Pointers and Linked Lists 13

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. This chapter shows you how to define and manipulate linked lists, which will serve to introduce you to a new way of using pointers.

PREREQUISITES

This chapter uses material from Chapters 2 through 12.

13.1 NODES AND LINKED LISTS

Useful dynamic variables are seldom of a simple type such as `int` or `double`, but are normally of some complex type such as an array, `struct`, or class type. You saw that dynamic variables of an array type can be useful. Dynamic variables of a `struct` or class type can also be useful, but in a different way. Dynamic variables that are either `struct`s or classes normally have one or more member variables that are pointer variables which connect them to other dynamic variables. For example, one such structure, which happens to contain a shopping list, is diagrammed in Display 13.1.

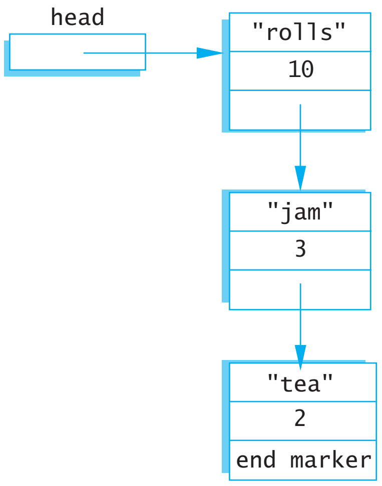
Nodes

A structure like the one shown in Display 13.1 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 13.1 contains a string, 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.

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 13.1, along with the type definition for a pointer to such nodes, can be as follows:

```
struct ListNode
{
    string item;
```

DISPLAY 13.1 Nodes and Pointers



```
        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 13.1 is not a node, but is 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 definition of the *struct* type ListNode is still blatantly circular. The definition uses the type name ListNode to define the member variable link. There is nothing wrong with this particular circularity, and it is allowed in C++. One indication that this definition is not logically inconsistent is the fact that you can draw pictures, like Display 13.1, that represent such structures.

We now have pointers inside of *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 above, the situation is as diagrammed in Display 13.1, 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:

```
(*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. So, the expression `*head` is 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`, and so `(*head).count` is the name of the *`int`* variable in the first node. The parentheses around `*head` are not optional. You want the dereferencing operator `*` to be performed before the dot operator. However, the dot operator has higher precedence than the dereferencing operator `*`, and so without the parentheses, the dot operator would be performed first (and that would produce an error). In the next paragraph, we will describe a shortcut notation that can avoid this worry about parentheses.

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. The **arrow operator** `->` combines the actions of a dereferencing operator `*` and a dot operator to specify a member of a dynamic *`struct`* or object that is pointed to by a given pointer. For example, the assignment statement above 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:

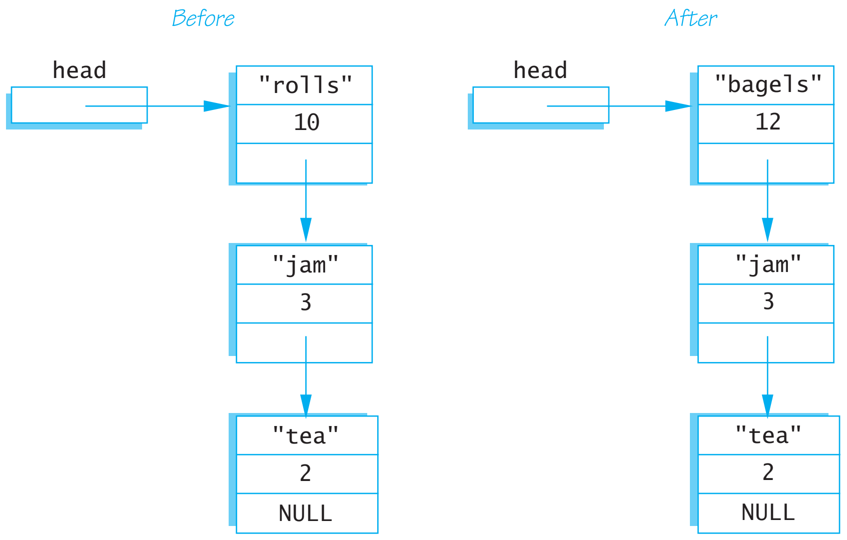
```
head->item = "bagels";
```

The result of these changes to the first node in the list is diagrammed in Display 13.2. Look at the pointer member in the last node in the lists shown in Display 13.2. This last node has the word `NULL` written where there should be a pointer. In Display 13.1 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 an end marker to signal the end of a linked list. `NULL` is a special defined constant that is part of the C++ language (provided as part of the required C++ libraries).

`NULL` is typically used for two different (but often coinciding) purposes. It 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

DISPLAY 13.2 Accessing Node Data

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



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 13.2, and when the program reaches the node that contains NULL, it knows that it has come to the end of the list.

NULL is 0

The constant NULL is actually the number 0, but we prefer to think of it and spell it as NULL. That makes it clear that you mean 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> or <cstddef> (or other suitable library) when you use NULL. No *using* directive is needed in order to make NULL available to your program code. In particular, it does not require *using namespace* std;, although other things in your code are likely to require something like *using namespace* std;.[1]

[1] The details are as follows: The definition of NULL is handled by the C++ preprocessor, which replaces NULL with 0. Thus, the compiler never actually sees "NULL" and so there are no namespace issues, and no *using* directive is needed.

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 as follows:

```
Pointer_Variable->Member_Name
```

The above refers to a member of the *struct* or 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'`:

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

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`.

nullptr

The fact that the constant NULL is actually the number 0 leads to an ambiguity problem. Consider the overloaded function below:

```
void func(int *p);
void func(int i);
```

Which function will be invoked if we call `func(NULL)`? Since NULL is the number 0, both are equally valid. C++11 resolves this problem by introducing a new constant, `nullptr`. `nullptr` is not the integer zero, but it is a literal constant used to represent a null pointer. Use `nullptr` anywhere you would have used NULL for a pointer. For example, we can write:

```
double *there = nullptr;
```

nullptr

`nullptr` is a special constant value that is used the same way as NULL, but it can only be assigned to a **pointer**. It is not the number 0. Use `nullptr` to differentiate between a null pointer and the number 0. `nullptr` was introduced in C++11.

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;
```

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 in order to set the value of the member variable next of this node equal to NULL?

3. Suppose that your program contains the type definitions and code given in Self-Test Exercise 1. Assuming that the value of the pointer variable head has not been changed, how can you destroy the dynamic variable pointed to by head and return the memory it uses to the freestore so that it can be reused to create new dynamic variables?

4. Given the following structure definition:

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

   write code to assign the string "Wilbur's brother Orville" to the member item of the node pointed to by head.

Linked Lists

Lists such as those shown in Display 13.2 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 the head of the list. The last node has no special name, but it does have a special property. The last node has NULL as the value of its member pointer variable. To test to see whether a node is the last node, you need only test to see if the pointer variable in the node is equal to NULL.

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 node than that used in Display 13.2. These nodes will contain only an integer and a pointer. The node and pointer type definitions that we will use are as follows:

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;
```

As a warm-up exercise, let's 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:

```
NodePtr head;
```

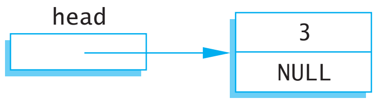
To create our first node, we use the operator *new* to create a new dynamic variable that will become the first node in our linked list.

```
head = new Node;
```

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

```
head->data = 3;
head->link = NULL;
```

Notice that the pointer member of this node is set equal to NULL. That is 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:



Our one-node list was built in a purely 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 head_insert(NodePtr& head, int the_number);
```

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.

To insert a new node into the linked list, our function will use the ***new*** operator to create a new node. The data is then copied into the new node, and the new node is inserted at the head of the list. When we insert nodes this way, the new node will be the first node in the list (that is, the head node) rather than the last node. Since dynamic variables have no names, we must use a local pointer variable to point to this node. If we call the local pointer variable `temp_ptr`, the new node can be referred to as `*temp_ptr`. The complete process can be summarized as follows:

Pseudocode for `head_insert` Function

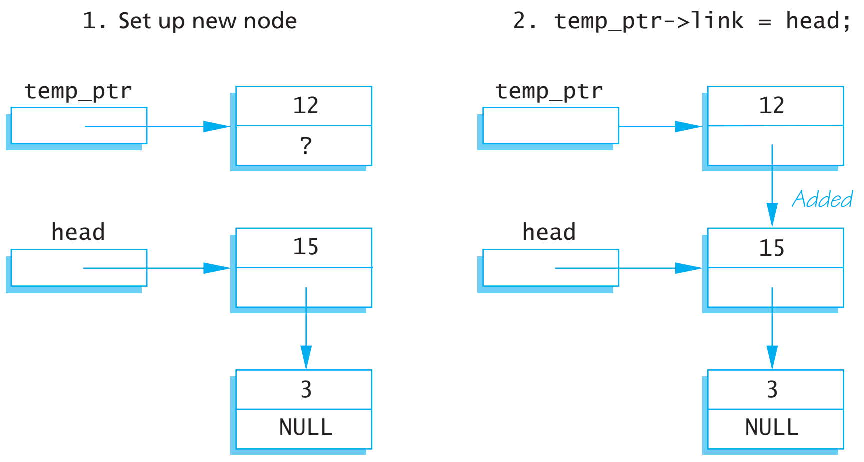
```
1. Create a new dynamic variable pointed to by temp_ptr. (This new
   dynamic variable is the new node. This new node can be referred to as
   *temp_ptr.)
2. Place the data in this new node.
3. Make the link member of this new node point to the head node (first
   node) of the original linked list.
4. Make the pointer variable named head point to the new node.
```

Display 13.3 contains a diagram of this algorithm. Steps 2 and 3 in the diagram can be expressed by these C++ assignment statements:

```
temp_ptr->link = head;
head = temp_ptr;
```

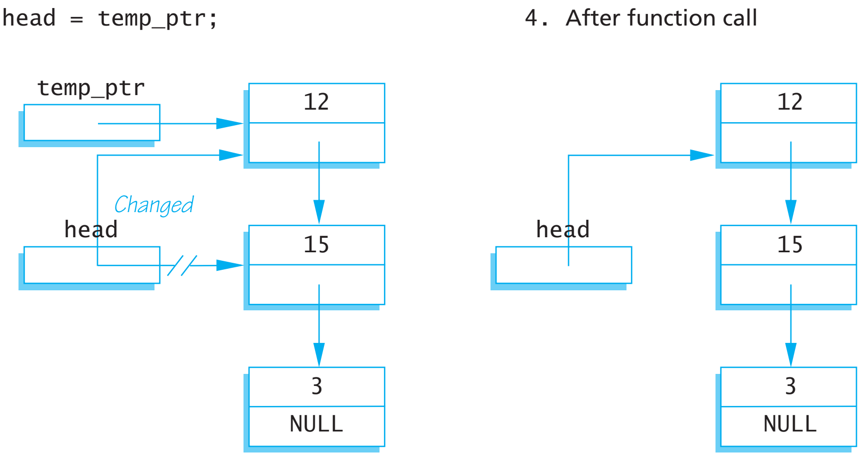
The complete function definition is given in Display 13.4.

DISPLAY 13.3 Adding a Node to a Linked List *(part 1 of 2)*



(continued)

DISPLAY 13.3 Adding a Node to a Linked List *(part 2 of 2)*



DISPLAY 13.4 Function to Add a Node at the Head of a Linked List

Function Declaration

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

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

Function Definition

```
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = head;
    head = temp_ptr;
}
```

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, you use the pointer `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 you set the value of `head` as follows:

```
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 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 `head_insert` in Display 13.4 was designed with nonempty lists as the model, but a check will show that it works for the empty list as well.

PITFALL Losing Nodes

You might be tempted to write the function definition for `head_insert` (Display 13.4) using the pointer variable `head` to construct the new node, instead of using the local pointer variable `temp_ptr`. If you were to try, you might start the function as follows:

```
head = new Node;
head->data = the_number;
```

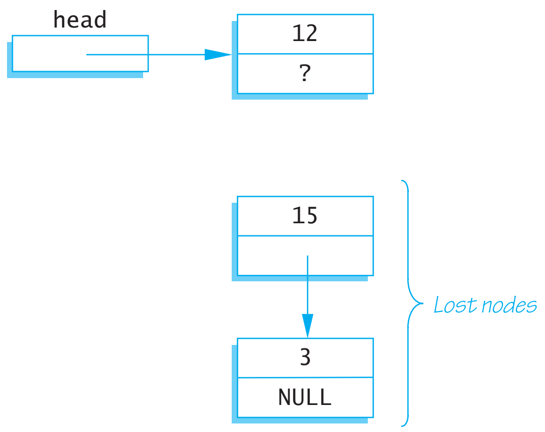
At this point the new node is constructed, contains the correct data, and is pointed to by the pointer `head`, all as it is supposed to be. All that is left to do is to attach the rest of the list to this node by setting the pointer member given below so that it points to what was formerly the first node of the list:

```
head->link
```

Display 13.5 shows the situation when the new data value is `12`. That illustration reveals 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 ending with that node), there is no way the program can reference this node. The node below this node is also lost. A program cannot make a pointer point to either of these nodes, nor can it access the data in these nodes, nor can it do anything else to the nodes. 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, causing abnormal termination. Worse, a memory leak (lost nodes) in an

DISPLAY 13.5 Lost Nodes



ordinary user's program can 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`. ■

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 `Node`, that we used in the previous subsections. (The definition of the node and pointer types is given in Display 13.4.) The function we design will have two arguments: for the linked list and the integer we want to locate. The function will return a pointer that points to the first node which contains that integer. If no node contains the integer, the function will return the pointer `NULL`. This way, our program can test to see whether the integer is on 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 is as follows:

```
NodePtr search(NodePtr 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. So 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 13.6. 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:

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;
```

In order 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->link
```

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 (member) variable. Hence, the following will move the pointer `here` to the next node in the list:

```
here = here->link;
```

Putting these pieces together yields the following refinement of the algorithm pseudocode:

Preliminary Version of the Code for the `search` Function

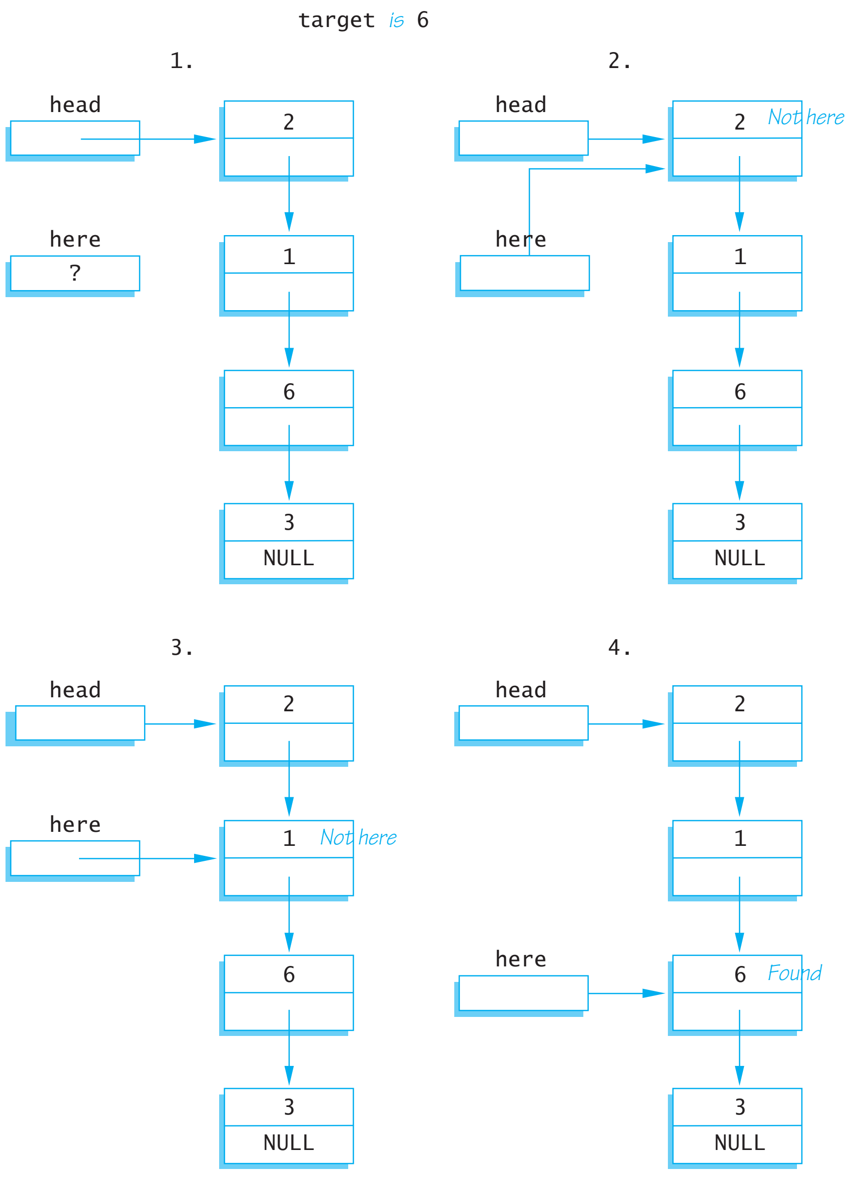
```
here = head;

while (here->data != target &&  here->link != NULL)
    here = here->link;

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

Notice the Boolean expression in the *`while`* statement. We test to see if `here` is not pointing to the last node by testing to see if the member variable `here->link` is not equal to `NULL`.

DISPLAY 13.6 Searching a Linked List



We still must go back and take care of the empty list. If we check our 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->data
here->link
```

When `here` is `NULL`, it is not pointing to any node, so there is no member named `data` nor any member named `link`. Hence, we make a special case of the empty list. The complete function definition is given in Display 13.7.

DISPLAY 13.7 Function to Locate a Node in a Linked List

Function Declaration

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

NodePtr search(NodePtr 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:
NodePtr search(NodePtr head, int target)
{
    NodePtr here = head;

    if (here == NULL)
    {
        return NULL;           Empty list case
    }
    else
    {
        while (here->data != target &&
               here->link != NULL)
            here = here->link;

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

Pointers as Iterators

An **iterator** is a construct 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. An iterator can be an object of some iterator class or something simpler, such as an array index or a pointer. Pointers provide a simple example of an iterator. In fact, a pointer is the prototypical example of an iterator. The basic ideas can be easily seen in the context of linked lists. 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:

```
Node_Type *iter;
for (iter = head; iter != NULL; iter = iter->link)
    Do whatever you want with the node pointed to by iter;
```

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 have been discussing, you could use

```
NodePtr iter; //Equivalent to: Node *iter;
for (iter = head; iter != NULL; iter = iter->link)
    cout << (iter->data);
```

The definition of `Node` and `NodePtr` are given in Display 13.7.

Inserting and Removing Nodes Inside a List

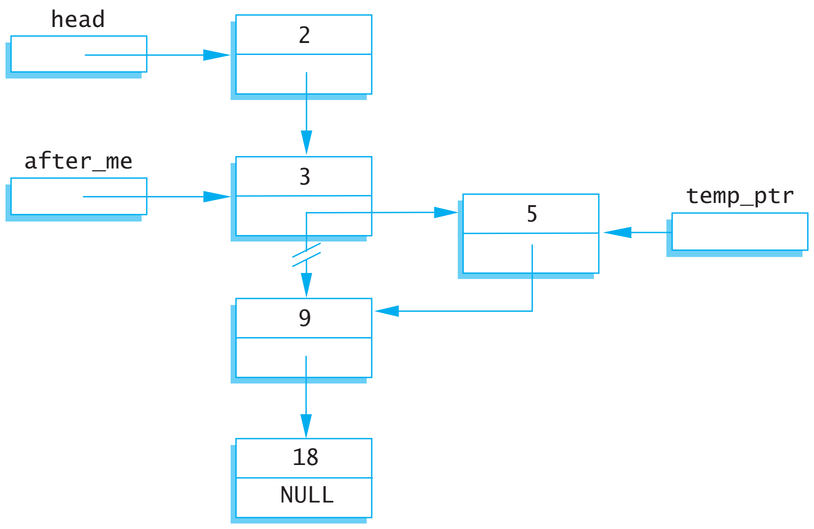
Inserting in the middle of a list

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 numeric order or alphabetical order, 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 `after_me` pointing to some node in the linked list. We want the new node to be placed after the node pointed to by `after_me`, as illustrated in Display 13.8. The same technique works for nodes with any kind of data, but to be concrete, we are using the same type of nodes as in previous subsections. The type definitions are given in Display 13.7. The function declaration for the function we want to define is:

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

A new node is set up the same way it was in the function `head_insert` in Display 13.4. The difference between this function and that one is that we now wish to insert the node not at the head of the list, but after the node

DISPLAY 13.8 Inserting in the Middle of a Linked List



pointed to by `after_me`. The way to do the insertion is shown in Display 13.8 and is expressed as follows in C++ code:

```
//add a link from the new node to the list:
temp_ptr->link = after_me->link;
//add a link from the list to the new node:
after_me->link = temp_ptr;
```

The order of these two assignment statements is critical. In the first assignment we want the pointer value `after_me->link` *before it is changed.* The complete function is given in Display 13.9.

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 `after_me` 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 `head_insert` given in Display 13.4 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 order or alphabetical order or 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 used an array, much, and in extreme cases all, of the array would have to be copied in order to make room for a new value in the correct spot. Despite the overhead involved in positioning the pointer `after_me`, 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 13.10 illustrates the method. Once the pointers `before` and `discard` have

DISPLAY 13.9 Function to Add a Node in the Middle of a Linked List

Function Declaration

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

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

Function Definition

```
void insert(NodePtr after_me, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = after_me->link;
    after_me->link = temp_ptr;
}
```

been positioned, all that is required to remove the node is the following statement:

```
before->link = discard->link;
```

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 it and return the memory it uses to the freestore; you can do this with a call to `delete` as follows:

```
delete discard;
```

PITFALL Using the Assignment Operator with Dynamic Data Structures

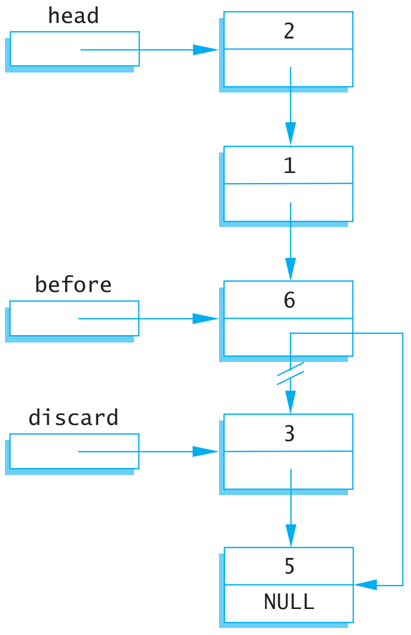
If `head1` and `head2` are pointer variables and `head1` points to the head node of a linked list, the following will make `head2` point to the same head node and hence the same linked list:

```
head2 = head1;
```

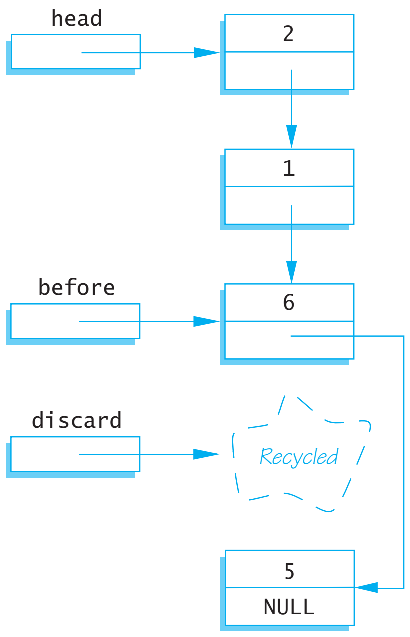
DISPLAY 13.10 Removing a Node

1. Position the pointer `discard` so that it points to the node to be deleted, and position the pointer before so that it points to the node before the one to be deleted.

2. `before->link = discard->link;`



3. `delete discard;`



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 list.

If `head1` points to a linked list and you want `head2` to point to a second, identical *copy* of this linked list, the assignment statement above will not work. Instead, you must copy the entire linked list node by node. Alternatively, you can overload the assignment operator = so that it means whatever you want it to mean. Overloading = is discussed in the subsection of Chapter 11 entitled "Overloading the Assignment Operator." ■

SELF-TEST EXERCISES

5. 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.

6. A linked list is normally given by giving 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?

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

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

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

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

8. Suppose your program contains type definitions and pointer variable declarations as in Self-Test Exercise 7. Suppose further that `p2` points to a node of type `Node` that is on a linked list and 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 on the linked list. (*Hint:* You might want to declare another pointer variable to use.)

9. Choose an answer and explain it.

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

   a. More efficient

   b. Less efficient

   c. About the same

   d. Dependent on the size of the two lists

Variations on Linked Lists

In this subsection we give you a hint of the many data structures that can be created using nodes and pointers. We briefly describe two additional data structures, the doubly linked list and the binary tree.

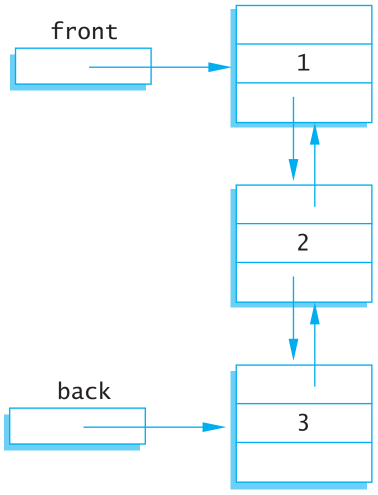
An ordinary linked list allows you to move down the list in only one direction (following the links). A node in a **doubly linked list** has two links, one link that points to the next node and one that points to the previous node. Diagrammatically, a doubly linked list looks like the sample list in Display 13.11. The node class for a doubly linked list could be as follows:

```
struct Node
{
    int data;
    Node *forward_link;
    Node *back_link;
};
```

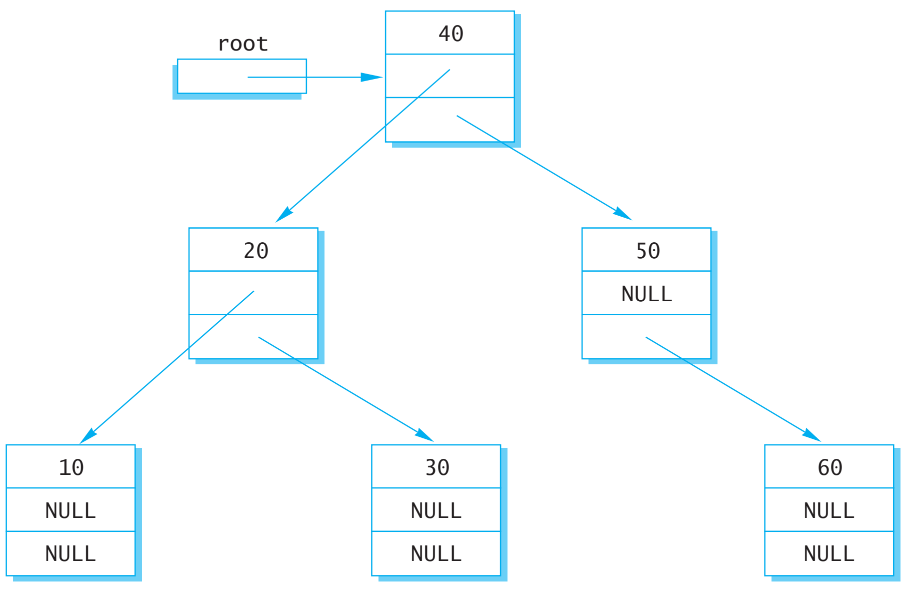
Rather than a single pointer to the head node, a doubly linked list normally has a pointer to each of the two end nodes. You can call these pointers `front` and `back`, although the choice of which is `front` and which is `back` is arbitrary. The definitions of constructors and some of the functions in the doubly linked list class will have to change (from the singly linked case) to accommodate the extra link.

A **tree** is a data structure that is structured as shown in Display 13.12. In particular, in a tree you can reach any node from the top (root) node by some path that follows the links. Note that there are no cycles in a tree. If you follow the links, you eventually get to an "end." Note that each node has two links that point

DISPLAY 13.11 A Doubly Linked List



DISPLAY 13.12 A Binary Tree



to other nodes (or the value `NULL`). This sort of tree is called a **binary tree,** because each node has exactly two links. There are other kinds of trees with different numbers of links in the nodes, but the binary tree is the most common case.

A tree is not a form of linked list, but does use links (pointers) in ways that are similar to how they are used in linked lists. The definition of the node type for a binary tree is essentially the same as what it is for a doubly linked list, but the two links are usually named using some form of the words *`left`* and *`right`*. The following is a node type that can be used for constructing a binary tree:

```
struct TreeNode
{
    int data;
    TreeNode *left_link;
    TreeNode *right_link;
};
```

In Display 13.12, the pointer named `root` points to the **root node** ("top node"). The root node serves a purpose similar to that of the head node in an ordinary linked list (Display 13.10). Any node in the tree can be reached from the root node by following the links.

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 tree branching structure. The secret to the terminology is to turn the picture (Display 13.12) upside down. The picture then does resemble the branching structure of a tree and the root node is where the tree's root would begin. The nodes at the ends of the branches with both link instance variables set to NULL are known as **leaf nodes**, a terminology that may now make some sense.

Although we do not have room to pursue the topic in this book, binary trees can be used to efficiently store and retrieve data.

Linked Lists of Classes

In the preceding examples we created linked lists by using a struct to hold the contents of a node within the list. It is possible to create the same data structures using a class instead of a struct. The logic is identical except the syntax of using and defining a class should be substituted in place of that for a struct.

VideoNote
Walkthrough of Linked Lists of Classes

Displays 13.13 and 13.14 illustrate how to define a Node class. The data variables are declared *private* using the principle of information hiding, and *public* methods have been created to access the data value and next node in the link. Display 13.15 creates a short list of five nodes by inserting new nodes

DISPLAY 13.13 Interface File for a Node Class

