Csci 1933 Project #3: A Linked List Sparse Matrix Implementation

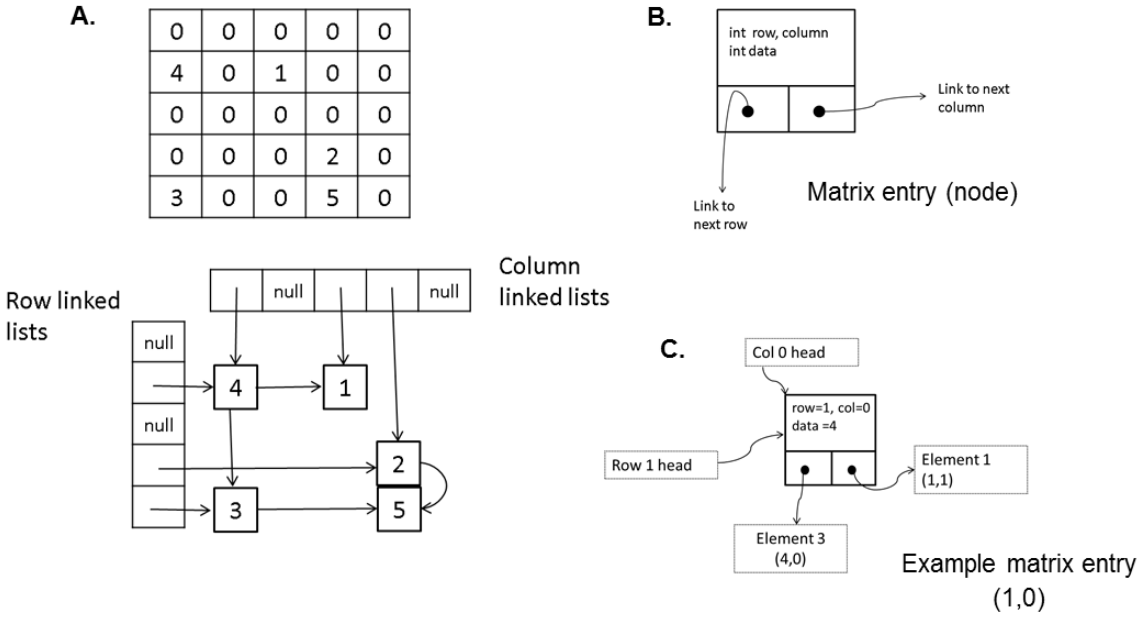
Due date: 11/7/2018, before midnight

100 points.

Summary. Many real-world applications rely on processing of large 2D matrices that often have few non-zero elements relative to the total number of entries in the matrix. For Project #3, you will use linked lists to implement a memory-efficient, sparse 2-dimensional matrix data structure. Then, you will test your data structure by loading in some large 800x800 matrices, which we have constructed to contain hidden patterns that will be visible if you have implemented your matrix correctly.

Overview of a Linked List Sparse Matrix Data Structure Design.

A typical linked list implementation of a sparse matrix relies on two sets of interconnected linked lists: one for the rows of the matrix, and one for the columns of the matrix. On both sides, an array can be used to keep track of the heads of each linked list, and each element is part of one column linked list as well as one row linked list. An example of the logical structure a linked list is shown below for a 5x5 matrix (Figure A).



Each of the matrix elements will need to store their data, row and column indices, and two links: one to the next element in that column (if it exists) and one to the next element in that row (if it exists) (Figure B). The data for the element (1,0) of the sparse matrix is shown as an example in Figure C. Your goal will be to implement a class MatrixEntry that stores all of this information as well as a class SparseIntMatrix that implements the linked list structure shown above.

Part I. (20 points) **Defining and Implementing a MatrixEntry class**

The MatrixEntry class contains all of the data members necessary for storing the data and links for each element of the sparse matrix. Define and implement this class including the following methods:

- MatrixEntry(int row, int col, int data) (constructor); input: the row, column, and data associated with this matrix element; output: an instance of the MatrixEntry class
- int getColumn(); input: none; output: an integer corresponding to the column of this entry
- void setColumn(int col); input: integer corresponding to the column of this entry; output: none
- int getRow(); input: none; output: an integer corresponding to the row of this entry
- void setRow(int row); input: integer corresponding to the row of this entry; output: none
- int getData(); input: none; output: an integer corresponding to the data associated with this entry
- void setData(int data); input: integer with the data for this matrix entry; output: none:
- MatrixEntry getNextRow(); input: none; output: output: a MatrixEntry reference for this entry's next row. (The next row is defined as the element with the same column number, and a different row number)
- void setNextRow(MatrixEntry el); input: MatrixEntry reference of this entry's next row; output; output: none
- MatrixEntry getNextCol(); input: none; output: a MatrixEntry reference for this entry's next column. (The next col is defined as the MatrixEntry element with the same row number, and a different col number)
- void setNextCol(MatrixEntry el); input: MatrixEntry reference of this entry's next column; output: none

Data members: any that are necessary to maintain the object's state and implement the methods above. The MatrixEntry class should be defined as a separate class from the SparseIntMatrix class described below, and all data members should be declared private. Your

SparseIntMatrix class client should interact with MatrixEntry objects through the public accessor/mutator methods.

Part II. (10 points) **Writing Unit Tests for MatrixEntry**

Before you implement your SparseIntMatrix class, you will need to write unit tests, using JUnit, to ensure your MatrixEntry class is working as expected. You are required to implement at least *four* unit tests, but are free to write more than that if you wish. Please write the tests in a java file named **MatrixEntryTest.java** The unit tests need to assert the following features:

- The MatrixEntry constructor properly initializes the row, column, and data member variables.
- setRow(), setCol(), and setData() properly update the MatrixEntry's row, column, and data variables.
- setNextColumn() updates the correct MatrixEntry reference variable, and getNextColumn() returns the correct MatrixEntry reference variable.
- setNextRow() updates the correct MatrixEntry reference variable, and getNextRow() returns the correct MatrixEntry reference variable.

