Unit 2 Arrays and Matrix

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To create an array, define the data type (like `int`) and specify the name of the array followed by **square brackets []**.

To insert values to it, use a comma-separated list, inside curly braces:

```
int myNumbers[] = {25, 50, 75, 100};
```

We have now created a variable that holds an array of four integers.

Access the Elements of an Array

To access an array element, refer to its **index number**.

Array indexes start with **0**: [0] is the first element. [1] is the second element, etc.

This statement accesses the value of the **first element [0]** in `myNumbers`:

Example

```
int myNumbers[] = {25, 50, 75, 100};
cout<< myNumbers[0];

// Outputs 25
```

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
myNumbers[0] = 33;
```

Example

```
int myNumbers[] = {25, 50, 75, 100};
myNumbers[0] = 33;
```

Loop Through an Array

You can loop through the array elements with the `for` loop.

The following example outputs all elements in the `myNumbers` array:

Example

```
int myNumbers[] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
  cout<<myNumbers[i];
}
```

Set Array Size

Another common way to create arrays, is to specify the size of the array, and add elements later:

Example

```
// Declare an array of four integers:
int myNumbers[4];

// Add elements
myNumbers[0] = 25;
myNumbers[1] = 50;
myNumbers[2] = 75;
myNumbers[3] = 100;
```

Multidimensional Arrays

A multidimensional array is basically an array of arrays.

Arrays can have any number of dimensions. In this chapter, we will introduce the most common; two-dimensional arrays (2D).

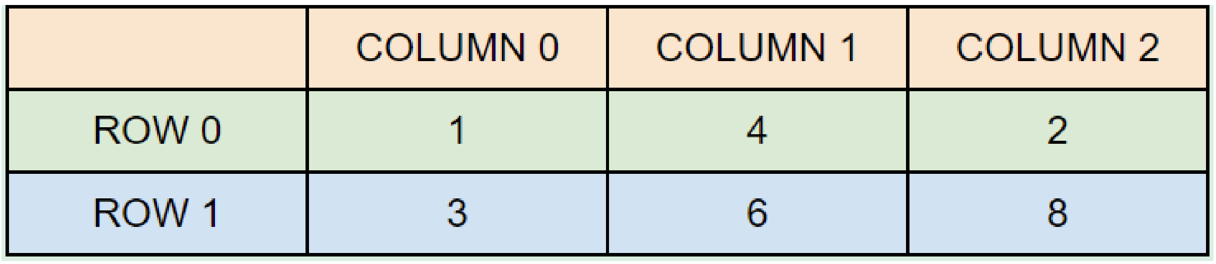
Two-Dimensional Arrays

A 2D array is also known as a matrix (a table of rows and columns).

To create a 2D array of integers, take a look at the following example:

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
```

The first dimension represents the number of rows **[2]**, while the second dimension represents the number of columns **[3]**. The values are placed in row-order, and can be visualized like this:



	COLUMN 0	COLUMN 1	COLUMN 2
ROW 0	1	4	2
ROW 1	3	6	8

Access the Elements of a 2D Array

To access an element of a two-dimensional array, you must specify the index number of both the row and column.

This statement accesses the value of the element in the **first row (0)** and **third column (2)** of the **matrix** array.

Example

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };

cout<<matrix[0][2];  // Outputs 2
```

Loop Through a 2D Array

To loop through a multi-dimensional array, you need one loop for each of the array's dimensions.

The following example outputs all elements in the **matrix** array:

Example

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };

int i, j;
for (i = 0; i < 2; i++) {
  for (j = 0; j < 3; j++) {
    cout<<matrix[i][j];
  }
}
```

Strings

Strings are used for storing text/characters.

For example, "Hello World" is a string of characters.

```
char greetings[] = "Hello World!";

char greetings[] = {'H', 'e', 'l', 'l', 'o', '
', 'W', 'o', 'r', 'l', 'd', '!', '\0'};
```

overloading

Types of overloading in are:

- Function overloading
- Operator overloading

Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in . It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof

- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}
     void operator ++()         {
        num = num+2;
     }
     void Print() {
         cout<<"The Count is: "<<num;
     }
};
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output:

```
The Count is: 10
```

Matrix addition

```
#include <iostream>
using namespace std;
class A
{

    int x;
      public:
      A(){}
    A(int i)
    {
       x=i;
    }
    void operator+(A);
    void display();
};

void A :: operator+(A a)
{

    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

Output:

```
The result of the addition of two objects is : 9
```

```
// C++ program for the matrix multiplication
#include "bits/stdc++.h"
#define rows 50
#define cols 50
using namespace std;

int N;

// Class for Matrix operator overloading
class Matrix {

    // For input Matrix
    int arr[rows][cols];

public:
    // Function to take input to arr[][]
    void input(vector<vector<int> >& A);
    void display();

    // Functions for operator overloading

    void operator*(Matrix x);
};

// Functions to get input to Matrix
// array arr[][]
void Matrix::input(vector<vector<int> >& A)

 void Matrix::operator*(Matrix x)

{
    // To store the multiplication
    // of Matrices
    int mat[N][N];

    // Traverse the Matrix x
    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            // Initialise current block
            // with value zero
            mat[i][j] = 0;

            for (int k = 0; k < N; k++) {
                mat[i][j] += arr[i][k]* (x.arr[k][j]);
            }
        }
    }
```

We can conceptualize a two-dimensional array as a rectangular grid of elements divided into rows and columns. We use the `(row, column)` notation to identify an element.

Creating two-dimensional arrays

There are three techniques for creating a two-dimensional array in Java:

- Using an initializer
- Using the keyword `new`
- Using the keyword `new` with an initializer

```
double[][] temperatures1 = { { 20.5, 30.6, 28.3 }, { -38.7, -18.3, -16.2 } };
```

- `double[][] temperatures2 = new double[2][3];`
- `double[][] temperatures3 = new double[][] { { 20.5, 30.6, 28.3 }, { -38.7, -18.3, -16.2 } };`

Initialization of a 3d array

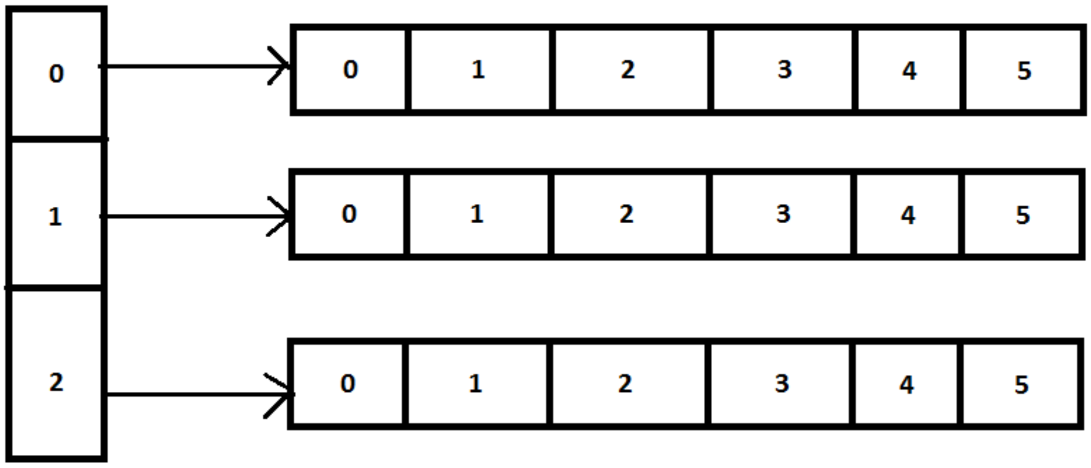
```
int test[2][3][4] = {
    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

```
// C Program to store and print 12 values entered by the user

#include <iostream.h>
int main()
{
  int test[2][3][2];

  cout<<"Enter 12 values: \n";

  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
      {
        cin>>test[i][j][k];
      }
    }
  }

  // Printing values with the proper index.

  cout<<"\nDisplaying values:\n";
  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
      {
        cout<<test[i][j][k];
      }
    }
  }
  return 0;
}
```

