資料結構
DATA STRUCTURE

Chap.02 Arrays and Structures

Ming-Han Tsai

2022 Fall

陣列(Array)

- 陣列(array)由一連串的索引(index)和值(value)構成
- 對每個索引index，必有一個相關聯的值value

- 如果可能的話，盡量使用連續空間的記憶體
  - 減少存取時間

Abstract Data Type *Array*

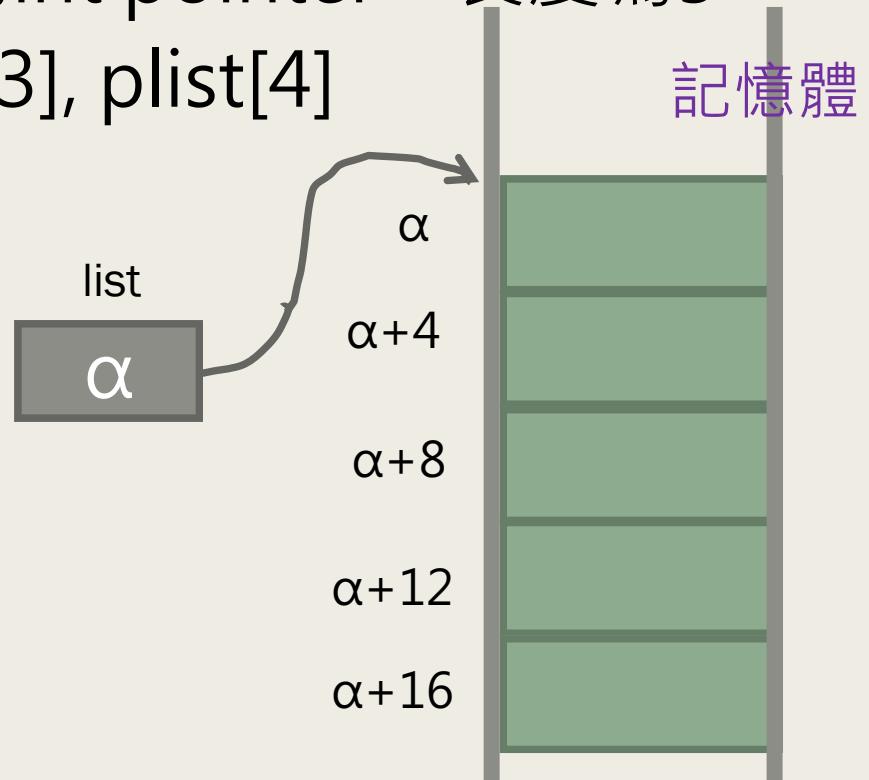
- **objects**：A set of pairs <index, value> where for each value of index there is a value from the set item. Index is a finite ordered set of one or more dimensions, for example, {0, … , n-1} for one dimension, {(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)} for two dimensions, etc.
- **Functions**：for all ***A*** ∈ 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 ith element is the size of the ith dimension. Items are undefined.
  - *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 in index) **return** an array that is identical to array A except the new pair <i, x> has been inserted
    else return error

Arrays in C

- int list[5], *plist[5];
- list[5]:　　名為list的陣列，類型為int，長度為5
- 　　list[0], list[1], list[2], list[3], list[4]
- *plist[5]:　名為plist的陣列，類型為int pointer，長度為5
- 　　plist[0], plist[1], plist[2], plist[3], plist[4]

Implementation of 1-D array’

- list[0]　　base address = α
- list[1]　　α + sizeof(int)
- list[2]　　α + 2*sizeof(int)
- list[3]　　α + 3*sizeof(int)
- list[4]　　α + 4*size(int)



Arrays in C

- 在C語言中，[]和* 其實有類似的意義
- int *list1 和 int list2[5]

  相同處:　　list1 和 list2 都是指標(pointer)

  不同處:　　list2 保留了五個整數的空間

- 標記:

  list2　　:　a pointer to list2[0]

  (list2 + i)　　:　a pointer to list2[i]　　(&list2[i])

  *(list2 + i)　　:　list2[i]

