CHAPTER 1:
INTRODUCTION TO DATA SRTUCTURE

Content

Introduction to DS

✅ Definition:

A **Data Structure** is a way of organizing and storing data so that it can be accessed and modified efficiently.

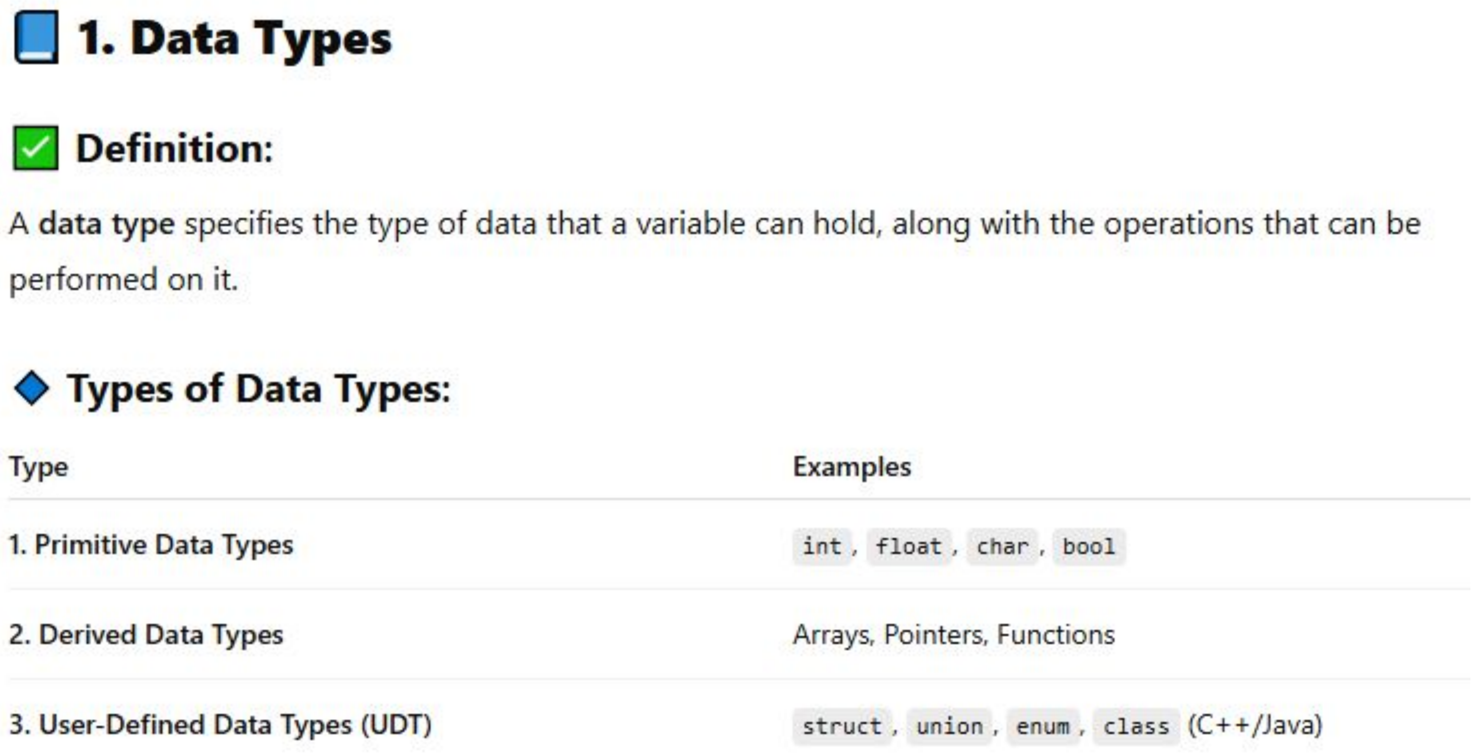
It defines:

- How data is stored in memory
- What operations can be performed on the data
- How fast these operations are (time complexity)

🧠 Need for Data Structures:

- To manage large amounts of data efficiently
- To perform operations like searching, sorting, insertion, deletion, etc., quickly
- To improve performance and optimize memory usage
- To make problem-solving easier and more structured

DATA TYPE



📘 1. Data Types

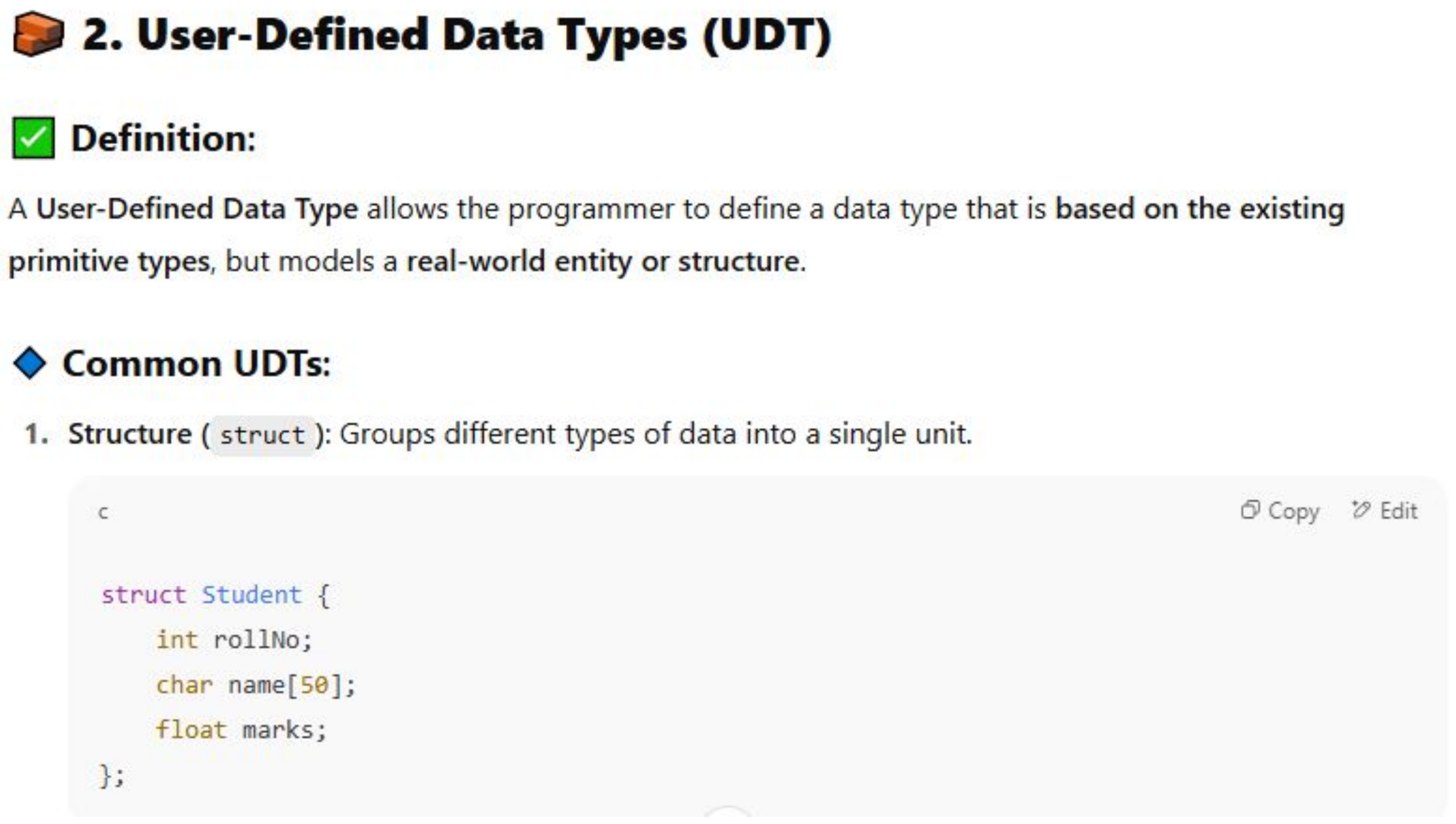
✅ Definition:

A **data type** specifies the type of data that a variable can hold, along with the operations that can be performed on it.

🔷 Types of Data Types:

Type	Examples
1. Primitive Data Types	`int`, `float`, `char`, `bool`
2. Derived Data Types	Arrays, Pointers, Functions
3. User-Defined Data Types (UDT)	`struct`, `union`, `enum`, `class` (C++/Java)

DATA TYPE



🧱 2. User-Defined Data Types (UDT)

✅ Definition:

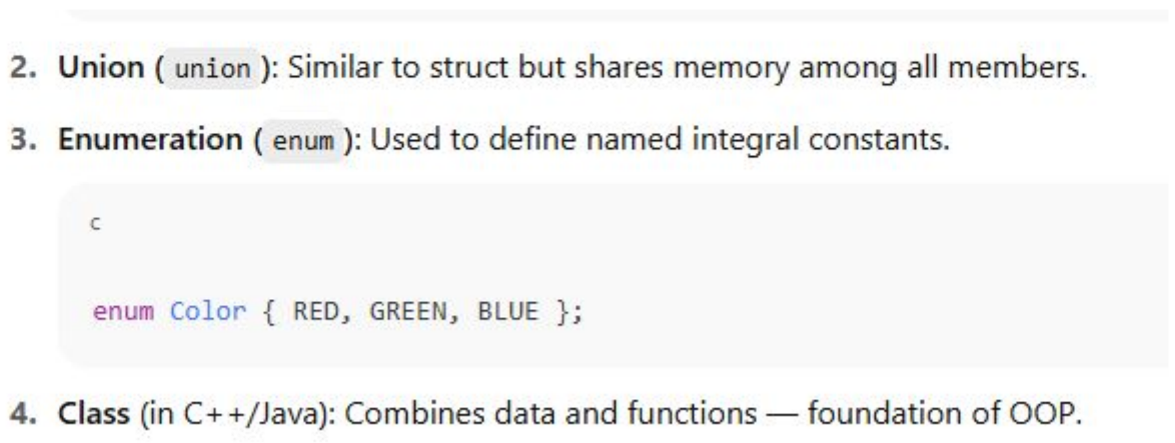
A **User-Defined Data Type** allows the programmer to define a data type that is **based on the existing primitive types**, but models a **real-world entity or structure**.

