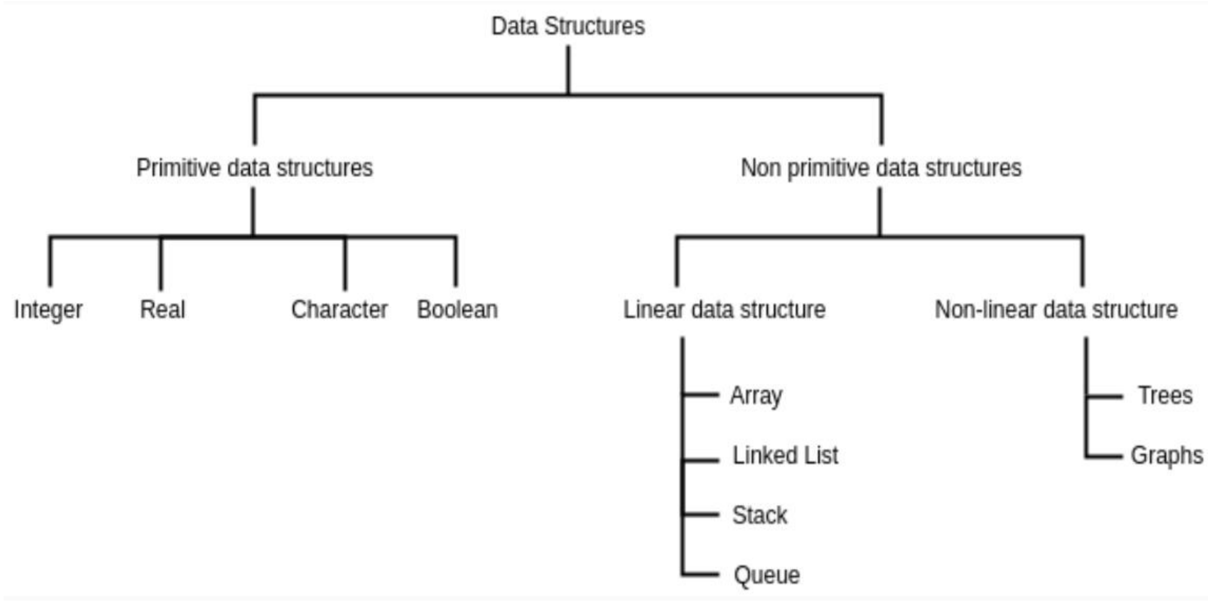
DATA STRUCTURES:

A data structure is an efficient way of storing and organizing the data elements in the computer memory so that it can be used efficiently

APPLICATIONS / CHARACTERISTICS OF DATA STRUCTURES

1. It provides a function that can be used to retrieve the individual data elements.
2. The data structure enables to solve the relationships between the data elements that are relevant to the solution of the problem.
3. Data structure helps to describe the operation that must be performed on the logically related data elements, the operations such as creating, displaying, inserting, deleting, retrieving etc.,
4. Data structure enables to devise the methods of representing the data elements in the memory of the computers, that will reduce the loss of fragmentation and also allows to select the memory configuration or storage structures.
5. Data structures give freedom to the programmer to decide any type of language that best suits for a particular problem.
6. The algorithm can be improved by structuring the data in such a way that the resulting operations can be efficiently carried out.
7. The data structure describes the physical and logical relationship between the data items. It also provides a mode of access to each element in the data structure.
8. Data structure helps in selection of an appropriate mathematical model for writing a program.

CLASSIFICATION OF DATA S TRUCTURES / ELEMENTARY ORGANISATION OF DATA STRUCTURES



PRIMITIVE DATA STRUCTURES

The data structure which can be directly operated by machine level instruction are called primitive data structures. Primitive data structures are the fundamental data types which are supported by a programming language.

In C language, the following are the primitive data structures.

1. Int 2. Float 3. Char 4. Double

Examples:

1. Int age = 10;
2. Float avg = 79.99;
3. Char gender ='M';
4. Double sal =3456789.3

OPERATIONS ON PRIMITIVE DATA STRUCTURES

1. **Creation:** This operation creates a data structure.
   Example: int I;
   It results in creation of memory space for 'I' during the compilation of declaration statement**.**
2. **Deletion:** It is used to destroy the data structure which is created. It leads to the efficient use of memory.
3. **Selection:** It is for accessing of data within a data structure. It depends on type of data structure being used.
4. **Update:** This operation is used to change data in the structure. An assignment operation is a good example of an update operation.
   e.g.: int i =0;
   i=5;

NON-PRIMITIVE DATA STRUCTURES

Non-primitive data structure are those data structures which are created using primitive data structures.

