배 열 과 구 조

* 구분

- 선형(linear) 과 비선형(non-linear)으로 구분한다.

- 자료구조내의 요소들이 순차적인 형식이면 선형, 아니면 비선형.

- 선형 자료구조에는 **배열과 연결 리스트**가 있다.

- **배열**: 순차적 메모리 주소에 의하여 요소들간의 관계를 표현하는 구조

- **연결 리스트**: 포인터를 사용하여 요소들간의 관계를 표현

- 비 선형 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.

* 추상데이터 타입으로서의 배열

- 일련의 연속적 메모리 위치: 구현중심
- C++배열: 인덱스 집합이 0부터 시작
- 함수:
  . 생성자: GeneralArray(j-dimension, j-range list, initial value)
  . 멤버 함수 (Retrieve(index), Store(index, item))

< 배열 추상데이터 타입(ADT) >

Class General*Array* {

//Objects: index의 각 값에 대하여 집합에 속한 한 값이 존재하는 <index, value>쌍의 집합.
// Index Set은 일차원/다차원의 유한 순서 집합.
//일차원의 경우 {0,1,2…n-1}, 이차원 (0,0)(0,1)….

Public:

GeneralArray (int j, RangeList list, float initValue=defaultvalue);
// **생성자 GeneralArray는** j차원의 배열생성.
// index 집합의 각index i에 대해 <i, initValue>를 배열에 삽입

***float* Retrieve(index i);**
// if ($i \in index$) return 배열의 인덱스 i 값과 관련된 항목
// else return 에러.

***void* Store(index i, float x);**
// if ($i \in index$) return 새로운 쌍<*i*, *x*> 삽입. else return 에러.
}
//end General*Array*

* 다항식 추상데이터 타입

배열은 자체가 자료구조이며, 다른 ADT의 구현에도 사용가능.

- 순서 리스트(또는 선형 리스트)에 배열의 사용

Ex) 순서 리스트:

한주일의 요일 (월, 화, 수…), 카드(Ace,2,3,4,..)

리스트 형태 : $(a_0,\ a_1,\ a2, \ldots)$

* 순서리스트의 일반적 구현

. 배열을 이용 (index i -> a_i, $0 \le i < n$)

. 순차 사상 (sequential mapping) //원소 a_i를 index i와 대응시킴

. 문제점: insert/delete의 overhead

● 순서리스트의 연산

① length *n*

② reading ($R \Rightarrow L$, $L \Rightarrow R$)

③ retrieve i^{th} element, $0 \le i < n$

④ update i^{th} element's value, $0 \le i < n$

⑤ insertion (*i 번째 위치*, $0 \le i \le n$)

⑥ deletion (*i 번째 항목*, $0 \le i < n$)

● 순서리스트를 필요로 하는 문제(예: 다항식)를 배열로 해결.

- ex) 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. Polynomial (다항식)

$$A(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x^1+a_0x^0$$

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

e : *exponent* (지수)

x : *variable x (변수)*

- **차수(*degree*): 다항식에서 가장 큰 지수**

● Polynomial 추상데이터 타입

Class Polynomial {

// Objects : A(x) = $a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x^1+a_0x^0$;

// $<e_i, a_i>$의 순서쌍으로 된 집합 ($a_i \in$coefficient, $e_i \in$ Exponent),

// Exponent는 ≥ 0 정수로 가정)

public:

Polynomial(); // 다항식 p(x)=0를 생성

Coefficient Coef(Exponent e); // e의 계수를 반환

Exponent LeadExp(); // 가장 큰 지수를 반환

Polynomial Add(Polynomial poly); // poly의 합을 반환

Polynomial Mult(Polynomial poly); // poly의 곱을 반환

float Eval(float f); // f를 대입한 값을 계산, 결과를 반환

}**;**

// polynomial ADT

2.1 다항식 표현

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

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

degree of A *n+1 coefficients*

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

a : polynomial 클래스 객체, n ≤ MaxDegree

a.degree = n **a.coef[i] =** a_{n-i}**,** **0** ≤ i ≤ n

* a.coef [i] 는 x^{n-i} 의 계수, 각 계수는 <u>**지수의 내림차순으로**</u> 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): 모든 차수에 대한 계수만 저장, 계수 배열 크기는 <u>실제 차수 크기로</u>

```
Polynomial::Polynomial(int d){
    degree = d;
    coef=new float[degree+1];
}
```

