Data Structure

- Data Structure refers to the organisation of data in computer memory.
- It is the way in which data is efficiently stored, processed and retrieved.

Definition

(1) A data structure is the logical or mathematical model of a particular organization of data.

(2) A data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

The choice of a particular data model depends on two considerations :

(1) The data structured must be rich enough in structure to reflect the relationship existing between the data.

(2) The data structures should be simple so that we can process data effectively whenever required.

Need of a Data Structure

- A data structure helps us to understand the relationship of one element with the other and organise it within the memory.
- The study of data structures includes :->

> Logical description of data structure

> Implementation of data structure

> Quantitative analysis of data structure which includes determining the amount of memory needed to store the data structure and the time required to process it.

Areas in which data structures are applied extensively are :->

- Database Management System
- Compiler Deligh
- Network Analysis
- Artificial Intelligence
- Operating System
- Graphics

Example: RDBMS → Array

Network data model → Graph

Hierarchical data model → Trees

Classification of Data Structure

Data structures are divided into two categories:

(1) Primitive data structures

(2) Non-primitive data structures

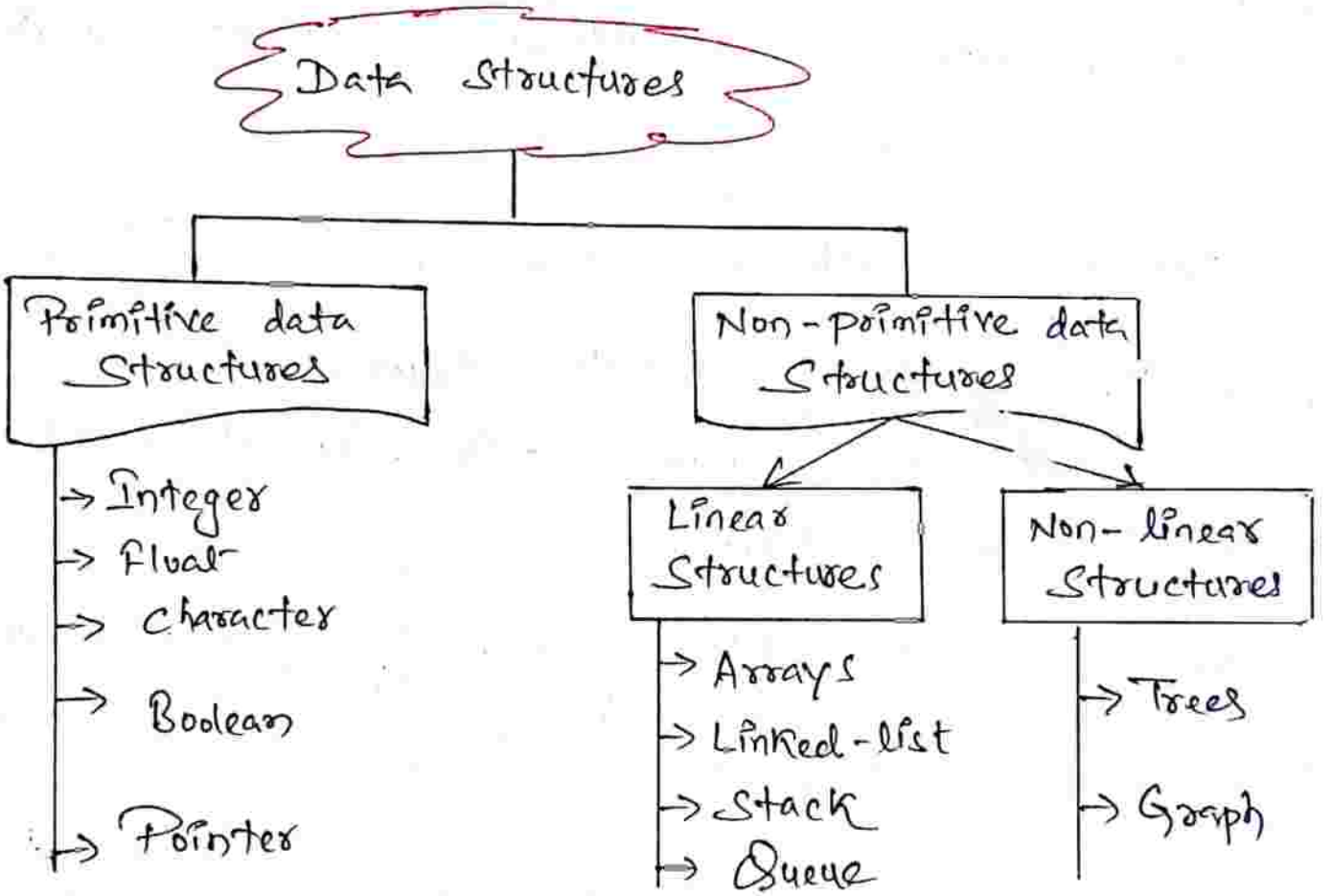


figure: classification of Data Structures

(1) Primitive Data Structures

Primitive data structures are the fundamental data types which are supported by a programming language.

Example: Integer, Float, character, Boolean, Pointer

(2) Non - Primitive Data Structures

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

Non - primitive data structures are further divided into linear and non- linear.

(2.1) Linear Data Structure

If the elements of a data structure are stored in a linear or sequential order, then it is called a linear data structures.

Example: Array, Linked-list, stack and Queue.

(2.2) Non - Linear Data Structure

If the elements of a data structure are not stored in a sequential order, then it is called non-linear data structures.

Example: Trees and Graphs

Operations on Data Structures

(1) Traversing :->

It means to access each data item exactly once so that it can be processed.

Example: to print names of all students in a class.

(2) Searching :->

It is used to find the location of one or more data items that satisfy the given constraint. Such demeut may or may not be present.

Example: to print names of all students who secured 100 marks in mathematics.

(3) Inserting :->

It is used to add new data items to the given list.

Example; to add the details of a new student who has recently joined the course.

(4) Deleting :->

It is used to remove (delete) a particular data item from the list.

(5) Sorting :→

Arranging the data elements in some logical order ie in ascending and descending order.

Example, sorting an array elements.

(6) Merging :→

Combine the data elements in two different sorted sets into a single sorted set.

NOTE

Data Type = Permitted Data values + operations

Data structure = Organized Data + Allowed Operations

Algorithm

Definition :- Set of instructions which can perform a specific task is known as algorithm.

Another definition is :- Step-by-step procedure to solve a particular problem is known as algorithm.

An algorithm should have following properties

(1) Finiteness :- An algorithm should terminates after finite number of steps.

(2) Definiteness :- Each step of the algorithm is precisely defined and unambiguous.

(3) Completeness / Generality :- Algorithm should be complete so that it can solve all the problems of the same type for which it is being designed.

(4) Effectiveness :- All the operations used in the algorithm are basic and feasible so that it can be implemented on the computer.

(5) Input :- There are zero or more quantities which are externally supplied.

(6) Output :- At least one quantity is produced. Outputs are generated in the intermediate as well as final steps of the algorithm.

Desirable Attributes for Algorithms

Although an algorithm may satisfy the criteria of precise, unambiguous, deterministic, and finite, it still may not be suitable for use as a computer solution to a problem.

The basic desirable attributes that an algorithm should meet are as follows :=>

(i) Generality :=>

An algorithm should solve a class of problems, not just one problem.

Example :=> Problem is to find average of numbers.

Algorithm 1 :=> finds average of four numbers.

Algorithm 2 :=> finds average of arbitrary numbers.

Here, Algorithm 2 is better than Algorithm 1 as Algorithm 2 is general taking any random values.

(ii) Good Structure

A well-structured algorithm should be created using good building blocks that make it easy to :

Explain
Understand
Test
Modify

The blocks, from which the algorithm is constructed, should be interconnected in such a way that one of them can be easily replaced with better line of codes.

(iii) **Efficiency**

An algorithm's speed of operation is important property. Algorithm should be efficient in terms of time and space.

(iv) **Robustness**

Resistance to failure when presented with invalid data.

Example of Algorithm

Algorithm which adds two numbers and store the result in a third variable.

Step 1: Start

Step 2: Read the first number in variable 'num_1'.

Step 3: Read the second number in variable 'num_2'.

Step 4: Perform the addition of num_1 and num_2 and store the result in variable 'sum'.

Step 5: Print the value of 'sum'.

Step 6: Stop

Efficiency of an Algorithm

The efficiency of an algorithm can be decided by measuring the performance of an algorithm.

We can measure the performance of an algorithm by computing 2 factors.

(1) Amount of time required by an algorithm to execute. (Time Complexity).

(2) Amount of space required by an algorithm (Space Complexity).

Types :->

① Worst case Efficiency :-> It is the maximum number of steps that an algorithm can take for any collection of data value.

② Best case Efficiency :-> It is the minimum number of steps that an algorithm can take for any collection of data value.

③ Average case Efficiency :-> It is the average number of steps that an algorithm can take for any collection of data value.

Complexity

Complexity of algorithm is a function of size of input of given problem instance which determines how much running time / memory space is needed by the algorithm in order to run completion.

Analysis of algorithm

- Modularity
- Correctness
- Functionality
- Robustness
- User Friendly
- Simplicity
- Extensibity

Time - Space Tradeoff

While developing any software, selection of the appropriate algorithm plays a vital role. An appropriate algorithm is the one that consumes less time and memory.

On the basis of complexity of algorithm we conclude which one is better.

Suppose we want to perform a task named "Operation One". There are two ways by which we can do it

(i) algo 1 (ii) algo 2

algo 1 performs the task within 10 seconds and it requires 100 bytes of memory while algo 2 performs the task within 12 seconds and it needs 110 bytes of memory.

