CC-213L

Data Structures and Algorithms

Laboratory 03

Abstract Data Types (ADT's)

Version: 1.0.0

Release Date: 04-10-2024

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- C++ Pointers
- Dynamic Memory Allocation
- Understanding the Concept of an ADT
- Arrays
- 1-D ,2-D and N-D Arrays
- Memory Representation of Arrays
- Sparse Matrices

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how s/he is doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course / Lab Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Teacher Assistant	Muhammad Tahir Mustafvi	Bcsf20m018@pucit.edu.pk

Background and Overview:

Pointers:

In C++, pointers are **variables** that hold **memory addresses**. They're declared with the data type they point to, initialized with **the address** of other variables, and can be used to access or modify the values at those addresses using the dereference operator. Null pointers indicate uninitialized or invalid addresses. Pointers are commonly used for dynamic memory allocation and can navigate arrays and data structures through pointer arithmetic. They also enable the use of function pointers for dynamic function invocation. However, proper handling is crucial to avoid null pointer errors and memory leaks. In modern C++, **smart pointers are recommended for safer memory management**, particularly in dynamic memory allocation scenarios.

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = 42;
    int* ptr = &x; // 'ptr' now points to the memory address of 'x'

    int y = *ptr; // 'y' is now 42, the value at the address pointed to by 'ptr'

    int* null_ptr = nullptr;

    //OR use NULL

    null_ptr = NULL;

}
```

Fig.01 (Pointers)

Dynamic Memory Allocation:

Dynamic memory allocation in programming refers to the process of requesting and managing memory during runtime rather than at compile time. It allows programs to allocate memory as needed, which can be particularly useful when you don't know the size of data structures or objects in advance. In languages like C++ and C, dynamic memory allocation is typically achieved using functions like **new** (in C++) and `malloc` (in C) to allocate memory on the heap and **delete** (in C++) and `free` (in C) to deallocate it when it's no longer needed. Here's an example in C++:

```cpp
#include <iostream>
using namespace std;
int main()
{
    int* ptr = new int;

    *ptr = 10;

    cout << *ptr << endl;

    delete ptr;

    int* arr = new int[10];

    delete[] arr;

