



*A.V. Parekh Technical Institute*  
(Department of Technical Education, Gujarat State)



- **Vision of Computer Department**
- Develop globally competent Computer Engineering Professionals who achieve excellence in an environment conducive for technical knowledge, skills, moral and ethical values with a focus to serve the society.
- **Mission of Computer Department**
- M1 : To provide state of the art infrastructure and facilities for imparting quality education and computer engineering skills for societal benefit.
- M2 : Adopt industry oriented curriculum with an exposure to technologies for building systems & application in computer engineering.
- M3 : To provide quality technical professional as per the industry and societal needs, encourage entrepreneurship, nurture innovation and life skills in consonance with latest interdisciplinary trends.

# **Data Structures and Algorithms**

## **(4330704 )**

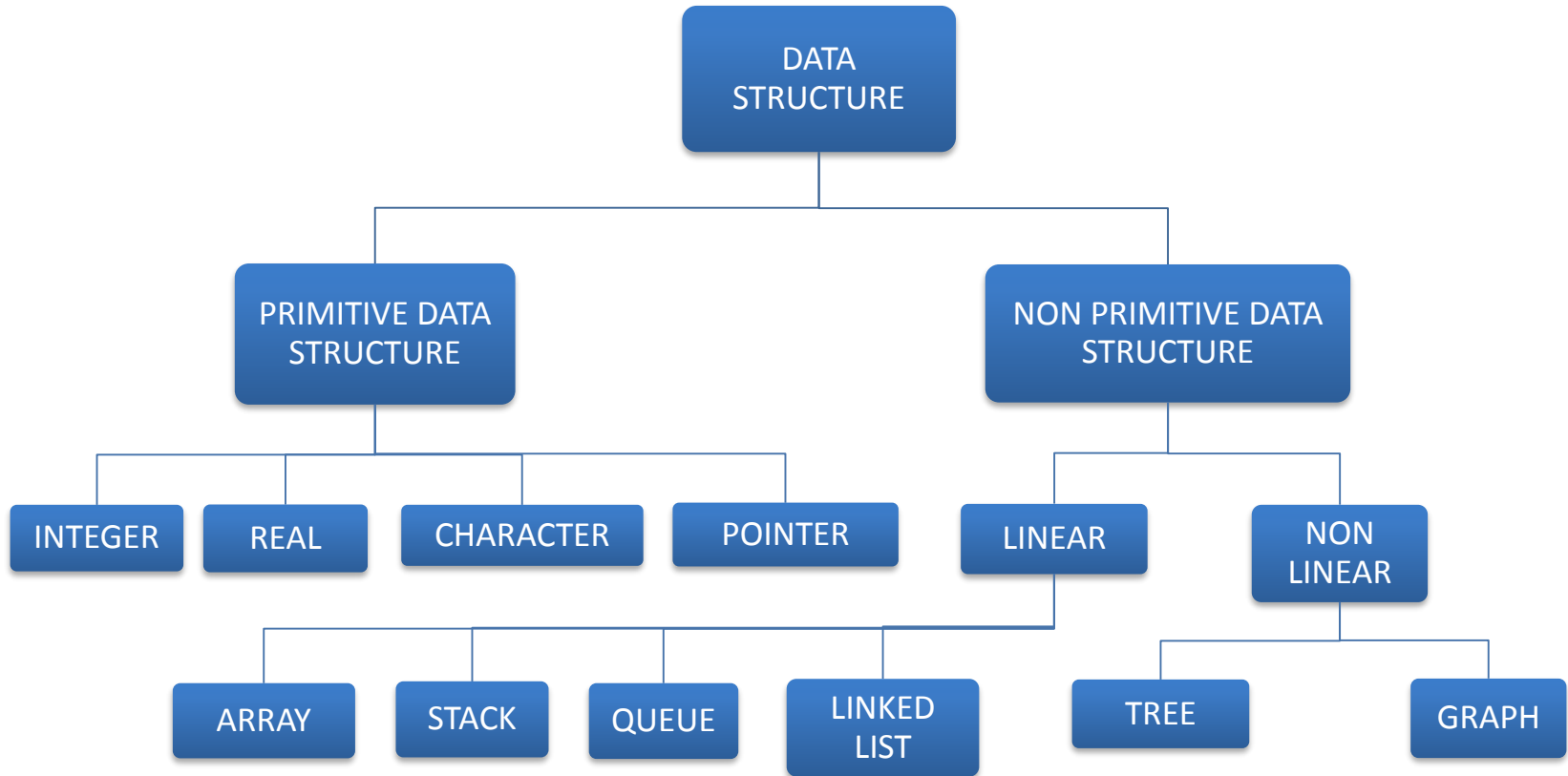
### **Course Outcomes**

1. Perform basic operations on arrays and strings.
2. Demonstrate algorithms to insert and delete elements from the stack and queue data structure.
3. Apply basic operations on the linked list data structure.
4. Illustrate algorithms to insert, delete and searching a node in tree.
5. Apply different sorting and searching algorithms to the small data sets.

# UNIT 1

## Basic Concepts Of Data Structure

# TYPES OF DATA STRUCTURE



# INTRODUCTION

- Data means collection of raw facts. Examples: marks of students, name of an employee etc.
- Examples:
  - a) To do list
  - b) Organize directories in computer
  - c) Stack of books
  - d) Dictionary
- So, Data structure is a specialized format for organizing and storing data in memory. How data is stored and organized matters as we want to perform operations on those data.
- **Definition:** Data structure is a way of storing and organizing data in a computer so that it can be used efficiently.

