배열과 구조

* 자료구조의 구분

- 선형(linear) 과 비선형(non-linear)으로 구분한다.
- 자료구조내의 요소들이 순차적인 형식이면 선형, 아니면 비선형.
- 선형자료구조에는 **배열과 연결 리스트**가 있다.
- **배열**: 순차적 메모리 주소에 의하여 요소들간의 관계를 표현하는 구조
- **연결리스트**: 포인터를 사용하여 요소들간의 관계를 표현
- 비선형 자료구조: 트리(tree) 와 그래프(graph)

1. 배열의 특징

- 연속적 기억장소(메모리 위치)의 집합
- 대부분의 언어에서 제공하는 가장 단순한 구조적 자료형
- 동일한 자료형 (Same data type for elements)
- 선언 시 크기 지정. 크기보다 많은 양의 자료 저장시 => overflow
- 정적 자료형 (compile 시 크기를 알아야 하고, 실행 되는 동안 크기가 변하지 않는다)
- Set of mappings between index and values; **<index, value>**

장점: 이해 쉽고, 사용하기 편함, 자료저장이 용이(예: A[4]=10)

단점: 동일한 자료만 저장, 미리 크기 선언(필요이상 크기 선언시, 공간낭비, 많은 자료이동으로 삽입 삭제 느림.

Structure *Array*

Objects: index의 각 값에 대하여 집합 item에 속한 한 값이 존재하는 <index, value>쌍의 집합. index는 일차원/다차원의 유한 순서 집합.

Functions:**모든** **A** ***∈Array, i∈ index, x∈ item, j, size∈ integer***

***Array* Create(j, list)**::= return j차원의 배열. list는 i번째 원소가 i번째 차원의 크기인 j-tuple이며 item들은 정의되지 않았음.

***Item* Retrieve(A, i)::=** if (*i* ∈ *index*)

return 배열 A의 인덱스 i 값과 관련된 항목.

else return 에러.

***Array* Store(A, i, x)::=** if (*i* ∈ *index*)

return 새로운 쌍<*i*, *x*>가 삽입된 배열 A.

else return 에러.

end *Array*

- **배열의 연산**
  - i. length n (길이 n의 연산)
  - ii. reading (R => L, L => R)
  - iii. retrieve i^{th} element, $0 \le i<n$
  - iv. update i^{th} element's value, $0\le i<n$
  - v. Insertion (i번째 위치, $0 \le i\le n$)
  - vi. deletion (i번째 항목, $0 \le i<n$)

2. 배열의 표현 (표현순서: 행, 열 우선)

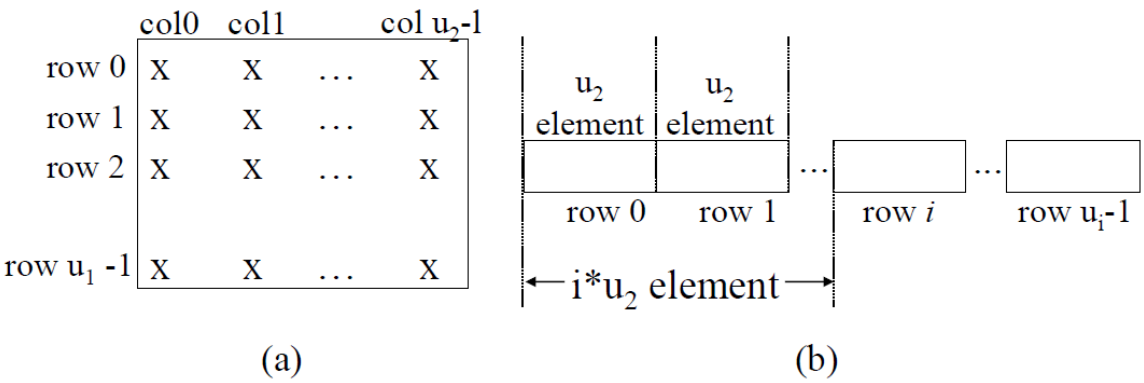
- 1차원 배열: A[i]

- α: a[0]의 주소(base address)
- 임의의 원소 A[i]의 주소 : α + i* (4byte)

배열주소	A[0]	A[1]	..	A[i]	..	$A[u_1\text{-}1]$
주소	α	α+1	..	α+i	..	$\alpha+u_1\text{-}1$

