

3.2 Singly Linked Lists

In the previous section, we presented the array data structure and discussed some of its applications. Arrays are nice and simple for storing things in a certain order, but they have drawbacks. They are not very adaptable. For instance, we have to fix the size n of an array in advance, which makes resizing an array difficult. (This drawback is remedied in STL vectors.) Insertions and deletions are difficult because elements need to be shifted around to make space for insertion or to fill empty positions after deletion. In this section, we explore an important alternate implementation of sequence, known as the singly linked list.

A **linked list**, in its simplest form, is a collection of **nodes** that together form a linear ordering. As in the children's game "Follow the Leader," each node stores a pointer, called *next*, to the next node of the list. In addition, each node stores its associated element. (See Figure 3.9.)



Figure 3.9: Example of a singly linked list of airport codes. The *next* pointers are shown as arrows. The null pointer is denoted by \emptyset .

The *next* pointer inside a node is a **link** or **pointer** to the next node of the list. Moving from one node to another by following a *next* reference is known as **link hopping** or **pointer hopping**. The first and last nodes of a linked list are called the **head** and **tail** of the list, respectively. Thus, we can link-hop through the list, starting at the head and ending at the tail. We can identify the tail as the node having a null *next* reference. The structure is called a **singly linked list** because each node stores a single link.

Like an array, a singly linked list maintains its elements in a certain order, as determined by the chain of *next* links. Unlike an array, a singly linked list does not have a predetermined fixed size. It can be resized by adding or removing nodes.

3.2.1 Implementing a Singly Linked List

Let us implement a singly linked list of strings. We first define a class `StringNode` shown in Code Fragment 3.13. The node stores two values, the member *elem* stores the element stored in this node, which in this case is a character string. (Later, in Section 3.2.4, we describe how to define nodes that can store arbitrary types of elements.) The member *next* stores a **pointer** to the next node of the list. We make the **linked list** class a friend, so that it can access the node's private members.

```

class StringNode {                                // a node in a list of strings
private:
    string elem;                                  // element value
    StringNode* next;                             // next item in the list

    friend class StringLinkedList;                // provide StringLinkedList access
};

```

Code Fragment 3.13: A node in a singly linked list of strings.

In Code Fragment 3.14, we define a class `StringLinkedList` for the actual linked list. It supports a number of member functions, including a constructor and destructor and functions for insertion and deletion. Their implementations are presented later. Its private data consists of a pointer to the head node of the list.

```

class StringLinkedList {                          // a linked list of strings
public:
    StringLinkedList();                           // empty list constructor
    ~StringLinkedList();                          // destructor
    bool empty() const;                           // is list empty?
    const string& front() const;                  // get front element
    void addFront(const string& e);               // add to front of list
    void removeFront();                           // remove front item list
private:
    StringNode* head;                             // pointer to the head of list
};

```

Code Fragment 3.14: A class definition for a singly linked list of strings.

A number of simple member functions are shown in Code Fragment 3.15. The list constructor creates an empty list by setting the head pointer to `NULL`. The destructor repeatedly removes elements from the list. It exploits the fact that the function `remove` (presented below) destroys the node that it removes. To test whether the list is empty, we simply test whether the head pointer is `NULL`.

```

StringLinkedList::StringLinkedList()              // constructor
: head(NULL) { }

StringLinkedList::~StringLinkedList()             // destructor
{ while (!empty()) removeFront(); }

bool StringLinkedList::empty() const              // is list empty?
{ return head == NULL; }

const string& StringLinkedList::front() const     // get front element
{ return head->elem; }

```

Code Fragment 3.15: Some simple member functions of class `StringLinkedList`.

3.2.2 Insertion to the Front of a Singly Linked List

We can easily insert an element at the head of a singly linked list. We first create a new node, and set its *elem* value to the desired string and set its *next* link to point to the current head of the list. We then set *head* to point to the new node. The process is illustrated in Figure 3.10.

