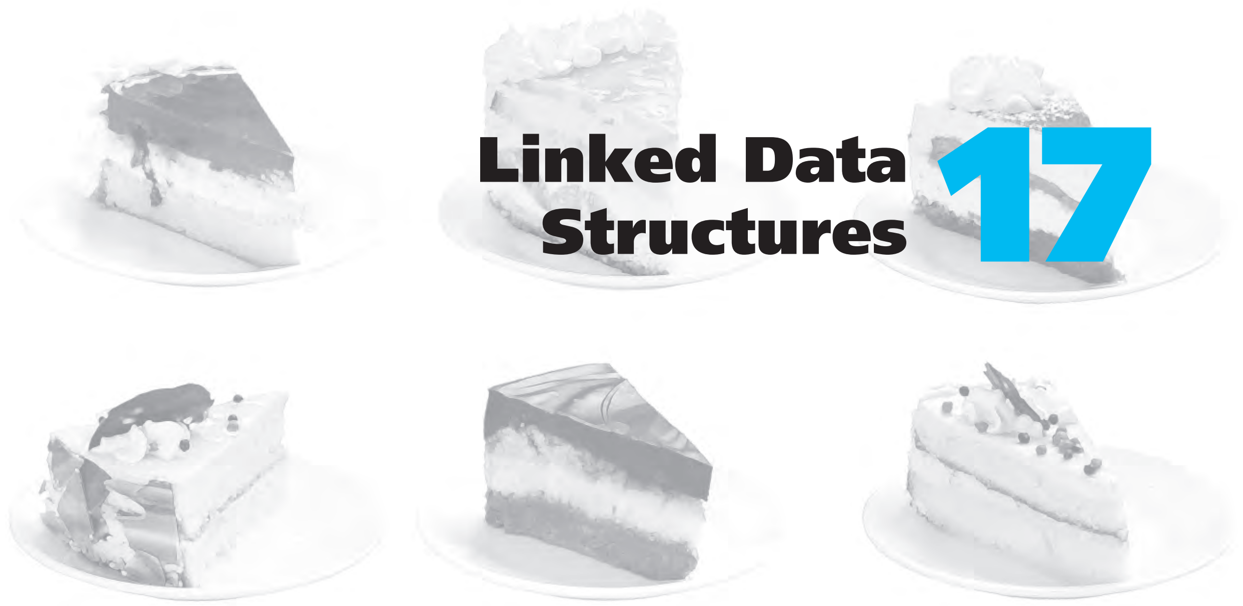


Linked Data Structures 17

17 Linked Data Structures

If somebody there chanced to be
Who loved me in a manner true
My heart would point him out to me
And I would point him out to you.

GILBERT AND SULLIVAN, *Ruddigore*

Introduction

A *linked list* is a list constructed using pointers. A linked list is not fixed in size but can grow and shrink while your program is running. A *tree* is another kind of data structure constructed using pointers. This chapter introduces the use of pointers for building such data structures. The Standard Template Library (STL) has predefined versions of these and other similar data structures. The STL is covered in Chapter 19. It often makes more sense to use the predefined data structures in the STL rather than defining your own. However, there are cases where you need to define your own data structures using pointers. (Somebody had to define the STL.) Also, this material will give you some insight into how the STL might have been defined and will introduce you to some basic widely used material.

Linked data structures produce their structures using dynamic variables, which are created with the `new` operator. The linked data structures use pointers to connect these variables. This gives you complete control over how you build and manage your data structures, including how you manage memory. This allows you to sometimes do things more efficiently. For example, it is easier and faster to insert a value into a sorted linked list than into a sorted array.

There are basically three ways to handle data structures of the kind discussed in this chapter:

1. The C-style approach of using global functions and `struct`s with everything public
2. Using classes with all member variables private and using accessor and mutator functions
3. Using friend classes (or something similar, such as private or protected inheritance or locally defined classes)

We give examples of all three methods. We introduce linked lists using method 1. We then present more details about basic linked lists and introduce both the stack and queue data structures using method 2. We give an alternate definition of our queue template class using friend classes (method 3), and also use friend classes (method 3) to present a tree template class. This way you can see the virtues and shortcomings of each approach. Our personal preference is to use friend classes, but each method has its own advocates.

Sections 17.1 through 17.3 do not use the material in Chapters 13 through 15 (recursion, inheritance, and polymorphism), with one small exception: We have marked our class destructors with the modifier `virtual` following the advice given in Chapter 15. If you have not yet read about virtual functions (Chapter 15), you can pretend that `"virtual"` does not appear in the code. For in the purposes of this chapter, it makes no difference whether `"virtual"` is present or not. Section 17.4 uses recursion (Chapter 13) but does not use Chapters 14 and 15.

17.1 Nodes and Linked Lists

dynamic data structure

A linked list, such as the one diagrammed in Display 17.1, is a simple example of a dynamic data structure. It is called a **dynamic data structure** because each of the boxes in Display 17.1 is a variable of a `struct` or class type that has been dynamically created with the `new` operator. In a dynamic data structure, these boxes, known as **nodes**, contain pointers, diagrammed as arrows, that point to other nodes. This section introduces the basic techniques for building and maintaining linked lists.

Nodes

node structures

A structure like the one shown in Display 17.1 consists of items that we have drawn as boxes connected by arrows. The boxes are called *nodes*, and the arrows represent pointers. Each of the nodes in Display 17.1 contains a string value, an integer, and a pointer that can point to other nodes of the same type. Note that pointers point to the entire node, not to the individual items (such as `10` or `"rolls"`) that are inside the node.

node type definition

Nodes are implemented in C++ as `struct`s or classes. For example, the `struct` type definitions for a node of the type shown in Display 17.1, along with the type definition for a pointer to such nodes, can be as follows:

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
}

Typedef ListNode* ListNodePtr;
```

The order of the type definitions is important. The definition of `ListNode` must come first, since it is used in the definition of `ListNodePtr`.

The box labeled `head` in Display 17.1 is not a node but a pointer variable that can point to a node. The pointer variable `head` is declared as follows:

```
ListNodePtr head;
```

Even though we have ordered the type definitions to avoid some illegal forms of circularity, the preceding definition of the struct type ListNode is still circular. The definition of the type ListNode uses the type name ListNode to define the member variable link. There is nothing wrong with this particular circularity, which is allowed in C++. One indication that this definition is not logically inconsistent is the fact that you can draw pictures, such as Display 17.1, that represent such structures.

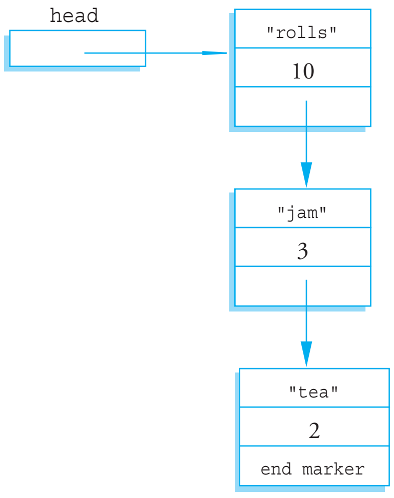
We now have pointers inside structs and have these pointers pointing to structs that contain pointers, and so forth. In such situations the syntax can sometimes get involved, but in all cases the syntax follows those few rules we have described for pointers and structs. As an illustration, suppose the declarations are as just shown, the situation is as diagrammed in Display 17.1, and you want to change the number in the first node from 10 to 12. One way to accomplish this is with the following statement:

changing node data

```
(*head).count = 12;
```

The expression on the left side of the assignment operator may require a bit of explanation. The variable head is a pointer variable. The expression *head is thus the thing it points to, namely the node (dynamic variable) containing "rolls" and the integer 10. This node, referred to by *head, is a struct, and the member variable of this struct, which contains a value of type int, is called count; therefore, (*head).count is the name of the int variable in the first node. The parentheses around *head are not optional. You want the dereferencing operation, *, to be performed before the dot operation. However, the dot operator has higher precedence than the dereferencing

Display 17.1 Nodes and Pointers



operator, `*`, and so without the parentheses, the dot operation would be performed first (which would produce an error). The next paragraph describes a shortcut notation that can avoid this worry about parentheses.

the -> operator

C++ has an operator that can be used with a pointer to simplify the notation for specifying the members of a `struct` or a class. Chapter 10 introduced the arrow operator, `->`, but we have not used it extensively before now. So, a review is in order. The arrow operator combines the actions of a dereferencing operator, `*`, and a dot operator to specify a member of a dynamic `struct` or class object that is pointed to by a given pointer. For example, the previous assignment statement for changing the number in the first node can be written more simply as

```
head->count = 12;
```

This assignment statement and the previous one mean the same thing, but this one is the form normally used.

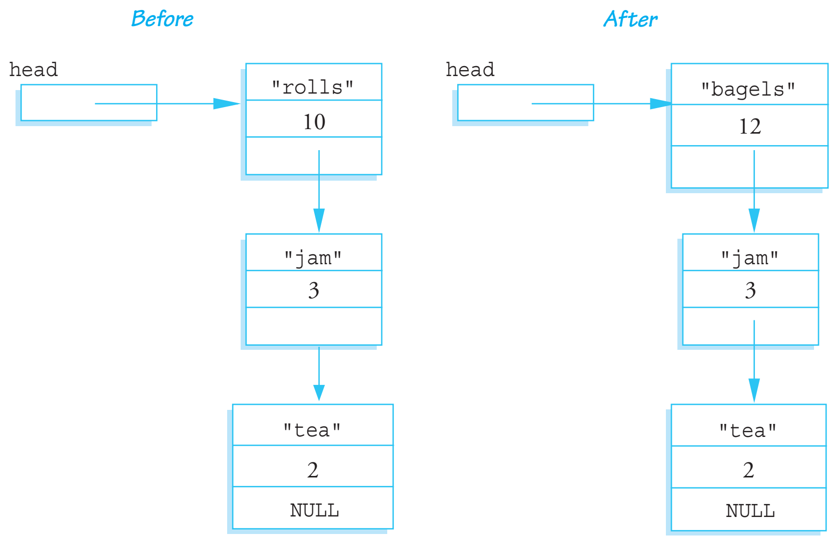
The string in the first node can be changed from `"rolls"` to `"bagels"` with the following statement:

```
head->item = "bagels";
```

The result of these changes to the first node in the list is diagrammed in Display 17.2.

Display 17.2 Accessing Node Data

```
head->count = 12;
head->item = "bagels";
```



The Arrow Operator, ->

The arrow operator, `->`, specifies a member of a `struct` or a member of a class object that is pointed to by a pointer variable. The syntax is

```
Pointer_Variable->Member_Name
```

This refers to a member of the `struct` or class object pointed to by the `Pointer_Variable`. Which member it refers to is given by the `Member_Name`. For example, suppose you have the following definition:

```
struct Record
{
    int number;
    char grade;
};
```

The following creates a dynamic variable of type `Record` and sets the member variables of the dynamic `struct` variable to `2001` and `'A'`:

black
text

```
Record *p;
p = new Record;
p->number = 2001;
p->grade = 'A';
```

NULL

Look at the pointer member in the last node in the list shown in Display 17.2. This last node has the word `NULL` written where there should be a pointer. In Display 17.1, we filled this position with the phrase "end marker," but "end marker" is not a C++ expression. In C++ programs we use the constant `NULL` as a marker to signal the end of a linked list (or the end of any other kind of linked data structure).

`NULL` is typically used for two different (but often coinciding) purposes. First, `NULL` is used to give a value to a pointer variable that otherwise would not have any value. This prevents an inadvertent reference to memory, since `NULL` is not the address of any memory location. The second category of use is that of an end marker. A program can step through the list of nodes as shown in Display 17.2 and know that it has come to the end of the list when the program reaches the node that contains `NULL`.

NULL is 0

As noted in Chapter 10, the constant `NULL` is actually the number 0, but we prefer to think of it and spell it as `NULL` to make it clear that it means this special-purpose value that you can assign to pointer variables. The definition of the identifier `NULL` is in a number of the standard libraries, such as `<iostream>` and `<cstddef>`, so you should use an `include` directive with either `<iostream>`, `<cstddef>`, or some other suitable library when you use `NULL`. The definition of `NULL` is handled by the C++ preprocessor, which replaces `NULL` with `0`. Thus, the compiler never actually sees `"NULL"`, so there are no namespace issues; therefore, no `using` directive is needed for `NULL`.

A pointer can be set to NULL using the assignment operator, as in the following, which declares a pointer variable called there and initializes it to NULL:

```
double *there = NULL;
```

The constant NULL can be assigned to a pointer variable of any pointer type.

NULL

NULL is a special constant value that is used to give a value to a pointer variable that would not otherwise have a value. NULL can be assigned to a pointer variable of any type. The identifier NULL is defined in a number of libraries including the library with header file <cstddef> and the library with header file <iostream>. The constant NULL is actually the number 0, but we prefer to think of it and spell it as NULL.

Linked Lists as Arguments

You should always keep one pointer variable pointing to the head of a linked list. This pointer variable is a way to name the linked list. When you write a function that takes a linked list as an argument, this pointer (which points to the head of the linked list) can be used as the linked list argument.

MyProgrammingLab™

Self-Test Exercises

1. Suppose your program contains the following type definitions:

```
struct Box
{
    string name;
    int number;
    Box *next;
};

typedef Box* BoxPtr;
```

What is the output produced by the following code?

```
BoxPtr head;
head = new Box;
head->name = "Sally";
head->number = 18;
cout << (*head).name << endl;
cout << head->name << endl;
cout << (*head).number << endl;
cout << head->number << endl;
```

(continued)

MyProgrammingLab™

Self-Test Exercises (continued)

2. Suppose that your program contains the type definitions and code given in Self-Test Exercise 1. That code creates a node that contains the string `"Sally"` and the number `18`. What code would you add to set the value of the member variable `next` of this node equal to `NULL`?

3. Consider the following structure definition:

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
ListNode *head = new ListNode;
```

   Give code to assign the string `"Wilbur's brother Orville"` to the member variable `item` of the variable to which `head` points.

Linked Lists

linked list

head

Lists such as those shown in Display 17.1 are called *linked lists*. A **linked list** is a list of nodes in which each node has a member variable that is a pointer that points to the next node in the list. The first node in a linked list is called the **head**, which is why the pointer variable that points to the first node is named `head`. Note that the pointer named `head` is not itself the head of the list but only points to it. The last node has no special name, but it does have a special property: It has `NULL` as the value of its member pointer variable. To test whether a node is the last node, you need only test whether the pointer variable in the node is equal to `NULL`.

node type definition

Our goal in this section is to write some basic functions for manipulating linked lists. For variety, and to simplify the notation, we will use a simpler type of data for the nodes than that used in Display 17.2. These nodes will contain only an integer and a pointer. However, we will make our nodes more complicated in one sense. We will make them objects of a class, rather than just a simple `struct`. The node and pointer type definitions that we will use are as follows:

```
class IntNode
{
public:
    IntNode( ) {}
    IntNode(int theData, IntNode* theLink)
             : data(theData), link(theLink) {}
    IntNode* getLink( ) const { return link; }
    int getData( ) const { return data; }
    void setData(int theData) { data = theData; }
    void setLink(IntNode* pointer) { link = pointer; }
private:
    int data;
    IntNode *link;
};
typedef IntNode* IntNodePtr;
```

Note that all the member functions in the class `IntNode` are simple enough to have inline definitions.

Notice the two-parameter constructor for the class `IntNode`. It will allow us to create nodes with a specified integer as data and with a specified link member. For example, if `p1` points to a node `n1`, then the following creates a new node pointed to by `p2` such that this new node has data `42` and has its link member pointing to `n1`:

```
IntNodePtr p2 = new IntNode(42, p1);
```

After we derive some basic functions for creating and manipulating linked lists with this node type, we will convert the node type and the functions to template versions so they will work to store any type of data in the nodes.

a one-node linked list

As a warm-up exercise, let us see how we might construct the start of a linked list with nodes of this type. We first declare a pointer variable, called `head`, that will point to the head of our linked list:

```
IntNodePtr head;
```

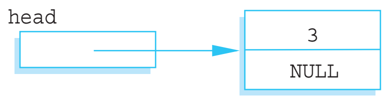
To create our first node, we use the operator `new` to create a new dynamic variable that will become the first node in our linked list:

```
head = new IntNode;
```

We then give values to the member variables of this new node:

```
head->setData(3);
head->setLink(NULL);
```

Notice that the pointer member of this node is set equal to `NULL` because this node is the last node in the list (as well as the first node in the list). At this stage our linked list looks like this:



That was more work than we needed to do. By using the `IntNode` constructor with two parameters, we can create our one-node linked list much easier. The following is an easier way to obtain the one-node linked list just pictured:

```
head = new IntNode(3, NULL);
```

As it turns out, we will always create new nodes using this two-argument constructor for `IntNode`. Many programs would even omit the zero-argument constructor from the definition of `IntNode` so that it would be impossible to create a node without specifying values for each member variable.

Our one-node list was built in an ad hoc way. To have a larger linked list, your program must be able to add nodes in a systematic way. We next describe one simple way to insert nodes in a linked list.

Inserting a Node at the Head of a List

In this subsection we assume that our linked list already contains one or more nodes, and we develop a function to add another node. The first parameter for the insertion function will be a call-by-reference parameter for a pointer variable that points to the head of the linked list—that is, a pointer variable that points to the first node in the linked list. The other parameter will give the number to be stored in the new node. The function declaration for our insertion function is as follows:

```
void headInsert(IntNodePtr& head, int theData);
```

To insert a new node into the linked list, our function will use the `new` operator and our two-argument constructor for `IntNode`. The new node will have `theData` as its data and will have its link member pointing to the first node in the linked list (before insertion). The dynamic variable is created as follows:

```
new IntNode(theData, head)
```

We want the pointer `head` to point to this new node, so the function body can simply be

```
{
    head = new IntNode(theData, head);
}
```

Display 17.3 contains a diagram of the action

```
head = new IntNode(theData, head);
```

when `theData` is `12`. The complete function definition is given in Display 17.4.