# PRIMITIVE DATA STRUCTURE

- The Data structures those are directly processed by machine using its instructions are known as primitive data structures.
- Primitive data structures are:
- **Integer:** Represents numerical values. Sign - magnitude method and 1's complement method are used to represent integer numbers.
  - ✓ Example: 1000
- **Real:** Number having fractional part. Real numbers are also stored in scientific format.
  - ✓ Example: 23.45

# PRIMITIVE DATA STRUCTURE

- **Character:** Used to store nonnumeric information. It can be letters [A-Z], [a-z], digits and special symbols. ASCII and EBCDIC are commonly known character set supported by computer.
- **Pointer:** Pointer is a variable which points to the memory address. This memory address is the location of other variable in memory. It provides faster insertion and deletion of elements.

# NON-PRIMITIVE DATA STRUCTURE

- The data structures those are not directly processed by machine using its instructions are known as non primitive data structure.
- Non-Primitive data structures are classified into linear and non-linear data structure.
- **Linear data structure:** In Linear data structures, the data items are processed in a linear fashion. Data are processed one by one sequentially.

Example: Array, Stack, Queue, List

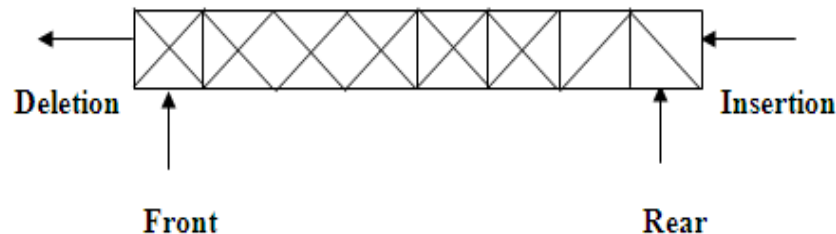
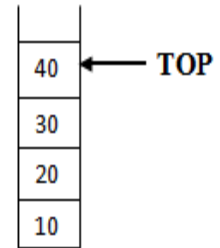
- **Non-linear data structure:** In Non-Linear data structures, the data items are processed in non-linear fashion.

Example: Tree, Graph



# LINEAR DATA STRUCTURE

- **Arrays:** An array is a collection of similar type of data elements. The data type of the array may be any valid data type like char, int, float or double.
- **Stack:** Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.
- **Queue:** Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

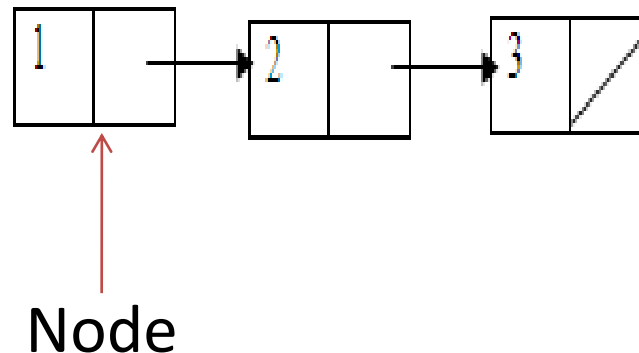


# LINEAR DATA STRUCTURE

- **Linked list:** It is dynamic data structure. A linked list is a collection of nodes. Each node has two fields:

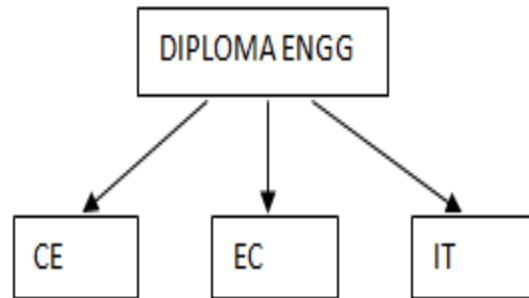
1) Data

2) Address, which contains the address of the next node.

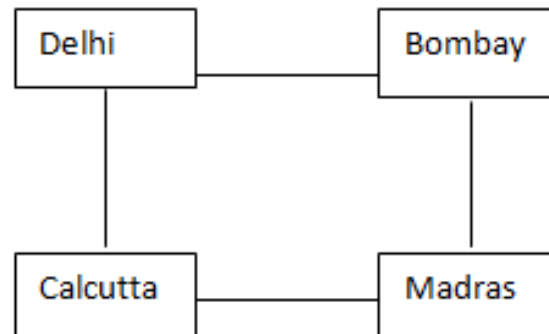


# NON-LINEAR DATA STRUCTURE

- **Tree:** A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order.



- **Graph:** A graph is a non-linear data structure which is a collection of vertices (also called nodes) and edges that connect these vertices.



# OPERATIONS ON DATA STRUCTURE

- **Traversing:** Access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.
- **Searching:** Find the location of data item within the data structure. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.
- **Insertion:** It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.
- **Deletion:** It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

# OPERATIONS ON DATA STRUCTURE

- **Sorting:** Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order.
- **Merging:** Lists of two sorted data items can be combined to form a single list of sorted data items.
- ❑ An abstract data type (ADT) is way of classifying data structures based on how they are used and the behaviors they provide. ADT is focusing on what it does and ignoring how it does its job.

# ALGORITHM

- Algorithm is a stepwise solution to a problem.
- Algorithm is a finite set of instructions that is followed to accomplish a particular task.
- In general terms, an algorithm provides a blueprint to write a program to solve a particular problem.
- Properties:
  1. **Input**- There should be 0 or more inputs supplied externally to the algorithm.
  2. **Output**- There should be at least 1 output obtained.
  3. **Definiteness**- Every step of the algorithm should be clear and well defined.
  4. **Finiteness**- The algorithm should have finite number of steps.
  5. **Correctness**- Every step of the algorithm must generate a correct output.

# FEATURES OF ALGORITHM

- **Name of Algorithm:** Every algorithm is given an identifying name, written in capital letters.

Example: STRLENTN(), INSERT(ARR, N)

- **Steps:** The algorithm is made of sequence of steps.
- **Assignment Statements:** The assignment statement is indicated by arrow. Example:  $X \leftarrow 10$
- Algorithm may employ one of the following control structures:
  - (a) sequence,
  - (b) decision
  - (c) repetition.

# SEQUENCE

- By sequence means that each step of an algorithm is executed in a specified order.
- Algorithm to add two numbers:  
Step 1: [Input first number]  
    read A  
Step 2: [Input second number]  
    read B  
Step 3: [Perform addition]  
     $SUM \leftarrow A + B$   
Step 4: [Display answer]  
    print SUM  
Step 5: END
- This algorithm performs the steps in a purely sequential order.



# DECISION

- Decision statements are used when the execution of a process depends on the outcome of some condition.
- IF construct: IF *condition* Then *process*
- IF-ELSE construct:     IF *condition*  
                                  Then *process1*  
                                  ELSE *process2*
- Algorithm to check equality of two numbers:  
    Step 1: [Input first number]  
            read A  
    Step 2: [Input second number]  
            read B  
    Step 3: [Check for equality]  
            IF A = B  
                print "Equal"  
            ELSE  
                print "Not Equal"  
    Step 4: END

# REPETITION

- Repetition involves executing one or more steps for a number of times. These loops execute one or more steps until some condition is true.
- Algorithm that prints the first 10 natural numbers.