Example of non-primitive data structures: Arrays, stacks, linked lists, queues, trees and graphs.

Non-primitive data structures are classified into two categories

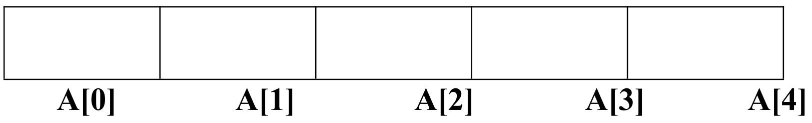
1. Linear Data Structure 2. Non-Linear Data Structure

LINEAR DATA STRUCTURES

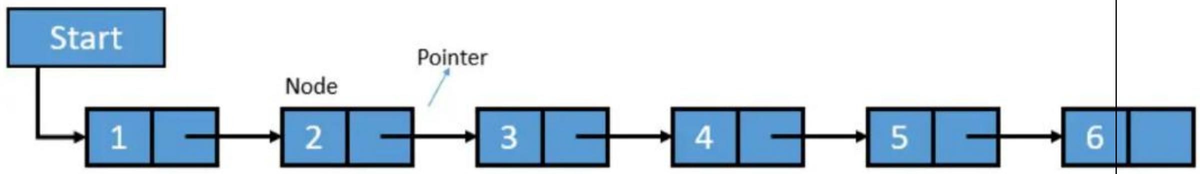
A Linear data structure is the one which establishes the relationship of adjacency between the elements, which means all the elements are stored in memory linearly or sequentially.

Examples: 1. Arrays 2. Linked List 3. Stack 4. Queues

1. **Arrays:** An array is a collection of homogeneous type of data elements in contiguous memory. An array is a linear data structure because all elements of an array are stored in linear order.
   Example: int A[5];

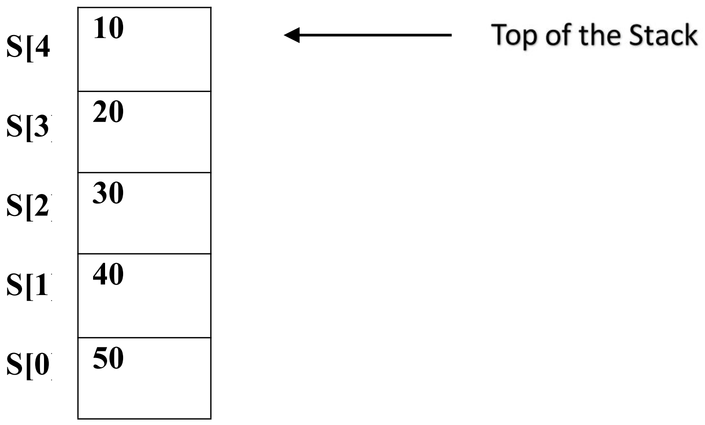


2. **Linked Lists:** Linked List is a sequence of nodes in which each node contains one or more data field and a pointer which points to the next node. Also, linked lists are dynamic, that is memory is allocated as and when required.
   In Linked list, every node contains the following two types of data:
   1) **Information Field / Data Field:** It contains the value of the node.
   2) **Link Field:** A pointer or link to the next node in the list.

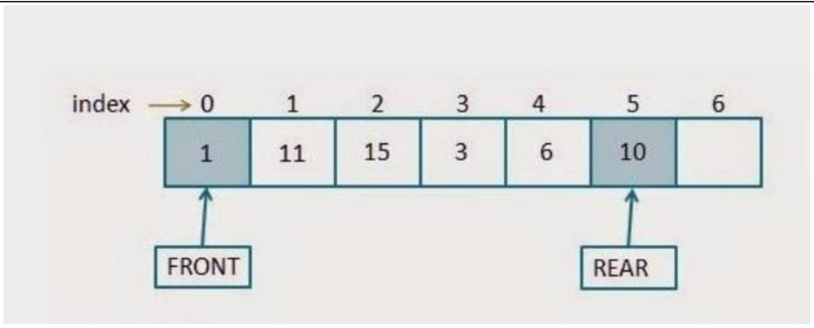


Simple Linked List

3. **STACKS:**
   Stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack. Stack follows Last in First Out(LIFO) model, because the last element which is added to the stack is the first element which is deleted from the stack.



