17 Advanced Data Representation

You will learn about the following in this chapter:

- Functions:

  More `malloc()`
- Using C to represent a variety of data types
- New algorithms and increasing your ability to develop programs conceptually
- Abstract data types (ADTs)

Learning a computer language is like learning music, carpentry, or engineering. At first, you work with the tools of the trade, playing scales, learning which end of the hammer to hold and which end to avoid, solving countless problems involving falling, sliding, and balanced objects. Acquiring and practicing skills is what you've been doing so far in this book, learning to create variables, structures, functions, and the like. Eventually, however, you move to a higher level in which using the tools is second nature and the real challenge is designing and creating a project. You develop an ability to see the project as a coherent whole. This chapter concentrates on that higher level. You may find the material covered here a little more challenging than the preceding chapters, but you may also find it more rewarding because it helps you move from the role of apprentice to the role of craftsperson.

We'll start by examining a vital aspect of program design: the way a program represents data. Often the most important aspect of program development is finding a good representation of the data manipulated by that program. Getting data representation right can make writing the rest of the program simple. By now you've seen C's built-in data types: simple variables, arrays, pointers, structures, and unions.

Finding the right data representation, however, often goes beyond simply selecting a type. You should also think about what operations will be necessary. That is, you should decide how to store the data, and you should define what operations are valid for the data type. For example, C implementations typically store both the C `int` type and the C pointer type as integers, but the two types have different sets of valid operations. You can multiply one integer by another, for example, but you can't multiply a pointer by a pointer. You can use the * operator to dereference a pointer, but that operation is meaningless for an integer. The C language defines the valid operations for its fundamental types. However, when you design a scheme to represent data, you might need to define the valid operations yourself. In C, you can do so by designing C functions to represent the desired operations. In short, then, designing a data type consists of deciding on how to store the data and of designing a set of functions to manage the data.

You will also look at some *algorithms*, recipes for manipulating data. As a programmer, you will acquire a repertoire of such recipes that you apply over and over again to similar problems.

This chapter looks into the process of designing data types, a process that matches algorithms to data representations. In it, you'll meet some common data forms, such as the queue, the list, and the binary search tree.

You'll also be introduced to the concept of the abstract data type (ADT). An ADT packages methods and data representations in a way that is problem oriented rather than language oriented. After you've designed an ADT, you can easily reuse it in different circumstances. Understanding ADTs prepares you conceptually for entering the world of object-oriented programming (OOP) and the C++ language.

Exploring Data Representation

Let's begin by thinking about data. Suppose you had to create an address book program. What data form would you use to store information? Because there's a variety of information associated with each entry, it makes sense to represent each entry with a structure. How do you represent several entries? With a standard array of structures? With a dynamic array? With some other form? Should the entries be alphabetized? Should you be able to search through the entries by ZIP Code? By area code? The actions you want to perform might affect how you decide to store the information. In short, you have a lot of design decisions to make before plunging into coding.

How would you represent a bitmapped graphics image that you want to store in memory? A bitmapped image is one in which each pixel on the screen is set individually. In the days of black-and-white screens, you could use one computer bit (1 or 0) to represent one pixel (on or off), hence the name *bitmapped*. With color monitors, it takes more than one bit to describe a single pixel. For example, you can get 256 colors if you dedicate 8 bits to each pixel. Now the industry has moved to 65,536 colors (16 bits per pixel), 16,777,216 colors (24 bits per pixel), 2,147,483,648 colors (32 bits per pixel), and even beyond. If you have 32-bit colors and if your monitor has a resolution of 2560×1440, you'll need nearly 118 million bits (14MB) to represent a single screen of bitmapped graphics. Is this the way to go, or can you develop a way of compressing the information? Should this compression be *lossless* (no data lost) or *lossy* (relatively unimportant data lost)? Again, you have a lot of design decisions to make before diving into coding.

Let's tackle a particular case of representing data. Suppose you want to write a program that enables you to enter a list of all the movies (including videotapes, DVDs, and Blu-ray) you've seen in a year. For each movie, you'd like to record a variety of information, such as the title, the year it was released, the director, the lead actors, the length, the kind of film (comedy, science fiction, romance, drivel, and so forth), your evaluation, and so on. That suggests using a structure for each film and an array of structures for the list. To simplify matters, let's limit the structure to two members: the film title and your evaluation, a ranking on a 0-to-10 scale. Listing 17.1 shows a bare-bones implementation using this approach.

Listing 17.1 **The `films1.c` Program**

```
/* films1.c -- using an array of structures */
#include <stdio.h>
#include <string.h>
#define TSIZE        45      /* size of array to hold title   */
#define FMAX         5       /* maximum number of film titles */

struct film {
    char title[TSIZE];
    int rating;
};
char * s_gets(char * st, int n);
int main(void)
{
    struct film movies[FMAX];
    int i = 0;
    int j;

    puts("Enter first movie title:");
    while (i < FMAX && s_gets(movies[i].title, TSIZE) != NULL &&
           movies[i].title[0] != '\0')
    {
        puts("Enter your rating <0-10>:");
        scanf("%d", &movies[i++].rating);
        while(getchar() != '\n')
            continue;
        puts("Enter next movie title (empty line to stop):");
    }
    if (i == 0)
        printf("No data entered. ");
    else
        printf ("Here is the movie list:\n");

    for (j = 0; j < i; j++)
        printf("Movie: %s  Rating: %d\n", movies[j].title,
               movies[j].rating);
    printf("Bye!\n");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

The program creates an array of structures and then fills the array with data entered by the user. Entry continues until the array is full (the FMAX test), until end-of-file (the NULL test) is reached, or until the user presses the Enter key at the beginning of a line (the '\0' test).

This formulation has some problems. First, the program will most likely waste a lot of space because most movies don't have titles 40 characters long, but some movies do have long titles, such as *The Discreet Charm of the Bourgeoisie* and *Won Ton Ton, The Dog Who Saved Hollywood*. Second, many people will find the limit of five movies a year too restrictive. Of course, you can increase that limit, but what would be a good value? Some people see 500 movies a year, so you could increase `FMAX` to 500, but that still might be too small for some, yet it might waste enormous amounts of memory for others. Also, some compilers set a default limit for the amount of memory available for automatic storage class variables such as `movies`, and such a large array could exceed that value. You can fix that by making the array a static or external array or by instructing the compiler to use a larger stack, but that's not fixing the real problem.

The real problem here is that the data representation is too inflexible. You have to make decisions at compile time that are better made at runtime. This suggests switching to a data representation that uses dynamic memory allocation. You could try something like this:

```
#define TSIZE  45           /* size of array to hold title  */
struct film {
    char title[TSIZE];
    int rating;
};
```

```
...
    int n, i;
    struct film * movies;   /* pointer to a structure       */
    ...
    printf("Enter the maximum number of movies you'll enter:\n");
    scanf("%d", &n);
    movies = (struct film *) malloc(n * sizeof(struct film));
```

Here, as in Chapter 12, “Storage Classes, Linkage, and Memory Management,” you can use the pointer `movies` just as though it were an array name:

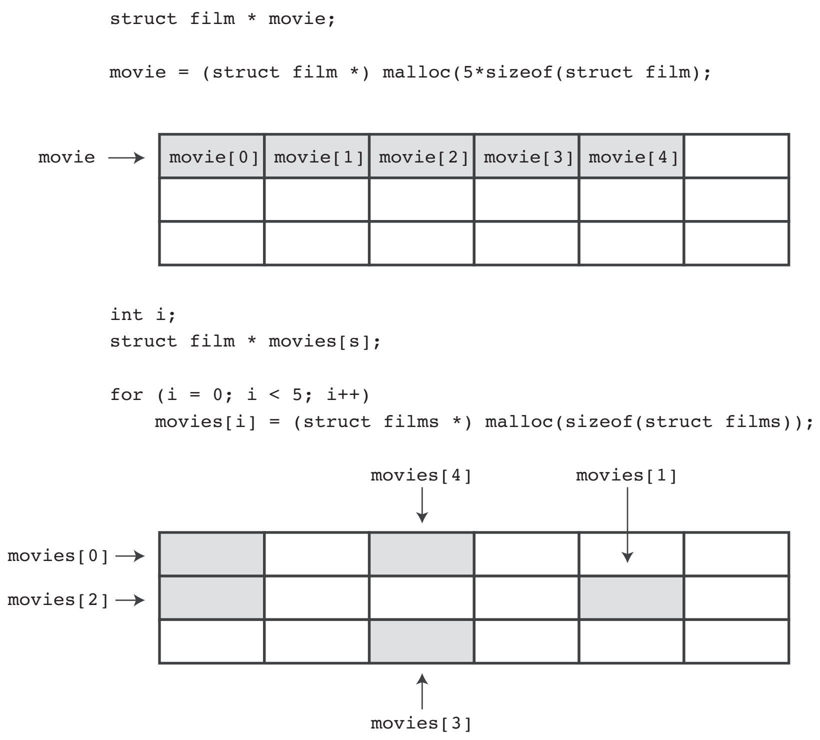
```
while (i < FMAX && s_gets(movies[i].title, TSIZE) != NULL &&
         movies[i].title[0] != '\0')
```

By using `malloc()`, you can postpone determining the number of elements until the program runs, so the program need not allocate 500 elements if only 20 are needed. However, it puts the burden on the user to supply a correct value for the number of entries.

Beyond the Array to the Linked List

Ideally, you’d like to be able to add data indefinitely (or until the program runs out of memory) without specifying in advance how many entries you’ll make and without committing the program to allocating huge chunks of memory unnecessarily. You can do this by calling `malloc()` after each entry and allocating just enough space to hold the new entry. If the user enters three films, the program calls `malloc()` three times. If the user enters 300 films, the program calls `malloc()` 300 times.

This fine idea raises a new problem. To see what it is, compare calling `malloc()` once, asking for enough space for 300 `film` structures, and calling `malloc()` 300 times, each time asking for enough space for one `film` structure. The first case allocates the memory as one contiguous memory block and all you need to keep track of the contents is a single pointer-to-`struct` variable (`film`) that points to the first structure in the block. Simple array notation lets the pointer access each structure in the block, as shown in the preceding code segment. The problem with the second approach is that there is no guarantee that consecutive calls to `malloc()` yield adjacent blocks of memory. This means the structures won’t necessarily be stored contiguously (see Figure 17.1). Therefore, instead of storing one pointer to a block of 300 structures, you need to store 300 pointers, one for each independently allocated structure!



```
struct film * movie;

movie = (struct film *) malloc(5*sizeof(struct film);
```

```
int i;
struct film * movies[s];

for (i = 0; i < 5; i++)
    movies[i] = (struct films *) malloc(sizeof(struct films));
```

Figure 17.1 Allocating structures in a block versus allocating them individually.

One solution, which we won't use, is to create a large array of pointers and assign values to the pointers, one by one, as new structures are allocated:

```
#define TSIZE  45                /* size of array to hold titles    */
#define FMAX   500               /* maximum number of film titles   */
struct film {
    char title[TSIZE];
    int rating;
};
...
    struct film * movies[FMAX]; /* array of pointers to structures */
    int i;
    ...
    movies[i] = (struct film *) malloc (sizeof (struct film));
```

This approach saves a lot of memory if you don't use the full allotment of pointers, because an array of 500 pointers takes much less memory than an array of 500 structures. It still wastes the space occupied by unused pointers, however, and it still imposes a 500-structure limit.

There's a better way. Each time you use `malloc()` to allocate space for a new structure, you can also allocate space for a new pointer. "But," you say, "then I need another pointer to keep track of the newly allocated pointer, and that needs a pointer to keep track of it, and so on." The trick to avoiding this potential problem is to redefine the structure so that each structure includes a pointer to the *next* structure. Then, each time you create a new structure, you can store its address in the preceding structure. In short, you need to redefine the `film` structure this way:

```
#define TSIZE  45      /* size of array to hold titles  */
struct film {
    char title[TSIZE];
    int rating;
    struct film * next;
};
```

True, a structure can't contain in itself a structure of the same type, but it can contain a pointer to a structure of the same type. Such a definition is the basis for defining a *linked list*—a list in which each item contains information describing where to find the next item.

Before looking at C code for a linked list, let's take a conceptual walk through such a list. Suppose a user enters `Modern Times` as a title and `10` as a rating. The program would allocate space for a `film` structure, copy the string `Modern Times` into the `title` member, and set the `rating` member to `10`. To indicate that no structure follows this one, the program would set the `next` member pointer to `NULL`. (`NULL`, recall, is a symbolic constant defined in the `stdio.h` file and represents the null pointer.) Of course, you need to keep track of where the first structure is stored. You can do this by assigning its address to a separate pointer that we'll refer to as the *head pointer*. The head pointer points to the first item in a linked list of items. Figure 17.2 represents how this structure looks. (The empty space in the `title` member is suppressed to save space in the figure.)

```
#define TSIZE 45
struct film {
   char title[TSIZE]
   int rating;
   struct film * next;
};
struct film * head;
```

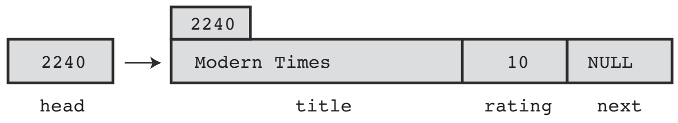


Figure 17.2 First item in a linked list.

Now suppose the user enters a second movie and rating—for example, `Midnight in Paris` and `8`. The program allocates space for a second `film` structure, storing the address of the new structure in the `next` member of the first structure (overwriting the `NULL` previously stored there) so that the `next` pointer of one structure points to the following structure in the linked list. Then the program copies `Midnight in Paris` and `8` to the new structure and sets its `next` member to `NULL`, indicating that it is now the last structure in the list. Figure 17.3 shows this list of two items.

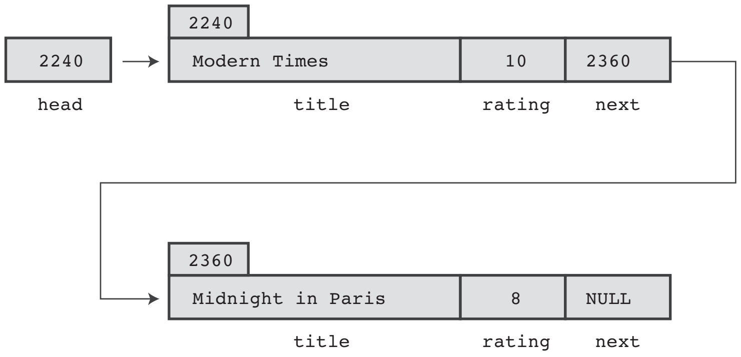


Figure 17.3 Linked list with two items.

Each new movie will be handled the same way. Its address will be stored in the preceding structure, the new information goes into the new structure, and its `next` member is set to `NULL`, setting up a linked list like that shown in Figure 17.4.

Suppose you want to display the list. Each time you display an item, you can use the address stored in the corresponding structure to locate the next item to be displayed. For this scheme to work, however, you need a pointer to keep track of the very first item in the list because no structure in the list stores the address of the first item. Fortunately, you've already accomplished this with the head pointer.

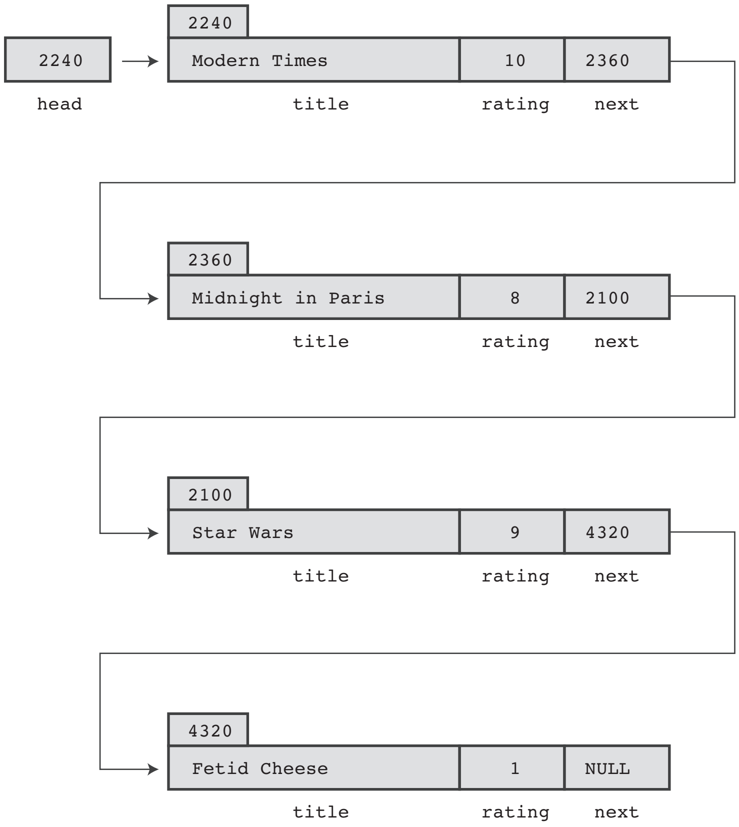


Figure 17.4 Linked list with several items.

Using a Linked List

Now that you have a picture of how a linked list works, let's implement it. Listing 17.2 modifies Listing 17.1 so that it uses a linked list instead of an array to hold the movie information.

Listing 17.2 **The `films2.c` Program**

```
/* films2.c -- using a linked list of structures */
#include <stdio.h>
#include <stdlib.h>      /* has the malloc prototype      */
#include <string.h>      /* has the strcpy prototype      */
#define TSIZE    45      /* size of array to hold title   */

struct film {
    char title[TSIZE];
    int rating;
    struct film * next;  /* points to next struct in list */
};
char * s_gets(char * st, int n);

int main(void)
{
    struct film * head = NULL;
    struct film * prev, * current;
    char input[TSIZE];

/* Gather  and store information          */
    puts("Enter first movie title:");
    while (s_gets(input, TSIZE) != NULL && input[0] != '\0')
    {
        current = (struct film *) malloc(sizeof(struct film));
        if (head == NULL)       /* first structure       */
            head = current;
        else                    /* subsequent structures */
            prev->next = current;
        current->next = NULL;
        strcpy(current->title, input);
        puts("Enter your rating <0-10>:");
        scanf("%d", &current->rating);
        while(getchar() != '\n')
            continue;
        puts("Enter next movie title (empty line to stop):");
        prev = current;
    }

/* Show list of movies                    */
    if (head == NULL)
        printf("No data entered. ");
    else
        printf ("Here is the movie list:\n");
    current = head;
    while (current != NULL)
    {
        printf("Movie: %s  Rating: %d\n",
               current->title, current->rating);
        current = current->next;
    }

/* Program done, so free allocated memory */
    current = head;
    while (current != NULL)
    {
        free(current);
        current = current->next;
    }
    printf("Bye!\n");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

The program performs two tasks using the linked list. First, it constructs the list and fills it with the incoming data. Second, it displays the list. Displaying is the simpler task, so let's look at it first.