Step 1: [Initialization]

$I \leftarrow 1, N \leftarrow 10$

Step 2: [Loop on control variable]

Repeat step 3 and 4 while  $I \leq N$

Step 3: [Display value]

Print  $I$

Step 4: [Increment value of  $I$ ]

$I \leftarrow I + 1$

Step 5: END

# COMPLEXITY

- An algorithm is basically a set of instructions that solve a problem.
- It is not uncommon to have **multiple algorithms to tackle the same problem**, but the choice of a particular algorithm must depend on the **time and space complexity** of the algorithm.
- **Time Complexity:** The time complexity of an algorithm is basically the amount of time taken by an algorithm to run as a function of the input size.
- **Space Complexity:** The space complexity of an algorithm is the amount of space or memory taken by an algorithm to run as a function of the input size.
- Time Complexity is more important than Space Complexity.

# TIME COMPLEXITY

- We can have three cases to analyze an algorithm:
  - 1) Worst Case
  - 2) Average Case
  - 3) Best Case
- **Worst Case Complexity-** It is the maximum time taken by an algorithm to solve a problem.
- The worst-case running time of an algorithm is an *upper bound* on the running time of an algorithm for any input.
- **Best case Complexity-** It is the minimum time taken by an algorithm to solve a problem.
- The best-case running time of an algorithm is *lower bound* on running time of an algorithm.
- **Average case Complexity-** It is in between best case and worst case that is average time taken by an algorithm to solve a problem.

# ASYMPTOTIC NOTATION

- To analyze the complexity of any algorithm, we use some standard notations, also known as **Asymptotic Notations**.
- Asymptotic analysis of algorithm is a method of defining the mathematical boundary of its run-time performance.
- Complexity can be expressed using a function  $f(n)$  where  $n$  is the input size.
- When we analyze any algorithm, we generally get a formula to represent the amount of time required for execution.
- If a function is linear (without any loops or recursions), the running time of that algorithm can be given as the number of instructions it contains.
- However, if an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm.

# ASYMPTOTIC NOTATION

- Example:

```
for(i=1;i<=100;i++)  
    printf("%d", i);
```

Here printf() statement will be executed 100 times. So time complexity is 100.

- So general formula is  $f(n) = n$ .

- Another Example:

```
for(i=0;i<10;i++){  
    for(j=0; j<10;j++){  
        printf("Hello");  
    }  
}
```

The outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times. Therefore, the efficiency here is 100.

- The generalized formula for quadratic loop can be given as  $f(n) = n^2$ .

# ASYMPTOTIC NOTATION

- The commonly used asymptotic notations used for calculating the running time complexity of an algorithm are:
  1. Big O Notation ( $O$ )
  2. Omega Notation ( $\Omega$ )
  3. Theta Notation ( $\theta$ )
- Big O Notation:  $O$  stands for “order of”.
- If an algorithm performs  $n^2$  operations for  $n$  elements, then that algorithm would be described as an  $O(n^2)$  algorithm.
- When expressing complexity using the Big O notation, constant multipliers are ignored. So, an  $O(4n)$  algorithm is equivalent to  $O(n)$ .

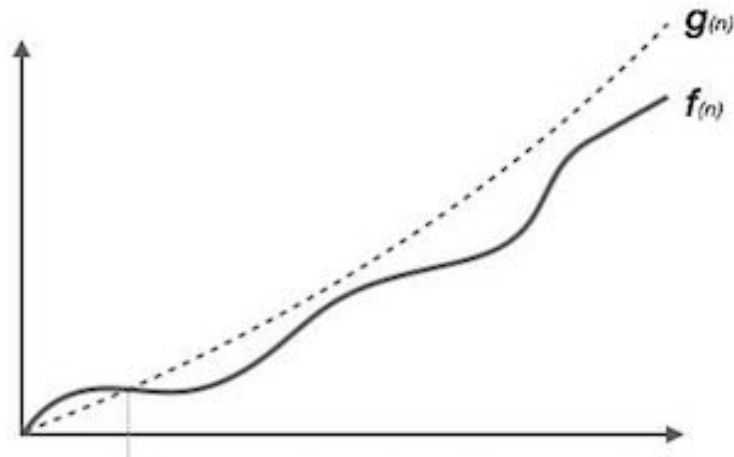
# BIG O NOTATION

- This notation is known as the upper bound of the algorithm.
- It measures the worst case of time complexity or the longest amount of time, algorithm takes to complete their operation.
- Definition: If  **$f(n)$**  and  **$g(n)$**  are the two functions defined for positive integer number  $n$ , then

**$f(n) = O(g(n))$** , i.e.  $f$  of  $n$  is Big-O of  $g$  of  $n$

if and only if positive constants  $c$  and  $n$  exist, such that  $f(n) \leq cg(n)$ .