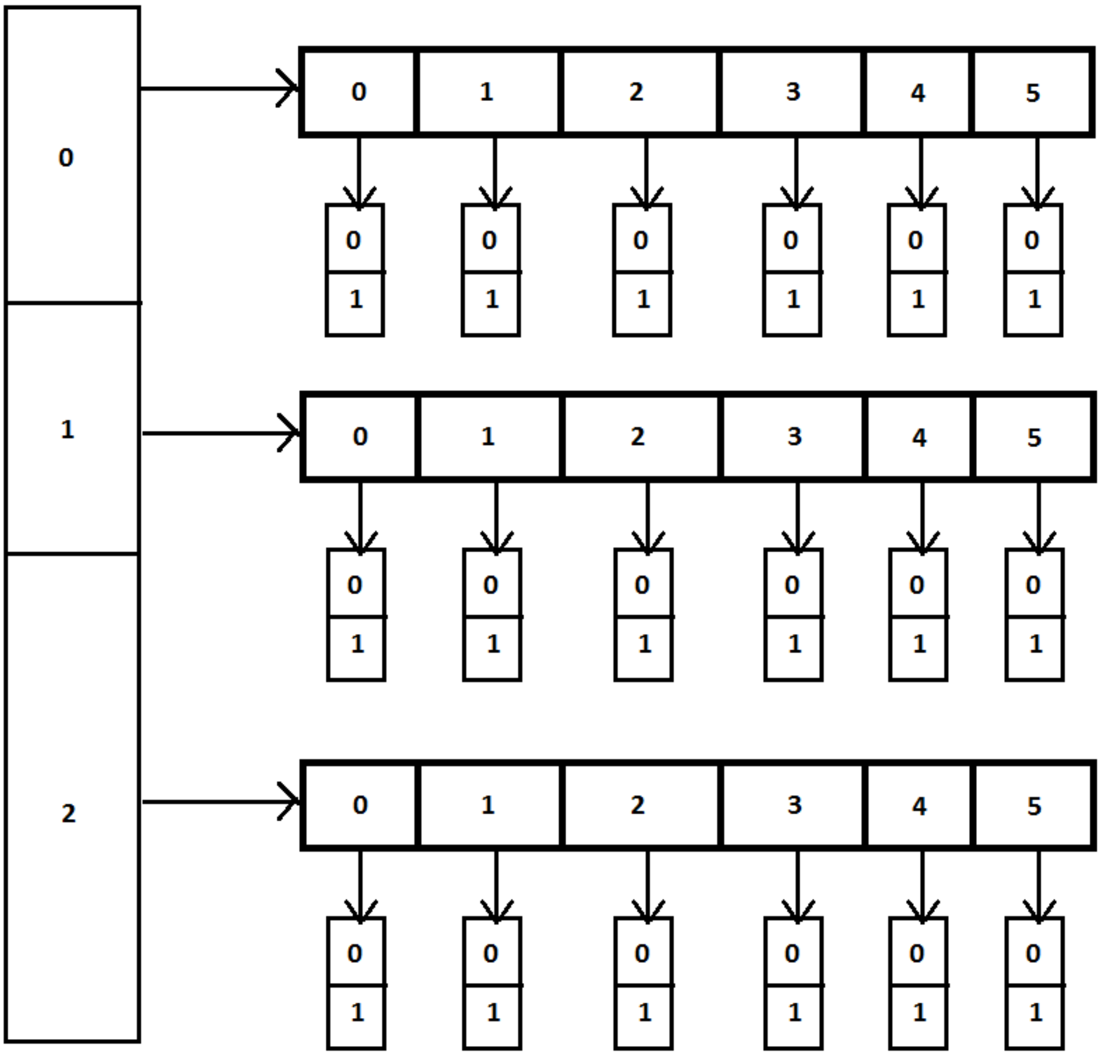
```
Enter 12 values:
1
2
3
4
5
6
7
8
9
10
11
12

Displaying Values:
test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
test[1][1][1] = 10
test[1][2][0] = 11
test[1][2][1] = 12
```

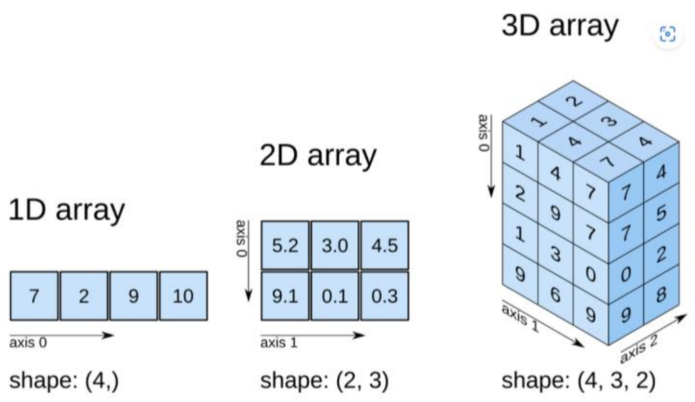
Visualizing n dimensional array

array[3][6]



array[3][6][2]





Mapping in data structure

Maps/Dictionaries

Maps (also known as Dictionaries) are data structures stores a collection of key-value pairs. Each key is unique and allows for quick access to values. A real life example of a map could be storing the grades for students in a class (student name is key, grade is value).

There are two main reasons why the map type can be valuable to C++ developers.

First, a map allows fast access to the value using the key. This property is useful when building any kind of index or reference.

Second, the map ensures that a key is unique across the entire data structure, which is an excellent technique for avoiding duplication of data. By virtue of these two advantages, a map is a common choice for a trading application in which we need to store stock prices by ticker symbol. If we are creating a weather application, a map would be an effective way to save and look up the current temperature in a set of cities around the world. In an e-commerce store, we'll likely need a map to find products by identifiers or categories.

How do we use a map in C++?

The primary operations are
a. creating a new map,
b. adding elements to and reading elements from a map
c. iterating through every element in a map. Let's take a look at each of these actions.

Constructing a map

```
#include <map>
```

```
#include <string>

using namespace std;



int main() {

  map<int, string> sample_map;

  sample_map.insert(pair<int, string>(1, "one"));

  sample_map.insert(pair<int, string>(2, "two"));



  cout << sample_map[1] << " " << sample_map[2] << endl;

}
```

In this example, we create a map that uses integers as keys and strings as values. We use the pair construct to create a key-value pair on the fly and insert that into our map.
The second often-used option is to initialize the map to a list of values at declaration.

```
#include <iostream>

#include <map>

#include <string>

using namespace std;



int main() {

  map<int, string> sample_map { { 1, "one"}, { 2, "two" } };



  cout << sample_map[1] << " " << sample_map[2] << endl;

}
```

Accessing map elements

In order to access the elements of the map, you can use array-style square brackets syntax:

```
...
cout << sample_map[1] << endl;
...
```

Another option, available as of C++11, is the at method:

```
...
cout << sample_map.at(1) << endl;
...
```

Inserting elements

When inserting elements into a map, it's common to use either the square brackets syntax or the insert method:

```
...
sample_map.insert(pair<int, string>(4, "four");
sample.map[5] = "five";
...
```

Iterating through elements

we can do this by using an *iterator*—a pointer that facilitates sequential access to a map's elements.
An iterator is bound to the shape of the map, so when creating an iterator we need to specify which kind of map the iterator is for. Once we have an iterator, we can use it to access both keys and values in a map. Here's what the code would look like:

```
int main() {

  map<int, string> sample_map { { 1, "one"}, { 2, "two" } };

  sample_map[3] = "three";

  sample_map.insert({ 4, "four" });
  map<int, string>::iterator it;

  for (it = sample_map.begin(); it != sample_map.end(); it++) {

    cout << it->second << " ";

  }

  cout << endl;

}
```

Special matrix

Sparse Matrix and its representations

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

```
0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0
```

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

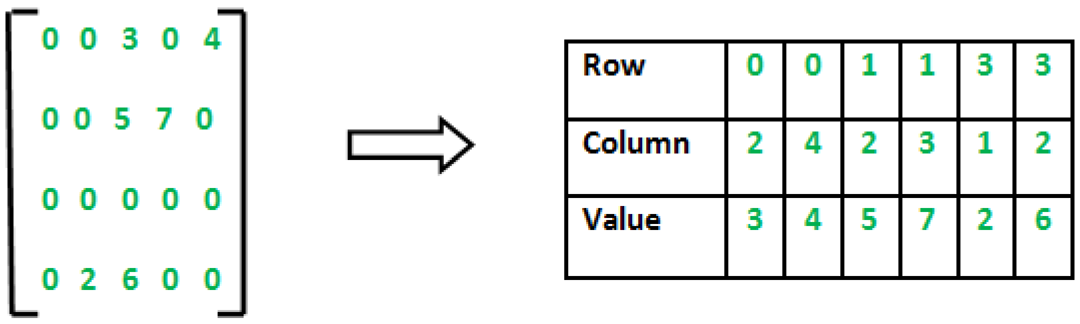
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays:

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

```
// C++ program for Sparse Matrix Representation
// using Array
#include <iostream>
using namespace std;

int main()
{
    // Assume 4x5 sparse matrix
    int sparseMatrix[4][5] =
    {
        {0 , 0 , 3 , 0 , 4 },
        {0 , 0 , 5 , 7 , 0 },
        {0 , 0 , 0 , 0 , 0 },
        {0 , 2 , 6 , 0 , 0 }
    };

    int size = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)
            if (sparseMatrix[i][j] != 0)
                size++;

    // number of columns in compactMatrix (size) must be
    // equal to number of non - zero elements in
    // sparseMatrix
    int compactMatrix[3][size];

    // Making of new matrix
    int k = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)
            if (sparseMatrix[i][j] != 0)
            {
                compactMatrix[0][k] = i;
                compactMatrix[1][k] = j;
                compactMatrix[2][k] = sparseMatrix[i][j];
                k++;
            }

    for (int i=0; i<3; i++)
    {
        for (int j=0; j<size; j++)
            cout <<" "<< compactMatrix[i][j];

        cout <<"\n";
    }
    return 0;
}
```

