Table of Contents

[1.0 Data Structure Used 1](#_Toc452214622)

[2.0 Classes Used 2](#_Toc452214623)

[2.1 ColumnNode Class 2](#_Toc452214624)

[2.2 RowNode Class 2](#_Toc452214625)

[2.3 SM Class 2](#_Toc452214626)

[2.4 Justification 3](#_Toc452214627)

[3.0 Functionalities Implementation 3](#_Toc452214628)

[3.1 SM Constructor 3](#_Toc452214629)

[3.2 Read Elements 4](#_Toc452214630)

[3.3 Print Matrix 5](#_Toc452214631)

[3.4 Adding sparse matrices together 6](#_Toc452214632)

[3.5 SM Destructor 6](#_Toc452214633)

[4.0 Main Function Snippet & Screenshots 7](#_Toc452214634)

[4.1 Snippet 7](#_Toc452214635)

[4.2 Screenshots 8](#_Toc452214636)

# 1.0 Data Structure Used

As the objective of this assignment was to store sparse matrix values in such a way to ensure the least amount of memory usage, while still being efficient, I had decided to use **linked lists** as the data structure to represent the sparse matrix and to store only the rows and columns that are going to hold a value, hence non-zero values only.

Essentially, there are two linked lists. The sparse matrix object consists of a linked list of the rows, which links each row node to the next respective row node, while each row node consists of a linked list of its column nodes, with each column node pointing to the next column node in that row node. This allows for only the rows that are going to hold columns that are going to hold values to be created and no other row nodes or column nodes which are unnecessary will be created. This will allow for only the storage space required to be used and as opposed to vectors or arrays, which will “reserve” the amount of memory (based on the number of rows and columns) that the objects will take in them, which are considered unnecessary memory space since sparse matrices are sparsely populated with values, hence not all of the reserved memory will be used, and it will contain random values, which are still going to occupy space in the memory. The usage of tail, which points to the last node of a linked list, has also been implemented, which allows direct access to the last node, which provides a much more efficient implementation of insertion and it can be utilized for adding two matrices, as the nodes are organized in an ascending order, and new nodes can always be inserted at the end, without having to traverse through the list.

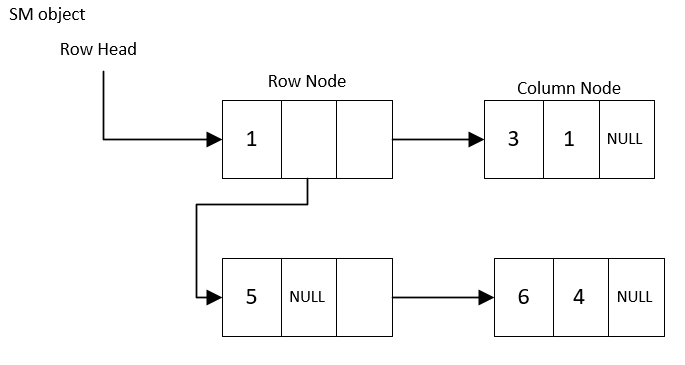


Figure 1. Simple Example of Adopted Data Structure

# 2.0 Classes Used

## 2.1 ColumnNode Class

The ColumnNode class is the basis of the data structure implemented, as in, it holds the value that is going to be inserted in the sparse matrix. This class consists of no functions, as it is the base, such as the NodeType would be the base of a linked list, it only holds a value, column number, and a pointer to the next column node. It consists of a parameterized constructor, which is used to assign the value and column number of the object.

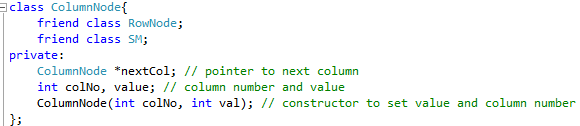


Figure 2. ColumnNode class

## 2.2 RowNode Class

The RowNode class is used as the linked list that will hold all the column nodes for that specific row. It consists of the linked list implementation required to store the column node. It has a row number, column size (which is the number of columns in that specific row), a pointer to the first column, and a pointer to the last column. It consists of deleteFirstCol function, which is used inside the destructor in order to destroy all the columns. The main function in this class is the insertColumn, which will allow to create or overwrite (using the overwriteColumn function) a column that the user inputs. Lastly, there is a printCol function, which will print out all the values in the columns of the specific row.