- This implies that  $f(n)$  does not grow faster than  $g(n)$ . Hence,  $g$  provides an upper bound.





# ARRAY

- **Definition:** Array is a collection of elements of same data type which are stored in consecutive memory location and are referred by common name.
- It is homogenous data structure.
- Index or subscript is used to access elements of an array.
- Array is classified into two categories:
  - 1) One dimensional array
  - 2) Two dimensional array

# CHARACTERISTICS OF ARRAY

- An array holds elements that have the same data type.
- Array elements are stored in subsequent memory locations.
- Array name represents the address of the starting element.
- Array size should be mentioned in the declaration. Array size must be a constant expression and not a variable.
- While declaring the 2D array, the number of columns should be specified and it's a mandatory. Whereas for number of rows there is no such rule.
- An Array index by default starts from 0 and ends with the index number N-1.
- The size of an Array cannot be changed at Run Time.

# ONE DIMENSIONAL ARRAY

- Syntax to declare an array: `data_type array_name[size];`
- Example: `int a[10];`
- Array name is an identifier which follows the rules of writing an identifier.
- Each integer takes 2 bytes so this array declaration reserves 20 bytes of memory for the array.
- The starting index for the array (i.e. 0) is known as ***Lower Bound (LB)*** and higher index n-1 is known as ***Upper Bound (UB)***.
- Number of elements in array is called array length and it is given by the following formula:  
$$\text{Length} = \text{UB} - \text{LB} + 1$$
- In our example, upper bound of an array is 9 and lower bound is 0. Hence, the length is  $9 - 0 + 1 = 10$ .

# STORAGE REPRESENTATION OF ONE DIMENSIONAL ARRAY

Example: `char ch[6] = {'a', 'b', 'c', 'd', 'e', 'f'};`

INDEX	0	1	2	3	4	5
ELEMENTS	a	b	c	d	e	f
ADDRESS	1100	1101	1102	1103	1104	1105

Address of first element is called Base Address(B).

Memory Space acquired by each element is called width(W). Here it is 1 byte.

- Address of an element of an array say “A[I]” is calculated using the following formula:

$$\text{Address of A [I]} = B + W * ( I - LB )$$

Where, B= Base address

W= Storage Size of one element stored in the array (in byte)

I = Subscript of element whose address is to be found

LB= Lower limit / Lower Bound of subscript, if not specified assume 0

# EXAMPLE

- Given the base address of an array B[1300...1900] as 1020 and size of each element is 2 bytes in memory. Find the address of B[1700].

- **Solution:**

Here  $I = 1700$ ,  $B = 1020$ ,  $W = 2$  and  $LB = 1300$ .

$$\begin{aligned}\text{Address of B[1700]} &= 1020 + 2 * (1700 - 1300) \\ &= 1020 + 2 * 400 \\ &= 1020 + 800 \\ &= 1820\end{aligned}$$

# OPERATIONS ON ARRAY

- Traversal
- Insert
- Delete
- Search
- Sort
- Merge

# TRAVERSAL

## Algorithm: TRAVERSE (A, LB, UB)

- A is an array
- LB is lower limit of an array
- UB is Upper limit of an array

Step-1) [Initialization]

COUNT ← LB

Step-2) [Perform Traversal]

Repeat step-3 and step-4 until  $\text{COUNT} \leq \text{UB}$

Step-3) [Display element of an array]

print A[COUNT]

Step-4) [Increment index value]

COUNT ← COUNT + 1

Step-5) [Finished]

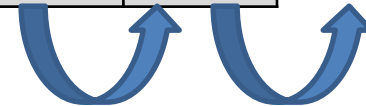
Exit

# INSERTION

This operation is used to add new element into one dimensional array.

