2장. 배열과 구조체

목차

- 배열**(Array)**의 정의
- 구조체**(Structure)**와 **Union**
- 다항식**(Polynomial)**의 표현
- 희소 행렬**(Sparse Matrix)**의 표현
- 다차원 배열**(Multidimensional Array)**

1. 배열의 정의

- 배열에 대한 **ADT** 정의**: ADT 2.1** 참조

- **C** 언어에서 배열
  - 배열의 선언
    - int list[5], *plist[5]; 주소 5개
    - 배열의 첨자(index)는 0부터 시작
  - 배열의 구현
    - list[0]의 주소 = α 라고 가정.
      list[1]의 주소 = α + sizeof(int)
    - list + i = &list[i], *(list + i) = list[i]
    - 배열을 인자로 전달: 배열의 시작주소가 복사
      $\Leftarrow$ Call-by Reference와 동일한 효과

ADT 2.1: 배열 ADT

ADT Array

객체: <첨자(index), 값(value)>의 쌍들의 집합. 각 첨자에 해당하는 배열 원소에 값들이 존재함. 첨자는 1차원 또는 다차원으로 정의되며, 1차원은 {0, …, n-1}, 2차원일 경우는 {(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), …} 등으로 표현함.

int A(10)(20)(5)

Creat(3, (10,20,5))

함수:

for all A ∈ Array, i ∈ index, x ∈ value, j, size ∈ integer

Array **Create(j, list)** ::= j-차원의 배열을 반환. list는 j-tuple로서 i번째 원소의 값은 i번째 차원의 크기를 나타냄. 배열 원소의 값들은 정의되지 않음.

Item **Retrieve(A, i)** ::= **if** (i ∈ index)
  return 배열 A의 첨자 i에 저장된 값
else return error

Array **Store(A, i, x)** ::= **if** (i ∈ index)
  return 배열 A에 <i, x>를 추가한 새로운 배열
else return error

A(X)= { {1,3,5}, {2,4,6} }

end Array

<2,7>

int A( ) = {1, 3, 7, 2}

<0,1> <1,3> <3,2>

index 값

(<0,0>,1)

(<0,1>,3)

(<1,1>,4)

배열의 동적 할당

- **C** 언어에서 **1**차원 배열의 동적 할당

```
int *A = (int *) malloc(sizeof(int) * 100);
int *B = (int *) calloc(100, sizeof(int));

A = (int *) realloc(A, sizeof(int) * 200);
free(A);
```

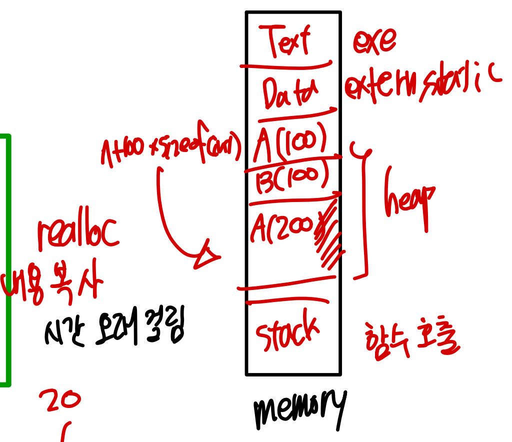
메모리 초기화 X

0으로 초기화

개수

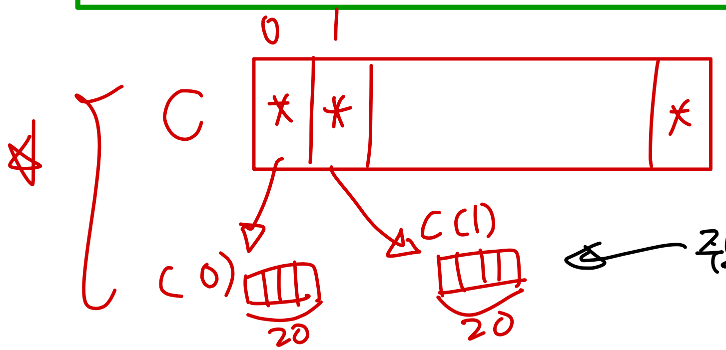
같을수도
다를수도
case by case

-1로 초기화하겠다
malloc 호출



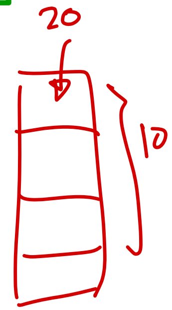
- **C** 언어에서 **2**차원 배열의 동적 할당

```
int **C = (int **) malloc(sizeof(int *) * 10);
for (int i = 0; i < 10; i++)
    C[i] = (int *) malloc(sizeof(int) * 20);
```



주소가 이어질수도
안이어질수도

int [10][20];



200개 연속적으로 할당

2. 구조체(Structure)와 Union

- 구조체
  - 하나 이상의 기본 자료형을 기반으로 사용자 정의 자료형을 만들 수 있는 문법 요소
  - 다양한 자료형을 포함 ↔ 배열: 동일한 자료형의 모음

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

```
human_being person;
strcpy(preson.name, "홍길동");
person.age = 21;
printf("%f", person.salary);
```

