

자료구조론

5장 순차 자료구조 방식

□ 이 장에서 다룰 내용

- ❖ 선형 리스트
- ❖ 선형 리스트의 구현
- ❖ 다항식의 순차 자료구조 표현
- ❖ 행렬의 순차 자료구조 표현

□ 선형 리스트

- ❖ 문제 해결을 위해서는 추상적으로 정의된 데이터와 연산을 구체적으로 구현해야 함
 - 연산의 구현은 데이터의 표현 방법에 따라 달라짐
 - 처리할 문제에 대해 데이터를 가장 효율적인 방법으로 표현하는 것이 중요
- ❖ 데이터를 구조화하는 가장 기본적인 방법은 데이터를 나열하는 것
 - 자료를 나열한 목록을 리스트(list)라고 한다.
 - 예) 동창의 이름을 나열 ➔ 동창 리스트

동창 리스트
김좌진
신채호
안중근
이봉창
한용운

□ 선형 리스트

❖ 선형 리스트(Linear List)

- 나열한 자료들 간에 앞뒤 관계가 1:1인 순서를 갖는 리스트
- 순서 리스트(Ordered List)
- 선형 리스트의 예

동창 선형 리스트	
1	김좌진
2	신채호
3	안중근
4	이봉창
5	한용운

□ 선형 리스트

❖ 선형 리스트의 표현 형식

리스트이름 = (원소 1, 원소 2, ..., 원소 n)

- 선형 리스트에서 원소를 나열한 순서는 원소들의 순서가 된다.
 - 예) 동창 선형 리스트
동창 = (김좌진, 신채호, 안중근, 이봉창, 한용운)
- 공백 리스트(Empty List) : 원소가 하나도 없는 리스트
 - 예) 친구 선형 리스트
친구 = ()

❖ 선형 리스트의 두 가지 구현 방식

- 순차(sequential) 자료구조 방식
- 연결(linked) 자료구조 방식 ➔ 6장에서 다룸

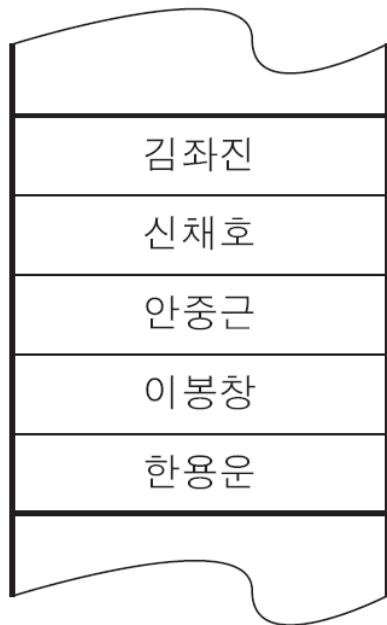
□ 선형 리스트 - 순차 자료구조

❖ 순차 자료구조

- 원소들의 논리적 순서와 같은 순서로 메모리에 저장

원소들의 논리적 순서 = 원소들이 저장된 물리적 순서

- 예) 동창 선형 리스트를 메모리에 순서대로 저장



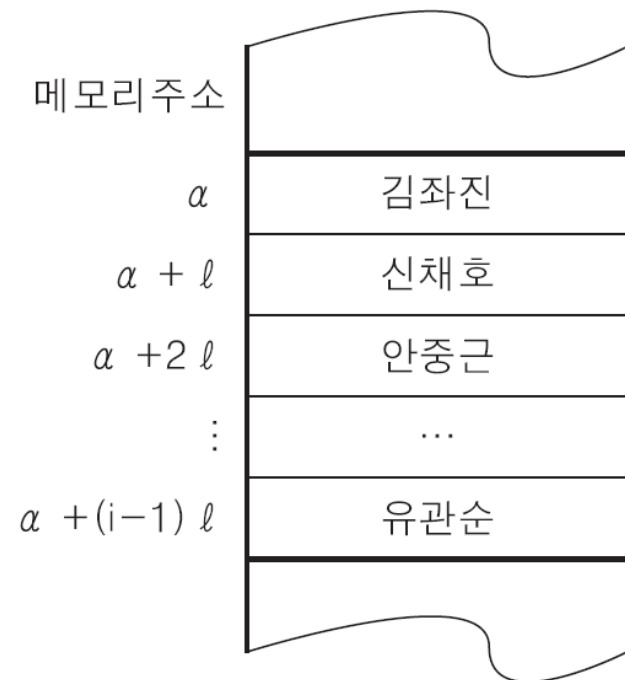
동창 = (김좌진, 신채호, 안중근, 이봉창, 한용운)

□ 선형 리스트 - 순차 자료구조

❖ 순차 자료구조에서는 원소 위치를 간단한 주소 계산으로 알아낼 수 있다.

