CHAPTER 2
ARRAYS AND STRUCTURES

2.1 The array as an abstract data type

- 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*.
- 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.
    - » 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 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.
  - Structure 2.1 shows a definition of the array ADT

– 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.”

---

structure *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 ith element is the the size of the ith 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*

Structure 2.1: Abstract Data Type *Array*

2.2 Structures and unions

- Structures
  - Arrays are collections of data of the same type. In C there is an alternate way of grouping data that permit the data to vary in type.
    - This mechanism is called the **struct**, short for structure.
  - A structure is a collection of data items, where each item is identified as to its type and name.
    - For example, the following *struct* creates a variable whose name is *person* and there has three fields:
      - Ê a name that is a character array
      - Ë an integer value representing the age of the person
      - Ì a float value representing the salary of the individual
      - »

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

– We may assign values to these fields as below.

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

or

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

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

human_being person1, person2;

– We can also embed a structure within a structure.

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

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

» A person born on February 11, 1994, would have values for the *date* **struct** set as

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

- Unions
  - 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 *human_being* to:

```
typedef struct sex_type {
  enum tag_field {female, male} sex;
  union {
     int children;
     int beard ;
     } u;
  };
typedef struct human_being {
      char name[10];
      int age;
      float salary;
      date dob;
      sex_type sex_info;
      };
human_being person1, person2;
```

Then we can assign values to person1 and person2, such as

person1.sex_info.sex = male;

person1.sex_info.u.beard = False;

and

person2.sex_info.sex = female;

person2.sex_info.u.children = 4;

2.3 The polynomial abstract data type

- Polynomial examples:
  - Two example polynomials are:
    - $A(x) = 3x^{20}+2x^5+4$ and $B(x) = x^4+10x^3+3x^2+1$
  - Assume that we have two polynomials, $A(\mathrm{x}) = \Sigma a^i x^i$ and $B(x) = \Sigma b^i x^i$ then:
    - $A(x) + B(x) = \Sigma(a^i + b^i)x^i$
    - $A(x) \cdot B(x) = \Sigma(a^i x^i \cdot \Sigma(b^j x^j))$
    - Similarly, we can define subtraction and division on polynomials, as well as many other operations.
- An ADT definition of a polynomial is contained in Structure 2.2. (next page)

structure *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 $\in$ *Polynomial*, *coef* $\in$ *Coefficients*, *expon* $\in$ *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* $\in$ *poly*) **return** its coefficient **else return** zero
Exponent Lead_Exp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** (*expon* $\in$ *poly*) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
Polynomial Remove(*poly*, *expon*)	::=	**if** (*expon* $\in$ *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 $\cdot$ *poly*2

end *Polynomial*

Structure 2.2: Abstract data type *Polynomial*

– There are two ways to create the type *polynomial* in C

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

drawback: the first representation may waste space.

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

- Examples:
  - Consider the two polynomials $A(x) = 2x^{1000} + 1$ and $B(x) = x^4 + 10x^3 + 3x^2 + 1$.
  - Figure 2.2 shows how these polynomials are stored in the array *terms*.

	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 2.2: Array representation of two polynomials

- 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 2.5) adds $A(x)$ and $B(x)$ term by term.

(next page)

```
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 2.5: Function to add two polynomials