A , B

← 구조체 복사될까? 됨

B = A

서로영향X

f(B) f(A)

call by value

배열 주소때문에

X

구조체 비교

if(A==B) 구조체검사X

- 구조체의 내용이 동일한 지를 검사하는 방법
  - 구조체의 모든 속성들이 같은 지를 하나씩 비교
  - memcmp를 사용

```
int humans_equal (human_being person1, human_being person2)
{
        if (strcmp (person1.name, person2.name))
                return FALSE;
        if (person1.age != person2.age)
                return FALSE;
        if (person1.salary != person2.salary)
                return FALSE;
        return TRUE;
}
```

더 편함

```
memcmp(&person1, &person2, sizeof(human_being))
```

비교할 바이트 수

비트로 비교

갈다면 0
person1<person2 0보다 작은 값
person1>person2 0보다 큰 값

(0~3, 4~7, 8 , ~11) = 12   (0, ~3, 4~7, 8, ~11,) = 12   (0,~1, 2~3, 4~7) = 8
구조체 4바이트로 맞춤

구조체의 내부 구현

- **Memory Alignment**를 고려

sizeof(int)=4
(char)=1
(short)=2

더 효율적

```
struct {
    int
    int
    char
} A;
```

sizeof(A)=9 ?
4의 배수
12

```
struct {
    char
    int
    char
} B;
```

0에서 1떨어진
4의 배수
4사용한 것처럼 보임
6 ?
12

```
struct {
    char
    short
    int
} C;
```

7 ?
8

char
char
int
8

int는 4의 배수 주소에서 할당
short 2의 배수
char 1의 배수

0 —
4 —
8 —
12 —

Self-Referential Structure

- 자기 참조 구조체
  - 구조체의 속성 중 하나가 스스로를 가리키는 구조체

- 예

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

- 연결 리스트(**4**장)의 구현에 많이 사용됨

unsigned int 32 char a[4]

32bits 8

Union

- **Union**의 필드들은 메모리를 공유
- 예:

모두 각각 메모리 할당

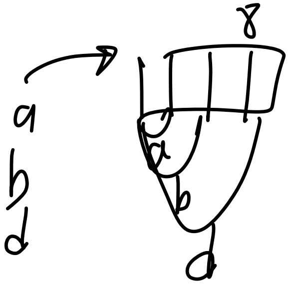
int 하나만 할당(메모리공유) 둘다 동시에 메모리 사용할때 없다

차이

```
struct human {
        enum {female, male} gender;
        union {
                int   children;
                int   beard;
        } u;
} person1, person2;

person1.gender = female; person1.u.children = 4;
person2.gender = male; person2.u.beard = FALSE;
```

union{
char a
int b
double d
}



- **C** 언어는 **union** 필드들의 적절한 사용을 검사하지 않음
  - person1.gender = female; person1.u.beard = TRUE;

3. 다항식의 표현

- 순서 리스트**(Ordered list)**란**?**
  - 데이터들의 순서가 유지되는 집합

- 순서 리스트의 예
  - 한 주의 요일들: (일, 월, 화, 수, 목, 금, 토)
  - 섞여진 카드들: (A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K)
  - 건물의 층: (지하실, 로비, 일층, 이층)
  - 미국의 2차 세계대전 참전 연도: (1941, 1942, 1943, 1944, 1945)
  - 스위스의 2차 세계대전 참전 연도

순서 리스트의 연산

- 순서 리스트에 적용 가능한 연산들 큰장점
  - 리스트의 길이 n의 계산 O(n)
  - 리스트의 모든 데이터들을 왼쪽에서 오른쪽으로 읽기 O(n)
  - 리스트로부터 i 번째 데이터를 검색, $0 \le i < n$ O(1)
  - 리스트로부터 i 번째 데이터를 대체, $0 \le i < n$ O(1)
  - 리스트의 i 번째 위치에 새로운 데이터를 추가 i=0 O(n) i=n O(1)
    - 그 결과로 기존의 i부터 $n-1$까지의 데이터들이 $i+1$부터 n까지 한칸씩 뒤로 이동
  - 리스트의 i 번째 데이터를 삭제 i=0 O(n) i=n O(1) 단점
    - 그 결과로 기존의 $i+1$부터 $n-1$까지의 데이터들이 i 부터 $n-2$까지 한칸씩 앞으로 이동

다항식 소개

- 순서화 리스트를 구현하는 두 가지 방법
  - 배열: i번째 데이터를 배열 인덱스 i에 저장
  - 연결 리스트
- 다항식의 정의

계수 (coefficient)
지수 (exponent)

  - $p(x) = a_n\, x^{e_n} + \ldots + a_1\, x^{e_1} + a_0$ 형태로 구성.
  - $a_i\ (0 \leq i \leq n)$는 0일 수 있으며, e_i는 정수
  - 다항식에 대한 ADT: ADT 2.2

ADT 2.2: 다항식 ADT (1)

ADT Polynomial 다항식

객체: $p(x) = a_1 x^{e_1} + \ldots + a_n x^{e_n}$; 순서화된 $< e_i, a_i >$ 쌍들의 집합으로 표현하되, a_i는 계수(coefficient) 이며, e_i는 지수(exponent)로 0보다 크거나 같은 정수임.

