of an equation in the system, we can multiply the jth row in $\mathbf{A}$ and the jth element in $\mathbf{b}$ by a constant without affecting the solution.

- Adding or subtracting any two equations in the system to does not affect the solution. Adding the ith equation to the jth equation corresponds to adding each element in the ith row of $\mathbf{A}$ to the corresponding values in the jth row, and adding the ith element in $\mathbf{b}$ to the jth element.

Gauss-Jordan elimination uses these operations to transform the original system of equations into one with the same solution, but where all the off-diagonal terms of the coefficients are zero. The right-hand side of the equation system after such a transformation divided by the diagonal terms is the solution to the system of linear equations. We present the method in two parts. First, we implement a simple (but slightly flawed) version of Gauss-Jordan elimination to show the basic steps of the method. In the next subsection we augment that method by a technique called *partial pivoting* to eliminate the flaw.

To see how these rules apply to a specific situation, consider a matrix $\mathbf{A}$ and a vector $\mathbf{b}$ as follows:

$$\mathbf{A} = \begin{pmatrix} 2 & 3 & 5 \\ 1 & 2 & 4 \\ 4 & 7 & 3 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 10 \\ 5 \\ 15 \end{pmatrix}$$

This represents the following system of equations:

$$\begin{aligned} 2x_0 + 3x_1 + 5x_2 &= 10 \\ x_0 + 2x_1 + 4x_2 &= 5 \\ 4x_0 + 7x_1 + 3x_2 &= 15 \end{aligned}$$

Gauss-Jordan elimination proceeds by taking half the first equation and subtracting it from the second equation. The goal of these operations is to transform element a_{10} in the coefficient matrix $\mathbf{A}$ to zero. These transformations would yield the equations

$$\begin{aligned} 2x_0 + 3x_1 + 5x_2 &= 10 \\ 0x_0 + .5x_1 + 1.5x_2 &= 0 \\ 4x_0 + 7x_1 + 3x_2 &= 15 \end{aligned}$$

The next step would be to transform a_{20} to zero by taking twice the first equation and subtracting it from the third. This would reduce the system of equations to

$$\begin{aligned} 2x_0 + 3x_1 + 5x_2 &= 10 \\ 0x_0 + .5x_1 + 1.5x_2 &= 0 \\ 0x_0 + 1x_1 - 7x_2 &= -5 \end{aligned}$$

The effect of these transformations has been to reduce all off-diagonal coefficients in the first column of $\mathbf{A}$ to zero. We call the element on the diagonal used to transform the off-diagonal elements in the same column the *pivot element*.

The next step in Gauss-Jordan elimination is to use the diagonal element in the second row as a pivot to reduce all off-diagonal terms in the *second* column to zero. Taking 6 times the second row and subtracting it from the first row yields

$$2x_0 + 0x_1 - 4x_2 = 10$$
$$0x_0 + .5x_1 + 1.5x_2 = 0$$
$$0x_0 + 1x_1 - 7x_2 = -5$$

Taking 2 times the second row and subtracting it from the third row produces

$$2x_0 + 0x_1 - 4x_2 = 10$$
$$0x_0 + .5x_1 + 1.5x_2 = 0$$
$$0x_0 + 0x_1 - 10x_2 = -5$$

The last step is to use the third equation to eliminate off-diagonal terms in the final column. Taking .4 times the third equation and subtracting it from the first yields

$$2x_0 + 0x_1 + 0x_2 = 12$$
$$0x_0 + .5x_1 + 1.5x_2 = 0$$
$$0x_0 + 0x_1 - 10x_2 = -5$$

Taking .15 times the third equation and adding it to the second produces

$$2x_0 + 0x_1 + 0x_2 = 12$$
$$0x_0 + .5x_1 + 0x_2 = -.75$$
$$0x_0 + 0x_1 - 10x_2 = -5$$

The final solution is found by dividing the right-hand side by the diagonal coefficients, resulting in the solution $x_0 = 6$, $x_1 = -1.5$, and $x_2 = .5$. In vector notation, this solution is written as $\mathbf{x} = (6, -1.5, .5)$.

The example above generalizes by noting that Gauss-Jordan elimination proceeds by using the ith diagonal element (a_{ii}) to eliminate the off-diagonal elements in the ith column of $\mathbf{A}$. The basic transformation for each row is of the form

$$\text{row}_j = \cancel{\text{row}_j\left(1 - \frac{a_{ji}}{a_{ii}}\right)} \quad \text{row}_j - \text{row}_i \frac{a_{ji}}{a_{ii}}$$

This algorithm is most easily implemented in C by creating a "working matrix" that contains all of the coefficients and the right-hand-side values. All transformations are done on the working matrix. This avoids changing values in the original arrays $\mathbf{A}$ and $\mathbf{b}$. The working matrix has n rows and $n + 1$ columns. The first n columns are the matrix $\mathbf{A}$, and the last column is $\mathbf{b}$. In our simple three equation example, the working matrix would be

$$\begin{pmatrix} 2 & 3 & 5 & 10 \\ 1 & 2 & 4 & 5 \\ 4 & 7 & 3 & 15 \end{pmatrix}$$

The C implementation copies values from **A** and **b** into the working matrix and then does the appropriate Gauss-Jordan elimination. It then computes the solution vector **x**. The function requires that the user define a working array with sufficient space. This working array is one of the arguments to the function. The implementation is as follows:

```
/* Gauss-Jordan without partial pivoting */
#define A(I,J) (*(a + (I)*n + (J)))
#define WORK(I,J) (*(work + (I)*(n+1) + (J)))

void simple_gauss(
  double *a,      /* the coefficients in A */
  double b[],     /* the right-hand side b */
  double *work,   /* the working array */
  double x[],     /* the solution array */
  int n)          /* number of equations */

{
  double m;
  int i,j,k;

/* set up working matrix */
  for(i=0; i<n; i++) {
    for(j=0; j<n; j++)
      WORK(i,j) = A(i,j);
    WORK(i,n) = b[i];
  }

/* loop through the rows of the working matrix */
  for(i=0; i<n; i++)
/* perform elimination */
    for(j=0; j<n; j++)
      if(j != i) {
        m = WORK(j,i)/WORK(i,i);
        for(k=i; k<=n; k++)
          WORK(j,k) -= m*WORK(i,k);
      }

/* compute solution as rhs divided by diagonal */

  for(i=0; i<n; i++)
    x[i] = WORK(i,n)/WORK(i,i);
}

#undef A
#undef WORK
```

The implementation uses two macros for accessing elements in the matrix of coefficients and the working matrix. Note that the macro **WORK** is written somewhat differently from the macro **A** because working matrix has **n+1** rather than **n** columns.

The function prototype for **simple_gauss()** is

```
void simple_gauss(double *, double *, double *,
                  double *, int);
```

and a code fragment that uses the function to solve our sample three-equation linear system would be

```
/* code using simple_gauss() */
double coeff[3][3] = { {2,3,5}, {1,2,4}, {4,7,3}};
double rhs[3] = {10,5,15};
double place[3][4];
double answer[3];
simple_gauss(&coeff[0][0],rhs,&place[0][0],answer, 3);
printf("Solution is: %lf %lf %lf\n", answer[0],
            answer[1], answer[2]);
```

Note that the two matrices, **coeff** and **place**, are passed to **simple_gauss()** in the form **&coeff[0][0]** and **&place[0][0]** rather than as **coeff** and **mat**. This is because, as discussed in Section 4.6.2, **coeff** and **mat** are not pointers to **double**s, but are pointers to *arrays* of **double**s. The function **simple_gauss()** is written to accept pointers to **double**s, which are then used as the base addresses of the matrices. Many C compilers will accept either form for the arguments even though the use of the matrix names alone is technically incorrect.

4.8.2 Partial Pivoting

The implementation of **simple_gauss()** unfortunately has a major flaw. *In particular, the method fails if any of the pivot elements is zero.* In this case the algorithm will attempt to divide an element of the coefficient matrix by zero, producing a floating-point error.

In order to avoid division by zero, we need to select a nonzero pivot element at each step. The simplest method is based on the observation that exchanging two rows in the working matrix leaves the system of equations unchanged. Thus, before pivoting on a diagonal element, we can look for a row in the working matrix that would produce a nonzero pivot element. Once we find this row, we can exchange it with the row that would have yielded a zero pivot value. This approach is called *partial pivoting*.

At each step of Gauss-Jordan elimination, all the rows below the one currently being considered are candidates for pivot elements. The rows above the current one already have been used for pivot elements. They cannot be used again because moving them to a different row of the working matrix will place their nonzero diagonal element in a nondiagonal position.

In general, there will be more than one candidate row that yields a nonzero pivot element. The selection of which one to use turns out to be significant. This is because without an appropriate selection, it is possible to show that Gauss-Jordan elimination

is numerically unstable. Roundoff errors tend to accumulate in ways that seriously compromise the accuracy of the method.

The best element to select on a theoretical basis is still unknown. However, the simple approach of selecting the pivot element with the greatest absolute magnitude tends to work well empirically. This is the method we will implement.

We also need to deal with the case where all of the candidate pivot elements are zero. This happens when the system of linear equations either has no solution or has an infinite number of solutions. One way to deal with this problem is to test whether the absolute value of the largest candidate pivot element exceeds zero. While this would work on a computer with infinite floating-point precision, a more useful test for real computers is to make sure that the largest candidate pivot element exceeds some tolerance that the user of the function specifies. This makes it possible to detect both cases where there is no solution or where the pivot element is so small that a numerically meaningful solution cannot be found. We will implement the latter technique by defining an argument `tol` which is the smallest allowed pivot element. Our algorithm will stop if a pivot smaller than `tol` in absolute value is encountered.

The basic steps of partial pivoting are as follows:

- **Step 1.** Set up a working matrix as in the simple version of Gauss-Jordan elimination.
- **Step 2.** For each row i (starting with row 0) in the working matrix:
  1. Check rows i through $n-1$ for the one which yields the pivot element with the largest absolute value. Call that row k.
  2. Swap row i with k.
  3. Perform Gauss-Jordan elimination using element ii of the working matrix as the pivot.
- **Step 3.** For each row i compute solution x_i as element in of the working matrix divided by element ii of the working matrix.

The implementation of this method is as follows:

```
/* implementation of Gauss-Jordan elimination
   with partial pivoting */

#include <math.h>
#define TRUE 1

#define FALSE 0
#define A(I,J) (*(a + (I)*n + (J)))
#define WORK(I,J) (*(work + (I)*(n+1) + (J)))

int gauss_jordan(
  double *a,      /* the coefficients in A */
  double b[],     /* the right-hand side b */
  double *work,   /* the working array */
  double x[],     /* the solution array */
```

```
  int n,            /* number of equations */
  double tol)       /* minimum allowed pivot element */

{
  double m, max, temp;
  int i,j,k, swap;

/* set up working matrix */
  for(i=0; i<n; i++) {
    for(j=0; j<n; j++)
      WORK(i,j) = A(i,j);
    WORK(i,n) = b[i];
  }

/* loop through the rows of the working matrix */
  for(i=0; i<n; i++) {
    max = -1.0;
    for(k=i; k<n; k++)
      if(fabs(WORK(k,i)) > max) {
        max = fabs(WORK(k,i));
        swap = k;
      }
/* check if pivot element is "large enough" */
    if(max <= tol)
      return(FALSE);

/* swap rows */
    if(swap != i)
      for(k=i; k<=n; k++) {
temp = WORK(i,k);
        WORK(i,k) = WORK(swap,k);
        WORK(swap,k) = temp;
      }

/* perform elimination */
    for(j=0; j<n; j++)
      if(j != i) {
        m = WORK(j,i)/WORK(i,i);
        for(k=i; k<=n; k++)
          WORK(j,k) -= m*WORK(i,k);
      }

  }
/* compute solution as rhs divided by diagonal */

  for(i=0; i<n; i++)
    x[i] = WORK(i,n)/WORK(i,i);
  return(TRUE);
}
```

```
#undef A
#undef WORK
```

The function **gauss_jordan()** returns an **int** which is **TRUE** if it found a solution without encountering a pivot element smaller than the tolerance **tol**, and returns **FALSE** otherwise. A typical usage for the function is shown in the following **main()** program.

```
/* example program using Gauss-Jordan elimination */
#include <stdio.h>
int gauss_jordan(double *, double *, double *,
                 double *, int, double);

double coeff[3][3] = { {2,3,5}, {1,2,4}, {4,7,3}};
double rhs[3] = {10,5,15};
double place[3][4];

main()
{
  double answer[3];
  if (gauss_jordan(&coeff[0][0], rhs, &place[0][0],
                   answer, 3, .00001))
    printf("Solution is: %lf %lf %lf\n",
                   answer[0], answer[1], answer[2]);
  else
    printf("No solution found\n");
}
```

4.8.3 Solving Systems of Linear Equations with the Same Coefficients

The Gauss-Jordan method generalizes quite easily to the situation where there are several systems of linear equations of the form

$$\begin{aligned}
\mathbf{A}\mathbf{x}_0 &= \mathbf{b}_0 \\
\mathbf{A}\mathbf{x}_1 &= \mathbf{b}_1 \\
\mathbf{A}\mathbf{x}_2 &= \mathbf{b}_2 \\
&\vdots \\
\mathbf{A}\mathbf{x}_{M-1} &= \mathbf{b}_{M-1}
\end{aligned}$$

Here, each of the systems of equations has the same matrix of coefficients but the right-hand sides are different, so the solutions are different. Systems of equations such as these could be solved one at a time using the Gauss-Jordan method. However, they can be solved far more efficiently by altering the Gauss-Jordan method so that the working matrix consists of the matrix **A** augmented by M columns, $\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{M-1}$. The basic steps of Gauss-Jordan are then applied to the working matrix, transforming all

the columns of the new working matrix. When the algorithm terminates, the solutions $\mathbf{x_0}, \mathbf{x_1}, \ldots, \mathbf{x_{M-1}}$ can be found from the added columns in the working matrix.

As an example, consider the solution of the following two systems of simultaneous equations:

$$2x_0 + 3x_1 + 5x_2 = 10$$
$$1x_0 + 2x_1 + 4x_2 = 5$$
$$4x_0 + 7x_1 + 3x_2 = 15$$

and

$$2x_0 + 3x_1 + 5x_2 = 9$$
$$1x_0 + 2x_1 + 4x_2 = 4$$
$$4x_0 + 7x_1 + 3x_2 = 27$$

These equation systems have exactly the same coefficients but different right-hand sides. To solve these efficiently, we would create a working matrix that has the original coefficients *plus* one column per equation set. In the example above, this matrix would be as follows:

$$\begin{pmatrix} 2 & 3 & 5 & 10 & 9 \\ 1 & 2 & 4 & 5 & 4 \\ 4 & 7 & 3 & 15 & 27 \end{pmatrix}$$

We would then perform the steps in Gaussian reduction, operating on this working matrix at each step. Each step in the reduction would modify all the columns. For example, in a simple Gauss-Jordan elimination (without partial pivoting), the first step would take half the first row and subtract it from the second row, yielding the matrix

$$\begin{pmatrix} 2 & 3 & 5 & 10 & 9 \\ 0 & .5 & 1.5 & 0 & -.5 \\ 4 & 7 & 3 & 15 & 27 \end{pmatrix}$$

This process would proceed until the matrix was reduced to the following:

$$\begin{pmatrix} 2 & 0 & 0 & 12 & 8 \\ 0 & .5 & 0 & -.75 & 1 \\ 0 & 0 & -10 & -5 & 10 \end{pmatrix}$$

The two solutions are found by dividing the diagonal elements into the right-hand-side columns, producing the vectors (6, -1.5, .5) and (4, 2, -1) as solutions to the two systems of linear equations.

4.8.4 Improvements in Gauss-Jordan

A relatively minor improvement in the Gauss-Jordan method, called *implicit pivoting*, first rescales all the equations so they are all the same order of magnitude, and then uses partial pivoting. This requires a little additional computational effort and storage to find and store the scaling factors. In addition, the final solution must be

rescaled back to the original units. The key advantage of this method is that the choice of the pivot element is independent of the scale of the independent variables.

A more sophisticated version of pivoting is called *full pivoting*. In this approach, exchanges of both rows *and columns* are considered in order to select the pivot element. Exchanging columns permutes the subscripts in the vector **x**. Implementing full pivoting therefore requires that the function keep track of which elements in the original vector **x** are stored in each column. The order of the independent variables must be updated each time two columns are swapped. The x's in the final solution are reordered to take into account the column swapping. Full pivoting makes it possible to select better pivot elements, thereby reducing the effects of roundoff errors.

The speed of all the pivoting methods can often be improved by a technique called *Gaussian elimination with backsubstitution*. This modification uses Gaussian elimination to reduce only the *lower triangle* of the system of equations to zero, reducing the original matrix of coefficients to one of the form

$$\begin{pmatrix} a'_{00} & a'_{01} & a'_{02} & \cdots & a'_{0,n-1} \\ 0 & a'_{11} & a'_{12} & \cdots & a'_{1,n-1} \\ 0 & 0 & a'_{22} & \cdots & a'_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a'_{n-1,n-1} \end{pmatrix}$$

where the notation a'_{ij} denotes the coefficients of the equations after the Gaussian elimination. The original coefficients in the vector **b** are also transformed in the elimination process into a new vector $\mathbf{b} = b'_0, b'_1, b_2, \ldots, b'_{n-1}$.

Once the matrix is in this form, the solution can be found by working backward, solving first for x_{n-1} as

$$x_{n-1} = \frac{b'_{n-1}}{a'_{n-1,n-1}}$$

This can be used to solve for x_{n-2} as

$$x_{n-2} = \frac{b'_{n-2} - a'_{n-2,n-1}x_{n-1}}{a'_{n-2,n-2}}$$

The backward substitution proceeds solving for x_{n-j} with the general formula

$$x_{n-j} = \frac{b'_{n-j} - \sum_{k=n-j+1}^{n-1} a'_{n-j,k}x_k}{a'_{n-j,n-j}}$$

How backsubstitution works can be seen using the now-familiar system of equations

$$\begin{aligned} 2x_0 + 3x_1 + 5x_2 &= 10 \\ 1x_0 + 2x_1 + 4x_2 &= 5 \\ 4x_0 + 7x_1 + 3x_2 &= 15 \end{aligned}$$

If we create the working matrix

$$\begin{pmatrix} 2 & 3 & 5 & 10 \\ 1 & 2 & 4 & 5 \\ 4 & 7 & 3 & 15 \end{pmatrix}$$

and reduce it by only generating zeros in the lower triangle, we obtain the matrix

$$\begin{pmatrix} 2 & 3 & 5 & 10 \\ 0 & .5 & 1.5 & 0 \\ 0 & 0 & -10 & -5 \end{pmatrix}$$

This corresponds to the equation system

$$2x_0 + 3x_1 + 5x_2 = 10$$

$$0x_0 + .5x_1 + 1.5x_2 = 0$$

$$0x_0 + 0x_1 - 10x_2 = -5$$

This can be solved first for the values of **x** as follows:

$$x_2 = \frac{-5}{-10} = .5$$

$$x_1 = \frac{0 - 1.5(.5)}{.5} = -1.5$$

$$x_0 = \frac{10 - 5(.5) - 3(-1.5)}{2} = 6$$

There are even more sophisticated methods that are used to solve very large systems of linear equations. One group of methods involves various forms of decomposition of the matrix of coefficients (called *LU decomposition*). Another set of approaches are iterative in nature, converging to a solution that is close to the true one. All of these techniques are beyond the scope of this book. The reader is referred to [Press, William H. 88].

4.9 MATRIX INVERSION

One of the common uses of Gauss-Jordan elimination is to find what is called the *matrix inverse*. For any matrix **A** that has an equal number of rows and columns, the inverse of **A** (denoted by $\mathbf{A}^{-1}$) is the matrix such that the product $\mathbf{AA}^{-1}$ is a matrix which has all ones along the diagonal and zeros for the rest of the elements. (A matrix with ones along its diagonal and zeros in its off-diagonal terms is called the *identity matrix*, and is usually denoted by **I**.)

