CHAPTER 2

ARRAYS AND STRUCTURES

2.1 ARRAYS

2.1.1 The Abstract Data Type

We begin our discussion by considering an array as an ADT. This is not the usual perspective since many programmers view an array only as "a consecutive set of memory locations." This is unfortunate because it clearly shows an emphasis on implementation issues. Thus, although an array is usually implemented as a consecutive set of memory locations, this is not always the case. Intuitively an array is a set of pairs, *<index, value>*, such that each index that is defined has a value associated with it. In mathematical terms, we call this a *correspondence* or a *mapping*. However, when considering an ADT we are more concerned with the operations that can be performed on an array. Aside from creating a new array, most languages provide only two standard operations for arrays, one that retrieves a value, and a second that stores a value. ADT 2.1 shows a definition of the array ADT.

The *Create(j, list)* function produces a new, empty array of the appropriate size. All of the items are initially undefined. *Retrieve* accepts an *array* and an *index*. It

ADT *Array* is

objects: A set of pairs <*index*, *value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, {0, $\cdots$, n–1} for one dimension, {(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)} for two dimensions, etc.

functions:

for all $A \in$ *Array*, $i \in$ *index*, $x \in$ *item*, j, *size* $\in$ integer

Array Create(*j*, *list*)	::=	**return** an array of *j* dimensions where *list* is a *j*-tuple whose *i*th element is the the size of the *i*th dimension. *Items* are undefined.
Item Retrieve(*A*, *i*)	::=	**if** (*i* $\in$ *index*) **return** the item associated with index value *i* in array *A* **else return** error
Array Store(*A*,*i*,*x*)	::=	**if** (*i* in *index*) **return** an array that is identical to array *A* except the new pair <*i*, *x*> has been inserted **else return** error.

end *Array*

ADT 2.1: Abstract Data Type *Array*

returns the value associated with the index if the index is valid, or an error if the index is invalid. *Store* accepts an *array*, an *index*, and an *item*, and returns the original array augmented with the new <*index*, *value*> pair. The advantage of this ADT definition is that it clearly points out the fact that the array is a more general structure than "a consecutive set of memory locations."

2.1.2 Arrays in C

We restrict ourselves initially to one-dimensional arrays. A one-dimensional array in C is declared implicitly by appending brackets to the name of a variable. For example,

```
int list[5], *plist[5];
```

declares two arrays each containing five elements. The first array defines five integers, while the second defines five pointers to integers. In C all arrays start at index 0, so *list*[0], *list*[1], *list*[2], *list*[3], and *list*[4] (abbreviated *list* [0:4]) are the names of the five array elements, each of which contains an integer value. Similarly, *plist* [0:4] are the names of five array elements, each of which contains a pointer to an integer.

We now consider the implementation of one-dimensional arrays. When the compiler encounters an array declaration such as the one used above to create *list*, it allocates five consecutive memory locations. Each memory location is large enough to hold a single integer. The address of the first element *list*[0], is called the *base address*. If the size of an integer on your machine is denoted by *sizeof(int)*, then the memory address of *list* [*i*] is $\alpha + i*sizeof$ (*int*), where α is the base address. In fact, when we write *list*[*i*] in a C program, C interprets it as a pointer to an integer whose address is *list* [*i*] is $\alpha +$ *i*sizeof* (*int*). Observe that there is a difference between a declaration such as

```
int *list1;
```

and

```
int list2[5];
```

The variables *list*1 and *list*2 are both pointers to an **int**, but in the second case five memory locations for holding integers have been reserved. *list*2 is a pointer to *list*2[0] and *list*2+*i* is a pointer to *list*2[*i*]. Notice that in C, we do not multiply the offset *i* with the size of the type to get to the appropriate element of the array. Thus, regardless of the type of the array *list*2, it is always the case that (*list* 2 + *i*) equals *&list* 2[*i*]. So, *(*list*2 + *i*) equals *list*2[*i*].

It is useful to consider the way C treats an array when it is a parameter to a function. All parameters of a C function must be declared within the function. However, the range of a one-dimensional array is defined only in the main program since new storage for an array is not allocated within a function. If the size of a one-dimensional array is needed, it must be either passed into the function as an argument or accessed as a glot al variable.

Consider Program 2.1. When *sum* is invoked, *input* = *&input* [0] is copied into a temporary location and associated with the formal parameter *list*. When *list*[*i*] occurs on the right-hand side of the equals sign, a dereference takes place and the value pointed at by (*list* + *i*) is returned. If *list*[*i*] appears on the left-hand side of the equals sign, then the value produced on the right-hand side is stored in the location (*list* + *i*). Thus in C, array parameters have their values altered, despite the fact that the parameter passing is done using *call-by-value*.

Example 2.1 [*One-dimensional array addressing*]: Assume that we have the following declaration:

```
int one[] = {0, 1, 2, 3, 4};
```

We would like to write a function that prints out both the address of the *i*th element of this array and the value found at this address. To do this, *print*1 (Program 2.2) uses pointer arithmetic. The function is invoked as *print*1(*&one* [0],5). As you can see from the **printf** statement, the address of the *i*th element is simply *ptr* + *i*. To obtain the

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
void main(void)
{
   int i;
   for (i = 0; i < MAX_SIZE; i++)
      input[i] = i;
   answer = sum(input, MAX_SIZE);
   printf("The sum is: %f\n", answer);
}
float sum(float list[], int n)
{
   int i;
   float tempsum = 0;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

Program 2.1: Example array program

value of the *i*th element, we use the dereferencing operator, *. Thus, *(*ptr* + *i*) indicates that we want the contents of the *ptr* + *i* position rather than the address.

```
void print1(int *ptr, int rows)
{/* print out a one-dimensional array using a pointer */
   int i;
   printf("Address Contents\n");
   for (i = 0; i < rows; i++)
      printf("%8u%5d\n", ptr + i, *(ptr + i));
   printf("\n");
}
```

Program 2.2: One-dimensional array accessed by address

Figure 2.1 shows the results we obtained when we ran *print1*. Notice that the addresses increase by four because each **int** is 4 bytes on our machine. □

Address	Contents
12244868	0
12344872	1
12344876	2
12344880	3
12344884	4

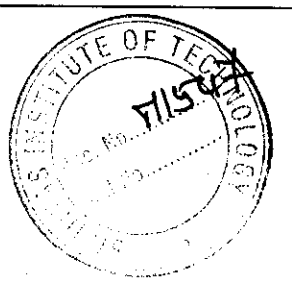


Figure 2.1: One-dimensional array addressing

2.2 DYNAMICALLY ALLOCATED ARRAYS

2.2.1 ONE-DIMENSIONAL ARRAYS

In Program 1.4, we defined the constant *MAX_SIZE* to have the value 101. As a result, the program can be used to sort a collection of up to 101 numbers. If the user wishes to sort more than 101 numbers, we have to change the definition of *MAX_SIZE* using some larger value and recompile the program. How large should this new value be? If we set *MAX_SIZE* to a very large number (say several million), we reduce the likelihood the program will fail at run time because the input value of n is less likely to exceed this large value of *MAX_SIZE*. However, we increase the likelihood the program may fail to compile for lack of memory for the array *list*. When writing computer programs, we often find ourselves in a situation where we cannot reliably determine how large an array to use. A good solution to this problem is to defer this decision to run time and allocate the array when we have a good estimate of the required array size. So, for example, we could change the first few lines of function *main* of Program 1.4 to:

```
int i,n,*list;
printf("Enter the number of numbers to generate: ");
scanf("%d",&n);
if( n < 1 ) {
  fprintf(stderr, "Improper value of n\n");
  exit(EXIT_FAILURE);
}
MALLOC(list, n * sizeof(int));
```

Now, the program fails only when $n<1$ or we do not have sufficient memory to hold the list of numbers that are to be sorted.

2.2.2 TWO-DIMENSIONAL ARRAYS

C uses the so-called array-of-arrays representation to represent a multidimensional array. In this representation, a two-dimensional array is represented as a one-dimensional array in which each element is, itself, a one-dimensional array. To represent the two-dimensional array

```
int x[3][5];
```

we actually create a one-dimensional array x whose length is 3; each element of x is a one-dimensional array whose length is 5. Figure 2.2 shows the memory structure. Four separate memory blocks are used. One block (the lightly shaded block) is large enough for three pointers and each of the remaining blocks is large enough for 5 **int**s.

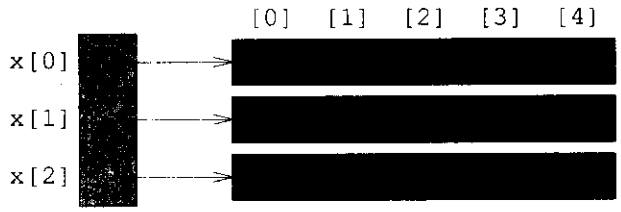


Figure 2.2: Array-of-arrays representation

C finds the element $x[i][j]$ by first accessing the pointer in $x[i]$. This pointer gives us the address, in memory, of the zeroth element of row i of the array. Then by adding j*sizeof* (*int*) to this pointer, the address of the $[j]$th element of row i (i.e., element $x[i][j]$) is determined. Program 2.3 gives a function that creates a two-dimensional array at run time.

This function may be used in the following way, for example. The second line allocates memory for a 5 by 10 two-dimensional array of integers and the third line assigns the value 6 to the [2][4] element of this array.

```
int **myArray;
myArray = make2dArray(5,10);
myArray[2][4] = 6;
```

C provides two additional memory allocation functions—*calloc* and *realloc*—that are useful in the context of dynamically allocated arrays. The function *calloc* allocates a user-specified amount of memory and initializes the allocated memory to 0 (i.e., all

```
int** make2dArray(int rows, int cols)
{/* create a two dimensional rows x cols array */
   int **x, i;

   /* get memory for row pointers */
   MALLOC(x, rows * sizeof (*x));;

   /* get memory for each row */
   for (i = 0; i < rows; i++)
     MALLOC(x[i], cols * sizeof(**x));
   return x;
}
```

Program 2.3: Dynamically create a two-dimensional array

allocated bits are set to 0); a pointer to the start of the allocated memory is returned. In case there is insufficient memory to make the allocation, the returned value is *NULL*. So, for example, the statements

```
int *x;
x = calloc(n, sizeof(int));
```

could be used to define a one-dimensional array of integers; the capacity of this array is n, and $x[0{:}n-1]$ are initially 0. As was the case with *malloc*, it is useful to define the macro *CALLOC* as below and use this macro to write clean robust programs.

```
#define CALLOC(p,n,s)\
        if (!((p) = calloc(n,s))) {\
               fprintf(stderr, "Insufficient memory"); \
               exit(EXIT_FAILURE);\
        }
```

The function *realloc* resizes memory previously allocated by either *malloc* or *calloc*. For example, the statement

```
realloc(p, s)
```

changes the size of the memory block pointed at by p to s. The contents of the first $\min\{s, oldSize\}$ bytes of the block are unchanged as a result of this resizing. When $s>oldSize$ the additional $s - oldSize$ have an unspecified value and when $s<oldSize$, the

rightmost *olsSize* − *s* bytes of the old block are freed. When *realloc* is able to do the resizing, it returns a pointer to the start of the new block and when it is unable to do the resizing, the old block is unchanged and the function returns the value *NULL*.

As with *malloc* and *calloc*, it is useful to define a macro *REALLOC* as below.

```
#define REALLOC(p,s)\
        if (!((p) = realloc(p,s))) {\
                fprintf(stderr, "Insufficient memory");\
                exit(EXIT_FAILURE);\
        }
```

A three-dimensional array is represented as a one-dimensional array, each of whose elements is a two-dimensional array. Each of these two-dimensional arrays is represented as shown in Figure 2.2.

EXERCISES

1. Make the fewest number of changes to Program 2.3 so as to obtain a function that creates a two-dimensional array all of whose elements are set to 0. Test your new function.
2. Let *length* [*i*] be the desired length (size or number of elements) of row *i* of a two-dimensional array. Write a funtion similar to Program 2.3 to create a two-dimensional array such that row *i* has *length* [*i*] elements, $0 \leq i < rows$. Test your code.
3. Rewrite the matrix add function of Program 1.16 using dynamically allocated arrays. The header for your function should be

   ```
   void add(int **a, int **b, int **c, int rows, int cols)
   ```

   Test your function
4. Rewrite the matrix multiplication function of Program 1.20 using dynamically allocated arrays. The header for your function should be

   ```
   void mult(int **a, int **b, int **c, int rows)
   ```

   where each matrix is a *rows* × *rows* matrix. Test your function
5. Rewrite the matrix transpose function of Program 1.22 using dynamically allocated arrays. The header for your function should be

   ```
   void transpose(int **a, int rows)
   ```

   Test your function

6. Write a matrix transpose function for matrices that may not be square. Use dynamically allocated arrays. The header for your function should be

```
void transpose(int **a, int **b, int rows, int cols)
```

where a is the $rows \times cols$ matrix that is to be transposed and b is the transposed matrix computed by the function. Note that the transposed matrix is a $cols \times rows$ matrix. Test your function

2.3 STRUCTURES AND UNIONS

2.3.1 Structures

Arrays are collections of data of the same type. In C there is an alternative way to group data that permits the data to vary in type. This mechanism is called the **struct**, short for structure. A structure (called a record in many other programming languages) is a collection of data items, where each item is identified as to its type and name. For example,

```
struct {
       char name[10];
       int age;
       float salary;
       } person;
```

creates a variable whose name is *person* and that has three fields:

- a name that is a character array
- an integer value representing the age of the person
- a **float** value representing the salary of the individual

We may assign values to these fields as below. Notice the use of the . as the structure member operator. We use this operator to select a particular member of the structure.

```
strcpy(person.name,"james");
person.age = 10;
person.salary = 35000;
```

We can create our own structure data types by using the **typedef** statement as below:

```
typedef struct {
        char name[10];
        int age;          float salary;
        } humanBeing;
```

This says that *humanBeing* is the name of the type defined by the structure definition, and we may follow this definition with declarations of variables such as:

```
humanBeing person1, person2;
```

We might have a program segment that says:

```
if (strcmp(person1.name, person2.name))
  printf("The two people do not have the same name\n");
else
  printf("The two people have the same name\n");
```

It would be nice if we could write `if (person1 == person2)` and have the entire structure checked for equality, or if we could write `person1 = person2` and have that mean that the value of every field of the structure of *person* 2 is assigned as the value of the corresponding field of *person* 1. ANSI C permits structure assignment, but most earlier versions of C do not. For older versions of C, we are forced to write the more detailed form:

```
strcpy(person1.name, person2.name);
person1.age = person2.age;
person1.salary = person2.salary;
```

While structures cannot be directly checked for equality or inequality, we can write a function (Program 2.4) to do this. *TRUE* and *FALSE* are defined as:

```
#define FALSE 0
#define TRUE 1
```

A typical function call might be:

```
if (humansEqual(person1,person2))
  printf("The two human beings are the same\n");
else
  printf("The two human beings are not the same\n");
```

```
int humansEqual(humanBeing person1,
                humanBeing person2)
{/* return TRUE if person1 and person2 are the same human
    being otherwise return FALSE */
   if (strcmp(person1.name, person2.name))
     return FALSE;
   if (person1.age != person2.age)
     return FALSE;
   if (person1.salary != person2.salary)
     return FALSE;
   return TRUE;
}
```

Program 2.4: Function to check equality of structures

We can also embed a structure within a structure. For example, associated with our *humanBeing* structure we may wish to include the date of his or her birth. We can do this by writing:

```
typedef struct {
        int month;
        int day;
        int year;
        } date;

typedef struct {
        char name[10];
        int age;
        float salary;
        date dob;
        } humanBeing;
```

A person born on February 11, 1944, would have the values for the *date* **struct** set as:

```
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

2.3.2 Unions

Continuing with our *humanBeing* example, it would be nice if we could distinguish between males and females. In the case of males we might ask whether they have a beard or not. In the case of females we might wish to know the number of children they have borne. This gives rise to another feature of C called a **union**. A **union** declaration is similar to a structure, but the fields of a **union** must share their memory space. This means that only one field of the **union** is "active" at any given time. For example, to add different fields for males and females we would change our definition of *humanBeing* to:

```
typedef struct {
   enum tagField {female, male} sex;
   union {
      int children;
      int beard ;
      } u;
   } sexType;
typedef struct {
   char name[10];
   int age;
   float salary;
   date dob;
   sexType sexInfo;
   } humanBeing;
humanBeing person1, person2;
```

We could assign values to *person*1 and *person*2 as:

```
person1.sexInfo.sex = male;
person1.sexInfo.u.beard = FALSE;
```

and

```
person2.sexInfo.sex = female;
person2.sexInfo.u.children = 4;
```

Notice that we first place a value in the tag field. This allows us to determine which field in the **union** is active. We then place a value in the appropriate field of the **union**. For example, if the value of *sexInfo.sex* was *male*, we would enter a *TRUE* or a *FALSE* in the *sexInfo.u.beard* field. Similarly, if the person was a *female*, we would enter an integer value in the *sexInfo.u.children* field. C does not verify that we use the appropriate field. For instance, we could place a value of *female* in the *sexInfo.sex* field, and then proceed to place a value of *TRUE* in the *sexInfo.u.beard* field. Although we know that this is not appropriate, C does not require us to use the correct fields of a **union**.

2.3.3 Internal Implementation Of Structures

In most cases you need not be concerned with exactly how the C compiler will store the fields of a structure in memory. Generally, if you have a structure definition such as:

```
struct {int i,j; float a, b;};
```

or

```
struct {int i; int j; float a; float b; };
```

these values will be stored in the same way using increasing address locations in the order specified in the structure definition. However, it is important to realize that holes or padding may actually occur within a structure to permit two consecutive components to be properly aligned within memory.

The size of an object of a **struct** or **union** type is the amount of storage necessary to represent the largest component, including any padding that may be required. Structures must begin and end on the same type of memory boundary, for example, an even byte boundary or an address that is a multiple of 4, 8, or 16.

2.3.4 Self-Referential Structures

A *self-referential structure* is one in which one or more of its components is a pointer to itself. Self-referential structures usually require dynamic storage management routines (*malloc* and *free*) to explicitly obtain and release memory. Consider as an example:

```
typedef struct {
        char data;
        struct list *link ;
        } list;
```

Each instance of the structure *list* will have two components, *data* and *link*. *data* is a single character, while *link* is a pointer to a *list* structure. The value of *link* is either the address in memory of an instance of *list* or the null pointer. Consider these statements, which create three structures and assign values to their respective fields:

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;
```

Structures *item* 1, *item* 2, and *item* 3 each contain the data item *a*, *b*, and *c*, respectively,

and the null pointer. We can attach these structures together by replacing the null *link* field in *item* 2 with one that points to *item* 3 and by replacing the null *link* field in *item* 1 with one that points to *item* 2.

```
item1.link = &item2;
item2.link = &item3;
```

We will see more of this linking in Chapter 4.

EXERCISES

1. Develop a structure to represent the planets in the solar system. Each planet has fields for the planet's name, its distance from the sun (in miles), and the number of moons it has. Place items in each the fields for the planets: Earth and Venus.
2. Modify the *humanBeing* structure so that we can include different information based on marital status. Marital status should be an enumerated type with fields: single, married, widowed, divorced. Use a **union** to include different information based on marital status as follows:
   - *Single*. No information needed.
   - *Married*. Include a marriage date field.
   - *Widowed*. Include marriage date and death of spouse date fields.
   - *Divorced*. Include divorce date and number of divorces fields.

   Assign values to the fields for some *person* of type *humanBeing*.
3. Develop a structure to represent each of the following geometric objects: *rectangle*, *triangle*, and *circle*.

2.4 POLYNOMIALS

2.4.1 The Abstract Data Type

Arrays are not only data structures in their own right, we can also use them to implement other abstract data types. For instance, let us consider one of the simplest and most commonly found data structures: the *ordered* or *linear list*. We can find many examples of this data structure, including:

- Days of the week: (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
- Values in a deck of cards: (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)

- Floors of a building: (basement, lobby, mezzanine, first, second)
- Years the United States fought in World War II: (1941, 1942, 1943, 1944, 1945)
- Years Switzerland fought in World War II: ( )

Notice that the years Switzerland fought in World War II is different because it contains no items. It is an example of an empty list, which we denote as ( ). The other lists all contain items that are written in the form $(item_0, item_1, \cdots, item_{n-1})$.

