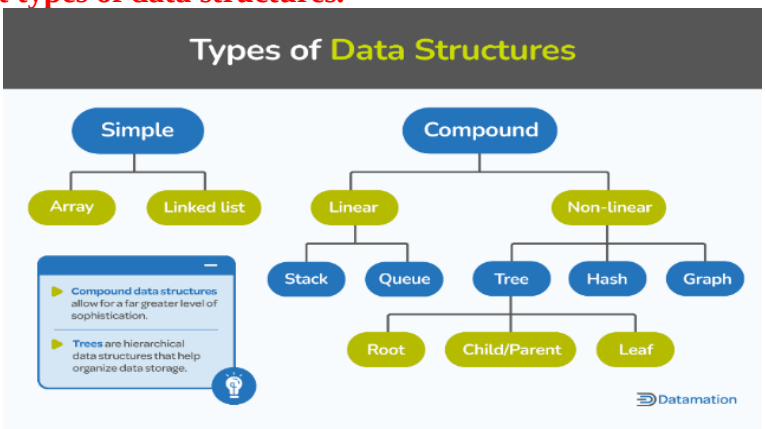
**Freshmen Engineering Department**

**Class : I B.Tech – II-Sem** **Academic Year : 2025- 2026**

**Course Title : Data Structures** **Course Code : 23ES1001**

## MODULE - I (2 Marks)

## INTRODUCTION

| Sno | Questions & Answers |
|---|---|
| 1 | **Define time and space complexity.** |
| Ans | **Time Complexity:**<br>It is the amount of time an algorithm takes to run as a function of input size.<br>**Space Complexity:**<br>It is the amount of memory an algorithm uses during execution. |
| 2 | **State the prerequisite for performing a binary search on a list of elements.** |
| Ans | The list must be sorted in ascending or descending order. The elements should allow direct access using index, so binary search works best on arrays, not linked lists. |
| 3 | **Define Data Structures.** |
| Ans | A data structure is a way of organizing, storing, and managing data in a computer so that it can be used efficiently. |
| 4 | **Write different types of data structures.** |
| Ans |  |
| 5 | **Define Abstract Data type.** |
| Ans | Abstract Data Type (ADT) is a data type that defines what data is stored and what operations can be performed on it, but hides how those operations are implemented from the user. |
| 6 | **Differentiate linear and nonlinear data structures.** |
| Ans | 2b Answer |

| | |
|---|---|
| **7**<br><br>**Ans** | **What is an array? What is the syntax to declare an array?**<br><br>**Array:**<br>An array is a collection of same type of elements stored in continuous memory locations and accessed using an index.<br><br>**Syntax to declare an array:**<br><br>datatype arrayname[size]; |
| **8**<br><br>**Ans** | **List out types of asymptotic notations.**<br><br>• Big-O Notation (O)<br>• Big-Omega Notation (Ω)<br>• Big-Theta Notation (Θ)<br>• Little-O Notation (o)<br>• Little-Omega Notation (ω) |
| **9**<br><br>**Ans** | **Define linear data structure with two examples.**<br>**Linear Data Structure:**<br>A linear data structure stores elements in a **sequential (line-by-line) manner**, where each element is connected to the next one.<br>**Examples:**<br><br>1. **Array** – Stores elements of the same type in contiguous memory locations.<br>2. **Stack** – Follows the **LIFO (Last In First Out)** principle. |
| **10**<br><br>**Ans** | **List out Abstract Data types.**<br><br>**Abstract Data Types (ADTs):**<br><br>1. List ADT – Stores an ordered collection of elements.<br>2. Stack ADT – Follows Last In First Out (LIFO) principle.<br>3. Queue ADT – Follows First In First Out (FIFO) principle.<br>4. Set ADT – Stores a collection of unique elements. |

# 1. Define Data Structures. Explain various types of data Structures.

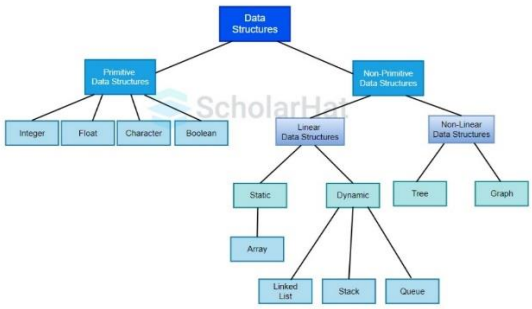
## Data Structure:

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.

## Types of data Structures

There are two types of data structures:

1. Primitive data structure
2. Non-primitive data structure




### 1. Primitive Data Structures

- Primitive data structures are basic and predefined data types
- Primitive data structures directly operated upon by machine instructions.
- They store the data of only one type i.e. built-in data types. Data types like integer, float, character, and Booleans come in this category.