For example, if a 2 by 2 matrix **A** is

$$A = \begin{pmatrix} 1 & 2 \\ 4 & 3 \end{pmatrix}$$

then the reader can verify that the inverse of $\mathbf{A}$ is

$$\begin{pmatrix} -3/5 & 2/5 \\ 4/5 & -1/5 \end{pmatrix}$$

Gauss-Jordan elimination can be used to find the inverse of a matrix by structuring the problem of finding the elements of $\mathbf{A}^{-1}$ as the solution to a series of systems of simultaneous linear equations. Each system of equations defines the conditions for one of the rows of the inverse of $\mathbf{A}$. To see this, define the elements of $\mathbf{A}$ as a_{ij} and denote the elements of $\mathbf{A}^{-1}$ as x_{ij}. Using this notation, the system of equations which provides the first row of $\mathbf{A}^{-1}$ is

$$\begin{aligned}
a_{00}x_{00} + a_{01}x_{10} + a_{02}x_{20} \; + \ldots + a_{0,n-1}x_{n-1,0} &= 1 \\
a_{00}x_{01} + a_{01}x_{11} + a_{02}x_{21} \; + \ldots + a_{0,n-1}x_{n-1,1} &= 0 \\
a_{00}x_{02} + a_{01}x_{12} + a_{02}x_{20} \; + \ldots + a_{0,n-1}x_{n-1,2} &= 0 \\
&\vdots \\
a_{00}x_{0,n-1} + a_{01}x_{1,n-1} + a_{02}x_{2,n-1} \; + \ldots + a_{0,n-1}x_{n-1,n-1} &= 0
\end{aligned}$$

This can be written more compactly in matrix form by defining $\mathbf{I}^j$ as a vector with a one in the jth element and a zero in all other elements. $\mathbf{I}^j$ is the jth column of the identity matrix. We also define $\mathbf{x}^j$ as the jth column of $\mathbf{A}^{-1}$. Using this notation we can write the system of equations above as

$$\mathbf{A}\mathbf{x}^0 = \mathbf{I}^0$$

The other columns of $\mathbf{A}^{-1}$ can be solved with equation systems of the same form:

$$\begin{aligned}
\mathbf{A}\mathbf{x}^1 &= \mathbf{I}^1 \\
\mathbf{A}\mathbf{x}^2 &= \mathbf{I}^2 \\
&\vdots \\
\mathbf{A}\mathbf{x}^{n-1} &= \mathbf{I}^{n-1}
\end{aligned}$$

This set of simultaneous equation systems is in exactly the same form as those discussed in the preceding section! It is a set of simultaneous equation systems where every equation system has exactly the same matrix of coefficients but where the right-hand sides of the systems differ. We can solve this using the same technique, creating a working matrix that has n additional columns, each corresponding to a column of the identity matrix. The resulting solutions will be the elements in the inverse of $\mathbf{A}$.

To see how this would work, consider again the example matrix

$$A = \begin{pmatrix} 1 & 2 \\ 4 & 3 \end{pmatrix}$$

This matrix can be inverted by setting up two systems of simultaneous equations as follows:

$$1x_0^0 + 2x_1^0 = 1$$

$$4x_0^0 + 3x_1^0 = 0$$

and

$$1x_0^1 + 2x_1^1 = 0$$

$$4x_0^1 + 3x_1^1 = 1$$

These two systems of equations have the same coefficients, so they can be solved by performing Gauss-Jordan reduction on the following working matrix:

$$\begin{pmatrix} 1 & 2 & 1 & 0 \\ 4 & 3 & 0 & 1 \end{pmatrix}$$

The reader can verify that this matrix reduces to

$$\begin{pmatrix} 1 & 0 & -3/5 & 2/5 \\ 0 & -5 & -4 & 1 \end{pmatrix}$$

which implies solutions for x_0^0, x_1^0, x_0^1 and x_1^1 as follows:

$$x_0^0 = -3/5$$

$$x_1^0 = 4/5$$

$$x_0^1 = 2/5$$

$$x_1^1 = -1/5$$

or in matrix form, the inverse matrix is

$$\begin{pmatrix} -3/5 & 2/5 \\ 4/5 & -1/5 \end{pmatrix}$$

4.10 PROGRAMMING PROJECT: SHORTEST PATHS IN A NETWORK

While most of our discussion of arrays has focused on their application to linear algebra, the concept of an array in C is far more general. In this section we develop a complete program to solve what is called the *all-to-all shortest path problem*. This problem arises frequently in transportation systems and logistics planning.

The shortest path problem is what is called a *network* (or *graph*) problem. Basically, a network is a series of points (often called *nodes*) connected by a set of *links* (often called *arcs*). Each link or arc has associated with it some cost, which may be measured in dollars, time, or some more abstract measure of the difficulty of moving

between the two points. In our formulation, each link will be unidirectional, allowing for the possibility that traveling from node i to node j may involve different costs than traveling from j to i.

The shortest path problem has numerous applications. For example, a trucking company may want to find the shortest path in the highway network to move goods from one point to another. Similarly, a telecommunications company may want to route phone messages on the least expensive lines. In these cases, the arcs represent physical entities such as highways or telephone cables.

Figure 4.7 depicts a sample network with five nodes numbered from 0 to 4 and a set of arcs. The label beside each arc is the cost associated with traversing it. For example, the arc connecting nodes 2 to node 3 has a cost of 2 units. Note that not all pairs of nodes have direct connections. For example, there is no arc connecting node 2 to node 4.

Networks are also used in more abstract ways. For example, the nodes might represent various stages in the completion of some construction project. Node 0 might represent the development site before any construction begins, and node 3 might represent the completed building. The arcs in the network might represent the costs of performing some element of the construction process, transforming the site from state i to state j. In this case, the shortest path from node 0 to node 3 would show the least expensive way to construct the building.

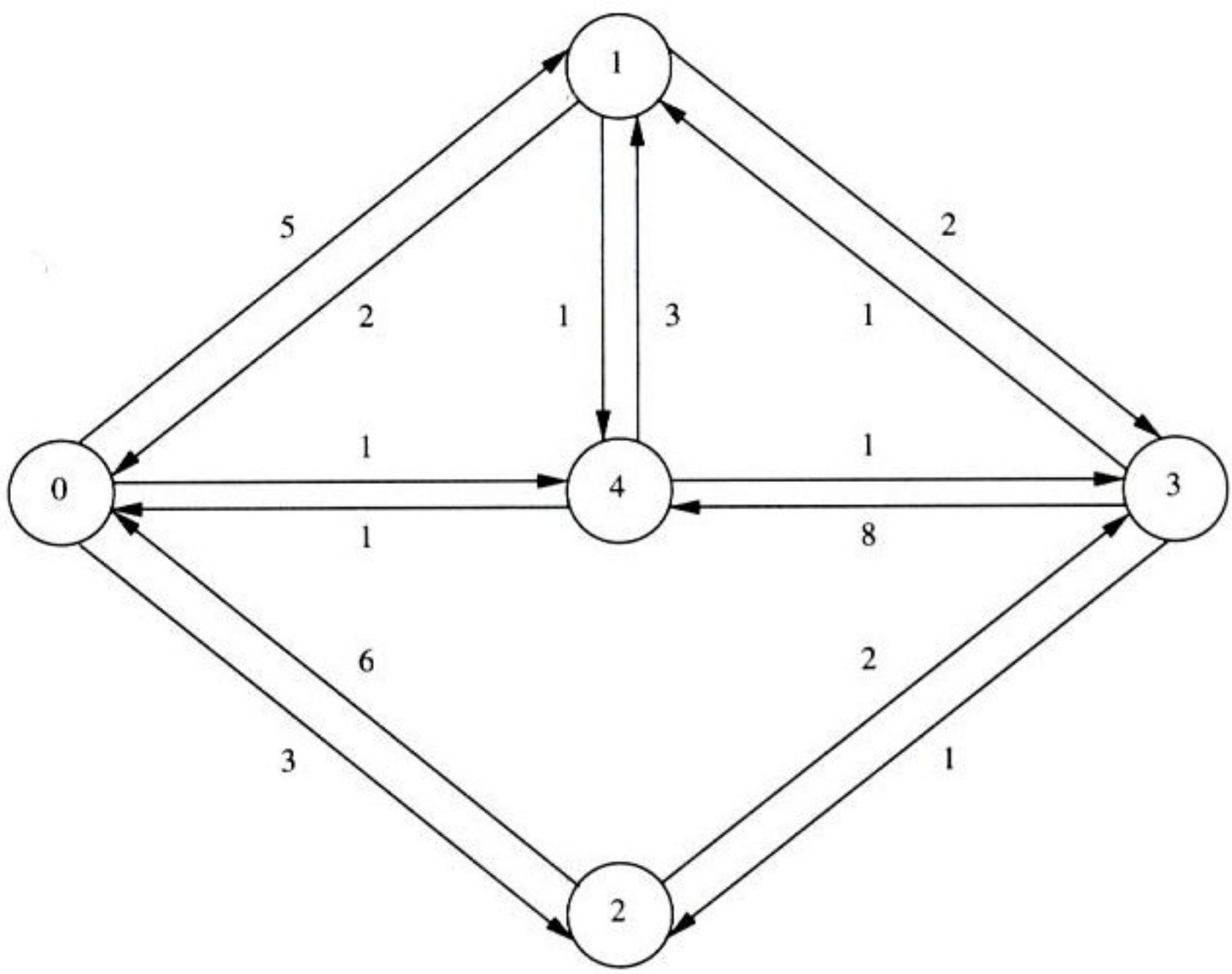

Figure 4.7 Example network.

4.10.1 The Floyd-Warshall Algorithm

Given a network, the all-to-all shortest path problem is to find the total cost of going from every node to every other node in the network. For some pairs of nodes, this can be found by inspecting the network. For example, the shortest way to get from node 4 to node 3 in Figure 4.7 is to traverse the arc connecting these two nodes, incurring a cost of 1 unit. In a large network that may consist of tens of thousands of arcs, finding the shortest path between various node pairs may be very complicated.

The method we will use to solve the all-to-all shortest path problem is called the *Floyd-Warshall algorithm* [Bertsekas, Dmitri P. 89]. It uses a two-dimensional array to represent the cost of traveling from every node to every other node. Each row of the two-dimensional matrix corresponds to an originating node, and each column corresponds to a destination node. For example, if we call the matrix of node-to-node costs **C**, the entries c_{ij} in **C** will hold the cost of traversing the shortest path from node i to node j.

The Floyd-Warshall algorithm begins by initializing the entries in the cost matrix with the costs of the arcs in the network. If we define n to be the number of nodes in the network, this initialization is as follows

- Initialize the diagonal elements $c_{00}, c_{11}, \ldots, c_{n-1,n-1}$ to zero. This reflects the fact that travel from any node to itself is costless.
- Each arc in the network connects some node i and some node j. For each arc, initialize the value c_{ij} in the cost matrix **C** to the cost of that arc.
- All other entries in the array should be set to some large value that exceeds the total cost of any final path in the network. For now, we will denote that value as ∞.

The Floyd-Warshall algorithm then examines every pair of nodes ij $(i \neq j)$, and for each node pair, tests whether the current cost of going from i to j can be reduced by using node 0 as an intermediate node. This corresponds to finding the following minimum:

$$\min(c_{ij}, c_{i0} + c_{0j})$$

Once this is done for all ij pairs, the revised matrix **C** now holds the cost of the shortest paths that use only node 0 as a possible intermediate node. The algorithm then performs the same operation on all node pairs using node 1 as another potential intermediate node. Nodes 2, 3, ..., $n - 1$ are each added, with the cost matrix updated at each stage. When this is completed, the final cost matrix includes the possibility of using all nodes as intermediates, and therefore contains the costs of the shortest paths through the network.

To see how these steps in the algorithm work, consider again the simple network shown in Figure 4.7. The initial cost matrix for this simple five-node network is as

follows:

$$\mathbf{C} = \begin{pmatrix} 0 & 5 & 3 & \infty & 1 \\ 2 & 0 & \infty & 2 & 1 \\ 6 & \infty & 0 & 2 & \infty \\ \infty & 1 & 1 & 0 & 8 \\ 1 & 3 & \infty & 1 & 0 \end{pmatrix}$$

After the initialization step, the algorithm would test alternative ways to get from every node i to every other node $j \neq i$ using node 0 as an intermediate node. For example, the initial cost of going from node 1 to node 2 is initialized to infinity because there is no arc in the network linking these nodes. The updating rule is to compare the current cost $c_{12} = \infty$ with $c_{10} + c_{02} = 2 + 3 = 5$. Since the use of node 0 as an intermediate node for getting from 1 to 2 reduces the current cost, the value 5 would replace c_{12}.

After all ij pairs are tested to see if using node 0 as an intermediate is less costly, the revised cost matrix would be as follows:

$$\mathbf{C} = \begin{pmatrix} 0 & 5 & 3 & \infty & 1 \\ 2 & 0 & 5 & 2 & 1 \\ 6 & 11 & 0 & 2 & 7 \\ \infty & 1 & 1 & 0 & 8 \\ 1 & 3 & 4 & 1 & 0 \end{pmatrix}$$

The algorithm then examines alternative paths using node 1 as an intermediate node. Since the cost matrix has already been revised using node 0 as an intermediate node, adding a test for node 1 yields the lengths of paths that can make use of *either* node 0 or node 1, or both. After all ij pairs are considered in this phase, the resulting cost matrix is

$$\mathbf{C} = \begin{pmatrix} 0 & 5 & 3 & 7 & 1 \\ 2 & 0 & 5 & 2 & 1 \\ 6 & 11 & 0 & 2 & 7 \\ 3 & 1 & 1 & 0 & 2 \\ 1 & 3 & 4 & 1 & 0 \end{pmatrix}$$

The updating process continues by testing nodes 2, 3, and 4 as possible intermediate nodes. When all five nodes have been considered, the resulting cost matrix will contain the costs of the shortest paths in the network. The reader can verify that the resulting matrix will be

$$\mathbf{C} = \begin{pmatrix} 0 & 3 & 3 & 2 & 1 \\ 2 & 0 & 3 & 2 & 1 \\ 5 & 3 & 0 & 2 & 4 \\ 3 & 1 & 1 & 0 & 2 \\ 1 & 2 & 2 & 1 & 0 \end{pmatrix}$$

There are several observations worth noting about the Floyd-Warshall algorithm:

- The Floyd-Warshall algorithm does not provide the actual paths that yield the lowest cost of getting between node pairs. It provides only the costs of those paths.
- The actual path that yields the least cost path may not be unique. For example, in the network shown in Figure 4.7, there are two paths between nodes 1 and 3 that each cost 2 units. The direct route uses the arc between nodes 1 and 3, while an indirect route uses node 4 as an intermediate node.
- The algorithm may not work when some of the arcs have negative costs. In particular, the algorithm requires that every series of arcs in the network that begins and ends at the same node have positive total cost. This condition is guaranteed when all arcs have positive costs.
- In some networks it may not be possible to reach every node from every other node. This will be detected in the algorithm because the initial infinite entries in the cost matrix will never be updated, leaving the value ∞ in the final cost matrix for such node pairs.
- The algorithm requires very large amounts of memory and computation time. The memory requirements grow as the square of the number of nodes in the network, and the computation time grows as the *cube* of the the number of nodes. Methods other than the Floyd-Warshall algorithm exist that are far more efficient for large networks, particularly for networks that have a relatively low ratio of arcs to nodes.

4.10.2 Implementation of the Floyd-Warshall Algorithm

Implementing the Floyd-Warshall algorithm in C is straightforward. We of course need to replace the value ∞ with some value. Since we will be using **int**'s for arc costs, we will select some positive integer value that is very large compared to the costs of arcs. The C header file **limits.h** has a defined constant **INT_MAX** (the largest representable integer). As explored in Exercise 4-22, the value **INT_MAX/2** is a reasonable choice for the "large value."

We begin the complete program with some **#include** directives, some **#define** directives, and function prototypes. These are as follows:

```
/* program to find the shortest path between all nodes
   in a network */

#include <stdio.h>
#include <limits.h>
#define MAXSIZE 100
#define LARGE_VALUE INT_MAX/2

/* prototype for Floyd_Warshall algorithm */
void floyd_warshall(int *, int, int);
```

```
int minint(int, int);     /* min of two ints */
void show_cost(int *, int, int); /* output matrix */
```

The value defined as **LARGE_VALUE** is used for infinity in the implementation. **MAXSIZE** is used to dimension the cost array; its value determines the largest network, measured in the number of nodes, that can be handled by the program. The three functions prototyped are:

- **floyd_warshall()**. This function implements the shortest path algorithm. Its arguments are a pointer to the first entry in the cost matrix, the number of rows in the cost matrix (declared in **main()**), and the number of nodes in the actual network.
- **minint()**. This function returns the minimum of its two **int** arguments.
- **show_cost()**. This function displays the cost matrix. Its arguments are the same as those for the function **floyd_warshall()**.

The next part of the program is a **main()** function that reads in the number of nodes in the network. This is followed by a series of **for** loops that initialize the cost matrix. The arcs in the network are then read in using **scanf()**, with each arc described by its beginning node, ending node, and cost. The **main()** function then invokes the function that implements the Floyd-Warshall algorithm, and then invokes a function that outputs the results.

The **main()** function for the program is as follows:

```
/* main() for Floyd-Warshall program */
main()
{
  int cost[MAXSIZE][MAXSIZE];  /* cost matrix */
  int i,j;                     /* subscripts */
  int flag = !EOF;  /* flag used in reading input */
  int arc_cost;     /* temporary variable */
  int nodes;        /* number of nodes in network */

/* read in number of nodes and
   initialize the cost matrix */
  scanf("%d", &nodes);

  for(i=0; i<nodes; i++)
    for(j=0; j<nodes; j++)
      if(i==j)
        cost[i][j] = 0;
      else
        cost[i][j] = LARGE_VALUE;

/* read in arcs in network */
  while(flag != EOF) {
```

```
    flag = scanf("%d %d %d", &i, &j, &arc_cost);
    if(flag != EOF)
      cost[i][j] = arc_cost;
  }

/*  run Floyd-Warshall algorithm */

  floyd_warshall(&cost[0][0], MAXSIZE, nodes);

/* output results */
  show_cost(&cost[0][0], MAXSIZE, nodes);

}
```

Note that the function defines a cost array that has a maximum size **MAXSIZE**, which is set here to 100. This maximum value can be redefined to handle larger networks.

The core of the program is the function that implements the Floyd-Warshall algorithm. This function is as follows:

```
/* this function implements the Floyd-Warshall algorithm
   for the all-to-all shortest path problem */

#define M(I,J) (*(m + (I)*ncols + (J)))

void floyd_warshall(
  int *m,          /* pointer to beginning of matrix */
  int ncols,       /* number of columns in matrix */
  int nodes)       /* number of nodes */
{
  int i,j,k;

  for(k=0; k<nodes; k++)  /* test intermediate nodes */
    for(i=0; i<nodes; i++)
      for(j=0; j<nodes; j++)
        if(i != j)          /* check costs using node k */
          M(i,j) = minint(M(i,j), M(i,k)+M(k,j));
}
#undef M
```

The two remaining functions, **minint()** and **show_cost()**, are straightforward. **minint()** simply takes the minimum of its two integer arguments and returns that minimum value. Its implementation is

```
/* this function takes the minimum of two integers */

    int minint(int x, int y)
    {
```

```
    if(x<y)
      return(x);
    return(y);
  }
```

The function **show_cost()** displays the resulting cost matrix. Because the cost matrix as defined in **main()** may have more rows than the number of nodes in the actual network, **show_cost()** must be able to display the square, upper left corner of the complete two-dimensional array. It therefore has as arguments a pointer to the first value in the array, the actual number of columns defined for the array, and the number of nodes in the network. As in the function **floyd_warshall()**, a macro is used to reference the actual elements within the array.

The code for **show_cost()** is as follows:

```
/* function to print matrix of integers */

#define MAT(I,J) (*(mat + (I)*ncols + (J)))

void show_cost(int *mat, int ncols, int nodes)
{
    int i,j;
    printf("Cost matrix is:\n");
    for(i=0; i<nodes; i++) {
      for(j=0; j<nodes; j++)
        printf("%5d ", MAT(i,j));
      printf("\n");
     }
}

#undef MAT
```

Our implementation of **show_cost()** has the unfortunate property of producing very long lines of output for large arrays. Exercise 4-11 at the end of this chapter explores the correction of this shortcoming.

4.11 PROGRAMMING PROJECT: SIMULATING A ROBOT IN A MAZE

In Chapter 1 we devised a hypothetical robot which was capable of executing eight distinct machine language instructions. In order to follow this programming project, the reader may wish to review the material in Section 1.5. The robot's machine language was used to write a program to find the way through a maze using the "right hand on wall" algorithm. In this section we develop a C program that simulates the machine language of the robot.

CHAPTER

5

DATA STRUCTURES

5.1 INTRODUCTION

In this chapter we introduce the concept of *data abstraction*. By this, we mean the creation of units of information which have a well-structured form and which can be used in many different contexts. Such units of information are often called *data structures*. These differ from simple variables which can only store single values. We also use the term *data objects* to describe units of information that are composite groups of related items and a set of operations that can be performed on those composite groups.

The C programming language provides a range of capabilities that make it easy to create and operate on data structures. While in a sense, arrays can be viewed as data structures, they are limited to storing a group of values that are *all of the same type*. While these are useful in constructing algorithms to solve many problems, there are situations in which we need more sophisticated ways of organizing information.