Output

```
 0 0 1 1 3 3

 2 4 2 3 1 2

 3 4 5 7 2 6
```

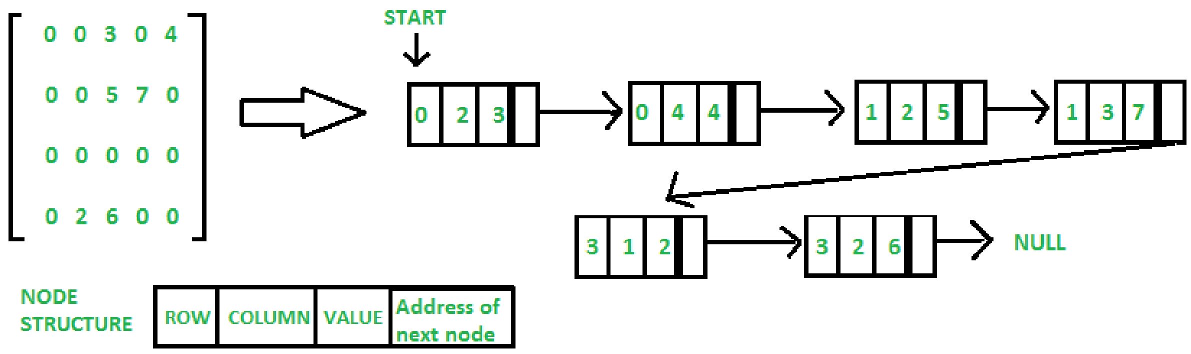
Time Complexity: O(NM), where N is the number of rows in the sparse matrix, and M is the number of columns in the sparse matrix.

Auxiliary Space: O(NM), where N is the number of rows in the sparse matrix, and M is the number of columns in the sparse matrix.

Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



```cpp
// C++ program for sparse matrix representation.
// Using Link list
#include<iostream>
using namespace std;

// Node class to represent link list
class Node
{
    public:
    int row;
    int col;
    int data;
    Node *next;
};

// Function to create new node
void create_new_node(Node **p, int row_index,
                     int col_index, int x)
{
    Node *temp = *p;
    Node *r;

    // If link list is empty then
    // create first node and assign value.
    if (temp == NULL)
    {
        temp = new Node();
        temp->row = row_index;
        temp->col = col_index;
        temp->data = x;
        temp->next = NULL;
        *p = temp;
    }

    // If link list is already created
    // then append newly created node
    else
    {
        while (temp->next != NULL)
            temp = temp->next;

        r = new Node();
        r->row = row_index;
        r->col = col_index;
        r->data = x;
        r->next = NULL;
        temp->next = r;
    }
}

// Function prints contents of linked list
// starting from start
void printList(Node *start)
{
    Node *ptr = start;
    cout << "row_position:";
    while (ptr != NULL)
    {
        cout << ptr->row << " ";
        ptr = ptr->next;
    }
    cout << endl;
    cout << "column_position:";

    ptr = start;
    while (ptr != NULL)
    {
        cout << ptr->col << " ";
        ptr = ptr->next;
    }
    cout << endl;
    cout << "Value:";
    ptr = start;

    while (ptr != NULL)
    {
        cout << ptr->data << " ";
        ptr = ptr->next;
    }
}

// Driver Code
int main()
{

    // 4x5 sparse matrix
    int sparseMatrix[4][5] = { { 0 , 0 , 3 , 0 , 4 },
                               { 0 , 0 , 5 , 7 , 0 },
                               { 0 , 0 , 0 , 0 , 0 },
                               { 0 , 2 , 6 , 0 , 0 } };

    // Creating head/first node of list as NULL
    Node *first = NULL;
    for(int i = 0; i < 4; i++)
    {
        for(int j = 0; j < 5; j++)
        {

            // Pass only those values which
            // are non - zero
            if (sparseMatrix[i][j] != 0)
                create_new_node(&first, i, j,
                                sparseMatrix[i][j]);
        }
    }
    printList(first);

    return 0;
}
```

Output

```
row_position:0 0 1 1 3 3

column_position:2 4 2 3 1 2

Value:3 4 5 7 2 6
```

Time Complexity: O(N*M), where N is the number of rows in the sparse matrix, and M is the number of columns in the sparse matrix.
Auxiliary Space: O(1)

Types of Sparse Matrices

There are different variations of sparse matrices, which depend on the nature of the sparsity of the matrices. Based on these properties, sparse matrices can be

- Regular sparse matrices
- Irregular sparse matrices / Non - regular sparse matrices

Regular sparse matrices

A regular sparse matri**x** is a square matrix with a well-defined sparsity pattern, i.e., non-zero elements occur in a well-defined pattern. The various types of regular sparse matrices are:

- Lower triangular regular sparse matrices

- Upper triangular regular sparse matrices
- Tri-diagonal regular sparse matrices

  Number of non-zero element in triangular matrix=

  1 + 2 + ………………… + i + ………………….. + (n-1) + n = n (n+1)/2

Lower triangular regular sparse matrices

A Lower regular sparse matrix is the one where all elements above the main diagonal are zero value. The following matrix is a lower triangular regular sparse matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 9 & -2 & 0 & 0 & 0 \\ -2 & 1 & 3 & 0 & 0 \\ 3 & 1 & -1 & 6 & 0 \\ 0 & 6 & 7 & 2 & 7 \end{bmatrix}$$

Lower Triangular Matrix

Storing Lower triangular regular sparse matrices

In a lower triangular regular sparse matrix, the non-zero elements are stored in a 1-dimensional array row by row.

Upper triangular regular sparse matrices

The Upper triangular regular sparse matrix is where all the elements below the main diagonal are zero value. Following matrix is the Upper triangular regular sparse matrix.

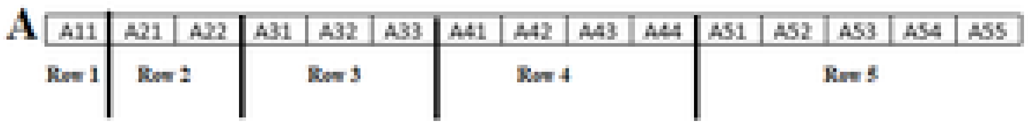
$$\begin{bmatrix} 2 & -1 & -5 & 9 & 7 \\ 0 & 4 & 2 & 4 & 6 \\ 0 & 0 & 3 & 1 & 5 \\ 0 & 0 & 0 & 6 & 4 \\ 0 & 0 & 0 & 0 & -2 \end{bmatrix}$$

Upper Triangular Matrix

Storage schemes of regular sparse matrices:

The efficient way to store the lower triangular matrix of size **N**:

- Count of non-zero elements = **1 + 2 + 3 + … + N = N * (N + 1) /2**.
- Count of **0**s = **N^2– (N * (N + 1) /2 = (N * (N – 1)/2**.



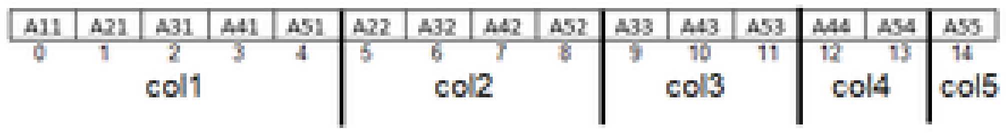
*Index of **Mat[i][j]** matrix in the array **A[]** = **[i*(i – 1)/2 + j – 1]***

```
// C++ program for the above approach
#include <bits/stdc++.h>
using namespace std;

// Dimensions of a matrix
static int N = 5;

// Structure of the efficient matrix
class Matrix {
  public:
  int* A;
  int size;
};

// Function to set the
// values in the Matrix
void Set(Matrix mat, int i, int j, int x)
{
  if (i >= j)
    mat.A[i * (i - 1) / 2 + j - 1] = x;
}

// Function to store the
// values in the Matrix
int Get(Matrix mat, int i, int j)
{
  if (i >= j)
    return mat.A[i * (i - 1) / 2 + j - 1];
  return 0;
}

// Function to display the
// elements of the matrix
void Display(Matrix mat)
{
  int i, j;
  // Traverse the matrix
  for (i = 1; i <= mat.size; i++) {
    for (j = 1; j <= mat.size; j++) {
      if (i >= j)
        cout << mat.A[i * (i - 1) / 2 + j - 1]
        << " ";
      else
        cout << 0 << " ";
    }
    cout << endl;
  }
}

// Function to generate an efficient matrix
Matrix createMat(vector<vector<int> >& Mat)
{
  // Declare efficient Matrix
  Matrix mat;
  // Initialize the Matrix
  mat.size = N;
  mat.A = new int[(mat.size * (mat.size + 1)) / 2];
  int i, j;
  // Set the values in matrix
  for (i = 1; i <= mat.size; i++)
    for (j = 1; j <= mat.size; j++)
      Set(mat, i, j, Mat[i - 1][j - 1]);
  // Return the matrix
  return mat;
}

// Driver Code
int main()
```

```
{
  vector<vector<int> > Mat = { { 1, 0, 0, 0, 0 },
                               { 1, 2, 0, 0, 0 },
                               { 1, 2, 3, 0, 0 },
                               { 1, 2, 3, 4, 0 },
                               { 1, 2, 3, 4, 5 } };

  // Stores the efficient matrix
  Matrix mat = createMat(Mat);

  // Print the Matrix
  Display(mat);
  return 0;
}
```