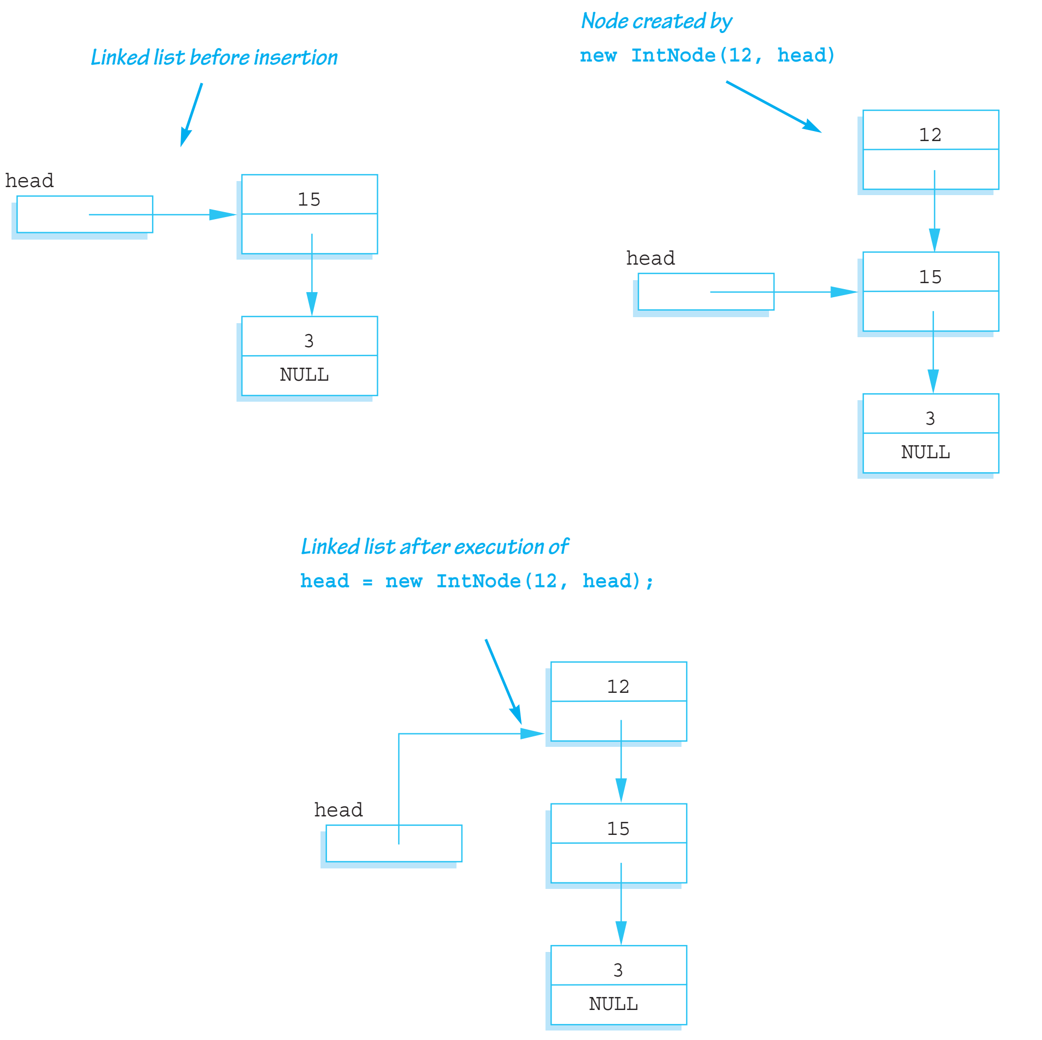
You will want to allow for the possibility that a list contains nothing. For example, a shopping list might have nothing in it because there is nothing to buy this week. A list with nothing in it is called an **empty list**. A linked list is named by naming a pointer that points to the head of the list, but an empty list has no head node. To specify an empty list, use the value `NULL`. If the pointer variable `head` is supposed to point to the head node of a linked list and you want to indicate that the list is empty, then set the value of `head` as follows:

empty list

```
head = NULL;
```

Whenever you design a function for manipulating a linked list, you should always check to see if it works on the empty list. If it does not, you may be able to add a special case for the empty list. If you cannot design the function to apply to the empty

Display 17.3 Adding a Node to the Head of a Linked List


Linked list before insertion
head
15
3
NULL
Node created by
new IntNode(12, head)
12
head
15
3
NULL
Linked list after execution of
head = new IntNode(12, head);
12
head
15
3
NULL

list, then your program must be designed to handle empty lists some other way or to avoid them completely. Fortunately, the empty list can often be treated just like any other list. For example, the function `headInsert` in Display 17.4 was designed with nonempty lists as the model, but a check will show that it works for the empty list as well.

Display 17.4 Functions for Adding a Node to a Linked List

NODE AND POINTER TYPE DEFINITIONS

```
class IntNode
{
public:
    IntNode( ) {}
    IntNode(int theData, IntNode* theLink)
            : data(theData), link(theLink) {}
    IntNode* getLink( ) const { return link; }
    int getData( ) const { return data; }
    void setData(int theData) { data = theData; }
    void setLink(IntNode* pointer) { link = pointer; }
private:
    int data;
    IntNode *link;
};

typedef IntNode* IntNodePtr;
```

FUNCTION TO ADD A NODE AT THE HEAD OF A LINKED LIST

FUNCTION DECLARATION

```
void headInsert(IntNodePtr& head, int theData);
//Precondition: The pointer variable head points to
//the head of a linked list.
//Postcondition: A new node containing theData
//has been added at the head of the linked list.
```

FUNCTION DEFINITION

```
void headInsert(IntNodePtr& head, int theData)
{
   head = new IntNode(theData, head);
}
```

FUNCTION TO ADD A NODE IN THE MIDDLE OF A LINKED LIST
FUNCTION DECLARATION

```
void insert(IntNodePtr afterMe, int theData);
//Precondition: afterMe points to a node in a linked list.
//Postcondition: A new node containing theData
//has been added after the node pointed to by afterMe.
```

FUNCTION DEFINITION

```
void insert(IntNodePtr afterMe, int theData)
{
    afterMe->setLink(new IntNode(theData, afterMe->getLink( )));
}
```

PITFALL: Losing Nodes

You might be tempted to write the function definition for `headInsert` (Display 17.4) using the zero-argument constructor to set the member variables of the new node. If you were to try, you might start the function as follows:

```
head = new IntNode;
head->setData(theData);
```

At this point, the new node is constructed, contains the correct data, and is pointed to by the pointer `head`—all as it is supposed to be. All that is left to do is attach the rest of the list to this node by setting the pointer member in this new node so that it points to what was formerly the first node of the list. You could do it with the following, if only you could figure out what pointer to put in place of the question mark:

```
head->setLink(?);
```

memory leak

Display 17.5 shows the situation when the new data value is `12` and illustrates the problem. If you were to proceed in this way, there would be nothing pointing to the node containing `15`. Since there is no named pointer pointing to it (or to a chain of pointers extending to that node), there is no way the program can reference this node. The node and all nodes below this node are lost. A program cannot make a pointer point to any of these nodes, nor can it access the data in these nodes or do anything else to them. It simply has no way to refer to the nodes. Such a situation ties up memory for the duration of the program. A program that loses nodes is sometimes said to have a **memory leak**. A significant memory leak can result in the program running out of memory and terminating abnormally. Worse, a memory leak (lost nodes) in an ordinary users program can, in rare situations, cause the operating system to crash. To avoid such lost nodes, the program must always keep some pointer pointing to the head of the list, usually the pointer in a pointer variable like `head`. ■

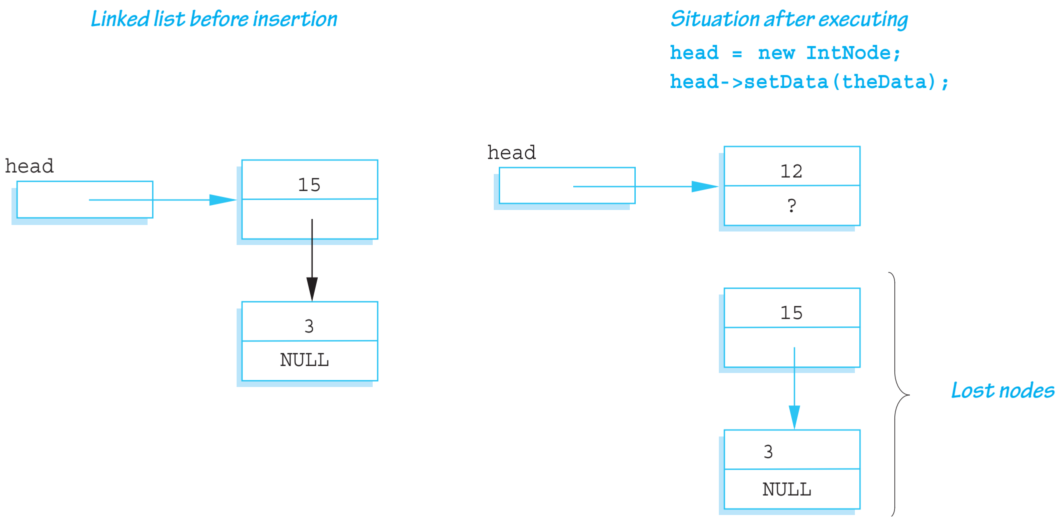
Inserting and Removing Nodes Inside a List

inserting in the middle of a list

We next design a function to insert a node at a specified place in a linked list. If you want the nodes in some particular order, such as numerical or alphabetical, you cannot simply insert the node at the beginning or end of the list. We will therefore design a function to insert a node after a specified node in the linked list.

We assume that some other function or program part has correctly placed a pointer called `afterMe` pointing to some node in the linked list. We want the new node to be placed after the node pointed to by `afterMe`, as illustrated in Display 17.6. The same technique works for nodes with any kind of data, but to be concrete, we are using

Display 17.5 Lost Nodes



the same type of nodes as in previous subsections. The type definitions are given in Display 17.4. The function declaration for the function we want to define is given in the following:

```
void insert(IntNodePtr afterMe, int theData);
//Precondition: afterMe points to a node in a linked list.
//Postcondition: A new node containing theData
//has been added after the node pointed to by afterMe.
```

The new node is inserted inside the list in basically the same way a node is added to the head (start) of a list, which we have already discussed. The only difference is that we use the pointer `afterMe->link` instead of the pointer `head`. The insertion is done as follows:

```
afterMe->setLink(new IntNode(theData, afterMe->getLink( )));
```

The details with `theData` equal to `5` are pictured in Display 17.6, and the final function definition is given in Display 17.4.

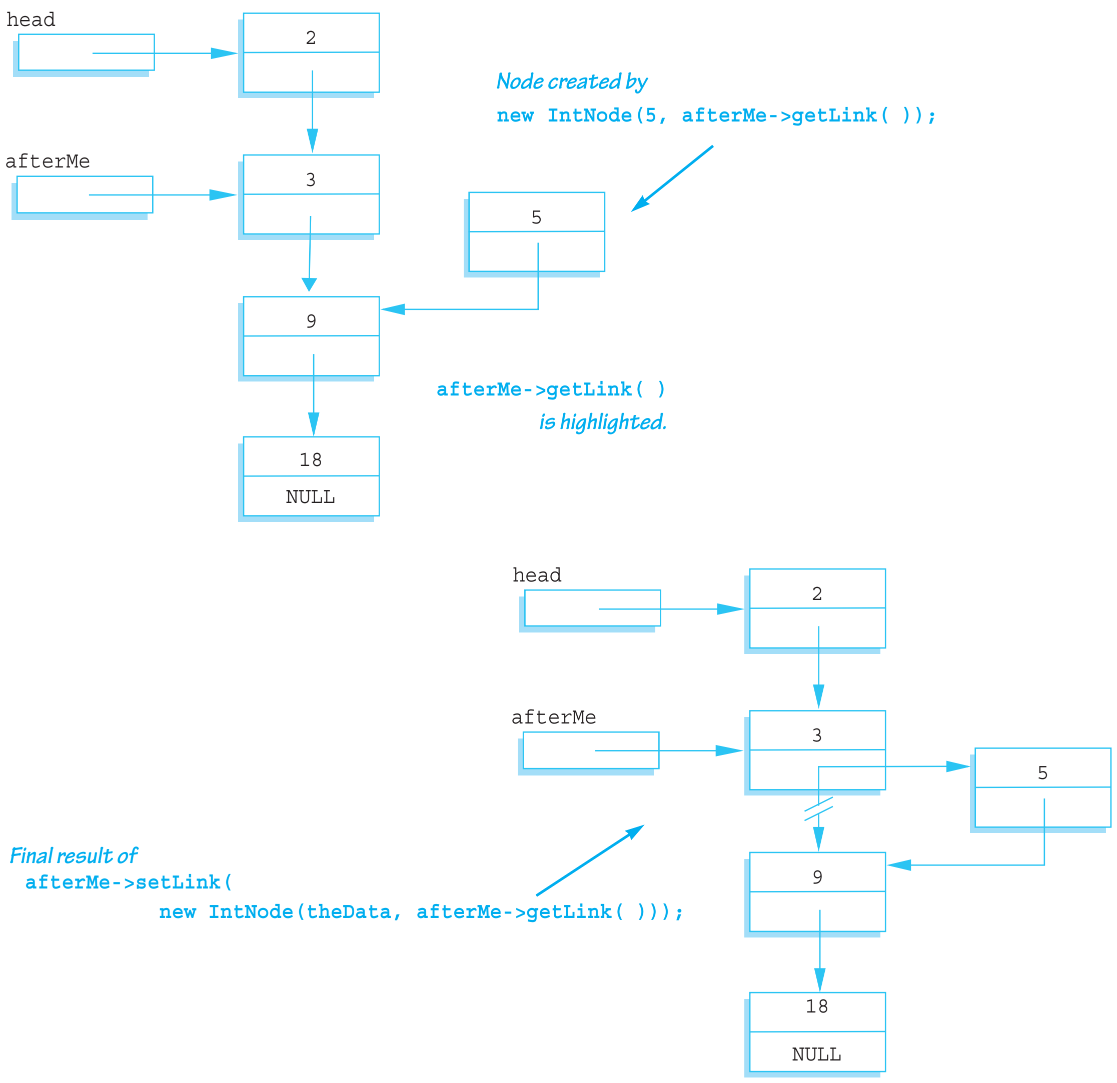
insertion at the ends

If you go through the code for the function `insert`, you will see that it works correctly even if the node pointed to by `afterMe` is the last node in the list. However, `insert` will not work for inserting a node at the beginning of a linked list. The function `headInsert` given in Display 17.4 can be used to insert a node at the beginning of a list.

comparison to arrays

By using the function `insert`, you can maintain a linked list in numerical or alphabetical order or in some other ordering. You can squeeze a new node into the correct position by simply adjusting two pointers. This is true no matter how long the linked list is or where in the list you want the new data to go. If you instead use an array, much—and in extreme cases, all—of the array would have to be copied in order

Display 17.6 Inserting in the Middle of a Linked List



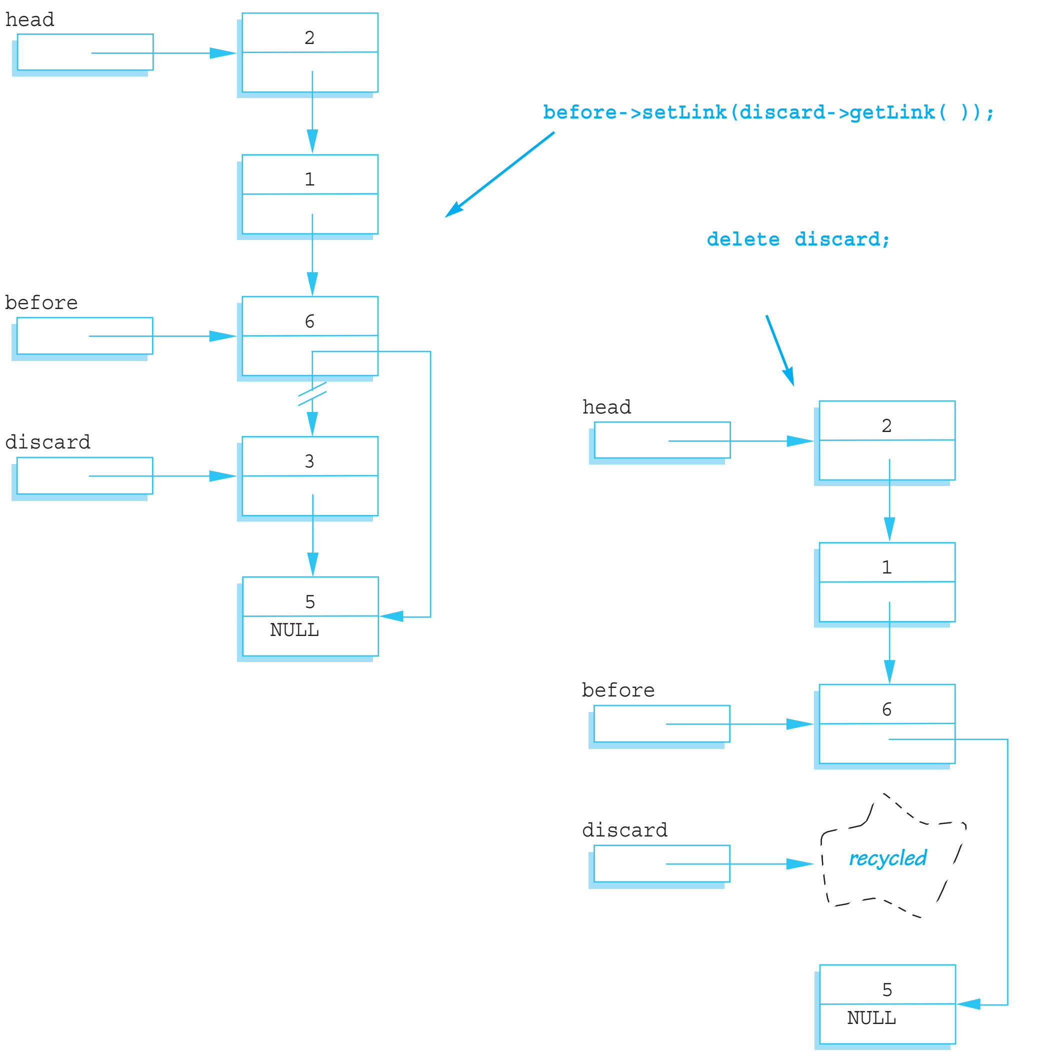
to make room for a new value in the correct spot. Despite the overhead involved in positioning the pointer `afterMe`, inserting into a linked list is frequently more efficient than inserting into an array.

removing a node

Removing a node from a linked list is also quite easy. Display 17.7 illustrates the method. Once the pointers `before` and `discard` have been positioned, all that is required to remove the node is the following statement:

```
before->setLink(discard->getLink( ));
```

Display 17.7 Removing a Node



This is sufficient to remove the node from the linked list. However, if you are not using this node for something else, you should destroy the node and return the memory it uses for recycling; you can do this with a call to `delete` as follows:

```
    delete discard;
```

As we noted in Chapter 10, the memory for dynamic variables is kept in an area of memory known as the *freestore*. Because the freestore is not unlimited, when a dynamic variable (node) is no longer needed by your program, you should return this memory for recycling using the `delete` operator. We include a review of the `delete` operator in the accompanying box.

The `delete` Operator

The `delete` operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore. The memory can then be reused to create new dynamic variables. For example, the following eliminates the dynamic variable pointed to by the pointer variable `p`:

```
delete p;
```

After a call to `delete`, the value of the pointer variable, like `p` just shown, is undefined.

PITFALL: Using the Assignment Operator with Dynamic Data Structures

If `head1` and `head2` are pointer variables and `head1` points to the head node of a linked list, the following will make `head2` point to the same head node and hence the same linked list:

```
head2 = head1;
```

However, you must remember that there is only one linked list, not two. If you change the linked list pointed to by `head1`, then you will also change the linked list pointed to by `head2`, because they are the same linked lists.

If `head1` points to a linked list and you want `head2` to point to a second, identical *copy* of this linked list, the preceding assignment statement will not work. Instead, you must copy the entire linked list node by node. ■

Searching a Linked List

Next we will design a function to search a linked list in order to locate a particular node. We will use the same node type, called `IntNode`, that we used in the previous subsections. (The definitions of the node and pointer types are given in Display 17.4.) The function we design will have two arguments: the linked list and the integer we want to locate. The function will return a pointer that points to the first node that contains that integer. If no node contains the integer, the function will return `NULL`.

This way our program can test whether the `int` is in the list by checking to see if the function returns a pointer value that is not equal to `NULL`. The function declaration and header comment for our function are as follows:

search

```
IntNodePtr search(IntNodePtr head, int target);
//Precondition: The pointer head points to the head of a
//linked list. The pointer variable in the last node is NULL.
//If the list is empty, then head is NULL.
//Returns a pointer that points to the first node that contains the
//target. If no node contains the target, the function returns NULL.
```

We will use a local pointer variable, called `here`, to move through the list looking for the `target`. The only way to move around a linked list, or any other data structure made up of nodes and pointers, is to follow the pointers. Thus, we will start with `here` pointing to the first node and move the pointer from node to node, following the pointer out of each node. This technique is diagrammed in Display 17.8.

