# UST
# Faculty of Engineering
# Software Engineering Department

## Data Structure and Algorithms

## Lecture 1

## Introduction to Data Structures and Algorithm Analysis

# Main Text Book

**Mark Allen Weiss**

*Data Structures and Algorithm Analysis in C++*, 4th Edition, Pearson.

# Additional References

**Mark Allen Weiss**
*Data Structures and Algorithm Analysis in Java*, 3rd Edition.

Thomas H. Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein
*Introduction to Algorithms (CLRS)*, MIT Press.

# Course Assessment Criteria

| | |
|---|---|
| ✓Tutorials | 20% |
| ✓Laboratory Work | 20% |
| ✓Final Exam | 60% |

# Prerequisite Courses

1. Programming Fundamentals.
2. Object Oriented Programming (OOP)

# Course Learning Outcomes

**After completing this course, students will be able to:**

- Explain the concept of Abstract Data Types (ADTs).
- Implement linear and non-linear data structures.
- Apply searching and sorting algorithms.
- Select appropriate data structures for software engineering problems.
- Analyze the performance of different algorithms.
- Solve programming problems using efficient data structures.

# What is Data Structure?

**Definition:**

- A **data structure** is a specialized way of **organizing, storing, and managing data** in a computer so that it can be accessed, processed, and modified efficiently.
- It provides a systematic method for handling data, allowing algorithms to perform operations such as searching, inserting, deleting, and updating efficiently.

# Simple Definition:

- A **data structure** is a method of organizing and storing data in a computer so that it can be used efficiently by algorithms and applications.

# Why Are Data Structures Important?

**Data structures are essential because they help computers:**

- Store large amounts of data efficiently.
- Access data quickly.
- Improve the performance of programs.
- Reduce memory usage.
- Solve complex computational problems effectively.
- Without appropriate data structures, even simple programs can become slow and inefficient.

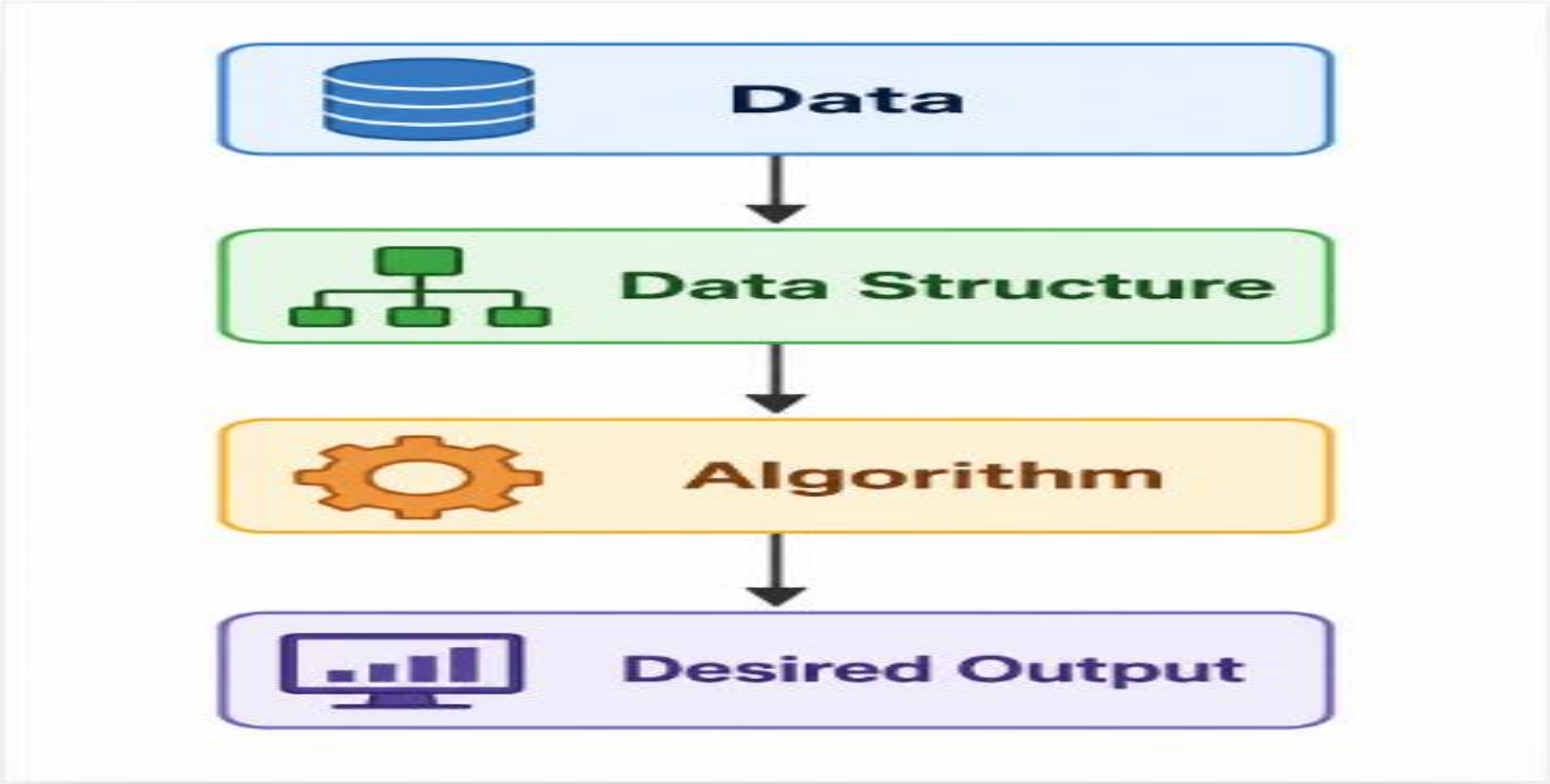
# Example:

- Think of a **library**.
- Books represent **data**.
- Shelves represent the **data structure**.
- The librarian represents the **algorithm**.
- If books are placed randomly, finding one book may take a long time. However, if books are organized alphabetically or by category, they can be found much faster.
- Similarly, a computer uses data structures to organize data for efficient processing.

# Data Structure and Algorithm Relationship

- A data structure stores and organizes data, while an algorithm defines the steps used to manipulate that data.

# Data Flow Process Diagram



# Common Operations on Data Structures

Most data structures support the following basic operations:

| Operation | Description |
|---|---|
| Traversing | Visiting each element in the structure |
| Searching | Finding a specific element |
| Insertion | Adding a new element |
| Deletion | Removing an existing element |
| Updating | Modifying the value of an element |
| Sorting | Arranging elements in a specific order |
| Merging | Combining two data structures |

# Examples of Data Structures

| Data Structure | Common Use |
| --- | --- |
| Array | Store fixed-size collections of data |
| Linked List | Dynamic data storage |
| Stack | Undo operations, function calls |
| Queue | Printer queues, scheduling |
| Tree | File systems, databases |
| Graph | Social networks, maps |
| Hash Table | Fast searching and lookup |

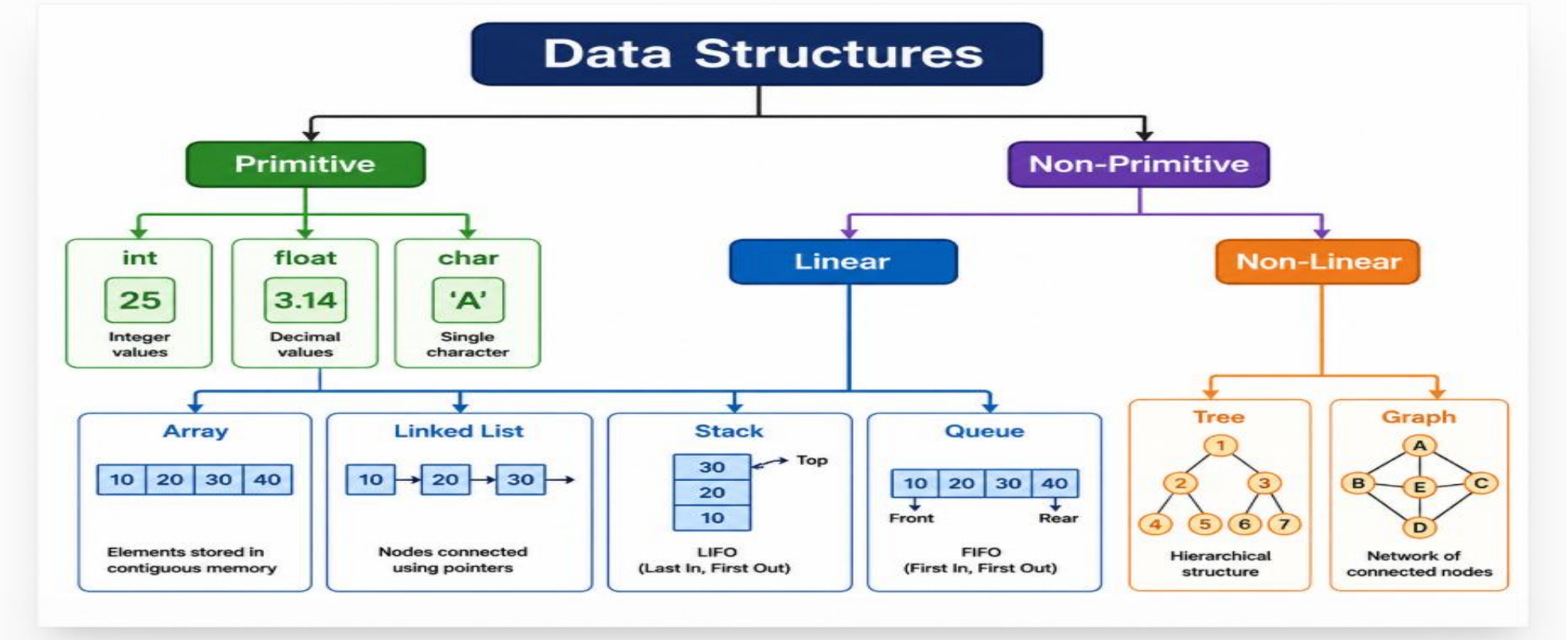
# Key Characteristics of Data Structures

