3장 순차 데이타 표현

순서

데이타의 표현

- **데이타와 연산자의 표현**
  - 고급적 표현 - 추상적, 논리적
  - 저급의 데이타와 연산자로 구현해야 실행될 수 있음
  - 연산자의 구현은 데이타의 표현 방법에 크게 의존

- **선형 리스트의 저급 표현의 분류**
  - 선형 리스트: 순서를 가진 원소들의 순열
  - 순차 표현 (sequential representation)
  - 연결 표현 (linked representation)

배열과 인덱스

- 배열의 특성
  - 순차적 메모리 할당 방식
  - <인덱스, 원소> 쌍의 집합 – 각 쌍은 인덱스와 그와 연관된 원소값의 대응 관계를 나타냄
  - 원소들이 모두 같은 타입, 같은 크기

- 인덱스 : 순서를 나타내는 원소의 유한 집합
  - 집합 내에서의 상대적 위치 식별
  - 원소 수가 한정되어 있어 항상 마지막 원소가 존재
  - 인덱스만으로 원하는 원소에 직접 접근이 가능 (정보 은닉 – 내부 구현의 문제는 사용자가 알 필요 없음)

배열 추상 데이타 타입 (1)

ADT Array
데이타 : <i∈Index, e∈Element> 쌍들의 집합. 여기서 Index는 순서를 나타내는 원소의 유한집합이고 Element는 타입이 같은 원소의 집합

연산 : a∈Array; i∈Index; x, e∈Element; n∈Integer;
create(n) ::= **return** an array to hold n elements;
retrieve(a,i)::= **if** (i∈Index) **then return** e where <i, e>∈a **else return** error;
store(a, i, e) ::= **if** (i∈Index) **then return** (a∪<i, e>) **else return** error;
End Array

배열 추상 데이타 타입 (2)

- 연산자
  - 연산자 create : n개의 원소들을 저장할 수 있는 공백 배열의 생성자
  - 연산자 retrieve : Array와 Index를 매개 변수로 받아 Index에 대응하는 원소를 찾아 반환하고 없으면 error를 반환
  - 연산자 store : Array, Index, Element를 매개 변수로 전달 받아 유효하면 <인덱스, 원소> 쌍이 되게 저장하고 Array를 반환

- 배열 ADT 정의의 이점
  - 배열에 포함되는 데이타와 연산들을 명확하게 정의
  - 배열의 본질을 더 정확하게 표현

배열의 표현

- 인덱스
  - 배열 내에서 원소의 상대적 위치를 나타냄
  - 인덱스가 하나의 값으로 표현되면 1차원, n개의 값으로 표현되면 n차원 배열이라 함

- **배열 a는 인덱스와 원소의 쌍(<i, v>)의 집합으로 정의**
  - a[i] : 인덱스 i에 대응하는 원소 v의 주소
  - <i, v>∊a 이면 a[i]=v
  - a[i] ← v : a[i]는 v가 저장될 주소
  - k ← a[i] : a[i]는 변수 k에 저장시킬 값을 검색해 올 주소, retrieve(a,i) 연산과 같은 기능 수행

1차원 배열

- **1차원 배열의 선언 : a[n]**
  - a : 배열 이름, n : 원소의 최대 수, 인덱스는 {0, 1, ..., n-1}
- **배열의 순차 표현(sequential representation)**
  - 연속적인 메모리 주소를 배열에 할당
  - 원소는 할당된 메모리 내에서 인덱스 순서에 따라 저장
  - 예) 1차원 배열의 순차 표현
    - a[0]의 주소(기준 주소: base address)가 α이면, a[i]의 주소는 α + i
    - 각 원소가 c개의 워드를 필요로 할 경우, a[i]의 주소는 α + c•i

주소	배열 a의 원소
α	a[0]
α+1	a[1]
. . .	. . .
α+i	a[i]
. . .	. . .
α+n-1	a[n-1]

2차원 배열 (1)

- **2차원 배열의 선언 : a[n_1, n_2]**
  - n_1 : 행(row)의 수, n_2 : 열(column)의 수, 원소 수 : $n_1 \cdot n_2$
  - 원소 : <행 인덱스(i), 열 인덱스(j)>를 기초로 a[i, j]로 표현

