Parul® University

Data Structure

Dr. Ghanshyam Rathod, Assistant Professor
IT and Computer Science

Parul®
University

DIGITAL LEARNING CONTENT

CHAPTER-1

Introduction of Data Structure

Terms

- **Data**: facts and statistics collected together for reference or analysis.
- Collection of values, for example, student's name and its id are the data about the student.
- **Information** : Organized data
- **Structure** : Way of organising information , so that it is easier to use.
- **Data organization**, in broad terms, refers to the method of classifying and organizing data sets to make them more useful.

What is Data Structure ?

What is Data Structure ?

- Data Structures are the programmatic way of storing and organizing data.
- Is a data organization , management and storage format that enables efficient access and modifications.
- Allows handling data in an efficient way.
- Plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

- **“Way we organize our data”**

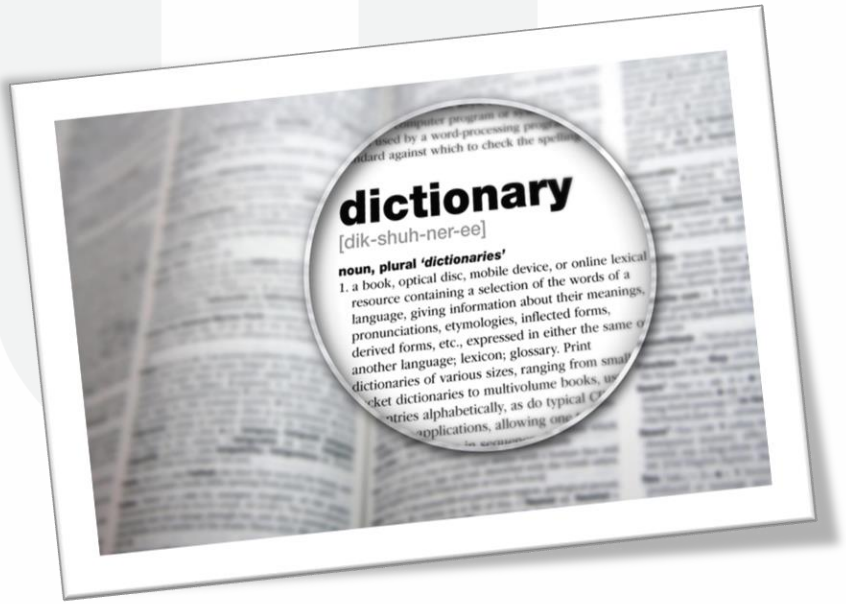
Need of Data Structure

- As applications are getting complex and amount of data is increasing day by day, there may arise the following problems:

1. Processor speed
2. Data Search
3. Multiple requests

Advantages of Data Structures

- Efficiency – access and store data
- Reusability
- Used to manage large amounts of data
- Specific data structure used for specific tasks



Applications of DS

- Compiler Design
- Operating System
- DBMS
- Simulation
- Network Analysis
- AI
- Graphs
- Numerical Analysis
- Statistical Analysis Package

Algorithm-Introduction

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Categories of Algorithm

- **Search** – Algorithm to search an item in a data structure.
- **Sort** – Algorithm to sort items in a certain order.
- **Insert** – Algorithm to insert item in a data structure.
- **Update** – Algorithm to update an existing item in a data structure.
- **Delete** – Algorithm to delete an existing item from a data structure.

Criteria for efficiency of the algorithm

- Correctness
- Implementation
- Simplicity
- Execution time
- Memory space required
- Alternative way of doing the same task.

Example

Problem – Design an algorithm to add two numbers and display the result.

```
Step 1 – START
Step 2 – declare three integers a, b & c
Step 3 – define values of a & b
Step 4 – add values of a & b
Step 5 – store output of step 4 to c
Step 6 – print c
Step 7 – STOP
```

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –

```
Step 1 – START ADD
Step 2 – get values of a & b
Step 3 – c ← a + b
Step 4 – display c
Step 5 – STOP
```

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Characteristics of Algorithm

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.

