자료구조 Data Structure | 조행래

배열과 구조체

배열을 이용한 희소 행렬의 표현

학습 목표

- 희소 행렬의 정의를 이해한다.
- 배열을 이용하여 희소 행렬을 표현할 수 있다.

1. 희소 행렬이란?

- **행렬의 표현**
  - 2차원 배열: A[MaxRows][MaxCols]

	col0	col1	col2
row0	-27	3	4
row1	6	82	-2
row2	109	-64	11
row3	12	8	9
row4	48	27	47

(a) A[5][3]

	col0	col1	col2	col3	col4	col5
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

(b) B[6][6]

  - 0이 많이 포함될 경우 ⇒ 희소 행렬
    - 예: 1000 * 1000 행렬에서 0이 아닌 원소가 10개?

2. 희소 행렬의 표현

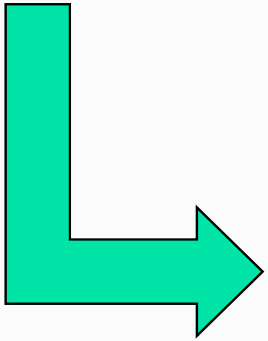
- **동기: 공간 낭비를 줄이자**
  - <row, column, value>의 쌍을 저장
  - 빠른 전치(transpose)를 위하여 row의 오름차순으로 저장

- **희소 행렬 ADT**

```
Sparse_Matrix Create(rows, cols) ::=

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

희소 행렬의 예

	col0	col1	col2	col3	col4	col5
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



	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

3. 행렬의 전치

- **전치 연산(Transposing a Matrix)**
  - 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!

전치 행렬의 예

	col0	col1	col2	col3	col4	col5
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

	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)

4. 전치 연산의 구현: Transpose

- **기본 개념**

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

	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)

Program: Transpose

```
void transpose( term a[ ], term b[ ] )// b = a^T
{
    int i, j, currentb;
    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 = a[0].value;          // 0이 아닌 원소 수는 a와 동일
    if (a[0].value > 0)    {
        currentb = 1;                 // 새로운 원소가 저장될 b의 위치.
        for ( i = 0;  i < a[0].col;  i++ )
            for (j = 1; j <= a[0].value; j++)
                if ( a[j].col == i ) {    //  현재 열의 원소 발견. b에 추가
                    b[currentb].row = a[j].col;
                    b[currentb].col = a[j].row;
                    b[currentb].value = a[j].value;
                    currentb++;     // b의 저장될 위치를 1 증가
                }
}   }
```

시간 복잡도 = O(columns * elements)

성능 분석

- **Transpose 알고리즘의 시간 복잡도**
  - O(columns * elements) ≅ O(columns^2 * rows)

- **2차원 배열에서 transpose의 구현**

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

복잡도 = O(rows*columns)

5. 전치 연산의 구현: Fast Transpose

- **기본 개념**
  - 각 column이 저장될 곳을 미리 파악 → Column Index
  - 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

	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)

Program: Fast Transpose

```
#define MAX_COL   50                    // 최대 열의 수 + 1
void  fast_transpose(term a[ ],  term b[ ])    //   b = a^T
{
    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
```

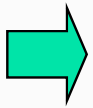
Program: Fast Transpose (계속)

```
    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;
    }
}   }
```

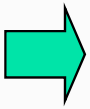
시간 복잡도 = O(columns + elements)

Fast Transpose의 동작 과정

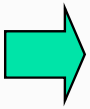
	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28



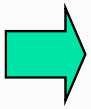
rowterms[0] = 2
rowterms[1] = 1
rowterms[2] = 2
rowterms[3] = 2
rowterms[4] = 0
rowterms[5] = 1



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

요약 정리

- 희소 행렬의 정의를 이해
- 구조체 배열로 희소 행렬을 표현하는 방법을 이해
- C-언어에서 희소 행렬의 전치 행렬을 구현하는 알고리즘 이해