- **2차원 배열을 1차원 메모리로 사상**
  - 행 우선 순서 (row major order)
  - 열 우선 순서 (column major order)

- **a[2, 3]의 행 우선 순서의 표현**
  - 3개의 원소로 된 2개의 행을 행 번호에 따라 한 줄로 연결
  - 즉, (<a[0, 0], a[0, 1], a[0, 2]>, <a[1, 0], a[1, 1], a[1, 2]> 순서
  - 열을 나타내는 오른편 인덱스가 항상 먼저 변함
  - a[0, 0]의 주소가 α라 할 때, 원소 a[1, 0]의 주소는 $\alpha + 1 \cdot 3 + 0 = \alpha + 3$이 됨
    - 원소의 행 인덱스 값 : 1, 배열의 열 수 : 3, 원소의 열 인덱스 값 : 0

2차원 배열 (2)

◆ 2차원 배열 a[n_1, n_2]

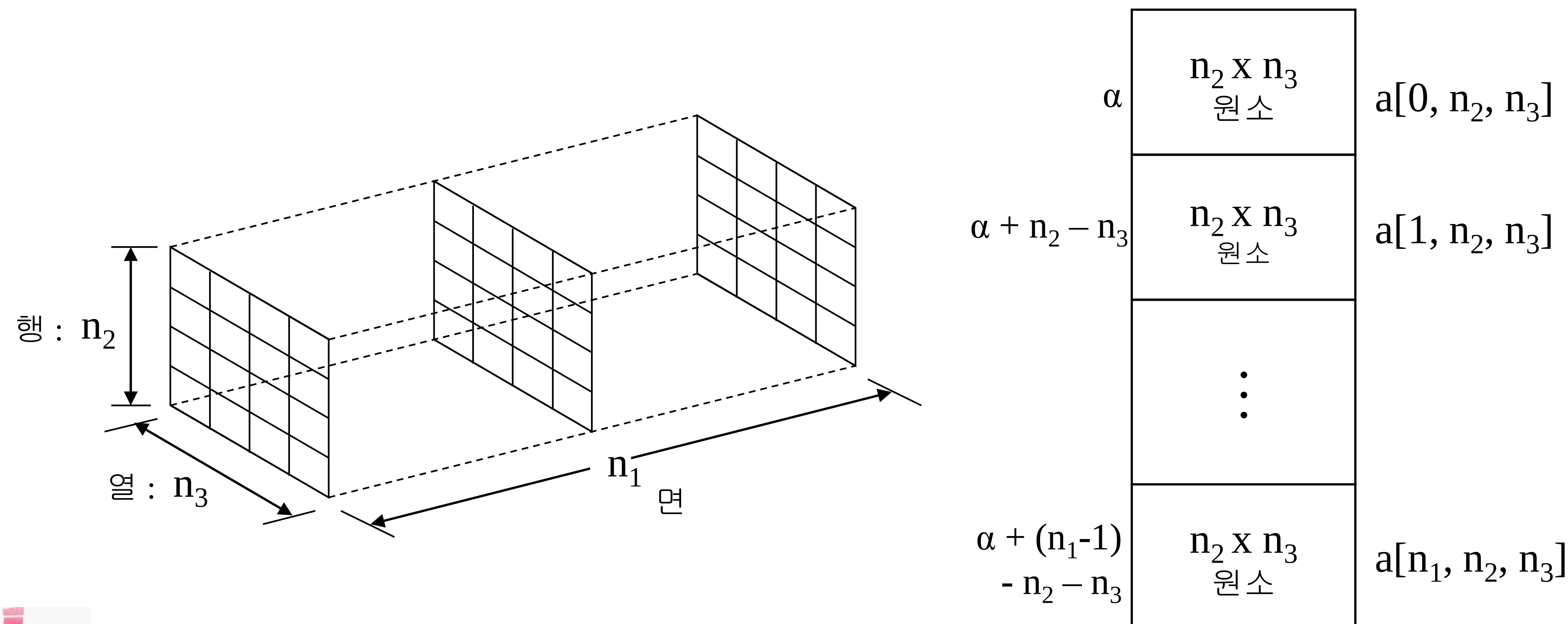
- a[0, 0]의 주소가 α일 때, 원소 a[i_1,i_2]의 주소 : $\alpha + i_1 \cdot n_2 + i_2$