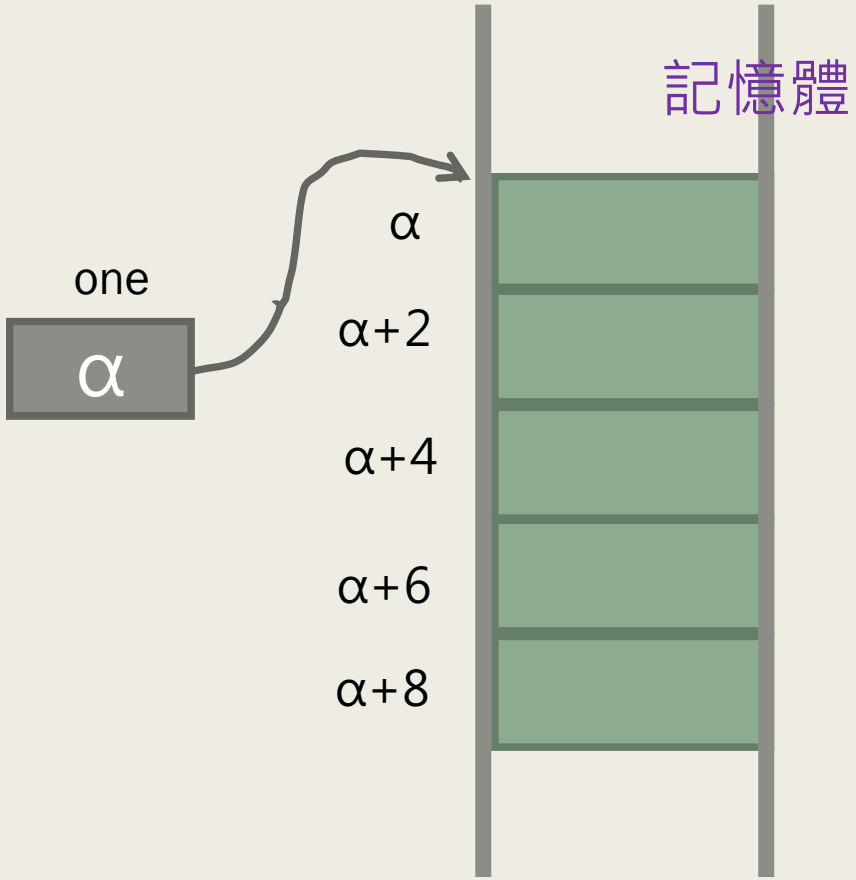
&：取址(address)

*：取值(value)

Example: 1-dimension array addressing

int one[] = {0, 1, 2, 3, 4};
Goal: print out address and value

```
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");
}
```



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

Address	Contents
1228	0
1230	1
1232	2
1234	3
1236	4

Structures (records)

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

- strcpy(person.name, “james”);
- person.age=10;
- person.salary=35000;

只宣告了一筆叫person的紀錄，型態為struct
無法再利用

Create structure data type

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

or

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

```
human_being person1, person2;
```

宣告了一個叫human_being的資料型態
宣告了兩筆叫person1、person2的紀錄，型態為human_being

Unions

- 類似於struct，但裡面所有資料共用記憶體空間，永遠只會有一個欄位有值(active)
- Example: Add fields for male and female.

```
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;
    }
```

← enum：列舉，表示可以用female/male代替0/1

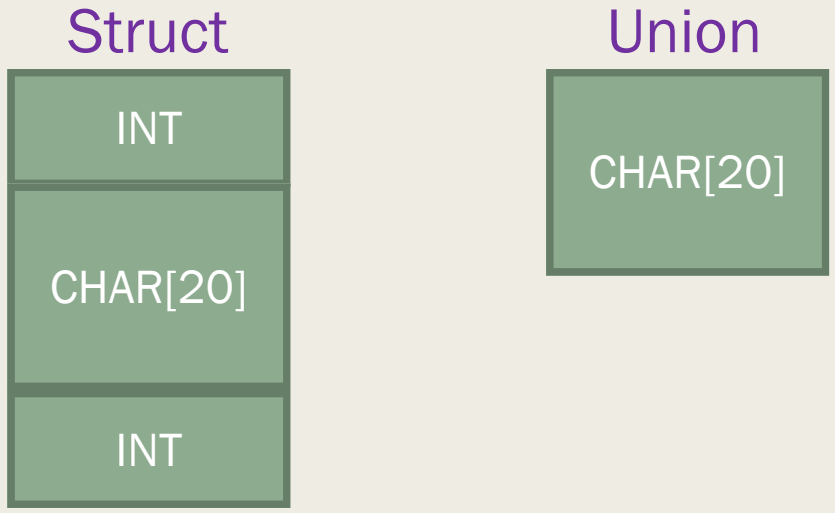
```
human_being person1, person2;
person1.sex_info.sex=male;
person1.sex_info.u.beard=FALSE;
```

print(person1.sex_info.u.children); ←無意義

Array/Struct/Union/Enum

- Array：連續記憶體空間，同一類型的資料
- Struct：不同類型的資料組合而成的一種自定義的資料型態，每個資料佔一個記憶體 = 同時都可以使用
- Union：不同類型的資料組合而成的一種自定義的資料型態，但多筆資料共用同一小塊記憶體 = 同時只有一個資料能用
- Enum：列舉代號讓程式更簡潔