```
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(1);
   }
   terms[avail].coef = coefficient;
   terms[avail++].expon = exponent;
}
```

Program 2.6: Function to add a new term

- **Analysis of padd:**

  The asymptotic computing time of this algorithm is $O(n+m)$.

2.4 The sparse matrix abstract data type

- Introduction
  - In mathematics, a matrix contains *m* rows and *n* columns of elements as illustrated in Figure 2.3
    - In this figure, 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.

$$
\begin{array}{c}
\\
\text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4}
\end{array}
\begin{array}{c}
\begin{array}{ccc} \text{col 0} & \text{col 1} & \text{col 2} \end{array} \\
\begin{bmatrix}
-27 & 3 & 4 \\
6 & 82 & -2 \\
109 & -64 & 11 \\
12 & 8 & 9 \\
48 & 27 & 47
\end{bmatrix}
\end{array}
$$

(a)

$$
\begin{array}{c}
\\
\text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \\ \text{row 5}
\end{array}
\begin{array}{c}
\begin{array}{cccccc} \text{col 0} & \text{col 1} & \text{col 2} & \text{col 3} & \text{col 4} & \text{col 5} \end{array} \\
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

(b)

Figure 2.3: Two matrices

- The standard representation of a matrix is 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 the standard representation.
  - For instance, if you look at Figure 2.3(b), you notice that it contains many zero entries.
  - We call this a *sparse matrix*.
- Since a sparse matrix wastes space, we must consider alternate forms of representation.
  - Our representation of sparse matrices should store only nonzero elements.

– Structure 2.3 contains our specification of the matrix ADT.

- A minimal set of operations includes matrix creation, addition, multiplication, and transpose.

---

structure *Sparse_Matrix* 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 $a, b \in$ *Sparse_Matrix*, $x \in$ *item*, *i*, *j*, *max_col*, *max_row* $\in$ *index*

Sparse_Matrix Create(*max_row*, *max_col*) ::=
return a *Sparse_Matrix* that can hold up to *max_items* = *max_row* $\times$ *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col*.

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

Sparse_Matrix 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

Sparse_Matrix 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.

Structure 2.3: Abstract data type *Sparse_Matrix*

- We implement the *Create* operation as below:

Sparse_Matrix Create(*max_row*, *max_col*) ::=

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

- Figure 2.4(a) shows how the sparse matrix of Figure 2.3(b) is represented in the array *a*.

	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 2.4: Sparse matrix and its transpose stored as triples

- Transposing a 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.
    - Figure 2.4(b) shows the transpose of the sample matrix.

	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 2.4: Sparse matrix and its transpose stored as triples

– An idea to transpose a matrix

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

– This algorithm is incorporated in transpose (Program 2.7).

```
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 2.7: Transpose of a sparse matrix

– **Analysis of *transpose*:**

The asymptotic time complexity is $O(columns \bullet elements)$.

– In fact, we can transpose a matrix represented as a sequence of triples in *O*(columns + elements) time.
  - This algorithm, *fast_transpose* is shown in Program 2.8.

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

column

element

column

element

+)

O(2col + 2elm)

=O(col+elm)

Program 2.8: Fast transpose of a sparse matrix

- Matrix Multiplication
  - **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 $0 \leq i < m$ and $0 \leq j < p$.
  - Example:

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

Figure 2.5: Multiplication of two sparse matrices

– The programs 2.9 and 2.10 can obtain the product matrix *D* which multiplies matrices *A* and *B*.

```
void mmult(term a[], term b[], term d[])
/* multiply two sparse matrices */
{
  int i, j, column, totalb = b[0].value, totald = 0;
  int rows_a = a[0].row, cols_a = a[0].col,
  totala = a[0].value; int  cols_b = b[0].col,
  int row_begin = 1, row = a[1].row, sum = 0;
  int new_b[MAX_TERMS][3];
  if (cols_a != b[0].row) {
    fprintf(stderr,"Incompatible matrices\n");
    exit(1);
  }
  fast_transpose(b,new_b);
  /* set boundary condition */
  a[totala+1].row = rows_a;
  new_b[totalb+1].row = cols_b;
  new_b[totalb+1].col = 0;
  for (i = 1; i <= totala; ) {
    column = new_b[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 = row_begin;
        for (; new_b[j].row == column; j++)
          ;
        column = new_b[j].row;
      }
      else if (new_b[j].row != column) {
        storesum(d, &totald, row, column, &sum);
        i = row_begin;
        column = new_b[j].row;
      }
      else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
              break;
        case 1 : /* advance to next term in b */
              j++;
      }
    } /* end of for j <= totalb+1 */
    for (; a[i].row == row; i++)
      ;
    row_begin = i; row = a[i].row;
  } /* end of for i<=totala */
  d[0].row = rows_a;
  d[0].col = cols_b; d[0].value =  totald;
}
```

Program 2.9: Sparse matrix multiplication

【課本 Program 2.9 有點囉嗦，課本的解釋也不很清楚，其實矩陣相乘的概念大家都懂，可以自己縯繹】

```
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(1);
      }
}
```

Program 2.10: *storesum* function

– Analysis of *mmult*:

The asymptotic time complexity is $O(cols\_b \bullet total\_a + rows\_a \bullet totalb)$.

