Chapter 2 : ARRAYS AND STRUCTURES

Data Structures Lecture Note

Prof. Sungwon Jung
Big Data Processing Laboratory
Dept. of Computer Science and Engineering
Sogang University

2.1 ARRAYS

2.1.1 The Abstract Data Type

- An *array* is usually viewed as "a consecutive set of memory locations" which is a usual implementation.

- An *array* as an ADT is a set of pairs, <*index, value*>, such that each index that is defined has a value associated with it.

- Aside from creating a new array, most languages provide only two standard operations for arrays,

  (1) *retrieving a value*
  (2) *storing a value*

■ <Abstract Data Type *Array*>

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 set of one or more dimensions, for example, {0, . . ., 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 ∈ *Array*, *i* ∈ *index*, *x* ∈ *item*, *j, size* ∈ integer

Array Create(*j, list*) ::= **return** an array of *j* dimensions where *list* is a *j*-tuple whose *i*th element is the size of the *i*th dimension. *Items* are undefined. B = Create(3,(4,5,7)) → B[4][5][7]
Item Retrieve(A, *i*) ::= **if** (*i* ∈ *index*) **return** the item associated with index value *i* in array A
else return error.
Array Store(A, *i, x*) ::= **if** (*i* ∈ *index*) **return** an array that is identical to array A except the new pair <*i, x*> has been inserted
else return error.

end *Array*

2.1.2 Arrays in C

■ Declaration of one-dimensional arrays in C :

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

■ Memory allocation of arrays :

Variable	Memory address
list[0]	base address = list
list[1]	list + sizeof(int)
list[2]	list + 2·sizeof(int)
list[3]	list + 3·sizeof(int)
list[4]	list + 4·sizeof(int)

C interprets list[i] as a pointer to an integer.

- Observe the difference between a declaration such as

  int *list1;

  and

  int list2[5];

Variables *list1* and *list2* are both pointers to an integer type object.
list2 is a pointer to *list2*[0] and *list2+i* is a pointer to *list2*[i].

Thus, (*list2+i*) equals &*list2*[i].
So, *(*list2+i*) equals *list2*[i].

- **[Program 2.1]**

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;
void main(void)
{
        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;
}
```

- 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 '=' in an assignment statement, a dereference takes place and the value pointed at by (*list+i*) is returned.
- If *list*[i] appears on the left-hand side of '=', then the value produced on the right-hand side is stored in the location (*list+i*).

- **Example 2.1 [One-dimensional array addressing]**

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

  A function that prints out both the address of the *i*th element of this and the value found at this address.

- **[Program 2.2]**

```
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");
}
```

[Figure 2.1] One-dimensional array addressing

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

2.2 DYNAMICALLY ALLOCATED ARRAYS

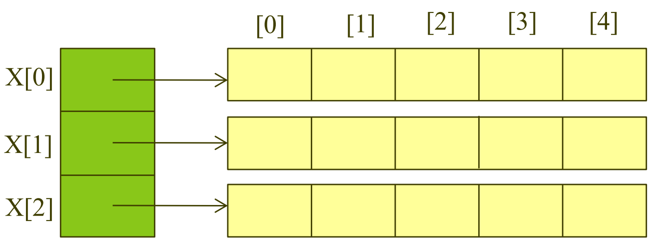
2.2.1 ONE-DIMENSIONAL ARRAYS

- If the user wishes to change array size, we have to change the definition of *MAX_SIZE* and recompile the program.
- 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.

```
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));
```

2.2.2 TWO-DIMENSIONAL ARRAYS

- A 2-D array is represented as a 1-D array in which each element is itself a 1-D array
- (e.g.) int x[3][5];



- A 3-D array is represented as a 1-D array in which each element is itself a 2-D array

- **[Program 2.3]**

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

cf.  x = (int **)malloc(rows*sizeof(*x));
```

=> int x[rows][cols]

■void* calloc(elt_count, elt_size)

→ allocates a region of memory large enough to hold an array of *elt_count* elements, each of size *elt_size*, and the region of memory is set to zero

■void* realloc(p, s)

→ changes the size of memory block pointed at by p to s

2.3 STRUCTURES AND UNIONS

2.3.1 Structures

- A *structure* (called a *record* in many other programming language) is a collection of data items, where each item is identified as to its *type* and *name*.
- For example, the following declaration creates a variable whose name is *person* with three fields.