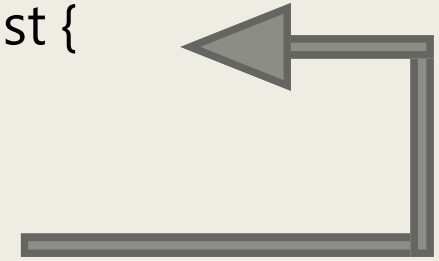
```
struct human {
    enum{female,male} sex;
    char[20] name;
    int name;
    };
```

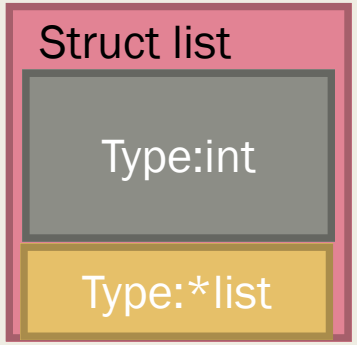


Self-Referential Structures

- 結構(struct)中包含指向自己類型的指標(pointer)

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

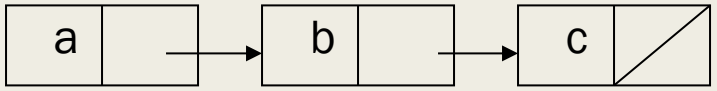





```
list item1, item2, item3;
item1.data=   'a'   ;
item2.data=   'b'   ;
item3.data=   'c'   ;
item1.link=item2.link=item3.link=NULL;

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

Construct a list with three nodes
item1.link=&item2;
item2.link=&item3;
malloc: obtain a node



Ordered List Examples

- 考慮某些情況下，我們需要序列：(item1,item2,item3,...,itemN)

- (水星、金星、地球、火星、木星、土星、天王星、海王星)
- (2,3,4,5,6,7,8,9,10,J,Q,K,A)
- (2010,2011,2012,...2020)

Ordered List的常見操作

- 找出List的長度n
- 從左到右(或是從右到左)循序讀取每一筆資料
- 擷取第i筆資料
- 儲存新的資料在i位置(修改)
- 插入新的資料在i位置，使得i以後的元素a_i, a_{i+1},a_{i+2},...,a_n 變成a_{i+1},a_{i+2},...,a_{n+1}(新增)
- 刪除某一筆位於i的資料，使得i以後的元素a_i, a_{i+1},a_{i+2},...,a_n 變成a_{i-1},a_{i-2},...,a_{n-1}

Think：Array辦的到嗎？　(1) ~ (4) Yes　(5)~(6) No

Recall：

- Ex: 建立一個可以儲存紀錄全班每個人成績的程式

```
int s9885123 = 90;
int s9885124 = 88;
int s9001245 = 79;
```

```
int sg[1000000];
sg[9885123] = 90;
sg[9885124] = 88;
sg[9001245] = 79;
```

```
Student A[40];
A[0].ID = 9885123;
A[0].grade = 90;
...
```

多項式表示法

- 思考：多項式表示法&加法&乘法
- F(x) = x^100 + x^10 + 4x + 1
- G(x) = x^35 + x^34 + x^2+24

- F(x) + G(x) = ?
- F(x) * G(x) = ?
- 怎麼用程式表示？

ADT Polynomial是

物件： $p(x) = a_1x^{e1} + \ldots + a_nx^{en}$; 一個成對序集 $<e_i, a_i>$，其中 a_i 屬於 Coefficients，e_i 屬於Exponents，e_i是大於或等於零的整數

函式：

對於所有的poly, poly1, poly2 ∈Polynomial，coef ∈Coefficients，expon∈Exponents

Polynomial Zero()	::=	return 多項式，p(x) = 0
Boolean IsZero(poly)	::=	if(poly) return FALSE　　else return TRUE
Coefficient Coef(poly,expon)	::=	if(expon ∈ poly) return它的係數 else return 0
Exponent LeadExp(poly)	::=	return poly的最高次方
Polynomial Attach(poly,coef,expon)	::=	if(expon ∈ poly) return錯誤 else return 插入<coef,expon>之後的多項式 poly
Polynomial Remove(poly,expon)	::=	if(expon ∈ poly) return刪除次方是expon的項後的多項式poly else return錯誤
Polynomial SingleMult(poly,coef,expon)	::=	return 多項式 poly．coef．x^{expon}
Polynomial Add(poly1,poly2)	::=	return 多項式 poly1 + poly2
Polynomial Mult(poly1,poly2)	::=	return 多項式 poly1．poly2

多項式加法

- F(x) = x^100 + x^10 + 4x + 1
- G(x) = x^35 + x^34 + x^2+24

- F(x) + G(x) = ?
- 怎麼用程式表示？

Polynomial Addition