We can perform many operations on lists, including:

- Finding the length, n, of a list.
- Reading the items in a list from left to right (or right to left).
- Retrieving the ith item from a list, $0 \le i < n$.
- Replacing the item in the ith position of a list, $0 \le i < n$.
- Inserting a new item in the ith position of a list, $0 \le i \le n$. The items previously numbered $i, i+1, \cdots, n-1$ become items numbered $i+1, i+2, \cdots, n$.
- Deleting an item from the ith position of a list, $0 \le i < n$. The items numbered $i+1, \cdots, n-1$ become items numbered $i, i+1, \cdots, n-2$.

Rather than state the formal specification of the ADT *list*, we want to explore briefly its implementation. Perhaps, the most common implementation is to represent an ordered list as an array where we associate the list element, $item_i$, with the array index i. We call this a sequential mapping because, assuming the standard implementation of an array, we are storing $item_i, item_{i+1}$ into consecutive slots i and $i+1$ of the array. Sequential mapping works well for most of the operations listed above. Thus, we can retrieve an item, replace an item, or find the length of a list, in constant time. We also can read the items in the list, from either direction, by simply changing subscripts in a controlled way. Only insertion and deletion pose problems since the sequential allocation forces us to move items so that the sequential mapping is preserved. It is precisely this overhead that leads us to consider nonsequential mappings of ordered lists in Chapter 4.

Let us jump right into a problem requiring ordered lists, which we will solve by using one-dimensional arrays. This problem has become the classical example for motivating the use of list processing techniques, which we will see in later chapters. Therefore, it makes sense to look at the problem and see why arrays offer only a partially adequate solution. The problem calls for building a set of functions that allow for the manipulation of symbolic polynomials. Viewed from a mathematical perspective, a polynomial is a sum of terms, where each term has a form ax^e, where x is the variable, a is the coefficient, and e is the exponent. Two example polynomials are:

$$A(x) = 3x^{20} + 2x^5 + 4 \text{ and } B(x) = x^4 + 10x^3 + 3x^2 + 1$$

The largest (or leading) exponent of a polynomial is called its *degree*. Coefficients that

are zero are not displayed. The term with exponent equal to zero does not show the variable since x raised to a power of zero is 1. There are standard mathematical definitions for the sum and product of polynomials. Assume that we have two polynomials, $A(x) = \sum a_i x^i$ and $B(x) = \sum b_i x^i$ then:

$$A(x) + B(x) = \sum (a_i + b_i) x^i$$

$$A(x) \cdot B(x) = \sum (a_i x^i \cdot \sum (b_j x^j))$$

Similarly, we can define subtraction and division on polynomials, as well as many other operations.

We begin with an ADT definition of a polynomial. The particular operations in part are a reflection of what will be needed in our subsequent programs to manipulate polynomials. The definition is contained in ADT 2.2.

2.4.2 Polynomial Representation

We are now ready to make some representation decisions. A very reasonable first decision requires unique exponents arranged in decreasing order. This requirement considerably simplifies many of the operations. Using our specification and this stipulation, we can write a version of *Add* that is closer to a C function (Program 2.5), but is still representation-independent.

This algorithm works by comparing terms from the two polynomials until one or both of the polynomials becomes empty. The **switch** statement performs the comparisons and adds the proper term to the new polynomial, *d*. If one of the polynomials becomes empty, we copy the remaining terms from the nonempty polynomial into *d*. With these insights, we now consider the representation question more carefully.

One way to represent polynomials in C is to use **typedef** to create the type *polynomial* as below:

```
#define MAX_DEGREE 101 /*Max degree of polynomial+1*/
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

Now if *a* is of type *polynomial* and $n <$ *MAX_DEGREE*, the polynomial $A(x) = \sum_{i=0}^{n} a_i x^i$ would be represented as:

```
a.degree = n
a.coef[i] = a_{n-i}, 0 ≤ i ≤ n
```

ADT *Polynomial* is

objects: $p(x) = a_1 x^{e_1} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $<e_i, a_i>$ where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers >= 0

functions:

for all *poly*, *poly*1, *poly*2 ∈ *Polynomial*, *coef* ∈ *Coefficients*, *expon* ∈ *Exponents*

Polynomial Zero()	::=	**return** the polynomial, $p(x) = 0$
Boolean IsZero(*poly*)	::=	**if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient Coef(*poly*,*expon*)	::=	**if** (*expon* ∈ *poly*) **return** its coefficient **else return** zero
Exponent LeadExp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** (*expon* ∈ *poly*) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
Polynomial Remove(*poly*, *expon*)	::=	**if** (*expon* ∈ *poly*) **return** the polynomial *poly* with the term whose exponent is *expon* deleted **else return** error
Polynomial SingleMult(*poly*, *coef*, *expon*)	::=	**return** the polynomial $poly \cdot coef \cdot x^{expon}$
Polynomial Add(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 + *poly*2
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 · *poly*2

end *Polynomial*

ADT 2.2: Abstract data type *Polynomial*

In this representation, we store the coefficients in order of decreasing exponents, such that $a.coef[i]$ is the coefficient of x^{n-i} provided a term with exponent $n-i$ exists; otherwise, $a.coef[i] = 0$. Although this representation leads to very simple algorithms for most of the operations, it wastes a lot of space. For instance, if $a.degree << MAX\_DEGREE$, (the double "less than" should be read as "is much less than"), then we will not need most of the positions in $a.coef[MAX\_DEGREE]$. The same argument

```
/* d = a + b, where a, b, and d are polynomials */
d = Zero()
while (! IsZero(a) && ! IsZero(b)) do {
   switch COMPARE(LeadExp(a), LeadExp(b)) {
      case -1: d =
         Attach(d,Coef(b,LeadExp(b)),LeadExp(b));
         b = Remove(b,LeadExp(b));
         break;
      case 0: sum = Coef( a, LeadExp(a))
                  + Coef(b, LeadExp(b));
         if (sum) {
            Attach(d,sum,LeadExp(a));
            a = Remove(a,LeadExp(a));
            b = Remove(b,LeadExp(b));
            }
         break;
      case 1: d =
         Attach(d,Coef(a,LeadExp(a)),LeadExp(a));
         a = Remove(a,LeadExp(a));
   }
}
insert any remaining terms of a or b into d
```

Program 2.5: Initial version of *padd* function

applies if the polynomial is sparse, that is, the number of terms with nonzero coefficient is small relative to the degree of the polynomial. To preserve space we devise an alternate representation that uses only one global array, *terms*, to store all our polynomials. The C declarations needed are:

```
MAX_TERMS 100 /*size of terms array*/
typedef struct {
        float coef;
        int expon;
        } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

Consider the two polynomials $A(x) = 2x^{1000} + 1$ and $B(x) = x^4 + 10x^3 + 3x^2 + 1$. Figure 2.3 shows how these polynomials are stored in the array *terms*. The index of the

first term of A and B is given by *startA* and *startB*, respectively, while *finishA* and *finishB* give the index of the last term of A and B. The index of the next free location in the array is given by *avail*. For our example, *startA* = 0, *finishA* = 1, *startB* = 2, *finishB* = 5, and *avail* = 6.

	startA ↓	*finishA* ↓	*startB* ↓			*finishB* ↓	*avail* ↓
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

Figure 2.3: Array representation of two polynomials

This representation does not impose any limit on the number of polynomials that we can place in *terms*. The only stipulation is that the total number of nonzero terms must be no more than *MAX_TERMS*. It is worth pointing out the difference between our specification and our representation. Our specification used *poly* to refer to a polynomial, and our representation translated *poly* into a <*start*, *finish*> pair. Therefore, to use $A(x)$ we must pass in *startA* and *finishA*. Any polynomial A that has n nonzero terms has *startA* and *finishA* such that *finishA* = *startA* + n − 1.

Before proceeding, we should evaluate our current representation. Is it any better than the representation that uses an array of coefficients for each polynomial? It certainly solves the problem of many zero terms since $A(x) = 2x^{1000} + 1$ uses only six units of storage: one for *startA*, one for *finishA*, two for the coefficients, and two for the exponents. However, when all the terms are nonzero, the current representation requires about twice as much space as the first one. Unless we know before hand that each of our polynomials has few zero terms, our current representation is probably better.

2.4.3 Polynomial Addition

We would now like to write a C function that adds two polynomials, A and B, represented as above to obtain $D = A + B$. To produce $D(x)$, *padd* (Program 2.6) adds $A(x)$ and $B(x)$ term by term. Starting at position *avail*, *attach* (Program 2.7) places the terms of D into the array, *terms*. If there is not enough space in *terms* to accommodate D, an error message is printed to the standard error device and we exit the program with an error condition.

```
void padd(int startA,int finishA,int startB, int finishB,
                                  int *startD,int *finishD)
{/* add A(x) and B(x) to obtain D(x) */
   float coefficient;
   *startD = avail;
   while (startA <= finishA && startB <= finishB)
      switch(COMPARE(terms[startA].expon,
                     terms[startB].expon)) {
         case -1: /* a expon < b expon */
               attach(terms[startB].coef,terms[startB].expon);
               startB++;
               break;
         case 0: /* equal exponents */
               coefficient = terms[startA].coef +
                             terms[startB].coef;
               if (coefficient)
                  attach(coefficient,terms[startA].expon);
               startA++;
               startB++;
               break;
         case 1: /* a expon > b expon */
               attach(terms[startA].coef,terms[startA].expon);
               startA++;
   }
   /* add in remaining terms of A(x) */
   for(; startA <= finishA; startA++)
      attach(terms[startA].coef,terms[startA].expon);
   /* add in remaining terms of B(x) */
   for( ; startB <= finishB; startB++)
      attach(terms[startB].coef, terms[startB].expon);
   *finishD = avail-1;
}
```

Program 2.6: Function to add two polynomials

Analysis of *padd*: Since the number of nonzero terms in A and in B are the most important factors in the time complexity, we will carry out the analysis using them. Therefore, let m and n be the number of nonzero terms in A and B, respectively. If $m > 0$ and $n > 0$, the **while** loop is entered. Each iteration of the loop requires O(1) time. At each iteration, we increment the value of *startA* or *startB* or both. Since the iteration terminates

```
void attach(float coefficient, int exponent)
{/* add a new term to the polynomial */
   if (avail >= MAX_TERMS) {
      fprintf(stderr,"Too many terms in the polynomial\n");
      exit(EXIT_FAILURE);
   }
   terms[avail].coef = coefficient;
   terms[avail++].expon = exponent;
}
```

Program 2.7: Function to add a new term

when either *startA* or *startB* exceeds *finishA* or *finishB*, respectively, the number of iterations is bounded by $m + n - 1$. This worst case occurs when:

$$A(x) = \sum_{i=0}^{n} x^{2i} \text{ and } B(x) = \sum_{i=0}^{n} x^{2i+1}$$

The time for the remaining two loops is bounded by $O(n + m)$ because we cannot iterate the first loop more than m times and the second more than n times. So, the asymptotic computing time of this algorithm is $O(n+m)$. □

Before proceeding let us briefly consider a few of the problems with the current representation. We have seen that, as we create polynomials, we increment *avail* until it equals *MAX_TERMS*. When this occurs, must we quit? Given the current representation, we must unless there are some polynomials that we no longer need. We could write a compaction function that would remove the unnecessary polynomials and create a large, continuous available space at one end of the array. However, this requires data movement which takes time. In addition, we also must change the values of *start* and *finish* for every polynomial that is moved. In Chapter 3, we let you experiment with some "simple" compacting routines.

EXERCISES

1. Consider the type definition

```
typedef struct {
        int degree;
        int capacity;
        float* coef;
        } dpolynomial;
```

where *coef* is the dynamically allocated one-dimensional array *coef* [0:*capacity* − 1]. Compare this representation for polynomials with the one using the type *polynomial*.

2. Write functions *readPoly* and *printPoly* that allow the user to create and print polynomials.
3. Write a function, *pmult*, that multiplies two polynomials. Figure out the computing time of your function.
4. Write a function, *peval*, that evaluates a polynomial at some value, x_0. Try to minimize the number of operations.
5. Let $A(x) = x^{2n} + x^{2n-2} + \cdots + x^2 + x^0$ and $B(x) = x^{2n+1} + x^{2n} + \cdots + x^3 + x$. For these polynomials, determine the exact number of times each statement of *padd* is executed.
6. The declarations that follow give us another representation of the polynomial ADT. *terms* [*i*][0].*expon* gives the number of nonzero terms in the *i*th polynomial. These terms are stored, in descending order of exponents, in positions *terms* [*i*][1], *terms* [*i*][2], $\cdots$. Create the functions *readPoly*, *printPoly*, *padd*, and *pmult* for this representation. Is this representation better or worse than the representation used in the text? (You may add declarations as necessary.)

```
#define MAX_TERMS 101 /* maximum number of terms + 1*/
#define MAX_POLYS 15 /* maximum number of
                        polynomials*/
typedef struct {
        float coef;
        int expon;
        } polynomial;
polynomial terms[MAX_POLYS][MAX_TERMS];
```

2.5 SPARSE MATRICES

2.5.1 The Abstract Data Type

We now turn our attention to a mathematical object that is used to solve many problems in the natural sciences, the matrix. As computer scientists, our interest centers not only on the specification of an appropriate ADT, but also on finding representations that let us efficiently perform the operations described in the specification.

In mathematics, a matrix contains m rows and n columns of elements as illustrated in Figure 2.4. In this figure, the elements are numbers. The first matrix has five rows and three columns; the second has six rows and six columns. In general, we write $m \times n$

(read "m by n") to designate a matrix with m rows and n columns. The total number of elements in such a matrix is mn. If m equals n, the matrix is square.

	col 0	col 1	col 2
row 0	−27	3	4
row 1	6	82	−2
row 2	109	−64	11
row 3	12	8	9
row 4	48	27	47

(a)

	col 0	col 1	col 2	col 3	col 4	col 5
row 0	15	0	0	22	0	−15
row 1	0	11	3	0	0	0
row 2	0	0	0	−6	0	0
row 3	0	0	0	0	0	0
row 4	91	0	0	0	0	0
row 5	0	0	28	0	0	0

(b)

Figure 2.4: Two matrices

When a matrix is represented as a two-dimensional array defined as *a[MAX_ROWS][MAX_COLS]*, we can locate quickly any element by writing *a[i][j]*, where *i* is the row index and *j* is the column index. However, there are some problems with this representation. For instance, if you look at the matrix of Figure 2.4(b), you notice that it contains many zero entries. We call this a *sparse matrix*. Although it is difficult to determine exactly whether a matrix is sparse or not, intuitively we can recognize a sparse matrix when we see one. In Figure 2.4(b), only 8 of 36 elements are nonzero and that certainly is sparse. When a sparse matrix is represented as a two-dimensional array, we waste space. For example, consider the space requirements necessary to store a 1000×1000 matrix that has only 2000 non-zero elements. The corresponding two-dimensional array requires space for 1,000,000 elements! We can do much better by using a representation in which only the nonzero elements are stored.

Before developing a particular representation, we first must consider the operations that we want to perform on these matrices. A minimal set of operations includes matrix creation, addition, multiplication, and transpose. ADT 2.3 contains our specification of the matrix ADT.

ADT *SparseMatrix* is

objects: a set of triples, <*row*, *column*, *value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.

functions:

for all $a, b \in$ *SparseMatrix*, $x \in$ *item*, i, j, *maxCol*, *maxRow* $\in$ *index*

SparseMatrix Create(*maxRow*, *maxCol*) ::=
return a *SparseMatrix* that can hold up to *maxItems* = *maxRow* × *maxCol* and whose maximum row size is *maxRow* and whose maximum column size is *maxCol*.

SparseMatrix Transpose(a) ::=
return the matrix produced by interchanging the row and column value of every triple.

SparseMatrix Add(a, b) ::=
if the dimensions of a and b are the same
return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.
else return error

SparseMatrix Multiply(a, b) ::=
if number of columns in a equals number of rows in b
return the matrix d produced by multiplying a by b according to the formula: $d[i][j] = \sum(a[i][k] \cdot b[k][j])$ where $d(i, j)$ is the (i, j)th element
else return error.

ADT 2.3: Abstract data type *SparseMatrix*

2.5.2 Sparse Matrix Representation

Before implementing any of the ADT operations, we must establish the representation of the sparse matrix. By examining Figure 2.4, we know that we can characterize uniquely any element within a matrix by using the triple <*row*, *col*, *value*>. This means that we can use an array of triples to represent a sparse matrix. Since we want our transpose operation to work efficiently, we should organize the triples so that the row indices are in ascending order. We can go one step further by also requiring that all the triples for any row be stored so that the column indices are in ascending order. In addition, to ensure

that the operations terminate, we must know the number of rows and columns, and the number of nonzero elements in the matrix. Putting all this information together suggests that we implement the *Create* operation as below:

SparseMatrix Create(*maxRow*, *maxCol*) ::=

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
typedef struct {
        int col;
        int row;
        int value;
        } term;
term a[MAX_TERMS];
```

Since *MAX_TERMS* is greater than eight, these statements can be used to represent the second sparse matrix from Figure 2.4. Figure 2.5(a) shows how this matrix is represented in the array *a*. Thus, *a*[0].*row* contains the number of rows; *a*[0].*col* contains the number of columns; and *a*[0].*value* contains the total number of nonzero entries. Positions 1 through 8 store the triples representing the nonzero entries. The row index is in the field *row*; the column index is in the field *col*; and the value is in the field *value*. The triples are ordered by row and within rows by columns.

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	−6
[8]	5	0	−15

(b)

Figure 2.5: Sparse matrix and its transpose stored as triples

2.5.3 Transposing A Matrix

Figure 2.5(b) shows the transpose of the sample matrix. To transpose a matrix we must interchange the rows and columns. This means that each element $a[i][j]$ in the original matrix becomes element $b[j][i]$ in the transpose matrix. Since we have organized the original matrix by rows, we might think that the following is a good algorithm for transposing a matrix:

```
for each row i
   take element <i, j, value> and store it
   as element <j, i, value> of the transpose;
```

Unfortunately, if we process the original matrix by the row indices we will not know exactly where to place element *<j, i, value>* in the transpose matrix until we have processed all the elements that precede it. For instance, in Figure 2.5, we have:

(0, 0, 15),	which becomes	(0, 0, 15)
(0, 3, 22),	which becomes	(3, 0, 22)
(0, 5, −15),	which becomes	(5, 0, −15)

If we place these triples consecutively in the transpose matrix, then, as we insert new triples, we must move elements to maintain the correct order. We can avoid this data movement by using the column indices to determine the placement of elements in the transpose matrix. This suggests the following algorithm:

```
for all elements in column j
   place element <i, j, value> in
   element <j, i, value>
```

The algorithm indicates that we should "find all the elements in column 0 and store them in row 0 of the transpose matrix, find all the elements in column 1 and store them in row 1, etc." Since the original matrix ordered the rows, the columns within each row of the transpose matrix will be arranged in ascending order as well. This algorithm is incorporated in *transpose* (Program 2.8). The first array, a, is the original array, while the second array, b, holds the transpose.

It is not too difficult to see that the function works correctly. The variable, *currentb*, holds the position in b that will contain the next transposed term. We generate the terms in b by rows, but since the rows in b correspond to the columns in a, we collect the nonzero terms for row i of b by collecting the nonzero terms from column i of a.