🔷 Common UDTs:

1. **Structure** (`struct`): Groups different types of data into a single unit.

```c
struct Student {
    int rollNo;
    char name[50];
    float marks;
};
```

DATA TYPE



2. **Union** (`union`): Similar to struct but shares memory among all members.
3. **Enumeration** (`enum`): Used to define named integral constants.

```c
enum Color { RED, GREEN, BLUE };
```

4. **Class** (in C++/Java): Combines data and functions — foundation of OOP.

Abstract Data type

✅ Definition:

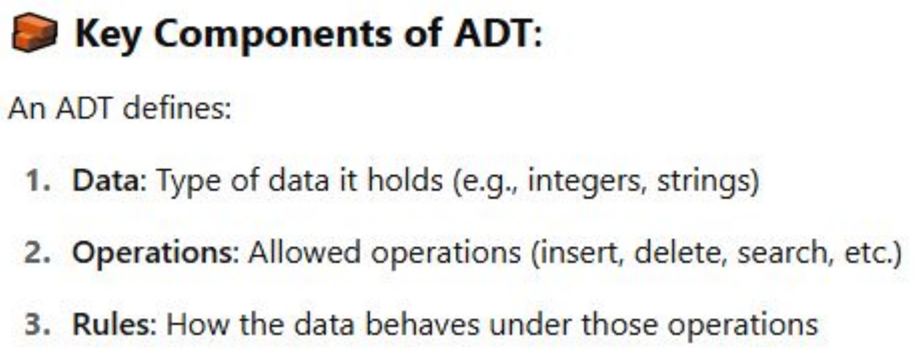
An **Abstract Data Type (ADT)** is a **logical or mathematical model** of a data structure that defines **what** operations can be performed on the data, but **not how** they are implemented.

> In simple terms, **ADT focuses on *what it does*, not *how it does it*.**

🧠 Why ADT?

- It hides the internal implementation details.
- Promotes **encapsulation** and **modular design**.
- Helps in writing cleaner and reusable code.

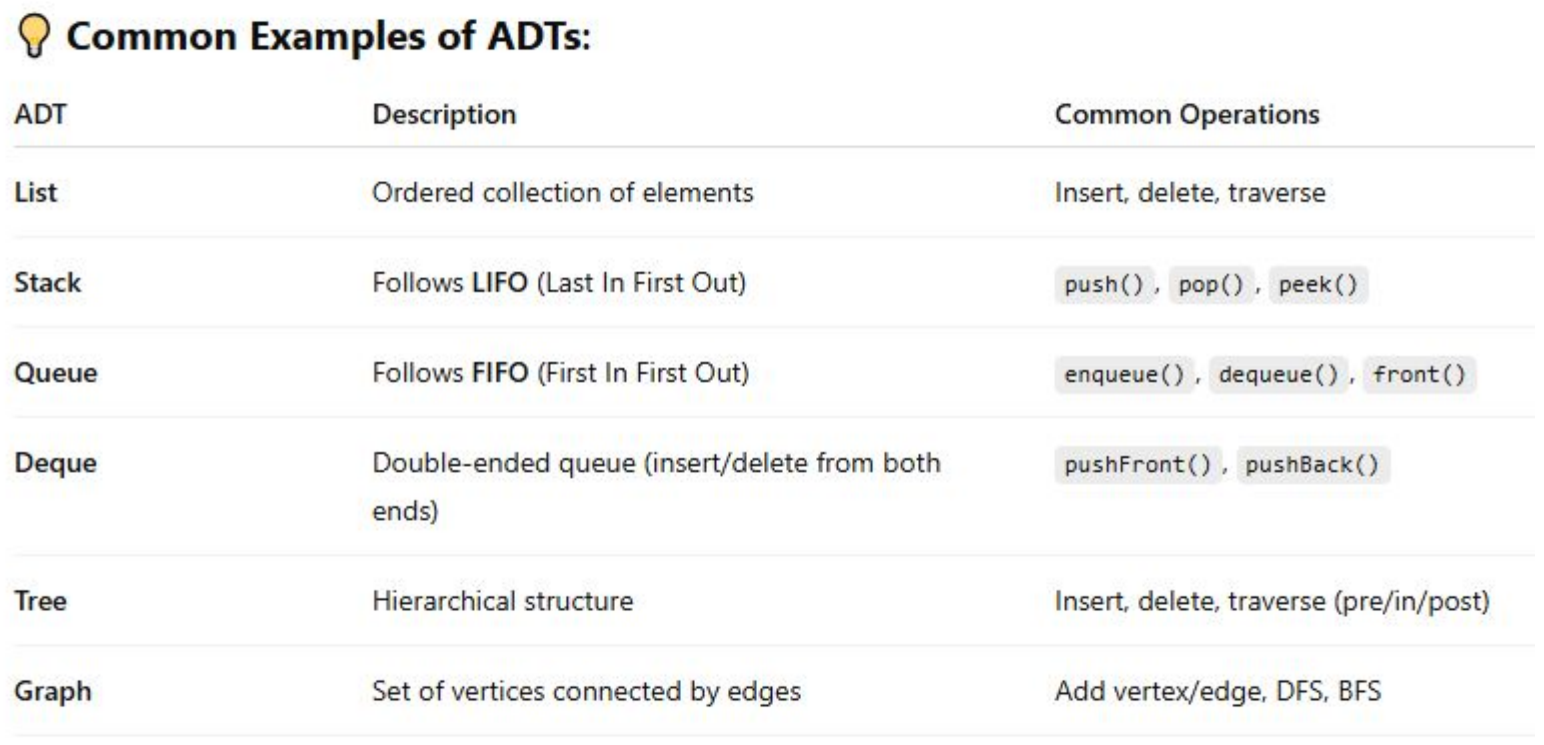
Abstract Data type



🧱 Key Components of ADT:

An ADT defines:

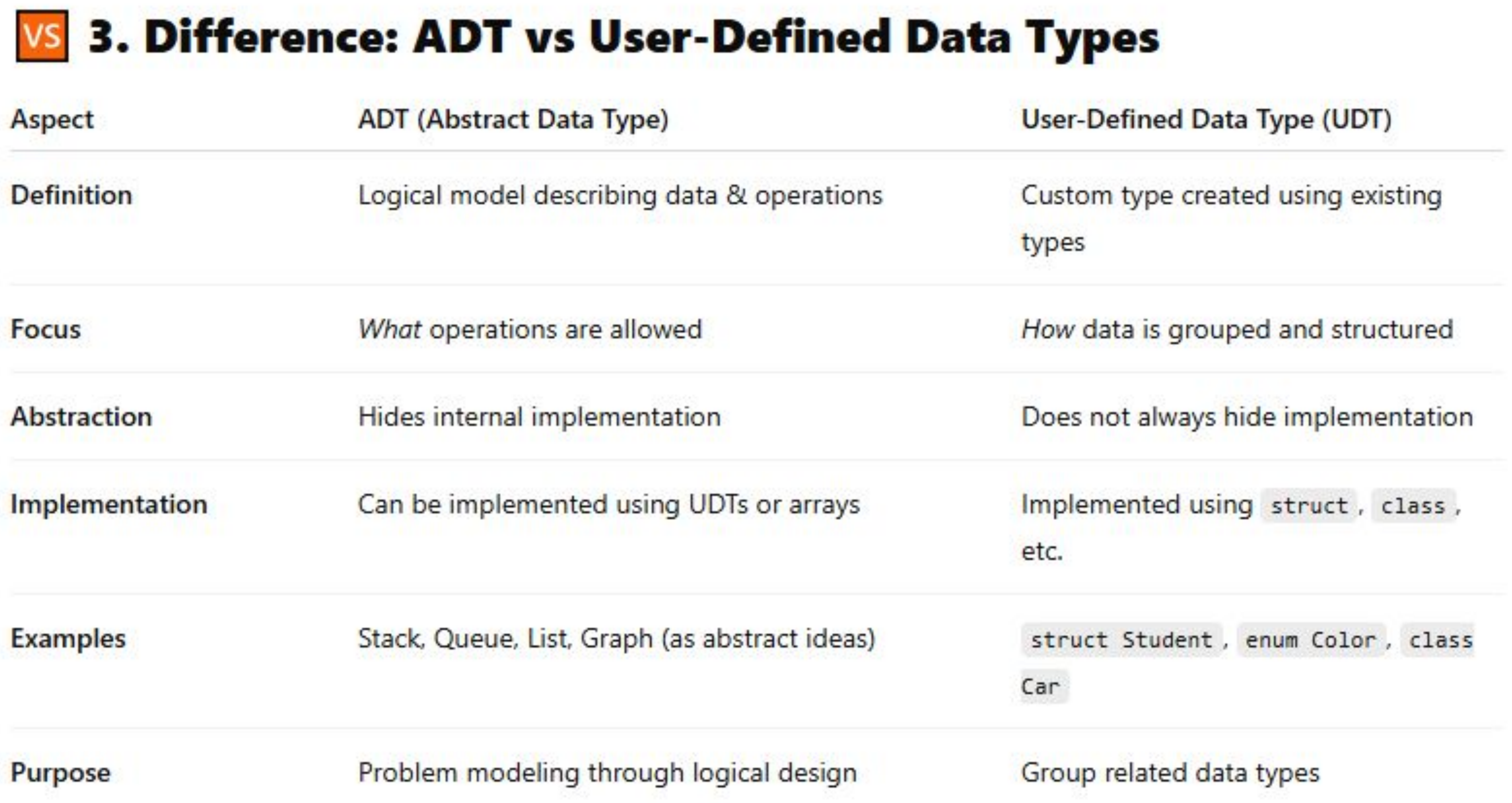
1. **Data**: Type of data it holds (e.g., integers, strings)
2. **Operations**: Allowed operations (insert, delete, search, etc.)
3. **Rules**: How the data behaves under those operations



💡 Common Examples of ADTs:

ADT	Description	Common Operations
List	Ordered collection of elements	Insert, delete, traverse
Stack	Follows **LIFO** (Last In First Out)	`push()`, `pop()`, `peek()`
Queue	Follows **FIFO** (First In First Out)	`enqueue()`, `dequeue()`, `front()`
Deque	Double-ended queue (insert/delete from both ends)	`pushFront()`, `pushBack()`
Tree	Hierarchical structure	Insert, delete, traverse (pre/in/post)
Graph	Set of vertices connected by edges	Add vertex/edge, DFS, BFS