	열0	열1	…	열n_2-1
행0	a[0,0]	a[0,1]	…	a[0, n_2-1]
행1	a[1,0]	a[1,1]	…	a[1, n_2-1]
행2	a[2,0]	a[2,1]	…	a[2, n_2-1]
	⋮	⋮	…	⋮
행n_1-1	a[n_1-1,0]	a[n_1-1,1]	…	a[n_1-1, n_2-1]

α	a[0,0]	
	⋮	행0
	a[0, n_2-1]	
α + n_2	a[1,0]	
	⋮	행1
	a[1, n_2-1]	
	⋮	
	a[n_1-10,0]	
	⋮	행n_1-1
α + $n_2 n_2$-1	a[n_1-1,n_2-1]	

3차원 배열

- **3차원 배열 $a[n_1, n_2, n_3]$: <면, 행, 열>**
  - n_1개의 2차원 배열(크기가 $n_2 X n_3$)을 차례로 1차원 메모리에 순차적으로 사상
  - a[0, 0, 0]의 주소를 α라 할 때, $a[i_1, i_2, i_3]$의 주소 : $\alpha + i_1 \cdot n_2 \cdot n_3 + i_2 \cdot n_3 + i_3$



Java에서의 배열 (1)

◆ Java의 배열

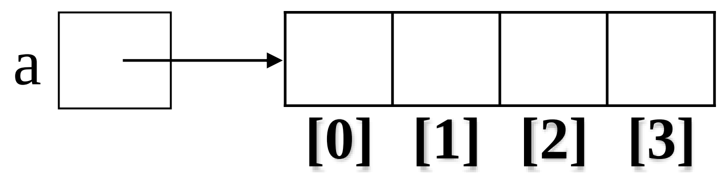
- 일정수의 컴포넌트를 순차적으로 정렬시킨 것
- 정수(int), 불리언(boolean), 문자(char), 부동소수(double) 등 원시 타입은 물론, 객체 타입의 배열도 허용
- 배열 변수는 객체를 참조하는 참조 변수와 똑같음

◆ 배열의 선언

- 예) 정수 배열 선언 - int[] a;

Java에서의 배열 (2)

- 배열의 생성
  - new 연산자
  - 예) int[] a;
    a = new int[4];



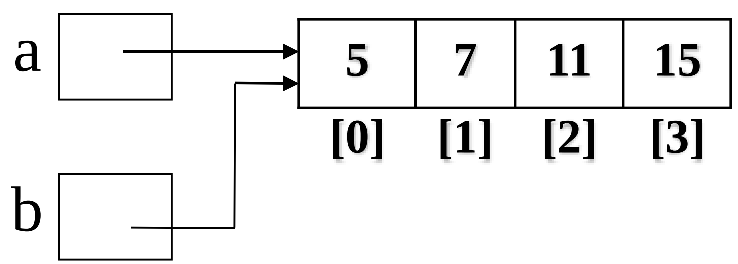
- 배열의 인스턴스 변수 length
  - 모든 배열이 생성될 때 내부적으로 가지게 되는 변수
  - 원소수를 표현
  - 예) a = new int[4]; 실행 뒤 a.length는 4가 됨
  - int[] a; 선언 후에는 a.length를 사용하면 a에 대한 객체가 생성되지 않았기 때문에 a는 null이 되어 length 접근 불가. 시스템은 에러 메시지를 생성

Java에서의 배열 (3)

