GLS UNIVERSITY

SEM III
DATA STRUCTURES
UNIT– I

Data ?

A Value or Set of Values

Example:

- 66
- 15/6/2016
- Pascal
- 21,31,41,51

Entity ?

An Entity is one that has certain attributes and which may be assigned value.

Example:

Entity - Student

Attributes -   Name   RollNo   BOD
  - ABC   888   5/6/16

Information ?

Meaningful data or processed data.

Information is used for data with its attributes.
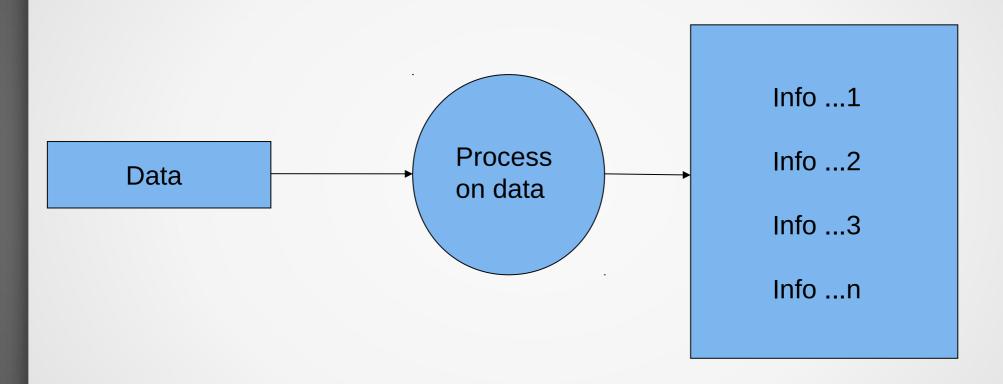
Example:

Data	Meaning
22	Age of person
Herry	Nick name of person
22/4/15	DOB of person

Relationship Between Data & info.

Algorithm ?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task.
An algorithm is a procedure or formula for solving a problem.

Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output.

Ex:

Search − Algorithm to search an item in a datastructure.

Sort − Algorithm to sort items in certain order

Insert − Algorithm to insert item in a datastructure

Update − Algorithm to update an existing item in a data structure

Delete − Algorithm to delete an existing item from a data structure

Problem − Design an algorithm to add two numbers and display 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

Problem Analysis

Generally , we define a PROBLEM a single statement that can be define in some general terms.

e.g. what is an output of the program??

write a program to traverse an elements in an array???

ANALYSIS

TO ANALYSIS A PROBLEM WE HAVE MEASURES WHICH IS AS FOLLOW:-

1.) BY USING AN ALGORITHM

An algoritjm is well defined steps for solving the computational problem in order to get better eficiency anda analysing of the PROBLEM..

Algortihm Analysis

- Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as mentioned below −

- A priori analysis − This is theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.

- A posterior analysis − This is empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm Compexity

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.

Time Factor – The time is measured by counting the number of key operations such as comparisons in sorting algorithm

Space Factor – The space is measured by counting the maximum memory space required by the algorithm.

Time Complexity

Time Com. T(p) is the time taken by a program P , the sum of its compile and execution times.

We need to determine the no of steps needed by a program to solve problem

1. New variable, global variable= 0

2. manually computer the number of times each statement will be execute

Cases

- **Best Case** com. of an algorithm is the function defined by the **minimum** number of steps taken on any instance of size n.

- **Worst case** com. of an algo. is the function defined by the **maximum** number of steps taken on any instance of size n.

- **Average case** com... **average** number of steps taken on any instance of size n

Space Complexity

amount of computer memory required during program execution as a function of input size

Compile Time: storage requirement of program at CT

Runtime:program need dynamic variable and dynamic data strucure

Space

- Program space: this is the memory occupied by the program itself.

- data space: this is the memory occupied by data members such as constant and variables.

- stack space: stack memory needed to save the function run-time.

Data types ?

- Data type specifies the type of data stored in a variable.