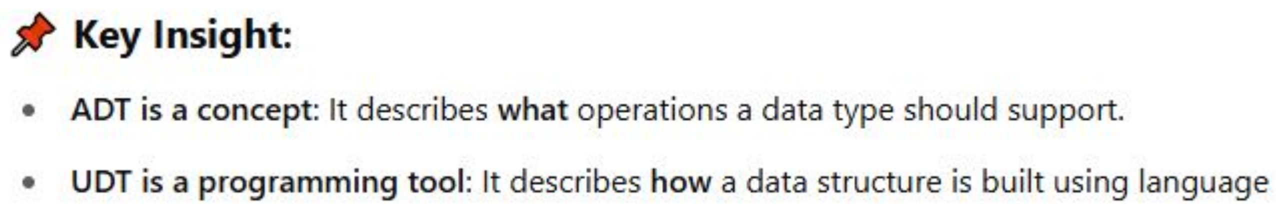
Diff between User Defined DT and ADT



🆚 3. Difference: ADT vs User-Defined Data Types

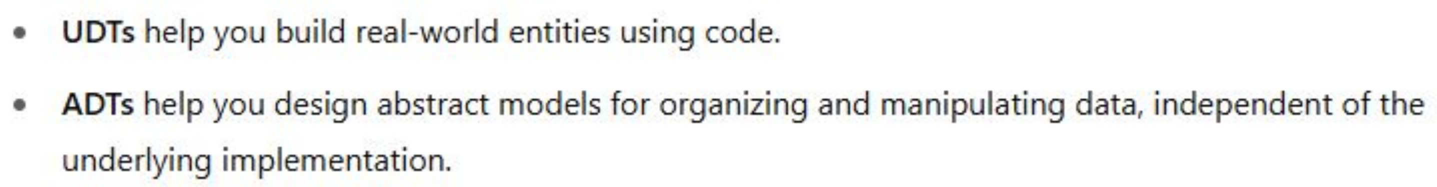
Aspect	ADT (Abstract Data Type)	User-Defined Data Type (UDT)
Definition	Logical model describing data & operations	Custom type created using existing types
Focus	*What* operations are allowed	*How* data is grouped and structured
Abstraction	Hides internal implementation	Does not always hide implementation
Implementation	Can be implemented using UDTs or arrays	Implemented using `struct`, `class`, etc.
Examples	Stack, Queue, List, Graph (as abstract ideas)	`struct Student`, `enum Color`, `class Car`
Purpose	Problem modeling through logical design	Group related data types

Diff between User Defined DT and ADT



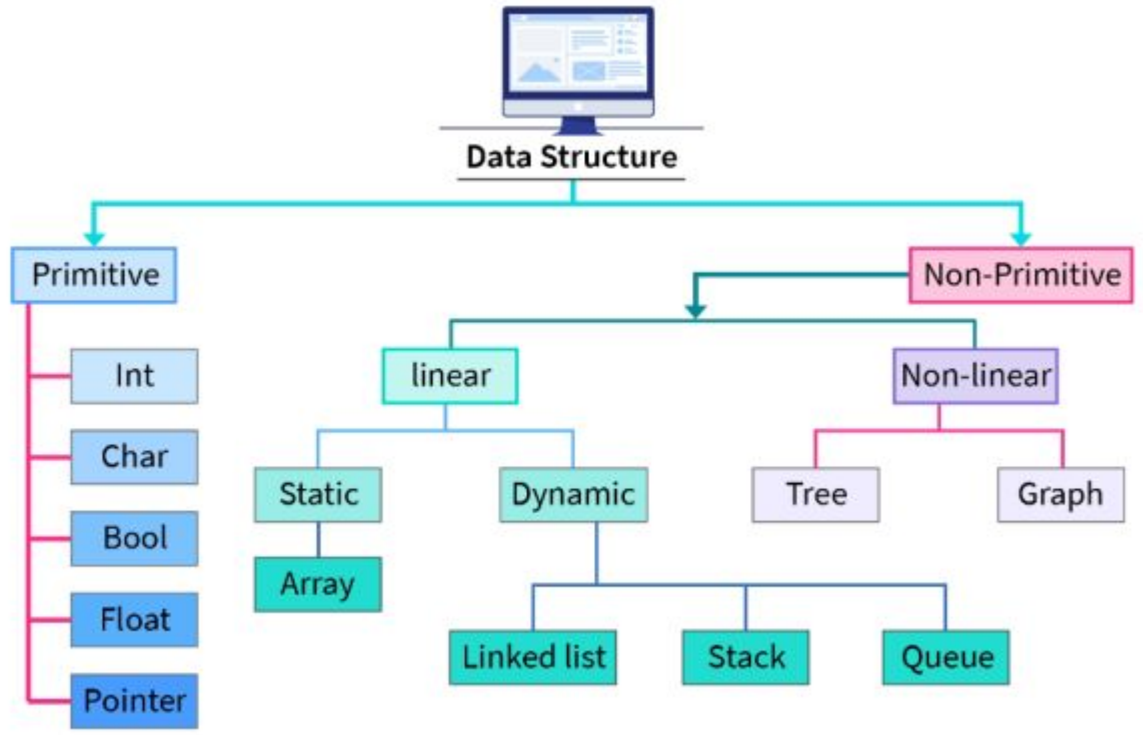
📌 **Key Insight:**

- **ADT is a concept:** It describes **what** operations a data type should support.
- **UDT is a programming tool:** It describes **how** a data structure is built using language

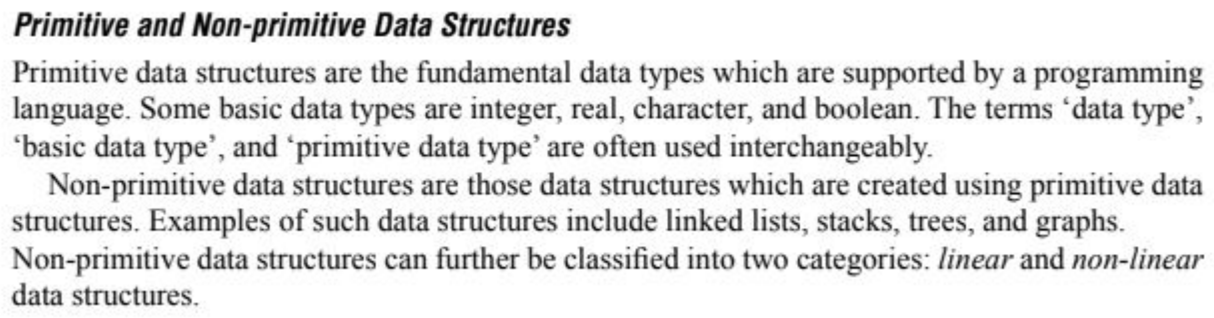


- **UDTs** help you build real-world entities using code.
- **ADTs** help you design abstract models for organizing and manipulating data, independent of the underlying implementation.

CLASSIFICATION OF DATA STRUCTURE



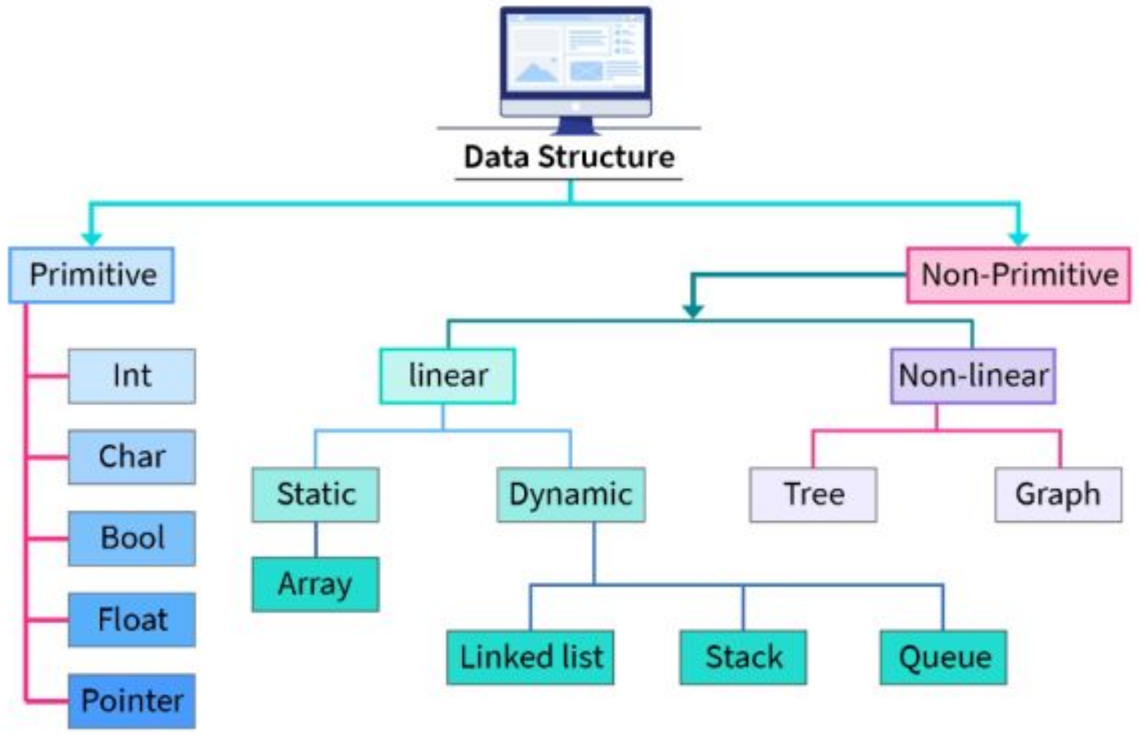
PRIMITIVE AND NON PRIMITIVE DS



Primitive and Non-primitive Data Structures

Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms 'data type', 'basic data type', and 'primitive data type' are often used interchangeably.

Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs. Non-primitive data structures can further be classified into two categories: *linear* and *non-linear* data structures.

