Data Structure

**Session : Data types, data structures, Abstract data types**

* **Introduction – topics outline**
  + Uses of data, information.
  + Introduction to Data Structure.
  + Introduction to abstract data types.
* **Data**
  + What is the most important factor for any event/task to take place?
* Any task or job done involves the movement/ processing of data.
* Definition : data is a collection of raw facts and figures. Ex. Number(10,20), Word(“apple”).
* What is Data?
  + Data means unprocessed information.
  + It doesn’t have meaning of its own until it is organized.
  + Ex. 90, 85, 70 – just numbers (data).
  + When we say “marks of student” id becomes information.
* Uses of Data.
  + To store information in computer (like student records, photos, videos).
  + To analyse and make decisions (sales data – which product sells more).
  + To reduce errors
  + To build applications
* Data is Expensive !!!
* In the context of an organization, data are the building blocks that, when aggregated, analyzed, and interpreted, becomes information.
* **Data management**
  + Definition : data management is the process of collecting, storing, organizing, and using data efficiently.
  + It makes sure data is accurate, available, and secure.
* **Why important?**
  + Without management -> data becomes messy and useless.
  + With management -> we can easily find, update, and use data.
* **Use of data management**
  + Helps id decision making.
  + Reduces redundancy (avoids storing the same data many times).
  + Increases security (only authorized people can access data).
  + Improves efficiency (faster access and updates).
* Ex.
  + **In a Banking Ststem:**
    - Customer info, transactions, loans –> all managed in a database.
    - Data management ensures no duplication, safe storage, and quick access.
* **Data structure in programming**
  + **Need of knowledge of data structures.**
    - Programs handle a lot of data -> we need a systematic way to store and use it.
    - Good data structures make programs **faster, memory-efficient, and easier to manage.**
    - Ex. Searching in a stored list (binary search) is faster than in an unsorted one.
  + **Data representation**
    - Data can be stored in **different formats** inside memory:
      * Primitive types : int float, char,etc.
      * Non-primitive types: arrays, linked list, trees, graphs.
    - Representation decides **how data is stored and accessed.**
  + **Relationships Among Data:**
    - Data items are not always isolated -> they often relate.
      * Ex. Student -> roll no, name, marks
    - Data structures define **how elements are connected:**
      * Sequential (array, list).
      * Hierarchical(tree).
      * Networked(graph).
  + **Organization of data:**
    - Organizing data properly = easier operations.
    - Common organizations:
      * **Linear –** elements in a sequence (array, stack, queue).
      * **Non-liner –** elements connected different ways (tree, graph).
    - Proper organization improves **search, insertion, deletion** performance.
  + **Definition of data structure**
    - A data structure is a way to store, organize, and manage data in computer memory so it can used efficiently.
    - Ex.
      * Array – stores elements In continuous memory.
      * Linked list – stores elements with pointers to next node.
  + **Rules for Data Access**
    - Each data structure has its own **rules to access and modify data:**
      * Array – access by index.
      * Stack – LIFO (Last in, First out).
      * Queue – FIFO (first in, first out).
      * Tree – follow parent-child links.
      * Graph – follow edges between nodes.

->>> Data Structure = Organized Data + Allowed Operations

- It is Important for the data to be arranged in an organized manner for the applications to make use of the data relationships.

- **Data Structure** studies the organization of data in computers, consisting, of data types, relationship between elements of this type, operations on data types.

- **Algorithms :**

- methods to operate on data structures,

- trade of between efficiency and simplicity,

- subtle interplay with data structure design.