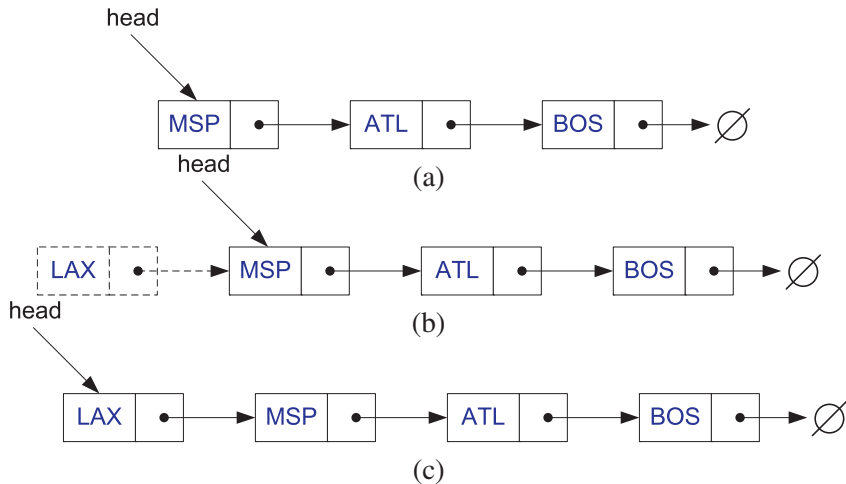


Figure 3.10: Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) creation of a new node; (c) after the insertion.

An implementation is shown in Code Fragment 3.16. Note that access to the private members *elem* and *next* of the `StringNode` class would normally be prohibited, but it is allowed here because `StringLinkedList` was declared to be a friend of `StringNode`.

```
void StringLinkedList::addFront(const string& e) { // add to front of list
    StringNode* v = new StringNode;              // create new node
    v->elem = e;                                   // store data
    v->next = head;                               // head now follows v
    head = v;                                     // v is now the head
}
```

Code Fragment 3.16: Insertion to the front of a singly linked list.

3.2.3 Removal from the Front of a Singly Linked List

Next, we consider how to remove an element from the front of a singly linked list. We essentially undo the operations performed for insertion. We first save a pointer

to the old head node and advance the head pointer to the next node in the list. We then delete the old head node. This operation is illustrated in Figure 3.11.

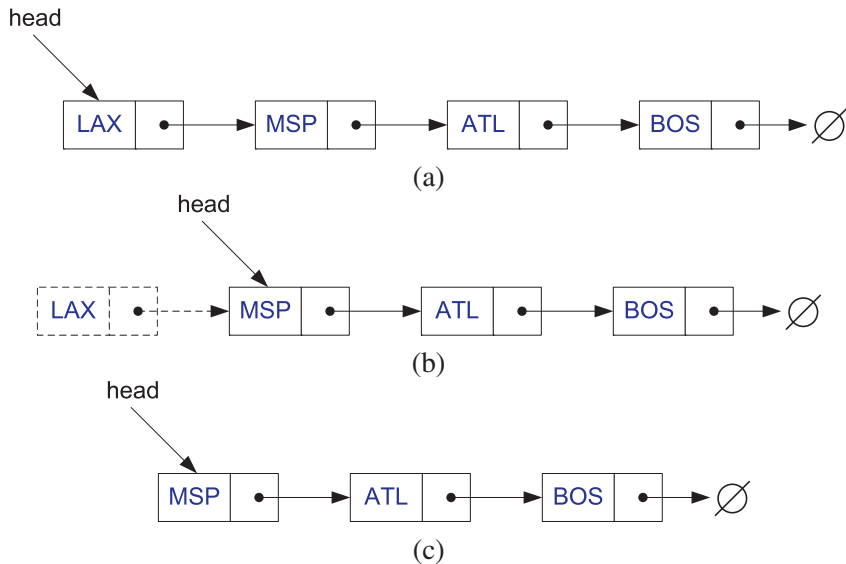


Figure 3.11: Removal of an element at the head of a singly linked list: (a) before the removal; (b) “linking out” the old new node; (c) after the removal.

An implementation of this operation is provided in Code Fragment 3.17. We assume that the user has checked that the list is nonempty before applying this operation. (A more careful implementation would throw an exception if the list were empty.) The function deletes the node in order to avoid any memory leaks. We do not return the value of the deleted node. If its value is desired, we can call the front function prior to the removal.

```

void StringLinkedList::removeFront() {           // remove front item
    StringNode* old = head;                       // save current head
    head = old->next;                             // skip over old head
    delete old;                                   // delete the old head
}

```

Code Fragment 3.17: Removal from the front of a singly linked list.