```
//This is the header file for Node.h. This is the interface for
//a node class that behaves similarly to the struct defined
//in Display 13.4
namespace linkedlistofclasses
{
    class Node
    {
    public:
        Node( );
        Node(int value, Node *next);
        //Constructors to initialize a node

        int getData( ) const;
        //Retrieve value for this node

        Node *getLink( ) const;
        //Retrieve next Node in the list

        void setData(int value);
        //Use to modify the value stored in the list

        void setLink(Node *next);
        //Use to change the reference to the next node

    private:
        int data;
        Node *link;
    };
    typedef Node* NodePtr;
} //linkedlistofclasses
//Node.h
```

DISPLAY 13.14 Implementation File for a Node Class

```
//This is the implementation file Node.cpp.
//It implements logic for the Node class. The interface
//file is in the header file Node.h
#include <iostream>
#include "Node.h"

namespace linkedlistofclasses
{
    Node::Node( ) : data(0), link(NULL)
    {
        //deliberately empty
    }

    Node::Node(int value, Node *next) : data(value), link(next)
    {
        //deliberately empty
    }

    //Accessor and Mutator methods follow

    int Node::getData( ) const
    {
        return data;
    }

    Node* Node::getLink( ) const
    {
        return link;
    }

    void Node::setData(int value)
    {
        data = value;
    }

    void Node::setLink(Node *next)
    {
        link = next;
    }
} //linkedlistofclasses
//Node.cpp
```

DISPLAY 13.15 Program Using the Node Class *(part 1 of 3)*

```
//This program demonstrates the creation of a linked list
//using the Node class. Five nodes are created, output, then
//destroyed.
```

(continued)

DISPLAY 13.15 Program Using the `Node` Class *(part 2 of 3)*

```
#include <iostream>
#include "Node.h"

using namespace std;
using namespace linkedlistofclasses;

//This function inserts a new node onto the head of the list
//and is a class-based version of the same function defined
//in Display 13.4.
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    //The constructor sets temp_ptr->link to head and
    //sets the data value to the_number
    temp_ptr = new Node(the_number, head);
    head = temp_ptr;
}

int main()
{
    NodePtr head, tmp;

    //Create a list of nodes 4 -> 3 -> 2 -> 1 -> 0
    head = new Node(0, NULL);
    for (int i = 1; i < 5; i++)
    {
        head_insert(head, i);
    }
    //Iterate through the list and display each value
    tmp = head;
    while (tmp != NULL)
    {
        cout << tmp->getData() << endl;
        tmp = tmp->getLink();
    }
    //Delete all nodes in the list before exiting
    //the program.
    tmp = head;
    while (tmp != NULL)
    {
        NodePtr nodeToDelete = tmp;
        tmp = tmp->getLink();
        delete nodeToDelete;
    }
    return 0;
}
```

(continued)

DISPLAY 13.15 Program Using the `Node` Class *(part 3 of 3)*

Sample Dialogue

```
4
3
2
1
0
```

onto the front of the list. The `head_insert` function is logically identical to the same function defined in Display 13.4 except the constructor defined for the `Node` class is used to set the data.

13.2 STACKS AND QUEUES

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

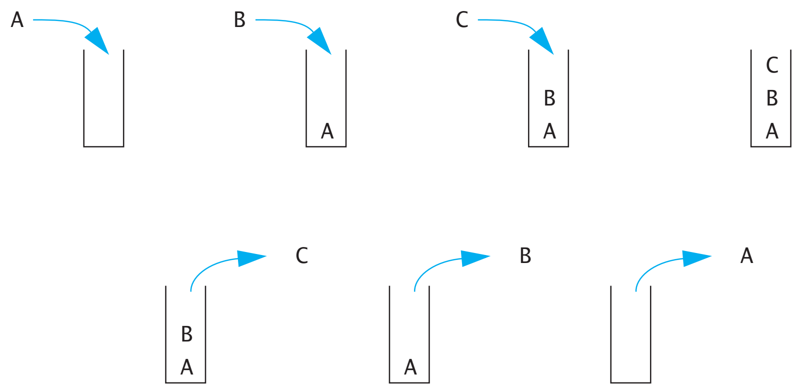
MATTHEW 19:30

Linked lists have many applications. In this section we give two samples of what they can be used for. We use linked lists to give implementations of two data structures known as a *stack* and a *queue*. In this section we always use regular linked lists and not doubly linked lists.

Stacks

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'`, `'B'`, and then `'A'`. This use of a stack is diagrammed in Display 13.16. As shown

DISPLAY 13.16 A Stack



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* (LIFO) data structure.

Stacks are used for many language processing tasks. In Chapter 14 we will discuss how the computer system uses a stack to keep track of C++ function calls. However, here we will do only one very simple application. Our goal in this example is to show you how you can use the linked list techniques to implement specific data structures; a stack is one simple example of the use of linked lists. You need not read Chapter 14 to understand this example.

PROGRAMMING EXAMPLE A Stack Class

The interface for our `Stack` class is given in Display 13.17. This particular stack is used to store data of type *char*. You can define a similar stack to store data of any other type. 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

DISPLAY 13.17 Interface File for a Stack Class *(part 1 of 2)*

```
//This is the header file stack.h. This is the interface for the class Stack,
//which is a class for a stack of symbols.
#ifndef STACK_H
#define STACK_H
namespace stacksavitch
{
    struct StackFrame
    {
        char data;
        StackFrame *link;
    };

    typedef StackFrame* StackFramePtr;

    class Stack
    {
    public:
        Stack( );
        //Initializes the object to an empty stack.
        Stack(const Stack& a_stack);
        //Copy constructor.

        ~Stack( );
        //Destroys the stack and returns all the memory to the freestore.
```

(continued)

DISPLAY 13.17 Interface File for a `Stack` Class *(part 2 of 2)*

```
            void push(char the_symbol);
            //Postcondition: the_symbol has been added to the stack.

            char pop( );
            //Precondition: The stack is not empty.
            //Returns the top symbol on the stack and removes that
            //top symbol from the stack.

            bool empty( ) const;
            //Returns true if the stack is empty. Returns false otherwise.
        private:
            StackFramePtr top;
        };
    }//stacksavitch

    #endif //STACK_H
```

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`.

The names `push` and `pop` derive from another 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, you 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.

Application program

Display 13.18 shows a simple program that illustrates how the `Stack` class is used. This program reads a word one letter at a time and places the letters in a stack. The program then removes the letters one by one and writes them to

DISPLAY 13.18 Program Using the `Stack` Class *(part 1 of 2)*

```
//Program to demonstrate use of the Stack class.
#include <iostream>
#include "stack.h"
using namespace std;
using namespace stacksavitch;

int main( )
{
```

(continued)

DISPLAY 13.18 Program Using the Stack Class *(part 2 of 2)*

```
    stack s;
    char next, ans;

    do
    {
        cout << "Enter a word: ";
        cin.get(next);
        while (next != '\n')
        {
            s.push(next);
            cin.get(next);
        }