    system("color 70");
    return 0;
}
```

Fig.02 (Dynamic Memory)

Explanation:

In our example code, we defined a pointer to integer and then dynamically allocated it a memory of integer. After dereferencing it we have assigned **a**n integer value to that memory. We have displayed

that value. Then we have deallocated it. Same we have done by allocating an array of integers dynamically and then assigned base address of array to pointer arr. After that we have deallocated it.

Fig.03 (Dynamic Memory)

Abstract Data Types:

An ADT is a language independent concept. As its specification can be written in any language.

Two necessary things for an ADT

a) Data

b) Operations On data

ADT on bigger level becomes class. The data becomes **data members / member variables** and

Operations become **member functions** or methods. ADT in any language is class So, one ADT can result into multiple classes based on the implementation i.e. one might use **any data structure.**

In C++, an Abstract Data Type (ADT) is a high-level description of a data structure and the operations that can be performed on it, without specifying the underlying implementation details. It abstracts away the internal workings of a data structure, allowing you to use it without needing to know how it is implemented. ADTs are a fundamental concept in computer science and are often used to create reusable, modular code.

Here are a few key points about ADTs in C++:

1. Encapsulation: ADTs encapsulate the data and operations on that data into a single unit. The user of the ADT interacts with it through a well-defined interface (public functions), and the internal details are hidden.

2. Data Abstraction: ADTs provide a way to represent complex data structures and their associated operations in a simplified and abstract manner. For example, a stack ADT may have operations like "push," "pop," and "top," abstracting the idea of a stack of elements.

3. Implementation Independence: Users of an ADT do not need to know how it is implemented internally. This allows for flexibility in changing the implementation details without affecting code that uses the ADT.

4. Separation of Concerns: ADTs promote a separation of concerns between the data structure and the algorithms that operate on it. This separation makes the code more modular and maintainable.

Key Features of an ADT

1. Conceptual Data Structures:

ADTs define data structures at a conceptual level without specifying how they are implemented. They describe what operations can be performed on the data and what the expected behavior of those operations is.

2. Separation of Interface and Implementation:

ADTs separate the interface (the set of operations that can be performed on the data) from the implementation (how those operations are carried out). This separation allows for flexibility in changing the underlying implementation without affecting the code that uses the ADT. For example, you can switch between array-based and linked-list-based implementations of a stack ADT without changing the way you use the stack

3. Reusability and Modularity:

ADTs promote code reusability and modularity. Once you've defined an ADT, you can use it in different parts of your code or in different projects without needing to reimplement the same data structure and operations. This saves development time and reduces the chance of introducing bugs.

4. Abstract Behavior:

ADTs define the expected behavior of operations rather than the specific algorithm used to achieve that behavior. For instance, the push operation for a stack ADT is expected to add an element to the top of the stack, but it doesn't dictate whether this is done through an array, linked list, or any other means.

5. Real-world Analogy:

ADTs often have real-world analogies. For example, a stack ADT is analogous to a physical stack of items where you can push items onto the top and pop items from the top. This analogy helps users understand the behavior of the ADT even if they don't know the underlying implementation.

6. Standardization:

ADTs can become standardized, and there are often libraries in C++ that provide commonly used ADTs such as lists, queues, and maps. These standardized ADTs are part of the Standard Template Library (STL) in C++.

7. Custom ADTs:

You can create custom ADTs tailored to the specific needs of your application. For example, if your application requires a specialized data structure, you can define an ADT for it, encapsulating the necessary data and operations.

Activities:

Pre-Lab Activities:

List ADT: The List ADT defines a set of operations that can be performed on a list, but it does not specify how these operations are implemented. Instead, it serves as a blueprint or contract for how a list should behave.

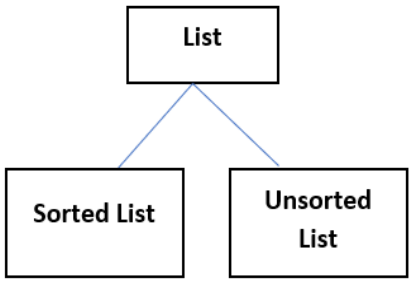


Fig.03 (List ADT)

Task 01: UnsortedList

Implement the **Unsorted List ADT** as given the prototype of the ADT. Kindly make sure the implement it as **Class Template.** The prototypes have been given in specific type **(int).**

Note: There shouldn`t any memory leak or dangling pointer in your implementation of ADT. All the corner cases should be handled properly. Try to throw Exceptions where necessary.

```
class UnsortedList
{
	int* arr;
	int maxSize, currSize;
public:
	UnsortedList();
	~UnsortedList();
	UnsortedList(const UnsortedList&);
	const UnsortedList& operator = (const UnsortedList&);
	bool insert(int val);
	bool isFull() const;
	bool isEmpty() const;
	void display() const;
	bool remove(int index);
};
```

The program should be multi files. You should make sure to write each and every method with efficient approach. There should not be extra space or time in your implementation of method being consumed. The credit of full marks only be given for efficient work and handling all types of problem instances for that algorithm.

Task 02: Polynomial ADT

You are required to implement the polynomial ADT. An array data structure is to be used for storing the terms in such a way that the coefficient of the term with exponent k will be stored in array at index

k.

For example, to store the polynomial 15x^2 + 5x. We'll have to store value 15 (which is co-efficient of 15x^2) at the index 2 and 5 (coefficient of 5x) at the index 1.

To figure out whether the term with exponent n exists in the polynomial or not, one has to simply check whether the value at index n is zero or not. If it's zero, it means that coefficient of term with degree n is zero which implies that the term does not exist.

Keeping in mind the above information, your job is to provide the implementation of the following methods.

Polynomial (int max_degree) ;

"max_degree" is not the actual degree of the polynomial. It is just the highest exponent a term can possibly have in the polynomial. It helps us to figure out the size of array to be created for storing our polynomial.

Polynomial (const Polynomial& obj) ; int get _degree() const;

This method will not return the max_degree argument that we received in the constructor, but the actual degree of the polynomial, which is the value of the highest exponent among all the terms.

Polynomial& operator = (const Polynomial& other);

void add_term(int coefficient , int degree) ;

Throw an exception if the degree of the term to be added to the polynomial is greater than size of the array we created in constructor. If the term already exists in polynomial, simply overwrite the coefficient.

void remove_term (int degree);

void set_coeffecient(int coefficient , int degree) ;

friend ostream& operator << (ostream& os, const Polynomial& poly);

Should print the polynomial on console as we usually write polynomials in mathematics such as 5x^4 + 3x^2 + 5.

double operator () (double x) const ;

Evaluates the value of polynomial for given value of x and returns the answer.

Polynomial operator + (const Polynomial& other) ;

Performs addition operation on polynomials and returns the new resultant polynomial. Write this function smartly to get the full credit :)

Polynomial operator * (const Polynomial& other) ;

~Polynomial ();

In-Lab Activities:

Arrays:

To store a group of data together in a **sequential manner in computer's memory**, arrays can be one of the possible data structures. Arrays enable us to organize more than one element in consecutive memory locations; hence, it is also termed as structured or composite data type. The only restriction is that all the elements we wish to store must be of the same data type. It can be thought of as a box with multiple compartments, where each compartment can hold one data item. Arrays support direct access to any of those data items just by specifying the name of the array and its index as the item's position **(sequence number as subscript).** Arrays are the most general and easy to use of all the data structures. An array as a data structure is defined as a set of **pairs (index, value)** such that with each index, a value is associated.

index—indicates the location of an element in an array

value—indicates the actual value of that data element

Here is an Example of C++ Array. How the Elements are accessed by the Indices.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      const int N{ 5 }; // the size of array should be constant
6
7      int arr[N] = {1,2,3,4,5}; // defining and initializing the Array
8
9      for (int i = 0; i < N; i++)
10     {
11         cout << arr[i] << endl;
12     }
13     return 0;
14 }
```