Algorithm Complexity

- Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

1. **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
2. **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

- The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

Asymptotic Analysis

- In mathematical analysis, asymptotic analysis of algorithm is a method of defining the mathematical boundation of its run-time performance.
- Using the asymptotic analysis, we can easily conclude about the average case, best case and worst case scenario of an algorithm.
- It is used to mathematically calculate the running time of any operation inside an algorithm.
- Usually the time required by an algorithm comes under three types:
  1. **Worst case:** It defines the input for which the algorithm takes the huge time.
  2. **Average case:** It takes average time for the program execution.
  3. **Best case:** It defines the input for which the algorithm takes the lowest time

Data type

- Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data.
- **There are two types of data types –**

1. Built-in Data Type
2. Derived Data Type

Built-in Data Type

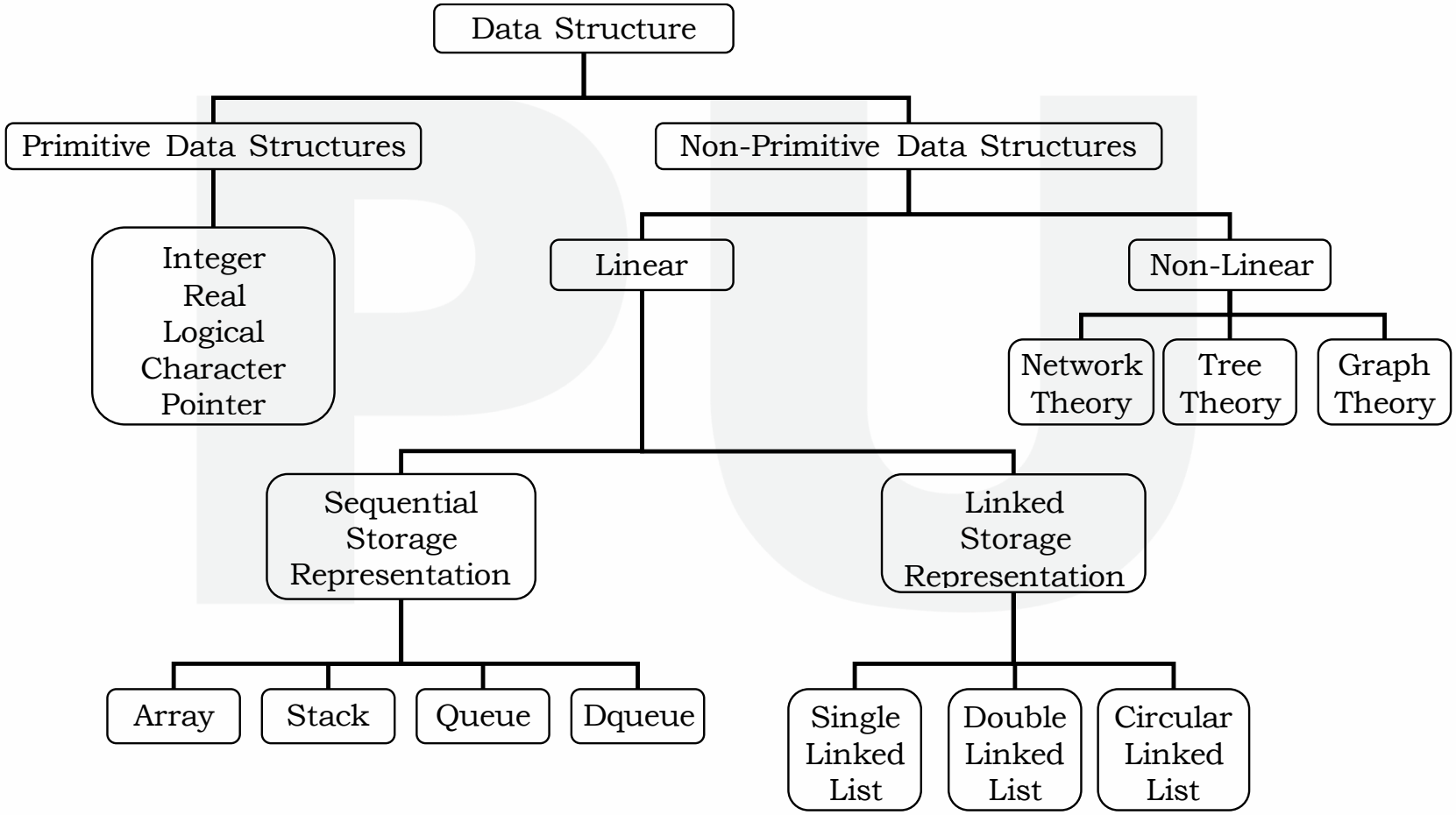
- Those data types for which a language has built-in support are known as Built-in Data types.
- For example, most of the languages provide the following built-in data types.