. 단점: 희소 다항식에서 기억 공간 낭비

(예) 다항식 $x^{1000}+1 \rightarrow$ coef에서 999개의 항목은 0

* <u>다항식의 표현 (3)</u>: 0이 아닌 계수-지수 쌍 저장

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

Where b_i **: 0 이 아닌계수** $(b_i \neq 0)$

e_i **: 지수, 내림차순,** $(e_{m-1} > e_{m-2} > \ldots > e_o \geq 0)$ $\quad 0 \leq i \leq m-1,$

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

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

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

```
MAX 100
typedef struct {
            float coef;
            int expon;
        } Polynomial;
Polynomial    terms[MAX];
```

```
class term{
  friend polynomial;
  private:
      float coef;
      int expon;
   };
```

ex) $3x^4 + 5x^2 + 6x + 4$ 의 경우

3	5	6	4
4	2	1	0

ex) 두개의 다항식 A(x), B(x) 표현

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

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	

startA = 0, finishA = 1, s tartB = 2, finishB = 5, Avail = 6

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

- N 개의 0 이 아닌 항을 가진 다항식 A는
  . finishA = startA + (n-1)의 식을 만족하는 startA와 finishA를 가짐
  . termArray에 저장될수 있는 다항식의 수는 maxterms
  . 장점: 많은 항이 0 이 아닌경우 우수
  . 단점: 모든항이 0이 아닐때, 표현2보다 두배의 저장장소 사용
  . 다항식 덧셈: A(X) + B(X) = D(X)를 구하는 C++ 함수
  . PADD: A(X)와 B(X)를 항별로 더하여 D(X)를 만드는 함수
  . 다항식은 0 이 아닌 항만 저장.

다항식 덧셈 D = A + B

void padd ( ); {
/* A(x) 와 B(x)를 더하여 D(x)를 생성한다 */
float coefficient;
*startd = avail; // beginning position of new polynomial D(x)

```
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) // if not 0
                     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;     } //end of D(x)
```

void attach(float coefficient,   int exponent) {

```
 /* 새 항을 다항식에 첨가한다. */
    if (avail >= MAX ) { 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:행의수, n:열의수 if m=n, 정방배열 (square matrix)

- Sparse Matrix(희소 행렬)
  [0 이아닌 원소수 / 전체 원소수] << small
  ➔ 0 아닌 원소만 저장 => 시간 /공간 절약

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

- 밀집 행렬 과 희소행렬(0 이 많은 행렬)의 예

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

Class *SparseMatrix*{
 // objects: 3원소쌍 <행, 열, 값>의 집합이다.
 // 행과 열은 정수, 값 또한 정수이다.
public:
 ***SparseMatrix*(int *MaxRow*, it *MaxCol*);**
 //생성자함수: MaxItems=MaxRow x MaxCol을 저장하는 SparseMatrix를 생성.
 SparseMatrix Transpose();
 // 모든 3원소 쌍의 행과 열의 값을 서로 교환
 ***SparseMatrix Add*(*SparseMatrix b*);**
 // if a와 b의 차원이 같으면 같은 행,열 값 가진 항들을 덧셈하고, else 오류발생
 ***SparseMatrix Multiply*(*SparseMatrix b*);**
 // if a 의 열 수와 b 행 수가 같으면 a와 b를 곱해서 생성된 행렬 d를 반환, else 오류를 발생한다}

- 효율적 **<u>희소행렬 표현방법</u>**
  - 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)

- C++ 표현