It is noteworthy that we cannot as easily delete the last node of a singly linked list, even if we had a pointer to it. In order to delete a node, we need to update the *next* link of the node immediately *preceding* the deleted node. Locating this node involves traversing the entire list and could take a long time. (We remedy this in Section 3.3 when we discuss doubly linked lists.)

3.2.4 Implementing a Generic Singly Linked List

The implementation of the singly linked list given in Section 3.2.1 assumes that the element type is a character string. It is easy to convert the implementation so that it works for an arbitrary element type through the use of C++’s template mechanism. The resulting generic singly linked list class is called `SLinkedList`.

We begin by presenting the node class, called `SNode`, in Code Fragment 3.18. The element type associated with each node is parameterized by the type variable `E`. In contrast to our earlier version in Code Fragment 3.13, references to the data type “string” have been replaced by “`E`.” When referring to our templated node and list class, we need to include the suffix “`<E>`.” For example, the class is `SLinkedList<E>` and the associated node is `SNode<E>`.

```
template <typename E>
class SNode {                                // singly linked list node
private:
    E elem;                                  // linked list element value
    SNode<E>* next;                          // next item in the list
    friend class SLinkedList<E>;            // provide SLinkedList access
};
```

Code Fragment 3.18: A node in a generic singly linked list.

The generic list class is presented in Code Fragment 3.19. As above, references to the specific element type “string” have been replaced by references to the generic type parameter “`E`.” To keep things simple, we have omitted housekeeping functions such as a copy constructor.

```
template <typename E>
class SLinkedList {                          // a singly linked list
public:
    SLinkedList();                          // empty list constructor
    ~SLinkedList();                        // destructor
    bool empty() const;                   // is list empty?
    const E& front() const;               // return front element
    void addFront(const E& e);            // add to front of list
    void removeFront();                   // remove front item list
private:
    SNode<E>* head;                      // head of the list
};
```

Code Fragment 3.19: A class definition for a generic singly linked list.

In Code Fragment 3.20, we present the class member functions. Note the similarity with Code Fragments 3.15 through 3.17. Observe that each definition is prefaced by the template specifier `template <typename E>`.

```

template <typename E>
SLinkedList<E>::SLinkedList()                                // constructor
    : head(NULL) { }

template <typename E>
bool SLinkedList<E>::empty() const                          // is list empty?
    { return head == NULL; }

template <typename E>
const E& SLinkedList<E>::front() const                      // return front element
    { return head->elem; }

template <typename E>
SLinkedList<E>::~~SLinkedList()                             // destructor
    { while (!empty()) removeFront(); }

template <typename E>
void SLinkedList<E>::addFront(const E& e) {                 // add to front of list
    SNode<E>* v = new SNode<E>;                             // create new node
    v->elem = e;                                              // store data
    v->next = head;                                          // head now follows v
    head = v;                                              // v is now the head
}

template <typename E>
void SLinkedList<E>::removeFront() {                       // remove front item
    SNode<E>* old = head;                                   // save current head
    head = old->next;                                       // skip over old head
    delete old;                                           // delete the old head
}

```

Code Fragment 3.20: Other member functions for a generic singly linked list.

We can generate singly linked lists of various types by simply setting the template parameter as desired as shown in the following code fragment.

```

SLinkedList<string> a;                                       // list of strings
a.addFront("MSP");
// ...
SLinkedList<int> b;                                         // list of integers
b.addFront(13);

```

Code Fragment 3.21: Examples using the generic singly linked list class.

Because templated classes carry a relatively high notational burden, we often sacrifice generality for simplicity, and avoid the use of templated classes in some of our examples.

3.3 Doubly Linked Lists

As we saw in the previous section, removing an element at the tail of a singly linked list is not easy. Indeed, it is time consuming to remove any node other than the head in a singly linked list, since we do not have a quick way of accessing the node immediately preceding the one we want to remove. There are many applications where we do not have quick access to such a predecessor node. For such applications, it would be nice to have a way of going both directions in a linked list.

There is a type of linked list that allows us to go in both directions—forward and reverse—in a linked list. It is the **doubly linked** list. In addition to its element member, a node in a doubly linked list stores two pointers, a *next* link and a *prev* link, which point to the next node in the list and the previous node in the list, respectively. Such lists allow for a great variety of quick update operations, including efficient insertion and removal at any given position.