Algorithm	Time (In seconds)	Space (in bytes)
algo 1	10	100
algo 2	12	110

It means that algo 1 is better than algo 2.

Now, Consider two different cases,

Algorithm	Time (In seconds)	Space (in bytes)
algo 1	10	100
algo 2	8	200

For algo 1, less memory is required in comparison with algo 2, while algo 2 is faster than algo 1.

Now Question arises which algo is better.

The answer is simple.
There are two criteria (a) Time and (b) space.
If time matters, select algo 2 and if space matters select algo (1).

Time And Space Complexity

Time Complexity of an algorithm is the running time of a program as a function of input size. ie. the number of machine instructions which a program executes is called its time complexity.

Space Complexity of an algorithm is the amount of computer memory that is required during the program execution as a function of input size.

The space needed by a program depends on the following two parts:

- Fixed Part :-> It varies from problem to problem. It includes the space needed for storing instructions, constants, variables and structures. variables (like array and structures).
- Variable Part :-> It varies from program to program. It includes space needed for recursion stack, and for structured variable that are allocated space dynamically during the runtime of a program.

Worst-Case Time Complexity

* This denotes the behaviour of an algorithm with respect to the worst possible case of input.
* The worst-case running time of an algorithm is an upper bound on the running time for any input.
* Worst-case running time gives an assurance that the algorithm will never go beyond this time limit.

Average-case Time Complexity

* The average-case running time of an algorithm is an estimate of the running time for an 'average' input.
* It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution.
* Average-case running time assumes that all inputs of a given size are equally likely.

Best-Case Time Complexity

* The term 'best-case performance' is used to analyse an algorithm under optimal conditions.

NOTE :=> (1) Best case: Algo produces result in minimum possible time.
(2) Worst case: Algo produces result in maximum possible time.
(3) Average case: Algo produces result in average case time requirement in between minimum and maximum.

Time - Space Trade-Off

The best algorithm to solve a particular problem is one that requires less memory space and takes less time to complete its execution.

But designing such an ideal algorithm is not a trivial task.

One may require less memory space while the other may require less CPU time to execute.

Hence, there exists a time-space trade-off among algorithms.

If space is a big constraint, then one might choose a program that takes less space at the cost of more CPU time.

If time is a major constraint, then one might choose a program that takes minimum time to execute at the cost of more space.

∴ It is basically a situation where either a space efficiency (memory utilization) can be achieved at the cost of time or time efficiency can be achieved at the cost of memory.

Example:→ Suppose in a file, if we store the uncompressed data then reading the data will be an efficient job but if the compressed data is stored then to read such data more time still be required.

executed

Q1. main()
{
1. x = y + z; ⇒ 1
}

how many times the statement is (runs) ⇒ order of magnitude.

only one statement

Ans:- O(1) is time Complexity.

e.g.(2)
```
main()
{
1. x = y+z;          ------> 1
for(i=1; i<=n; i++)
  {
2. x = y+z;          ------> n
  }
}
```

(n+1) = O(n)

→ constant is removed.

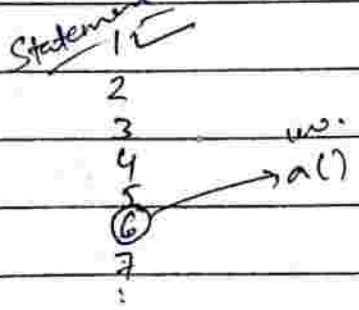
e.g.(3) main ()
{
1. x = y + z; ⟶ 1
for (i = 1; i ≤ n; i++)
{
2. x = y + z; ⟶ n
for (i = 1; i ≤ n; i++)
{
for (j = 1; j ≤ n; j++)
{
3. x = y + z; ⟶ n^2.
}
}
}

$n^2 + n + 1 = O(n^2)$

is larger

i = 1	i = 2	i = 3	...	i = n
j = 1, 2, 3, ... n	j = 1, 2, 3, ... n	j = 1, 2, 3, ... n		j = 1, 2, 3, ... n

temporal locality of references : loop
Spatial " " " : next statement.



cache ← contains next statement to be executed.

Many fun() calls ⇒ more miss

Finding time complexity means find largest loop

e.g (4) Find time Complexity

```
main ()
{
  for (i=1; i≤n; i++)
  {
    for (j=1; j≤n/2; j++)
    {
      1. x=y+z;
    }
  }
}
```

→ $n/2$

→ $n \times \frac{n}{2} = \frac{n^2}{2}$

$= \frac{1}{2} \times n^2$ (constant)

$= O(n^2)$

e.g (5)

```
main ()
{
  for (i=1; i≤n; i++)
  {
    for (j=1; j≤10; j++)
    {
      for (K=1; K≤133; K++)
      {
        1 x=y+z;
      }
    }
  }
}
```

→ 133

→ $133n$

→ $133n^2$ $\simeq O(n^2)$

i=1	i=2	i=3	----	i=n
j=1	j=1,2	j=1,2,3		j=1,2,3,...n
K=133	K=133,133	K=133,133,133		K=133,133,...133

total $= 1 \times 133 + 2 \times 133 + 3 \times 133 + 4 \times 133 + \dots + n \times 133$

$= 133(1+2+3+\dots n)$

Date

$= n^2 + n$

$= O(n^2)$

GATE

```
main()
{
for (i=1; i<=n; i++)
{
for (j=1; j<=i^2; j++)
{
for (k=1; k<=n; k++)
{
x = y + z;
}
}
}
}
```

i=1	i=2	i=3	i=4	...	i=n
j=1	j=1,2,3,4	j=1,2,3,...9	j=1,2,...4²	..	j=1,2,...n²
k=n	k=n,n,n,n	k=n,n,...n	k=n,n,...n	.	k=n,n,...n

total $= 1*n + 2^2*n + 3^2*n + 4^2*n + \cdots + n^2*n$

$= n[1^2 + 2^2 + 3^2 + \cdots + n^2]$

$= n\left[\frac{n(n+1)(2n+1)}{6}\right]$

$= O(n^4)$

Date

GATE

```
main()
{
for(i=1; i<=n; i++)
{
for(i=1; i<=n^2; i++)
{
for(i=1; i<=n^3; i++)
{
x=y+z;
}
}
}
}
```

$n^3+2 \leq n$ false

n^3+2

$n^3+1 \leq n^2$ ⇒ false

n^3+1

$\Rightarrow n^3$

1st for loop, 2nd for loop.

$\therefore 1 * 1 * n^3 = O(n^3)$

```
i=1
j=n^2
k=n^2
main()
{
for(i=1; i<=n; i++)
{
for(k=1; k<=n^2; k++)
{
for(i=1; i<=n^3; i++)
{
x=y+z;
}
}
}
}
```

$1 * n^2 * n^3 = O(n^5)$

Asymptotic Notations

(1) Big - Oh - Notations (O)
(2) Omega - Notations (Ω)
(3) Theta - Notations (θ)

(1) Big-Oh- Notations (O)

Let $f(n)$ and $g(n)$ be two positive functions.

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

(or)

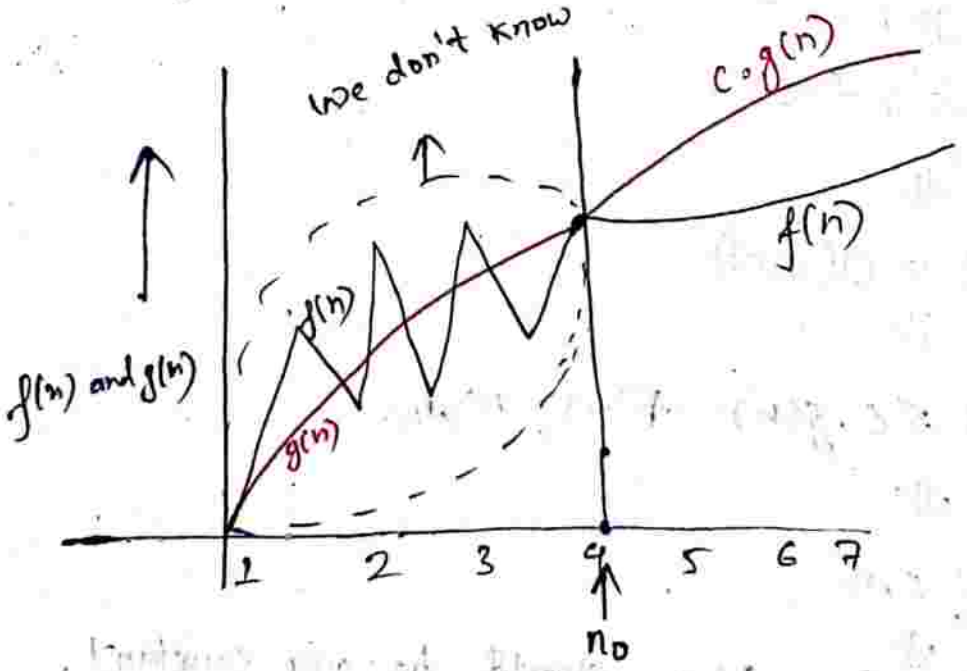
$f(n)$ is order of $g(n)$ → $g(n)$ is upper bound

if and only if

$f(n) \leq c \cdot g(n)$ $\forall n$, $n \geq n_0$

where c is a positive constant and $c > 0$.

n_0 is a positive integer constant and $n_0 \geq 1$



Example (1) $f(n) = n + 10$

$g(n) = n$

$f(n) = O(g(n)) \Rightarrow f(n) \leq c \cdot g(n)$ $\forall n$, $n \geq n_0$

$\therefore n+10 \leq c \cdot n \quad \forall n, n \geq n_0$

$\Downarrow$

$\therefore n+10 = O(n).$

$n+10 \leq 2n$

$10 \leq n$

$n_0 = 10$

Example(2) $f(n) = n$

$g(n) = n+10$

$\Downarrow$

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

$\Downarrow$

$f(n) \leq c \cdot g(n) \quad \forall n \quad n \geq n_0$

$\Downarrow$

$n \leq c \cdot (n+10) \quad \forall n \quad \boxed{n_0 = 1}$

$\Downarrow$

$\therefore n = O(n+10)$

Example(3) $f(n) = n^2$

$g(n) = n$

$\Downarrow$

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

$\Downarrow$

$f(n) \leq c \cdot g(n) \quad \forall n, n \geq n_0$

$\Downarrow$

$n^2 \leq c \cdot n$

$\Downarrow$

n

$\therefore n^2 \neq O(n)$

$\because c$ should be only constant.
It does not support function i/e n.