함수:

for all poly, poly1, poly2 ∈ Polynomial, coef ∈ Coefficients, expon ∈ Exponents

poly = {(3,2)(1,-5)(0,1)}

$p(x) = 2x^3 - 5x + 1$

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

다항식에서 지수가 expon인 계수를 반환

coef(poly,1) = -5

coef(ploy,2) = 0

Exponent Lead_Exp(poly) ::= **return** the largest exponent in poly
지수

Lead_Exp(ploy) = 3

poly={(3,2)(1,-5)(0,1)}

$p(x)=2x^3-5x+1$

ADT 2.2: 다항식 ADT (2)

Attach(p,5,1) X

(p,5,2)

계수 지수

Polynomial Attach(poly, coef, expon) ::= 계수가 coef고 지수가 expon인 항 추가

if (expon ∈ poly) **return** error

else return the polynomial poly

with the term <coef, expon> inserted

Polynomial Remove(poly, expon) ::= 지수가 expon인 항 제거

if (expon ∈ poly) **return** the polynomial poly

with the term whose exponent expon is deleted

else return error

$(2x^3-5x+1)\times x^2$
$=2x^5-5x^3+x^2$

Polynomial SingleMult(poly, coef, expon) ::= coef, expon인 항 하나와 주어진 다항식 poly를 곱합

return the polynomial poly · coef · x^{expon}

Polynomial Add(poly1, poly2) ::=

return the polynomial poly1 + poly2

Polynomial Mult(poly1, poly2) ::=

return the polynomial poly1 · poly2

end Polynomial

C 언어에서 다항식 구현 (1)

- 방법 **1:** 모든 지수의 계수들을 내림차순으로 저장

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

Lead_exp

x^3+1
[3, (1, 0, 0, 1)]

- 예: $2x^3 + x^2 - 1$ → [3, (2, 1, 0, -1)]
- 문제점?
  - $x^{100} + 1$

99개
[100, (1, 0 ⋯ 0, 1)]

C 언어에서 다항식 구현 (2)

x^3-2x^2+5
(1,3)(-2,2)(5,0) 6개
3[1,-2,0,5] 5개
빈공간이 많을때
장단점
식이 촘촘히 있을때

- 지수와 계수를 모두 저장

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

$x^{100} + 1 \rightarrow \{(1, 100), (1, 0)\}$

계수 지수

다항식의 모든 항들의 계수가 **0**이 아닌 경우?

C=A+B

- 모든 다항식을 저장하기 위한 전역변수 terms[] 사용
- 각 다항식은 <start, end>의 쌍으로 표현

	starta	finisha	startb			finishb	**avail**
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	

A B

Program 2.5: 다항식의 합(초기 버전)

```
d = Zero();   // d = a + b, a와 b, 그리고 d는 다항식
while (!IsZero(a) && !IsZero(b)) do  {          가장 큰항
    switch (COMPARE(Lead_Exp(a), Lead_Exp(b)))   {
        case -1:  d = Attach(d, Coef(b, Lead_Exp(b)), Lead_Exp(b));
         a<b      b = Remove(b, Lead_Exp(b)); 계수        지수
                  break;   제거
        case 0:   sum = Coef(a, Lead_Exp(a)) + Coef(b, Lead_Exp(b));
         a==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>b      a = Remove(a, Lead_Exp(a));
    }
}              // a와 b의 남은 항들을 d에 추가
```

$3x^2+x-5$

$-5x+6$

큰지수 부터

$3x^2+6x+1$

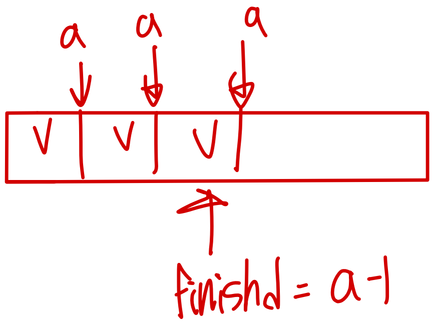
padd (sa, fa, sb, fb, & sd, & fd)

Program 2.6: 다항식의 합(1)

```
void padd(int starta, int finisha, int startb, int finishb, int *startd, int *finishd)      // d = a
    + b. 그런데 startd와 finishd는 왜 포인터일까?
{
    float coefficient;
    *startd = avail;        // avail은 terms[]에서 비어있는 공간의 색인
    while (starta <= finisha && startb <= finishb)
        switch (COMPARE(terms[starta].expon, terms[startb].expon))  ← lead_exp(a,b)
        {
            case -1: // a.expon < b.expon
                    attach(terms[startb].coef, terms[startb].expon);
                    startb++;  break;        // startb를 증가시키는 이유?
            case 0: // equal exponents
                    coefficient = terms[starta].coef + terms[startb].coef;
                    if (coefficient) attach(coefficient, terms[starta].expon);
                    starta++;  startb++;     // starta와 startb를 모두 증가
                    break;
                                                          = startb
```

