also, write a program to test these functions. (Use either the `class unorderedLinkedList` or the `class orderedLinkedList` to test your function.)

4. (**Printing a single linked list backward**) Include the functions `reversePrint` and `recursiveReversePrint`, as discussed in this chapter, in the `class linkedListType`. Also, write a program function to print a (single) linked list backward. (Use either the `class unorderedLinkedList` or the `class orderedLinkedList` to test your function.)

5. (**Dividing a linked list into two sublists of almost equal sizes)**

   a. Add the operation `divideMid` to the `class linkedListType` as follows:

```
void divideMid(linkedListType<Type> &sublist);
  //This operation divides the given list into two sublists
  //of (almost) equal sizes.
  //Postcondition: first points to the first node and last
  //               points to the last node of the first
  //               sublist.
  //               sublist.first points to the first node
  //               and sublist.last points to the last node
  //               of the second sublist.
```

   Consider the following statements:

```
unorderedLinkedList<int> myList;
unorderedLinkedList<int> subList;
```

   Suppose `myList` points to the list with elements `34 65 27 89 12` (in this order). The statement:

```
myList.divideMid(subList);
```

   divides `myList` into two sublists: `myList` points to the list with the elements `34 65 27`, and `subList` points to the sublist with the elements `89 12`.

   b. Write the definition of the function template to implement the operation `divideMid`. Also, write a program to test your function.

6. (**Splitting a linked list, at a given node, into two sublists**)

a. Add the following operation to the class `linkedListType`:

```
void divideAt(linkedListType<Type> &secondList,
              const Type& item);
  //Divide the list at the node with the info item into two
  //sublists.
   //Postcondition: first and last point to the first and
  //               last nodes of the first sublist.
  //               secondList.first and secondList.last
  //               point to the first and last nodes of the
  //               second sublist.
```

Consider the following statements:

```
unorderedLinkedList<int> myList;
unorderedLinkedList<int> otherList;
```

Suppose `myList` points to the list with the elements `34 65 18 39 27 89 12` (in this order). The statement:

```
myList.divideAt(otherList, 18);
```

divides `myList` into two sublists: `myList` points to the list with the elements `34 65`, and `otherList` points to the sublist with the elements `18 39 27 89 12`.

b. Write the definition of the function template to implement the operation `divideAt`. Also, write a program to test your function.

7. a. Add the following operation to the class `orderedLinkedList`:

```
void mergeLists(orderedLinkedList<Type> &list1,
                orderedLinkedList<Type> &list2);
   //This function creates a new list by merging the
   //elements of list1 and list2.
   //Postcondition: first points to the merged list
   //               list1 and list2 are empty
```

Consider the following statements:

```
orderedLinkedList<int> newList;
orderedLinkedList<int> list1;
orderedLinkedList<int> list2;
```

Suppose `list1` points to the list with the elements `2 6 7`, and `list2` points to the list with the elements `3 5 8`. The statement:

```
newList.mergeLists(list1, list2);
```

creates a new linked list with the elements in the order `2 3 5 6 7 8`, and the object `newList` points to this list. Also, after the preceding statement executes, `list1` and `list2` are empty.

b. Write the definition of the function template `mergeLists` to implement the operation `mergeLists`.

8. The function `insert` of the `class orderedLinkedList` does not check if the item to be inserted is already in the list; that is, it does not check for duplicates. Rewrite the definition of the function `insert` so that before inserting the item, it checks whether the item to be inserted is already in the list. If the item to be inserted is already in the list, the function outputs an appropriate error message. Also, write a program to test your function.

9. In this chapter, the class to implement the nodes of a linked list is defined as a `struct`. The following rewrites the definition of the `struct nodeType` so that it is declared as a class and the member variables are `private`.

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

    void setInfo(const Type& elem);
      //Function to set the info of the node.
      //Postcondition: info = elem;

    Type getInfo() const;
      //Function to return the info of the node.
      //Postcondition: The value of info is returned.

    void setLink(nodeType<Type> *ptr);
      //Function to set the link of the node.
      //Postcondition: link = ptr;

    nodeType<Type>* getLink() const;
      //Function to return the link of the node.
      //Postcondition: The value of link is returned.

    nodeType();
       //Default constructor
       //Postcondition: link = NULL;

    nodeType(const Type& elem, nodeType<Type> *ptr);
       //Constructor with parameters
       //Sets info to point to the object elem points to, and
       //link is set to point to the object ptr points to.
       //Postcondition: info = elem; link = ptr
    nodeType(const nodeType<Type> &otherNode);
      //Copy constructor
    ~nodeType();
      //Destructor

private:
    Type info;
    nodeType<Type> *link;
};
```

Write the definitions of the member functions of the `class nodeType`. Also, write a program to test your class.

10. Programming Exercise 9 asks you to redefine the class to implement the nodes of a linked list so that the instance variables are `private`. Therefore, the `class linkedListType` and its derived `class`es `unorderedLinkedList` and `orderedLinkedList` can no longer directly access the instance variables of the `class nodeType`. Rewrite the definitions of these classes so that they use the member functions of the `class nodeType` to access the info and link fields of a node. Also, write programs to test various operations of the classes `unorderedLinkedList` and `orderedLinkeList`.
11. Write the definitions of the function `copyList`, the copy constructor, and the function to overload the assignment operator for the `class doublyLinkedList`.
12. Write a program to test various operations of the `class doublyLinkedList`.
13. (**Circular linked lists**) This chapter defined and identified various operations on a circular linked list.
    a. Write the definitions of the `class circularLinkedList` and its member functions. (You may assume that the elements of the circular linked list are in ascending order.)
    b. Write a program to test various operations of the class defined in (a).
14. (**Video Store programming example**)
    a. Complete the design and implementation of the `class customerType` defined in the Video Store programming example.
    b. Design and implement the `class customerListType` to create and maintain a list of customers for the video store.
15. (**Video Store programming example**) Complete the design and implementation of the video store program. In other words, write a program that uses the classes designed in the Video Store programming example and in Programming Exercise 14 to make a video store operational.

16. Extend the `class linkedListType` by adding the following function:

```
void rotate();
//Function to remove the first node of a linked list and put it
//at the end of the linked list.
```

    Also write a program to test your function. Use the `class unorderedLinkedList` to create a linked list.

17. Write a program that prompts the user to input a string and then outputs the string in the pig Latin form. The rules for converting a string into pig Latin form are described in Programming Example: Pig Latin Strings of Chapter 8. Your program must store the characters of a string into a linked and use the function `rotate`, as described in Programming Exercise 16, to rotate the string.