```
void transpose(term a[], term b[])
{/* b is set to the transpose of a */
   int n,i,j, currentb;
   n = a[0].value;       /* total number of elements */
   b[0].row = a[0].col; /* rows in b = columns in a */
   b[0].col = a[0].row; /* columns in b = rows in a */
   b[0].value = n;
   if (n > 0 )  { /* non zero matrix */
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by the columns in a */
         for (j = 1; j <= n; j++)
         /* find elements from the current column */
            if (a[j].col == i) {
            /* element is in current column, add it to b */
               b[currentb].row = a[j].col;
               b[currentb].col = a[j].row;
               b[currentb].value = a[j].value;
               currentb++;
            }
   }
}
```

Program 2.8: Transpose of a sparse matrix

Analysis of *transpose:* Determining the computing time of this algorithm is easy since the nested **for** loops are the decisive factor. The remaining statements (two **if** statements and several assignment statements) require only constant time. We can see that the outer **for** loop is iterated $a[0].col$ times, where $a[0].col$ holds the number of columns in the original matrix. In addition, one iteration of the inner **for** loop requires $a[0].value$ time, where $a[0].value$ holds the number of elements in the original matrix. Therefore, the total time for the nested **for** loops is $columns \cdot elements$. Hence, the asymptotic time complexity is $O(columns \cdot elements)$. □

We now have a matrix transpose algorithm with a computing time of $O(columns \cdot elements)$. This time is a little disturbing since we know that if we represented our matrices as two-dimensional arrays of size $rows \times columns$, we could obtain the transpose in $O(rows \cdot columns)$ time. The algorithm to accomplish this has the simple form:

```
for (j = 0; j < columns; j++)
  for (i = 0; i < rows; i++)
    b[j][i] = a[i][j];
```

The O(*columns* · *elements*) time for our transpose function becomes O($columns^2 \cdot rows$) when the number of elements is of the order *columns* · *rows*. Perhaps, to conserve space, we have traded away too much time. Actually, we can create a much better algorithm by using a little more storage. In fact, we can transpose a matrix represented as a sequence of triples in O(*columns* + *elements*) time. This algorithm, *fastTranspose* (Program 2.9), proceeds by first determining the number of elements in each column of the original matrix. This gives us the number of elements in each row of the transpose matrix. From this information, we can determine the starting position of each row in the transpose matrix. We now can move the elements in the original matrix one by one into their correct position in the transpose matrix. We assume that the number of columns in the original matrix never exceeds *MAX_COL*.

```
void fastTranspose(term a[], term b[])
{/* the transpose of a is placed in b */
  int rowTerms[MAX_COL], startingPos[MAX_COL];
  int i,j, numCols = a[0].col, numTerms = a[0].value;
  b[0].row = numCols;  b[0].col = a[0].row;
  b[0].value = numTerms;
  if (numTerms > 0) { /* nonzero matrix */
    for (i = 0; i < numCols; i++)
      rowTerms[i] = 0;
    for (i = 1; i <= numTerms; i++)
      rowTerms[a[i].col]++;
    startingPos[0] = 1;
    for (i = 1; i < numCols; i++)
      startingPos[i] =
                  startingPos[i-1] + rowTerms[i-1];
    for (i = 1; i <= numTerms; i++) {
      j = startingPos[a[i].col]++;
      b[j].row = a[i].col;   b[j].col = a[i].row;
      b[j].value = a[i].value;
    }
  }
}
```

Program 2.9: Fast transpose of a sparse matrix

Analysis of *fastTranspose:* We can verify that *fastTranspose* works correctly from the preceding discussion and the observation that the starting point of row i, $i > 1$ of the transpose matrix is *rowTerms*[$i-1$] + *startingPos*[$i-1$], where *rowTerms*[$i-1$] is the number of elements in row $i-1$ and *startingPos*[$i-1$] is the starting point of row $i-1$. The first two **for** loops compute the values for *rowTerms*, the third **for** loop carries out the computation of *startingPos*, and the last **for** loop places the triples into the transpose matrix. These four loops determine the computing time of *fastTranspose*. The bodies of the loops are executed *numCols*, *numTerms*, *numCols* − 1, and *numTerms* times, respectively. Since the statements within the loops require only constant time, the computing time for the algorithm is O(*columns* + *elements*). The time becomes O(*columns* · *rows*) when the number of elements is of the order *columns* · *rows*. This time equals that of the two-dimensional array representation, although *fastTranspose* has a larger constant factor. However, when the number of elements is sufficiently small compared to the maximum of *columns* · *rows*, *fastTranspose* is much faster. Thus, in this representation we save both time and space. This was not true of *transpose* since the number of elements is usually greater than max{*columns, rows*} and *columns* · *elements* is always at least *columns* · *rows*. In addition, the constant factor for *transpose* is bigger than that found in the two-dimensional array representation. However, *transpose* requires less space than *fastTranspose* since the latter function must allocate space for the *rowTerms* and *startingPos* arrays. We can reduce this space to one array if we put the starting positions into the space used by the row terms as we calculate each starting position. □

If we try the algorithm on the sparse matrix of Figure 2.5(a), then after the execution of the third **for** loop, the values of *rowTerms* and *startingPos* are:

	[0]	[1]	[2]	[3]	[4]	[5]
rowTerms =	2	1	2	2	0	1
startingPos =	1	3	4	6	8	8

The number of entries in row i of the transpose is contained in *rowTerms*[i]. The starting position for row i of the transpose is held by *startingPos*[i].

2.5.4 Matrix Multiplication

A second operation that arises frequently is matrix multiplication, which is defined below.

Definition: Given A and B where A is $m \times n$ and B is $n \times p$, the product matrix D has dimension $m \times p$. Its $<i, j>$ element is :

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik}\, b_{kj}$$

for $0 \le i < m$ and $0 \le j < p$. □

The product of two sparse matrices may no longer be sparse, as Figure 2.6 shows.

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 2.6: Multiplication of two sparse matrices

We would like to multiply two sparse matrices represented as an ordered list (Figure 2.5). We need to compute the elements of D by rows so that we can store them in their proper place without moving previously computed elements. To do this we pick a row of A and find all elements in column j of B for $j = 0, 1, \cdots, colsB - 1$. Normally, we would have to scan all of B to find all the elements in column j. However, we can avoid this by first computing the transpose of B. This puts all column elements in consecutive order. Once we have located the elements of row i of A and column j of B we just do a merge operation similar to that used in the polynomial addition of Section 2.2. (We explore an alternate approach in the exercises at the end of this section.)

To obtain the product matrix D, *mmult* (Program 2.10) multiplies matrices A and B using the strategy outlined above. We store the matrices A, B, and D in the arrays a, b, and d, respectively. To place a triple in d and to reset *sum* to 0, *mmult* uses *storeSum* (Program 2.11). In addition, *mmult* uses several local variables that we will describe briefly. The variable *row* is the row of A that we are currently multiplying with the columns in B. The variable *rowBegin* is the position in a of the first element of the current row, and the variable *column* is the column of B that we are currently multiplying with a row in A. The variable *totalD* is the current number of elements in the product matrix D. The variables i and j are used to examine successively elements from a row of A and a column of B. Finally, the variable *newB* is the sparse matrix that is the transpose of b. Notice that we have introduced an additional term into both a (a[*totalA*+1].row = *rowsA*;) and *newB* (*newB*[*totalB*+1].row = *colsB*;). These dummy terms serve as sentinels that enable us to obtain an elegant algorithm.

```
void mmult(term a[], term b[], term d[])
{/* multiply two sparse matrices */
   int i, j, column, totalB = b[0].value, totalD = 0;
   int rowsA = a[0].row, colsA = a[0].col,
   totalA = a[0].value; int  colsB = b[0].col,
   int rowBegin = 1, row = a[1].row, sum = 0;
   int newB[MAX_TERMS][3];
   if (colsA != b[0].row) {
      fprintf(stderr,"Incompatible matrices\n");
      exit(EXIT_FAILURE);
   }
   fastTranspose(b,newB);
   /* set boundary condition */
   a[totalA+1].row = rowsA;
   newB[totalB+1].row = colsB;
   newB[totalB+1].col = 0;
   for (i = 1; i <= totalA; ) {
      column = newB[1].row;
      for (j = 1; j <= totalB+1;) {
      /* multiply row of a by column of b */
         if (a[i].row != row) {
            storeSum(d,&totalD,row,column,&sum);
            i = rowBegin;
            for (; newB[j].row == column; j++)
               ;
            column = newB[j].row;
         }
         else if (newB[j].row != column) {
            storeSum(d, &totalD, row, column, &sum);
            i = rowBegin;
            column = newB[j].row;
         }
         else switch (COMPARE(a[i].col, newB[j].col)) {
            case -1: /* go to next term in a */
                  i++;  break;
            case 0: /* add terms, go to next term in a and b*/
                  sum += ( a[i++].value * newB[j++].value);
                  break;
            case 1 : /* advance to next term in b */
                  j++;
         }
```

```
      }  /* end of for j <= totalB+1 */
      for (; a[i].row == row; i++)
         ;
      rowBegin = i; row = a[i].row;
   } /* end of for i<=totalA */
   d[0].row = rowsA;
   d[0].col = colsB; d[0].value =  totalD;
}
```

Program 2.10: Sparse matrix multiplication

```
void storeSum(term d[], int *totalD, int row, int column,
                                     int *sum)
{/* if *sum != 0, then it along with its row and column
    position is stored as the *totalD+1 entry in d */
   if (*sum)
      if (*totalD < MAX_TERMS) {
         d[++*totalD].row = row;
         d[*totalD].col = column;
         d[*totalD].value = *sum;
         *sum = 0;
      }
      else {
         fprintf(stderr,"Numbers of terms in product
                                exceeds %d\n",MAX_TERMS);
         exit(EXIT_FAILURE);
      }
}
```

Program 2.11: *storeSum* function

Analysis of *mmult:* We leave the correctness proof of *mmult* as an exercise and consider only its complexity. Besides the space needed for a, b, d, and a few simple variables, we also need space to store the transpose matrix *newB*. We also must include the additional space required by *fastTranspose*. The exercises explore a strategy for *mmult* that does not explicitly compute *newB*.

We can see that the lines before the first **for** loop require only O(*colsB* + *totalB*) time, which is the time needed to transpose b. The outer **for** loop is executed *totalA* times. At each iteration either i or j or both increase by 1, or i and *column* are reset. The maximum total increment in j over the entire loop is *totalB* + 1. If *termsRow* is the total

number of terms in the current row of A, then i can increase at most *termsRow* times before i moves to the next row of A. When this happens, we reset i to *rowBegin*, and, at the same time, advance *column* to the next column. Thus, this resetting takes place at most *colsB* time, and the total maximum increment in i is *colsB*termsRow*. Therefore, the maximum number of iterations of the outer **for** loop is *colsB* + *colsB*termsRow* + *totalB*. The time for the inner loop during the multiplication of the current row is O(*colsB*termsRow* + *totalB*), and the time to advance to the next row is O(*termsRow*). Thus, the time for one iteration of the outer **for** loop is O(*colsB*termsRow* + *totalB*). The overall time for this loop is:

$$O\left(\sum_{row}(colsB \cdot termsRow + totalB)\right) = O(colsB \cdot totalA + rowsA \cdot totalB) \quad \square$$

Once again we can compare this time with the computing time required to multiply matrices using the standard array representation. The classic multiplication algorithm is:

```
for (i = 0; i < rowsA; i++)
   for (j = 0; j < colsB; j++) {
      sum = 0;
      for (k = 0; k < colsA; k++)
         sum += (a[i][k] * b[k][j]);
         d[i][j] = sum;
   }
```

This algorithm takes O(*rowsA* · *colsA* · *colsB*) time. Since *totalA* ≤ *colsA* · *rowsA* and *totalB* ≤ *colsA* · *colsB*, the time for *mmult* is O(*rowsA* · *colsA* · *colsB*). However, its constant factor is greater than that of the classic algorithm. In the worst case, when *totalA* = *colsA* · *rowsA* or *totalB* = *colsA* · *colsB*, *mmult* is slower by a constant factor. However, when *totalA* and *totalB* are sufficiently smaller than their maximum value, that is, A and B are sparse, *mmult* outperforms the classic algorithm. The analysis of *mmult* is not trivial. It introduces some new concepts in algorithm analysis and you should make sure that you understand the analysis.

This representation of sparse matrices permits us to perform operations such as addition, transpose, and multiplication efficiently. However, there are other considerations that make this representation undesirable in certain applications. Since the number of terms in a sparse matrix is variable, we would like to represent all our sparse matrices in one array as we did for polynomials in Section 2.2. This would enable us to make efficient utilization of space. However, when this is done we run into difficulties in allocating space from the array to any individual matrix. These difficulties also occur with the polynomial representation and will become even more obvious when we study a similar representation for multiple stacks and queues in Section 3.4.

EXERCISES

1. Write C functions *readMatrix*, *printMatrix*, and *search* that read triples into a new sparse matrix, print out the terms in a sparse matrix, and search for a value in a sparse matrix. Analyze the computing time of each of these functions.
2. Rewrite *fastTranspose* so that it uses only one array rather than the two arrays required to hold *rowTerms* and *startingPos*.
3. Develop a correctness proof for the *mmult* function.
4. Analyze the time and space requirements of *fastTranspose*. What can you say about the existence of a faster algorithm?
5. Use the concept of an array of starting positions found in *fastTranspose* to rewrite *mmult* so that it multiplies sparse matrices A and B without transposing B. What is the computing time of your function?
6. As an alternate sparse matrix representation we keep only the nonzero terms in a one-dimensional array, *value*, in the order described in the text. In addition, we also maintain a two-dimensional array, *bits* [*rows*][*columns*], such that *bits*[i][j] = 0 if $a[i][j] = 0$ and *bits*[i][j] = 1 if $a[i][j] \neq 0$. Figure 2.7 illustrates the representation for the sparse matrix of Figure 2.5(b).

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 15 \\ 22 \\ -15 \\ 11 \\ 3 \\ -6 \\ 91 \\ 28 \end{bmatrix}$$

Figure 2.7: Alternate representation of a sparse matrix

(a) On a computer with w bits per word, how much storage is needed to represent a sparse matrix, A, with t nonzero terms?

(b) Write a C function to add two sparse matrices A and B represented as in Figure 2.7 to obtain $D = A + B$. How much time does your algorithm take?

(c) Discuss the merits of this representation versus the one used in the text. Consider the space and time requirements for such operations as random access, add, multiply, and transpose. Note that we can improve the random access time by keeping another array, *ra*, such that *ra*[i] = number of nonzero terms in rows 0 through $i - 1$.

2.6 REPRESENTATION OF MULTIDIMENSIONAL ARRAYS

In C, multidimensional arrays are represented using the array-of-arrays representation (Section 2.2.2). An alternative to the array-of-arrays representation is to map all elements of a multidimensional array into an ordered or linear list. The linear list is then stored in consecutuve memory just as we store a one-dimensional array. This mapping of a multidimensional array to memory requires a more complex addressing formula that required by the mapping of a one-dimensional array to memory. If an array is declared $a[upper_0][upper_1] \cdots [upper_{n-1}]$, then it is easy to see that the number of elements in the array is:

$$\prod_{i=0}^{n-1} upper_i$$

where Π is the product of the $upper_i$'s. For instance, if we declare a as $a[10][10][10]$, then we require $10 \cdot 10 \cdot 10 = 1000$ units of storage to hold the array. There are two common ways to represent multidimensional arrays: *row major order* and *column major order*. We consider only row major order here, leaving column major order for the exercises.

As its name implies, row major order stores multidimensional arrays by rows. For instance, we interpret the two-dimensional array $A[upper_0][upper_1]$ as $upper_0$ rows, $row_0, row_1, \cdots, row_{upper_0-1}$, each row containing $upper_1$ elements.

If we assume that α is the address of $A[0][0]$, then the address of $A[i][0]$ is $\alpha + i \cdot upper_1$ because there are i rows, each of size $upper_1$, preceding the first element in the ith row. Notice that we haven't multiplied by the element size. This follows C convention in which the size of the elements is automatically accounted for. The address of an arbitrary element, $a[i][j]$, is $\alpha + i \cdot upper_1 + j$.

To represent a three-dimensional array, $A[upper_0][upper_1][upper_2]$, we interpret the array as $upper_0$ two-dimensional arrays of dimension $upper_1 \times upper_2$. To locate $a[i][j][k]$, we first obtain $\alpha + i \cdot upper1 \cdot upper2$ as the address of $a[i][0][0]$ because there are i two-dimensional arrays of size $upper_1 \cdot upper_2$ preceding this element. Combining this formula with the formula for addressing a two-dimensional array, we obtain:

$$\alpha + i \cdot upper_1 \cdot upper_2 + j \cdot upper_2 + k$$

as the address of $a[i][j][k]$.

Generalizing on the preceding discussion, we can obtain the addressing formula for any element $A[i_0][i_1] \ldots [i_{n-1}]$ in an n-dimensional array declared as:

$$A[upper_0][upper_1] \ldots [upper_{n-1}]$$

If α is the address for $A[0][0] \ldots [0]$ then the address of $a[i_0][0][0] \ldots [0]$ is:

$$\alpha + i_0\ upper_1\ upper_2 \ldots upper\ n-1$$

The address of $a[i_0][i_1][0]\ldots[0]$ is:

$$\alpha + i_0\ upper_1\ upper_2 \ldots upper_{n-1} + i_1\ upper_2\ upper_3 \ldots upper_{n-1}$$

Repeating in this way the address for $A[i_0][i_1]\ldots[i_{n-1}]$ is:

$$\begin{aligned}
&\alpha + i_0 upper_1 upper_2 \ldots upper_{n-1} \\
&\quad + i_1 upper_2 upper_3 \ldots upper_{n-1} \\
&\quad + i_2 upper_3 upper_4 \ldots upper_{n-1} \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&\quad + i_{n-2} upper_{n-1} \\
&\quad + i_{n-1}
\end{aligned}$$

$$= \alpha + \sum_{j=0}^{n-1} i_j a_j \text{ where: } \begin{cases} a_j = \prod_{k=j+1}^{n-1} upper_k & 0 \le j < n-1 \\ a_{n-1} = 1 \end{cases}$$

Notice that a_j may be computed from a_{j+1}, $0 \le j < n-1$, using only one multiplication as $a_j = upper_{j+1} \cdot a_{j+1}$. Thus, a compiler will initially take the declared bounds $upper_0, \ldots, upper_{n-1}$ and use them to compute the constants $a_0 \ldots a_{n-2}$ using $n-2$ multiplications. The address of $a[i_0]\ldots a[i_{n-1}]$ can be computed using the formula, requiring $n-1$ more multiplications and n additions and n subtractions.

EXERCISES

1. Assume that we have a one-dimensional array, *a*[*MAX_SIZE*]. Normally, the subscripts for this array vary from 0 to *MAX_SIZE* − 1. However, by using pointer arithmetic we can create arrays with arbitrary bounds. Indicate how to create an array, and obtain subscripts for an array, that has bounds between −10 to 10. That is, we view the subscripts as having the values −10, −9, −8, ⋯ , 8, 9, 10.
2. Extend the results from Exercise 1 to create a two-dimensional array where row and column subscripts each range from −10 to 10.
3. Obtain an addressing formula for the element $a[i_0][i_1]\ldots[i_{n-1}]$ in an array declared as $a[upper_0]\ldots a[upper_{n-1}]$. Assume a column major representation of the array with one word per element and α the address of $a[0][0]\ldots[0]$. In column major order, the entries are stored by columns first. For example, the array $a[3][3]$ would be stored as $a[0][0]$, $a[1][0]$, $a[2][0]$, $a[0][1]$, $a[1][1]$, $a[2][1]$, $a[0][2]$, $a[1][2]$, $a[2][2]$.

2.7 STRINGS

2.7.1 The Abstract Data Type

Thus far, we have considered only ADTs whose component elements were numeric. For example, we created a sparse matrix ADT and represented it as an array of triples *<row, col, value>*. In this section, we turn our attention to a data type, the string, whose component elements are characters. As an ADT, we define a string to have the form, $S = s_0, \ldots, s_{n-1}$, where s_i are characters taken from the character set of the programming language. If $n = 0$, then S is an empty or null string.

There are several useful operations we could specify for strings. Some of these operations are similar to those required for other ADTs: creating a new empty string, reading a string or printing it out, appending two strings together (called *concatenation*), or copying a string. However, there are other operations that are unique to our new ADT, including comparing strings, inserting a substring into a string, removing a substring from a string, or finding a pattern in a string. We have listed the essential operations in ADT 2.4, which contains our specification of the string ADT. Actually there are many more operations on strings, as we shall see when we look at part of C's string library in Figure 2.8.

2.7.2 Strings in C

In C, we represent strings as character arrays terminated with the null character \0. For instance, suppose we had the strings:

```
#define MAX_SIZE 100 /*maximum size of string */
char s[MAX_SIZE] = {"dog"};
char t[MAX_SIZE] = {"house"};
```

Figure 2.9 shows how these strings would be represented internally in memory. Notice that we have included array bounds for the two strings. Technically, we could have declared the arrays with the statements:

```
char s[] = {"dog"};
char t[] = {"house"}
```

Using these declarations, the C compiler would have allocated just enough space to hold each word including the null character. Now suppose we want to concatenate these strings together to produce the new string, "doghouse." To do this we use the C function *strcat* (See Figure 2.8). Two strings are joined together by *strcat* (*s*, *t*), which stores the result in *s*. Although *s* has increased in length by five, we have no additional space in *s* to store the extra five characters. Our compiler handled this problem inelegantly: it

ADT *String* is
objects: a finite set of zero or more characters.
functions:
for all $s, t \in$ *String*, $i, j, m \in$ non-negative integers

String Null(m)	::=	**return** a string whose maximum length is m characters, but is initially set to *NULL* We write *NULL* as "".
Integer Compare(s, t)	::=	**if** s equals t **return** 0 **else if** s precedes t **return** −1 **else return** +1
Boolean IsNull(s)	::=	**if** (Compare(s, *NULL*)) **return** *FALSE* **else return** *TRUE*
Integer Length(s)	::=	**if** (Compare(s, *NULL*)) **return** the number of characters in s **else return** 0.
String Concat(s, t)	::=	**if** (Compare(t, *NULL*)) **return** a string whose elements are those of s followed by those of t **else return** s.
String Substr(s, i, j)	::=	**if** ($(j > 0)$ && $(i+j-1) <$ Length(s)) **return** the string containing the characters of s at positions $i, i+1, \cdots, i+j-1$. **else return** *NULL*.

ADT 2.4: Abstract data type *String*

simply overwrote the memory to fit in the extra five characters. Since we declared t immediately after s, this meant that part of the word "house" disappeared.

We have already seen that C provides a built-in function to perform concatenation. In addition to this function, C provides several other string functions which we access through the statement *#include <string.h>*. Figure 2.8 contains a brief summary of these functions (we have excluded string conversion functions such as *atoi*). For each function, we have provided a generic function declaration and a brief description. Rather than discussing each function separately, we next look at an example that uses several of them.

Example 2.2 [***String insertion***]: Assume that we have two strings, say *string* 1 and *string* 2, and that we want to insert *string* 2 into *string* 1 starting at the ith position of *string* 1. We begin with the declarations:

Function	Description
*char *strcat(char *dest, char *src)*	concatenate *dest* and *src* strings; return result in *dest*
*char *strncat(char *dest, char *src, int n)*	concatenate *dest* and *n* characters from *src*; return result in *dest*
*char *strcmp(char *str1, char *str2)*	compare two strings; return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 0 if *str1* > *str2*
*char *strncmp(char *str1, char *str2, int n)*	compare first *n* characters return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 1 if *str1* > *str2*
*char *strcpy(char *dest, char *src)*	copy *src* into *dest*; return *dest*
*char *strncpy(char *dest, char *src, int n)*	copy *n* characters from *src* string into *dest*; return *dest*;
*size_t strlen(char *s)*	return the length of a *s*
*char *strchr(char *s, int c)*	return pointer to the first occurrence of *c* in *s*; return *NULL* if not present
*char *strrchr(char *s, int c)*	return pointer to last occurrence of *c* in *s*; return *NULL* if not present
*char *strtok(char *s, char *delimiters)*	return a token from *s*; token is surrounded by *delimiters*
*char *strstr(char *s, char *pat)*	return pointer to start of *pat* in *s*
*size_t strspn(char *s, char *spanset)*	scan *s* for characters in *spanset*; return length of span
*size_t strcspn(char *s, char *spanset)*	scan *s* for characters not in *spanset*; return length of span
*char *strpbrk(char *s, char *spanset)*	scan *s* for characters in *spanset*; return pointer to first occurrence of a character from *spanset*

Figure 2.8: C string functions

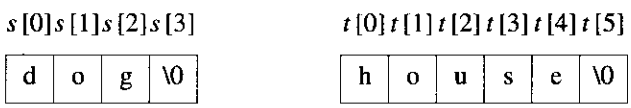


Figure 2.9: String representation in C

```
#include <string.h>
#define MAX_SIZE 100 /*size of largest string*/
char string1[MAX_SIZE], *s = string1;
char string2[MAX_SIZE],  *t = string2;
```

In addition to creating the two strings, we also have created a pointer for each string.

Now suppose that the first string contains "amobile" and the second contains "uto" (Figure 2.10). We want to insert "uto" starting at position 1 of the first string, thereby producing the word "automobile." We can accomplish this using only three function calls, as Figure 2.10 illustrates. Thus, in Figure 2.10(a), we assume that we have an empty string that is pointed to by *temp*. We use *strncpy* to copy the first i characters from s into *temp*. Since $i = 1$, this produces the string "a." In Figure 2.10(b), we concatenate *temp* and t to produce the string "auto." Finally, we append the remainder of s to *temp*. Since *strncat* copied the first i characters, the remainder of the string is at address $(s + i)$. The final result is shown in Figure 2.10(c).

Program 2.12 inserts one string into another. This particular function is not normally found in *<string.h>*. Since either of the strings could be empty, we also include statements that check for these conditions. It is worth pointing out that the call *strnins*$(s, t, 0)$ is equivalent to *strcat*(t, s). Program 2.12 is presented as an example of manipulating strings. It should never be used in practice as it is wasteful in its use of time and space. Try to revise it so the string *temp* is not required. □

2.7.3 Pattern Matching

Now let us develop an algorithm for a more sophisticated application of strings. Assume that we have two strings, *string* and *pat*, where *pat* is a pattern to be searched for in *string*. The easiest way to determine if *pat* is in *string* is to use the built-in function *strstr*. If we have the following declarations:

```
char pat[MAX_SIZE], string[MAX_SIZE], *t;
```

then we use the following statements to determine if *pat* is in *string*:

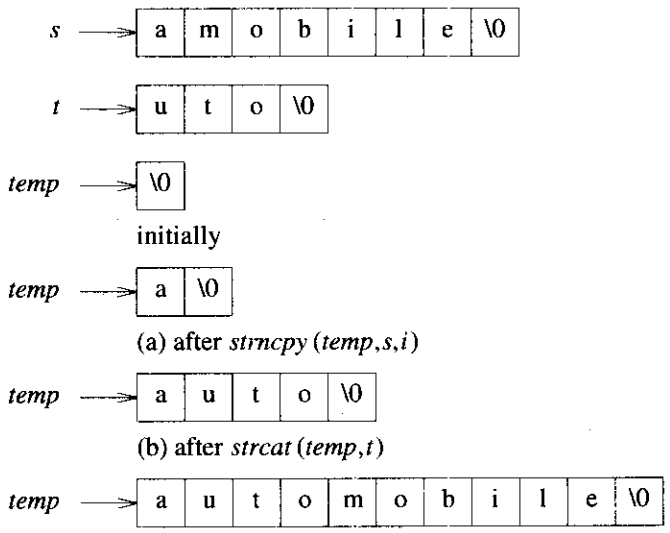


Figure 2.10: String insertion example

```
if (t = strstr(string,pat))
   printf("The string from strstr is: %s\n",t);
else
   printf("The pattern was not found with strstr\n");
```

The call (*t* = *strstr* (*string*,*pat*)) returns a null pointer if *pat* is not in *string*. If *pat* is in *string*, *t* holds a pointer to the start of *pat* in *string*. The entire string beginning at position *t* is printed out.

Although *strstr* seems ideally suited to pattern matching, we may want to develop our own pattern matching function because there are several different methods for implementing a pattern matching function. The easiest but least efficient method sequentially examines each character of the string until it finds the pattern or it reaches the end of the string. (We explore this approach in the Exercises.) If *pat* is not in *string*, this method has a computing time of $O(n \cdot m)$ where n is the length of *pat* and m is the length of *string*. We can do much better than this, if we create our own pattern matching function.

We can improve on an exhaustive pattern matching technique by quitting when

```
void strnins(char *s, char *t, int i)
{/* insert string t into string s at position i */
   char string[MAX_SIZE], *temp = string;

   if (i < 0 && i > strlen(s)) {
      fprintf(stderr,"Position is out of bounds \n");
      exit(EXIT_FAILURE);
   }
   if (!strlen(s))
      strcpy(s,t);
   else if (strlen(t)) {
      strncpy(temp, s,i);
      strcat(temp,t);
      strcat(temp, (s+i));
      strcpy(s, temp);
   }
}
```

Program 2.12: String insertion function

strlen (*pat*) is greater than the number of remaining characters in the string. Checking the first and last characters of *pat* and *string* before we check the remaining characters is a second improvement. These changes are incorporated in *nfind* (Program 2.13).

Example 2.3 [***Simulation of nfind***]: Suppose *pat* = "aab" and *string* = "ababbaabaa." Figure 2.11 shows how *nfind* compares the characters from *pat* with those of *string*. The end of the *string* and *pat* arrays are held by *lasts* and *lastp*, respectively. First *nfind* compares *string* [*endmatch*] and *pat* [*lastp*]. If they match, *nfind* uses i and j to move through the two strings until a mismatch occurs or until all of *pat* has been matched. The variable *start* is used to reset i if a mismatch occurs. □

Analysis of *nfind*: If we apply *nfind* to *string* = "aa $\cdots$ a" and *pat* = "a $\cdots$ ab", then the computing time for these strings is linear in the length of the string $O(m)$, which is certainly far better than the sequential method. Although the improvements we made over the sequential method speed up processing on the average, the worst case computing time is still $O(n \cdot m)$. □

Ideally, we would like an algorithm that works in $O(strlen(string) + strlen(pat))$ time. This is optimal for this problem as in the worst case it is necessary to look at all characters in the pattern and string at least once. We want to search the string for the pattern without moving backwards in the string. That is, if a mismatch occurs we want

```
int nfind(char *string, char *pat)
{/* match the last character of pattern first, and
    then match from the beginning */
   int i,j,start = 0;
   int lasts = strlen(string)-1;
   int lastp = strlen(pat)-1;
   int endmatch = lastp;

   for (i = 0; endmatch <= lasts; endmatch++, start++) {
      if (string[endmatch] == pat[lastp])
         for (j = 0, i = start; j < lastp &&
                     string[i] == pat[j]; i++,j++)
            ;
      if (j == lastp)
         return start; /* successful */
      }
      return -1;
}
```

Program 2.13: Pattern matching by checking end indices first

to use our knowledge of the characters in the pattern and the position in the pattern where the mismatch occurred to determine where we should continue the search. Knuth, Morris, and Pratt have developed a pattern matching algorithm that works in this way and has linear complexity. Using their example, suppose

$$pat = \text{'}a\ b\ c\ a\ b\ c\ a\ c\ a\ b\text{'}$$

Let $s = s_0\, s_2 \cdots s_{m-1}$ be the string and assume that we are currently determining whether or not there is a match beginning at s_i. If $s_i \neq a$ then, clearly, we may proceed by comparing s_{i+1} and a. Similarly if $s_i = a$ and $s_{i+1} \neq b$ then we may proceed by comparing s_{i+1} and a. If $s_i s_{i+1} = ab$ and $s_{i+2} \neq c$ then we have the situation:

$s =$	'-	a	b	?	?	?	.	.	.	.	?'
$pat =$		'a	b	c	a	b	c	a	c	a	b'

The ? implies that we do not know what the character in s is. The first ? in s represents s_{i+2} and $s_{i+2} \neq c$. At this point we know that we may continue the search for a match by comparing the first character in *pat* with s_{i+2}. There is no need to compare this character

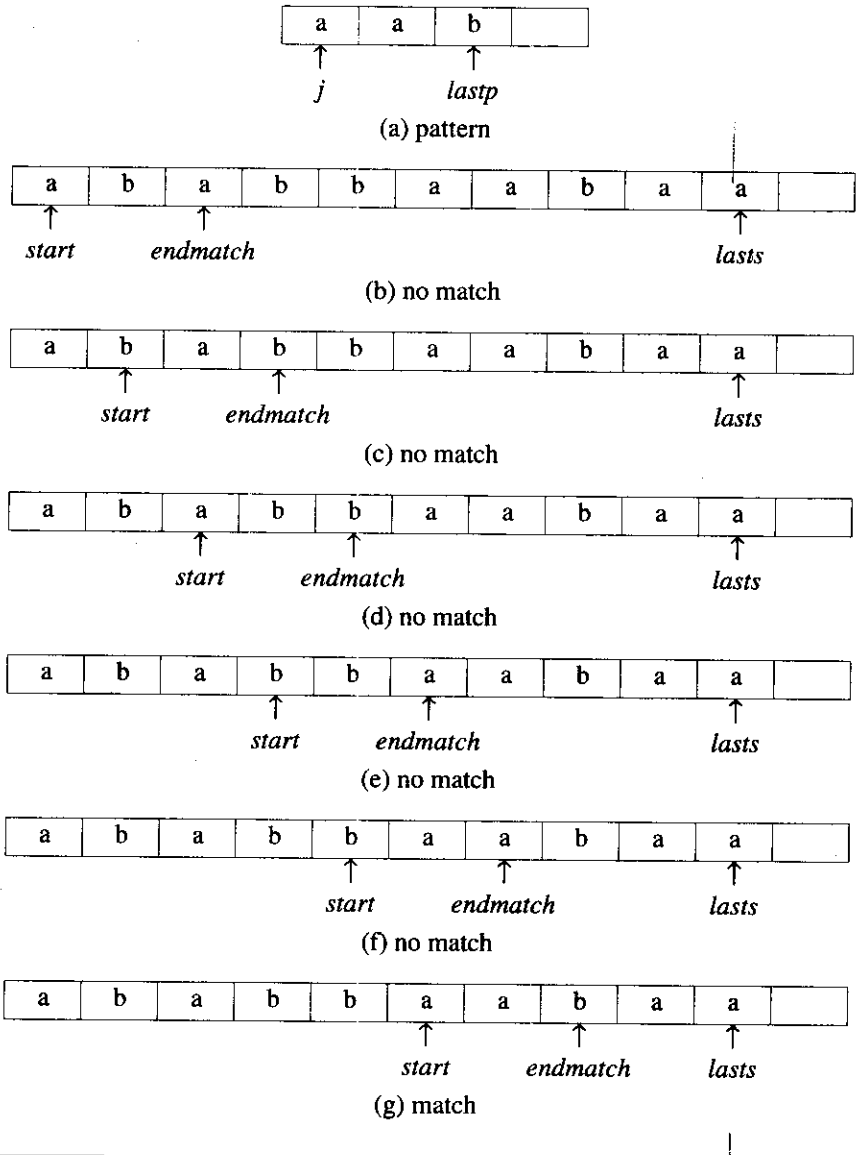


Figure 2.11: Simulation of *nfind*

of *pat* with s_{i+1} as we already know that s_{i+1} is the same as the second character of *pat*, *b*, and so $s_{i+1} \neq a$. Let us try this again assuming a match of the first four characters in *pat* followed by a nonmatch, i.e., $s_{i+4} \neq b$. We now have the situation:

s =	'-	a	b	c	a	?	?	.	.	.	?'
pat =		'a	b	c	a	b	c	a	c	a	b'

We observe that the search for a match can proceed by comparing s_{i+4} and the second character in *pat*, *b*. This is the first place a partial match can occur by sliding the pattern *pat* towards the right. Thus, by knowing the characters in the patterm and the position in the pattern where a mismatch occurs with a character in *s* we can determine where in the pattern to continue the search for a match without moving backwards in *s*. To formalize this, we define a failure function for a pattern.

Definition: If $p = p_0 p_1 \cdots p_{n-1}$ is a pattern, then its *failure function, f*, is defined as:

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0 p_1 \cdots p_i = p_{j-i} p_{j-i+2} \cdots p_j \text{ if such an } i \geq 0 \text{ exists} \\ -1 \qquad \text{otherwise} \end{cases} \quad \square$$

For the example pattern, *pat* = *abcabcacab*, we have:

j	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
f	−1	−1	−1	0	1	2	3	−1	0	1

From the definition of the failure function, we arrive at the following rule for pattern matching: *If a partial match is found such that* $s_{i-j} \cdots s_{i-1} = p_0 p_1 \cdots p_{j-1}$ *and* $s_i \neq p_j$ *then matching may be resumed by comparing* s_i *and* $p_{f(j-1)+1}$ *if* $j \neq 0$. *If* $j = 0$, *then we may continue by comparing* s_{i+1} *and* p_0. This pattern matching rule translates into function *pmatch* (Program 2.14). The following declarations are assumed:

```
#include <stdio.h>
#include <string.h>
#define max_string_size 100
#define max_pattern_size 100
int pmatch();
void fail();
int failure[max_pattern_size];
char string[max_string_size];
char pat[max_pattern_size];
```

```
int pmatch(char *string, char *pat)
{/* Knuth, Morris, Pratt string matching algorithm */
    int i = 0, j = 0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    while ( i < lens && j < lenp ) {
        if (string[i] == pat[j]) {
            i++; j++; }
        else if (j == 0) i++;
            else j = failure[j-1]+1;
    }
    return ( (j == lenp) ? (i-lenp) : -1);
}
```

Program 2.14: Knuth, Morris, Pratt pattern matching algorithm

Note that we do not keep a pointer to the start of the pattern in the string. Instead we use the statement:

```
return ( (j == lenp) ? (i - lenp) : -1);
```

This statement checks to see whether or not we found the pattern. If we didn't find the pattern, the pattern index index j is not equal to the length of the pattern and we return -1. If we found the pattern, then the starting position is i – the length of the pattern.

Analysis of *pmatch*: The **while** loop is iterated until the end of either the string or the pattern is reached. Since i is never decreased, the lines that increase i cannot be executed more than $m = strlen(string)$ times. The resetting of j to $failure[j-1]+1$ decreases the value of j. So, this cannot be done more times than j is incremented by the statement $j++$ as otherwise, j falls off the pattern. Each time the statement $j++$ is executed, i is also incremented. So, j cannot be incremented more than m times. Consequently, no statement of Program 2.14 is executed more than m times. Hence the complexity of function *pmatch* is $O(m) = O(strlen(string))$. □

From the analysis of *pmatch*, it follows that if we can compute the failure function in $O(strlen(pat))$ time, then the entire pattern matching process will have a computing time proportional to the sum of the lengths of the string and pattern. Fortunately, there is a fast way to compute the failure function. This is based upon the following restatement of the failure function:

$$f(j) = \begin{cases} -1 & \text{if } j = 0 \\ f^m(j-1)+1 & \text{where } m \text{ is the least integer } k \text{ for which } p_{f^k(j-1)+1} = p_j \\ -1 & \text{if there is no } k \text{ satisfying the above} \end{cases}$$

(note that $f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j))$).

This definition yields the function in Program 2.15 for computing the failure function of a pattern.