- The data type can be classified into two types Primitive data type and Non-Primitive data type

- PRIMITIVE DATATYPE

The primitive data types are the basic data types that are available in most of the programming languages.

The primitive data types are used to represent single values.

- **Integer:** This is used to represent a number without decimal point.
  - Eg: 12, 90

- **Float and Double:** This is used to represent a number with decimal point.
  - Eg: 45.1, 67.3

- **Character :** This is used to represent single character
  - Eg: 'C', 'a'

- **Boolean:** This is used represent logical values either true or false.

NON-PRIMITIVE DATATYPES

The data types that are derived from primary data types are known as non-Primitive data types. These datatypes are used to store group of values.

The non-primitive data types are

Arrays
Structure
Union
linked list
Stacks
Queue etc

Data Structure

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.

A data structure is a specialized format for organizing and storing data.

General data structure types include array, file, record, table, tree.

Definition

A data structure is a specialized format for organizing and storing data.

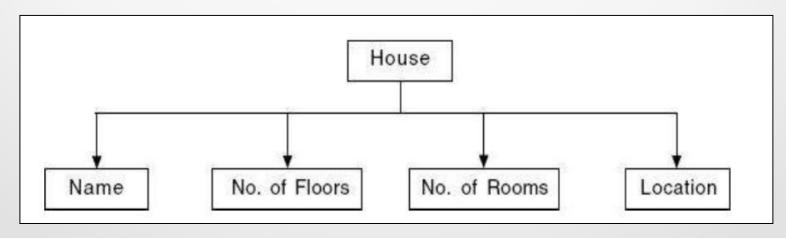Domain(D): range of value that the data may have.

Function(F): set of operations which may legally be applied to element of the data object.

Axioms(A): set of rules with which the different operations belonging to f can actually be implement

A Data Structure D is a triplet that is D=(D,F,A) where D is a set of data object, F is a set of functions and A is set of rules to implement the functions

Data Structure

✂ Data structures are a method of representing of **logical relationships between individual data elements related to the solution of a given** problem.

✂ Data structures are the most convenient way to handle data of different types.

✂ For example, the characteristics of a house can be represented by the house name, house number, location, number of floors, number of rooms on each floor, kind of fencing to the house—either with brick walls or wire, electrification—either underground or open, whether a balcony has been provided or not, etc.
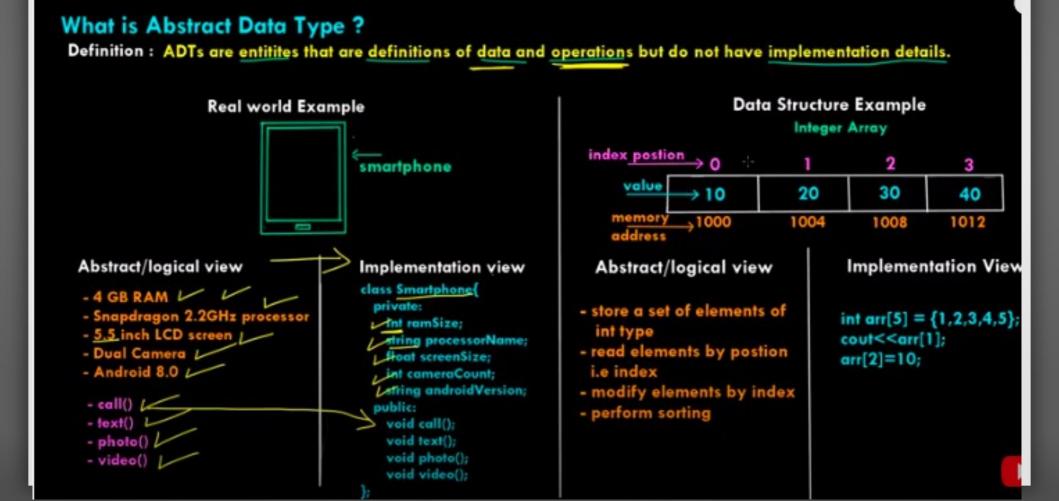
Abstract Data Types(ADT)