◆ **배열 참조**

- 배열 변수 a, b에 대해 b = a; 가 실행되면 b는 a가 참조하고 있는 배열을 똑같이 참조. a == b는 true가 됨
- 예

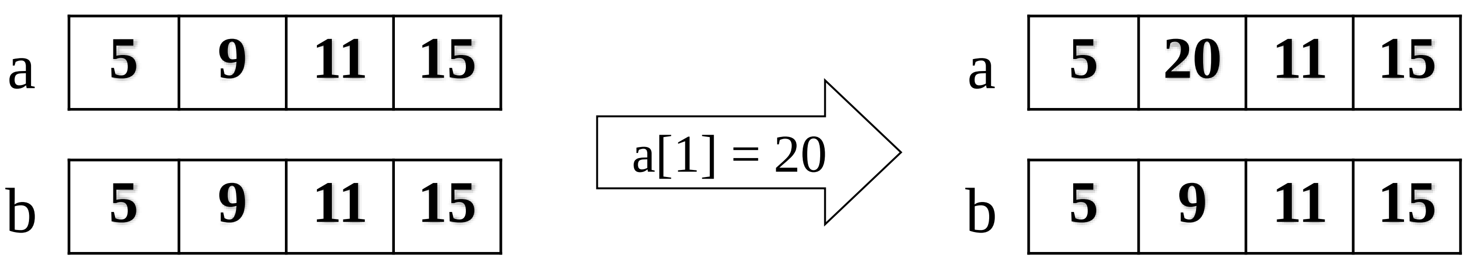
```
int[] a;
int[] b;
a = new int[4];   // 4개의 정수 원소에 대한 메모리 할당
a[0] = 5;   a[1] = 7;   a[2] = 11;   a[3] = 15;
b = a;
```



- 배열a 원소의 변경은 다른 배열 변수 b에 영향을 줌

Java에서의 배열 (4)

- 배열 복제
  - Java에서는 객체를 복제할 수 있게 메소드 clone()을 제공
  - 예) b = (int[]) a.clone();
    - (int[]) : clone() 메소드가 복제한 배열이 정수 배열이라는 것을 명세
    - 이제 두 개의 배열이 만들어졌기 때문에 원소의 변경은 서로 영향을 주지 않음.



선형 리스트

- **선형 리스트 (linear list)**
  - 순서를 가진 원소들의 순열(sequence)
  - 물리적 순서가 아닌 원소의 특성에 의한 논리적 순서를 의미
  - 리스트는 기본적으로 순서 개념을 가지므로 선형 리스트라고 볼 수 있음

- **리스트 $L=(e_1, e_2, \ldots , e_n)$**
  - L은 리스트 이름, e_i는 리스트 원소
  - 공백 리스트(empty list, 원소가 하나도 없는 리스트)의 표현 : L=( )
  - 리스트의 각 원소는 선행자(predecessor)와 후속자(successor)를 가짐
  - 예
    - 자료 구조 강의 요일 = (월요일, 수요일, 금요일)
    - 토요일 강의 과목 = ()

리스트 ADT 구현을 위한 연산

- **createList():** 초기에 공백리스트 **L**을 생성
- **isEmpty(L):** 리스트 **L**이 공백인지 아닌지 결정
- **length(L):** 리스트 **L**의 길이를 계산함. 여기서 리스트 길이는 리스트에 포함된 원소의 수. 공백리스트의 길이는 **0**
- **retrieve(L, i):** 리스트 **L**의 **i**번째 원소를 검색 **(1≤i≤L**의 길이**)**
- **replace(L, x, y):** 리스트 **L**의 원소 **x**를 새로운 원소 **y**로 대체
- **delete(L, x):** 공백이 아닌 리스트 **L**로부터 원소 **x**를 제거. 이 때 리스트 **L**의 길이는 하나 감소
- **insert(L, i, x):** 새로운 원소 **x**를 리스트 **L**의 지정된 위치 **i**에 삽입. 이 때 리스트 원소 $e_i, e_{i+1}, \ldots, e_n$은 $e_{i+1}, \ldots, e_n, e_{n+1}$로 되고, 리스트 **L**의 길이는 하나 증가

리스트의 표현

- **배열을 사용해 표현 (순차 표현 리스트)**
  - 리스트 원소 e_i와 e_{i+1}이 인덱스 i-1과 i에 대응되게 연속적으로 저장
  - 원소의 물리적 순서로 논리적 순서를 나타냄 (순서를 표시하기 위한 특별한 장치가 필요 없음)
  - 삽입, 삭제시에 후속 원소들을 한자리씩 밀거나 당겨야 하는 오버헤드가 치명적인 약점

$$L = (e_1, e_2, \ldots, e_n)$$

e_1	e_2	e_3	…	e_n
L[0]	L[1]	L[2]		L[n-1]