- ✓Organize data efficiently.
- ✓Improve program performance.
- ✓Simplify data management.
- ✓Support efficient algorithm execution.
- ✓Optimize memory usage.

# Types of Data Structures

- Data structures can be classified into different categories based on how data is organized, stored, and accessed.
- The most common classification is into **Primitive** and **Non-Primitive** data structures.

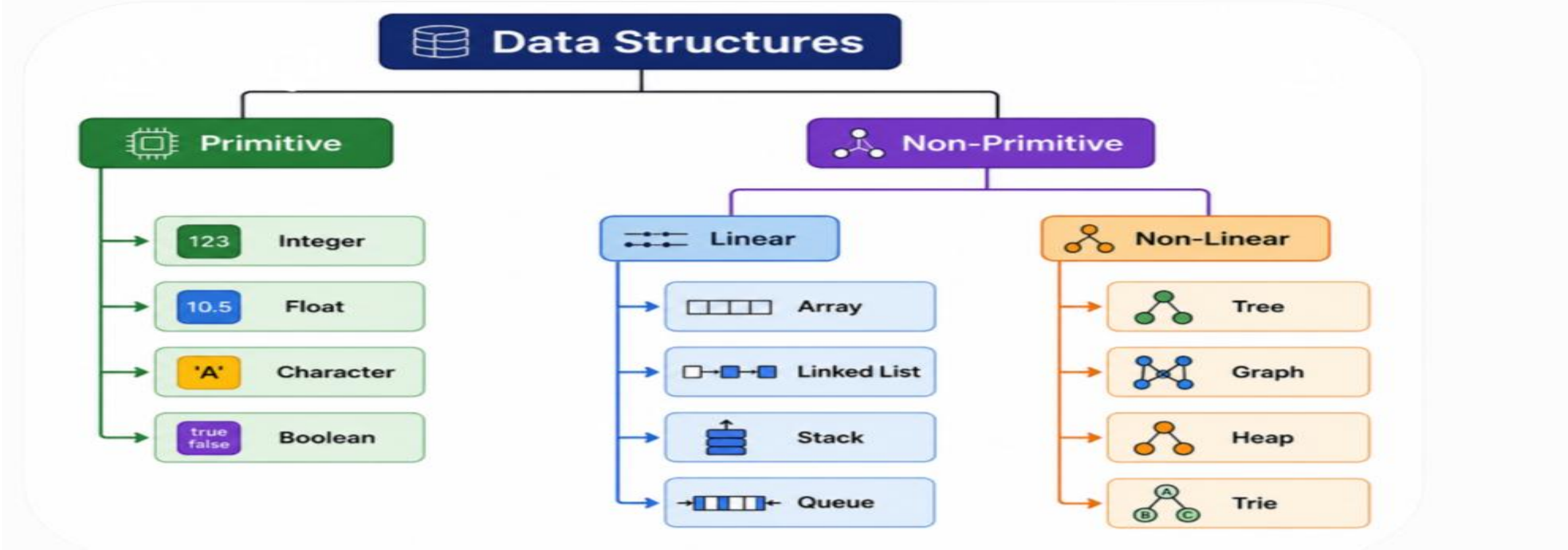
# Data Structures Types



# Primitive data vs Non-Primitive data

- **Primitive data structures** are the fundamental data types used to store individual values.
- **Non-Primitive data structures** are built from primitive types and are designed to store and organize multiple related values efficiently.