Since empty lists present some minor problems that would clutter our discussion, we will at first assume that the linked list contains at least one node. Later we will come back and make sure the algorithm works for the empty list as well. This search technique yields the following algorithm:

algorithm

Pseudocode for `search` Function

```
Make the pointer variable here point to the head node (that is, first node) of the linked list.

while (here is not pointing to a node containing target
                    and here is not pointing to the last node)
{
      Make here point to the next node in the list.
}
if (the node pointed to by here contains target)
      return here;
else
      return NULL;
```

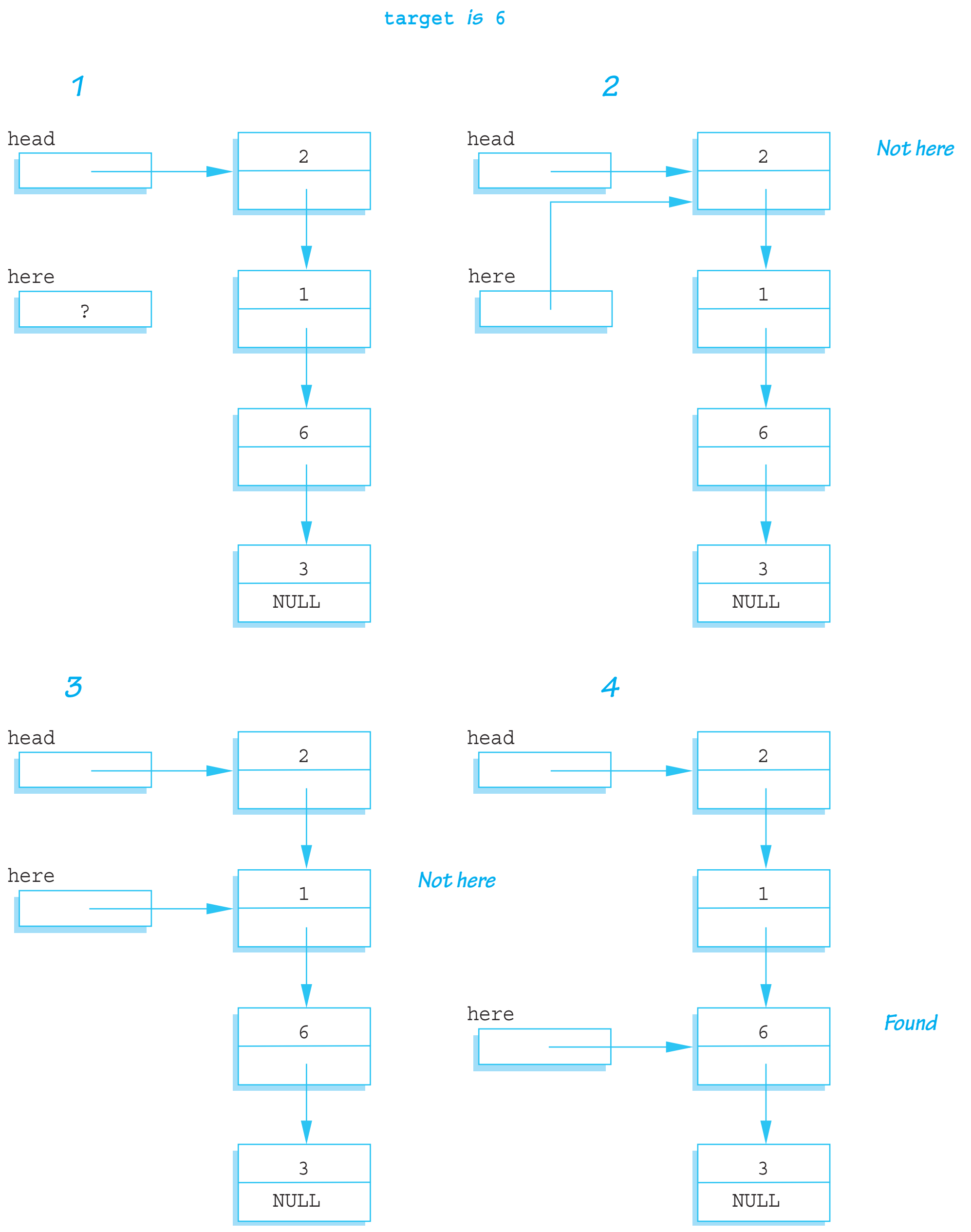
To move the pointer `here` to the next node, we must think in terms of the named pointers we have available. The next node is the one pointed to by the pointer member of the node currently pointed to by `here`. The pointer member of the node currently pointed to by `here` is given by the expression

```
here->getLink( )
```

To move `here` to the next node, we want to change `here` so that it points to the node that is pointed to by the above-named pointer. Hence, the following will move the pointer `here` to the next node in the list:

```
here = here->getLink( );
```

Display 17.8 Searching a Linked List



Putting these pieces together yields the following refinement of the algorithm pseudocode for the `search` function:

algorithm refinement

```
here = head;

while (here->getData( ) != target && here->getLink( ) != NULL)
    here = here->getLink( );

if (here->getData( ) == target)
    return here;
else
    return NULL;
```

Notice the Boolean expression in the `while` statement. We test to see if `here` is pointing to the last node by testing to see if the member variable `here->getLink( )` is equal to `NULL`.

empty list

We still must go back and take care of the empty list. If we check the previous code, we find that there is a problem with the empty list. If the list is empty, then `here` is equal to `NULL` and hence the following expressions are undefined:

```
here->getData( )
here->getLink( )
```

When `here` is `NULL`, it is not pointing to any node, so there is no data member or link member. Hence, we make a special case of the empty list. The complete function definition is given in Display 17.9.

Doubly Linked Lists

doubly linked list

An ordinary linked list allows you to move down the list in only one direction (following the links). A **doubly linked list** has one link that is a pointer to the next node and an additional link that is a pointer to the previous node. In some cases the link to the previous node can simplify our code. For example, if removing a node from the list, we will no longer need to have a `before` variable to remember the node that links to the node we wish to discard. Diagrammatically, a doubly linked list looks like the sample list in Display 17.10.

Display 17.9 **Function to Locate a Node in a Linked List** (part 1 of 2)

FUNCTION DECLARATION

```
IntNodePtr search(IntNodePtr head, int target);
//Precondition: The pointer head points to the head of a
//linked list. The pointer variable in the last node is NULL.
//If the list is empty, then head is NULL.
//Returns a pointer that points to the first node that contains the
//target. If no node contains the target, the function returns NULL.
```

FUNCTION DEFINITION

```
//Uses cstddef:
```

Display 17.9 **Function to Locate a Node in a Linked List** (part 2 of 2)

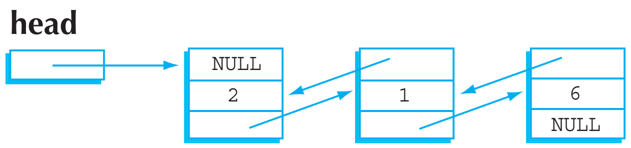
```
IntNodePtr search(IntNodePtr head, int target)
{
        IntNodePtr here = head;

        if (here == NULL) //if empty list
        {
            return NULL;
        }
        else
        {
           while (here->getData( ) != target && here->getLink( ) != NULL)
                 here = here->getLink( );

           if (here->getData( ) == target)
               return here;
           else
               return NULL;
        }
}
```

The definitions of `IntNode` *and* `IntNodePtr` *are given in Display 17.4.*

Display 17.10 **A Doubly Linked List**



The node class for a doubly linked list of integers can be defined as follows:

```
class DoublyLinkedIntNode
{
  public:
    DoublyLinkedIntNode ( ) {}
    DoublyLinkedIntNode (int theData, DoublyLinkedIntNode* previous,
                         DoublyLinkedIntNode* next)
                : data(theData), nextLink(next), previousLink(previous) {}
    DoublyLinkedIntNode* getNextLink( ) const { return nextLink; }
    DoublyLinkedIntNode* getPreviousLink( ) const
         { return previousLink; }
    int getData( ) const { return data; }
    void setData(int theData) { data = theData; }
    void setNextLink(DoublyLinkedIntNode* pointer)
                { nextLink = pointer; }
```

ok as set

```
    void setPreviousLink(DoublyLinkedIntNode* pointer)
                            { previousLink = pointer; }
  private:
    int data;
    DoublyLinkedIntNode *nextLink;
    DoublyLinkedIntNode *previousLink;
};
typedef DoublyLinkedIntNode* DoublyLinkedIntNodePtr;
```

The code is almost identical to the version for the singly linked list except that we have added a private member variable, `previousLink`, to store a link to the previous node in the list. The functions `setPreviousLink` and `getPreviousLink` have been added to get and set the link, along with an additional parameter to the constructor to initialize `previousLink`. What used to be called `link` has also been renamed `nextLink` to differentiate between linking to the previous node or to the next node.

Adding a Node to a Doubly Linked List

adding a node to a doubly linked list

To add a new `DoublyLinkedIntNode` to the front of the list, we must set links on two nodes instead of one. The general process is shown in Display 17.11. The declaration for the insertion function is basically the same as in the singly linked case:

```
    void headInsert(DoublyLinkedIntNodePtr& head, int theData);
```

First, we create a new node whose `nextLink` points to the old head and whose `previousLink` is `NULL`, because it will become the new head:

```
    DoublyLinkedIntNode* newHead = new DoublyLinkedIntNode
          (theData, NULL, head);
```

The old head has to link its previous pointer to the new head:

```
    head->setPreviousLink(newHead);
```

Finally, we set `head` to the new head:

```
    head = newHead;
```

The complete function definition is given in Display 17.13.

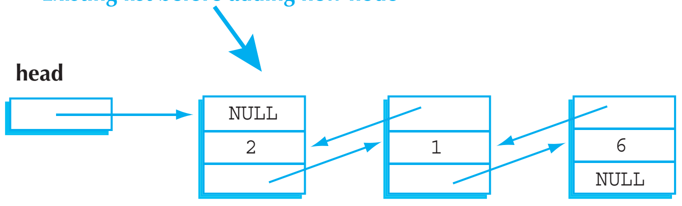
Deleting a Node from a Doubly Linked List

deleting a node from a doubly linked list

To remove a node from the doubly linked list also requires updating the references on both sides of the node we wish to delete. Thanks to the backward link we do not need a separate variable to keep track of the previous node in the list as we did for the singly linked list. The general process of deleting a node referenced by `position` is shown in Display 17.12. Note that some special cases must be handled separately, such as deleting a node from the beginning or the end of the list.

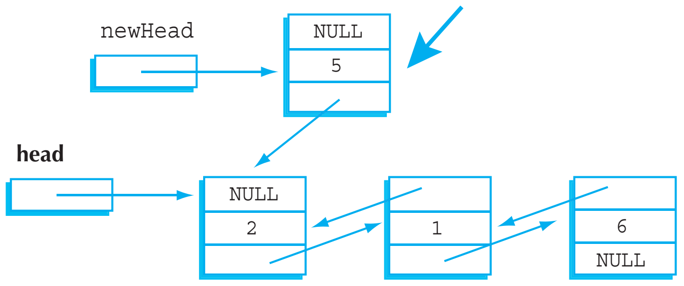
Display 17.11 Adding a Node to the Front of a Doubly Linked List

Existing list before adding new node



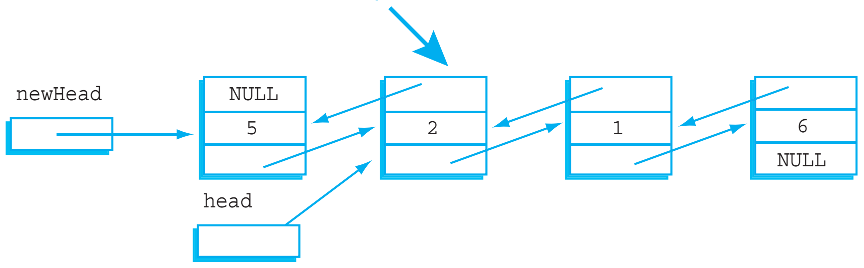
Node created by

```
newHead = new DoublyLinkedIntNode(5, NULL, head);
```



Set the previous link of the original head node

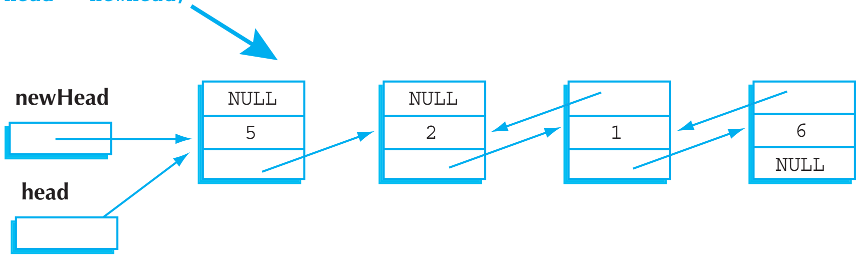
```
head->setPreviousNode(newHead);
```



both in code

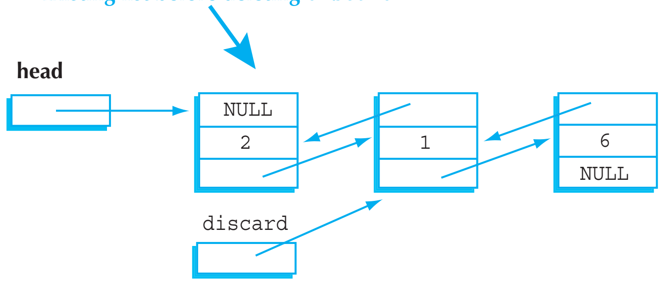
Set head to newHead

```
head = newHead;
```



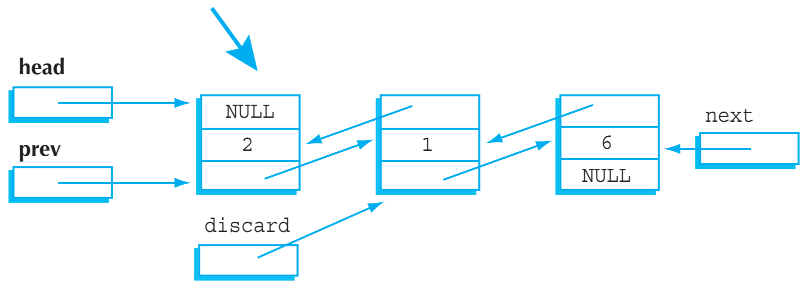
Display 17.12 Deleting a Node from a Doubly Linked List

Existing list before deleting `discard`



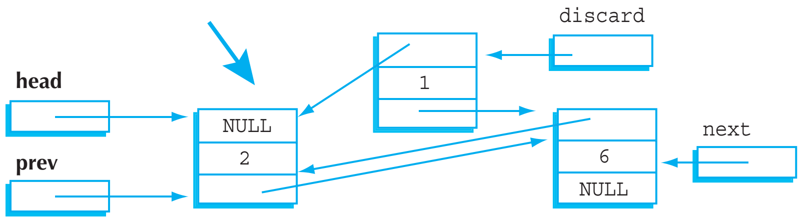
Set pointers to the previous and next nodes

```
DoublyLinkedIntNodePtr prev = discard->getPreviousLink( );
DoublyLinkedIntNodePtr next = discard->getNextLink( );
```



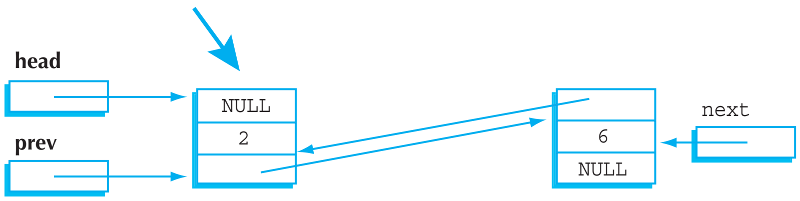
Bypass discard code

```
prev->setNextLink(next);
next->setPreviousLink(prev);
```



Delete discard code

```
delete discard;
```



ok as set

The function declaration for our delete function is now

```
void delete(DoublyLinkedIntNodePtr& head,
       DoublyLinkedIntNodePtr discard);
```

The parameter `discard` is a pointer to the node we wish to remove. We must also input the `head` of the list to handle the case where `discard` is the same as `head`:

```
if (head == discard)
{
   head = head->getNextLink( );
   head->setPreviousLink(NULL);
}
```

In this case we have to advance `head` to the next node in the list. We then set the previous link to `NULL` because there is no prior node. In the more general case, the variable `discard` points to any other node that is not the head. We handle this case by redirecting the links to bypass `discard`, as shown in Display 17.12:

```
else
{
   DoublyLinkedIntNodePtr prev = discard->getPreviousLink( );
   DoublyLinkedIntNodePtr next = discard->getNextLink( );
   prev->setNextLink(next);
   if (next != NULL)
   {
     next->setPreviousLink(prev);
   }
```

The previous node now links to `discard`'s next node, and the next node now links to `discard`'s previous node. Since `discard` might be the last node in the list, we have to check and make sure that `next != NULL` so we do not try to dereference a `NULL` pointer in the function `setPreviousLink`.

The complete function definition is given in Display 17.13.

Display 17.13 **Functions to Add and Remove a Node from a Doubly Linked List** (part 1 of 3)

NODE AND POINTER TYPE DEFINITIONS

```
class DoublyLinkedIntNode
{
public:
    DoublyLinkedIntNode ( ) {}
    DoublyLinkedIntNode (int theData, DoublyLinkedIntNode* previous,
                         DoublyLinkedIntNode* next)
            : data(theData), nextLink(next), previousLink(previous) {}
    DoublyLinkedIntNode* getNextLink( ) const
            { return nextLink; }
```

(continued)

Display 17.13 **Functions to Add and Remove a Node from a Doubly Linked List** (part 2 of 3)

```
    DoublyLinkedIntNode* getPreviousLink( ) const
             { return previousLink; }
    int getData( ) const
             { return data; }
    void setData(int theData)
             { data = theData; }
    void setNextLink(DoublyLinkedIntNode* pointer)
             { nextLink = pointer; }
    void setPreviousLink(DoublyLinkedIntNode* pointer)
             { previousLink = pointer; }
private:
    int data;
    DoublyLinkedIntNode *nextLink;
    DoublyLinkedIntNode *previousLink;
}
typedef DoublyLinkedIntNode* DoublyLinkedIntNodePtr;
```

FUNCTION TO ADD A NODE AT THE HEAD OF A LINKED LIST

FUNCTION DECLARATION

```
void headInsert(DoublyLinkedIntNode& head, int theData);
//Precondition: The pointer variable head points to
//the head of a linked list.
//Postcondition: A new node containing theData
//has been added at the head of the linked list.
```

FUNCTION DEFINITION

```
void headInsert(DoublyLinkedIntNodePtr& head, int theData)
{
     DoublyLinkedIntNode* newHead = new DoublyLinkedIntNode (
     theData, NULL, head);          ⟶   indent line
     head->setPreviousLink(newHead);
     head = newHead;
}
```

FUNCTION TO REMOVE A NODE

FUNCTION DECLARATION

```
void deleteNode(DoublyLinkedIntNodePtr& head,
                DoublyLinkedIntNodePtr discard);
//Precondition: The pointer variable head points to
//the head of a linked list and discard points to the node to remove.
//Postcondition: The node pointed to by discard is removed from the list.
```

Display 17.13 **Functions to Add and Remove a Node from a Doubly Linked List** (part 3 of 3)

FUNCTION DEFINITION

```
void deleteNode(DoublyLinkedIntNodePtr& head,
                DoublyLinkedIntNodePtr discard);
{
    if (head == discard)
    {
      head = head->getNextLink( );
      head->setPreviousLink(NULL);
    }
    else
    {
      DoublyLinkedIntNodePtr prev = discard->getPreviousLink( );
      DoublyLinkedIntNodePtr next = discard->getNextLink( );
      prev->setNextLink(next);
      if (next != NULL)
      {
        next->setPreviousLink(prev);
      }
    }
    delete discard;
}
```

MyProgrammingLab™

Self-Test Exercises

4. Write type definitions for the nodes and pointers in a linked list. Call the node type `NodeType` and call the pointer type `PointerType`. The linked lists will be lists of letters.

5. A linked list is normally referred to via a pointer that points to the first node in the list, but an empty list has no first node. What pointer value is normally used to represent an empty list?

6. Suppose your program contains the following type definitions and pointer variable declarations:

```
struct Node
{
   double data;
   Node *next;
}

typedef Node* Pointer;
Pointer p1, p2;
```

(continued)

MyProgrammingLab™

Self-Test Exercises (continued)

Suppose p1 points to a node of the above type that is in a linked list. Write code that will make p1 point to the next node in this linked list. (The pointer p2 is for the next exercise and has nothing to do with this exercise.)

7. Suppose your program contains type definitions and pointer variable declarations as in Self-Test Exercise 6. Suppose further that p2 points to a node of the above type that is in a linked list and that is not the last node on the list. Write code that will delete the node *after* the node pointed to by p2. After this code is executed, the linked list should be the same, except that there will be one less node in the linked list. (*Hint*: You may want to declare another pointer variable to use.)