```
/* 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));
  }
}
```

if (*expon* ∈ *poly*) **return** error
else return the polynomial *poly* with the term <*coef*, *expon*> inserted

if (*expon* ∈ *poly*)
return the polynomial *poly* with the term whose exponent is *expon* deleted
else return error

advantage: easy implementation
disadvantage: waste space when sparse

- *insert any remaining terms of a or b into d*
- ***Program 2.4 :**Initial version of padd function(p.62)*

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

$D(x) = A(x) + B(x)$

```
d = Zero( ) ; // Let d is a Null polynomial
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));
   }
}
```

A(x) = ZERO

B(x) = ZERO

Typedef polynormial：

- 陣列表示
- Struct 表示法1：

```
#define MAX_DEGREE 101

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

$a.degree = n$

$a.coef[i] = a_{n-1}, 0 \leq \mathrm{i} \leq \mathrm{n}$

缺點：這個表示法很浪費空間!!

Typedef polynormial：

- Struct 表示法2：

```
typedef struct{
    float coef;
    int expon;
    } polynomial;

polynomial terms[MAX_TERMS];
int avail = 0;
```

$avail$：*有用到的項的數目*

$terms[i].expon = p$

$terms[i].coef = a_i$ for $a_i x^p$

Typedef polynormial：

- Struct 表示法2：

```
typedef struct{
    float coef;
    int expon;
    } polynomial;

polynomial terms[MAX_TERMS];
int avail = 0;
```

$avail$：*有用到的項的數目*

$terms[i].expon = p$

$terms[i].coef = a_i$ for $a_i x^p$

Typedef polynormial：

- 用單一個大陣列儲存所有的多項式

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

specification	representation
poly	<start, finish>
A	<0,1>
B	<2,5>

storage requirements: start, finish, 2*(finish-start+1)

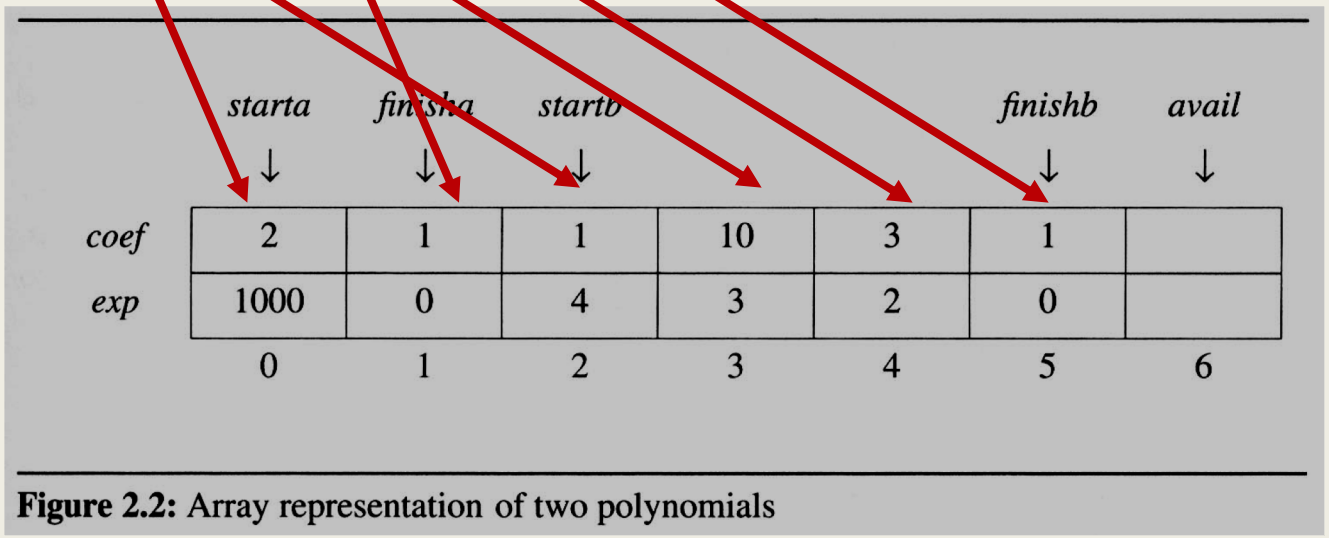


Figure 2.2: Array representation of two polynomials

	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

- 如何用C語言表示一個能將AB相加的函數padd並將結果儲存在D：$D = A + B$.
  - *To produce D(x), padd (Program 2.5) adds A(x) and B(x) term by term.*

```
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

Analysis: $O(n+m)$
where n (m) is the number of nonzeros in A (B).

