TECHNICAL REPORT

HOMEWORK 2

2071008 SEOYEONKIM

Homework 2_1 Transpose of SparseMatrix

[Setting Initial value]

main()함수에서 먼저 SparseMatrix 타입의 구조체 변수 B의 값을 초기화해주어야한다.

Sparse Matrix 타입안에 있는 변수는 총 4가지로 아래와같다.

SparseMatirx	
type	**name**
element[Max term]	data
int	rows
int	cols
int	terms

: rows cols terms는 각각 matrix하나의 row수, column의 수를 저장하고, terms 변수에는 matrix에서 0이 아닌 변수들의 수를 저장한다.

element 타입의 구조체배열에는 matrix에 들어가는 수들의 정보를 담으며, element 타입의 구조체 안에 있는 변수는 총 3가지로 아래와 같다.

element	
type	**name**
int	row
int	col
int	value

: matrix에 들어가는 수에 대한 정보로 해당하는 element의 위치를 row와 col에 저장하고 그 값을 value에 저장한다. 예를 들어, row의 index1에 위치하고, column의 index2에 위치하는 수 21을 저장하고 싶다면 element타입안에 row = 1; col =2; value =21; 으로 저장하면 된다. 즉, element 배열의 크기는 행렬에 들어갈 수 있는 최대 term의 개수가 되고, MAX_TERMS의 크기만큼 Matrix안에 들어가는 숫자를 저장할 수 있다.

정리하자면, SparseMatrix 타입의 변수를 선언할때에는 차례대로 {data, row-size, col-size, the number of terms} 를 초기화해주는 것이 필요하다. 그리고 Matrix에 들어가는 수에 대한 정보 data는 element array 형식으로 들어가는데, 해당 배열의 각 요소에는 Matrix에 들어갈 수에 대한 정보를 {row, col, value} 꼴로 저장해주면 된다. 예를들어 다음과 같은 선언

```
SparseMatirx B = { {{0,3,7},{1,0,9},{1,5,8},{3,0,6},{3,1,5},{4,5,1},{5,3,2}}, 6, 6, 7 };
```

은 아래와같이 7개의 term을 가지는 6x6 크기의 Sparse matrix를 만들겠다는 것을 의미한다.

```
[input]
0 0 0 7 0 0
9 0 0 0 0 8
0 0 0 0 0 0
6 5 0 0 0 0
0 0 0 0 0 1
0 0 2 0 0 0
```

[explanation of the codes and analysis]

: theoretical explanation and performance analysis , the reason for setting parameters, the result and performance analysis

1.SparseMatirx transpose(SparseMatirx B)

: 행렬의 transpose연산을 수행해주는 함수

```
SparseMatirx transpose(SparseMatirx m) {
	SparseMatirx transposed;
	// swithching row size and col size
	transposed.rows = m.cols;
	transposed.cols = m.rows;
	transposed.terms = m.terms;


	for (int i = 0; i < m.terms; i++) {
		transposed.data[i].value = m.data[i].value;
		transposed.data[i].row = m.data[i].col;
		transposed.data[i].col = m.data[i].row;
	}

	sort(transposed.data, transposed.data + transposed.terms,cmp); //in row-wize manner
	return transposed;

}
```

: parameter인 SparseMatrix타입의 변수 m에 transpose operation을 적용한 결과를 저장할 구조체변수 transposed를 선언한다. transpose Matrix는 해당 Matrix의 주 대각선을 대칭으로 뒤집어서 얻을 수 있다. 즉, row와 column을 교환하여 얻는 matrix이다. 따라서 M*N matrix가 N*M

형태가 되고, 한 element의 row와 col의 값은 뒤바뀐다. 따라서 transposed의 row는 m의 col로, transposed col은 m의 row로 설정한다**. (M*N -> N*M)** 그리고 m에 저장된 모든 element에 접근하여, element의 위치값도 col과 row를 바꾸어서 transposed에 저장한다 **(B[i][j] -> B[j][i])** 이때 행렬에 0이아닌 숫자만 바꿔주기 위해 m의 terms의 값만큼 반복한다. 그러나 위와 같은 과정을 거치면, 본래 m의 data에는 값들이 in row-wise manner였지만 transposed의 data에서는 in row-wise manner로 저장되지 않게 된다. 단순히 m이 갖는 value의 row와 col값을 바꾼 data를 transposed에 저장했을뿐이기 때문이다. 따라서 data를 in row-wise manner로 정렬해주기 위해서 cmp함수를 정의하고 sort함수를 호출한다. cmp함수는 element배열이 행우선으로 정렬되도록 정의되어있으며, 내용은 아래와 같다.

```
bool cmp(element a, element b) {
	if (a.row != b.row)
		return a.row < b.row;
	return a.col < b.col;
}
```

: cmp함수는 다음과 같은 규칙으로 정렬을 수행한다.