4. **QUEUES:** A Queue is a linear collection of data elements in which the element inserted first will be the element that is taken out first, Queue follows FIFO model(First In First Out). Queue is a popular linear data structure in which element is inserted from one end called REAR end and the deletion of the element takes place from the other end called FRONT end.

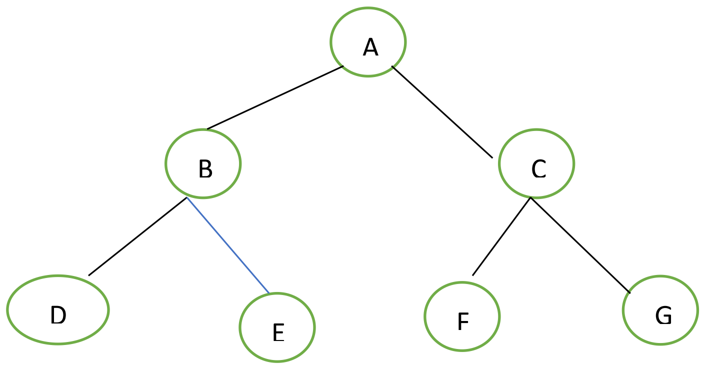


NON-LINEAR DATA STRUCTURE:

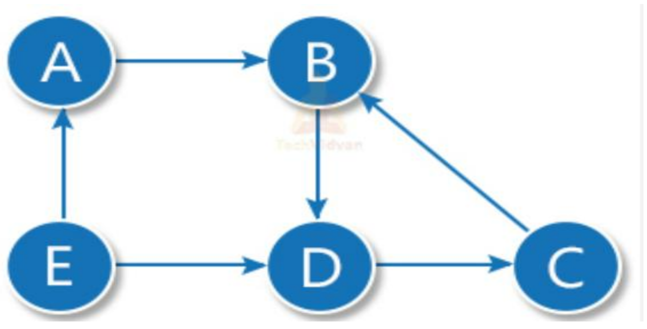
Any data structure which establishes the relationships other than the adjacent relationship is called non-linear data structure. If the elements of a data structure are not stored in a sequential order then it is non- linear data structure.

Example: 1. Trees 2. Graphs

TREES: A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order. One of the node is designated as root node and the remaining nodes can be the sub-tree of the root.



GRAPHS: A graph is a general tree with no parent-child relationship. It is a non-linear data structure which consists of vertices called nodes and the edges which connect those vertices to one another.



V(G)={A, B, C, D,E} E(G)={AB,AE,ED, BD, DC, BC}

OPERATIONS ON DATA STRUCTURE / NON-PRIMITIVE DATA STRUCTURE

Data appearing in the data structure are processed by the operations, the operations on data structures are classified as

1. INSERTING: Adding the new element to the existing data structure.
   Example: To add the details of a new student who has recently joined the course
2. Deleting: Deleting the element from a data structure
   Example: To delete the name of a student who has left the course
3. Traversing: Process of accessing each data item exactly once so that it can be processed.
   Example: To print the names of all the students in a class.
4. SEARCHING: Search a particular element in the list of elements
   Example: To find the names of all the students who secured 100 marks in English.
5. SORTING: Arranging the elements in ascending or descending order
   Example: Arranging the student name in alphabetical order.
6. MERGING: Combining the two different sorted lists into single sorted list.

ABSTRACT DATA TYPE: It is a specification of set of data and the set of operations that can be performed on the data. It is organised in a way that specification and operations are separated from representation of values and implementation of the operations.

ALGORITHM:

An algorithm is a well-defined sequential computational technique that accepts a value or a collection of values as input and produces the output(s) needed to solve a problem

NEED OF ALGORITHM / CHARACTERISTICS OF ALGORITHM

- Efficiency: Algorithms can perform tasks quickly and accurately, making them an essential tool for tasks that require a lot of calculations or data processing.
- Consistency: Algorithms are repeatable and produce consistent results every time they are executed. This is important when dealing with large amounts of data or complex processes.
- Scalability: Algorithms can be scaled up to handle large datasets or complex problems, which makes them useful for applications that require processing large volumes of data.
- Automation: Algorithms can automate repetitive tasks, reducing the need for human intervention and freeing up time for other tasks.
- Standardization: Algorithms can be standardized and shared among different teams or organizations, making it easier for people to collaborate and share knowledge.