```
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

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

$B(x) = x4+10x3+3x2+1$

starta = 2 finisha = 1

startb = 6 finishb = 5

starta <= finisha = FALSE

startb <= finishb = FALSE

Term

$2x^{1000}$
1
x^4
$10x^3$
$3x^2$
1
$2x^{1000}$
x^4
$10x^3$
$10x^2$
2

```
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;
}
```

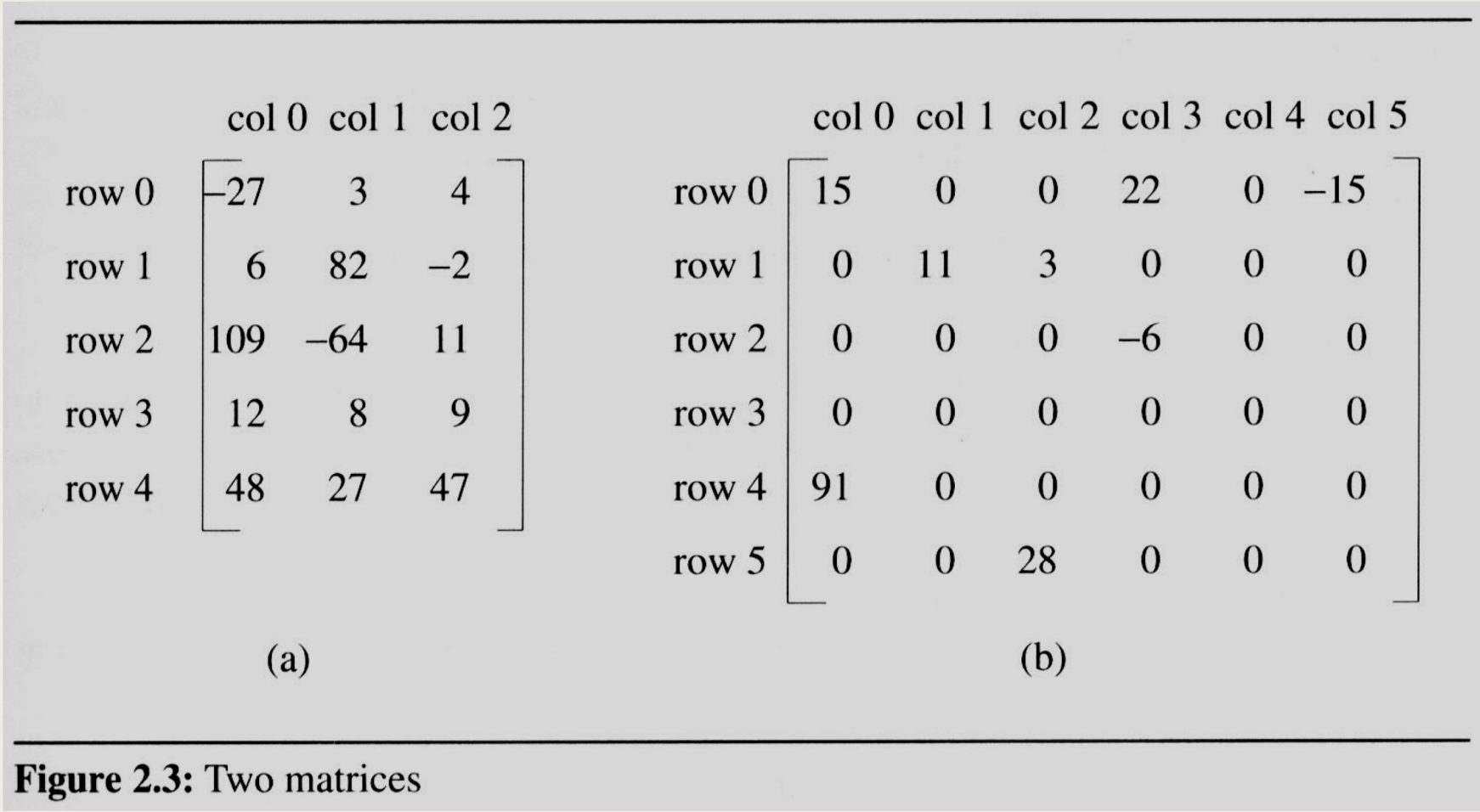
Program 2.6: Function to add a new term

問題：陣列大小是有限的，當多項式總項數超過上限MAX_TERMS的時候需要壓縮

移動資料需要時間

Sparce Matrix

- 思考：矩陣表示法&加法&乘法



	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 2.3: Two matrices

Sparse Matrix

- 數學上，矩陣表示由m列(m rows) n行(n columns)個元素構成的資料型態，我們以mxn表示一個矩陣具有m rows and n cloumns
- 稀疏矩陣(Sparse Matrix)：0很多的矩陣

	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 2.3: Two matrices

Sparse Matrix

- 最簡單的矩陣表示法是使用二維陣列(two dimensional array) defined as

  a[*MAX_ROWS*][*MAX_COLS*]

  - *簡單使用 a[i][j] 即可存取任一位置*
- 缺點：處理稀疏矩陣浪費空間
  - *Hint：只儲存非零的元素？*
  - *Each element is characterized by <row, col, value>.*

The Sparse Matrix ADT

- Structure 2.3 contains our specification of the matrix ADT.
  - *A minimal set of operations*
    - Matrix creation
    - Addition
    - Multiplication
    - 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* × *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*

The Sparse Matrix ADT

- 建立矩陣：Create
- 對每個原本陣列的元素a[i,j]=x，我們需要一個三元數(triple)去儲存(i,j,x)，因此：

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];
```

- 其中a[0].col = MAX_COL，a[0].row=MAX_ROW，a[0].value=NUM_OF_ELEMENTS

The Sparse Matrix ADT

- Figure 2.4(a) shows how the sparse matrix of Figure 2.3(b) is represented in the array *a[MAX_TERMS]*.
  - *Each element is characterized by <row, col, value>.*

	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)

row, column in ascending order

# of rows (columns) # of nonzero terms

	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)

transpose

	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