Fig.04 (Arrays)

Explanation:

In line number 7 a static array at stack has been defined and initialized with integer values. In line number 9 a for loop have been used to display contents of Array. Attention Here the subscript/index ([]) operator have been used. But you can also use pointer notation to access the elements of Array.

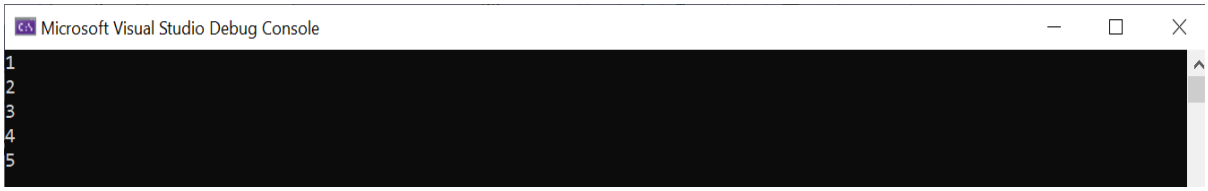


Fig. 05 (output)

Memory Representation of 1-D array:

The address of the i^{th} element is calculated by the following formula:

(Base address) + (Offset of i^{th} element from base address)

Here, base address is the address of the first element where array storage starts.

Address of A[i] = Base + i ×Size of element

All the elements of the array must be properly initialized before referring in any expression. It is important to note that arrays and their sizes are mostly defined statically, so it is not possible to change the size at the time of execution.

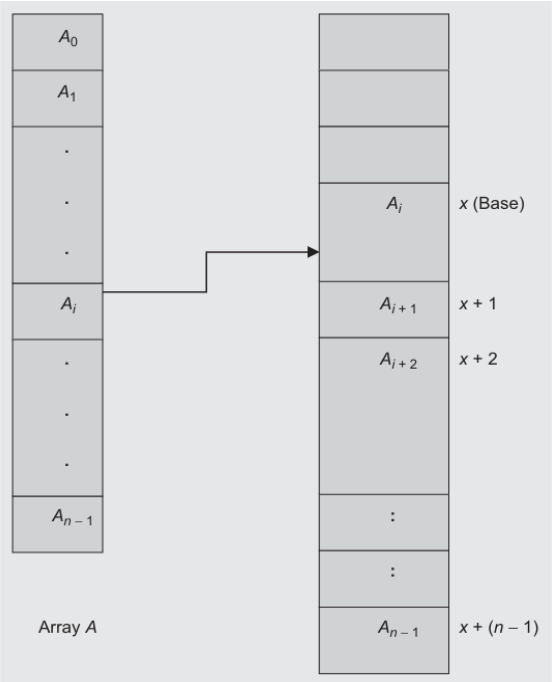


Fig.06 (1-D Array memory Address)