(2) Omega Notations

$f(n) = \Omega(g(n))$

or

$f(n)$ is omega of $g(n)$ → $g(n)$ is lower bound

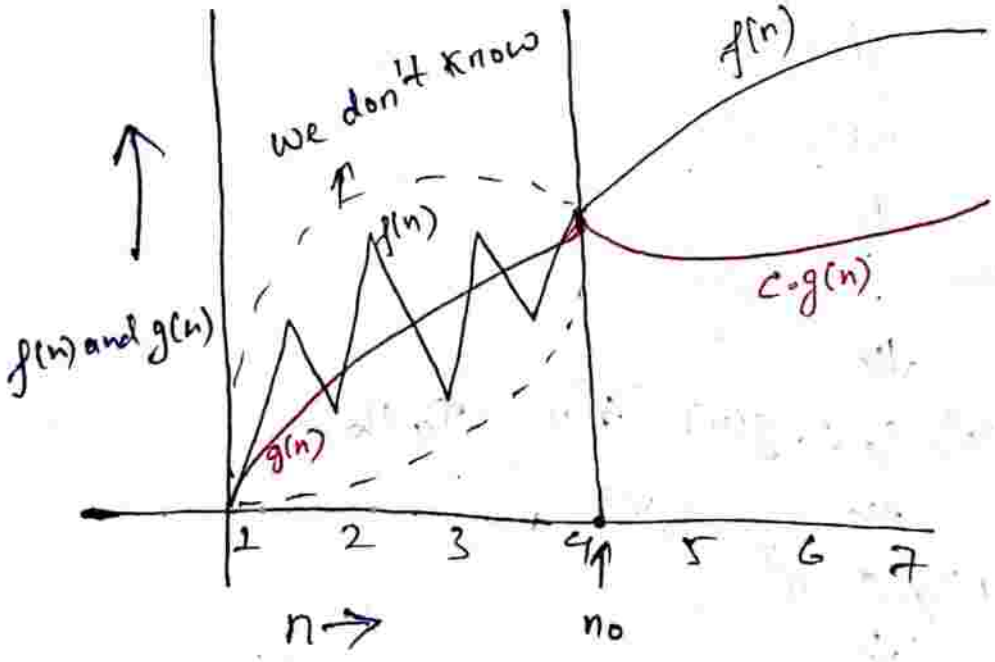
⇓

if and only if

$f(n) \geq c \cdot g(n)$, $\forall n$ $n \geq n_0$

where c and n_0 are two positive constants and

$c > 0$ and $n_0 \geq 1$



Example (1) $f(n) = n$

$g(n) = n+10$

⇓

$f(n) = \Omega(g(n))$

⇓

$f(n) \geq c \cdot g(n)$ $\forall n$ $n \geq n_0$

⇓

$n \geq c \cdot (n+10)$ $\forall n$ $n \geq 10$

⇓

$1/2$ $\therefore n = \Omega(n+10)$

n_0 → from which n value it hold true.

Example (2) $f(n) = n$

$g(n) = n$

$\Downarrow$

$f(n) = \Omega(g(n))$

$\Downarrow$

$f(n) \geq c \cdot g(n) \quad \forall n \quad n \geq n_0$

$\Downarrow$

$n \geq c \cdot n \quad \forall n \quad n \geq 1$

$\Downarrow$

1

$\therefore n = \Omega(n)$

Example (3) $f(n) = n$

$g(n) = n^2$

$\Downarrow$

$f(n) = \Omega(g(n))$

$\Downarrow$

$f(n) \geq c \cdot g(n) \quad \forall n \quad n \geq n_0$

$\Downarrow$

$n \geq c \cdot n^2$

$\Downarrow$

~~$\frac{1}{n}$~~

$\forall n$ is a function.
c only hold constant value.

$\therefore n \neq \Omega(n^2)$

(3) Theta - Notation (Θ)

$f(n) = \Theta(g(n))$

(or)

$f(n)$ is theta of $g(n)$ $\Rightarrow$ $g(n)$ is both upper bound and lower bound.

⇓

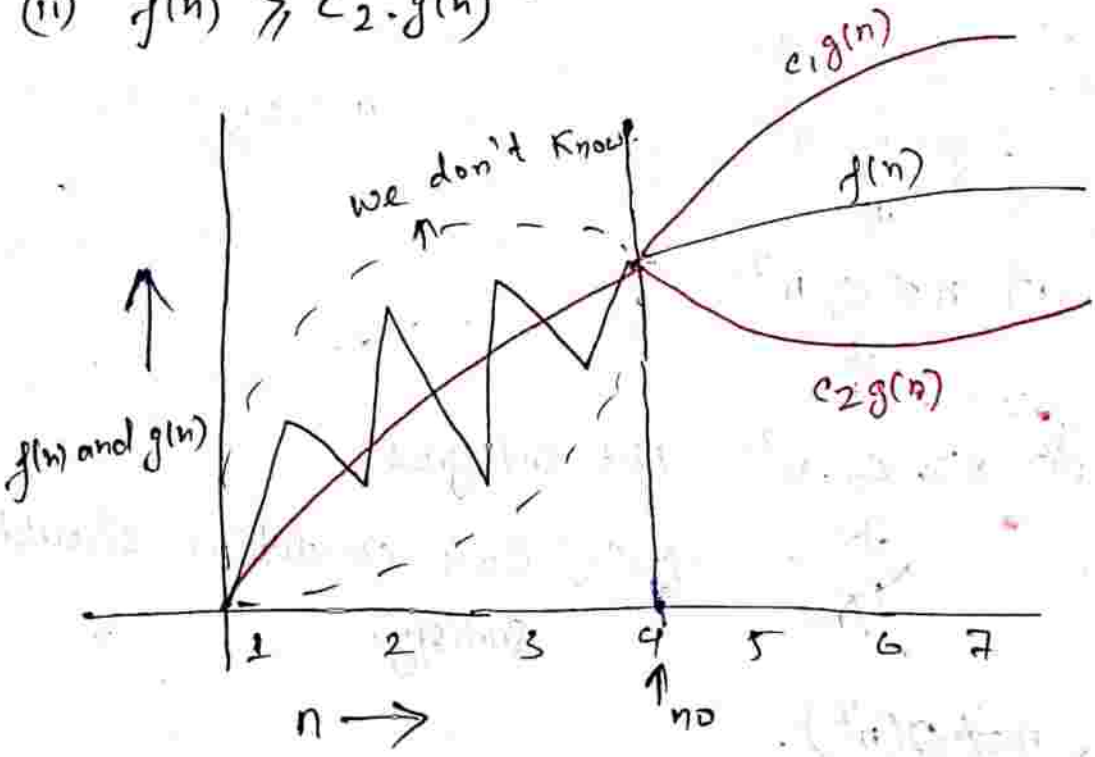
if and only if

(i) $f(n) \leq c_1 g(n)$

and

(ii) $f(n) \geq c_2 \cdot g(n)$

$\forall n, \; n \geq n_0$



Example (1) $f(n) = n$, $g(n) = n+10$ } are equal using asymptotically (i.e. remove const. 10).

⇓

(i) $n \leq c_1 \cdot (n+10)$ ∴ true for $n_0 > 1$

⇓

1

(ii) $n \geq c_2 \cdot (n+10)$ ∴ true for $n_0 > 10$

⇓

$\frac{1}{2}$

$n_0 = 10$

∴ $n = \Theta(n+10)$

Example (2) $f(n)=n$, $g(n)=n$ ⇕ are equal using both mathematically and asymptotically.

(i) $n \leq c_1 \cdot n$

$\downarrow$

1

(ii) $n \geq c_2 \cdot n$ $\quad n_0 = 1$

$\downarrow$

1

$\therefore n = \Theta(n)$

Example (3) $f(n) = n$

$g(n) = n^2$

(i) $n \leq c_1 n^2$

$\downarrow$

1

(ii) $n \geq c_2 \cdot n^2$ Not satisfied

$\downarrow$

1/n

Note: Both conditions should satisfy.

$\therefore n \neq \Theta(n^2)$.

Note:- (1) Upper bound means worst case

(2) lower bound means best case

Types of Time Complexity

1. Constant T.C $\longrightarrow O(1)$
2. Logarithmic T.C $\longrightarrow O(\log n)$
3. Linear T.C $\longrightarrow O(n)$
4. Quadratic T.C $\longrightarrow O(n^2)$
5. Cubic T.C $\longrightarrow O(n^3)$
6. Polynomial T.C $\longrightarrow O(n^K)$
7. Exponential T.C $\longrightarrow O(C^n)$

where K is const. and $K > 0$

C is const. and $C > 1$

Asymptotic Notation

- Asymptotic is defined as a way to describe the behaviour of functions in the limit or without bounds.
- Asymptotic notation is a short hand way to represent Time Complexity.
- Using Asymptotic notation, we can give Time Complexity as "Fastest Possible", "slowest Possible", "Average Time".
- Various notations such as O, Ω, Θ are the asymptotic Notations.