Theoritical concept
study or view two ways
2 ways of looking at Data structures
Abstarct Models/mathematical/logical view/specification/rules/regulations
Implementation using practicaly using some programming language
And will implement according to rules

ADT : ADTs are entities that are definitions of data and operations but do not have implemention details. Only logical/ abstarct view
Entites that have definitions for data and operations but dont have impemenation

We know what we are going to store, operations and the way we r going to strore depending on data structure but dont have implementation
The reason behind not having implementation
Because every programming have Different programming language has different implementation strategies

Abstract Data Types(ADT)

Abstract Data Types(ADT)

To simplify the process of solving the problems, we combine the data structures along with their operations and call it as ADT.

Abstract Data type (ADT) is a type (or class) for objects
whose behavior is defined by a set of value and a set of operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.

It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

It is called "abstract" because it gives an implementation independent view.
The process of providing only the essentials and hiding the details is known as abstraction.

The user of data type need not know that data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented.

So a user only needs to know what a data type can do but not how it will do it.

We can think of ADT as a black box which hides the inner structure and design of the data type.

Example: List ADT, Stack ADT, Queue ADT.

Stack ADT

A Stack contains elements of same type arranged in sequential order. All operations takes place at a single end that is top of the stack and following operations can be performed:
push() - Insert an element at one end of the stack called top.
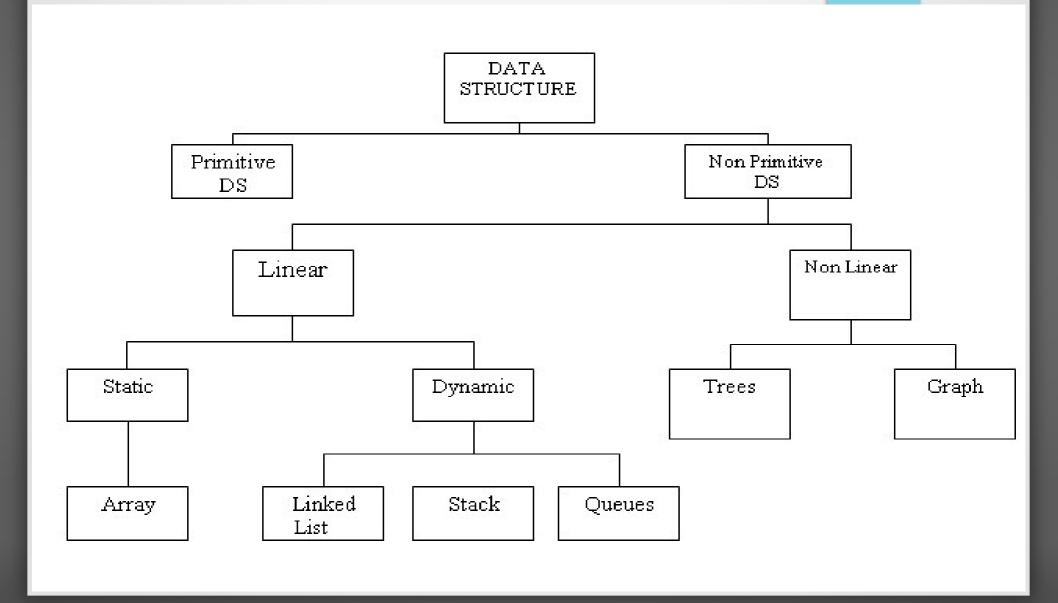pop() - Remove and return the element at the top of the stack, if it is not empty.
peep() - Return the element at the top of the stack without removing it, if the stack is not empty.
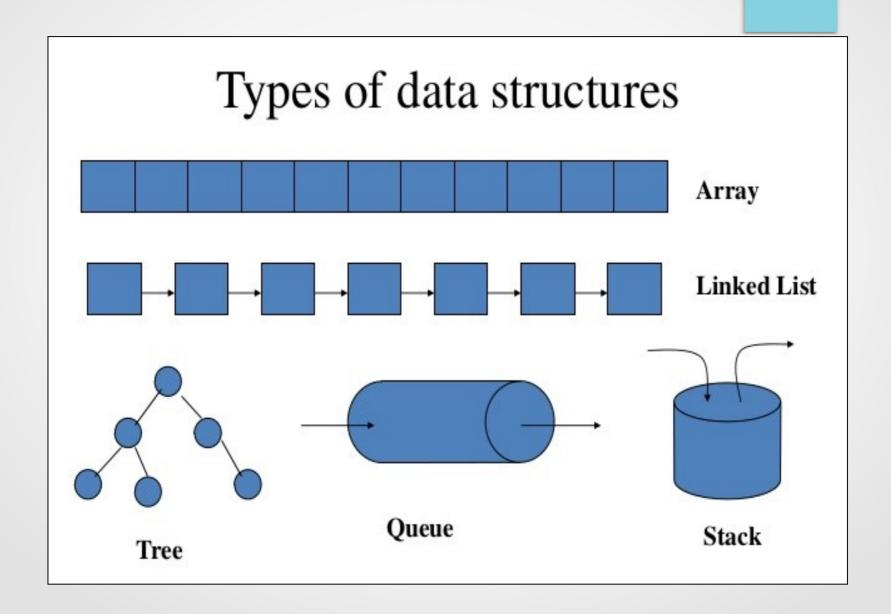size() - Return the number of elements in the stack.
isEmpty() - Return true if the stack is empty, otherwise return false.
isFull() - Return true if the stack is full, otherwise return false.