Memory Representation of Two-dimensional Arrays:

Let us consider a two-dimensional array A of dimension m × n. Though the array is multidimensional, it is usually stored in memory as a one-dimensional array. A multidimensional array is represented in memory as a sequence of m × n consecutive memory locations. The elements of a multidimensional array can be stored in the memory as

1. Row-major representation
2. Column-major representation

1. Row Major Representation:

In row-major representation the elements of Array are stored row-wise, that is, elements of the 0th row, 1st row, 2nd row, 3rd row, and so on till the m^{th} row.

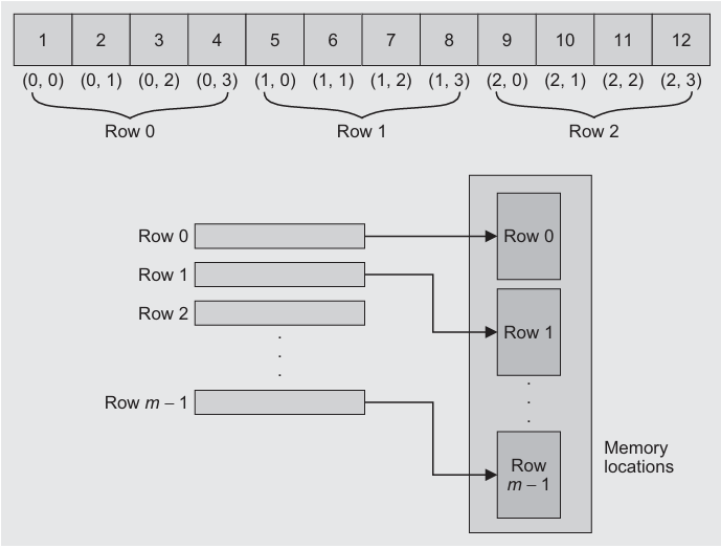


Fig.07 (row-major memory representation)

Address of (A[i][j]) = Base Address + Offset

= Base Address + (Number of rows placed before ith row x Size of row) x (Size of element) + (Number of elements placed before in jth element in jth column) x (Size of element)

Here, size of a row is actually the number of columns n. The base is the address of A[0][0].

Address of A[i][j] = Base + (i × n × Size of element) + (j × Size of element)

Where "i" is the i^{th} row, we are looking for, "n" is the number of elements in each row i.e. columns, "size of element" is size of each element in the array e.g. 4 bytes for an integer, and "j" is the j^{th} column.

2. Column-major representation:

In column-major representation, m × n elements of a two-dimensional array A are stored as one single row of columns. The elements are stored in the memory as a sequence: first the elements of column 0, then the elements of column 1, and so on, till the elements of column n - 1.

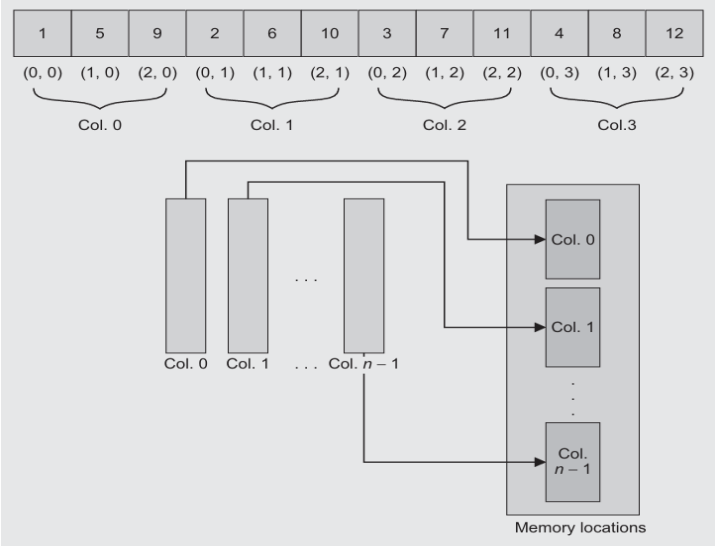


Fig.08 (column-major memory representation)

The address of A[i][j] is computed as

Address of (A[i][j]) = Base Address + Offset

= Base Address + (Number of columns placed before jth column x Size of column) x (Size of element) + (Number of elements placed before in ith element in ith row) x (Size of element)

Here, size of a columns is actually the number of rows m. The base is the address of A[0][0].