Question on Big-Oh (O) Notations

Q1 Let $f(n) = 3n+2$ $g(n) = n$

Given that $f(n) = O(g(n))$

Calculate c and n_0.

Soln Let $f(n) = 3n+2$ $g(n) = n$

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

$\Rightarrow f(n) \leq c.g(n) \quad \forall n \quad n \geq n_0$

$\Rightarrow 3n+2 \leq c.n$

$\therefore c = 4$ (least upper bound or tightest bound)

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

$\Rightarrow n \geq 2$

Comparing $n \geq 2$ with $n \geq n_0$

$\therefore n_0 = 2$

Q2. Let $f(n) = n^2+5n$ $g(n) = n^2$

Find the value of c and n_0 such that $f(n) = O(g(n))$

Soln

$f(n) = n^2+5n$

$g(n) = n^2$

$\because f(n) = O(g(n))$

$\Rightarrow f(n) \leq c.g(n) \quad \forall n \quad n \geq n_0$

$\Rightarrow n^2+5n \leq c.n^2$

$\therefore \boxed{c = 2}$

$\Rightarrow n^2+5n \leq 2.n^2$

$\Rightarrow n^2 \geq 5n$

$\Rightarrow n \geq 5$

(OR)

n	f(n)	c.g(n)
1	5	2
2	14	8
5	50	50

Comparing $n \geq 5$ with $n \geq n_0$

$\therefore \boxed{n_0 = 5}$

Q3 Let $f(n) = n^2$ $g(n) = n$

check $f(n) = O(g(n))$.

Soln

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

$f(n) \leq c.g(n)$

$n^2 \leq c.n \quad \forall n \quad n \geq n_0$

to fulfill above condition 'c' must be equal to 'n'.

But $c = n$ is not possible. $\because$ c must be constant.

$\therefore n^2$

Q4 For the function defined by

$f(n) = 5n^3 + 6n^2 + 1$

Show that $f(n) = O(n^3)$.

Soln $f(n) = 5n^3 + 6n^2 + 1$

$\therefore f(n) = 5n^3 + 6n^2 + 1 \leq 5n^3 + 6n^3 + 1.n^3 \quad \forall\ n \geq 1$

$\Rightarrow f(n) \leq 12n^3 \quad \forall\ n \geq 1$

here $c = 12$, $n_0 = 1$ and $g(n) = n^3$

$\therefore f(n) = O(n^3)$.

Questions on Omega (Ω) Notation

Q1 Let $f(n) = n$ $g(n) = n+10$

Find c & n_0 if $f(n) = \Omega g(n)$.

Soln $f(n) = \Omega g(n)$

$f(n) \geq c.g(n)$

$n \geq c.(n+10)$

$\therefore \boxed{c = \frac{1}{2}}$ [$\because$ $n \geq c.(n+10)$ $\Rightarrow 2n \geq c(n+10)$ i.e if we multiply 2 on LHS, then LHS becomes greater than R.H.S

Here, c is called Greatest Lower Bound

$\therefore 2n \geq n+10$

$\Rightarrow n \geq 10$

Comparing $n \geq 10$ with $n \geq n_0$

$\therefore \boxed{n_0 = 10}$

Q2. Let $f(n)=n$ $g(n)=n$

Find c and n_0 if $f(n) = \Omega g(n)$

Soln $f(n) = \Omega g(n)$

$\Rightarrow f(n) \geq c.g(n)$

$\Rightarrow n \geq c.n$

$\therefore c = 1$ and $n_0 = 1$ ($\because$ minimum value of $n_0 = 1$).

Q3 Let $f(n) = n$ $g(n) = n^2$

Check $f(n) = \Omega g(n)$

Soln $f(n) = \Omega g(n)$

$\Rightarrow f(n) \geq c.g(n)$

$\Rightarrow n \geq c.n^2$

Here condition hold true for $c = \frac{1}{n}$ but c must be a constant.

$\therefore f(n) = \Omega g(n)$ is false.

Questions on Theta (Θ) Notation

Q1 Let $f(n) = n$ $\quad g(n) = n+10$

Find the value of c_1 and c_2 and n_0 such that-

$$f(n) = \Theta g(n)$$

Soln :-> $\quad f(n) = \Theta g(n)$

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Case (1) $\quad c_1 g(n) \leq f(n)$

$$\Rightarrow c_1 (n+10) \leq n$$

$\therefore c_1 = \frac{1}{2}$ [$\because c_1(n+10) \leq n$
$(n+10) \leq 2n$ i.e. to make RHS greater, multiply RHS by 2.

$$\Rightarrow n+10 \leq 2n$$

$$n \geq 10$$

Comparing $n \geq 10$ with $n \geq n_0$

$$\Rightarrow n_0 = 10$$

Case (2) $\quad f(n) \leq c_2 g(n)$

$$\Rightarrow n \leq c_2 (n+10)$$

$\therefore c_2 = 1$ and $n_0 = 1$ ($\because$ minimum n_0 value $= 1$).

Combing case (1) and case (2)

$\therefore c_1 = \frac{1}{2}$, $c_2 = 1$ and $n_0 = 10$

Q2 Let $f(n) = n$ $g(n) = n$

Given that $f(n) = \Theta g(n)$. Find c_1, c_2 and n_0.

Soln $f(n) = \Theta g(n)$

$\therefore c_1 g(n) \leq f(n) \leq c_2 g(n)$

Case (1) $c_1 g(n) \leq f(n)$

$\Rightarrow c_1 n \leq n$ $\forall n$ $n \geq n_0$

$\therefore c_1 = 1$ and $n_0 = 1$

Case (2) $f(n) \leq c_2 g(n)$

$n \leq c_2 n$ $\forall n$ $n \geq n_0$

$c_2 = 1$ and $n_0 = 1$

$\therefore$ Combining case (1) and case (2)

$c_1 = 1$, $c_2 = 1$, $n_0 = 1$

Q3 Let $f(n) = n$ $g(n) = n^2$

check for $f(n) = \Theta g(n)$.

Soln $c_1 g(n) \leq f(n) \leq c_2 g(n)$

case (1) $\therefore c_1 g(n) \leq f(n)$

$c_1 n^2 \leq n$

$\therefore c_1 \neq \frac{1}{n}$ (not possible)

case (2) $f(n) \leq c_2 g(n)$

$n \leq c_2 \cdot n^2$

$c_2 = 1$

Since case (1) is not satisfied, $\therefore f(n) \neq \Theta g(n)$

Abstract Data Type (ADT)

The abstract data type is a triple of DFA where

D = set of Domains

f = set of functions

A = Axioms (what is to be is mentioned but how it is to be done not mentioned).

ie ADT = Type + function names + Behaviour of each function.

ADT for set

Abstract DataType Set

{

Instances: Set is a collection of integer type of elements.

precondition : None

Operations:

1. Store(): this operation is for storing the integer element in a set.
2. Retrieve(): this operation is for retrieving the desired element from set.
3. Display(): this operation is for displaying the contents of set.

ADT of Array

Abstract data type linear list

{

Instances : Ordered finite collections of zero or more elements.

Operations:

1. IsEmpty( ): -> returns TRUE if Array is Empty.

otherwise return False.

2. size() :→ return the list size (i.e. number of elements in the list).

3. get(index) :→ return index element of list.

4. index (x) :→ return index of first occurence of x in the list.
   else return -1 if x is not in the list.

5. insert(index)

6. delete (index)

7. traverse ()

ABSTRACT DATA TYPES (ADT)

* An abstract data types (ADT) refers to a set of data values and associated operations that are specified accurately, independent of any particular implementation.

* With an ADT, we know that what a specific data type can do but how it actually does it is hidden.

* ADT consists of a set of definitions that allow us to use the functions while hiding the implementation.

For example

ADT of an array are Insert, Delete, traverse, merge, sort.

ADT of a stack are PUSH for insertion, POP for deletion, peep for accessing Top of stack, IsEmpty, IsFull.

ADT of a Queue are Enqueue for insertion, Dequeue for Deletion.

Abstract Data Type Model

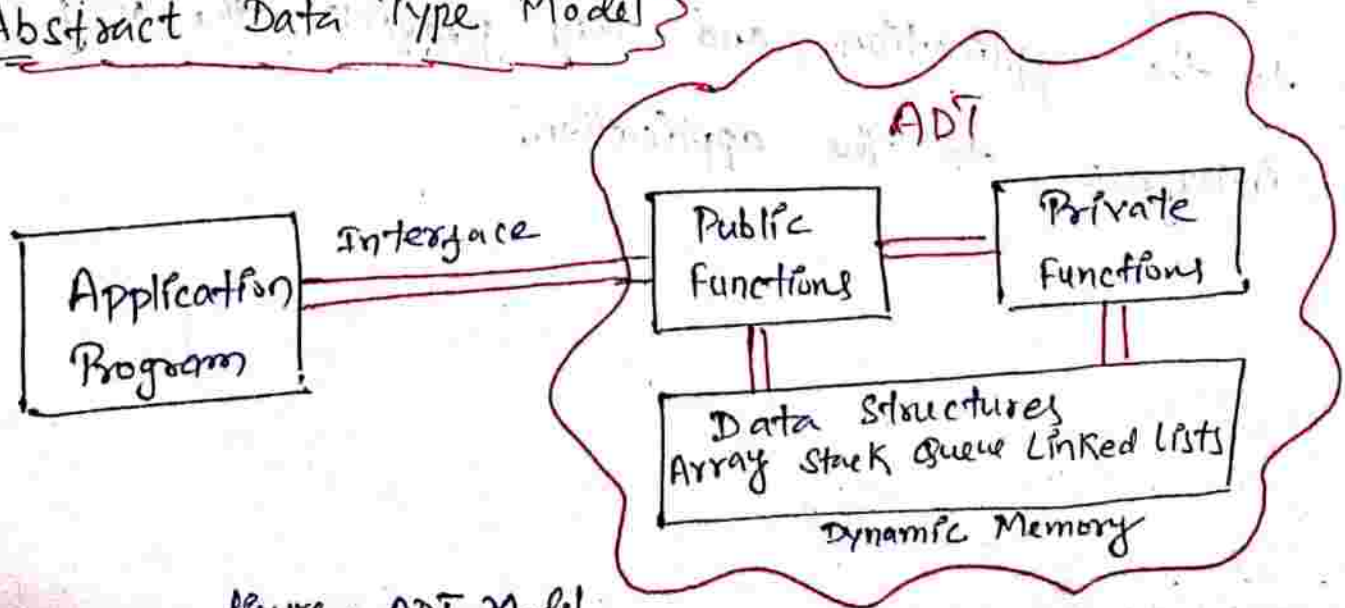


figure. ADT Model

There are two different parts of the ADT modul — functions (public and private) and data structures. Both are contained within the ADT modul itself, and do not come within the scope of the application program.