1. Integers
2. Boolean (true, false)
3. Floating (Decimal numbers)
4. Character and Strings

Derived Data Type

- Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types.
- These data types are normally built by the combination of primary or built-in data types and associated operations on them.
- For example –
    1. Linked List
    2. Array: int arr[1000];
    3. Stack
    4. Queue

Classification of Data Structure



Primitive Data Structure

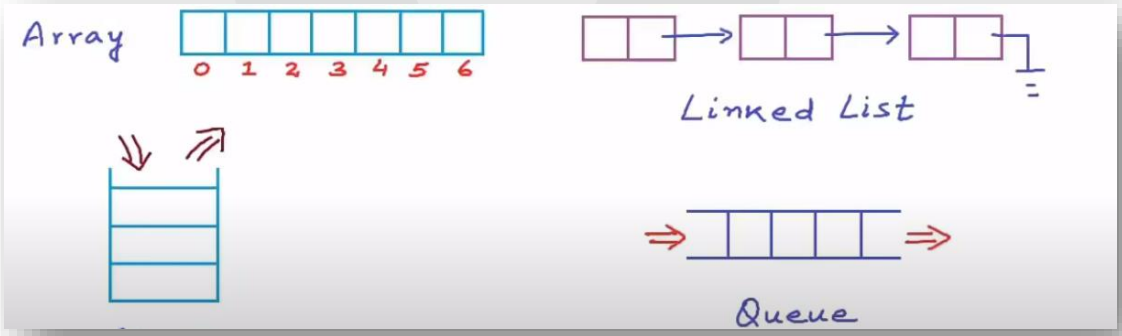
- There are basic structures and directly operated upon by the machine level instructions.
- In general, there are different representation on different computers.
- It is a fundamental data type.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Non-Primitive Data Structure

- These are derived from the primitive data structures.
- The non primitive data structures are not directly operated by machine level instruction.
- The non primitive data structure is also known as complex or composite data structure.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.
- Lists, Stack, Queue, Tree, Graph are example of non-primitive data structures.

Linear Data Structure

- Data structure where data elements are arranged sequentially or linearly where the elements are attached to its previous and next adjacent in what is called a linear data structure.
- In linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only.
- Linear data structures are easy to implement because computer memory is arranged in a linear way. Its examples are array, stack, queue, linked list, etc.



Non-Linear Data Structure

- Data structures where data elements are not arranged sequentially or linearly are called non-linear data structures.
- In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only.
- Non-linear data structures are not easy to implement in comparison to linear data structure.
- It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs

Operations on DS

Basic operations on DS:

- Creation
- Selection
- Updation
- Deletion

Other operations on DS:

- Searching
- Sorting
- Merging

Creation

- This operation is used to create a data structure.
- e.g. In ‘C’ language using declaration statement
  int n = 45;
- Once any data is declared, it is associated with memory space, address and the value.

1010	45	n
Address	Value	Name

Selection

- This operation is used to access data within data structure
- Selection is used to refer the value stored in data structure
- e.g. created data structure is int a = 10;
- So value 10 is referred from created data structure.

Updation

- Once the structure created user can update / change the data.
- a = 10;

Deletion

- Once the structure is used and its purpose is over, it is to be destroyed.