```
void fail(char *pat)
{/* compute the pattern's failure function */
    int n = strlen(pat);
    failure[0] = -1;
    for (j=1; j < n; j++) {
        i = failure[j-1];
        while ((pat[j] != pat[i+1]) && (i >= 0))
            i = failure[i];
        if (pat[j] == pat[i+1])
            failure[j] = i+1;
        else failure[j] = -1;
    }
}
```

Program 2.15: Computing the failure function

Analysis of *fail*: In each iteration of the **while** loop the value of i decreases (by the definition of f). The variable i is reset at the beginning of each iteration of the **for** loop. However, it is either reset to -1 (initially or when the previous iteration of the **for** loop goes through the last **else** clause) or it is reset to a value 1 greater than its terminal value on the previous iteration (i.e., when the statement *failure*$[j] = i+1$ is executed). Since the **for** loop is iterated only $n-1$ (n is the length of the pattern) times, the value of i has a total increment of at most $n-1$. Hence it cannot be decremented more than $n-1$ times. Consequently the **while** loop is iterated at most $n-1$ times over the whole algorithm and the computing time of *fail* is $O(n) = O(strlen(pat))$. □

Note that when the failure function is not known in advance, the time to first compute this function and then perform a pettern match is $O(strlen(pat) + strlen(string))$.

EXERCISES

1. Write a function that accepts as input a *string* and determines the frequency of occurrence of each of the distinct characters in *string*. Test your function using suitable data.
2. Write a function, *strndel*, that accepts a *string* and two integers, *start* and *length*. Return a new string that is equivalent to the original string, except that *length* characters beginning at *start* have been removed.
3. Write a function, *strdel*, that accepts a *string* and a *character*. The function returns *string* with the first occurrence of *character* removed.
4. Write a function, *strpos* 1, that accepts a *string* and a *character*. The function returns an integer that represents the position of the first occurrence of *character* in *string*. If *character* is not in *string*, it returns -1. You may not use the function *strpos* which is part of the traditional <*string.h*> library, but not the ANSI C one.
5. Write a function, *strchr* 1, that does the same thing as *strpos* 1 except that it returns a pointer to *character*. If *character* is not in the list it returns *NULL*. You may not use the built-in function *strchr*.
6. Modify Program 2.12 so that it does not use a temporary string *temp*. Compare the complexity of your new function with that of the old one.
7. Write a function, *strsearch*, that uses the sequential method for pattern matching. That is, assuming we have a *string* and a *pattern*, *strsearch* examines each character in *string* until it either finds the *pattern* or it reaches the end of the *string*.
8. Show that the computing time for *nfind* is $O(n \cdot m)$ where n and m are, respectively, the lengths of the string and the pattern. Find a string and a pattern for which this is true.
9. Compute the failure function for each of the following patterns:
   (a) *a a a a b*
   (b) *a b a b a a*
   (c) *a b a a b a a b*
10. Show the equivalence of the two definitions for the failure function.

2.8 REFERENCES AND SELECTED READINGS

The Knuth, Morris, Pratt pattern-matching algorithm can be found in "Fast pattern matching in strings," *SIAM Journal on Computing*, 6:2, 1977, pp. 323-350. A discussion of the Knuth Morris Pratt algorithm, along with other string matching algorithms, may be found in *Introduction to Algorithms* Second Edition, by T. Cormen, C. Leiserson, R. Rivest and C. Stein, McGraw Hill, New York, 2002.

2.9 ADDITIONAL EXERCISES

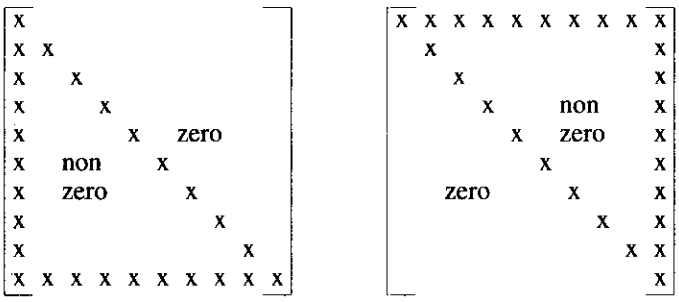
1. Given an array $a[n]$ produce the array $z[n]$ such that $z[0] = a[n-1], z[1] = a[n-2], \cdots, z[n-2] = a[1], z[n-1] = a[0]$. Use a minimal amount of storage.
2. An $m \times n$ matrix is said to have a saddle point if some entry $a[i][j]$ is the smallest value in row i and the largest value in column j. Write a C function that determines the location of a saddle point if one exists. What is the computing time of your method?

Exercises 3 through 8 explore the representation of various types of matrices that are frequently used in the solution of problems in the natural sciences.

3. A *triangular matrix* is one in which either all the elements above the main diagonal or all the elements below the main diagonal of a square matrix are zero. Figure 2.12 shows a lower and an upper triangular matrix. In a lower triangular matrix, a, with n rows, the maximum number of nonzero terms in row i is $i+1$. Thus, the total number of nonzero terms is

$$d = \sum_{i=0}^{n-1} (i+1) = n(n+1)/2.$$

Since storing a triangular matrix as a two dimensional array wastes space, we would like to find a way to store only the nonzero terms in the triangular matrix. Find an addressing formula for the elements a_{ij} so that they can be stored by rows in an array $b[n(n+1)/2-1]$, with $a[0][0]$ being stored in $b[0]$.



lower triangular upper triangular

Figure 2.12: Lower and upper triangular matrices

4. Let a and b be two lower triangular matrices, each with n rows. The total number of elements in the lower triangles is $n(n+1)$. Devise a scheme to represent both triangles in an array $d[n-1][n]$. [Hint: Represent the triangle of a in the lower triangle of d and the transpose b in the upper triangle of d.] Write algorithms to determine the values of $a[i][j]$, $b[i][j]$, $0 \leq i, j < n$.

5. A *tridiagonal matrix* is a square matrix in which all elements that are not on the major diagonal and the two diagonals adjacent to it are zero (Figure 2.13). The elements in the band formed by these three diagonals are represented by rows in an array, b, with $a[0][0]$ being stored in $b[0]$. Obtain an algorithm to determine the value of $a[i][j]$, $0 \leq i, j < n$ from the array b.

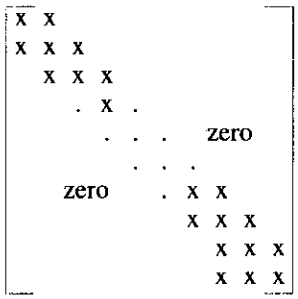


Figure 2.13: Tridiagonal matrix

6. A *square band matrix* $D_{n,a}$ is an $n \times n$ matrix in which all the nonzero terms lie in a band centered around the main diagonal. The band includes the main diagonal and $a-1$ diagonals below and above the main diagonal (Figure 2.14).
   (a) How many elements are there in the band $D_{n,a}$?
   (b) What is the relationship between i and j for elements $d_{i,j}$ in the band $D_{n,a}$?
   (c) Assume that the band of $D_{n,a}$ is stored sequentially in an array b by diagonals, starting with the lowermost diagonal. For example, the band matrix, $D_{4,3}$ of Figure 2.14 would have the following representation.

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]	b[10]	b[11]	b[12]	b[13]
9	7	8	3	6	6	0	2	8	7	4	9	8	4

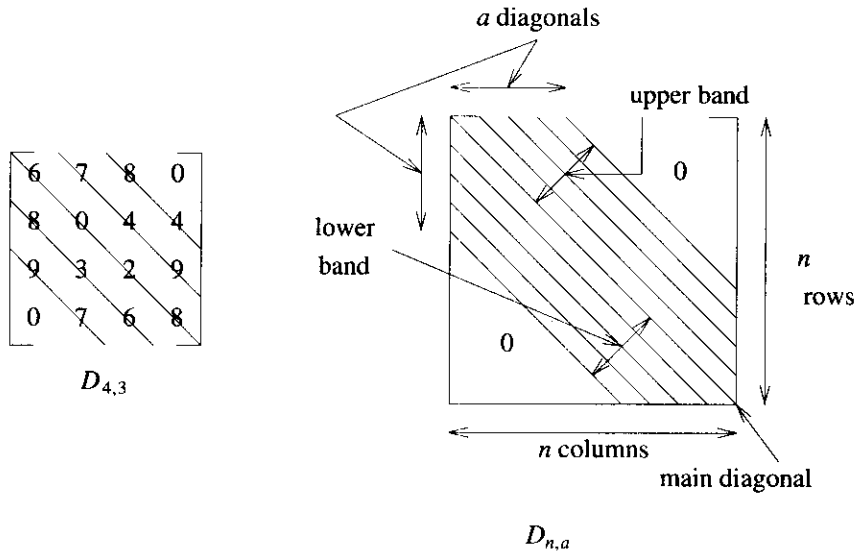
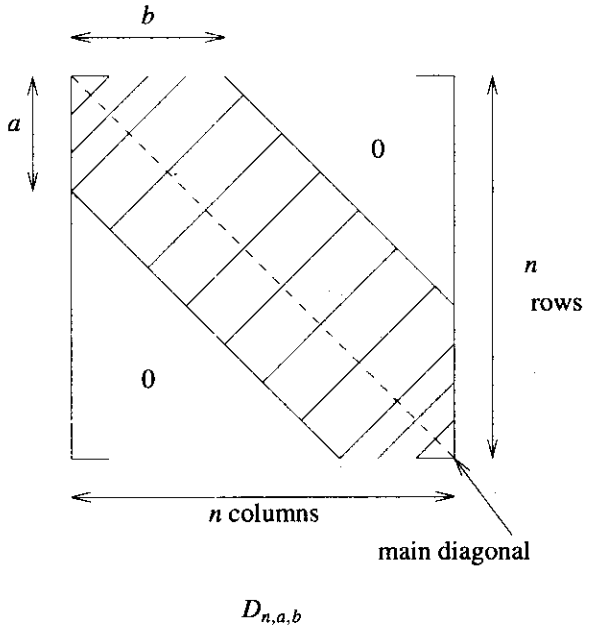


Figure 2.14: Square band matrix

d_{20} d_{31} d_{10} d_{21} d_{32} d_{00} d_{11} d_{22} d_{33} d_{01} d_{12} d_{23} d_{02} d_{13}

Obtain an addressing formula for the location of an element, $d_{i,j}$, in the lower band of $D_{n,a}$ (location(d_{10}) = 2 in the example above).

7. A *generalized band matrix* $D_{n,a,b}$ is an $n \times n$ matrix in which all the nonzero terms lie in a band made up of $a-1$ diagonals below the main diagonal, the main diagonal, and $b-1$ bands above the main diagonal (Figure 2.15).
   (a) How many elements are there in the band of $D_{n,a,b}$?
   (b) What is the relationship between i and j for the elements d_{ij} in the band of $D_{n,a,b}$?
   (c) Obtain a sequential representation of the band $D_{n,a,b}$ in the one dimensional array e. For this representation, write a C function *value* (n, a, b, i, j, e) that determines the value of element d_{ij} in the matrix $D_{n,a,b}$. The band of $D_{n,a,b}$ is represented in the array e.
8. A complex-valued matrix X is represented by a pair of matrices $<a, b>$, where a and b contain real values. Write a function that computes the product of two complex-valued matrices $<a, b>$ and $<d, e>$, where $<a, b> * <d, e> = (a + ib)$



$D_{n,a,b}$

Figure 2.15: Generalized band matrix

$*\ (d + ie) = (ad - be) + i\,(ae + bd)$. Determine the number of additions and multiplications if the matrices are all $n \times n$.

9. § [***Programming project***] There are a number of problems, known collectively as "random walk" problems, that have been of longstanding interest to the mathematical community. All but the most simple of these are extremely difficult to solve, and, for the most part, they remain largely unsolved. One such problem may be stated as:

   A (drunken) cockroach is placed on a given square in the middle of a tile floor in a rectangular room of size $n \times m$ tiles. The bug wanders (possibly in search of an aspirin) randomly from tile to tile throughout the room. Assuming that he may move from his present tile to any of the eight tiles surrounding him (unless he is against a wall) with equal probability, how long will it take him to touch every tile on the floor at least once?

   Hard as this problem may be to solve by pure probability techniques, it is quite

easy to solve using a computer. The technique for doing so is called "simulation." This technique is widely used in industry to predict traffic flow, inventory control, and so forth. The problem may be simulated using the following method:

An $n \times m$ array *count* is used to represent the number of times our cockroach has reached each tile on the floor. All the cells of this array are initialized to zero. The position of the bug on the floor is represented by the coordinates (*ibug*, *jbug*). The eight possible moves of the bug are represented by the tiles located at (*ibug* + *imove* [*k*], *jbug* + *jmove* [*k*]), where $0 \le k \le 7$, and

imove[0] = −1	*jmove*[0] = 1
imove[1] = 0	*jmove*[1] = 1
imove[2] = 1	*jmove*[2] = 1
imove[3] = 1	*jmove*[3] = 0
imove[4] = 1	*jmove*[4] = −1
imove[5] = 0	*jmove*[5] = −1
imove[6] = −1	*jmove*[6] = −1
imove[7] = −1	*jmove*[7] = 0

A random walk to any one of the eight neighbor squares is simulated by generating a random value for *k*, lying between 0 and 7. Of course, the bug cannot move outside the room, so that coordinates that lead up a wall must be ignored, and a new random combination formed. Each time a square is entered, the count for that square is incremented so that a nonzero entry shows the number of times the bug has landed on that square. When every square has been entered at least once, the experiment is complete.

Write a program to perform the specified simulation experiment. Your program MUST:

(a) handle all values of n and m, $2 < n \le 40$, $2 \le m \le 20$;

(b) perform the experiment for (1) $n = 15$, $m = 15$, starting point (10, 10), and (2) $n = 39$, $m = 19$, starting point (1, 1);

(c) have an iteration limit, that is, a maximum number of squares that the bug may enter during the experiment. This ensures that your program will terminate. A maximum of 50,000 is appropriate for this exercise.

For each experiment, print (1) the total number of legal moves that the cockroach makes and (2) the final count array. This will show the "density" of the walk, that is, the number of times each tile on the floor was touched during the experiment. This exercise was contributed by Olson.

10. § [***Programming project***] Chess provides the setting for many fascinating diversions that are quite independent of the game itself. Many of these are based on the strange "L-shaped" move of the knight. A classic example is the problem of the "knight's tour," which has captured the attention of mathematicians and puzzle enthusiasts since the beginning of the eighteenth century. Briefly stated, the problem requires us to move the knight, beginning from any given square on the chessboard, successively to all 64 squares, touching each square once and only once. Usually we represent a solution by placing the numbers 0, 1, $\cdots$, 63 in the squares of the chess board to indicate the order in which the squares are reached. One of the more ingenious methods for solving the problem of the knight's tour was given by J. C. Warnsdorff in 1823. His rule stated that the knight must always move to one of the squares from which there are the fewest exits to squares not already traversed.

The goal of this programming project is to implement Warnsdorff's rule. The ensuing discussion will be easier to follow, however, if you try to construct a solution to the problem by hand, before reading any further.

The crucial decision in solving this problem concerns the data representation. Figure 2.16 shows the chess board represented as a two-dimensional array.

The eight possible moves of a knight on square (4, 2) are also shown in this figure. In general, a knight may move to one of the squares $(i-2, j+1)$, $(i-1, j+2)$, $(i+1, j+2)$, $(i+2, j+1)$, $(i+2, j-1)$, $(i+1, j-2)$, $(i-1, j-2)$, $(i-2, j-1)$. However, notice that if (i, j) is located near one of the board's edges, some of these possibilities could move the knight off the board, and, of course, this is not permitted. We can represent easily the eight possible knight moves by two arrays *ktmove* 1 and *ktmove* 2 as:

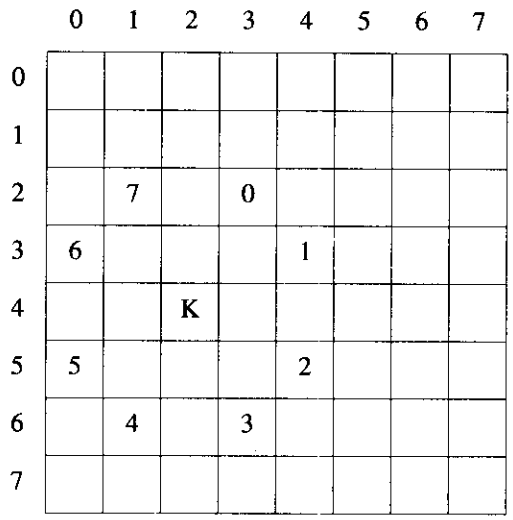


Figure 2.16: Legal moves for a knight

ktmove 1	*ktmove* 2
−2	1
−1	2
1	2
2	1
2	−1
1	−2
−1	−2
−2	−1

Then a knight at (i, j) may move to $(i + ktmove[k], j + ktmove2[k])$, where k is some value between 0 and 7, provided that the new square lies on the chess board. Below is a description of an algorithm for solving the knight's tour problem using Warnsdorff's rule. The data representation discussed in the previous section is assumed.

(a) [***Initialize chessboard***] For $0 \leq i, j \leq 7$ set $board[i][j]$ to 0.

(b) [***Set starting position***] Read and print (i, j) and then set $board[i][j]$ to 0.

(c) [***Loop***] For $1 \le m \le 63$, do steps (d) through (g).

(d) [***Form a set of possible next squares***] Test each of the eight squares one knight's move away from (i, j) and form a list of the possibilities for the next square $(nexti[l], nextj[l])$. Let *npos* be the number of possibilities. (That is, after performing this step we have $nexti[l] = i + ktmove\ 1[k]$ and $nextj[l] = j + ktmove\ 2[k]$, for certain values of k between 0 and 7. Some of the squares $(i + ktmove\ 1[k], j + ktmove\ 2[k])$ may be impossible because they lie off the chessboard or because they have been occupied previously by the knight, that is, they contain a nonzero number. In every case we will have $0 \le npos \le 8$.)

(e) [***Test special cases***] If $npos = 0$, the knight's tour has come to a premature end; report failure and go to step (h). If $npos = 1$, there is only one next move; set *min* to 1 and go to step (g).

(f) [***Find next square with minimum number of exits***] For $1 \le l \le npos$, set $exits[l]$ to the number of exits from square $(nexti[l], nextj[l])$. That is, for each of the values of l, examine each of the next squares $(nexti[l] + ktmove\ 1[k], nextj[l] + ktmove\ 2[k])$ to see if it is an exit from $(nexti[l], nextj[l])$, and count the number of such exits in $exits[l]$. (Recall that a square is an exit if it lies on the chessboard and has not been occupied previously by the knight.) Finally, set *min* to the location of the minimum value of exits. (If there is more than one occurrence of the minimum value, let *min* denote the first such occurrence. Although this does not guarantee a solution, the chances of completing the tour are very good.)

(g) [***Move knight***] Set $i = nexti[min]$, $j = nextj[min]$, and $board[i][j] = m$. Thus, (i, j) denotes the new position of the knight, and $board[i][j]$ records the move in proper sequence.

(h) [***Print***] Print out the board showing the solution to the knight's tour, and then terminate the algorithm.

Write a C program that corresponds to the algorithm. This exercise was contributed by Legenhausen and Rebman.

CHAPTER 3

STACKS AND QUEUES

3.1 STACKS

In this chapter we look at two data types that are frequently found in computer science. These data types, the stack and the queue, are special cases of the more general data type, *ordered list*, that we discussed in Chapter 2. Recall that $A = a_0, a_1, \cdots, a_{n-1}$ is an ordered list of $n \geq 0$ elements. We refer to the a_i as *atoms* or *elements* that are taken from some set. The null or empty list, denoted by ( ), has $n = 0$ elements. In this section we begin by defining the ADT *Stack* and follow with its implementation. Then, we look at the queue.

A *stack* is an ordered list in which insertions (also called pushes and adds) and deletions (also called pops and removes) are made at one end called the *top*. Given a stack $S = (a_0, \cdots, a_{n-1})$, we say that a_0 is the bottom element, a_{n-1} is the top element, and a_i is on top of element a_{i-1}, $0 < i < n$. The restrictions on the stack imply that if we add the elements A, B, C, D, E to the stack, in that order, then E is the first element we delete from the stack. Figure 3.1 illustrates this sequence of operations. Since the last element inserted into a stack is the first element removed, a stack is also known as a *Last-In-First-Out (LIFO)* list.

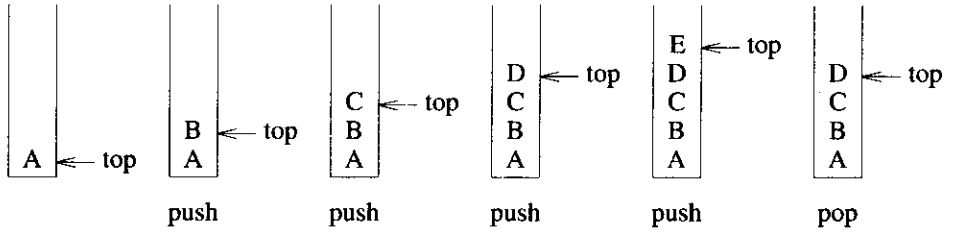


Figure 3.1: Inserting and deleting elements in a stack

Example 3.1 [*System stack*]: Before we discuss the stack ADT, we look at a special stack, called the system stack, that is used by a program at run-time to process function calls. Whenever a function is invoked, the program creates a structure, referred to as an *activation record* or a *stack frame*, and places it on top of the system stack. Initially, the activation record for the invoked function contains only a pointer to the previous stack frame and a return address. The previous stack frame pointer points to the stack frame of the invoking function, while the return address contains the location of the statement to be executed after the function terminates. Since only one function executes at any given time, the function whose stack frame is on top of the system stack is chosen. If this function invokes another function, the local variables, except those declared static, and the parameters of the invoking function are added to its stack frame. A new stack frame is then created for the invoked function and placed on top of the system stack. When this function terminates, its stack frame is removed and the processing of the invoking function, which is again on top of the stack, continues. A simple example illustrates this process.

Assume that we have a main function that invokes function *a1*. Figure 3.2(a) shows the system stack before *a1* is invoked; Figure 3.2(b) shows the system stack after *a1* has been invoked. Frame pointer *fp* is a pointer to the current stack frame. The system also maintains separately a stack pointer, *sp*, which we have not illustrated.

Since all functions are stored similarly in the system stack, it makes no difference if the invoking function calls itself. That is, a recursive call requires no special strategy; the run-time program simply creates a new stack frame for each recursive call. However, recursion can consume a significant portion of the memory allocated to the system stack; it could consume the entire available memory. □

Our discussion of the system stack suggests several operations that we include in the ADT specification (ADT 3.1).

The easiest way to implement this ADT is by using a one-dimensional array, say,

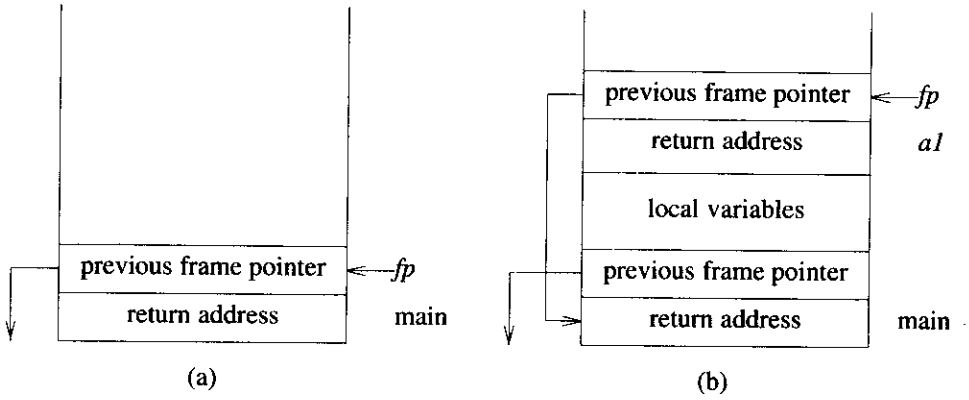


Figure 3.2: System stack after function call

stack[*MAX_STACK_SIZE*], where *MAX_STACK_SIZE* is the maximum number of entries. The first, or bottom, element of the stack is stored in *stack*[0], the second in *stack*[1], and the *i*th in *stack*[*i*−1]. Associated with the array is a variable, *top*, which points to the top element in the stack. Initially, *top* is set to −1 to denote an empty stack. Given this representation, we can implement the operations in ADT 3.1 as follows. Notice that we have specified that *element* is a structure that consists of only a *key* field. Ordinarily, we would not create a structure with a single field. However, we use *element* in this and subsequent chapters as a template whose fields we may add to or modify to meet the requirements of our application.

Stack CreateS(*maxStackSize*) ::=

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
typedef struct {
        int key;
        /* other fields */
        } element;
element stack[MAX_STACK_SIZE];
int top = -1;
```

