# ARRAYS AND STRUCTURES

2.1 Arrays

2.2 Dynamically Allocated Array

2.3 Structures and Unions

2.4 Polynomials

2.5 Sparse Matrices

2.6 Representation of Multidimensional Arrays

# 2.4.1 THE ABSTRACT DATA TYPE

# Ordered (linear) List

- An ordered set of data items
  - Ex)

    Days of the week: (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)

  - Denote as ($\text{item}_0$, $\text{item}_1$, $\text{item}_2$, …, $\text{item}_{n-1}$)
  - Empty list : ( )

- Operations on ordered list: p65
  - ....

We can perform many operations on lists, including:

- Finding the length, $n$, of a list. 원소들의 개수
- Reading the items in a list from left to right (or right to left). 거꾸로 읽기
- Retrieving the $i$th item from a list, $0 \leq i < n$. i번째 데이터 검색
- Replacing the item in the $i$th position of a list, $0 \leq i < n$. i번째 데이터 변경
- Inserting a new item in the $i$th position of a list, $0 \leq i \leq n$. The items previously numbered $i, i+1, \cdots, n-1$ become items numbered $i+1, i+2, \cdots, n$. i번째에 데이터 넣기
- Deleting an item from the $i$th position of a list, $0 \leq i < n$. The items numbered $i+1, \cdots, n-1$ become items numbered $i, i+1, \cdots, n-2$. i번째 데이터 삭제

# Implementation of Ordered List

- Array (sequential mapping)
  – Associate the list element, $item_i$, with the array index $i$

| | | | | | | |
|---|---|---|---|---|---|---|
| | 3 | 2 | 1 | 2 | 2 | 3 |
| Index → | 0 | 1 | 2 | 3 | 4 | 5 |

  – Retrieve/replace an item, or find the length of a list in a constant time
  – Problems in insertion and deletion : …

- Linked List (non-sequential mapping)
  – Chapter 4

# Application: Polynomials

- Ex)
  $A(x) = 3x^{20}+2x^{5}+4, \quad B(x) = x^{4}+10x^{3}+3x^{2}+1$

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 x^0$$

$ax^e$    a : coefficient $a_n \neq 0$
e : exponent – unique
x : variable x

- Assume    $A(x) = \Sigma a_i x^i$ and $B(x) = \Sigma b_i x^i$
  then    $A(x) + B(x) = \Sigma(a_i + b_i)x^i$
  $A(x) \cdot B(x) = \Sigma(a_i x^i \cdot \Sigma(b_j x^j))$

$A(x) = 3x^{20}+2x^5+4$

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

**ADT** *Polynomial* is

**objects**: $p(x) = a_1x^{e_1} + \cdots + a_nx^{e_n}$; a set of ordered pairs of $<e_i, a_i>$ where $a_i$ in *Coefficients* and $e_i$ in *Exponents*, $e_i$ are integers >= 0

**functions**:

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

| | | |
|---|---|---|
| *Polynomial* Zero() | ::= | **return** the polynomial, $p(x) = 0$ |
| *Boolean* IsZero(*poly*) | ::= | **if** (*poly*) **return** *FALSE* **else return** *TRUE* |
| *Coefficient* Coef(*poly*,*expon*) | ::= | **if** (*expon* ∈ *poly*) **return** its coefficient **else return** zero |
| *Exponent* LeadExp(*poly*) | ::= | **return** the largest exponent in *poly* |
| *Polynomial* Attach(*poly*, *coef*, *expon*) | ::= | **if** (*expon* ∈ *poly*) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted |
| *Polynomial* Remove(*poly*, *expon*) | ::= | **if** (*expon* ∈ *poly*) **return** the polynomial *poly* with the term whose exponent is *expon* deleted **else return** error |
| *Polynomial* SingleMult(*poly*, *coef*, *expon*) | ::= | **return** the polynomial $poly \cdot coef \cdot x^{expon}$ |
| *Polynomial* Add(*poly*1, *poly*2) | ::= | **return** the polynomial $poly1 + poly2$ |
| *Polynomial* Mult(*poly*1, *poly*2) | ::= | **return** the polynomial $poly1 \cdot poly2$ |

**end** *Polynomial*

**ADT 2.2:** Abstract data type *Polynomial*

## 2.4.2 POLYNOMIAL REPRESENTATION

# Polynomial Representation

- Representation (I)