Time Complexity: *$O(N^2)$*
Auxiliary Space: *$O(N^2)$*

Efficient method to store a Lower Triangular Matrix using row-major mapping

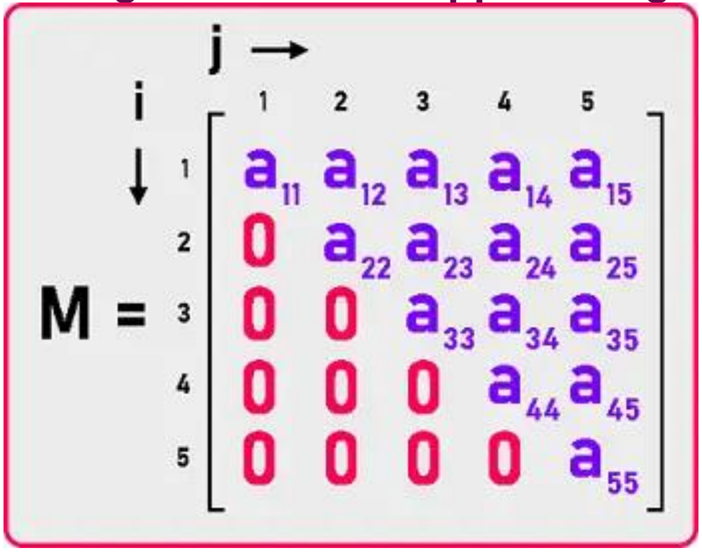


*index of **Mat[i][j]** matrix in the array A[] = [n*(j-1)-(((j-2)*(j-1))/2)+ (i-j))]*

```
// C++ program for the above approach

// Dimensions of the matrix
const int N = 5;

// Structure of a memory
// efficient matrix
struct Matrix {
    int* A;
    int size;
};

// Function to set the
// values in the Matrix
void Set(struct Matrix* m, int i,
         int j, int x)
{
    if (i >= j)
        m->A[((m->size)*(j-1)-(((j-2)
              *(j-1))/2)+(i-j))] = x;
```

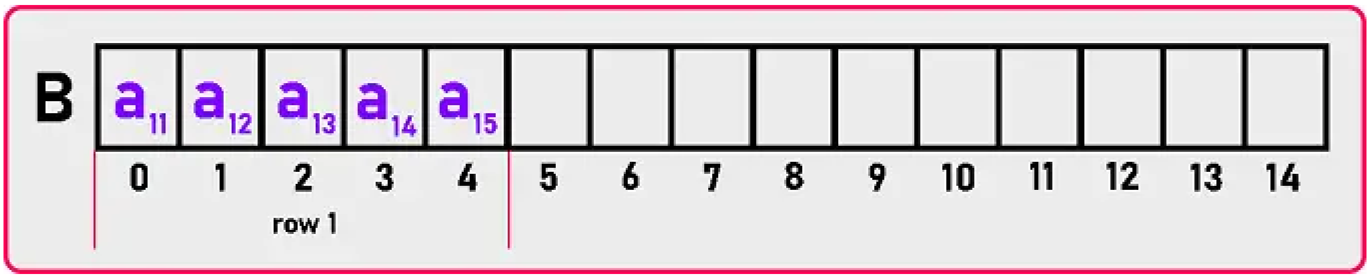
```
}

// Function to store the
// values in the Matrix
int Get(struct Matrix m, int i, int j)
{
    if (i >= j)
        return m.A[((m.size)*(j-1)-(((j-2)
                   *(j-1))/2)+(i-j))];
    else
        return 0;
}

// Function to display the
// elements of the matrix
void Display(struct Matrix m)
{
    // Traverse the matrix
    for (int i = 1; i <= m.size; i++)
    {
        for (int j = 1; j <= m.size; j++)
        {
            if (i >= j)
                cout<< m.A[((m.size)*(j-1)-(((j-2)
                       *(j-1))/2)+(i-j))] <<" ";
            else
                cout<<"0 ";
        }
        cout<<endl;
    }
}


// Function to generate an efficient matrix
struct Matrix createMat(int Mat[N][N])
{
    // Declare efficient Matrix
    struct Matrix mat;

    // Initialize the Matrix
    mat.size = N;
    mat.A = (int*)malloc(
        mat.size * (mat.size + 1) / 2
        * sizeof(int));

    // Set the values in matrix
    for (int i = 1; i <= mat.size; i++) {
        for (int j = 1; j <= mat.size; j++) {

            Set(&mat, i, j, Mat[i - 1][j - 1]);
        }
    }

    // Return the matrix
    return mat;
}


// Driver Code
int main()
{

    // Given Input
    int Mat[5][5] = { { 1, 0, 0, 0, 0 },
                      { 1, 2, 0, 0, 0 },
                      { 1, 2, 3, 0, 0 },
                      { 1, 2, 3, 4, 0 },
                      { 1, 2, 3, 4, 5 } };

    // Function call to create a memory
    // efficient matrix
    struct Matrix mat = createMat(Mat);

    // Function call to
      // print the Matrix
    Display(mat);

    return 0;
}
```

Storage schemes of upper triangular matrices:

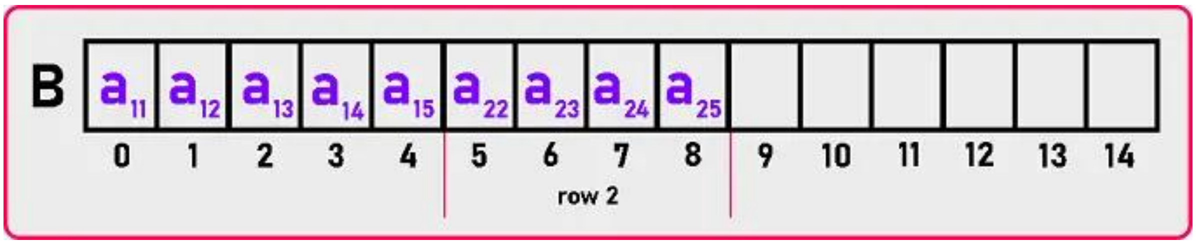


Using row major mapping:

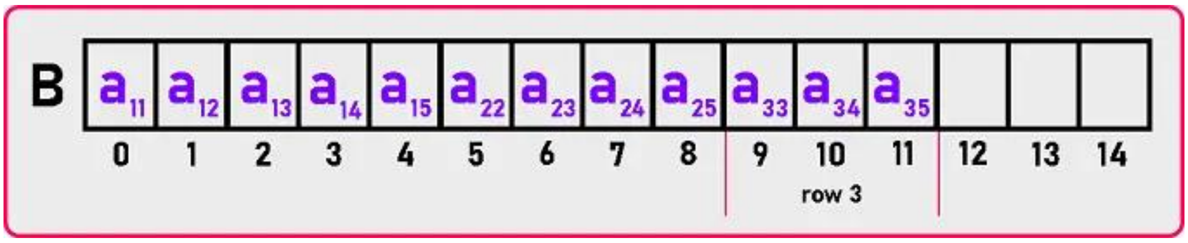
1st row:



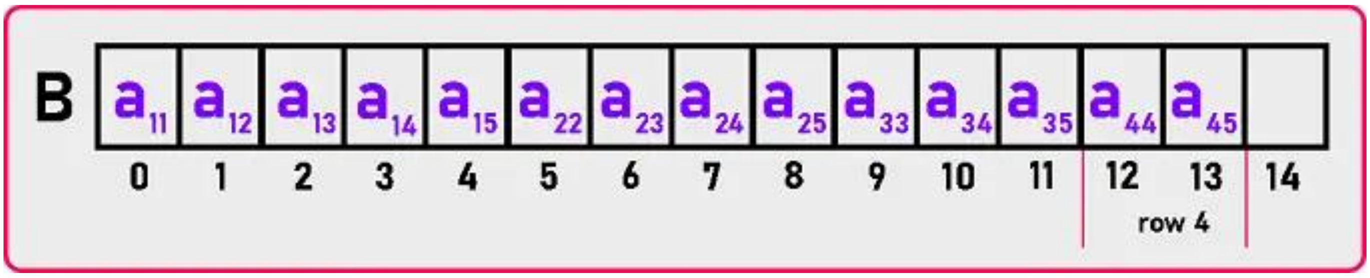
2nd row:


B
a_{11} a_{12} a_{13} a_{14} a_{15} a_{22} a_{23} a_{24} a_{25}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
row 2