**Program = data structures + algorithm.**

* **Data Structure Types:**

1. **Primitive data structure**
   * **Definition :** Basic data types provided by programming languages.
   * Directly represent **single values.**
   * **Ex.** 
     + Integer -> whole number (….-2,-1,0,1,2,….)
     + Boolean -> logical values (true/false).
     + Character -> single alphabet, digit, or symbol (‘a, ‘9’, ‘$’).
2. **Non-Primitive Data Structures**
   * **Definition :** more complex data structures built using primitive types.
   * Can store large amounts of data in an organized way.
3. **Simple Data Structures**
   * + store data in structured but straight forward ways.
4. String -> Sequence of characters. Ex. “Hello”.
5. Array -> Collection of similar data elements stored at continuous memory locations. Ex. Marks[5].
6. Record (or structures) -> Collection of related data items of different types Ex. Student = {RollNo, Name, Marks}.
7. **Compound Data Structures:**
   * + Store data in multiple related elements.
     + Can be divided into Linear and Non-Linear.
8. **Linear Data Structures**
   * + Data arranged in sequential order.
     + Easy traversal but can slow for insertions deletions.
   1. **Stack ->** Works on LIFO (Last In, First Out). Ex. Undo operations in Word.
   2. **Queue ->** Works on FIFO (First In, First Out). Ex. People standing in line.
   3. **Linked List ->** Collection of nodes connected with pointers. Ex. Music playlist (next -> previous songs).
9. **Non-Linear Data Structures**
   1. **Binary**
      1. **Binary Tree ->** Each node has max 2 children.
      2. **Binary search Tree ->** special binary tree where: left child < root < right child.
   2. **N-ary(Generalized)**
      1. **Graph ->** collection of nodes connected by edges. Ex. Social media friends network.
      2. **General tree ->** a treewith any number of children. Ex. Folder structure in computer.
      3. **M-way search tree ->** generalized BST with multiple children per node.
      4. **B-tree ->** balanced search tree used in databases.
      5. **B, B+, Tree\* ->** variants of B-tree with improved search and storage.
      6. **Trie ->** tree structure for storing strings efficiently (used in autocomplete, dictionary).
10. **File Organizations data structures:**
    * + Used in secondary storage (disk) to manage large datasets.
        - **Sequential file organization**
          * Records stored one after another.
          * Ex. Roll numbers list.
        - **Relative file organization**
          * Records stored at calculated addresses.
          * Faster access using formula (like hashing).
        - **Indexed sequential file organization**
          * Index created for faster searching + sequential storage.
          * Ex. Book index + pages.
        - **Multi-key file Organization**
          * Multiple indexes maintained for different keys.
          * Ex. Searching student data by RollNo or Name.

Data Structures

│

├── 1. Primitive

│ ├─ Integer

│ ├─ Boolean

│ └─ Character

│

└── 2. Non-Primitive

├─ (A) Simple

│ ├─ String

│ ├─ Array

│ └─ Record

│

├─ (B) Compound

│ ├─ Linear

│ │ ├─ Stack

│ │ ├─ Queue

│ │ └─ Linked List

│ └─ Non-Linear

│ ├─ Binary

│ │ ├─ Binary Tree

│ │ └─ Binary Search Tree

│ └─ N-ary

│ ├─ Graph

│ ├─ General Tree

│ ├─ M-way Search Tree

│ ├─ B-Tree

│ ├─ B\*, B+ Tree

│ └─ Trie

│

└─ (C) File Organization

├─ Sequential

├─ Relative

├─ Indexed Sequential

└─ Multi-key

* **Abstract Data**
  + Abstract means not showing details – only focusing on what data represent, not how it id stored.
  + Abstract data means we hide the implementation details and only show the data and its operations.
  + Ex. Suppose you use a stack. You only know you can use operations like push, pop, peek you don’t need to know how It is stored inside.
  + So, abstract data = concept or idea of the data, without worrying about how it works internally.
* **Abstract data Types:**
  + Definition: an abstract data types defines what operation can be performed on data and what kind of data it holds, but does not define how these operations are implemented.
  + It is like a blueprint or model of a data structure.
  + Ex. Stack ADT push(), pop(), peek(), isEmpty().
* **key points / features:** 
  + ADT tells what to do not how to do it.
  + It provides abstraction and encapsulation (hides internal details).
  + ADT improves code reusability and maintainability.
  + EX of common ADTs:
    - Stack
    - Queue
    - List
    - Tree
    - Graph

In short:

* Abstract data = idea of data without implementation.
* Abstract data type = model that defines what operations can be done on the data (not how).