8. Suppose your program contains the following type definitions and pointer variable declarations:

```
class Node
{
public:
    Node(double theData, Node* theLink)
            : data(theData), next(theLink) {}
    Node* getLink( ) const { return next; }
    double getData( ) const { return data; }
    void setData(double theData) { data = theData; }
    void setLink(Node* pointer) { next = pointer; }
private:
    double data;
    Node *next;

}
typedef Node* Pointer;
Pointer p1, p2;
```

Suppose p1 points to a node of the above type that is in a linked list. Write code that will make p1 point to the next node in this linked list. (The pointer p2 is for the next exercise and has nothing to do with this exercise.)

9. Suppose your program contains type definitions and pointer variable declarations as in Self-Test Exercise 8. Suppose further that p2 points to a node of the previous type that is in a linked list and that is not the last node on the list. Write code that will delete the node *after* the node pointed to by p2. After this code is executed, the linked list should be the same, except that there will be one less node in the linked list. (*Hint*: You may want to declare another pointer variable to use.)

MyProgrammingLab™

Self-Test Exercises (continued)

10. Choose an ending to the following statement, and explain:

    For a large array and a large list holding the same type objects, inserting a new object at a known location into the middle of a linked list compared to insertion in an array is

    a. more efficient.
    b. less efficient.
    c. about the same.

11. Complete the body of the following function:

    ```
    void insert(DoublyLinkedIntNodePtr afterMe, int theData);
    ```

    The function should insert a new node with the value in `theData` after the node `afterMe` in a doubly linked list.

12. What operations are easier to implement with a doubly linked list than with a singly linked list? What operations are more difficult?

EXAMPLE: A Generic Sorting Template Version of Linked List Tools

It is a routine matter to convert our type definitions and function definitions to templates so that they will work for linked lists with data of any type `T` in the nodes. However, there are some details to worry about. The heart of what you need to do is replace the data type of the data in a node (the type `int` in Display 17.4) by a type parameter `T` and insert the following at the appropriate locations:

```
template<class T>
```

However, you should also do a few more things to account for the fact that the type `T` might be a class type. Since the type `T` might be a class type, a value parameter of type `T` should be changed to a constant reference parameter and a returned type of type `T` should have a `const` added so that it is returned by constant value. (The reason for returning by `const` value is explained in Chapter 8.)

The final templates with the changes we described are shown in Displays 17.14 and 17.15. It was necessary to do one more change from the simple case of a linked list of integers. Since template `typedef`s are not implemented in most compilers, we have not been able to use them. This means that on occasion we needed to use the following hard-to-read parameter type specification:

```
Node<T>*&
```

(continued)

EXAMPLE: (continued)

This is a call-by-reference parameter for a pointer to a node of type `Node<T>`. Next, we have reproduced a function declaration from Display 17.15 so you can see this parameter type specification in context:

```
template<class T>
void headInsert(Node<T>*& head, const T& theData);
```

Display 17.14 **Interface File for a Linked List Library** (part 1 of 2)

```
//This is the header file listtools.h. This contains type definitions
//and function declarations for manipulating a linked list to store
//data of any type T. The linked list is given as a pointer of type
//Node<T>* that points to the head (first) node of the list. The
//implementation of the functions is given in the file listtools.cpp
#ifndef LISTTOOLS_H
#define LISTTOOLS_H

namespace LinkedListSavitch
{
    template<class T>
    class Node
    {
    public:
        Node(const T& theData, Node<T>* theLink) : data(theData),
                                                   link(theLink){}
        Node<T>* getLink( ) const { return link; }
        const T getData( ) const { return data; }
        void setData(const T& theData) { data = theData; }
        void setLink(Node<T>* pointer) { link = pointer; }
    private:
        T data;
        Node<T> *link;
    };
    template<class T>
    void headInsert(Node<T>*& head, const T& theData);
    //Precondition: The pointer variable head points to
    //the head of a linked list.
    //Postcondition: A new node containing theData
    //has been added at the head of the linked list.

    template<class T>
    void insert(Node<T>* afterMe, const T& theData);
    //Precondition: afterMe points to a node in a linked list.
    //Postcondition: A new node containing theData
    //has been added after the node pointed to by afterMe.
```

It would be acceptable to use `T` as a parameter type where we have used `const T&`. We used a constant reference parameter because we anticipate that `T` will frequently be a class type.

Display 17.14 **Interface File for a Linked List Library** (part 2 of 2)

```
    template<class T>
    void deleteNode(Node<T>* before);
    //Precondition: The pointer before points to a node that has
    //at least one node after it in the linked list.
    //Postcondition: The node after the node pointed to by before
    //has been removed from the linked list and its storage
    //returned to the freestore.

    template<class T>
    void deleteFirstNode(Node<T>*& head);
    //Precondition: The pointer head points to the first
    //node in a linked list with at least one node.
    //Postcondition: The node pointed to by head has been removed
    //from the linked list and its storage returned to the freestore.

    template<class T>
    Node<T>* search(Node<T>* head, const T& target);
    //Precondition: The pointer head points to the head of a linked list.
    //The pointer variable in the last node is NULL.
    //== is defined for type T.
    //(== is used as the criterion for being equal.)
    //If the list is empty, then head is NULL.
    //Returns a pointer that points to the first node that
    //is equal to the target. If no node equals the target,
    //then the function returns NULL.
}//LinkedListSavitch

#endif //LISTTOOLS_H
```

Display 17.15 **Implementation File for a Linked List Library** (part 1 of 2)

```
//This is the implementation file listtools.cpp. This file contains
//function definitions for the functions declared in listtools.h.
#include <cstddef>
#include "listtools.h"

namespace LinkedListSavitch
{
    template<class T>
    void headInsert(Node<T>*& head, const T& theData)
    {
        head = new Node<T>(theData, head);
    }
```

(continued)

Display 17.15 **Implementation File for a Linked List Library** (part 2 of 2)

```
    template<class T>
    void insert(Node<T>* afterMe, const T& theData)
    {
        afterMe->setLink(new Node<T>(theData, afterMe->getLink( )));
    }

    template<class T>
    void deleteNode(Node<T>* before)
    {
        Node<T> *discard;
        discard = before->getLink( );
        before->setLink(discard->getLink( ));
        delete discard;
    }

    template<class T>
    void deleteFirstNode(Node<T>*& head)
    {
        Node<T> *discard;
        discard = head;
        head = head->getLink( );
        delete discard;
    }

    //Uses cstddef:
    template<class T>
    Node<T>* search(Node<T>* head, const T& target)
    {
        Node<T>* here = head;
        if (here == NULL) //if empty list
        {
            return NULL;
        }
        else
        {
            while (here->getData( ) != target && here->getLink( ) != NULL)
                here = here->getLink( );

            if (here->getData( ) == target)
                return here;
            else
                return NULL;
        }
    }
}//LinkedListSavitch
```

17.2 Linked List Applications

But many who are first now will be last, and many who are last now will be first.

Matthew 19:30

First come first served

A common (and more secular) saying

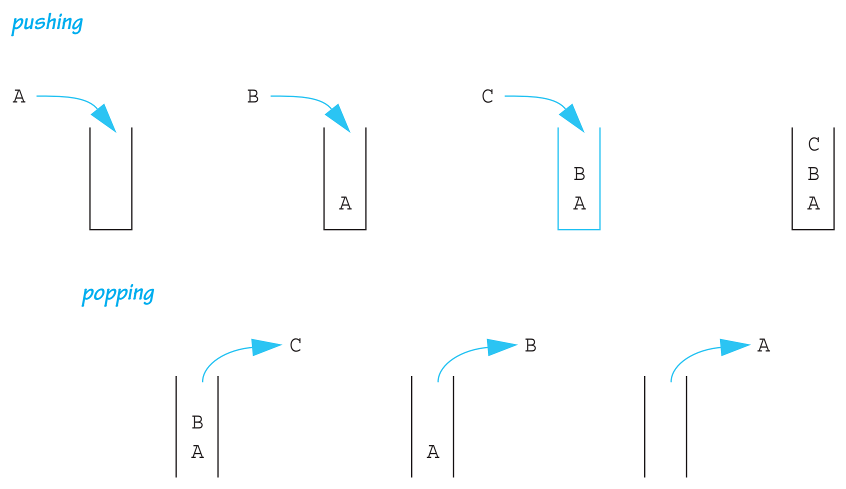
Linked lists have many applications. This section presents only a few small examples of their use—namely, common data structures that all use a linked list as the heart of their implementation.

EXAMPLE: A Stack Template Class

A **stack** is a data structure that retrieves data in the reverse of the order in which the data is stored. Suppose you place the letters `'A'`, `'B'`, and then `'C'` in a stack. When you take these letters out of the stack, they will be removed in the order `'C'`, then `'B'`, and then `'A'`. This use of a stack is diagrammed in Display 17.16. As shown there, you can think of a stack as a hole in the ground. In order to get something out of the stack, you must first remove the items on top of the one you want. For this reason a stack is often called a *last-in/first-out* data structure.

(continued)

Display 17.16 A Stack



EXAMPLE: (continued)

Stacks are used for many language processing tasks. Chapter 13 discussed how the computer system uses a stack to keep track of C++ function calls. However, here we will only be concerned with one very simple application. Our goal in this example is to show how you can use the linked list techniques to implement specific data structures, such as a stack.

The interface for our `stack` class is given in Display 17.17. This is a template class with a type parameter `T` for the type of data stored in the stack. One item stored in the stack is a value of type `T`. In the example we present, `T` is replaced by the type `char`. However, in most applications, an item stored in the stack is likely to be a `struct` or class object. Each record (item of type `T`) that is stored in the stack is called a **stack frame**, which will explain why we occasionally use `stackFrame` as an identifier name in the definition of the stack template class. There are two basic operations you can perform on a stack: adding an item to the stack and removing an item from the stack. Adding an item is called **pushing** the item onto the stack, and so we called the member function that does this `push`. Removing an item from a stack is called **popping** the item off the stack, and so we called the member function that does this `pop`.

push

pop

The names `push` and `pop` derive from a particular way of visualizing a stack. A stack is analogous to a mechanism that is sometimes used to hold plates in a cafeteria. The mechanism stores plates in a hole in the countertop. There is a spring underneath the plates with its tension adjusted so that only the top plate protrudes above the countertop. If this sort of mechanism were used as a stack data structure, the data would be written on plates (which might violate some health laws, but still makes a good analogy). To add a plate to the stack, put it on top of the other plates, and the weight of this new plate pushes down the spring. When you remove a plate, the plate below it pops into view.

Display 17.18 shows a simple program that illustrates how the `Stack` class is used. This program reads a line of text one character at a time and places the characters in a stack. The program then removes the characters one by one and writes them to the screen. Because data is removed from a stack in the reverse of the order in which it enters the stack, the output shows the line written backward. We have `#included` the implementation of the `Stack` class in our application program, as we normally do with template classes. That means we cannot run or even compile our application program until we do the implementation of our `Stack` class template.

The definitions of the member functions for the template class `Stack` are given in the implementation file shown in Display 17.19. Our stack class is implemented as a linked list in which the head of the list serves as the `top` of the stack. The member

(continued on page 769)

Display 17.17 Interface File for a `Stack` Template Class

```
//This is the header file stack.h. This is the interface for the class
//Stack, which is a template class for a stack of items of type T.
#ifndef STACK_H
#define STACK_H

namespace StackSavitch
{
    template<class T>
    class Node
    {
    public:
Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
        Node<T>* getLink( ) const { return link; }
        const T getData( ) const { return data; }
        void setData(const T& theData) { data = theData; }
        void setLink(Node<T>* pointer) { link = pointer; }
    private:
        T data;
        Node<T> *link;
    };
       template<class T>
    class Stack
    {
 public:
        Stack( );
        //Initializes the object to an empty stack.

        Stack(const Stack<T>& aStack);

        Stack<T>& operator =(const Stack<T>& rightSide);

        virtual ~Stack( );

        void push(T stackFrame);
        //Postcondition: stackFrame has been added to the stack.

        T pop( );
        //Precondition: The stack is not empty.
        //Returns the top stack frame and removes that top
        //stack frame from the stack.
        bool isEmpty( ) const;
        //Returns true if the stack is empty. Returns false otherwise.
    private:
        Node<T> *top;
    };

} //StackSavitch
#endif //STACK_H
```

You might prefer to replace the parameter type `T` *with* `const T&`.

Copy Constructor.

The destructor destroys the stack and returns all the memory to the freestore.

Display 17.18 **Program Using the `Stack` Template Class** (part 1 of 2)

```
//Program to demonstrate use of the Stack template class.
#include <iostream>
#include "stack.h"
#include "stack.cpp"
using std::cin;
using std::cout;
using std::endl;
using StackSavitch::Stack;
int main( )
{
    char next, ans;

    do
    {
        Stack<char> s;
        cout << "Enter a line of text:\n";
        cin.get(next);
        while (next != '\n')
        {
            s.push(next);
            cin.get(next);
        }

        cout << "Written backward that is:\n";
        while ( ! s.isEmpty( ) )
            cout << s.pop( );
        cout << endl;

        cout << "Again?(y/n): ";
        cin >> ans;
        cin.ignore(10000, '\n');
    }while (ans != 'n' && ans != 'N');

    return 0;
}
```

Sample Dialogue

```
Enter a line of text:
straw
Written backward that is:
warts
Again?(y/n): y
```

The `ignore` member of `cin` is discussed in Chapter 9. It discards input remaining on the line.

Display 17.18 **Program Using the `Stack` Template Class** (part 2 of 2)

```
Enter a line of text:
I love C++
Written backward that is:
++C evol I
Again?(y/n): n
```

Display 17.19 **Implementation of the `Stack` Template Class** (part 1 of 2)

```
//This is the implementation file stack.cpp.
//This is the implementation of the template class Stack.
//The interface for the template class Stack is in the header file
//stack.h.

#include <iostream>
#include <cstdlib>
#include <cstddef>
#include "stack.h"
using std::cout;

namespace StackSavitch
{

   //Uses cstddef:
    template<class T>
    Stack<T>::Stack( ) : top(NULL)
    {
        //Intentionally empty
    }

    template<class T>
    Stack<T>::Stack(const Stack<T>& aStack)
<The definition of the copy constructor is Self-Test Exercise 14.>
    template<class T>
    Stack<T>& Stack<T>::operator =(const Stack<T>& rightSide)
<The definition of the overloaded assignment operator is Self-Test Exercise 15.>

    template<class T>
    Stack<T>::~Stack( )
    {
        T next;
```

(continued)

Display 17.19 **Implementation of the `Stack` Template Class** (part 2 of 2)

```
        while (! isEmpty( ))
           next = pop( );//pop calls delete.
    }

    //Uses cstddef:
    template<class T>
    bool Stack<T>::isEmpty( ) const
    {
        return (top == NULL);
    }
    template<class T>
    void Stack<T>::push(T stackFrame)
<The rest of the definition is Self-Test Exercise 13.>

    //Uses cstdlib and iostream:
    template<class T>
    T Stack<T>::pop( )
    {
        if (isEmpty( ))
        {
            cout << "Error: popping an empty stack.\n";
            exit(1);
        }

        T result = top->getData( );

        Node<T> *discard;
        discard = top;
        top = top->getLink( );

        delete discard;
        return result;
    }
}   //StackSavitch
```

EXAMPLE: (continued)

variable `top` is a pointer that points to the head of the linked list. The pointer `top` serves the same purpose as the pointer `head` did in our previous discussions of linked lists.

Self-Test Exercise 13 is to write the definition of the member function `push`. However, we have already given the algorithm for this task. The code for the `push` member function is essentially the same as the function `headInsert` shown in Display 17.15, except that in the member function `push` we use a pointer named `top` in place of a pointer named `head`.

An empty stack is just an empty linked list, so an empty stack is implemented by setting the pointer top equal to `NULL`. Once you realize that `NULL` represents the empty stack, the implementations of the default constructor and of the member function `empty` are obvious.

The definition of the copy constructor is a bit more complicated but does not use any techniques we have not already discussed. The details are left to Self-Test Exercise 14.

The `pop` member function first checks to see if the stack is empty. If it is not empty, it proceeds to remove the top character in the stack. It sets the local variable result equal to the top symbol on the stack as follows:

```
T result = top->getData( );
```

After the data in the top node is saved in the variable `result`, the pointer `top` is moved to the next node in the linked list, effectively removing the top node from the list. The pointer `top` is moved with the statement

```
top = top->getLink( );
```

However, before the pointer `top` is moved, a temporary pointer, called `discard`, is positioned so that it points to the node that is about to be removed from the list. The storage for the removed node can then be recycled with the following call to `delete`:

```
delete discard;
```

Each node that is removed from the linked list by the member function `pop` has its memory recycled with a call to `delete`, so all that the destructor needs to do is remove each item from the stack with a call to `pop`. Each node will then have its memory returned to the freestore for recycling.

Stacks

A *stack* is a last-in/first-out data structure; that is, data items are retrieved in the opposite order to which they were placed in the stack.

Push and Pop

Adding a data item to a stack data structure is referred to as *pushing* the data item onto the stack. Removing a data item from a stack is referred to as *popping* the item off the stack.

MyProgrammingLab™

Self-Test Exercises

13. Give the definition of the member function `push` of the template class `Stack` described in Displays 17.17 and 17.19.
14. Give the definition of the copy constructor for the template class `Stack` described in Displays 17.17 and 17.19.
15. Give the definition of the overloaded assignment operator for the template class `Stack` described in Displays 17.17 and 17.19.

EXAMPLE: A Queue Template Class

A stack is a last-in/first-out data structure. Another common data structure is a **queue**, which handles data in a *first-in/first-out* fashion. A queue can be implemented with a linked list in a manner similar to our implementation of the `Stack` template class. However, a queue needs a pointer at both the head of the list and at the end of the linked list, since action takes place in both locations. It is easier to remove a node from the head of a linked list than from the other end of the linked list. Therefore, our implementation will remove nodes from the head of the list (which we will now call the **front** of the list) and will add nodes to the other end of the list, which we will now call the **back** of the list (or the back of the queue).

The definition of the `Queue` template class is given in Display 17.20. A sample application that uses the class `Queue` is shown in Display 17.21. The definitions of the member functions are left as Self-Test Exercises (but remember that the answers are given at the end of the chapter should you have any problems filling in the details).

Display 17.20 **Interface File for a Queue Template Class** (part 1 of 2)

```

//This is the header file queue.h. This is the interface for the class
//Queue, which is a template class for a queue of items of type T.
#ifndef QUEUE_H
#define QUEUE_H

namespace QueueSavitch
{
    template<class T>
    class Node
    {
    public:
        Node(T theData, Node<T>* theLink) : data(theData),
        link(theLink){}
        Node<T>* getLink( ) const { return link; }
        const T getData( ) const { return data; }
        void setData(const T& theData) { data = theData; }
        void setLink(Node<T>* pointer) { link = pointer; }
    private:
        T data;
        Node<T> *link;
    };

     template<class T>
     class Queue
     {
     public:
         Queue( );
         //Initializes the object to an empty queue.

         Queue(const Queue<T>& aQueue);

         Queue<T>& operator =(const Queue<T>& rightSide);

         virtual ~Queue( );


         void add(T item);
         //Postcondition: item has been added to the back of the queue.

         T remove( );
         //Precondition: The queue is not empty.
         //Returns the item at the front of the queue
         //and removes that item from the queue.
```

This is the same definition of the template class `Node` *that we gave for the stack interface in Display 17.17. See the "Tip: A Comment on Namespaces" for a discussion of this duplication.*

You might prefer to replace the parameter type `T` *with* `const T&`.

Copy constructor.

The destructor destroys the queue and returns all the memory to the freestore.