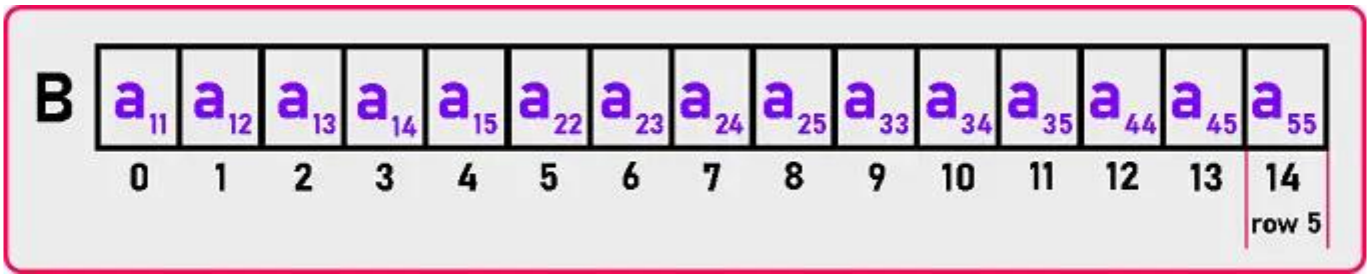
3rd row:

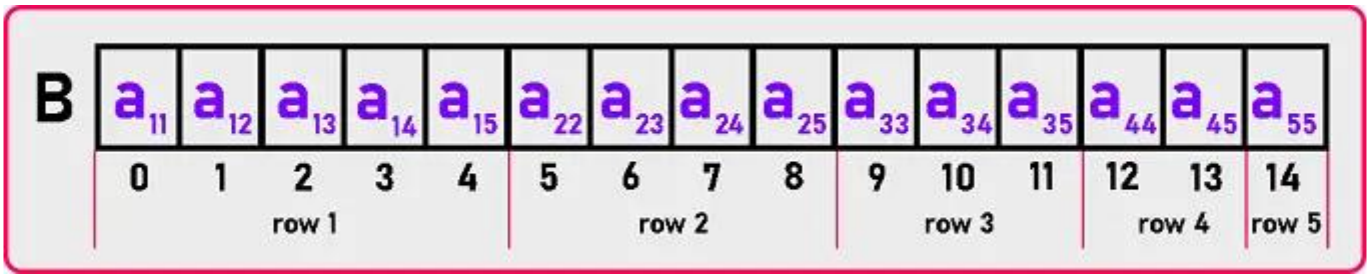

B
a_{11} a_{12} a_{13} a_{14} a_{15} a_{22} a_{23} a_{24} a_{25} a_{33} a_{34} a_{35}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
row 3

4th row:


B
a_{11} a_{12} a_{13} a_{14} a_{15} a_{22} a_{23} a_{24} a_{25} a_{33} a_{34} a_{35} a_{44} a_{45}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
row 4

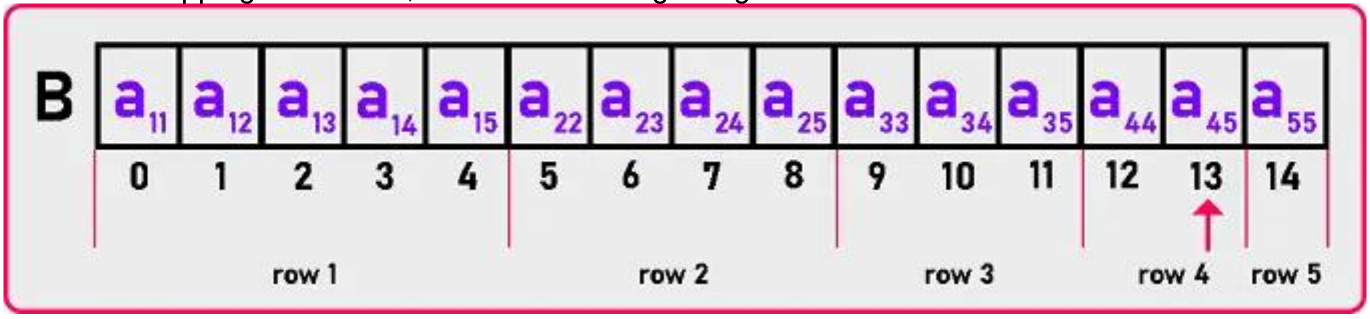
5th row:


B
a_{11} a_{12} a_{13} a_{14} a_{15} a_{22} a_{23} a_{24} a_{25} a_{33} a_{34} a_{35} a_{44} a_{45} a_{55}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
row 5


B
a_{11} a_{12} a_{13} a_{14} a_{15} a_{22} a_{23} a_{24} a_{25} a_{33} a_{34} a_{35} a_{44} a_{45} a_{55}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
row 1
row 2
row 3
row 4
row 5

We want to find the indices of **M [4][5]**. **M [4][5]** means the element which is present in the 4th row and 5th column. As we can see in the array, **M [4][5]** is present on the 13th index. The element is present in the 4th row. So, we need to skip **1st**, **2nd**, and **3rd** row to reach on **4th** row.

1st row having **5** elements: **Index (M [4][5]) = 5 +**
2nd row having 4 elements: **Index (M [4][5]) = 5 + 4**
3rd row having 3 elements: **Index (M [4][5]) = 5 + 4 + 3**
Now after skipping the 3 rows, we are on the beginning of 4th row:


B
a_{11} a_{12} a_{13} a_{14} a_{15} a_{22} a_{23} a_{24} a_{25} a_{33} a_{34} a_{35} a_{44} a_{45} a_{55}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
row 1
row 2
row 3
row 4
row 5

Now how much we should move ahead reach **a_{45}**? Just one element. So,
Index (M [4][5]) = [5 + 4 + 3] + 1

Index (M [i][j]) = [(i-1) * n – (i-2)*(i-1) / 2] + (j – i)

Using column major mapping:

```
M[i][j]={j(j-1)/2}+i-1
```

Tri-diagonal matrix

Tri-diagonal matrix is also another type of a sparse matrix, where elements with a non-zero value appear only on the diagonal or immediately below or above the diagonal.
In a tridiagonal matrix, Arr i,j=0, where |i – j| > 1.
For matrix being a tridiagonal matrix element should present
(a) On the main diagonal means all non-zero elements at i=j and at all rest place zero. In this case, the total number of non-zero elements is n.
(b) at above the main diagonal means all non-zero elements at i=j–1. In this case, the total number of non-zero elements is n-1.

(c) at below the main diagonal means all non-zero elements for i=j+1. In this case, the total number of non-zero elements is n-1.

$$M = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & a_{11} & a_{12} & 0 & 0 & 0 \\ 2 & a_{21} & a_{22} & a_{23} & 0 & 0 \\ 3 & 0 & a_{32} & a_{33} & a_{34} & 0 \\ 4 & 0 & 0 & a_{43} & a_{44} & a_{45} \\ 5 & 0 & 0 & 0 & a_{54} & a_{55} \end{array}$$

1. **Main Diagonal:** row number is equal to column number **(i = j).**
2. **Lower Diagonal:** row number – column number = 1 **(i – j = 1).**
3. **Upper Diagonal:** row number – column number = -1 **(i – j = -1).**

How many non-zero elements are there?
= n + n-1 + n-1.
= 3n – 2 non-zero elements are there.

We can do the mapping between a two-dimensional matrix and a one-dimensional array the following ways:

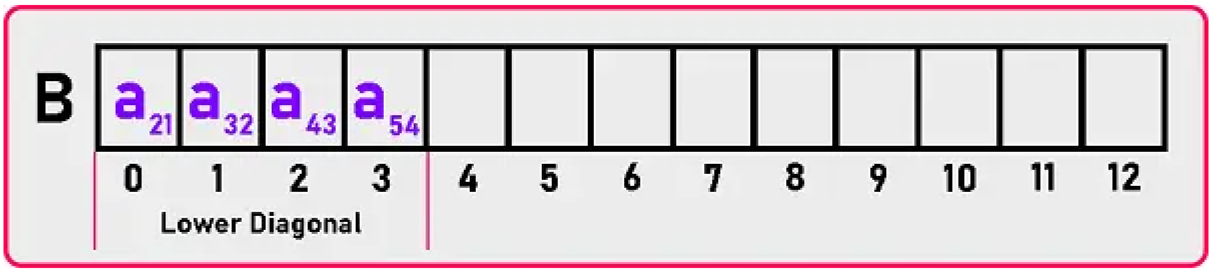
(a) Row-wise mapping— Here the contents of array Arr[] will be {1, 1, 5, 2, 8, 8, 3, 2, 4, 1, 5, 7, 9}

(b) Column-wise mapping— Here the contents of array Arr[] will be {1, 5, 1, 2, 8, 8, 3, 4, 2, 1, 7, 5, 9}

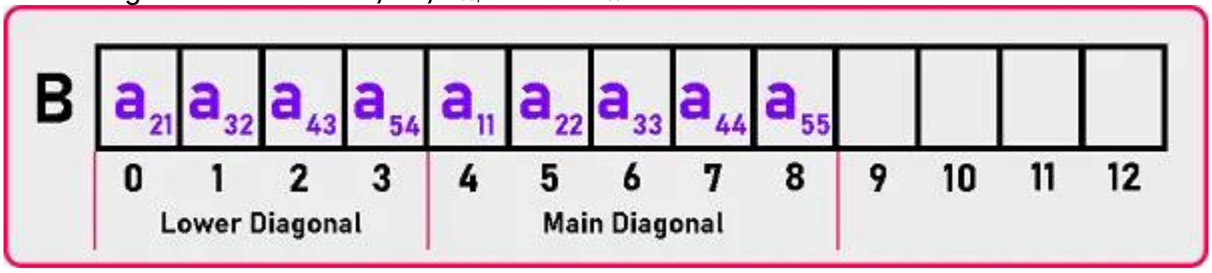
(c) Diagonal-wise mapping— Here the contents of array Arr[] will be {1, 8, 2, 5, 1, 2, 3, 1, 9, 5, 8, 4, 7}

Store The Elements Diagonal By Diagonal.