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

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

    return 0;
}
```

<The `ignore` *member of* `cin` *is discussed in Chapter 8. It discards input remaining on the current input line up to 10,000 characters or until a return is entered. It also discards the return (*`'\n'`*) at the end of the line.>*

Sample Dialogue

```
Enter a word: straw
Written backward that is: warts
Again?(y/n): y
Enter a word: C++
Written backward that is: ++C
Again?(y/n): n
```

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 word written backward.

Implementation

As shown in Display 13.19, 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 variable top is a pointer that points to the head of the linked list.

DISPLAY 13.19 Implementation of the Stack Class *(part 1 of 2)*

```
//This is the implementation file stack.cpp.
//This is the implementation of the class Stack.
//The interface for the class Stack is in the header file stack.h.
#include <iostream>
#include <cstddef>
#include "stack.h"
using namespace std;

namespace stacksavitch
{
    //Uses cstddef:
    Stack::Stack( ) : top(NULL)
    {
        //Body intentionally empty.
    }

    Stack::Stack(const Stack& a_stack)
```

<The definition of the copy constructor is Self-Test Exercise 11.>

```
    Stack::~Stack( )
    {
        char next;
        while (! empty( ))
            next = pop( ); //pop calls delete.
    }

    //Uses cstddef:
    bool Stack::empty( ) const
    {
        return (top == NULL);
    }

    void Stack::push(char the_symbol)
```

<The rest of the definition is Self-Test Exercise 10.>

```
    //Uses iostream:
    char Stack::pop( )
    {
        if (empty( ))
        {
            cout << "Error: popping an empty stack.\n";
            exit(1);
        }

```

(continued)

DISPLAY 13.19 Implementation of the `Stack` Class *(part 2 of 2)*

```
            char result = top->data;

            StackFramePtr temp_ptr;
            temp_ptr = top;
            top = top->link;

            delete temp_ptr;

            return result;
        }
    }//stacksavitch
```

Writing the definition of the member function `push` is Self-Test Exercise 10. However, we have already given the algorithm for this task. The code for the `push` member function is essentially the same as the function `head_insert` shown in Display 13.4, 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 complicated but does not use any techniques we have not already discussed. The details are left to Self-Test Exercise 11.

The `pop` member function first checks to see if the stack is empty. If the stack 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. That is done as follows:

```
char result = top->data;
```

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

```
top = top->link;
```

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

```
delete temp_ptr;
```

Each node that is removed from the linked list by the member function `pop` is destroyed with a call to *`delete`*. Thus, 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.

SELF-TEST EXERCISES

10. Give the definition of the member function push of the class Stack described in Display 13.17.

11. Give the definition of the copy constructor for the class Stack described in Display 13.17.

Queues

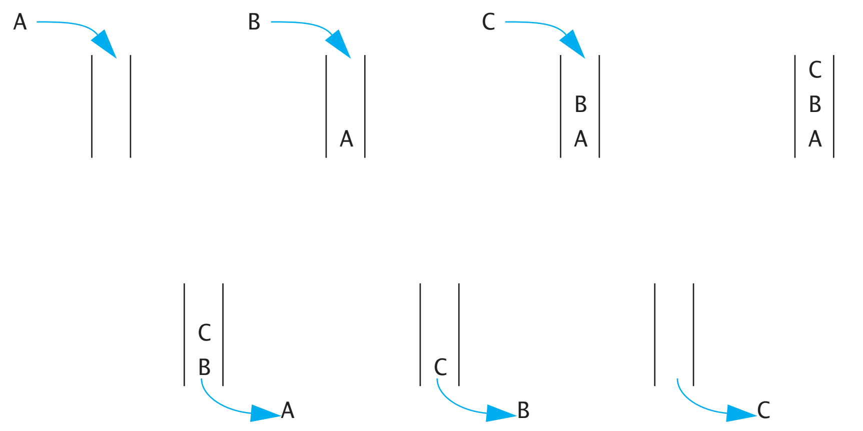
A stack is a last-in/first-out data structure. Another common data structure is a **queue**, which handles data in a first-in/first-out (FIFO) fashion. A queue behaves exactly the same as a line of people waiting for a bank teller or other service. The people are served in the order they enter the line (the queue). The operation of a queue is diagrammed in Display 13.20.

A queue can be implemented with a linked list in a manner that is similar to our implementation of the `Stack` class. However, a queue needs a pointer at both the head of the list and at the other 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. So, our implementation will remove a node from the head of the list (which we will now call the **front** of the list) and we 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).

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.

DISPLAY 13.20 A Queue



PROGRAMMING EXAMPLE A Queue Class

The interface for our queue class is given in Display 13.21. This particular queue is used to store data of type *char*. You can define a similar queue to store data of any other type. There are two basic operations you can perform on a queue: adding an item to the end of the queue and removing an item from the front of the queue.

DISPLAY 13.21 Interface File for a Queue Class

```
//This is the header file queue.h. This is the interface for the class Queue,
//which is a class for a queue of symbols.
#ifndef QUEUE_H
#define QUEUE_H
namespace queuesavitch
{
    struct QueueNode
    {
        char data;
        QueueNode *link;
    };
    typedef QueueNode* QueueNodePtr;

    class Queue
    {
    public:
        Queue();
        //Initializes the object to an empty queue.
        Queue(const Queue& aQueue);
        ~Queue();
        void add(char item);
        //Postcondition: item has been added to the back of the queue.
        char remove();
        //Precondition: The queue is not empty.
        //Returns the item at the front of the queue and
        //removes that item from the queue.
        bool empty() const;
        //Returns true if the queue is empty. Returns false otherwise.
    private:
        QueueNodePtr front; //Points to the head of a linked list.
                            //Items are removed at the head
        QueueNodePtr back;  //Points to the node at the other end of the
                            //linked list. Items are added at this end.
    };
}//queuesavitch
#endif //QUEUE_H
```

Display 13.22 shows a simple program that illustrates how the queue class is used. This program reads a word one letter at a time and places the letters in a queue. The program then removes the letters one by one and writes them to the screen. Because data is removed from a queue in the order in which it enters the queue, the output shows the letters in the word in the same order that the user entered them. It is good to contrast this application of a queue with a similar application using a stack that we gave in Display 13.18.

Application program

DISPLAY 13.22 Program Using the Queue Class *(part 1 of 2)*

```
//Program to demonstrate use of the Queue class.
#include <iostream>
#include "queue.h"
using namespace std;
using namespace queuesavitch;

int main()
{
    Queue q;
    char next, ans;

    do
    {
        cout << "Enter a word: ";
        cin.get(next);
        while (next != '\n')
        {
            q.add(next);
            cin.get(next);
        }

        cout << "You entered:: ";
        while ( ! q.empty() )
            cout << q.remove();
        cout << endl;

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

    return 0;
}
```

<The `ignore` *member of* `cin` *is discussed in Chapter 8. It discards input remaining on the current input line up to 10,000 characters or until a return is entered. It also discards the return (* `'\n'`*) at the end of the line.>*

(continued)

DISPLAY 13.22 Program Using the `Queue` Class *(part 2 of 2)*

Sample Dialogue

```
Enter a word: straw
You entered: straw
Again?(y/n): y
Enter a word: C++
You entered: C++
Again?(y/n): n
```

Implementation

As shown in Displays 13.21 and 13.23, our queue class is implemented as a linked list in which the head of the list serves as the front of the queue. The member variable `front` is a pointer that points to the head of the linked list. Nodes are removed at the head of the linked list. The member variable `back` is a pointer that points to the node at the other end of the linked list. Nodes are added at this end of the linked list.

An empty queue is just an empty linked list, so an empty queue is implemented by setting the pointers `front` and `back` equal to `NULL`. The rest of the details of the implementation are similar to things we have seen before.

DISPLAY 13.23 Implementation of the `Queue` Class *(part 1 of 3)*

