

UNIT 1

Introduction to Data Structure

(2 Hours) — Data Structures and Algorithms (CACS201)

1.1 Definition of Data Structure

A **data structure** is a specific, systematic way of organizing, storing, and managing data in a computer so that it can be accessed and modified efficiently. In other words, it is a logical or mathematical arrangement of data elements in the memory of a computer, together with the set of operations required to manipulate them.

Formally, a data structure can be defined as a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. Every program works with data, and how that data is arranged directly affects how fast and how easily a program can search, insert, delete, or update information.

Example — Why Organization Matters

Suppose 10,000 student records need to be searched by roll number.

If the records are stored as an **unsorted list**, the computer may have to check every single record — up to 10,000 comparisons in the worst case.

If the same records are stored in a **sorted array** or a **binary search tree**, the search can be completed in about 14 comparisons using binary search ($\log_2 10000 \approx 14$).

Note: The choice of data structure can change an algorithm's running time from thousands of steps to just a handful of steps for the same task.

1.2 Abstract Data Type (ADT)

An **Abstract Data Type (ADT)** is a theoretical, mathematical model of a data structure that specifies *what* operations can be performed on the data, without specifying *how* those operations are implemented internally. It separates the logical (abstract) view of data from its physical (concrete) implementation.

An ADT is defined by three components:

- **Data** — the collection of values the type can hold.
- **Operations** — the set of actions that can be performed on the data (e.g., insert, delete, search).

- **Semantics / Rules** — the behaviour and constraints that govern how the operations affect the data (e.g., a Stack always removes the most recently added element first).

Because an ADT hides implementation details, the same ADT can be implemented in more than one way — for example, a Stack ADT can be implemented using an array or using a linked list, and a user of the Stack does not need to know which one is used underneath.

Example — Stack as an Abstract Data Type

Data: a collection of elements arranged in Last-In-First-Out (LIFO) order.

Operations: push(item), pop(), peek(), isEmpty().

Rule: the element removed by pop() must always be the last one inserted by push().

The user only needs to know *what* push() and pop() do — not whether the Stack is built internally on an array or a linked list.

ADT	Typical Operations	Underlying Implementation
List	insert, delete, traverse, search	Array or Linked List
Stack	push, pop, peek, isEmpty	Array or Linked List
Queue	enqueue, dequeue, front, isEmpty	Array or Linked List
Tree	insert, delete, search, traverse	Linked Nodes (pointers)
Graph	addVertex, addEdge, traverse (BFS/DFS)	Adjacency List / Matrix

1.3 Classification of Data Structures

Data structures are broadly classified into **primitive** and **non-primitive** types. Primitive data structures (int, float, char, boolean) are the basic building blocks directly supported by a programming language. Non-primitive data structures are built using primitive types and are further divided into **linear** structures, where elements are arranged sequentially (Array, Stack, Queue, Linked List), and **non-linear** structures, where elements are arranged hierarchically or in a networked fashion (Tree, Graph).

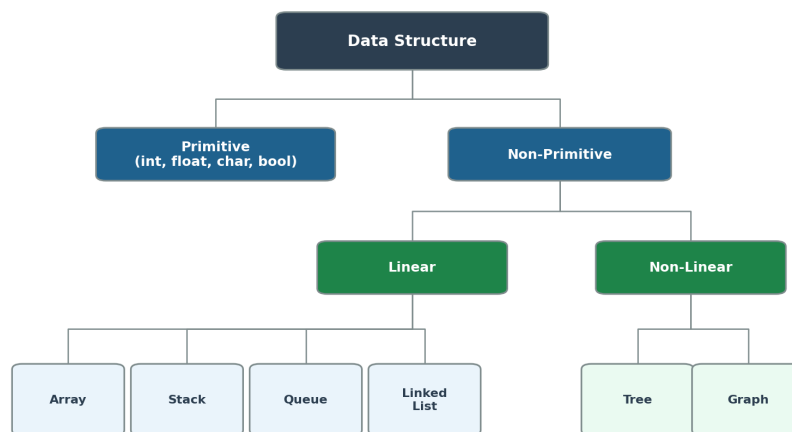


Figure 1.1 — Classification of Data Structures

1.4 Importance of Data Structure

The choice of an appropriate data structure is one of the most important decisions in designing efficient software. Its importance can be understood through the following points:

- **Efficiency:** A well-chosen data structure reduces the time and memory required to perform operations such as searching, sorting, insertion, and deletion.
- **Reusability and Abstraction:** ADTs allow data structures to be reused across programs without depending on internal implementation details.
- **Manageability of Large Data:** Real-world applications (databases, search engines, operating systems) deal with massive volumes of data that must be organized systematically.
- **Foundation for Algorithm Design:** Almost every algorithm — sorting, searching, graph traversal, dynamic programming — is built on top of one or more data structures.
- **Resource Optimization:** Proper structures help minimize CPU time and memory usage, which is critical in systems with limited resources such as embedded devices.

Illustration — Choosing the Right Structure

Task: A web browser needs to implement the "Back" button functionality.

Best-fit structure: Stack — because the most recently visited page must be the first one returned to (LIFO behaviour), which matches exactly how a Stack ADT works.

Wrong choice: using a plain unordered list would require extra logic to track visiting order, making the operation slower and more error-prone.

Note: A rule of thumb: analyse how data will be accessed (by order added, by key, by position, or by relationship) before selecting a data structure — this decision affects every operation built on top of it.

Key Takeaways

- ✓ A data structure organizes data in memory so it can be accessed and modified efficiently.
- ✓ An Abstract Data Type (ADT) defines *what* operations are available, hiding *how* they are implemented.
- ✓ Data structures are classified as Primitive and Non-Primitive; Non-Primitive further splits into Linear (Array, Stack, Queue, Linked List) and Non-Linear (Tree, Graph).
- ✓ Choosing the right data structure directly affects the speed, memory usage, and scalability of a program.

— *End of Unit 1* —

bcalink.com