Operations on Non-Primitive Data Structure

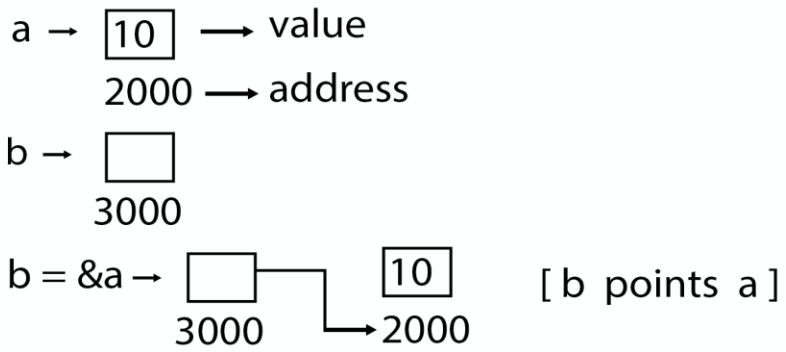
Traversing	Processing each element exactly once.
Sorting	Arranging elements in some logical order. (Ascending or descending)
Merging	Combine the element in two different sort lists into a single sorted list.
Searching	Finding the location of element.
Insertion	adding a new element to the data structure.
Deletion	Removing an item from the data structure.

A	B	C	D	E
120	121	122	123	124

```
char item;
int I;
printf(“Enter Item to Search:”);
scanf(“%c”,&item);
for(i=0; i<5;i++)
{
        if(a[i] == item)
                printf(“Index of %c is %d”,item,i);
        else
                printf(“Item not found”);
}
```

Pointers

- Pointer is used to points the address of the value stored anywhere in the computer memory.
- To obtain the value stored at the location is known as dereferencing the pointer.




```
int a = 5, *b;
b = &a;
printf("%d",a);
printf("%d", *b);
```

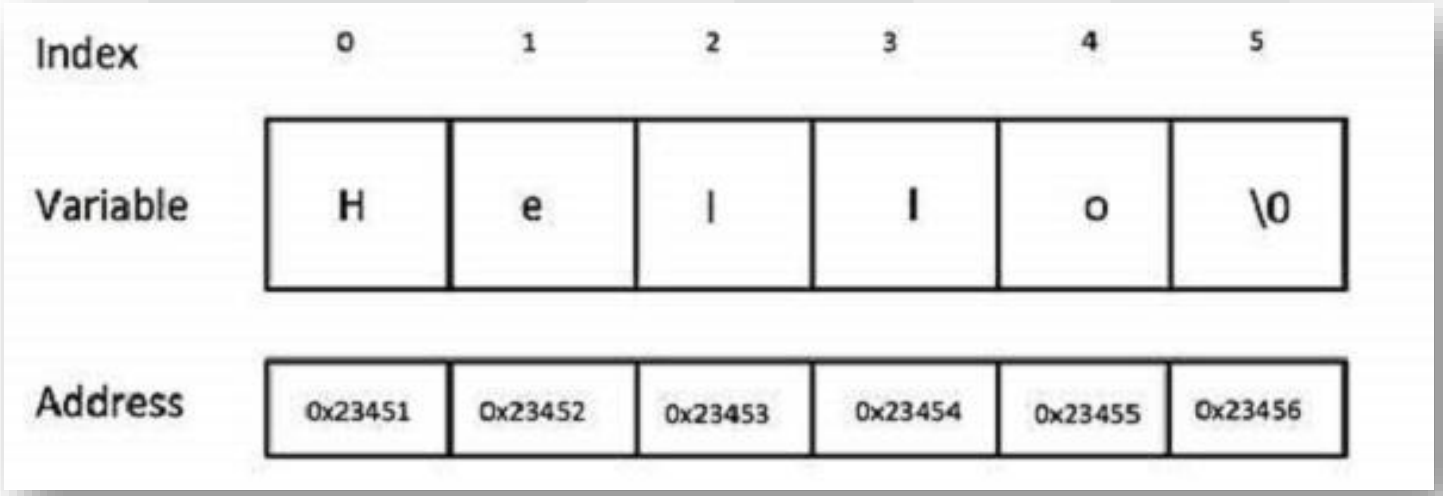
Strings

- In computer programming, a string is traditionally a sequence of characters, either as a literal constant or as some kind of variable.
- A string is generally considered as a data type and is often implemented as an array data structure of bytes (or words) that stores a sequence of elements, typically characters, using some character encoding.
- String may also denote more general arrays or other sequence (or list) data types and structures.