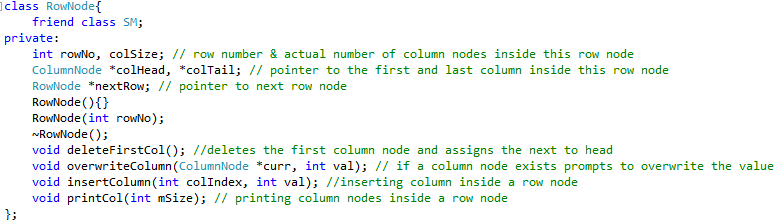


Figure 3. RowNode Class

## 2.3 SM Class

The SM class is the main body of the sparse matrix representation, it consists of all the necessary functions, such as readElements, addSM, and printMatrix, while it consists of private functions, which are used within those functions and the destructor of the class. It consists of a row head pointer and a row tail pointer, which are used to access the first row node of the sparse matrix (and last row node, which is used to create a more efficient the insert implementation).

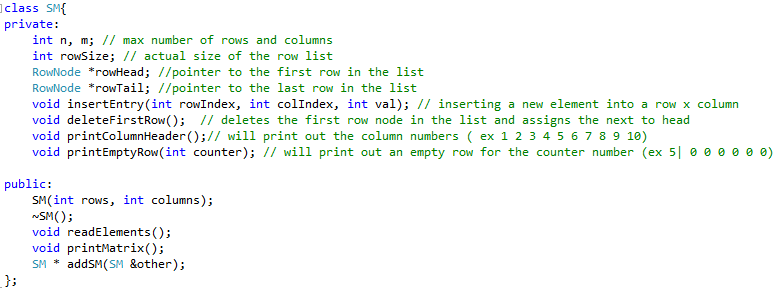


Figure 4. SM Class

## 2.4 Justification

The reason I have chosen to create a ColumnNode class, RowNode class beside the SM class, is because each of the classes have their own set of attributes and functions, and as the logic behind the data structure implemented is that the row node will link all the column nodes in that particular node together, so it is more or less a linked list on its own, with ColumnNode being the node that it is linking together, and the SM class will link all the row nodes together, each of which have their own linked list of column nodes. Alongside that, all the ColumnNode class is declared as friend to the SM and RowNode class, so that its attributes can be used in both classes, and RowNode class is declared as a friend class to the SM class, so that its attributes and functions can be used in the SM class later. Lastly, the reason that ColumnNode class and RowNode class are not combined together into a single class (for example NodeType class) is because they have different data members, as a row node will have 2 pointers, and 1 integer value, while a column node class will have 2 integer values and a single pointer.

# 3.0 Functionalities Implementation

## 3.1 SM Constructor

The sparse matrix is constructed with the number of rows and columns, and the values are assigned to the respective data members inside the SM class, and all the pointers are set to null.

## 3.2 Read Elements

The read element function allows the user to insert an element into the sparse matrix, by getting the triple values, which are row number, column number, and value, and creating a row node, if it does not exist, and a column node, if it does not exist, inside the row node and store the value. The row nodes are all linked together, while the column nodes for each of the existing rows are linked together inside the row node itself. It will also ensure that the elements are placed consecutively in their respective position to ensure better efficiency, such as placing row 5 between 4 and 6. Although quite rare, but still possible, if an element already exists, then it will prompt the user if he/she wants to overwrite it.