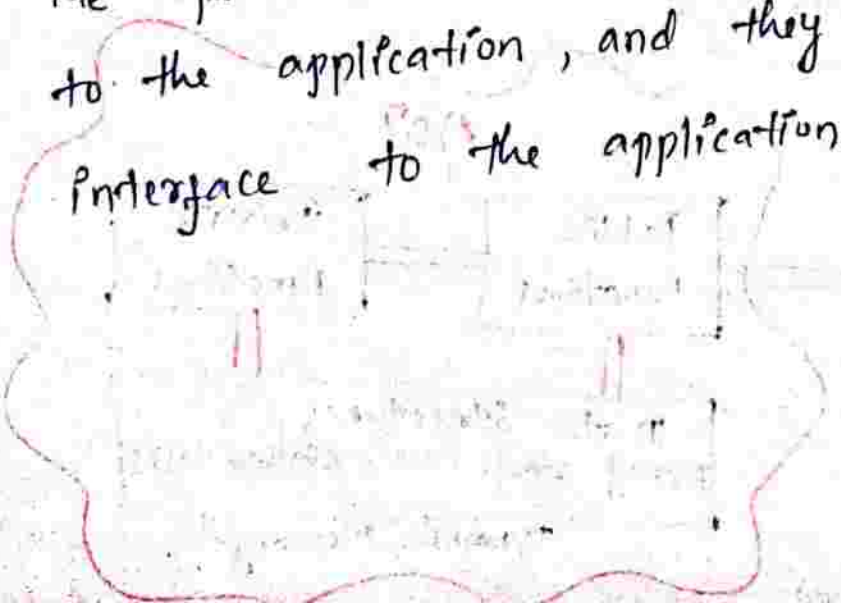
Data structures are available to all of the ADTs functions as required and a function may call on any other function to accomplish its task.

Data structures and functions are within the scope of each other.

Data are entered, accessed, modified and deleted through the external application programming interface. This interface can only access the public functions.

For each ADT operation, there is an algorithm that performs its specific task.

The operation name and parameters are available to the application, and they provide the only interface to the application.



Arrays

- An array is a collection of similar data elements.
- These data elements have the same data type.
- The elements of the array are stored in consecutive memory locations are called index or subscript.

Declaration of Arrays

Syntax is datatype Arrayname [size];

Note: In C, the array index starts from zero.

example, int marks [10];

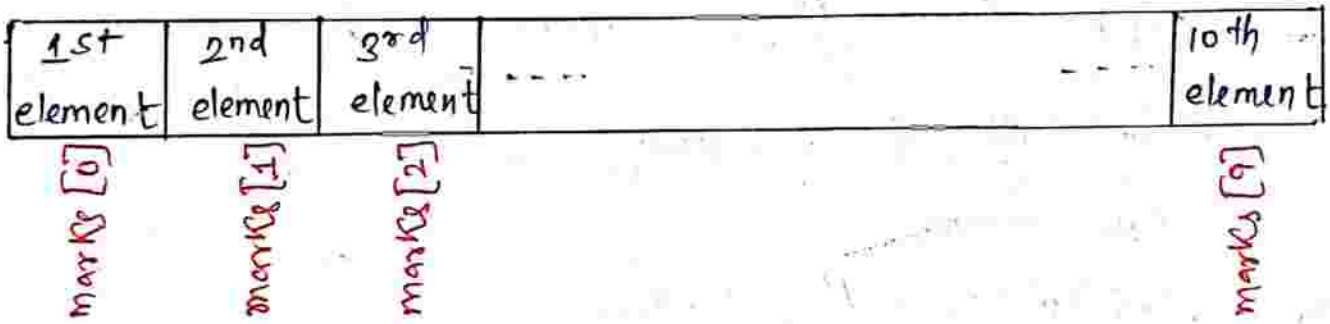


fig. Memory Representation of an array of 10 elements.

NOTE: The size of array is always a constant and not a variable. This is because the fixed amount of memory will be allocated to the array before execution of program.

The method of allocating the memory is known as static allocation of memory.

NOTE: (1) If an element of an array is referenced by single subscript, then the array is known as one-dimensional array. (or) Linear Array.

(2) If an element of an array is referenced by two subscripts., then the array is known as two-dimensional array (or) Matrix

(3) If an element of an array is referenced by more than two subscripts, then the array is known as multi-dimensional arrays.

Example: int a[10]; // 1-D array

int a[10][10]; // 2-D array
↑ Row ↑ Column

int a[10][10][10]; // 3-D array
↑ Table or Page ↑ Row ↑ Column

No. of 2D Arrays (or) Table or Page

NOTE: Array of arrays is 2-D array.

Array of arrays of arrays is known as 3D array.

3D arrays is collection of 2D arrays.

2D arrays is collection of 1D array.

One - Dimensional Arrays

For Reading an array of 'n' elements

```
for (i=0; i<n; i++)
{
    scanf("%d", &a[i]);
}
```

For Printing (or) writing an array

```
for (i=0; i<n; i++)
{
    printf("%d", a[i]);
}
```

Address Calculation in 1D Array

Array A [lb ... ub]

Base Address = BA

size of each element = w

$\therefore$ Address (A[i]) = BA + (i - lb) * w

Q1. For an array declared as int arr[50], calculate the address of arr[35], if Base Address (arr) = 1000, and w = 2.

Solution:-> Address (arr[35]) = 1000 + (35-0) * 2

= 1000 + 70

= 1070

Q2. Let the base address of the first element of the array is 400 and each element of the array occupies 2 bytes in the memory, then find the address of 4th element in the array A[10].

Solution :=>

$$\text{Address}(A[i]) = BA + (i - lb) * w$$

$$\therefore \text{Address}(A[3]) = 400 + (3-0) * 2$$
$$= 400 + 6$$
$$= 406$$

Q3. A[-25, ... , +75]

Base Address = 1000

Size of each element, $w = 10$.

$$\therefore \text{Address}(A[60]) = 1000 + (60 - (-25)) * 10$$
$$= 1000 + 85 * 10$$
$$= 1850.$$

NOTE: Length of Array = UB - LB + 1.

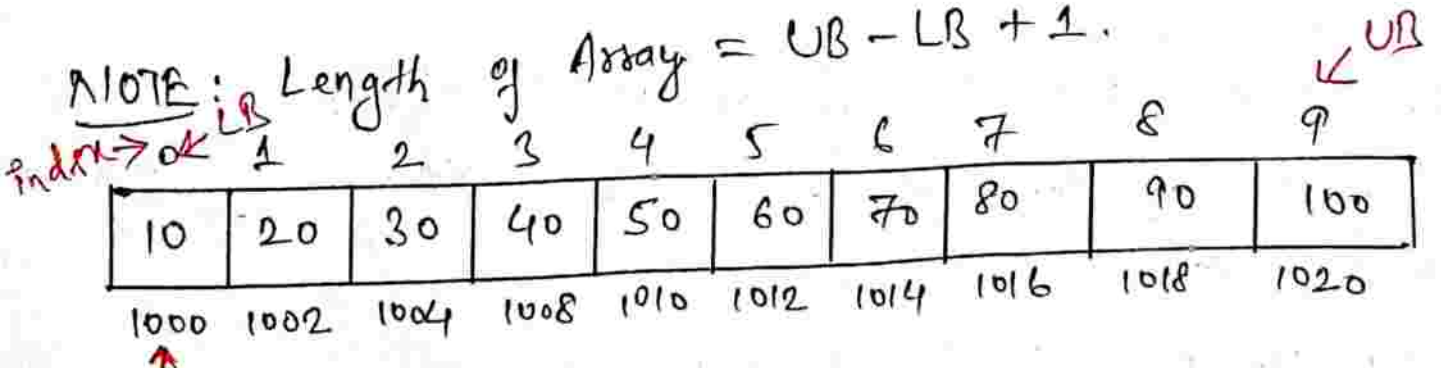


fig. Memory Representation of int A[10].

$$\therefore \text{Length} = (9-0) + 1 \quad \text{(+1 to include 10th element)}$$
$$= 10$$

Two Dimensional Arrays

Two-dimensional array is an array of one-dimensional arrays.