**Example:**

ARRAY	VALUE	10	20	30	40	50
A	INDEX	0	1	2	3	4



ARRAY	VALUE	10	20	30	40	40	50
A	INDEX	0	1	2	3	4	5

ARRAY	VALUE	10	20	30	35	40	50
A	INDEX	0	1	2	3	4	5



# INSERTION

Algorithm: INSERT (A, N, POS, VALUE)

- A is an array
- N indicates number of elements in an array
- POS indicates position at which you want to insert element
- VALUE indicates value (element) to be inserted

Step-1) [Initialization]

TEMP  $\leftarrow$  N-1

Step-2) [Move elements one position right]

Repeat While (TEMP  $\geq$  POS)

A [TEMP + 1]  $\leftarrow$  A[TEMP]

TEMP  $\leftarrow$  TEMP - 1

Step-3) [Insert element]

A [POS]  $\leftarrow$  VALUE

Step-4) [Increase size of array]

N  $\leftarrow$  N + 1

Step-5) [Finished]

Exit


ARRAY	VALUE	10	20	30	40	50
A	INDEX	0	1	2	3	4

# DELETION

This operation is used to remove new element from already existing an array.

**Example:**

ARRAY	VALUE	10	20	30	40	50
A	INDEX	0	1	2	3	4



ARRAY	VALUE	10	30	40	50
A	INDEX	0	1	2	3

# DELETION

Algorithm: DELETE (A, N, POS)

- A is an array
- POS indicates position at which you want to delete an element
- N indicates number of elements in an array

Step-1) [Initialization]

TEMP ← POS

Step-2) [Move elements one position left]

Repeat While (TEMP ≤ N-1)

A [TEMP] ← A[TEMP+1]

TEMP ← TEMP + 1

Step-3) [Decrease size of array]

N ← N - 1

Step-4) [Finished]

Exit

ARRAY	VALUE	10	20	30	40	50
A	INDEX	0	1	2	3	4

# MERGE

This operation is used to join elements of two array.

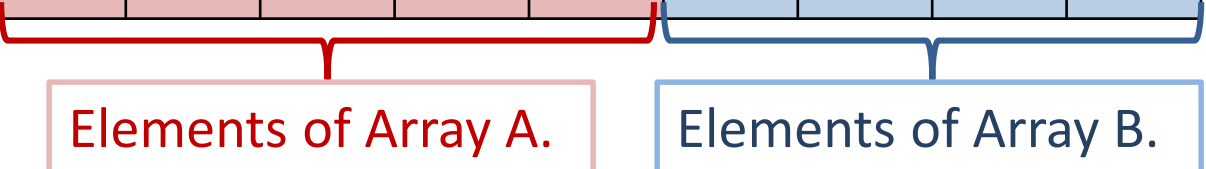
**Example:**

ARRAY A	VALUE	10	20	30	40	50
	INDEX	0	1	2	3	4

ARRAY B	VALUE	11	22	33	44
	INDEX	0	1	2	3

After merge operation, new array C will contain 9 elements

ARRAY C	VALUE	10	20	30	40	50	11	22	33	44
	INDEX	0	1	2	3	4	5	6	7	8



Elements of Array A.

Elements of Array B.

# MERGE

Algorithm: MERGE(A, N, B, M)

- A and B are array
- N indicates number of elements in an array A
- M indicates number of elements in an array B

Step-1) [Initialization]

$I \leftarrow 0, J \leftarrow 0, K \leftarrow 0$

Step-2) [Copy elements of array A to array C]

Repeat While ( $I \leq N-1$ )

$C[K] \leftarrow A[I]$

$I \leftarrow I+1$

$K \leftarrow K+1$

Step-3) [Copy elements of array B to array C]

Repeat While ( $J \leq M-1$ )

$C[K] \leftarrow B[J]$

$J \leftarrow J+1$

$K \leftarrow K+1$

# MERGE

Step-4) [Initialization]

$I \leftarrow 0$

Step-5) [Display elements of array C]

Repeat While ( $I \leq K-1$ )

print C[I]

$I \leftarrow I+1$

Step-6) [Finished]

Exit

# SEARCH

This operation is used to search particular element in one dimensional array.

**Example:**

ARRAY	VALUE	10	20	30	40	50
A	INDEX	0	1	2	3	4

- To search given value(i.e. key), we have to compare value with each element of an array.
- **CASE 1:** KEY = 40  
Here KEY is available in array, so return the position of that value.
- **CASE 2:** KEY = 90  
As KEY is not available in array, return -1/0.