Many interesting data abstractions are far more complicated than simple variables and arrays. Consider, for example, a large database used to store airline reservations. This database has information about passengers' names, flight times, the names of standby passengers, seat assignments, fares, and other items. While it is possible to construct a program to manage this database using just simple variables and arrays, it would be far more useful to think of the information as organized into larger units such

as complete reservations (consisting of names, flight numbers, flight times, fare codes, etc.), flight manifests (consisting of the names of all passengers booked on a flight), standby lists (a list of names for each flight) and plane schedules (a list of flights for each plane in the fleet). We could then think about operations on these data abstractions such as, "add passenger p to flight y" or "assign seat q on flight f to passenger p."

Certain ways of organizing information have proven particularly useful in solving various information management problems. For example, many situations require creation of something called a *queue*. New entries are added to the end of a queue, and items are removed from the front of the queue. In the context of an airline management system, information about people waiting for standby seats might be organized as a queue, with people getting seats freed by cancellations on a first come, first served basis.

In order to implement various data structures, we will need to present some new elements in the C language. In particular, we will require the ability to define a composite of existing C data types called a `struct`. These units allow grouping together information of dissimilar types, such as an integer, an array of `double`s, and a character string. The entire group can be treated as a variable, and arrays of this type of variable can be created. We will also need the ability to create and destroy new data structures as a program executes. This feature is called *dynamic memory allocation*. C provides a library of routines that permit such memory usage.

We will also introduce features of C that make development and maintenance of data structures easier. We will, for example, adopt the approach of creating small *libraries* of functions which manipulate each type of data structure. Later in the chapter, we will show how these libraries can be kept in separate source code files from the functions that use the library. This approach is common in developing large programs, where keeping all the source code in a single file would be unmanageable.

Segregating the functions that create and manipulate the data structures into a separate library allows us to create data structures as abstract entities whose internal structure need not be known to other programmers. Only the functions that control the data structures need be known for the library to be useful. The partitioning of programming projects into separate units that communicate through function calls is crucial to effective implementation of large software systems.

After introducing some additional features of the C language, we will explore the most useful organizations of data structures. The particular forms we will study are:

- *Stacks*. These are data structures that organize information on a first in, last out basis. New items are added to the top of a stack, and items are removed from the top of the stack.
- *Ragged arrays*. This data structure is used to hold an array of arrays, where each of the arrays may have a different length. Ragged arrays are often used to hold multiple lines of text, where the number of characters in the lines varies.
- *Queues*. As discussed in the example above, queues are first in, first out data structures. New items are added to the end of the queue, and items are removed from the front.

- *Lists*. These are general-purpose data structures, with many different variants. Basically, a list is a group of data elements linked together in some way. We will develop a simple single linked list.
- *Trees*. These are also general-purpose data structures. Each element of a tree can have one or more elements linked to it. These linked elements are called its *children*. We will consider in detail the *binary tree*, where any element can have at most two children.

At the end of this chapter we develop some example applications that use many of the data structures. These programming projects will illustrate how data structures and the library of functions that operate on them can be used as building blocks to solve quite complicated problems. Using data structures as part of algorithms makes it possible to solve these problems in ways that are surprisingly straightforward.

5.2 THE BASICS OF C STRUCTURES

C

In this section we show how a **struct** is defined in the C language and illustrate how members of the **struct** are referenced. We also show a simple example of the use of **structs**.

5.2.1 Creating a Structure in C

A **struct** in C is a variable which can consist of many components of different types. We can think of the creation of a **struct** in C as consisting of two steps. First, we define a *template* for the **struct**. A template declares a structure type, including what is in that structure type. The components of a **struct** are called its *members*. The template does not allocate any storage for a **struct**. The second step defines variables which are **structs** of a particular type, allocating storage for them.

For example, suppose we want to define a **struct** that consists of information about some household, including an **int** for number of household members, a **float** for the household's annual income, and another **int** for the number of adults in the household. One way to do this in C is to define a template for a **struct** of this type as follows:

```
/* structure template declaration */

struct household {
  int total;      /* number of household members */
  float income;   /* household income */
  int adults;     /* number of adults */
  };    /* don't forget this last semicolon */
```

The template above does not create any new variables; it simply defines the format for **structs** of type **household**. It declares that **structs** of type **household** will contain three *members*: an **int** named **total**, a **float** named **income**, and an **int** named **adults**. The name **household** is called a *structure tag*.

Once the template for a **struct** is declared, we can define variables which are of type **struct household** as follows:

```
struct household a,b,c;
```

The statement defines three variables, **a**, **b**, and **c**, each of which is a **struct** of type **household**.

The two steps of declaration and definition can be combined in different ways. For example, C allows you to declare the template and define variables in a single statement, as in

```
/* template declaration and variable definition */

struct household {
  int total;      /* number of household members */
  float income;   /* household income */
  int adults;     /* number of adults */
  } a, b, c;
```

In this declaration, the structure tag **household** is declared and the variables **a**, **b**, and **c** are defined in a single statement.

You are also allowed to omit the structure tag entirely, as in the statement

```
/* variable definition  without tag */

struct {
  int total;      /* number of household members */
  float income;   /* household income */
  int adults;     /* number of adults */
  } a, b, c;
```

However, when you omit the declaration of the structure tag, you need to repeat the structure template when you want to define other variables of the same type. We will always use structure templates to identify tags and encourage you to do likewise.

C also allows you to create an array of structures. For example, once a template for a **struct** of type **household** has been declared, you can define a variable **harray** as an array of 10 **structs** of type **household** as follows:

```
struct household harray[10];
```

The variable **harray** consists of elements **harray[0]**, **harray[1]**, ..., **harray[9]**, each of which is a **struct**. All the rules for using arrays in C discussed in Chapter 4 apply to arrays of **structs**, including how arrays are passed to functions. We explore this in Section 5.2.6.

A **struct** in C can also have one or more **structs** as members. If the tag **household** is defined earlier, then we could declare another **struct** which consists of a **household** and other elements. For example, we might have a student record that includes the identification number, age, and grade point average of a student along with the information about the student's household. A structure declaration for this would be as follows:

```
/* tag declaration of struct containing a struct */

struct student {
  struct household family; /* household information */
  long int idnum;          /* id number */
  int age;                 /* student's age */
  float gpa;               /* grade point average */
  };
```

The structure type **student** contains a member **family** which itself is a structure of type **household**.

5.2.2 Accessing Members of a Structure

In addition to needing a mechanism for defining a **struct**, we need a way to access the members of the **struct** as variables. C provides the *dot operator* as one such mechanism. For example, the notation **a.income** is an expression which evaluates as the member **income** of **struct a**. Thus, assuming the structure tag **household** has been declared as above, the following are all legal C statements:

```
/* accessing members of a struct */

struct household a,b,c;
a.income = 54000.0;
b.adults = 2;
b.total = 7;
printf("Number of children is %d\n", b.total-b.adults);
scanf("%d %f %d", &c.total, &c.income, &c.adults);
```

The dot operator (along with parentheses denoting function calls, square brackets denoting array references, and a second mechanism for referencing members of a structure to be introduced shortly) have the highest precedence. Thus, in the expression **&c.total**, the member **total** of the **struct c** is evaluated first; the address

operator (&) is then applied to create a pointer to the memory location where `c.total` is stored. In the example above, `scanf()` reads a floating-point value into that memory location.

The dot operator associates from left to right. Thus, if the structure tag `student` is defined as in Section 5.2.1, then the code fragment

```
struct student s;
s.family.adults = 2;
```

defines `s` to be a structure of type `student` and sets the member `adults` of the member `family` to 2.

The name of a member of a `struct` is local to that `struct`. Thus, we could define the template for another `struct` with the tag `staff`, and that `struct` could have as one of its members a variable called `idnum`. The member named `idnum` in the structure `staff` would be entirely different from the member with the same name in the structure `student`. The C compiler can always distinguish between the two members named `idnum` because references to a member always require the structure name as well as the member name. For example, if `a` is a `struct` of type `student`, and `d` is a `struct` of type `staff`, then `a.idnum` is a reference to member `idnum` in a `student` structure, while `d.idnum` is a reference to member `idnum` of a `staff` structure.

5.2.3 Initialization of a struct

A `struct` can be initialized in the same way as an array. For example, we could initialize a `struct` of type `household` along with its definition as follows:

```
struct household a={ 5, 35000.0, 2};
```

This definition initializes `a.total` to 5, `a.income` to 35000 and `a.adults` to 2. Similarly, a `struct` of type `student` could be initialized as

```
struct student s = { {5, 35000.0, 2}, 245323, 19, 3.54};
```

The inner curly braces enclosing the initialization of the `household` structure within the `student` structure are optional. We encourage their inclusion to make the composition of the members of the `struct` clear.

Arrays of structures get initialized just as any other array. For example, we could initialize an array of `household` structures as

```
struct household h[] = { {4, 20000.0, 1},
                         {7, 29000.0, 2},
                         {1, 35000.0, 1},
                         {5, 95000.0, 2}
                       };
```

As with other array initialization, the C compiler will automatically set the number of entries in an array of structures to the number of initializers if the size is omitted.

5.2.4 Structure Pointers and the Member Access Operator

As with any object in C, you can create a *pointer* to a **struct**. The definition

```
struct household *p;
```

defines the variable **p** as a pointer to a **struct** of type household. This definition makes sense only if the template for the structure tag **household** has already been declared.

A pointer to a **struct** can be used like any other pointer. For example, if **a** has been defined as a **struct** of type **household**, then the statement

```
p = &a;  /* p set to address of struct a */
```

sets the value of **p** to the address of **a**. Once **p** is set to the address of **a**, we can use the *member access operator* as another way of accessing the members of a structure. This operator, also called the *arrow operator*, is denoted as **->** (a hyphen followed immediately by a greater than symbol). Thus, assuming that the variables **a** and **p** are declared as above and **p** is assigned the address of **a**, the statements

```
/* fragment showing use of pointer to struct */

p->total = 3;
p->adults = 1;
printf("Number of adults is %d\n", p->adults);
scanf("%f", &p->income);
```

are all valid in C.

You can also use pointer arithmetic with pointers to structures. For example, the following fragment of C code would read in values for 10 different households into an array of structures.

```
/* fragment using address arithmetic with
   pointer to structure */
/* tag household is assumed to be declared earlier */

struct household harray[10];
struct household *p;

for(p=harray; (p-harray)<10; p++)
  scanf("%d %f %d", &p->total, &p->income, &p->adults);
```

You can also use the dereferencing operator (*) with a pointer to a **struct**. For example, if **p** points to a **struct** of type **household**, then the statement

```
(*p).total =  7;
```

sets the member **total** of the structure pointed to by **p** to 7. The parentheses are necessary because the dot operator(.) has higher precedence than the dereferencing operator (*). Omitting the parentheses is an error because the dot operator requires that the left-hand side be a structure, not a pointer to a structure.

In fact, the notation

```
(*p).total
```

is entirely equivalent to

```
p->total
```

The arrow notation is used far more often than the combination of the dereferencing and dot operators because it is more compact, clearer, and avoids the potential for forgetting the parentheses.

The member access operator is in the group of operators with the highest precedence. It associates from left to right. For example, suppose that we have two structures defined as follows:

```
/* two structure templates */

struct type1 {
     double a;
     double b; };

struct type2 {
     int c;
     struct type1 *d;};
```

In this example, member **d** of **struct type2** is a pointer to a **struct** with template **type1**. In this case, we might have the following C fragment:

```
/* fragment using member access operator */

struct type1 s;     /* a type1 struct */
struct type2 t;     /* a type2 struct */
struct type2 *pt;   /* a pointer to a type2 struct */

pt = &t;            /* pt is address of t */
t.d = &s;   /* member d of t assigned address of s */
pt->d->a = 17.5;    /* equivalent of s.a = 17.5 */
```

The three variable definitions create two variables which are structures and a pointer to one type of structure. The first assignment statment in this example assigns **pt** the address of the **type2** structure **t**. The member **d** of structure **t** is then assigned the address of **s**. Thus, member **d** in **t** points to structure **s**. The third assignment statement assigns member **a** of **struct s** the value 17.5. This is equivalent to the statement

```
s.a = 17.5;  /* equivalent to pt->d->a = 17.5 */
```

5.2.5 Allowable Operations on Structures

While C allows you to define a variable that is a **struct**, not all operations can be performed on such variables. More specifically, C allows you to perform only the three following operations on **structs**:

- Accessing a member of a structure.
- Taking the address of a structure.
- Copying a structure for the purposes of:
  - Assigning one structure to another of the same type.
  - Passing a structure as an argument to a function.
  - Returning it as the result of a function.

The third of these allowed operations, copying a structure, allows you to have a **struct** on the left-hand side of an assignment statement. For example, imagine that we are using the **student struct** defined earlier in a program that keeps records of all students registered at a university. Assuming that the template for **student** were already defined, we might have two such structures called **old_student** and **new_student**. If the variable **new_student** is to be used to store information about the sister of the **old_student**, then much of the data about the **old_student** could be used in filling in the members of the **new_student**. For example, we might use the code fragment

```
/* code fragment showing assignment of structs */

struct student new_student, old_student;
    .
    .
    .
new_student.family = old_student.family;

scanf("%d %d %f", &new_student.idnum, &new_student.age,
          &new_student.gpa);
```

The statement assigning the **struct household** from **old_student.family** to **new_student.family** copies all the members of the

structure `old_student.family` into the corresponding members of `new_student.family`. This avoids the need to write separate statements to copy each member one at a time. It also allows us to change the template for the `household struct` and still have this fragment of code do what we want.

We explore the use of `struct`s as function arguments and returned values in the next subsection.

5.2.6 Using Structures as Function Arguments and Returned Values

Note that C does not allow you to do arithmetic or logical operations on `struct`s even when such operations make sense. For example, we could define a template for a `struct` which is a point in three dimensions as follows:

```
/* struct template for a point in three dimensions */

struct point3D {
  double x;    /* x coordinate */
  double y;    /* y coordinate */
  double z;    /* z coordinate */
  };
```

Given two structures `p1` and `p2`, the operation `p1+p2` makes logical sense but is not allowed in C.[1] Instead, you would have to write a function in C that had two `point3D` structures as arguments and returned a `struct` of type `point3D`. You could then invoke this function whenever you wanted to add two points together as shown in the following example. (The code below assumes that the structure template for `point3D` has already been declared.)

```
/* function that adds two structs representing
      3D points and returns the sum as a struct */

struct point3D add_points(
  struct point3D a,
  struct point3D b)
{
  struct point3D sum; /* holds sum of points */
  sum.x = a.x + b.x;  /* add x coordinates */
  sum.y = a.y + b.y;  /* add y coordinates */
  sum.z = a.z + b.z;  /* add z coordinates */

  return(sum);        /* return sum */
}
```

[1]C++, an extension of the C language, provides direct support for defining operations such as `p1+p2`. This is discussed in Chapter 9.

This function could then be used as in the following program. For brevity we have not repeated the declaration of the template for the **struct point3D** and the function **add_points()**.

```
/* main() function showing use of function
   involving structs */

#include <stdio.h>

   /* code for template of point3D and the
      implementation of add_points() assumed
      to be here */

/* function prototype for add_points() */
struct point3D add_points(struct point3D,
                          struct point3D);

main()
{

/* two points to be added */
  struct point3D p1  = {4.0, 5.0, 7.4};
  struct point3D p2 =  {-3.5, 7.2, 1.6};

  struct point3D result;  /* result of addition */

  result = add_points(p1, p2);
  printf("Result is (%lf,%lf,%lf)\n", result.x,
         result.y, result.z);
}
```

When a **struct** is used as an argument to a function, the *entire contents of the structure* are copied during the function invocation. This is quite different from how arrays are passed as arguments, where only the address of the first element is passed. Thus, the function **add_points()** above receives a complete copy of each of the two structures that are its arguments. If it modified these copies (which it does not), the changes would not affect the values of the arguments in the function which called **add_points()**.

Copying the entire contents of a **struct** as part of the function invocation process can be time consuming, particularly if the **struct** has many members. Frequently, a function that has one or more **structs** as its arguments does not need complete copies. For this reason, it is far more common for functions to use *pointers* to a **struct** rather than a **struct** itself as arguments. It is almost always faster to pass a single pointer to a function than it is to pass a complete copy of a structure.

For example, the function **add_points()** could be implemented using pointers to structures for the arguments. A revised version, named **add_points_new()**, would

be as follows:

```
/* function uses two pointers to structs representing
      3D points and returns the sum as a struct */

struct point3D add_points_new(
  struct point3D *pa,  /* first pointer to struct */
  struct point3D *pb)  /* second pointer to struct */
{
  struct point3D sum; /* holds sum of points */
  sum.x = pa->x + pb->x;  /* add x coordinates */
  sum.y = pa->y + pb->y;  /* add y coordinates */
  sum.z = pa->z + pb->z;  /* add z coordinates */

  return(sum);          /* return sum */
}
```

The function prototype for this new version and a typical invocation are given in the following code fragment.

```
/* code fragment showing passing pointers to structs */

/*function prototype */
struct point3D add_points_new(struct point3D *,
                              struct point3D *);

/* typical use of new function */
struct point3D p1  = {4.0, 5.0, 7.4};
struct point3D p2 =  {-3.5, 7.2, 1.6};

struct point3D result;  /* result of addition */

result = add_points_new(&p1, &p2);
```

Note that the address operator (`&`) is used in the function invocation to create a pointer to each of the `structs`. Also, the type of each argument given in the function prototype is a pointer to a `struct` of type `point3D`.

5.3 PROGRAMMING PROJECT: A STACK

C

In this section we describe and implement a simple version of a data structure called a *stack*. This is probably the simplest form of data structure that is widely used. Stacks are data structures to which entries can be added or removed. Adding an entry to a stack is referred to as *pushing* it on the stack. Removing an entry is referred to as *popping* the stack. Stacks use *last in, first out* discipline to determine the order in which previously pushed entries are popped.

Figure 5.1 illustrates how a stack works. Part (a) of the figure shows a stack containing three names: `"Deborah"`, `"Amy"`, and `"David"`. Part (b) shows what happens when the stack is popped: the most recently entered name, `"Deborah"`, is removed from the stack. Part (c) of the figure illustrates the pushing of `"John"` onto the stack. Since `"John"` is the last name added, it would be the first one removed if the stack were popped.

In the somewhat contrived example in this section, we will consider a situation where airline passengers arrive at an airport to board some flight. When these passengers check in, their arrival is assumed to be recorded in a computer system. Because the flight may be overbooked, we may need to *bump* some passengers off the flight after they have arrived and checked in. For the purposes of this example, we assume that the passengers who checked in last are the ones bumped off the flight.