Types of Data Structures

Types of Data Structures: Linear & Non-Linear

✂A **linear** data structure traverses the data elements sequentially, in which only one data element can directly be reached. Ex: Arrays, Linked Lists.

✂Linked list is an example of linear data storage or structure. Linked list stores data in an organized a linear fashion. They store data in the form of a list.

✂**Non-Linear** data structure: Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure. Ex: Trees, Graphs.

✂Non Linear data structure- Tree data structure is an example of a non linear data structure. A tree has one node called as root node that is the starting point that holds data and links to other nodes.
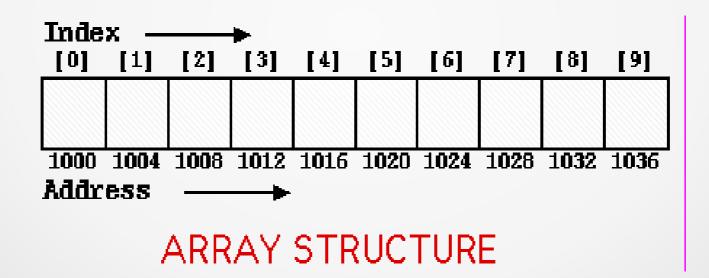
Types of Data Structures: Linear & Non-Linear

Types of Data Structures: Linear & Non-Linear

- **Linear Data Structure:**
  - An **arrays** is a collection of data elements where each element could be identified using an index.
  - A **linked list** is a sequence of nodes, where each node is made up of a data element and a reference to the next node in the sequence.
  - A **stack** is actually a list where data elements can only be added or removed from the top of the list.
  - A **queue** is also a list, where data elements can be added from one end of the list and removed from the other end of the list.
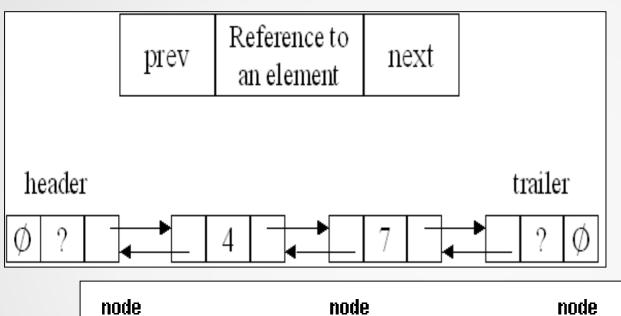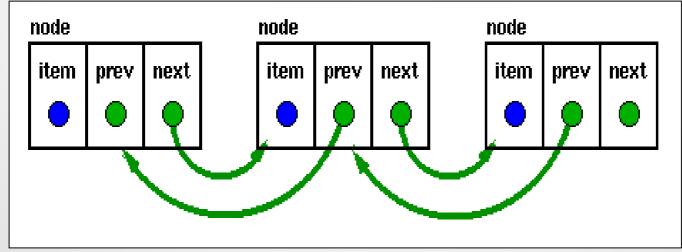
- **Non-Linear Data Structure:**
  - A **tree** is a data structure that is made up of a set of linked nodes, which can be used to represent a hierarchical relationship among data elements.
  - A **graph** is a data structure that is made up of a finite set of edges and vertices. Edges represent connections or relationships among vertices that stores data elements.