Boolean IsEmpty(Stack) ::= `top < 0;`

Boolean IsFull(Stack) ::= `top >= MAX_STACK_SIZE-1;`

ADT *Stack* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *stack* ∈ *Stack*, *item* ∈ *element*, *maxStackSize* ∈ positive integer
    Stack CreateS(*maxStackSize*) ::=
                create an empty stack whose maximum size is *maxStackSize*
    Boolean IsFull(*stack*, *maxStackSize*) ::=
                if (number of elements in *stack* == *maxStackSize*)
                return *TRUE*
                else return *FALSE*
    Stack Push(*stack*, *item*) ::=
                if (IsFull(*stack*)) *stackFull*
                else insert *item* into top of *stack* and **return**
    Boolean IsEmpty(*stack*) ::=
                if (*stack* == CreateS(*maxStackSize*))
                return *TRUE*
                else return *FALSE*
    Element Pop(*stack*) ::=
                if (IsEmpty(*stack*)) **return**
                else remove and return the element at the top of the stack.

ADT 3.1: Abstract data type *Stack*

The *IsEmpty* and *IsFull* operations are simple, and we will implement them directly in the *push* (Program 3.1) and *pop* (Program 3.2) functions. Each of these functions assumes that the variables *stack* and *top* are global. The functions are short and require little explanation. Function *push* checks to see if the stack is full. If it is, it calls *stackFull* (Program 3.3), which prints an error message and terminates execution. When the stack is not full, we increment *top* and assign *item* to *stack* [*top*]. Implementation of the pop operation parallels that of the push operation. The code of Program 3.2 assumes that the *stackEmpty* function prints an error message and returns an item of type *element* with a *key* field that contains an error code. Typical function calls would be *push* (*item*); and *item* = *pop* ();.

EXERCISES

1. Implement the *stackEmpty* function.
2. Using Figures 3.1 and 3.2 as examples, show the status of the system stack after each function call for the iterative and recursive functions to compute binomial coefficients (Exercise 9, Section 1.2). You do not need to show the stack frame

```
void push(element item)
{/* add an item to the global stack */
   if (top >= MAX_STACK_SIZE-1)
      stackFull();
   stack[++top] = item;
}
```

Program 3.1: Add an item to a stack

```
element pop()
{/* delete and return the top element from the stack */
   if (top == -1)
      return stackEmpty(); /* returns an error key */
   return stack[top--];
}
```

Program 3.2: Delete from a stack

```
void stackFull()
{
   fprintf(stderr, "Stack is full, cannot add element");
   exit(EXIT_FAILURE);
}
```

Program 3.3: Stack full

itself for each function call. Simply add the name of the function to the stack to show its invocation and remove the name from the stack to show its termination.

3. The Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, $\cdots$, is defined as $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$, $i \geq 2$. Write a recursive function, *fibon* (n), that returns the nth fibonacci number. Show the status of the system stack for the call *fibon* (4) (see Exercise 2). What can you say about the efficiency of this function?

4. Consider the railroad switching network given in Figure 3.3. Railroad cars numbered 0, 1, $\cdots$, $n-1$ are at the right. Each car is brought into the stack and removed at any time. For instance, if $n = 3$, we could move in 0, move in 1, move in 2, and then take the cars out, producing the new order 2, 1, 0. For $n = 3$ and $n = 4$, what are the possible permutations of the cars that can be obtained? Are any permutations not possible?

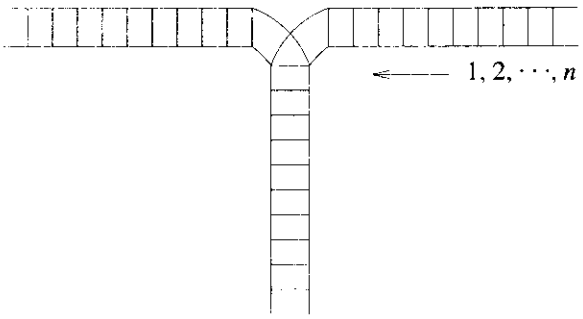


Figure 3.3: Railroad switching network

3.2 STACKS USING DYNAMIC ARRAYS

A shortcoming of the stack implementation of the preceding section is the need to know, at compile time, a good bound (*MAX_STACK_SIZE*) on how large the stack will become. We can overcome this shortcoming by using a dynamically allocated array for the elements and then increasing the size of this array as needed. The following implementation of *CreateS*, *IsEmpty*, and *IsFull* uses a dynamically allocated array *stack* whose initial capacity (i.e., maximum number of stack elements that may be stored in the array) is 1. Specific applications may dictate other choices for the initial capacity.

Stack CreateS() ::=
```
typedef struct {
        int key;
        /* other fields */
        } element;
element *stack;
MALLOC(stack, sizeof(*stack));
int capacity = 1;
int top = -1;
```

Boolean IsEmpty(Stack) ::= `top < 0;`

Boolean IsFull(Stack) ::= `top >= capacity-1;`

While we must alter the code for the *push* function (Program 3.1) to use the new test for a full stack (replace *MAX_STACK_SIZE* with *capacity*), the code for the *pop* function (Program 3.2) is unchanged. Additionally, the code for *stackFull* is changed. The new code for *stackFull* attempts to increase the capacity of the array *stack* so that we can add an additional element to the stack. Before we can increase the capacity of an array, we must decide what the new capacity should be. In *array doubling*, we double array capacity whenever it becomes necessary to increase the capacity of an array. Program 3.4 gives the code for *stackFull* when array doubling is used.

```
void stackFull()
{
   REALLOC(stack, 2 * capacity * sizeof(*stack))
   capacity *= 2;
}
```

Program 3.4: Stack full with array doubling

Although it may appear that a lot of time is spent doubling the capacity of *stack*, this is actually not the case. In the worst case, the *realloc* function needs to allocate 2**capacity* **sizeof* (**stack*) bytes of memory and copy *capacity* **sizeof* (**stack*)) bytes of memory from the old array into the new one. Under the assumptions that memory may be allocated in $O(1)$ time and that a stack element can be copied in $O(1)$ time, the time required by array doubling is $O(capacity)$. Initially, *capacity* is 1. Suppose that when we are done with all the stack pushes we wish to perform, *capacity* is 2^k for some k, $k>0$. The total time spent over all array doublings is $O(\sum_{i=1}^{k} 2^i) = O(2^{k+1}) = O(2^k)$. Since the total number of pushes is more than 2^{k-1} (otherwise the array capacity would not have been doubled from 2^{k-1} to 2^k), the total time spend in array doubling is $O(n)$, where n is the total number of pushes. Hence, even with the time spent on array doubling added in, the total run time of *push* over all n pushes is $O(n)$. Notice that this conclusion remains valid whenever *stackFull* resizes the stack array by a factor $c>1$ ($c=2$ in Program 3.4).

EXERCISES

1. Let S be a stack whose initial capacity is 1 and that array doubling is used to increase the stack's capacity whenever an elemenet is added to a full stack. Let $n=2^k+1$, where k is a positive integer, be the maximum number of elements on S

during the execution of some program. How much memory is needed for this program to run successfully (consider only the memory needed for the stack and the array doubling operation)? How much memory is needed when using the representation of Section 3.1 (assume we can determine k without running the program)?

2. Prove that whenever *stackFull* resizes the stack array by a factor $c>1$, the total time for all invocastions of *push* (Program 3.1) is $O(n)$, where n is the number of pushes to the stack. In the initial configuration, te stack is empty and *capacity* $=1$.
3. Suppose that we modify Program 3.4 so that the size of *stack* is increased by an additive amount $c*sizeof(stack)$. Show that the time for n pushes is $O(n^2)$ when the initial configuration is an empty stack and *capacity* $=1$.

3.3 QUEUES

A *queue* is an ordered list in which insertions (also called additions, puts, and pushes) and deletions (also called removals and pops) take place at different ends. The end at which new elements are added is called the *rear*, and that from which old elements are deleted is called the *front*. The restrictions on a queue imply that if we insert A, B, C, D, and E in that order, then A is the first element deleted from the queue. Figure 3.4 illustrates this sequence of events. Since the first element inserted into a queue is the first element removed, queues are also known as *First-In-First-Out (FIFO)* lists. The ADT specification of the queue appears in ADT 3.2.

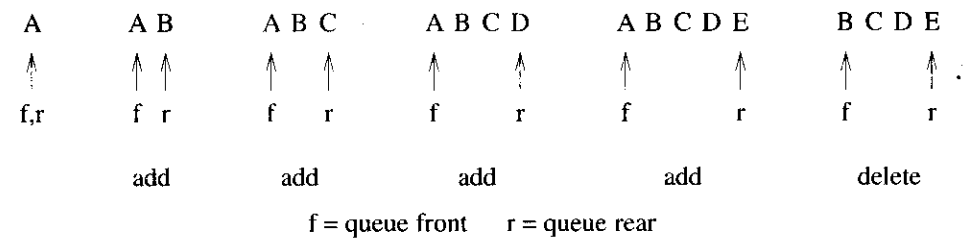


Figure 3.4: Inserting and deleting elements in a queue

The representation of a queue in sequential locations is more difficult than that of the stack. The simplest scheme employs a one-dimensional array and two variables, *front* and *rear*. Given this representation, we can define the queue operations in ADT 3.2 as:

ADT *Queue* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *queue* ∈ *Queue*, *item* ∈ *element*, *maxQueueSize* ∈ positive integer
    Queue CreateQ(*maxQueueSize*) ::=
                create an empty queue whose maximum size is *maxQueueSize*
    Boolean IsFullQ(*queue*, *maxQueueSize*) ::=
                if (number of elements in *queue* == *maxQueueSize*)
                return *TRUE*
                else return *FALSE*
    Queue AddQ(*queue*, *item*) ::=
                if (IsFullQ(*queue*)) *queueFull*
                else insert *item* at rear of *queue* and return *queue*
    Boolean IsEmptyQ(*queue*) ::=
                if (*queue* == CreateQ(*maxQueueSize*))
                return *TRUE*
                else return *FALSE*
    Element DeleteQ(*queue*) ::=
                if (IsEmptyQ(*queue*)) **return**
                else remove and return the *item* at front of queue.

ADT 3.2: Abstract data type *Queue*

Queue CreateQ(*maxQueueSize*) ::=

```
#define MAX_QUEUE_SIZE 100 /* maximum queue size */
typedef struct {
        int key;
        /* other fields */
        } element;
element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
```

Boolean IsEmptyQ(*queue*) ::= `front == rear`

Boolean IsFullQ(*queue*) ::= `rear == MAX_QUEUE_SIZE-1`

Since the *IsEmptyQ* and *IsFullQ* operations are quite simple, we again implement them directly in the *addq* (Program 3.5) and *deleteq* (Program 3.6) functions. The implementation of *queueFull* is similar to that of *stackFull* (Program 3.3). Functions *addq* and *deleteq* are structurally similar to *push* and *pop* on stacks. While the stack uses the

variable *top* in both *push* and *pop*, the queue increments *rear* in *addq* and *front* in *deleteq*. Typical function calls would be *addq* (*item*); and *item* = *deleteq* ();.

```
void addq(element item)
{/* add an item to the queue */
   if (rear == MAX_QUEUE_SIZE-1)
      queueFull();
   queue[++rear] = item;
}
```

Program 3.5: Add to a queue

```
element deleteq()
{/* remove element at the front of the queue */
   if (front == rear)
      return queueEmpty(); /* return an error key */
   return queue[++front];
}
```

Program 3.6: Delete from a queue

This sequential representation of a queue has pitfalls that are best illustrated by an example.

Example 3.2 [*Job scheduling*]: Queues are frequently used in computer programming, and a typical example is the creation of a job queue by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system. Figure 3.5 illustrates how an operating system might process jobs if it used a sequential representation for its queue.

It should be obvious that as jobs enter and leave the system, the queue gradually shifts to the right. This means that eventually the rear index equals *MAX_QUEUE_SIZE* − 1, suggesting that the queue is full. In this case, *queueFull* should move the entire queue to the left so that the first element is again at *queue* [0] and *front* is at − 1. It should also recalculate *rear* so that it is correctly positioned. Shifting an array is very time-consuming, particularly when there are many elements in it. In fact, *queueFull* has a worst case complexity of O(*MAX_QUEUE_SIZE*). □

We can obtain a more efficient queue representation if we we permit the queue to wrap around the end of the array. At this time it is convenient to think of the array

front	*rear*	*Q*[0]	*Q*[1]	*Q*[2]	*Q*[3]	Comments
−1	−1					queue is empty
−1	0	J1				Job 1 is added
−1	1	J1	J2			Job 2 is added
−1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Figure 3.5: Insertion and deletion from a sequential queue

positions as arranged in a circle (Figure 3.6) rather than in a straight line (Figure 3.4). In Figure 3.6, we have changed the convention for the variable *front*. This variable now points one position counterclockwise from the location of the front element in the queue. The convention for *rear* is unchanged. This change simplifies the codes slightly.

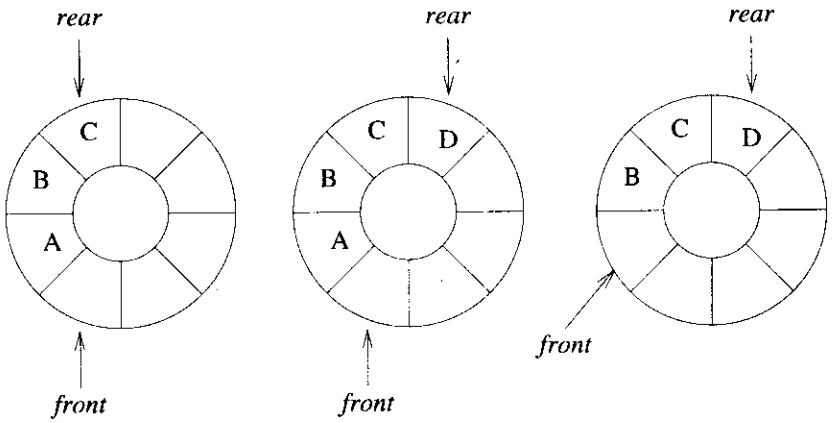


Figure 3.6: Circular queue

When the array is viewed as a circle, each array position has a next and a previous position. The position next to position *MAX_QUEUE_SIZE* − 1 is 0, and the position that precedes 0 is *MAX_QUEUE_SIZE* − 1. When the queue rear is at *MAX_QUEUE_SIZE* − 1, the next element is put into position 0. To work with a

circular queue, we must be able to move the variables *front* and *rear* from their current position to the next position (clockwise). This may be done using code such as

```
if (rear == MAX_QUEUE_SIZE - 1) rear = 0;
else rear++;
```

Using the modulus operator, which computes remainders, this code is equivalent to (*rear*+1) % *MAX_QUEUE_SIZE*. With our conventions for *front* and *rear*, we see that the front element of the queue is located one position clockwise from *front* and the rear element is at position *rear*.

To determine a suitable test for an empty queue, we experiment with the queues of Figure 3.6. To delete an element, we advance *front* one position clockwise and to add an element, we advance *rear* one position clockwise and insert at the new position. If we perform 3 deletions from the queue of Figure 3.6(c) in this fashion, we will see that the queue becomes empty and that *front* = *rear*. When we do 5 additions to the queue of Figure 3.6(b), the queue becomes full and *front* = *rear*. So, we cannot distinguish between an empty and a full queue. To avoid the resulting confusion, we shall increase the capacity of a queue just before it becomes full. Consequently, *front* == *rear* iff the queue is empty. The initial value for both *front* and *rear* is 0. Progrms 3.7 and 3.8, respectively, given the codes to add and delete. The code for *queueFull* is similar to that of the *stackFull* code of Program 3.3.

```
void addq(element item)
{/* add an item to the queue */
   rear = (rear+1) % MAX_QUEUE_SIZE;
   if (front == rear)
      queueFull(); /* print error and exit */
   queue[rear] = item;
}
```

Program 3.7: Add to a circular queue

Observe that the test for a full queue in *addq* and the test for an empty queue in *deleteq* are the same. In the case of *addq*, however, when *front* = **rear* is evaluated and found to be true, there is actually one space free (*queue*[*rear*]) since the first element in the queue is not at *queue*[*front*] but is one position clockwise from this point. As remarked earlier, if we insert an item here, then we will not be able to distinguish between the cases of full and empty, since the insertion would leave *front* equal to *rear*. To avoid this we signal *queueFull*, thus permitting a maximum of *MAX_QUEUE_SIZE* − 1 rather than *MAX_QUEUE_SIZE* elements in the queue at any time. We leave the implementation of *queueFull* as an exercise.

```
element deleteq()
{/* remove front element from the queue */
   element item;
      if (front == rear)
         return queueEmpty(); /* return an error key */
      front = (front+1) % MAX_QUEUE_SIZE;
      return queue[front];
}
```

Program 3.8: Delete from a circular queue

EXERCISES

1. Implement the *queueFull* and *queueEmpty* functions for the noncircular queue.
2. Implement the *queueFull* and *queueEmpty* functions for the circular queue.
3. Using the noncircular queue implementation, produce a series of adds and deletes that requires O(*MAX_QUEUE_SIZE*) for each add. (Hint: Start with a full queue.)
4. A *double-ended queue (deque)* is a linear list in which additions and deletions may be made at either end. Obtain a data representation mapping a deque into a one-dimensional array. Write functions that add and delete elements from either end of the deque.
5. We can maintain a linear list circularly in an array, *circle* [*MAX_SIZE*]. We set up *front* and *rear* indices similar to those used for a circular queue.
   (a) Obtain a formula in terms of *front*, *rear*, and *MAX_SIZE* for the number of elements in the list.
   (b) Write a function that deletes the *k*th element in the list.
   (c) Write a function that inserts an element, *item*, immediately after the *k*th element.
   (d) What is the time complexity of your functions for (b) and (c)?