Displaying a List

The idea is to begin by setting a pointer (call it `current`) to point to the first structure. Because the head pointer (call it `head`) already points there, this code suffices:

```
current = head;
```

Then you can use pointer notation to access the members of that structure:

```
printf("Movie: %s  Rating: %d\n", current->title, current->rating);
```

The next step is to reset the `current` pointer to point to the next structure in the list. That information is stored in the `next` member of the structure, so this code accomplishes the task:

```
current = current->next;
```

After this is accomplished, repeat the whole process. When the last item in the list is displayed, `current` will be set to `NULL`, because that's the value of the `next` member of the final structure.

You can use that fact to terminate the printing. Here's all the code films2.c uses to display the list:

```
while (current != NULL)
{
    printf("Movie: %s  Rating: %d\n", current->title, current->rating);
    current = current->next;
}
```

Why not just use head instead of creating a new pointer (current) to march through the list? Because using head would change the value of head, and the program would no longer have a way to find the beginning of the list.

Creating the List

Creating the list involves three steps:

1. Use malloc() to allocate enough space for a structure.
2. Store the address of the structure.
3. Copy the correct information into the structure.

There's no point in creating a structure if none is needed, so the program uses temporary storage (the input array) to get the user's choice for a movie name. If the user simulates EOF from the keyboard or enters an empty line, the input loop quits:

```
while (s_gets(input, TSIZE) != NULL && input[0] != '\0')
```

If there is input, the program requests space for a structure and assigns its address to the pointer variable current:

```
current = (struct film *) malloc(sizeof(struct film));
```

The address of the very first structure should be stored in the pointer variable head. The address of each subsequent structure should be stored in the next member of the structure that precedes it. Therefore, the program needs a way to know whether it's dealing with the first structure or not. A simple way is to initialize the head pointer to NULL when the program starts. Then the program can use the value of head to decide what to do:

```
if (head == NULL)       /* first structure       */
   head = current;
else                    /* subsequent structures */
   prev->next = current;
```

In this code, prev is a pointer that points to the structure allocated the previous time.

Next, you have to set the structure members to the proper values. In particular, you should set the next member to NULL to indicate that the current structure is the last one in the list. You should copy the film title from the `input` array to the `title` member, and you should get a value for the `rating` member. The following code does these things:

```
current->next = NULL;
strcpy(current->title, input);
puts("Enter your rating <0-10>:");
scanf("%d", &current->rating);
```

Because the call to `s_gets()` limits the input to `TSIZE - 1` characters, the string in the `input` array will fit into the `title` member, so it's safe to use `strcpy()`.

Finally, you should prepare the program for the next cycle of the input loop. In particular, you need to set `prev` to point to the current structure, because it will become the previous structure after the next movie name is entered and the next structure is allocated. The program sets this pointer at the end of the loop:

```
prev = current;
```

Does it work? Here is a sample run:

```
Enter first movie title:
Spirited Away
Enter your rating <0-10>:
9
Enter next movie title (empty line to stop):
The Duelists
Enter your rating <0-10>:
8
Enter next movie title (empty line to stop):
Devil Dog: The Mound of Hound
Enter your rating <0-10>:
1
Enter next movie title (empty line to stop):

Here is the movie list:
Movie: Spirited Away  Rating: 9
Movie: The Duelists  Rating: 8
Movie: Devil Dog: The Mound of Hound  Rating: 1
Bye!
```

Freeing List Memory

In many environments the program will free the memory used by `malloc()` when the program terminates, but it's best to get into the habit of balancing calls to `malloc()` with calls to `free()`. Therefore, the program cleans up its memory use by applying `free()` to each of the allocated structures:

```
current = head;
while (current != NULL)
```

```
{
    free(current);
    current = current->next;
}
```

Afterthoughts

The `films2.c` program is a bit skimpy. For example, it fails to check whether `malloc()` finds the requested memory, and it doesn't have any provisions for deleting items from the list. These failings can be fixed, however. For example, you can add code that checks whether `malloc()`'s return value is `NULL` (the sign it failed to obtain the memory you wanted). If the program needs to delete entries, you can write some more code to do that.

This ad hoc approach to solving problems and adding features as the need arises isn't always the best programming method. On the other hand, you usually can't anticipate everything a program needs to do. As programming projects get larger, the model of a programmer or programming team planning everything in advance becomes more and more unrealistic. It has been observed that the most successful large programs are those that evolved step-by-step from successful small programs.

Given that you may have to revise your plans, it's a good idea to develop your original ideas in a way that simplifies modification. The example in Listing 17.2 doesn't follow this precept. In particular, it tends to intermingle coding details and the conceptual model. For example, in the sample program, the conceptual model is that you add items to a list. The program obscures that interface by pushing details such as `malloc()` and the `current->next` pointer into the foreground. It would be nice if you could write a program in a way that made it obvious you're adding something to a list and in which bookkeeping details, such as calling memory-management functions and setting pointers, were hidden. Separating the user interface from the details will make the program easier to understand and to update. By making a fresh start, you can meet these targets. Let's see how.

Abstract Data Types (ADTs)

In programming, you try to match the data type to the needs of a programming problem. For example, you would use the `int` type to represent the number of shoes you own and the `float` or `double` type to represent your average cost per pair of shoes. In the movie examples, the data formed a list of items, each of which consisted of a movie name (a C string) and rating (an `int`). No basic C type matches that description, so we defined a structure to represent individual items, and then we devised a couple methods for tying together a series of structures to form a list. In essence, we used C's capabilities to design a new data type that matched our needs, but we did so unsystematically. Now we'll take a more systematic approach to defining types.

What constitutes a type? A *type* specifies two kinds of information: a set of properties and a set of operations. For example, the `int` type's property is that it represents an integer value and, therefore, shares the properties of integers. The allowed arithmetic operations are changing the sign, adding two `int`s, subtracting two `int`s, multiplying two `int`s, dividing one `int` by another, and taking the modulus of one `int` with respect to another. When you declare a variable to be an `int`, you're saying that these and only these operations can affect it.

Note Integer Properties

Behind the C `int` type is a more abstract concept, that of the *integer*. Mathematicians can, and do, define the properties of integers in a formal abstract manner. For example, if N and M are integers, N + M = M + N, or for every two integers N and M, there is an integer S, such that N + M = S. If N + M = S and if N + Q = S, then M = Q. You can think of mathematics as supplying the abstract concept of the integer and of C as supplying an implementation of that concept. For example, C provides a means of storing an integer and of performing integer operations such as addition and multiplication. Note that providing support for arithmetic operations is an essential part of representing integers. The `int` type would be much less useful if all you could do was store a value but not use it in arithmetic expressions. Note also that the implementation doesn't do a perfect job of representing integers. For example, there are an infinite number of integers, but a 2-byte `int` can represent only 65,536 of them; don't confuse the abstract idea with a particular implementation.

Suppose you want to define a new data type. First, you have to provide a way to store the data, perhaps by designing a structure. Second, you have to provide ways of manipulating the data. For example, consider the `films2.c` program (Listing 17.2). It has a linked set of structures to hold the information and supplies code for adding information and displaying information. This program, however, doesn't do these things in a way that makes it clear we were creating a new type. What should we have done?

Computer science has developed a very successful way to define new data types. It's a three-step process that moves from the abstract to the concrete:

1. Provide an abstract description of the type's properties and of the operations you can perform on the type. This description shouldn't be tied to any particular implementation. It shouldn't even be tied to a particular programming language. Such a formal abstract description is called an *abstract data type* (ADT).
2. Develop a programming interface that implements the ADT. That is, indicate how to store the data and describe a set of functions that perform the desired operations. In C, for example, you might supply a structure definition along with prototypes for functions to manipulate the structures. These functions play the same role for the user-defined type that C's built-in operators play for the fundamental C types. Someone who wants to use the new type will use this interface for her or his programming.
3. Write code to implement the interface. This step is essential, of course, but the programmer using the new type need not be aware of the details of the implementation.

Let's work through an example to see how this process works. Because we've already invested some effort into the movie listing example, let's redo it using the new approach.

Getting Abstract

Basically, all you need for the movie project is a list of items. Each item contains a movie name and a rating. You need to be able to add new items to the end of the list, and you need to be able to display the contents of the list. Let's call the abstract type that will handle these needs a *list*. What properties should a list have? Clearly, a list should be able to hold a sequence of items. That is, a list can hold several items, and these items are arranged in some kind of order, so you can speak of the first item in a list or of the second item or of the last item. Next, the list type should support operations such as adding an item to the list. Here are some useful operations:

- Initializing a list to empty
- Adding an item to the end of a list
- Determining whether the list is empty
- Determining whether the list is full
- Determining how many items are in the list
- Visiting each item in a list to perform some action, such as displaying the item

We don't need any further operations for this project, but a more general list of operations for lists might include the following:

- Inserting an item anywhere in the list
- Removing an item from the list
- Retrieving an item from the list (list left unaltered)
- Replacing one item in the list with another
- Searching for an item in the list

The informal, but abstract, definition of a list, then, is that it is a data object capable of holding a sequence of items and to which you can apply any of the preceding operations. This definition doesn't state what kind of items can be stored in the list. It doesn't specify whether an array or a linked set of structures or some other data form should be used to hold the items. It doesn't dictate what method to use, for example, to find the number of elements in a list. These matters are all details left to the implementation.

To keep the example simple, let's adopt a simplified list as the abstract data type, one that embodies only the features needed for the movie project. Here's a summary of the type:

Type Name:	Simple List
Type Properties:	Can hold a sequence of items.
Type Operations:	Initialize list to empty.
	Determine whether list is empty.
	Determine whether list is full.

Determine number of items in the list.

Add item to end of list.

Traverse list, processing each item in list.

Empty the list.

The next step is to develop a C-language interface for the simple list ADT.

Building an Interface

The interface for the simple list has two parts. The first part describes how the data will be represented, and the second part describes functions that implement the ADT operations. For example, there will be functions for adding an item to a list and for reporting the number of items in the list. The interface design should parallel the ADT description as closely as possible. Therefore, it should be expressed in terms of some general `Item` type instead of in terms of some specific type, such as `int` or `struct film`. One way to do this is to use C's `typedef` facility to define `Item` as the needed type:

```
#define TSIZE  45      /* size of array to hold title   */
struct film
{
    char title[TSIZE];
    int rating;
};

typedef struct film Item;
```

Then you can use the `Item` type for the rest of the definitions. If you later want a list of some other form of data, you can redefine the `Item` type and leave the rest of the interface definition unchanged.

Having defined `Item`, you now have to decide how to store items of that type. This step really belongs to the implementation stage, but making a decision now makes the example easier to follow. The linked structure approach worked pretty well in the `films2.c` program, so let's adapt it as shown here:

```
typedef struct node
{
    Item item;
   struct node * next;
} Node;
typedef Node * List;
```

In a linked list implementation, each link is called a *node*. Each node contains information that forms the contents of the list along with a pointer to the next node. To emphasize this terminology, we've used the tag name `node` for a node structure, and we've used `typedef` to make

`Node` the type name for a `struct node` structure. Finally, to manage a linked list, we need a pointer to its beginning, and we've used `typedef` to make `List` the name for a pointer of this type. Therefore, the declaration

```
List movies;
```

establishes `movies` as a pointer suitable for referring to a linked list.

Is this the only way to define the `List` type? No. For example, you could incorporate a variable to keep track of the number of entries:

```
typedef struct list
{
    Node * head;   /* pointer to head of list        */
    int size;      /* number of entries in list      */
} List;            /* alternative definition of list */
```

You could add a second pointer to keep track of the end of the list. Later, you'll see an example that does that. For now, let's stick to the first definition of a `List` type. The important point is that you should think of the declaration

```
List movies;
```

as establishing a list, not as establishing a pointer to a node or as establishing a structure. The exact data representation of `movies` is an implementation detail that should be invisible at the interface level.

For example, a program should initialize the head pointer to `NULL` when starting out, but you should not use code like this:

```
movies = NULL;
```

Why not? Because later you might find you like the structure implementation of a `List` type better, and that would require the following initializations:

```
movies.next = NULL;
movies.size = 0;
```

Anyone using the `List` type shouldn't have to worry about such details. Instead, they should be able do something along the following lines:

```
InitializeList(movies);
```

Programmers need to know only that they should use the `InitializeList()` function to initialize a list. They don't have to know the exact data implementation of a `List` variable. This is an example of *data hiding*, the art of concealing details of data representation from the higher levels of programming.

To guide the user, you can supply a function prototype along these lines:

```
/* operation:        initialize a list                  */
/* preconditions:    plist points to a list             */
/* postconditions:   the list is initialized to empty   */
void InitializeList(List * plist);
```

There are three points you should notice. First, the comments outline *preconditions*—that is, conditions that should hold before the function is called. Here, for example, you need a list to initialize. Second, the comments outline *postconditions*—that is, conditions that should hold after the function executes. Finally, the function uses a pointer to a list instead of a list as its argument, so this would be the function call:

```
InitializeList(&movies);
```

The reason is that C passes arguments by value, so the only way a C function can alter a variable in the calling program is by using a pointer to that variable. Here the restrictions of the language make the interface deviate slightly from the abstract description.

The C way to tie all the type and function information into a single package is to place the type definitions and function prototypes (including precondition and postcondition comments) in a header file. This file should supply all the information a programmer needs to use the type. Listing 17.3 shows a header file for the simple `list` type. It defines a particular structure as the `Item` type, and then it defines `Node` in terms of `Item` and it defines `List` in terms of `Node`. The functions representing list operations then use `Item` types and `List` types as arguments. If the function needs to modify an argument, it uses a pointer to the corresponding type instead of using the type directly. The file capitalizes each function name as a way of marking it as part of an interface package. Also, the file uses the `#ifndef` technique discussed in Chapter 16, "The C Preprocessor and the C Library," to protect against multiple inclusions of a file. If your compiler doesn't support the C99 `bool` type, you can replace

```
#include <stdbool.h>     /* C99 feature         */
```

with this in the header file:

```
enum bool {false, true}; /* define bool as type, false, true as values */
```

Listing 17.3 **The `list.h` Interface Header File**

```
/* list.h -- header file for a simple list type */
#ifndef LIST_H_
#define LIST_H_
#include <stdbool.h>     /* C99 feature         */

/* program-specific declarations */

#define TSIZE      45    /* size of array to hold title  */
struct film
{
    char title[TSIZE];
    int rating;
};
```

```
/* general type definitions */

typedef struct film Item;

typedef struct node
{
    Item item;
    struct node * next;
} Node;

typedef Node * List;

/* function prototypes */

/* operation:        initialize a list                           */
/* preconditions:    plist points to a list                      */
/* postconditions:   the list is initialized to empty            */
void InitializeList(List * plist);

/* operation:        determine if list is empty                  */
/*                   plist points to an initialized list         */
/* postconditions:   function returns True if list is empty      */
/*                   and returns False otherwise                 */
bool ListIsEmpty(const List *plist);

/* operation:        determine if list is full                   */
/*                   plist points to an initialized list         */
/* postconditions:   function returns True if list is full       */
/*                   and returns False otherwise                 */
bool ListIsFull(const List *plist);

/* operation:        determine number of items in list           */
/*                   plist points to an initialized list         */
/* postconditions:   function returns number of items in list    */
unsigned int ListItemCount(const List *plist);

/* operation:        add item to end of list                     */
/* preconditions:    item is an item to be added to list         */
/*                   plist points to an initialized list         */
/* postconditions:   if possible, function adds item to end      */
/*                   of list and returns True; otherwise the     */
/*                   function returns False                      */
bool AddItem(Item item, List * plist);

/* operation:        apply a function to each item in list       */
/*                   plist points to an initialized list         */
/*                    pfun points to a function that takes an    */
/*                    Item argument and has no return value      */
/* postcondition:     the function pointed to by pfun is         */
/*                    executed once for each item in the list    */
void Traverse (const List *plist, void (* pfun)(Item item) );

/* operation:         free allocated memory, if any              */
/*                    plist points to an initialized list        */
/* postconditions:    any memory allocated for the list is freed */
/*                    and the list is set to empty               */
void EmptyTheList(List * plist);

#endif
```

Only the `InitializeList()`, `AddItem()`, and `EmptyTheList()` functions modify the list, so, technically, they are the only methods requiring a pointer argument. However, it can get confusing if the user has to remember to pass a `List` argument to some functions and an address of a `List` as the argument to others. So, to simplify the user's responsibilities, all the functions use pointer arguments.