---------------------------------------------------------------------------------------------------------------------

* **Algorithm**
  + An algorithm id finite sequence of well-defined instructions designed to perform a specific task or solve a particular problem.
  + In simple words, it is a step-by-step procedure that transforms input data into desired output
  + It defines the logic of computation without depending on any particular programming language.
  + Ex.
    - To find the sum of two numbers.
    - Read two numbers a and b.
    - Add them -> c = a + b
    - Display c
  + **Need for an algorithm**
    - It helps in planning the logic before actual coding.
    - Makes programs easier to understand and debug.
    - Increases efficiency and reliability of solutions.
    - Allows comparison between different approaches to choose the best one.
* **Characteristics of good algorithm:**

1. Unambiguous
   * Every step in algorithm should be clearly defined and understandable.
   * There should be no confusion or ambiguity in meaning of any instruction.
   * Each operation must have a single interpretation.
   * If the same algorithm is executed multiple times, the out come of each step should always be clear.
   * Ex. Instead of saying “process the data “ say “sort the list in ascending order.”
2. Input
   * An algorithm should have zero or more inputs.
   * These are the initial data values supplied before the algorithm starts execution.
   * The input gives the algorithm the necessary information to begin processing.
   * Ex. In an algorithm to find the average of numbers, the list of numbers is the input.
3. Output
   * Every algorithm must produce al least one output.
   * The output represent the result obtained after performing the steps of the algorithm.
   * Without output, the algorithm serves no purpose.
   * Ex. In an average-finding algorithm, the final computed average value is the output
4. Finiteness
   * An algorithm must terminate after a finite number of steps.
   * It should not enter into an infinite loop or continue indefinitely.
   * Finiteness ensures that the algorithm gives the result in a limited amount of time.
   * Ex. A loop that repeats 10 times is finite, but loop that run “until the sun rises” is not
5. Effectiveness
   * Each step of the algorithm must be basic and feasible.
   * It should be possible to perform each operation manually or by computer within a reasonable time.
   * No step should be too complex or acstract.
   * Ex. “add two numbers” Is an effective step, but “think of a large number” id not
6. Generality
   * An algorithm should be general enough to handle a wide range of inputs for a particular problem type.
   * Is should not ne designed only for specific data or a single case.
   * This property ensures the reusability of the algorithm.
   * Ex. A shorting algorithm should sort any list of numbers, not just one specific list.
7. Sequential
   * The steps of an algorithm must be logically ordered.
   * Each step should follow the pervious one in a proper sequence to ensure correct results.
   * Skipping or rearranging steps can lead to incorrect output.
   * Ex. You cannot divide numbers before reading them – reading must come first.
8. Deterministic
   * For a given input, an algorithm should always produce the same output every times it is executed.
   * There should be no randomness or uncertainty involved in steps.
   * This makes the algorithm predictable and reliable.
   * Ex. An algorithm to find factorial(5) should always give 120.
9. Independent
   * An algorithm should be independent of any programming language or machine
   * It focuses only on logical steps, not on syntax or implementation.
   * This allows it to be implemented later in any language like c, java, or python.
   * Ex. Writing “step : read a and b” id independent, while “int a = scan.nextint()” is language dependent.

* **Importance of alogorithm analysis**

1. Efficiency measurement:
   * Algorithm analysis helps to measure how efficiently an algorithm performs in terms of execution time and memory usage.
   * Efficient algorithm are faster and require fewer resources.
   * Ex. Comparing sorting algorithms like bubble sort and quick sort helps us see that Quick sort is more efficient for large data sets.
2. Optimization
   * By analysing an algorithm, er can optimize it – that is, improve its performance by reducing the time and space complexity.
   * This helps in developing high-performance applications.
   * Ex, improving a loop or using a better data structure can make a program rum much faster.
3. Resource Utilization
   * Algorithm analysis helps to understand how much memory, CPU time, and other resources an algorithm consumes.
   * This ensures that the algorithm can run smoothly even on systems with limited resources.
4. Scalability :
   * A well-analyzed algorithm performs efficiently even when the input size grows.
   * This means it can handle large data without drastically increasing execution time.
   * Hence, analysis ensures scalability of software.
   * Ex. An algorithm with O(n) time complexity scales better than one with O(n2)when input increases.
5. Comparison Between algorithms
   * When multiple algorithms solve the same problem, analysis helps in comparing their performance and selecting the most suitable one.
   * Ex. For searching, binary search ( O(log n) ) is better then linear search (O(n)).
6. Predicting Performance
   * Before actual implementation, analysis allows us to predict how an algorithm will behave for different input sizes.
   * This helps developers design system that can meet performance goals.
7. Cost and time saving
   * Efficient algorithms reduce execution time and resource cost, making applications more practical and user-friendly.
   * This is especially important in real-time systems, where delay can lead to failure
8. Foundation for algorithm design
   * Analysis gives insights into how algorithms work, helping programmers design better, faster, and more reliable algorithms in the future.