Program 2.6: 다항식의 합(2)

```
        case 1: //  a.expon > b expon
                attach(terms[starta].coef,  terms[starta].expon);
                starta++;     // starta만 증가
    }
    //  a의 나머지 항들을 d에 모두 추가. 항이 없을 경우?
    for( ; starta <= finisha; starta++ )
            attach(terms[starta].coef,  terms[starta].expon);
    // b의 나머지 항들을 d에 모두 추가.
    for( ; startb <= finishb; startb++)
            attach(terms[startb].coef,  terms[startb].expon);
    *finishd = avail-1;                // avail-1에는 뭐가 들어 있을까?
}
```



Program 2.6: 다항식의 합(3)

```
void attach(float coefficient,  int exponent)
{
    // 다항식에 새로운 항을 추가하는 함수
    if (avail  >=  MAX_TERMS)    {
        fprintf(stderr, "Too many terms in the polynomial ₩n.");
        exit(1);
    }
    terms[avail].coef = coefficient;
    terms[avail++].expon = exponent;    // avail은 여기에서 증가됨.
}
```

	starta		finisha	startb		finishb	startd avail					
coef	3	2	1	8	-3	10						
exp	14	8	0	14	10	6						

		starta	finisha		startb	finishb	startd	avail				
coef	3	2	1	8	-3	10	11					
exp	14	8	0	14	10	6	14					

		starta	finisha			startb finishb	startd		avail			
coef	3	2	1	8	-3	10	11	-3				
exp	14	8	0	14	10	6	14	10				

			starta finisha			startb finishb	startd			avail		
coef	3	2	1	8	-3	10	11	-3	2			
exp	14	8	0	14	10	6	14	10	8			

			starta finisha			finishb	startb startd				avail	
coef	3	2	1	8	-3	10	11	-3	2	10		
exp	14	8	0	14	10	6	14	10	8	6		

while문 종료

			finisha	starta		finishb	startb startd				finishd	avail
coef	3	2	1	8	-3	10	11	-3	2	10	1	
exp	14	8	0	14	10	6	14	10	8	6	0	

4. 희소 행렬(Sparse Matrix)의 표현

4.1 희소 행렬의 정의

- 행렬의 표현: a[MaxRows][MaxCols]
  - 0이 많이 포함될 경우 ⇒ 희소 행렬

$$
\begin{array}{cc}
 & \begin{array}{ccc} col0 & col1 & col2 \end{array} \\
\begin{array}{c} row0 \\ row1 \\ row2 \\ row3 \\ row4 \end{array} &
\begin{bmatrix}
-27 & 3 & 4 \\
6 & 82 & -2 \\
109 & -64 & 11 \\
12 & 8 & 9 \\
48 & 27 & 47
\end{bmatrix}
\end{array}
$$

(a)

$$
\begin{array}{cc}
 & \begin{array}{cccccc} col0 & col1 & col2 & col3 & col4 & col5 \end{array} \\
\begin{array}{c} row0 \\ row1 \\ row2 \\ row3 \\ row4 \\ row5 \end{array} &
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

(b)

선형

degree(최고 항의 지수), coef(계수) 개수
x^3 + x^2 + x^1 + 4 //4개 할당

연산: retrieval
replace

$\text{sizeof(int)} = 4\text{byte}$

$\text{sizeof(int*)} = 4\text{byte}$

방법1

coef 지수

exp 계수

$8 \times n$

$8n < 4(n+2)$

$n < \frac{n+2}{2}$

+1??

방법2

degree 최고 항의 지수

coef 메모리할당

$(4 + 4 \times (n+1))$

메모리 때문에?

$4(n+2)$

$100x^5 + 60x$

poly.degree $= n$

poly.coef[i] $= a_{n-i}$

$(5, (100, 0, 0, 0, 60, 0))$

C 언어에서 희소 행렬의 구현

- 낭비되는 공간을 줄이자. (1000 * 1000 행렬)
  - <row, column, value>의 쌍을 저장
  - 빠른 전치(transpose)를 위하여 row의 오름차순으로 저장
- **ADT 2.3:** 희소 행렬 **ADT**

Sparse_Matrix Create (max_row, max_col) ::=

```
#define MAX_TERMS 101
typedef struct {
        int     row;     위치
        int     col;
        int     value;
} term;
term a[MAX_TERMS];
```

Example 1

$$
\begin{array}{c|cccccc}
 & \text{col0} & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col5} \\
\hline
\text{row0} & 15 & 0 & 0 & 22 & 0 & -15 \\
\text{row1} & 0 & 11 & 3 & 0 & 0 & 0 \\
\text{row2} & 0 & 0 & 0 & -6 & 0 & 0 \\
\text{row3} & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{row4} & 91 & 0 & 0 & 0 & 0 & 0 \\
\text{row5} & 0 & 0 & 28 & 0 & 0 & 0
\end{array}
$$

6x6=36

희소행렬

	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

이아닌 date 개수

9x3=27

ADT 2.3: 희소 행렬 ADT

ADT Sparse_Matrix

객체: <row, column, value> 쌍들의 집합으로, row와 column은 고유한 정수 조합이며, value는 그 row와 column에 저장된 값.