COMPLEXITY OF ALGORITHM

Complexity in algorithms refers to the amount of resources (such as time or memory) required to solve a problem or perform a task. The most common measure of complexity are time complexity and space complexity

TIME COMPLEXITY:

The amount of time required by an algorithm to complete its execution is called time complexity.

SPACE COMPLEXITY:

The amount of memory space required by an algorithm to complete its execution is called space complexity.

TIME COMPLEXITY

Time Complexity: The amount of time required by an algorithm to complete its execution is called time complexity.

Three classification of time complexity,

a) Best Case Time complexity: If an algorithm requires minimum amount of time for its execution is called as best-case Time Complexity.
b) Worst Case Time Complexity: If an algorithm requires maximum amount of time for its execution, it is called as Worst-case Time Complexity
c) Average Case Time Complexity: If an algorithm requires average amount of time for its execution, it is called as Average Case Time complexity.

Time complexity is measured in two ways,

1. Frequency count or Step Count
2. Asymptotic Notation

FREQUENCY COUNT METHOD:

In this method, we count the number of times each instruction is executed. Based on that we will calculate the Time Complexity.

RULES:

1. For comment line and declaration statement step count is 0.
2. For Assignment statement and return statement step count is 1
3. Ignore lower order exponent when higher order exponent is present
4. Ignore constant multiplier

Example 1:

```
Sum (int a[], int n)
{
S = 0;                  -------------------→ 1
For(i=0;i<n;i++)      --------------------→ n+1
```

```
S = s +a[i];                -------------------→ n
Return s;                   ------------------→ 1
                                             -------------------
                                                   2n + 3
                                             --------------------
```

T(P) = O(n)

Example 2:

```
Void matadd(int a[][], int b[][])
{
Int c[][];
For(i=0;i<n;i++)                 ----------------→ (n+1)     = n +1
{
        For(j=0;j<n;j++)       ----------------→ n (n+1)  = n^2 + n
              {
                  C[i][j]=a[i][j] + b[i][j];    ----→ n * n    = n^2
              }
}
}
                                                        -------------
                                                   2n^2 + 2n +1  = 2n^2
```

T(P) = O(n2)

ASYMPTOTIC NOTATION

Asymptotic notation is one of the methods, used to measure the time complexity of an algorithm

Types:

1. Big Oh Notation (O)
2. Big Omega Notation (Ω)
3. Big Theta Notation (Θ)

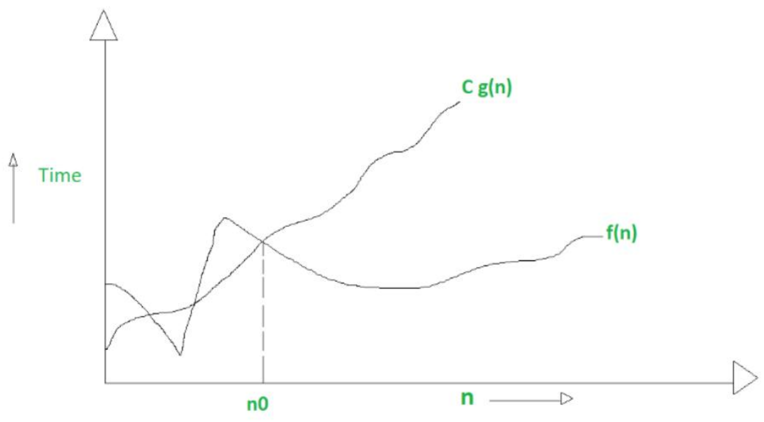
Big Oh Notation:

- It is used to define the upper bound of algorithm in run time
- It always indicates the maximum amount of time for its execution
- Suitable for Worst-Case time complexity.
- It is represented by the symbol ‘O’

Definition: Let f(n), g(n) be two non-negative function, then f(n) = O(g(n)) if there exists two constants C, n such that

$$\boxed{F(n) \le C * g(n) ,\ \text{for all } n \ge n_0}$$

Graphical Notation:



Example:

F(n) = 3n + 2

G(n) = n

According to Big Oh Notation

f(n) = O(g(n))

$f(n) \le C * g(n)$, for all $n > n0$

$3n + 2 \le C * n$

Let us assume C = 4 for

$3n + 2 \le 4n$

If $n_0 = 1$,

$5 \le 4$ (false)