# Primitive vs Non-Primitive Data Structures



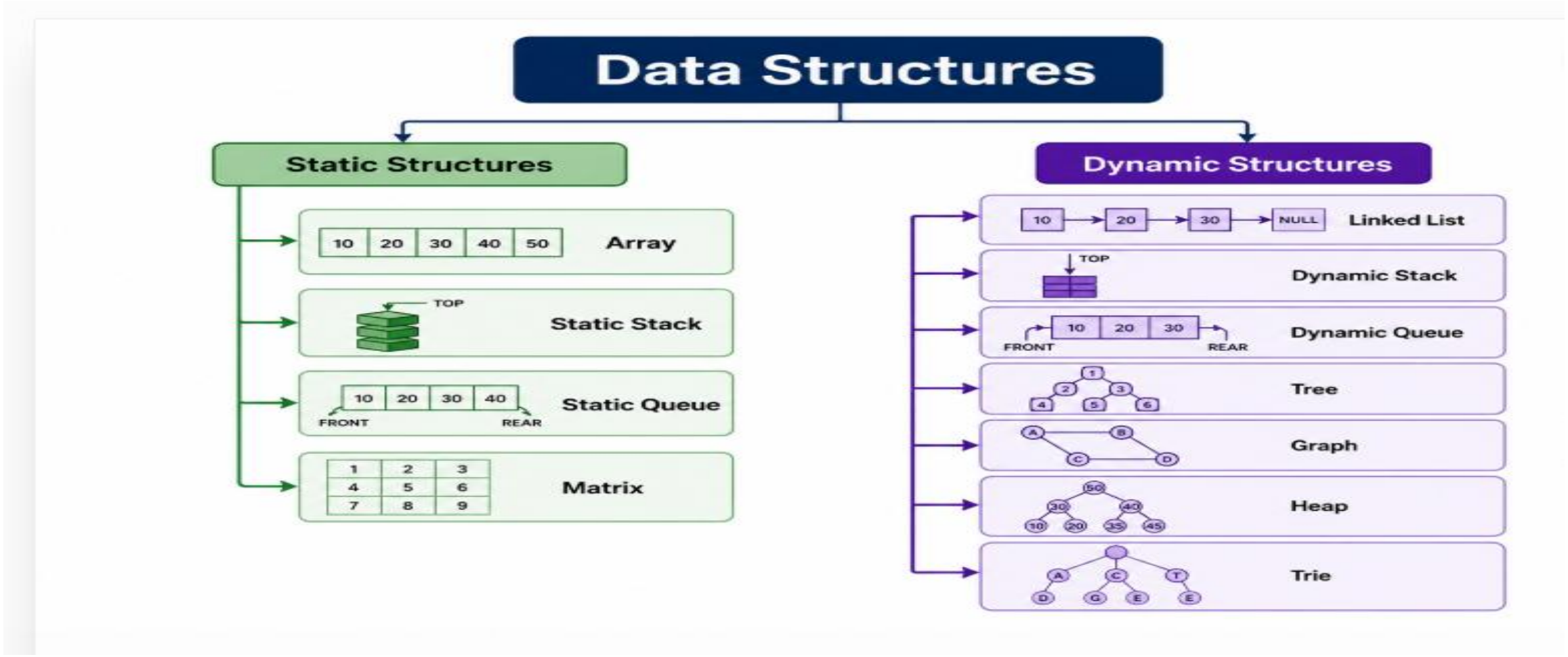
# Key Difference

| Feature | Primitive | Non-Primitive |
| --- | --- | --- |
| Complexity | Simple | Complex |
| Data Stored | Single value | Multiple values |
| Memory Usage | Low | Higher |
| Size | Fixed | Fixed or Dynamic |

# Static data vs Dynamic data

- **Static data structures** have a fixed size, making them simple and fast, but they may waste memory if the allocated space is not fully utilized.
- **Dynamic data structures** can expand or shrink at runtime, providing greater flexibility and better memory utilization, although they require more complex memory management.

# Static data vs Dynamic data



# Abstract Data Type (ADT)

- 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 without specifying how those operations are implemented.
- In simple terms: ADT describes what a data structure does, while the data structure describes how it does it.

# Characteristics of ADTs

- Defines **data** and **operations**.
- Hides implementation details (**data abstraction**).
- Focuses on functionality rather than memory representation.
- Can have multiple implementations.

# ADT Example

| Abstract Data Type (ADT) | Possible Implementations |
| --- | --- |
| List | Array, Linked List |
| Stack | Array, Linked List |
| Queue | Array, Linked List |
| Deque | Circular Array, Doubly Linked List |
| Priority Queue | Heap, Sorted Array |
| Dictionary (Map) | Hash Table, Binary Search Tree |
| Set | Hash Table, Tree |

# ADT vs Data Structure

| Abstract Data Type (ADT) | Data Structure |
| --- | --- |
| Defines **what** operations are performed | Defines **how** the operations are implemented |
| Logical concept | Physical implementation |
| Independent of programming language | Language-dependent implementation |
| Focuses on behavior | Focuses on memory organization and algorithms |
| Example: Stack ADT | Example: Stack implemented using an Array or Linked List |

# Relationship Between ADT and Data Structure