1010101010
0101010101
1010101010
0101010101

Infographics

Visual Learning
Better understanding through visual content.

Simplified Concepts
Complex topics explained in simple and visual form.

Quick Revision
Easy to revise important points and diagrams.

Exam Ready
Helps in faster revision and better exam preparation.

DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS

Unit-I


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using nameepace std;
int main()
{
cout << "Hello, World!";
return 0;
}

DATA STRUCTURE

— INTRODUCTION TO DATA STRUCTURE —

A **Data Structure** is a way of **organizing** and **storing** data in a computer so that it can be used **efficiently**.

WHAT IS DATA STRUCTURE?

A data structure is a **collection** of data values, the **relationships** among them, and the **functions** or **operations** that can be applied to the data.

In simple words:

It is a way to **store**, **organize** and manage data in memory so that it can be accessed and modified efficiently.

WHY DO WE NEED DATA STRUCTURES?

- **Efficiency** – Helps in processing data faster and using memory wisely.
- **Organization** – Keeps data well structured and easy to manage.
- **Easy Access** – Allows quick searching, insertion, deletion and other operations.
- **Problem Solving** – Essential for solving complex real-world problems.

CHARACTERISTICS OF DATA STRUCTURE

Organization
Data is organized in a specific manner.

Storage
Data is stored in computer memory.

Operations
Supports various operations like insert, delete, search, update.

Efficiency
Designed to perform operations in the most efficient way possible.

Reusability
Data structures can be reused in different applications.

TYPES OF DATA STRUCTURES

Data structures are broadly classified into two types:

1. Linear Data Structure

Elements are arranged in a sequential manner.

Examples:

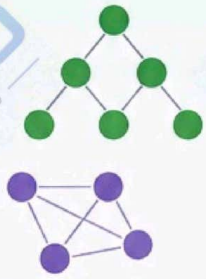
- Array
- Linked List
- Stack
- Queue

2. Non-Linear Data Structure

Elements are not arranged sequentially.

Examples:

- Tree
- Graph
- Heap



COMMON OPERATIONS

- **Insert** – Add a new element into the data structure.
- **Delete** – Remove an element from the data structure.
- **Search** – Find the location of an element.
- **Update** – Modify the value of an existing element.
- **Traverse** – Access each element exactly once.

KEY TAKEAWAY Data structures are the foundation of efficient programming. Choosing the right data structure for the right problem makes programs faster, simpler and more effective.

TYPES OF DATA STRUCTURE

Primitive & Non-Primitive

Data structures are of **two main types** based on the type of data they store and the operations that can be performed.

1. PRIMITIVE DATA STRUCTURE

Primitive data structures are the basic data types that store a **single value**.

Examples

123	3.14	'A'	True
Integer (int)	Float (float)	Character (char)	Boolean (bool)

Features

- Store a single value.
- Built-in data types of a programming language.
- Directly supported by the system.
- Fixed size and simple.

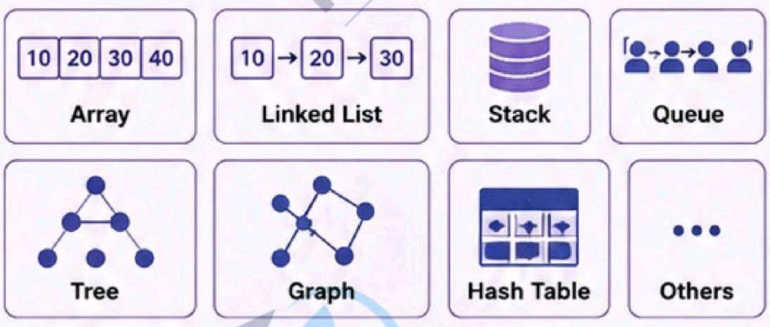
Examples in C/Java

```
int a = 10;    float b = 3.5;    char ch = 'A';    boolean flag = true;
```

2. NON-PRIMITIVE DATA STRUCTURE

Non-primitive data structures are derived types that can store **multiple values**.

Examples



Features

- Store multiple values (collection of data).
- User-defined or derived data types.
- More complex.
- Require more memory.
- Can represent relationships between data.