- 2차원 배열: A[i][j]

- α: A[0][0]의 주소
- 임의의 원소 A[i][0]의 주소 : α +i*u2



$a[u_1][u_2]$의 순차적 표현

row major:

α (base address)

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3

Col major

α (base address)

0,0	1,0	2,0	0,1	1,1	2,1	0,2	1,2	2,2	0,3	1,3	2,3

- 3차원 배열: a[i]j][k], - n차원 배열: a[i1][i2]…

*** 순서리스트의 일반적 구현**

. 배열을 이용: 인덱스 i, $0 \le i < n$

. 순차 사상(sequential mapping)

. 문제점: insert/delete의 overhead

- **Structure**

*** Struct** - 타입이 다른 데이타를 그룹화

```
- struct {
    char name[10];          ex) strcpy(person.name, "james");
    int age;                    person.age=10;
    float salary;               person.salary= 3000;
} person;
```

*** nested structure 허용**

- 자체 참조 구조 (self_referential structure)

(구성요소 중 자신을 가리키는 포인터가 존재하는 구조)

```
struct list {
    char data;
    list *link; };      /* 링크드 리스트
```

- 문자열 추상 데이터 타입 (string data type)

문자열(string): $S=s_0,\ldots,s_{n-1}$의 형태, (n=0, 공백 문자열)

s_i : 문자 집합의 원소

Structure **String** is
Objects: a finite set of 0 or more characters
Functions: forall s,t:string, i,j,m: non-negative integers
String Null(m) : initialize
Integer Compare(s,t): return 0 if s==t, else (return –1 if s proceed t else returns +1)
Boolen IsNull(s)::= if (compare(s, NULL)) returns False
Else return True
Integer Length(s)::= If (compare(s, NULL)) return 0
Else return number of characters
String Concat(s)::= If(compare(s, NULL)) return s
Else return append t at end of s
String Substr(s,i,j)::= if((j>0)&&(i+j-1) <length(s))
return null; else return string from the index I to the index i+j-1.

```
Ex) #include <string.h>
#define MAX_SIZE 100 // Max size of the string

char string1 [MAX_SIZE] = {"amobile"}, *s = string1;
char string2 [MAX_SIZE] = {"uto"}, *t = string2;
strings (s, t, 1);

void strings (char *s, char *t, int i){
   char string[MAX_SIZE], *temp = string;
   if (!strlen (s))    strcpy (s, t);    //if string s is empty
      else if (strlen (t)) {
         strncpy (temp, s, i);    //copy 1 ch. From s to temp
         strcat (temp, t);        // make new temp+t
         strcpy (s, temp):        // write s to temp
```

2. Polynomial (다항식)

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

ax^e *a* : *coefficient (계수)* $a_n >= 0$

e : *exponent* (지수)

x : *variable x (변수)*

- 차수(*degree*): 다항식에서 0 이 아닌 가장 큰 지수

- 다항식의 덧셈과 곱셈

  합: $A(x) + B(x) = \sum (a_i + b_i) \odot x^i$

  곱: $A(x) \odot B(x) = \sum (a_i \odot \sum (b_j \odot x^j))$

2.1 다항식 표현

- <u>다항식의 표현 (1)</u>: 모든 지수에 대한 계수만 저장, 계수 배열 크기는 최대로

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

degree of A   *n+1 coefficients*

```
private:
    int degree;          //degree ≤  MaxDegree
    float coef [MaxDegree + 1];   // 계수 배열
```

a : polynomial 클래스 객체, $n \leq$ MaxDegree

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

* a.coef[i]는 x^{n-i} 의 계수, 각 계수는 지수의 내림차순으로 저장

장점: 다항식에 대한 연산이 간단.