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

- The structure member operator · is used to select a particular member of the structure.

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

- Creating new structure data types by using the *typedef* statement :

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

```
humanBeing  person1,  person2;
```



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

* Entire structure operation ?

<person1 = person2>.   person1 = person2;

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

<person1 == person2>. if (person1 == person2)

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

[Program 2.4]

```
int humans_equal(human_being person1, human_being 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;
}

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


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

2.3.2 Unions

- Fields share their memory space → only one field of union is active at any given time

```
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;

person1.sex_info.sex = male;
person1.sex_info.u.beard = FALSE;

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

2.3.3 Internal Implementation of Structures

- In most cases we need not be concerned with exactly how the C compiler will store the fields of structure in memory.
- Generally, the values will be stored in the same way using increasing address location in the order specified in the structure definition.

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 structure usually require dynamic storage management routine (*malloc* and *free*) to explicitly obtain and release memory.

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

list  item1, item2, item3;

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

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


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.
- One of the simplest and most commonly found data structures:
  ordered list or *linear list*.

$$(item_0, item_1, \cdots, item_{n-1})$$

- Examples :
  - 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 the building :
    (basement, lobby, mezzanine, first, second)

  etc.

- Possible operations on the ordered lists :
  - Finding the length, *n*, of a list.
  - Reading the items in a list from left to right (or right to left).
  - Retrieving the *i*th item from a list, $0 \le i < n$.
  - Replacing the item in the *i*th position of a list, $0 \le i < n$.
  - Inserting a new item in the *i*th position of a list, $0 \le i < n$.
    The items previously numbered $i, i+1, \ldots, n-1$ become items numbered $i+1, i+2, \ldots, n$.
  - Deleting an item from the *i*th position of a list, $0 \le i < n$.
    The items previously numbered $i+1, \ldots, n$ become items numbered $i, i+1, \ldots, n-1$.

- Implementations (ways to represent an ordered list) :
  - Sequential mapping
  - Nonsequential mapping

- A *polynomial* (viewed from a mathematical perspective) is a sum of terms, where each term has a form ax^e,
  where *x* is a variable, *a* is the coefficient, and *e* is the exponent.

  For example :

  A(X) = $3X^2 + 2X^5 + 4$
  B(X) = $X^4 + 10X^3 + 3X^2 + 1$

  Standard mathematical definitions for sum and product of polynomials.

  For A(x) = $\sum a_i x^i$ and B(x) = $\sum b_i x^i$
  A(x) + B(x) = $\sum (a_i + b_i) x^i$
  A(x)·B(x) = $\sum (a_i x^i \bullet (\sum b_i x^i))$

[ADT 2.2] Abstract Data Type *Polynomial*

ADT *Polynomial* is

Objects : $p(x) = a_1 x^{e_1} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $< a_i, e_i >$
where a_i in *Coefficients* and e_i in *Exponents*, 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
Coefficients Coef(*poly,expon*)	::= **if** (*expon* ∈ *poly*) **return** its coefficient **else return** zero
Exponent LeadExp(*poly*)	::= **return** the largest exponen 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(*poly1,poly2*)	::= **return** the polynomial *poly1* + *poly2*
Polynomial Mult(*poly1,poly2*)	::= **return** the polynomial *poly1* · *poly2*

end *Polynomial*

2.4.2 Polynomial Representation

- **[Program 2.5] Initial version of padd function**

$A(X) = 3X^2 + 2X^5 + 4$
$B(X) = X^4 + 10X^3 + 3X^2 + 1$

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


- Representation
  Exponents are uniquely arranged in decreasing order.

<Dense Representation>

Include all the terms in a polynomial :

A(x) = $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$, where $a_n \neq 0$

```
#define MAX_DEGREE 1001
typedef  struct  {
    int  degree;
    float  coef[MAX_DEGREE];
    }  polynomial;
```

Let *a* be a variable of type polynomial.
We can represent the polynomial A(x) = $\sum a_i x^i$ in *a*,
by setting a.degree = *n* and a.coef[i] = a_{n-i}, $0 \le i \le n$.