A

r	c	value
0	0	$
0	1	#
0	2	@
1	0	*
1	1	*
1	2	*
*	*	*
*	*	*

$$\begin{vmatrix} \$ \; \# \; @ \\ \cdots\cdots\cdots \\ \cdots\cdots\cdots \end{vmatrix}$$

B

r	c	value
0	0	…
0	1	…
0	2	…
1	0	*
1	1	*
1	2	*
*	*	*
*	*	*

$$\begin{vmatrix} \$ & \$ & \$ & \$ & \$ & \$ & \$ & \$ \\ \# & \# & \# & \# & \# & \# & \# & \# \\ @ & @ & @ & @ & @ & @ & @ & @ \end{vmatrix}$$

Sparse Matrix Multiplicatoin

【課本 Program 2.9 有點囉嗦，課本的解釋也不很清楚，其實矩陣相乘的概念大家都懂，可以自己繽繹】

$$\underset{a}{\begin{bmatrix} 3 & 1 & 4 \\ 2 & 5 & 8 \\ 6 & 0 & 9 \end{bmatrix}} \times \underset{b}{\begin{bmatrix} 4 & 3 \\ 5 & 1 \\ 7 & 2 \end{bmatrix}} = \underset{d}{\begin{bmatrix} \% & \$ \\ \# & @ \\ * & \& \end{bmatrix}}$$

【這二個矩陣雖然不是稀疏矩陣，但做法相同！】

$$d[i, j] = \sum_k a[i, k] \times b[k, j] = \sum_k a[i, k] \times b^T[\, j, k]$$

這是說，若把 b 矩陣作了 transpost. 則相乘的指標要更改

例

$$d[i, j] = \sum_k a[i, k] \times b[k, j] = \sum_k a[i, k] \times b^T[j, k]$$

a	Row	Col	Val
a[0]	0	0	3
a[1]	0	1	1
a[2]	0	2	4
a[3]	1	0	2
a[4]	1	1	5
a[5]	1	2	8
a[6]	2	0	6
a[7]	2	1	0
a[8]	2	2	9

b	Row	Col	Val
b[0]	0	0	4
b[1]	0	1	3
b[2]	1	0	5
b[3]	1	1	1
b[4]	2	0	7
b[5]	2	1	2

b^T	Row	Col	Val
b[0]	0	0	4
b[1]	0	1	5
b[2]	0	2	7
b[3]	1	0	3
b[4]	1	1	1
b[5]	1	2	2

1) 用b[k, j] 求： $d[i, j] = \sum_{k} a[i, k] \times b[k, j]$

$d[2,1] = \sum a[i, k] \times b[k, j] = \sum a[2, k] \times b[k, 1]$

$= 6*3 + 0*1 + 9*2$

◆ 缺點是要用到的 b[k,1] 是 b[0,1], b[1,1], b[2,1]，分別落在 b 矩陣的三個隔開的位置，若 b 是個較大的矩陣時，access them become costly.

2) 用b^T[j, k] 求： $d[i, j] = \sum_{k} a[i, k] \times b^T[j, k]$

$d[2,1] = \sum a[i, k] \times b^T[j, k] = \sum a[2, k] \times b^T[1, k]$

$= 6*3 + 0*1 + 9*2$

◆ 優點是要用到的 b^T[1, k] 是 b[1,0], b[1,1], b[1,2]，是在 b^T 矩陣的連續位置，access them become easy.

2.5 Representation of multidimensional array

– The internal representation of multidimensional arrays requires more complex addressing formula.

  - If an array is declared $a[upper_0][upper_1]\ldots[upper_n]$, 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.

  - Example:

    If we declare *a* as $a[10][10][10]$, then we require 10*10*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.
  - 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, …, $row_{upper0-1}$, each row containing *upper1* elements.
  - If we assume that α is the address of A[0][0], then the address of $A[i][0]$ is $\alpha + i \bullet upper_1$ 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.
    - The address of an arbitrary element, $a[i][j]$, is $\alpha + i \bullet upper_1 + j$.

- 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 $\alpha + i \bullet upper_1 \bullet upper_2$ as the address of $a[i][0][0]$ because there are i two dimensional arrays of size $upper_1 \bullet upper_2$ preceding this element.
  - Combining this formula with the formula for addressing a two-dimensional array, we obtain

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

    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:

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

  – The address for $A[i_0][i_1]\ldots[i_{n-1}]$ is:

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

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

2.6 The string abstract data type

– 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.
  - We have listed the essential operations in Structure 2.4.
  - Actually there are many more operation on strings, as we shall see when we look at part of C's string library in Figure 2.7. (next page)

structure *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*.

Structure 2.4: Abstract data type *String*

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 2.7: C string functions

- 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 2.8 shows how these string would be represented internally in memory.