If $n_0 = 2$,

$8 \le 8$

If $n_0 = 3$,

$11 \le 12$

So, $f(n) \le C * g(n)$, for all $n_0 \ge 2$ and C=4

Big Omega Notation:

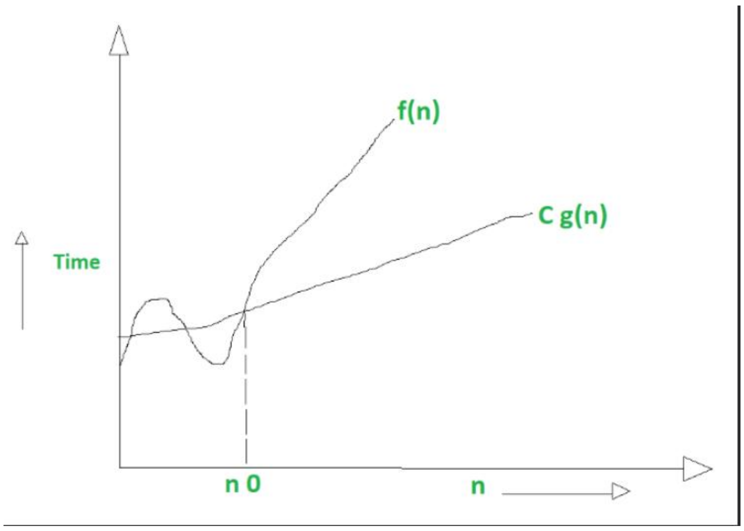
- It is used to define the lower bound of algorithm

- It always indicates the minimum amount of time for its execution
- Suitable for Best-Case time complexity.
- It is represented by the symbol 'Ω'

Definition: Let f(n), g(n) be two non-negative function, then f(n) = Ω (g(n)) if there exists two constants C, n such that

$$\boxed{F(n) \geq C * g(n)\ ,\ \text{for all } n \geq n_0}$$

Graphical Notation:



Example:

F(n) = 3n + 2

G(n) = n

According to Big Omega Notation

f(n) = Ω(g(n))

$f(n) \geq C * g(n)$, for all $n > n0$

$3n + 2 \geq C * n$

Let us assume C = 1

$3n + 2 \geq n$

If $n_0 = 1$

$5 \geq 1$ (True)

If $n_0 = 2$

$8 \geq 2$ (True)

So, $f(n) \leq C * g(n)$, for all $n_0 \geq 1$ and $C = 1$

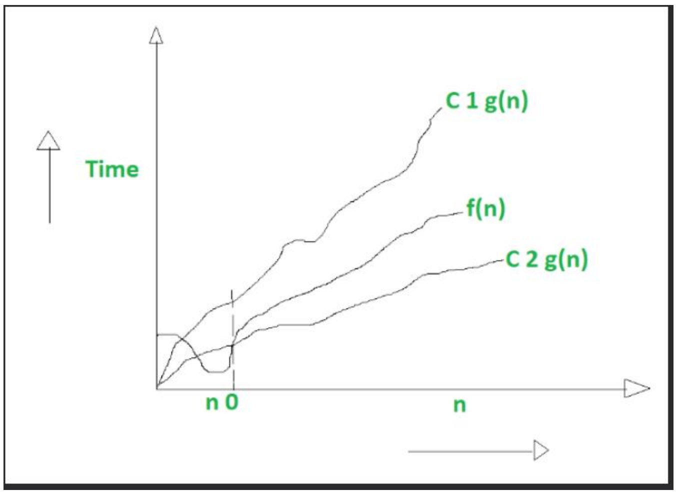
Big Theta Notation (Θ):

- It is used to define the average bound of algorithm in run time
- It always indicates the average amount of time for its execution
- Suitable for average-Case time complexity.
- It is represented by the symbol 'Θ'

Definition: Let f(n), g(n) be two non-negative function, then f(n) = Θ (g(n)) if there exists two constants C, n such that

$$\boxed{C1 * g(n) \leq F(n) \leq C2 * g(n)\ ,\ \text{for all } n \geq n_0}$$

Graphical Representation:



Example:

F(n) = 3n + 2

G(n) = n

According to Big Theta Notation

f(n) = Θ (g(n))

$C1 * g(n) \leq F(n) \leq C2 * g(n)$, for all $n > n_0$

Let us assume C1 = 1, C2 = 4

$1 * n \leq 3n + 2 \leq 4 * n$

$n \leq 3n + 2 \leq 4n$

if $n_0 = 1$

$1 \leq 5 \leq 4$ (False)

If $n_0 = 2$

$2 \leq 8 \leq 8$ (True)

$C1 * g(n) \leq F(n) \leq C2 * g(n)$, for all C1 = 1, C2 = 4 and $n_0 > 2$

SPACE COMPLEXITY

Space Complexity: The amount of memory space required by an algorithm to complete its execution is called space complexity.