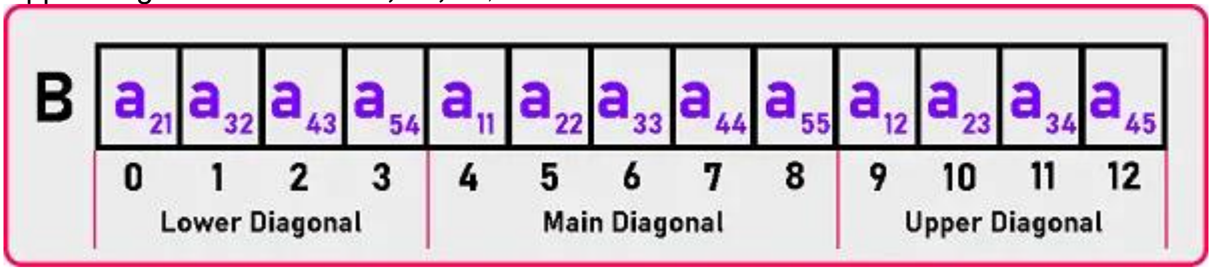
We can pick up either the upper diagonal and main then lower or first lower than main and then upper. So let us store first lower diagonal elements. We will store lower diagonal elements **a_{21}, a_{32}, a_{43}, and a_{54}** as:



Main Diagonal elements **a_{11}, a_{22}, a_{33}, a_{44} and a_{55}** as:


B
a_{21} a_{32} a_{43} a_{54} a_{11} a_{22} a_{33} a_{44} a_{55}
0 1 2 3 4 5 6 7 8 9 10 11 12
Lower Diagonal
Main Diagonal

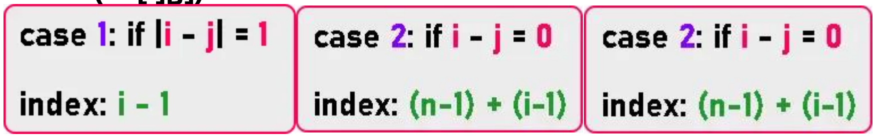
Upper diagonal elements **a_{12}, a_{23}, a_{34}, and a_{45}** as:


B
a_{21} a_{32} a_{43} a_{54} a_{11} a_{22} a_{33} a_{44} a_{55} a_{12} a_{23} a_{34} a_{45}
0 1 2 3 4 5 6 7 8 9 10 11 12
Lower Diagonal
Main Diagonal
Upper Diagonal

Now how to map these?

There must be some formula, we cannot come up with a single formula for all the diagonals. We have to handle them separately. Simply we go diagonal by diagonal.

Index (M [i][j]):

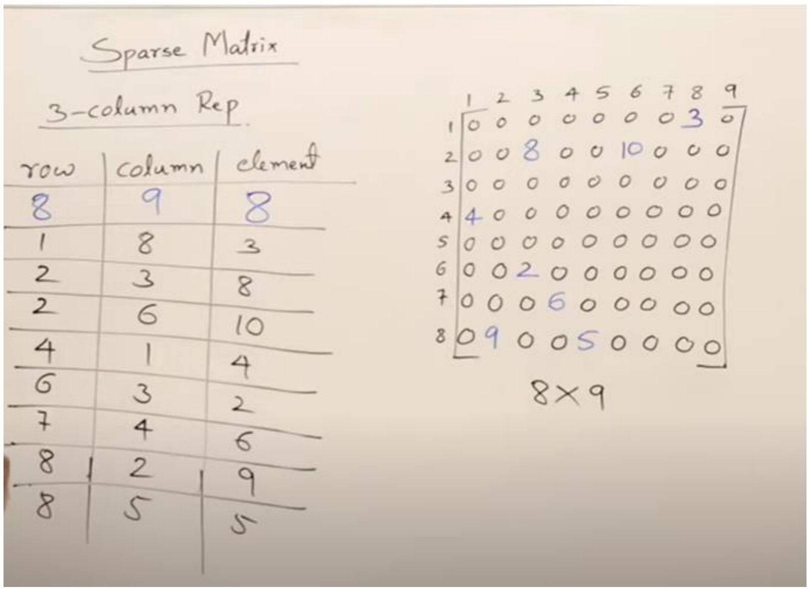


case 1: if |i - j| = 1
index: i - 1

case 2: if i - j = 0
index: (n-1) + (i-1)

case 2: if i - j = 0
index: (n-1) + (i-1)

```
Assignment:
1. wap for row and column major mapping of lower and upper
triangular matrix

Irrregular sparse matrix
```


Sparse Matrix

3-column Rep

row	column	element
8	9	8
1	8	3
2	3	8
2	6	10
4	1	4
6	3	2
7	4	6
8	2	9
8	5	5

$$\begin{array}{c|ccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 2 & 0 & 0 & 8 & 0 & 0 & 10 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 9 & 0 & 0 & 5 & 0 & 0 & 0 & 0 \end{array}$$

8×9

Sparse matrix triplet or column representation

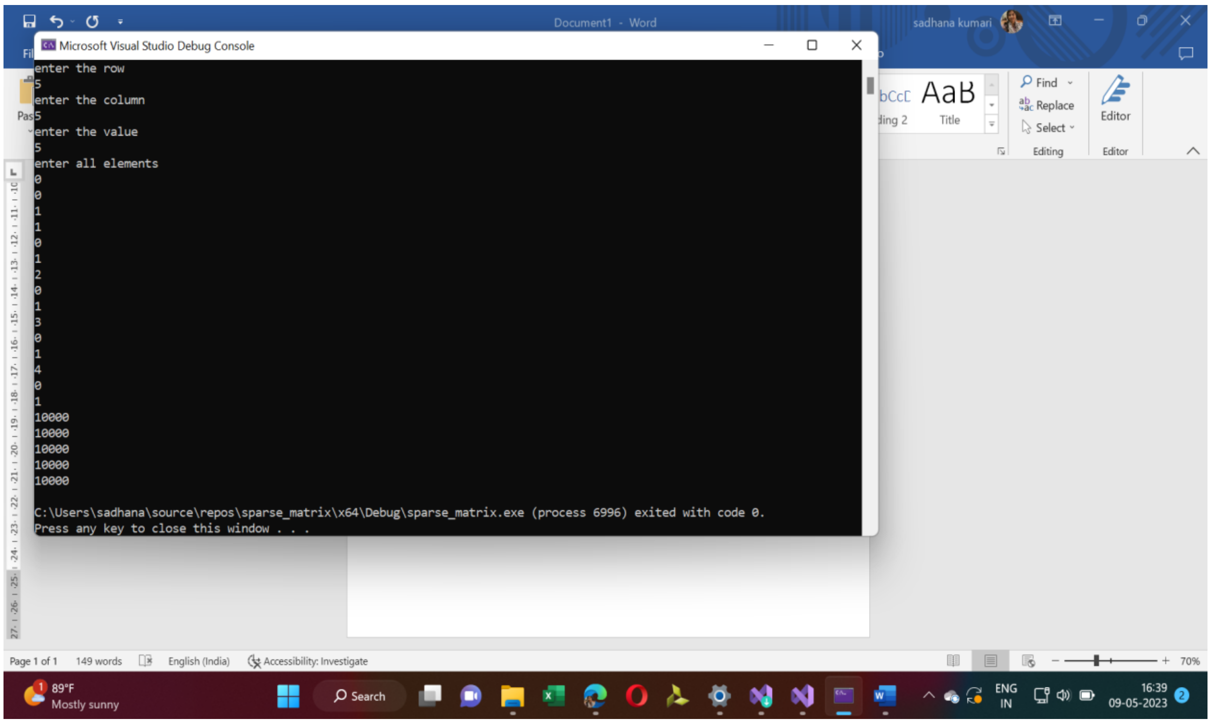
First row first column represents number of rows, First row second column represents number of column, First row third column represents number of non-zero element

Rest other rows show triplet (row index, column index and value at that particular index)