Types of Data Structures: Linear & Non-Linear

In **linear data structures, data elements are organized sequentially and therefore they are easy to implement in the computer's memory. In nonlinear data structures, a data element can be attached to several other data elements to represent specific relationships that exist among them.**
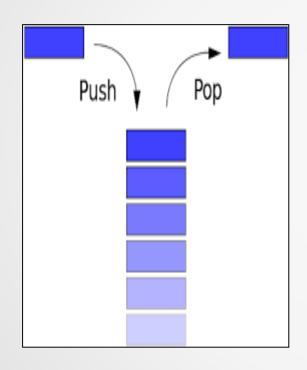

ARRAY STRUCTURE

Linear: LINKED LIST
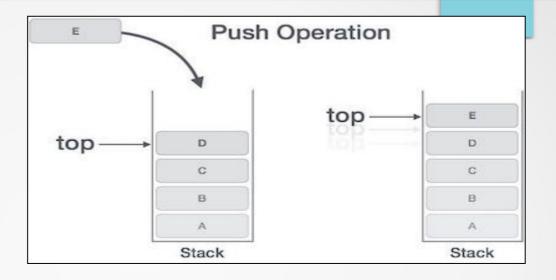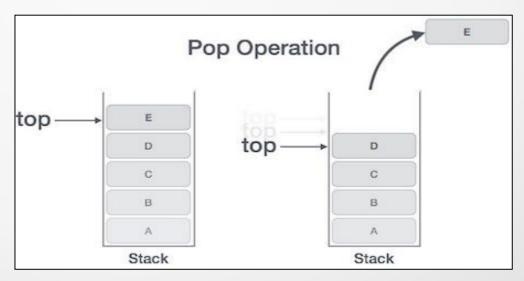
Linked List

Linear: STACK

Stack

Linear: QUEUE

enqueue() operation

dequeue() operation

REAR

FRONT

enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .
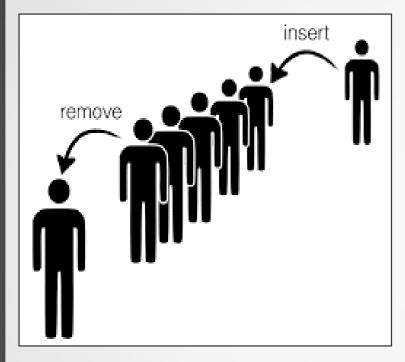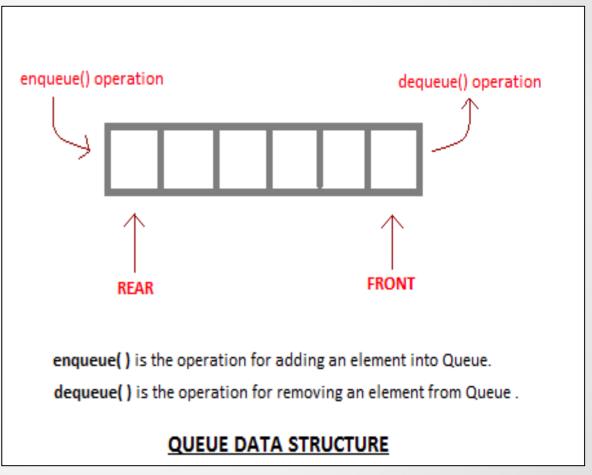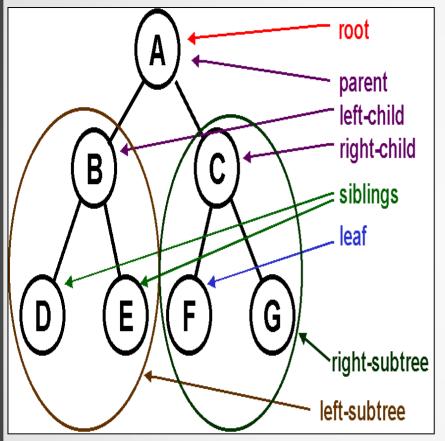
QUEUE DATA STRUCTURE