* **Multiple solutions to the same problem**
* Introduction
  + In computer science, there is often more than one way to solve the same problem.
  + Each solution may use a different algorithm, logic, or data structure.
  + These are called multiple solutions for the same problem.
  + However, all these solutions may produce the same result but differ in how efficiently they use time, memory, and other resources.
  + That is why we analyze different solutions to find the best and most efficient one.
* Definition
  + When a problem can be solved by two or more algorithms, each having a different approach, they are said to be multiple solutions to that problem.
  + All of them will give the same output, but their performance, speed, and complexity may vary.
* Why multiple solutions exits
  + Different programming logics can achieve the same goal.
  + Each algorithm may have a different trade-off between time and space.
  + The nature of input data (size, type, order) affects which algorithm works best.
  + Some algorithms are designed for simplicity, while other for efficiency.
* Importance of having multiple solutions.
  + Flexibility developers can choose the best algorithm according to the situation.
  + Optimization: allows comparison to select the fastest or least memory-consuming.
  + Scalability – different algorithms perform better with different input sizes.
  + Problem Understanding – multiple solutions help understand different ways to approach to problem
  + Performance tuning – in critical systems, choosing the right algorithm can greatly improve performance.
* **Evaluation beyond running time**
* Introduction:
  + When we evaluate an algorithm, the most common factor considered is its running time, i.e, how long it takes to execute.
  + However, running time along does not fully describe the performance of an algorithm.
  + In real-world applications, we must also consider other factors like memory usage, implementation complexity, scalability, reliability, and maintainability.
  + Therefore, algorithm evaluation must go beyond running time to judge its overall effectiveness.
* Meaning
  + Evaluation beyond running time means analyzing an algorithm based on multiple performance aspects, not only on how fast it runs.
  + These aspects help determine whether an algorithm is practical, efficient, and suitable for real-world use.
* Factors considered beyond running time

1. Space complexity
   * It measures the amount of memory (ram) an algorithm needs to execute.
   * Some algorithms may run faster but use a lot of memory.
   * In memory-limited environment efficiency is very important
   * Ex. Merge sort Is fast but uses extra memory, while quick sort uses less memory.
2. Implementation complexity
   * It refers to how easy or difficult it is to implement an algorithm.
   * A simple algorithm is easier to write, debug, and maintain.
   * Sometimes, a slightly slower but simpler algorithm is preferred over a fast but complex one.
   * Ex. Bubble sort is easier to implement than quick sort, even though it is slower.
3. Readability and maintainability
   * The algorithm should be easy to understand and modify.
   * Readable algorithms are important for team project and long-term software maintenance.
   * Complex algorithms may be hard to maintain even if they perform well.
4. Scalability.
   * Scalability means the ability of an algorithm to handle larger input sizes efficiently.
   * A scalable algorithm performs well even when data size grows exponentially.
   * Evaluation must include how the algorithm behaves for small, medium , and large input.
   * Ex. An algorithm that runs fine for 1000 elements but become too slow for 1,000,000 element is not scalability.
5. Portability
   * A good algorithm should be portable, meaning it can run on different hardware or operating systems with minimal changes.
   * This ensure the algorithm can be reused across different enviroments.
6. Reliability and accuracy
   * The algorithm should always give the correct and expected output for all valid input.
   * It should handle error, exceptions, or invalid data gracefully without crashing.
7. Flexibility and adaptability
   * An algorithm should be flexible enough to adapt when the problem changes slightly.
   * Highly rigid algorithms may not be useful if the requirements evolve.
   * Ex. A sorting algorithm that can easily switch between ascending and seceding order is more flexible.
8. Resource utilization
   * Apart form time and memory, other resources like disk space network bandwidth, and power consumption can also matter.
   * Especially in mobile and IOT applications, low energy usage is an important factor.
9. Security
   * Some algorithms must also be analyzed for security aspects – to ensure they are not vulnerable to attacks or misuse.
   * This is especially true for cryptographic and data-processing algorithms.