Address of A[i][j] = Base + (j × m × Size of element) + (i × Size of element)

For size of the element = 1, the address is

Address of A[i][j] = Base + (j × m) + i

N-Dimensional Arrays:

An n-dimensional m_1 x m_2 x m_3 . . . x m_n array A is a collection of m_1 x m_2 x m_3 . . . x m_n elements in which each element is specified by a list of n integers such as k_1, k_2 , k_3 . . . , k_n called subscripts where 0 <= k_1 <= m_1 – 1, 0 <= k_2 <= m_2 – 1, . . . , 0 <= k_n <= m_n – 1. The elements of array A with subscripts k_1, k_2 , k_3 . . . , k_n is denoted by A[k_1] [k_2] [k_3] . . . [k_n].

Address of $A[k_1][k_2][k_3]\ldots[k_n]$ = $X + (k_1 \times m_2 \times m_3 \times \ldots \times m_n) + (k_2 \times m_3 \times m_4 \times \ldots \times m_n) + (k_3 \times m_4 \times m_5 \times \ldots \times m_n) + (k_4 \times m_5 \times m_6 \times \ldots \times m_n) + \ldots + k_n$

Memory address	Array elements	$m_1 = 2, m_2 = 3, m_3 = 4$
Base	*A*[0][0][0]	
Base + 1	*A*[0][0][1]	
Base + 2	*A*[0][0][2]	
Base + 3	*A*[0][0][3]	
Base + 4	*A*[0][1][0]	Base + $m_3 \times 1$
Base + 5	*A*[0][1][1]	
Base + 6	*A*[0][1][2]	
Base + 7	*A*[0][1][3]	
Base + 8	*A*[0][2][0]	Base + $m_3 \times 2$
Base + 9	*A*[0][2][1]	
Base + 10	*A*[0][2][2]	
Base + 11	*A*[0][2][3]	
Base + 12	*A*[1][0][0]	Base + $m_3 \times 3$
Base + 13	*A*[1][0][1]	
Base + 14	*A*[1][0][2]	
Base + 15	*A*[1][0][3]	
Base + 16	*A*[1][1][0]	Base + $m_3 \times 3 + m_2$
Base + 17	*A*[1][1][1]	
Base + 18	*A*[1][1][2]	
Base + 19	*A*[1][1][3]	
Base + 20	*A*[1][2][0]	
Base + 21	*A*[1][2][1]	
Base + 22	*A*[1][2][2]	
Base + 23	*A*[1][2][3]	

Fig.09 (3-D Array memory representation)

Task 01: Print Dimensions

Write a function that should print the row major based ND to linear index mapping formula against a given number of dimensions.

void rowMajorND (int dimensions);

A function call rowMajorND (3) should print: **i1 D2 D3 + i2 D3 + i3.**

Task 02: Dynamically Allocated 3-D Array

Write a C++ program that should dynamically allocate 3-D Array of integers. The elements of the Array should be randomly initialized. Print elements of Array. Finally Deallocate the Array.

Note: There should not any dangling pointer or memory Leak in your program.

```
void allocateArray (int*** arr, int n = 3);
void initializeArray (int*** arr, int n = 3);
void printArray (int*** arr, int n = 3);
void deallocateArray (int*** arr, int n = 3);
```

You should use pointer notation of Array instead of **subscript/index** Operator to Access the Elements of Array in all functions.