1. 행 값이 다르다면, 더 작은 행값을 가진 elment가 더 앞에 온다.

2. 행 값이 같다면, 더 작은 열값을 가진 element가 더 앞에 온다.

결국 이 모든 과정을 거쳐 transposed변수는 변수m의 row와 col이 뒤집힌 결과인 transpose operation을 한 결과를 행 우선방법으로 담은 matrix 가되고, 그 구조체 transposed를 return해서, transpose연산이 완료된 SparseMatrix타입의 변수를 main함수의 변수에 저장할 수 있다.

2.void printSparseMatirx(SparseMatirx m)

: SparseMatirx 를 denseMatrix 형태로 출력해주는 함수.

```
void printSparseMatirx(SparseMatirx m) {
	cout << "========================="<<'\n';
	int pos=0;
	for (int i = 0; i < m.rows; i++) {
		for (int j = 0; j < m.cols; j++) {
			if (i == m.data[pos].row && j == m.data[pos].col)
				cout << m.data[pos++].value<<' ';
			else {
				cout << "0" <<' ';

			}

		}
```

```
                cout << '\n';
        }
        cout << "========================"<<'\n';
}
```

: 우선 주어진 matrix(m)의 data를 순차적으로 접근하기 위해 인덱스값을 담을 pos변수를 선언한다. 그리고 반복문은 총 row * col 번 수행되며, 각 반복문에서는 i행 j번째 값이 data에 존재하는지 확인하는 과정을 거친다. 이때, SparseMatirx의 element가 row-wize-manner로 정렬되어 있기 때문에 이번에 출력할 행,열 값 (i,j) 이 data[pos]와 일치하는지 단 한번의 비교를 통해 m.data에 i행 j번째 원소가 존재하는지를 알 수 있다. 만일 data[pos] 가 이번에 출력할 행,열의 값과 일치한다면, value값을 출력하고 pos값을 증가시킨다. 증가된 pos값을 통해 다음번 반복때 data의 다음원소에 접근할 수 있도록 한다. 만일 일치하지 않다면, i행 j번째 원소는 주어진 matirx에 존재하지 않다는 뜻이 되므로 0을 출력한다.

Homework 2_2 Dynamic allocation of 3D array

[Setting Initial value]

```
//you can define the size of matrix by changing below value
#define ROW 3
#define COL 3
#define HEIGHT 3
```

: these affects overall function

For example , if you want to make array of size 4x5x6, change value as below

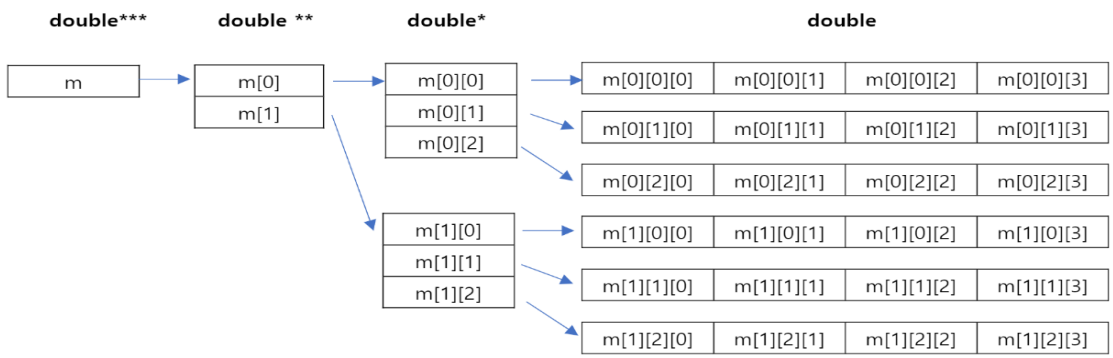
```
#define ROW 4
#define COL 5
#define HEIGHT 6
```

: Then, the all functions work based on new ROW,COL,HEIGHT

[explanation of the codes and analysis]

1. mem_alloc_3D_double

다음은 2X3X4 크기의 3차원 배열을 만들기 위한 함수 mem_alloc_3D_double()에서 나타나는 메모리 할당을 나타낸 그림이다.