One of the prototypes in the header file is a bit more complex than the others:

```
/* operation:         apply a function to each item in list      */
/*                    plist points to an initialized list        */
/*                    pfun points to a function that takes an    */
/*                    Item argument and has no return value      */
/* postcondition:     the function pointed to by pfun is         */
/*                    executed once for each item in the list    */
void Traverse (const List *plist, void (* pfun)(Item item) );
```

The argument `pfun` is a pointer to a function. In particular, it is a pointer to a function that takes an `item` value as an argument and that has no return value. As you might recall from Chapter 14, "Structures and Other Data Forms," you can pass a pointer to a function as an argument to a second function, and the second function can then use the pointed-to function. Here, for example, you can let `pfun` point to a function that displays an item. The `Traverse()` function would then apply this function to each item in the list, thus displaying the whole list.

Using the Interface

Our claim is that you should be able to use this interface to write a program without knowing any further details—for example, without knowing how the functions are written. Let's write a new version of the movie program right now before we write the supporting functions. Because the interface is in terms of `List` and `Item` types, the program should be phrased in those terms. Here's a pseudocode representation of one possible plan:

```
Create a List variable.
Create an Item variable.
```

```
Initialize the list to empty.
While the list isn't full and while there's more input:
    Read the input into the Item variable.
    Add the item to the end of the list.
Visit each item in the list and display it.
```

The program shown in Listing 17.4 follows this basic plan, with some error-checking. Note how it makes use of the interface described in the `list.h` file (Listing 17.3). Also note that the listing has code for the `showmovies()` function, which conforms to the prototype required by `Traverse()`. Therefore, the program can pass the pointer `showmovies` to `Traverse()` so that `Traverse()` can apply the `showmovies()` function to each item in the list. (Recall that the name of a function is a pointer to the function.)

Listing 17.4 **The `films3.c` Program**

```
/* films3.c -- using an ADT-style linked list */
/* compile with list.c                        */
#include <stdio.h>
#include <stdlib.h>    /* prototype for exit() */
#include "list.h"      /* defines List, Item   */
void showmovies(Item item);
char * s_gets(char * st, int n);
int main(void)
{
    List movies;
    Item temp;


/* initialize       */
    InitializeList(&movies);
    if (ListIsFull(&movies))
    {
        fprintf(stderr,"No memory available! Bye!\n");
        exit(1);
    }

/* gather and store */
    puts("Enter first movie title:");
    while (s_gets(temp.title, TSIZE) != NULL && temp.title[0] != '\0')
    {
        puts("Enter your rating <0-10>:");
        scanf("%d", &temp.rating);
        while(getchar() != '\n')
            continue;
        if (AddItem(temp, &movies)==false)
        {
            fprintf(stderr,"Problem allocating memory\n");
            break;
        }
        if (ListIsFull(&movies))
        {
            puts("The list is now full.");
            break;
        }
        puts("Enter next movie title (empty line to stop):");
    }

/* display          */
    if (ListIsEmpty(&movies))
        printf("No data entered. ");
    else
    {
        printf ("Here is the movie list:\n");
        Traverse(&movies, showmovies);
    }
    printf("You entered %d movies.\n", ListItemCount(&movies));


/* clean up         */
    EmptyTheList(&movies);
    printf("Bye!\n");

    return 0;
}

void showmovies(Item item)
{
    printf("Movie: %s  Rating: %d\n", item.title,
            item.rating);
}
char * s_gets(char * st, int n)

    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
            continue;          // dispose of rest of line
    }
    return ret_val;
}
```

Implementing the Interface

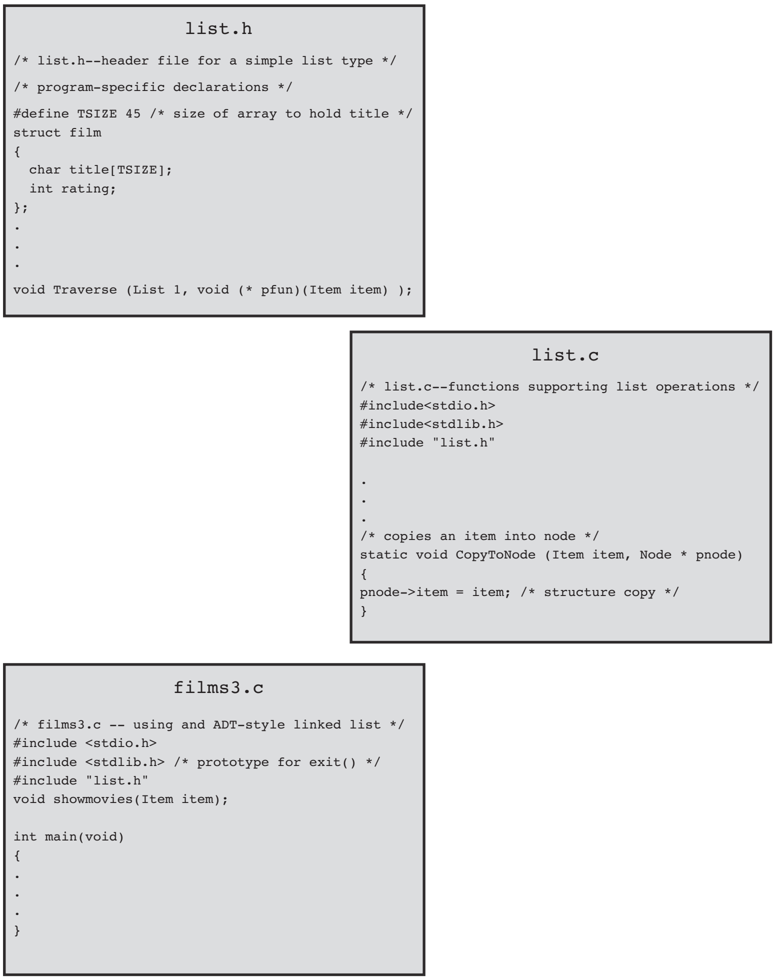
Of course, you still have to implement the `List` interface. The C approach is to collect the function definitions in a file called `list.c`. The complete program, then, consists of three files: `list.h`, which defines the data structures and provides prototypes for the user interface, `list.c`, which provides the function code to implement the interface, and `films3.c`, which is a source code file that applies the list interface to a particular programming problem. Listing 17.5 shows one possible implementation of `list.c`. To run the program, you must compile both `films3.c` and `list.c` and link them. (You might want to review the discussion in Chapter 9, "Functions," on compiling multiple-file programs.) Together, the files `list.h`, `list.c`, and `films3.c` constitute a complete program (see Figure 17.5).

Listing 17.5 **The `list.c` Implementation File**

```
/* list.c -- functions supporting list operations */
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

/* local function prototype */
static void CopyToNode(Item item, Node * pnode);

/* interface functions   */
/* set the list to empty */
void InitializeList(List * plist)
{
    * plist = NULL;
}

/* returns true if list is empty */
bool ListIsEmpty(const List * plist)
{
    if (*plist == NULL)
        return true;
    else
        return false;
}
```

```
/* returns true if list is full */
bool ListIsFull(const List * plist)
{
    Node * pt;
    bool full;

    pt = (Node *) malloc(sizeof(Node));
    if (pt == NULL)
        full = true;
    else
        full = false;
    free(pt);

    return full;
}

/* returns number of nodes */
unsigned int ListItemCount(const List * plist)
{
    unsigned int count = 0;
    Node * pnode = *plist;    /* set to start of list */

    while (pnode != NULL)
    {
        ++count;
        pnode = pnode->next;  /* set to next node     */
    }

    return count;
}

/* creates node to hold item and adds it to the end of */
/* the list pointed to by plist (slow implementation)  */
bool AddItem(Item item, List * plist)
{
    Node * pnew;
    Node * scan = *plist;

    pnew = (Node *) malloc(sizeof(Node));
    if (pnew == NULL)
        return false;     /* quit function on failure  */

    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (scan == NULL)          /* empty list, so place */
```

```
        *plist = pnew;         /* pnew at head of list */
    else
    {
        while (scan->next != NULL)
            scan = scan->next;  /* find end of list    */
        scan->next = pnew;      /* add pnew to end     */
    }

    return true;
}

/* visit each node and execute function pointed to by pfun */
void Traverse  (const List * plist, void (* pfun)(Item item) )
{
    Node * pnode = *plist;    /* set to start of list   */

    while (pnode != NULL)
    {
        (*pfun)(pnode->item); /* apply function to item */
        pnode = pnode->next;  /* advance to next item   */
    }
}

/* free memory allocated by malloc() */
/* set list pointer to NULL          */
void EmptyTheList(List * plist)
{
    Node * psave;

    while (*plist != NULL)
    {
        psave = (*plist)->next; /* save address of next node */
        free(*plist);           /* free current node         */
        *plist = psave;         /* advance to next node      */
    }
}

/* local function definition  */
/* copies an item into a node */
static void CopyToNode(Item item, Node * pnode)
{
    pnode->item = item;  /* structure copy */
}
```



list.h

```
/* list.h--header file for a simple list type */

/* program-specific declarations */

#define TSIZE 45 /* size of array to hold title */
struct film
{
  char title[TSIZE];
  int rating;
};
.
.
.
void Traverse (List 1, void (* pfun)(Item item) );
```

list.c

```
/* list.c--functions supporting list operations */
#include<stdio.h>
#include<stdlib.h>
#include "list.h"

.
.
.
/* copies an item into node */
static void CopyToNode (Item item, Node * pnode)
{
pnode->item = item; /* structure copy */
}
```

films3.c

```
/* films3.c -- using and ADT-style linked list */
#include <stdio.h>
#include <stdlib.h> /* prototype for exit() */
#include "list.h"
void showmovies(Item item);

int main(void)
{
.
.
.
}
```

Figure 17.5 The three parts of a program package.

Program Notes

The `list.c` file has many interesting points. For one, it illustrates when you might use functions with internal linkage. As described in Chapter 12, functions with internal linkage are known only in the file where they are defined. When implementing an interface, you might find it convenient sometimes to write auxiliary functions that aren't part of the official interface. For instance, the example uses the function `CopyToNode()` to copy a type `Item` value to a type `Item` variable. Because this function is part of the implementation but not part of the interface, we hid it in the `list.c` file by using the `static` storage class qualifier. Now, let's examine the other functions.

The `InitializeList()` function initializes a list to empty. In our implementation, that means setting a type `List` variable to `NULL`. As mentioned earlier, this requires passing a pointer to the `List` variable to the function.

The `ListIsEmpty()` function is quite simple, but it does depend on the list variable being set to `NULL` when the list is empty. Therefore, it's important to initialize a list before first using the `ListIsEmpty()` function. Also, if you were to extend the interface to include deleting items, you should make sure the deletion function resets the list to empty when the last item is deleted. With a linked list, the size of the list is limited by the amount of memory available. The `ListIsFull()` function tries to allocate enough space for a new item. If it fails, the list is full. If it succeeds, it has to free the memory it just allocated so that it is available for a real item.

The `ListItemCount()` function uses the usual linked-list algorithm to traverse the list, counting items as it goes:

```
unsigned int ListItemCount(const List * plist)
{
    unsigned int count = 0;
    Node * pnode = *plist;    /* set to start of list */

    while (pnode != NULL)
    {
        ++count;
        pnode = pnode->next;  /* set to next node     */
    }

    return count;
}
```

The `AddItem()` function is the most elaborate of the group:

```
bool AddItem(Item item, List * plist)
{
    Node * pnew;
    Node * scan = *plist;

    pnew = (Node *) malloc(sizeof(Node));
    if (pnew == NULL)
        return false;     /* quit function on failure  */

    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (scan == NULL)          /* empty list, so place */
        *plist = pnew;         /* pnew at head of list */
    else
    {
        while (scan->next != NULL)
```

```
            scan = scan->next;  /* find end of list    */
        scan->next = pnew;      /* add pnew to end     */
    }

    return true;
}
```

The first thing the `AddItem()` function does is allocate space for a new node. If this succeeds, the function uses `CopyToNode()` to copy the item to the node. Then it sets the `next` member of the node to `NULL`. This, as you'll recall, indicates that the node is the last node in the linked list. Finally, after creating the node and assigning the correct values to its members, the function attaches the node to the end of the list. If the item is the first item added to the list, the program sets the head pointer to the first item. (Remember, `AddItem()` is called with the address of the head pointer as its second argument, so `* plist` is the value of the head pointer.) Otherwise, the code marches through the linked list until it finds the item having its `next` member set to `NULL`. That node is currently the last node, so the function resets its `next` member to point to the new node.

Good programming practice dictates that you call `ListIsFull()` before trying to add an item to the list. However, a user might fail to observe this dictate, so `AddItem()` checks for itself whether `malloc()` has succeeded. Also, it's possible a user might do something else to allocate memory between calling `ListIsFull()` and calling `AddItem()`, so it's best to check whether `malloc()` worked.

The `Traverse()` function is similar to the `ListItemCount()` function with the addition of applying a function to each item in the list:

```
void Traverse  (const List * plist, void (* pfun)(Item item) )
{
    Node * pnode = *plist;    /* set to start of list   */

    while (pnode != NULL)
    {
        (*pfun)(pnode->item); /* apply function to item */
        pnode = pnode->next;  /* advance to next item   */
    }
}
```

Recall that `pnode->item` represents the data stored in a node and that `pnode->next` identifies the next node in the linked list. For example, the function call

```
Traverse(movies, showmovies);
```

applies the `showmovies()` function to each item in the list.

Finally, the `EmptyTheList()` function frees the memory previously allocated using `malloc()`:

```
void EmptyTheList(List * plist)
{
    Node * psave;

    while (*plist != NULL)
    {
        psave = (*plist)->next; /* save address of next node */
        free(*plist);           /* free current node         */
        *plist = psave;         /* advance to next node      */
    }
}
```

The implementation indicates an empty list by having the `List` variable being set to `NULL`. Therefore, this function needs to be passed the address of the `List` variable to be able to reset it. Because `List` already is a pointer, `plist` is a pointer to a pointer. Thus, within the code, the expression `*plist` is type pointer-to-`Node`. When the list terminates, `*plist` is `NULL`, meaning the original actual argument is now set to `NULL`.

The code saves the address of the next node because the call to `free()`, in principle, may make the contents of the current node (the one pointed to by `*plist`) no longer available.

Note The Limitations of `const`

Several of the list-handling functions have `const List * plist` for a parameter. This indicates the intent that these functions don't alter the list. Here, `const` does provide some protection. It prevents `*plist` (the quantity to which `plist` points) from being changed. In this program, `plist` points to `movies`, so `const` prevents those functions from changing `movies`, which, in turn, points to the first link in the list. Therefore, code such as this is not allowed in, say, `ListItemCount()`:

```
*plist = (*plist)->next;   // not allowed if *plist is const
```

This is good, because changing `*plist`, and, hence, `movies`, would cause the program to lose track of the data. However, the fact that `*plist` and `movies` are treated as `const` doesn't mean that data pointed to by `*plist` or `movies` is `const`. For example, code such as this is allowed:

```
(*plist)->item.rating = 3; // allowed even if *plist is const
```

That's because this code doesn't change `*plist`; it changes data that `*plist` points to. The moral is that you can't necessarily rely on `const` to catch programming errors that accidentally modify data.

Contemplating Your Work

Take a little time now to evaluate what the ADT approach has done for you. First, compare Listing 17.2 with Listing 17.4. Both programs use the same fundamental method (dynamic allocation of linked structures) to solve the movie listing problem, but Listing 17.2 exposes all the programming plumbing, putting `malloc()` and `prev->next` into public view. Listing 17.4, on the other hand, hides these details and expresses the program in a language that relates directly to the tasks. That is, it talks about creating a list and adding items to the list, not about calling memory functions or resetting pointers. In short, Listing 17.4 expresses the program in terms of the problem to be solved, not in terms of the low-level tools needed to solve the problem. The ADT version is oriented to the end user's concerns and is much easier to read.

Next, the `list.h` and `list.c` files together constitute a reusable resource. If you need another simple list, just haul out these files. Suppose you need to store an inventory of your relatives: names, relationships, addresses, and phone numbers. First, you would go to the `list.h` file and redefine the `Item` type:

```
typedef struct itemtag
{
   char fname[14];
   char lname [24];
   char relationship[36];
   char address [60];
   char phonenum[20];
} Item;
```

Next... well, that's all you have to do in this case because all the simple list functions are defined in terms of the `Item` type. In some cases, you would also have to redefine the `CopyToNode()` function. For example, if an item were an array, you couldn't copy it by assignment.

Another important point is that the user interface is defined in terms of abstract list operations, not in terms of some particular set of data representations and algorithms. This leaves you free to fiddle with the implementation without having to redo the final program. For example, the current `AddItem()` function is a bit inefficient because it always starts at the beginning of the list and then searches for the end. You can fix this problem by keeping track of the end of the list. For example, you can redefine the `List` type this way:

```
typedef struct list
{
    Node * head;      /* points to head of list */
    Node * end;       /* points to end of list  */
} List;
```

Of course, you would then have to rewrite the list-processing functions using this new definition, but you wouldn't have to change a thing in Listing 17.4. This sort of isolating implementation from the final interface is particularly useful for large programming projects. It's called *data hiding* because the detailed data representation is hidden from the final user.

Note that this particular ADT doesn't even force you to implement the simple list as a linked list. Here's another possibility:

```
#define MAXSIZE 100
typedef struct list
{
    Item entries[MAXSIZE];   /* array of items          */
    int items;              /* number of items in list */
} List;
```

Again, this would require rewriting the `list.c` file, but the program using the list doesn't need to be changed.

Finally, think of the benefits this approach provides for the program-development process. If something is not working right, you probably can localize the problem to a single function. If you think of a better way to do one of the tasks, such as adding an item, you just have to rewrite that one function. If you need a new feature, you can think in terms of adding a new function to the package. If you think that an array or double-linked list would be better, you can rewrite the implementation without having to modify the programs that use the implementation.

Getting Queued with an ADT

The abstract data type approach to programming in C, as you've seen, involves the following three steps:

1. Describing a type, including its operations, in an abstract, general fashion
2. Devising a function interface to represent the new type
3. Writing detailed code to implement the interface

You've seen this approach applied to a simple list. Now, apply it to something slightly more complex: the queue.