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

※ **A = (n, $\underline{a_n, a_{n-1}, \ldots, a_1, a_0}$)**

degree of A        n+1 coefficients

ex) $A(x) = x^4+10x^3+3x^2+1$   : n = 4

A = (4, 1, 10, 3, 0, 1)   : 6 elements

```c
#define MAX_DEGREE 101
typedef struct  {
        int degree;
        float coef[MAX_DEGREE];
} polynomial;
polynomial a;
```

- $A(x) = \sum a_i x^i$,
  **a.degree** = n, **a.coef[i]** = $\mathbf{a_{n-i}}$, $0 \le i \le n$

- Ex)
  $A(x)=11x^8+5x^6+x^5+2x^4-3x^2+x+10$

| Index *i* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Coefficient | 11 | 0 | 5 | 1 | 2 | 0 | -3 | 1 | 10 |

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

- Problems: 이게 아주 작으면... 나머지 공간은 아까움
  - If a.degree << MAX_DEGREE: …
  - If $A(x)= x^{99}+78$: …

텀이 2개인데 나머지는 다 0

- Representation II

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

– Use one global array *term* to store all polynomials

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

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

| poly | <start, finish> |
|---|---|
| A | <0,1> |
| B | <2,5> |

| | *startA* ↓ | *finishA* ↓ | *startB* ↓ | | | *finishB* ↓ | *avail* ← 비어있는 인덱스 ↓ |
|---|---|---|---|---|---|---|---|
| *coef* | 2 | 1 | 1 | 10 | 3 | 1 | |
| *exp* | 1000 | 0 | 4 | 3 | 2 | 0 | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

**Figure 2.3:** Array representation of two polynomials

$$A(x) = 2x^{1000} + 2x^3$$

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

$$A(x)+B(x) = \ldots$$

`#define COMPARE(x, y) ( ((x) < (y)) ? -1 : ((x) == (y) ? 0: 1 ) )` ※ p.12

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

2, 1000이 d에 옮기고

$2x^{1000}$은 제거해줌

$A(x) = 2x^{1000} + 2x^3$

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

$A(x)+B(x) = \ldots$

Program 2.5: Initial version of *padd* function (representation-independent )

## 2.4.3 POLYNOMIAL ADDITION

# Polynomial Addition

| | 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 |

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

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

$D(x) =$

| | 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 |

시작 index 끝 index

b terms[2].coef = 1, terms[2].expon = 4

a terms[0].coef = 2, terms[0].expon = 1000

```
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, term[startb].expon )  ) {
2 3 4        case -1:   /* a expon < b expon */
                     attach(terms[startb].coef, terms[startb].expon);
                     startb++; break;
5            case 0:   /* equal exponents */
                     coefficient = terms[starta].coef + terms[startb].coef;
                     if(coefficient) attach(coefficient, terms[starta].expon);
                     starta++; startb++; break;
1            case 1:   /* a expon > b expon */
                     attach(terms[starta].coef, terms[starta].expon);
                     starta++;
        }
```

Ex)
$A(x) = 2x^{1000}+9x^{75}$
$B(x) = x^4+10x^3+3x^2+1$

```
  /* add in remaining terms of A(x) */
  for(; starta <= finisha; starta++)
        attach(terms[starta].coef, term[starta].expon);

  /* add in remaining terms of B(x) */
  for(; startb <= finishb; startb++)
        attach(terms[startb].coef, terms[startb].expon);

 *finishd = avail – 1;
}
```

A+B 한 텀의 가장 마지막 index

**Program 2.6**: Function to add 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 |

```
typedef struct {
        float coef;
        int expon;
 } polynomial;
 polynomial terms[MAX_TERMS];
```

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

2

1000

**Program 2.7:** Function to add a new term

- Analysis of *padd*
  - $m$, $n$: the number of nonzero terms in A and B
  - Time complexity:

    … $O(m+n)$
  - The worst case occurs when:

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

텀이 모두 다를때

(=지수가 모두 다를때)

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

**Program 2.6:** Function to add two polynomials

# ARRAYS AND STRUCTURES

2.1 Arrays

2.2 Dynamically Allocated Array

2.3 Structures and Unions

2.4 Polynomials

2.5 Sparse Matrices

2.6 Representation of Multidimensional Arrays

# 2.5.1 The Abstract Data Type

- m×n matrix
  - A matrix contains *m* rows and *n* columns of elements

→ 0이 spar많비..