m은 3중 포인터로 address of address of address를 의미하고,

m[]은 2중 포인터로 address of address를 의미하며,

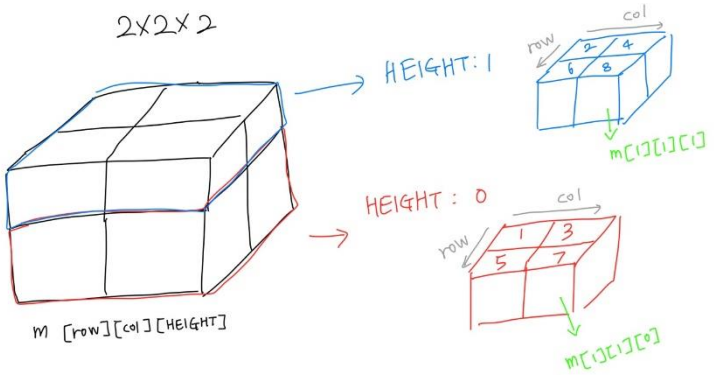
m[][]은 포인터로 address를 의미한다.

즉, 배열의 m[][][]꼴의 값에 접근하기 위한 address는 m[][]이고, m[][]의 address에 접근하기위한 address는 m[]이고 (address of address) m[]의 address에 접근하기위한 address는 m이다. (address of address of address) 따라서, malloc을 이용해서 주소를 할당해줄때, double*** 타입인 m은 row-size만큼의 double **을 필요로하므로, `double*** m = (double***)malloc(sizeof(double**) * ROW);` 로 메모리 할당을 해준다. 이때, (double***)을 붙이는 이유는, malloc은 void값의 형태로 return되어 데이터형을 맞춰주기 위해서이다. 마찬가지로 m[i]는 double **타입으로, col-size만큼의 double *을 필요로하므로 , `m[i] = (double**)malloc(sizeof(double*) * COL);` 로 메모리 할당을 해준다. m[i][j]는 double* 타입으로, height-size만큼의 double 공간을 필요로 하므로, `m[i][j] = (double*)malloc(sizeof(double) * HEIGHT);` 로 메모리 할당을 해준다. 마지막으로 이 함수의 리턴값은 double*** 타입인데 , address of address of address 를 의미하는 m을 return하여, main함수의 double***타입의 변수 A, B, C에서 할당된 3D 배열에 접근할 수 있도록 한다.

2. void setValue3D(double* m)**

for문을 이용하여, 3차원 배열 m의 모든 요소에 사용자의 입력값을 할당한다. 입력의 형식은 matrix[i][j][k] = input 이다. 다음은 ROW,COL,HEIGHT을 각각 2로 설정했을 때 나타나는 출력이다.

```
Matrix A
set own your value
m[0][0][0] :1
m[0][0][1] :2
m[0][1][0] :3
m[0][1][1] :4
m[1][0][0] :5
m[1][0][1] :6
m[1][1][0] :7
m[1][1][1] :8
```



:오른쪽 그림은 왼쪽 INPUT을 가지는 3차원 배열을 그림으로 나타낸것이다.

3. void print_3D_matrix(double* m)**

: 주어진 매트릭스에 저장된 값을 층우선방식으로 출력해주는 함수이다.

[예시]

4. double* addition_3D(double*** a, double*** b)**

할당이 완료된 double***타입의 두개의 변수를 addition_3D함수의 parameter로 사용한다. 그리고 `double*** sum = mem_alloc_3D_double();` 로 매트릭스의 덧셈연산값을 저장할 새로운 3D배열을 만든다. matrix의 덧셈은 같은 자리에 있는 element들끼리의 합이기 때문에, `sum[i][j][k] = a[i][j][k] + b[i][j][k];` 로 수행되어질 수 있다. 그리고 `return sum;` 을 통하여 덧셈연산을 수행한 matrix의 address of address of address를 리턴한다. main함수에서 `double*** SUM = addition_3D(A, B);` 같은 방식으로 쓰이는데, addition_3D 함수를 통해 A matrix와 B matrix를 더하는 연산을 한 새로운 matrix 를 가리키는 주소값을 SUM 에 저장하겠다는 의미이다.

5. void deallocate_3D_double(double* m)**

할당된 메모리를 해제할 때에는, 할당과 반대의 순서로 메모리를 해제해야한다. 메모리 할당의 과정이 m -> m[i] -> m[i][j]였다면 메모리 해제의 과정은 m[i][j] -> m[i] -> m이다. 따라서 중첩 반복문을 이용하여 먼저 free(m[i][j])를 해주고, 그 밖에 반복문에서 free(m[i]) 그리고 반복문의 밖에서 free(m)으로 3차원 배열의 메모리 공간을 모두 해제한다. 결국 변수 m은 메모리가 할당되지 않은 상태, 즉 아 무것도 가리키지 않는 NULL포인터로 만들어야 하므로 `m = NULL;` 을 마지막줄에 넣어준다