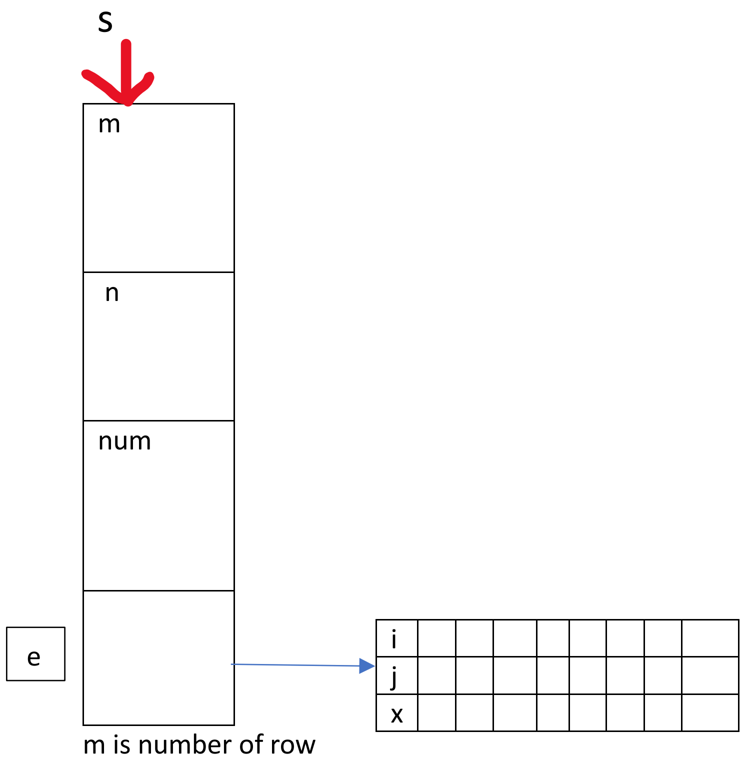
Create a sparse matrix dynamically

```
#include<iostream>
using namespace std;
struct element
{
int i;//row index
int j;//column index
int x;// value of non-zero elemet
};
struct sparse {
	int m;
	int n;
	int num;
	struct element* e;
};
void create(struct sparse *s)
{
	cout << "enter the row" << endl;
	cin >> s->m;
	cout << "enter the column" << endl;
	cin >> s->n;
	cout << "enter the value " << endl;

	cin >> s->num;
	s->e = new element[s->num];
	cout << "enter all elements" << endl;
	for (int i = 0; i < s->num; i++)
	{
		cin >> s->e[i].i >> s->e[i].j >> s->e[i].x;

	}
}
void display (struct sparse s)
```

```
{
        int i, j, k = 0;
        for (i=0;i<s.m;i++)
        {
                for (j = 0; j < s.n; j++)
                {
                        if (i == s.e[k].i && j == s.e[k].j)
                                cout << s.e[k++].x;
                        else
                                cout << "0";
                }
                cout << "\n";

        }
        }

int main()
{
        struct sparse s;
        create(&s);
        display(s);
        return 0;

}


//Output window
```



```
enter the row
5
enter the column
5
enter the value
5
enter all elements
0
0
1
1
0
1
2
0
1
3
0
1
4
0
1
10000
10000
10000
10000
10000

C:\Users\sadhana\source\repos\sparse_matrix\x64\Debug\sparse_matrix.exe (process 6996) exited with code 0.
Press any key to close this window . . .
```



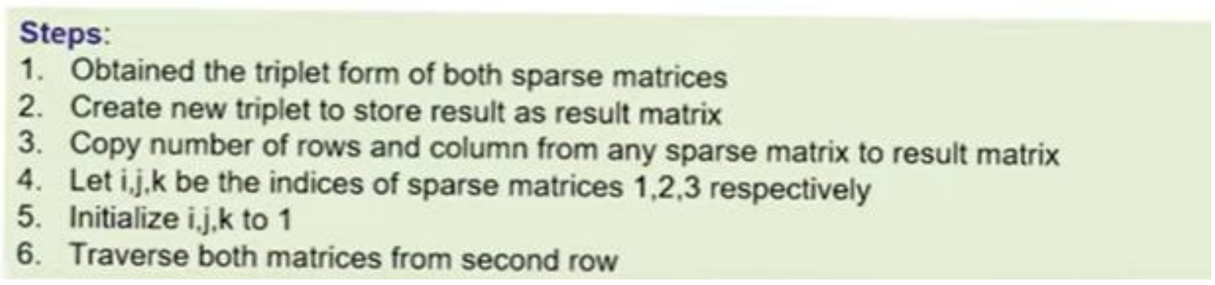
m is number of row

n is number of column

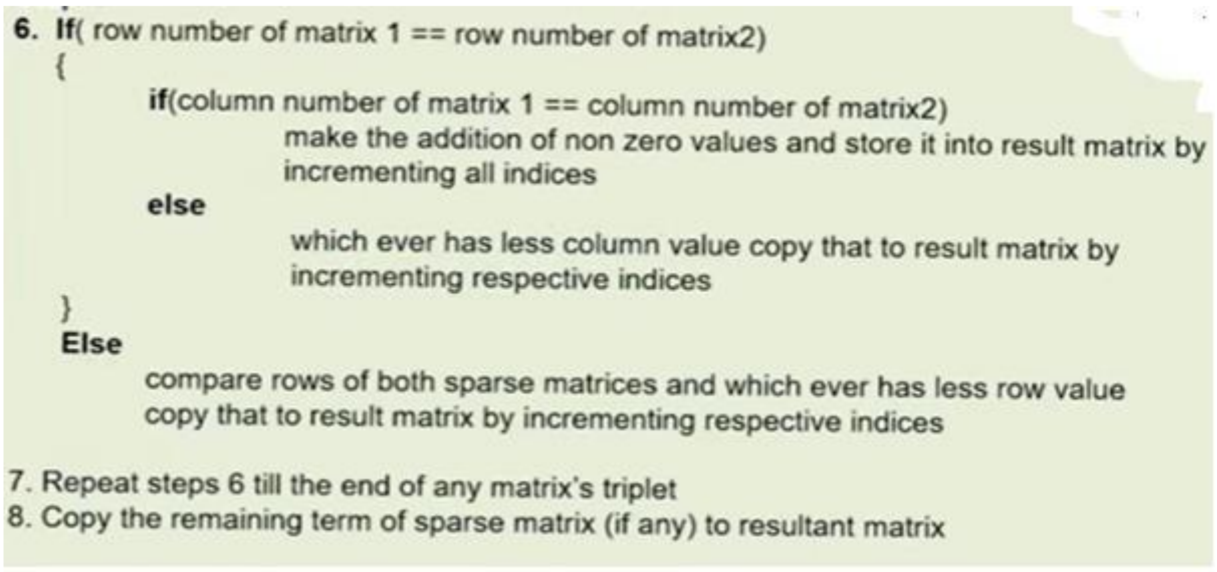
num is number of non-zero element

Addition of sparse matrix:

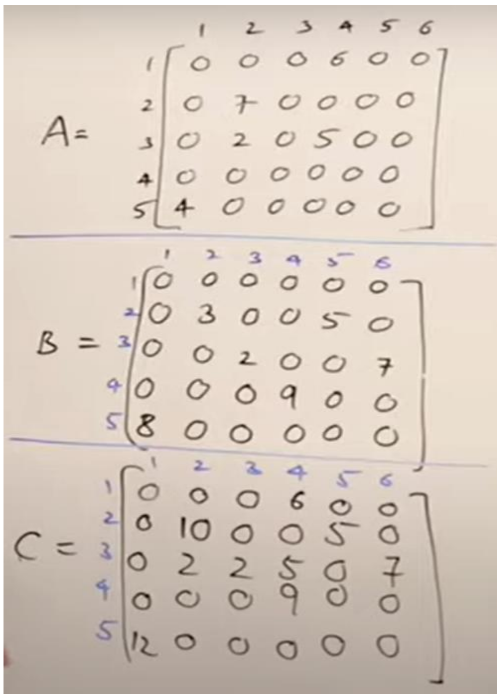
Algorithm



Steps:
1. Obtained the triplet form of both sparse matrices
2. Create new triplet to store result as result matrix
3. Copy number of rows and column from any sparse matrix to result matrix
4. Let i,j,k be the indices of sparse matrices 1,2,3 respectively
5. Initialize i,j,k to 1
6. Traverse both matrices from second row