Header and Trailer Sentinels

To simplify programming, it is convenient to add special nodes at both ends of a doubly linked list: a **header** node just before the head of the list, and a **trailer** node just after the tail of the list. These “dummy” or **sentinel** nodes do not store any elements. They provide quick access to the first and last nodes of the list. In particular, the header’s *next* pointer points to the first node of the list, and the *prev* pointer of the trailer node points to the last node of the list. An example is shown in Figure 3.12.

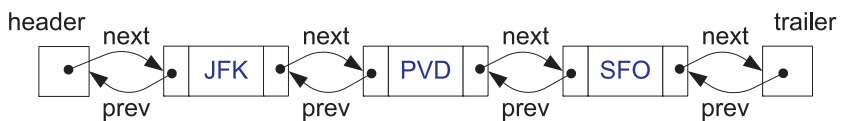


Figure 3.12: A doubly linked list with sentinels, *header* and *trailer*, marking the ends of the list. An empty list would have these sentinels pointing to each other. We do not show the null *prev* pointer for the *header* nor do we show the null *next* pointer for the *trailer*.

3.3.1 Insertion into a Doubly Linked List

Because of its double link structure, it is possible to insert a node at any position within a doubly linked list. Given a node v of a doubly linked list (which could possibly be the header, but not the trailer), let z be a new node that we wish to insert

immediately after v . Let w be the node following v , that is, w is the node pointed to by v 's next link. (This node exists, since we have sentinels.) To insert z after v , we link it into the current list, by performing the following operations:

- Make z 's *prev* link point to v
- Make z 's *next* link point to w
- Make w 's *prev* link point to z
- Make v 's *next* link point to z

This process is illustrated in Figure 3.13, where v points to the node JFK, w points to PVD, and z points to the new node BWI. Observe that this process works if v is any node ranging from the header to the node just prior to the trailer.

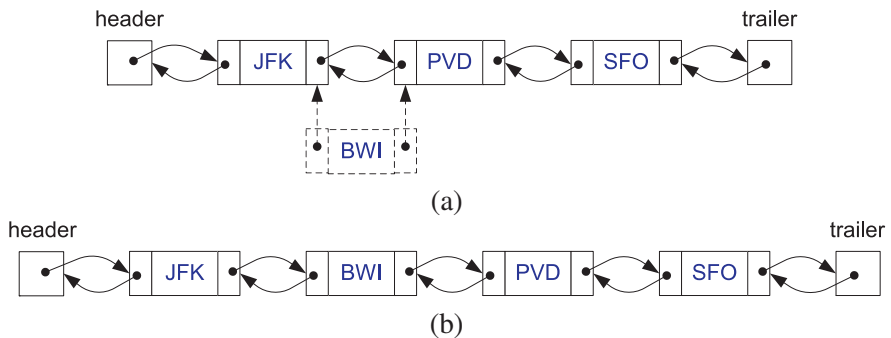


Figure 3.13: Adding a new node after the node storing JFK: (a) creating a new node with element BWI and linking it in; (b) after the insertion.

3.3.2 Removal from a Doubly Linked List

Likewise, it is easy to remove a node v from a doubly linked list. Let u be the node just prior to v , and w be the node just following v . (These nodes exist, since we have sentinels.) To remove node v , we simply have u and w point to each other instead of to v . We refer to this operation as the **linking out** of v . We perform the following operations.

- Make w 's *prev* link point to u
- Make u 's *next* link point to w
- Delete node v

This process is illustrated in Figure 3.14, where v is the node PVD, u is the node JFK, and w is the node SFO. Observe that this process works if v is any node from the header to the tail node (the node just prior to the trailer).