# SEARCH

## ■ CASE 1: KEY = 40

Here value is available in array, so return the position of that value.

ARRAY	VALUE	10	20	30	40	50
A	INDEX	0	1	2	3	4

### Value of I

### COMPARISON

0	A[0] = KEY	10 = 40	False
1	A[1] = KEY	20 = 40	False
2	A[2] = KEY	30 = 40	False
3	A[3] = KEY	40 = 40	True
<b>So return 3.</b>			



# SEARCH

## ■ CASE 1: KEY = 90

Here value is available in array, so return the position of that value.

ARRAY	VALUE	10	20	30	40	50
A	INDEX	0	1	2	3	4

**Value of I**

**COMPARISON**

0	A[0] = KEY	10 = 90	False
1	A[1] = KEY	20 = 90	False
2	A[2] = KEY	30 = 90	False
3	A[3] = KEY	40 = 90	False
4	A[4] = KEY	50 = 90	False

**As we reach end of the array and KEY is not found in array, return-1.**

# SEARCH

Algorithm: SEARCH (A, N, KEY)

- A is an array
- N indicates number of elements in an array
- KEY indicates value (element) to be searched

Step-1) [Initialization]

$I \leftarrow 0$

Step-2) [Traverse the array to search value]

Repeat step-3 and step-4 until  $I \leq N-1$

Step-3) [Compare array element with value]

IF  $A[I] = \text{KEY}$

THEN return I

Step-4) [Increment index]

$I \leftarrow I + 1$

Step-5) [Finished]

return -1

# TWO DIMENSIONAL ARRAY

- Syntax: `data_type array_name[row_size][column_size];`
- Example: `int arr[][4];`

	Column 0	Column 1	Column 2
Row 0	<code>arr[0][0]</code>	<code>arr[0][1]</code>	<code>arr[0][2]</code>
Row 1	<code>arr[1][0]</code>	<code>arr[1][1]</code>	<code>arr[1][2]</code>
Row 2	<code>arr[2][0]</code>	<code>arr[2][1]</code>	<code>arr[2][2]</code>

- There are two ways array can be stored in memory.
  1. Row Major Order
  2. Column Major Order

# ROW MAJOR ARRAY

- Example:

		Column Index		
		0	1	2
Row Index	0	10	1	12
	1	3	4	7
	2	2	6	8

- In Row major array, elements are stored row by row in memory.

INDEX	0, 0	0, 1	0, 2	1, 0	1, 1	1, 2	2, 0	2, 1	2, 2
ELEMENTS	10	1	12	3	4	7	2	6	8
ADDRESS	1100	1102	1104	1106	1108	1110	1112	1114	1116

Row 0                      Row 1                      Row 2

# ROW MAJOR ARRAY

- Address of an element of array (ARR[I][J]) is calculated using following equation.
- **Address of ARR[I][J] = B + W \* [ N \* ( I – LBr ) + ( J – LBc ) ]**
- $ARR[1][0] = 1100 + 2 * [ 3 * (1-0) + (0-0) ] = 1100 + 6 = 1106$

Where

B = Base address

I = Row subscript of element whose address is to be found

J = Column subscript of element whose address is to be found

W = Storage Size of one element stored in the array (in byte)

LBr = Lower limit of row/start row index of matrix, if not given assume 0 (zero)

LBc = Lower limit of column/start column index of matrix, if not given assume 0 (zero)

N = Number of columns of the given matrix

# EXAMPLE

- Consider a  $20 * 5$  two-dimensional array marks which has its base address = 1000 and the size of an element = 2. Now compute the address of the element, marks[18][4] assuming that the elements are stored in row major order.

- **Solution:**

$$\text{Address of } A[I][J] = B + W * [N * (I - LBr) + (J - LBC)]$$

$$\begin{aligned}\text{Address of marks}[18][4] &= 1000 + 2 * [5 * (18 - 0) + (4 - 0)] \\ &= 1000 + 2 * [5 * 18 + 4] \\ &= 1000 + 2 * 94 \\ &= 1000 + 188\end{aligned}$$

$$\text{Address of marks}[18][4] = 1188$$

# COLUMN MAJOR ARRAY

- Example:

		Column Index		
Row Index		0	1	2
	0	10	1	12
	1	3	4	7
	2	2	6	8

- In Column major array, elements are stored column by column in memory.

