CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 4

In this assignment you will create a calculator for performing matrix operations that exploits the (expected) sparseness of its matrix operands. An $n \times n$ square matrix is said to be *sparse* if its number of non-zero entries is small compared to the total number of entries n^2. The result will be a C program capable of performing fast matrix operations, even on very large matrices, provided that they are sparse.

Given $n \times n$ matrices *A* and *B*, their product $C = A \cdot B$ is the $n \times n$ matrix whose ij^{th} entry is given by

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Thus the element in the i^{th} row and j^{th} column of *C* is the vector dot product of the i^{th} row of *A* with the j^{th} column of *B*. If we consider addition and multiplication of real numbers to be our basic operations, then the above formula can be computed in time $\Theta(n^3)$, which is impractical for matrix sizes *n* of more than a few thousand. If it so happens that *A* and *B* are sparse, then a great many of these arithmetic operations involve adding or multiplying by zero, hence are unnecessary.

The sum *S*, and difference *D*, of *A* and *B* are the $n \times n$ matrices having ij^{th} entries:

$$S_{ij} = A_{ij} + B_{ij} \qquad \text{and} \qquad D_{ij} = A_{ij} - B_{ij}$$

The scalar product of a real number *x* with *A* is denoted xA, and has ij^{th} entry $(xA)_{ij} = x \cdot A_{ij}$. The transpose of *A*, denoted A^T, is the matrix whose ij^{th} entry is the ji^{th} entry of *A*: $(A^T)_{ij} = A_{ji}$. Thus, the rows of *A* are the columns of A^T, and the columns of *A* are the rows of A^T. Each of these operations can be computed in time $\Theta(n^2)$, and just as for multiplication, their cost can be improved upon significantly when *A* and *B* are sparse.

As one would expect, the cost of a matrix operation depends heavily on the choice of data structure used to represent its matrix operands. There are several ways to represent a square matrix with real entries. The standard approach is to use a 2-dimensional $n \times n$ array of doubles. The advantage of this representation is that all of the above matrix operations have a straight-forward implementation using nested loops.

In this project however, will use a very different representation. Instead we will represent a matrix as an array of Lists. Each List represents one row of the Matrix, but only the non-zero entries will be stored in that List. Thus a List element must store, along with a matrix value, the index of the column index at which that value resides. For example, the matrix below is represented by the following array of Lists.

$$M = \begin{bmatrix} 1.0 & 0.0 & 2.0 \\ 3.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 5.0 \end{bmatrix} \qquad \text{Array of Lists: } \begin{bmatrix} 1: & (1, 1.0) & (3, 2.0) \\ 2: & (1, 3.0) & \\ 3: & (2, 4.0) & (3, 5.0) \end{bmatrix}$$

Each List element is an ordered pair of the form (column index, value). This method results in a substantial space savings when the Matrix is sparse. In addition, the standard matrix operations defined above can be performed more efficiently on sparse matrices. However, as we will see, the matrix operations are significantly more complex when using this representation. The trade-off then, is a gain in space and time efficiency for sparse matrices, at the expense of more complicated algorithms for performing standard

matrix operations. Designing these algorithms in terms of our List ADT operations will constitute the bulk of your work on this assignment.

List ADT changes

It will be necessary to first make some important alterations to your List ADT from pa1. First you will convert it from a List of ints to a List of generic pointers. This entails changing certain field types, declaration statements, method parameters, and return types from `int` to `void*`. The objects pointed to by these List elements will be defined in the Matrix ADT specified below. Second, it will be necessary to eliminate the List operations equals() and copyList(). The problem with these functions is that the List no longer knows what it is a list of, and therefore can only perform "shallow" versions of these operations, i.e. compare or copy pointers, not what they point to. For the same reason, function printList() is no longer required. You may choose to include it though for diagnostic purposes. All other List operations from pa1 will be retained. A typical loop a client of the new List ADT might now appear as:

```
void* p;
moveFront(L);
while( position(L)>=0 ){
   p = get(L);
   // dereference p and do something with the object it points to
   moveNext(L);
}
```

Be sure to test your modified List ADT before you use it in the Matrix ADT described below.