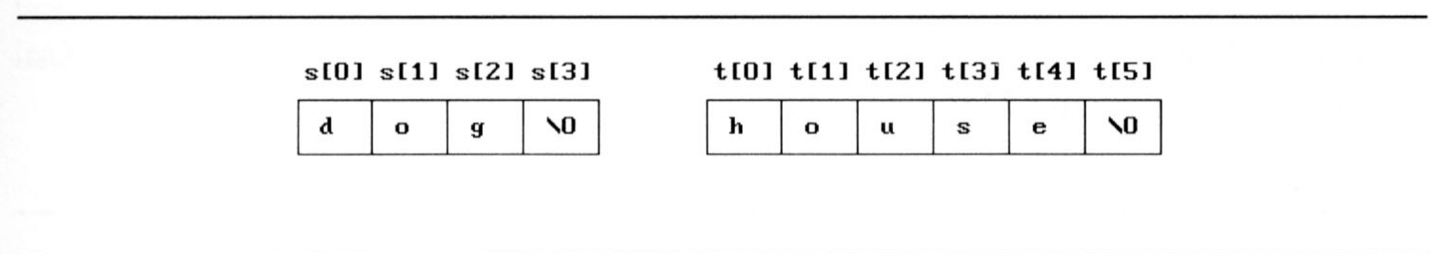

Figure 2.8: String representation in C

- **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 *i*th position of *string 1*.
  - We begin with the declarations:

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

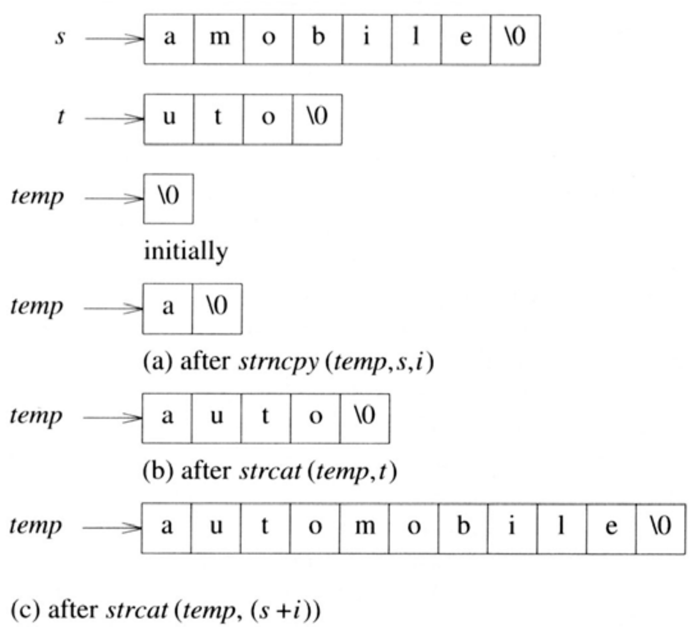

Figure 2.9: String insertion example

- Pattern matching
  - 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*.

```
/* declarations */
char pat[MAX_SIZE], string[MAX_SIZE], *t;
/* statements to determine if pat is in string */
if (t = strstr(string, pat))
   printf(“The string from strstr is: %s\n”, t);
else
   printf(“The pattern was not found with strstr\n”);
```

    - There are two reasons why we may want to develop our own pattern matching function:
      - The function *strstr* is new to ANSI C. Therefore, it may not be available with the compiler we use.
      - The easiest but least efficient method sequentially examines each character of string until it finds the pattern or reach the end of the string. (The time complexity is high.)