INDEX	0, 0	1, 0	2, 0	0, 1	1, 1	2, 1	0, 2	1, 2	2, 2
ELEMENTS	10	3	2	1	4	6	12	7	8
ADDRESS	1100	1102	1104	1106	1108	1110	1112	1114	1116

Column 0      Column 1      Column 2

# COLUMN MAJOR ARRAY

- Address of an element of array (ARR[I][J]) is calculated using following equation.

- **Address of ARR[I][J] = B + W \* [( I – LBr ) + M \* ( J – LBc )]**

Where

B = Base address

I = Row subscript of element whose address is to be found

J = Column subscript of element whose address is to be found

W = Storage Size of one element stored in the array (in byte)

LBr = Lower limit of row/start row index of matrix, if not given assume 0 (zero)

LBc = Lower limit of column/start column index of matrix, if not given assume 0 (zero)

M = Number of rows of the given matrix



# EXAMPLE

- Consider a  $20 * 5$  two-dimensional array marks which has its base address = 1000 and the size of an element = 2. Now compute the address of the element, marks[18][4] assuming that the elements are stored in column major order.

- **Solution:**

$$\text{Address of } A[I][J] = B + W * [(I - LBr) + M * (J - LBC)]$$

$$\begin{aligned}\text{Address of marks}[18][4] &= 1000 + 2 * [(18 - 0) + 20 * (4 - 0)] \\ &= 1000 + 2 * [18 + 80] \\ &= 1000 + 2 * [98] \\ &= 1000 + 196\end{aligned}$$

$$\text{Address of marks}[18][4] = 1196$$

# BINARY SEARCH

- Given a sorted array `arr[]` of  $n$  elements, write a function to search a given element  $x$  in `arr[]`.
- A simple approach is to do linear search. The time complexity of above algorithm is  $O(n)$ . Another approach to perform the same task is using Binary Search.
- Binary search is an efficient algorithm for finding an item from a sorted list of items.
- It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.
- A binary search or half-interval search algorithm finds the position of a specified input value (the search key) within sorted array.

# BINARY SEARCH

- For binary search, the array should be arranged in ascending or descending order.
- In each step, the algorithm compares the search key value with the middle element of the array.
- If the key matches, then a matching element has been found and its index or position, is returned.
- Otherwise, if the search key is less than the middle element, then the algorithm repeats its action on the left sub-array of the middle element or, if the search key is greater, then algorithm repeats on the right sub-array.
- If the remaining array to be searched is empty, then the key can not be found in the array and a special "not found" indication is returned.

# BINARY SEARCH

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	H=9
23 > 16 take 2 <sup>nd</sup> half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5	6	M=7	8	H=9
23 > 56 take 1 <sup>st</sup> half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5, M=5	H=6	7	8	9
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91

# BINARY SEARCH ALGORITHM

Algorithm: BINARY\_SEARCH(K, N, X)

- K is an array
- N indicates number of elements in an array
- X indicates value (element) to be searched
- LOW, HIGH and MIDDLE denotes the lower, upper and middle limit of the list.

Step-1) [Initialization]

$LOW \leftarrow 0$

$HIGH \leftarrow N-1$

Step-2) [Perform Search]

Repeat through step 4 while  $LOW \leq HIGH$

Step-3) [Obtain Index of midpoint interval]

$MIDDLE \leftarrow [(LOW + HIGH)/2]$

# BINARY SEARCH ALGORITHM

Step-4) [Compare]

If  $X < K$  [MIDDLE] then

$HIGH \leftarrow MIDDLE - 1$

Else if  $X > K$  [MIDDLE] then

$LOW \leftarrow MIDDLE + 1$

Else

Write "Successful Search"

Return (MIDDLE)

Step-5) [Unsuccessful Search]

Write "Unsuccessful Search"

Return -1

# QUESTIONS!!

- An array  $X[-15 \dots 10, 15 \dots 40]$  requires one byte of storage. If beginning location is 1500, the location of  $X[8, 20]$  would be?
- An array  $P[20][30]$  is stored in the memory along the column with each of the element occupying 4 bytes, find out the memory location for the element  $P[5][15]$  if an element  $P[2][20]$  is stored at the memory location 5000.

Queries ???