**Example:** int a;

In a variable "a" we can store only one value.

**Integer:** It is used to represent number without decimal part.

**Example:** 2, 5, 78 etc.,

**Float:** It is used to represent number with decimal part.

**Example:** 2.34, 56.78, 78.1 etc.,

**Character:** Character represents a single character (like a letter, symbol, or digit).

**Example:** 'a', 'b', etc.,

**Boolean:** It is used to represent logical values. i.e. true or false

## 2. Non-Primitive Data Structures

Non-Primitive data derived from primitive data structures. They can store multiple values and are more complex. structures are

The non-primitive data structure is divided into two types:

- ❖ Linear data structure
- ❖ Non-linear data structure

**Linear Data Structure:** In linear data structures, elements are stored sequentially, one after another.

**Based on memory allocation, they are divided into:**

a) Static Data Structure b) Dynamic Data Structure

### a) Static Data Structure:

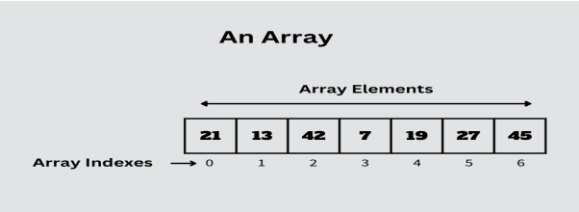
- Size is fixed
- Memory allocated at compile time
- Size cannot be changed

**Example of Static Data structure is Array.**

**Arrays:**

- ➢ Array is a collection of same type of elements
- ➢ Elements are stored in contiguous memory locations
- ➢ Each element is accessed using an index
- ➢ **Example:**

age[0], age[1], age[2], … age[6]




**Array Length** = 7

**First Index** = 0

**Last index** = 6

### b) Dynamic Data Structure:

- Size is not fixed
- Memory allocated at run time
- Size can increase or decrease

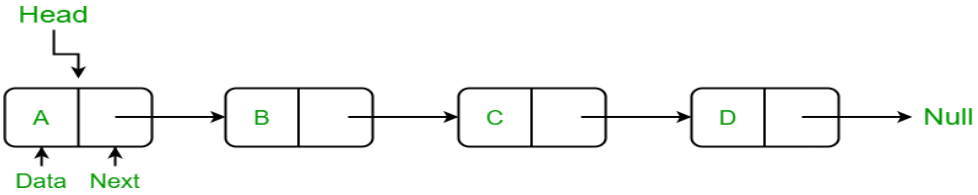
**Examples**

- Linked List
- Stack
- Queue

## Example of Dynamic Data Stucture is linked list, Stack, Queue

### Linked List :

- ✓ Elements are stored as nodes
- ✓ Each node contains data and a link to the next node
- ✓ First node is called Head
- ✓ Last node points to NULL

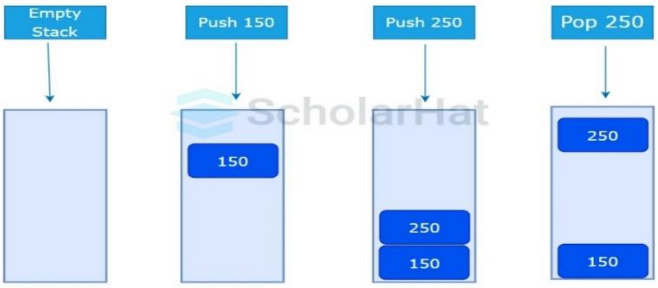



### Stack:

- ✓ Stack follows LIFO / FILO (The order may be LIFO (LAST IN FIRST OUT)(or)FILO (FIRST IN LAST OUT)
- ✓ Insertion and deletion happen at one end (Top)

**LIFO:** Where the last element added is the first to be removed.

**FILO:** Where the first element added is the last to be removed.




## Operations

**Push:** Add an element to the top of the stack.

**Pop:** Remove the top element from the stack.

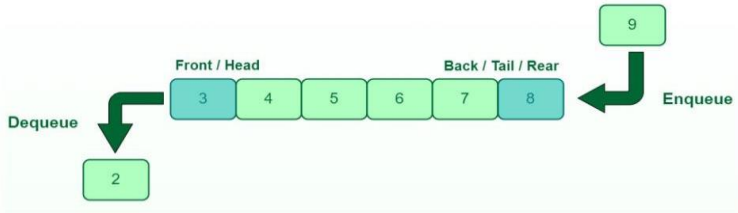
- ✓ **Example:** Stack of books

### Queue:

- ✓ Queue is a linear data structure which elements can be inserted only at one end called **rear** and deleted only at the other end called front.
- ✓ It follows First-In-First-Out (FIFO)
- ✓ In this rule, the element which is added first will be removed first.