– A second improvement is checking the first and last characters of *pat* and *string* before we check the remaining characters.
  - These changes are incorporated in nfind (Program 2.12)

```
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 2.12: Pattern matching by checking end indices first

- **Example 2.3 [*Simulation of nfind*]:**
  - Suppose *pat* = “aab” and *string* = “ababbaabaa”. Figure 2.10 shows how *nfind* compares the characters from *pat* with those of *string*.

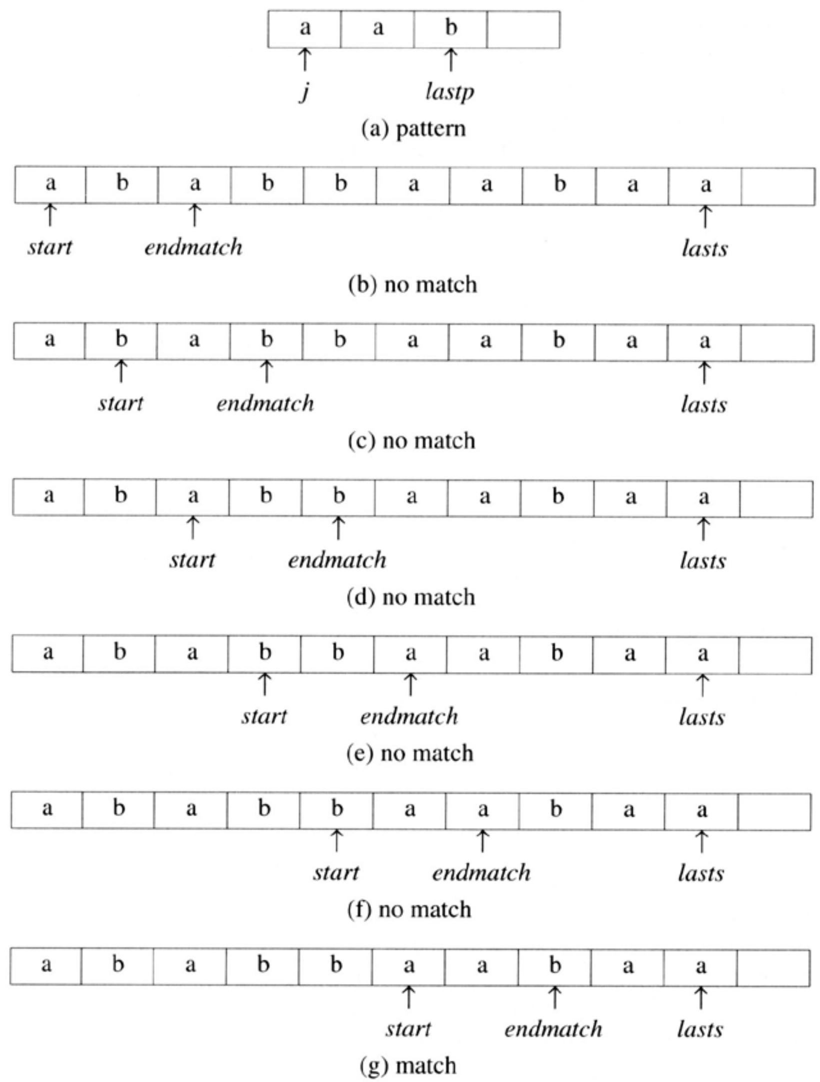

Figure 2.10: Simulation of *nfind*

- **Analysis of *nfind*:**
  - The worst case of computing time of *nfind* is O($n \bullet m$), where n is the length of *pat* and m is the length of *string*.

– Knuth, Morris, and Pratt have developed a pattern matching algorithm that works in O(*strlen*(*string*) + *strlen*(*pat*)).

- Example:

pat = '*abcabcacab*'

Let $s=s_0s_1\ldots s_{m-1}$ 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, we may proceed by comparing s_{i+1} and a.

» 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*'

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}.

» If we have 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.

– **Definition:**

- If $p=p_0p_1\ldots p_{n-1}$ is a pattern, then its failure function, f, is defined as:

$f(j)$ = largest $i < j$ such that $p_0p_1 \ldots p_i = p_{j-i}p_{j-i+1}\ldots p_j$, if such an i >=0 exists

= -1, otherwise

- Example:

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

若 ……s_{i-j}…….s_{i-1} s_i………

……$p_0 p_1$…..p_{j-1} …..

Matching may be resumed by comparing (i.e., matching 可以直接跳到):

- **s_i and $p_{f(j-1)+1}$, if j <> 0**
- **s_{i-j+1} and p_0 , if j=0** (i.e., 連一個都沒有比對到，故最上面一行的 s_{i-j} 即 s_i。所以接下來由隔壁的 s_{i-j+1} 和 p_0 去比對(所以課本只寫s_{i+1}))

在此例中， s = ……a b c a ? ? ? ? ….

p = a b c a b ……..

j=0 1 2 3 4 ….

s = …. a b c a ? ?

P = ab ……

因match到 s_{i-1}, 故 s_i 即第一個 “?”處，即

而可以比對到 p_{j-1}是在j為3之處，故j=4;

故 $p_{f(j-1)+1} = p_{f(3)+1} = p_{0+1} = p_1$, 亦即 matching 可移到 s_i 和 p_1 之處來比。

```
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 2.13: Knuth, Morris, Pratt pattern matching algorithm

- **Analysis of *pmatch*:**

  The complexity of function *pmatch* is O(*strlen*(*string*)).