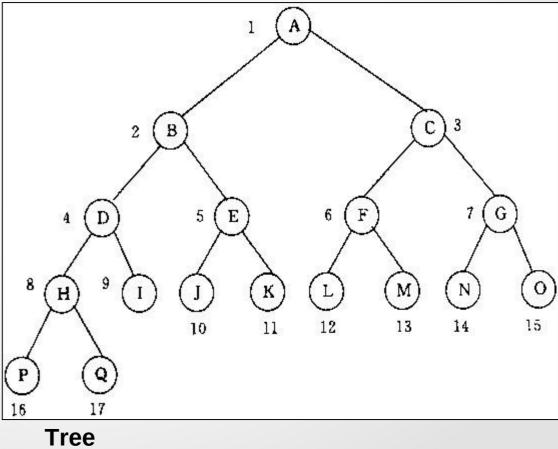**Queue**

Non-Linear: TREE

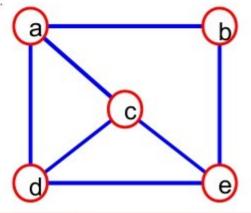
Tree

Non-Linear: GRAPH

What is a Graph?

- A graph G = (V,E) is composed of:
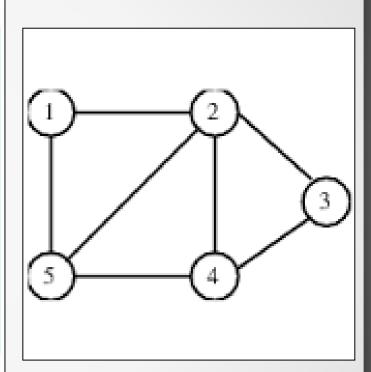
  V: set of **vertices**

  E: set of **edges** connecting the **vertices** in V

- An **edge** e = (u,v) is a pair of **vertices**

- Example:

V= {a,b,c,d,e}

E= {(a,b),(a,c), (a,d), (b,e),(c,d),(c,e), (d,e)}

Graph

Arrays

- **An array is a finite ordered collection of homogenous data elements that provides direct access to any of its elements.**
  - **Finite:** number of elements of an array
  - **Ordered Collection:** arrangement of all the elements in an array is specific.
  - **Homogeneous:** All the elements of an array should be of the same data type.
  - **Size of an array:** The maximum number of elements that would be stored in an array is the size of that array. It is also the length of that array.
  - **Base:** The base address of an array is the memory location where the first element of an array is stored.
  - **Datatype of an array:** The data type of an arrya indicates the data type of elements stored in that array.
  - **Index:** A user can access the elements of an array by index/subscript like a[0], a[1]....

