Sparse Matrix Assignment

Assignment objective:

Implement a sparse matrix of integers along with the arithmetic operations to perform on sparse matrices. Let $A_{m, n}$ be a matrix of m × n integer elements organized in m rows and n columns. If many of the matrix elements are 0, we call the matrix sparse. To store a sparse matrix in memory, one can do better than storing all the elements since many of them are already known to be 0. Thus, the required space reduces from m x n (for a general matrix) to n + m + k_A, where k_A is the actual number of non–zero elements in the sparse matrix A. For such efficient storage one can use a data structure using linked lists as illustrated in Figure 1.

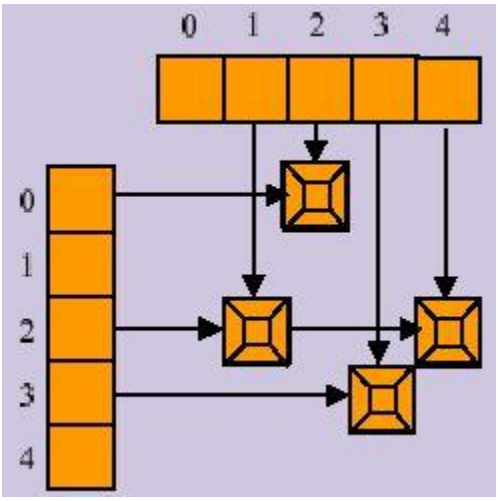

Figure 1. A 5 x 5 sparse matrix represented with linked lists.

Design:

1. Using the sparse matrix example illustrated in Figure 1, the element in cell (2,1) is the first non–zero element on row 2. It is the first element in the linked list for row 2 as well as the only element of the linked list for column 1. The next non–zero element on row 2 is in cell (2,4) which is in the linked list for row 2 after the element in cell (2, 1). The element in cell (2,4) is the only element of the linked list for column 4. The element in cell (0, 2) is the only non–zero element on row 0 as well as column 2. The element in cell (3, 3) is the only non–zero element on row 3 as well as column 3. Finally, row 1, row 4, and column 0 have no non–zero elements.

2. Create the Element class as defined by its UML class diagram given in figure 2 below. You must make the Element class a private nested class of the SparseMatrix class.

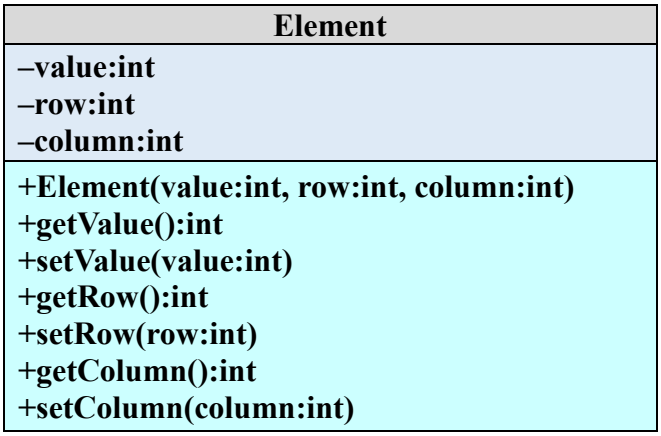

Figure 2. Element's UML class diagram

3. Implement the SparseMatrix class for handling sparse matrices as defined by its UML class diagram given in figure 3 below utilizing Java's linked list data structure:

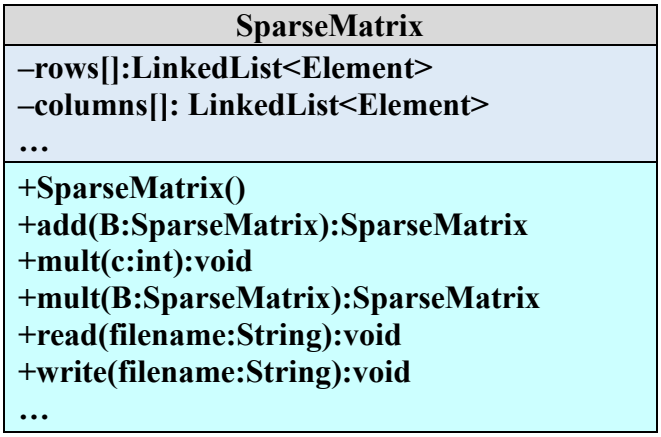

Figure 3. SparseMatrix's UML class diagram

You can add additional attributes to SparseMatrix but they must be declared private. You can add additional (helper) methods to SparseMatrix but they must be declared private.