- 선형 리스트가 저장된 시작 주소 : α
- 각 원소의 크기 : ℓ
- 1번째 원소의 주소 = α
- 2번째 원소의 주소 = $\alpha + \ell$
- 3번째 원소의 주소 = $\alpha + 2 \times \ell$

→ i 번째 원소의 주소 = $\alpha + (i-1) \times \ell$

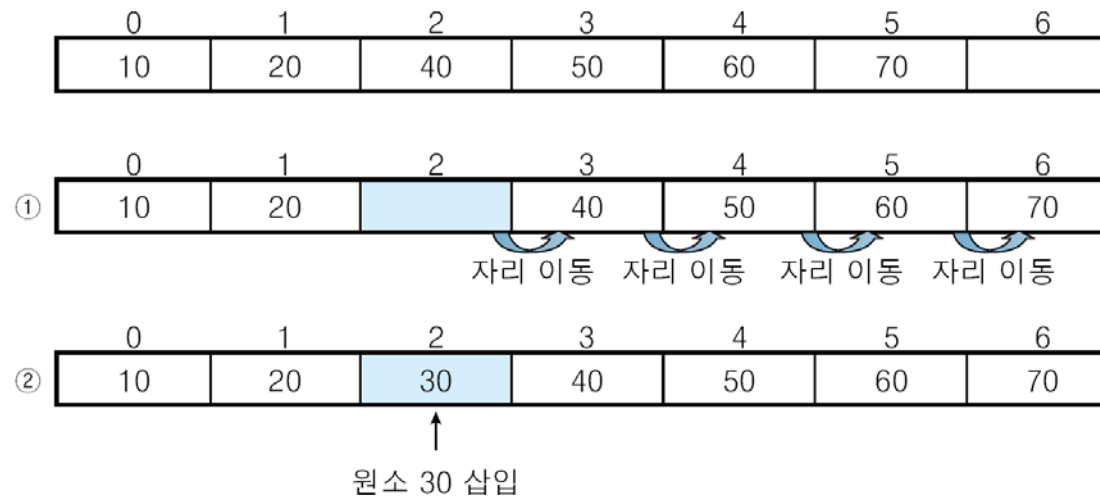


□ 선형 리스트 - 순차 자료구조

❖ 원소 삽입

- 선형 리스트 중간에 원소가 삽입되면, 그 이후의 원소들은 한자리씩 자리를 뒤로 이동하여 물리적 순서를 논리적 순서와 일치시킨다.

- ① 원소를 삽입할 빈 자리 만들기
- ② 준비한 빈 자리에 원소 삽입하기



- 원소 수가 n 일 때, k 번 자리에 삽입시 자료 이동횟수 = $n-k$
- 삽입 연산 수행 시간 = $O(n)$

□ 선형 리스트 - 순차 자료구조

❖ 원소 삭제

- 선형 리스트의 중간에서 원소가 삭제되면, 그 이후의 원소들을 한자리씩 앞으로 이동하여 물리적 순서를 논리적 순서와 일치시킨다.