Defining the Queue Abstract Data Type

A *queue* is a list with two special properties. First, new items can be added only to the end of the list. In this respect, the queue is like the simple list. Second, items can be removed from the list only at the beginning. You can visualize a queue as a line of people buying tickets to a theater. You join the line at the end, and you leave the line at the front, after purchasing your tickets. A queue is a *first in, first out* (FIFO) data form, just the way a movie line is (if no one cuts into the line). Once again, let's frame an informal, abstract definition, as shown here:

Type Name:	Queue
Type Properties:	Can hold an ordered sequence of items.
Type Operations:	Initialize queue to empty.
	Determine whether queue is empty.
	Determine whether queue is full.
	Determine number of items in the queue.
	Add item to rear of queue.
	Remove and recover item from front of queue.
	Empty the queue.

Defining an Interface

The interface definition will go into a file called `queue.h`. We'll use C's `typedef` facility to create names for two types: `Item` and `Queue`. The exact implementation for the corresponding structures should be part of the `queue.h` file, but conceptually, designing the structures is part of the detailed implementation stage. For the moment, just assume that the types have been defined and concentrate on the function prototypes.

First, consider initialization. It involves altering a `Queue` type, so the function should take the address of a `Queue` as an argument:

```
void InitializeQueue (Queue * pq);
```

Next, determining whether the queue is empty or full involves a function that should return a true or false value. Here we assume that the C99 `stdbool.h` header file is available. If it's not, you can use type `int` or define a `bool` type yourself. Because the function doesn't alter the queue, it can take a `Queue` argument. On the other hand, it can be faster and less memory intensive to just pass the address of a `Queue`, depending on how large a `Queue`-type object is. Let's try that approach. Another advantage is that this way all the functions will take an address as an argument. To indicate that these functions don't change a queue, you can, and should, use the `const` qualifier:

```
bool QueueIsFull(const Queue * pq);
bool QueueIsEmpty (const Queue * pq);
```

Paraphrasing, the pointer `pq` points to a `Queue` data object that cannot be altered through the agency of `pq`. You can define a similar prototype for a function that returns the number of items in a queue:

```
int QueueItemCount(const Queue * pq);
```

Adding an item to the end of the queue involves identifying the item and the queue. This time the queue is altered, so using a pointer is necessary, not optional. The function could be type `void`, or you can use the return value to indicate whether the operation of adding an item succeeded. Let's take the second approach:

```
bool EnQueue(Item item, Queue * pq);
```

Finally, removing an item can be done several ways. If the item is defined as a structure or as one of the fundamental types, it could be returned by the function. The function argument could be either a `Queue` or a pointer to a `Queue`. Therefore, one possible prototype is this:

```
Item DeQueue(Queue q);
```

However, the following prototype is a bit more general:

```
bool DeQueue(Item * pitem, Queue * pq);
```

The item removed from the queue goes to the location pointed to by the `pitem` pointer, and the return value indicates whether the operation succeeded.

The only argument that should be needed for a function to empty the queue is the queue's address, suggesting this prototype:

```
void EmptyTheQueue(Queue * pq);
```

Implementing the Interface Data Representation

The first step is deciding what C data form to use for a queue. One possibility is an array. The advantages to arrays are that they're easy to use and that adding an item to the end of an array's filled portion is easy. The problem comes with removing an item from the front of the queue. In the analogy of people in a ticket line, removing an item from the front of the queue consists of copying the value of the first element of the array (simple) and then moving each item left in the array one element toward the front. Although this is easy to program, it wastes a lot of computer time (see Figure 17.6).

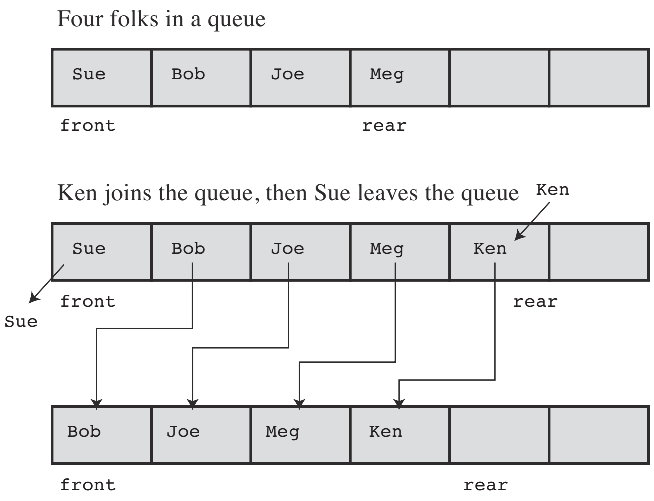


Figure 17.6 Using an array as a queue.

A second way to handle the removal problem in an array implementation is to leave the remaining elements where they are and, instead, change which element you call the front (see Figure 17.7). This method's problem is that the vacated elements become dead space, so the available space in the queue keeps decreasing.

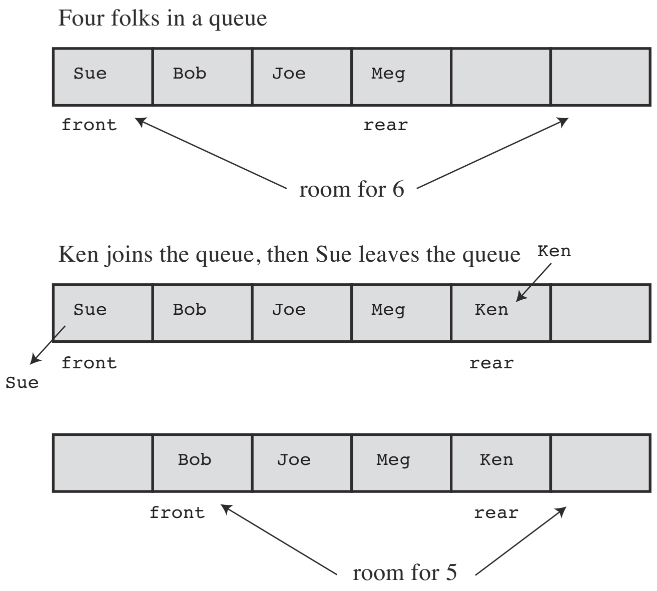


Figure 17.7 Redefining the front element.

A clever solution to the dead space problem is to make the queue *circular*. This means wrapping around from the end of the array to the beginning. That is, consider the first element of the array as immediately following the last element so that when you reach the end of the array, you can start adding items to the beginning elements if they have been vacated (see Figure 17.8). You can imagine drawing the array on a strip of paper, and then pasting one end of the array to the other to form a band. Of course, you now have to do some fancy bookkeeping to make sure the end of the queue doesn't pass the front.

Yet another solution is to use a linked list. This has the advantage that removing the front item doesn't require moving all the other items. Instead, you just reset the front pointer to point to the new first element. Because we've already been working with linked lists, we'll take this track. To test our ideas, we'll start with a queue of integers:

```
typedef int Item;
```

A linked list is built from nodes, so let's define a node next:

```
typedef struct node
{
    Item item;
    struct node * next;
} Node;
```

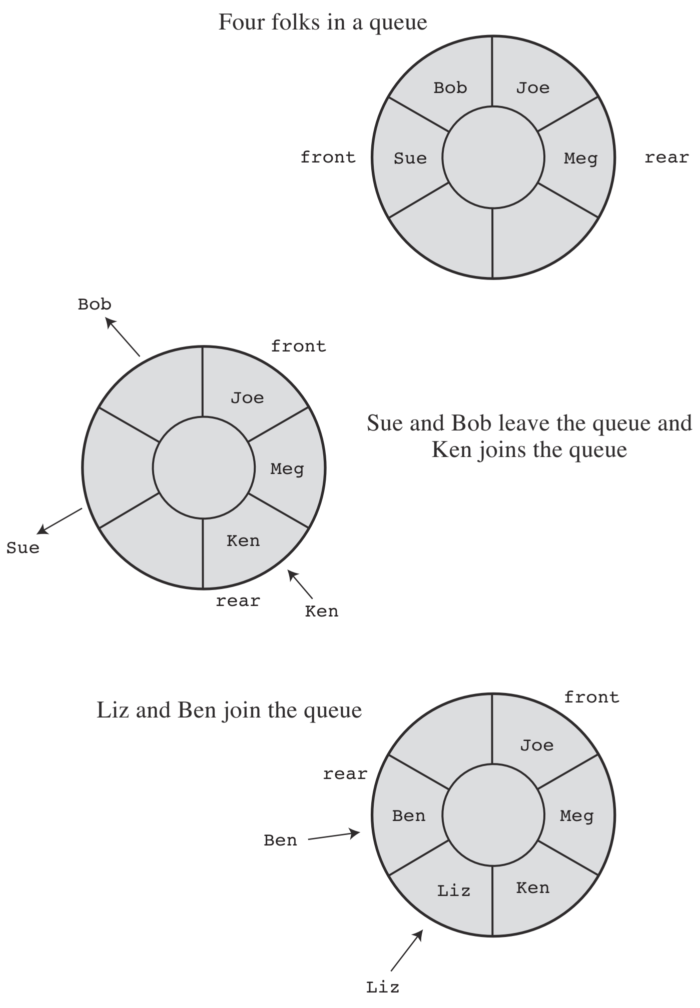

Four folks in a queue
Bob
Joe
front
Sue
Meg
rear
Bob
front
Joe
Meg
Sue and Bob leave the queue and Ken joins the queue
Sue
Ken
rear
Ken
front
Liz and Ben join the queue
Joe
rear
Ben
Meg
Ben
Liz
Ken
Liz
Circular queue wraps around to front of array

Figure 17.8 A circular queue.

For the queue, you need to keep track of the front and rear items. You can use pointers to do this. Also, you can use a counter to keep track of the number of items in a queue. Thus, the structure will have two pointer members and one type `int` member:

```
typedef struct queue
{
    Node * front;   /* pointer to front of queue */
    Node * rear;    /* pointer to rear of queue  */
    int items;      /* number of items in queue  */
} Queue;
```

Note that a `Queue` is a structure with three members, so the earlier decision to use pointers to queues instead of entire queues as arguments is a time and space saver.

Next, think about the size of a queue. With a linked list, the amount of available memory sets the limit, but often a much smaller size is more appropriate. For example, you might use a queue to simulate airplanes waiting to land at an airport. If the number of waiting planes gets too large, new arrivals might be rerouted to other airports. We'll set a maximum queue size of 10. Listing 17.6 contains the definitions and prototypes for the queue interface. It leaves open the exact definition of the `Item` type. When using the interface, you would insert the appropriate definition for your particular program.

Listing 17.6 **The `queue.h` Interface Header File**

```
/* queue.h -- interface for a queue */
#ifndef _QUEUE_H_
#define _QUEUE_H_
#include <stdbool.h>

/* INSERT ITEM TYPE HERE */
/* FOR EXAMPLE, */
typedef int Item;  // for use_q.c
/* OR typedef struct item {int gumption; int charisma;} Item; */

#define MAXQUEUE 10

typedef struct node
{
    Item item;
    struct node * next;
} Node;

typedef struct queue
{
    Node * front;  /* pointer to front of queue  */
    Node * rear;   /* pointer to rear of queue   */
    int items;     /* number of items in queue   */
} Queue;

/* operation:        initialize the queue                       */
/* precondition:     pq points to a queue                       */
/* postcondition:    queue is initialized to being empty        */
void InitializeQueue(Queue * pq);

/* operation:        check if queue is full                     */
/* precondition:     pq points to previously initialized queue  */
/* postcondition:   returns True if queue is full, else False   */
bool QueueIsFull(const Queue * pq);

/* operation:        check if queue is empty                    */
/* precondition:     pq points to previously initialized queue  */
/* postcondition:    returns True if queue is empty, else False */
bool QueueIsEmpty(const Queue *pq);

/* operation:        determine number of items in queue         */
/* precondition:     pq points to previously initialized queue  */
/* postcondition:    returns number of items in queue           */
int QueueItemCount(const Queue * pq);

/* operation:        add item to rear of queue                  */
/* precondition:     pq points to previously initialized queue  */
/*                   item is to be placed at rear of queue      */
/* postcondition:    if queue is not empty, item is placed at   */
/*                   rear of queue and function returns         */
/*                   True; otherwise, queue is unchanged and    */
/*                   function returns False                     */
bool EnQueue(Item item, Queue * pq);

/* operation:        remove item from front of queue            */
/* precondition:     pq points to previously initialized queue  */
/* postcondition:    if queue is not empty, item at head of     */
/*                   queue is copied to *pitem and deleted from */
/*                   queue, and function returns True; if the   */
/*                   operation empties the queue, the queue is  */
/*                   reset to empty. If the queue is empty to   */
/*                   begin with, queue is unchanged and the     */
/*                   function returns False                     */
bool DeQueue(Item *pitem, Queue * pq);

/* operation:        empty the queue                            */
/* precondition:     pq points to previously initialized queue  */
/* postconditions:   the queue is empty                         */
void EmptyTheQueue(Queue * pq);

#endif
```

Implementing the Interface Functions

Now we can get down to writing the interface code. First, initializing a queue to "empty" means setting the front and rear pointers to `NULL` and setting the item count (the `items` member) to `0`:

```
void InitializeQueue(Queue * pq)
{
    pq->front = pq->rear = NULL;
    pq->items = 0;
}
```

Next, the `items` member makes it easy to check for a full queue or empty queue and to return the number of items in a queue:

```
bool QueueIsFull(const Queue * pq)
{
    return pq->items == MAXQUEUE;
}

bool QueueIsEmpty(const Queue * pq)
{
    return pq->items == 0;
}

int QueueItemCount(const Queue * pq)
{
    return pq->items;
}
```

Adding an item to the queue involves the following steps:

1. Creating a new node.
2. Copying the item to the node.
3. Setting the node's `next` pointer to `NULL`, identifying the node as the last in the list.
4. Setting the current rear node's `next` pointer to point to the new node, linking the new node to the queue.
5. Setting the `rear` pointer to the new node, making it easy to find the last node.
6. Adding 1 to the item count.

Also, the function has to handle two special cases. First, if the queue is empty, the `front` pointer should be set to point to the new node. That's because when there is just one node, that node is both the front and the rear of the queue. Second, if the function is unable to obtain memory for the node, it should do something. Because we envision using small queues, such failure should be rare, so we'll simply have the function terminate the program if the program runs out of memory. Here's the code for `EnQueue()`:

```
bool EnQueue(Item item, Queue * pq)
{
    Node * pnew;
    if (QueueIsFull(pq))
        return false;
    pnew = (Node *) malloc( sizeof(Node));
    if (pnew == NULL)
    {
        fprintf(stderr,"Unable to allocate memory!\n");
        exit(1);
    }
    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (QueueIsEmpty(pq))
        pq->front = pnew;           /* item goes to front     */
    else
        pq->rear->next = pnew;      /* link at end of queue   */
    pq->rear = pnew;                /* record location of end */
    pq->items++;                    /* one more item in queue */

    return true;
}
```

The `CopyToNode()` function is a static function to handle copying the item to a node:

```
static void CopyToNode(Item item, Node * pn)
{
    pn->item = item;
}
```

Removing an item from the front of the queue involves the following steps:

1. Copying the item to a waiting variable
2. Freeing the memory used by the vacated node
3. Resetting the front pointer to the next item in the queue
4. Resetting the front and rear pointers to `NULL` if the last item is removed
5. Decrementing the item count

Here's code that does all these things:

```
bool DeQueue(Item * pitem, Queue * pq)
{
    Node * pt;

    if (QueueIsEmpty(pq))
        return false;
    CopyToItem(pq->front, pitem);
    pt = pq->front;
    pq->front = pq->front->next;
    free(pt);
    pq->items--;
    if (pq->items == 0)
        pq->rear = NULL;

    return true;
}
```

There are a couple of pointer facts you should note. First, the code doesn't explicitly set the `front` pointer to `NULL` when the last item is deleted. That's because it already sets the `front` pointer to the `next` pointer of the node being deleted. If that node is the last node, its `next` pointer is `NULL`, so the `front` pointer gets set to `NULL`. Second, the code uses a temporary pointer (`pt`) to keep track of the deleted node's location. That's because the official pointer to the first node (`pq->front`) gets reset to point to the next node, so without the temporary pointer, the program would lose track of which block of memory to free.

We can use the `DeQueue()` function to empty a queue. Just use a loop calling `DeQueue()` until the queue is empty:

```
void EmptyTheQueue(Queue * pq)
{
    Item dummy;
    while (!QueueIsEmpty(pq))
        DeQueue(&dummy, pq);
}
```

Note Keeping Your ADT Pure

After you've defined an ADT interface, you should use only the functions of the interface to handle the data type. Note, for example, that `Dequeue()` depends on the `EnQueue()` function doing its job of setting pointers correctly and setting the `next` pointer of the `rear` node to `NULL`. If, in a program using the ADT, you decided to manipulate parts of the queue directly, you might mess up the coordination between the functions in the interface package.

Listing 17.7 shows all the functions of the interface, including the `CopyToItem()` function used in `EnQueue()`.