연산: a, b, c ∈ Sparse_Matrix, i, j, rows, cols ∈ index, v ∈ value

Sparse_Matrix Create(rows, cols) ::=
    rows × cols개의 항목을 저장할 수 있는 행렬 반환

Sparse_Matrix Transpose(a) ::=
    a[i][j] = v일 때 c[j][i] = v인 전치 행렬 c를 반환 $A : A^T$
    row ↔ col

Sparse_Matrix Add(a, b) ::=
    if (a와 b의 dimension이 동일) c[i][j] = a[i][j] + b[i][j]인 c 반환
    else **return** error

Sparse_Matrix Multiply(a, b) ::=
    if (a의 열의 수 == b의 행의 수) c[i][j] = Σ(a[i][k] · b[k][j])인
        c 반환
    else **return** error

4.2 행렬의 전치(Transposing a Matrix)

- 전치 연산의 특징
  - row와 column을 교환

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

  - <j, i, value>를 희소 행렬 M의 어디에 저장?
    - (0, 0, 15) → (0, 0, 15)
    - (0, 3, 22) → (3, 0, 22)
    - (0, 5, -15) → (5, 0, -15)
  - 전치 연산을 위한 연속적인 삽입으로 인해 기존에 저장된 항목들의 이동이 불가피!

Example

$O(n^2)$

$$
\begin{array}{c|cccccc}
 & col0 & col1 & col2 & col3 & col4 & col5 \\
\hline
row0 & 15 & 0 & 0 & 22 & 0 & -15 \\
row1 & 0 & 11 & 3 & 0 & 0 & 0 \\
row2 & 0 & 0 & 0 & -6 & 0 & 0 \\
row3 & 0 & 0 & 0 & 0 & 0 & 0 \\
row4 & 91 & 0 & 0 & 0 & 0 & 0 \\
row5 & 0 & 0 & 28 & 0 & 0 & 0
\end{array}
$$

n

$k \approx n^2$

row+col $= k$

선택 $= k^2$

qsort: $k + k\log_2 k$

$n^2 \cdot \log_2 n^2$

k

	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)

전치 연산의 구현: Transpose

for all elements in column j
    place element <i, j, value> in
    element <j, i, value> of the transpose

- **Program 2.8: Transpose**
  - O(columns * elements) ≅ O(columns^2 * rows)

n $\times\ n^2$ 0이 아닌 개수 $= n^3$(최악)   col   row

- **2**차원 배열에서 **transpose**의 구현

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

n^2

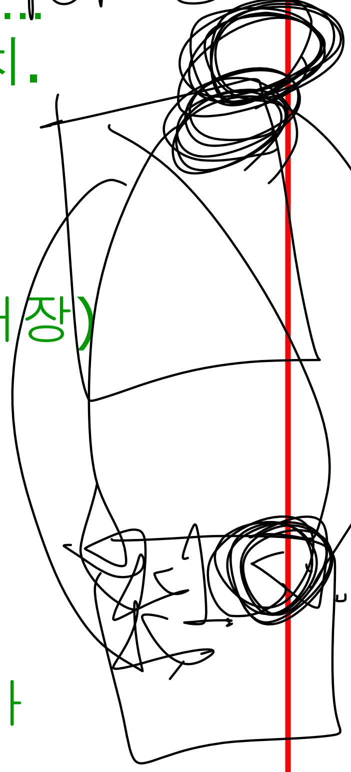
복잡도 = O(rows*columns)

Program 2.8: Transpose (1)

```
void transpose( term a[ ], term b[ ] )
//   a는 입력 행렬, b는 출력 행렬. b = aT
{
    int count, i, j, currentb;
    count = a[0].value;        // a에서 0이 아닌 원소의 수
    b[0].row = a[0].col;       // b의 행의 수 = a의 열의 수
    b[0].col = a[0].row;       // b의 열의 수 = a의 행의 수
    b[0].value = count;        // 0이 아닌 원소 수는 a와 동일
```

Program 2.8: Transpose (2)

row col value



```
if (count > 0)    {   // a가 empty matrix가 아니라면...
    currentb = 1;    // 새로운 원소가 b에 저장될 위치.
    for ( i = 0;  i < a[0].col;  i++ ) {
        // a의 열 순서로 전치 연산 실행 (i: 현재 열)
        for (j = 1; j <= count; j++) {
            //  a에서 현재 열을 찾자. (a는 행 순서로 저장)
            if ( a[j].col == i )     {
                //  현재 열의 원소 발견. b에 추가하자.
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++; // b의 저장될 위치를 1 증가
            }
    } } }
}
```

Complexity = O(columns * elements) → $O(n^3)$

전치 연산의 구현: Fast Transpose

- 기본 개념
  - 각 column이 저장될 곳을 미리 파악
  - Column Index 저장을 위한 추가적인 공간 사용

	[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: Fast Transpose**
  - O(columns + elements) ≅ O(columns * rows)
  - 배열을 하나만 사용하여 구현 가능