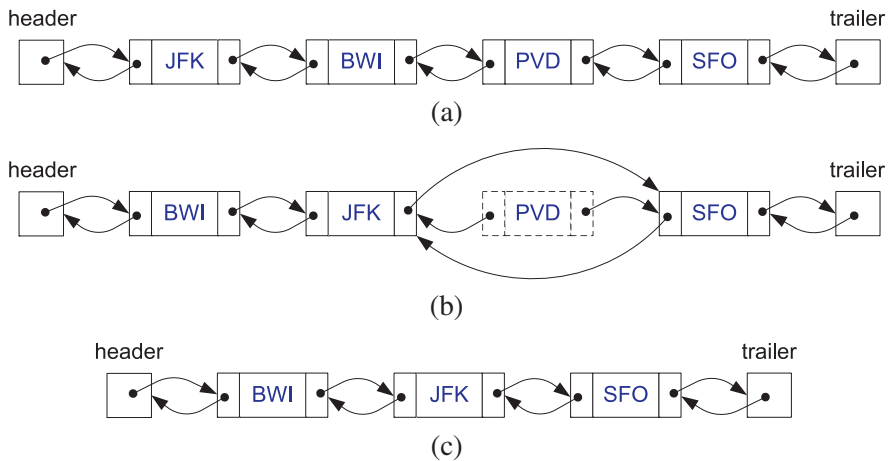


Figure 3.14: Removing the node storing PVD: (a) before the removal; (b) linking out the old node; (c) after node deletion.

3.3.3 A C++ Implementation

Let us consider how to implement a doubly linked list in C++. First, we present a C++ class for a node of the list in Code Fragment 3.22. To keep the code simple, we have chosen not to derive a templated class as we did in Section 3.2.1 for singly linked lists. Instead, we provide a **typedef** statement that defines the element type, called `Elem`. We define it to be a string, but any other type could be used instead. Each node stores an element. It also contains pointers to both the previous and next nodes of the list. We declare `DLinkedList` to be a friend, so it can access the node's private members.

```
typedef string Elem;           // list element type
class DNode {                 // doubly linked list node
private:
    Elem elem;                // node element value
    DNode* prev;              // previous node in list
    DNode* next;              // next node in list
    friend class DLinkedList;  // allow DLinkedList access
};
```

Code Fragment 3.22: C++ implementation of a doubly linked list node.

Next, we present the definition of the doubly linked list class, `DLinkedList`, in Code Fragment 3.23. In addition to a constructor and destructor, the public members consist of a function that indicates whether the list is currently empty

(meaning that it has no nodes other than the sentinels) and accessors to retrieve the front and back elements. We also provide methods for inserting and removing elements from the front and back of the list. There are two private data members, *header* and *trailer*, which point to the sentinels. Finally, we provide two protected utility member functions, *add* and *remove*. They are used internally by the class and by its subclasses, but they cannot be invoked from outside the class.

```

class DLinkedList {                                // doubly linked list
public:
    DLinkedList();                                // constructor
    ~DLinkedList();                               // destructor
    bool empty() const;                           // is list empty?
    const Elem& front() const;                     // get front element
    const Elem& back() const;                     // get back element
    void addFront(const Elem& e);                  // add to front of list
    void addBack(const Elem& e);                  // add to back of list
    void removeFront();                           // remove from front
    void removeBack();                           // remove from back
private:                                         // local type definitions
    DNode* header;                               // list sentinels
    DNode* trailer;
protected:                                     // local utilities
    void add(DNode* v, const Elem& e);            // insert new node before v
    void remove(DNode* v);                       // remove node v
};

```

Code Fragment 3.23: Implementation of a doubly linked list class.

Let us begin by presenting the class constructor and destructor as shown in Code Fragment 3.24. The constructor creates the sentinel nodes and sets each to point to the other, and the destructor removes all but the sentinel nodes.

```

DLinkedList::DLinkedList() {                      // constructor
    header = new DNode;                           // create sentinels
    trailer = new DNode;
    header->next = trailer;                         // have them point to each other
    trailer->prev = header;
}

DLinkedList::~DLinkedList() {                     // destructor
    while (!empty()) removeFront();               // remove all but sentinels
    delete header;                                // remove the sentinels
    delete trailer;
}

```

Code Fragment 3.24: Class constructor and destructor.

Next, in Code Fragment 3.25 we show the basic class accessors. To determine whether the list is empty, we check that there is no node between the two sentinels. We do this by testing whether the trailer follows immediately after the header. To access the front element of the list, we return the element associated with the node that follows the list header. To access the back element, we return the element associated with node that precedes the trailer. Both operations assume that the list is nonempty. We could have enhanced these functions by throwing an exception if an attempt is made to access the front or back of an empty list, just as we did in Code Fragment 3.6.

```

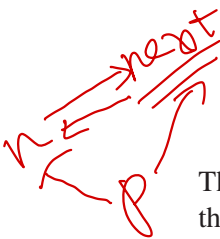
bool DLinkedList::empty() const           // is list empty?
{ return (header->next == trailer); }

const Elem& DLinkedList::front() const     // get front element
{ return header->next->elem; }

const Elem& DLinkedList::back() const      // get back element
{ return trailer->prev->elem; }

```

Code Fragment 3.25: Accessor functions for the doubly linked list class.