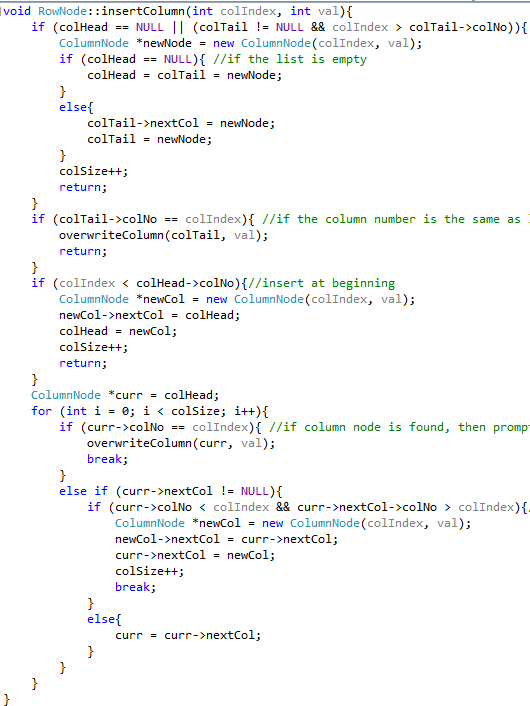


Figure 5. Inserting Column function different cases (inside RowNode class)

## 3.3 Print Matrix

The print function allows for all the rows and columns to be printed out in a tabular form by iterating through all the column nodes of all the rows, and whenever one does not exist, it will print a zero instead, otherwise it will print the value. If a whole row does not exist, then it will print a whole row as zeros.

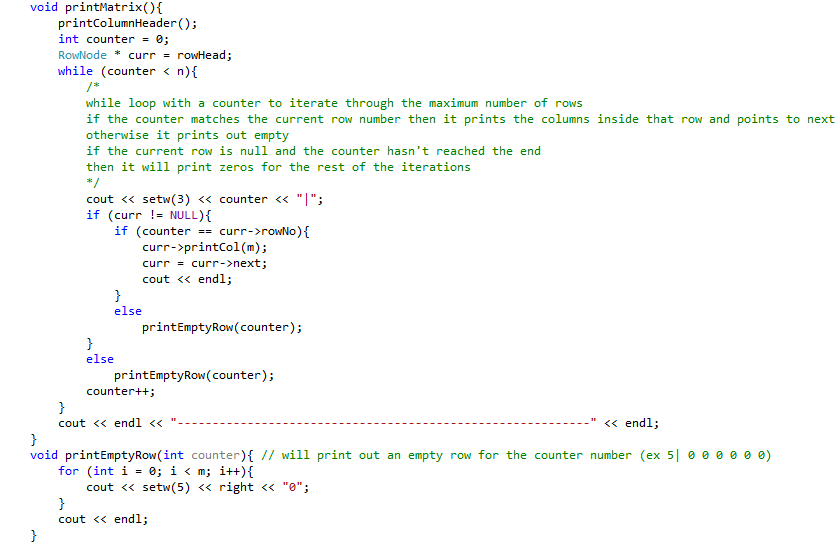


Figure 6. Main Print Matrix implementation code (inside SM class)

## 3.4 Adding sparse matrices together

Two matrices can be added together if their dimensions match, by creating a new sparse matrix with column nodes and row nodes that exist in either both or one of the matrices, and the elements in the column nodes of row nodes that are shared between both sparse matrices are summed up.