List class (Array 이용)

List.java

ListTest.java

다항식 추상 데이타 타입 (1)

- 다항식
  - 수학적으로 ax^e형식을 가진 항(term)의 합으로 정의
- 다항식 추상 데이타 타입

ADT Polynomial
데이타 : 지수-계수의 순서 쌍 <ei∈Exponent, ai∈Coefficient>의 집합으로 표현된
    다항식 $p(x)=a_0x^{e0}+a_1x^{e1}+\ldots+a_nx^{en}$.. 여기서 e_i는 음이 아닌 정수.

연산 : $p, p_1, p_2 \in$Polynomial; $a \in$Coefficient; $e \in$Exponent;
    zeroP() ::= **return** polynomial p(x)=0;
    isZeroP(p) ::= **if** (p) then **return false**
        else return true;
    coef(p, e) ::= **if (**<e, a>∈p) **then return** a
        else return 0;

다항식 추상 데이타 타입 (2)

maxExp(p) ::= **return** max(p.Exponent);
addTerm(p, a, e) ::= **if** (e∈p.Exponent) **then return** error
else return p after inserting the term <e, a>;
delTerm(p,e) ::= **if**(e∈p.Exponent) **then return** p after removing the term <e, a>
else return error;
sMult(p, a, e) ::= **return** ($p*ax^e$);
polyAdd(p1, p2) ::= **return** ($p_1 + p_2$);
polyMult(p1, p2) ::= **return** ($p_1 * p_2$);

End Polynomial


Poly.java

다항식 연산의 구현

- 다항식 생성
  - 모든 다항식은 zeroP()와 addTerm() 연산을 통해 생성됨
  - 예) $2x^3 + 4x^2 + 5$
    - addTerm(addTerm(addTerm(zeroP(), 2, 3), 4, 2), 5, 0)

```
p = zeroP();
p.addTerm(2, 3);
p.addTerm(4, 2);
p.addTerm(5, 0);
```

- 연산자 구현시 가정
  - 모든 항은 지수에 따라 내림차순으로 정렬
  - 모든 항의 지수는 다름
  - 계수가 0인 항은 포함하지 않음

두 다항식의 덧셈 연산

```
polyAdd(p1, p2)
// p3 = p1 + p2 : 다항식 p1과 p2를 더한 결과 p3을 반환
  p3←zeroP()
  while (not isPzero(p1) and not isPzero(p2)) do {
    case {
      maxExp(p1) < maxExp(p2) :
                p3 ← addTerm(p3, coef(p2, maxExp(p2)), maxExp(p2));
                p2 ← delTerm(p2, maxExp(p2));
      maxExp(p1) = maxExp(p2) :
                sum ← coef(p1, maxExp(p1)) + coef(p2, maxExp(p2));
                if (sum≠0) then p3 ← addTerm(p3, sum, maxExp(p1));
          p1 ← delTerm(p1, maxExp(p1));
          p2 ← delTerm(p2, maxExp(p2));
      maxExp(p2) > maxExp(p2) :
                p3 ← addTerm(p3, coef(p1, maxExp(p1)), maxExp(p1));
                p1 ← delTerm(p1, maxExp(p1));
    }
  }
  if (not isPzero(p1)) then p1의 나머지 항들을 p3에 복사
  else if (not isPzero(p2)) then p2의 나머지 항들을 p3에 복사;
  return p3;
End polyAdd
```

다항식 표현 방법

- **계수만 저장하는 방법**
  - 지수의 내림차순에 따라 다항식의 계수만 표현
  - 차수가 n인 항은 지수 n+1개에 해당하는 계수만을 순서리스트에 표현
  - 장점 : 다항식의 덧셈이나 곱셈 연산이 간단하고 지수를 별도로 저장할 필요가 없음
  - 단점 : 0인 항이 많은 희소 다항식인 경우 공간 낭비
    - 예) $x^{1000} + 1$: 원소 2개만 0이 아니고 나머지 999개가 모두 0
- **지수-계수 쌍 표현 방법**
  - 0이 아닌 항만 유지
  - 단점 : 다항식 연산의 알고리즘이 복잡 (지수 검사 필요)
  - 장점 : 효율적으로 저장 공간 활용
  - 보통 이 방법을 더 선호