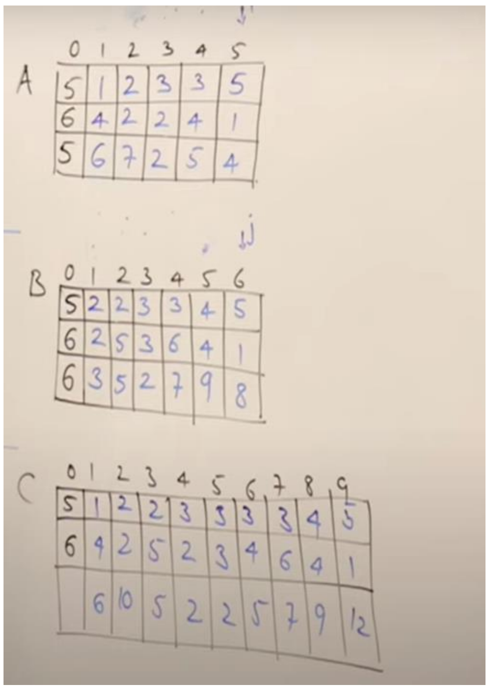
```
6. If( row number of matrix 1 == row number of matrix2)
   {
        if(column number of matrix 1 == column number of matrix2)
                 make the addition of non zero values and store it into result matrix by
                 incrementing all indices
        else
                 which ever has less column value copy that to result matrix by
                 incrementing respective indices
   }
   Else
        compare rows of both sparse matrices and which ever has less row value
        copy that to result matrix by incrementing respective indices

7. Repeat steps 6 till the end of any matrix's triplet
8. Copy the remaining term of sparse matrix (if any) to resultant matrix
```



$$A = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 0 & 0 & 6 & 0 & 0 \\ 2 & 0 & 7 & 0 & 0 & 0 & 0 \\ 3 & 0 & 2 & 0 & 5 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 4 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$B = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 3 & 0 & 0 & 5 & 0 \\ 3 & 0 & 0 & 2 & 0 & 0 & 7 \\ 4 & 0 & 0 & 0 & 9 & 0 & 0 \\ 5 & 8 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$C = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 0 & 0 & 6 & 0 & 0 \\ 2 & 0 & 10 & 0 & 0 & 5 & 0 \\ 3 & 0 & 2 & 2 & 5 & 0 & 7 \\ 4 & 0 & 0 & 0 & 9 & 0 & 0 \\ 5 & 12 & 0 & 0 & 0 & 0 & 0 \end{array}$$



A

0	1	2	3	4	5
5	1	2	3	3	5
6	4	2	2	4	1
5	6	7	2	5	4

B

0	1	2	3	4	5	6
5	2	2	3	3	4	5
6	2	5	3	6	4	1
6	3	5	2	7	9	8

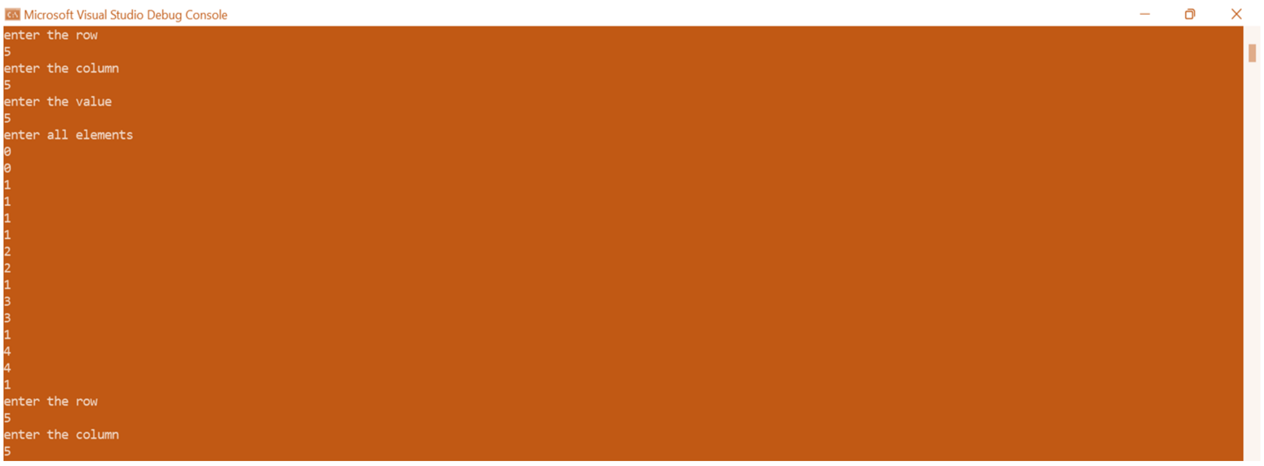
C

0	1	2	3	4	5	6	7	8	9
5	1	2	2	3	3	3	3	4	5
6	4	2	5	2	3	4	6	4	1
	6	10	5	2	2	5	7	9	12

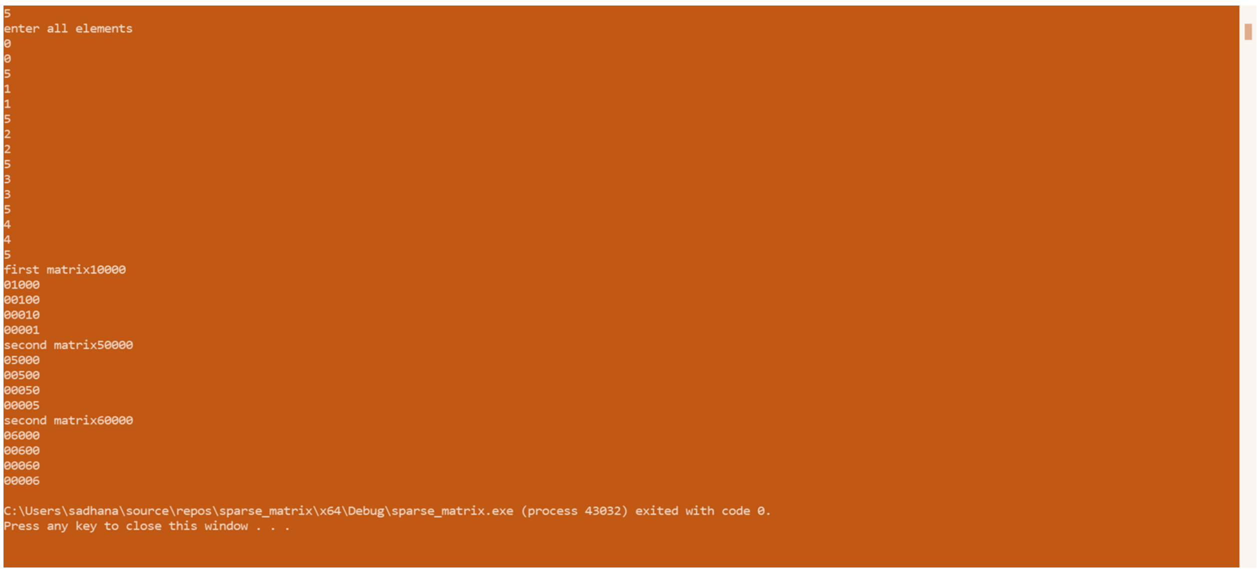
c=A+B

```
#include<iostream>
using namespace std;
struct element
{
	int i;
int j;
int x;
};
struct sparse {
	int m;
	int n;
	int num;
	struct element* e;
};
void create(struct sparse *s)
{
	cout << "enter the row" << endl;
	cin >> s->m;
```

```
        cout << "enter the column" << endl;
        cin >> s->n;
        cout << "enter the value " << endl;

        cin >> s->num;
        s->e = new element[s->num];
        cout << "enter all elements" << endl;
        for (int i = 0; i < s->num; i++)
        {
                cin >> s->e[i].i >> s->e[i].j >> s->e[i].x;

        }
}
void display (struct sparse s)
{
        int i, j, k = 0;
        for (i=0;i<s.m;i++)
        {
                for (j = 0; j < s.n; j++)
                {
                        if (i == s.e[k].i && j == s.e[k].j)
                                cout << s.e[k++].x;
                        else
                                cout << "0";
                }
                cout << "\n";

        }

}
struct sparse *add(struct sparse* s1, struct sparse* s2) {
        struct sparse* sum;
        int i, j, k;
        i = j = k = 0;

        sum = new struct sparse;
        sum->e = new element[s1->num + s2->num];
        while (i < s1->num && j < s2->num)
        {
                if (s1->e[i].i < s2->e[j].i)
                        sum->e[k++] = s1->e[i++];
                else if(s1->e[i].i > s2->e[j].i)
                        sum->e[k++] = s1->e[j++];
                else {
                        if (s1->e[i].j < s2->e[j].j)
                                sum->e[k++] = s1->e[i++];
                        else if (s1->e[i].j > s2->e[j].j)
                                sum->e[k++] = s1->e[j++];
                        else
                        {
                                sum->e[k] = s1->e[i];
                                sum->e[k++].x= s1->e[i++].x + s2->e[j++].x;

                        }
                }
        }
        for (; i < s1->num; i++)
                sum->e[k++] = s1->e[i];
        for (; j < s2->num; j++)
                sum->e[k++] = s2->e[i];
        sum->m = s1->m;
        sum->n = s1->n;
        sum->num = k;
        return sum;
}

int main()
{
        struct sparse s1,s2,*s3;
        create(&s1);
        create(&s2);
        s3 = add(&s1, &s2);
        cout << "first matrix";
        display(s1);
        cout << "second matrix";
        display(s2);

        cout << "second matrix";
        display(*s3);

        return 0;

}
```



```
enter the row
5
enter the column
5
enter the value
5
enter all elements
0
0
1
1
1
1
2
2
1
3
3
1
4
4
1
enter the row
5
enter the column
5
```



```
5
enter all elements
0
0
5
1
1
5
2
2
5
3
3
5
4
4
5
first matrix10000
01000
00100
00010
00001
second matrix50000
05000
00500
00050
00005
second matrix60000
06000
00600
00060
00006

C:\Users\sadhana\source\repos\sparse_matrix\x64\Debug\sparse_matrix.exe (process 43032) exited with code 0.
Press any key to close this window . . .
```