- Sparse matrix: $\dfrac{\text{no. of non-zero elements}}{\text{no. of total elements}} \ll \text{small}$

Not sparse matrix

| | 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)

Sparse matrix

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

(b)

**Figure 2.4:** Two matrices

- The standard representation of a matrix
  - A two dimensional array: ***a*[MAX_ROWS][MAX_COLS]**
  - We can locate quickly any element by writing $a[i\ ][\ j\ ]$

- In case of a sparse matrix: …

- Ex) a 1000×1000 matrix with only 2000 non-zero elements;
  - Problem: … 남는 메모리 공간이 너무많음

**ADT** SparseMatrix is

**object**: a set of triples, *<row, column, value>*

**functions** :

sparseMatrix Create(maxRow, maxCol) ::=

sparseMatrix Transpose(a) ::=

sparseMatrix Add(a,b) ::=

sparseMatrix Multiply(a,b) ::=

**ADT 2.3:** Abstract data type *SparseMatrix*

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

# 2.5.2 Sparse Matrix Representation

- *Create* operation :

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

# of rows (columns)

# of nonzero entries

| | 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 |

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

(b)

# 2.5.3 Transposing A Matrix

a[1].row=0 a[1].col=0 a[1].value=15
a[2].row=0 a[2].col=3 a[2].value=22

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

(a)

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

(b)

**Figure 2.5:** Sparse matrix and its transpose stored as triples

a

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

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

- Ex)

  (0, 0, 15) → (0, 0, 15)

  (0, 3, 22) → (3, 0, 22)

  (0, 5, -15) → (5, 0, -15)

  (1, 1, 11) → (1, 1, 11)

  …

- Problem: …

| | 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)

| | | | |
|---|---|---|---|
| *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)

- Algorithm:

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

(a)

| | row | col | value |
|---|---|---|---|
| b[0] | 6 | 6 | 8 |
| [1] | | | |
| [2] | | | |
| [3] | | | |
| [4] | | | |
| [5] | | | |
| [6] | | | |
| [7] | | | |
| [8] | | | |

```
void transpose(term a[], term b[])
{/ * b is set to the transpose of a * /
      int n, i, j, currentb;
      n = a[0].value;
```

a[0].value에는 데이터의 개수가 들어있음

```
      b[0].row = a[0].col;
      b[0].col = a[0].row;
      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 ) {
                              b[currentb].row = a[j].col;
                              b[currentb].col = a[j].row;
                              b[currentb].value = a[j].value;
                              currentb++;
                        }
      }
}
```

| | 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 |

**Program 2.8:** Transpose of a sparse matrix

```
void transpose(term a[], term b[])
{/ * b is set to the transpose of a * /
    int n, i, j, currentb;
    n = a[0].value;
    b[0].row = a[0].col;
    b[0].col = a[0].row;
    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 ) {
                    b[currentb].row = a[j].col;
                    b[currentb].col = a[j].row;
                    b[currentb].value = a[j].value;
                    currentb++;
                }
    }
}
```

Time Complexity:
→…

Time Complexity (in non-sparcse):
→…

**Program 2.8:** Transpose of a sparse matrix

- If we represented our matrices as two-dimensional arrays of size *rows*×*columns,*

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

Time Complexity: ….

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

- Fast transpose

| | 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] | | | |
| [2] | | | |
| → [3] | | | |
| → [4] | | | |
| [5] | | | |
| → [6] | | | |
| [7] | | | |
| → [8] | | | |

1) calculation of rowTerms

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

2) calculation of startingPos

| | 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] | | | |
| [2] | | | |
| [3] | | | |
| [4] | | | |
| [5] | | | |
| [6] | | | |
| [7] | | | |
| [8] | | | |

b에 row갈 a의 col term이 몇개인지

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

```
for(i=1; i <= numTerms; i++)
        rowTerms[ a[i].col ]++;
```

| | 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] | | | |
| [2] | | | |
| → [3] | | | |
| → [4] | | | |
| [5] | | | |
| → [6] | | | |
| [7] | | | |
| → [8] | | | |

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

```
startingPos[0] = 1;

for(i=1;  i<numCols;  i++)
        startingPos[i]  =  startingPos[i-1] + rowTerms[i-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] | | | |
| [2] | | | |
| [3] | | | |
| [4] | | | |
| [5] | | | |
| [6] | | | |
| [7] | | | |
| [8] | | | |

→ 원래 term의 col

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

```
for(i=1;  i<=numTerms;  i++)  {
                    j = startingPos[a[i].col]++;
                    b[j].row    = a[i].col;
                    b[j].col    = a[i].row;
                    b[j].value = a[i].value;
}
```