Declaration Syntax

datatype arrayname [row-size][column-size];

e.g. int a[3][5];

Rows / Columns	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

- 2D arrays are also known as Matrix.
- 2D array is treated as a collection of 1D arrays.
- 2D array is stored in memory in sequential order.
- There are two ways of storing 2D array in the memory:
  - (i) Row-Major Order
  - (ii) Column-Major Order

Row-Major Order

The elements of array are stored row-by-row.

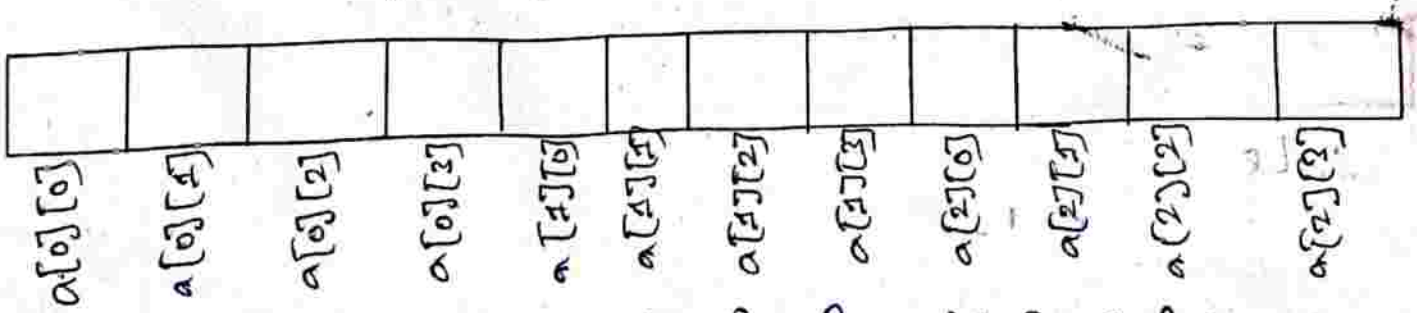


Fig. Elements of 3×4 2D array in Row-Major Order.

(6)

Column-Major Order

The elements of array are stored column-by-column.

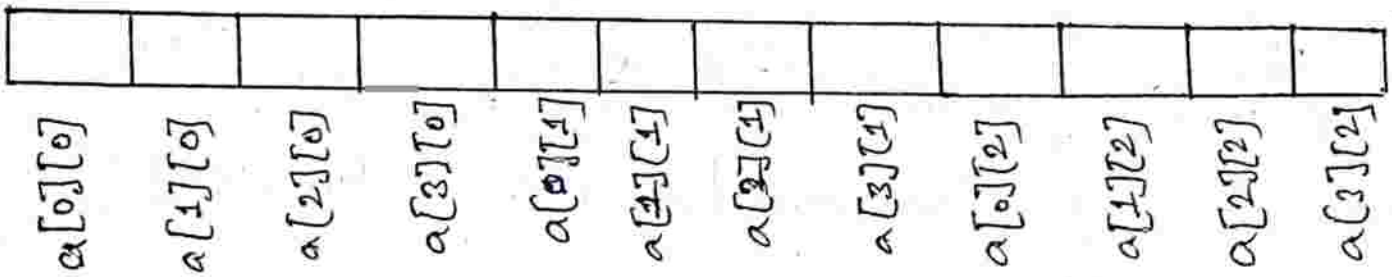


fig. Elements of a 4X3 2D array in column-major order.

Address Calculation in 2D Array

$A[lb_1 \ldots ub_1, lb_2 \ldots ub_2]$

Base Address = BA

Element size = w

No. of Rows (nr) = $ub_1 - lb_1 + 1$

No. of Columns (nc) = $ub_2 - lb_2 + 1$

∴ Row-Major Order is

$$\text{Address}(A[i][j]) = BA + [(i - lb_1) * nc + (j - lb_2)] * w$$

∴ Column-Major Order is

$$\text{Address}(A[i][j]) = BA + [(i - lb_1) + nr * (j - lb_2)] * w$$

2D-Array Address Calculation

- Row-Major Order
- Column-Major Order

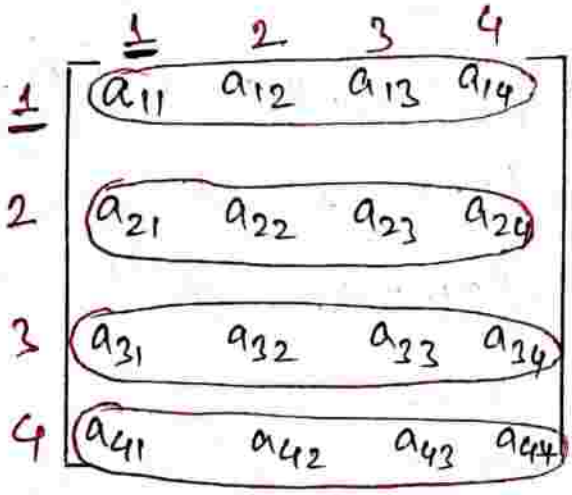
Row-Major Order

int a[4][4]

2 Bytes

$nr = (4-1)+1 = 4$

$nc = (4-1)+1 = 4$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a11	a12	a13	a14	a21	a22	a23	a24	a31	a32	a33	a34	a41	a42	a43	a44
1000	02	04	06	08	10	12	14	16	18	20	22	24	26	28	30

$\therefore$ Loc a[4][3] $= 1000 + [(4-1)*4 + (3-1)]*2$

$= 1000 + (3*4+2)2$

$= 1000 + 14*2 = 1028$

Loc a[2][4] $= 1000 + [(2-1)*4 + (4-1)]*2$

$= 1000 + (4+3)*2$

$= 1000 + 14$

$= 1014$

Example: $A[25 \ldots 79, 55 \ldots 124]$ (Row: 25 ... 79, Column: 55 ... 124)

LB_1 UB_1 LB_2 UB_2

BA = 1000 and w = 10 Bytes

Loc A[55][75] = ?

Soln $nr = UB_1 - LB_1 + 1 = 79 - 25 + 1 = 55$

$nc = UB_2 - LB_2 + 1 = 124 - 55 + 1 = 70$

$\therefore$ Loc A[55][75] $= 1000 + [(55-25) \times 70 + (75-55)] * 10$

$= 1000 + (30 \times 70 + 20) \times 10$

$= 1000 + (2100 + 20) \times 10$

$= 1000 + 2120 \times 10$

$= 22200.$

$\therefore$ General Formula for Address Calculation of 2-D Array using Row-Major.

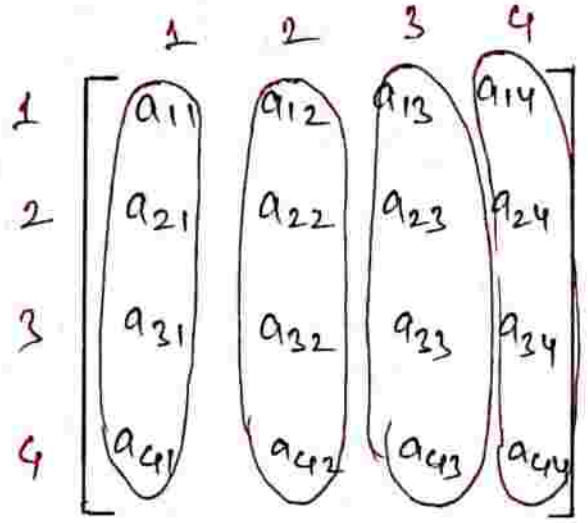
$A[\underbrace{LB_1 \ldots UB_1}_{\text{Rows}}, \underbrace{LB_2 \ldots UB_2}_{\text{Columns}}]$ and Base Address = BA, width = w Bytes

$\therefore nr = UB_1 - LB_1 + 1$ $\quad nc = UB_2 - LB_2 + 1$

$$\text{Loc}(A[i][j]) = BA + [(i - LB_1) \times nc + (j - LB_2)] \times w$$

Column-Major Order

int a[4][4]



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a_{11}	a_{21}	a_{31}	a_{41}	a_{12}	a_{22}	a_{32}	a_{42}	a_{13}	a_{23}	a_{33}	a_{43}	a_{14}	a_{24}	a_{34}	a_{44}
1000	02	04	06	08	10	12	14	16	18	20	22	24	26	28	30

$(LB_1 \ldots UB_1 , LB_2 \ldots UB_2)$

$LB_1 \ldots UB_1$: Rows, $LB_2 \ldots UB_2$: Column

$(1 \ldots 4 , 1 \ldots 4)$

$nr = 4-1+1 = 4$ $\quad nc = 4-1+1 = 4$

i/e $nr = UB_1 - LB_1 + 1$ $\quad nc = UB_2 - LB_2 + 1$

$$Addr(a[3][4]) = 1000 + [(4-1) \times \underset{nr}{4} + (3-1)] \times 2$$

$(4-1)$: no. of columns before a_{34}; $(3-1)$: no. of Rows traversed i/e a_{14} and a_{24}

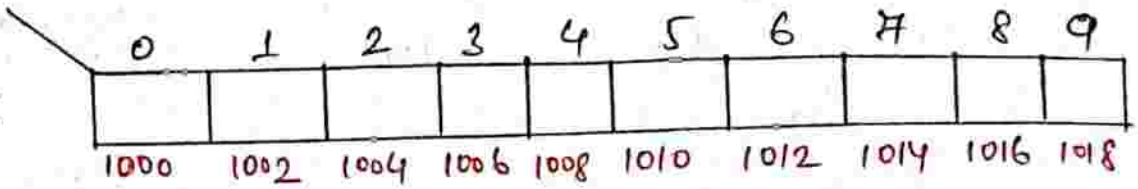
$$= 1000 + (3 \times 4 + 2) \times 2$$

$$= 1000 + 14 \times 2$$

$$= 1028.$$

$$\boxed{Addr(a[i][j]) = BA + [(j - LB_2) \times nr + (i - LB_1)] \times w}$$

1-D Array Address Calculation



int a[10];
- int → 2 Bytes → Width (w)
- 10 elements

$a[7] = \overbrace{1000}^{BA} + (7-0) * \overbrace{2}^{w}$

(7-0) ← No. of elements before 7.