- 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 or if the polynomial is sparse.
  Examples:

```
#define MAX_DEGREE 10001
typedef struct {
    int degree;
    float coef[MAX_DEGREE];
    } polynomial;
```

Let *a* be a variable of type polynomial.
We can represent the polynomial A(x) = $\sum a_i x^i$ in *a*,
by setting a.degree = *n* and a.coef[i] = a_{n-i}, 0≤*i*≤*n*.

$$A(x) = 2x^{1000} + 1$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$

struct {int degree; float coef[1001];} polys;

degree = 1000

0	1	2	3	…	998	999	1000
2	0	0	0	…	0	0	1

degree = 4

0	1	2	3	4	…	999	1000
1	10	2	0	1	0	0	0

- **<Sparse Representation>**

  To preserve space, we use only one global array to store all our polynomials.

```
#define  MAX_TERMS  100
typedef  struct {
        float  coef;
        int  expon;
        } polynomial;
polynomial  terms[MAX_TERMS];
int  avail = 0;
```

[Figure 2.2] : Array representation of two polynomials

	starta ↓	finisha ↓	startb ↓			finishb ↓	avail ↓			
coef	2	1	1	10	3	1				
expon	1000	0	4	3	2	0				
	0	1	2	3	4	5	6	7	8	

Examples : $A(x) = 2x^{1000} + 1, B(x) = x^4 + 10x^3 + 3x^2 + 1$

To represent a zero polynomial *c*, set *startc* > *finishc*.

2.4.3 Polynomial Addition

[Program 2.6] : Function to add two polynomials

```
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 2.7] : Function to add a new term**

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

- Analysis of *padd* :

  Time complexity is O(n+m), where *m* and *n* are the number of terms in A and B, respectively.

  When *avail* > MAX_TERMS, must we quit?

2.5 THE SPARSE MATRIX

2.5.1 The Abstract Data Type

- A *sparse matrix* is a matrix which contains many zero entries.
- If a two-dimensional array is used to represent a sparse matrix, a lot of space is used to store the same value 0 and this implementation does not work when the matrices are large since most compilers impose limits on array sizes.

[Figure 2.3]

	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)

[ADT 2.3] ADT *Sparse Matrix*

ADT *Sparse_Matrix* is

objects : a set of triples, <*row, column, value*>, where *row* and *column* are integers and from 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*×*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 dimension 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.

end *Sparse_matrix*

2.5.2 Sparse Matrix Representation

- We can characterize uniquely any element within a matrix by a triple <*row, col, value*>.
  Thus we can use an array of triples.

- We organize the triples so that row indices are in ascending order and among those with the same row indices are ordered in ascending order of column indices.

- To insure that the operations terminate, we must know the number of rows and columns, and the number of nonzero elements in the matrix.

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];
```


	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

[Figure 2.5] For example,

	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)

2.5.3 Transposing A Matrix

- < A simple algorithm >

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

We will not know exactly where to place element <j, i, value> in the transpose until we have processed all the elements that precede it.

For instance,

(0, 0, 15) becomes (0, 0, 15)
(0, 3, 22) becomes (3, 0, 22)
(0, 5,-15) becomes (5, 0,-15)

Consecutive insertions are required.
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.

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

- **[Program 2.8]**

```
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)  {                /*  nonzero matrix  */
        currentb = 1;
        for (i=0; i<a[0].col; i++)  /*  transpose by 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++;
                }
        }
}
```

Time complexity :   O(*columns·elements*).
    If *elements* = O(*rows·columns*), then
        O(*columns·elements*) becomes O(*rows·columns²*).

- A transpose algorithm using dense representation :

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

Time complexity : O(*rows·columns*).

- <A much better algorithm by using a little more storage>

This algorithm, *fast_transpose*, proceeds by first determining the number of elements in each column of the original matrix.

This number gives the number of elements in each row of the transpose matrix.

	0	1	2	3	4	5
row-terms	2	1	2	2	0	1
starting_pos	1	3	4	6	8	8

- **[Program 2.9]**

```
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;
            }
    }
}
```

	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

	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

- Time complexity : O(*columns + elements*).
  If *elements* = O(*rows·columns*), then
  O(*columns +elements*) becomes O(*rows·columns*).

- Additional arrays, *row_terms* and *starting_pos*, are used.

- We can reduce this space to one array
  if we put the starting positions into the space used by *row_terms*.

2.5.4 Matrix Multiplication

- **Definition :**
  Given two matrices 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 $0 \le j < p$.