**Operations:**

- ✓ **Enqueue** – Insert element
- ✓ **Dequeue** – Remove element



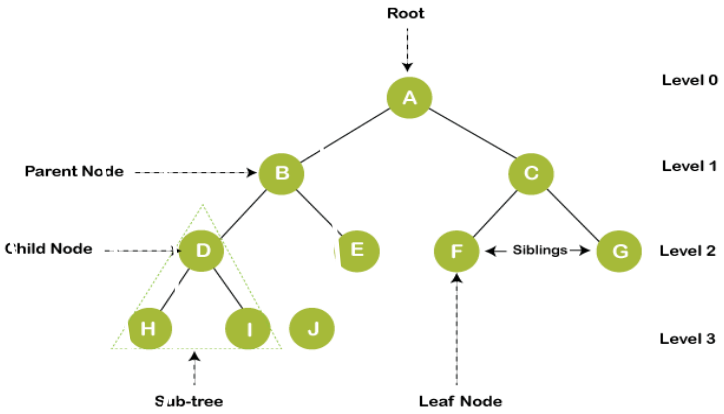

Queue Data Structure

## Non Linear Data Structure:

❖ In non-linear data structures, elements are not stored sequentially and can have multiple connections.

### Trees:

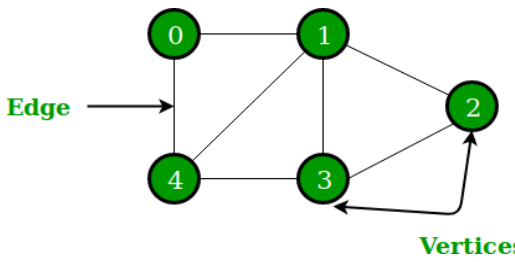
- ✓ Tree is a hierarchical data structure
- ✓ Top node is called Root
- ✓ Nodes have parent and child relationships




### Graphs:

- ✓ Graph consists of vertices (nodes) and edges
- ✓ Edges show the relationship between nodes
- ✓ Can be directed or undirected

.



## 2. a) Define ADT. Give an example for it.

- Abstract Data Type (ADT) is a data type that defines what data is stored and what operations can be performed on it, but hides how those operations are implemented from the user.

### Real-Life Example: TV Remote Control

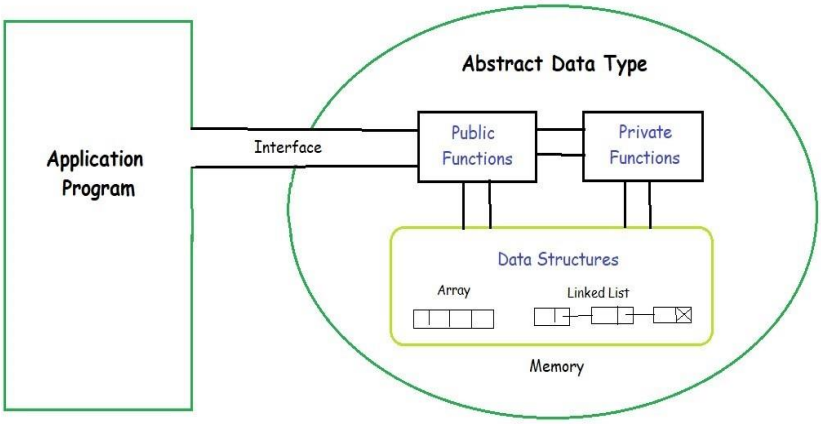
- Using a TV remote, we can perform operations such as:

  Switching the TV ON / OFF

  Increasing or decreasing the volume
- These actions represent the operations available to the user.
- The user:

  Does not know how the TV works internally

  Does not need to know the internal circuits or logic
- The user only interacts with the **buttons on the remote**.




## 1. Application Program

- This is the user program.
- The user writes code here.
- The user does not access data directly.

The application program communicates with ADT only through an interface.

## 2. Interface

- Interface acts as a bridge between:

  Application Program

  Abstract Data Type (ADT)

The user calls functions through this interface.

## 3. Abstract Data Type (ADT)

ADT is shown as a **big boundary** which hides internal details.

Inside ADT, there are **three main parts**:

**Public Functions**

- These are the functions visible to the user.
- The user can call these functions.

**Examples:**

- push()
- pop()
- insert()
- delete()
- get()

These define what operations can be performed.

## 5. Private Functions

- These functions are not visible to the user.
- They are used internally by ADT.
- They help in implementing public functions.

User cannot access these functions directly.

## 6. Data Structures

- Actual data is stored here.
- ADT may use:
  **Array**
  **Linked List**
  or any other data structure

The user does **not know** which data structure is used.