- **예**
  - $2x^4 + 10x^3 + x^2 + 6$
  - 계수 표현 : (2, 10, 1, 0, 6)
  - 지수-계수 표현 : (4, 2, 3, 10, 2, 1, 0, 6)또는 (4, 2), (3, 10), (2, 1), (0, 6)
    – 배열로 표현

희소 행렬 (sparse matrix)

- **행렬 (matrix)**
  - m X n 행렬 : m개의 행과 n개의 열로 구성
  - m과 n이 같은 행렬은 정방 행렬(square matrix)
  - 2차원 배열로 표현
  - 행렬을 a[m, n]으로, 각 원소는 a[i, j] (i는 행, j는 열)로 표현

- **희소 행렬 (sparse matrix)**
  - 행렬의 원소가 대부분 0인 값
  - 일반적인 2차원 배열로 나타낼 때 공간의 낭비가 심함. 행렬의 원소가 0이 아닌 원소만 저장하는 방법이 필요

희소 행렬 추상 데이타 타입

ADT Sparse_Matrix

데이타 : 3원소쌍 <i, j, v>의 집합. i∈Row, j∈Column, v∈Value,
Row={0, 1, ..., m-1}, Column={0, 1, ..., n-1}

연산 : a, b ∈ Sparse_Matrix; c∈Matrix; u, v ∈ Value;
i∈Row; j∈Column;

spCreate(m, n) ::= return an empty sparse matrix with m×n;
spTranspose(a) ::= return c where c[j, i]=v when a[i, j]=v;
spAdd(a, b) ::= if (a.dimension = b.dimension) then return c where
c[i, j]=v+u when a[i, j]=v and b[i, j]=u
else return error
spMult(a, b) := if (a.no_of_cols = b.no_of_rows) then return c where
c[i, j] = Σ (a[i, k]·b[k, j]), p=a.no_of_cols
else return error;

End Sparse_Matrix

희소 행렬의 표현 방법

- **행렬은 <행, 열, 값> 3원소쌍으로 원소를 식별**
  - 0이 아닌 원소에 대해 열이 3인 2차원 배열로 표현
  - 효율적인 연산을 위해 행과 열을 오름차순으로 저장

$$b_{7X6}=\begin{bmatrix} 76 & 0 & 0 & 0 & 13 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 25 & 0 & 0 & 0 & 0 \\ -19 & 0 & 0 & 56 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 13 \\ 0 & 0 & 3 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{c[9,3]}$$

	0	1	2
0	7	6	8
1	0	0	76
2	0	4	13
3	2	5	3
4	3	1	25
5	4	0	-19
6	4	3	56
7	5	5	13
8	6	2	3

- 첫번째 행은 희소 행렬의 정보를 저장하는 데 사용
  c[0, 0], c[0, 1]은 각각 희소 행렬 b의 행수와 열수, c[0, 2]는 0이 아닌 원소수를 나타냄

희소 행렬의 전치

- **전치 행렬 (transposed matrix)**
  - 원소의 행과 열을 교환시킨 행렬
  - 원소 <i, j, v> → <j, i, v>

- **간단한 전치 행렬 알고리즘 (행 우선 표현)**

```
for (j ←0; j  <= n-1 ; j ←j+1 ) do
    for (i ←0; i <=m-1 ; i ←i+1 ) do
        b[j, i] ←  a[i, j];
```

- **희소 행렬의 전치**
  - 행 우선, 행 내에선 열 오름차순으로 저장된 경우, 각 열별로 차례로 모든 원소를 찾아 행의 오름차순으로 저장하는 작업을 반복