$S(P) = C + S_P$

C = Constant Part (or) Independent Part (or) Fixed Part.   Example: int a = 10;

S_p = Instance Part (or) Dependent Part (or) Variable Part.   Example int a[ ];

Formula for finding the space complexity is $S(P) = C + S_P$

Example 1:

```
Sum (int a, int b, int c)
{
a = 10;
b = 20;
c = a + b;
}
```

C (Constant part are a, b, c) = 3 units of memory

$S_P = 0$

$S(P) = C + S_P$

$= 3 + 0$

$= 3$

$S(P) = O(1)$

Example 2:

```
Sum (int a[], int n)
{
Total =0;
For I = 0 to n
Total = total + a[i];
}
```

C (Constant part are n, total, i) = 3 Units of memory

$S_P = 5n$

$S(P) = C + SP$

= 3 + 5n

= 5n

S(P) = O(n)

ARRAYS

DEFINITION OF ARRAYS:

Array is a collection of homogeneous elements, where each element is stored in consecutive memory location.

Length if an array = UB – LB + 1

UB- Upper bound(largest Index)

LB – Lower Bound(smallest Index)

DECLARATION OF ARRAYS:

Syntax:

Datatype arrayname[size];

Datatype – may be int, float, char etc.,

Arrayname – name of the array

Size – number of elements that array can hold

Example:

Int regno[14];

Float average[10];

Char name[40];

ARRAY AS ABSTRACT DATA TYPE(ADT):

An array is a fixed-size sequence of elements of the same type. It is a fundamental abstract data type. The basic operations include direct access to each element in the array by specifying its position so that values can be retrieved from or stored in that position,

REPRESENTATION OF LINEAR ARRAYS IN MEMORY

A single dimensional array is a linear array consisting of related and similar data items. In memory, all the data items are stored in contiguous memory locations one after the other.

Example:

Int value[5]={85, 98, 79, 88, 90};

Memory representation:

Array Name $\longrightarrow$ **value[0] value[1] value[2] value[3] value[4]**

Data $\longrightarrow$

85	**98**	**79**	**88**	**90**

Address $\longrightarrow$ **1000-1004 1004-1008 1008-1012 1012-1016 1016-1020**

4 bytes 4 bytes 4 bytes 4 bytes 4 bytes

Memory requirement depends on the data items stored and number of items.

In the above example, the type of value is int, and it will take 4 bytes. The starting address of the memory block is 1000, the next element will be in 1004 and so on.

Memory for array = n* sizeof(data type)

n= number of data items

data type=type of data items.

Memory of array value=5 * 4

=20 bytes

We can calculate the address of any element using the below formula

Address(A[i]) = Base(A) + I * size of element

Base(A) = Address of the first array element is base address.

To find the address of 3rd element:

Address(value[3]) = 1000 + 3 * 4

= 1000 +12

= 1012

OPERATIONS N LINEAR ARRAYS

a) Traversal: Processing each element in the array
b) Search: Finding the location of the element with the given value in the array
c) Insertion: Adding new element into an array
d) Deletion: Removing an element from an array.
e) Sorting: Arranging the element in ascending or descending order
f) Merging: Combining two arrays into single array

TRAVERSING OF LINEAR ARRAY:

Traversing a linear array is moving through or accessing all the elements in the array sequentially.

Algorithm:

TRAVERSE (A, LB, UB, N)

A – Linear Array

N – Number of elements in the array

LB – Lower Bound

UB – Upper Bound

```
STEP 1: Repeat FOR I = LB to UB
                PROCESS to A[i]
        END OF FOR LOOP
STEP 2: EXIT
```

INSERTION OF ELEMENT IN AN ARRAY

Inserting an element means, adding new element in the array. To insert the element at our desired location, all the elements must be moved to next locations to add new element and to keep their order.