## 7. Memory

- Data structures are stored in memory.
- Memory management is handled internally by ADT.

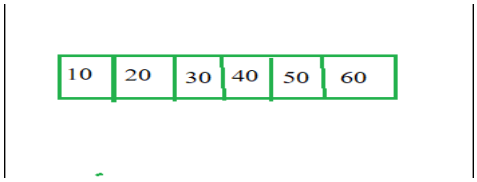
### Types of ADTs

The three commonly used Abstract Data Types (ADTs) are:

1. List ADT
2. Stack ADT
3. Queue ADT

## 1. List ADT

A List stores elements one after another in order.




*Vies of list*

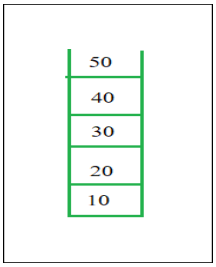
**List Operations**

- **get()** – Get an element from the list
- **insert()** – Add an element to the list
- **remove()** – Delete an element from the list
- **removeAt()** – Delete element at a given position
- **replace()** – Change an element
- **size()** – Number of elements in the list
- **isEmpty()** – Checks whether list is empty
- **isFull()** – Checks whether list is full

## 2.Stack ADT

A Stack works on LIFO

*Last In First Out* (like plates stacked one above another).




*View of stack*

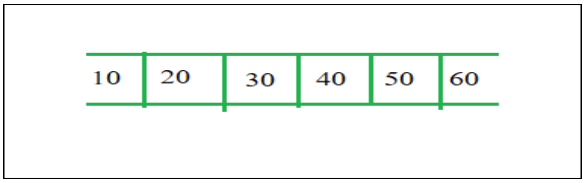
**Stack Operations**

- **push()** – Add element at the top
- **pop()** – Remove element from the top
- **peek()** – See top element
- **size()** – Number of elements
- **isEmpty()** – Checks if stack is empty
- **isFull()** – Checks if stack is full

## 3. Queue ADT

A **Queue** works on **FIFO**

*First In First Out* (like a line at a bus stop).




*View of Queue*

**Queue Operations**

- **enqueue()** – Add element at the end
- **dequeue()** – Remove element from the front
- **peek()** – See first element
- **size()** – Number of elements
- **isEmpty()** – Checks if queue is empty
- **isFull()** – Checks if queue is full

## 2. b) Compare and contrast linear and non-linear data structures.

| S. No | Linear Data Structure | Non Linear Data Structure |
|---|---|---|
| 1 | Data elements are arranged in a single sequence (line) | Data elements are arranged in a hierarchical or network form |
| 2 | Single level structure | Multiple levels structure |
| 3 | Easy to understand and implement | Difficult to understand and implement |
| 4 | Traversal is done in one direction in a single run | Traversal requires multiple paths |
| 5 | Memory usage is less efficient | Memory usage is more efficient |
| 6 | Used for simple data storage | Used for complex data representation |
| 7 | Examples: Array, Stack, Queue, Linked List | Examples: Tree, Graph |

## 3. a) List out the advantages of Abstract data type?

**Hides details**

We don't need to know *how* data is stored. We only use operations like insert, delete, display.

**Protects data**

Data cannot be changed directly. It can be changed only using given functions → safer.

**Easy to understand programs**

Big programs are divided into small parts, so programs become simple.

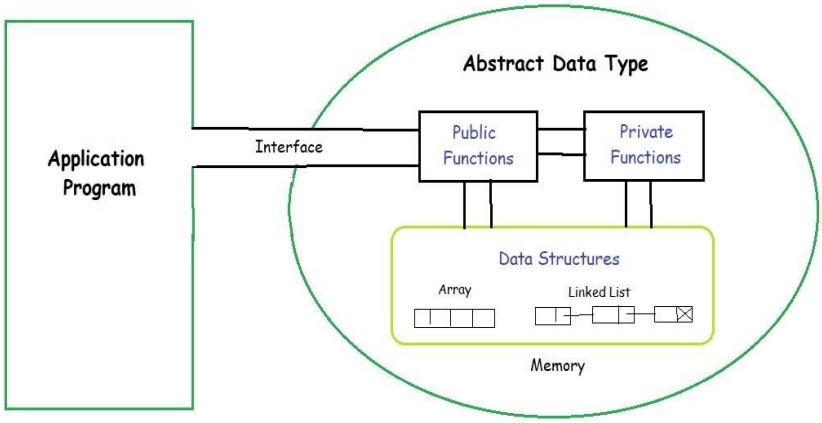
**Easy to change**

If we change the internal code, the program still works without changes.

**Reuse code**

Same ADT (like Stack, Queue) can be used in many programs.

## 3. b) Explain Characteristics of ADTs.




### 1. Application Program

- This is the user program.
- The user writes code here.
- The user does not access data directly.