Program 2.9: Fast Transpose (1)

```
#define MAX_COL  50  // 최대 열의 수 + 1
void  fast_transpose(term a[ ],  term b[ ])
{   //   a는 입력 행렬, b는 출력 행렬. b = aT
    int  row_terms[MAX_COL];       // a의 열의 원소 수 저장
    int  starting_pos[MAX_COL];    // 각 열의 시작위치 저장
    int  i, j, num_col = a[0].col, num_terms = a[0].value;

    b[0].row = num_col;   b[0].col = a[0].row;
    b[0].value = num_terms;

    if (num_terms > 0) {    // a에 0이 아닌 원소들이 존재
        for(i = 0; i < num_col; i++)
          row_terms[i] = 0; // 초기화
```

공간 복잡도 = O(n)

O(n)
num_term

Program 2.9: Fast Transpose (2)

O(num_terms)

O(col)

O(num_terms)

```
    for (i = 1;  i <= num_terms;  i++)
       row_terms[a[i].col]++; // a의 각 열의 원소 수를 계산

    // row_terms를 이용하여 시작위치 계산
    starting_pos[0] = 1;
    for (i = 1; i < num_col; i++)
       starting_pos[i] = starting_pos[i-1] + row_terms[i-1];

    // 시작위치를 이용하여 특정 원소의 저장위치 파악.
    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;
       // b에서 동일한 행에 대해 열 순서로 저장되는가?
}  }  }
```

O(col + num_terms)

Fast Transpose의 동작 과정

0:1
1:3
2:4

	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

rowterms[0] = 2
rowterms[1] = 1
rowterms[2] = 2
rowterms[3] = 2
rowterms[4] = 0
rowterms[5] = 1

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

a[1] (0, 0, 15) → b[1] (0, 0, 15)
a[2] (0, 3, 22) → b[6] (3, 0, 22)
a[3] (0, 5, -15) → b[8] (5, 0, -15)
a[4] (1, 1, 11) → b[3] (1, 1, 11)
a[5] (1, 2, 3) → b[4] (2, 1, 3)
a[6] (2, 3, -6) → b[7] (3, 2, -6)
a[7] (4, 0, 91) → b[2] (0, 4, 91)
a[8] (5, 2, 28) → b[5] (2, 5, 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

Fast Transpose의 개선

```
for (i = 1;  i <= num_terms;  i++)
    row_terms[a[ i ].col]++;

tmp1 = 1;
for(i = 0;  i < num_cols;  i++) {
        tmp2 = row_terms[i];
        row_terms[i] = tmp1;
        tmp1 += tmp2;
    }

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

O(n)

starting_pos[] 배열을 사용하지 않고, tmp1과 tmp2 두 개의 변수로 해결. Transpose()와 동일한 공간 복잡도를 가짐.

4.3 희소 행렬의 곱셈

- 행렬의 곱셈 방법

  A = m × n 행렬, B = n × p 행렬, A × B의 결과를 D라고 할 때, D = m × p 행렬임. D의 <i, j> element 원소:

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

  for $0 \leq i < m$ and $0 \leq j < p$.

- 희소 행렬의 곱셈 결과는 희소 행렬이 아닐 수 있음.

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

희소 행렬의 곱셈의 예

(0,0)⇒(0,0)
(0,3)⇒(3,0)
(0,5)⇒(5,0)

col 순서로 저장

$$
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\times
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
3 & 0 \\
0 & 1 \\
2 & 0 \\
0 & 0
\end{bmatrix}
$$

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

b^T

	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]	2	6	5
[1]	0	0	1
[2]	0	2	3
[3]	0	4	2
[4]	1	1	1
[5]	1	3	1

희소 행렬의 곱셈의 예

$$
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\times
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
3 & 0 \\
0 & 1 \\
2 & 0 \\
0 & 0
\end{bmatrix}
$$

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

	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
[9]	6		

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

Program 2.10: 희소 행렬의 곱셈 (1)

a[] b[]
23 41
12 52

23 8
14 4

```
void mmult(term a[], term b[], term d[])
{         // a, b: 입력, d: 출력, d = a * b
    int i, j, column, totald = 0;
    int totala = a[0].value, totalb = b[0].value;
    int row_begin = 1, row = a[1].row, sum = 0;    // row: a의 현재 행
    term new_b[MAX_TERMS];

    if (a[0].col != b[0].row) {      // a의 열의 수와 b의 행의 수는 동일
         fprintf(stderr, "Incompatible matricses\n"); exit(1);
    }
    fast_transpose(b, new_b);

    // 경계 조건을 설정
    a[totala+1].row = a[0].row;
    new_b[totalb+1].row = b[0].col;
```

b^T

a[5].row=2

new_b[5]=2

Program 2.10: 희소 행렬의 곱셈 (2)

i=2 j=2

row_begin=1 row=0

```
for (i = 1; i <= totala; ) {
      column = new_b[1].row;           // b의 현재 열
      for (j = 1; j <= totalb + 1; ) {
        // a의 현재 행과 b의 현재 열에 대해 곱셈 수행
        if (a[i].row != row) {  // a의 현재 행을 벗어남.
              storesum(d, &totald, row, column, &sum);
              i = row_begin;  // b는 다음 열로. a는 원 위치로.
              for (; new_b[j].row == cloumn; j++) ;
              column = new_b[j].row;
        }
        else if (new_b[j].row != column) { // b의 현재 열을 벗어남.
              storesum(d, &totald, row, column, &sum);
              i = row_begin;  // a는 원 위치
              column = new_b[j].row  // b는 다음 열로.
        }
```

i=2

i=1

j=4

column=1

Program 2.10: 희소 행렬의 곱셈 (3)