$= 1000 + 14$

$= 1014$

$a[5] = 1000 + (5-0) * 2$

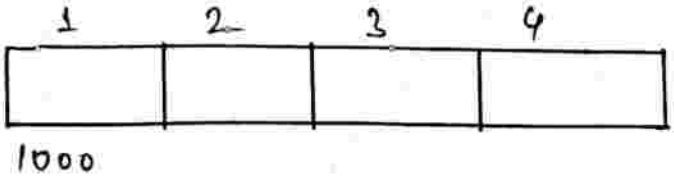
$= 1010$

Why Array index always starts with Zero?

Ans:-> (7-0) → 7, (5-0) → 5. If array starts with zero, then the subtraction operator use is wastage. i.e. No need to use '-' operator.

$\therefore a[9] = 1000 + 9 * 2$

$= 1018.$



$a[4] = \text{Base Address} + (4-1) * w$

↑ Here '-' operator is mandatory.

NOTE:-> a[7] means before index 7, there are 7 elements.

a[9] means before index 9, there are 9 elements.

$\therefore$ to include a[9] element, we have to add +1 to 9.

i.e total elements $= 9 + \overbrace{1}^{\text{to include a[9]}}$

$= 10$ ← (9-0) ← UB-LB

Example $A[-25 \ldots\ldots +75]$ $n = UB - LB + 1$

$= 75 - (-25) + 1$ self
$= 101$

B.A = 1000 (LB = -25, UB = +75)

w = 10 Bytes

∴ Loc A[60] = ?

indicates total element before 60.

$$\text{Loc } A[60] = 1000 + [(60 - (-25)) * 10$$

$$= 1000 + 85 * 10$$

$$= 1850.$$

NOTE :⇒ Formula for Address Calculation in 1-D Array

A [LB, ... UB] and Base Address = B.A

width = w

$$\text{Loc } A[i] = BA + (i - LB) * w$$

$$\text{no. of elements} = UB - LB + 1$$

Q1 Consider a 20×5 2D array. Base Address is 1000, size of an element = 2. Calculate the address of element marks [18][4] by Row-Major order and column-Major order.

Sol Given that BA = 1000

$w = 2$

$nr = 20$

$nc = 5$

Row-Major Order

Address(A[i][j]) = BA + [(i-0)nc + (j-0)] * w

∴ Address(marks[18][4]) = 1000 + [(18-0)5 + (4-0)] * 2

= 1000 + [18*5 + 4] * 2

= 1000 + 94 * 2

= 1188.

Column-Major Order

Address(A[i][j]) = BA + [(i-0) + nr(j-0)] * w

∴ Address(marks[18][4]) = 1000 + [(18-0) + 20(4-0)] * 2

= 1000 + [18 + 80] * 2

= 1000 + 98 * 2

= 1196.

Q2 Consider a two-dimensional array arr[10][10], which has base address = 1000 and number of bytes per element of the array = 2. Compute the address of element arr[8][5] by Row-Major order and column-Major order.

Sol :⇒ Given that, BA = 1000

$w = 2$

$nr = 10$

$nc = 10$

Row-Major order

Address (A[i][j]) = BA + [(i-0)nc + (j-0)] * w

∴ Address (arr[8][5]) = 1000 + [(8-0)10 + (5-0)] * 2

= 1000 + [80+5] * 2

= 1000 + 85 * 2

= 1170

Column-Major order

Address (A[i][j]) = BA + [(i-0) + nr(j-0)] * w

∴ Address (arr[8][5]) = 1000 + [(8-0) + 10(5-0)] * 2

= 1000 + [8+50] * 2

= 1000 + 58 * 2

= 1116.

Q1. Consider a 20X5 2D array. Base Address is 1000, size of an element = 2. Calculate the address of element marks[18][4] by Row-Major Order and column-Major order. Given that Array index starts from (1,1).

Sol: Given that, BA = 1000

$w = 2$

$mr = 20$

$nc = 5$

Row-Major Order

$$\text{Address}(A[i][j]) = BA + [(i-1)*nc + (j-1)]*w$$

$$\therefore \text{Address}(\text{marks}[18][4]) = 1000 + [(18-1)5 + (4-1)]*2$$

$$= 1000 + [17*5 + 3]*2$$

$$= 1000 + 176$$

$$= 1176.$$

Column-Major order

$$\text{Address}(A[i][j]) = BA + [(i-1) + (j-1)*mr]*w$$

$$\therefore \text{Address}(\text{marks}[18][4]) = 1000 + [(18-1) + (4-1)20]*2$$

$$= 1000 + [17 + 3*20]*2$$

$$= 1000 + (17+60)*2$$

$$= 1000 + 77*2$$

$$= 1000 + 154$$

$$= 1154$$

Q2 Consider a two-dimensional array arr[10][10], which has base address = 1000 and number of bytes per element of the array = 2. Compute the address of element arr[8][5] by Row-Major order and column-Major order. Given that Array index starts from (1,1).

Sol: Given that, BA = 1000

$w = 2$

$nr = 10$

$nc = 10$

Row-Major Order

Address(A[i][j]) = BA + [(i-1)nc + (j-1)] * w

∴ Address(arr[8][5]) = 1000 + [(8-1)10 + (5-1)] * 2

= 1000 + [70 + 4] * 2

= 1000 + 74 * 2

= 1000 + 148

= 1148

Column-Major Order

Address(A[i][j]) = BA + [(i-1) + nr(j-1)] * w

∴ Address(arr[8][5]) = 1000 + [(8-1) + 10(5-1)] * 2

= 1000 + [7 + 40] * 2

= 1000 + 47 * 2

= 1000 + 94

= 1094.

Q3 Each element of an array a[20][50] requires 4 bytes of storage. Base Address of data is 2000. Determine the location of a[10][10] when the array is stored as

(i) Row-Major order (ii) Column-Major order

Given that Array index starts from (1,1).

Sol: Given that BA = 2000
w = 4
nr = 20
nc = 50

Row-Major order

Address (A[i][j]) = BA + [(i-1)*nc + (j-1)]*w

∴ Address (a[10][10]) = 2000 + [(10-1)50 + (10-1)]*4
= 2000 + [450 + 9]*4
= 2000 + 459*4
= 2000 + 1836
= 3836

Column-Major order

Address (A[i][j]) = BA + [(i-1) + nr(j-1)]*w

∴ Address (a[10][10]) = 2000 + [(10-1) + 20(10-1)]*4
= 2000 + [9 + 180]*4
= 2000 + 189*4
= 2000 + 756
= 2756

Q4. Consider the 2D array

A [25 ..... 79, 55 ..... 124]

Base Address = 1000

Size of each element w = 100

Find Loc (A[55][75]) by using (i) Row-Major order

(ii) Column-Major order.

Sol:

A [25 ..... 79, 55 ..... 124]

(First Row, Last Row, First Col, Last Col)

No. of Rows, $n_r = ub_1 - lb_1 + 1$

$= \text{Last Row} - \text{First Row} + 1$

$= 79 - 25 + 1$

$= 55$

No. of Columns, $n_c = ub_2 - lb_2 + 1$

$= \text{Last Col} - \text{First Col} + 1$

$= 124 - 55 + 1$

$= 70$

Row-major order

$\text{Address}(A[i][j]) = BA + [(i - lb_1) n_c + (j - lb_2)] * w$

$\therefore \text{Address}(A[55][75]) = 1000 + [(55 - 25)70 + (75 - 55)] * 100$

$= 1000 + [30 \times 70 + 20] * 100$

$= 1000 + (2100 + 20) * 100$

$= 1000 + 212000$

$= 213000$

Column-major order

Address (A[i][j]) = BA + [(i - lb1) + nr (j - lb2)] * w

∴ Address (A[55][75]) = 1000 + [(55 - 25) + 55(75 - 55)] * 100

= 1000 + [30 + 55 * 20] * 100

= 1000 + (30 + 1100) 100

= 1000 + 113000

= 114000

Q5 Consider the 2D array

A[-5 .... +5, -5 .... +5]

Base Address = 1000

Size of each element = 10.

Find Loc (A[4][-4]) using (i) Row-Major Order

(ii) Column-Major order

Sol

A[-5 .... +5, -5 .... +5]

lb1 ub1 lb2 ub2

BA = 1000

w = 10

nr = 5 - (-5) + 1 = 11

nc = 5 - (-5) + 1 = 11

Row-Major order

Address (A[i][j]) = BA + [(i - lb1) nc + (j - lb2)] w

∴ Address (A[4][-4]) = 1000 + [(4 - (-5)) * 11 + (-4 - (-5))] * 10

= 1000 + [9 * 11 + 1] * 10

= 1000 + 1000

= 2000

Column-Major order

$$\text{Address}(A[i][j]) = BA + [(i - lb_1) + n_r(j - lb_2)] * w$$

$$\therefore \text{Address}(A[4][-4]) = 1000 + [(4 - (-5)) + 11(-4 - (-5))] * 10$$

$$= 1000 + [9 + 11 * 1] * 10$$

$$= 1000 + 200$$

$$= 1200$$

Important Questions

Q1 Consider the linear arrays AAA(5:50), BBB(-5:10) and CCC(18).

(a) Find the number of elements in each array.

(b) Suppose Base(AAA) = 300 and w = 4 words per memory cell for AAA. Find the address of AAA[15], AAA[35], and AAA[55].