In Section 3.3.1, we discussed how to insert a node into a doubly linked list. The local utility function `add`, which is shown in Code Fragment 3.26, implements this operation. In order to add a node to the front of the list, we create a new node, and insert it immediately after the header, or equivalently, immediately before the node that follows the header. In order to add a new node to the back of the list, we create a new node, and insert it immediately before the trailer.

```

// insert new node before v
void DLinkedList::add(DNode* v, const Elem& e) {
    DNode* u = new DNode; u->elem = e; // create a new node for e
    u->next = v;                        // link u in between v
    u->prev = v->prev;                  // ...and v->prev
    v->prev->next = v->prev = u;
}

void DLinkedList::addFront(const Elem& e) // add to front of list
{ add(header->next, e); }

void DLinkedList::addBack(const Elem& e) // add to back of list
{ add(trailer, e); }

```

Code Fragment 3.26: Inserting a new node into a doubly linked list. The protected utility function `add` inserts a node z before an arbitrary node v . The public member functions `addFront` and `addBack` both invoke this utility function.

Observe that the above code works even if the list is empty (meaning that the only nodes are the header and trailer). For example, if `addBack` is invoked on an empty list, then the value of `trailer->prev` is a pointer to the list header. Thus, the node is added between the header and trailer as desired. One of the major advantages of providing sentinel nodes is to avoid handling of special cases, which would otherwise be needed.

Finally, let us discuss deletion. In Section 3.3.2, we showed how to remove an arbitrary node from a doubly linked list. In Code Fragment 3.27, we present local utility function `remove`, which performs the operation. In addition to linking out the node, it also deletes the node. The public member functions `removeFront` and `removeBack` are implemented by deleting the nodes immediately following the header and immediately preceding the trailer, respectively.

```

void DLinkedList::remove(DNode* v) {           // remove node v
    DNode* u = v->prev;                         // predecessor
    DNode* w = v->next;                         // successor
    u->next = w;                                // unlink v from list
    w->prev = u;
    delete v;
}

void DLinkedList::removeFront()                // remove from front
{ remove(header->next); }

void DLinkedList::removeBack()                 // remove from back
{ remove(trailer->prev); }
```

Code Fragment 3.27: Removing a node from a doubly linked list. The local utility function `remove` removes the node `v`. The public member functions `removeFront` and `removeBack` invoke this utility function.

There are many more features that we could have added to our simple implementation of a doubly linked list. Although we have provided access to the ends of the list, we have not provided any mechanism for accessing or modifying elements in the middle of the list. Later, in Chapter 6, we discuss the concept of iterators, which provides a mechanism for accessing arbitrary elements of a list.

We have also performed no error checking in our implementation. It is the user's responsibility not to attempt to access or remove elements from an empty list. In a more robust implementation of a doubly linked list, we would design the member functions `front`, `back`, `removeFront`, and `removeBack` to throw an exception when an attempt is made to perform one of these functions on an empty list. Nonetheless, this simple implementation illustrates how easy it is to manipulate this useful data structure.

3.4 Circularly Linked Lists and List Reversal

In this section, we study some applications and extensions of linked lists.

3.4.1 Circularly Linked Lists

A **circularly linked list** has the same kind of nodes as a singly linked list. That is, each node in a circularly linked list has a next pointer and an element value. But, rather than having a head or tail, the nodes of a circularly linked list are linked into a cycle. If we traverse the nodes of a circularly linked list from any node by following **next** pointers, we eventually visit all the nodes and cycle back to the node from which we started.

Even though a circularly linked list has no beginning or end, we nevertheless need some node to be marked as a special node, which we call the **cursor**. The cursor node allows us to have a place to start from if we ever need to traverse a circularly linked list.