```
        else switch (COMPARE(a[i].col, new_b[j].col)){
            case -1:        // a[i].col < new_b[j].col. a 증가
              i++;  break;
            case 0:         // 계산 후, a와 b를 모두 진행
              sum += (a[i++].value * new_b[j++].value); break;
            case 1:         // a[i].col > new_b[j].col. b 증가
              j++;
        }
      }
      for (; a[i].row == row; i++) ;    // b의 모든 원소를 처리한 후,
      row_begin = i;  row = a[i].row; // a의 현재 행을 다음 행으로.
  }   // end of for i <= totala
  d[0].row = a[0].row;
  d[0].col = b[0].col; d[0].value = totald;
}
```

0 1 0 1

i=3 j=3

a[1].value * new_b[1].value

a[2].value * new_b[2].value

0 == 0

i == 0

=1 =0

Program 2.10: 희소 행렬의 곱셈 (4)

```
void storesum(term d[], int *totald, int row, int column, int *sum)
{
    /* sum이 0이 아니면, d 배열의 *totald+1 위치에 row, column 값과 함께 저장 */
    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 2.10의 분석 0이 아닌 행의 원소

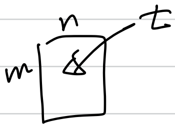
- **Complexity**
  - $O(\Sigma_{row}(cols\_b*termsrow + totalb))$ $= O(n^3)$

    $= O(\underbrace{cols\_b}_{n}*\underbrace{totala}_{n^2} + \underbrace{rows\_a}_{n}*\underbrace{totalb}_{n^2})$
- 전통적인 행렬 곱셈 알고리즘

```
for (i = 0; i < row_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;
    }
```

$O(n^3)$

  - Complexity: O(rows_a * cols_b * cols_a)
  - Non-sparse matrix
    - totala = rows_a * cols_a
    - totalb = rows_b * cols_b



배열의크기: $3t$

5. 다차원 배열

- 다차원 배열을 저장하는 두 가지 방법

Memory 1차원

A[2][4]

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]

A[10][20][30]

A(0)( )( ) ← 600

A( )(0)( ) ← 200

300 ?

행 우선 순서
(Row major order)

A[**0**][0]	A[0][*]
A[**0**][1]	
A[**0**][2]	
A[**0**][3]	
A[**1**][0]	A[1][*]
A[**1**][1]	
A[**1**][2]	
A[**1**][3]	

열 우선 순서
(Column major order)

A[0][**0**]	A[*][0]
A[1][**0**]	
A[0][**1**]	A[*][1]
A[1][**1**]	
A[0][**2**]	A[*][2]
A[1][**2**]	
A[0][**3**]	A[*][3]
A[1][**3**]	

A(100) A(5)

① size 확인

② 5번째로 이동

A(10)(20)

A(0)(0) =α

A(7)(0)=α+7×20

A(0)( )

A(6)

A(7)(0)

행 우선 순서의 주소 계산

- **2차원 배열: A[$\mathbf{upper_0}$][$\mathbf{upper_1}$]**
  - α가 A[0][0]의 주소라고 가정
  - A[i][0]의 주소 = α + i * $upper_1$
  - A[i][j]의 주소 = α + i * $upper_1$ + j
- **3차원 배열: A[$\mathbf{upper_0}$][$\mathbf{upper_1}$][$\mathbf{upper_2}$]**
  - α가 A[0][0][0]의 주소라고 가정
  - A[i][j][k]의 주소 = α + i * $upper_1$ * $upper_2$ + j * $upper_2$ + k
- 예: A[10][20][30] 배열에서 A[0][0][0]의 주소가 α
  - A[4][2][5]의 주소 = α + 4 * 600 + 2 * 30 + 5

    = α + 2465

0, 1, 2, 3

A(4)(0)(0)

α+4×600+ 2×30+5

행 우선(Row Major) 저장의 규칙

col Major
반대로 곱하면 나옴
$upper_0 \times upper_1$

- 다차원 배열: **A[upper$_0$][upper$_1$]…[upper$_{n-1}$]**
  - α가 A[0][0]…[0]의 주소라고 가정
  - Address of A[i_0][i_1]…[i_{n-1}] =