단점: 저장공간 낭비 (a.degree가 maxdegree보다 아주 작을 때)

예) **$3x^4 + 5x^2 + 6x + 4$ 의 경우 A = (4, 3, 0, 5, 6, 4)**

degree	**4**										
coef	0	0	0	0	0	0	**3**	**0**	5	**6**	**4**
	10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

문제점) $A(x) = x^{1000} + 1$: n = 1000

A = (1000, 1, $\underbrace{0,\ldots,0}_{999의\ 엔트리는\ 0}$, 1)

⇨ *a.coef[MAX_DEGREE]*의 대부분이 필요없음

* 다항식의 표현 (2): 0이 아닌 계수-지수 쌍 저장

$$A(x) = b_{m-1}x^{e_{m-1}} + b_{m-2}x^{e_{m-2}} + \ \ldots \ + b_0x^{e_0}$$

Where $b_i \neq 0$, $0 \leq i \leq m-1$, $e_{m-1} > e_{m-2} > \ldots > e_0 \geq 0$

$A = (m,\ e_{m-1}, b_{m-1},\ e_{m-2}, b_{m-2}, \ldots,\ e_0, b_0)$

no. of non-zero terms

예) $A(x) = x^4+10x^3+3x^2+1$ **A = (4, 4, 1, 3, 10, 2, 3, 0, 1)**

$A(x) = x^{1000} + 1$ **A = (2, 1000, 1, 0, 1)**

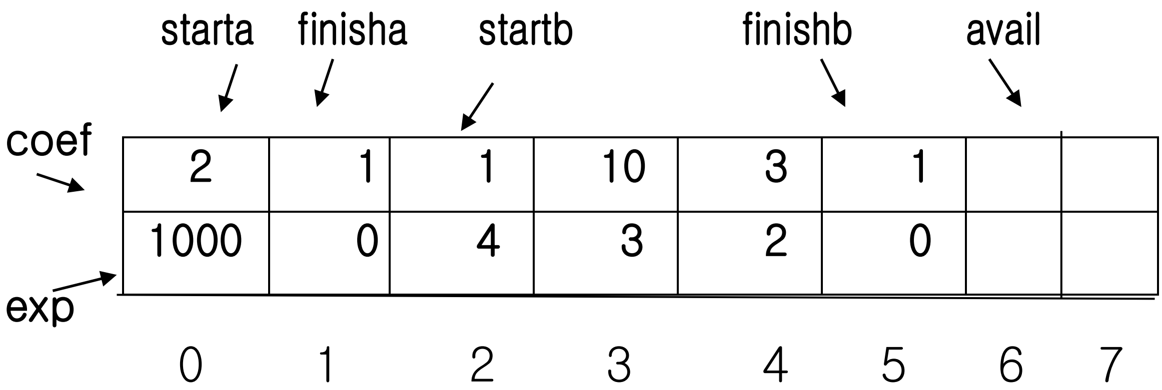
```
MAX_TERMS 100          /* 항 배열의 크기 */
typedef struct {
            float coef;
            int expon;
         } Polynomial;
Polynomial   terms[MAX_TERMS];
int   avail = 0;
```

예) $3x^4 + 5x^2 + 6x + 4$ 의 경우

coef	3	5	6	4
expon	4	2	1	0

ex) 두개의 다항식 A(x), B(x)

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



startA = 0, finishA = 1, startB = 2, finishB = 5, Avail = 6

A(x) : <*starta*, *finisha*> B(x): <startb, finishb>

다항식 덧셈 D = A + B

```
void padd (int starta, int finisha, int startb, int finishb, int *startd,
int *finishd);

{   /* A(x) 와 B(x)를 더하여 D(x)를 생성한다 */
    float coefficient;          *startd = avail;

while (starta <= finisha && startb <= finishb)

    switch(COMPARE(terms[starta].expon, terms[startb].expon))
    {
       case -1: /* a의 expon이 b의 expon보다 작은 경우 */
              attach(terms[startb].coef, terms[startb].expon);
              startb++;   break;

       case 0: /* 지수가 같은 경우 */
              coefficient= terms[starta].coef + terms[startb].coef;
              if(coefficient)
                   attach(coefficient, terms[starta].expon);
                   starta++;   startb++; break;

      case 1: /*   a의 expon이 b의 expon보다 큰 경우 */
             attach(terms[starta].coef, terms[starta].expon);
             starta++;
     }

 /* A(x)의 나머지 항들을 첨가한다 */
    for(; starta <= finisha; starta++)
         attach(terms[starta].coef, terms[starta].expon);

 /* B(x)의 나머지 항들을 첨가한다 */
  for(; startb <= finishb; startb++)
         attach(terms[startb].coef,terms[startb].expon);
  *finishd = avail-1;     }
```

```
void attach(float coefficient, int exponent) {
    if (avail >= MAX_TERMS) {
        cout<< "too many elements.. ";     break; }
    terms[avail++].coef = coefficient;
    terms[avail++].expon = exponent;
}
```

3. 희소행렬 (Sparse Matrix)

- *m × n 행렬 A ≡ A[MAX_ROWS][MAX_COLS]*

  m=n: 정방 배열. //m:행의수, n:열의수

- Sparse Matrix(희소 행렬)

  [0 이아닌 원소수 / 전체 원소수] << small

  ➔ 0 아닌 원소만 저장 => 시간 /공간 절약

- 행렬연산: creation, addition, multiplication, transpose,..