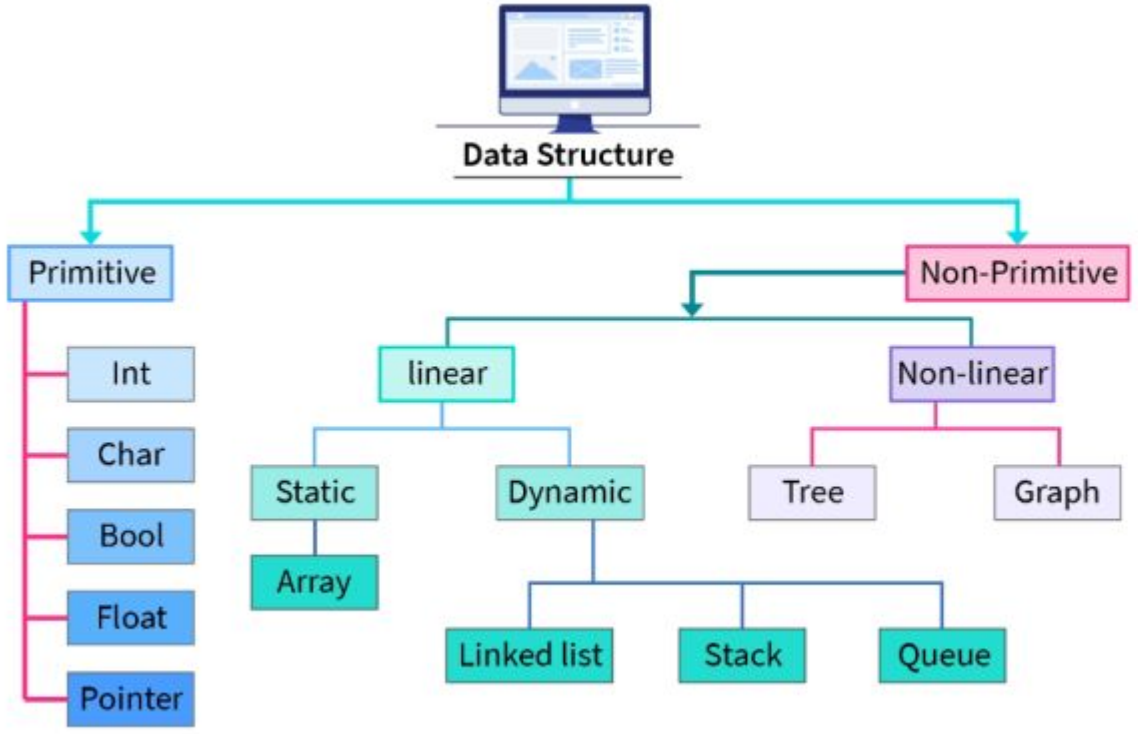


PRIMITIVE DS

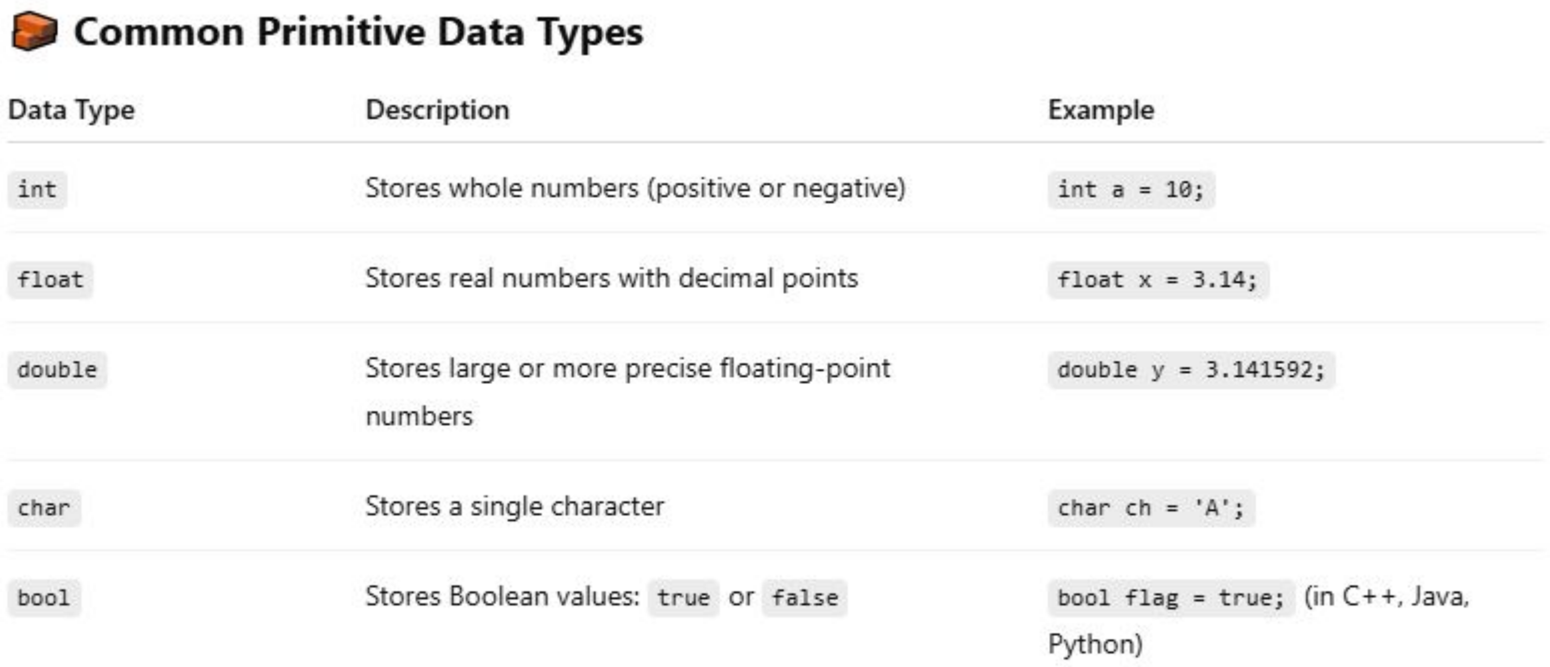
📘 Primitive Data Types

✅ Definition:

Primitive Data Types are the **basic building blocks** of data types in any programming language.
They are **predefined by the language** and used to represent **simple, single values** like integers, characters, or floating-point numbers.

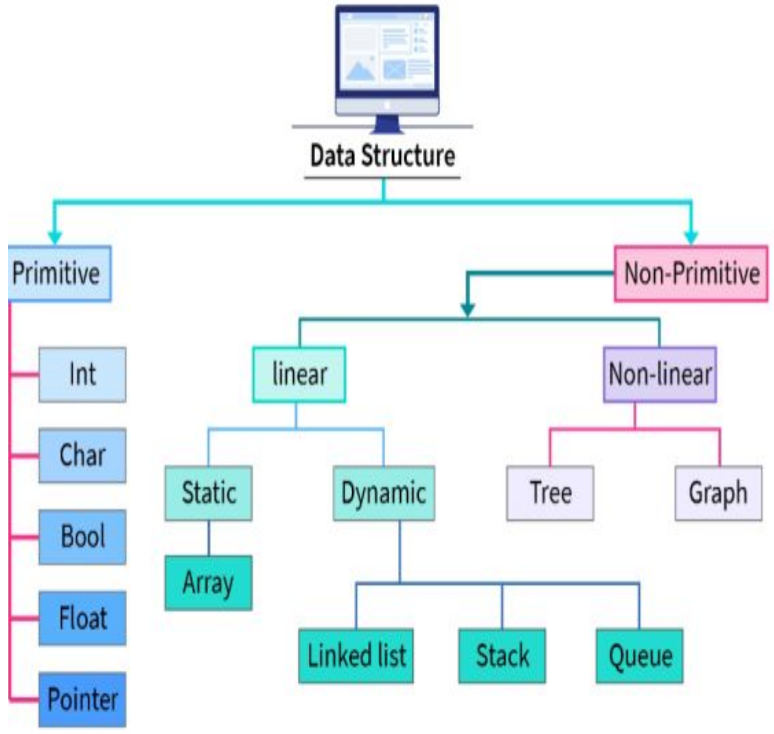


PRIMITIVE DS

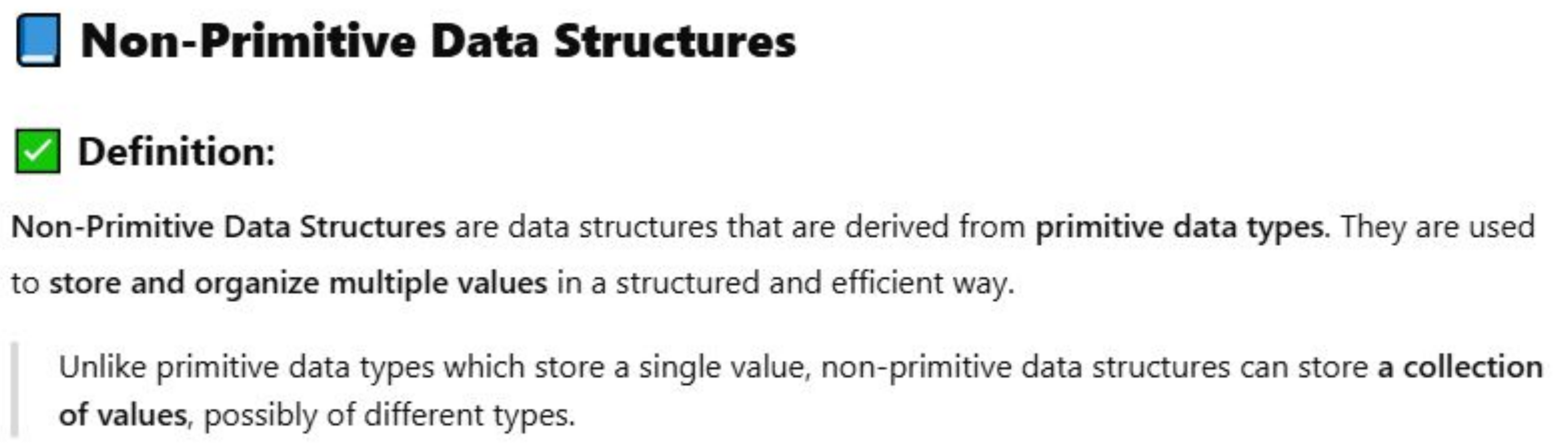


Common Primitive Data Types

Data Type	Description	Example
`int`	Stores whole numbers (positive or negative)	`int a = 10;`
`float`	Stores real numbers with decimal points	`float x = 3.14;`
`double`	Stores large or more precise floating-point numbers	`double y = 3.141592;`
`char`	Stores a single character	`char ch = 'A';`
`bool`	Stores Boolean values: `true` or `false`	`bool flag = true;` (in C++, Java, Python)