Listing 17.7 **The `queue.c` Implementation File**

```
/* queue.c -- the Queue type implementation*/
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

/* local functions */
static void CopyToNode(Item item, Node * pn);
static void CopyToItem(Node * pn, Item * pi);

void InitializeQueue(Queue * pq)
{
    pq->front = pq->rear = NULL;
    pq->items = 0;
}

bool QueueIsFull(const Queue * pq)
{
    return pq->items == MAXQUEUE;
}

bool QueueIsEmpty(const Queue * pq)
{
    return pq->items == 0;
}

int QueueItemCount(const Queue * pq)
{
    return pq->items;
}

bool EnQueue(Item item, Queue * pq)
{
    Node * pnew;

    if (QueueIsFull(pq))
        return false;
    pnew = (Node *) malloc( sizeof(Node));
    if (pnew == NULL)
    {
        fprintf(stderr,"Unable to allocate memory!\n");
        exit(1);
    }
    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (QueueIsEmpty(pq))
        pq->front = pnew;           /* item goes to front     */
    else
        pq->rear->next = pnew;      /* link at end of queue   */
    pq->rear = pnew;                /* record location of end */
    pq->items++;                    /* one more item in queue */

    return true;
}
```

```
bool DeQueue(Item * pitem, Queue * pq)
{
    Node * pt;

    if (QueueIsEmpty(pq))
        return false;
    CopyToItem(pq->front, pitem);
    pt = pq->front;
    pq->front = pq->front->next;
    free(pt);
    pq->items--;
    if (pq->items == 0)
        pq->rear = NULL;

    return true;
}

/* empty the queue                */
void EmptyTheQueue(Queue * pq)
{
    Item dummy;
    while (!QueueIsEmpty(pq))
        DeQueue(&dummy, pq);
}

/* Local functions                 */

static void CopyToNode(Item item, Node * pn)
{
    pn->item = item;
}

static void CopyToItem(Node * pn, Item * pi)
{
    *pi = pn->item;
}
```

Testing the Queue

It's a good idea to test a new design, such as the queue package, before inserting it into a critical program. One approach to testing is writing a short program, sometimes called a *driver*, whose sole purpose is to test the package. For example, Listing 17.8 uses a queue that enables you to add and delete integers. Before using the program, make sure the following line is present in `queue.h`:

```
typedef int item;
```

Remember, too, that you have to link queue.c and use_q.c.

Listing 17.8 **The use_q.c Program**

```
/* use_q.c -- driver testing the Queue interface */
/* compile with queue.c                          */
#include <stdio.h>
#include "queue.h"  /* defines Queue, Item       */

int main(void)
{
    Queue line;
    Item temp;
    char ch;

    InitializeQueue(&line);
    puts("Testing the Queue interface. Type a to add a value,");
    puts("type d to delete a value, and type q to quit.");
    while ((ch = getchar()) != 'q')
    {
        if (ch != 'a' && ch != 'd')   /* ignore other input */
            continue;
        if ( ch == 'a')
        {
            printf("Integer to add: ");
            scanf("%d", &temp);
            if (!QueueIsFull(&line))
            {
                printf("Putting %d into queue\n", temp);
                EnQueue(temp,&line);
            }
           else
               puts("Queue is full!");
        }
        else
        {
            if (QueueIsEmpty(&line))
                puts("Nothing to delete!");
            else
            {
                 DeQueue(&temp,&line);
                 printf("Removing %d from queue\n", temp);
            }
        }
        printf("%d items in queue\n", QueueItemCount(&line));
        puts("Type a to add, d to delete, q to quit:");
    }
    EmptyTheQueue(&line);
    puts("Bye!");

    return 0;
}
```

Here is a sample run. You should also test to see that the implementation behaves correctly when the queue is full.

```
Testing the Queue interface. Type a to add a value,
type d to delete a value, and type q to quit.
a
Integer to add: 40
Putting 40 into queue
1 items in queue
Type a to add, d to delete, q to quit:
a
Integer to add: 20
Putting 20 into queue
2 items in queue
Type a to add, d to delete, q to quit:
a
Integer to add: 55
Putting 55 into queue
3 items in queue
Type a to add, d to delete, q to quit:
d
Removing 40 from queue
2 items in queue
Type a to add, d to delete, q to quit:
d
Removing 20 from queue
1 items in queue
Type a to add, d to delete, q to quit:
d
Removing 55 from queue
0 items in queue
Type a to add, d to delete, q to quit:
d
Nothing to delete!
0 items in queue
Type a to add, d to delete, q to quit:
q
Bye!
```

Simulating with a Queue

Well, the queue works! Now let's do something more interesting with it. Many real-life situations involve queues. For example, customers queue in banks and in supermarkets, airplanes queue at airports, and tasks queue in multitasking computer systems. You can use the queue package to simulate such situations.

Suppose, for example, that Sigmund Landers has set up an advice booth in a mall. Customers can purchase one, two, or three minutes of advice. To ensure a free flow of foot traffic, mall regulations limit the number of customers waiting in line to 10 (conveniently equal to the program's maximum queue size). Suppose people show up randomly and that the time they want to spend in consultation is spread randomly over the three choices (one, two, or three minutes). How many customers, on average, will Sigmund handle in an hour? How long, on average, will customers have to wait? How long, on average, will the line be? These are the sort of questions a queue simulation can answer.

First, let's decide what to put in the queue. You can describe each customer in terms of the time when he or she joins the queue and in terms of how many minutes of consultation he or she wants. This suggests the following definition for the `Item` type:

```
typedef struct item
{
    long arrive;      /* the time when a customer joins the queue   */
    int processtime;  /* the number of consultation minutes desired */
} Item;
```

To convert the queue package to handle this structure, instead of the `int` type the last example used, all you have to do is replace the former `typedef` for `Item` with the one shown here. After that's done, you don't have to worry about the detailed mechanics of a queue. Instead, you can proceed to the real problem—simulating Sigmund's waiting line.

Here's one approach. Let time move in one-minute increments. Each minute, check to see whether a new customer has arrived. If a customer arrives and the queue isn't full, add the customer to the queue. This involves recording in an `Item` structure the customer's arrival time and the amount of consultation time the customer wants, and then adding the item to the queue. If the queue is full, however, turn the customer away. For bookkeeping, keep track of the total number of customers and the total number of "turnaways" (people who can't get in line because it is full).

Next, process the front of the queue. That is, if the queue isn't empty and if Sigmund isn't occupied with a previous customer, remove the item at the front of the queue. The item, recall, contains the time when the customer joined the queue. By comparing this time with the current time, you get the number of minutes the customer has been in the queue. The item also contains the number of consultation minutes the customer wants, which determines how long Sigmund will be occupied with the new customer. Use a variable to keep track of this waiting time. If Sigmund is busy, no one is "dequeued." However, the variable keeping track of the waiting time should be decremented.

The core code can look like this, with each cycle corresponding to one minute of activity:

```
for (cycle = 0; cycle < cyclelimit; cycle++)
{
    if (newcustomer(min_per_cust))
    {
        if (QueueIsFull(&line))
            turnaways++;
        else
        {
            customers++;
            temp = customertime(cycle);
            EnQueue(temp, &line);
        }
    }
    if (wait_time <= 0 && !QueueIsEmpty(&line))
    {
        DeQueue (&temp, &line);
        wait_time = temp.processtime;
        line_wait += cycle - temp.arrive;
        served++;
    }
    if (wait_time > 0)
        wait_time––;
    sum_line += QueueItemCount(&line);
}
```

Note that the time resolution is relatively coarse (one minute) so that the maximum number of customers per hour is just 60.

Here are the meanings of some of the variables and functions:

- `min_per_cust` is the average number of minutes between customer arrivals.
- `newcustomer()` uses the C `rand()` function to determine whether a customer shows up during this particular minute.
- `turnaways` is the number of arrivals turned away.
- `customers` is the number of arrivals who join the queue.
- `temp` is an `Item` variable describing the new customer.
- `customertime()` sets the `arrive` and `processtime` members of the `temp` structure.
- `wait_time` is the number of minutes remaining until Sigmund finishes with the current client.
- `line_wait` is the cumulative time spent in line by all customers to date.
- `served` is the number of clients actually served.
- `sum_line` is the cumulative length of the line to date.

Think of how much messier and more obscure this code would look if it were sprinkled with `malloc()` and `free()` functions and pointers to nodes. Having the queue package enables you to concentrate on the simulation problem, not on programming details.

Listing 17.9 shows the complete code for the mall advice booth simulation. It uses the standard `rand()`, `srand()`, and `time()` functions to generate random values, following the method suggested in Chapter 12. To use the program, remember to update the `Item` definition in `queue.h` with the following:

```
typedef struct item
{
    long arrive;      // the time when a customer joins the queue
    int processtime;  // the number of consultation minutes desired
} Item;
```

Also remember to link the code for `mall.c` with `queue.c`.

Listing 17.9 **The `mall.c` Program**

```
// mall.c -- use the Queue interface
// compile with queue.c
#include <stdio.h>
#include <stdlib.h>    // for rand() and srand()
#include <time.h>      // for time()
#include "queue.h"     // change Item typedef
#define MIN_PER_HR 60.0

bool newcustomer(double x);   // is there a new customer?
Item customertime(long when); // set customer parameters

int main(void)
{
    Queue line;
    Item temp;                // new customer data
    int hours;                // hours of simulation
    int perhour;              // average # of arrivals per hour
    long cycle, cyclelimit;   // loop counter, limit
    long turnaways = 0;       // turned away by full queue
    long customers = 0;       // joined the queue
    long served = 0;          // served during the simulation
    long sum_line = 0;        // cumulative line length
    int wait_time = 0;        // time until Sigmund is free
    double min_per_cust;      // average time between arrivals
    long line_wait = 0;       // cumulative time in line

    InitializeQueue(&line);
    srand((unsigned int) time(0)); // random initializing of rand()
```

```
puts("Case Study: Sigmund Lander's Advice Booth");
puts("Enter the number of simulation hours:");
scanf("%d", &hours);
cyclelimit = MIN_PER_HR * hours;
puts("Enter the average number of customers per hour:");
scanf("%d", &perhour);
min_per_cust = MIN_PER_HR / perhour;

for (cycle = 0; cycle < cyclelimit; cycle++)
{
    if (newcustomer(min_per_cust))
    {
        if (QueueIsFull(&line))
            turnaways++;
        else
        {
            customers++;
            temp = customertime(cycle);
            EnQueue(temp, &line);
        }
    }
    if (wait_time <= 0 && !QueueIsEmpty(&line))
    {
        DeQueue (&temp, &line);
        wait_time = temp.processtime;
        line_wait += cycle - temp.arrive;
        served++;
    }
    if (wait_time > 0)
        wait_time--;
    sum_line += QueueItemCount(&line);
}

if (customers > 0)
{
    printf("customers accepted: %ld\n", customers);
    printf("  customers served: %ld\n", served);
    printf("       turnaways: %ld\n", turnaways);
    printf("average queue size: %.2f\n",
           (double) sum_line / cyclelimit);
    printf(" average wait time: %.2f minutes\n",
           (double) line_wait / served);
}
else
    puts("No customers!");
EmptyTheQueue(&line);
puts("Bye!");
```

```
    return 0;
}

// x = average time, in minutes, between customers
// return value is true if customer shows up this minute
bool newcustomer(double x)
{
    if (rand() * x / RAND_MAX < 1)
        return true;
    else
        return false;
}

// when is the time at which the customer arrives
// function returns an Item structure with the arrival time
// set to when and the processing time set to a random value
// in the range 1 - 3
Item customertime(long when)
{
    Item cust;

    cust.processtime = rand() % 3 + 1;
    cust.arrive = when;

    return cust;
}
```

The program enables you to specify the number of hours to simulate and the average number of customers per hour. Choosing a large number of hours gives you good average values, and choosing a small number of hours shows the sort of random variation you can get from hour to hour. The following runs illustrate these points. Note that the average queue sizes and wait times for 80 hours are about the same as for 800 hours, but that the two one-hour samples differ quite a bit from each other and from the long-term averages. That's because smaller statistical samples tend to have larger relative variations.

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
80
Enter the average number of customers per hour:
20
customers accepted: 1633
  customers served: 1633
       turnaways: 0
average queue size: 0.46
average wait time: 1.35 minutes
```

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
800
Enter the average number of customers per hour:
20
customers accepted: 16020
  customers served: 16019
       turnaways: 0
average queue size: 0.44
average wait time: 1.32 minutes

Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
1
Enter the average number of customers per hour:
20
customers accepted: 20
  customers served: 20
       turnaways: 0
average queue size: 0.23
average wait time: 0.70 minutes

Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
1
Enter the average number of customers per hour:
20
customers accepted: 22
  customers served: 22
       turnaways: 0
average queue size: 0.75
average wait time: 2.05 minutes
```

Another way to use the program is to keep the numbers of hours constant but to try different average numbers of customers per hour. Here are two sample runs exploring this variation:

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
80
Enter the average number of customers per hour:
25
customers accepted: 1960
  customers served: 1959
       turnaways: 3
average queue size: 1.43
average wait time: 3.50 minutes
```

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
80
Enter the average number of customers per hour:
30
customers accepted: 2376
  customers served: 2373
       turnaways: 94
average queue size: 5.85
average wait time: 11.83 minutes
```

Note how the average wait time takes a sharp upturn as the frequency of customers increases. The average wait for 20 customers per hour (80-hour simulation) was 1.35 minutes. It climbs to 3.50 minutes at 25 customers per hour and soars to 11.83 minutes at 30 customers an hour. Also, the number of turnaways climbs from 0 to 3 to 94. Sigmund could use this sort of analysis to decide whether he needs a second booth.

The Linked List Versus the Array

Many programming problems, such as creating a list or a queue, can be handled with a linked list—by which we mean a linked sequence of dynamically allocated structures—or with an array. Each form has its strengths and weaknesses, so the choice of which to use depends on the particular requirements of a problem. Table 17.1 summarizes the qualities of linked lists and arrays.

Table 17.1 **Comparing Arrays to Linked Lists**

Data Form	Pros	Cons
Array	Directly supported by C. Provides random access. at compile time.	Size determined Inserting and deleting elements is time consuming
Linked list	Size determined during runtime. Inserting and deleting elements is quick.	No random access. User must provide programming support.

Take a closer look at the process of inserting and deleting elements. To insert an element in an array, you have to move elements to make way for the new element, as shown in Figure 17.9. The closer to the front the new element goes, the more elements have to be moved. To insert a node in a linked list, however, you just have to assign values to two pointers, as shown in Figure 17.10. Similarly, removing an element from an array involves a wholesale relocation of elements, but removing a node from a linked list involves resetting a pointer and freeing the memory used by the deleted node.

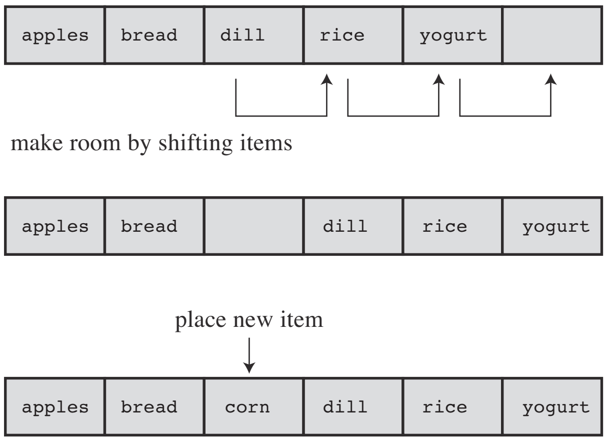


Figure 17.9 Inserting an element into an array.

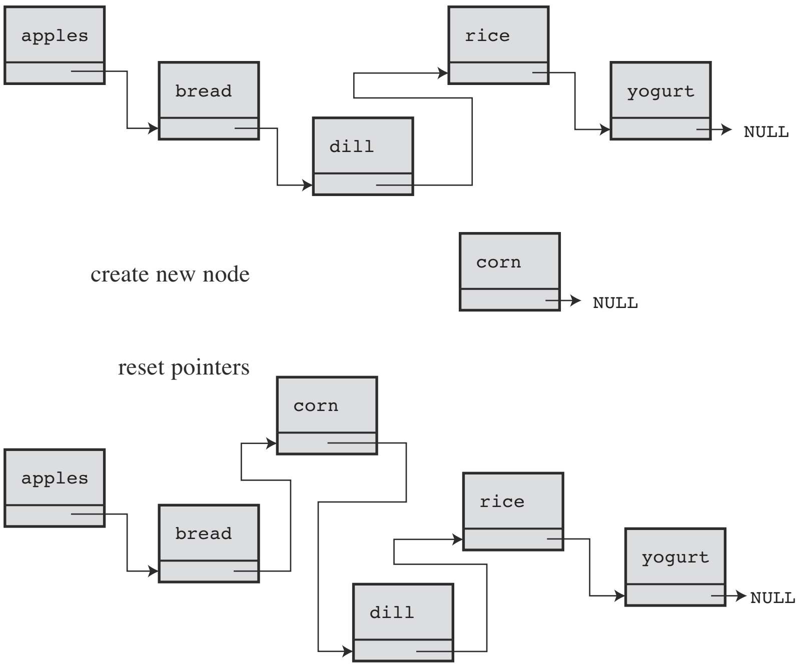


Figure 17.10 Inserting an element into a linked list.

Next, consider how to access the members of a list. With an array, you can use the array index to access any element immediately. This is called *random access*. With a linked list, you have to start at the top of the list and then move from node to node until you get to the node you want, which is termed *sequential access*. You can have sequential access with an array, too. Just increment the array index by one step each to move through the array in order. For some situations, sequential access is sufficient. For example, if you want to display every item in a list, sequential access is fine. Other situations greatly favor random access, as you will see next.

Suppose you want to search a list for a particular item. One algorithm is to start at the beginning of the list and search through it in sequence, called a *sequential search*. If the items aren't arranged in some sort of order, a sequential search is about all you can do. If the sought-for item isn't in the list, you'll have to look at every item in the list before concluding the item isn't there. (Concurrent programming could help here, as different CPUs could search different parts of the list simultaneously.)

You can improve the sequential search by sorting the list first. That way, you can terminate a search if you haven't found an item by the time you reach an item that would come later. For example, suppose you're seeking *Susan* in an alphabetical list. Starting from the top of the list, you look at each item and eventually encounter *Sylvia* without finding *Susan*. At that point you can quit searching because *Susan*, if in the list, would precede *Sylvia*. On average, this method would cut search times in half for attempting to find items not in the list.