```
void fasttranspose(term a[], term b[])   {
    int rowTerms[MAX_COL], startingPos[MAX_COL];
    int i, j, numCol = a[0].col,  numTerms = a[0].value;
    b[0].row = numCols;  b[0].col = a[0].row;      (= a[0].col
    b[0].value = numTerms;
    if (numTerms >0)   {
        for(i=0; i < numCols; i++)  rowTerms[i] = 0;
        for(i=1; i <= numTerms; i++)  rowTerms[a[i].col]++;
        startingPos[0] = 1;
        for(i=1;  i<numCols;  i++)                  startingPos[1] = 1+2=3
            startingPos[i]  =  startingPos[i-1] + rowTerms[i-1];
        for(i=1;  i<=numTerms;  i++)  {
            j = startingPos[a[i].col]++;            j=1
            b[j].row = a[i].col;   b[j].col = a[i].row;
            b[j].value = a[i].value;
        }
    }
}
```

**Program 2.9:** Fast transpose of a sparse matrix

a의 col의 data의 타입에따라 b의 row에 들어갈 데이터의 타입을 결정

| | 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 |

| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| *rowTerms* = | 2 | 1 | 2 | 2 | 0 | 1 |
| *startingPos* = | 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)

- Analysis of *fastTranspose*
  - Additional rowTerms and startingPos arrays are required
  - Computing time
    : O(...)
  - when # of elements == columns · rows (non-sparse)
    : O(...)
  - when the number of elements is sufficiently small
    : *fastTranspose* is much faster

$O(2 \cdot col + 2 \cdot elements)$

$= O(a+b)$

\# elements = columns · rows (non-sparse)

$= O(n^2)$

∴ sparse 일때만 효율적으로 동작함

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

**Program 2.9:** Fast transpose of a sparse matrix

# ARRAYS AND STRUCTURES

2.1 Arrays

2.2 Dynamically Allocated Array

2.3 Structures and Unions

2.4 Polynomials

2.5 Sparse Matrices

2.6 Representation of Multidimensional Arrays

- If an array is declared $a$[upper$_0$][upper$_1$]….[upper$_{n-1}$] the number of elements $= \prod_{i=0}^{n-1} upper_i$
  - Ex) a[10][10][10] → 10·10·10 = 1000

- Two ways to represent multidimensional arrays:
  - Row major order: A[upper0][upper1]   Row 우선

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A[0][0] | A[0][1] | ··· | A[0][upper$_1$-1] | | | ···· | |

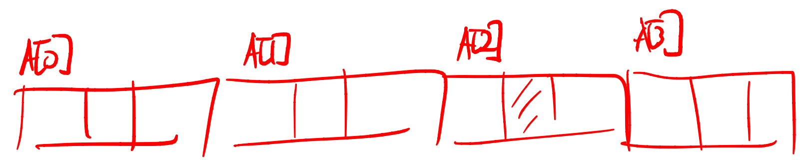
  - Column major order:   Col 우선

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A[0][0] | A[1][0] | ··· | A[upper$_0$-1][0] | | | ···· | |

- Addressing formula: …

- $A[upper_0\ ][upper_1]$
  - A[0][0]: $\boldsymbol{\alpha}$ (base address)
  - A[i][0]: $\boldsymbol{\alpha} + i \cdot upper_1$
  - A[i][j]: $\boldsymbol{\alpha} + i \cdot upper_1 + j$

- ex) A[3][3]
  - A[0][0] : A (base address)
  - A[2][2] : $A + 2 \cdot 3 + 2$




$A + 2 \times 3 + 2$

- $A[upper_0][upper_1][upper_2]$
  - Interpreted as $upper_0$ two-dimensional arrays
  - A[0][0][0]: $\boldsymbol{\alpha}$
  - A[i][0][0]: $\boldsymbol{\alpha} + i \cdot upper_1 \cdot upper_2$
  - A[i][j][k]: $\boldsymbol{\alpha} + i \cdot upper_1 \cdot upper_2 + j \cdot upper_2 + k$

- ex) A[3][4][5]
  - A[0][0][0] : A
  - A[2][3][4] : $A + 2 \cdot 4 \cdot 5 + 3 \cdot 5 + 4$

- $A[i_0][i_1]\ldots[i_{n-1}]$:

$$\boldsymbol{\alpha} + i_0 upper_1 upper_2 \ldots upper_{n-1}$$
$$+ i_1 upper_2 upper_3 \ldots upper_{n-1}$$
$$+ i_2 upper_3 upper_4 \ldots upper_{n-1}$$
$$\vdots$$
$$+ i_{n-2} upper_{n-1}$$
$$+ i_{n-1}$$

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

# ARRAYS AND STRUCTURES

2.1 Arrays

2.2 Dynamically Allocated Array

2.3 Structures and Unions

2.4 Polynomials

2.5 Sparse Matrices

2.6 Representation of Multidimensional Arrays