Here is the use of pointer Notation instead of Subscript.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int arr[2][2] = { {1,2},{3,4} };
6      cout << *(*(arr + 1) + 1) << endl;
7      return 0;
8  }
```

Fig.10 (Array Pointer representation)

Explanation:

In above code in line number 5 a static 2-D Array has been defined and initialized with integer values.

In line number 5 the values of the index [1][1] have been displayed through the pointer notation.

Fig.11 (Output)

Task 03: 2-D to 1-D mapping

Create a class of 2D Arrays. This class should use dynamic linear array at the backend. Use Column major mapping to map 2D array to 1 1D. You must develop the following functionalities for the array class

- Constructor, destructor, copy-constructor.
- getIndexValue (i, j) Get the value stored at ith row and jth column.
- setIndexValue (i, j, val) Set the value at ith row and jth column.
- printArray () Print array in 2D format.
- addressOfIndex (i, j, StartIndex) Calculate the address of ith, jth index.
- Overload + Operator
- printSubArray (r1, r2, c1, c2) Print subarray in 2D format.
- clear (m, n) clear the m to n rows and columns of the matrix.

Post-Lab Activities:

Sparse Matrices:

Sparse matrices are matrices in which a significant number of elements are zero or have some default value. These matrices are common in various scientific and engineering applications, such as finite element analysis, network analysis, and image processing, where the data is inherently sparse. Storing sparse matrices efficiently is essential to save memory and improve computational performance. In C++, there are several ways to represent sparse matrices:

1. Compressed Sparse Row (CSR) Format:

In the CSR format, you store three arrays:

- **data**: An array containing non-zero values.
- **indices**: An array containing the column indices of the corresponding values.
- **Index pointers**: An array that points to the beginning of each row in the **data** and **indices** arrays.

This format is efficient for matrix-vector multiplication operations and is widely used in numerical computing libraries like Intel Math Kernel Library (MKL) and SciPy.

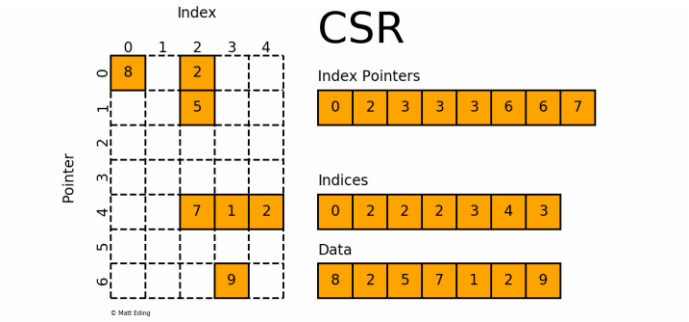


Fig.12 (Compressed Sparse Row)

2. Compressed Sparse Column (CSC) Format:

Similar to CSR, the CSC format stores non-zero values and column indices, but it uses a different indexing scheme. It is often more efficient for some operations like backslash (\) in MATLAB.

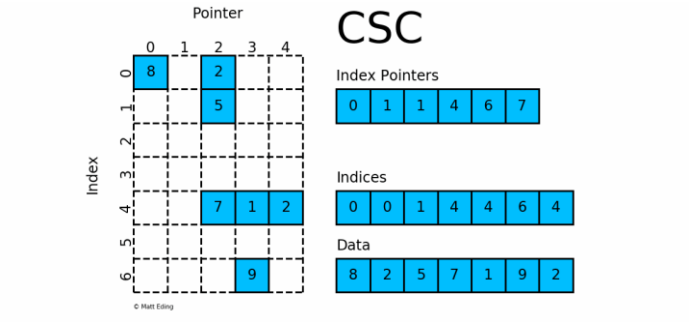


Fig.13 (Compressed Sparse Column)

3. List of Lists (LIL) Format:

In this format, you use a list of lists to represent the sparse matrix. Each list corresponds to a row and contains the non-zero elements along with their column indices. This format is straightforward but not very memory efficient.

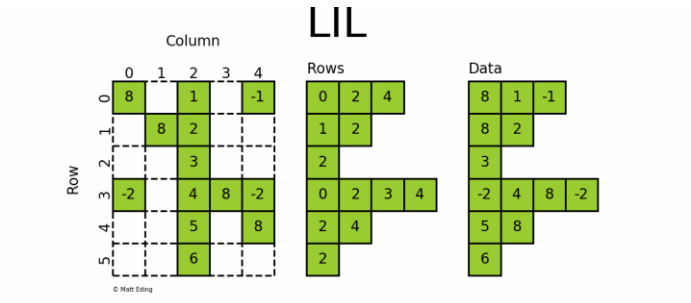


Fig.14 (List of List)

4. Coordinate List (COO) Format:

The COO format stores the (row, column, value) triplets of non-zero elements in a list. This format is simple but can be inefficient for certain operations.

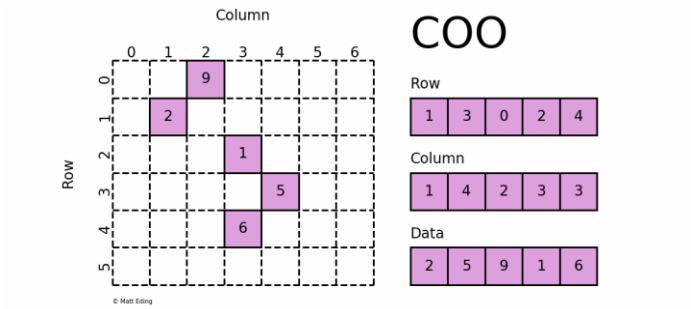


Fig.15 (Coordinate List)