4. The SparseMatrix **constructor** creates an empty sparse matrix.

5. The SparseMatrix **add** method adds two sparse matrices together returning the resulting SparseMatrix. For example, A.add(B) adds the two sparse matrices A and B of the same size m x n, returning the resulting sparse matrix C. This operation needs to have time complexity $O(n+m+k_A+k_B)$, not $O(m*n)$, where k_A is the actual number of non–zero elements in the sparse matrix A and k_B is the actual number of non–zero elements in the sparse matrix B.

6. The SparseMatrix **mult(c:int)** method multiplies the elements of a sparse matrix by a constant c. For example, A.mult(c) multiplies the elements of the sparse matrix A by the constant c. This operation needs to have time complexity O(*1*), not O(*m*n*), and not even O(k_A), where k_A is the actual number of non–zero elements in the sparse matrix A. Think about what this requires from the SparseMatrix class.

7. The SparseMatrix **mult(B:SparseMatrix)** method multiplies two sparse matrices together returning the resulting SparseMatrix. For example, A.mult(B) multiplies two sparse matrices A of size m x n and B of size n x p, returning the resulting sparse matrix C of size m x p. This multiplication needs to have as good a time complexity as you can manage.

8. The SparseMatrix **read** method reads in a sparse matrix from a text file. For example, A.read(f) reads in a sparse matrix from input file f into SparseMatrix A. This operation needs to have time complexity O($n+m+k_A$), not O(*m*n*), where k_A is the actual number of non–zero elements in the sparse matrix A. Number 11 below describes the format of the input file f.

9. The SparseMatrix **write** method writes the sparse matrix to a text file. For example, A.write(f) writes the SparseMatrix A out to file f using the same exact format as the input files described in number 11 below. The file produced by the write method must be able to be read using the SparseMatrix read method. This method, like A.read(f), needs to have time complexity O($n+m+k_A$), not O(*m*n*), where k_A is the actual number of non–zero elements in the sparse matrix A.

10. In this assignment in Blackboard are two zip files containing two sets of test case files: large sparse matrices in **Large_Test_Files.zip** and small sparse matrices in **Small_Test_Files.zip**.
    a. A test case file with **addn** in the filename is the sparse matrix resulting from adding together two sparse matrices. For example, **A_addn_B_1_large.txt** is the resulting sparse matrix from adding the sparse matrix in file **A_test_1_large.txt** to the sparse matrix in file **B_test_1_large.txt**.
    b. A test case file with **mult** in the filename is the sparse matrix resulting from multiplying together two sparse matrices. For example, **A_mult_C_1_large.txt** is the resulting sparse matrix from multiplying together the sparse matrix in file **A_test_1_large.txt** with the sparse matrix in file **C_test_1_large.txt**.
    c. A test case file with **mult_n** in the filename is the sparse matrix resulting from multiplying a sparse matrix by a constant. For example, **A_mult_n_1_large.txt** is the resulting sparse matrix from multiplying the sparse matrix in file **A_test_1_large.txt** by the constant listed in the comment line in file **A_mult_n_1_large.txt**.
    d. The test case files with **test** in the filename are the sparse matrices to test the operations of your SparseMatrix class. You should first use the small test case files to test your SparseMatrix code. When satisfied that your code works, then use the large test case files.
    e. You need to write a driver class (containing the **public static void main(String args[])** method) called **Assignment1** to test your SparseMatrix class. The driver class must be in a separate file (**Assignment1.java**) from the SparseMatrix class. The driver code needs to test each SparseMatrix operation: read, add, the two mult operations, and write. I'll use your driver class as well as my own version of a driver class to test your SparseMatrix class.