Examples in C/Java

```
int arr[4] = {10, 20, 30, 40};                    // Array
struct Node { int data; struct Node* next; };     // Linked List
Stack s;    Queue q;    Tree t;    Graph g;
```

PRIMITIVE vs NON-PRIMITIVE DATA STRUCTURE

Aspect	Primitive Data Structure	Non-Primitive Data Structure
Definition	Stores a single value.	Stores multiple values (collection of data).
Built-in / User-defined	Built-in (predefined).	User-defined.
Size	Fixed size.	Variable size.
Complexity	Simple.	Complex.
Memory	Less memory.	More memory.
Examples	int, float, char, bool	Array, Linked List, Stack, Queue, Tree, Graph, etc.
Supported Operations	Basic operations only.	Many operations (insert, delete, search, traverse, etc.).

SUMMARY

Primitive data structures hold a single value and are simple.
Non-primitive data structures hold multiple values and help in organizing complex data efficiently.

LINEAR AND NON-LINEAR DATA STRUCTURE

Data structures are broadly classified based on the **arrangement** of elements and the **relationships** between them.

1. LINEAR DATA STRUCTURE

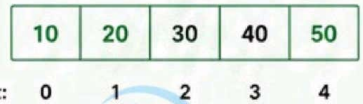
In linear data structures, elements are arranged in a **sequential order**. Each element (except the first and last) has exactly **one predecessor** and **one successor**.

Characteristics

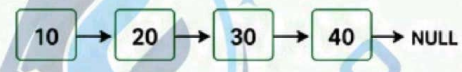
- Elements are stored in a sequence.
- There is a one-to-one relationship between elements.
- Traversal is done in a single path.

Examples

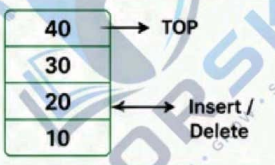
- **Array**



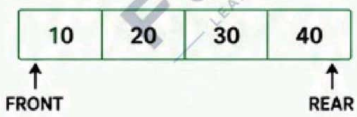
- **Linked List**



- **Stack**



- **Queue**



Applications

Managing lists of data

Function calls, undo operations

CPU scheduling, printer queue

2. NON-LINEAR DATA STRUCTURE

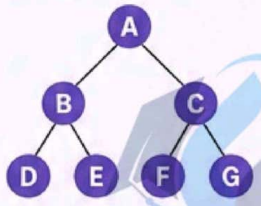
In non-linear data structures, elements are **not** arranged in a **sequential order**. One element can be connected to **multiple other elements**.

Characteristics

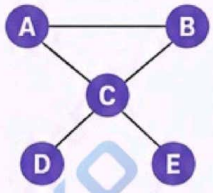
- Elements are not stored in sequence.
- There is a one-to-many or many-to-many relationship.
- Traversal may have multiple paths.

Examples

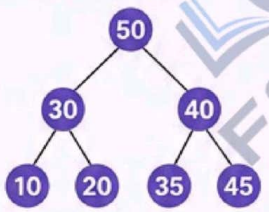
- **Tree**



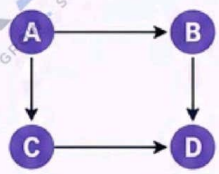
- **Graph**



- **Heap**



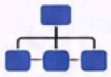
- **Graph (Directed)**



Applications

File systems, organization hierarchies

Network routing, social networks



Compilers, AI, databases

LINEAR vs NON-LINEAR DATA STRUCTURE

Aspect	Linear Data Structure	Non-Linear Data Structure
Arrangement	Elements are arranged in a sequence.	Elements are not arranged in sequence.
Relationship	One-to-one relationship.	One-to-many or many-to-many relationship.
Traversal	Single path.	Multiple paths.
Complexity	Generally simpler.	Generally more complex.
Memory Usage	Less memory (in most cases).	More memory (due to links/pointers).
Examples	Array, Linked List, Stack, Queue.	Tree, Graph, Heap, etc.
Applications	Simple applications, linear processing.	Complex applications, hierarchical relationships.

KEY TAKEAWAY Use linear data structures when data is sequential.
Use non-linear data structures when data has hierarchical or network relationships.

DATA STRUCTURE OPERATIONS

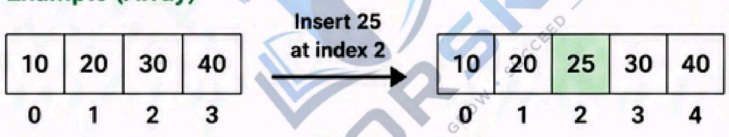
Operations are the basic actions that can be performed on data structures to manipulate the data stored in them.

COMMON OPERATIONS

1 INSERT

Insertion is the process of adding a new element at a specific position in the data structure.