– There is a fast way to compute 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 $f^m=f(f^{m-1}(j))$).

```
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 2.14: Computing the failure function

– **Analysis of *fail*:**

The complexity of function *fail* is O(*strlen*(*pat*)).

KMP (Knuth-Morris-Pratt) Algorithm

T = a b a b d b c d e

P = a b a b a

f(x) = 最多有幾個字母，能使 prefix 和 suffix 是一樣的？

Step 1. 設定 P 的 e(x) and f(x) 值。

e(x)： 0, 1, 2, 3, ……

f(x)： a => 0

a b => 0

a b a => 1

a b a b => 2

P 本身不用求 f 值 ~~a b a b a => 3~~

以abab 為例，最多可選到2個字母時， prefix 是 ab，且suffix 也是 ab，二者相等。

故 f(x) = 2.

當選到三個字母時，prefix是

aba, 而suffix 是 bab，二者不相等了。

T = a b a b d b c d e

e: 0 1 2 3 4

P = a b a b a

f: -1 $\underbrace{0\ \ 0\ \ 1\ \ 2}$ ← 這是將前頁的 f 值抄過來，**<u>但第一個固定是 “-1”</u>**

Step 2. 比對時，從左到右，找到第一個P不等於T的字母，假設其 f 值是 k. 將 P 往右移到 e 值 = k. 但若 k = -1, 則將 P 往右移一個字母

T = a b a b $\boxed{d}$ b c d e

e: 0 1 2 3 4

P = a b a b a

f: -1 0 0 1 2

前四個字母，P 都等於T。第五個字母不相等。該字母的 f(x)=2. 故應將 e = 2 的那個字母（i.e,. a）移到 f = 2 的那位置。

Step 3. 繼續往下比對時，不必再從 P 的第一個字母比對起了。可以從 Step 2 裡發生比對錯誤的字母開始往下比即可。亦即，

T = a b a b d b c d e

e: 0 1 2 3 4

P = a b a b a

f: -1 0 0 1 2

接著就是一再重覆這些步驟。

因此，下一步是比較 d 與 a。

因為不相等，故將如紅色箭頭所示找到 e = f 之處. 故P 將會向右移二個字母。然後再將 f 歸位。

至此已比對完畢，此例裡我們並沒有找到完全相等的字串。

$$
\begin{array}{llccccccccc}
 & T = & a & b & a & b & d & b & c & d & e \\
e: & & & & & & 0 & 1 & 2 & 3 & 4 \\
 & P = & & & & & a & b & a & b & a \\
f: & & & & & & -1 & 0 & 0 & 1 & 2
\end{array}
$$

若 T 後面還有字母，還可以再比對下去，那麼，因為 d ≠ a, 而 a 的 f 值是 -1. 逢 -1, 則 P 往右一格。繼續下去。

Example 2

T = a a a a a a a b

e 0 1 2 3 4

P = a a a a b

f -1 $\underbrace{0\ \ 1\ \ 2\ \ 3}$

求 f 值

a → 0

a a → 1

a a a → 2

a a a a → 3

~~a a a a b~~

不能包含整個字串；所以不包括prefix和suffix都是aa 的case

T = a a a a a a a b

e 0 1 2 3 4

P = a a a a b

f -1 0 1 2 3

前四個字母都相等，第五個字母不相等，該處 f = 3. 故應將 e = 3 之位置對齊到 f = 3 處。亦即往右一格。

T = a a a a a a a b

e 0 1 2 3 4

P = a a a a b

f -1 0 1 2 3

但好處是繼續往下比對時，只需從 e=3處往下比對即可。前面三個 a 皆可跳過。

Example 3

T = g h j k n o p q

e 0 1 2 3 4

P = a b c d e

f -1 0 0 0 0

求 f 值

a ➔ 0

a b ➔ 0

a b c ➔ 0

a b c d ➔ 0

~~a b c d e~~

T =		g	h	j	k	n	o	p	q
e(x)	-1	0	1	2	3	4			
P =		a	b	c	d	e			
f(x)		-1	0	0	0	0			

T =	g	h	j	k	n	o	p	q
e(x)	-1	0	1	2	3	4		
P =		a	b	c	d	e		
f(x)	-1	0	0	0	0			

Step 1. 比對 g 與 a，不相等。

Step 2. 應該將 a 的 f(x) 值（i.e., -1) 與e(x) 比對，找到相等的。但 <u>e(x) 的 -1 在那裡</u>？ <u>**在 0 的左邊**</u>。

Step 3. 故，移動時將 e(x) = -1 對準 f(x) = -1，亦即，只能將 P 往右移動一格。然後再把 f(x) 右移歸位

Step 4. Repeat until the end.