- char name[20];

Storage representation of Strings

- In C string is stored in an array of characters. Each character in a string occupies one location in an array.
- The null character '\0' is put after the last character. The '\0' character is included in the size of the string in 'C' language but it is extra in C++.
- This is done so that program can tell when the end of the string has been reached.
- For example, the string "Hello" is stored as



Strings

- Since the string contains 5 characters. it requires a character array of size 6 to store it. the last character in a string is always a NULL('\0') character.
- Always keep in mind that the '\0' is not included in the length if the string, so here the length of the string is 5 only.
- Notice above that the indexes of the string starts from 0 not one so don't confuse yourself with index and length of string.

String / Text handling functions

- #include<string.h>
- include this library to your program to use some of the inbuilt string manipulation functions present in the library.
- There are about 20-22 inbuilt string manipulating functions present in this library.

1. strlen("name of string")
2. strcpy(dest, source)
3. strcmp(string1, string2)

int strlen(string)

- This function accepts string as parameter and return integer i.e. the length of String passed to it.
- Example

```
#include<stdio.h>
#include<string.h>
void main()
{
        char string[]="spark";
        int len;
        len=strlen(string);
        printf("length of %s is %d\t", string, len);
}
```

Output::length of spark is 5.

strcpy(destination string, source string)

- This function accepts 2 strings as parameter, 1st one is destination string and 2nd is source string.
- This function copies source string to destination string.
- Example

```
#include<stdio.h>
#include<string.h>
void main()
{
    char src[]="spark", dest[15];
    strcpy(dest,src);
    printf("%s is copied to dest string\t",dest);
}
```

Output: spark is copied to dest string.

strcmp(str1, str2)

- This function is used to compare two strings "str1" and "str2". this function returns zero("0") if the two compared strings are equal, else some none zero integer.
- Declaration and usage of strcmp() function
  strcmp(str1 , str2); //declaration //using this function to check given two strings are equal or not.
- str1=Abc
- str2=abc

```
if( strcmp( str1, str2 )==0)
{
        printf("string %s and string %s are equal\n",str1,str2);
}
else
        printf("string %s and string %s are not equal\n",str1,str2);
```

char str1[] = “abc”
char str2[] = “Abc”

a - A = 97 – 65 = 32
b – b = 66 – 66 = 00
c – c = 67 – 67 = 00

32 + 0 + 0 = 32

char str1[] = “Abc”
char str2[] = “abc”

A - a = 65 – 97 = -32
b – b = 66 – 66 = 00
c – c = 67 – 67 = 00

-32 + 0 + 0 = -32

char str1[] = “abc”
char str2[] = “abc”

a - a = 65 – 65 = 00
b – b = 66 – 66 = 00
c – c = 67 – 67 = 00

0 + 0 + 0 = 00

strcat(Destination string, Source string)

- This function accepts two strings source string is appended to the destination string.
- Example

```
#include<stdio.h>
#include<string.h>
void main()
{
    char src[]=“C", dest[]=“Programming in ";
    strcat(dest,src);
    printf("concatenated string is %s",dest);
}
```

Output: concatenated string is “Programming in C”.

Keyword in Context (KWIC) Indexing

- An important area in computer science is information retrieval. In information retrieval applications, a data base, which may contain a wide variety of data structures, which is maintained on an on-line basis using large random-access (e.g., disk) files.
- These files are searched for requested information based on index items generated from a user query.
- One method of indexing that is widely used in library systems is the permuted or KWIC (key-word-in-context) indexing scheme. A KWIC index provides the context surrounding each occurrence of each word.
- Example: In KWIC scheme a title of a book is “Introduction to Data Structures with Applications”, where each item of phrase is scanned for keywords and reproduced once in a permuted fashion for each keyword.