Arrays

✗ **Synatx:** type arrayName [arraySize];

✗ **Example:** double salary[15000];

✗ **Initializing an Array:**
    int age[5]={22,25,30,32,35};

                                                OR

    int newArray[5];
    int n = 0;
    // Initializing elements of array seperately
    for(n=0;n<sizeof(newArray);n++)
    {
      newArray[n] = n;
    }

Two-Dimensional Arrays:

✄The simplest form of the multidimensional array is the two-dimensional array.
✄It is a list of one-dimensional arrays.
✄To declare a two-dimensional integer array of size x,y :
   ✄type arrayName [x][y];
   ✄A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array a, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Initializing Two-Dimensional Arrays

✂Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
 };
```
                                OR
```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Sparse Matrix

✂A matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a mXn matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

✂**"Sparse matrix is a matrix which contains very few non-zero elements".**

✂**"Sparse matrix is two-dimensional array in which most of the elements are zero".**

✂**Implementated two ways**
✂**Array**
✂**Linked List**

Sparse Matrix

- **Array representation of a sparse matrix**

  - 2D array can be represented using three rows. First row indicating row of non-zero element, second row representing column of non-zero element and third row depicting the non-zero value.

- **Example**

$$\begin{bmatrix} 0 & 0 & 5 & 2 \\ 0 & 0 & 0 & 6 \\ 9 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 \end{bmatrix}$$

Row	0	0	1	2	3
Column	2	3	3	0	0
Value	5	2	6	9	7

Sparse Matrix

- **Array representation of a sparse matrix**

  - 2D array can be represented using three rows. First row indicating row of non-zero element, second row representing column of non-zero element and third row depicting the non-zero value.

- **Example**

$$\begin{bmatrix} 0 & 0 & 5 & 2 \\ 0 & 0 & 0 & 6 \\ 9 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 \end{bmatrix}$$
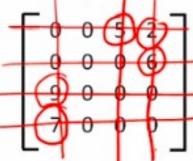
Row	0	0	1	2	3
Column	2	3	3	0	0
Value	5	2	6	9	7

Row	Column	Value
0	2	5
0	3	2
1	3	6
2	0	9
3	0	7

Sparse Matrix

- **Assignment 1: Write a program to check whether the given matrix is sparse matrix or not.**
- Hint: Number of ZEROs should be more than half of the total elements in the matrix

```
int isSparse(int A[row][col])
{
int i,j, count=0;
for(i=0; i<row; ++i)
{for(j=0; j<col; ++j)
        if(A[i][j]==0)count++;
}
if(count>=(row*col)/2)return count;
else return -1;
}
```

Sparse Matrix

- **Assignment 1: Write a program to check whether the given matrix is sparse matrix or not.**
- Hint: Number of ZEROs should be more than half of the total elements in the matrix

```c
int isSparse(int A[row][col])
{
int i,j, count=0;
for(i=0; i<row; ++i)
{for(j=0; j<col; ++j)
        if(A[i][j]==0)count++;
}
if(count>=(row*col)/2)return count;
else return -1;
}
```

```c
/* sparse matrix */
#include<stdio.h>
#define row 4
#define col 3
int main()
{
    int S[row][col]={0,5,0,0,0,8,0,0,0,1,0,0},status;
    status = isSparse(S);
    if(status==-1)printf("Not a sparse matrix");
    else printf("Sparse matrix: With %d zero elements
from total %d elements", status, row*col);
    return 0;
}
```

Sparse Matrix

- **Types of Sparse Matrix:**
  - Triangular Matrix
    - Lower Matrix
      - Lower Left
      - Lower Right
    - Upper Matrix
      - Upper Left
      - Upper Right
  - Band Matrix
    - Diagonal
      - Identity
      - scalar
    - Tri-diognal

Sparse Matrix: Applications

✂You really cannot represent very large high dimensional matrices (when most of them have zeroes) in memory and do manipulations on them.

✂Computer Graphics: It's a simple play of matrices multiplying themselves.

✂Recommendor Systems: Everytime Google or Amazon recommends you something, be rest-assured, lots of sparse vectors (matrices) were multiplied to yield that result for you.

✂Machine Learning - Again, almost every learning method relies heavily on Sparse vectors and matrices operations.

✂Information Retrieval - An index is nothing but an intelligently crafted Sparse Matrix.

✂There are tens of other use-cases (Social Networks, Maps and Graph based applications) where you will be required to store relationships between n objects, all of them use sparse matrices.