- 밀집 행렬과 희소행렬의 예

$$\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix} \quad \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

- 효율적 희소행렬 표현

  ⅰ) <i, j, value> : 3-tuples (triples)로 식별가능

  ⅱ) no. of rows (행의 수)

  ⅲ) no. of columns (열의 수)

  ⅳ) no. of non-zero elements

  ⅴ) ordering (column major or row major)

• *sparse matrix (row major) : 행 순서로 저장*

	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	0	0
2	0	0	0	-6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

(row)	*(col)*	*(value)*
6	6	8
0	0	15
0	3	22
0	5	-15
1	1	11
1	2	3
2	3	-6
4	0	91
5	2	28

Sparse Matrix ‘a’

• *sparse matrix (column major): 열 순서로 저장*

(transpose)

(col)	*(row)*	*(value)*
6	6	8
0	0	15
0	4	91
1	1	11
2	1	3
2	5	28
3	0	22
3	2	-6
5	0	-15

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

Sparse Matrix ‘b’

- <u>희소 행렬의 전치 (Transpose)</u>

원래의 행렬 각행 i 에 대하여 원소(i, j, 값)을 전치행렬의 원소 (j, i, 값)으로 저장

```
void transpose( SMarray a[], SMarray b[])
/* a 를 전치시켜 b 를 생성, 예:(0,3,22) -> (3,0,22) */ {
   int    i, j, currentb;
     b[0].row = a[0].col;
     b[0].col = a[0].row;
     b[0].value = a[0].value;

   if (a[0].value    >    0)    { /* 0 이 아닌 행렬 */
      currentb = 1;
      for (i =0; i <a[0].col; i++) /* a 에서 열별로 전치*/
         for (j = 1;    j  ≤  a[0].value;    j++)
               /* 현재의 열로부터 원소를 찾는다. */
               if (a[j].col ==i ) {
                  /*현재 열에 있는 원소를 b 에 첨가 */
                  b[currentb].row = a[j].col;
                  b[currentb].col = a[j].row;
                  b[currentb].value=a[j].value;
                  currentb++;
            }
         }
 }
```

분석: 시간 복잡도 O(col * elements)

- Better Transpose Algorithm (개선된 알고리즘)

```
void fast_transpose(term a[], term b[])
{
    /* a 를 전치시켜 b 에 저장 */
int row_terms[MAX_COL],   starting_pos[MAX_COL];
int i,j, num_col = a[0].col,      num_terms = a[0].value;

b[0].row = num_cols;   //6
b[0].col = a[0].row;      //6
b[0].value = num_terms; //8

if (num_terms > 0) { /* 0 이 아닌 행렬 */
    for(i = 0; i < num_cols; i++)
        row_terms[i] = 0;
    for(i = 1; i <= num_terms; i++) /* 각 row terms 위한 값
        row_terms[a[i].col]++;

    starting_pos[0] = 1;     /* 각 row terms 시작점 구함
    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++) { /* A 를 B 로 옮김
        j = starting_pos[a[i].col]++;
        b[j].row = a[i].col;  b[j].col = a[i].row;
        b[j].value = a[i].value;
    }
} }
```

- **O(*columns* + *elements*)**

더 빨리 변형할 수 있는 알고리즘

A matrix

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

B matrix (=A^T)

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

Starting_pos:	1	3	4	6	8	8
A의 col. Idx 값:	0	1	2	3	4	5
Row_terms:	2	1	2	2	0	1

이걸 만들면, B의 시작 주소로 쓰이게 된다.

순서

1. A 행렬을 읽고, row_terms와 starting_pos를 만든다.
2. A 행렬을 읽고, 특정 col.#를 가진 element를 세어서

   B의 특정 row의 element로 삽입하고, 삽입 수 만큼

   starting_pos를 증가시킨다.