While this is not exactly how airlines operate in overbooking situations, we will use it as an example of a programming problem where a stack might be useful. We will use a stack to store the arriving passengers. When a passenger checks in, he or she will be pushed onto the stack. At boarding time, we might then have to pop off the

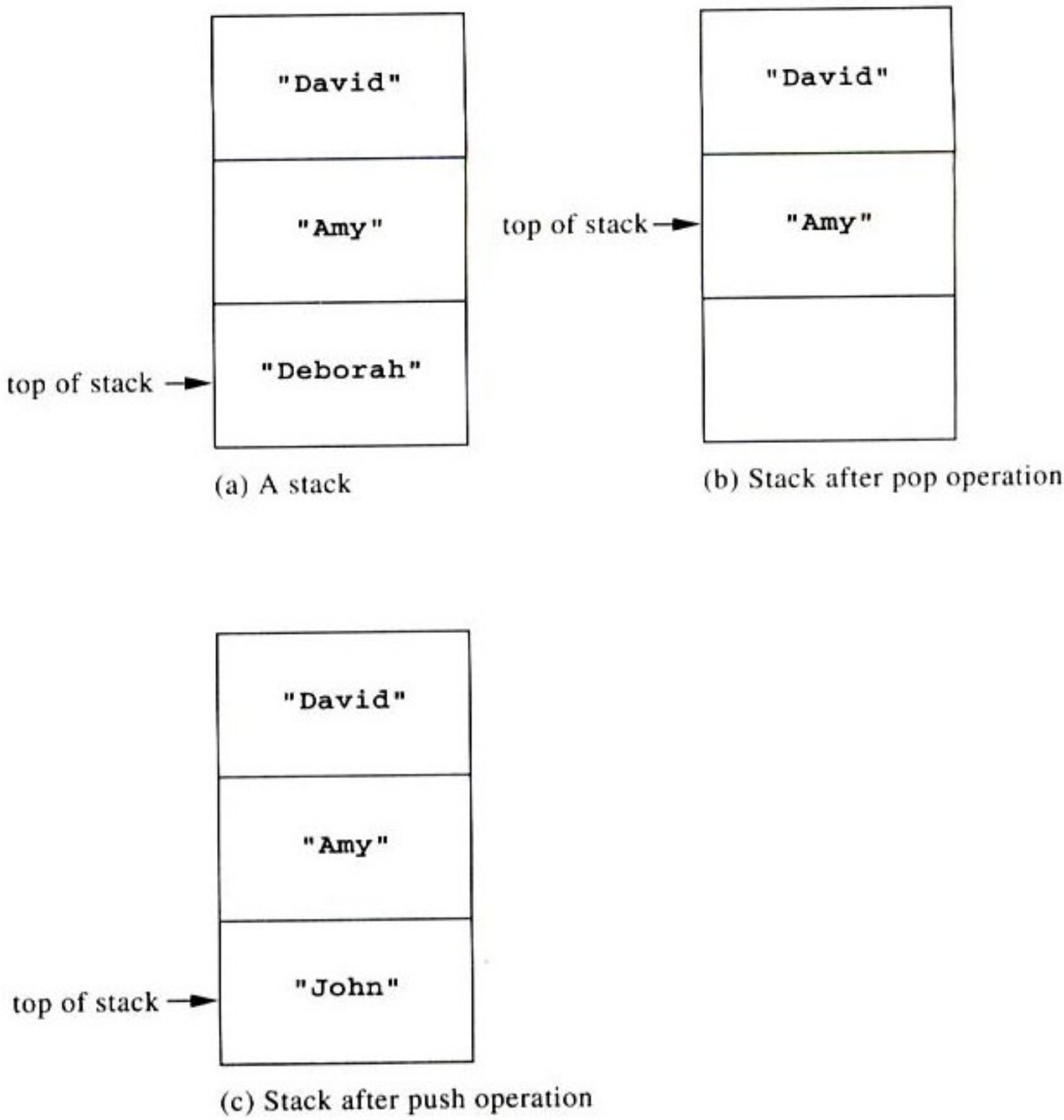

Figure 5.1 Example of a stack.

passengers who arrived most recently to reduce the total number of passengers to the number of seats on the aircraft.

We begin our implementation by declaring the template of a **struct** named **pass** to hold passenger information. We will store the first 80 characters of the passenger's name, the number of items of luggage (up to the allowed limit of two pieces), and the tag numbers of the luggage. This will allow us to unload the luggage of the bumped passengers from the aircraft. We will assume that luggage tag numbers are always positive, so that values of zero are entered for the tag numbers when there is no luggage. The template for this **struct** is as follows:

```
/* template for passenger struct */

struct pass {
  char name[80];   /* name of passenger */
  int pieces;      /* number of pieces of luggage */
  int tag1;        /* luggage tag number 1 */
  int tag2;        /* luggage tag number 2 */
  };
```

Another **struct** will be used to store the actual stack. The stack will have some maximum capacity which we will call **MAXSTACK**. The template for this **struct** (which we will call **stack_pass**) is given below.

```
/* template for a stack of passengers */

#define MAXSTACK 300  /* maximum  size of stack */

struct stack_pass {
  int count;  /* number of entries in stack */
  struct pass entries[MAXSTACK];
  };
```

The member **count** is a count of the number of elements in the stack. It should be initialized to zero, denoting an empty stack. (We show a more elegant way to initialize data structures in Section 5.9.) The array named **entries** is an array of **structs** of type **pass**. It will be used to hold the information about passengers in the stack.

The next step is to implement a set of functions that manipulate the stack. We will develop four distinct functions:

- **init_stack()**—initializes the stack
- **push()**—pushes a new passenger on the stack
- **pop()**—pops a passenger from the stack
- **give_count()**—returns the number of passengers on the stack

The four functions will all have a pointer to a stack as their first argument. Using a pointer rather than the stack itself will vastly reduce the amount of work done in copying the arguments when the functions are invoked. **give_count()** will return an

int which is the number of items currently on the stack. **push()** will have a pointer to a **struct** of type **pass** to be pushed on the stack as an additional argument and will have no returned value. **pop()** will return a **struct** of type **pass** that was popped off the stack.

Both **push()** and **pop()** have to check for error conditions. It is not permissible to try to pop something from a stack which is empty or to push something on a stack that is full. In this implementation we will treat attempts at illegal pushes and pops as fatal errors and exit from the program. In Section 5.9 we develop a more general implementation that handles these situations in a much more elegant way.

The implementations of **push()**, **pop()** and **give_count()** are listed below. They assume that the templates for **pass** and **stack_pass** are as given above.

```
/* function to initialize the stack */
void init_stack(struct stack_pass *s)
{
  s->count = 0;
}


/* function to give count of stack entries */
int give_count(struct stack_pass *s)
{
  return(s->count);
}

/* function to pop from stack */

struct pass pop(struct stack_pass *s)
{

/* If stack isn't empty, pop it */
  if(s->count > 0) {
    s->count--;
    return(s->entries[s->count]);
  }
  else {    /* Output error message and exit */
    printf("Error-Attempt to pop empty stack\n");
    exit(-1);
  }
}

/* function to push on stack */

void push(struct stack_pass *s, struct pass *p)
{

/* If stack isn't full, push item on it */
  if(s->count < MAXSTACK) {
```

```
        s->entries[s->count] = *p;
        s->count++;
    }
    else {    /* output error message*/
        printf("Error-Attempt to push on full stack\n");
        exit(-2);
    }
}
```

A few things are worth noting about these implementations. The function **exit()** is part of the standard C library. It causes a program to terminate. The value of its integer argument is returned to the operating system, or what is often referred to as the *command shell*, which processes operating system commands. In some operating systems, the value returned by **exit()** can be used by the command shell to handle error conditions in some way.

When an item is popped from the stack, **s->count**, the count of the number of entries in the stack, is first decremented by one. The element **s->count** in array of passenger structures is returned in the statement

```
return(s->entries[s->count]);
```

The decrement of the count is done first because array subscripts in C begin with 0. Information about the passenger who arrived first is stored in element 0 of the array, not element 1.

In the implementation of **push()**, the pointer to the passenger structure is dereferenced in the assignment statement

```
s->entries[s->count] = *p;
```

The right-hand side of this assignment, ***p**, is the dereferencing of the pointer **p**, which produces a **struct** of type **pass**. This is assigned to element **s->count** of the array **s->entries**.

The last step in the program development is to create the **main()** program that accepts the input of the passengers as they arrive using **scanf()** and pushes them on the stack. The user is first prompted to input the capacity of the airplane. The user then enters information for each passenger who checks in; each passenger is pushed onto a stack. When the arrival gate for passengers is closed, the user of the program enters the appropriate keystrokes so that **scanf()** returns **EOF**, the "end of file" indicator (see Section 2.14). The program then checks if it is necessary to bump passengers off the flight. If so, it uses **pop()** to list the passengers who arrived last.

The implementation of **main()** along with the appropriate **#include** statements and function prototypes are given below. We assume that the templates for the **pass** and **stack_pass structs** have already been declared.

```
/* function prototypes for stack management */
void init_stack(struct stack_pass *);
```

```
int give_count(struct stack_pass *);
void push(struct stack_pass *, struct pass *);
struct pass pop(struct stack_pass *);

/* main program for example */
main()
{
  struct pass p;    /* holds a single passenger */
  struct stack_pass stack; /* the stack */
  int flag=!EOF;    /* indicator when EOF reached */
  int seats;        /* number of seats on plane */

/* initialize stack and get capacity of aircraft */

  init_stack(&stack);
  printf("Enter number of seats on plane:\n");
  scanf("%d", &seats);

/* read in the passengers until EOF found */

  while(flag != EOF) {
      flag = scanf("%s %d %d %d", p.name, &p.pieces,
                  &p.tag1, &p.tag2);
      if(flag != EOF) push(&stack, &p);
  }

/* pop passengers who are bumped */

  while(give_count(&stack) > seats) {
    p = pop(&stack);
    printf("Passenger %s is bumped. Remove luggage
           %d and %d\n", p.name, p.tag1, p.tag2);
    }
}
```

The **main()** program initializes the stack and reads in the capacity of the aircraft. It then reads in the data for the various passengers. If there are fewer than two items of luggage, it is assumed that zeros are entered for the corresponding tag numbers. The variable **flag** is used to test whether the end-of-file is reached. Once **scanf()** returns the end-of-file indicator **EOF**, the extra passengers are popped off the stack until the number allowed to board is less than or equal to the seating capacity of the aircraft.

5.3.1 An Aside on Modularity in Software Design

Note that the functions used to initialize and manipulate a stack in the example above are quite simple. One might reasonably ask, "Why bother to create separate functions for these operations rather than using statements in the **main()** program that

access the internal structure of the stack?" The answer to this question is crucial to understanding the usefulness and importance of building large software systems out of simple, well-defined modules.

By using only functions to initialize and manipulate the data structure, we can separate the *use* of the stack functions from the *internal structure* of the stacks. As long as we keep the function names and arguments unchanged, we are free to alter how the data structure is implemented. While this may seem artificial in our simple example, it is extremely important in larger software systems. It allows one programmer to work on the implementation of a data structure while other programmers use these data structures in their parts of the software. As long as the two parts of the programming team agree on the functions that control the data structure, the implementer of the data structure library is free to change the implementation at any time. In effect, the functions become a *contract* between the data structure implementer and the rest of the programming team. How that contract is fulfilled can change without affecting the rest of the team as long as the functions do what is agreed upon.

Figure 5.2 illustrates how this separation of function interface and implementation works. One part of the programming team (Group A in the figure) defines a set of functions that operates on the data structure. The other part of the team (Group B) operates on the data structures *only* by using the interface functions. This allows changes in the internal details of the implementation to be made by Group A independently from the rest of the software development team. However, it requires that programmers of the data structure functions provide a complete and reliable set of interface functions, and that the other part of the team avoid the temptation to make use of the internal structure of the current implementation of the data structure. This discipline is crucial to the software design and implementation process for large projects. Failure to enforce such discipline is a major reason why software projects often take longer than projected and have unexpected errors.

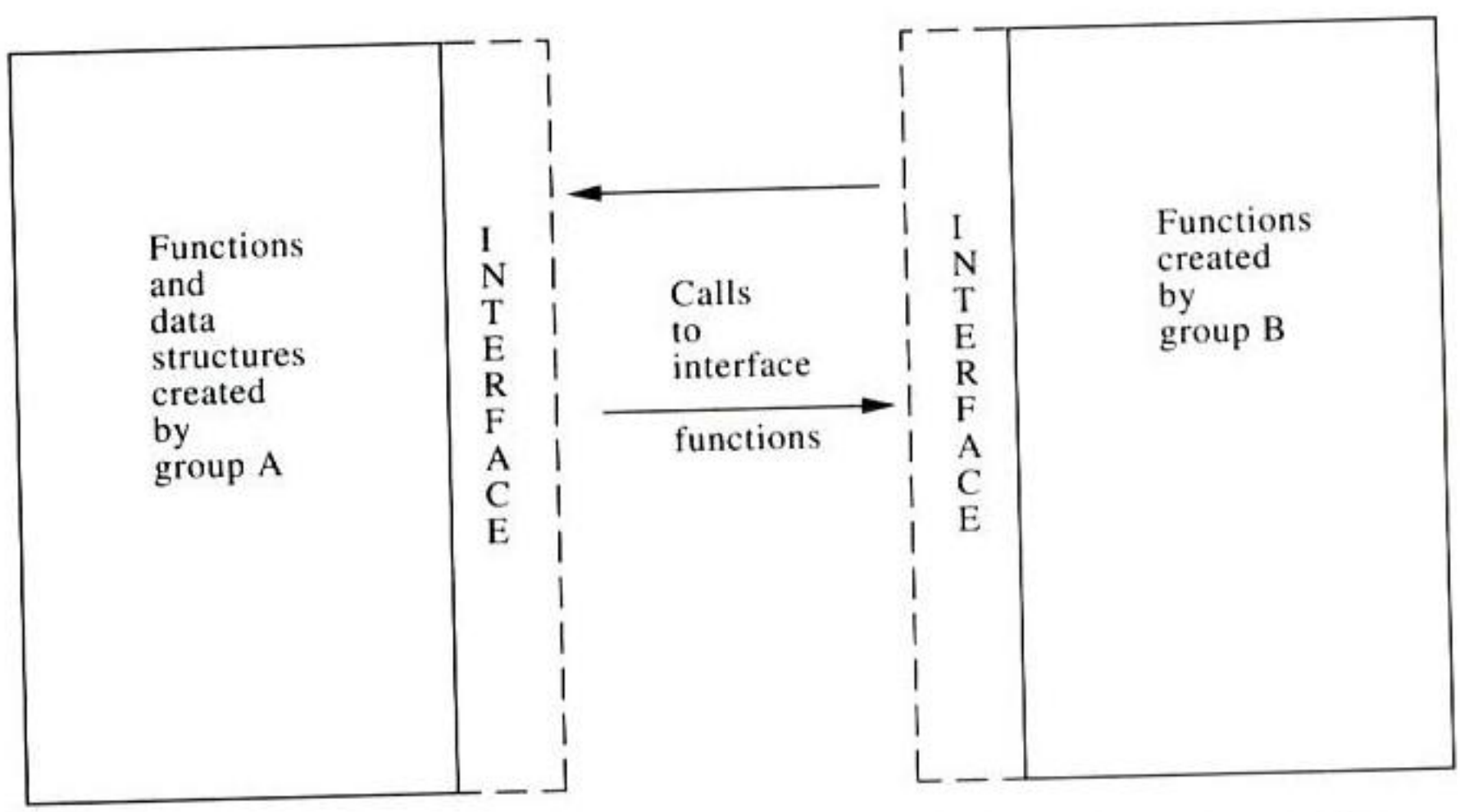

Figure 5.2 Programming interfaces in software design and implementation.

Even when a software project is small enough to be done by a single individual, the discipline of "walling off" some aspects of a program and defining a function interface to it can be invaluable. It allows a programmer to focus his or her attention on one aspect of a large program without worrying about the other aspects. It also makes detection of errors and later maintenance of the software far easier. We will encourage this style in our examples throughout this and subsequent chapters.

5.4 DYNAMIC MEMORY ALLOCATION AND VOID POINTERS

C

The implementation of a stack given above has several shortcomings. First, it requires that the maximum size of the stack (**MAXSTACK**) be fixed. In more realistic applications, there would be many aircraft with different seating capacities. Requiring that the standard stack be dimensioned to the largest possible aircraft would waste enormous amounts of storage. In addition, a real airline would have periods where many aircraft would be boarding. Each one would require a separate stack. At some times of the day, we might need many stacks, while at others we would need just a few. Ideally, we would be able to create stacks *dynamically* and have those stacks be as large or as small as needed. This will require what is called *dynamic memory allocation*.

In all the programs up until this point in the book, arrays and structures had to be declared as part of the program. Memory for these variables was allocated prior to the program's execution. The amount of storage needed by these variables had to be specified before the program was executed or was allocated automatically when functions were invoked. Dynamic memory allocation allows you to *control* the allocation of memory while the program is running. In this way, the amount of storage you need can reflect inputs from the user. In addition to permitting you to allocate memory as a program executes, you can also free up memory that you previously allocated but no longer need. This allows the memory to be reused for other purposes later in the program's execution.

One way to think of dynamic memory allocation is to imagine that there is a pool of unused memory available to a C program while it is executing. This pool is managed by a group of functions in the standard C library. Every C compiler provides these functions for dynamic memory allocation. Of course, the size of the memory pool and the internal details of how it is managed will vary across different computers and compilers. All the C programmer needs to care about is the programming interface to the library of functions used to access this pool. In this section we explore the most useful functions in this library.

5.4.1 Void Pointers and malloc()

Most pointers in C point to objects of a particular type, such as an **int**, a **float**, an array, or a **struct** with some known template. C also allows you to define a general-purpose pointer called a **void** *pointer*. Pointers to **void** are used extensively by the library functions that manage dynamically allocated memory. A pointer to **void**

can be declared just like any other pointer, as in the declaration

```
void *vp;
```

The standard library function **malloc()**, the simplest dynamic memory function, returns a pointer to **void** that is the starting address of the allocated memory. You give as an argument the number of bytes of memory you want, and **malloc()** returns a **void** pointer to that memory. If it cannot allocate the memory requested, **malloc()** returns the pointer value **NULL**. The memory provided by successful invocations of **malloc()** is uninitialized. This means that its contents are entirely indeterminate.

Pointers to **void** can be assigned to pointers of any type. For example, the following code fragment calls **malloc()** to allocate 1000 bytes of memory, and assigns the **void** pointer returned by **malloc()** to a pointer to an **int**. For the moment, we assume that each **int** occupies four bytes. (In the next subsection we show how to avoid the need to make such machine-dependent assumptions.) It then reads 250 **int**s, using pointer arithmetic to compute the address into which each **int** should be read. The **void** pointer returned by **malloc()** will automatically be converted to a pointer to an **int** by the assignment operator.

```
/* example using void pointer */

#include <stdio.h>
#include <stdlib.h>

int *pn;              /* pointer to int */
int i;                /* an int used as a counter */

pn = malloc(1000);

if(pn != NULL)
  for(i=0; i<250; i++)
    scanf("%d", pn+i);
```

The header file **stdlib.h** contains the function prototypes for the dynamic memory allocation functions. Thus, any program using dynamic memory allocation should use the preprocessor directive

```
#include <stdlib.h>
```

before any of the memory management functions are invoked. We will place this directive at the beginning of our source code files along with the other **#include** directives.

Note that we check whether **malloc()** was successful in allocating the requested memory by making sure that the pointer it returned is not **NULL**. Attempting to access memory that was not allocated successfully is a grievous error, resulting in symptoms

ranging from programs that abort abruptly to those that bring the entire computer to a halt. *You should always check to make sure that the memory you requested could be provided before you attempt to access that memory.*

Pointers to **void** can be thought of as generic pointers. They are often used when you have to manage pieces of memory without knowing the type of object that will be stored in it.

5.4.2 The sizeof Operator

C provides a special operator to assist in computing the amount of memory you will need to allocate for a particular purpose. For example, C does not standardize how many bytes are used to store an **int** or a **double**. If you want to allocate enough memory to hold **n ints**, you need to know how many bytes a single **int** requires. The **sizeof** operator provides this.

For example, suppose that you wanted to generalize the code fragment in the preceding subsection to make it into a function that allocates space for **n ints** and reads values using **scanf()** into those **ints**. The function could then return a pointer to the beginning address of the array of **ints**. This pointer could then be used just the way you might use the beginning address of a regular array. The advantage is that with an array you have to know the size of the array when you write the program. Using dynamic memory allocation, you can create the array during the program's execution.

You can use the **sizeof** operator and **malloc()** to accomplish this as follows:

```
/* function to allocate n ints
   and read values into them */

#include <stdio.h>
#include <stdlib.h>

int *make_ints(int n)
{
  int *pn;              /* pointer to int */
  int i;

/* allocate memory */
  pn = malloc(n*sizeof(int));

/* read values into the allocated memory */
  if(pn != NULL)
    for(i=0; i<n; i++)
      scanf("%d", pn+i);

return(pn);
}
```

Note that we again check whether **malloc()** returned a **NULL** pointer indicating that it failed to allocate the requested memory.

We explore the potential application of dynamic memory allocation and the **sizeof** operator in generalizing some of the linear algebra functions from Chapter 3 in the exercises at the end of this chapter.

The **sizeof** operator can be used with any C type, including any pointer or **struct** template. You can also use **sizeof** to find the number of bytes occupied by any defined variable, including arrays or structures. For example, the fragment

```
double mat[20][40][10];
printf("Bytes in mat = %d\n", sizeof(mat))
```

will output the number of bytes in the array.[2]

The **sizeof** operator has lower precedence than the structure member (**.**), array element (**[]**), function invocation (**()**), and member access (**->**) operators. The parentheses after the **sizeof** keyword are optional when the operand referenced by **sizeof** is a variable rather than a variable type such as **int** or **float**.

The **sizeof** operator is extremely useful in creating programs that can be moved across different computers without having to be rewritten. It automatically accounts for differences in the storage required for objects, avoiding the need to code the number of bytes a particular machine uses to store any given data type.

5.4.3 calloc(), realloc(), and free()

There are three other functions in the standard library used in managing memory dynamically. The first of these is **calloc()**. It has two arguments: the number of elements to allocate and the size (in bytes) of each element. It allows you to allocate arrays in a more straightforward fashion than using **malloc()**.

If possible, **calloc()** allocates sufficient memory to hold the requested number of elements. It returns a pointer to **void** that points to the starting address of the requested memory. If **calloc()** cannot provide this memory, it returns the value **NULL**. Unlike **malloc()**, the memory provided by successful invocations of **calloc()** is initialized to all zero values.