- <Matrix Multiplication Algorithm using dense representation>

```
for (i = 0; i < rows_a; i++) {
      for (j = 0; j < cols_b; j++) {
            sum = 0;
            for (k = 0; k < cols_a; k++)
                  sum += a[i][k]*b[k][j];
            d[i][j] = sum;
      }
}
```

Time Complexity : O($rows\_a \cdot cols\_a \cdot cols\_b$)

Note that the product of two sparse matrices may no longer be sparse.
For example :

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

- <Multiplying two sparse matrices represented as an ordered list>

  Need to compute the elements of D by rows so that we can store them in their proper place without moving previously computed elements.

A

rows_a	cols_a	totala
rows_a		

B

rows_b	cols_b	totalb

B^T

cols_b	rows_b	totalb
cols_b	-1	

D

rows_a	cols_b	totald

- **[Program 2.10]**

Matrices A, B, and D are stored in the arrays *a, b*, and *d*, respectively.
Transpose of B is stored in *new_b*.

Variables used :

row - the row of A that we are currently multiplying with the columns in B.

row_begin - the position in *a* of the first element of the current row.

column - the column of B that we are currently multiplying with a row in A.

totald - the current number of elements in the product matrix D.

i, j - pointers which are used to examine successively elements from a row of A and a column 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;
  term  new_b[MAX_TERMS];
  if (col_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 = -1;
```

	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

	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


```
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 : /* go 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 = row_a;  d[0].col = cols_b;
d[0].value = totald;
}
```


A =

7	0	0	5
4	5	0	1
0	0	5	0

B =

0	1	0
9	9	7
0	0	1
1	0	0

D =

5	7	0
46	49	35
0	0	5

A

3	4	6
0	0	7
0	3	5
1	0	4
1	1	5
1	3	1
2	2	5
3	0	0

B^T

3	4	6
0	1	9
0	3	1
1	0	1
1	1	9
2	1	7
2	2	1
3	-1	0

D

0	0	0
0	0	5
0	1	7
1	0	46
1	1	49

row = 0   column = 0   rowBegin = 1   sum = 0

- Notice that we have introduced an additional term into both *a* and *new_b* :

```
a[totala+1].row = rows_a;
new_b[totalb+1].row = cols_b;
new_b[totalb+1].col = -1;
```

- Time complexity :

lines before the *for* loop:

fast transpose - O(*cols_b + totalb*) time.

the outer *for* loop is iterated *totala* times:

at each iteration - one row of the product matrix D is computed

by the inner *for* loop in which at each iteration

either *i* or *j* or both increase by 1,

or *i* is reset to *row_begin*.

The maximum total increment in *j* is *totalb*+1.
Let r_k be the number of terms in row *k*.
Then when row *k* is processed, *i* can increase at most r_k times
and *i* is reset to *row_begin* at most *cols_b* times.
Thus the maximum total increment in *i* is *cols_b*$\cdot r_k$.
The inner *for* loop requires O(*cols_b*$\cdot r_k$+ *totalb*) time.
column is reset.
Therefore the outer *for* loop requires

$$\sum_{k=0}^{rows\_a-1} O(cols\_b \cdot r_k + totalb)$$

$$= O(cols\_b \cdot \sum_{k=0}^{rows\_a-1} r_k + rows\_a \cdot totalb)$$

$$= O(cols\_b \cdot totala + rows\_a \cdot totalb).$$

Note that if *totala* = O(*rows_a*$\cdot$*cols_a*) and *totalb*=O(*rows_b*$\cdot$*cols_b*) and cols_a == rows_b, its complexity becomes O(*rows_a*$\cdot$*cols_a*$\cdot$*cols_b*).

2.6 REPRESENTATION OF MULTIDIMENSIONAL ARRAYS

- If an array is declared a[$upper_0$][$upper_1$]⋯[$upper_{n-1}$],

  the number of elements in the array is $\prod_{i=0}^{n-1} upper_i$

- Two common ways to represent multidimensional arrays :
  row major order
  column major order

- <row major order>
  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 $upper_1$ elements.

  If we assume that α is the address of a[0][0],
  then the address of a[i][j] is α + i·$upper_1$ + 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 $upper_1 \times upper_2$.
  Then the address of a[i][j][k] is
  α + i·$upper_1$·$upper_2$ + j·$upper_2$ + k.

- ***column major order address:***
  - $a + j \cdot upper_0 + i$
  - $a + k \cdot upper_0 \cdot upper_1 + j \cdot upper_0 + i$
-

- Generalizing on the preceding discussion, we can obtain the addressing formula for any element $a[i_0][i_1]\cdots[i_{n-1}]$ in an array declared as $a[upper_0][upper_1]\cdots[upper_{n-1}]$.

  If α is the address of $a[0][0] \ldots [0]$, the address of $a[i_0][i_1]\cdots[i_{n-1}]$ is :

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

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

- Example:
  - Given $a[upper_0][upper_1][upper_2][upper_3]$ array, the address of a[i][j][k][m] is

$row\ major\ order\ address\!:\ a + i \cdot upper_1 \cdot upper_2 \cdot upper_3 + j \cdot upper_2 \cdot upper_3 + k \cdot upper_3 + m$

$column\ major\ order\ address\!:\ a + m \cdot upper_0 \cdot upper_1 \cdot upper_2 + k \cdot upper_0 \cdot upper_1 + j \cdot upper_0 + i$

2.7 STRINGS

2.7.1 The Abstract Data Type

A *string* is a finite sequence of zero or more characters,

$S = s_0, \ldots, s_{n-1}$, where s_i are characters.

- **[ADT2.4] Abstract data type *String* :**

ADT *String* is
objects : a finite sequence 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,…. i+j-1.
else return *NULL*.

C provides many string operations in its library : see Fig. 2.8 (string.h)

2.7.2 Strings in C

■<Representation>

In C, we represent strings as character arrays terminated with the null character .

For instance,