You are free (and encouraged) to use existing JUnit tests we have provided you for this project as well as previous projects and labs as a resource for creating your testing class.

Hint: Because the row, col, data, nextRow, and nextCol variables in your MatrixEntry class must be private, you will need to use your getter methods (i.e. getRow() and getCol()) to ensure the appropriate setter methods are working).

Part III. (50 points) **Defining and Implementing a SparseIntMatrix class**

Your third goal is to design and implement the SparseIntMatrix class, which should provide the basic functionality of a matrix, but use the linked list structure described above (NOTE: you should not just use a 2D array here—you will receive 0 points for a 2D array-based solution, however you may use two single dimension arrays as shown in the overview).

Note: For full credit, you must keep properly link both rows and columns for each element in the matrix.

The SparseIntMatrix class should have all of the following methods:

- SparseIntMatrix(int numRows, int numCols) (constructor); Input: integers with the number of rows and number of columns in this matrix; output: instance of this class
- SparseIntMatrix(int numRows, int numCols, String inputFile) (constructor); Input: integers with the number of rows and number of columns in this matrix, and a String with the filename of a file with matrix data. The format of the input file should be comma-

delimited lines with the row,column,data of each element. This constructor should populate the matrix with this data; output: instance of this class

- int getElement(int row, int col); input: integers with the row and column of the desired element; output: corresponding element (integer) if one exists or zero otherwise.
- boolean setElement(int row, int col, int data); input: integers with the row and column of the element to be set and an integer with the matrix data; output: boolean indicating if operation was successful (e.g. row/col were in the correct range)
- boolean removeElement(int row, int col, int data); input: integers with the row and column of the element to be removed; output: boolean indicating if an element was removed or not (false indicates that the element didn't previously exist or that the row/col were out of range, true indicates that it did and has been removed). Any links to/from the element that was removed should be properly updated in the matrix.
- int getNumCols(); input: none; output: integer with the number of columns in this matrix.
- int getNumRows(); input: none; output: integer with the number of rows in this matrix
- boolean plus(SparseIntMatrix otherMat); input: another sparse matrix to be added to the current one. This method should update the state of the current object; output: boolean indicating if addition was successful (matrices should be identical dimensions)
- boolean minus(SparseIntMatrix otherMat); input: another sparse matrix to be subtracted from the current one. This method should update the state of the current object; output: boolean indicating if addition was successful (matrices should be identical dimensions)

Data members: any that are necessary to implement the methods above using the linked list structure shown in Figure A. All data members should be declared private.

Part IV. (10 points) **Testing your SparseIntMatrix Implementation**

Now that you've finished implementing the SparseIntMatrix class, write a main method in a separate class to test its functionality. We have provided three data files for you to test your implementation: matrix1_data.txt, matrix2_data.txt, and matrix2_noise.txt. All three of these files are comma-delimited with the following format on each line: <row>,<column>,<matrix element>, and they each contain a 800x800 sparse integer matrix.

We have also provided a helper class *MatrixViewer* and *EasyBufferedImage* which will allow you to visualize the patterns in these matrices. If you have implemented your matrix construction correctly matrix1_data.txt will contain an obvious pattern, which you can view by putting the following code inside a main method:

```
SparseIntMatrix mat = new SparseIntMatrix(800,800,"matrix1_data.txt");
MatrixViewer.show(mat);
```

Matrix2_data.txt also contains a pattern, but it is obscured by random noise. All you'll need to do is load in the matrix in matrix2_noise.txt and subtract this matrix off of matrix2_data (i.e. matrix2_data.minus(matrix2_noise). Check for a pattern before and after removing the noise— if you've implemented your matrix arithmetic correctly, the pattern should be obvious. Be sure to save all of the code that you use to check these patterns in your main method. To receive the full 10 points for this section, you must successfully load and properly use all three example files described above.

Part V. (10 points) **Analysis and Comparison of Space Efficiency of your SparseIntMatrix class**

Answer the following questions about the memory used by your SparseIntMatrix implementation relative to a standard 2-dimensional integer array. Please include your answers in the header of your SparseIntMatrix.java file.

Assume that for the MatrixEntry class, each of the following require 1 memory unit: row label, column label, matrix element data, link to next row element, link to next column element. Ignore the overhead of the array that stores the links to the row/column linked lists— only consider the memory used in storing matrix elements. Also, you can assume that each element of a 2D array will require 1 memory unit. Be sure to justify how you arrived at your answers to receive full credit.

(a) For a square matrix of *N x N* elements with *m* non-zero elements, how much memory is required for the SparseIntMatrix implementation? How much for a standard 2D array implementation?

(b) For a square matrix, assume *N=100,000* and *m=1,000,000*. Is the SparseIntMatrix implementation more space-efficient, and if so, by how much? (i.e. was it worth all of the effort you just went through implementing it?) For what value of *m* does the 2D array implementation become more space-efficient than the SparseIntMatrix implementation?

Submitting your finished assignment:

Once you've completed Project #3, create a zip file with all of your Java files (.java) and submit them through Moodle.

Working with a partner:

As discussed in lecture, you may work with one partner to complete this assignment (max team size = 2). If you choose to work as a team, please only turn in one copy of your assignment. At the top of your class definition that implements your main program, include in the comments both of your names and student IDs. In doing so, you are attesting to the fact that both of you have contributed substantially to completion of the project and that both of you understand all code that has been implemented.