File Formats

The top level client module for this project will be called Sparse.c. It will take two command line arguments giving the names of the input and output files, respectively. The input file will begin with a single line containing three integers *n*, *a* and *b*, separated by spaces. The second line will be blank, and the following *a* lines will specify the non-zero entries of an $n \times n$ matrix A. Each of these lines will contain a space separated list of three numbers: two integers and a double, giving the row, column, and value of the corresponding matrix entry. After another blank line, there will follow *b* lines specifying the non-zero entries of an $n \times n$ matrix B. For example, the two matrices

$$A = \begin{bmatrix} 0.0 & 0.0 & 4.9 \\ -9.3 & 0.0 & 0.0 \\ 0.0 & -2.4 & -5.2 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 1.3 & 0.0 & -8.6 \\ 6.2 & 8.0 & 0.0 \end{bmatrix}$$

are encoded by the following input file:

```
3 4 4

3 3 -5.2
3 2 -2.4
2 1 -9.3
1 3 4.9

2 3 -8.6
3 2 8.0
2 1 1.3
3 1 6.2
```

Your program Sparse.c will read an input file as above, initialize and build the Array-of-Lists representation of the matrices A and B, then calculate and print the following matrices to the output file: A, B, $(1.5)A$, $A + B$, $A + A$, $B - A$, $A - A$, A^T, AB and B^2. The output file corresponding to the above input file is illustrated below.

```
A has 4 non-zero entries:
1: (3, 4.9)
2: (1, -9.3)
3: (2, -2.4) (3, -5.2)

B has 4 non-zero entries:
2: (1, 1.3) (3, -8.6)
3: (1, 6.2) (2, 8.0)

(1.5)*A =
1: (3, 7.4)
2: (1, -14.0)
3: (2, -3.6) (3, -7.8)

A+B =
1: (3, 4.9)
2: (1, -8.0) (3, -8.6)
3: (1, 6.2) (2, 5.6) (3, -5.2)

A+A =
1: (3, 9.8)
2: (1, -18.6)
3: (2, -4.8) (3, -10.4)

B-A =
1: (3, -4.9)
2: (1, 10.6) (3, -8.6)
3: (1, 6.2) (2, 10.4) (3, 5.2)

A-A =

Transpose(A) =
1: (2, -9.3)
2: (3, -2.4)
3: (1, 4.9) (3, -5.2)

A*B =
1: (1, 30.4) (2, 39.2)
3: (1, -35.4) (2, -41.6) (3, 20.6)

B*B =
2: (1, -53.3) (2, -68.8)
3: (1, 10.4) (3, -68.8)
```

Notice that each row is to be printed in column sorted order, and zero rows are skipped altogether. A zero matrix will cause no output to be printed, as seen by the matrix $A - A$ above. Note that, unlike the output file, the input file may give matrix entries in any order.