The application program communicates with ADT only through an interface.

### 2. Interface

- Interface acts as a bridge between:

  Application Program

  Abstract Data Type (ADT)

The user calls functions through this interface.

### 3. Abstract Data Type (ADT)

ADT is shown as a **big boundary** which hides internal details.

Inside ADT, there are **three main parts**:

**Public Functions**

- These are the functions visible to the user.
- The user can call these functions.

**Examples:**

- push()
- pop()
- insert()
- delete()
- get()

These define what operations can be performed.

## 5. Private Functions

- These functions are not visible to the user.
- They are used internally by ADT.
- They help in implementing public functions.

User cannot access these functions directly.

## 6. Data Structures

- Actual data is stored here.
- ADT may use:
  Array
  Linked List
  or any other data structure

The user does not know which data structure is used.

## 7. Memory

- Data structures are stored in memory.
- Memory management is handled internally by ADT.

## 4 a) Briefly discuss various asymptotic notations with examples.

### Asymptotic Notations

- ❖ Asymptotic notations are used to measure time complexity of an algorithm.
- ❖ They describe how an algorithm's running time grows as the input size increases.
- ❖ Constants and lower-order terms are ignored in asymptotic analysis.
- ❖ Asymptotic notations are used to find best case, average case, and worst-case time complexity.

**They are as follows**

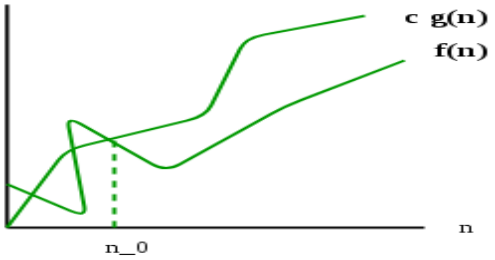
1. Big - Oh (O)
2. Big - Omega (Ω)
3. Big - Theta (Θ)
4. Little oh Notation (o)
5. Little omega Notation (w)

### Big - Oh Notation (O)

- Represented by O**.**
- Shows the upper bound of an algorithm's running time.
- It tells the maximum time an algorithm can take.
- Used for worst case time complexity**.**
- .

**Definition:**

Let **f(n)** and **g(n)** be two non-negative functions. **f(n)** is said to be **O(g(n)),** if and only if there exists two positive constants **c** and **'$n_0$'** such that, **$f(n) \leq c * g(n)$** for all non-negative values of n, where, **$n \geq n_0$**




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

**Example**

| **Let f(n)=3n+2 and g(n)=4n** | |
|---|---|
| Substituting n=1 | Substituting n=2 |
| f(n)=3n+2 | f(n)=3n+2 |
| f(1)=3(1)+2 = 5 | f(2)=3(2)+2 = 8 |
| g(n)=4n | g(n)=4n |
| g(1)=4(1)=4 | g(2)=4(2)=8 |
| Since f(n)>g(n) | Since f(n)=g(n) |
| The above condition fails | n=2 satisfy the above condition |
| n=1 does not satisfy the above condition | 3n+2 ≤ 4(n) |
| [Where c=4, g(n)=n, and $n_0$=2]<br>Hence **f(n) ≤ O(g(n))**<br>Therefore, the time complexity for f(n) = 3n+2 is O(n) | |

## Big - Omega Notation (Ω)

- Big-Omega notation is represented by the symbol **Ω**.
- It shows the lower bound of an algorithm's running time.
- It tells the minimum time an algorithm will take.
- It is used to represent the best case time complexity**.**

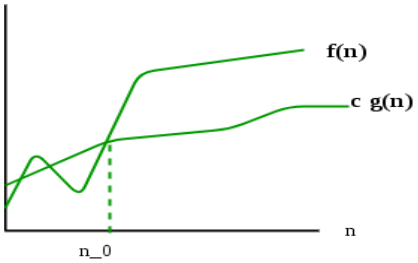
Big - Omega Notation can be defined as follows...

**f(n) = Ω(g(n))**

**Definition:** Let **f(n)** and **g(n)** be two non-negative functions.

We say **f(n) = Ω(g(n))** if there exist two positive constants **c** and $\mathbf{n_0}$ such that:

**f(n) ≥ c × g(n)** for all **n ≥** $\mathbf{n_0}$




f(n) = Omega(g(n))

**Example:**

f(n) = 3n+2 g(n) = n **f(n) = Ω(g(n))**

**f(n) ≥c*g(n) for all, n≥n0**

3n+2 ≥ c*n (let c=1)

3n+2 ≥ 1*n (n=1)

3(1)+2≥1*1

3+2≥1

5≥1 (condition satisfied)

f(n) ≥c*g(n) for all, n≥1

## Big - Theta Notation (Θ)

- Big-Theta notation is represented by the symbol **Θ.**
- It shows the exact or average running time of an algorithm.
- It gives both upper and lower bounds of time complexity.
- It is used to represent the average case time complexity.