(continued)

Display 17.20 **Interface File for a Queue Template Class** (part 2 of 2)

```
            bool isEmpty( ) const;
            //Returns true if the queue is empty. Returns false otherwise.
        private:
            Node<T> *front; //Points to the head of a linked list.
                            //Items are removed at the head
            Node<T> *back;  //Points to the node at the other end of the
                            //linked list.
                            //Items are added at this end.
        };

    } //QueueSavitch

    #endif //QUEUE_H
```

Display 17.21 **Program Using the Queue Template Class** (part 1 of 2)

```
//Program to demonstrate use of the Queue template class.
#include <iostream>
#include "queue.h"
#include "queue.cpp"
using std::cin;
using std::cout;
using std::endl;
using QueueSavitch::Queue;

int main( )
{
    char next, ans;

    do
    {
        Queue<char> q;
        cout << "Enter a line of text:\n";
        cin.get(next);
        while (next != '\n')
        {
            q.add(next);
            cin.get(next);
        }

        cout << "You entered:\n";
        while ( ! q.isEmpty( ) )
            cout << q.remove( );
        cout << endl;
```

Contrast this with the similar program using a stack instead of a queue that we gave in Display 17.18.

Display 17.21 **Program Using the Queue Template Class** (part 2 of 2)

```
            cout << "Again?(y/n): ";
            cin >> ans;
            cin.ignore(10000, '\n');
        } while (ans != 'n' && ans != 'N');

        return 0;
    }
```

Sample Dialogue

```
Enter a line of text:
straw
You entered:
straw
Again?(y/n): y
Enter a line of text:
I love C++
You entered:
I love C++
Again?(y/n): n
```

Queue

A *queue* is a *first-in/first-out* data structure; that is, the data items are removed from the queue in the same order that they were added to the queue.

TIP: A Comment on Namespaces

Notice that both of the namespaces `StackSavitch` (Display 17.17) and `QueueSavitch` (Display 17.20) define a template class called `Node`. As it turns out, the two definitions of `Node` are the same, but the point discussed here is the same whether the two definitions are the same or different. C++ does not allow you to define the same identifier twice, even if the two definitions are the same, unless the two names are somehow distinguished. In this case, the two definitions are allowed because they are in two different namespaces. It is even legal to use both the `Stack` template class and the `Queue` template class in the same program. However, you should use

```
using StackSavitch::Stack;
using QueueSavitch::Queue;
```

(continued)

TIP: (continued)

rather than

```
using namespace StackSavitch;
using namespace QueueSavitch;
```

Most compilers will allow either set of `using` directives if you do not use the identifier `Node`, but the second set of `using` directives provides two definitions of the identifier `Node` and therefore should be avoided.

It would be fine to use either, but not both, of the following:

```
using StackSavitch::Node;
```

or

```
using QueueSavitch::Node;
```

MyProgrammingLab™

Self-Test Exercises

16. Give the definitions for the default (zero-argument) constructor and the member functions `Queue<T>::isEmpty` for the template class `Queue` (Display 17.20).
17. Give the definitions for the member functions `Queue<T>::add` and `Queue<T>::remove` for the template class `Queue` (Display 17.20).
18. Give the definition for the destructor for the template class `Queue` (Display 17.20).
19. Give the definition for the copy constructor for the template class `Queue` (Display 17.20).
20. Give the definition for the overloaded assignment operator for the template class `Queue` (Display 17.20).

Friend Classes and Similar Alternatives

You may have found it a nuisance to use the accessor and mutator functions `getLink` and `setLink` in the template class `Node` (see Display 17.17 or Display 17.20). You might be tempted to avoid the invocations of `getLink` and `setLink` by simply making the member variable `link` of the class `Node` public instead of private. Before you abandon the principle of making all member variables private, note two things. First, using `getLink` and `setLink` is not really any harder for you, the programmer, than directly accessing the links in the nodes. (However, `getLink` and `setLink` do introduce some overhead and so may slightly reduce efficiency.) Second, there is a way to avoid using `getLink` and `setLink` and instead directly access the links of nodes without making the `link` member variable public. Let us explore this second possibility.

Chapter 8 discussed friend functions. As you will recall, if `f` is a friend function of a class `C`, then `f` is not a member function of `C`; however, when you write the definition friend class

of the function f, you can access private members of C just as you can in the definitions of member functions of C. A class can be a **friend** of another class in the same way that a function can be a friend of a class. If the class F is a friend of the class C, then every member function of the class F is a friend of the class C. Thus, if, for example, the Queue template class were a friend of the Node template class, then the private link member variables would be directly available in the definitions of the member functions of Queue. The details are outlined in Display 17.22.

Display 17.22 **A Queue Template Class as a Friend of the Node Class** (part 1 of 2)

```
//This is the header file queue.h. This is the interface for the class
//Queue, which is a template class for a queue of items of type T.
#ifndef QUEUE_H
#define QUEUE_H

namespace QueueSavitch
{
    template<class T>
    class Queue;

    template<class T>
    class Node
    {
    public:
        Node(T theData, Node<T>* theLink) : data(theData),
                               link(theLink){}
        friend class Queue<T>;
    private:
        T data;
        Node<T> *link;
    };

    template<class T>
    class Queue
    {
<The definition of the template class Queue is identical to the one given in Display 17.20.
However, the definitions of the member functions will be different from the ones we gave
(in the Self-Test Exercises) for the nonfriend version of Queue.>
    }
}     //QueueSavitch
```

A forward declaration. Do not forget the semicolon.

This is an alternate approach to that given in Display 17.20. In this version, the Queue template class is a friend of the Node template class.

If Node<T> is only used in the definition of the friend class Queue<T>, there is no need for mutator or accessor functions.

```
#endif //QUEUE_H
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include "queue.h"
using std::cout;
namespace QueueSavitch
```

The implementation file would contain these definitions and the definitions of the other member functions similarly modified to allow access by name to the link and data member variables of the nodes.

(continued)

Display 17.22 A Queue Template Class as a Friend of the `Node` Class (part 2 of 2)

```
{
    template<class T> //Uses cstddef:
    void Queue<T>::add(T item)
    {
       if (isEmpty( ))
            front = back = new Node<T>(item, NULL);
        else
        {
           back->link = new Node<T>(item, NULL);
           back = back->link;
        }
    }

    template<class T>   //Uses cstdlib and iostream:
    T Queue<T>::remove( )
    {
        if (isEmpty( ))
        {
            cout << "Error: Removing an item from an empty queue.\n";
            exit(1);
        }

        T result = front->data;

        Node<T> *discard;
        discard = front;
        front = front->link;
        if (front == NULL) //if you removed the last node
            back = NULL;

        delete discard;
        return result;
    }
} //QueueSavitch
```

If efficiency is a major issue, you might want to use `(front == NULL)` *instead of* `(isEmpty())`.

Contrast these implementations with the ones given as the answer to Self-Test Exercise 17.

forward declaration

When one class is a friend of another class, it is typical for the classes to reference each other in their class definitions. This requires that you include a **forward declaration** to the class or class template defined second, as illustrated in Display 17.22. Note that the forward declaration is just the heading of the class or class template definition followed by a semicolon. A complete example using a friend class is given in Section 17.4 (see the programming example “A Tree Template Class”).

Two approaches that serve pretty much the same purpose as friend classes and that can be used in pretty much the same way with classes and template classes such as `Node` and `Queue` are (1) using protected or private inheritance to derive `Queue` from `Node`, and (2) giving the definition of `Node` within the definition of `Queue`, so that `Node` is a local class (template) definition. (Protected inheritance is discussed in Chapter 14, and classes defined locally within a class are discussed in Chapter 7.)

EXAMPLE: Hash Tables with Chaining

hash table

hash map

A **hash table** or **hash map** is a data structure that efficiently stores and retrieves data from memory. There are many ways to construct a hash table; in this section we will use an array in combination with singly linked lists. In Section 17.1, we searched a linked list by iterating through every node in the list looking for a target. This process might require the examination of every node in the list, a potentially time-consuming process if the list is very long. In contrast, a hash table has the potential to find the target very quickly, although in a worst-case (but highly unlikely) scenario our implementation would run as slowly as using a singly linked list.

hash function

An object is stored in a hash table by associating it with a *key*. Given the key, we can retrieve the object. Ideally, the key is unique to each object. If the object has no intrinsically unique key, then we can use a **hash function** to compute one. In most cases the hash function computes a number.

collision

chaining

For example, let us use a hash table to store a dictionary of words. Such a hash table might be useful to make a spell-checker—words missing from the hash table might not be spelled correctly. We will construct the hash table with a fixed array, where each array element references a linked list. The key computed by the hash function will map to the index of the array. The actual data will be stored in a linked list at the hash value's index. Display 17.23 illustrates the idea with a fixed array of ten entries. Initially each entry of the array `hasharray` contains a reference to an empty singly linked list. First, we add the word "cat," which has been assigned the key or hash value of 2 (we will show how this was computed shortly). Next, we add "dog" and "bird," which are assigned hash values of 4 and 7, respectively. Each of these strings is inserted as the head of the linked list using the hash value as the index in the array. Finally, we add "turtle," which also has a hash of 2. Since "cat" is already stored at index 2, we now have a **collision**. Both "turtle" and "cat" map to the same index in the array. When this occurs in a hash table with **chaining**, we simply insert the new node onto the existing linked list. In our example, there are now two nodes at index 2: "turtle" and "cat."

To retrieve a value from the hash table, we first compute the hash value of the target. Next we sequentially search the linked list that is stored at `hasharray[hashvalue]` for the target. If the target is not found in this linked list, then the target is not stored in the hash table. If the size of the linked list is small, then the retrieval process will be quick.

(continued)

EXAMPLE: (continued)

A HASH FUNCTION FOR STRINGS

A simple way to compute a numeric hash value for a string is to sum the ASCII value of every character in the string and then compute the modulus of the sum using the size of the fixed array. A subset of ASCII codes is given in Appendix 3. Code to compute the hash value is shown in the function `computeHash`.

```
int computeHash(string s)
{
      int hash = 0;
      for (int i = 0; i < s.length( ); i++)
      {
          hash = hash + s[i];
      }
      return hash % SIZE;   //SIZE = 10 in example
}
```

For example, the ASCII codes for the string "dog" are

```
d   ->   100
o   ->   111
g   ->   103
```

The hash function is computed as

```
Sum             = 100 + 111 + 103 = 314
Hash = Sum % 10 = 314 % 10        = 4
```

In this example, we first compute an unbounded value, the sum of the ASCII values in the string. However, the array was defined to hold a finite number of elements. To scale the sum to the size of the array, we compute the modulus of the sum with respect to the size of the array, which is 10 in the example. In practice, the size of the array is generally a prime number larger than the number of items that will be put into the hash table[1]. The computed hash value of 4 serves as a fingerprint for the string "dog." However, other strings may also map to the same value. For example, we can verify that "cat" maps to (99 + 97 + 116) % 10 = 2 and "turtle" maps to (116 + 117 + 114 + 116 + 108 + 101) % 10 = 2.

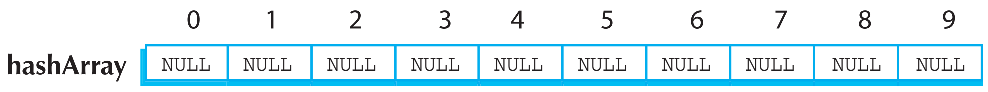
A complete code listing for a hash table class is given in Displays 17.24 and 17.25. A demo is shown in Display 17.26. The hash table definition uses an array where each element is a `Node` class defined in Display 17.14. The linked list is implemented using the generic linked list library defined in Displays 17.14 and 17.15.

[1]A prime number avoids common divisors after modulus that can lead to collisions.

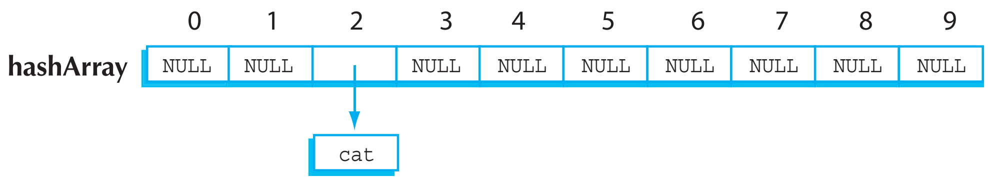
Display 17.23 Constructing a Hash Table

Existing hash table with 10 empty linked lists

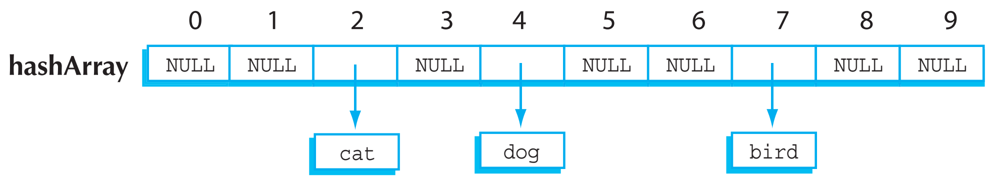
```
Node<string> *hashArray[10];
for (int i=0; i<10; i++) hashArray[i] = NULL;
```



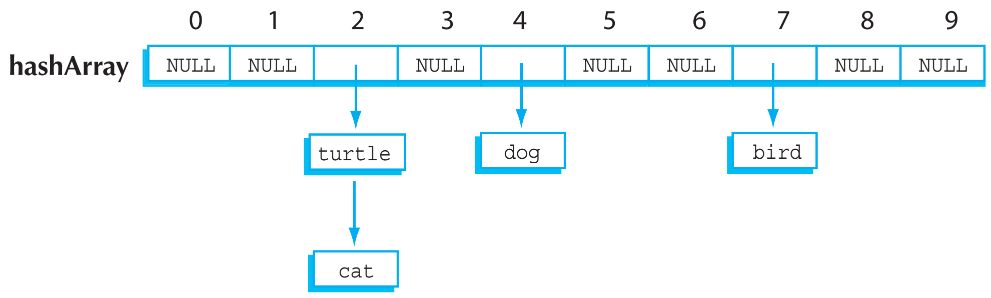
After adding "cat" with a hash of 2



After adding "dog" with a hash of 4 and "bird" with a hash of 7



After adding "turtle" with a hash of 2 - collision and chained to linked list with "cat"



Display 17.24 Interface File for a `HashTable` Class

```
//This is the header file hashtable.h. This is the interface
//for the class HashTable, which is a class for a hash table
//of strings.
#ifndef HASHTABLE_H
#define HASHTABLE_H

#include <string>
#include "listtools.h"

using LinkedListSavitch::Node;
using std::string;

namespace HashTableSavitch
{
 const int SIZE = 10;  //Maximum size of the hash table array

 class HashTable
 {
  public:
      HashTable( );   //Initialize empty hash table.

      //Normally a copy constructor and overloaded assignment
      //operator would be included. They have been omitted
      //to save space.

      virtual ~HashTable( );   //Destructor destroys hash table.

      bool containsString(string target) const;
      //Returns true if target is in the hash table,
      //false otherwise.

      void put(string s);
      //Adds a new string to the hash table.

  private:
      Node<string> *hashArray[SIZE];      //The actual hash table
      static int computeHash(string s);   //Compute a hash value
 };   //HashTable
}   //HashTableSavitch
#endif   //HASHTABLE_H
```

The library "listtools.h" is the linked list library interface from Display 17.14.

Display 17.25 Implementation of the `HashTable` Class (part 1 of 2)

```
//This is the implementation file hashtable.cpp.
//This is the implementation of the class HashTable.

#include <string>
#include "listtools.h"
#include "hashtable.h"

using LinkedListSavitch::Node;
using LinkedListSavitch::search;
using LinkedListSavitch::headInsert;
using std::string;

namespace HashTableSavitch
{
    HashTable::HashTable( )
    {
        for (int i = 0; i < SIZE; i++)
        {
           hashArray[i] = NULL;
        }
    }

    HashTable::~HashTable( )
    {
        for (int i=0; i<SIZE; i++)
        {
            Node<string> *next = hashArray[i];
            while (next != NULL)
            {
                Node<string> *discard = next;
                next = next->getLink( );
                delete discard;
            }
        }
    }

    int HashTable::computeHash(string s)
    {
        int hash = 0;
        for (int i = 0; i < s.length( ); i++)
        {
            hash = hash + s[i];
        }
        return hash % SIZE;
    }
```

(continued)

Display 17.25 **Implementation of the `HashTable` Class** (part 2 of 2)

```
        void HashTable::put(string s)
    {
    int hash = computeHash(s);
    if (search(hashArray[hash], s)==NULL)
    {
        //Only add the target if it's not in the list
        headInsert(hashArray[hash], s);
    }
}
//HashTableSavitch
```

Display 17.26 **Hash Table Demonstration** (part 1 of 2)

```
//Program to demonstrate use of the HashTable class

#include <string>
#include <iostream>
#include "hashtable.h"
#include "listtools.cpp"
#include "hashtable.cpp"
using std::string;
using std::cout;
using std::endl;
using HashTableSavitch::HashTable;

int main( )
{
    HashTable h;

    cout << "Adding dog, cat, turtle, bird" << endl;
    h.put("dog");
    h.put("cat");
    h.put("turtle");
    h.put("bird");
    cout << "Contains dog? " << h.containsString("dog") << endl;
    cout << "Contains cat? " << h.containsString("cat") << endl;
    cout << "Contains turtle? " << h.containsString("turtle") << endl;
    cout << "Contains bird? " << h.containsString("bird") << endl;

    cout << "Contains fish? " << h.containsString("fish") << endl;
    cout << "Contains cow? " << h.containsString("cow") << endl;

    return 0;
}
```

Display 17.26 **Hash Table Demonstration** (part 2 of 2)

Sample Dialogue

```
Adding dog, cat, turtle, bird
Contains dog? 1
Contains cat? 1
Contains turtle? 1
Contains bird? 1
Contains fish? 0
Contains cow? 0
```

Hash Table

A *hash table* is a data structure that associates a data item with a key. The key is computed by a hash function.

Efficiency of Hash Tables

The efficiency of our hash table depends on several factors. First, let us examine some extreme cases. The worst-case run-time performance occurs if every item inserted into the table has the same hash key. Everything will then be stored in a single linked list, and the `find` operation may require searching through each item in the list. Fortunately, if the items that we insert are somewhat random, the possibility that all of them hash to the same key is highly unlikely. In contrast, the best-case run-time performance occurs if every item inserted into the table has a different hash key. This means that there will be no collisions, so the `find` operation will only need to search through a one-item list because the target will always be the first node in the linked list.

We can decrease the chance of collisions by using a better hash function. For example, the simple hash function that sums each letter of a string ignores the ordering of the letters. The words “rat” and “tar” would hash to the same value. A better hash function for a string `s` is to multiply each letter by an increasing weight depending on the position in the word:

```
int hash = 0;
for (int i = 0; i < s.length( ); i++)
{
    hash = 31 * hash + s[i];
}
```

Another way to decrease the chance of collisions is by making the hash table bigger. For example, if the hash table array had a 10,000-entry capacity but we were only inserting 1000 items, then the probability of a collision would be much smaller than if the hash table array could store only 1000 entries. However, inserting only 1000 items time-space tradeoff

in a 10,000-entry hash table would mean 9000 memory locations will go unused, which is a waste of memory. This illustrates the **time-space tradeoff**. It is usually possible to increase run-time performance at the expense of memory space, and vice versa.

MyProgrammingLab™

Self-Test Exercises

21. Suppose that every student in your university is assigned a unique nine-digit ID number. You would like to create a hash table that indexes ID numbers to an object representing a student. The hash table has a size of *N* where *N* has fewer than nine digits. Describe a simple hash function that you can use to map from ID number to a hash index.

22. Write an `outputHashTable( )` function for the `HashTable` class that outputs every item stored in the hash table.

EXAMPLE: A Set Template Class

A *set* is a collection of elements in which no element occurs more than once. Many problems in computer science can be solved with the aid of a set data structure. A variation on linked lists is a straightforward way to implement a set. In this implementation, the items in each set are stored using a singly linked list. The `data` variable for each node simply contains an item in the set.

Display 17.27 illustrates two sample sets stored using this data structure. The set `round` contains "peas," "ball," and "pie" while the set `green` contains "peas" and "grass." The string "peas" is in both sets because it is both round and green. Note that if the data type used to fill the `Node` template is a pointer to an object, then multiple lists might reference a common object instead of creating multiple copies of the same object in each list.

FUNDAMENTAL SET OPERATIONS

Some fundamental operations that our set class should support are

`add element`. Add a new item into a set.
`contains`. Determine if a target item is a member of the set.
`union`. Return a set that is the union of two sets.
`intersection`. Return a set that is the intersection of two sets.