Matrix ADT Specifications

In addition to the main program Sparse.c and the altered List.c from pa1, you will implement a Matrix ADT in a file called Matrix.c. This ADT will contain a private inner struct (similar to Node in your List ADT) encapsulating the column and value information corresponding to a matrix entry. You can give this inner struct any name you wish, but I will refer to it here as Entry. Thus Entry will have two fields of types int and double respectively. Your Matrix ADT will represent a matrix as an array of Lists of pointers to Entry objects. It is required that each List be maintained in column sorted order. Your Matrix ADT will export the following operations.

```
// Constructors-Destructors ---------------------------------------------------

// newMatrix()
// Returns a reference to a new n by n Matrix in the zero state.
Matrix newMatrix(int n);

// freeMatrix()
// Frees all heap memory associated with *pM, sets *pM to NULL.
void freeMatrix(Matrix* pM);


// Access functions -----------------------------------------------------------

// dimension()
// Returns the number of rows and columns of square Matrix M.
int dimension(Matrix M);

// numNonZero()
// Returns the number of non-zero elements in M.
int numNonZero(Matrix M);

// equals()
// Returns true if matrices A and B are equal, false otherwise.
bool equals(Matrix A, Matrix B);


// Manipulation procedures ----------------------------------------------------

// makeZero()
// Resets M to the zero Matrix.
void makeZero(Matrix M);

// changeEntry()
// Changes the ith row, jth column of M to be the value x.
// Pre: 1<=i<=dimension(M), 1<=j<=dimension(M)
void changeEntry(Matrix M, int i, int j, double x);


// Matrix Arithmetic operations -----------------------------------------------

// copy()
// Returns a reference to a new Matrix having the same entries as A.
Matrix copy(Matrix A);

// transpose()
// Returns a reference to a new Matrix representing the transpose
// of A.
Matrix transpose(Matrix A);
```

```
// scalarMult()
// Returns a reference to a new Matrix representing xA.
Matrix scalarMult(double x, Matrix A);

// sum()
// Returns a reference to a new Matrix representing A+B.
// pre: dimension(A)==dimension(B)
Matrix sum(Matrix A, Matrix B);

// diff()
// Returns a reference to a new Matrix representing A-B.
// pre: dimension(A)==dimension(B)
Matrix diff(Matrix A, Matrix B);

// product()
// Returns a reference to a new Matrix representing AB
// pre: dimension(A)==dimension(B)
Matrix product(Matrix A, Matrix B);


// Other operations -----------------------------------------------------------

// printMatrix()
// Prints a string representation of Matrix M to filestream out. Zero rows are
// not printed.  Each non-zero row is represented as a line consisting of the
// row number, followed by a colon, a space, then a space separated list of pairs
// "(col, val)" giving the column numbers and non-zero values in that row. The
// double val will be rounded to 1 decimal point.
void printMatrix(FILE* out, Matrix M);
```

It is required that your program perform these operations efficiently. Let n be the number of rows in A, and let a and b denote the number of non-zero entries in A and B respectively. Then the worst case run times of the above functions should have the following asymptotic growth rates.

`changeEntry(A, i, j, x):`	$\Theta(a)$
`copy(A):`	$\Theta(n + a)$
`transpose(A):`	$\Theta(n + a)$
`scalarMult(x, A):`	$\Theta(n + a)$
`sum(A, B):`	$\Theta(n + a + b)$
`diff(A, B):`	$\Theta(n + a + b)$
`product(A, B):`	$\Theta(n^2 + a \cdot b)$

See the handout entitled *Algorithm Runtime and Efficiency* posted on the class webpage for an explanation of the Θ notation above. It will be helpful to include a private function in Matrix.c with signature

```
double vectorDot(List P, List Q)
```

that computes the vector dot product of two matrix rows represented by Lists P and Q. Use this function together with function `transpose()` to help implement `product()`. Similar helper functions for the operations `sum()` and `diff()` will also be useful, and will be discussed in class.

What to Turn In

Your project will be structured in three files: Sparse.c, Matrix.c, and List.c (together with header files Matrix.h and List.h). The main program Sparse.c, will handle the input and output files and is the client of the Matrix ADT, which is itself a client of the modified List ADT. Note that Sparse is not itself a direct client of List, since it need not call any List operations. You will also write a separate client module called MatrixTest.c your Matrix ADT in isolation. Students often ask what should be the contents of such a test client. You should include enough calls to ADT operations to convince the grader that you did in fact test your Matrix ADT module in isolation before using it in the larger project. *The best way to do this is to actually use the test client for this purpose*. At minimum it should call every public function at least once. The webpage link Examples/pa4 contains the file MatrixClient.c, which should be considered to be weak test of your Matrix ADT, and are not fully adquate for your testing purposes. A number of matched pairs of input/output files are also included, along with a Python script for creating random input files.

As usual submit a README.md file and a Makefile that creates an executable called `Sparse`. A Makefile is also included in Examples/pa4 which you may alter as you see fit. Thus in all, eight files will be submitted.

`Sparse.c`	written by you
`Matrix.c`	" " "
`Matrix.h`	" " "
`MatrixTest.c`	" " "
`List.c`	" " "
`List.h`	" " "
`README.md`	" " "
`Makefile`	provided, change if you like

Push these files to the pa4 folder in your git repository on git.ucsc.edu by the due date. As always, start early and ask questions.