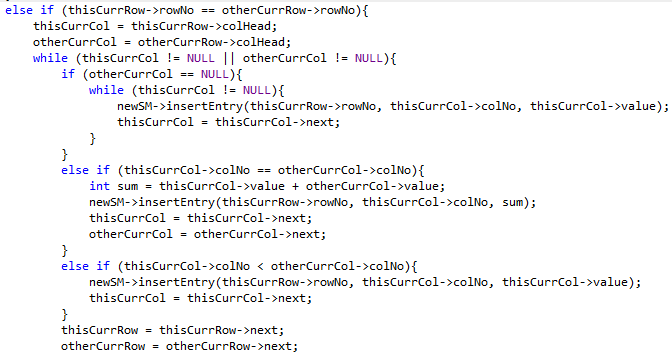


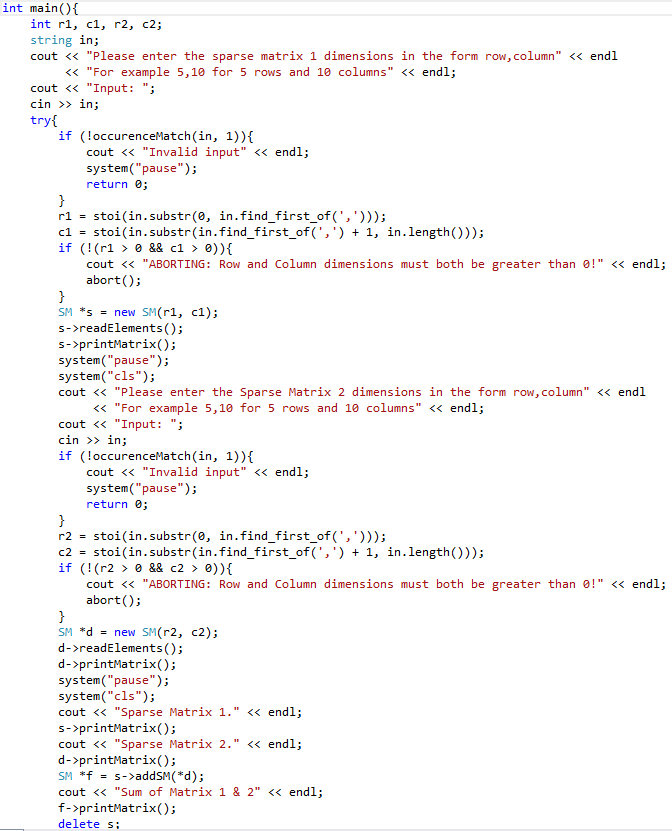
Figure 7. Adding Implementation – when both matrices have the **same row node**

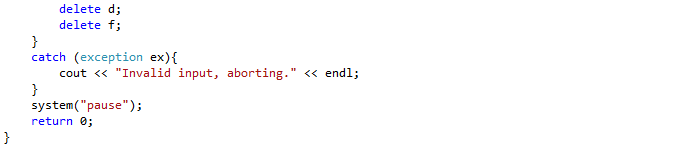
## 3.5 SM Destructor

The destructor allows for all the dynamically allocated memory to be freed up, by deleting all the column nodes for each row, and deleting the row node afterwards, before deleting the sparse matrix object itself.

# 4.0 Main Function Snippet & Screenshots

## 4.1 Snippet



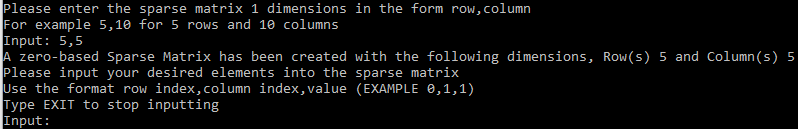


## 4.2 Screenshots

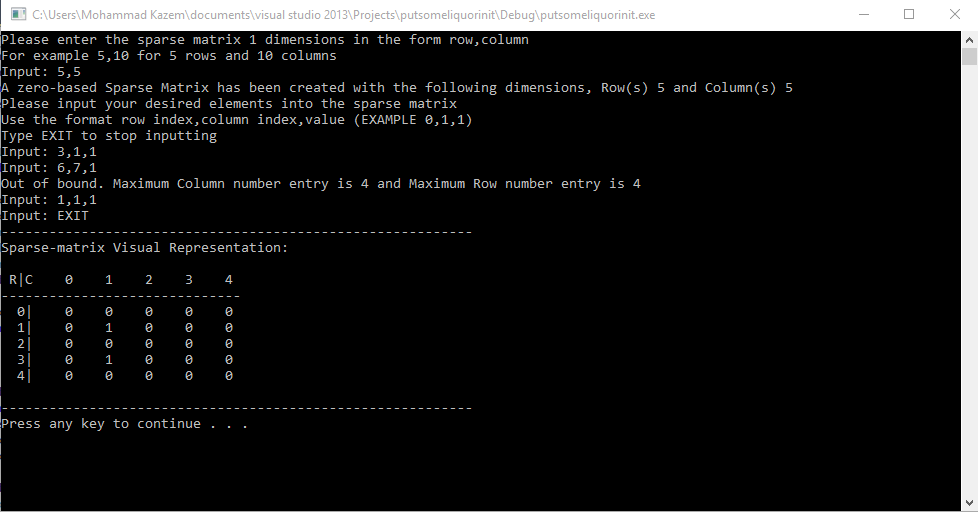
1. Initially, it will prompt the user to input the number of rows and columns for the first matrix