NON-PRIMITIVE DS



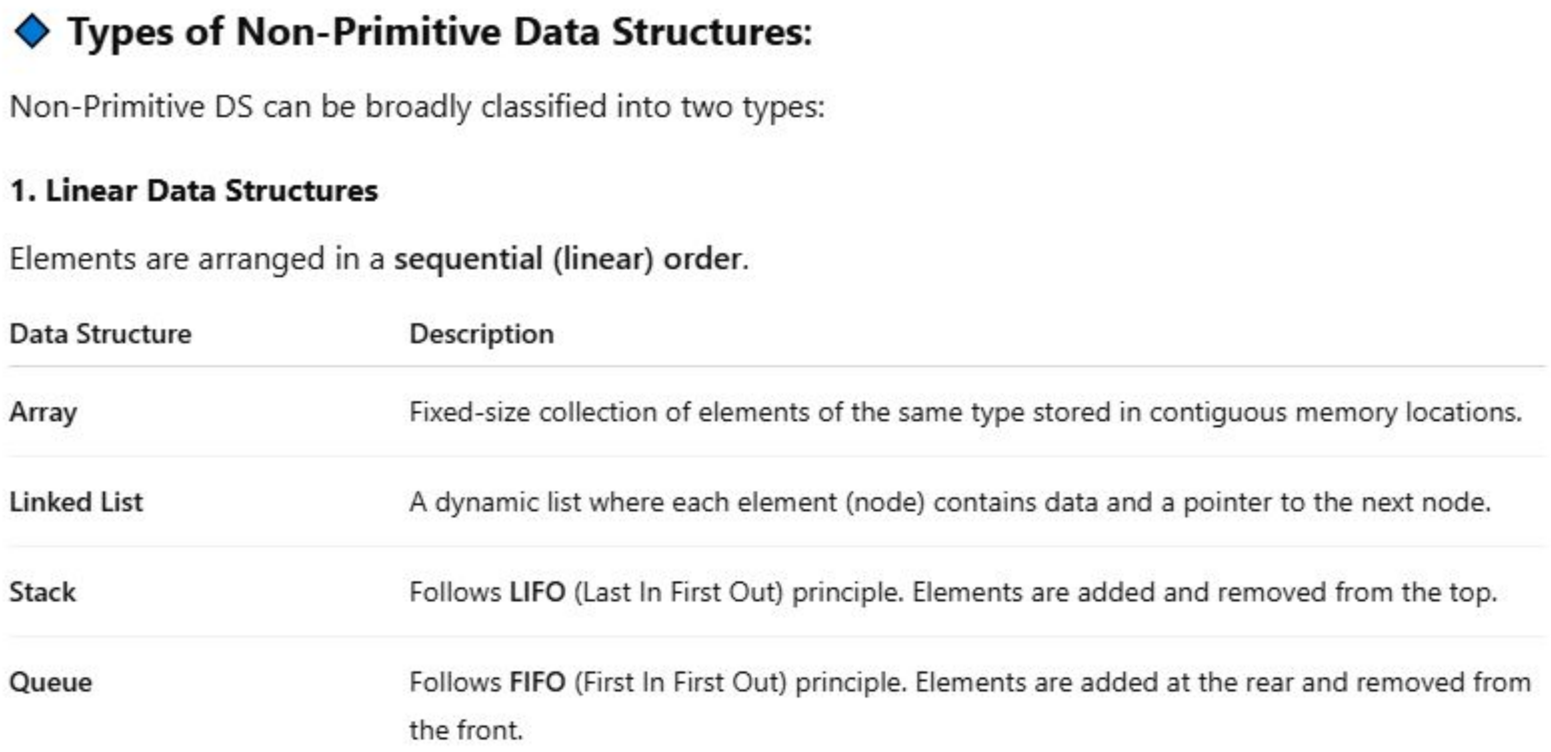
📘 Non-Primitive Data Structures

✅ Definition:

Non-Primitive Data Structures are data structures that are derived from **primitive data types**. They are used to **store and organize multiple values** in a structured and efficient way.

> Unlike primitive data types which store a single value, non-primitive data structures can store **a collection of values**, possibly of different types.

TYPES OF NON-PRIMITIVE DS



🔷 Types of Non-Primitive Data Structures:

Non-Primitive DS can be broadly classified into two types:

1. Linear Data Structures

Elements are arranged in a **sequential (linear) order**.

Data Structure	Description
Array	Fixed-size collection of elements of the same type stored in contiguous memory locations.
Linked List	A dynamic list where each element (node) contains data and a pointer to the next node.
Stack	Follows **LIFO** (Last In First Out) principle. Elements are added and removed from the top.
Queue	Follows **FIFO** (First In First Out) principle. Elements are added at the rear and removed from the front.

TYPES OF NON-PRIMITIVE DS

2. Non-Linear Data Structures

Elements are not arranged sequentially. They are arranged in a **hierarchical or networked fashion.**

Data Structure	Description
Tree	A hierarchical structure where each node points to its child nodes. Example: Binary Tree, Binary Search Tree, AVL Tree.
Graph	A collection of nodes (vertices) connected by edges. Can represent networks like social media, roads, etc.

🧠 Characteristics of Non-Primitive Data Structures:

- Can store multiple values
- Can store heterogeneous data (in some cases like structures or classes)
- Can grow dynamically (Linked Lists, Trees)
- Useful for solving complex problems and managing large data

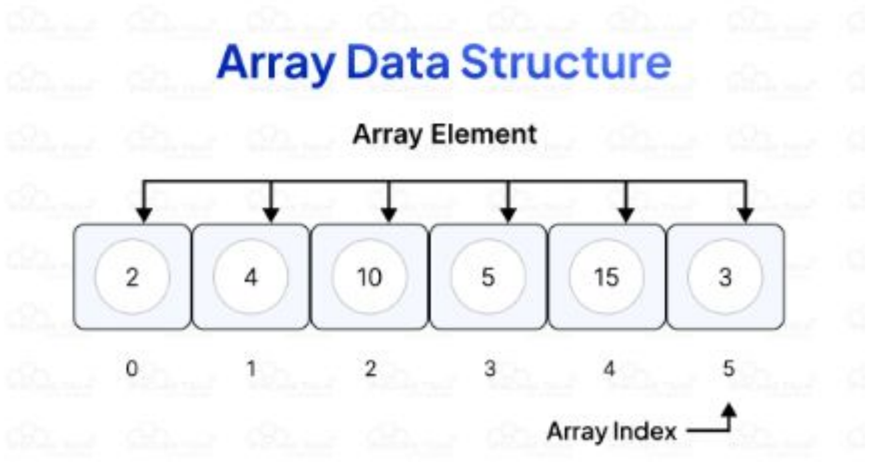
ARRAY

📘 ARRAY – Data Structure

✅ Definition:

An **array** is a **linear data structure** that stores a **fixed-size** collection of **elements of the same data type** in **contiguous memory locations**.

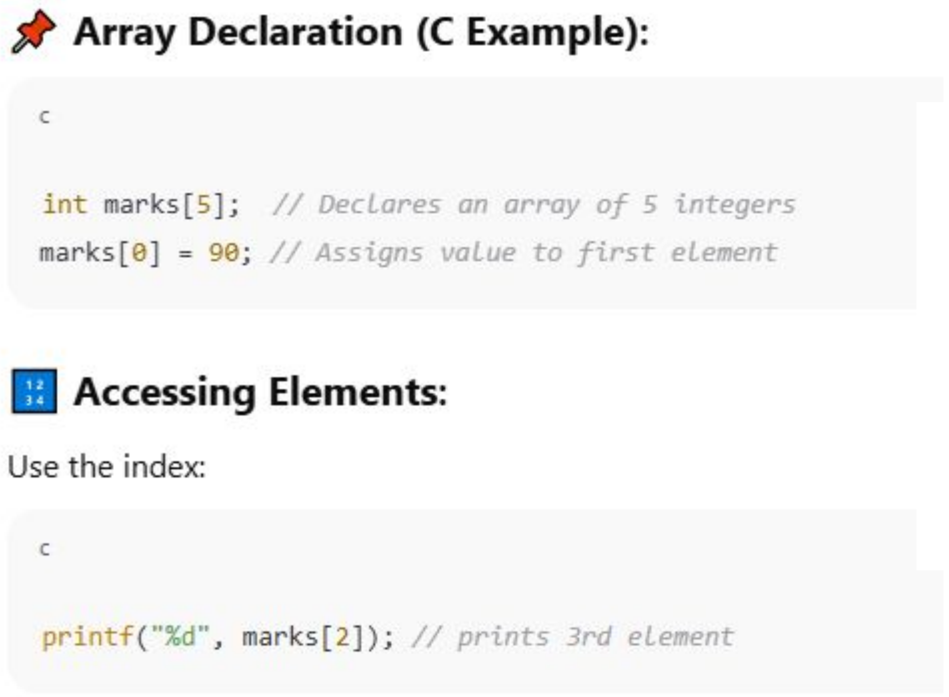
> Each element is accessed using an **index**, starting from `0`.



🧠 Key Features of Arrays:

- Stores **multiple elements** under one variable name.
- All elements must be of the **same type** (e.g., all `int` or all `float`).
- Elements are stored in **adjacent memory locations**.
- Provides **random access** using indexing.