```
class SparseMatrix;    // forward declaration

class MatrixTerm {
friend class SparseMatrix;
private:
      int row, col, value;
};
```

- Class *SparseMatrix* 내부 정의

```
private:
   int Rows, Cols, Terms;    //rows: 행의수, col: 열의수
   MatrixTerm    smArray[MaxTerms];
```

<u>• sparse matrix (row major) : 행 순서로 저장</u>

	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’

• *<u>sparse matrix (column major)</u>*: *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

Sparse Matrix ‘b’

● <u>희소 행렬의 전치 (Transpose)</u>

원래의 행렬 각행 i 에 대하여 원소(i, j, 값)을 전치행렬(Transpose matrix)의 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;                      // 6 6 8
  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++; } }}
```

- 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_cols = 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;       // number of terms 초기화

    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;
    }
} }
```

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

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

4. 배열의 표현

- 다차원 배열의 표현

  Addressing formula: $A[p_1..q_1][p_2..q_2],\cdots,[p_n..q_n]$

. 배열 A의 총 원소수: $\prod_{i=1}^{n}(q_i - p_i + 1)$

. **<u>표현 순서: 행/열 우선</u>**-> 사전 순서(lexicographic order)

- 행 우선(row major order)

(ex) A[4..5][2..4][1..2][3..4]

. 총 원소 수: 2*3*2*2=24

. 저장 순서: 4213, 4214, …, 5423, 5424

-> 사전순서(lexicographic order)

*** 단순화를 위한 가정: 차원 i의 인덱스: 0에서 u_i-1**

- 1차원 배열 a[u_1]

– α: A[0]의 주소

– 임의의 원소 A[i]의 주소 : α +i

배열원소:	A[0]	A[1]	.. A[i]….	A[u_1-1]
주소:	α	α+1	…α+I	…α+u_1-1

- 2차원 배열 A[u_1][u_2]

. α : A[0][0]의 주소

. A[i][o]의 주소: α+ i*u_2 .A[i][j]의 주소: α+ i*u_2+j

- 3차원 배열 A[u_1][u_2][u_3]

. α : A[0][0][0]의 주소

. A[i][0][0]의 주소: α +i

. A[i][j][k]의 주소: $\alpha + iu_2u_3 + ju_3 + k$

- n차원 배열 A[u_1][u_2]…,[u_n]

. α ; A[0][0],…,[0]의 주소

. A[i_1][0][0]…,[0]의 주소: $\alpha + i_1u_2u_3 \ldots u_n$

. A[i_1][i_2][0],…,[0]의 주소: $\alpha + i_1u_2u_3 \ldots u_n + i_2u_3u_4 \ldots u_n$

- **문자열 추상 데이터 타입 (string data type)**

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

  s_i : 문자 집합의 원소 // if n=0, then empty or NULL

  - "abc" 형태
  - 내부적으로는 char의 배열로 표현
  - 맨 뒤에 NULL 문자 사용

. 문자열 연산

생성, 문자열의 읽기 또는 출력,
두 문자열 접합(concatenation), 문자열 복사(copy),
문자열 비교(compare),
문자열에 일부 문자열 삽입(insert)
문자열로부터 일부 문자열 삭제(delete)
문자열에서 특정 패턴 식별(find)

```
class String{
    // objects :  0개 이상의 문자들의 유한 순서 집합
public:
   String(char *init, int m);
            // 길이 m인 문자열 init로 초기화하는 생성자
   int operator==(String t);   // if (문자열이 t와 동등하면)
            1(TRUE)을 반환,   else 0(FALSE)을 반환
   int Length();               //  스트링의 문자수를 반환
   String Concat(String t); // 뒤에 t를 붙인 문자열 반환
   String Substr(int i, int j); //  substring 반환
  int Find(String pat);      //  substring이 pat 와 부합시,  i 반환,
          else  -1 반환
```

*** 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;
```