轉置矩陣Transpose

- Transpose a Matrix
  - *For each row i*
    - take element <i, j, value> and store it in element <j, i, value> of the transpose.
    - difficulty: where to put <j, i, value>
      (0, 0, 15) ====> (0, 0, 15)
      (0, 3, 22) ====> (3, 0, 22)
      (0, 5, -15) ====> (5, 0, -15)
      (1, 1, 11) ====> (1, 1, 11)
      Move elements down very often.
  - *For all elements in column j,*
    - place element <i, j, value> in element <j, i, value>

2.4 The sparse matrix ADT (7/18)

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

Assign
A[i][j] to B[j][i]

place element <i, j, value>
in element <j, i, value>

For all columns i
For all elements in column j

Scan the array "columns" times.
The array has "elements" elements.

```
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++;
            }
   }
}
```

==> O(columns*elements)

Program 2.7: Transpose of a sparse matrix

EX: A[6][6] transpose to B[6][6]

i=1 j=8
a[i].col = 2 != i

Matrix 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

Row Col Value

0	6	6	8
1	0	0	15
2	0	4	91
3	1	1	11

```
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++;
          }
   }
}
```

Set Up row & column in B[6][6]

And So on...

Program 2.7: Transpose of a sparse matrix

2.4 The sparse matrix ADT (8/18)

- 和二維陣列表示法比較
  - *O(columns*elements)* vs. *O(columns*rows)*
  - *elements --> columns * rows when non-sparse, O(columns^2*rows)*
- 問題：需要掃描整個陣列 “columns” 次.
  - *In fact, we can transpose a matrix represented as a sequence of triples in O(columns + elements) time.*
- 解決方法：
  - *先計算每個column裡面到底有多少元素*
  - *接著找出轉置矩陣中，每一個元素的新的起始位置*

  (根據該行到底有幾個元素)

2.4 The sparse matrix ADT (9/18)

- Compared with 2-D array representation:
  O(columns+elements) vs. O(columns*rows)
  elements --> columns * rows O(columns*rows)

Cost:
Additional row_terms and starting_pos arrays are required.
Let the two arrays row_terms and starting_pos be shared.

For columns

For elements

Buildup row_term & starting_pos

For columns

transpose For elements