There are two positions of particular interest in a circular list. The first is the element that is referenced by the cursor, which is called the **back**, and the element immediately following this in the circular order, which is called the **front**. Although it may seem odd to think of a circular list as having a front and a back, observe that, if we were to cut the link between the node referenced by the cursor and this node's immediate successor, the result would be a singly linked list from the front node to the back node.

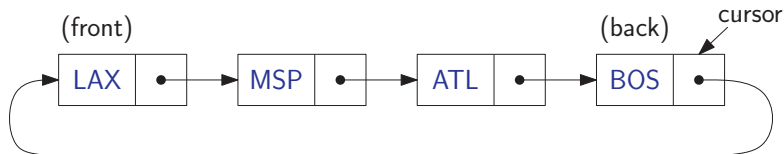


Figure 3.15: A circularly linked list. The node referenced by the cursor is called the back, and the node immediately following is called the front.

We define the following functions for a circularly linked list:

front(): Return the element referenced by the cursor; an error results if the list is empty.

back(): Return the element immediately after the cursor; an error results if the list is empty.

advance(): Advance the cursor to the next node in the list.

- add(*e*):** Insert a new node with element *e* immediately after the cursor; if the list is empty, then this node becomes the cursor and its *next* pointer points to itself.
- remove():** Remove the node immediately after the cursor (not the cursor itself, unless it is the only node); if the list becomes empty, the cursor is set to *null*.

In Code Fragment 3.28, we show a C++ implementation of a node of a circularly linked list, assuming that each node contains a single string. The node structure is essentially identical to that of a singly linked list (recall Code Fragment 3.13). To keep the code simple, we have not implemented a templated class. Instead, we provide a **typedef** statement that defines the element type *Elem* to be the base type of the list, which in this case is a string.

```
typedef string Elem;           // element type
class CNode {                 // circularly linked list node
private:
    Elem elem;                // linked list element value
    CNode* next;              // next item in the list

    friend class CircleList;   // provide CircleList access
};
```

Code Fragment 3.28: A node of a circularly linked list.

Next, in Code Fragment 3.29, we present the class definition for a circularly linked list called *CircleList*. In addition to the above functions, the class provides a constructor, a destructor, and a function to detect whether the list is empty. The private member consists of the cursor, which points to some node of the list.

```
class CircleList {             // a circularly linked list
public:
    CircleList();               // constructor
    ~CircleList();              // destructor
    bool empty() const;         // is list empty?
    const Elem& front() const;   // element at cursor
    const Elem& back() const;    // element following cursor
    void advance();              // advance cursor
    void add(const Elem& e);      // add after cursor
    void remove();               // remove node after cursor
private:
    CNode* cursor;              // the cursor
};
```

Code Fragment 3.29: Implementation of a circularly linked list class.

Code Fragment 3.30 presents the class's constructor and destructor. The constructor generates an empty list by setting the cursor to NULL. The destructor iteratively removes nodes until the list is empty. We exploit the fact that the member function `remove` (given below) deletes the node that it removes.

```
CircleList::CircleList()           // constructor
: cursor(NULL) { }
CircleList::~CircleList()         // destructor
{ while (!empty()) remove(); }
```

Code Fragment 3.30: The constructor and destructor.

We present a number of simple member functions in Code Fragment 3.31. To determine whether the list is empty, we test whether the cursor is NULL. The advance function advances the cursor to the next element.

```
bool CircleList::empty() const    // is list empty?
{ return cursor == NULL; }
const Elem& CircleList::back() const // element at cursor
{ return cursor->elem; }
const Elem& CircleList::front() const // element following cursor
{ return cursor->next->elem; }
void CircleList::advance()        // advance cursor
{ cursor = cursor->next; }
```

Code Fragment 3.31: Simple member functions.

Next, let us consider insertion. Recall that insertions to the circularly linked list occur *after* the cursor. We begin by creating a new node and initializing its data member. If the list is empty, we create a new node that points to itself. We then direct the cursor to point to this element. Otherwise, we link the new node just after the cursor. The code is presented in Code Fragment 3.32.

```
void CircleList::add(const Elem& e) { // add after cursor
    CNode* v = new CNode;           // create a new node
    v->elem = e;
    if (cursor == NULL) {           // list is empty?
        v->next = v;                 // v points to itself
        cursor = v;                 // cursor points to v
    }
    else {                           // list is nonempty?
        v->next = cursor->next;      // link in v after cursor
        cursor->next = v;
    }
}
```

Code Fragment 3.32: Inserting a node just after the cursor of a circularly linked list.