5. Triangular Matrix Formats (CSR, CSC, etc.):

If your sparse matrix is triangular (upper or lower), you can use the CSR or CSC format and ignore the zero elements below or above the main diagonal. In a Lower triangular sparse matrix, all elements above the main diagonal have a zero value. This type of sparse matrix is also known as a lower triangular matrix. In a lower triangular matrix, Array [i],[j] = 0 where i<j.

Similarly, in an upper triangular matrix, Array [i],[j] = 0 where i>j. Similarly, a 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, Array [i][j] = 0, where |i – j| > 1.

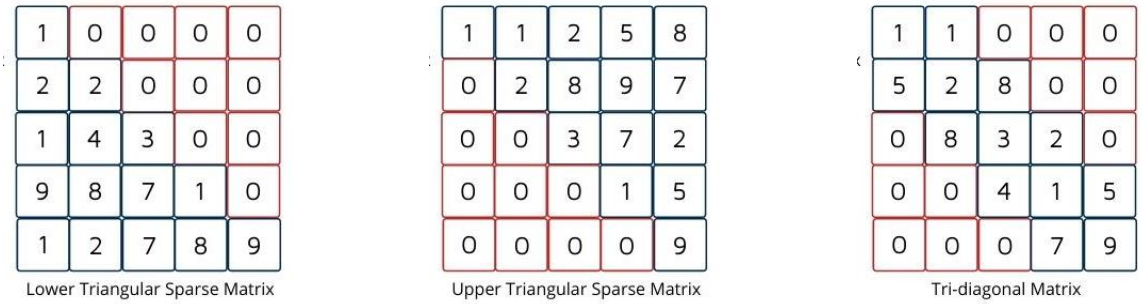


Fig.16 (Triangular Matrix)

6. Dictionary of Keys (DOK) Format:

DOK format uses a dictionary (associative container) to store (row, column) keys and their corresponding values. This format is useful for constructing a sparse matrix incrementally but may not be as efficient for matrix operations.

7. Eigen Sparse Matrix:

If you're using the Eigen library in C++, it provides a built-in SparseMatrix class that can efficiently represent and operate on sparse matrices. It supports various storage schemes, including compressed formats.

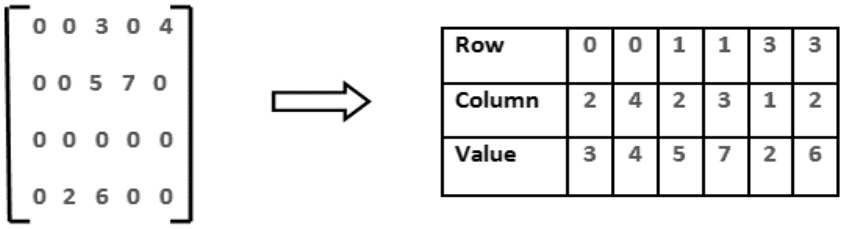
How to Select a Sparse Matrix Representation:

Choosing the right representation depends on your specific application and the operations you need to perform on the sparse matrix. The choice of format can significantly impact memory usage and computational performance. You may also need to consider libraries or frameworks like Eigen or Boost uBLAS that provide efficient implementations of sparse matrix operations.

In this Lab we will be using 2-D array-based representation of Sparse Matrix.

Each element of the matrix is uniquely characterized by its row and column positions. So a triple (i, j, value) can easily represent the non-zero elements of the matrix. In the sparse representation of a matrix, there are three columns. In the first row, we always specify the number of rows, columns, and non-zero elements (No_Of_Non ZeroValues) in columns 1, 2, and 3, respectively. From the second row onwards, we store each non-zero element by its triple (i, j, value).

So in a sparse matrix, there are three columns and (No_Of_NonZeroValues + 1) rows. In general, for space reliability, 3 × (No_Of_NonZeroValues + 1) should always be less than or equal to m x n where m = number of rows and n = number of columns. No_Of_NonZeroValues = Number of non-zero elements. In brief, for the alternate representation, we should have 3 × (No_Of_NonZeroValues + 1) <= m ×n



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

Fig.17 (Sparse Matrices)

In left picture the Matrix is normal, in the right figure the matrix is sparse.

Task 01: Sparse Matrix

Create a 2D matrix class with well-defined data members and member functions to represent the matrix. You have been provided with a template for two classes, **which you can adjust as needed.** Consider how to track the count of non-zero elements in the matrix and contemplate the concept of composition. Should the Sparse Matrix class serve as a composite, with the Matrix class being composed within it?