ARRAY



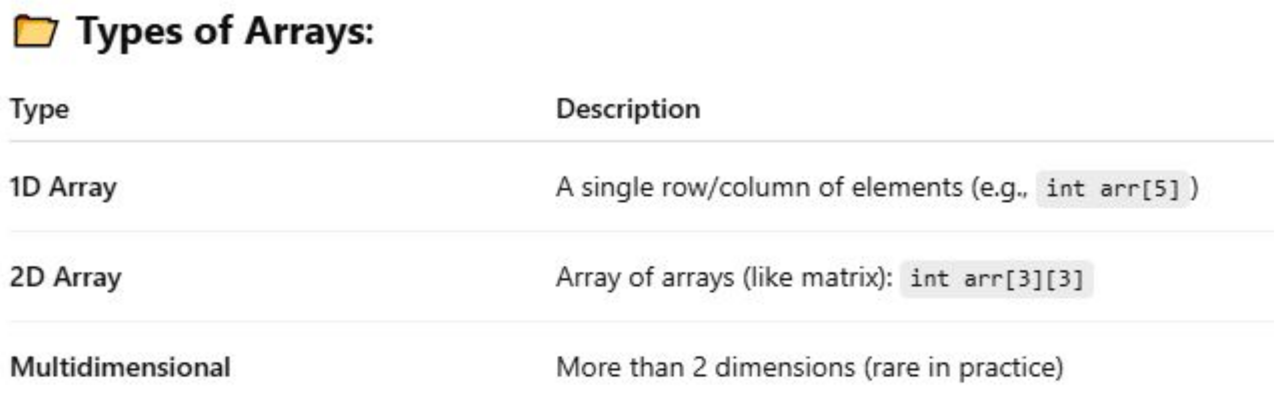
📌 Array Declaration (C Example):

```c
int marks[5];   // Declares an array of 5 integers
marks[0] = 90; // Assigns value to first element
```

🔢 Accessing Elements:

Use the index:

```c
printf("%d", marks[2]); // prints 3rd element
```



📂 Types of Arrays:

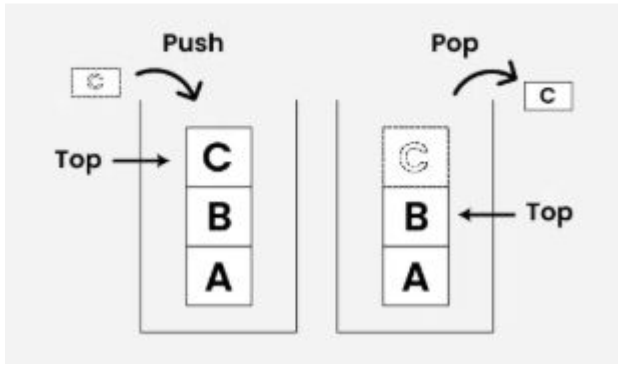
Type	Description
1D Array	A single row/column of elements (e.g., `int arr[5]`)
2D Array	Array of arrays (like matrix): `int arr[3][3]`
Multidimensional	More than 2 dimensions (rare in practice)

STACK

A **stack** is a **linear data structure** that follows the **LIFO (Last In, First Out)** principle.

> The **last element inserted** (pushed) is the **first one removed** (popped).

Imagine a **stack of plates** – you add (push) plates on top and remove (pop) from the top only.



🧠 Key Operations in Stack:

Operation	Description
`push(x)`	Inserts an element `x` on top of the stack
`pop()`	Removes the top element from the stack
`peek()` / `top()`	Returns the top element without removing it
`isEmpty()`	Checks if the stack is empty
`isFull()` *(optional, for fixed-size stacks)*	Checks if the stack is full

🧱 Implementation Methods:

1. **Array-based Stack**: Fixed size
2. **Linked List-based Stack**: Dynamic size

QUEUES

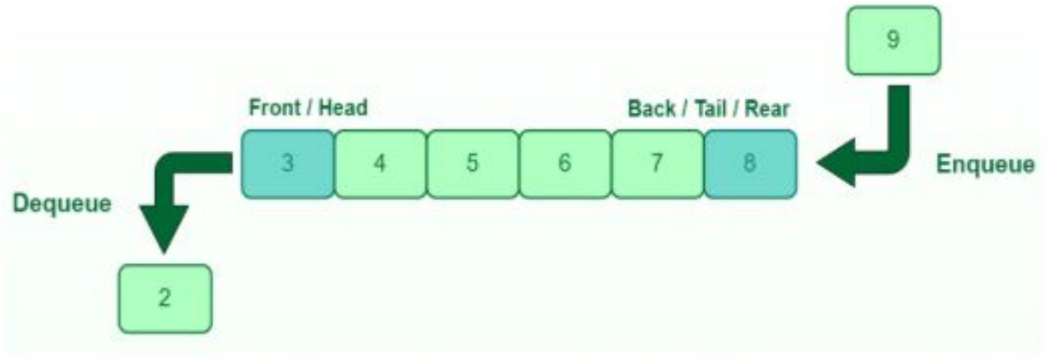
✅ Definition:

A **queue** is a **linear data structure** that follows the **FIFO (First In, First Out)** principle.

> The element inserted **first** is removed **first**, just like a real-world queue (e.g., line at a ticket counter).

🧠 Key Operations in Queue:

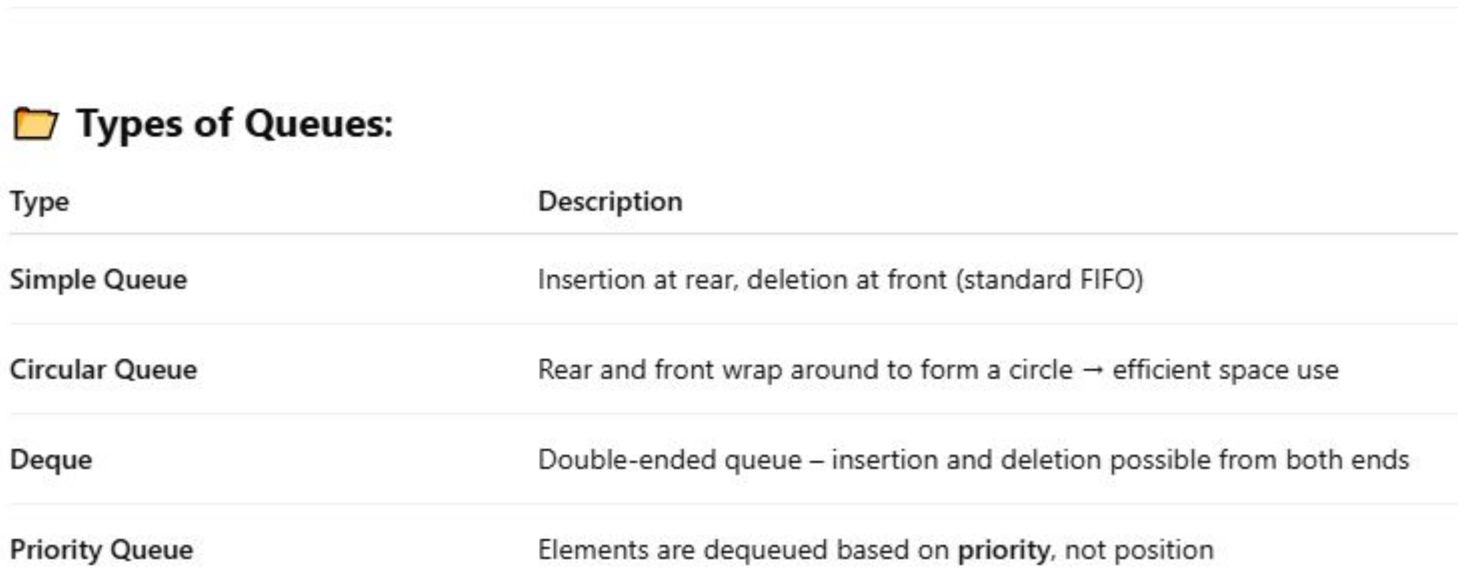
Operation	Description
`enqueue(x)`	Add an element `x` to the **rear** of the queue
`dequeue()`	Remove and return the element from the **front**
`front()`	View the front element without removing it
`isEmpty()`	Check if the queue is empty
`isFull()` *(in fixed-size queues)*	Check if the queue is full



Queue Data Structure

TRUST

QUEUES



📂 Types of Queues:

Type	Description
Simple Queue	Insertion at rear, deletion at front (standard FIFO)
Circular Queue	Rear and front wrap around to form a circle → efficient space use
Deque	Double-ended queue – insertion and deletion possible from both ends
Priority Queue	Elements are dequeued based on **priority**, not position

LINKED LIST

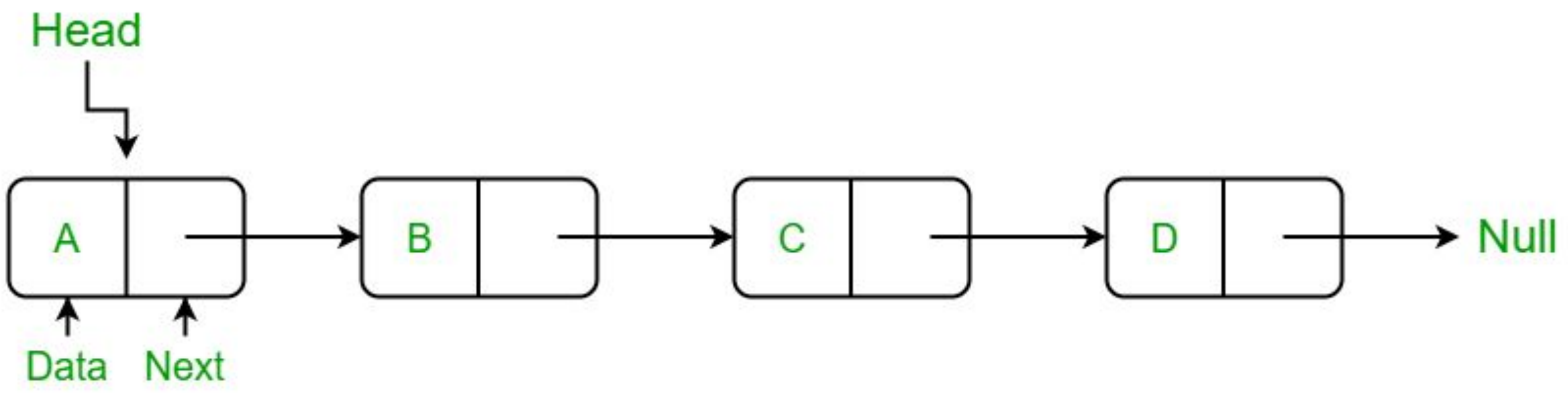
✅ Definition:

A **Linked List** is a **linear data structure** in which **elements (nodes)** are stored **non-contiguously in memory** and are linked together using **pointers**.

> Each node contains **data** and a **reference (pointer)** to the next node in the sequence.

