CMPS 101
Algorithms and Abstract Data Types
Programming Assignment 3

In this assignment you will create a calculator for performing matrix operations that exploits the (expected) sparseness of it's matrix operands. An $n \times n$ square matrix is said to be *sparse* if the number of non-zero entries (abbreviated NNZ) is small compared to the total number n^2 of entries. The result will be a Java program capable of performing fast matrix operations, even on very large matrices, provided 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 to, 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*: $(A^T)_{ij} = A_{ji}$. In other words, 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 the matrix operands. There are several ways to represent a 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. This project will use a very different representation however. Here you will represent a matrix as a 1-dimensional array of Lists. Each List will represent one row of the Matrix, but only the non-zero entries will be stored. Therefore List elements must store not just the matrix entries, but the columns in which those entries reside. For example, the matrix below would have the following representation as an array of Lists.

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

This method obviously 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. As you will see though, the matrix operations are much more difficult to implement 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 List operations will constitute the majority of the work you do on this assignment.

It will be necessary to make some minor changes to your List ADT from pa1. First you must convert your List ADT from a List of ints to a List of Objects. This entails changing certain field types, declaration statements, method parameters, and return types from int to Object. The Objects referred to by these List elements will be defined in the Matrix ADT specified below. Second, it will be necessary to eliminate the List operations copy() and cat() (which was optional anyway.) All other List operations from pa1 will be retained. The equals() operation however will be altered slightly so as to override, rather than overload Object's built in equals() method. This is done by changing it's signature from `boolean equals(List L)`, as in pa1 to `public boolean equals(Object x)`, which is it's signature in the superclass Object. Indeed, all equals() methods in this project should carry this same signature.

File Formats

The top level client module for this project will be called Sparse.java. 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, will follow *b* lines specifying the non-zero entries of an $n \times n$ matrix *B*. For example, the two matrices

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}$$

are encoded by the following input file:

```
3 9 5

1 1 1.0
1 2 2.0
1 3 3.0
2 1 4.0
2 2 5.0
2 3 6.0
3 1 7.0
3 2 8.0
3 3 9.0

1 1 1.0
1 3 1.0
3 1 1.0
3 2 1.0
3 3 1.0
```

Your program will read an input file such 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 format is illustrated by the following example, which corresponds to the above input file.

```
A has 9 non-zero entries:
1: (1, 1.0) (2, 2.0) (3, 3.0)
2: (1, 4.0) (2, 5.0) (3, 6.0)
3: (1, 7.0) (2, 8.0) (3, 9.0)

B has 5 non-zero entries:
1: (1, 1.0) (3, 1.0)
3: (1, 1.0) (2, 1.0) (3, 1.0)

(1.5)*A =
1: (1, 1.5) (2, 3.0) (3, 4.5)
2: (1, 6.0) (2, 7.5) (3, 9.0)
3: (1, 10.5) (2, 12.0) (3, 13.5)

A+B =
1: (1, 2.0) (2, 2.0) (3, 4.0)
2: (1, 4.0) (2, 5.0) (3, 6.0)
3: (1, 8.0) (2, 9.0) (3, 10.0)

A+A =
1: (1, 2.0) (2, 4.0) (3, 6.0)
2: (1, 8.0) (2, 10.0) (3, 12.0)
3: (1, 14.0) (2, 16.0) (3, 18.0)

B-A =
1: (2, -2.0) (3, -2.0)
2: (1, -4.0) (2, -5.0) (3, -6.0)
3: (1, -6.0) (2, -7.0) (3, -8.0)

A-A =

Transpose(A) =
1: (1, 1.0) (2, 4.0) (3, 7.0)
2: (1, 2.0) (2, 5.0) (3, 8.0)
3: (1, 3.0) (2, 6.0) (3, 9.0)

A*B =
1: (1, 4.0) (2, 3.0) (3, 4.0)
2: (1, 10.0) (2, 6.0) (3, 10.0)
3: (1, 16.0) (2, 9.0) (3, 16.0)

B*B =
1: (1, 2.0) (2, 1.0) (3, 2.0)
3: (1, 2.0) (2, 1.0) (3, 2.0)
```

Notice that the rows are to be printed in column sorted order, and zero rows are skipped altogether. On the other hand, the input file may give the matrix entries in any order.

Matrix ADT Specifications

In addition to the main program Sparse.java and the altered List.java from pa1, you will implement a Matrix ADT in a file called Matrix.java, which defines the Matrix class. This class will contain a private inner class (similar to Node in your List ADT) that encapsulates the column and value information corresponding to a matrix entry. You may give this inner class any name you wish, but I will refer to it here as Entry. Thus Entry will have two fields that store types int and double respectively. Entry must also contain its own equals() and toString() methods which override the corresponding methods in the Object superclass. Your Matrix class will represent a matrix as an Array-of-Lists of Entry Objects. It is highly recommended that these Lists be maintained in column sorted order. Your Matrix ADT will export the following operations.

```
// Constructor
Matrix(int n) // Makes a new n x n zero Matrix.  pre: n>=1

// Access functions
int getSize() // Returns n, the number of rows and columns of this Matrix
int getNNZ() // Returns the number of non-zero entries in this Matrix
public boolean equals(Object x) // overrides Object's equals() method

// Manipulation procedures
void makeZero() // sets this Matrix to the zero state
Matrix copy()// returns a new Matrix having the same entries as this Matrix
void changeEntry(int i, int j, double x)
    // changes ith row, jth column of this Matrix to x
    // pre: 1<=i<=getSize(), 1<=j<=getSize()
Matrix scalarMult(double x)
    // returns a new Matrix that is the scalar product of this Matrix with x
Matrix add(Matrix M)
    // returns a new Matrix that is the sum of this Matrix with M
    // pre: getSize()==M.getSize()
Matrix sub(Matrix M)
    // returns a new Matrix that is the difference of this Matrix with M
    // pre: getSize()==M.getSize()
Matrix transpose()
    // returns a new Matrix that is the transpose of this Matrix
Matrix mult(Matrix M)
    // returns a new Matrix that is the product of this Matrix with M
    // pre: getSize()==M.getSize()

// Other functions
public String toString() // overrides Object's toString() method
```

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 functions copy(), changeEntry(), scalarMult(), and transpose() should have cost $\Theta(n+a)$ when applied to A, in worst case. Functions add() and sub() can be implemented to run in time $\Theta(n+a+b)$, and mult() should run in time $\Theta(n+a\cdot b)$. It will be helpful to include a private function with signature

```
private static double dot(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 `mult()`. Similar helper functions for the operations `add()` and `sub()` will also be highly useful.

What to Turn In

Your project will be structured in three files: `Sparse.java`, `Matrix.java`, and `List.java`. The main program, Sparse, will handle the input and output files and is the client of Matrix, which is itself the client of List. Note that Sparse is not itself a direct client of List, since it need not call any List operations. You will also write separate client modules `ListTest.java` and `MatrixTest.java` to test the List and Matrix ADTs in isolation. Students often ask what should be the contents of these test files. In each case include enough calls to ADT operations to convince the grader that you did in fact test your List and Matrix ADT modules. The best way to do this is to *actually use* them for this purpose. At minimum they should call every public function in their respective ADT modules at least once.

Also submit a `README` file and a `Makefile` that creates an executable jar file called `Sparse`. Thus seven files in all will be turned in:

```
Sparse.java
Matrix.java
List.java
MatrixTest.java
ListTest.java
Makefile
README
```

Submit these to the assignment name pa3 by the due date. As always, start early and ask questions.