ALGORITHM:

INSERT (A, N, LOC, ITEM)

A – Linear Array

N – Number of element in the array

LOC – location

ITEM – Item to be inserted

```
STEP 1: Set I =N - 1
STEP 2: Repeat while ( I >= LOC-1)
        Set A[i+1]=A[I]
        Set I =I – 1
END of While Loop
STEP 3: Set A[LOC] = ITEM
STEP 4: Set N = N + 1
```

STEP 5: Exit

DELETION OF ELEMENT IN AN ARRAY:

Deleting means removing an element from an array. To delete the element from the desired location, then the subsequent elements must be moved one location left to keep the order of array.

ALGORITHM:

DELETE (A, N, LOC, ITEM)

A – Linear Array

N – Number of elements in the array

LOC – location

ITEM – Item to be deleted

```
STEP 1: Set ITEM = A[LOC]
STEP 2: Repeat For I = LOC - 1 to N
STEP 3: Set A[I] = A[I +1]
          END of FOR Loop
STEP 4: Set N=N-1
STEP 5: Exit
```

MULTIDIMENSIONAL ARRAYS

A multi-dimensional array is an array with more than one level or dimension. For example, a 2D array, or two-dimensional array, is an array of arrays, meaning it is a matrix of rows and columns (think of a table). A 3D array adds another dimension, turning it into an array of arrays of arrays.

HOW TO DECLARE A MULTIDIMENSIONAL ARRAY

SYNTAX: data type array_name[d1][d2][d3][d4]……[dn];

Data type – may be int, float, char, double etc.,

Array_name – name of the array

d1, d2, d3….dn – size of dimensions

Example: int table[5][5][10];

Float A[3][4][4][2];

Size of the array table contains 5 * 5 * 10 = 250 elements

Size of the array A contains 3 * 4 * 4 * 2 =144 elements

DEFINTION OF TWO-DIMENSIONAL ARRAY:

A two-dimensional m x n array A is a collection of m, n elements such that each element is specified by pair of integers (I and j) called subscripts with the property that

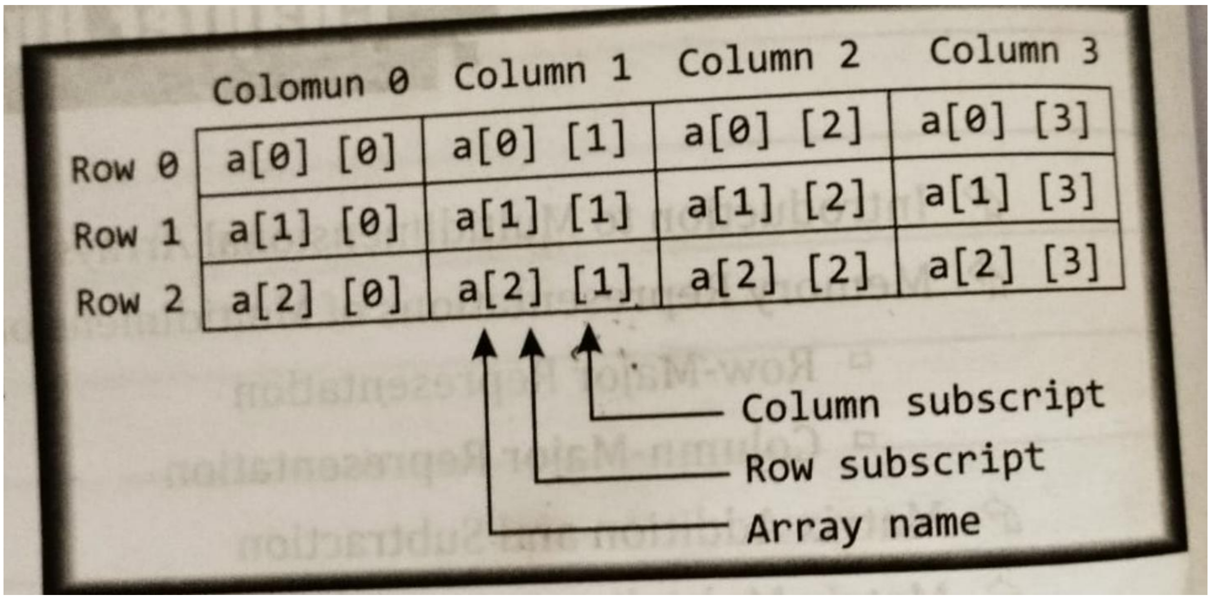
0 <= I <= m and 0 <= j <=n

The elements of an array A with subscripts I and j are denoted by a[I][j]

Example:

Int A[3][4];

Integer array A contains 3 rows and 4 columns



	Colomun 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]

MEMORY REPRESENTATIONS OF MULTIDIMENSIONAL ARRAYS

Multidimensional arrays are stored in the memory in the following two ways:

1. Row-Major Representation
2. Column Major Representation

ROW-MAJOR REPRESENTATION:

In this representation, the arrays are stored in the memory in terms of row design. First the first row of the array is stored in the memory then the second row and so on.