With an ordered list, you can do much better than a sequential search by using the *binary search* method. Here's how it works. First, call the list item you want to find the *target* and assume the list is in alphabetical order. Next, pick the item halfway down the list and compare it to the target. If the two are the same, the search is over. If the list item comes before the target alphabetically, the target, if it's in the list, must be in the second half. If the list item follows the target alphabetically, the target must be in the first half. Either way, the comparison rules out half the list as a place to search. Next, apply the method again. That is, choose an item midway in the half of the list that remains. Again, this method either finds the item or rules out half the remaining list. Proceed in this fashion until you find the item or until you've eliminated the whole list (see Figure 17.11). This method is quite efficient. Suppose, for example, that the list is 127 items long. A sequential search, on the average, would take 64 comparisons before finding an item or ruling out its presence. The binary search method, on the other hand, will take at most seven comparisons. The first comparison prunes the possible matches to 63, the second comparison cuts the possible matches to 31, and so on, until the sixth comparison cuts down the possibilities to 1. The seventh comparison then determines whether the one remaining choice is the target. In general, *n* comparisons let you process an array with 2^n-1 members, so the advantage of a binary search over a sequential search gets greater the longer the list is.

It's simple to implement a binary search with an array, because you can use the array index to determine the midpoint of any list or subdivision of a list. Add the subscripts of the initial and final elements of the subdivision and divide by 2. For example, in a list of 100 elements, the first index is 0, the final index is 99, and the initial guess would be (0 + 99) / 2, or 49 (integer division). If the element having index 49 were too far down the alphabet, the correct choice must be in the range 0–48, so the next guess would be (0 + 48) / 2, or 24. If element 24 were too early in the alphabet, the next guess would be (25 + 48) / 2, or 36. This is where the random access feature of the array comes into play. It enables you to jump from one location to another without visiting every location in between. Linked lists, which support only sequential access, don't provide a means to jump to the midpoint of a list, so you can't use the binary search technique with linked lists.

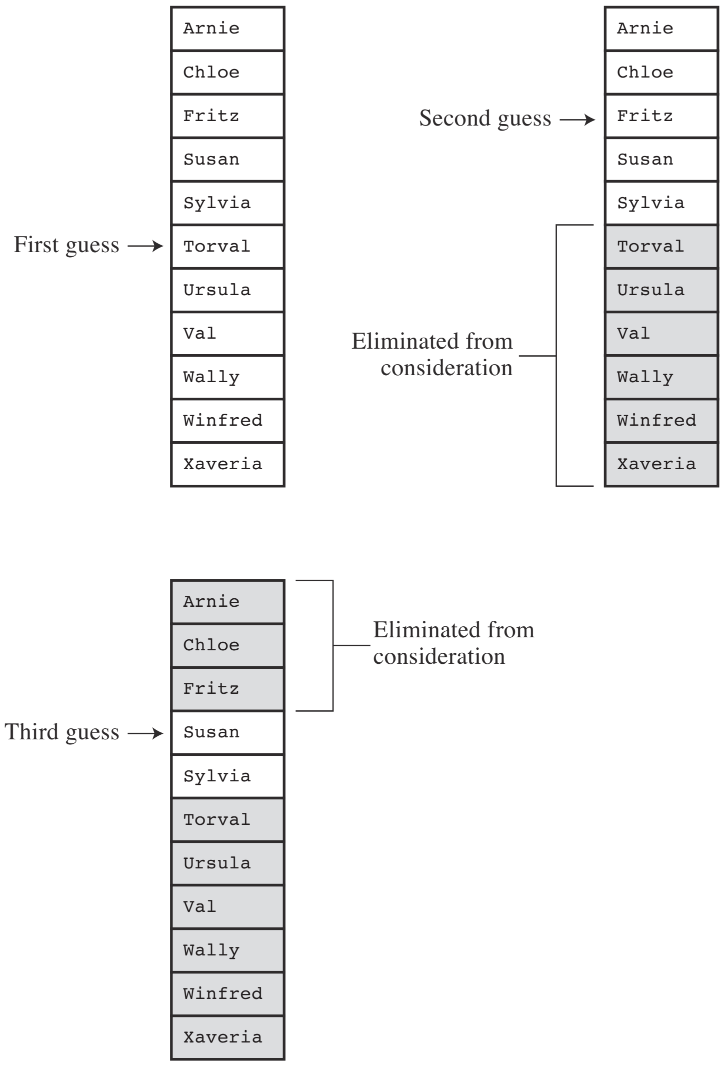


Figure 17.11 A binary search for Susan.

You can see, then, that the choice of data type depends on the problem. If the situation calls for a list that is continuously resized with frequent insertions and deletions but that isn't searched often, the linked list is the better choice. If the situation calls for a stable list with only occasional insertions and deletions but that has to be searched often, an array is the better choice.

What if you need a data form that supports frequent insertions and deletions and frequent searches? Neither a linked list nor an array is ideal for that set of purposes. Another form—the binary search tree—may be just what you need.

Binary Search Trees

The *binary search tree* is a linked structure that incorporates the binary search strategy. Each node in the tree contains an item and two pointers to other nodes, called *child nodes*. Figure 17.12 shows how the nodes in a binary search tree are linked. The idea is that each node has two child nodes—a left node and a right node. The ordering comes from the fact that the item in a left node precedes the item in the parent node, and the item in the right node follows the item in the parent node. This relationship holds for every node with children. Furthermore, all items that can trace their ancestry back to a left node of a parent contain items that precede the parent item in order, and every item descended from the right node contains items that follow the parent item in order. The tree in Figure 17.12 stores words in this fashion. The top of the tree, in an interesting inversion of botany, is called the *root*. A tree is a *hierarchical* organization, meaning that the data is organized in ranks, or levels, with each rank, in general, having ranks above and below it. If a binary search tree is fully populated, each level has twice as many nodes as the level above it.

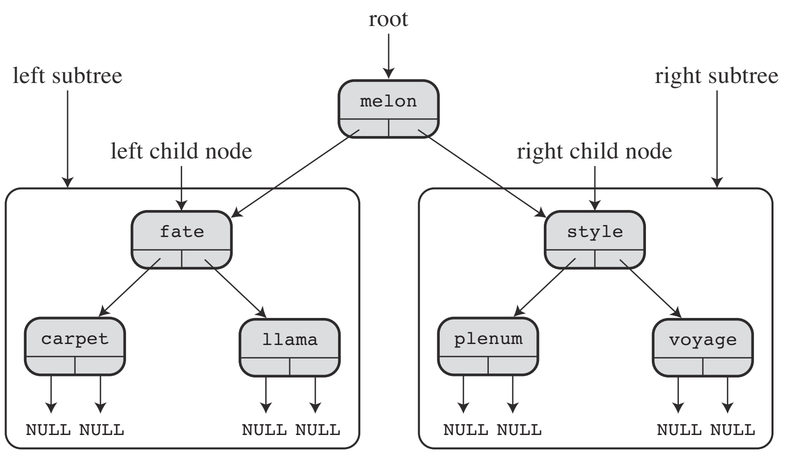


Figure 17.12 A binary search tree storing words.

Each node in the binary search tree is itself the root of the nodes descending from it, making the node and its descendants a *subtree*. In Figure 17.12, for example, the nodes containing the words *fate*, *carpet*, and *llama* form the left subtree of the whole tree, and the word *voyage* is the right subtree of the *style-plenum-voyage* subtree.

Suppose you want to find an item—call it the *target*—in such a tree. If the item precedes the root item, you need to search only the left half of the tree, and if the target follows the root item, you need to search only the right subtree of the root node. Therefore, one comparison eliminates half the tree. Suppose you search the left half. That means comparing the target with the item in the left child. If the target precedes the left-child item, you need to search only the left half of its descendants, and so on. As with the binary search, each comparison cuts the number of potential matches in half.

Let's apply this method to see whether the word *puppy* is in the tree shown in Figure 17.12. Comparing *puppy* to *melon* (the root node item), you see that *puppy*, if present, must be in the right half of the tree. Therefore, you go to the right child and compare *puppy* to *style*. In this case, *puppy* precedes the node item, so you must follow the link to the left node. There you find *plenum*, which precedes *puppy*. You now have to follow the right branch for that node, but it is empty, so three comparisons show you that *puppy* is not in the tree.

A binary search tree, then, combines a linked structure with binary search efficiency. The programming price is that putting a tree together is more involved than creating a linked list. Let's make a binary tree for the next, and final, ADT project.

A Binary Tree ADT

As usual, we'll start by defining a binary tree in general terms. This particular definition assumes the tree contains no duplicate items. Many of the operations are the same as list operations. The difference is in the hierarchical arrangement of data. Here is an informal summary of this ADT:

Type Name:	Binary Search Tree
Type Properties:	A binary tree is either an empty set of nodes (an empty tree) or a set of nodes with one node designated the root.
	Each node has exactly two trees, called the *left subtree* and the *right subtree*, descending from it.
	Each subtree is itself a binary tree, which includes the possibility of being an empty tree.
	A binary search tree is an ordered binary tree in which each node contains an item, in which all items in the left subtree precede the root item, and in which the root item precedes all items in the right subtree.
Type Operations:	Initializing tree to empty.
	Determining whether tree is empty.
	Determining whether tree is full.

Determining the number of items in the tree.

Adding an item to the tree.

Removing an item from the tree.

Searching the tree for an item.

Visiting each item in the tree.

Emptying the tree.

The Binary Search Tree Interface

In principle, you can implement a binary search tree in a variety of ways. You can even implement one as an array by manipulating array indices. But the most direct way to implement a binary search tree is by using dynamically allocated nodes linked together by using pointers, so we'll start with definitions like these:

```
typedef SOMETHING Item;

typedef struct trnode
{
    Item item;
    struct trnode * left;
    struct trnode * right;
} Trn;

typedef struct tree
{
    Trnode * root;
    int size;
} Tree;
```

Each node contains an item, a pointer to the left child node, and a pointer to the right child node. You could define a `Tree` to be type pointer-to-`Trnode`, because you only need to know the location of the root node to access the entire tree. Using a structure with a size member, however, makes it simpler to keep track of the size of the tree.

The example we'll be developing is maintaining the roster of the Nerfville Pet Club, with each item consisting of a pet name and a pet kind. With that in mind, we can set up the interface shown in Listing 17.10. We've limited the tree size to 10. The small size makes it easier to test whether the program behaves correctly when the tree fills. You can always set `MAXITEMS` to a larger value, if necessary.

Listing 17.10 **The `tree.h` Interface Header File**

```
/* tree.h -- binary search tree                          */
/*           no duplicate items are allowed in this tree */
```

```
#ifndef _TREE_H_
#define _TREE_H_
#include <stdbool.h>

/* redefine Item as appropriate */
typedef struct item
{
    char petname[20];
    char petkind[20];
} Item;

#define MAXITEMS 10

typedef struct trnode
{
    Item item;
    struct trnode * left;  /* pointer to right branch  */
    struct trnode * right; /* pointer to left branch   */
} Trnode;

typedef struct tree
{
    Trnode * root;         /* pointer to root of tree  */
    int size;              /* number of items in tree  */
} Tree;

/* function prototypes */

/* operation:      initialize a tree to empty          */
/* preconditions:  ptree points to a tree              */
/* postconditions: the tree is initialized to empty    */
void InitializeTree(Tree * ptree);

/* operation:      determine if tree is empty          */
/* preconditions:  ptree points to a tree              */
/* postconditions: function returns true if tree is    */
/*                 empty and returns false otherwise   */
bool TreeIsEmpty(const Tree * ptree);

/* operation:      determine if tree is full           */
/* preconditions:  ptree points to a tree              */
/* postconditions: function returns true if tree is    */
/*                 full and returns false otherwise    */
bool TreeIsFull(const Tree * ptree);

/* operation:      determine number of items in tree   */
/* preconditions:  ptree points to a tree              */
/* postconditions: function returns number of items in */
/*                 tree                                 */
int TreeItemCount(const Tree * ptree);

/* operation:      add an item to a tree                */
/* preconditions:  pi is address of item to be added    */
/*                 ptree points to an initialized tree */
/* postconditions: if possible, function adds item to  */
/*                 tree and returns true; otherwise,   */
/*                 the function returns false          */
bool AddItem(const Item * pi, Tree * ptree);

/* operation: find an item in a tree                    */
/* preconditions:  pi points to an item                 */
/*                 ptree points to an initialized tree */
/* postconditions: function returns true if item is in */
/*                 tree and returns false otherwise    */
bool InTree(const Item * pi, const Tree * ptree);

/* operation:      delete an item from a tree           */
/* preconditions:  pi is address of item to be deleted */
/*                 ptree points to an initialized tree */
/* postconditions: if possible, function deletes item  */
/*                 from tree and returns true;         */
/*                 otherwise the function returns false*/
bool DeleteItem(const Item * pi, Tree * ptree);

/* operation:      apply a function to each item in    */
/*                 the tree                            */
/* preconditions:  ptree points to a tree              */
/*                 pfun points to a function that takes*/
/*                 an Item argument and has no return  */
/*                 value                               */
/* postcondition:  the function pointed to by pfun is  */
/*                 executed once for each item in tree */
void Traverse (const Tree * ptree, void (* pfun)(Item item));

/* operation:      delete everything from a tree       */
/* preconditions:  ptree points to an initialized tree */
/* postconditions: tree is empty                       */
void DeleteAll(Tree * ptree);

#endif
```

The Binary Tree Implementation

Next, proceed to the task of implementing the splendid functions outlined in `tree.h`. The `InitializeTree()`, `EmptyTree()`, `FullTree()`, and `TreeItems()` functions are pretty simple, working like their counterparts for the list and queue ADTs, so we'll concentrate on the remaining ones.

Adding an Item

When adding an item to the tree, you should first check whether the tree has room for a new node. Then, because the binary search tree is defined so that it has no duplicate items, you should check that the item is not already in the tree. If the new item clears these first two hurdles, you create a new node, copy the item to the node, and set the node's left and right pointers to `NULL`. This indicates that the node has no children. Then you should update the `size` member of the `Tree` structure to mark the adding of a new item. Next, you have to find where the node should be located in the tree. If the tree is empty, you should set the root pointer to point to the new node. Otherwise, look through the tree for a place to add the node. The `AddItem()` function follows this recipe, offloading some of the work to functions not yet defined: `SeekItem()`, `MakeNode()`, and `AddNode()`.

```
bool AddItem(const Item * pi, Tree * ptree)
{
    Trnode * new_node;

    if  (TreeIsFull(ptree))
    {
        fprintf(stderr,"Tree is full\n");
        return false;             /* early return           */
    }
    if (SeekItem(pi, ptree).child != NULL)
    {
        fprintf(stderr, "Attempted to add duplicate item\n");
        return false;             /* early return           */
    }
    new_node = MakeNode(pi);      /* points to new node     */
    if (new_node == NULL)
    {
        fprintf(stderr, "Couldn't create node\n");
        return false;             /* early return           */
    }
    /* succeeded in creating a new node */
    ptree->size++;

    if (ptree->root == NULL)      /* case 1: tree is empty  */
        ptree->root = new_node;   /* new node is tree root  */
    else                          /* case 2: not empty      */
        AddNode(new_node,ptree->root); /* add node to tree  */
    return true;                  /* successful return      */
}
```

The SeekItem(), MakeNode(), and AddNode() functions are not part of the public interface for the Tree type. Instead, they are static functions hidden in the tree.c file. They deal with implementation details, such as nodes, pointers, and structures, that don't belong in the public interface.