희소 행렬 전치 알고리즘

```
transposeS(a[])
  // 배열로 표현한 희소 행렬 a[t+1, 3]을 전치 희소 행렬 b[t+1, 3]으로 변환
  m ← a[0, 0];   n ← a[0, 1];   // m : a의 행 수,  n :  a[]의 열수
  t ← a[0, 2];        // a[]의 0이 아닌 원소 수
  b[0, 0] ← n;        // b[]의 행 수 ← a[]의 열 수
  b[0, 1] ← m;        // b[]의 열 수 ← a[]의 행 수
  b[0, 2] ← t;        // b[]의 0이 아닌 원소 수
  if (t > 0) then {
    k ← 1;          // b[]에 대해 현재 행의 위치를 지시
    for (p ← 0; p < n; p ← p+1) do { // a[]의 열 별로 처리
      for (i ← 1; i <= t; i ← i+1) do { // 0이 아닌 원소 수에 대해
        if a[i, 1] = p then {  // 열 p의 원소 발견
          b[k, 0] ← a[i, 1];  b[k, 1] ← a[i, 0]; b[k, 2] ← a[i, 2];
          k ← k+1;
        }
      }
    }
  }
  return b[];
End transposeS()
```

행렬 전치 알고리즘의 시간 복잡도

- **일반 m X n행렬에 대한 전치 행렬 알고리즘**
  - 중첩된 for 루프의 영향으로 O(m・n)
- **희소 행렬의 전치 알고리즘**
  - 희소 행렬 전치 알고리즘 Transpose
    - O(n・t) (t는 0이 아닌 원소수)
  - 최악의 경우
    - t = m・n이 될 수 있으므로 O(n・t) = O(m・n^2)이 되어 O(m・n)보다 커질 수 있음
  - 개선된 방법
    - 각 열에 대해 0이 아닌 원소수를 먼저 계산하면 이것이 곧 전치 행렬 B에서 각 행의 원소수가 됨.
    - 이 정보를 통해 배열 b[]에 저장시킬 각 행의 시작점(인덱스)을 결정할 수 있음.
    - 그런 다음 배열 a[]의 원소를 하나씩 차례로 배열 b[]로 변환
    - 시간 복잡도는 O(n + t)

희소 행렬 연산의 Java 구현

- **3원소쌍의 표현**
  - 3원소쌍을 2차원 배열로 표현 가능
  - 객체지향 프로그래밍 언어 환경에서는 클래스로 표현하는 것이 바람직함
- **Triple 클래스**
  - <행, 열, 값> 을 가지는 자료 구조와 생성자를 포함한 클래스

```
public class Triple {
    int row, col, value;
            public Triple() {
                row = col =  value= 0;
            }
            public Triple(int r, int c, int v) {
                row = r;
                col = c;
                value = v;
            }
        }
```

희소 행렬 전치 프로그램 (1)

```
public class SparseMatrix {
    int Nrows, Ncols, Nterms, idx;
    Triple[] a;
    public SparseMatrix( int rows, int cols, int terms ) {
        // 생성자. 각 변수 초기화
        Nrows = rows; Ncols=cols; Nterms=terms;
        idx = 0;
        a = new Triple[Nterms];
    }

    public void displayMatrix() {
        // 행의 수, 열의 수, 0이 아닌 원소수를 출력
        System.out.println("Number of rows : "+Nrows);
        System.out.println("Number of columns : "+Ncols);
        System.out.println("Number of non-zero terms : "+Nterms);
            // 행렬의 내용을 "[인덱스] 행 열 값"으로 출력
        for (int i = 0; i < Nterms; i++)
            System.out.println("["+i+"]"+"  "+a[i].row
            +"  "+a[i].col+"  "+a[i].value);
    }
```

희소 행렬 전치 프로그램 (2)

```
public void storeTriple( int r, int c, int v ) {
        if (idx >= Nterms) {   // 에러 출력 후 프로그램 종료
           System.out.println("Error : too many terms..");
           System.exit(-1);
        }
        a[idx++] = new Triple( r, c, v );
    }

    public SparseMatrix transpose() {
        int i, j;
        int [] RowTerms = new int[Ncols];
        int [] RowBegins = new int[Ncols];
        SparseMatrix b = new SparseMatrix( Ncols, Nrows, Nterms );
        if (Nterms > 0) {
           for (i = 0; i < Ncols; i++)  RowTerms[i] = 0;
           for (i = 0; i < Nterms; i++) RowTerms[a[i].col]++;
           RowBegins[0] = 0;
```