  Introduction to Data Structures with Applications // An
  Data Structures with Applications // An Introduction to
  Structures with Applications // An Introduction to Data
  Applications // An Introduction to Data Structures

Keyword in Context (KWIC) Indexing

- Underscored word in each index is called index word.
- Keywords are the meaning full words, are the main words which represents the subject of document.
- ‘a’, ‘for’, ‘to’, ‘the’, ‘an’, ‘and’, ‘with’ are ordinary words to represent the subject of document.
- KWIC generator creates 4 arrays, which stores below information:
  - ORD_WORDS : Ordered list of Ordinary words
  - KEYWORDS : Ordered list of main words
  - TITLE : Complete book title
  - T_INDEX : List of array indices for the titles

Array

- An array is a ***fixed-size*** sequenced collection of elements of the same data type that share a common name.
- Examples:
  - List of temperatures recorded every hour in a day, or a month, or a year.
  - List of employees in an organization.
  - List of products and their cost sold by a store.
  - Test scores of a class of students.
  - List of customers and their telephone numbers.
- For instance, we can use an array name salary to represent a set of salaries of a group of employees in an organization. We can refer to the individual salaries writing a number called ***index*** or ***subscript*** in brackets after the array name. For example:

```
int salary[10]
```

- represents the salary of 10 employees and individual elements are called ***elements***.

Types of Arrays

- One – Dimensional Arrays
  - A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a One-Dimensional array.

    int arr[5]
- Two – Dimensional Arrays
  - A list of items can be given one variable name using two subscripts and such a variable is called a two-Dimensional array.
  - In two dimensional array will be created using rows and columns, in below example there are 3 rows and 5 columns.

    int arr[3][5]
- Multi – Dimensional Arrays
  - A list of items can be given one variable name using more then two subscripts and such a variable is called a multi-Dimensional array.

    int arr[2][3][5]

Storage Representation of 1-D Array

- Consider a one-dimensional array of integer type which requires 2 bytes in memory for each element.
- The lowest index value is called lower bound and the highest index value is called upper bound.

Let

L_0 = Starting address of an array in memory (Base Address)

A_i = the i^{th} element in the array for which we want to find the address.

c = No of bytes required for each element in array (In our case 2 Byte)

b = Lower bound of the array.

Then the address of element A_i is given by:

$$loc(A_i) = L_0 + c^* (i-b)$$

Storage Representation of 1-D Array

E.g. Consider an array A[10] with L_0 = 1026, c = 2 Bytes, b = 1

Then the address for element A [5] is obtained as follows:

A [5] = L_0 + c * (i-b)

A [5] = 1026 + 2 * (5-1)

A [5] = 1026 + 2 * 4

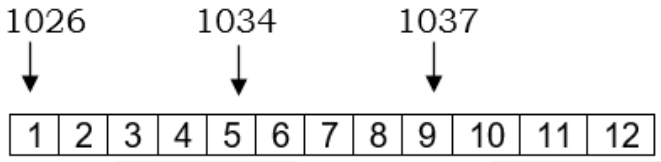
A [5] = 1026 + 8 = 1034.

Storage Representation of 2-D Array

- Consider the following two dimensional array A[m][n] where m = 3 and n =4
- There are two techniques for storing two dimensional arrays

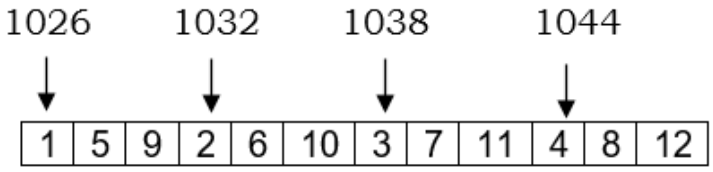
A. Row Major Order

	1	2	3	4
1	1	2	3	4
2	5	6	7	8
3	9	10	11	12



B. Column Major Order.

	1	2	3	4
1	1	2	3	4
2	5	6	7	8
3	9	10	11	12



Storage Representation of 2-D Array (Row Major)

To calculate a particular address in memory for the Aijth element use the following formula:-

L_0 = Starting address of an array in memory (Base Address)

A_{ij} = The ith element in the array for which we want to find the address

b_1 = Lower bound of the row

b_2 = Lower bound of the column

u_1 = Upper bound of the row

u_2 = Upper bound of the column

c = No of bytes required for each element in array (In our case 2 Byte)

Storage Representation of 2-D Array (Row Major)

e.g. Loc(A23) can be calculated as follows:-

$Loc(A_{ij}) = L_0 + [(i-b_1) * (u_2 - b_2 + 1) + (j - 1)] * c$

Loc(A23) = 1026 + [(2-1) * (4-1+1) + (3-1)] *2

= 1026 + [(1) * (4) + (2)] *2

= 1026 + [4 + 2] *2

= 1026 + 12

= 1038

A23 = element 7.