11. The test case files have the following format:
    a. The line(s) starting with the pound sign (#) contains comments. Comment lines are always the first lines of the file. A file may or may not have comment lines. The read method disregards all comment lines. The write method doesn't need to write a comment line (but can) except when the SparseMatrix **mult(c:int)** method executes for the sparse matrix written. In this case, the write method writes a comment line at the beginning of the text file containing the scaling (constant) factor. For example, if A.mult(20) executes during a test run then the write method writes the following comment line when writing the sparse matrix A to a file.
    ```
    # const = 20
    ```
    b. The first line not starting with a pound sign defines the size of the matrix: the number of rows, the number of columns, and the number of non–zero elements in the sparse matrix. For example:
    ```
    m = 3   n = 10   numItems = 18
    ```
    is a sparse matrix with 3 rows, 10 columns, and 18 non–zero elements.
    c. Each line (after the above first defining line) contains information about exactly 1 non–zero element: the row and column coordinates of the element, and the element's value. For example:
    ```
    2,8 = 5708
    ```
    is the sparse matrix element in row 2 and column 8 with value 5708.
    d. The elements in the test file are ordered by column first, and then by row. That is, element (4,1) precedes element (2,3) since column 1 comes before column 3, which itself precedes element (5,3) since both have the same column 3 but row 2 precedes row 5. Examine the test case files for more examples.

12. **Tip:** Make your code as modular as possible, not placing all your code in a single method. You can create as many methods as you need in addition to the methods already required in the class. Methods being reasonably small follow the guidance that **A function does one thing and does it well**. You will lose a lot of points for code readability if you don't make your code as modular as possible. But do not go overboard creating methods. Your common sense guides your creation of methods.

13. Do **not** type any comments in your program. If you do a good job of programming by following the advice in number 12 above, then it will be easy for me to determine the task of your code.

Grading Criteria:

The assignment is worth a total of 20 points, broken down as follows:

1. If your code does not implement the task described in this assignment, then the grade for the assignment is zero.
2. If your program does not compile successfully then the grade for the assignment is zero.
3. If your program produces runtime errors which prevents the grader from determining if your code works properly then the grade for the assignment is zero.

If the program compiles successfully and executes without significant runtime errors, then the grade computes as follows:
Followed proper submission instructions, 3 points:

1. Was the file submitted a zip file?
2. The zip file has the correct filename.
3. The contents of the zip file are in the correct format.

Code implementation and Program execution, 12 points:

- The code performs all the tasks as described in the assignment description.
- The code is free from logical errors.
- The sparse matrix operations have the appropriate run-time.
- Program input, the program properly processes the input.
- Program output, the program produces the proper results for the assignment.

Code readability, 5 points:

- Good variable, method, and class names.
- Variables, classes, and methods that have a single small purpose.
- Consistent indentation and formatting style.
- Reduction of the nesting level in code.

Late submission penalty: assignments submitted after the due date are subjected to a 2-point deduction for each day late.

Late submission policy: you **CAN** submit your assignment early, before the due date. You are given plenty of time to complete the assignment well before the due date. Therefore, I do **NOT** accept any reason for not counting late points if you decide to wait until the due date (and the last possible moment) to submit your assignment and something happens to cause you to submit your assignment late. I only use the date submitted, ignoring the time as well as Blackboard's late submission label.

Submission Instructions:

Go to the folder containing your **Assignment1.java** and **SparseMatrix.java** files, select them, and place only them in a Zip file. The file can **NOT** be a **7z** or **rar** file! Follow the directions below for creating a zip file depending on the operating system running on the computer containing your assignment's two java code files.

Creating a Zip file in Microsoft Windows (any version):

1. Right-click any of the two files to display a pop-up menu.
2. Click on **Send to**.
3. Click on **Compressed (zipped) Folder**.
4. Rename your Zip file as described below.
5. Follow the directions below to submit your assignment.

Creating a Zip file in MacOS:

1. Click **File** on the menu bar.
2. Click on **Compress**.
3. MacOS creates the file **Archive.zip**.
4. Rename **Archive** as described below.
5. Follow the directions below to submit your assignment.

Save the Zip file with the filename having the following format:
your last name,
followed by an underscore _,
followed by your first name,
followed by an underscore _,
followed by the word **Assignment1**.
For example, if your name is John Doe then the filename would be: **Doe_John_Assignment1**

Once you submit your assignment you will not be able to resubmit it!
Make absolutely sure the assignment you want to submit is the assignment you want graded.
There will be **NO** exceptions to this rule!

You will submit your Zip file via your CUNY Blackboard account.
The only accepted submission method!

Follow these instructions:

Log onto your CUNY Blackboard account.
Click on the CSCI 313 course link in the list of courses you are taking this semester.
Click on the **Assignments** tab in the red area on the left side of the webpage.
You will see the **Sparse Matrix Assignment**.
Click on the assignment.
Upload your Zip file and then click the submit button to submit your assignment.

Due Date: Submit this assignment on or before 11:59 p.m. Thursday, April 4, 2024.