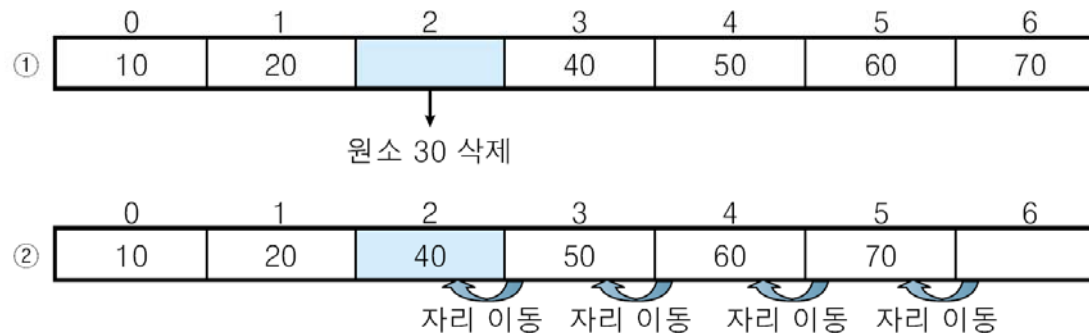
① 원소 삭제하기

② 삭제한 빈 자리 채우기

(a) 원소 삭제 전

0	1	2	3	4	5	6
10	20	30	40	50	60	70

(b) 원소 삭제 후



- 원소 수가 n 일 때, k 번 자리 삭제시 자료 이동횟수 = $n-1-k$
- 삭제 연산 수행 시간 = $O(n)$

□ 선형 리스트의 구현

❖ 순차 자료구조

- 배열을 사용하여 선형 리스트를 구현
 - 배열 - <인덱스, 원소>의 순서쌍의 집합
 - 배열의 인덱스 - 배열 원소의 순서
- 배열의 인덱스는 배열 원소의 순서
- 1차원 배열
 - 인덱스를 하나만 사용하여 원소의 순서를 구별할 수 있으면 1차원 배열을 사용하여 표현할 수 있다.
- 2차원 배열
 - 인덱스를 두 개 사용하여 원소의 순서를 구별할 수 있으면 2차원 배열을 사용하여 표현할 수 있다.
- 3차원 배열, ...

□ 선형 리스트의 구현

❖ 1차원 배열을 이용한 선형 리스트의 구현

- 예) 분기별 노트북 판매량

분기	1/4분기	2/4분기	3/4분기	4/4분기
판매량	157	209	251	312

- 1차원 배열을 이용한 구현

```
int[] sale = new int[4];  
sale[0] = 157;  
sale[1] = 209;  
sale[2] = 251;  
sale[3] = 312
```

int[] sale;
sale = new int[4];

또는

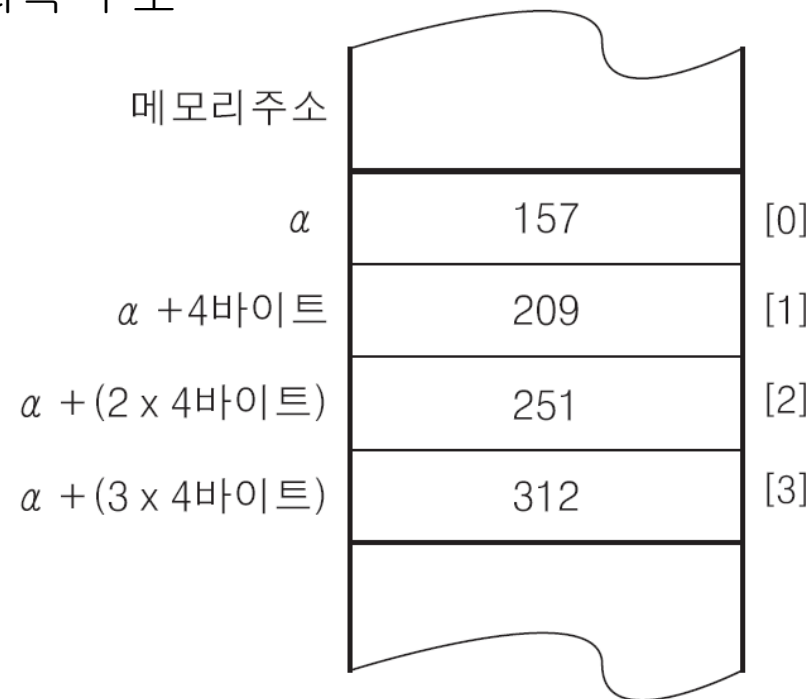
```
int[] sale = {157, 209, 251, 312};  
int[] sale = new int[] {157, 209, 251, 312};
```

□ 선형 리스트의 구현

➡ 논리적 구조

	[0]	[1]	[2]	[3]
sale	157	209	251	312

➡ 물리적 구조



□ 선형 리스트의 구현

- 분기별 판매량 선형 리스트 프로그램

```
public class Ex5_1 {  
    public static void main(String[] args) {  
        int[] sale = {157, 209, 251, 312};  
        for(int i=0; i<4; i++)  
            System.out.println(sale[i]);  
    }  
}
```

실행 결과 :

157
209
251
312

□ 선형 리스트의 구현

❖ 2차원 배열을 이용한 선형 리스트의 구현

- 예) 2007~2008년 분기별 노트북 판매량

년 \ 분기	1/4분기	2/4분기	3/4분기	4/4분기
2007년	63	84	140	130
2008년	157	209	251	312

- 2차원 배열을 이용한 구현
int[][] sale = new int[2][4];
sale[0][0] = 63;
sale[0][1] = 84;
...
sale[1][3] = 312;