```
2   class Matrix
3   {
4   private:
5       int** data;
6       int rows;
7       int cols;
8
9   public:
10      Matrix(int numRows=0, int numCols=0);
11      ~Matrix();
12
13      void setValue(int row, int col, int value);
14      int getValue(int row, int col) const;
15      void display() const;
16  };
17
18
```

Fig.18 (Matrix)

Also, overload [][] operator for your matrix Class.

Define another class named SparseMatrix.

```
class SparseMatrix
{
private:
    int non_zero;
    const int colms { 3 };
    int** S_Mat;
public:
    SparseMatrix(int non_zero=0); // the constructor
    void Read_SparseMatrix();

    SparseMatrix  AddSparseMatrix(SparseMatrix B);

    void display() const;
};
```

Fig.19 (Sparse Matrix)

Instantiate two Matrix class objects and initialize them with random values obtained from a file. For instance, if you're creating a 10x10 matrix, it should consist of 100 elements. Out of these 100 elements, ensure that at least 85 percent of them are zeros.

The data of matrices should be read from a text file. Format of file should be like this for a 3x3 matrix.

```
0 3 0
0 0 0
1 0 0
```

Fig.20 (File Format)

Every line represents a row and number of elements in each line tell about elements that should be inserted in matrix in each row. This matrix will be formed from above file data.

0	3	0
0	0	0
1	0	0

You are tasked with populating two matrices using data from separate files. After successfully populating these matrices, convert them into valid Sparse Matrix objects (e.g., named sp_A, etc.). Once the data is read into these two matrices, each object of the Sparse Matrix class should now effectively represent a Sparse Matrix.

Following this, invoke a method within the Sparse Matrix class that performs the addition of these two Sparse Matrices and returns the resulting matrix. To conclude, display the initial two original matrices, the corresponding Sparse Matrices, and the resultant matrix obtained from the addition operation.

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab and Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:** **[20 marks]**
  - Task 01: Unsorted List ADT [10 marks]
  - Task 02: Polynomial ADT [10 marks]
- **Division of In-Lab marks:** **[30 marks]**
  - Task 01: Print Dimensions [05 marks]
  - Task 02: 3-D Array [10 marks]
  - Task 03:2-D to 1-D Mapping [15 marks]
- **Division of Post-Lab marks:** **[20 marks]**
  - Task 01: Sparse Matrix [20 marks]

References and Additional Material:

- Abstract Data Types
  https://www.geeksforgeeks.org/abstract-data-types/

- Sparse Matrices
  https://www.geeksforgeeks.org/sparse-matrix-representation/

Lab Time Activity Simulation Log:

- Slot – 01 – 02:15 – 02:30: Class Settlement
- Slot – 02 – 02:30 – 03:00: In-Lab Task 01
- Slot – 03 – 03:00 – 03:45: In-Lab Task 02
- Slot – 04 – 03:45 – 04:30: In-Lab Task 03
- Slot – 05 – 04:30 – 04:45: Discussion on Post-Lab Task
- Slot – 12 – 04:45 – 05:15: Evaluation