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

Due date: 3/30/2022, 11:55pm

Total Points: 100

Instructions:

Please read and understand these expectations thoroughly. Failure to follow these instructions could negatively impact your grade. Rules detailed in the course syllabus also apply but will not necessarily be repeated here.

• **Due:** The project is due on Wednesday, March 30th by midnight on Canvas.

• **Submission:** Please submit a .zip file on Canvas containing your src/ folder with all .java files and the README.txt file. If you are working with a partner, only one student should submit a project on Canvas. Submissions in the incorrect format may receive a penalty. Submissions with missing code will also be penalized. Make sure you submit all your source code!

• **Partners:** You may work alone or with one partner. Please place your partner's and your names and x500s in a comment in each .java file you submit and in the README. Do not share code with students other than your partner.

• **Code Sharing:** If you use online resources to share code with your partner, please ensure the code is private. Public repositories containing your code are prohibited because other students may copy your work.

• **Java Version:** The TAs will grade your code using Java version 11. Please ensure you have this version of Java installed.

• **Questions:** Please post questions related to this project on the Piazza forum.

Summary.

Many real-world applications rely on processing large 2D matrices with 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 1000x1000 matrices. We have constructed these matrices to contain patterns that make up a hidden image that will be visible if you have implemented your matrix correctly.

Files Provided:

1. MatrixViewer.java
2. EasyBufferedImage.java

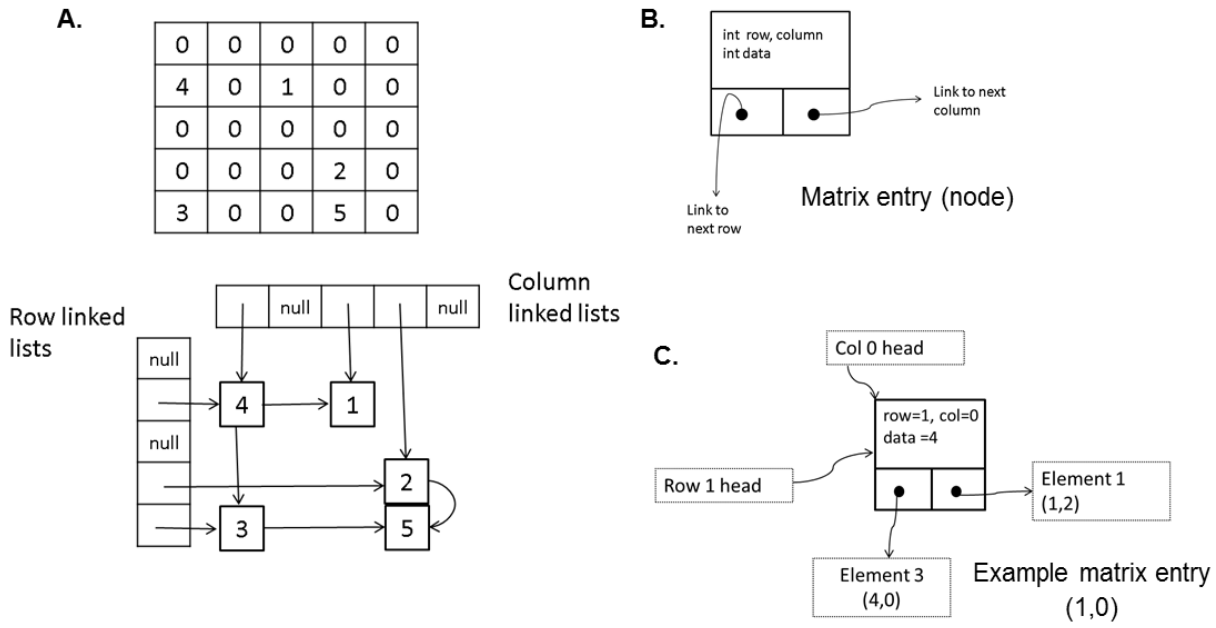
3. matrix1_data.txt, matrix2_data.txt, matrix2_noise.txt

Files To Create:

1. MatrixEntry.java
2. MatrixEntryTest.java
3. SparseIntMatrix.java
4. SparseIntMatrixTest.java

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 matrix element is in one column linked list as well as in one row linked list. An example of the logical structure for this implementation of linked lists is shown below for a 5x5 sparse 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). Figure C shows an example of the data for element (1,0) in the sparse matrix. 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 and necessary member variables:

- MatrixEntry(int row, int col, int data) (constructor); input: the row, column, and data associated with this matrix element
- 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: 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: 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**. You must use public accessor/mutator methods to work with MatrixEntry objects in your SparseIntMatrix class.

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

Before you implement your SparseIntMatrix class, you will need to write unit tests to ensure your MatrixEntry class works as expected. You must implement at least ***four*** **unit tests**, but are free to write more than that. Please write the tests in a java file named **MatrixEntryTest.java.**

The unit tests need to assure that the following features function correctly:

- 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 not required to follow any specific format for your test class. The only requirement is that running your test class shows the user whether or not MatrixEntryTest.java is working correctly.

Hint: Because the row, col, data, nextRow, and nextCol variables in your MatrixEntry class must be private, you will need to use your public accessor methods (e.g. 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 will provide the basic functionality of a matrix, but use the linked list structure described above

Important Reminders:

- you should **not just use a 2D array here**—you will receive no credit for a 2D array-based solution, however you may use two single dimension arrays as shown in the overview).
- For full credit, you must 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
- 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, and data of each element. This constructor should populate the matrix with this data

- 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 (i.e. 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 which 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 have finished implementing the SparseIntMatrix class, write a main method in **SparseIntMatrixTest.java** 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 contain a 1000x1000 sparse integer matrix. You should assume zero-based indexing for the rows/columns, which means that the element in the first row/first column is (0,0). Your code should discard any input data elements that are out-of-range.

We have also provided the helper classes:

1. *MatrixViewer*
2. *EasyBufferedImage*

to help you visualize the patterns in these matrices. If you have implemented your matrix construction correctly, you should be able to view a clear pattern in matrix1_data.txt after adding the following code inside a main method:

```
SparseIntMatrix mat = new SparseIntMatrix(1000,1000,"matrix1_data.txt");
MatrixViewer.show(mat);
```

Matrix2_data.txt also contains a pattern, but it is obscured by random noise. You will need to load in the matrix in matrix2_noise.txt and subtract this matrix off of matrix2_data (i.e. matrix2_data.minus(matrix2_noise) to view it clearly. 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 your SparseIntMatrix implementation uses relative to a standard 2-dimensional int array. Please include your answers in your README 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, and each element of a 2D array. 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. Be sure to fully explain 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 Canvas. You are required to include a README.txt file containing the following:

• Group member's names and x500s

• Contributions of each partner (if working with a partner)

• How to compile and run your program

• Any assumptions

• Additional features that you implemented (if applicable)

• Any known bugs or defects in the program

• Any outside sources (aside from course resources) consulted for ideas used in the project, in the format:

- idea1: source

- idea2: source

- idea3: source

• Include the statement: "I(We) certify that the information contained in this README file is complete and accurate. I(We) have both read and followed the course policies in the 'Academic Integrity - Course Policy' section of the course syllabus." and type your name(s) underneath.

• Include your answers to the analysis questions listed above.

There is a ten-point deduction for a missing or incomplete README.

Working with a partner:

As discussed in lecture, you may work with one partner to complete this assignment. 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.