For example, if you wanted to allocate an array of 25 **structs** of type **point3D** as defined in Section 5.2.6, you would do so in the following fragment of code:

```
/* allocating an array of 25 structs */

#include <stdlib.h>

/* struct template for a point in three dimensions */
```

[2]As a technical matter, the value provided by the **sizeof** operator [as well as integer arguments to the library functions **malloc()**, **calloc()**, and **realloc()**] are of an unsigned integer type called **size_t**. This issue is discussed in Appendix A (see Section A.12).

```
struct point3D {
  double x;   /* x coordinate */
  double y;   /* y coordinate */
  double z;   /* z coordinate */
  };

/* declare pointer to structure of type point3D */
struct point3D *p;

p =  calloc(25, sizeof struct point3D);
```

The pointer **p** could then be used as the starting address of an array of 25 **structs** each of type **point3D**.

The library function **realloc()** changes the size of a previously allocated piece of memory. It has two arguments: a pointer to the beginning address of the previously allocated memory, and the new size (measured in bytes) requested. It returns a pointer to **void**, which is the starting address of the new memory.

realloc() will not change the entries in the existing part of memory up to the minimum of the old and new sizes. For example, if you had previously allocated memory for 10 **doubles** and then invoked **realloc()** requesting space for 5 **doubles**, the first five entries in the memory pointed to by the value returned from **realloc()** would be the same as the first five entries in the original memory. Similarly, if you used **realloc()** to get space for 15 **doubles**, the first 10 entries in the memory returned would be the same as in the original memory.

If the requested size of memory is larger than the old size, then the additional memory provided by **realloc()** is not initialized. If **realloc()** cannot provide the new memory, it leaves the old memory unchanged and returns **NULL**.

Suppose that we had allocated memory for 25 **structs** of type **point3D**, and we later needed to store the coordinates for 30 points. If **p** pointed to the memory originally allocated in the fragment above, we could increase the size of this in the following statement:

```
struct point3D *q;

q = realloc(p, 30*sizeof struct point3D);
```

Note that if **realloc()** is successful, the first 25 entries in the memory pointed to by **q** will be the same as those pointed to previously by **p**. Also, it is important to check whether **q** is **NULL** before we attempt to use it in case **realloc()** was unable to provide the requested memory. When **realloc()** fails, the original memory pointed to by **p** is unchanged. As with **malloc()**, the functions **calloc()** and **realloc()** always provide a block of contiguous memory locations. This memory can therefore be treated as an array.

The last memory allocation function is **free()**. It deallocates memory that was previously allocated by **malloc()**, **calloc()**, and **realloc()**. It has a single argument which is a pointer to the memory allocated previously. **free()** does not return a value.

For example, if you no longer needed the memory pointed to by **q** in the example above, you could return it to the memory allocation system for later reuse with the following statement:

```
free(q);
```

It is an error to try to use the functions **free()** or **realloc()** on a segment of memory that was not previously allocated.

5.5 RAGGED ARRAYS IN C

One data structure that gets widely used in text processing is called a *ragged array*. Consider, for example, an application in which we need to store a series of words. Since words vary in length, it would be inefficient to create an array of character arrays each long enough to store the longest possible word. Instead, we generally create a data structure to hold the words. This data structure is an array of pointers to **char**. Each pointer in the array points to a character array that is allocated dynamically when the size of the word to be stored is known. Figure 5.3 illustrates how such an array would be organized.

To create a ragged array from a series of words input by a user, we use a character array (called a *buffer*) with some fixed length equal to the longest string we are likely to encounter (plus one additional array element for the null character). After we input a word into the buffer, we determine its length using the standard library function

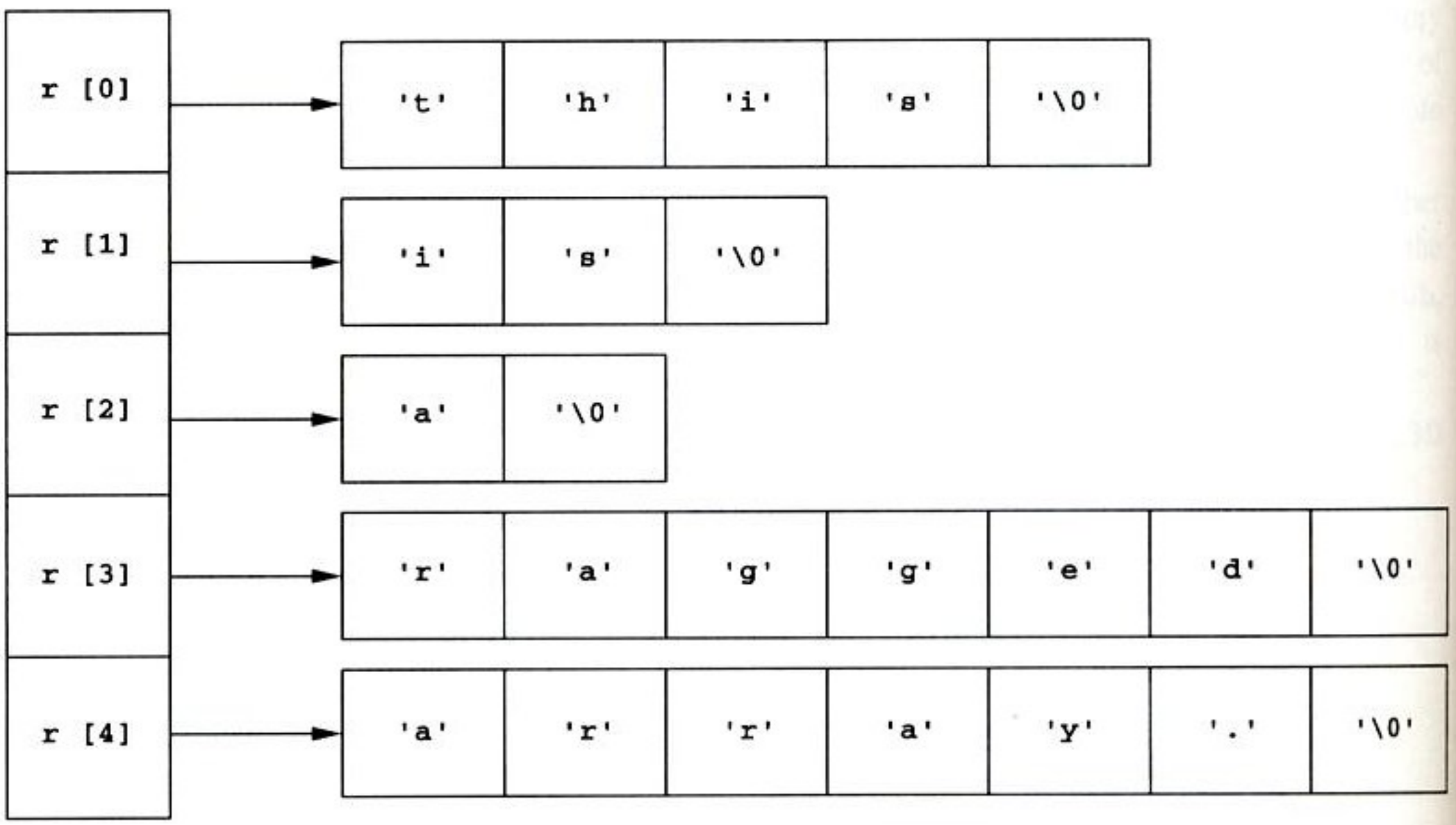

Figure 5.3 Ragged array.

strlen(), allocate memory for the word using **malloc()**, and then copy the word from the buffer into the newly allocated memory.

The function **ragged_array()** given below creates a ragged array from a series of input words (separated from each other by white space). It has as arguments **r_array**, a pointer to the beginning of the array of character pointers, and **maxwords**, the greatest number of words to be read into the array. The function returns an **int** equal to the number of words actually read into the array.

```
#include <stdio.h>  /* input/output library */
#include <stdlib.h> /* memory allocation library */
#include <string.h> /* string library */

#define LONGWORD 80 /* longest allowed word */

/* function to create ragged array of strings */
int ragged_array(
  char *r_array[], /* array of character pointers */
  int maxwords)    /* max number of words to be read */
{
  int nwords=0;       /* counter of words read */
  int flag;           /* flag for end of file */
  char buffer[LONGWORD+1];   /* buffer */

/* read in first word up to maximum allowed length */
  flag = scanf("%s", buffer);

/*loop until end of file reached */
  while(flag != EOF && nwords < maxwords) {
    r_array[nwords] = malloc(strlen(buffer)+1);
    if(r_array[nwords] != NULL){
        strcpy(r_array[nwords], buffer);
        nwords++;
    }
    else
        flag = EOF;
  }
  return(nwords);
}
```

The crucial statements in this function are

```
r_array[nwords] = malloc(strlen(buffer)+1);
```

and

```
strcpy(r_array[nwords], buffer);
```

The first of these uses **malloc()** to allocate sufficient space for the string in the buffer. Note that we add one to the value returned by **strlen()** to have space for the null character. The pointer returned by **malloc()** is assigned to the next free element of the array of character pointers, **r_array**. The second key statement uses the library function **strcpy()** to copy the contents of the buffer into the next element of the ragged array.

5.6 ARGUMENTS TO main()

C

Up until this point, we have written complete programs as though the function **main()** did not have any arguments. In fact, **main()** can have two arguments that are used to pass information between the command interpreter (or shell) being used to invoke programs and the C program itself. For example, some computer systems have a command for printing a file of the form

```
print filename
```

where **filename** is the name of the file to be printed. If the **print** command is implemented in C, there needs to be some way for the name of the file to be printed to be communicated to that C program.

A **main()** function in C actually has two arguments, which by custom are named **argc** and **argv**. The value of **argc** passed to **main()** is an **int** equal to the number of strings that appeared on the command line invoking the C program, including the name of the command program. In the foregoing example of a **print** command, the value passed to **argc** would be 2. (Each string is a series of characters not separated by white space.) The argument **argv** is a ragged array (i.e., an array of pointers to strings). This ragged array contains the values of the separate strings in the command line. In the example above, the first element in the ragged array would be the string **"print"** and the second would be **"filename"**.

The following program shows how the arguments to **main()** might be used. It simply prints out the values of the arguments to **main()**.

```
/* program to print out arguments to main() */

#include <stdio.h>

main(
  int argc,        /* number of arguments to main */
  char *argv[])    /* ragged array  */
{
  int i;           /* used for looping */
  printf("Number of arguments to main is %d\n", argc);
```

```
/* loop through elements in ragged array */
  for(i=0; i<argc; i++)
    printf("String %d is %s\n", i, argv[i]);
}
```

A more realistic use would be to pass the program user's selections among various options such as the files to be operated on. For example, many C compilers are themselves written in C. The names of the files to be compiled, the various libraries to be used, and the name of the executable file to be created are often options passed using the **argc** and **argv** arguments.

The memory in which the ragged array is stored is allocated by the command processor, not by the C program itself. All of the entries in the ragged array are complete strings, including the terminating null character.

5.7 THE typedef STATEMENT

C

Another feature of C that makes it more convenient to create and use libraries of functions that operate on data structures is the language's ability to give a name to a new type of data, and then to use that derived data type in the same way as the standard types C provides. The **typedef** statement allows you to create and name derived data types.

Suppose, for example, that you wanted to define a new data type called **Complex** to hold a complex number consisting of real and imaginary parts. We might first create a template for a **struct** for this data object as follows:

```
/* struct to hold complex number */
struct real_and_imaginary {
  double real;       /* real part */
  double imag; /* imaginary part */
};
```

Following this declaration, we could use a **typedef** statement to declare that **Complex** refers to objects of this type:

```
typedef struct real_and_imaginary Complex;
```

Once these statements appear in a source code file, we are then free to use the data type **Complex** just as we could use any other data type. For example, all of the following would be allowed:

```
/* using Complex data type */

/* prototype for function */
Complex add_complex(Complex, Complex);
```

```
/* function adding two Complex values */
Complex add_complex(Complex a, Complex b)
{
  Complex sum;
  sum.imag = a.imag + b.imag;
  sum.real = a.real + b.real;
  return(sum);
}

/* fragment dynamically allocating space
   for a Complex value */

Complex *p;  /* pointer to a Complex value */

/* use malloc to get memory */
p = malloc(sizeof (Complex));
if(p != NULL){
    p->real = 1.0;
    p->imag = 0.0;
}
```

While not required, we will adopt the convention that data types created by the **typedef** mechanism will always have names that begin with a capital letter and have the remainder of their name in lowercase. This will distinguish them from values created using **#define** preprocessor directives.

The most valuable aspect of the **typedef** capability is that it allows you to define new data types and then use them without being overly concerned with their internal structure. This is particularly useful for creating libraries of functions that manipulate data structures when the library will be used by others. You can give a library's users the appropriate **typedef** statements and let them use variables with types such as **Stacks** or **Lists** in their code. We will demonstrate the usefulness of this approach in Section 5.9 by reimplementing the stack functions in a more elegant fashion.

5.8 C PROGRAMS WITH MULTIPLE SOURCE FILES

C

C allows the programmer to combine source code from multiple files to create a single executable program. These source files (formally termed *translation units*) can contain different parts of the complete program, thereby facilitating development of large programs by more than one programmer. In addition, the programmer can compile each of the source code files separately and link them together into a single executable program at a later stage. Several rules apply to programs that are spread across multiple files.

First, variables that are defined *external* to any function can be referenced in other source files. However, since each file can be compiled independently, you must inform

the compiler that the variable is "defined elsewhere" and you must declare the type of the variable. For example, suppose that you have one source file named **a.c** with an external variable **i** defined as

```
/* definition of i in file a.c */
int i;
```

If you wanted to use that variable in another source file **b.c**, then you would have to include the following declaration in that file.

```
/* declaration of i in file b.c */
external int i;
```

When file **b.c** is compiled, there would be no memory allocated to store the value of variable **i**. Instead, the compiler generates an *external reference* to **i** which would be found in some other source code module. The location of **i** would be resolved when the complete executable program was linked together from the separately compiled parts.

Second, all functions are by default considered to be external. This means that you can use these functions in other source code files. You should, however, include the function prototype in each source code file. This is often done by creating a separate **#include** file that is included with each source code file. For example, you might have a **#include** file named **c.h** which has all the function prototypes for the complete program. This would typically be kept in the same directory as the source code files. You would then have the preprocessor directive

```
/* include file in the same directory */
#include "c.h"
```

at the beginning of each source code file that made up the complete program. Note that the name **c.h** is enclosed in quotes rather than less than and greater than symbols. This indicates that the file **c.h** is not one of the standard header files, and that the file's name should be found by whatever file naming conventions are used by your computer's operating system.

The third rule allows some functions not to be treated as external. To do this, you define the function to be **static**, as in the following example:

```
/* declaration of a function which
   will not be external */
static double squares(double x)
{
  return (x*x);
}
```

Functions that are defined as static can be used only in the source code file in which they are defined. In effect, they are "hidden" from the other source code files.

Such functions are most often used when you are creating libraries of functions for someone else and you do not want that person to invoke some of the functions you use internally in your library.

5.9 PROGRAMMING PROJECT: STACKS REVISITED

We can now use the dynamic memory allocation functions, the **typedef** feature, and the ability to create separately compiled collections of functions to create a more general stack manipulation library. In particular, the new implementation will allocate memory for stacks dynamically and will automatically extend the size of those stacks when needed. In addition, the stack library will return error codes when unexpected conditions arise rather then simply terminating. This allows the programmer using the library to take whatever action is appropriate in the application.

The code for the stack manipulation is more complicated than the example presented in Section 5.3. The additional complexity of the implementation is offset by added generality of the stack functions. This tradeoff is often favorable because data structure manipulation functions can be used in many parts of a large software project. It therefore makes sense to invest the time to code these functions in a very general style.

Our implementation will create a separate header file **stack.h** and a file named **stack.c** containing the functions to manipulate the stack. The assumption is that users of the library will **#include** the header file in their programs, eliminating the need for reentering the various structure template declarations and the function prototypes. The following is a listing of the contents of the header file **stack.h**.

```
/*  header file stack.h */

/* template for passenger struct */
struct pass {
  char name[80];   /* name of passenger */
  int pieces;      /* number of pieces of luggage */
  int tag1;        /* luggage tag number 1 */
  int tag2;        /* luggage tag number 2 */
  };

/* template for a stack of passengers */
struct stack_pass {
  int count;   /* number of entries in stack */
  int size;    /* maximum size of stack */
  int inc;     /* increment by which stack grows */
  struct pass *entries; /* pointer to the last */
  };                    /* item in stack */

/* typedef for stack of passengers */
typedef struct stack_pass * Stack;
```

```
/* prototypes for stack functions */
Stack make_stack(int, int);
int give_count(Stack);
int push(Stack, struct pass *);
int pop(Stack, struct pass *);
void destroy_stack(Stack);

#define TRUE 1
#define FALSE 0
```

The **struct** with the template **pass** is the same as in our earlier implementation. However, the **struct** template for **stack_pass** now has four entries:

- **count**—the number of items in the stack.
- **size**—the maximum number of items that currently can be held in the stack.
- **inc**—the number of items by which the stack should grow if it needs to be extended.
- **entries**—a pointer to an array of structures of type **pass** containing the information about stacked passengers.

Figure 5.4 illustrates how the stack structure is organized. It shows a stack that has capacity for 50 passengers. There are three passengers named **"Jefferson"**, **"Hancock"**, and **"Adams"** currently in the stack, with **"Adams"** at the top of the stack. The stack will be incremented by enough space to hold 10 additional passengers each time it fills up.

The **typedef** statement defines a **Stack** to be a pointer to a **struct** of type **stack_pass**. This will allow us to declare variables as **Stack**s in our programs.

The header file also gives the function prototypes for the five stack manipulation functions. These functions are as follows:

- **make_stack()**. This function has two integers as its arguments. The first gives the initial size of the stack and the second is a *growth* increment describing how many additional spaces in the stack should be added when the stack grows. **make_stack()** returns a newly created **Stack** if it can successfully allocate memory for it or the value **NULL** if it fails to allocate the requested memory.
- **give_count()**. Takes a **Stack** as its argument and returns the number of items currently in a **Stack**.
- **push()**. This function pushes a new passenger on a **Stack**. Its two arguments are the **Stack** onto which the passenger is to be pushed and a pointer to a **struct** of type **pass** holding the data for the passenger. **push()** returns an **int** which will have a value which is **TRUE** (i.e., nonzero) if the item is pushed on the **Stack** successfully, and **FALSE** (i.e., zero) otherwise.
- **pop()**. This function pops an item off a stack. Its first argument is a **Stack** from which the passenger is popped. Its second argument is a pointer to a **struct** of type **pass** into which the passenger data is placed. **pop()** returns an **int** which will have a value that is **TRUE** (i.e., nonzero) if an item can be popped from the **Stack**, and **FALSE** (i.e., zero) otherwise.