```
//This is the implementation file queue.cpp.
//This is the implementation of the class Queue.
//The interface for the class Queue is in the header file queue.h.
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include "queue.h"
using namespace std;

namespace queuesavitch
{
    //Uses cstddef:
    Queue::Queue() : front(NULL), back(NULL)
    {
        //Intentionally empty.
    }

    Queue::Queue(const Queue& aQueue)
            <The definition of the copy constructor is Self-Test Exercise 12.>
```

(continued)

DISPLAY 13.23 Implementation of the `Queue` Class *(part 2 of 3)*

```

Queue::~Queue()
        <The definition of the destructor is Self-Test Exercise 13.>

//Uses cstddef:
bool Queue::empty() const
{
    return (back == NULL); //front == NULL would also work
}

//Uses cstddef:
void Queue::add(char item)
{
    if (empty())
    {
        front = new QueueNode;
        front->data = item;
        front->link = NULL;
        back = front;
    }

    else
    {
        QueueNodePtr temp_ptr;
        temp_ptr = new QueueNode;
        temp_ptr->data = item;
        temp_ptr->link = NULL;
        back->link = temp_ptr;
        back = temp_ptr;
    }
}

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

    char result = front->data;

    QueueNodePtr discard;
    discard = front;
    front = front->link;
```

(continued)

DISPLAY 13.23 Implementation of the Queue Class *(part 3 of 3)*

```
        if (front == NULL) //if you removed the last node
            back = NULL;

        delete discard;

        return result;
    }
}//queuesavitch
```

SELF-TEST EXERCISES

12. Give the definition of the copy constructor for the class `Queue` described in Display 13.21.

13. Give the definition of the destructor for the class `Queue` described in Display 13.21.

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` or `nullptr`.
- A stack is a first-in/last-out data structure. A stack can be implemented using a linked list.
- A queue is a first-in/first-out data structure. A queue can be implemented using a linked list.

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. *`delete`* `head`;

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

5.
```
struct NodeType
{
    char data;
    NodeType *link;
};

typedef NodeType* PointerType;
```

6. The pointer value `NULL` is used to indicate an empty list.

7. `p1 = p1-> next;`

8.
```
Pointer discard;
discard = p2->next;
//discard now 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;
```

9. 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.

10.
```
//Uses cstddef:
void Stack::push(char the_symbol)
{
    StackFramePtr temp_ptr;
    temp_ptr = new StackFrame;
```

```
        temp_ptr->data = the_symbol;

        temp_ptr->link = top;
        top = temp_ptr;
    }
```

11.
```
    //Uses cstddef:
    Stack::Stack(const Stack& a_stack)
    {
        if (a_stack.top == NULL)
            top = NULL;
        else
        {
            StackFramePtr temp = a_stack.top;//temp moves
                //through the nodes from top to bottom of
                //a_stack.
            StackFramePtr end;//Points to end of the new stack.

            end = new StackFrame;
            end->data = temp->data;
            top = end;
            //First node created and filled with data.
            //New nodes are now added AFTER this first node.

            temp = temp->link;
            while (temp != NULL)
            {
                end->link = new StackFrame;
                end = end->link;
                end->data = temp->data;
                temp = temp->link;
            }
            end->link = NULL;
        }
    }
```

12.
```
    //Uses cstddef:
    Queue::Queue(const Queue&aQueue)
    {
        if (aQueue.empty( ))
            front = back = NULL;
        else
        {
            QueueNodePtr temp_ptr_old = aQueue.front;
            //temp_ptr_old moves through the nodes
            //from front to back of aQueue.
            QueueNodePtr temp_ptr_new;
            //temp_ptr_new is used to create new nodes.

            back = new QueueNode;
            back->data = temp_ptr_old->data;
            back->link = NULL;
            front = back;
            //First node created and filled with data.
            //New nodes are now added AFTER this first node.

            temp_ptr_old = temp_ptr_old->link;
            //temp_ptr_old now points to second
            //node or NULL if there is no second node.

            while (temp_ptr_old != NULL)
            {
                temp_ptr_new = new QueueNode;
                temp_ptr_new->data = temp_ptr_old->data;
                temp_ptr_new->link = NULL;
                back->link = temp_ptr_new;
                back = temp_ptr_new;
                temp_ptr_old = temp_ptr_old->link;
            }
        }
    }
```

13.
```
Queue::~Queue( )
{
    char next;
    while (! empty( ))
        next = remove( );//remove calls delete.
}
```

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. The following program creates a linked list with three names:

```
#include <iostream>
#include <string>
using namespace std;

struct Node
{
    string name;
    Node *link;
};

typedef Node* NodePtr;

int main()
{
            NodePtr listPtr, tempPtr;
            listPtr = new Node;
            listPtr->name = "Emily";

            tempPtr = new Node;
            tempPtr->name = "James";
            listPtr->link = tempPtr;

            tempPtr->link = new Node;
            tempPtr = tempPtr->link;
            tempPtr->name = "Joules";
            tempPtr->link = NULL;

            return 0;

}
```

Add code to the `main` function that:

a. Outputs in order all names in the list.

b. Inserts the name "Joshua" in the list after "James" then outputs the modified list.

c. Deletes the node with "Joules" then outputs the modified list.

d. Deletes all nodes in the list.

2. Re-do Practice Program 1, but instead of a `struct`, use a class named `Node`. Your class should have appropriate member functions to set the name and the link to the next node in the list. You might also consider adding a constructor that can set the name and link.

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects 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 `merge_lists` 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 of 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 whose objects represent 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 notation above, 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 zero, 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 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 zero 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. In this project you will redo Programming Project 8 from Chapter 11 using a linked list instead of an array. As noted there, this is a linked list of *double* items. This fact may imply changes in some of the member functions. The members are as follows: a default constructor; a member function named `add_item` to add a *double* to the list; a test for a full list that is a Boolean-valued function named `full()`; and a *friend* function overloading the insertion operator <<.

5. A harder version of Programming Project 4 would be to write a class named `List`, similar to Project 4, but with all the following member functions:

   - Default constructor, `List();`
   - `double List::front();`, which returns the first item in the list
   - `double List::back();`, which returns the last item in the list
   - `double List::current();`, which returns the "current" item
   - `void List::advance();`, which advances the item that `current()` returns
   - `void List::reset();` to make `current()` return the first item in the list
   - `void List::insert(double after_me, double insert_me);`, which inserts `insert_me` into the list after `after_me` and increments the *private*: variable `count`.
   - `int size();`, which returns the number of items in the list
   - `friend istream& operator <<(istream& ins, double write_me);`

   The private data members should include the following:

```
node* head;
node* current;
int count;
```

and possibly one more pointer.

You will need the following *struct* (outside the list class) for the linked list nodes:

```
struct node
{
    double item;
    node *next;
};
```

Incremental development is essential to all projects of any size, and this is no exception. Write the definition for the `List` class, but do not implement any members yet. Place this class definition in a file `list.h`. Then `#include "list.h"` in a file that contains `int main(){}`. Compile your file. This will find syntax errors and many typographical errors that would cause untold difficulty if you attempted to implement members without this check. Then you should implement and compile one member at a time, until you have enough to write test code in your main function.

6. 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 eliminate every third suitor. When she reached the end of the line she would continue counting from the beginning.

VideoNote
Solution to Programming Project 13.6

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. A circular linked list is a linked list where the link field of the last node in the list refers to the node that is the head of the list. 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. Consider the possibility that you may need to delete the `"head"` node in the list.

7. Redo (or do for the first time) Programming Project 5 from Chapter 9. However, instead of a dynamic array to store the list of user IDs for each computer station, use a linked list. The node for the lists should contain the station number and user ID of the person logged in on that station. If nobody is logged on to a computer station, then no entry should exist in the linked list for that computer station.

8. Modify or rewrite the `Queue` class (Display 13.21 through 13.23) to simulate customer arrivals at the Department of Motor Vehicles (DMV) counter. As customers arrive, they are given a ticket number starting at 1 and incrementing with each new customer. When a customer service agent is free, the customer with the next ticket number is called. This system results in a FIFO queue of customers ordered by ticket number. Write a program that implements the queue and simulates customers entering and leaving the queue. Input into the queue should be the ticket number and a timestamp when the ticket was entered into the queue. A ticket and its corresponding timestamp is removed when a customer service agent handles the next customer. Your program should save the length of time the last three customers spent waiting in the queue. Every time a ticket is removed from the queue, update these times and output the average of the last three customers as an estimate of how long it will take until the next customer is handled. If nobody is in the queue, output that the line is empty.

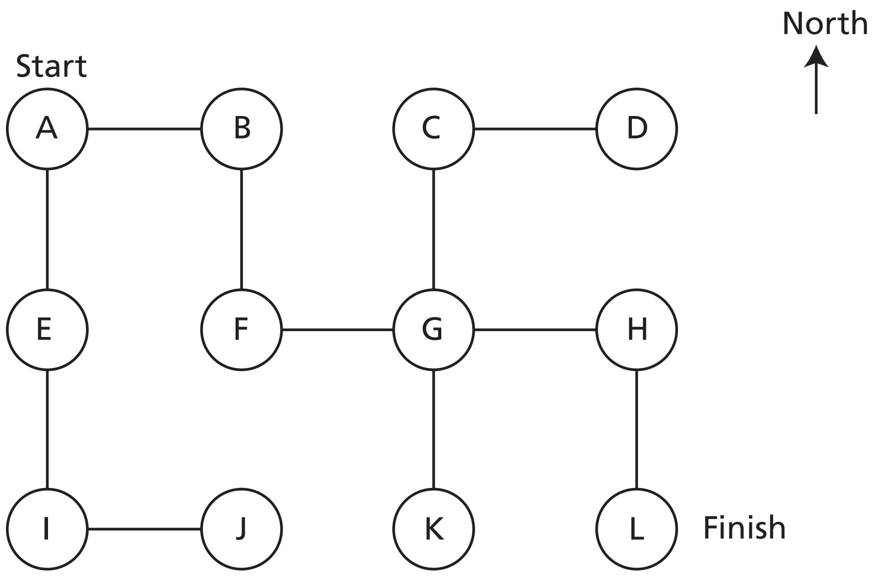
   Code to compute a timestamp based on the computer's clock is given below. The `time(NULL)` function returns the number of seconds since January 1, 1970, on most implementations of C++:

```
#include <ctime>
...
int main()
{
    long seconds;
    seconds = static_cast<long>(time(NULL));
    cout << "Seconds since 1/1/1970: " << seconds << endl;
    return 0;
}
Sample execution is shown here:
The line is empty.
Enter '1' to simulate a customer's arrival, '2' to help the
next customer, or '3' to quit.
1
Customer 1 entered the queue at time 100000044.
Enter '1' to simulate a customer's arrival, '2' to help the
next customer, or '3' to quit.
1
Customer 2 entered the queue at time 100000049.
Enter '1' to simulate a customer's arrival, '2' to help the
next customer, or '3' to quit.
1
```

```
Customer 3 entered the queue at time 100000055.
Enter '1' to simulate a customer's arrival, '2' to help the
next customer, or '3' to quit.
2
Customer 1 is being helped at time 100000069. Wait time = 25
seconds.
The estimated wait time for customer 2 is 25 seconds.
Enter '1' to simulate a customer's arrival, '2' to help the
next customer, or '3' to quit.
2
Customer 2 is being helped at time 100000076. Wait time = 27
seconds.
The estimated wait time for customer 3 is 26 seconds.
Enter '1' to simulate a customer's arrival, '2' to help the
next customer, or '3' to quit.
1
Customer 4 entered the queue at time 100000080.
Enter '1' to simulate a customer's arrival, '2' to help the
next customer, or '3' to quit.
2
Customer 3 is being helped at time 100000099. Wait time = 44
seconds.
The estimated wait time for customer 4 is 32 seconds.
```

9. 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 graph.

VideoNote
Solution to Programming Project 13.9



Write a program that implements the maze using nodes and pointers. Each node in the graph will correspond to a node in your code that is implemented in the form of a class or `struct`. The edges correspond to bidirectional links that point from one node to another. 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 here.

```
You are in room A of a maze of twisty little passages, all alike.
You can go (E)ast, (S)outh, or (Q)uit.
E
You are in room B of a maze of twisty little passages, all alike.
You can go (W)est, (S)outh, or (Q)uit.
S
You are in room F of a maze of twisty little passages, all alike.
You can go (E)ast, (N)orth, or (Q)uit.
E
```

10. Reverse Polish Notation (RPN), or postfix notation, is a format to specify mathematical expressions. In RPN, the operator comes after the operands instead of the normal format in which the operator is between the operands (this is called *infix* notation). Starting with an empty stack, a RPN calculator can be implemented with the following rules:

- If a number is input, push it on the stack.
- If `"+"` is input, then pop the last two operands off the stack, add them, and push the result on the stack.
- If `"-"` is input then pop `value1`, pop `value2`, then push `value2-value1` on the stack.
- If `"*"` is input, then pop the last two operands off the stack, multiply them, and push the result on the stack
- If `"/"` is input them pop value1, pop `value2`, then push `value2/value1` on the stack
- If `"q"` is input, then stop inputting values, print out the top of the stack, and exit the program

Modify the `Stack` class given in Section 13.2 to store integers instead of characters. Use the modified stack to implement a RPN calculator. Output an appropriate error message if there are not two operands on the stack when given an operator. Here is a sample input and output that is equivalent to `((10 - (2 + 3)) * 2)/5`:

```
10
2
3
+
```

(continued)

```
-
2
*
5
/
q
The top of the stack is:  2
```

11. You should complete the Programming Project 10 before attempting this one. Write a program that converts a fully parenthesized mathematical infix expression into an equivalent postfix expression and then evaluates the postfix expression. A fully parenthesized expression is one in which parentheses surround every operator and its operands. Starting with an empty stack of strings to store operators and an empty queue of strings to store the postfix expression, the conversion can be implemented with the following rules:

    - If `"("` is input, then ignore it.
    - If a number is input, then add it to the queue.
    - If an operator (either `"*"`, `"+"`, `"-"`, or `"/"`) is input, then push it on the stack.
    - If `")"` is input, then pop the operator from the stack and add it to the queue.
    - If `"q"` is input, then exit.

    When the final operator is popped from the stack, the queue contains the equivalent postfix expression. Use your solution from Programming Project 10 to evaluate it. You will need to convert a string object to an integer. Use the `c_str()` function to convert the string to a C string, and then use the `atoi` function to convert the C string into an integer. Refer to Chapter 8 for details.

    Sample output is shown below for `((10 - (2 + 3)) * 2)`,which translates to the postfix expression `10 2 3 + - 2 *`:

```
(
(
10
-
(
2
+
3
)
)
*
2
)
q
The expression evaluates to 10
```