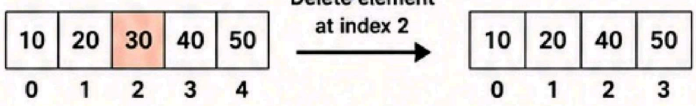
Example (Array)



2 DELETE

Deletion is the process of removing an element from the data structure.

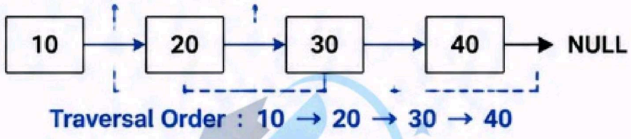
Example (Array)



3 TRAVERSE

Traversal is the process of visiting each element of the data structure exactly once in a particular order.

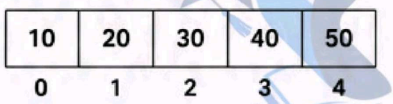
Example (Linked List)



4 SEARCH

Searching is the process of finding the location (index or address) of an element in the data structure.

Example (Array)



Search key = 40
Found at index 3.

5 UPDATE

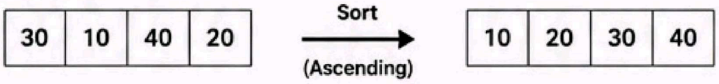
Updating is the process of modifying the value of an existing element.

Example (Array)

6 SORT

Sorting is the process of arranging elements in ascending of descending order.

Example (Array)



SUMMARY TABLE

Operation	Purpose	Where Used
1 Insert	Add a new element	Array, Linked List, Stack, Queue, Tree, Graph, etc.
2 Delete	Remove an existing element	All data structures
3 Traverse	Visit each element exactly once	Linked List, Tree, Graph, Array, etc.
4 Search	Find the position of an element	Array, Linked List, Tree, Graph, etc.
5 Update	Modify an existing element	Array, Linked List, Tree, Graph, etc.
6 Sort	Arrange elements in order	Array, List, etc.

KEY TAKEAWAY

These basic operations allow efficient manipulation of data in different data structures. The choice of data structure depends on the type of operations to be performed and the problem to be solved.

ARRAY

Definition and Concepts, Memory Representations

1. DEFINITION

An **array** is a collection of **similar data elements** stored in **contiguous memory locations** and accessed using a **common name**.

2. CONCEPTS

- All elements are of the same data type.
- Elements are stored in contiguous memory locations.
- Elements are accessed using an index (or subscript).
- The index of the first element is 0.
- For an array of size N, valid indices are 0 to N–1.
- Direct access is possible. Any element can be accessed in constant time O(1).

3. BASIC TERMINOLOGIES

- **Element** : Each value stored in an array.
- **Index** : Position of an element in the array.
- **Size** : Total number of elements in the array.
- **Array Name** : Name used to represent the array.
- **Base Address** : Address of the first memory location of the array.
- **Contiguous** : Consecutive memory locations.

Example

Let A be an array of size 5.

A = [	10	20	30	40	50]
Index:	0	1	2	3	4

Access:

A[0] = 10
A[2] = 30
A[4] = 50

4. MEMORY REPRESENTATIONS OF ARRAY

4.1 ONE-DIMENSIONAL ARRAY (1D)

An array of elements arranged in a single row.

Example: A[5] = { 10, 20, 30, 40, 50 }

Array Elements	10	20	30	40	50
Index	0	1	2	3	4
Memory Address	1000	1004	1008	1012	1016

- Assume base address (BA) = 1000
- If each integer occupies 4 bytes.

4.2 TWO-DIMENSIONAL ARRAY (2D)

An array of elements arranged in rows and columns (matrix form).

Example: A[3][4]

Row Index (i) \ Column Index (j)	0	1	2	3
0	10	20	30	40
1	50	60	70	80
2	90	100	110	120

Access:

A[0][0] = 10
A[1][2] = 70
A[2][3] = 120

- Stored in memory in **row-major order** (row by row).

4.3 ADDRESS CALCULATION

Let,
BA = Base Address of array
W = Size of each element in bytes
i, j = Row and Column indices
N = Number of columns

(a) 1D Array

Address of A[i]

$$= BA + i \times W$$

Where, $0 \le i < N$

(b) 2D Array (Row-Major Order)

Address of A[i][j]

$$= BA + ((i \times N) + j) \times W$$

Where, $0 \le i < Rows$, $0 \le j < Columns$

Note:

Row-major order stores row by row in contiguous memory locations.