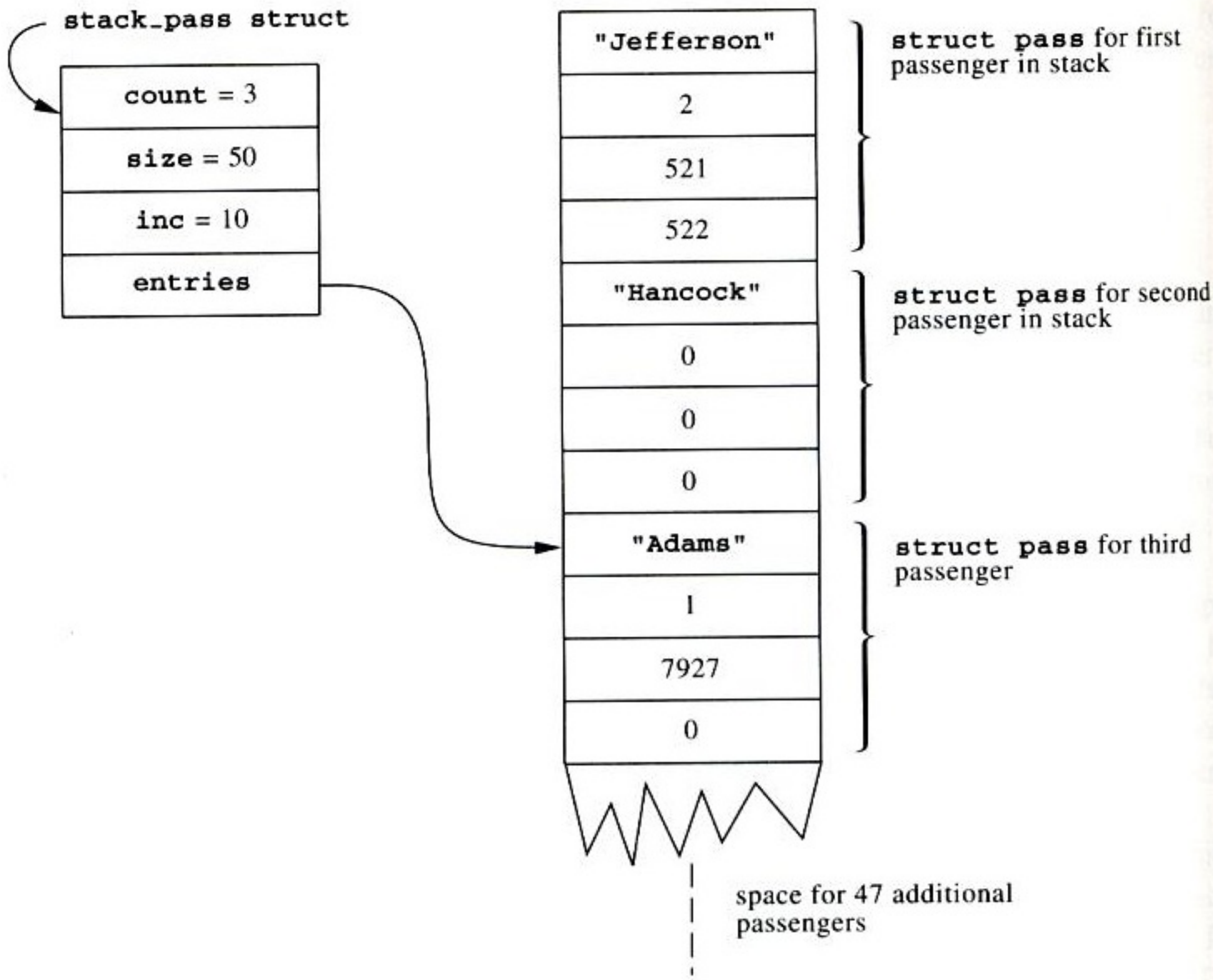

Figure 5.4 Revised implementation of a stack of passengers.

- `destroy_stack()`. Destroys a `Stack`. This function has as its argument the `Stack` to be destroyed and does not return a value.

The next file is the actual implementation of the stack functions. This is in the file `stack.c`. The implementation begins with the following `#include` statements:

```
/* #include statements for stack manipulation functions
      in file stack.c */

#include <stdio.h>
#include <stdlib.h>
#include "stack.h"
```

The C code for the function `make_stack()` follows.

```
/* function to make a stack */
Stack make_stack(
   int initial_size,    /* initial size of stack */
   int increment)       /* growth increment */
```

```
{
  Stack s;

/* allocate memory for stack */
  s = malloc(sizeof(struct stack_pass));
/* check if stack can be allocated */
  if(s != NULL) {
/* allocate memory for data in stack */
    s->entries=calloc(initial_size,sizeof(struct pass));
    if(s->entries == NULL) { /* check allocation */
      free(s);
      s = NULL;
      }
    else {
      s->size = initial_size;
      s->inc = increment;
      }
  }
  return(s);
}
```

This function begins by allocating memory for a **Stack**. If this is successful, then it allocates sufficient memory for an array of **initial_size** structures of type **pass**. Each one of these array entries can hold information about a passenger. If this memory allocation fails (as indicated by **calloc()** returning a **NULL** value), then the memory allocated for the **Stack** is freed and the **NULL** value is returned. Otherwise, the size of the stack and the size of the increments to be added when the stack fills up are set, and the **Stack** is returned.

The second function, **give_count()** is essentially unchanged from the version described in Section 5.3. The C code is as follows:

```
/* function to give count of stack entries */
int give_count(Stack s)
{
  return(s->count);
}
```

The third function, **pop()**, first checks if the stack is empty. If not, then the item is popped from the stack and the value **TRUE** is returned. Otherwise, **FALSE** is returned.

```
/* function to pop from stack */
int pop(
  Stack s,            /* stack to pop */
  struct pass *p)     /* pointer to where popped */
{                     /* data should be put */
```

```
/* if stack isn't empty, pop it */
  if(s->count > 0) {
    s->count--;
    *p = s->entries[s->count];
    return(TRUE);
  }
  else
    return(FALSE);  /* stack was empty */
}
```

Note the statement

```
*p = s->entries[s->count];
```

Since **p** is a pointer to a structure, the value of ***p** is a structure. Thus the statement assigns the complete contents of the structure **s->entries[s->count]** to the structure pointed to by **p**.

The fourth function, **push()**, must deal with the possibility that the stack is full. In this case, it must use the library function **realloc()** to extend the size of the array **entries** in the stack. As with any other memory allocation function calls, care must be taken to make sure that the additional memory could be allocated. The implementation of **push()** is as follows.

```
/* function to push on stack */
int push(
  Stack s,          /* stack to push data on to */
  struct pass *p)/* pointer to data to push on stack */
{
  struct pass *temp;
/* if stack if full,increase size of allocated memory */
  if(s->count == s->size) {
    temp = realloc(s->entries,
                   (s->size+s->inc)*sizeof(struct pass));
    if(temp == NULL)
      return(FALSE);
    else
      s->entries = temp;
  }
  s->entries[s->count] = *p; /* add to stack */
  s->count++;
  return(TRUE);
}
```

The last function destroys a **Stack** by freeing the previously allocated memory. This must be done in two steps. First, the memory allocated for the array **entries** must be freed. Then the memory for the structure containing the stack can be freed. The code for this is provided below.

```
/* function to destroy stack */
void destroy_stack(Stack s)
{
  free(s->entries); /* free memory for data */
  free(s);          /* free memory for stack */
}
```

Note that it would be an error to first free the memory pointed to by **s**. Once **s** is freed, the reference **s->entries** is no longer valid.

Once this library is complete, we can then write other programs that make use of it. A realistic application would make stacks for each arriving airplane and keep adding passengers to these stacks as they checked in. Below we show the simpler case introduced in Section 5.3 using the stack library for a single flight.

```
/* using a stack for airline passengers */

#include <stdio.h>
#include "stack.h"

main()
{
  struct pass p;        /* holds a single passenger */
  Stack  s;             /* the stack */
  int flag1=!EOF;       /* indicator when EOF reached */
  int flag2;            /* indicator for illegal push */
  int seats;            /* number of seats on plane */

/* initialize stack and get capacity of aircraft */

  printf("Enter number of seats on plane:\n");
  scanf("%d", &seats);
  s = make_stack(seats, 10);

/* read in the passengers until EOF found */

  while(flag1 != EOF) {
    flag1 = scanf("%s %d %d %d", p.name, &p.pieces,
                  &p.tag1, &p.tag2);
    if(flag1 != EOF)
      if(!push(s, &p))
        printf("Can not add passenger to stack\n");
  }

/* pop passengers who are bumped */

  while(give_count(s) > seats) {
    pop(s, &p);
```

```
        printf("Passenger %s is bumped. Remove luggage ");
        printf("%d and %d\n", p.name, p.tag1, p.tag2);
    }
}
```

5.10 LINEAR LINKED LISTS

C

A *linear linked list* consists of a series of items (often called *nodes*), where each node contains a pointer to the next node. Such data structures are frequently used in situations where the number of items to be stored varies as the program executes. A linear linked list can grow as needed, with new items added to either the front or the end of the list. When some items on the list are no longer needed, they can be removed, allowing the list to shrink. Figure 5.5a to d show how a linear linked list is organized. The figure shows a sequence of operations on an initial list, with items added to the beginning and end of the list, and an item deleted from the list.

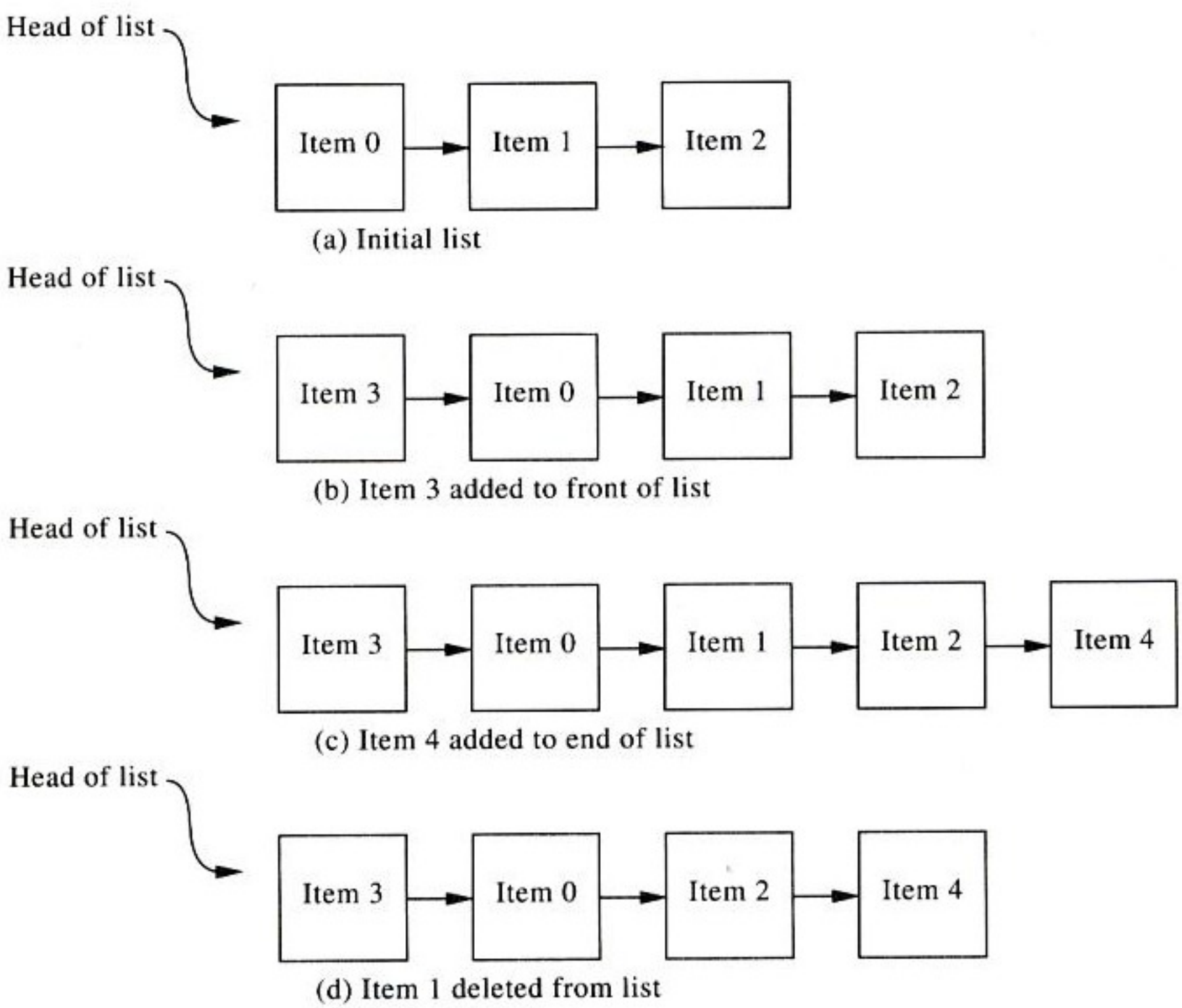

Figure 5.5 Linear linked list.

CHAPTER

6

SORTING AND SEARCHING

6.1 INTRODUCTION

A significant aspect of many problems solved using computation involve *sorting* and *searching*. Sorting is generally used to organize large volumes of data such as billing records, where having information sorted makes it easy to combine data from different sources. Sorting is also frequently used as the first step in searching for a particular item. Both searching and sorting algorithms play important roles in the implementation of computer databases.

While searching and sorting are essentially trivial for very small problems, they become time consuming for realistically sized problems, where thousands or millions of pieces of information must be organized. Knuth[Knuth, Donald E. 73a], for example, has estimated that up to a quarter of all computation time is spent on sorting. In this chapter we introduce several of the more common sorting algorithms and show how they can be implemented efficiently in C.

While sorting is an important component in solving many problems, searching is of even more obvious value. For example, in the airline reservation problem described in Chapter 5, rapid access to information on available seats on a particular flight, as well as the reservation information for specific passengers, is essential. This capability depends on fast and flexible searching algorithms. In this chapter several approaches to the searching problem are presented, including the use of a particular type of tree called

a *binary search tree*. Another technique, called *hashing*, is also described. This method uses some additional computer memory beyond the minimum necessary to speed up searching significantly.

Because searching and sorting involve processing large volumes of information that are most often to be stored permanently, it is natural to introduce the idea of *files* in this chapter as well. Files make it possible to process quantities of data that are too large to be kept in a computer's memory. This is often the case for computer databases. In addition, files are often used to store results of algorithms in a form suitable for further processing by other programs.

Files are used to represent information most commonly written to and read from secondary storage devices such as floppy or hard disks. The C programming language provides a rich set of library functions for the creation and manipulation of files. The most useful portions of this library are discussed in Section 6.2.

Sections 6.3 and 6.4 cover some of the most widely used algorithms for sorting and searching, respectively. Section 6.5 describes the use of binary search trees as data structures for rapid retrieval of information. In Section 6.6 we present an introduction to hashing methods. This is followed by a programming project in Section 6.7, where hashing is used to extend the algorithm for solving the 15-tile puzzle introduced in Chapter 5 to eliminate the possibility of cycling.

6.2 FILES IN C

Files allow the storage and processing of large amounts of data typically associated with most real problems. Beyond having capacity considerably in excess of the memory on most computers, data in files also has the property of *persistence*. That is, because files are written to secondary storage devices, data stored in files is maintained even when the computer is shut off.

Up until this point in this book we have relied on the standard C library functions **`printf()`** and **`scanf()`** to obtain input from the keyboard for our programs and to output the results. As will be made clearer below, these functions are simply restricted versions of the general file processing functions which are available in C. For example, many applications require dealing with several files simultaneously. C provides considerable flexibility in reading and writing to different files which are active at the same time.

In C a file is treated as a *stream* of characters which are processed sequentially. In addition to other, less important operations, the standard C library provides functions that allow the programmer to operate on a file in any of the following ways:

- Open it. Opening a file establishes a link between the program and the file. The open operation generally will create the file if it does not already exist. If the file exists, the open operation will determine whether the user running the C program has access privileges to the file.

- Read from it. A read operation closely parallels what happens when the **scanf()** function is invoked except that input is taken from the file rather than from the keyboard.
- Write to it. A write operation closely parallels what happens when the **printf()** function is invoked except that output is directed to the file rather than to the display.
- Close it. This breaks the link between the program and the file. Further reading or writing from the file after it is closed is not allowed unless the file is opened again.

Access to any file for these operations is through something called a *file pointer*. The file pointer data type (denoted by **FILE *** in a C variable definition) is defined in the header file **stdio.h**, most often through a **typedef** statement. In most cases, the file pointer points to a **struct** which is also defined in the **stdio.h** header file. This **struct** contains information on the file itself, including the current place in the stream of characters where processing(reading or writing) is taking place.

Since the **FILE struct** is defined in **stdio.h**, this header file must be included whenever a program includes file processing. So, for example, two file pointers, **fpin** and **fpout**, could be declared in the following code fragment:

```
#include <stdio.h>  /* defines FILE pointer */
.
.
FILE *fpin, *fpout; /* two FILE pointers */
```

After this fragment the variables **fpin** and **fpout** can be used to access files. Each of the principal functions that operate on files is described below. All of the prototypes for these functions are in **stdio.h**.

6.2.1 Opening a File

Before we can do anything with a file, we must first open it by making use of the standard C library function **fopen()**. **fopen()** returns a file pointer and takes two arguments:

- A character string which is the name of the file to be opened and subsequently processed.
- A character string indicating the *mode* in which the file may be operated on. The three most common modes of use are:
  - **"r"** opens the file for reading, beginning at the start of file.
  - **"w"** opens the file for writing.
  - **"a"** opens the file for appending (i.e., adding to the end of the file).

The effect of **fopen()** depends on several factors, including whether or not the file already exists. The most important rule in using **fopen()** is that *opening an existing file for writing results in the erasure of the current contents of the file.* One of

the most common ways to delete a file accidentally is to open it for writing when you really intended to open it either for reading or appending.

If a nonexistent file is opened for writing, then a new file with that name will be created. **fopen()** may be unsuccessful in opening the file referred to by the first argument. For example, an attempt to open a nonexistent file for reading will always fail. Another possible reason for a file opening to fail is that on some larger computer systems, one must have the correct permissions in order to open a file.[1] In these cases **fopen()** will return the **NULL** pointer. Before making use of the file pointer returned by **fopen()** it is essential to check whether the attempt to open the file has been successful.

The following fragment of code shows how to open three files, one for reading, one for writing, and one for appending:

```
/* fragment showing use of fopen() */
#include <stdio.h>
.
.
FILE *fpi, *fpo, *fpa;
char input[80];

/* read name of data file into input */
scanf ("%s", input);


fpi = fopen( input, "r");
if (fpi == NULL)
   printf("File error, unable to open file %s\n",input);
   exit (-1);
   }
fpo = fopen("results", "w");
if (fpo == NULL)
   printf("File error, unable to open file results\n");
   exit (-2);
   }
fpa = fopen("output", "a");
if (fpa == NULL)
   printf("File error, unable to open file output\n");
   exit (-3);
   }
.
.
```

As shown, this code reads in the name of a file and opens that file for reading. It then opens a file named **"results"** for writing and a file named **"output"** for

[1]Personal computer systems such as IBM PCs, PC-compatibles, and Apple computers do not generally enforce permissions except in the case of floppy disks, which can be "write locked." All multiuser computer systems, including UNIX workstations, have file access systems.

appending. As with any other variables, the file pointers **fpi**, **fpo**, and **fpa** can be reassigned values later in the program once processing the file has been completed and the file has been closed.

When a C program is started three "files" are opened automatically: **stdin**, **stdout**, and **stderr**. These are the standard devices used for input, output, and error messages, respectively. Normally, the keyboard is the standard input file and the screen is the standard output and standard error message file.

6.2.2 Reading from a File

Reading from a file is accomplished with the function **fscanf()** which is identical to **scanf()** except for the inclusion of the file pointer as a new first argument. [2] Thus, if the preceding code fragment also included the following declaration:

```
int a[100]; /* array of integers */
int i;      /* a counter */
```

we could read integer data from the file named in the character string **input** simply by adding the following code to the example above:

```
int test;  /* used as a value to test for end of file */

for (i=0; i<100; i++) {
  test = fscanf(fpi, "%d", &a[i]);
  if(test == EOF)     /* check if EOF is reached */
    exit(-4);
  }
```

As with **scanf()**, the function **fscanf()** returns an integer which is the number of items read successfully from the input file. It is important to check this integer to determine whether the end-of-file value (**EOF**) is returned from **fscanf()**. This occurs when a read operation would go beyond the last item in the file.

6.2.3 Writing to a File

Writing to a file is accomplished through the function **fprintf()**. We add a new first argument, the file pointer for the file to be written to. [3] To illustrate using the preceding code fragments, if we add the declaration

```
double b[100], c[100];
```

[2]In fact, **scanf()** is generally defined with a preprocessor macro that uses **fscanf()** with the file pointer set to **stdin**.

[3]As with reading from a file, **printf()** is usually defined by a preprocessor macro that uses **fprintf()** with the file pointer set to **stdout**.

we could write to the two files **results** and **output** by adding the following code after these files are opened successfully:

```
for (i=0; i<100; i++){
   fprintf(fpo, "%lf \n", b[i]);
   fprintf(fpa, "%lf \n", c[i]);
  }
```

As with **printf()**, the function **fprintf()** returns an integer which is the number of characters successfully written to the file or a negative value if an error occurred. The fragment above ignores these returned values.

6.2.4 Closing a File

When operations on a file have been completed, the file can be closed with the **fclose()** function with the single argument being the file pointer for the appropriate file. After a file has been closed the pointer can be recycled to access another file. Any open files are closed automatically when the program's execution ends. Referring back to the previous code, the input file could be closed by using the following statement:

```
fclose(fpi);
```

The file-closing function returns an integer which is set to **EOF** if some error occurs. This returned value is often ignored by programmers.