```
#define MAX_SIZE 100
        char s[MAX_SIZE] = "dog";
        char t[MAX_SIZE] = "house";
```

Internal representation in C :

[Figure 2.8]

s[0]	s[1]	s[2]	s[3]
d	o	g	\0

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]
h	o	u	s	e	\0

- Alternative declaration :

  char s[] = "dog";

  char t[] = "house";

Concatenating these two strings by calling *strcat*(s,t) which stores the result in *s*. This produces the new string, "doghouse".

Although *s* has increased in length by five,
we have no additional space in *s* to store the extra five characters.

Most of *C* compilers simply *overwrite* the memory
to fit in the extra five characters.

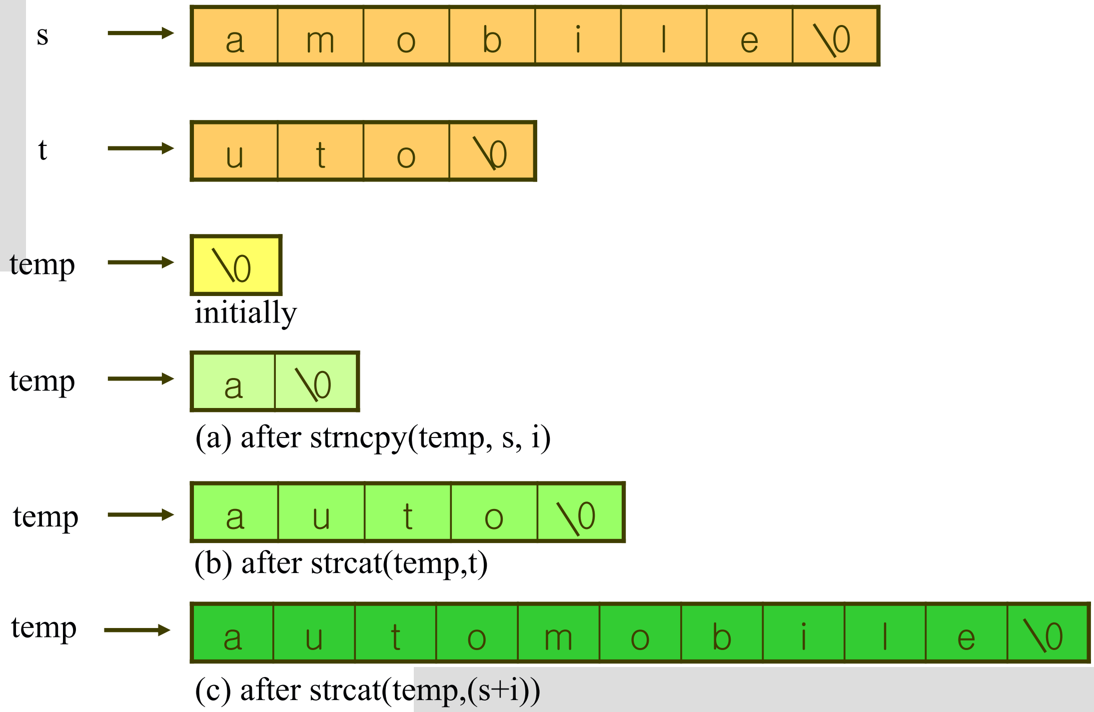
- C provides built-in other string functions which we access through the statement

  #include <string.h>

- **Example 2.2 [String insertion]**

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

strnins (*s*, *t*, 1)



- **[Program 2.12]**