We should also include a way to iterate through each element in the set. Other useful set operations include functions to retrieve the cardinality of the set and to remove items from the set. The implementation of these operations is given as an exercise in Programming Project 17.7.

EXAMPLE: (continued)

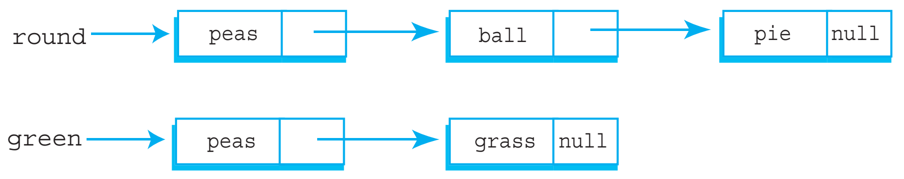
The code for implementing a generic set of elements appears in Displays 17.28 and 17.29. The `Set` class uses the linked list tools from Display 17.14. The `add` function simply adds a node to the front of the linked list, but only if the item is not already in the set. The `contains` function uses the `search` function from the linked list library. We simply loop through every item in the list looking for the target.

The `union` function combines the elements in the calling objects set with the elements from the set of the input argument, `otherSet`. To `union` these sets we first create a new empty `Set` object. Next, we iterate through both the calling object's set and `otherSet`'s set. All elements are added to the new set. The `add` function enforces uniqueness so we do not have to check for duplicate elements in the `union` function.

The `intersection` function is similar to the `union` function in that it also creates a new, empty `Set` object. In this case, we populate the set with items that are common to both the calling object's set and `otherSet`'s set. This is accomplished by iterating through every item in the calling object's set. For each item, we invoke the `contains` function for `otherSet`. If `contains` returns `true`, then the item is in both sets and can be added to the new set.

A short demonstration program is in Display 17.30.

Display 17.27 **Set Implementation Using Linked Lists**



Set

A set is an unordered collection of data elements.

Display 17.28 Interface File for a Set Template Class

```
//This is the header file set.h. This is the interface
//for the class Set, which is a class for a generic set.
#ifndef SET_H
#define SET_H

#include "listtools.h"
using LinkedListSavitch::Node;

namespace SetSavitch
{
  template<class T>
  class Set
  {
   public:
      Set( ) { head = NULL; } //Initialize empty set.

      //Normally a copy constructor and overloaded assignment
      //operator would be included. They have been omitted
      //to save space.

      virtual ~Set( ); //Destructor destroys set.
      bool contains(T target) const;
      //Returns true if target is in the set, false otherwise.

      void add(T item);
      //Adds a new element to the set.

      void output( );
      //Outputs the set to the console.

      Set<T>* setUnion(const Set<T>& otherSet);
      //Union calling object's set with otherSet
      //and return a pointer to the new set.

      Set<T>* setIntersection(const Set<T>& otherSet);
      //Intersect calling object's set with otherSet
      //and return a pointer to the new set.
   private:
       Node<T> *head;
  };   //Set
}    //SetSavitch
#endif //SET_H
```

The library "listtools.h" is the linked list library interface from Display 17.14.

Display 17.29 **Implementation File for a Set Template Class** (part 1 of 2)

```
//This is the implementation file set.cpp.
//This is the implementation of the class Set.

#include <iostream>
#include "listtools.h"
#include "set.h"
using std::cout;
using std::endl;
using LinkedListSavitch::Node;
using LinkedListSavitch::search;
using LinkedListSavitch::headInsert;

namespace SetSavitch
{

    template<class T>
    Set<T>::~Set( )
    {
        Node<T> *toDelete = head;
        while (head != NULL)
        {
          head = head->getLink( );
          delete toDelete;
          toDelete = head;
        }
    }

    template<class T>
    bool Set<T>::contains(T target) const
    {
        Node<T>* result = search(head, target);
        if (result == NULL)
           return false;
        else
           return true;
    }

    void Set<T>::output( )
    {
        Node<T> *iterator = head;
        while (iterator != NULL)
        {
          cout << iterator->getData( ) << " ";
          iterator = iterator->getLink( );
        }
```

(continued)

Display 17.29 **Implementation File for a Set Template Class** (part 2 of 2)

```
       cout << endl;
   }

   template<class T>
   void Set<T>::add(T item)
   {
       if (search(head, item)==NULL)
       {
         //Only add the target if it's not in the list
         headInsert(head, item);
       }
   }

   template<class T>
   Set<T>* Set<T>::setUnion(const Set<T>& otherSet)
   {
       Set<T> *unionSet = new Set<T>( );
       Node<T>* iterator = head;
       while (iterator != NULL)
       {
           unionSet->add(iterator->getData( ));
           iterator = iterator->getLink( );
       }
       iterator = otherSet.head;
       while (iterator != NULL)
       {
           unionSet->add(iterator->getData( ));
           iterator = iterator->getLink( );
       }
       return unionSet;
   }

   template<class T>
   Set<T>* Set<T>::setIntersection(const Set<T>& otherSet)
   {
       Set<T> *interSet = new Set<T>( );
       Node<T>* iterator = head;
       while (iterator != NULL)
       {
          if (otherSet.contains(iterator->getData( )))
          {
           interSet->add(iterator->getData( ));
          }
          iterator = iterator->getLink( );
       }
       return interSet;
   }
}  //SetSavitch
```

Display 17.30 **Program Using the Set Template Class** (part 1 of 2)

```
//Program to demonstrate use of the Set class

#include <iostream>
#include <string>
#include "set.h"
#include "listtools.cpp"
#include "set.cpp"
using std::cout;
using std::endl;
using std::string;
using namespace SetSavitch;

int main( )
{
      Set<string> round; //Round things
      Set<string> green; //Green things

      round.add("peas"); //Sample data for both sets
      round.add("ball");
      round.add("pie");
      round.add("grapes");
      green.add("peas");
      green.add("grapes");
      green.add("garden hose");
      green.add("grass");

      cout << "Contents of set round: ";
      round.output( );
      cout << "Contents of set green: ";
      green.output( );

      cout << "ball in set round? " <<
              round.contains("ball") << endl;
      cout << "ball in set green? " <<
              green.contains("ball") << endl;

      cout << "ball and peas in same set? ";
      if ((round.contains("ball") && round.contains("peas")) ||
          (green.contains("ball") && green.contains("peas")))
          cout << " true" << endl;
      else
          cout << " false" << endl;

      cout << "pie and grass in same set? ";
      if ((round.contains("pie") && round.contains("grass")) ||
          (green.contains("pie") && green.contains("grass")))
          cout << " true" << endl;
```

(continued)

Display 17.30 **Program Using the Set Template Class** (part 2 of 2)

```
        else
            cout << "  false" << endl;

        cout << "Union of green and round: " << endl;
        Set<string> *unionset = round.setUnion(green);
        unionset->output( );
        delete unionset;

        cout << "Intersection of green and round: " << endl;
        Set<string> *interset = round.setIntersection(green);
        interset->output( );
        delete interset;

        return 0;
    }
```

Sample Dialogue

```
Contents of set round: grapes pie ball peas
Contents of set green: grass garden hose grapes peas
ball in set round? 1
ball in set green? 0
ball and peas in same set? true
pie and grass in same set? false
Union of green and round:
garden hose grass peas ball pie grapes
Intersection of green and round:
peas grapes
```

Some compilers may output `true` *and* `false` *instead of* `1` *and* `0`.

Efficiency of Sets Using Linked Lists

set

We can analyze the run-time efficiency of our set data structure in terms of the fundamental set operations. Adding an item to the set always inserts a new node on the front of the list. This requires setting only one link on the linked list. The `contains` function iterates through the entire set looking for the target, which may require examining every node in the list. When we invoke the `setUnion` function for sets A and B, it iterates through both sets and adds each item into a new set. If there are n items in set A and m items in set B, then this requires examining $n + m$ items. However, there is a hidden cost because the `add` function searches through its entire list for any duplicates before a new item is added. This cost becomes significant as the number of items added to the new set increases. Finally, the `setIntersection` function applied to sets A and B invokes the `contains` function of set B for each item in set A. Since the `contains` function requires examining up to m nodes for each item in set A, then `setIntersection` requires examining at most $m \times n$ nodes. These are inefficient functions in our implementation of sets. A different approach to represent the set—for example, one that used hash tables instead of a linked list—could result in a `setIntersection` function that examines at most $n + m$ nodes. Nevertheless, our linked list implementation would probably be fine for an application that uses small sets or for an application that does not frequently invoke the `setIntersection` function, and we have the benefit of relatively simple code that is easy to understand.

If we really needed the efficiency, we could maintain the same interface to the `Set<T>` class but replace our linked list implementation with something else. If we used the hash table implementation from Display 17.25, the `contains` function would run much more quickly. However, switching to a hash table makes it more difficult to iterate through the set of items. Instead of traversing a single linked list to retrieve every item in the set, the hash table version must now iterate through the hash table array and then, for each index in the array, iterate through the linked list at that index. Examination of each entry in the hash table array takes extra time that was not necessary in the singly linked list implementation of a set. So while we have decreased the number of steps it takes to look up an item, we have increased the number of steps it takes to iterate over every item. If this were troublesome, you could overcome this problem with an implementation of `Set<T>` that used both a linked list (to facilitate iteration) and a hash table (for fast lookup). However, the complexity of the code is significantly increased using such an approach. You are asked to explore the hash table implementation in Programming Project 17.10.

MyProgrammingLab™

Self-Test Exercises

23. Write a function named `difference` for the `Set` class that returns the difference between two sets. The function should return a pointer to a new set that has items from the first set that are not in the second set.
For example, if `setA` contains {1, 2, 3, 4} and `setB` contains {2, 4, 5}, then `setA.difference(setB)` should return the set {1, 3}.

17.3 Iterators

The white rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.
"Begin at the beginning," the King said, very gravely, "And go on till you come to the end: then stop."

LEWIS CARROLL, *Alice in Wonderland*

iterator

An important notion in data structures is that of an iterator. An **iterator** is a construct (typically an object of some iterator class) that allows you to cycle through the data items stored in a data structure so that you can perform whatever action you want on each data item.

Iterator

An *iterator* is a construct (typically an object of some iterator class) that allows you to cycle through the data items stored in a data structure so that you can perform whatever action you want on each data item in the data structure.

Pointers as Iterators

The basic idea, and in fact the prototypical model, for iterators can easily be seen in the context of linked lists. A linked list is one of the prototypical data structures, and a pointer is a prototypical example of an iterator. You can use a pointer as an iterator by moving through the linked list one node at a time starting at the head of the list and cycling through all the nodes in the list. The general outline is as follows:

ok as set

```
Node_Type *iterator;
for (iterator = Head; iterator != NULL;
         iterator = iterator->Link)
      Do whatever you want with the node pointed to by iterator;
```

where *Head* is a pointer to the head node of the linked list and *Link* is the name of the member variable of a node that points to the next node in the list.

For example, to output the data in all the nodes in a linked list of the kind we discussed in Section 17.1, you could use the following:

```
IntNode *iterator;
for(iterator = head; iterator != NULL;
                     iterator = iterator->getLink( ))
    cout << (iterator->getData( ));
```

The definition of `IntNode` is given in Display 17.4.

Note that you test to see if two pointers are pointing to the same node by comparing them with the equal operator, `==`. A pointer is a memory address. If two pointer variables contain the same memory address, then they compare as equal and point to the same node. Similarly, you can use `!=` to compare two pointers to see if they do not point to the same node.

Iterator Classes

iterator class

An **iterator class** is a more versatile and more general notion than a pointer. It very often does have a pointer member variable as the heart of its data, as in the next programming example, but that is not required. For example, the heart of the iterator might be an array index. An iterator class has functions and overloaded operators that allow you to use pointer syntax with objects of the iterator class no matter what you use for the underlying data structure, node type, or basic location marker (pointer or array index or whatever). Moreover, it provides a general framework that can be used across a wide range of data structures.

An iterator class typically has the following overloaded operators:

`++` Overloaded increment operator, which advances the iterator to the next item.

`--` Overloaded decrement operator, which moves the iterator to the previous item.

`==` Overloaded equality operator to compare two iterators and return `true` if they both point to the same item.

`!=` Overloaded not-equal operator to compare two iterators and return `true` if they do not point to the same item.

`*` Overloaded dereferencing operator that gives access to one item. (Often it returns a reference to allow both read and write access.)

When thinking of this list of operators, you can use a linked list as a concrete example. In that case, remember that the items in the list are the data in the list, not the entire nodes and not the pointer members of the nodes. Everything but the data items is implementation detail that is meant to be hidden from the programmer who uses the iterator and data structure classes.

An iterator is used in conjunction with some particular structure class that stores data items of some type. The data structure class normally has the following member functions that provide iterators for objects of that class:

`begin( )`: A member function that takes no argument and returns an iterator that is located at ("points to") the first item in the data structure.

`end( )`: A member function that takes no argument and returns an iterator that can be used to test for having cycled through all items in the data structure. If `i` is an iterator and it has been advanced *beyond* the last item in the data structure, then `i` should equal `end( )`.

Using an iterator, you can cycle through the items in a data structure `ds` as follows:

```
for (i = ds.begin( ); i != ds.end( ); i++)
     process *i //*i is the current data item.
```

Iterator Class

An iterator class typically has the following overloaded operators: `++`, move to next item; `-`, move to previous item; `==`, overloaded equality; `!=`, overloaded not-equal operator; and `*`, overloaded dereferencing operator that gives access to one data item.

The data structure corresponding to an iterator class typically has the following two member functions: `begin( )`, which returns an iterator that is located at ("points to") the first item in the data structure; and `end( )`, which returns an iterator that can be used to test for having cycled through all items in the data structure. If `i` is an iterator and it has been advanced *beyond* the last item in the data structure, then `i` should equal `end( )`.

Using an iterator, you can cycle through the items in a data structure `ds` as follows:

```
for (i = ds.begin( ); i != ds.end( ); i++)
    process *i //*i is the current data item.
```

where i is an iterator. Chapter 19 discusses iterators with a few more items and refinements than these, but these will do for an introduction.

This abstract discussion will not come alive until we give an example. So, let us walk through one.

EXAMPLE: An Iterator Class

Display 17.31 contains the definition of an iterator class that can be used for data structures (such as a stack or queue) that are based on a linked list. We have placed the node class and the iterator class into a namespace of their own. This makes sense, since the iterator is intimately related to the node class and since any class that uses this node class can also use the iterator class. This iterator class does not have a decrement operator, because a definition of a decrement operator depends on the details of the linked list and does not depend solely on the type Node<T>. (There is nothing wrong with having the definition of the iterator depend on the underlying linked list. We have just decided to avoid this complication.)

As you can see, the template class ListIterator is essentially a pointer wrapped in a class so that it can have the needed member operators. The definitions of the overload operators are straightforward and in fact so short that we have defined all of them as inline functions. Note that the dereferencing operator, *, produces the data member variable of the node pointed to. Only the data member variable is data. The pointer member variable in a node is part of the implementation detail that the user programmer should not need to be concerned with.

You can use the ListIterator class as an iterator for any class based on a linked list that uses the template class Node. As an example, we have rewritten the template class Queue so that it has iterator facilities. The interface for the template class Queue is given in Display 17.32. This definition of the Queue template is the same as our previous version (Display 17.20) except that we have added a type definition as well as the following two member functions:

```
Iterator begin( ) const { return Iterator(front); }
Iterator end( ) const { return Iterator( ); }
//The end iterator has end( ).current == NULL.
```

Let us discuss the member functions first.

The member function begin( ) returns an iterator located at ("pointing to") the front node of the queue, which is the head node of the underlying linked list. Each application of the increment operator, ++, moves the iterator to the next node. Thus, you can move through the nodes, and hence the data, in a queue named q as follows:

```
 for (i = q.begin( ); Stopping_Condition; i++)
     process *i    //*i is the current data item.
```

where i is a variable of the iterator type.

(continued)

Display 17.31 **An Iterator Class for Linked Lists** (part 1 of 2)

```
//This is the header file iterator.h. This is the interface for the
//class ListIterator, which is a template class for an iterator to use
//with linked lists of items of type T. This file also contains the
//node type for a linked list
#ifndef ITERATOR_H
#define ITERATOR_H

namespace ListNodeSavitch
{
    template<class T>
    class Node
    {
    public:
        Node(T theData, Node<T>* theLink) : data(theData),
                 link(theLink){}
        Node<T>* getLink( ) const { return link; }
        const T& getData( ) const { return data; }
        void setData(const T& theData) { data = theData; }
        void setLink(Node<T>* pointer) { link = pointer; }
    private:
        T data;
        Node<T> *link;
    };

    template<class T>
    class ListIterator
    {
    public:
        ListIterator( ) : current(NULL) {}

        ListIterator(Node<T>* initial) : current(initial) {}
        const T& operator *( ) const { return current->getData( ); }
        //Precondition: Not equal to the default constructor object;
        //that is, current != NULL.
        ListIterator& operator ++( ) //Prefix form
        {
            current = current->getLink( );
            return *this;
        }
        ListIterator operator ++(int) //Postfix form
        {
            ListIterator startVersion(current);
            current = current->getLink( );
            return startVersion;
```

*Note that the dereferencing operator * produces the data member of the node, not the entire node. This version does not allow you to change the data in the node.*

(continued)

Display 17.31 **An Iterator Class for Linked Lists** (part 2 of 2)

```
            }
            bool operator ==(const ListIterator& rightSide) const
            { return (current == rightSide.current); }

            bool operator !=(const ListIterator& rightSide) const
            { return (current != rightSide.current); }

            //The default assignment operator and copy constructor
            //should work correctly for ListIterator.
        private:
            Node<T> *current;
        };

    } //ListNodeSavitch

    #endif  //ITERATOR_H
```

EXAMPLE: (continued)

The member function `end( )` returns an iterator whose current member variable is `NULL`. Thus, when the iterator `i` has passed the last node, the Boolean expression

```
i != q.end( )
```

changes from `true` to `false`. This is the desired *Stopping_Condition*. This queue class and iterator class allow you to cycle through the data in the queue in the way we outlined for an iterator:

```
for (i = q.begin( ); i != q.end( ); i++)
    process *i //*i is the current data item.
```

Note that `i` is not equal to `q.end( )` when `i` is at the last node. The iterator `i` is not equal to `q.end( )` until `i` has been advanced one position past the last node. To remember this detail, think of `q.end( )` as being an end marker like `NULL`; in this case, it is essentially a version of `NULL`. A sample program that uses such a `for` loop is shown in Display 17.33.

Notice the type definition in our new queue template class:

```
typedef ListIterator<T> Iterator;
```

(continued on page 799)

Display 17.32 Interface File for a Queue with Iterators Template Class

```
//This is the header file queue.h. This is the interface for the class
//Queue, which is a template class for a queue of items of type T,
//including iterators.
#ifndef QUEUE_H
#define QUEUE_H
#include "iterator.h"
using namespace ListNodeSavitch;

namespace QueueSavitch
{
    template<class T>
    class Queue
    {
    public:
        typedef ListIterator<T> Iterator;

        Queue( );
        Queue(const Queue<T>& aQueue);
        Queue<T>& operator =(const Queue<T>& rightSide);
        virtual ~Queue( );
        void add(T item);
        T remove( );
        bool isEmpty( ) const;

        Iterator begin( ) const { return Iterator(front);}
        Iterator end( ) const { return Iterator( ); }
        //The end iterator has end( ).current == NULL.
        //Note that you cannot dereference the end iterator.
    private:
        Node<T> *front;//Points to the head of a linked list.
                       //Items are removed at the head.
        Node<T> *back; //Points to the node at the other end of
                       //the linked list.
                       //Items are added at this end.
    };

}  //QueueSavitch

#endif //QUEUE_H
```

The definitions of `Node<T>` *and* `ListIterator<T>` *are in the namespace* `List NodeSavitch` *in the file* `iterator.h.`

Display 17.33 Program Using the Queue Template Class with Iterators

```
//Program to demonstrate use of the Queue template class with iterators.
#include <iostream>
#include "queue.h" //not needed
#include "queue.cpp"
#include "iterator.h" //not needed
using std::cin;
using std::cout;
using std::endl;
using namespace QueueSavitch;
int main( )
{
    char next, ans;
    do
    {
        Queue<char> q;
        cout << "Enter a line of text:\n";
        cin.get(next);
        while (next != '\n')
        {
            q.add(next);
            cin.get(next);
        }

        cout << "You entered:\n";
        Queue<char>::Iterator i;

        for (i = q.begin( ); i != q.end( ); i++)
            cout << *i;
        cout << endl;

        cout << "Again?(y/n): ";
        cin >> ans;
        cin.ignore(10000, '\n');
    }while (ans != 'n' && ans != 'N');

    return 0;
}
```

Even though they are not needed, many programmers prefer to include these `include` directives for the sake of documentation.