또는

```
int[][] sale = { {63, 84, 140, 130},  
                 {157, 209, 251, 312} };
```

□ 선형 리스트의 구현

➡ 논리적 구조

	[0]	[1]	[2]	[3]
sale [0]	63	84	140	130
[1]	157	209	251	312

➡ 물리적 구조

물리순서		논리순서	
0	63	[0][0]	행0
1	84	[0][1]	
2	140	[0][2]	
3	130	[0][3]	
4	157	[1][0]	행1
5	209	[1][1]	
6	251	[1][2]	
7	312	[1][3]	

(a) 행 우선 순서 방법

물리순서		논리순서	
0	63	[0][0]	열0
1	157	[1][0]	
2	84	[0][1]	열1
3	209	[1][1]	
4	140	[0][2]	열2
5	251	[1][2]	
6	130	[0][3]	열3
7	312	[1][3]	

(b) 열 우선 순서 방법

□ 선형 리스트의 구현

❖ 2차원 배열의 물리적 저장 방법

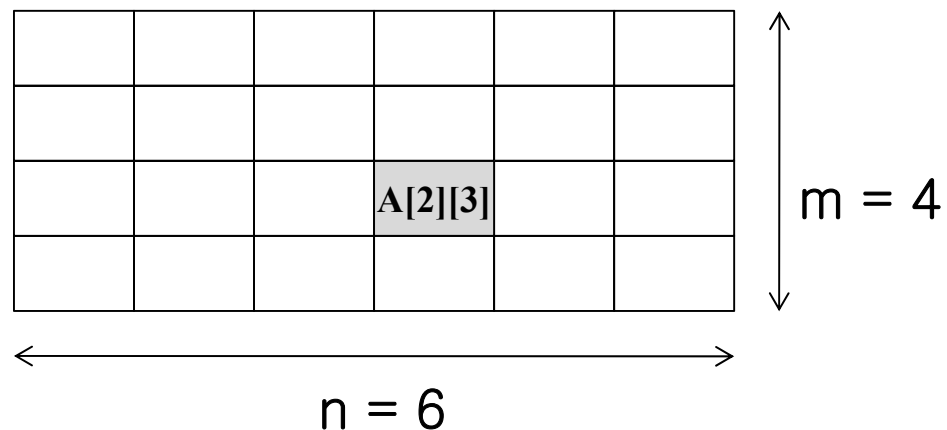
- 행의 개수가 m 이고 열의 개수가 n 인 2차원 배열 A 의 시작주소가 α 이고 각 원소의 크기가 ℓ 일 때, $A[i][j]$ 의 위치는?

- 행 우선 순서 방법(row-major order)

$$\alpha + (i \times n + j) \times \ell$$

- 열 우선 순서 방법(column-major order)

$$\alpha + (j \times m + i) \times \ell$$



□ 선형 리스트의 구현

- 2007, 2008년 분기별 판매량 선형 리스트 프로그램

```
public class Ex5_2 {  
    public static void main(String[] args) {  
        int[][] sale = { {63, 84, 140, 130},  
                          {157, 209, 251, 312} };  
        for(int i=0; i<2; i++) {  
            for(int j=0; j<4; j++)  
                System.out.println(sale[i][j]);  
            System.out.println();  
        }  
    }  
}
```

실행 결과 :

63
84
140
130

157
209
251
312

□ 선형 리스트의 구현

❖ 3차원 배열을 이용한 선형 리스트의 구현

- 예) 2007~2008년, 1팀과 2팀의 분기별 노트북 판매량

1팀				
년 \ 분기	1/4분기	2/4분기	3/4분기	4/4분기
2007년	63	84	140	130
2008년	157	209	251	312

2팀				
년 \ 분기	1/4분기	2/4분기	3/4분기	4/4분기
2007년	59	80	130	135
2008년	149	187	239	310

□ 선형 리스트의 구현

- 3차원 배열을 이용한 구현

```
int[][][] sale = new int[2][2][4];
```

```
sale[0][0][0] = 63;
```

```
sale[0][0][1] = 84;
```

...