희소 행렬 전치 프로그램 (3)

```
        for (i = 1; i < Ncols; i++)
            RowBegins[i] = RowBegins[i-1]+RowTerms[i-1];
        for (i = 0; i < Nterms; i++) {
             j = RowBegins[a[i].col]++;
            b.a[j] =
                new Triple(a[i].col,
                     a[i].row, a[i].value);
        }
      }
      return b;
   }
} // end SparseMatrix class
```

희소 행렬 전치 프로그램 (4)

- **SparseMatrix 클래스**
  - SparseMatrix(**int** rows, **int** cols, **int** terms);
    - Nrows, Ncols, idx를 초기화
    - Nterms크기의 Triple타입의 배열 생성
  - void displayMatrix();
    - 행렬의 내용을 화면에 표시
  - void storeTriple(**int** r, **int** c, **int** v);
    - r, c, val 값으로 Triple의 생성자를 통해 객체를 생성
    - 배열 a[]에 실제 객체의 주소 저장
  - SparseMatrix spTranspose();
    - 주어진 희소 행렬을 전치 시켜 희소 행렬 b로 반환
    - 전치 행렬 b에서의 행의 크기(RowTerms), 행의 출발점(RowBegin) 계산
    - 실제로 주어진 희소 행렬에서부터 b로 3원소쌍을 이동

```
b.a[j] = new Triple();
b.a[j].row = a[i].col;
b.a[j].col = a[i].row;
b.a[j].value = a[i].value;
```

희소 행렬 전치 프로그램 (5)

```
public class SparseTrans {

  public static void main(String args[]) {
    SparseMatrix b;
    SparseMatrix a = new SparseMatrix(7, 6, 8);

    a.storeTriple (0, 0, 76 );
    a.storeTriple (0, 4, 13 );
    a.storeTriple (2, 5, 3  );
    a.storeTriple (3, 1, 25 );
    a.storeTriple (4, 0, -19 );
    a.storeTriple (4, 3, 56 );
    a.storeTriple (5, 5, 13 );
    a.storeTriple (6, 2, 3 );

    a.displayMatrix();
    b = a.transpose();
    b.displayMatrix();
  } // end main()
} // end SparseTrans class
```

희소 행렬 전치 프로그램 (6)

- **Sparse 클래스**
  - SparseMatrix 클래스를 이용하여 7X6 행렬a를 전치시키는 클래스
  - 5행 : 행수가 7, 열수가 6, 0이 아닌 원소수가 8인 SparseMatrix 객체를 생성
  - 7-14행 : 행렬 a를 입력
  - 16행 : 전치시킬 희소 행렬 a를 출력
  - 17행 : 행렬 a에 대해 spTranpose() 메소드를 기동시켜 그 결과를 희소 행렬 b에 지정
  - 18행 : 전치된 희소 행렬 b를 출력

희소 행렬 전치 프로그램 (7)

◆ 실행 결과

- 원래의 행렬

Number of rows : 7
Number of colums : 6
Number of non-zero terms : 8
[0] 0 0 76
[1] 0 4 13
[2] 2 5 3
[3] 3 1 25
[4] 4 0 -19
[5] 4 3 56
[6] 5 5 13
[7] 6 2 3

- 전치된 행렬

Number of rows : 6
Number of colums : 7
Number of non-zero terms : 8
[0] 0 0 76
[1] 0 4 -19
[2] 1 3 25
[3] 2 6 3
[4] 3 4 56
[5] 4 0 13
[6] 5 2 3
[7] 5 5 13

희소 행렬 전치 프로그램 (8)

- **transpose() 메소드**
  - RowBegin[i] = RowBegin[i-1] + RowTerms[i-1]
    - RowBegin[i] : 행렬 b의 i행의 시작 위치
    - RowTerms[i-1] : b의 행 i-1에 속하게 될 원소수

row :	[0]	[1]	[2]	[3]	[4]	[5]
RowTerms =	2	1	1	1	1	2
RowBegins =	0	2	3	4	5	6

  - 시간 복잡도 : O(Ncols + Nterms)
    - 각각 Ncols, Nterms, Ncols-1, Nterms번 실행되는 4개의 루프가 있으므로

MatrixTrio.java SparseMatrix.java SparseMatrixTest.java