If your compiler is unhappy with `Queue<char>::Iterator i;` try using namespace `ListNodeSavitch;` `ListIterator<char> i;`

Sample Dialogue

```
Enter a line of text:
Where shall I begin?
You entered:
Where shall I begin?
Again?(y/n): y
Enter a line of text:
Begin at the beginning
You entered:
Begin at the beginning
Again?(y/n): n
```

EXAMPLE: (continued)

This `typedef` is not absolutely necessary. You can always use `ListIterator<T>` instead of the type name `Iterator`. However, this type definition does make for cleaner code. With this type definition, an iterator for the class `Queue<char>` is written

```
Queue<char>::Iterator i;
```

This makes it clear with which class the iterator is meant to be used.

The implementation of our new template class `Queue` is given in Display 17.34. Since the only member functions we added to this new `Queue` class are defined inline, the implementation file contains nothing really new, but we include the implementation file to show how it is laid out and to show which directives it would include.

Display 17.34 **Implementation File for a Queue with Iterators Template Class** (part 1 of 2)

```
//This is the file queue.cpp. This is the implementation of the
//template class Queue. The interface for the template class Queue is
//in the header file queue.h.
#include <iostream>
#include <cstdlib>
#include <cstddef>                The member function definitions are the same as in
#include "queue.h"                the previous version of the Queue template. This is
using std::cout;                  given to show the file layout and use of namespaces.

using namespace ListNodeSavitch;
namespace QueueSavitch
{
    template<class T>
    Queue<T>::Queue( ) : front(NULL), back(NULL)
<The rest of the definition is given in the answer to Self-Test Exercise 16.>

    template<class T>
    Queue<T>::Queue(const Queue<T>& aQueue)
<The rest of the definition is given in the answer to Self-Test Exercise 19.>

    template<class T>
    Queue<T>& Queue<T>::operator =(const Queue<T>& rightSide)
<The rest of the definition is given in the answer to Self-Test Exercise 20.>
    template<class T>
    Queue<T>::~Queue( )
```

(continued)

Display 17.34 **Implementation File for a Queue with Iterators Template Class** (part 2 of 2)

```
<The rest of the definition is given in the answer to Self-Test Exercise 18.>
    template<class T>
    bool Queue<T>::isEmpty( ) const
<The rest of the definition is given in the answer to Self-Test Exercise 16.>

    template<class T>
    void Queue<T>::add(T item)
<The rest of the definition is given in the answer to Self-Test Exercise 17.>

    template<class T>
    T Queue<T>::remove( )
<The rest of the definition is given in the answer to Self-Test Exercise 17.>
}//QueueSavitch
#endif//QUEUE_H
```

MyProgrammingLab™

Self-Test Exercises

24. Write the definition of the template function `inQ` shown here. Use iterators. Use the definition of `Queue` given in Display 17.32.

```
template<class T>
bool inQ(Queue<T> q, T target);
//Returns true if target is in the queue q;
//otherwise, returns false.
```

17.4 Trees

I think that I shall never see a data structure as useful as a tree.

Anonymous

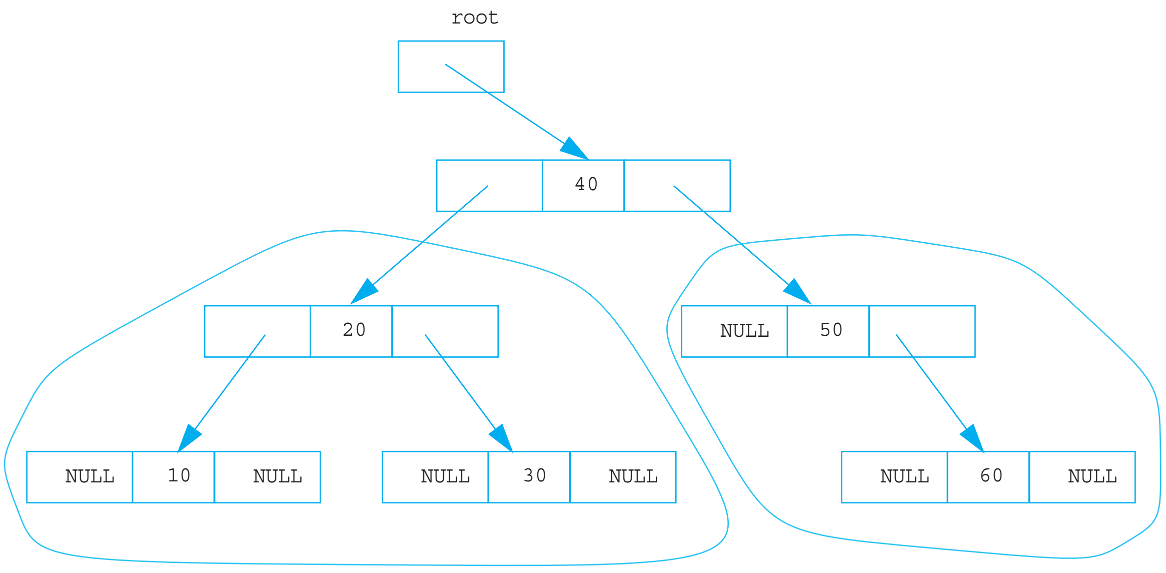
A detailed treatment of trees is beyond the scope of this chapter. The goal of this chapter is to teach you the basic techniques for constructing and manipulating data structures based on nodes and pointers. The linked list served as a good example for our discussion. However, there is one detail about the nodes in a singly linked list that is quite restricted: They have only one pointer member variable to point to another node. A tree node has two (and in some applications more than two) member variables for pointers to other nodes. Moreover, trees are a very important and widely used data structure. So, we will briefly outline the general techniques used to construct and manipulate trees.

This section uses recursion, which is covered in Chapter 13.

Tree Properties

A tree is a data structure that is structured as shown in Display 17.35. Note that a tree must have the sort of structure illustrated in Display 17.35. In particular, in a tree you can reach any node from the top (root) node by some path that follows the links (pointers). Note that there are no cycles in a tree. If you follow the pointers, you eventually get to an end. A definition for a node class for this sort of tree of `ints` is also shown in Display 17.35. Note that each node has two links (two pointers) coming from it. This sort of tree is called a binary tree because it has exactly two link member

Display 17.35 **A Binary Tree**



```
class IntTreeNode
{
public:
    IntTreeNode(int theData, IntTreeNode* left, IntTreeNode* right)
        : data(theData), leftLink(left), rightLink(right){}
private:
    int data;
    IntTreeNode *leftLink;
    IntTreeNode *rightLink;
};


IntTreeNode *root;
```

variables. There are other kinds of trees with different numbers of link member variables, but the **binary tree** is the most common case.

binary tree

The pointer named `root` serves a purpose similar to that of the pointer `head` in a linked list (Display 17.1). The node pointed to by the `root` pointer is called the **root node**. Note that the pointer `root` is not itself the root node, but rather points to the root node. Any node in the tree can be reached from the root node by following the links.

root node

The term *tree* may seem like a misnomer. The root is at the top of the tree and the branching structure looks more like a root branching structure than a true tree branching structure. The secret to the terminology is to turn the picture (Display 17.35) upside down. The picture then does resemble the branching structure of a tree and the root node is where the trees root would begin. The nodes at the ends of the branches with both link member variables set to `NULL` are known as **leaf nodes**, a terminology that may now make some sense.

leaf node

By analogy to an empty linked list, an empty tree is denoted by setting the pointer variable `root` equal to `NULL`.

empty tree

Note that a tree has a recursive structure. Each tree has two subtrees whose root nodes are the nodes pointed to by the `leftLink` and `rightLink` of the root node. These two subtrees are circled in Display 17.35. This natural recursive structure make trees particularly amenable to recursive algorithms. For example, consider the task of searching the tree in such a way that you visit each node and do something with the data in the node (such as writing it to the screen). The general plan of attack is as follows:

preorder

Preorder Processing

1. Process the data in the root node.
2. Process the left subtree.
3. Process the right subtree.

You can obtain a number of variants on this search process by varying the order of these three steps. Two more versions are given next.

in order

In-order Processing

1. Process the left subtree.
2. Process the data in the root node.
3. Process the right subtree.

postorder

Postorder Processing

1. Process the left subtree.
2. Process the right subtree.
3. Process the data in the root node.

The tree in Display 17.35 has stored each number in the tree in a special way known as the **Binary Search Tree Storage Rule**. The rule is given in the accompanying box. A tree that satisfies the Binary Search Tree Storage Rule is referred to as a **binary search tree**.

binary search tree

Binary Search Tree Storage Rule

Binary Search Tree Storage Rule

1. All the values in the left subtree are less than the value in the root node.
2. All the values in the right subtree are greater than or equal to the value in the root node.
3. This rule applies recursively to each of the two subtrees.

(The base case for the recursion is an empty tree, which is always considered to satisfy the rule.)

Note that if a tree satisfies the Binary Search Tree Storage Rule and you output the values using the in-order processing method, the numbers will be output in order from smallest to largest.

For trees that follow the Binary Search Tree Storage Rule that are short and fat rather than long and thin, values can be very quickly retrieved from the tree using a binary search algorithm that is similar in spirit to the binary search algorithm presented in Display 13.5. The topic of searching and maintaining a binary storage tree to realize this efficiency is a large topic that goes beyond what we have room for here.

EXAMPLE: A Tree Template Class

Display 17.36 contains the definition of a template class for a binary search tree. In this example, we have made the `SearchTree` class a friend class of the `TreeNode` class. This allows us to access the node member variables by name in the definitions of the tree class member variables. The implementation of this `SearchTree` class is given in Display 17.37, and a demonstration program is given in Display 17.38.

This template class is designed to give you the flavor of tree processing, but it is not really a complete example. A real class would have more member functions. In particular, a real tree class would have a copy constructor and an overloaded assignment operator. We have omitted these to conserve space.

There are some things to observe about the function definitions in the class `SearchTree`. The functions `insert` and `inTree` are overloaded. The single-argument versions are the ones we need. However, the clearest algorithms are recursive, and the recursive algorithms require one additional parameter for the root of a subtree. Therefore, we defined private helping functions with two arguments for each of these functions and implemented the recursive algorithms in the two-parameter function. The single-parameter function then simply makes a call to the two-parameter version with the subtree root parameter set equal to the root of the entire tree. A similar situation holds for the overloaded member function name `inorderShow`. The function `deleteSubtree` serves a similar purpose for the destructor function.

(continued on page 808)

Display 17.36 Interface File for a Tree Template Class

```
//Header file tree.h. The only way to insert data in a tree is with the
//insert function. So, the tree satisfies the Binary Search Tree Storage
//Rule. The function inTree depends on this. < must be defined and
//give a well-behaved ordering to the type T.
#ifndef TREE_H
#define TREE_H
namespace TreeSavitch
{
    template<class T>
    class SearchTree;//forward declaration

    template<class T>
    class TreeNode
    {
    public:
        TreeNode( ) : root(NULL){}
        TreeNode(T theData, TreeNode<T>* left, TreeNode<T>* right)
            : data(theData), leftLink(left), rightLink(right){}
        friend class SearchTree<T>;
    private:
        T data;
        TreeNode<T> *leftLink;
        TreeNode<T> *rightLink;
    };

    template<class T>
    class SearchTree
    {
    public:
        SearchTree( ) : root(NULL){}
        virtual ~SearchTree( );
        void insert(T item);//Adds item to the tree.
        bool inTree(T item) const;
        void inorderShow( ) const;
    private:
        void insert(T item, TreeNode<T>*& subTreeRoot);
        bool inTree(T item, TreeNode<T>* subTreeRoot) const;
        void deleteSubtree(TreeNode<T>*& subTreeRoot);
        void inorderShow(TreeNode<T>* subTreeRoot) const;
        TreeNode<T> *root;
    };

}//TreeSavitch

#endif
```

The `SearchTree` template class should have a copy constructor, an overloading of the assignment operator, and other member functions. However, we have omitted these functions to keep this example short. A real template class would contain more member functions and overloaded operators.

Display 17.37 **Implementation File for a Tree Template Class** (part 1 of 2)

```
//This is the implementation file tree.cpp. This is the implementation
//for the template class SearchTree. The interface is in the file tree.h.
namespace TreeSavitch
{
    template<class T>
    void SearchTree<T>::insert(T item, TreeNode<T>*& subTreeRoot)
    {
        if (subTreeRoot == NULL)
            subTreeRoot = new TreeNode<T>(item, NULL, NULL);
        else if (item < subTreeRoot->data)
            insert(item, subTreeRoot->leftLink);
        else//item >= subTreeRoot->data
            insert(item, subTreeRoot->rightLink);
    }

    template<class T>
    void SearchTree<T>::insert(T item)
    {
        insert(item, root);
    }

    template<class T>
    bool SearchTree<T>::inTree(T item, TreeNode<T>* subTreeRoot) const
    {
        if (subTreeRoot == NULL)
            return false;
        else if (subTreeRoot->data == item)
            return true;
        else if (item < subTreeRoot->data)
            return inTree(item, subTreeRoot->leftLink);
        else//item >= link->data
            return inTree(item, subTreeRoot->rightLink);
    }

    template<class T>
    bool SearchTree<T>::inTree(T item) const
    {
        return inTree(item, root);
```

If all data is entered using the function `insert`, the tree will satisfy the Binary Search Tree Storage Rule.

The function `in Tree` uses a binary search algorithm that is a variant of the one given in Display 13.5.

(continued)

Display 17.37 **Implementation File for a Tree Template Class** (part 2 of 2)

```
    }
    template<class T>//uses iostream:
    void SearchTree<T>::inorderShow(TreeNode<T>* subTreeRoot) const
    {
        if (subTreeRoot != NULL)
        {
            inorderShow(subTreeRoot->leftLink);
            cout << subTreeRoot->data << " ";
            inorderShow(subTreeRoot->rightLink);
        }
    }

    template<class T>//uses iostream:
    void SearchTree<T>::inorderShow( ) const
    {
        inorderShow(root);
    }

    template<class T>
    void SearchTree<T>::deleteSubtree(TreeNode<T>*& subTreeRoot)
    {
        if (subTreeRoot != NULL)
        {
            deleteSubtree(subTreeRoot->leftLink);

            deleteSubtree(subTreeRoot->rightLink);

            //subTreeRoot now points to a one node tree.
            delete subTreeRoot;
            subTreeRoot = NULL;
        }
    }

    template<class T>
    SearchTree<T>::~SearchTree( )
    {
        deleteSubtree(root);
    }
}//TreeSavitch
```

Uses in-order traversal of the tree.

Uses postorder traversal of the tree.

Display 17.38 Demonstration Program for the Tree Template Class

```
//Demonstration program for the Tree template class.
#include <iostream>
#include "tree.h"
#include "tree.cpp"
using std::cout;
using std::cin;
using std::endl;
using TreeSavitch::SearchTree;

int main( )
{
    SearchTree<int> t;

    cout << "Enter a list of nonnegative integers.\n"
         << "Place a negative integer at the end.\n";
    int next;
    cin >> next;
    while (next >= 0)
    {
        t.insert(next);
        cin >> next;
    }

    cout << "In sorted order: \n";
    t.inorderShow( );
    cout << endl;

    return 0;
}
```

Sample Dialogue

```
Enter a list of nonnegative integers.
Place a negative integer at the end.
40  30  20  10  11  22  33  44  -1
In sorted order:
10  11  20  22  30  33  40  44
```

EXAMPLE: (continued)

Finally, it is important to note that the `insert` function builds a tree that satisfies the Binary Search Tree Storage Rule. Since `insert` is the only function available to build trees for this template class, objects of this tree template class will always satisfy the Binary Search Tree Storage Rule. The function `inTree` uses the fact that the tree satisfies the Binary Search Tree Storage Rule in its algorithms. This makes searching the tree very efficient. Of course this means that the < operator must be defined for the type `T` of data stored in the tree. To make things work correctly, the operation < should satisfy the following rules when applied to values of type `T`:

- *Transitivity*: $a < b$ and $b < c$ implies $a < c$.
- *Antisymmetry*: If a and b are not equal, then either $a < b$ or $b < a$, but not both.
- *Irreflexive*: You never have $a < a$.

Most natural orders satisfy these rules.[2]

MyProgrammingLab™

Self-Test Exercises

25. Define the following member functions, which could be added to the class `SearchTree` in Display 17.36. These functions display the data encountered in a pre- and postorder traversal of the tree, respectively. Define a private helping function for each function, as we did for `SearchTree<T>::inorderShow`.

```
void SearchTree<T>::preorderShow( ) const
void SearchTree<T>::postorderShow( )const
```

Chapter Summary

- A *node* is a `struct` or class object that has one or more member variables that are pointer variables. These nodes can be connected by their member pointer variables to produce data structures that can grow and shrink in size while your program is running.
- A *linked list* is a list of nodes in which each node contains a pointer to the next node in the list.
- The end of a linked list (or other linked data structure) is indicated by setting the pointer member variable equal to `NULL`.

[2]Note that you normally have both a "less-than-or-equal" relation and a "less-than" relation. These rules apply only to the "less-than" relation. You can actually make do with an even weaker notion of ordering known as a *strict weak ordering*, which is defined in Chapter 19, but that is more detail than you need for normally encountered orderings.

- Nodes in a *doubly linked list* have two links—one to the previous node in the list and one to the next node. This makes operations such as insertion and deletion slightly easier.
- A *stack* is a first-in/last-out data structure. A *queue* is a first-in/first-out data structure. Both can be implemented using a linked list.
- A *hash table* is a data structure that is used to store objects and retrieve them efficiently. A *hash function* is used to map an object to a value that can then be used to index the object.
- Linked lists can be used to implement sets, including common operations such as `union`, `intersection`, and `set` membership.
- An *iterator* is a construct (typically an object of some iterator class) that allows you to cycle through data items stored in a data structure.
- A *tree* is a data structure whose nodes have two (or more) member variables for pointers to other nodes. If a tree satisfies the *Binary Search Tree Storage Rule*, then a function can be designed to rapidly find data in the tree.

Answers to Self-Test Exercises

1.
```
Sally
Sally
18
18
```
Note that `(*head).name` and `head->name` mean the same thing. Similarly, `(*head).number` and `head->number` mean the same thing.

2. The best answer is
```
head->next = NULL;
```
However, the following is also correct:
```
(*head).next = NULL;
```

3. `head->item = "Wilbur's brother Orville";`

4.
```
class NodeType
{
public:
    NodeType( ){}
    NodeType(char theData, NodeType* theLink)
            : data(theData), link(theLink){}
    NodeType* getLink( ) const { return link; }
```

```
        char getData( ) const { return data; }
        void setData(char theData) { data = theData; }
        void setLink(NodeType* pointer) { link = pointer; }
    private:
        char data;
        NodeType *link;
    };
    typedef NodeType* PointerType;
```

5. The value `NULL` is used to indicate an empty list.
6. `p1 = p1-> next;`
7.
```
Pointer discard;
discard = p2->next;//discard points to the node to be deleted.
p2->next = discard->next;
```
   This is sufficient to delete the node from the linked list. However, if you are not using this node for something else, you should destroy the node with a call to `delete` as follows:
```
delete discard;
```
8. `p1 = p1->getLink( );`
9.
```
Pointer discard;
discard = p2->getLink( );//points to node to be deleted.
p2->setLink(discard->getLink( ));
```
   This is sufficient to delete the node from the linked list. However, if you are not using this node for something else, you should destroy the node with a call to `delete` as follows:
```
delete discard;
```
10. a. Inserting a new item at a known location into a large linked list is more efficient than inserting into a large array. If you are inserting into a list, you have about five operations, most of which are pointer assignments, regardless of the list size. If you insert into an array, on the average you have to move about half the array entries to insert a data item.