6.2.5 An Example Using Files

These basic file operations described above are illustrated in the following function, which simply copies one file into another on a character-by-character basis. The names of both files are passed as string arguments to the function. The function returns an **int** whose value indicates whether or not the copying has been successful.

```
/* copy in_file into out_file,
   returning an integer flag
             = 0 for a successful copy
             = 1 for failure to open in_file
             = 2 for failure to open out_file */

#include <stdio.h>

int copy_text(
  char *in_file,      /* name of input file */
  char *out_file)     /* name of output file */
{
 FILE *fpi, *fpo;
 char c;
```

```
/* open input file */
 fpi = fopen(in_file, "r");
 if (fpi == NULL) {
    printf("ERROR: couldn't open input file\n");
    return(1);
    }

/* open output file */
 fpo = fopen(out_file, "w");
 if (fpo == NULL) {
    printf("ERROR: couldn't open output file\n");
    return(2);
    }

/* copy input file to output file */
 while (fscanf(fpi, "%c", &c) != EOF)
    fprintf(fpo, "%c", c);

/* close files */
 fclose(fpi);
 fclose(fpo);
 return(0);
}
```

Note that the function **copy_text()** checks only for problems in opening files. A more rigorously coded version should check for reading and writing errors as well. For example, the function **fprintf()** may fail because the disk to which it is writing is full.

6.3 SORTING

Sorting is required in the solution of many computational problems, but perhaps its most obvious use is to facilitate searching. Without prior sorting any search must be on a sequential, item-by-item basis; this is known as a *linear search*. While a linear search may be acceptable in terms of computation time for some very small problems, it will be extremely inefficient for most applications—hence the need to sort.

Before presenting specific sorting algorithms we will define some basic terminology to be used throughout this chapter and distinguish among alternative sorting contexts which can influence choice of sorting algorithm. First some definitions:

- *Records.* These are the items to be sorted, generally data structures which include several pieces of information. In many cases these structures will be quite complex. For example, records in an airline reservation system might include: the passenger's name, home and work telephone numbers, the number of passengers traveling together, origin city, destination city, flight number,

and fare paid as well as information on associated flight legs. The size and complexity of a record can influence the best approach to sorting.

- *Keys.* These are the data attributes which are used for sorting. The *primary key* is the first attribute used in the sort, while *secondary keys* are those used to resolve ties based on the primary key. For example, in a mailing list the primary sort key might be the zipcode and the name of the individual might be the secondary key used to organize records with the same zipcode.

In this discussion we focus on *internal sorts*, in which all records to be sorted are stored in random access memory, as distinct from *external sorts*, which involve secondary storage. Our focus on internal sorting is because with the rapid decline in the cost of random access memory, more problems are now solvable with internal sorting algorithms. In addition, consideration of the bottleneck caused by the movement of data between memory and secondary storage is outside the scope of an introductory textbook.

It is also important to recognize other characteristics of sorting problems beyond the obvious issues of number of records and their individual composition. One of these characteristics is whether or not there is some structure in the initial data set. For example, a database which is being frequently updated through the addition of new records, with sorting following each new set of additions, presents a much easier problem than a one-time sort through a randomly ordered database. This type of problem structure can have a significant impact on the efficiency of different algorithms.

Finally, we come to the sorting algorithms themselves. In this book we do not attempt to be comprehensive in our treatment. Rather, we present several of the more useful approaches at different levels of complexity. (See [Knuth, Donald E. 73a] or [Sedgewick, Robert 90] for more comprehensive treatments of sorting algorithms.) It is usual to distinguish between *elementary* and *advanced* sorting methods on the basis of their relative complexity as well as their efficiency for large, randomly ordered data sets.

- Elementary methods. These methods:
  - Usually require on the order of n^2 steps to sort n randomly arranged records.
  - Are suitable for small files (i.e., less than about 50 records).
  - Are suitable for larger, partially sorted files.
- Advanced methods. These usually require on the order of $n \log n$ steps to sort n randomly arranged records.

It is customary to use "Big-O" notation to denote how performance of an algorithm varies with the size of a problem. Thus, elementary sorting methods in which the computation time increases with the square of the number of records to be sorted are said to be $O(n^2)$, while advanced are $O(nlogn)$. The difference between elementary and advanced sorts is not trivial. Particularly for the very large data sets that characterize many commercial applications, algorithms which require $O(n^2)$ can be *much* slower than those which take $O(n \log n)$. The following table compares the values of these two functions for different values of n. For values of n over 100,000, the ratio of the times is nearly a factor of 10^4.

COMPARISON OF COMPUTATION TIMES

n	n^2	$n \log n$
10	100	23.0259
100	10,000	460.517
1,000	10^6	6,907.76
10,000	10^8	92,103.4
100,000	10^{10}	1.15129×10^6

In the remainder of this section we present insertion sort, shellsort, and selection sort as examples of elementary methods, and quicksort as an advanced method.

6.3.1 Insertion Sort

Insertion sort is a very simple but flexible algorithm which, as with the remaining elementary methods, can be implemented with data organized either in an array or in a linked list. For ease of presentation, we focus on the array implementation, although several linked list examples are also presented.

Insertion sort moves through the array record by record, at each iteration inserting the new record in the appropriate place in the already sorted subarray. This is analogous to the way many people sort a hand of playing cards.

Figure 6.1 illustrates how an array of eight integer values changes as it is insertion sorted. The initial array is unaffected in the first iteration because the first two values (65 and 70) are in the correct order. At iteration 2, the value 50 is inserted as element

Initial Array: 65 70 50 85 55 80 45 90

Iteration	State of Array							
1	65	70	50	85	55	80	45	90
2	50	65	70	85	55	80	45	90
3	50	65	70	85	55	80	45	90
4	50	55	65	70	85	80	45	90
5	50	55	65	70	80	85	45	90
6	45	50	55	65	70	80	85	90
7	45	50	55	65	70	80	85	90

Figure 6.1 Insertion sort.

zero because it is the minimum of the first three entries in the array. At each subsequent iteration, the next element in the array is inserted into the sorted subarray.

Two implementations of this algorithm are shown below. The first implementation operates directly on the array of records, moving each element into its correct position. This is presented first to show clearly the structure of the method. The example sorts an array of **double**s in ascending numerical order.

```
/* function for insertion sort directly moving records.
   Sort is in ascending numerical order */

void insertion_sort_direct(
  double data[],      /* array to be sorted */
  int n)              /* number of elements in array */
{
 int position,index;
 double value;

/* outer loop over all records (except first) */

 for (position = 1; position < n; position++){
    /* value of record to be inserted */

      value = data[position];

/* inner loop over previously sorted records
   to find  place for insertion of new record */

     for(index=position; index>0 && data[index-1]>value;
           index--)
       data[index] = data[index-1];
     data[index] = value;
   }
}
```

A useful approach to attempt to improve the efficiency of this function is to focus on the inner loop, which will, of course, be executed most frequently. In this case part of the test for whether the loop should terminate could be eliminated by reversing the direction of the outer loop [i.e., starting from the last array element rather than the first and setting a *sentinel marker* as the $(n + 1)$th element of the array]. The sentinel value would be set high enough to be greater than the highest possible value of any record.[4] Thus the sentinel would automatically terminate the inner loop when the **index** reached $n + 1$ without having an explicit test for the value of **index**. In some cases this would be straightforward since a value could readily be found beyond the range of the other records, but in other cases this would not be so easy. Here we see the classic tradeoff

[4]The value **DBL_MAX**, the largest representable **double**, defined in the standard header file **float.h** is an obvious choice for the sentinel value in this example.

between writing general functions which will work for a wide range of applications versus writing special functions which will work more efficiently for limited cases. In general we prefer to present code which can have multiple uses even if some price is paid in efficiency.

Our second implementation of insertion sort uses an array of pointers to access the records in the array, and the sort occurs on the pointers rather than on the records directly. This will be preferred in many cases for several reasons. First, we will frequently want to sort the same data set using different keys and use the sets of sorted lists simultaneously in later phases of the larger program. This is obviously impossible following the strategy of directly reorganizing the list unless we create duplicate lists. Duplication, however will consume large amounts of memory in the case where the records are large data structures. Duplication of an array of pointers which can then be sorted on different keys will usually be much more efficient. Second, again in the case of large structures, using pointers enables us to avoid the extra computation time associated with shifting these records around during the insertion sorting process. The final reason applies when the data is stored as a linked list. In this case the sorted array of pointers can be used to access the correct record in a search without scanning the linked list.

The example below assumes that the set of records to be sorted are character strings.

```
/* insertion sort of character strings
      using array of pointers to access records */

#include <stdio.h>
#include <string.h>

void insertion_sort_indirect(
  char *array[], /* array of pointers to character strings
  int n)         /* number of elements in array */
{
 int position, index;
 char *value;

/* loop over all records (except the first) */
 for (position = 1; position < n; position++){
   value = array[position];

/* inner loop of sort */
   for(index = position; index > 0 &&
         strcmp(array[index-1],value) > 0; index--)
     array[index] = array[index-1];
   array[index] = value;
  }
}
```

With this code we have overcome the problem of moving the records around in memory, but there are still several other obstacles to overcome before we have a general-purpose implementation of insertion sort. First, we have used arrays of records to be sorted, with each record consisting of only a single variable; in general, we need to be able to sort data structure records which are stored as linked lists as well as arrays. Second, we have "hard-wired" into the code the variable on which to sort. Of course, when the record is a single variable there is no choice in the sorting key. Each of these limitations is dealt with in the following generalized implementation of insertion sort.

6.3.2 General Insertion Sort

In this implementation we start with a linked list of known length **num** with each node defined to have the following structure:

```
struct list_node {
        int x;
        int y;
        struct list_node *next;
        };

typedef struct list_node * List;
```

When sorting a linked list it is usual to set up an array of pointers with each pointer assigned the address of one record in the list. The sorting occurs in the array of pointers, which can then be used in any subsequent search to locate the appropriate record without the linked list itself ever having been altered. The following fragment of code uses **calloc()** to obtain the memory necessary to set up the pointer array and then initializes this array:

```
#include <stdlib.h>

List *record; /* used to store the address of the array
                 of pointers once memory is allocated */
List present, head;
int num, i /* number of records and counter */
.
.

/* allocate memory for elements of List, where num
   is the number of nodes in the linked list */
record =  calloc(num,sizeof(List));
.
.

/* initialize array of pointers to access each record */
present = head;
```

```
for(i = 0; present != NULL; i++){
  record[i] = present;
   present = present->next;
    }
```

The function **insertion_sort_general()** shown below can be used to sort the records on either key **x** or **y**. The function uses a pointer to a function called **key()**, which compares two records and returns an integer which is TRUE if the first record is "greater" (i.e., higher in the sorting order) than the second record and FALSE otherwise. The code for this generalized insertion sort is as follows:

```
/* generalized insertion sort */
void insertion_sort_general(
  List record[],              /* array of pointers */
  int num,                    /* number of records */
  int (*key)(List, List) ) /* pointer to function
                               which takes Lists
                               as arguments
                               and returns an int */
{
  List old_node;
  int position, index;

  /* loop through all entries */
  for (position = 1; position < num; position++) {
       old_node = record[position];

  /* find correct positions */
       for (index = position; index > 0 &&
              key(record[index-1], old_node);index--)
           record[index] = record[index-1];
       record[index] = old_node;
    }
}
```

The arguments to **insertion_sort_general()** include the array of pointers to the nodes on the linked list and a pointer to a function **key()**. The second component of the logical test for termination of the inner loop of the sort uses the pointer to **key()** to generalize the record key comparison.

To show how this could be used, the following fragment of **main()** code together with the two functions would result in the linked list being sorted twice, first by key **x** in ascending order and second by key **y** in descending order:

```
{
/* fragment of main */
int by_x (List, List);
int by_y (List, List);
```

```
void insertion_sort_general (List[], int,
      int (*) (List, List));
.
.
/* sort by x in ascending order */
insertion_sort_general(record, num, by_x);
.
.
/* sort by y in descending order */
insertion_sort_general(record, num, by_y);
.
.
}

/* by_x compares the x values of two structs,
   returning TRUE if first value exceeds second */

int by_x( List node1, List node2)
{
  return ( node1->x > node2->x);
}
/* by_y compares the y values of two structs,
   returning TRUE if first value is less than second */

int by_y(List node1, List node2)
{
  return ( node1->y < node2->y);
}
```

This approach can be followed to generalize any of the remaining sorting algorithms, which, for the sake of clarity, will be presented for arrays of single key records.

6.3.3 Shellsort

Insertion sort is not very efficient, except for partially ordered data sets, because it exchanges only adjacent records. So, for example, if the initial data set of n items is reverse ordered, it will take n cycles through the inner loop to place the last record in its correct place. Shellsort is a simple extension which can considerably improve the efficiency of insertion sort by exchanging records which are farther apart. The approach is to sort all records which are h records apart. If, for example, $h = 13$, records 0,13,26... will be sorted after the first pass through the array. After completion of all iterations for $h = 13$, all sequences of records separated by h will be sorted (including sequences 1,14,27..., and 2,15,28..., etc.). In subsequent iterations, h is systematically reduced until the $h = 1$ iteration is completed. At this point the data set will be fully sorted.

The following implementation of shellsort sorts an array of n doubles:

```
/* this function sorts an array of n doubles with
   the gap size being halved at each iteration  */

void shell_sort(
  double data[],   /* array to be sorted */
  int n)           /* number of entries in array */
{
  double temp;
  int h, i, j;  /* h is gap size at each iteration */

  /* outer loop varies gap size */
    for (h = n/2; h > 0; h /= 2) {

  /* middle loop covers all sequences for a gap size h */
      for (i = h; i < n; i++) {

  /* inner loop uses a simple exchange sort */
          for(j=i-h; j>=0 && data[j]>data[j+h]; j -= h) {
            temp = data[j];
            data[j] = data[j+h];
            data[j+h] = temp;
            }
        }
    }
  }
```

Figure 6.2 shows how shellsort works on an array of eight integers. The state of the array after each execution of the outer loop is illustrated.

While the principle of shellsort is well illustrated with the code and example above, the actual sequence of values of h used in the outer loop is far from ideal. Sedgewick [Sedgewick, Robert 90] notes that a much better sequence for h is the reverse of the sequence 1,4,13,40,121,... . In this sequence the $(i+1)$th number is given by $h_{i+1} = 3h_i + 1$. For a given problem size n, the sequence of values of h would start with the second largest number in the given sequence that is smaller than the array size.

As an aside, when $h = 1$, shellsort is often called a *bubble sort*.

Initial Array:	65	70	50	85	55	80	45	90
After h = 4	55	70	45	85	65	80	50	90
After h = 2	45	70	50	80	55	85	65	90
After h = 1	45	50	55	65	70	80	85	90

Figure 6.2 Shellsort.

6.3.4 Selection Sort

The final elementary sorting method presented is selection sort, which follows a straightforward strategy of finding the minimum value among the record keys and making that the first record, then finding the minimum of the remaining elements and making that the second record, etc. Selection sort is implemented for an array of integers in the following function.

```
/* this function implements selection
   sort for array of ints */

void selection_sort(
  int data[],      /* array to be sorted */
  int n)           /* number of elements in array */
{

  int min, i ,j;
  int temp;

/* outer loop over all array elements */
  for (i = 0; i < n; i++) {
    min = i;

/* loop over remaining elements to find minimum value */
    for (j = i + 1; j < n; j++)
      if (data[j] < data[min])
        min = j;

/* insert minimum at correct place */
    temp = data[min];
    data[min] = data[i];
    data[i] = temp;
  }
}
```

Figure 6.3 illustrates the major steps in a selection sort on the same array of eight integers used in earlier examples. After the first iteration, the minimum value in the array (45) is moved to the element zero. In the second iteration the next smallest value is moved to element one of the array. This process continues until the last array element is reached. In this particular example, the array is completely sorted after the fourth iteration. However, the remaining iterations would still be executed.

6.3.5 Quicksort

The only advanced sorting algorithm we will present is *quicksort*, which is perhaps the most widely applied sorting method for large problems. Quicksort was developed by C.

Initial Array: 65 70 50 85 55 80 45 90

Iteration	State of Array							
1	45	70	50	85	55	80	65	90
2	45	50	70	85	55	80	65	90
3	45	50	55	85	70	80	65	90
4	45	50	55	65	70	80	85	90
5	45	50	55	65	70	80	85	90
6	45	50	55	65	70	80	85	90
7	45	50	55	65	70	80	85	90

Figure 6.3 Selection sort.

A. R. Hoare[Hoare, C.A.R. 62] in the early 1960s and is the classic embodiment of a "divide and conquer" algorithmic strategy.

The central idea in quicksort is to partition the data set into two parts and sort them independently. This partitioning process is repeated until all the subproblems are of size one, when they are, by definition, sorted. The key to this method is to find an efficient way to partition a data set into two parts such that:

- Everything in one subarray is smaller than a known value called the *pivot element*.
- Everything in the other subarray is greater than the pivot value.

Assuming that the records are stored in an array, the quicksort partitioning method works as follows:

- Choose an element in the array to be the pivot element. For convenience we will choose the first element in the array.
- Start one integer array index (**`lowindex`**) at the second element of the array and another index (**`highindex`**) at the last element.
- Decrease **`highindex`** until an element is found with value *less than* the pivot value or **`highindex`** equals **`lowindex`**. Increase **`lowindex`** until an element *greater than* the pivot is found or until the value **`highindex`** is reached. We refer to this process as *scanning*.
- If the two array indices **`highindex`** and **`lowindex`** are different, the two array elements are out of place and are exchanged. Repeat the previous step, resuming the two scans where they had been interrupted.

- If the two indices are the same and the value in the array they both reference is *less than* the pivot value, exchange the pivot element and the array element. The array is now partitioned by the element referenced by **highindex** (or **lowindex**). The current location of the pivot element separates the array into a portion less than the pivot and a portion greater than the pivot. Moreover, the pivot element is in its correct place in the array once it is fully sorted.

The following example illustrates this partitioning process. Suppose that the original array consists of the following eight elements:

65 70 50 85 55 80 45 90

The pivot element will be 65 (element zero in the array), and the scans will first be halted at elements one and six of the array (45 and 70), which will then be exchanged. After this first exchange the array will be as follows:

65 45 50 85 55 80 70 90

The scan will next be halted at elements three and four of the array (85 and 55), which will then be exchanged to produce

65 45 50 55 85 80 70 90

The scan will next be stopped with both indices referencing element three (55), which is then exchanged with the pivot element (since it is smaller than it) to produce

55 45 50 65 85 80 70 90

This array is now partitioned about the original pivot value (65), which is in its correct position in the final, sorted array. The partitioning process can now be applied to the subarray

55 45 50

as well as to the subarray

85 80 70 90

The recursive nature of quicksort is clear from this simple example. We can apply the partitioning recursively to each subarray until each subarray has only a single element. At this point, the complete array will be sorted. This structure will be used in the implementation of quicksort below.

The code for quicksort uses three distinct functions. Function **quicksort()** executes the basic recursion, invoking **partition()** to partition the array and then invoking **quicksort()** again for each of the subarrays created. The function **exchange()** is used to interchange two elements of the array. The arguments to **quicksort()** are the array to be sorted and the number of elements in the array. The code for sorting an array of integers is as follows:

```
/* quicksort for an array of num ints */
```

```
/* function prototypes */
void quicksort (int [], int);
void partition (int *,int,int *([]),
                int *,int *([]),int*);
void exchange (int [], int, int);

void quicksort(
     int data[], /* data is array of ints to sort */
     int num)    /* number of elements in data */
{
  int *low, *high;
  int numlow, numhigh;

  if (num > 1) /* a subarray of size one is solved */
  {
    partition(data, num, &low, &numlow,
              &high, &numhigh);
    quicksort(low, numlow);
    quicksort(high, numhigh);
  }
}
```

The second function, **partition()**, has six arguments. The first two, **array** and **num**, are the array to be partitioned and the number of elements in that array. The remaining four arguments are all pointers. They are used to return the results of the partitioning to the function which invokes **partition()**. The pointer **low** points to the beginning of the first subarray (the one containing all values lower than the pivot element after the partitioning is complete). The argument **numlow** is a pointer to the number of elements in the array pointed to by **low**. The argument **high** points to the second subarray created by partitioning (the one with all values greater than the pivot) and the argument **numhigh** points to the number of elements in this subarray.

partition() scans the elements in the array, invoking the function **exchange()** to exchange elements which are out of place. The C code is as follows:

```
/* partition() applies the partitioning method
     to an array with num elements */

void partition(
  int *array,      /* array to be partitioned */
  int num,         /* number of elements in array */
  int *(low[]),    /* pointer to the left subarray */
  int *numlow,     /* pointer to the number of
                      elements in the left subarray */
  int *(high[]),   /* pointer to the right subarray */
  int *numhigh)    /* pointer to the number of
                      elements in the right subarray */
```

```
/*  values resulting from the partition
    returned using pointers */
{
  int lowindex, highindex, pivot;
/* initialize pivot value and array subscripts */
  pivot = array[0];
  lowindex = 1;
  highindex = num - 1;

/* highindex scans down and lowindex
   up the array until stopped */
  while (lowindex < highindex) {
    while (highindex > lowindex &&
           array[highindex] > pivot)
      highindex--;
    while (lowindex < highindex &&
           array[lowindex] <= pivot)
      lowindex++;
 /* if needed exchange the two out of place elements */
    if (highindex > lowindex)
      exchange (array, lowindex, highindex);
  }

/* see if pivot and value pointed
       to by should be exchanged */

  if (pivot > array[lowindex]) {
    exchange(array, 0, lowindex);
    highindex++;
   }

/* set return values */
  *low = &(array[0]);
  *high = &(array[highindex]);
  *numlow = lowindex;
  *numhigh = num - highindex;
 }
```

The third function used to implement a quicksort is **exchange()**, which exchanges two elements of an array. Its arguments are the array where the exchange is to take place and the subscripts of the two elements to be exchanged. The code is as follows:

```
/* exchange()swaps two array elements */
void exchange(
     int array[], /* array to be operated on */
     int index1,  /* index of one exchange element */
     int index2)  /* index of other exchange element */
```

```
{
  int temp;
  temp = array[index1];
  array[index1] = array[index2];
  array[index2] = temp;

}
```

While this implementation presents the basic quicksort algorithm, there are many ways to improve its performance (see Sedgewick [Sedgewick, Robert 90] for a full discussion of these options). Several of the more important ones are discussed briefly below.