📌 Types of Linked Lists:

Type	Description
Singly Linked List	Each node points to the next node only
Doubly Linked List	Each node points to both previous and next node
Circular Linked List	Last node links back to the first node (circular structure)



Somaiya TRUST

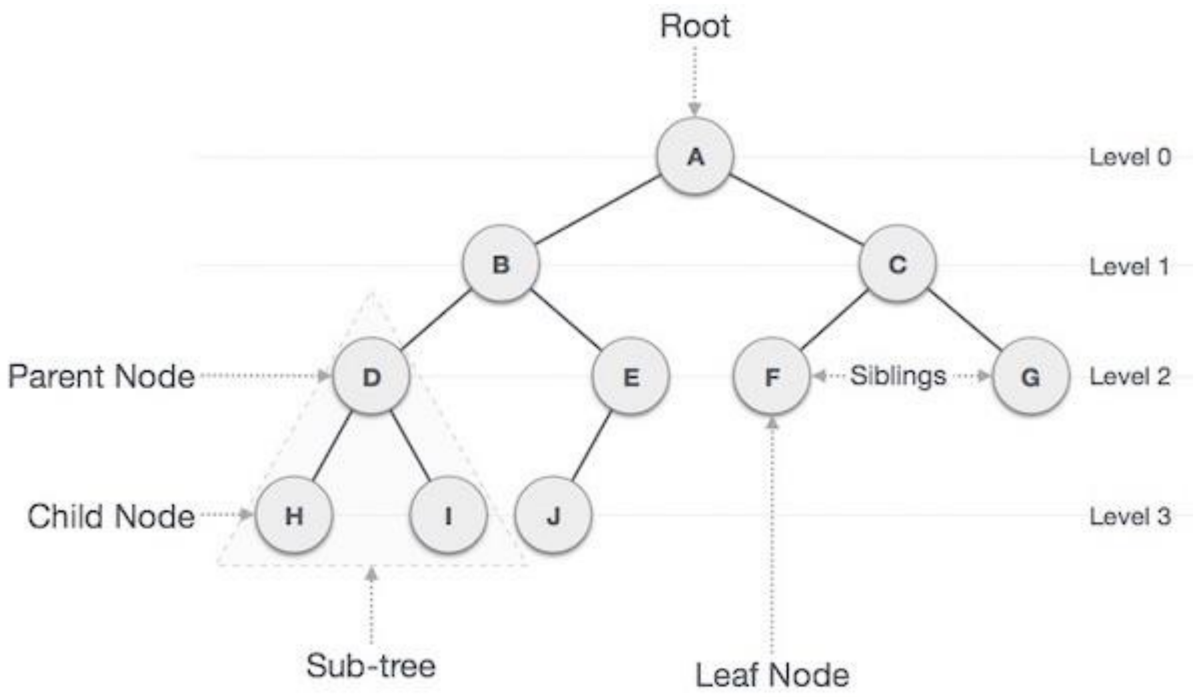
TREE

🌳 TREE – Data Structure

✅ Definition:

A **Tree** is a **non-linear hierarchical data structure** consisting of nodes, where:

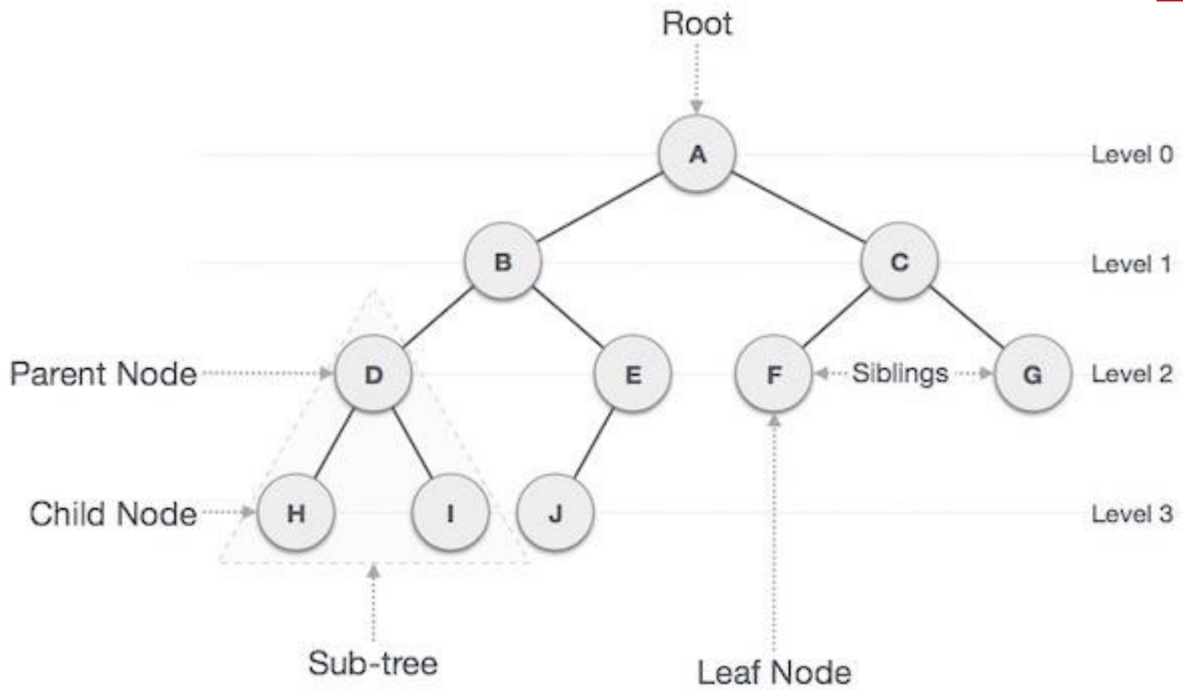
- The **topmost node** is called the **root**.
- Each node contains **data** and **links (edges)** to its **child nodes**.
- There are **no cycles** — a tree is an **acyclic graph**.



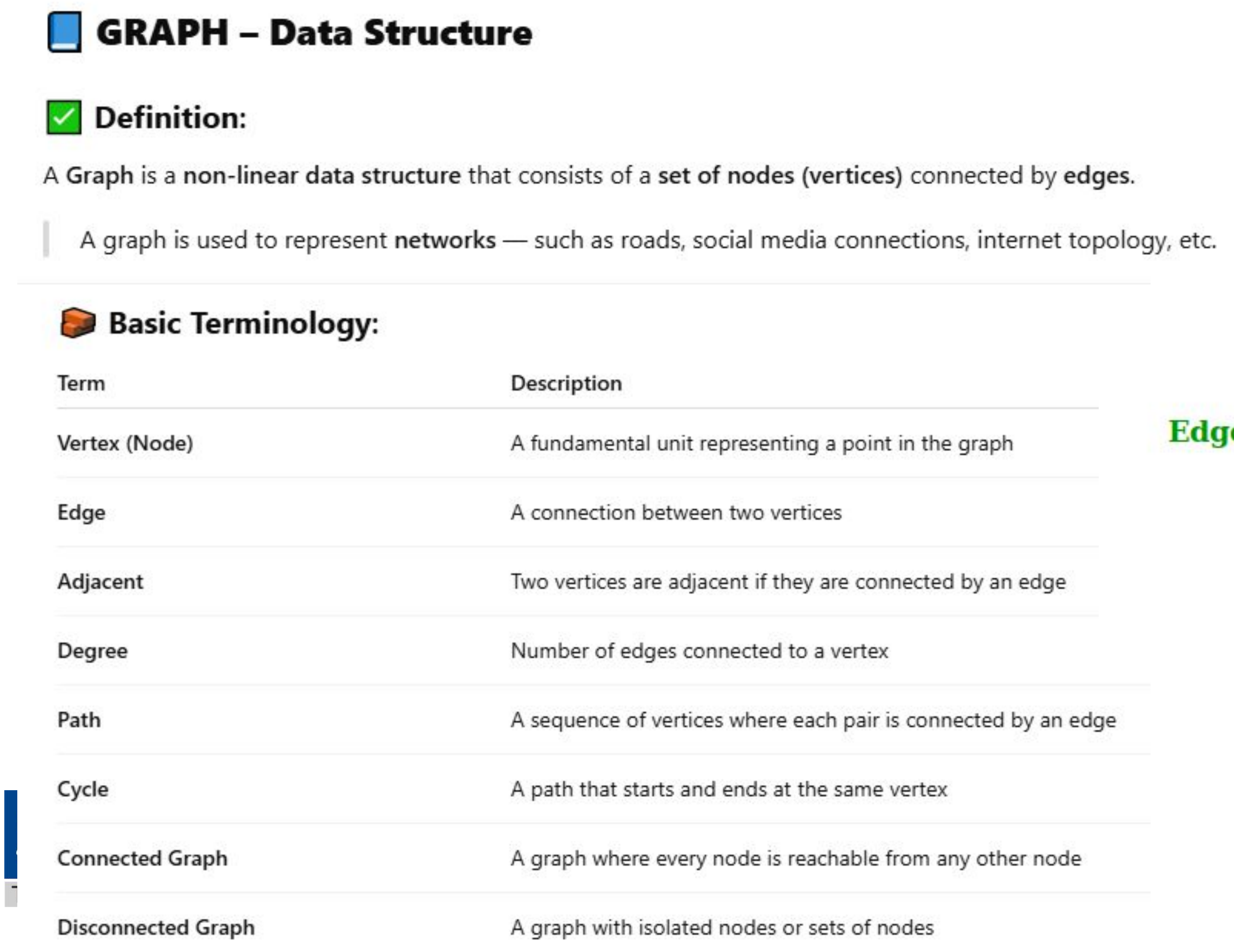
TREE

Basic Terminology:

Term	Description
Root	The topmost node (starting point of the tree)
Node	Basic unit containing data
Parent	A node that has children
Child	A node that descends from a parent
Leaf	A node with no children
Edge	Connection between parent and child node
Subtree	A tree formed by any node and its descendants
Level	Distance from root (root is level 0)
Height	Longest path from root to a leaf



GRAPH



📘 GRAPH – Data Structure

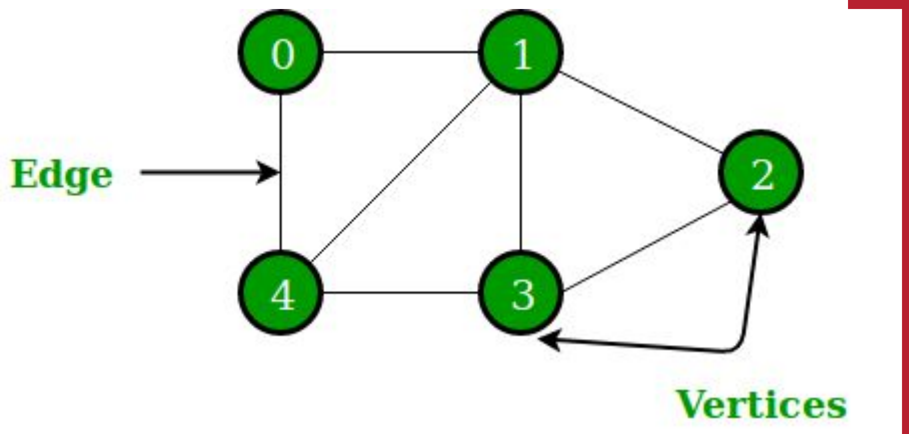
✅ Definition:

A **Graph** is a **non-linear data structure** that consists of a **set of nodes (vertices)** connected by **edges**.