또는

```
int[][][] sale = { {{63, 84, 140, 130}, {157, 209, 251, 312}},  
                   {{59, 80, 130, 135}, {149, 187, 239, 310}} };
```

➡ 논리적 구조



□ 선형 리스트의 구현

➡ 물리적 구조

물리 순서		논리 순서	
0	63	[0][0][0]	면0
1	84	[0][0][1]	
2	140	[0][0][2]	
3	130	[0][0][3]	
4	157	[0][1][0]	
5	209	[0][1][1]	
6	251	[0][1][2]	
7	312	[0][1][3]	
8	59	[1][0][0]	면1
9	80	[1][0][1]	
10	130	[1][0][2]	
11	135	[1][0][3]	
12	149	[1][1][0]	
13	187	[1][1][1]	
14	239	[1][1][2]	
15	310	[1][1][3]	

(a) 면 우선 순서 방법

물리 순서		논리 순서	
0	63	[0][0][0]	열0
1	59	[1][0][0]	
2	157	[0][1][0]	
3	149	[1][1][0]	
4	84	[0][0][1]	열1
5	80	[1][0][1]	
6	209	[0][1][1]	
7	187	[1][1][1]	
8	140	[0][0][2]	열2
9	130	[1][0][2]	
10	251	[0][1][2]	
11	239	[1][1][2]	
12	130	[0][0][3]	열3
13	135	[1][0][3]	
14	312	[0][1][3]	
15	310	[1][1][3]	

(b) 열 우선 순서 방법

□ 선형 리스트의 구현

❖ 3차원 배열의 물리적 저장 방법

- 면의 개수가 n_i , 행의 개수가 n_j , 열의 개수가 n_k 인 3차원 배열 A의 시작주소가 α 이고 각 원소의 크기가 ℓ 일 때, **$A[i][j][k]$** 의 위치는?

- 면 우선 순서 방법

$$\alpha + \{(i \times n_j \times n_k) + (j \times n_k) + k\} \times \ell$$

- 열 우선 순서 방법

$$\alpha + \{(k \times n_j \times n_i) + (j \times n_i) + i\} \times \ell$$

□ 선형 리스트의 구현

- 1, 2팀의 2007, 2008년 분기별 판매량 선형 리스트 프로그램

```
public class Ex5_3{
    public static void main(String[] args) {
        int[][][] sale = { { {63, 84, 140, 130}, {157, 209, 251, 312} },
                           { {59, 80, 130, 135}, {149, 187, 239, 310} } };

        for(int i=0; i<2; i++) { // 팀
            for(int j=0; j<2; j++) { // 연도
                for(int k=0; k<4; k++) { // 분기
                    System.out.println(sale[i][j][k]);
                }
            }
        }
    }
}
```

□ 다항식의 순차 자료구조 구현

❖ 다항식 (polynomial)

- aX^e 형식의 항들의 합으로 구성된 식
 - a : 계수 (coefficient)
 - X : 변수 (variable)
 - e : 지수 (exponent)
- 예) $P(X) = 4X^3 + X^2 - 2.6X + 7$
- 지수에 따라 내림차순으로 항을 나열
- 다항식의 차수 = 가장 큰 지수값
- 다항식 항의 최대 개수 = (차수 + 1)개

□ 다항식의 순차 자료구조 구현

ADT Polynomial

데이터 : 지수(e_i)-계수(a_i)의 순서쌍 $\langle e_i, a_i \rangle$ 의 집합으로 표현된 다항식

$$p(X) = a_0X^{e_0} + a_1X^{e_1} + \dots + a_iX^{e_i} + \dots + a_nX^{e_n} \text{ (} e_i \text{는 음이 아닌 정수)}$$

연산 : $p, p_1, p_2 \in \text{Polynomial}; \quad a \in \text{Coefficient}; \quad e \in \text{Exponent};$

// p, p_1, p_2 는 다항식이고, a 는 계수, e 는 지수를 나타낸다.

$\text{zeroP}() ::= \text{return polynomial } p(X)=0;$

// 공백 다항식($p(x)=0$)을 만드는 연산

$\text{isZeroP}(p) ::= \text{if } (p) \text{ then false}$
 $\quad \text{else return true};$

// 다항식 p 가 0(공백 다항식)인지 아닌지 검사하여 0이면 true 를 반환하는 연산

$\text{coef}(p, e) ::= \text{if } (\langle e, a \rangle \in p) \text{ then return } a$
 $\quad \text{else return } 0;$

// 다항식 p 에서 지수가 e 인 항의 계수 a 를 구하는 연산. 지수가 e 인 항이 없으면 0 반환

□ 다항식의 순차 자료구조 구현

maxExp(p) ::= return max(p.Exponent);

// 다항식 p에서 최대 지수를 구하는 연산

addTerm(p,a,e) ::= if ($e \in p.Exponent$) then return error

else return p after inserting the term $\langle e,a \rangle$;