3.4 CIRCULAR QUEUES USING DYNAMICALLY ALLOCATED ARRAYS

Suppose that a dynamically allocated array is used to hold the queue elements. Let *capacity* be the number of positions in the array *queue*. To add an element to a full queue, we must first increase the size of this array using a function such as *realloc*. As with dynamically allocated stacks, we use array doubling. However, it isn't sufficient to simply double array size using *realloc*. Consider the full queue of Figure 3.7(a). This

figure shows a queue with seven elements in an array whose capacity is 8. To visualize array doubling when a circular queue is used, it is better to flatten out the array as in Figure 3.7(b). Figure 3.7(c) shows the array after array doubling by *realloc*.

To get a proper circular queue configuration, we must slide the elements in the right segment (i.e., elements *A* and *B*) to the right end of the array as in Figure 3.7(d). The array doubling and the slide to the right together copy at most 2**capacity* − 2 elements. The number of elements copied can be limited to *capacity* − 1 by customizing the array doubling code so as to obtain the configuration of Figure 3.7(e). This configuration may be obtained as follows:

(1) Create a new array *newQueue* of twice the capacity.

(2) Copy the second segment (i.e., the elements *queue* [*front* + 1] through *queue* [*capacity* − 1]) to positions in *newQueue* beginning at 0.

(3) Copy the first segment (i.e., the elements *queue* [0] through *queue* [*rear*]) to positions in *newQueue* beginning at *capacity*−*front*−1.

Program 3.9 gives the code to add to a circular queue using a dynamically allocated array. Program 3.10 gives the code for *queueFull*. The function *copy* (*a*,*b*,*c*) copies elements from locations *a* through *b*−1 to locations beginning at *c*. Program 3.10 obtains the configuration of Figure 3.7(e).

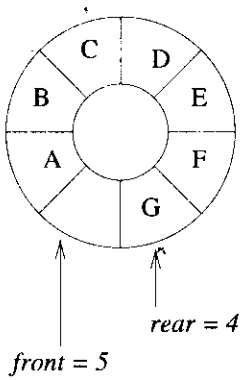
EXERCISES

1. Write and test code that implements the function *copy* used in Program 3.10. Your code should work correctly even when there is some overlap between the memory being copied from and that being copied to.
2. Write and test code for all of the queue functions specified in the queue ADT (ADT 3.2). In addition, include code for the function *queueFront* () that returns the element at the front of the queue but does not delete this element from the queue. In case the queue is empty, your function should print an error message and terminate. You should use array doubling whenever an attempt is made to add an element to a full queue.

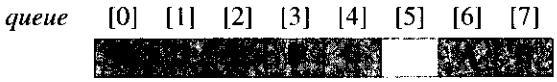
3.5 A MAZING PROBLEM

Mazes have been an intriguing subject for many years. Experimental psychologists train rats to search mazes for food, and many a mystery novelist has used an English country garden maze as the setting for a murder. We also are interested in mazes since they present a nice application of stacks. In this section, we develop a program that runs a maze. Although this program takes many false paths before it finds a correct one, once found it can correctly rerun the maze without taking any false paths.

In creating this program the first issue that confronts us is the representation of the

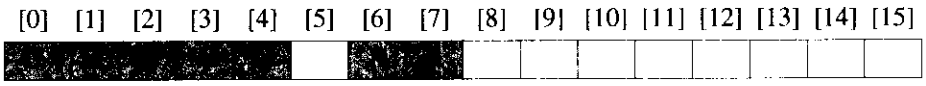


(a) A full circular queue



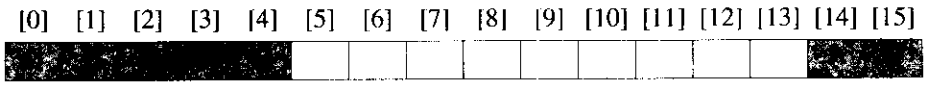
front = 5, *rear* = 4

(b) Flattened view of circular full queue



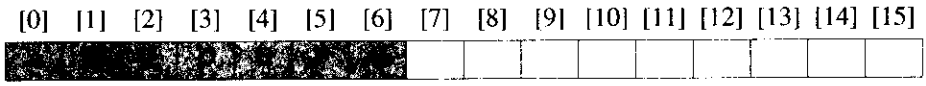
front = 5, *rear* = 4

(c) After array doubling



front = 13, *rear* = 4

(d) After shifting right segment



front = 15, *rear* = 6

(e) Alternative configuration

Figure 3.7: Doubling queue capacity

```
void addq(element item)
{/* add an item to the queue */
   rear = (rear+1) % capacity;
   if (front == rear)
      queueFull(); /* double capacity */
   queue[rear] = item;
}
```

Program 3.9: Add to a circular queue

```
void queueFull()
{
   /* allocate an array with twice the capacity */
   element* newQueue;
   MALLOC(newQueue, 2 * capacity * sizeof(*queue));

   /* copy from queue to newQueue */
   int start = (front+1)  % capacity;
   if (start  <  2)
      /* no wrap around */
      copy(queue+start, queue+start+capacity-1, newQueue);
   else
   {/* queue wraps around */
      copy(queue+start, queue+capacity, newQueue);
      copy(queue, queue+rear+1, newQueue+capacity-start);
   }

   /* switch to newQueue */
   front = 2 * capacity - 1;
   rear = capacity - 2;
   capacity *= 2;
   free(queue);
   queue = newQueue;
}
```

Program 3.10: Doubling queue capacity

maze. The most obvious choice is a two dimensional array in which zeros represent the open paths and ones the barriers. Figure 3.8 shows a simple maze. We assume that the rat starts at the top left and is to exit at the bottom right. With the maze represented as a two-dimensional array, the location of the rat in the maze can at any time be described by the row and column position. If X marks the spot of our current location, *maze*[*row*][*col*], then Figure 3.9 shows the possible moves from this position. We use compass points to specify the eight directions of movement: north, northeast, east, southeast, south, southwest, west, and northwest, or N, NE, E, SE, S, SW, W, NW.

entrance

0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

exit

Figure 3.8: An example maze (can you find a path?)

We must be careful here because not every position has eight neighbors. If [*row*,*col*] is on a border then less than eight, and possibly only three, neighbors exist. To avoid checking for these border conditions we can surround the maze by a border of ones. Thus an $m \times p$ maze will require an $(m+2) \times (p+2)$ array. The entrance is at position [1][1] and the exit at [m][p].

Another device that will simplify the problem is to predefine the possible directions to move in an array, *move*, as in Figure 3.10. This is obtained from Figure 3.9. We represent the eight possible directions of movement by the numbers from 0 to 7. For each direction, we indicate the vertical and horizontal offset. The C declarations needed to create this table are:

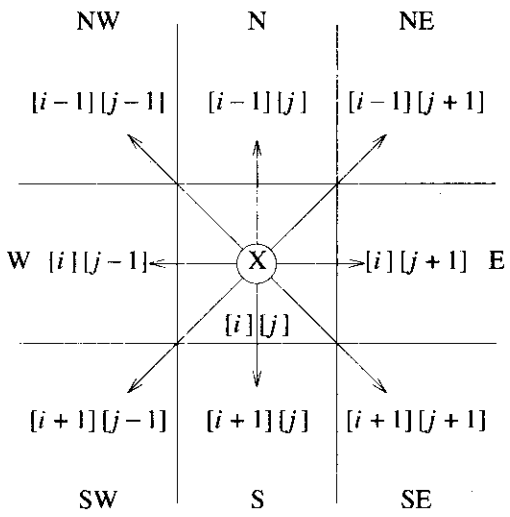


Figure 3.9: Allowable moves

Name	Dir	*move[dir].vert*	*move[dir].horiz*
N	0	−1	0
NE	1	−1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	−1
W	6	0	−1
NW	7	−1	−1

Figure 3.10: Table of moves

```
typedef struct {
        short int vert;
        short int horiz;
        } offsets;
offsets move[8]; /* array of moves for each direction */
```

We assume that *move* is initialized according to the data provided in Figure 3.10. This means that if we are at position, *maze*[*row*][*col*], and we wish to find the position of the next move, *maze*[*nextRow*][*nextCol*], we set:

```
nextRow = row + move[dir].vert;
nextCol = col + move[dir].horiz;
```

As we move through the maze, we may have the choice of several directions of movement. Since we do not know which choice is best, we save our current position and arbitrarily pick a possible move. By saving our current position, we can return to it and try another path if we take a hopeless path. We examine the possible moves starting from the north and moving clockwise. Since we do not want to return to a previously tried path, we maintain a second two-dimensional array, *mark*, to record the maze positions already checked. We initialize this array's entries to zero. When we visit a position, *maze*[*row*][*col*], we change *mark*[*row*][*col*] to one. Program 3.11 is our initial attempt at a maze traversal algorithm. *EXIT_ROW* and *EXIT_COL* give the coordinates of the maze exit.

Although this algorithm describes the essential processing, we must still resolve several issues. Our first concern is with the representation of the stack. Examining Program 3.11, we see that the stack functions created in Sections 3.1 and 3.2 will work if we redefine *element* as:

```
typedef struct {
        short int row;
        short int col;
        short int dir;
        } element;
```

If we use the stack implementation of Sction 3.1, we also need to determine a reasonable bound for the stack size. While such a bound is not required when array doubling is used as in Section 3.2, we will need more memory on our computer to guarantee successful completion of the program (see Exercise 1 of Section 3.2). Since each position in the maze is visited no more than once, the stack need have only as many positions as there are zeroes in the maze. The maze of Figure 3.11 has only one entrance to exit path. When searching this maze for an entrance to exit path, all positions (except the exit) with value zero will be on the stack when the exit is reached. Since, an $m \times p$ maze, can have at most mp zeroes, it is sufficient for the stack to have this capacity.

Program 3.12 contains the maze search algorithm. We assume that the arrays, *maze*, *mark*, *move*, and *stack*, along with the constants *EXIT_ROW*, *EXIT_COL*, *TRUE*, and *FALSE*, and the variable, *top*, are declared as global. Notice that *path* uses a variable *found* that is initially set to zero (i.e., *FALSE*). If we find a path through the maze, we set this variable to *TRUE*, thereby allowing us to exit both **while** loops gracefully.

```
initialize a stack to the maze's entrance coordinates and
direction to north;
while (stack is not empty) {
   /* move to position at top of stack */
   <row,col,dir> = delete from top of stack;
   while (there are more moves from current position) {
      <nextRow, nextCol> = coordinates of next move;
      dir = direction of move;
      if ((nextRow == EXIT_ROW) && (nextCol == EXIT_COL))
         success;
      if (maze[nextRow][nextCol] == 0 &&
                  mark[nextRow][nextCol] == 0) {
      /* legal move and haven't been there */
         mark[nextRow][nextCol] = 1;
         /* save current position and direction */
         add <row,col,dir> to the top of the stack;
         row = nextRow;
         col = nextCol;
         dir = north;
      }
   }
}
printf("No path found\n");
```

Program 3.11: Initial maze algorithm

Analysis of *path*: The size of the maze determines the computing time of *path*. Since each position within the maze is visited no more than once, the worst case complexity of the algorithm is $O(mp)$ where m and p are, respectively, the number of rows and columns of the maze. □

EXERCISES

1. Describe how you could model a maze with horizontal and vertical walls by a matrix whose entries are zeroes and ones. What moves are permitted in your matrix model? Provide an example maze together with its matrix model.
2. Do the previous exercise for the case of mazes that have walls that are at 45 and 135 degrees in addition to horizontal and vertical ones.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 3.11: Simple maze with a long path

3. What is the maximum path length from start to finish for any maze of dimensions *rows* × *columns*?
4. (a) Find a path through the maze of Figure 3.8.
   (b) Trace the action of function path on the maze of Figure 3.8. Compare this to your own attempt in (a).
5. § [***Programming project***] Using the information provided in the text, write a complete program to search a maze. Print out the entrance to exit path if successful.

3.6 EVALUATION OF EXPRESSIONS

3.6.1 Expressions

The representation and evaluation of expressions is of great interest to computer scientists. As programmers, we write complex expressions such as:

$$((rear+1==front) \;||\; ((rear==MAX\_QUEUE\_SIZE-1) \;\&\&\; !front)) \qquad (3.1)$$

or complex assignment statements such as:

$$x=a/b-c+d*e-a*c \qquad (3.2)$$

If we examine expression (3.1), we notice that it contains operators (==, +, −, ||, &&, !), operands (*rear*, *front*, *MAX_QUEUE_SIZE*), and parentheses. The same is true of the statement (3.2), although the operands and operators have changed, and there are no parentheses.

The first problem with understanding the meaning of these or any other expressions and statements is figuring out the order in which the operations are performed. For instance, assume that $a = 4$, $b = c = 2$, $d = e = 3$ in statement (3.2). We want to find the value of x. Is it

```
void path(void)
{/* output a path through the maze if such a path exists */
   int i, row, col, nextRow, nextCol, dir, found = FALSE;
   element position;
   mark[1][1] = 1; top = 0;
   stack[0].row = 1;  stack[0].col = 1;  stack[0].dir = 1;
   while (top > -1 && !found) {
      position = pop();
      row = position.row;  col = position.col;
      dir = position.dir;
      while (dir <  8 && !found) {
         /* move in direction dir */
         nextRow = row + move[dir].vert;
         nextCol = col + move[dir].horiz;
         if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
         else if ( !maze[nextRow][nextCol] &&
         ! mark[nextRow][nextCol]) {
            mark[nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
         }
         else ++dir;
      }
   }
   if (found) {
      printf("The path is:\n");
      printf("row  col\n");
      for (i = 0; i <= top; i++)
         printf("%2d%5d",stack[i].row, stack[i].col);
      printf("%2d%5d\n",row,col);
      printf("%2d%5d\n",EXIT_ROW,EXIT_COL);
   }
   else printf("The maze does not have a path\n");
}
```

Program 3.12: Maze search function

$$((4/2)-2)+(3*3)-(4*2)$$
$$=0+9-8$$
$$=1$$

or

$$(4/(2-2+3))*(3-4)*2$$
$$=(4/3)*(-1)*2$$
$$=-2.66666\cdots$$

Most of us would pick the first answer because we know that division is carried out before subtraction, and multiplication before addition. If we wanted the second answer, we would have written (3.2) differently, using parentheses to change the order of evaluation:

$$x=((a/(b-c+d))*(e-a)*c \tag{3.3}$$

Within any programming language, there is a precedence hierarchy that determines the order in which we evaluate operators. Figure 3.12 contains the precedence hierarchy for C. We have arranged the operators from highest precedence to lowest. Operators with the same precedence appear in the same box. For instance, the highest precedence operators are function calls, array elements, and structure or union members, while the comma operator has the lowest precedence. Operators with highest precedence are evaluated first. The associativity column indicates how we evaluate operators with the same precedence. For instance, the multiplicative operators have left-to-right associativity. This means that the expression *a*b/c%d/e* is equivalent to ((((*a*b*)/*c*)%*d*)/*e*). In other words, we evaluate the operator that is furthest to the left first. With right associative operators of the same precedence. we evaluate the operator furthest to the right first. Parentheses are used to override precedence, and expressions are always evaluated from the innermost parenthesized expression first.

3.6.2 Evaluating Postfix Expressions

The standard way of writing expressions is known as infix notation because in it we place a binary operator in-between its two operands. We have used this notation for all of the expressions written thus far. Although infix notation is the most common way of writing expressions, it is not the one used by compilers to evaluate expressions. Instead compilers typically use a parenthesis-free notation referred to as postfix. In this notation, each operator appears after its operands. Figure 3.13 contains several infix expressions and their postfix equivalents.

Before writing a function that translates expressions from infix to postfix, we tackle the easier task of evaluating postfix expressions. This evaluation process is much simpler than the evaluation of infix expressions because there are no parentheses to consider. To evaluate an expression we make a single left-to-right scan of it. We place the operands on a stack until we find an operator. We then remove, from the stack, the

Token	Operator	Precedence[1]	Associativity
() [] → . 	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

1. The precedence column is taken from Harbison and Steele.
2. Postfix form
3. Prefix form

Figure 3.12: Precedence hierarchy for C

correct number of operands for the operator, perform the operation, and place the result back on the stack. We continue in this fashion until we reach the end of the expression. We then remove the answer from the top of the stack. Figure 3.14 shows this processing when the input is the nine character string 6 2/3–4 2*+.

Infix	Postfix
2+3*4	2 3 4*+
*a*b +5	*ab*5+
(1+2)*7	1 2+7*
*a*b/c	*ab*c/
((*a*/(*b* −*c* +*d*))*(*e* −*a*)*c	*abc* −*d* +/*ea* −**c**
a/*b* −*c* +*d***e* −*a***c*	*ab*/*c* −*de***ac***−

Figure 3.13: Infix and postfix notation

Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
–	6/2–3			0
4	6/2–3	4		1
2	6/2–3	4	2	2
*	6/2–3	4*2		1
+	6/2–3+4*2			0

Figure 3.14: Postfix evaluation

We now consider the representation of both the stack and the expression. To simplify our task we assume that the expression contains only the binary operators +, –, *, /, and % and that the operands in the expression are single digit integers as in Figure 3.14. This permits us to represent the expression as a character array. The operands are stored on a stack of type **int** until they are needed. We may use either of the representations of Sections 3.1 and 3.2. It is convenient to define the enumerated type *precedence*, which lists the operators by mnemonics, as below:

```
typedef enum {lparen, rparen, plus, minus, times, divide,
              mod, eos, operand} precedence;
```

Although we will use it to process tokens (operators, operands, and parentheses) in this example, its real importance becomes evident when we translate infix expressions into postfix ones. Besides the usual operators, the enumerated type also includes an end-of-string (*eos*) operator.

The function *eval* (Program 3.13) contains the code to evaluate a postfix expression. Since an operand (*symbol*) is initially a character, we must convert it into a single digit integer. We use the statement, *symbol* – '0', to accomplish this task. The statement takes the ASCII value of *symbol* and subtracts the ASCII value of '0', which is 48, from it. For example, suppose *symbol* = '1'. The character, '1', has an ASCII value of 49. Therefore, the statement *symbol* – '0' produces as result the number 1.

We use an auxiliary function, *getToken* (Program 3.14), to obtain tokens from the expression string. If the token is an operand, we convert it to a number and add it to the stack. Otherwise, we remove two operands from the stack, perform the specified operation, and place the result back on the stack. When we have reached the end of expression, we remove the result from the stack.

3.6.3 Infix to Postfix

We can describe an algorithm for producing a postfix expression from an infix one as follows:

(1) Fully parenthesize the expression.

(2) Move all binary operators so that they replace their corresponding right parentheses.

(3) Delete all parentheses.