    For small lists, the answer is c, about the same.
11.
```
void insert(DoublyLinkedIntNodePtr afterMe, int theData)
{
  DoublyLinkedIntNode* newNode = new
            DoublyLinkedIntNode(theData, afterMe,
            afterMe->getNext
        Link( ));
          afterMe->setNextLink(newNode);
          if (newNode->getNextLink( ) != NULL)
```
join (no space, it is getNextLink)

```
            {
                newNode->getNextLink( )->setPreviousLink(newNode);
            }
}
```

12. Insertion and deletion are slightly easier with the doubly linked list because we no longer need a separate variable to keep track of the previous node. Instead, we can access this node through the previous link. However, all operations require updating more links (e.g., both the next and previous instead of just the previous).
13. Note that this function is essentially the same as `headInsert` in Display 17.15.

```
template<class T>
void Stack<T>::push(T stackFrame)
{
    top = new Node<T>(stackFrame, top);
}
```

14.

```
//Uses cstddef:
template<class T>
Stack<T>::Stack(const Stack<T>& aStack)
{
    if (aStack.isEmpty( ))
        top = NULL;
    else
    {
        Node<T> *temp = aStack.top;//temp moves through
               //the nodes from top to bottom of aStack.
        Node<T> *end;//Points to end of the new stack.

        end = new Node<T>(temp->getData( ), NULL);
        top = end;
        //First node created and filled with data.
        //New nodes are now added AFTER this first node.

        temp = temp->getLink( );//move temp to second node
                      //or NULL if there is no second node.
        while (temp != NULL)
        {
            end->setLink(
                      new Node<T>(temp->getData( ), NULL));
            temp = temp->getLink( );
            end = end->getLink( );
        }
        //end->link == NULL;
    }
}
```

15.
```
template<class T>
Stack<T>& Stack<T>::operator =(const Stack<T>& rightSide)
{
    if (top == rightSide.top)//if two stacks are the same
        return *this;
    else   //send left side back to freestore
    {
        T next;
        while (! isEmpty( ))
            next = pop( );//remove calls delete.
    }
    if (rightSide.isEmpty( ))
    {
        top = NULL;
        return *this;
    }
    else
    {
        Node<T> *temp = rightSide.top;//temp moves through
             //the nodes from front top to bottom of rightSide.
        Node<T> *end;//Points to end of the left-side stack.
        end = new Node<T>(temp->getData( ), NULL);
        top = end;;
        //First node created and filled with data.
        //New nodes are now added AFTER this first node.
        temp = temp->getLink( );//Move temp to second node
                //or set to NULL if there is no second node.
        while (temp != NULL)
        {
            end->setLink(
                     new Node<T>(temp->getData( ),NULL));
            temp = temp->getLink( );
            end = end->getLink( );
        }
         //end->link == NULL;
         return *this;
    }
}
```

16. The following should be placed in the namespace `QueueSavitch`:

```
//Uses cstddef:
template<class T>
Queue<T>::Queue( ) : front(NULL), back(NULL)
```

```
{
    //Intentionally empty.
}
//Uses cstddef:
template<class T>
bool Queue<T>::isEmpty( ) const
{
    return (back == NULL);//front == NULL would also work
}
```

17. The following should be placed in the namespace `QueueSavitch`:

```
//Uses cstddef:
template<class T>
void Queue<T>::add(T item)
{
    if (isEmpty( ))
       front = back = new Node<T>(item, NULL);//Sets both
                     //front and back to point to the only node
    else
    {
       back->setLink(new Node<T>(item, NULL));
       back = back->getLink( );
    }
}
//Uses cstdlib and iostream:
template<class T>
T Queue<T>::remove( )
{
    if (isEmpty( ))
    {
        cout << "Error: Removing an item from an empty queue.\n";
        exit(1);
    }
    T result = front->getData( );
    Node<T> *discard;
    discard = front;
    front = front->getLink( );
    if (front == NULL)//if you removed the last node
        back = NULL;
    delete discard;
    return result;
}
```

18. The following should be placed in the namespace `QueueSavitch`:

```
template<class T>
Queue<T>::~Queue( )
{
    T next;
    while (! isEmpty( ))
       next = remove( );//remove calls delete.
}
```

19. The following should be placed in the namespace `QueueSavitch`:

```
//Uses cstddef:
template<class T>
Queue<T>::Queue(const Queue<T>& aQueue)
{
    if (aQueue.isEmpty( ))
        front = back = NULL;
    else
    {
        Node<T> *temp = aQueue.front;//temp moves
        //through the nodes from front to back of aQueue.

        back = new Node<T>(temp->getData( ), NULL);
        front = back;
        //First node created and filled with data.
        //New nodes are now added AFTER this first node.

        temp = temp->getLink( );//temp now points to second
                //node or NULL if there is no second node.

        while (temp != NULL)
        {
            back->setLink(new Node<T>(temp->getData( ), NULL));
            back = back->getLink( );
            temp = temp->getLink( );
        }
        //back->link == NULL
    }
}
```

20. The following should be placed in the namespace `QueueSavitch`:

```
//Uses cstddef:
template<class T>
Queue<T>& Queue<T>::operator =(const Queue<T>& rightSide)
```

```
{
    if (front == rightSide.front)//if the queues are the same
        return *this;
    else//send left side back to freestore
    {
        T next;
        while (! isEmpty( ))
            next = remove( );//remove calls delete.
    }
    if (rightSide.isEmpty( ))
    {
        front = back = NULL;
        return *this;
    }
    else
    {
        Node<T> *temp = rightSide.front;//temp moves
          //through the nodes from front to back of rightSide.
        back = new Node<T>(temp->getData( ), NULL);
        front = back;
        //First node created and filled with data.
        //New nodes are now added AFTER this first node.
        temp = temp->getLink( );//temp now points to second
                   //node or NULL if there is no second node.
        while (temp != NULL)
        {
            back->setLink(
                     new Node<T>(temp->getData( ), NULL));
            back = back->getLink( );
            temp = temp->getLink( );
        }
        //back->link == NULL;
        return *this;
    }
}
```

21. The simplest hash function is to map the ID number to the range of the hash table using the modulus operator:

```
hash = ID % N; //N is the hash table size.
```

22.
```
void HashTable::outputHashTable( )
{
    for (int i=0; i<SIZE; i++)
    {
      Node<string> *next = hashArray[i];
      cout << "In slot " << i << endl;
      cout << " ";
      while (next != NULL)
      {
        cout << next->getData( ) << " ";
        next = next->getLink( );
      }
    }
           cout << endl;
}
```

23. This code is similar to `intersection`, but adds elements if they are not in `otherSet`:

```
template<class T>
Set<T>* Set<T>::setDifference(const Set<T>& otherSet)
{
 Set<T> *diffSet = new Set<T>( );
 Node<T>* iterator = head;
 while (iterator != NULL)
 {
    if (!otherSet.contains(iterator->getData( )))
    {
     diffSet->add(iterator->getData( ));
    }
    iterator = iterator->getLink( );
 }
 return diffSet;
}
```

24.
```
using namespace ListNodeSavitch;
using namespace QueueSavitch;
template<class T>
bool inQ(Queue<T> q, T target)
{
    Queue<T>::Iterator i;
    i = q.begin( );
    while ((i != q.end( )) && (*i != target))
              i++;
    return (i != q.end( ));
}
```

Note that the following `return` statement does not work, since it can cause a dereferencing of `NULL`, which is illegal. The error would be a run-time error, not a compiler error.

```
return (*i == target);
```

25. The template class `SearchTree` needs function declarations added. These are just the definitions.

```
template<class T>//uses iostream:
void SearchTree<T>::preorderShow( ) const
{
    preorderShow(root);
}
template<class T>//uses iostream:
void SearchTree<T>::preorderShow(
                          TreeNode<T>* subTreeRoot) const
{
    if (subTreeRoot != NULL)
    {
        cout << subTreeRoot->data << " ";
        preorderShow(subTreeRoot->leftLink);
        preorderShow(subTreeRoot->rightLink);
    }
}
template<class T>//uses iostream:
void SearchTree<T>::postorderShow( ) const
{
    postorderShow(root);
}
template<class T>//uses iostream:
void SearchTree<T>::postorderShow(
                           TreeNode<T>* subTreeRoot) const
{
    if (subTreeRoot != NULL)
    {
        postorderShow(subTreeRoot->leftLink);
        postorderShow(subTreeRoot->rightLink);
        cout << subTreeRoot->data << " ";
    }
}
```

MyProgrammingLab™ **Programming Projects**

Visit www.myprogramminglab.com to complete select exercises online and get instant feedback.

1. Write a `void` function that takes a linked list of integers and reverses the order of its nodes. The function will have one call-by-reference parameter that is a pointer to the head of the list. After the function is called, this pointer will point to the head of a linked list that has the same nodes as the original list but in the reverse of the order they had in the original list. Note that your function will neither create nor destroy any nodes. It will simply rearrange nodes. Place your function in a suitable test program.
2. Write a function called `mergeLists` that takes two call-by-reference arguments that are pointer variables that point to the heads of linked lists of values of type `int`. The two linked lists are assumed to be sorted so that the number at the head is the smallest number, the number in the next node is the next smallest, and so forth. The function returns a pointer to the head of a new linked list that contains all the nodes in the original two lists. The nodes in this longer list are also sorted from smallest to largest values. Note that your function will neither create nor destroy any nodes. When the function call ends, the two pointer variable arguments should have the value `NULL`.
3. Design and implement a class that is a class for polynomials. The polynomial

   $$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$$

   will be implemented as a linked list. Each node will contain an `int` value for the power of x and an `int` value for the corresponding coefficient. The class operations should include addition, subtraction, multiplication, and evaluation of a polynomial. Overload the operators `+`, `-`, and `*` for addition, subtraction, and multiplication. Evaluation of a polynomial is implemented as a member function with one argument of type `int`. The evaluation member function returns the value obtained by plugging in its argument for x and performing the indicated operations.

   Include four constructors: a default constructor, a copy constructor, a constructor with a single argument of type `int` that produces the polynomial that has only one constant term that is equal to the constructor argument, and a constructor with two arguments of type `int` that produces the one-term polynomial whose coefficient and exponent are given by the two arguments. (In the previous notation, the polynomial produced by the one-argument constructor is of the simple form consisting of only a_0. The polynomial produced by the two-argument constructor is of the slightly more complicated form $a_n x^n$.) Include a suitable destructor. Include member functions to input and output polynomials.

   When the user inputs a polynomial, the user types in the following:

   a_n`x^`n `+` a_{n-1}`x^`n`-1+...+`a_0

However, if a coefficient a_i is 0, the user may omit the term a_i`x^ i`. For example, the polynomial

$3x^4 + 7x^2 + 5$

can be input as

```
3x^4 + 7x^2 + 5
```

It could also be input as

```
3x^4 + 0x^3 + 7x^2 + 0x^1 + 5
```

If a coefficient is negative, a minus sign is used in place of a plus sign, as in the following examples:

```
3x^5 - 7x^3 + 2x^1 - 8
```

```
-7x^4 + 5x^2 + 9
```

A minus sign at the front of the polynomial, as in the second of the previous two examples, applies only to the first coefficient; it does not negate the entire polynomial. Polynomials are output in the same format. In the case of output, the terms with 0 coefficients are not output. To simplify input, you can assume that polynomials are always entered one per line and that there will always be a constant term a_0. If there is no constant term, the user enters 0 for the constant term, as in the following:

```
12x^8 + 3x^2 + 0
```

4. a. The annotation in Display 17.36 says that a real `SearchTree` template class should have a copy constructor, an overloaded assignment operator, other overloaded operators, and other member functions. Obtain the code for Display 17.36 and add declarations for the following functions and overloaded operators: the default constructor, copy constructor, `delete`, overloaded operator, `=`, `makeEmpty`, `height`, `size`, `preOrderTraversal`, `inOrderTraversal`, and `postOrderTraversal`. The functions `preOrderTraversal`, `inOrder-Traversal`, and `postOrderTraversal` each call a global function `process` to process the nodes as they are encountered. The function `process` is a friend of the `SearchTree` class. For this exercise, it is only a stub.

   Supply preconditions and postconditions for these functions describing what each function should do.

   The function `height` has no parameters and returns the height of the tree. The height of the tree is the maximum of the heights of all the nodes. The height of a node is the number of links between it and the root node.

   The function `size` has no parameters and returns the number of nodes in the tree.

   The function `makeEmpty` removes all the nodes from the tree and returns the memory used by the nodes for reuse. The `makeEmpty` function leaves the root pointer with the value `NULL`.

b. Implement the member and friend functions and overloaded operators. Note that some of the functions listed here are already implemented in the text. You should make full use of the text's code. You should test your package thoroughly.

c. Design and implement an iterator class for the tree class. You will need to decide what a `begin` and `end` element means for your `searchTree`, and what will be the next element the `++` operator will point to.

Hint 1: You might maintain a private size variable that is increased by insertion and decreased by deletion, and whose value is returned by the `size` function. An alternative (use this if you know calls to size will be quite infrequent) is to calculate the size when you need it by traversing the tree. Similar techniques, though with more sophisticated details, can be used to implement the `height` function.

Hint 2: Do these a few members at a time. Compile and test after doing each group of a few members. You will be glad you did it this way.

Hint 3: Before you write the operator, `=`, and copy constructor, note that their jobs have a common task—duplicating another tree. Write a `copyTree` function that abstracts out the common task of the copy constructor and operator, `=`. Then write these two important functions using the common code.

Hint 4: The function `makeEmpty` and the destructor have a common tree destruction task.

VideoNote
Solution to Programming Project 17.5

5. In an ancient land, the beautiful princess Eve had many suitors. She decided on the following procedure to determine which suitor she would marry. First, all of the suitors would be lined up one after the other and assigned numbers. The first suitor would be number 1, the second number 2, and so on up to the last suitor, number *n*. Starting at the first suitor, she would then count three suitors down the line (because of the three letters in her name) and the third suitor would be eliminated from winning her hand and removed from the line. Eve would then continue, counting three more suitors, and eliminating every third suitor. When she reached the end of the line, she would continue counting from the beginning.

For example, if there were six suitors, then the elimination process would proceed as follows:

123456	Initial list of suitors, start counting from 1
12456	Suitor 3 eliminated, continue counting from 4
1245	Suitor 6 eliminated, continue counting from 1
125	Suitor 4 eliminated, continue counting from 5
15	Suitor 2 eliminated, continue counting from 5
1	Suitor 5 eliminated, 1 is the lucky winner

Write a program that creates a circular linked list of nodes to determine which position you should stand in to marry the princess if there are *n* suitors. Your program should simulate the elimination process by deleting the node that corresponds to the suitor that is eliminated for each step in the process. Be careful that you do not leave any memory leaks.

6. Modify the `Queue Template` class given in Section 17.2 so that it implements a **priority queue**. A priority queue is similar to a regular queue except that each item added to the queue also has an associated priority. For this problem, make the priority an integer where 0 is the highest priority and larger values are increasingly lower in priority.

   The `remove` function should return and remove the item that has the highest priority. For example,

   ```
   q.add('X', 10);
   q.add('Y', 1);
   q.add('Z', 3);
   
   cout << q.remove( ); //Returns Y
   cout << q.remove( ); //Returns Z
   cout << q.remove( ); //Returns X
   ```

   Test your queue on data with priorities in various orders (e.g., ascending, descending, mixed). You can implement the priority queue by performing a linear search in the `remove( )` function. In future courses, you may study a data structure called a *heap* that affords a more efficient way to implement a priority queue.

code

7. Add a remove function, a `cardinality` function, and an iterator for the `Set` class given in Displays 17.28 and 17.29. Write a suitable test program.

8. The hash table from Displays 17.24 and 17.25 hashed a string to an integer and stored the same string in the hash table. Modify the program so that instead of storing strings it stores `Employee` objects. Define the `Employee` class so that it contains private string member variables for the combined first and last name, job title, and phone number. Include functions to get and set these member variables. Use the employee name as the input to the hash function. The modification will require changes to the linked list, since the `LinkedList2` class created only linked lists of strings. For the most generality, modify the hash table so that it uses generic types. You will also need to add a `get` function to the `HashTable` class that returns the `Employee` object stored in the hash table that corresponds to the input name. The `Employee` class may require defining the == and != operators. Test your program by adding and retrieving several names, including names that hash to the same slot in the hash table.

9. Displays 17.24 through 17.26 provide the beginnings of a spell-checker. Refine the program to make it more useful. The modified program should read in a text file, parse each word, and determine whether each word is in the hash table and if not

output the line number and word of the potentially misspelled word. Discard any punctuation in the original text file. Use the `words.txt` file, which can be found on the website accompanying the textbook and on the book's Web site, as the basis for the hash table dictionary. The file contains 45,407 common words and names in the English language. Note that some words are capitalized. Test your spell-checker on a short text document.

10. Change the `Set<T>` class of Displays 17.28 and 17.29 so that internally it uses a hash table to store its data instead of a linked list. The headers of the public functions should remain the same so that a program such as the demonstration in Display 17.30 will work without any changes. Add a constructor that allows the user of the new `Set<T>` class to specify the size of the hash table array.

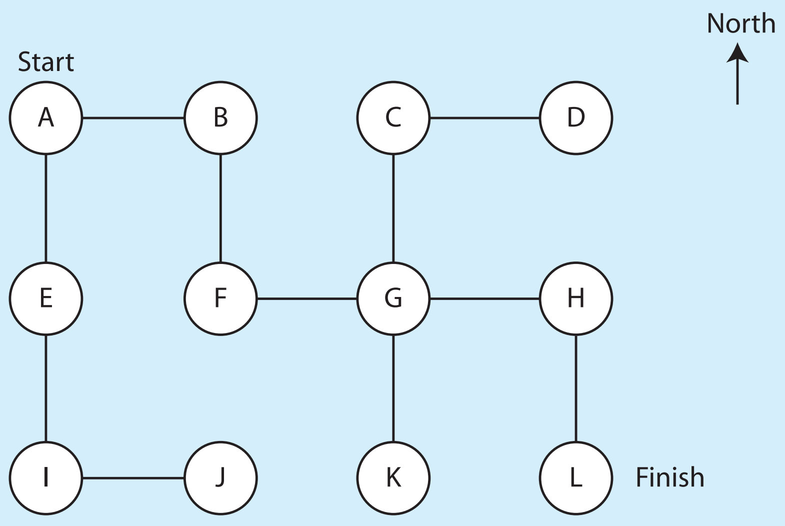
    The class for type `T` must override the `<<` operator. To convert the return value of `<<` to a string, do the following:

    ```
    # include <sstream>
    ...
    stringstream temp;
    temp << instance of Class;
    string s = temp.str();
    ```

    For an additional challenge, implement the set using both a hash table and a linked list. Items added to the set should be stored using both data structures. Any operation requiring lookup of an item should use the hash table, and any operation requiring iteration through the items should use the linked list.

VideoNote
Solution to Programming Project 17.11

11. The following figure is called a graph. The circles are called nodes and the lines are called edges. An edge connects two nodes. You can interpret the graph as a maze of rooms and passages. The nodes can be thought of as rooms and an edge connects one room to another. Note that each node has at most four edges in the following graph.



Write a program that implements the previous maze using references to instances of a Node class. Each node in the graph will correspond to an instance of Node. The edges correspond to links that connect one node to another and can be represented in Node as instance variables that reference another Node class. Start the user in node A. The user's goal is to reach the finish in node L. The program should output possible moves in the north, south, east, or west direction. Sample execution is shown next.

```
You are in room A of a maze of twisty little passages, all alike.
You can go east or south.
E
You are in room B of a maze of twisty little passages, all alike.
You can go west or south.
S
You are in room F of a maze of twisty little passages, all alike.
You can go north or east.
E
```

12. First, complete Programming Project 17.11. Then, write a recursive algorithm that finds a path from node A to node L. Your algorithm should work with any pair of start and finish nodes, not just nodes A and L. Your algorithm should also work if there are loops such as a connection between nodes E and F.