```
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))  {
          fprint(stderr, "Position is out of bounds ");
          exit(1);
      }
      if (!strlen(s))
            strcpy(s, t);
      else if (strlen(t))  {
           strncpy(temp, s, i);
           strcat(temp, t);
           strcat(temp, (s+i));
           strcpy(s, temp);
      }
}
```

2.7.3 Pattern Matching

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

To determine if *pat* is in *string* :

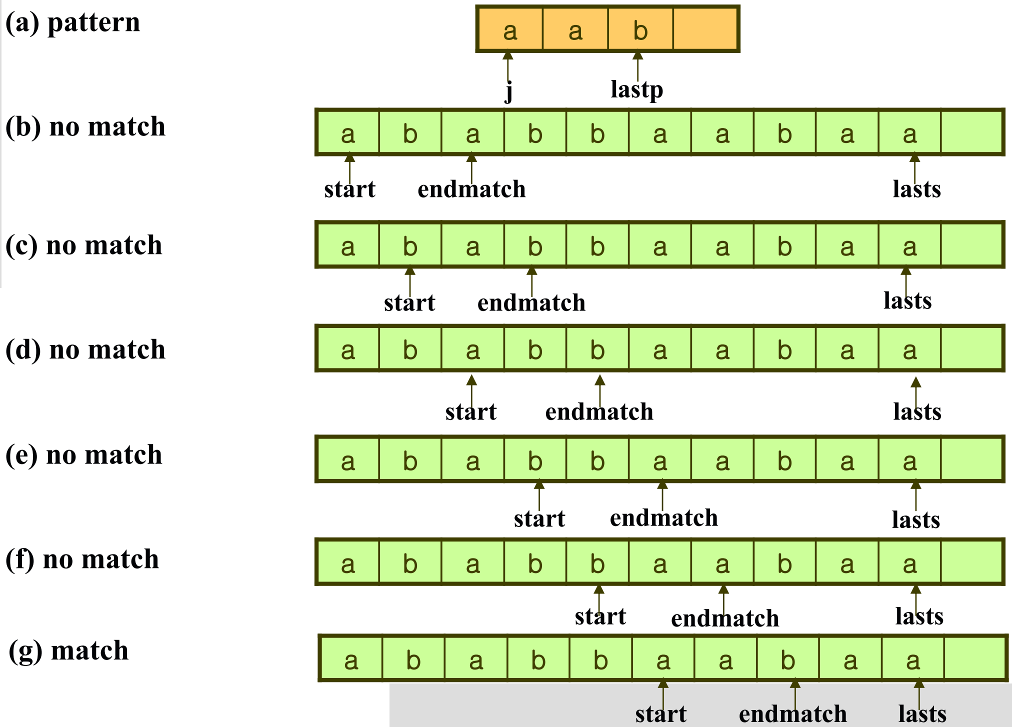
```
    if (t = strstr(string, pat))
        printf("The string from strstr is: %s", t);
    else
        printf("The pattern was not found with strstr");
```

The call (*t = strstr(string, pat)*) returns

a null pointer if *pat* is not in *string*.

a pointer to the start of *pat* in *string* if *pat* is in *string*.

- Reasons of developing our own pattern matching function:
  (1) The function *strstr* may not be available with the compiler we are using.
  (2) There are several different methods
  for implementing a pattern matching function.
- A simple matching algorithm :
  At each position *i* of *string*, check if *pat* == *string*[*i*+*strlen*(*pat*)-1].
- If *pat* is not in *string*, this algorithm has a computing time of O(*nm*),
  where *n* is the length of *pat* and *m* is the length of *string*.
- Improvements :
  1. Quitting when *strlen*(*pat*) is greater than the number of remaining characters in the string.
  2. Checking the first and last characters of pat before we checking the remaining characters.



[Program 2.13]