For example, $a/b-c+d*e-a*c$ when fully parenthesized becomes:

$$((((a/b)-c)+(d*e))-a*c))$$

Performing steps 2 and 3 gives:

$$ab/c-de*+ac*-$$

Although this algorithm works well when done by hand, it is inefficient on a computer because it requires two passes. The first pass reads the expression and parenthesizes it, while the second moves the operators. Since the order of operands is the same in infix and postfix, we can form the postfix equivalent by scanning the infix expression left-to-right. During this scan, operands are passed to the output expression as they are encountered. However, the order in which the operators are output depends

```
int eval(void)
{/* evaluate a postfix expression, expr, maintained as a
    global variable. '\0' is the the end of the expression.
    The stack and top of the stack are global variables.
    getToken is used to return the token type and
    the character symbol. Operands are assumed to be single
    character digits */
  precedence token;
  char symbol;
  int op1, op2;
  int n = 0; /* counter for the expression string */
  int top = -1;
  token = getToken(&symbol, &n);
  while (token != eos) {
    if (token == operand)
      push(symbol-'0'); /* stack insert */
    else {
      /* pop two operands, perform operation, and
         push result to the stack */
      op2 = pop(); /* stack delete */
      op1 = pop();
      switch(token) {
        case plus: push(op1+op2);
                   break;
        case minus: push(op1-op2);
                    break;
        case times: push(op1*op2);
                    break;
        case divide: push(op1/op2);
                     break;
        case mod: push(op1%op2);
      }
    }
    token = getToken(&symbol, &n);
  }
  return pop(); /* return result */
}
```

Program 3.13: Function to evaluate a postfix expression

```
precedence getToken(char *symbol, int *n)
{/* get the next token, symbol is the character
    representation, which is returned, the token is
    represented by its enumerated value, which
    is returned in the function name */
   *symbol = expr[(*n)++];
   switch (*symbol) {
      case '(' : return lparen;
      case ')' : return rparen;
      case '+' : return plus;
      case '-' : return minus;
      case '/' : return divide;
      case '*' : return times;
      case '%' : return mod;
      case ' ' : return eos;
      default  : return operand; /* no error checking,
                                    default is operand */
   }
}
```

Program 3.14: Function to get a token from the input string

on their precedence. Since we must output the higher precedence operators first, we save operators until we know their correct placement. A stack is one way of doing this, but removing operators correctly is problematic. Two examples illustrate the problem.

Example 3.3 [*Simple expression*]: Suppose we have the simple expression $a+b*c$, which yields $abc*+$ in postfix. As Figure 3.15 illustrates, the operands are output immediately, but the two operators need to be reversed. In general, operators with higher precedence must be output before those with lower precedence. Therefore, we stack operators as long as the precedence of the operator at the top of the stack is less than the precedence of the incoming operator. In this particular example, the unstacking occurs only when we reach the end of the expression. At this point, the two operators are removed. Since the operator with the higher precedence is on top of the stack, it is removed first. □

Example 3.4 [*Parenthesized expression*]: Parentheses make the translation process more difficult because the equivalent postfix expression will be parenthesis-free. We use as our example the expression $a*(b+c)*d$, which yields $abc+*d*$ in postfix. Figure 3.16 shows the translation process. Notice that we stack operators until we reach the

Token	Stack [0]	[1]	[2]	Top	Output
a				−1	*a*
+	+			0	*a*
b	+			0	*ab*
*	+	*		1	*ab*
c	+	*		1	*abc*
eos				−1	*abc**+

Figure 3.15: Translation of $a+b*c$ to postfix

right parenthesis. At this point we unstack until we reach the corresponding left parenthesis. We then delete the left parenthesis from the stack. (The right parenthesis is never put on the stack.) This leaves us with only the $*d$ remaining in the infix expression. Since the two multiplications have equal precedences, one is output before the d, the second is placed on the stack and removed after the d is output. □

Token	Stack [0]	[1]	[2]	Top	Output
a				−1	*a*
*	*			0	*a*
(	*	(		1	*a*
b	*	(		1	*ab*
+	*	(	+	2	*ab*
c	*	(	+	2	*abc*
)	*			0	*abc* +
*	*			0	*abc* +*
d	*			0	*abc* +**d*
eos	*			0	*abc* +**d**

Figure 3.16: Translation of $a*(b+c)*d$ to postfix

The analysis of the two examples suggests a precedence-based scheme for stacking and unstacking operators. The left parenthesis complicates matters because it behaves like a low-precedence operator when it is on the stack, and a high-precedence

one when it is not. It is placed in the stack whenever it is found in the expression, but it is unstacked only when its matching right parenthesis is found. Thus, we have two types of precedence, an *in-stack precedence* (*isp*) and an *incoming precedence* (*icp*). The declarations that establish these precedences are:

```
/* isp and icp arrays -- index is value of precedence
   lparen, rparen, plus, minus, times, divide, mod, eos */
int isp[] = {0,19,12,12,13,13,13,0};
int icp[] = {20,19,12,12,13,13,13,0};
```

Notice that we are now using the stack to store the mnemonic for the token. That is, the data type of the stack elements is *precedence*. Since the value of a variable of an enumerated type is simply the integer corresponding to the position of the value in the enumerated type, we can use the mnemonic as an index into the two arrays. For example, *isp* [*plus*] is translated into *isp* [2], which gives us an in-stack precedence of 12. The precedences are taken from Figure 3.12, but we have added precedences for the left and right parentheses and the *eos* marker. We give the right parenthesis an in-stack and incoming precedence (19) that is greater than the precedence of any operator in Figure 3.12. We give the left parenthesis an instack precedence of zero, and an incoming precedence (20) greater than that of the right parenthesis. In addition, because we want unstacking to occur when we reach the end of the string, we give the *eos* token a low precedence (0). These precedences suggest that we remove an operator from the stack only if its instack precedence is greater than or equal to the incoming precedence of the new operator.

The function *postfix* (Program 3.15) converts an infix expression into a postfix one using the process just discussed. This function invokes a function, *print_token*, to print out the character associated with the enumerated type. That is, *print_token* reverses the process used in *get_token*.

Analysis of *postfix*: Let n be the number of tokens in the expression. $\Theta(n)$ time is spent extracting tokens and outputting them. Besides this, time is spent in the two **while** loops. The total time spent here is $\Theta(n)$ as the number of tokens that get stacked and unstacked is linear in n. So, the complexity of function *postfix* is $\Theta(n)$. □

EXERCISES

1. Write the postfix form of the following expressions:
   (a) $a * b * c$
   (b) $-a + b - c + d$
   (c) $a * -b + c$
   (d) $(a + b) * d + e / (f + a * d) + c$

```
void postfix(void)
{/* output the postfix of the expression. The expression
    string, the stack, and top are global */
   char symbol;
   precedence token;
   int n = 0;
   int top = 0;    /* place eos on stack */
   stack[0] = eos;
   for (token = getToken(&symbol, &n); token != eos;
                          token = getToken(&symbol,&n)) {
      if (token == operand)
         printf("%c", symbol);
      else if (token == rparen) {
         /* unstack tokens until left parenthesis */
         while (stack[top] != lparen)
            printToken(pop());
         pop();  /* discard the left parenthesis */
      }
      else {
         /* remove and print symbols whose isp is greater
            than or equal to the current token's icp */
         while(isp[stack[top]] >= icp[token])
           printToken(pop());
         push(token);
      }
   }
   while ( (token = pop()) != eos)
     printToken(token);
   printf("\n");
}
```

Program 3.15: Function to convert from infix to postfix

(e) $a \&\& b \| c \| !(e > f)$ (assuming C precedence)

(f) $!(a \&\& !((b < c) \| (c > d))) \| (c < e)$

2. Write the *print_token* function used in *postfix* (Program 3.15).
3. Use the precedences of Figure 3.12 together with those for '(', ')', and \0 to answer the following:

   (a) In the postfix function, what is the maximum number of elements that can be

on the stack at any time if the input expression, *expr*, has n operators and an unlimited number of nested parentheses?

(b) What is the answer to (a) if *expr* has n operators and the depth of the nesting of parentheses is at most six?

4. Rewrite the *eval* function so that it evaluates the unary operators + and −.

5. § Rewrite the *postfix* function so that it works with the following operators, besides those used in the text: &&, !!, <<, >>, <=, !=, <, >, <=, and >=. (Hint: Write the equation so that the operators, operands, and parentheses are separated with a space, for example, $a + b > c$. Then review the functions in <*string.h*>.)

6. Another expression form that is easy to evaluate and is parenthesis-free is known as prefix. In prefix notation, the operators precede their operands. Figure 3.17 shows several infix expressions and their prefix equivalents. Notice that the order of operands is the same in infix and prefix.

Infix	Prefix
*a*b/c*	*/*abc*
*a/b−c+d*e−a*c*	*−+−/abc*de*ac*
a(b+c)/d−g*	*−/*a+bcdg*

Figure 3.17: Infix and postfix expressions

(a) Write the prefix form of the expressions in Exercise 1.

(b) Write a C function that evaluates a prefix expression, *expr*. (Hint: Scan *expr* from right to left.)

(c) Write a C function that transforms an infix expression, *expr*, into its prefix equivalent.

What is the time complexity of your functions for (b) and (c)? How much space is needed by each of these functions?

7. Write a C function that transforms a prefix expression into a postfix one. Carefully state any assumptions you make regarding the input. How much time and space does your function take?

8. Write a C function that transforms a postfix expression into a prefix one. How much time and space does your function take?

9. Write a C function that transforms a postfix expression into a fully parenthesized infix expression. A fully parenthesized expression is one in which all the subexpressions are surrounded by parentheses. For example, $a+b+c$ becomes

$((a+b)+c)$. Analyze the time and space complexity of your function.

10. Write a C function that transforms a prefix expression into a fully parenthesized infix expression. Analyze the time and space complexity of your function.
11. § Repeat Exercise 5, but this time transform the infix expression into prefix.

3.7 MULTIPLE STACKS AND QUEUES

Until now we have been concerned only with the representations of a single stack or a single queue. In both cases, we have seen that it is possible to obtain efficient sequential representations. We would now like to examine the case of multiple stacks. (We leave the consideration of multiple queues as an exercise.) We again examine only sequential mappings of stacks into an array, *memory*[*MEMORY_SIZE*]. If we have only two stacks to represent, the solution is simple. We use *memory*[0] for the bottom element of the first stack, and *memory*[*MEMORY_SIZE* − *1*] for the bottom element of the second stack. The first stack grows toward *memory*[*MEMORY_SIZE* − *1*] and the second grows toward *memory*[0]. With this representation, we can efficiently use all the available space.

Representing more than two stacks within the same array poses problems since we no longer have an obvious point for the bottom element of each stack. Assuming that we have n stacks, we can divide the available memory into n segments. This initial division may be done in proportion to the expected sizes of the various stacks, if this is known. Otherwise, we may divide the memory into equal segments.

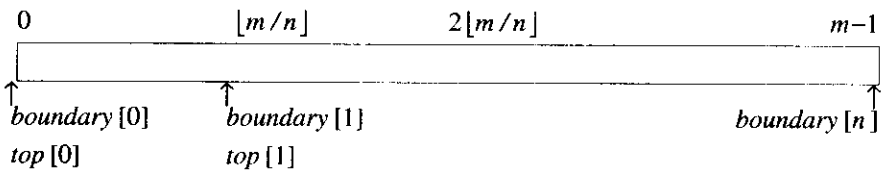
Assume that i refers to the stack number of one of the n stacks. To establish this stack, we must create indices for both the bottom and top positions of this stack. The convention we use is that *boundary*[i], $0 \le i <$ *MAX_STACKS*, points to the position immediately to the left of the bottom element of stack i, while *top*[i], $0 \le i <$ *MAX_STACKS* points to the top element. Stack i is empty iff *boundary*[i] = *top*[i]. The relevant declarations are:

```
#define MEMORY_SIZE 100 /* size of memory */
#define MAX_STACKS 10 /* max number of stacks plus 1 */
/* global memory declaration */
element memory[MEMORY_SIZE];
int top[MAX_STACKS];
int boundary[MAX_STACKS];
int n;  /* number of stacks entered by the user */
```

To divide the array into roughly equal segments we use the following code:

```
top[0] = boundary[0] = -1;
for (j = 1; j < n; j++)
   top[j] = boundary[j] = (MEMORY_SIZE/n)*j;
boundary[n] = MEMORY_SIZE-1;
```

Figure 3.18 shows this initial configuration. In the figure, n is the number of stacks entered by the user, $n < MAX\_STACKS$, and $m = MEMORY\_SIZE$. Stack i can grow from *boundary*[i] + 1 to *boundary*[i + 1] before it is full. Since we need a boundary for the last stack, we set *boundary*[n] to $MEMORY\_SIZE - 1$. Programs 3.16 and 3.17 implement the add and delete operations for this representation.



All stacks are empty and divided into roughly equal segments.

Figure 3.18: Initial configuration for n stacks in *memory*[m].

```
void push(int i, element item)
{/* add an item to the ith stack */
   if (top[i] == boundary[i+1])
      stackFull(i);
   memory[++top[i]] = item;
}
```

Program 3.16: Add an item to the ith stack

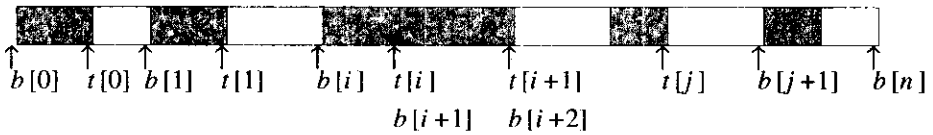
The *push* (Program 3.16) and *pop* (Program 3.17) functions for multiple stacks appear to be as simple as those we used for the representation of a single stack. However, this is not really the case because the *top*[i] == *boundary*[i+1] condition in *push* implies only that a particular stack ran out of memory, not that the entire memory is full. In fact, there may be a lot of unused space between other stacks in array *memory* (see Figure 3.19). Therefore, we create an error recovery function, *stackFull*, which determines if there is any free space in memory. If there is space available, it should shift the stacks so that space is allocated to the full stack.

There are several ways that we can design *stackFull* so that we can add elements to this stack until the array is full. We outline one method here. Other methods are discussed in the exercises. We can guarantee that *stackFull* adds elements as long as there is free space in array *memory* if we:

(1) Determine the least, j, $i < j < n$, such that there is free space between stacks j and j

```
element pop(int i)
{/* remove top element from the ith stack */
   if (top[i] == boundary[i])
      return stackEmpty(i);
   return memory[top[i]--];
}
```

Program 3.17: Delete an item from the *i*th stack



b = boundary, t = top

Figure 3.19: Configuration when stack i meets stack $i + 1$, but the memory is not full

+ 1. That is, $top[j] < boundary[j+1]$. If there is such a j, then move stacks $i+1$, $i+2$, $\cdots$, j one position to the right (treating $memory[0]$ as leftmost and $memory[MEMORY\_SIZE - 1]$ as rightmost). This creates a space between stacks i and $i+1$.

(2) If there is no j as in (1), then look to the left of stack i. Find the largest j such that $0 \leq j < i$ and there is space between stacks j and $j+1$. That is, $top[j] < boundary[j+1]$. If there is such a j, then move stacks $j+1$, $j+2$, $\cdots$, i one space to the left. This also creates a space between stacks i and $i+1$.

(3) If there is no j satisfying either condition (1) or condition (2), then all *MEMORY_SIZE* spaces of memory are utilized and there is no free space. In this case *stackFull* terminates with an error message.

We leave the implementation of *stackFull* as an exercise. However, it should be clear that the worst case performance of this representation for the n stacks together will be poor. In fact, in the worst case, the function has a time complexity of O(*MEMORY_SIZE*).

EXERCISES

1. We must represent two stacks in an array, *memory*[*MEMORY_SIZE*]. Write C functions that add and delete an item from stack i, $0 \leq i < n$. Your functions should be able to add elements to the stacks as long as the total number of elements in both stacks is less than *MEMORY_SIZE* − *1*.
2. Obtain a data representation that maps a stack and a queue into a single array, *memory*[*MEMORY_SIZE*]. Write C functions that add and delete elements from these two data objects. What can you say about the suitability of your data representation?
3. Write a C function that implements the *stackFull* strategy discussed in the text.
4. Using the add and delete functions discussed in the text and *stackFull* from Exercise 3, produce a sequence of additions/deletions that requires O(*MEMORY_SIZE*) time for each add. Assume that you have two stacks and that your are starting from a configuration representing a full utilization of *memory*[*MEMORY_SIZE*].
5. Rewrite the *push* and *stackFull* functions so that the *push* function terminates if there are fewer than c_1 free spaces left in memory. The empirically determined constant, c_1 shows when it is futile to move items in memory. Substitute a small constant of your choice.
6. Design a data representation that sequentially maps n queues into an array *memory*[*MEMORY_SIZE*]. Represent each queue as a circular queue within memory. Write functions *addq*, *deleteq*, and *queueFull* for this representation.

3.8 ADDITIONAL EXERCISES

1. § [***Programming project***] [Landweber] People have spent so much time playing solitaire that the gambling casinos are now capitalizing on this human weakness. A form of solitaire is described below. You must write a C program that plays this game, thus freeing hours of time for people to return to more useful endeavors.

   To begin the game, 28 cards are dealt into seven piles. The leftmost pile has one card, the next pile has two cards, and so forth, up to seven cards in the rightmost pile. Only the uppermost card of each of the seven piles is turned face-up. The cards are dealt left-to-right, one card to each pile, dealing one less pile each time, and turning the first card in each round face-up. You may build descending sequences of red on black or black on red from the top face-up card of each pile. For example, you may place either the eight of diamonds or the eight of hearts on the nine of spades or the nine of clubs. All face-up cards on a pile are moved as a unit and may be placed on another pile according to the bottom face-up card. For example, the seven of clubs on the eight of hearts may be moved as a unit onto the nine of clubs or the nine of spades.

Whenever a face-down card is uncovered, it is turned face-up. If one pile is removed completely, a face-up king may be moved from a pile (together with all cards above it) or the top of the waste pile (see below) into the vacated space. There are four output piles, one for each suite, and the object of the game is to get as many cards as possible into the output piles. Each time an ace appears at the top of a pile or the top of the stack it is moved into the appropriate output pile. Cards are added to the output piles in sequence, the suit for each pile being determined by the ace on the bottom.

From the rest of the deck, called the stock, cards are turned up one by one and placed face-up on a waste pile. You may always play cards off the top of the waste pile, but only one at a time. Begin by moving a card from the stock to the top of the waste pile. If you can ever make more than one possible play, make them in the following order:

(a) Move a card from the top of a playing pile or from the top of the waste pile to an output pile. If the waste pile becomes empty, move a card from the stock to the waste pile.

(b) Move a card from the top of the waste pile to the leftmost playing pile to which it can be moved. If the waste pile becomes empty, move a card from the stock to the waste pile.

(c) Find the leftmost playing pile that can be moved and place it on top of the leftmost playing pile to which it can be moved.

(d) Try (a), (b), and (c) in sequence, restarting with (a) whenever a move is made.

(e) If no move is made via (a) through (d), move a card from the stock to the waste pile and retry (a).

Only the top card of the playing piles or the waste pile may be played to an output pile. Once placed on an output pile, a card may not be withdrawn to help elsewhere. The game is over when either all the cards have been played to the output piles, or the stock pile has been exhausted and no more cards can be moved.

When played for money, the player pays the house \$52 at the beginning, and wins \$5 for every card played to the output piles. Write your program so that it will play several games and determine your net winnings. Use a random number generator to shuffle the deck. Output a complete record of two games in easily understandable form. Include as output the number of games played and the net winnings (+ or −).

2. § [***Programming project***] [Landweber] We want to simulate an airport landing and takeoff pattern. The airport has three runways, runway 0, runway 1, and runway 2. There are four landing holding patterns, two for each of the first two runways

Arriving planes enter one of the holding pattern queues, where the queues are to be as close in size as possible. When a plane enters a holding queue, it is assigned an integer identification number and an integer giving the number of time units the plane can remain in the queue before it must land (because of low fuel level). There is also a queue for takeoffs for each of the three runways. Planes arriving in a takeoff queue are assigned an integer identification number. The takeoff queues should be kept approximately the same size.

For each time period, no more than three planes may arrive at the landing queues and no more than three planes may enter the takeoff queues. Each runway can handle one takeoff or landing at each time slot. Runway 2 is used for takeoffs except when a plane is low on fuel. During each time period, planes in either landing queue whose air time has reached zero must be given priority over other landings and takeoffs. If only one plane is in this category, runway 2 is used. If there is more than one plane, then the other runways are also used.

Use successive even(odd) integers for identification numbers of the planes arriving at takeoff (landing) queues. At each time unit assume that arriving planes are entered into queues before takeoffs or landings occur. Try to design your algorithm so that neither landing nor takeoff queues grow excessively. However, arriving planes must be placed at the ends of queues and the queues cannot be reordered.

Your output should label clearly what occurs during each time unit. Periodically you should also output:

(a) the contents of each queue

(b) the average takeoff waiting time

(c) the average landing waiting time

(d) the number of planes that have crashed (run out of fuel and there was no open runway) since the last time period.