```
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;
      }
   }
}
```

Program 2.8: Fast transpose of a sparse matrix

2.4 The sparse matrix ADT (10/18)

- After the execution of the third **for** loop, the values of *row_terms* and *starting_pos* are:

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	1	3	4	6	8	8

	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)

transpose →

	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

Matrix 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

	[0]	[1]	[2]	[3]	[4]	[5]	
row_terms	2	1	2	2	0	1	#col = 6
starting_pos	1	3	4	6	8	8	#term = 6

```
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;
      }
   }
}
```

Program 2.8: Fast transpose of a sparse matrix

I = 8

Matrix 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

	Row	Col	Value
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

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	3	4	6	8	8	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;
      }
   }
}
```

Program 2.8: Fast transpose of a sparse matrix

2.4 The sparse matrix ADT (11/18)

- 2.4.3 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 $\langle i, j \rangle$ element is

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

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

2.4 The sparse matrix ADT (12/18)

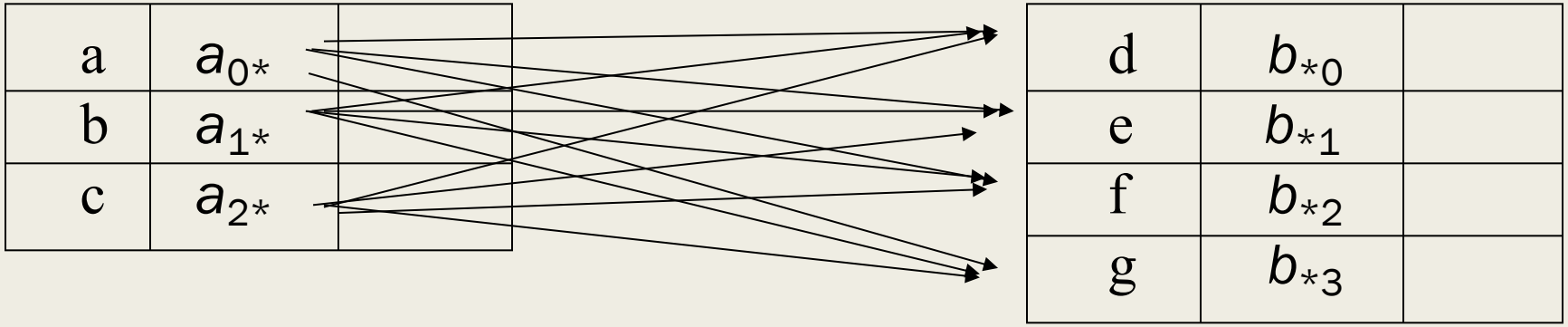
- Sparse Matrix Multiplication
  - *Definition:* $[D]_{m*p}=[A]_{m*n}* [B]_{n*p}$
  - *Procedure: Fix a row of A and find all elements in column j of B for j=0, 1, …, p-1.*
  - *Alternative 1.*
    Scan all of B to find all elements in j.
  - *Alternative 2.*
    Compute the transpose of B.
    (Put all column elements consecutively)
    - 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 2.2

2.4 The sparse matrix ADT (13/18)

- General case:

$$d_{ij}=a_{i0}*b_{0j}+a_{i1}*b_{1j}+\ldots+a_{i(n\text{-}1)}*b_{(n\text{-}1)j}$$

  - *Array A row i * Array J col j*
  - *可以想成Array row j * Array J轉置 row j*



The multiply operation generate entries:
a*d , a*e , a*f , a*g , b*d , b*e , b*f , b*g , c*d , c*e , c*f , c*g

The sparse matrix ADT (14/18)

- An Example

$$A = \begin{bmatrix} 1 & 0 & 2 \\ -1 & 4 & 6 \end{bmatrix} \quad B^T = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 0 & 2 \\ -1 & 0 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

	row	col	value
a[0]	2	3	5
[1]	0	0	1
[2]	0	2	2
[3]	1	0	-1
[4]	1	1	4
[5]	1	2	6

	row	col	value
b_t[0]	3	3	4
b_t[1]	0	0	3
b_t[2]	0	1	-1
b_t[3]	2	0	2
b_t[4]	2	2	5

	row	col	value
b[0]	3	3	4
b[1]	0	0	3
b[2]	0	2	2
b[3]	1	0	-1
b[4]	2	2	5

- An Example

$$A = \begin{matrix} 1 & 0 & 2 \\ -1 & 4 & 6 \end{matrix} \qquad B^T = \begin{matrix} 3 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 5 \end{matrix} \qquad B = \begin{matrix} 3 & 0 & 2 \\ -1 & 0 & 0 \\ 0 & 0 & 5 \end{matrix}$$