// 다항식 p에 지수가 e인 항이 없는 경우에 새로운 항 $\langle e,a \rangle$ 를 추가하는 연산

delTerm(p,e) ::= if ($e \in p.Exponent$) then return p after removing the term $\langle e,a \rangle$

else return error;

// 다항식 p에서 지수가 e인 항 $\langle e,a \rangle$ 를 삭제하는 연산

multTerm(p,a,e) ::= return ($p * aX^e$);

// 다항식 p의 모든 항에 aX^e 항을 곱하는 연산

addPoly(p1,p2) ::= return ($p1 + p2$);

// 두 다항식 p1과 p2의 합을 구하는 연산

multPoly(p1,p2) ::= return ($p1 * p2$);

// 두 다항식 p1과 p2의 곱을 구하는 연산

End Polynomial

□ 다항식의 순차 자료구조 구현

❖ 다항식의 표현

- 각 항의 지수와 계수의 쌍에 대한 선형 리스트

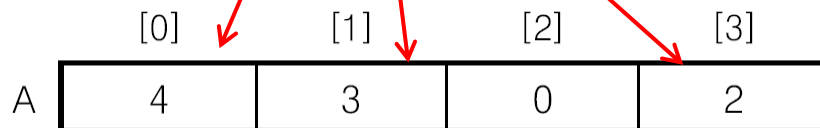
- 예) $A(x) = 4x^3 + 3x^2 + 2$

$$p = ((3,4), (2,3), (0,2))$$

❖ 1차원 배열을 이용한 다항식 표현

- 차수가 n 인 다항식을 $(n+1)$ 개의 원소를 가지는 1차원 배열로 표현
- 배열 인덱스 i 위치에 지수 $(n-i)$ 인 항의 계수 저장

- 예) $A(x) = 4x^3 + 3x^2 + 2$



□ 다항식의 순차 자료구조 구현

❖ 희소 다항식에 대한 1차원 배열 저장

- 예) $B(x) = 3x^{1000} + x + 4$

	[0]	[1]	[2]	[3]	...	[997]	[998]	[999]	[1000]
B	3	0	0	0	...	0	0	1	4

- 1001개의 배열 원소 중 3개만 사용하므로 메모리 낭비

❖ 2차원 배열을 이용한 다항식 표현

- 다항식의 각 항에 대한 (지수, 계수)의 쌍을 2차원 배열에 저장

- 예) $B(x) = 3x^{1000} + x + 4$ 의 2차원 배열 표현

	[0]	[1]	
[0]	1000	3	☞ $3x^{1000}$
[1]	1	1	☞ x
[2]	0	4	☞ 4

- 연산 구현은 복잡하지만,
- 희소 다항식의 경우, 1차원 배열 구현보다 메모리 필요량 감소

□ 다항식의 순차 자료구조 구현

```
addPoly(A, B) // 다항식의 덧셈 알고리즘: A와 B를 더하여 결과 다항식 C를 반환
  C ← zeroP();
  while (not isZeroP(A) and not isZeroP(B)) do {
    case {
      maxExp(A) < maxExp(B) :
        C ← addTerm(C, coef(B, maxExp(B)), maxExp(B));
        B ← delTerm(B, maxExp(B));
      maxExp(A) = maxExp(B) :
        sum ← coef(A, maxExp(A)) + coef(B, maxExp(B));
        if (sum ≠ 0) then C ← addTerm(C, sum, maxExp(A));
        A ← delTerm(A, maxExp(A));
        B ← delTerm(B, maxExp(B));
      maxExp(A) > maxExp(B) :
        C ← addTerm(C, coef(A, maxExp(A)), maxExp(A));
        A ← delTerm(A, maxExp(A));
    }
  }
  if (not isZeroP(A)) then A의 나머지 항들을 C에 복사
  else if (not isZeroP(B)) then B의 나머지 항들을 C에 복사
  return C;
End addPoly()
```

□ 다항식의 순차 자료구조 구현

- 다항식의 덧셈 프로그램

```
public class Ex5_4{
    public static void main(String[] args) {
        double[] coefArrayA = {9, 3, 5, 0};
        double[] coefArrayB = {8, 1, -3, 2.5, 1.8};

        Polynomial a = new Polynomial(3, coefArrayA);
        Polynomial b = new Polynomial(4, coefArrayB);
        Polynomial c = a.addPoly(b);

        System.out.println("A(x)=" + a);
        System.out.println("B(x)=" + b);
        System.out.println("C(x)=" + c);
    }
}
```