Finally, we consider removal. We assume that the user has checked that the list is nonempty before invoking this function. (A more careful implementation would throw an exception if the list is empty.) There are two cases. If this is the last node of the list (which can be tested by checking that the node to be removed points to itself) we set the cursor to NULL. Otherwise, we link the cursor's next pointer to skip over the removed node. We then delete the node. The code is presented in Code Fragment 3.33.

```

void CircleList::remove() {           // remove node after cursor
    CNode* old = cursor->next;         // the node being removed
    if (old == cursor)                 // removing the only node?
        cursor = NULL;                // list is now empty
    else
        cursor->next = old->next;       // link out the old node
    delete old;                        // delete the old node
}

```

Code Fragment 3.33: Removing the node following the cursor.

To keep the code simple, we have omitted error checking. In front, back, and advance, we should first test whether the list is empty, since otherwise the cursor pointer will be NULL. In the first two cases, we should throw some sort of exception. In the case of advance, if the list is empty, we can simply return.

Maintaining a Playlist for a Digital Audio Player

To help illustrate the use of our CircleList implementation of the circularly linked list, let us consider how to build a simple interface for maintaining a playlist for a digital audio player, also known as an MP3 player. The songs of the player are stored in a circular list. The cursor points to the current song. By advancing the cursor, we can move from one song to the next. We can also add new songs and remove songs by invoking the member functions insert and remove, respectively. Of course, a complete implementation would need to provide a method for playing the current song, but our purpose is to illustrate how the circularly linked list can be applied to this task.

To make this more concrete, suppose that you have a friend who loves retro music, and you want to create a playlist of songs from the bygone Disco Era. The main program is presented Code Fragment 3.34. We declare an object *playList* to be a CircleList. The constructor creates an empty playlist. We proceed to add three songs, “Stayin Alive,” “Le Freak,” and “Jive Talkin.” The comments on the right show the current contents of the list in square brackets. The first entry of the list is the element immediately following the cursor (which is where insertion and removal occur), and the last entry in the list is cursor (which is indicated with an asterisk).

Suppose that we decide to replace “Stayin Alive” with “Disco Inferno.” We advance the cursor twice so that “Stayin Alive” comes immediately after the cursor. We then remove this entry and insert its replacement.

```
int main() {
    CircleList playList;           // []
    playList.add("Stayin Alive");  // [Stayin Alive*]
    playList.add("Le Freak");      // [Le Freak, Stayin Alive*]
    playList.add("Jive Talkin");   // [Jive Talkin, Le Freak, Stayin Alive*]

    playList.advance();            // [Le Freak, Stayin Alive, Jive Talkin*]
    playList.advance();            // [Stayin Alive, Jive Talkin, Le Freak*]
    playList.remove();             // [Jive Talkin, Le Freak*]
    playList.add("Disco Inferno"); // [Disco Inferno, Jive Talkin, Le Freak*]
    return EXIT_SUCCESS;
}
```

Code Fragment 3.34: Using the CircleList class to implement a playlist for a digital audio player.

3.4.2 Reversing a Linked List

As another example of the manipulation of linked lists, we present a simple function for reversing the elements of a doubly linked list. Given a list L , our approach involves first copying the contents of L in reverse order into a temporary list T , and then copying the contents of T back into L (but without reversing).

To achieve the initial reversed copy, we repeatedly extract the first element of L and copy it to the front of T . (To see why this works, observe that the later an element appears in L , the earlier it will appear in T .) To copy the contents of T back to L , we repeatedly extract elements from the front of T , but this time we copy each one to the back of list L . Our C++ implementation is presented in Code Fragment 3.35.

```
void listReverse(DLinkedList& L) {           // reverse a list
    DLinkedList T;                          // temporary list
    while (!L.empty()) {                    // reverse L into T
        string s = L.front(); L.removeFront();
        T.addFront(s);
    }
    while (!T.empty()) {                    // copy T back to L
        string s = T.front(); T.removeFront();
        L.addBack(s);
    }
}
```

Code Fragment 3.35: A function that reverses the contents of a doubly linked list L .