5. EXAMPLE: ADDRESS CALCULATION

(a) 1D Array

A[5], BA = 1000, W = 4 bytes
Find address of A[3]
Address = 1000 + 3 × 4 = 1012

(b) 2D Array

A[3][4], BA = 2000, W = 4 bytes, N = 4 (columns)
Find address of A[2][3]
Address = 2000 + ((2 × 4) + 3) × 4
= 2000 + (11 × 4) = 2044

6. CHARACTERISTICS OF ARRAY

- Fixed size (static memory allocation).
- Homogeneous elements (same data type).
- Contiguous memory allocation.
- Random access: O(1) time.
- Efficient traversal using loops.
- Insertion and deletion are costly (because of shifting).

7. COMMON OPERATIONS

- Traversal (visit each element)
- Insertion (at specific position)
- Deletion (at specific position)
- Searching (linear / binary)
- Sorting (ascending / descending)
- Updating (modify element)

KEY TAKEAWAY

Arrays store similar elements in contiguous memory locations and provide direct access to any element using its index. Understanding their memory representation is essential for efficient programming.

ARRAY OPERATIONS

Traversing, Insertion, Deletion

An array is a collection of **similar elements** stored in **contiguous memory** locations and accessed using an **index**.

1. TRAVERSING AN ARRAY

Traversing means **visiting each element** of the array exactly once in a specific order (usually from first to last).

Example

Let A = | 10 | 20 | 30 | 40 | 50 |

Index: 0 1 2 3 4

Traversal Order: 10 → 20 → 30 → 40 → 50

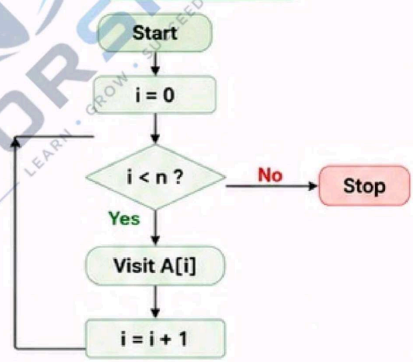
ALGORITHM (TRAVERSAL)

```
for i = 0 to n-1
    visit A[i]
end for
```

PSEUDOCODE

```
for i ← 0 to n - 1
    print A[i]
end for
```

Flow / Illustration



2. INSERTION IN AN ARRAY

Insertion means **adding a new element** at a specific position. Elements from that position are shifted one place to the right.

Example

Insert 25 at index 2 in A = [10 | 20 | 30 | 40 | 50] (Size = 5)

Step 1: Shift elements to the right from index 2

Index:	0	1	2 →	3 →	4 →
	10	20	30	40	50

Step 2: Insert 25 at index 2

Index:	0	1	(2)	3	4	5
	10	20	25	30	40	50

ALGORITHM (INSERTION)

1. Check if the array is full.
2. For i = n - 1 down to pos
   A[i + 1] = A[i]
3. A[pos] = newElement
4. n = n + 1

PSEUDOCODE

```
if n == MAX
    print "Array Overflow"
else
    for i ← n-1 downto pos
        A[i+1] ← A[i]
    A[pos] ← newElement
    n ← n + 1
end if
```

Illustration

Before Insertion

Index:	0	1	2	3	4
	10	20	30	40	50

↑ Insert 25 at index 2

After Insertion

Index:	0	1	2	3	4	5
	10	20	25	30	40	50

3. DELETION FROM AN ARRAY

Deletion means **removing an element** from a specific position. Elements after that position are shifted one place to the left.

Example

Delete element at index 2 (value 30) from A = [10 | 20 | 30 | 40 | 50] (Size = 5)

Step 1: Shift elements to the left from index 2

Index:	0	1	2 →	3 →	4
	10	20	30	40	50

Step 2: Reduce size by 1

Index:	0	1	2	3
	10	20	40	50

ALGORITHM (DELETION)

1. Check if the array is empty.
2. For i = pos to n - 2
   A[i] = A[i + 1]
3. n = n - 1

PSEUDOCODE

```
if n == 0
    print "Array Underflow"
else
    for i ← pos to n-2
        A[i] ← A[i+1]
    n ← n - 1
end if
```

Illustration

Before Deletion

Index:	0	1	2	3	4
	10	20	30	40	50

↑ Delete element at index 2 (30)

After Deletion

Index:	0	1	2	3	
	10	20	40	50	

KEY TAKEAWAY

- Traversing visits each element of the array.
- Insertion adds a new element and may require shifting.
- Deletion removes an element and may require shifting.