□ 다항식의 순차 자료구조 구현

```
class Polynomial {  
    // 1차원 배열을 이용한 다항식의 표현  
  
    private int degree;  
    private double[] coef;  
  
    public Polynomial(int degreeValue,  
                      double[] coefArray){  
        degree = degreeValue;  
        coef = coefArray;  
    }  
  
    public Polynomial(int degreeValue){  
        degree = degreeValue;  
        coef = new double[degree+1];  
        for(int i=0; i<=degree; i++)  
            coef[i] = 0;  
    }  
  
    public int getDegree(){  
        return degree;  
    }  
  
    public double getCoef(int expo){  
        return coef[expo];  
    }  
}
```

```
    public void setCoef(int expo, double coefValue){  
        coef[expo] = coefValue;  
    }  
  
    public Polynomial addPoly(Polynomial b) {  
        ...  
        Polynomial c ;  
        ...  
        return c;  
    }  
  
    public void addTerm(int expo, double coefValue) {  
        ...  
    }  
  
    public void delTerm(int expo) {  
        ...  
    }  
}
```

□ 다항식의 순차 자료구조 표현

- 다항식의 덧셈 실행 결과 $C(x) = A(x) + B(x)$
 - $A(x) = 9x^3 + 3x^2 + 5x$
 - $B(x) = 8x^4 + x^3 - 3x^2 + 2.5x + 1.8$
 - $C(x) = 8x^4 + 10x^3 + 7.5x + 1.8$

□ 행렬의 순차 자료구조 표현

❖ 행렬(matrix)

- $m \times n$ 행렬
 - m : 행의 개수
 - n : 열의 개수
 - 원소의 개수 : $(m \times n)$ 개

$$A = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{vmatrix}$$

- 정방 행렬(square matrix) : 행과 열의 개수가 같은 행렬

□ 행렬의 순차 자료구조 표현

❖ 전치 행렬

- 행렬의 행과 열을 서로 교환하여 구성한 행렬
- 행렬 A의 모든 원소의 위치 (i, j)를 (j, i)와 교환
- m×n 행렬 A의 전치행렬은 n×m 행렬 A'

$$A = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{vmatrix}$$

$$A' = \begin{vmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{vmatrix}$$

- 예)

$$A = \begin{vmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{vmatrix} \quad \text{전치행렬 변환} \longrightarrow \quad A' = \begin{vmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{vmatrix}$$

□ 행렬의 순차 자료구조 표현

❖ 2차원 배열로 행렬을 표현하는 방법

- $m \times n$ 행렬을 m 행 n 열의 2차원 배열로 표현

- 예)

$$A = \begin{vmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{vmatrix} \quad \rightarrow \quad \begin{matrix} & [0] & [1] & [2] & [3] \\ A[3][4] & [0] & [1] & [2] & [3] \\ & [0] & [1] & [2] & [3] \\ & [1] & [2] & [3] & [4] \\ & [2] & [3] & [4] & [5] \end{matrix}$$

(a) 3×4 행렬 A와 배열 A[3][4]

$$B = \begin{vmatrix} 0 & 0 & 2 & 0 & 0 & 0 & 12 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 \\ 23 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 31 & 0 & 0 & 0 \\ 0 & 14 & 0 & 0 & 0 & 25 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 \\ 52 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 11 & 0 & 0 \end{vmatrix} \quad \rightarrow \quad \begin{matrix} & [0] & [1] & [2] & [3] & [4] & [5] & [6] \\ B[8][7] & [0] & [1] & [2] & [3] & [4] & [5] & [6] \\ & [0] & [1] & [2] & [3] & [4] & [5] & [6] \\ & [1] & [2] & [3] & [4] & [5] & [6] & [7] \\ & [2] & [3] & [4] & [5] & [6] & [7] & [8] \\ & [3] & [4] & [5] & [6] & [7] & [8] & [9] \\ & [4] & [5] & [6] & [7] & [8] & [9] & [10] \\ & [5] & [6] & [7] & [8] & [9] & [10] & [11] \\ & [6] & [7] & [8] & [9] & [10] & [11] & [12] \\ & [7] & [8] & [9] & [10] & [11] & [12] & [13] \end{matrix}$$

(b) 8×7 행렬 B와 배열 B[8][7] --- B는 희소 행렬 : 배열의 원소 56개 중에 실제 사용하는 것은 0이 아닌 원소를 저장하는 10개 뿐임