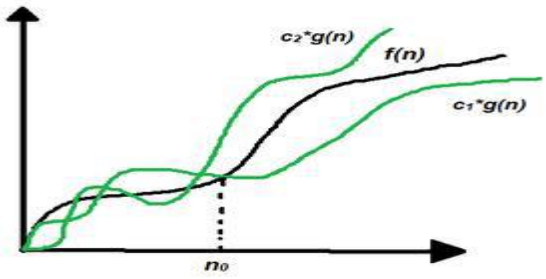
$$f(n) = \Theta(g(n))$$

**Definition:**

Let **f(n)** and **g(n)** be two non-negative functions.

We say **f(n) = Θ(g(n))** if there exist positive constants $c_1$, $c_2$, and $n_0$ such that:

$c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for all $n \geq n_0$




**Example:**

**c1*g(n) ≤f(n) ≤c2*g(n)**

f(n)=3n+2 g(n)=n

$c_1 = 1$ $c_2 = 4$

1*n ≤ 3n+2 ≤ 4*n

1*1 ≤ 3(1)+2 ≤ 4*1 (let n=1)

1≤ 5 ≤ 4 (false)

1*2 ≤ 3(2)+2 ≤ 4*2 (let n=2)

2 ≤ 8 ≤ 8 (condition true)

## 4 b) Write an algorithm for determining the transpose of a matrix using a multi-dimensional array.

**Definition:** The transpose of a matrix is a new matrix obtained by interchanging rows and columns of the original matrix.

**Example**

**Original Matrix A**

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

## Transpose Matrix $A^T$

$$A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

**Algorithm: Transpose of a Matrix**

**Step 1:** Start

**Step 2:** Read number of rows **m** and columns **n**

**Step 3:** Declare two 2-D arrays

Matrix **A[m][n]** and Transpose **T[n][m]**

**Step 4:** Read all elements of matrix **A**

**Step 5:** For **i = 0 to m−1**

For **j = 0 to n−1**

Set **T[j][i] = A[i][j]**

**Step 6:** Display the transpose matrix **T**

**Step 7:** Stop**.**

**Source Code:**

```
#include <stdio.h>
int main() {
  int a[10][10], transpose[10][10], r, c;
  printf("Enter rows and columns: ");
  scanf("%d %d", &r, &c);
  // asssigning elements to the matrix
  printf("\nEnter matrix elements:\n");
  for (int i = 0; i < r; i++)
  for (int j = 0; j < c; j++) {
    printf("Enter element a%d%d: ", i + 1, j + 1);
    scanf("%d", &a[i][j]);
  }
  // printing the matrix a[][]
  printf("\nEntered matrix: \n");
  for (int i = 0; i < r; i++)
  for (int j = 0; j < c; j++) {
    printf("%d  ", a[i][j]);
    if (j == c - 1)
    printf("\n");
  }
  // computing the transpose
  for (int i = 0; i < r; i++)
  for (int j = 0; j < c; j++) {
    transpose[j][i] = a[i][j];
  }
  // printing the transpose
  printf("\nTranspose of the matrix:\n");
  for (int i = 0; i < c; i++)
  for (int j = 0; j < r; j++) {
```

```
    printf("%d  ", transpose[i][j]);
    if (j == r - 1)
    printf("\n");
  }
 return 0;
}
```

Output:

```
Enter rows and columns: 3
2

Enter matrix elements:
Enter element a11: 4
Enter element a12: 5
Enter element a21: 8
Enter element a22: 9
Enter element a31: 7
Enter element a32: 8

Entered matrix:
4  5
8  9
7  8

Transpose of the matrix:
4  8  7
5  9  8
```

## 5) Explain in brief about multi-dimensional arrays with an example.

**Types of Arrays**

Arrays are used to store multiple values of the same data type in a single variable.

There are two types of arrays:

Single Dimensional Array

Multi-Dimensional Array

**a) Single Dimensional Array**

A single dimensional array is used to store data in a linear form.

- It uses only one subscript (index)
- It is also called a linear array
- Elements are stored in one row

**Syntax**

datatype array_name[size];

**Example**

int mark[5] = {40, 60, 80, 70, 90};

Here,

- mark is the array name
- 5 is the size of the array
- Values are stored in one line

**b) Multi-Dimensional Array**

A multi-dimensional array is an array that has more than one dimension.

It is an array of arrays.