- Selection of pivot element. Ideally, the array to be sorted would be partitioned into approximately equal-size subarrays at each partition stage, since this will minimize computation time. One reasonably straightforward way to improve on the "take the first element as the pivot" rule is to select the *median* of three elements to be the partition element. For example, one could take the median of the first, last, and middle elements in the array, exchange this median with the first element (if necessary), and proceed as before. This approach would tend to make the subarrays closer in size.
- Order of subarray processing. In the recursive implementation of quicksort there is no preference given to short subarrays in the processing sequence. However, such a priority could minimize the number of **`quicksort()`** invocations active concurrently. Since an executing C program must allocate memory for the arguments and local variables for every active invocation of a function, this approach will reduce the memory required by the quicksort algorithm.
- Sorting of small subarrays. Although quicksort is efficient for larger data sets, each step in quicksort takes considerable computational effort, making it less efficient than simpler methods for small data sets. This deficiency in quicksort can be corrected by using a simple sorting method on subproblems smaller than a given size.

Implementation of these improvements is explored in Exercise 6-9.

6.4 SIMPLE SEARCHING METHODS

As stated earlier, one of the most common reasons for sorting is to improve the efficiency of subsequent searches of the data. The objective of any search is to locate a specific record identified through one or more keys. Making searches more efficient is of practical importance whenever large databases are involved, particularly when access to the record must be very fast. Examples of this include telephone reservation systems, such as for a chain of hotels or for an airline. These searches are often characterized by the need to use more than one key as alternative means to identify a specific record.

For example, in the hotel reservation case, a reservation record would include the hotel name and location, the customer name and associated data, the date of the booking, and the room rate. The record will also be uniquely identified by a reservation number. It should be possible to access a specific record through any one of several keys, so it would be unwise to design the search process to work efficiently for one key but inefficiently for others.

In other software systems this may be a less severe problem. For example, in a bank there may be a strong reason to make accessing information on an account difficult to protect the privacy of the customer. In this case the data structure may be set up specifically to allow access only through the account number. If such a restriction is included, it may result in a much more efficient data storage design and corresponding search process.

The characteristics of any specific problem will clearly have a major role in determining the organization and content of the database as well as the appropriate search algorithm to be employed. Among the important problem characteristics are:

- Frequency of accessing any record. If records are likely to be accessed only infrequently, then the efficiency of the search process becomes less important.
- Stability of the database. In the airline reservation case, for example, reservations are likely to be changed frequently, whereas in the banking application, information about owners of accounts will be quite stable.
- Nature of the access process. This includes whether or not there is a maximum acceptable time for access to information and the type of processing required after a record being searched for is found.

We will return to the appropriateness of each data organization option and each search strategy at the end of this chapter.

Sorting and searching are strongly interrelated. The type of sort implemented must be consistent with the type of search to be used. In the preceding section the focus was on producing a sorted array of data structures or a sorted array of pointers to nodes on a linked list. As will be shown in this section there are other approaches to searching which require a radically different type of data organization than sorting. In all cases, however, we will assume that each record consists of a data structure which includes the *search key* along with other information. A basic principle in searching is first to organize the database in such a way that the search can quickly be focused on the part of the database which is likely to contain the record of interest. As will be shown, this general objective can be pursued through very different strategies.

Many tasks which do not initially appear to be searches do, in fact, require search as an element. For example:

- Inserting a new record in an already structured database. If the structure of the database is to be preserved for future processing, then the correct location for the insertion must be determined. This requires a search of the existing records.
- Deleting an existing record. First the correct record must be found as a result of a search.

These different types of search-related activities impose different demands on the search process, depending upon the organization of the data. For example, record deletion in an array is more difficult than in a linked list, as is inserting a new record. Hence the more dynamic the database, the greater the attraction of a linked list implementation. Search techniques can be classified into elementary and advanced, with the main distinction being the access time for very large databases. Elementary methods include binary search and binary search trees, both of which are presented in this section. For the sake of completeness we start with a short discussion of linear search.

6.4.1 Linear Search

Linear search is the sequential process of scanning through the records, starting at the first record, until either a match is found or the search is completed unsuccessfully. Linear search is the only option if the records are in random order or if the data is stored in a linear linked list for which no pointer array has been established. This approach may be appropriate if the size of the data set is guaranteed to be small and the content of the database is highly dynamic. In this case the effort of continually sorting the list may *not* be justified. However this is an extremely artificial scenario, particularly when one recognizes that a highly dynamic small database implies frequent deletion of records, which is itself a task requiring a search.

The following function implements a linear search on an array of data structures which include the following members:

```
/* structure template for typical data record */
struct node {
  int phone_num;
  char name[80];
  .....            /* other variables in the record  */
  .....            /*   would go here                */
};
```

The search will use either the telephone number (**phone_num**) or the name (**name**) as the key, depending on the value of a flag which is passed as one of the search function's arguments. Because the type of item to be searched for may be either an integer or a character string, we use a **void** pointer as an argument to the searching function. This **void** pointer contains the address of the item to be searched for. The value pointed to by this pointer is cast as an **int** when the search is for a phone number, and as a character string when the search is for a name.

If the item being searched for is found, then the linear search function returns the subscript in the array of records of the found item. If the item is not found, then the linear search returns the special code **NOT_FOUND**, which is set to negative one. The

implementation of the linear search is as follows:

```
/* linear search method */

#include <string.h>

#define SEARCH_BY_PHONE 0
#define SEARCH_BY_NAME  1
#define NOT_FOUND -1    /* indicates value not found */

int lin_search(
    struct node data[], /* array of structs to search */
    int n,              /* number of elements in array */
    int flag,           /* type of  search key */
    void *target)       /* pointer to search target */
{
  int i = 0;

/* loop through all records */
  for(i=0; i<n; i++)
    if(flag == SEARCH_BY_PHONE &&
       (data[i].phone_num == *(int *)target))
      return(i);
    else if(flag == SEARCH_BY_NAME &&
       !strcmp(data[i].name, target))
      return(i);

  return(NOT_FOUND);
}
```

Note that the header file **string.h** is included. This provides the function prototype for **strcmp()**, the string comparison function in the standard library. The prototype for this library function automatically casts the argument **target** from a pointer to **void** to a pointer to **char** when **stremp()** is invoked.

A second aspect of the linear search function worth noting is the expression ***(int *)target** used in the **if** statement. This construction casts the **void** pointer **target** as a pointer to an integer, and then uses the indirection operator (*****) to find the integer value pointer to by **target**. It would be incorrect to code this expression as **(int)*target** because the indirection operator cannot be applied to a **void** pointer.

The approach of using flags and void pointers can be used to write more general functions for any type of sorting or searching method, but for the sake of clarity in presentation, for the remainder of this section we assume that search is based on a single key, either an **int** or a string. Some of the exercises at the end of the chapter explore generalizations of the functions in the text to allow searching for items of different types.

Before leaving the rather pedestrian linear search method for more efficient approaches, we should note that for an array of n items linear search requires on average $(n+1)/2$ comparisons for a successful outcome and $n+1$ comparisons for an unsuccessful search. Given the availability of more efficient algorithms, linear search is unattractive for all but the smallest problems.

6.4.2 Binary Search

Binary search is a straightforward procedure for locating an element in an already sorted array. It is directly analogous to the bisection method for root finding presented in Section 3.5.2, proceeding by systematically halving the search domain at each iteration until either the correct record is found or the search concludes unsuccessfully.

The function **bin_search()** shown below implements a binary search using the integer key **phone_num** in the array of data structures of type **struct node** used in the preceding subsection. The three arguments to the function are the array of records to be searched, the number of elements in the array, and the phone number being searched for. **bin_search()** returns an integer equal to the index in the array where the record with the requested phone number is found or a special code (set to negative one in the example below) when the record cannot be found. As with any application of the binary search method, we assume that array of records is sorted on the search key.

```
#define NOT_FOUND -1     /* indicates value not found */

int bin_search(
   struct node data[], /* array to be searched */
   int n,              /* number of elements in array */
   int target)         /* value being search for */
{
   int low,high,middle;

   low = 0;
   high = n-1;

/* terminate search when either we have narrowed to
   a single record or when we have found target */

   while (low <= high) {
     middle = (low+high)/2;
     if (target < data[middle].phone_num)
       high = middle-1;
     else if ( target > data[middle].phone_num)
       low = middle+1;
```

```
    else
      return(middle);  /* search was successful */
  }
  return (NOT_FOUND);  /* search unsuccessful */
}
```

Binary search is much more efficient than linear search since it requires $O(\log n)$ iterations in the worst case when there is no match. It is an attractive approach to searching when the database is reasonably stable but large and when many searches are required. On the other hand, if the database is subject to frequent modification while the search itself is still efficient, the need to resort the array as records are added and deleted detracts seriously from its appeal. In such situations a method which does not require maintaining a sorted array would be much more attractive. In the remainder of this chapter we present several such approaches.

6.5 BINARY SEARCH TREES

In Section 5.12 we introduced the *binary search tree* as a special type of tree with the following properties:

- Each node has at most two nodes at the next-lower level (this is a requirement for any binary tree).
- Every node in the left subtree has a key value *no larger than* the root node; every node in the right subtree has a key value *no smaller than* the root node. This property applies recursively to each subtree of the complete tree.

Binary search trees allow a search process which retains the efficiency of the binary search method but overcomes some of the disadvantages of requiring data representation in array form. However, the basic deficiency of binary search (that the array remain sorted) is replaced with another problem with binary search trees: how to maintain trees that satisfy the properties listed above as records are added to, and removed from, the data set. A second problem that is introduced is due to the multiplicity of binary search trees which can represent the same data set. Figure 6.4 illustrates this characteristic of binary search trees. All three trees satisfy the properties of a binary search tree, and all contain the same key integer data. Tree *a* is a balanced tree since all the leaf nodes exist at the lowest two levels, whereas tree *c* is a degenerate form of tree which is also a linked list since each node has at most a single descendant. Tree *b* is intermediate in terms of the number of levels at which nodes exist.

Figure 6.4 also illustrates several other important properties of binary search trees. First, the smallest key of a binary search tree is found in the node farthest to the left in the tree, and the largest key is found in the node farthest to the right. Note that these minimum and maximum key nodes may be at any level in the tree, depending on the tree structure. So in tree *a* the maximum node is at the lowest level, whereas in *c*, it is at the highest level. Second, the efficiency of a search in a binary search tree

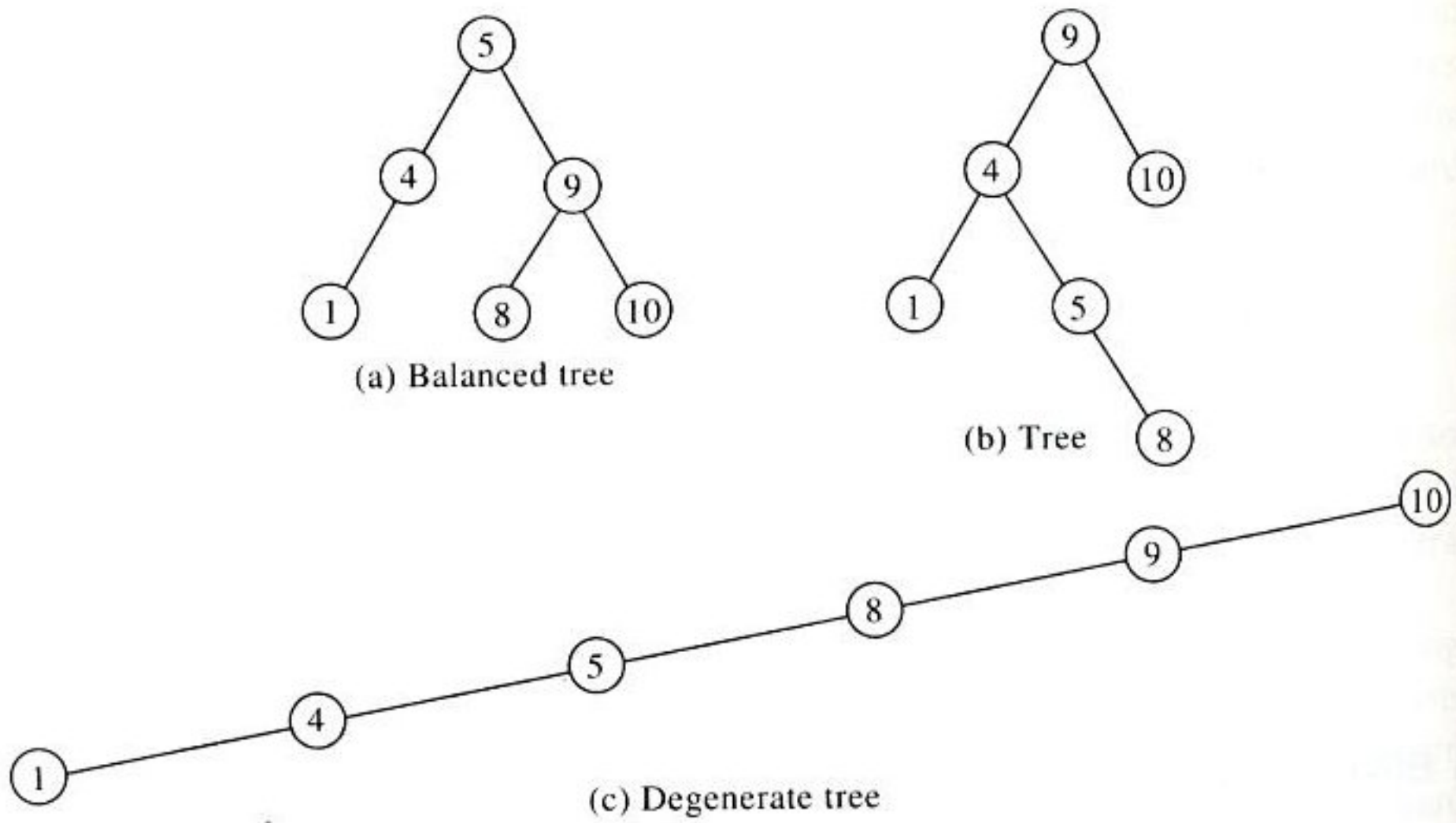

Figure 6.4 Alternative binary search tree representations.

will depend on the tree structure. For a given number of nodes, n, the most efficient tree structure will be a balanced tree which will have $\log_2(n+1)$ levels. In this case the computation time for a search will be $O(\log n)$, compared with $O(n)$ in the fully degenerate case. Thus an important issue to address is how to obtain a tree which has a balanced structure initially. A second issue relevant if the data in the tree changes is how to maintain balance in the face of subsequent additions and deletions. While it is not essential to obtain a fully balanced tree, it is important to achieve a structure more like that of tree *a* than tree *c* in Figure 6.4.

A key characteristic of binary search trees which can greatly facilitate writing functions that operate on them is their highly recursive nature. This is because the subtree of any node in a binary search tree is itself a binary search tree. This can readily be seen by reference to any of the example trees in Figure 6.4. To illustrate the value of this property, we will show how a sorted list can easily be obtained from any binary search tree.

Any systematic process of visiting all nodes in a tree is referred to as a *traversal* of the tree. To obtain a sorted list of all nodes in a binary search tree we apply an *in-order* traversal, which means that for any subtree the nodes are visited in the order

- Left node
- Root node
- Right node

In this, and subsequent, examples of binary search trees we will use the following data structure, in which we define only the node pointers and the search key, in this case an individual's social security number:

```
/* structure template for binary tree nodes */
struct node {
  int soc_sec;
  struct node *left;
  struct node *right;
};

typedef struct node * Bin_tree;
```

The following function would execute an in-order traversal of a binary search tree. It prints the social security numbers in ascending order. The function **empty_tree()** simply tests for an empty tree as did the function **empty_list()** for lists in Chapter 5.

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0

/* prints sorted items in a binary tree */

void tree_print(Bin_tree t)
{
  /* first check for an empty tree */
  if (! empty_tree (t)){
     tree_print(t->left);
     printf("%d\n",t->soc_sec);
     tree_print(t->right);
  }
}

/* determines if tree is empty */

int empty_tree (Bin_tree b)
{
  if (b == NULL)
     return (TRUE);
  return (FALSE);
}
```

In order to use binary search trees we need to develop algorithms that accomplish the following tasks:

- Search for a record with a specific key value
- Add a node to an existing tree
- Delete a node from an existing tree

These functions are presented in the following subsections.

6.5.1 Searching in a Binary Search Tree

The result of a successful search in a binary search tree will be a pointer to the record sought. This is different from the array-based searches, in which an array index is used to identify the record in question. In fact, most binary search tree processing functions will communicate with the calling function through pointers to nodes in the tree. The search process takes advantage of the structure of the binary search tree in the following steps:

1. Check to see if the root node is empty. If so, return.
2. Compare the search key with the root key.
3. If it matches, end the search.
4. If it is smaller, search the left subtree.
5. Otherwise, search the right subtree.

Eventually, the search will either conclude with a match or reach a leaf node without a match, in which case the search has been unsuccessful. Our implementation of `find_node()` has as arguments the tree to be searched and the integer value to be searched for. It returns a pointer to the node in the tree where the item is located or **NULL** if the item is not found. The function takes advantage of the recursive structure of the tree, calling itself recursively as it works its way from the root to the item being searched for. The code is as follows:

```
/* finds a node in a binary search tree */

Bin_tree find_node(
  Bin_tree t,            /* tree to be searched */
  int value)             /* value to search for */
{
  if (empty_tree(t)) /* tree is empty */
     return(t);
/* check the node */

  else {
    if(t->soc_sec == value) /* match has been found */
      return(t);
    else if(t->soc_sec > value) /* search left subtree */
      return(find_node( t->left, value));
    else                              /* search right subtree */
      return(find_node( t->right, value));
  }
}
```

As with any function that returns a pointer which may be **NULL**, it is essential that the calling function check to see if a **NULL** pointer has been returned before further processing is done. Again, while this function will eventually find a match if it exists,

its efficiency will depend on the tree structure, so we turn next to the topics of building and modifying binary search trees.

6.5.2 Adding a Node to a Binary Search Tree

The most straightforward operation on an existing binary search tree is to add a new node. In this case we cannot drastically affect the degree of balance of the tree, since there will be only a single node added. One simple approach is to add the node as a child of some existing node without shifting anything else in the tree. This strategy could also be used repeatedly to build a tree from scratch, but in this case there needs to be some consideration of the impact of the sequence of additions on the tree structure. We reserve consideration of this problem to Section 6.5.3.

Given an existing tree, there will be just one location at which the new node can be added which preserves the properties of the binary search tree. The task of the function **add_node()** is to find this location and set up the information associated with the new node. To preserve the binary structure of the tree, the new node must be added as a child of a node which has at most one existing child. The value of the search key for the new record is used to find the correct location for the new node. In the following implementation of **add_node()** we assume that the **struct** to hold the new node has already been set up, including initialization of its right and left pointers to **NULL**. A pointer to this node is passed to **add-node()** as an argument.

```
/* adds a new node to a binary search tree */

void add_node(
  Bin_tree t,          /* root of the tree */
  Bin_tree new_node)   /* struct to be added to tree */
{
  Bin_tree prev;  /* used to access new node's parent */

 /* find correct place for new node */
  while(t != NULL) {
    if(new_node->soc_sec < t->soc_sec) {
      prev = t;
      t = t->left;
    }
    else {
      prev = t;
      t = t->right;
    }
  }

 /* next set up new node pointer in parent node */
    if(new_node->soc_sec < prev->soc_sec)
      prev->left = new_node;
```