The MakeNode() function is pretty simple. It handles the dynamic memory allocation and the initialization of the node. The function argument is a pointer to the new item, and the function's return value is a pointer to the new node. Recall that malloc() returns the null pointer if it can't make the requested allocation. The MakeNode() function initializes the new node only if memory allocation succeeds. Here is the code for MakeNode():

```
static Trnode * MakeNode(const Item * pi)
{
    Trnode * new_node;

    new_node = (Trnode *) malloc(sizeof(Trnode));
    if (new_node != NULL)
    {
        new_node->item = *pi;
        new_node->left = NULL;
        new_node->right = NULL;
    }

    return new_node;
}
```

The AddNode() function is the second most difficult function in the binary search tree package. It has to determine where the new node goes and then has to add it. In particular, it needs to compare the new item with the root item to see whether the new item goes into the left subtree or the right subtree. If the item were a number, you could use < and > to make comparisons. If the item were a string, you could use strcmp() to make comparisons. But the item is a structure containing two strings, so you'll have to define your own functions for making comparisons. The ToLeft() function, to be defined later, returns True if the new item should be in the left subtree, and the ToRight() function returns True if the new item should be in the right subtree. These two functions are analogous to < and >, respectively. Suppose the new item goes to the left subtree. It could be that the left subtree is empty. In that case, the function just makes the left child pointer point to the new node. What if the left subtree isn't empty? Then the function should compare the new item to the item in the left child node, deciding whether the new item should go in the left subtree or right subtree of the child node. This process should continue until the function arrives at an empty subtree, at which point the new node can be added. One way to implement this search is to use recursion—that is, apply the AddNode() function to a child node instead of to the root node. The recursive series of function calls ends when a left or right subtree is empty—that is, when `root->left` or `root->right` is `NULL`. Keep in mind that `root` is a pointer to the top of the current subtree, so it points to a new, and lower-level, subtree each recursive call. (You might want to review the discussion of recursion in Chapter 9.)

```
static void AddNode (Trnode * new_node, Trnode * root)
{
    if (ToLeft(&new_node->item, &root->item))
    {
        if (root->left == NULL)      /* empty subtree       */
            root->left = new_node;   /* so add node here    */
        else
            AddNode(new_node, root->left);/* else process subtree*/
    }
    else if (ToRight(&new_node->item, &root->item))
    {
        if (root->right == NULL)
            root->right = new_node;
        else
            AddNode(new_node, root->right);
    }
    else                        /* should be no duplicates */
    {
        fprintf(stderr,"location error in AddNode()\n");
        exit(1);
    }
}
```

The `ToLeft()` and `ToRight()`functions depend on the nature of the `Item` type. The members of the Nerfville Pet Club will be ordered alphabetically by name. If two pets have the same name, order them by kind. If they are also the same kind, then the two items are duplicates, which aren't allowed in the basic search tree. Recall that the standard C library function `strcmp()` returns a negative number if the string represented by the first argument precedes the second string, returns zero if the two strings are the same, and returns a positive number if the first string follows the second. The `ToRight()` function has similar code. Using these two functions instead of making comparisons directly in `AddNode()` makes the code easier to adapt to new requirements. Instead of rewriting `AddNode()` when a different form of comparison is needed, you rewrite `ToLeft()` and `ToRight()`.

```
static bool ToLeft(const Item * i1, const Item * i2)
{
    int comp1;

    if ((comp1 = strcmp(i1->petname, i2->petname)) < 0)
        return true;
    else if (comp1 == 0 &&
                strcmp(i1->petkind, i2->petkind) < 0 )
        return true;
    else
        return false;
}
```

Finding an Item

Three of the interface functions involve searching the tree for a particular item: `AddItem()`, `InTree()`, and `DeleteItem()`. This implementation uses a `SeekItem()` function to provide that service. The `DeleteItem()` function has an additional requirement: It needs to know the parent node of the deleted item so that the parent's child pointer can be updated when the child is deleted. Therefore, we designed `SeekItem()` to return a structure containing two pointers: one pointing to the node containing the item (`NULL` if the item isn't found) and one pointing to the parent node (`NULL` if the node is the root and has no parent). The structure type is defined as follows:

```
typedef struct pair {
    Trnode * parent;
    Trnode * child;
} Pair;
```

The `SeekItem()` function can be implemented recursively. However, to expose you to a variety of programming techniques, we'll use a `while` loop to handle descending through the tree. Like `AddNode()`, `SeekItem()` uses `ToLeft()` and `ToRight()` to navigate through the tree. `SeekItem()` initially sets the `look.child` pointer to point to the root of the tree, and then it resets `look.child` to successive subtrees as it traces the path to where the item should be found. Meanwhile, `look.parent` is set to point to successive parent nodes. If no matching item is found, `look.child` will be `NULL`. If the matching item is in the root node, `look.parent` is `NULL` because the root node has no parent. Here is the code for `SeekItem()`:

```
static Pair SeekItem(const Item * pi, const Tree * ptree)
{
    Pair look;
    look.parent = NULL;
    look.child = ptree->root;

    if (look.child == NULL)
        return look;                        /* early return   */

    while (look.child != NULL)
    {
        if (ToLeft(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->left;
        }
        else if (ToRight(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->right;
        }
        else       /* must be same if not to left or right    */
            break; /* look.child is address of node with item */
    }

    return look;                       /* successful return   */
}
```

Note that because the `SeekItem()` function returns a structure, it can be used with the structure membership operator. For example, the `AddItem()` function used the following code:

```
if (SeekItem(pi, ptree).child != NULL)
```

After you have `SeekItem()`, it's simple to code the `InTree()` public interface function:

```
bool InTree(const Item * pi, const Tree * ptree)
{
    return (SeekItem(pi, ptree).child == NULL) ? false : true;
}
```

Considerations in Deleting an Item

Removing an item is the most difficult of the tasks because you have to reconnect the remaining subtrees to form a valid tree. Before attempting to program this task, it's a good idea to develop a visual picture of what has to be done.

Figure 17.13 illustrates the simplest case. Here the node to be deleted has no children. Such a node is called a *leaf*. All that has to be done in this case is to reset a pointer in the parent node to `NULL` and to use the `free()` function to reclaim the memory used by the deleted node.

Next in complexity is deleting a node with one child. Deleting the node leaves the child subtree separate from the rest of the tree. To fix this, the address of the child subtree needs to be stored in the parent node at the location formerly occupied by the address of the deleted node (see Figure 17.14).

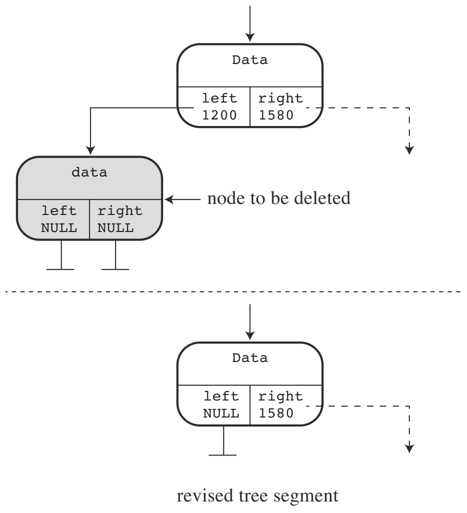


Figure 17.13 Deleting a leaf.

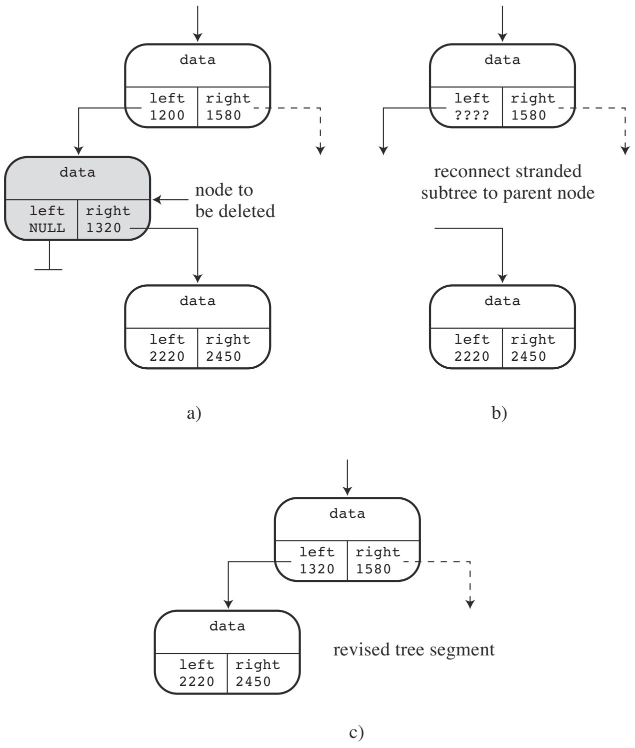


Figure 17.14 Deleting a one-child node.

The final case is deleting a node with two subtrees. One subtree, say the left, can be attached to where the deleted node was formerly attached. But where should the remaining subtree go? Keep in mind the basic design of a tree. Every item in a left subtree precedes the item in the parent node, and every item in a right subtree follows the item in the parent node. This means that every item in the right subtree comes after every item in the left subtree. Also, because the right subtree once was part of the subtree headed by the deleted node, every item in the right subtree comes before the parent node of the deleted node. Imagine coming down the tree looking for where to place the head of the right subtree. It comes before the parent node, so you have to go down the left subtree from there. However, it comes after every item in the left subtree, so you have to take the right branch of the left subtree and see whether it has an opening for a new node. If not, you must go down the right side of the left subtree until you do find an opening. Figure 17.15 illustrates the approach.

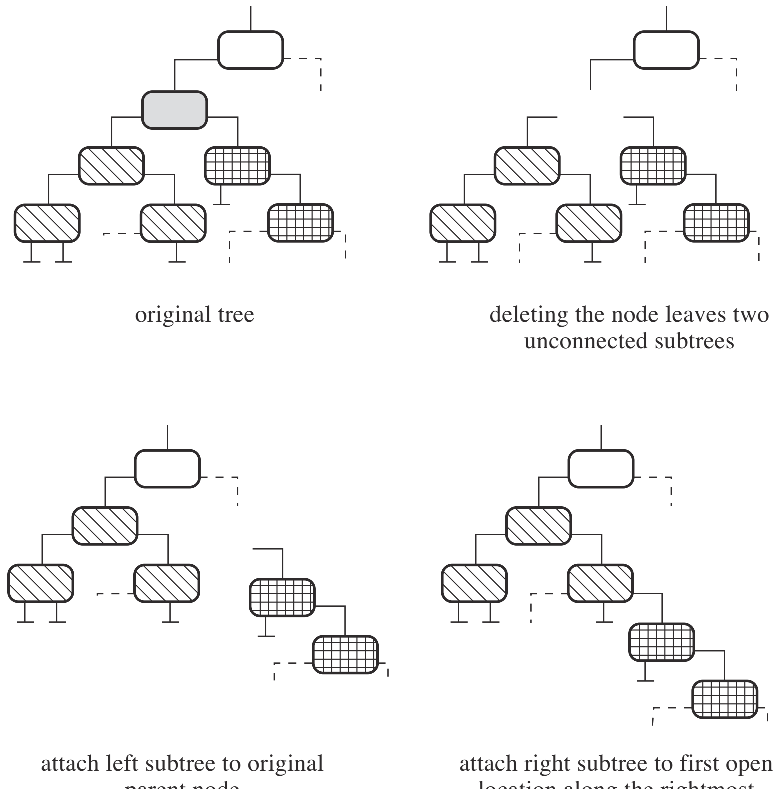

original tree
deleting the node leaves two unconnected subtrees
attach left subtree to original parent node
attach right subtree to first open location along the rightmost branches of first subtree

Figure 17.15 Deleting a two-child node.

Deleting a Node

Now you can begin to plan the necessary functions, separating the job into two tasks. One is associating a particular item with the node to be deleted, and the second is actually deleting the node. One point to note is that all the cases involve modifying a pointer in the parent node, which has two important consequences:

- The program has to identify the parent node of the node to be deleted.
- To modify the pointer, the code must pass the *address* of that pointer to the deleting function.

We'll come back to the first point later. Meanwhile, the pointer to be modified is itself of type `Trnode *`, or pointer-to-`Trnode`. Because the function argument is the address of that pointer, the argument will be of type `Trnode **`, or pointer-to-pointer-to-`Trnode`. Assuming you have the proper address available, you can write the deletion function as the following:

```
static void DeleteNode(Trnode **ptr)
/* ptr is address of parent member pointing to target node  */
{
    Trnode * temp;

    if ( (*ptr)->left == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->right;
        free(temp);
    }
    else if ( (*ptr)->right == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->left;
        free(temp);
    }
    else    /* deleted node has two children */
    {
        /* find where to reattach right subtree */
        for (temp = (*ptr)->left; temp->right != NULL;
             temp = temp->right)
            continue;
        temp->right = (*ptr)->right;
        temp = *ptr;
        *ptr =(*ptr)->left;
        free(temp);
    }
}
```

This function explicitly handles three cases: a node with no left child, a node with no right child, and a node with two children. A node with no children can be considered a special case of a node with no left child. If the node has no left child, the code assigns the address of the right child to the parent pointer. But if the node also has no right child, that pointer is `NULL`, which is the proper value for the no-child case.

Notice that the code uses a temporary pointer to keep track of the address of the deleted node. After the parent pointer (`*ptr`) is reset, the program would lose track of where the deleted node is, but you need that information for the `free()` function. So the program stores the original value of `*ptr` in `temp` and then uses `temp` to free the memory used for the deleted node.

The code for the two-child case first uses the `temp` pointer in a `for` loop to search down the right side of the left subtree for an empty spot. When it finds an empty spot, it attaches the right subtree there. Then it reuses `temp` to keep track of where the deleted node is. Next, it attaches the left subtree to the parent and then frees the node pointed to by `temp`.

Note that because `ptr` is type `Trnode **`, `*ptr` is of type `Trnode *`, making it the same type as `temp`.

Deleting an Item

The remaining part of the problem is associating a node with a particular item. You can use the `SeekItem()` function to do so. Recall that it returns a structure containing a pointer to the parent node and a pointer to the node containing the item. Then you can use the parent node pointer to get the proper address to pass to the `DeleteNode()` function. The `DeleteItem()` function, shown here, follows this plan:

```
bool DeleteItem(const Item * pi, Tree * ptree)
{
    Pair look;

    look = SeekItem(pi, ptree);
    if (look.child == NULL)
        return false;

    if (look.parent == NULL)      /* delete root item       */
        DeleteNode(&ptree->root);
    else if (look.parent->left == look.child)
        DeleteNode(&look.parent->left);
    else
        DeleteNode(&look.parent->right);
    ptree->size--;

    return true;
}
```

First, the return value of the `SeekItem()` function is assigned to the `look` structure variable. If `look.child` is `NULL`, the search failed to find the item, and the `DeleteItem()` function quits, returning `false`. If the `Item` is found, the function handles three cases. First, a `NULL` value for `look.parent` means the item was found in the root node. In this case, there is no parent node to update. Instead, the program has to update the root pointer in the `Tree` structure. Therefore, the function passes the address of that pointer to the `DeleteNode()` function. Otherwise, the program determines whether the node to be deleted is the left child or the right child of the parent, and then it passes the address of the appropriate pointer.

Note that the public interface function (`DeleteItem()`) speaks in terms of end-user concerns (items and trees), and the hidden `DeleteNode()` function handles the nitty-gritty of pointer shuffling.

Traversing the Tree

Traversing a tree is more involved than traversing a linked list because each node has two branches to follow. This branching nature makes divide-and-conquer recursion (Chapter 9) a natural choice for handling the problem. At each node, the function should do the following:

- Process the item in the node.
- Process the left subtree (a recursive call).
- Process the right subtree (a recursive call).

You can break this process down into two functions: `Traverse()` and `InOrder()`. Note that the `InOrder()` function processes the left subtree, then processes the item, and then processes the right subtree. This order results in traversing the tree in alphabetic order. If you have the time, you might want to see what happens if you use different orders, such as item-left-right and left-right-item.

```
void Traverse (const Tree * ptree, void (* pfun)(Item item))
{

    if (ptree != NULL)
        InOrder(ptree->root, pfun);
}
static void InOrder(const Trnode * root, void (* pfun)(Item item))
{
    if (root != NULL)
    {
        InOrder(root->left, pfun);
        (*pfun)(root->item);
        InOrder(root->right, pfun);
    }
}
```

Emptying the Tree

Emptying the tree is basically the same process as traversing it. That is, the code needs to visit each node and apply `free()` to it. It also needs to reset the members of the `Tree` structure to indicate an empty `Tree`. The `DeleteAll()` function takes care of the `Tree` structure and passes off the task of freeing memory to `DeleteAllNodes()`. The latter function has the same design as `InOrder()`. It does save the pointer value `root->right` so that it is still available after the root is freed. Here is the code for these two functions:

```
void DeleteAll(Tree * ptree)
{
    if (ptree != NULL)
        DeleteAllNodes(ptree->root);
    ptree->root = NULL;
    ptree->size = 0;
}

static void DeleteAllNodes(Trnode * root)
{
    Trnode * pright;

    if (root != NULL)
    {
        pright = root->right;
        DeleteAllNodes(root->left);
        free(root);
        DeleteAllNodes(pright);
    }
}
```

The Complete Package

Listing 17.11 shows the entire `tree.c` code. Together, `tree.h` and `tree.c` constitute a tree programming package.