Soln (a) The number of elements in array is equal to the length of array.

$$\text{Length} = UB - LB + 1$$

$$\therefore \text{Length}(AAA) = 50 - 5 + 1 = 46$$

$$\text{Length}(BBB) = 10 - (-5) + 1 = 16$$

$$\text{Length}(CCC) = 18 - 1 + 1 = 18 \quad (LB = 1)$$

(b) $Loc(AAA[K]) = Base(AAA) + w(K - LB)$

$\therefore Loc(AAA[15]) = 300 + 4(15-5)$

$= 300 + 4*10 = 340$

$Loc(AAA[35]) = 300 + 4(35-5)$

$= 300 + 4*30 = 420$

AAA [55] is not an element of AAA, since 55 exceeds UB = 50.

Q2 Consider the linear arrays XXX(-10:10), YYY(1935:1985), ZZZ(35).

(a) Find the number of elements in each array.

(b) Suppose Base(YYY) = 400 and w = 4 words per memory cell for YYY. Find the address of YYY[1942], YYY[1977], and YYY[1988].

N-Dimensional array address calculation

Array is $A(m_1, m_2, \ldots m_n)$

we have to find address of $A[K_1][K_2]\ldots[K_n]$

where $1 \le K_1 \le m_1$, ...

$1 \le K_2 \le m_2$ ...

$1 \le K_n \le m_n$

$\therefore L_i = UB - LB + 1$ and $E_i = K_i - LB$

Row-Major Order

Address of $A[K_1, K_2, \ldots K_n]$

$$= BA + w * [(((\ldots(E_1 L_2 + E_2)L_3 + E_3)L_4 \ldots E_{N-1})L_N + E_N)$$

Example: $A(2:8, -4:1, 6:10)$, $BA = 200$, $w = 4$

Find address $A[5, -1, 8]$.

Soln $L_1 = 8 - 2 + 1 = 7$ $L_2 = 1 - (-4) + 1 = 6$ $L_3 = 10 - 6 + 1 = 5$

$E_1 = 5 - 2 = 3$ $E_2 = -1 - (-4) = 3$ $E_3 = 8 - 6 = 2$

$E_1 L_2 = 18$

$E_1 L_2 + E_2 = 18 + 3 = 21$

$(E_1 L_2 + E_2) L_3 = 21 \times 5 = 105$

$(E_1 L_2 + E_2) L_3 + E_3 = 105 + 2 = 107$

$\therefore$ Address $(A[5, -1, 8]) = 200 + 4 \times 107$

$= 200 + 428 = 628.$

Column-Major Order

Address $A[K_1, K_2, \ldots K_n] = BA + \omega * [((\ldots E_N L_{N-1} + E_{N-1}) L_{N-2} \ldots E_2) L_1 + E_1)]$

Example:⇒ $A(1:8, -5:5, -10:5)$, $BA = 400$, $\omega = 4$

Find address of $A[3][3][3]$

Soln $L_1 = 8-1+1 = 8$, $L_2 = 5-(-5)+1 = 11$, $L_3 = 5-(-10)+1 = 16$

$E_1 = 3-1 = 2$, $E_2 = 3-(-5) = 8$, $E_3 = 3-(-10) = 13$

$E_3 L_2 = 13 \times 11 = 143$

$E_3 L_2 + E_2 = 143 + 8 = 151$

$(E_3 L_2 + E_2) L_1 = 151 \times 8 = 1208$

$(E_3 L_2 + E_2) L_1 + E_1 = 1208 + 2 = 1210$

$\therefore$ Address $(A[3][3][3]) = 400 + 4 * 1210$

$= 5240$

(16)

Q An array S[10][30] is stored in the memory along the column with each of its element occupying 2 bytes. Find out the memory location of S[5][10], if element S[2][5] is stored at the location 8200.

Soln:⇒ Array S[10][30]

No. of rows, nr = 10

No. of columns, nc = 30

w = 2

Address (S[2][5]) = 8200

Given that array elements are stored in column-major order.

$\therefore BA + w[(i-0) + nr(j-0)] = 8200$

$\Rightarrow BA + 2[(2-0) + 10(5-0)] = 8200$

$\Rightarrow BA + 2[2 + 50] = 8200$

$\Rightarrow BA + 2 * 52 = 8200$

$\Rightarrow BA + 104 = 8200$

$\Rightarrow BA = 8096$

$\therefore$ Base Address of Array = 8096

Address (S[5][10]) $= BA + w[(i-0) + nr(j-0)]$

$= 8096 + 2[(5-0) + 10(10-0)]$

$= 8096 + 2[5 + 100]$

$= 8096 + 2 * 105$

$= 8096 + 210$

$= 8306$

Q Consider the following multidimensional arrays:

$X(-5:5, 3:33)$ $Y(3:10, 1:15, 10:20)$

(a) Find the length of each dimension and the number of elements in X and Y.

(b) Suppose Base(Y) = 400 and there are $w = 4$ words per memory location. Find the effective indices E_1, E_2, E_3 and the address of $Y[5, 10, 15]$ assuming

(i) Y is stored in row-major order

(ii) Y is stored in column-major order.

Sparse Matrix And its Representation

A matrix is a two-dimensional data object made of m rows and n columns.

∴ total elements = m x n

If most of the elements of the matrix have 0 value, then it is called a Sparse matrix.

Why do we use sparse matrix instead of simple matrix?

Ans:→ (1) Storage :→ There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

(2) Computing Time :→ Computing time can be saved by logically designing a data structure traversing only non-zero elements.

Example of a Sparse Matrix

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

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeros in the matrix are of no use in most of the cases.

So, instead by storing zeroes with non-zero elements we only store non-zero elements.
This means storing non-zero elements with triples (Row, Column, value).

Sparse Matrix Representation

(1) Array Representation
(2) Linked List Representation

Method 1 : Using Arrays

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

(i) Row : Index of row, where non-zero element is located.

(ii) Column : Index of column, where non-zero element is located.

(iii) Value : Value of non-zero element located at index (row, column).

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

⇨

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

Method 2 : Using Linked List

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

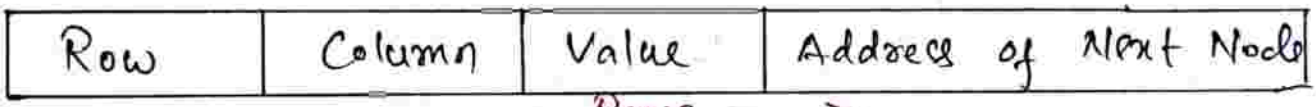
(i) **Row** :→ Index of row, where non-zero element is located.

(ii) **Column** :→ Index of column, where non-zero element is located.

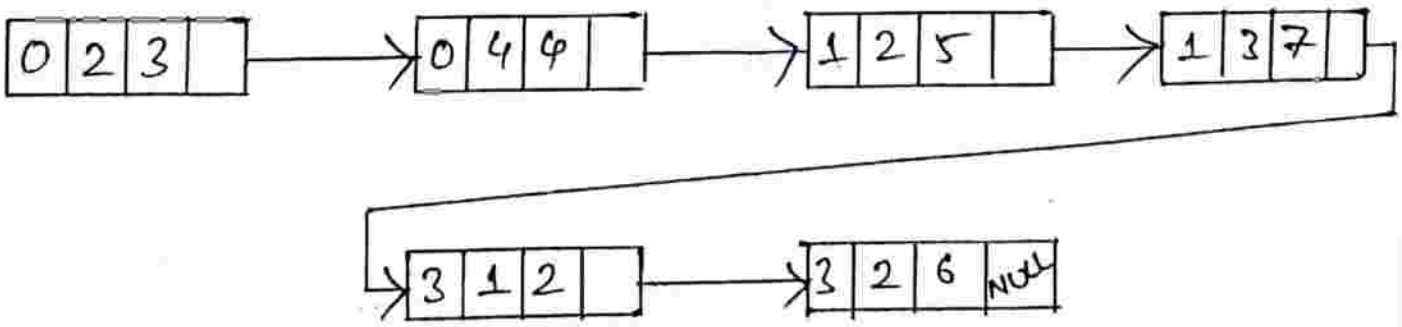
(iii) **Value** :→ Value of the non-zero element located at index (row, column).

(iv) **Next Node** :→ Address of next node.

Node Structure is



Column \ Rows	0	1	2	3	4
0	0	0	3	0	4
1	0	0	5	7	0
2	0	0	0	0	0
3	0	2	6	0	0



Different Forms of a Sparse Matrix

(1) **Diagonal Matrix** :→ When the non-zero elements are stored on the leading diagonal of the matrix.

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

(2) **Lower Triangular Matrix** :→ Where the non-zero elements are placed below the leading diagonal.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 9 & 3 & 0 & 0 \\ 6 & 0 & 12 & 0 \end{bmatrix}$$

(3) **Upper Triangular Matrix** :→ In this, the non-zero elements are placed above the leading diagonal.

$$\begin{bmatrix} 0 & 2 & 0 & 1 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Applications of Array

1. Array is used to store the number of elements belonging to same data type.
2. Array used for maintaining multiple variable names using single name.
3. Arrays are used to perform Matrix Operations.
4. Arrays are used to implement Search Algorithms —
   (a) Linear Search
   (b) Binary Search
5. Arrays are used to implement Sorting Algorithms- we use single dimensional arrays to implement sorting algorithms like
   Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Merge Sort, Heap Sort etc.
6. Arrays are used to implement Data Structures— we use single dimensional arrays to implement data structures like
   Stack Using Array
   Queue using Array
7. Arrays are used to implement CPU scheduling Algorithms like FCFS, SJF, SRTF, Round Robin etc.