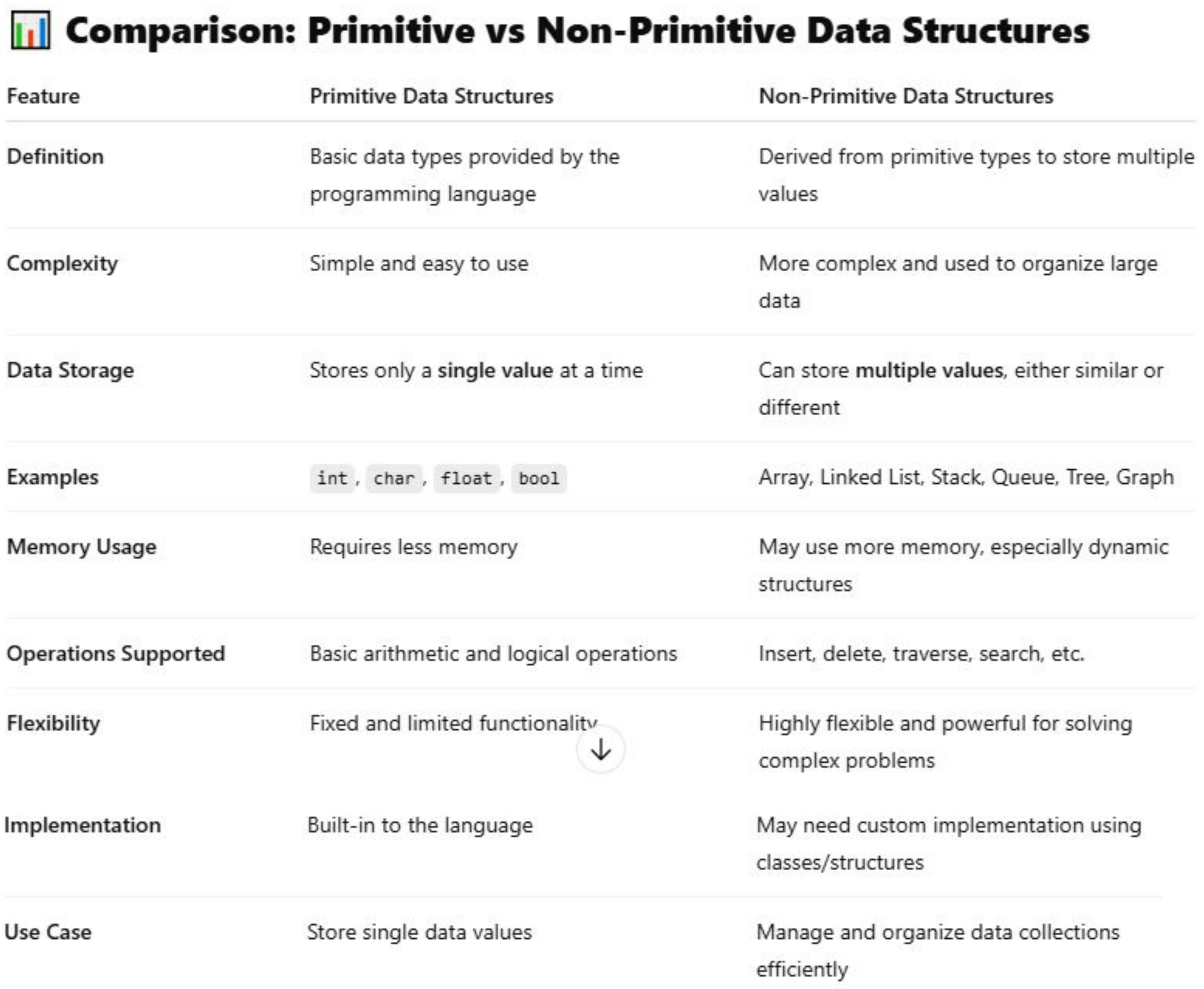
> A graph is used to represent **networks** — such as roads, social media connections, internet topology, etc.

🧱 Basic Terminology:

Term	Description
Vertex (Node)	A fundamental unit representing a point in the graph
Edge	A connection between two vertices
Adjacent	Two vertices are adjacent if they are connected by an edge
Degree	Number of edges connected to a vertex
Path	A sequence of vertices where each pair is connected by an edge
Cycle	A path that starts and ends at the same vertex
Connected Graph	A graph where every node is reachable from any other node
Disconnected Graph	A graph with isolated nodes or sets of nodes



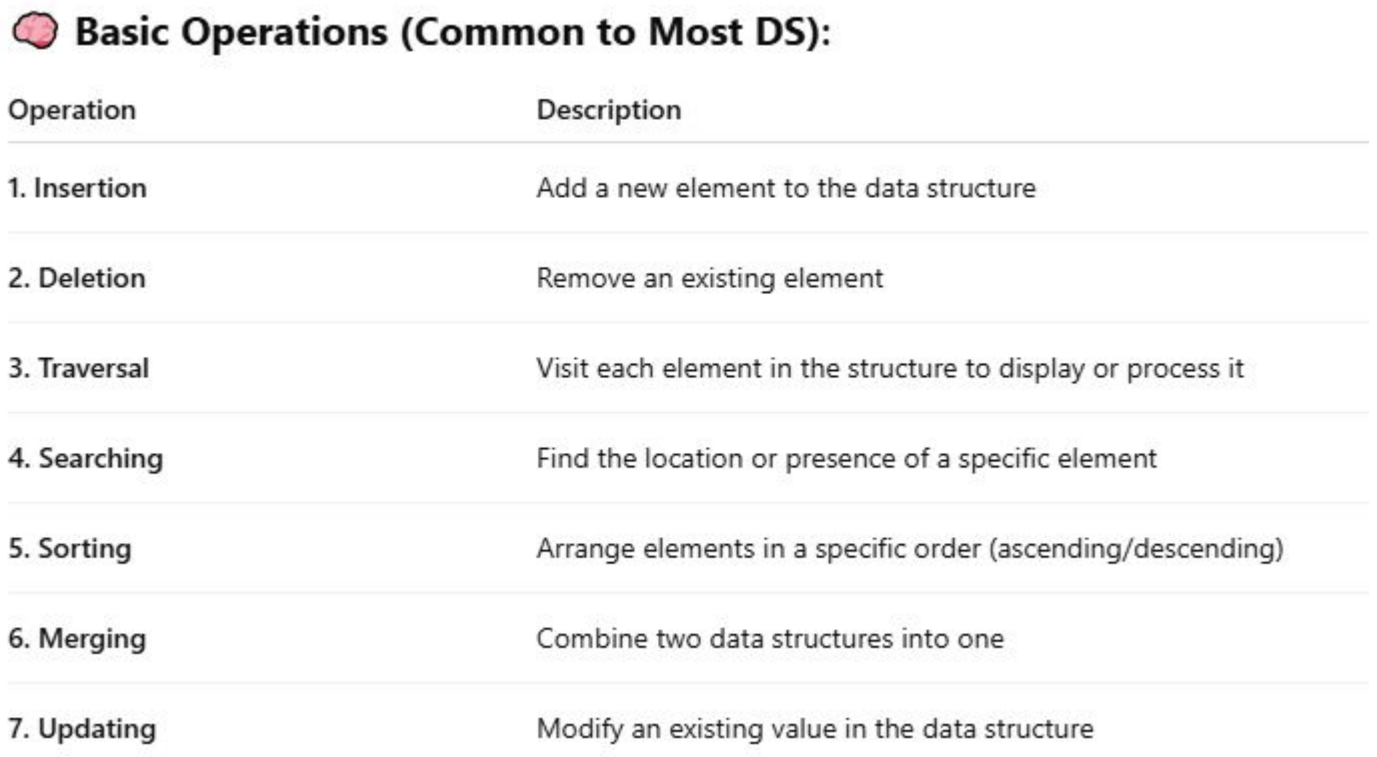
COMPARISON: PRIMITIVE AND NON PRIMITIVE DS



📊 Comparison: Primitive vs Non-Primitive Data Structures

Feature	Primitive Data Structures	Non-Primitive Data Structures
Definition	Basic data types provided by the programming language	Derived from primitive types to store multiple values
Complexity	Simple and easy to use	More complex and used to organize large data
Data Storage	Stores only a **single value** at a time	Can store **multiple values**, either similar or different
Examples	`int`, `char`, `float`, `bool`	Array, Linked List, Stack, Queue, Tree, Graph
Memory Usage	Requires less memory	May use more memory, especially dynamic structures
Operations Supported	Basic arithmetic and logical operations	Insert, delete, traverse, search, etc.
Flexibility	Fixed and limited functionality	Highly flexible and powerful for solving complex problems
Implementation	Built-in to the language	May need custom implementation using classes/structures
Use Case	Store single data values	Manage and organize data collections efficiently

OPERATIONS ON DS



🧠 Basic Operations (Common to Most DS):

Operation	Description
1. Insertion	Add a new element to the data structure
2. Deletion	Remove an existing element
3. Traversal	Visit each element in the structure to display or process it
4. Searching	Find the location or presence of a specific element
5. Sorting	Arrange elements in a specific order (ascending/descending)
6. Merging	Combine two data structures into one
7. Updating	Modify an existing value in the data structure

SAMPLE QUESTIONS

QUESTION NO.	SAMPLE QUESTIONS MODULE 1
1	Explain Stack and Queue as Abstract data types
2	Explain the types of Data Structures with diagram
3	Explain the types of Linear DS
4	Explain the types of Nonlinear DS
5	What do you understand about stack overflow and underflow?
6	Explain Abstract data types