B1 = 1, b2 = 1 u1 = 3, u2 = 4

Storage Representation 2-D Array (Column Major)

e.g. Loc(A21) can be calculated as follows :

$= Loc(A_{ij}) = L_0 + [(i-b_1) * (u_2 - b_2 + 1) + (j - 1)] * c$

$Loc(A_{ij}) = L_0 + [(j-b_1) * (u_1 - b_1 + 1) + (i - 1)] * c$

Loc(A21) = 1026 + [(1-1) * (3-1+1) + (2-1)] *c

= 1026 + [(0) * (3) + (1)] *2

= 1026 + [1]*2

= 1028

A21 = element 5

B1 = 1, b2 = 1 u1 = 3, u2 = 4

Special types of Array – Sparse Matrix

- A matrix is called a two-dimensional data object made up of m rows and n columns, therefore having total m x n values.
- If in my matrix most of the elements of the matrix have 0 value, then it is called a sparse matrix.
- Here instead of storing zeroes with non-zero elements, we only store non-zero elements.
- That means we will store non-zero elements with triples- (Row, Column, value).
- 2D array will used to represent a sparse matrix in which there are three rows named as: row, column, and value.

Sparse Matrix

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

Sparse Matrix →

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

Table Structure

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

Sparse Matrix →

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

- In the above figure, we can observe a 5x4 sparse matrix containing 7 non-zero elements and 13 zero elements.
- The above matrix occupies 5 x 4 = 20 memory space.
- Increasing the size of matrix will increase the wastage space.

Sparse Matrix

- In the above structure, first column represents the rows, the second column represents the columns, and the third column represents the non-zero value.
- The first row of the table represents the triplets. The first triplet represents that the value 4 is stored at 0th row and 1st column.
- Similarly, the second triplet represents that the value 5 is stored at the 0^{th} row and 3^{rd} column. In a similar manner, all triplets represent the stored location of the non-zero elements in the matrix.

Sparse Matrix

- The size of the table depends upon the total number of non-zero elements in the given sparse matrix.
- Above table occupies 8x3 = 24 memory space which is more than the space occupied by the sparse matrix.
- So, what's the benefit of using the sparse matrix?
- Consider the case if the matrix is 8*8 and there are only 8 non-zero elements in the matrix.
- Then the space occupied by the sparse matrix would be 8 * 8 = 64
- But in sparse matrix the space occupied by the table represented using triplets would be 8 *3 = 24.

Triangular Matrix

- **Matrix** is defined as a rectangular array of numbers that are arranged in rows and columns.
- The size of a matrix can be determined by the number of rows and columns in it. An **"m by n" matrix has "m" rows and "n" columns** and is written as an "m × n" matrix.
- For example, a matrix of order "5 × 6" has five rows and six columns.
- **Triangular matrix** is a special case of a square matrix, where all elements above or below the principal diagonal are zeros.
- An upper triangular matrix is a square matrix, whose all elements below the principal diagonal are zeros.
- A lower triangular matrix is a square matrix, whose all elements above the principal diagonal are zeros. The matrices in the image given below are upper triangular and lower triangular matrices of order "4 × 4."

Triangular Matrix

- The matrices in the image given below are upper triangular and lower triangular matrices of order "4 × 4."