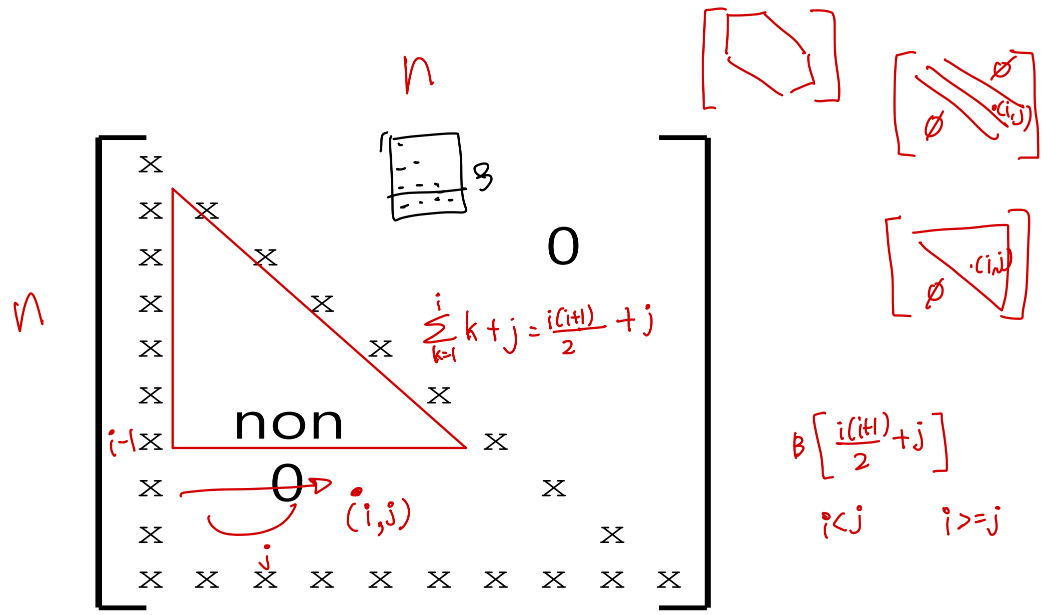
$$
\begin{aligned}
\alpha &+ i_0\, upper_1\, upper_2 \ldots upper_{n-1} \\
&+ i_1\, upper_2\, upper_3 \ldots upper_{n-1} \\
&+ i_2\, upper_3\, upper_4 \ldots upper_{n-1} \\
&+ \ldots \\
&+ i_{n-2}\, upper_{n-1} \\
&+ i_{n-1}
\end{aligned}
$$

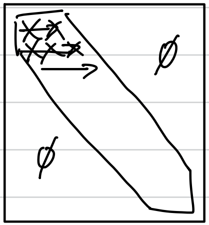
$$= \alpha + \sum_{j=0}^{n-1} i_j a_j \quad \text{where } a_j = \prod_{k=j+1}^{n-1} upper_k \quad (0 \le j < n-1)$$

$$a_{n-1} = 1$$

배열의 주소 계산 문제의 예

- 아래 그림과 같은 삼각 행렬(triangular matrix)의 경우, A[n][n] 형태의 이차원 행렬 대비 기억 공간을 절약하기 위하여 0이 아닌 데이터만 일차원 배열 B[n(n+1)/2]에 저장하고자 한다. A[0][0]은 B[0]에 저장하고, 0이 아닌 A의 데이터들을 <u>행 우선 순서</u>로 B에 저장할 때, A[i][j]는 B의 어느 위치에 저장되는가? 단, i ≥ j이다.





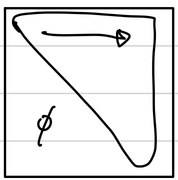
A[50][49]   B

0: 2개   1~49: 3개   50 = 49×3+2+49

$B[0 : 3n-3]$

100

100



(i, j)

A(90, 93)

$\sum_{k=0}^{k=i-1} (100-k)$

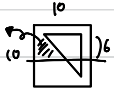
$\frac{(100-(i-1))\times(100-i)}{2} = \frac{(101-i)(100-i)}{2}$

A(6, 9)

10

10

6



$\sum_{k=1}^{i} 10 - \sum_{k=0}^{i-1} k$

i=6 $\sum_{k=0}^{6} 10 - \sum_{k=0}^{5} k$

$10\times6 - \frac{5\times6}{2} = 60-15=45$

list[6] = *(list+6)

≠ (list+6)

ptr

list(0)+3

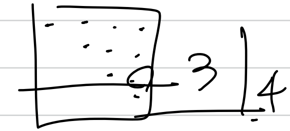
[4 7 (5)(3)]

0~1=15 ×2=30

(2, 3, 1)

3×3=9

+1=1



$\sum_{k=1}^{100}$ $\frac{100(101)}{2}$

$- \sum_{k=1}^{10} \frac{10(11)}{2} - 7$

$*(a+2)[3]$

$(a[0]+5\times2)[3]$

$*(\&a[0][0]+5\times2+3)$

$*(a+5\times2+3)$

$*A+k$ $\&A[0]+k$ 주소 값

$A[k]$

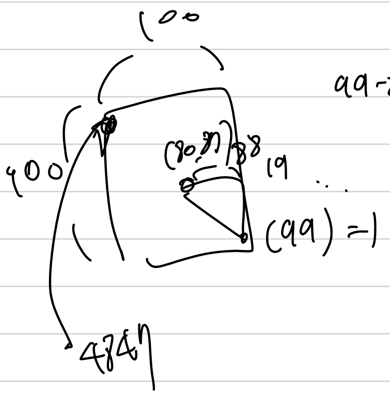
$*(A+k)$

$a[3][5]$ 선언 $a[2][3]$ 다른값

$A[100][100]$

$B[5050]$

$A[80][87] \rightarrow B(x)$



$99-88=12$

$\frac{19\times20}{2}=190+12=202$

$5050-202=4848$