**Examples:**

- Two-Dimensional Array (2-D)
- Three-Dimensional Array (3-D)

**General Syntax**

datatype array_name[size1][size2]...[sizen];

**i) Two-Dimensional Array (2-D Array)**

A 2-D array stores data in the form of rows and columns.

It is also called a matrix.

**Syntax**

datatype array_name[rows][columns];

Where:

- datatype → type of data
- rows → number of rows
- columns → number of columns

**Example**

int a[2][3];

This means:

- 2 rows
- 3 columns
- Total elements = 2 × 3 = 6

**Initialization of 2-D Array**

```
int a[3][4] =
{
 {1, 2, 3, 4},
 {5, 6, 7, 8},
 {9, 10, 11, 12}
};
```

This array has:

- 3 rows
- 4 columns

**ii) Three-Dimensional Array (3-D Array)**

A 3-D array is an array of 2-D arrays.

It is used to store data in three dimensions.

**Syntax**

datatype array_name[x][y][z];

**Example**

```
int arr[3][3][3] =
{
 {{11,12,13},{14,15,16},{17,18,19}},
 {{21,22,23},{24,25,26},{27,28,29}},
 {{31,32,33},{34,35,36},{37,38,39}}
};
```

This array contains **27 elements**.

## 6) Describe the significance of time and space complexity with an example.

**Introduction**

Time and Space Complexity are used to analyze the efficiency of an algorithm. They help us understand how much time and memory an algorithm requires as the input size increases.

### 1) Time Complexity

Time complexity means how much time an algorithm takes to run.

**Why Time Complexity is Important**

- It tells how fast the program works
- It helps to compare two algorithms
- It shows what happens when input size becomes large
- It helps to choose a faster algorithm
- It reduces waiting time

**Example (Time Complexity)**

**Linear Search**

• If element is found first → **O(1)** (best case)
• If element is found last → **O(n)** (worst case)
This shows time changes with input size.

### 2) Space Complexity

Space complexity means how much memory an algorithm uses.

**Why Space Complexity is Important**

- It tells how much memory is needed
- Useful for computers with less memory
- It helps to save memory
- It makes the program efficient

**Example (Space Complexity)**

```
int sum = 0;
```
• Uses one variable → **O(1)** space

```
int arr[n];
```
• Uses array of size n → **O(n)** space

## 7) Explain Linear search algorithm in detail with an Example Program

**Linear Search Algorithm**

Linear search is a simple searching technique.
It searches an element by checking each element one by one from the beginning of the array until the element is found or the array ends.

**Algorithm: Linear Search**

**Step 1:** Start
**Step 2:** Read number of elements n
**Step 3:** Read array elements
**Step 4:** Read search element key

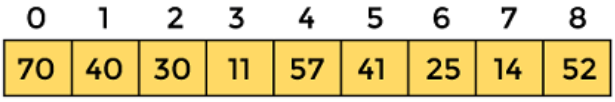
**Step 5:** Compare key with each element of array
**Step 6:**
If match found → print position and stop
Else continue searching
**Step 7:** If element not found → print "Not Found"
**Step 8:** Stop

## Working of Linear search

Now, let's see the working of the linear search Algorithm.

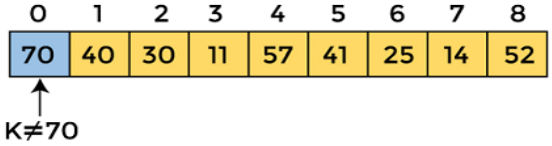
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

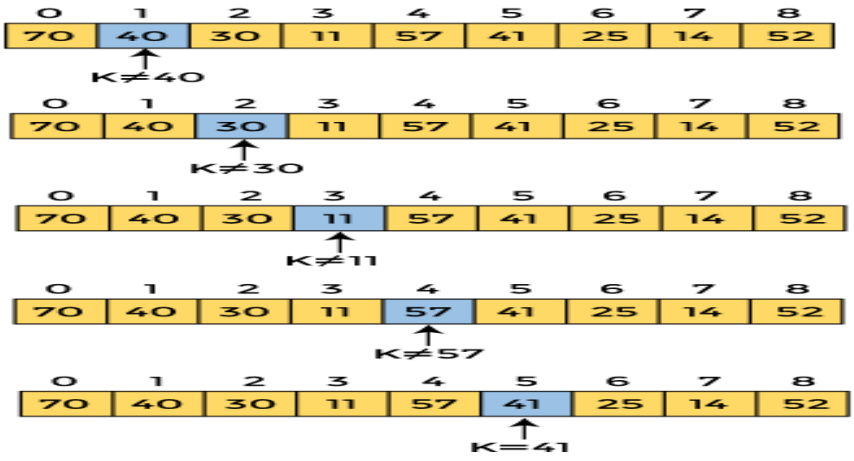



Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.




The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.