	row	col	value		row	col	value
a[0]	2	3	5	b_t[0]	3	3	4
[1]	0	0	1	b_t[1]	0	0	3
[2]	0	2	2	b_t[2]	0	1	-1
[3]	1	0	-1	b_t[3]	2	0	2
[4]	1	1	4	b_t[4]	2	2	5
[5]	1	2	6				

	row	col	value		row	col	value		row	col	value
a[0]	2	3	5	b_t[0]	3	3	4	b[0]	3	3	4
[1]	0	0	1	b_t[1]	0	0	3	b[1]	0	0	3
[2]	0	2	2	b_t[2]	0	1	-1	b[2]	0	2	2
[3]	1	0	-1	b_t[3]	2	0	2	b[3]	1	0	-1
[4]	1	1	4	b_t[4]	2	2	5	b[4]	2	2	5
[5]	1	2	6	[5]	3	0					
[6]	2										

```
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;
```

Totala = 5

Totalb = 4

Totald = 0

rows_a = 2

cols_a = 3

cols_b = 3

row_begin = 1

row = 0

A裡面的元素個數
B裡面的元素個數
D裡面的元素個數
row_a = 2 a的row數
cols_a = 3 a的cols數
cols_b = 3 b的cols數
目前要開始處理的row
是存在陣列第幾個位置(index)
目前要處理第幾行row

Totala = 5 rows_a = 2
Totalb = 4 cols_a = 3
Totald = 0 cols_b = 3

row_begin = 1
row = 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

Variable	Value
i	1 2 3
j	1 2 3 4
column	0 1 2
row	0 1
sum	0 12
row_begin	1 3

d[1] 0 0 3
d[2] 0 2 12

6

a[0]	2	3	5
[1]	0	0	1
[2]	0	2	2
[3]	1	0	-1
[4]	1	1	4
[5]	1	2	6
[6]	2		
b_t[0]	3	3	4
b_t[1]	0	0	3
b_t[2]	0	1	-1
b_t[3]	2	0	2
b_t[4]	2	2	5
b_t[5]	3	0	

A[0][0]*B[0][0]

A[0][0]*B[0][2]

A[0][2]*B[2][2]

And So on…

2.4 The sparse matrix ADT (15/18)

- 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;
}
```

a × b

Program 2.9: Sparse matrix multiplication

2.4 The sparse matrix ADT (16/18)

```
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

Analyzing the algorithm

- cols_b * termsrow1 + totalb +
  cols_b * termsrow2 + totalb +
  … +
  cols_b * termsrowp + totalb
  = cols_b * (termsrow1 + termsrow2 + … + termsrowp)+
  rows_a * totalb
  = cols_b * totala + row_a * totalb

  O(cols_b * totala + rows_a * totalb)

```
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

$$A = \begin{bmatrix} 1 & 0 & 2 \\ -1 & 4 & 6 \end{bmatrix} \quad B^T = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 0 & 2 \\ -1 & 0 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

	row	col	value
a[0]	2	3	5
[1]	0	0	1
[2]	0	2	2
[3]	1	0	-1
[4]	1	1	4
[5]	1	2	6

	row	col	value
b_t[0]	3	3	4
b_t[1]	0	0	3
b_t[2]	0	1	-1
b_t[3]	2	0	2
b_t[4]	2	2	5

	row	col	value
b[0]	3	3	4
b[1]	0	0	3
b[2]	0	2	2
b[3]	1	0	-1
b[4]	2	2	5

2.4 The sparse matrix ADT (18/18)

- Compared with matrix multiplication using array
  - *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;
    }
  - *O(rows_a * cols_a * cols_b)* *vs.*
    *O(cols_b * total_a + rows_a * total_b)*
  - *optimal case:*
    *total_a < rows_a * cols_a total_b < cols_a * cols_b*
  - *worse case:*
    *total_a --> rows_a * cols_a, or*
    *total_b --> cols_a * cols_b*

Practice

- 若陣列元素A(5,3)在記憶體位址是5314，而元素A(8,5)是在5422；假設每個元素占用四個位元組(Bytes)，請問元素A(2,7)的位址為？
- (1) A(8,5) = A(5,3) + [(8-5)*n + (5-3)] * 4R

  row-major

  5422 = 5314 + 12n + 8 => 100 = 12n n = 100/12 ...X

- (2) A(8,5) = A(5,3) + [(5-3)*m + (8-5)] * 4

  column-major

  5422 = 5314 + 8m + 12 => 96=8m m = 12

  A(2,7) = A(5,3) + [(7-3)*12 + (2-5)] * 4

  = 5314 + [48-3]*4 = 5314+180 = 5494