```
int nfind(chaqr * string, char *pat)
{
/*  match the last character of the 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;
 }
```

- Analysis of *nfind* :

For *string* = "*aa…a*" and *pat* = "*aa…ab*", the computing time is O(*m*).

Bur for *string* = "*aa…a*" and *pat* = "*aa…aba*",

the computing time is still O(*nm*).

- **<KMP Algorithm>**
  - When a mismatch occurs, 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.
  - We want to search the string for the pattern without moving backwards in the string

pat = 'a b c a b c a c a b'

p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9

if $s_i \neq p_0$, ?

if $s_i = p_0$ and $s_{i+1} \neq p_1$, ?

if $s_i = p_0$, $s_{i+1} = p_1$, and $s_{i+2} \neq p_2$, ?

if $s_i = p_0$, $s_{i+1} = p_1$, $s_{i+2} = p_2$, $s_{i+3} = p_3$, and $s_{i+4} \neq p_4$, ?

- **Definition :**
  If $p = p_0p_1p_2 \; . \; . \; . \; p_{n-1}$ is a pattern, then its *failure function, f,* is defined as :

$$f(j) = \begin{cases} \text{largest } i<j \text{ such that } p_0p_1 \ldots p_i = p_{j-i}p_{j-i+1} \ldots p_j & \text{if such an } i \geq 0 \text{ exists} \\ -1 & \text{otherwise} \end{cases}$$

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

- A rule for pattern matching :
  If a partial match is found such that $s_{i-j} \ldots s_{i-1} = p_0p_1 \ldots p_{j-1}$ *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.

How KMP Algorithm works

string	d	e	a	b	c	d	e	f	g	h	a	b	c	d	e	f	g	h	…	…

pattern	a	b	c	d	e	f	g	h	a	b	c	d	x	f	g
failure	-1	-1	-1	-1	-1	-1	-1	-1	0	1	2	3	-1	-1	-1

When (pattern[12] == x) != (string[14] == e),
First, compute j = failure[11] +1 = 3 + 1 = 4.
Then, start to compare (pattern[4] == e) with (string[14] == e)

string	a	b	a	b	b	a	a	b	a	a
pattern	a	a	b	a	a					
fail	-1	0	-1	0	1					

string	a	b	a	b	b	a	a	b	a	a
pattern	a	a	b	a	a					

string	A	C	G	G	G	G	G	T	G	C
pattern	G	G	G	G	T	G				
fail	-1	0	1	2	-1	0				

string	A	C	G	G	G	G	G	T	G	C
pattern	G	G	G	G	T	G				

- Assumed declarations:

```
#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];
```


- **[Program 2.14]**

```
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);
}
```

- Analysis of *pmatch* :

  The *while* loop is iterated until the end of either the string or the pattern is reached.

  In each iteration, one of the following three actions occurs :

  1) increment i.
  2) increment both i and j.
  3) reset j to failure[j-1]+1

     -- this cannot be done more than j is incremented by the statement j++ as otherwise, j falls off the pattern.

  Note that j cannot be incremented more than m = *strlen*(*string*) times.

  Hence the complexity of *pmatch* is O(m).

- **<u>Another definition of the failure function</u>**:

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

- **[program 2.15]**

```
void fail(char *pat)
{
/*  compute the pattern's failure function  */
   int i, 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;
   }
}
```


	i								j
0	1	2	3	4	5	6	7	8	9
a	b	c	a	b	c	a	c	a	b
-1	-1	-1	0	1	2	3	-1	0	1

i					j				
0	1	2	3	4	5				
g	g	g	g	t	g				
-1	0	1	2	-1	0				

				i						j
0	1	2	3	4	5	6	7	8	9	10
a	a	b	a	a	c	a	a	b	a	a
-1	0	-1	0	1	-1	0	1	2	3	4

- 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

or it is reset to a value 1 greater than

its terminal value on the previous iteration.

Since the *for* loop is iterated only *n*-1 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

Hence the complexity of *fail* is O(*n*) = O(*strlen*(*pat*))