Example: int A[3][3];

A[0][0]	A[0][1]	A[0[2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]

A[3][3] = {1, 2, 3,

4, 5, 6,

7, 8, 9}

It will be represented in memory with row major representation as follows,

1	2	3	4	5	6	7	8	9

Location[I,j] = Base Address(A) + {(I * colsize) + j} * word size

COLUMN – MAJOR REPRESENTATION:

In this representation, the arrays are stored in the memory in terms of the column design. First the first column of the array is stored in the memory then the second column and so on.

A[3][3] = {1, 2, 3,

4, 5, 6,

7, 8, 9}

It will be represented in memory with row major representation as follows,

1	4	7	2	5	8	3	6	9

Location[I,j] = Base Address(A) + {(I * Rowsize) + j} * word size

MATRIX ADDITION:

ALGORITHM:

MATRIX ADDITION(A,B,M,N,X,Y)

A – Two dimensional array with M rows and N columns

B – Two dimensional array with X rows and Y columns

```
STEP 1: If (M ≠ X) or (N ≠ Y) Then
STEP 2:         Print: Addition is not Possible
STEP 3: EXIT
            [End of IF]
STEP 4: Repeat For I = 1 to M
STEP 5: Repeat For J = 1 to N
STEP 6:       Set c[i][j] = A[i] [j] + B[i] [j]
           [End of For loop I]
```

```
        [End of For loop J]
STEP 7: EXIT
```

MATRIX SUBTRACTION:

ALGORITHM:

MATRIX SUBTRACTION(A,B,M,N,X,Y)

A – Two-dimensional array with M rows and N columns

B – Two-dimensional array with X rows and Y columns

```
STEP 1: If (M ≠ X) or (N ≠ Y) Then
STEP 2:         Print: Addition is not Possible
STEP 3: EXIT
         [End of IF]
STEP 4: Repeat For I = 1 to M
STEP 5: Repeat For J = 1 to N
STEP 6:     Set c[i][j] = A[i] [j] - B[i] [j]
        [End of For loop I]
        [End of For loop J]
STEP 7: EXIT
```

MATRIX MULTIPLICATION:

ALGORITHM:

MATRIX MULTIPLICATION (A,B,M,N,X,Y)

A – Two-dimensional array with M rows and N columns

B – Two-dimensional array with X rows and Y columns

```
STEP 1: If N ≠ X Then
STEP 2:        Print: Multiplication is not possible
STEP 3: Else
STEP 4: Repeat For I = 1 To N
STEP 5: Repeat For J = 1 To X
STEP 6:         Set c[i][j] =0
STEP 7:             Repeat For K = 1 to Y
STEP 8:                 Set C[I] [J] = c[I] [J] + A[I] [K] * B[k] [J]
                    End of For loop K
               [End of For loop J]
           [End of For loop I]
         [End of If)
STEP 9: EXIT
```

SPARSE MATRIX

Matrices which contain high number of zero entries are called sparse matrix. A matrix which contains more zero elements than non-zero elements are referred as sparse matrix.

Example: int A[6][5]

$$A = \begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 2 & 9 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 2 & 0 & 0 \\ 4 & 0 & 8 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \end{array} \xrightarrow{} \text{Column numbers}$$

$\downarrow$ **Row numbers**

If we represent the above matrix in memory, it requires 30 * 4 = 120 bytes,

The above matrix can be represented as

$$\text{Sparse (A)} = \begin{array}{ccc} \text{Row Number(i)} & \text{Column Number(j)} & \text{Value} \\ 1 & 2 & 1 \\ 2 & 0 & 9 \\ 3 & 2 & 2 \\ 4 & 1 & 8 \end{array}$$

The above matrix contains 12 elements and the memory requirement is 48 bytes. This representation saved around 72 bytes of memory.