Listing 17.11 **The `tree.c` Implementation File**

```
/* tree.c -- tree support functions */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

/* local data type */
typedef struct pair {
    Trnode * parent;
    Trnode * child;
```

```
} Pair;

/* protototypes for local functions */
static Trnode * MakeNode(const Item * pi);
static bool ToLeft(const Item * i1, const Item * i2);
static bool ToRight(const Item * i1, const Item * i2);
static void AddNode (Trnode * new_node, Trnode * root);
static void InOrder(const Trnode * root, void (* pfun)(Item item));
static Pair SeekItem(const Item * pi, const Tree * ptree);
static void DeleteNode(Trnode **ptr);
static void DeleteAllNodes(Trnode * ptr);

/* function definitions */
void InitializeTree(Tree * ptree)
{
    ptree->root = NULL;
    ptree->size = 0;
}

bool TreeIsEmpty(const Tree * ptree)
{
    if (ptree->root == NULL)
        return true;
    else
        return false;
}

bool TreeIsFull(const Tree * ptree)
{
    if (ptree->size == MAXITEMS)
        return true;
    else
        return false;
}

int TreeItemCount(const Tree * ptree)
{
    return ptree->size;
}

bool AddItem(const Item * pi, Tree * ptree)
{
    Trnode * new_node;

    if  (TreeIsFull(ptree))
    {
        fprintf(stderr,"Tree is full\n");
        return false;             /* early return           */
    }
    if (SeekItem(pi, ptree).child != NULL)
    {
        fprintf(stderr, "Attempted to add duplicate item\n");
        return false;             /* early return           */
    }
    new_node = MakeNode(pi);      /* points to new node     */
    if (new_node == NULL)
    {
        fprintf(stderr, "Couldn't create node\n");
        return false;             /* early return           */
    }
    /* succeeded in creating a new node */
    ptree->size++;

    if (ptree->root == NULL)      /* case 1: tree is empty  */
        ptree->root = new_node;   /* new node is tree root  */
    else                          /* case 2: not empty      */
        AddNode(new_node,ptree->root); /* add node to tree  */

    return true;                  /* successful return      */
}

bool InTree(const Item * pi, const Tree * ptree)
{
    return (SeekItem(pi, ptree).child == NULL) ? false : true;
}

bool DeleteItem(const Item * pi, Tree * ptree)
{
    Pair look;

    look = SeekItem(pi, ptree);
    if (look.child == NULL)
        return false;

    if (look.parent == NULL)      /* delete root item       */
        DeleteNode(&ptree->root);
    else if (look.parent->left == look.child)
        DeleteNode(&look.parent->left);
    else
        DeleteNode(&look.parent->right);
    ptree->size--;

    return true;
}
```

```
void Traverse (const Tree * ptree, void (* pfun)(Item item))
{

    if (ptree != NULL)
        InOrder(ptree->root, pfun);
}

void DeleteAll(Tree * ptree)
{
    if (ptree != NULL)
        DeleteAllNodes(ptree->root);
    ptree->root = NULL;
    ptree->size = 0;
}


/* local functions */
static void InOrder(const Trnode * root, void (* pfun)(Item item))
{
    if (root != NULL)
    {
        InOrder(root->left, pfun);
        (*pfun)(root->item);
        InOrder(root->right, pfun);
    }
}

static void DeleteAllNodes(Trnode * root)
{
    Trnode * pright;

    if (root != NULL)
    {
        pright = root->right;
        DeleteAllNodes(root->left);
        free(root);
        DeleteAllNodes(pright);
    }
}

static void AddNode (Trnode * new_node, Trnode * root)
{
    if (ToLeft(&new_node->item, &root->item))
    {
        if (root->left == NULL)      /* empty subtree       */
            root->left = new_node;   /* so add node here    */
        else
            AddNode(new_node, root->left);/* else process subtree*/
    }
    else if (ToRight(&new_node->item, &root->item))
    {
        if (root->right == NULL)
            root->right = new_node;
        else
            AddNode(new_node, root->right);
    }
    else                         /* should be no duplicates */
    {
        fprintf(stderr,"location error in AddNode()\n");
        exit(1);
    }
}\
static bool ToLeft(const Item * i1, const Item * i2)
{
    int comp1;

    if ((comp1 = strcmp(i1->petname, i2->petname)) < 0)
        return true;
    else if (comp1 == 0 &&
             strcmp(i1->petkind, i2->petkind) < 0 )
        return true;
    else
        return false;
}

static bool ToRight(const Item * i1, const Item * i2)
{
    int comp1;

    if ((comp1 = strcmp(i1->petname, i2->petname)) > 0)
        return true;
    else if (comp1 == 0 &&
             strcmp(i1->petkind, i2->petkind) > 0 )
        return true;
    else
        return false;
}

static Trnode * MakeNode(const Item * pi)
{
    Trnode * new_node;

    new_node = (Trnode *) malloc(sizeof(Trnode));
    if (new_node != NULL)
    {
        new_node->item = *pi;
        new_node->left = NULL;
        new_node->right = NULL;
    }

    return new_node;
}

static Pair SeekItem(const Item * pi, const Tree * ptree)
{
    Pair look;
    look.parent = NULL;
    look.child = ptree->root;

    if (look.child == NULL)
        return look;                        /* early return   */

    while (look.child != NULL)
    {
        if (ToLeft(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->left;
        }
        else if (ToRight(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->right;
        }
        else       /* must be same if not to left or right    */
            break; /* look.child is address of node with item */
    }

    return look;                       /* successful return   */
}

static void DeleteNode(Trnode **ptr)
/* ptr is address of parent member pointing to target node  */
{
    Trnode * temp;

    if ( (*ptr)->left == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->right;
        free(temp);
    }
    else if ( (*ptr)->right == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->left;
        free(temp);
    }
    else    /* deleted node has two children */
    {
        /* find where to reattach right subtree */
        for (temp = (*ptr)->left; temp->right != NULL;
             temp = temp->right)
            continue;
        temp->right = (*ptr)->right;
        temp = *ptr;
        *ptr =(*ptr)->left;
        free(temp);
    }
}
```

Trying the Tree

Now that you have the interface and the function implementations, let's use them. The program in Listing 17.12 uses a menu to offer a choice of adding pets to the club membership roster, listing members, reporting the number of members, checking for membership, and quitting. The brief `main()` function concentrates on the essential program outline. Supporting functions do most of the work.

Listing 17.12 **The `petclub.c` Program**

```
/* petclub.c -- use a binary search tree */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "tree.h"

char menu(void);
void addpet(Tree * pt);
void droppet(Tree * pt);
void showpets(const Tree * pt);
void findpet(const Tree * pt);
void printitem(Item item);
void uppercase(char * str);
char * s_gets(char * st, int n);
```

```
int main(void)
{
    Tree pets;
    char choice;

    InitializeTree(&pets);
    while ((choice = menu()) != 'q')
    {
        switch (choice)
        {
            case 'a' :  addpet(&pets);
                break;
            case 'l' :  showpets(&pets);
                break;
            case 'f' :  findpet(&pets);
                break;
            case 'n' :  printf("%d pets in club\n",
                               TreeItemCount(&pets));
                break;
            case 'd' :  droppet(&pets);
                break;
            default  :  puts("Switching error");
        }
    }
    DeleteAll(&pets);
    puts("Bye.");

    return 0;
}

char menu(void)
{
    int ch;

    puts("Nerfville Pet Club Membership Program");
    puts("Enter the letter corresponding to your choice:");
    puts("a) add a pet          l) show list of pets");
    puts("n) number of pets     f) find pets");
    puts("d) delete a pet       q) quit");
    while ((ch = getchar()) != EOF)
    {
        while (getchar() != '\n')  /* discard rest of line */
            continue;
        ch = tolower(ch);
        if (strchr("alrfndq",ch) == NULL)
            puts("Please enter an a, l, f, n, d, or q:");
        else
            break;
    }
    if (ch == EOF)       /* make EOF cause program to quit */
        ch = 'q';

    return ch;
}

void addpet(Tree * pt)
{
    Item temp;

    if (TreeIsFull(pt))
        puts("No room in the club!");
    else
    {
        puts("Please enter name of pet:");
        s_gets(temp.petname,SLEN);
        puts("Please enter pet kind:");
        s_gets(temp.petkind,SLEN);
        uppercase(temp.petname);
        uppercase(temp.petkind);
        AddItem(&temp, pt);
    }
}

void showpets(const Tree * pt)
{
    if (TreeIsEmpty(pt))
        puts("No entries!");
    else
        Traverse(pt, printitem);
}

void printitem(Item item)
{
    printf("Pet: %-19s  Kind: %-19s\n", item.petname,
           item.petkind);
}

void findpet(const Tree * pt)
{
    Item temp;

    if (TreeIsEmpty(pt))
    {
        puts("No entries!");
        return;     /* quit function if tree is empty */
    }

    puts("Please enter name of pet you wish to find:");
    s_gets(temp.petname, SLEN);
    puts("Please enter pet kind:");
    s_gets(temp.petkind, SLEN);
    uppercase(temp.petname);
    uppercase(temp.petkind);
    printf("%s the %s ", temp.petname, temp.petkind);
    if (InTree(&temp, pt))
        printf("is a member.\n");
    else
        printf("is not a member.\n");
}

void droppet(Tree * pt)
{
    Item temp;

    if (TreeIsEmpty(pt))
    {
        puts("No entries!");
        return;     /* quit function if tree is empty */
    }

    puts("Please enter name of pet you wish to delete:");
    s_gets(temp.petname, SLEN);
    puts("Please enter pet kind:");
    s_gets(temp.petkind, SLEN);
    uppercase(temp.petname);
    uppercase(temp.petkind);
    printf("%s the %s ", temp.petname, temp.petkind);
    if (DeleteItem(&temp, pt))
        printf("is dropped from the club.\n");
    else
        printf("is not a member.\n");
}

void uppercase(char * str)
{
    while (*str)
    {
        *str = toupper(*str);
        str++;
    }
```

```
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

The program converts all letters to uppercase so that *SNUFFY*, *Snuffy*, and *snuffy* are not considered distinct names. Here is a sample run:

```
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet          l) show list of pets
n) number of pets     f) find pets
q) quit
a
Please enter name of pet:
Quincy
Please enter pet kind:
pig
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet          l) show list of pets
n) number of pets     f) find pets
q) quit
a
Please enter name of pet:
Bennie Haha
Please enter pet kind:
parrot
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet          l) show list of pets
n) number of pets     f) find pets
```

```
q) quit
a
Please enter name of pet:
Hiram Jinx
Please enter pet kind:
domestic cat
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet          l) show list of pets
n) number of pets     f) find pets
q) quit
n
3 pets in club
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet          l) show list of pets
n) number of pets     f) find pets
q) quit
l
Pet: BENNIE HAHA          Kind: PARROT
Pet: HIRAM JINX           Kind: DOMESTIC CAT
Pet: QUINCY               Kind: PIG
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet          l) show list of pets
n) number of pets     f) find pets
q)  quit
q
Bye.
```

Tree Thoughts

The binary search tree has some drawbacks. For example, the binary search tree is efficient only if it is fully populated, or *balanced*. Suppose you're storing words that are entered randomly. Chances are the tree will have a fairly bushy look, as in Figure 17.12. Now suppose you enter data in alphabetical order. Then each new node would be added to the right, and the tree might look like Figure 17.16. The Figure 17.12 tree is said to be *balanced*, and the Figure 17.16 tree is *unbalanced*. Searching this tree is no more effective than sequentially searching a linked list.

One way to avoid stringy trees is use more care when building a tree. If a tree or subtree begins to get too unbalanced on one side or the other, rearrange the nodes to restore a better balance. Similarly, you might need to rearrange the tree after a deletion. The Russian mathematicians Adel'son-Vel'skii and Landis developed an algorithm to do this. Trees built with their method are called *AVL trees*. It takes longer to build a balanced tree because of the extra restructuring, but you ensure maximum, or nearly maximum, search efficiency.

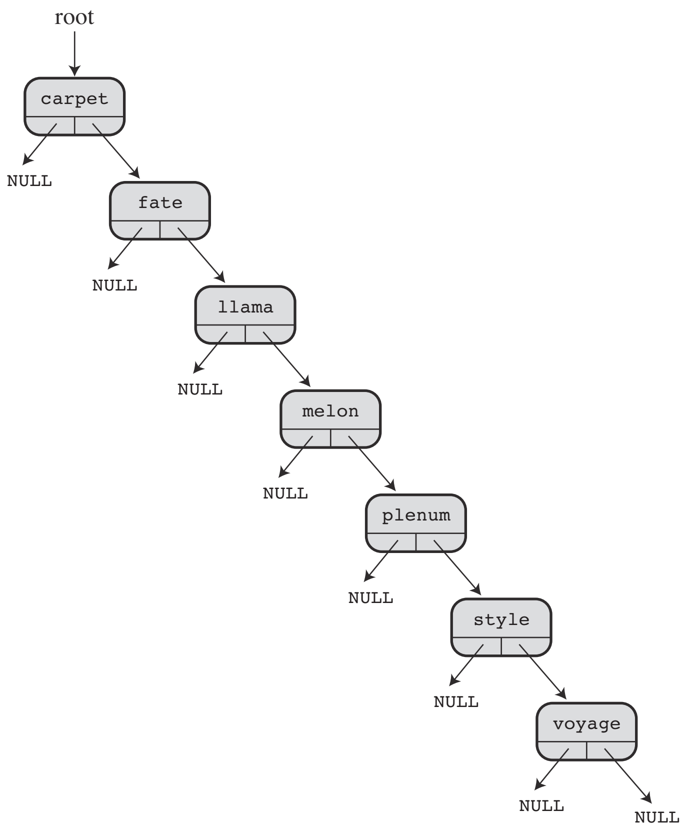


Figure 17.16 A badly unbalanced binary search tree.

You might want a binary search tree that does allow duplicate items. Suppose, for example, that you wanted to analyze some text by tracking how many times each word in the text appears. One approach is to define `Item` as a structure that holds one word and a number. The first time a word is encountered, it's added to the tree, and the number is set to `1`. The next time the same word is encountered, the program finds the node containing the word and increments the number. It doesn't take much work to modify the basic binary search tree to behave in this fashion.

For another possible variation, consider the Nerfville Pet Club. The example ordered the tree by both name and kind, so it could hold Sam the cat in one node, Sam the dog in another node, and Sam the goat in a third node. You couldn't have two cats called Sam, however. Another approach is to order the tree just by name. Making that change alone would allow for only one Sam, regardless of kind, but you could then define `Item` to be a list of structures instead of being a single structure. The first time a Sally shows up, the program would create a new node, then create a new list, and then add Sally and her kind to the list. The next Sally that shows up would be directed to the same node and added to the list.

Tip Add-On Libraries

You've probably concluded that implementing an ADT such as a linked list or a tree is hard work with many, many opportunities to err. Add-on libraries provide an alternative approach: Let someone else do the work and testing. Having gone through the two relatively simple examples in this chapter, you are in a better position to understand and appreciate such libraries.

Other Directions

In this book, we've covered the essential features of C, but we've only touched upon the library. The ANSI C library contains scores of useful functions. Most implementations also offer extensive libraries of functions specific to particular systems. Windows-based compilers support the Windows graphic interface. Macintosh C compilers provide functions to access the Macintosh toolbox to facilitate producing programs with the standard Macintosh interface or for IOS systems, such as iPhones and iPads. Similarly, there are tools for creating Linux programs with graphical interfaces. Take the time to explore what your system has to offer. If it doesn't have what you want, make your own functions. That's part of C. If you think you can do a better job on, say, an input function, do it! And as you refine and polish your programming technique, you will go from C to shining C.

If you've found the concepts of lists, queues, and trees exciting and useful, you might want to read a book or take a course on advanced programming techniques. Computer scientists have invested a lot of energy and talent into developing and analyzing algorithms and ways of representing data. You may find that someone has already developed exactly the tool you need.

After you are comfortable with C, you might want to investigate C++, Objective C, or Java. These *object-oriented* languages have their roots in C. C already has data objects ranging in complexity from a simple `char` variable to large and intricate structures. Object-oriented languages carry the idea of the object even further. For example, the properties of an object include not only what kinds of information it can hold, but also what kinds of operations can be performed on it. The ADTs in this chapter follow that pattern. Also, objects can inherit properties from other objects. OOP carries modularizing to a higher level of abstraction than does C, and it facilitates writing large programs.

You might want to check out the bibliography in Reference Section I, "Additional Reading," for books that might further your interests.

Key Concepts

A data type is characterized by how the data is structured and stored and also by what operations are possible. An abstract data type (ADT) specifies in an abstract manner the properties and operations characterizing a type. Conceptually, you can translate an ADT to a particular programming language in two steps. The first step is defining the programming interface. In C, you can do this by using a header file to define type names and to provide function prototypes that correspond to the allowed operations. The second step is implementing the interface. In C, you can do this with a source code file that supplies the function definitions corresponding to the prototypes.

Summary

The list, the queue, and the binary tree are examples of ADTs commonly used in computer programming. Often they are implemented using dynamic memory allocation and linked structures, but sometimes implementing them with an array is a better choice.

When you program using a particular type (say, a queue or a tree), you should write the program in terms of the type interface. That way, you can modify and improve the implementation without having to alter programs by using the interface.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. What's involved in defining a data type?

2. Why can the linked list in Listing 17.2 be traversed in only one direction? How could you modify the `struct film` definition so that the list could be traversed in both directions?

3. What's an ADT?

4. The `QueueIsEmpty()` function took a pointer to a `queue` structure as an argument, but it could have been written so that it took a `queue` structure rather than a pointer as an argument. What are the advantages and disadvantages of each approach?

5. The *stack* is another data form from the list family. In a stack, additions and deletions can be made from only one end of the list. Items are said to be "pushed onto" the top of the stack and to be "popped off" the stack. Therefore, the stack is a LIFO structure (that is, *last in, first out*).

   a. Devise an ADT for a stack.

   b. Devise a C programming interface for a stack, i.e., a `stack.h` header file.

6. What is the maximum number of comparisons a sequential search and a binary search would need to determine that a particular item is not in a sorted list of three items? 1,023 items? 65,535 items?

7. Suppose a program constructs a binary search tree of words, using the algorithm developed in this chapter. Draw four trees, one for each of the following word entry orderings:

   a. nice food roam dodge gate office wave

   b. wave roam office nice gate food dodge

   c. food dodge roam wave office gate nice

   d. nice roam office food wave gate dodge

8. Consider the binary trees constructed in Review Question 7. What would each one look like after the word *food* is removed from each tree using the algorithm from this chapter?

Programming Exercises

1. Modify Listing 17.2 so that it displays the movie list both in the original order and in reverse order. One approach is to modify the linked-list definition so that the list can be traversed in both directions. Another approach is to use recursion.

2. Suppose `list.h` (Listing 17.3) uses the following definition of a list:

```
typedef struct list
{
    Node * head;    /* points to head of list */
    Node * end;     /* points to end of list  */
} List;
```

   Rewrite the `list.c` (Listing 17.5) functions to fit this definition and test the resulting code with the `films3.c` (Listing 17.4) program.

3. Suppose `list.h` (Listing 17.3) uses the following definition of a list:

```
#define MAXSIZE 100
typedef struct list
{
    Item entries[MAXSIZE];    /* array of items          */
    int items;                /* number of items in list */
} List;
```

   Rewrite the `list.c` (Listing 17.5) functions to fit this definition and test the resulting code with the `films3.c` (Listing 17.4) program.

4. Rewrite `mall.c` (Listing 17.7) so that it simulates a double booth having two queues.

5. Write a program that lets you input a string. The program then should push the characters of the string onto a stack, one by one (see review question 5), and then pop the characters from the stack and display them. This results in displaying the string in reverse order.

6. Write a function that takes three arguments: the name of an array of sorted integers, the number of elements of the array, and an integer to seek. The function returns the value `1` if the integer is in the array, and `0` if it isn't. Have the function use the binary search technique.

7. Write a program that opens and reads a text file and records how many times each word occurs in the file. Use a binary search tree modified to store both a word and the number of times it occurs. After the program has read the file, it should offer a menu with three choices. The first is to list all the words along with the number of occurrences. The second is to let you enter a word, with the program reporting how many times the word occurred in the file. The third choice is to quit.

8. Modify the Pet Club program so that all pets with the same name are stored in a list in the same node. When the user chooses to find a pet, the program should request the pet name and then list all pets (along with their kinds) having that name.