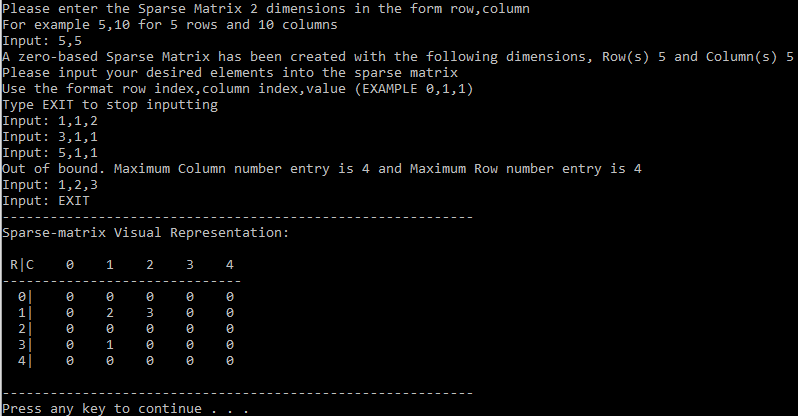
2. Afterwards, it will create the sparse matrix with the desired number of rows and columns, and will prompt the user to input the row index, column index, and value in the format provided.



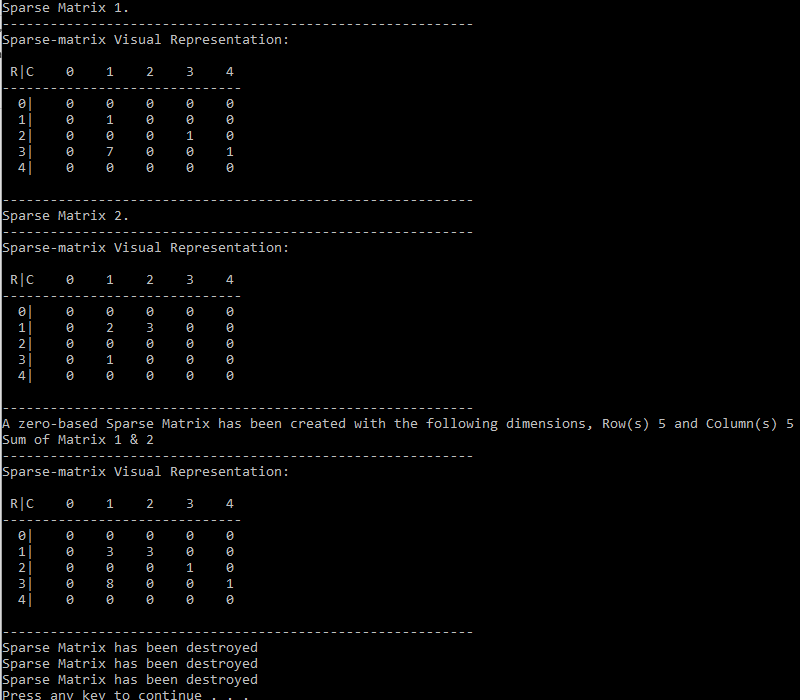
3. The user can input as many triplets as they desire, if the row number or column number is out of bound, it will output a message. If the user attempts to input a value into a row number, column number that already exists, the system will prompt them if they want to overwrite the old value. Lastly, it will print out the matrix once the user inputs exit in any case (EXIT,exit,Exit,eXIT, etc)



4. After the first sparse matrix is printed, it will prompt the user the same set of details as the first one for the second sparse matrix.



5. After the second sparse matrix has been created and printed, it will show the user the addition of the two matrices, and it will destroy all three matrices.



6. If the two matrices are not of the same dimension, then it will alert the user and abort the system.