Now, the element to be searched is found. So algorithm will return the index of the element matched.

## Example:

```
#include <stdio.h>
int main()
{
   int a[10], n, key, i;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   printf("Enter array elements:\n");
   for(i = 0; i < n; i++)
      scanf("%d", &a[i]);
   printf("Enter element to search: ");
   scanf("%d", &key);
   for(i = 0; i < n; i++)
   {
      if(a[i] == key)
      {
         printf("Element found at position %d", i + 1);
         return 0;
      }
   }
   printf("Element not found");
   return 0;
}
```

## 8) Write a Program to reverse the elements of an array.

**Definition:** Reversing the elements of an array means rearranging the elements such that the first element becomes the last, the second element becomes the second-to-last, and so on, until the entire array is reversed.

**Algorithm**

**Step 1:** Start
**Step 2:** Read the size of the array
**Step 3:** Read array elements
**Step 4:** Print the elements from last index to first index
**Step 5:** Stop

**Source Code:**

```
#include <stdio.h>
void main()
{
   int s[10], size, i;
   scanf("%d", &size);

   for(i = 0; i < size; i++)
      scanf("%d", &s[i]);
```

```
    for(i = size - 1; i >= 0; i--)
        printf("%d ", s[i]);
}
```
**Input:**

10 20 30 40 50

**Output:**

50 40 30 20 10

## 9) Explain various operations of arrays.

An array is a collection of elements of the same data type stored in contiguous memory locations. Various operations are performed on arrays to store, access, modify, and process data.

### 1) Traversing an Array

Accessing and processing each element of the array one by one.

**Example:**

Printing all elements of an array.

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

### 2) Insertion in an Array

Adding a new element at a specific position in the array.

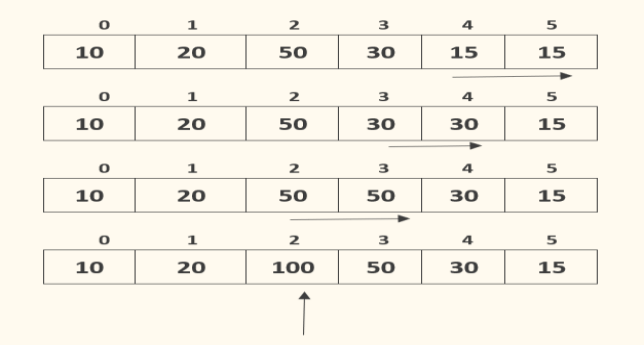
**Steps:**

- Shift elements to the right
- Insert the new element at required position

**Example:** Consider linear array A as below:

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 10 | 20 | 50 | 30 | 15 |

New element to be inserted is 100 and location for insertion is 2. So, shift the elements from 4th location to 2nd location downwards by 1 place. And then insert 100 at 3rd location. It is shown below:



## 3) Deletion from an Array

Removing an element from a given position.

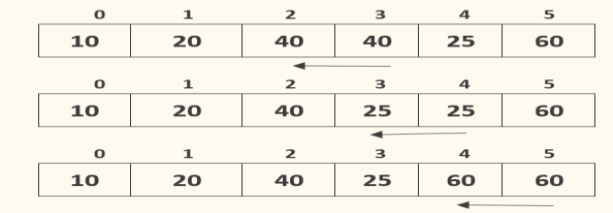
**Steps:**

- Remove the element
- Shift remaining elements to the left

## Example:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 10 | 20 | 50 | 40 | 25 | 60 |

The element to be deleted is 50 which is at 2nd location. So shift the elements from 3rd to 5th location upwards by 1 place. It is shown below:




After deletion the array will be:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 10 | 20 | 40 | 25 | 60 | |

## 4) Searching in an Array

Finding the **position of a required element** in the array.

**Methods:**

- Linear Search
- Binary Search

.

## Example:

We have linear array A as below:

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 15 | 50 | 35 | 20 | 25 |

Suppose item to be searched is 20. We will start from beginning and will compare 20 with each element. This process will continue until element is found or array is finished. Here:

1) **Compare 20 with 15**

20 # 15, go to next element.

2) **Compare 20 with 50**

20 # 50, go to next element.

3) **Compare 20 with 35**

20 #35, go to next element.

4) **Compare 20 with 20**

20 = 20, so 20 is found and its location is 4.

## 5) Sorting an Array

Arranging elements in ascending or descending order.

**Example:**

10 40 30 20 → 10 20 30 40

## 6) Merging Arrays

Combining two arrays into one.

**Example:**

A = {1,2} and B = {3,4} → C = {1,2,3,4}

## 7) Reversing an Array

Arranging elements in reverse order.

**Example:**

10 20 30 → 30 20 10

## Advantages of Array Operations

- Easy to store large data
- Fast access using index
- Simple data processing