Upper Triangular Matrix

$$U = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{bmatrix}_{4\times4}$$

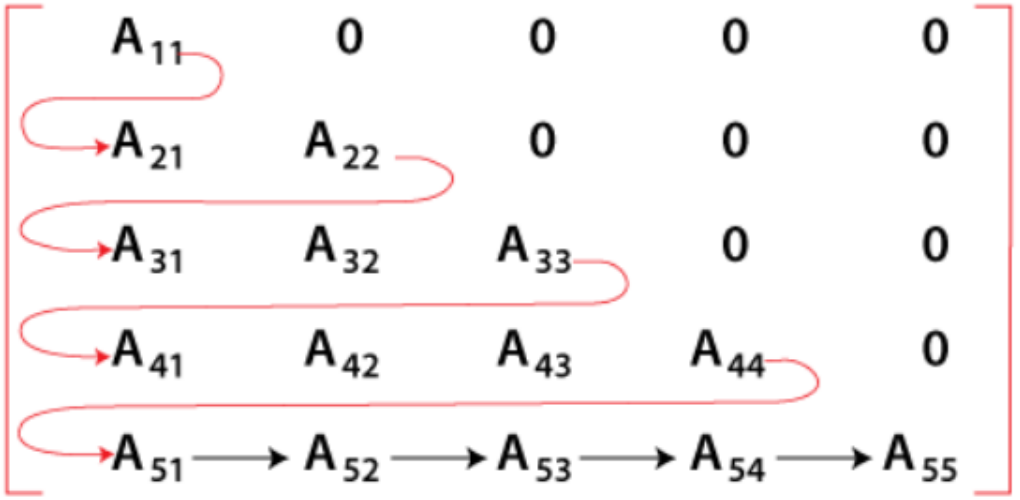
Lower Triangular Matrix

$$L = \begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}_{4\times4}$$

Triangular Matrix

Storing Lower triangular matrices

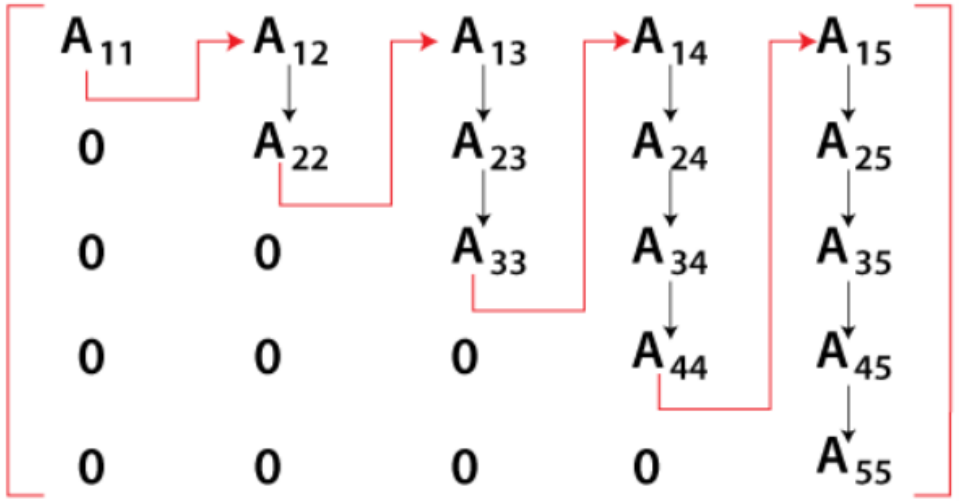
- In a lower triangular matrix, the non-zero elements are stored in a 1-dimensional array **row by row.**
- For example: The 5 by 5 lower triangular matrix as shown in the figure is stored in one-dimensional array B is:
- B = { A11, A21, A22, A31, A32, A33, A41, A42, A43, A44, A51, A52, A53, A54, A55}



Triangular Matrix

Storing Upper triangular matrices

- In a lower triangular matrix, the non-zero elements are stored in a 1-dimensional array **column by column**.
- For example: The 5 by 5 upper triangular matrix as shown in the figure is stored in one-dimensional array B is:



- B = { A11, A12, A22, A13, A23, A33, A14, A24, A34, A44, A15, A25, A35, A45, A55}
- As calculated, for storing non-zero elements, **N*(N + 1)/2** space is needed.
- Taking the above example, **N = 5**. Array of size **5 * (5 + 1)/2 = 15** is required to store the non-zero elements.
- So, for integer array **15 * 2 Bytes = 30 Bytes** are required.

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in