□ 행렬의 순차 자료구조 표현

- 희소 행렬(sparse matrix)에 대한 2차원 배열 표현
 - 크기가 $m \times n$ 인 2차원 배열에 저장하는 경우 메모리 낭비
 - 0이 아닌 원소만 <행번호, 열번호, 원소>쌍으로 배열에 저장하면 필요한 메모리를 줄일 수 있다.
 - 행렬의 <전체 행의 개수, 전체 열의 개수, 0이 아닌 원소의 개수> 정보를 0번 행에 저장

B[8][7]

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	
[0]	0	0	2	0	0	0	12	<0, 2, 2>
[1]	0	0	0	0	7	0	0	<0, 6, 12>
[2]	23	0	0	0	0	0	0	<1, 4, 7>
[3]	0	0	0	31	0	0	0	<2, 0, 23>
[4]	0	14	0	0	0	25	0	<3, 3, 31>
[5]	0	0	0	0	0	0	6	<4, 1, 14>
[6]	52	0	0	0	0	0	0	<4, 5, 25>
[7]	0	0	0	0	11	0	0	<5, 6, 6>
								<6, 0, 52>
								<7, 4, 11>



	[0]	[1]	[2]
[0]	8	7	10
[1]	0	2	2
[2]	0	6	12
[3]	1	4	7
[4]	2	0	23
[5]	3	3	31
[6]	4	1	14
[7]	4	5	25
[8]	5	6	6
[9]	6	0	52
[10]	7	4	11

□ 행렬의 순차 자료구조 표현

- 희소 행렬의 추상 자료형

ADT SparseMatrix

데이터 : 3원소쌍 <행 인덱스, 열 인덱스, 원소 값>의 집합

연산 : $a, b \in \text{SparseMatrix}$; $c \in \text{Matrix}$; $u, v \in \text{value}$; $i \in \text{Row}$; $j \in \text{Column}$;

// a와 b는 희소행렬, c는 행렬, u와 v는 행렬의 원소값, i와 j는 행 인덱스와 열 인덱스

$\text{createSM}(m, n) ::= \text{return an empty sparse matrix with } m \times n$;

// $m \times n$ 의 공백 희소행렬을 만드는 연산

$\text{transposeSM}(a) ::= \text{return } c \text{ where } c[j, i] = v \text{ when } a[i, j] = v$;

// $a[i, j] = v$ 를 $c[j, i] = v$ 로 전치시킨, 전치행렬 c를 구하는 연산

$\text{addSM}(a, b) ::= \text{if } (a.\text{dimension} = b.\text{dimension})$

then return c where $c[i, j] = v + u$ where $a[i, j] = v$ and $b[i, j] = u$

else return error;

// 차원이 같은 희소행렬 a와 b를 합한 행렬 c를 구하는 연산

$\text{multiSM}(a, b) ::= \text{if } (a.n = b.m) \text{ then return } c \text{ where } c[i, j] = a[i, k] \times b[k, j]$;

else return error;

// a의 열의 개수(n)와 b의 행의 개수(m)가 같은 경우에 두 행렬의 곱 c를 구하는 연산

End SparseMatrix

□ 행렬의 순차 자료구조 표현

```
transposeSM(a[]) // 희소행렬의 전치 연산 알고리즘
  m ← a[0,0]; // 희소행렬 a의 행 수
  n ← a[0,1]; // 희소행렬 a의 열 수
  v ← a[0,2]; // 희소행렬 a에서 0이 아닌 원소 수
  b[0,0] ← n; // 전치행렬 b의 행 수 지정
  b[0,1] ← m; // 전치행렬 b의 열 수 지정
  b[0,2] ← v; // 전치행렬 b의 0이 아닌 원소 수 지정
  if (v > 0) then { // 0이 아닌 원소가 있는 경우에만 전치 연산 수행
    p ← 1;
    for (i ← 0; i < n; i ← i+1) do { // 희소행렬 a의 열별로 전치 반복 수행
      for (j ← 1; j ≤ v; j ← j+1) do { // 0이 아닌 원소 수에 대해서만 반복 수행
        if (a[j,1]=i) then { // 현재의 열에 속하는 원소가 있으면 b[]에 삽입
          b[p,0] ← a[j,1];
          b[p,1] ← a[j,0];
          b[p,2] ← a[j,2];
          p ← p+1;
        }
      }
    }
  }
  return b[];
End transposeSM()
```