Data structure

A **data structure** is a way of organizing and storing data so that it can be used efficiently. It helps in managing data for operations like searching, sorting, and modifying while optimizing memory usage.

1. Field:

It represents a single piece of information about an entity.
Example: In a student record, fields could be **Name, Age, Roll Number, and Address**.

2. Record:

A **record** is a collection of related fields that together describe a single entity.
⇕ Example: A student's record may have fields like **"Ali, 20, 101, Lahore"**.

3. File:

A **file** is a collection of related records stored together. It is used to manage and organize data efficiently.

Entity

An **entity** is anything that has a unique identity and can store data about it. It can be a real-world object, person, place, or concept.

Examples of Entities:

1. **Person** – Student, Teacher, Employee

Difference Between Linear and Non-Linear Data Structure

Feature	Linear Data Structure	Non-Linear Data Structure
Definition	Data elements are arranged sequentially, one after another.	Data elements are connected in a hierarchical or complex manner.
Storage	Stored in a continuous memory location.	Stored in random memory locations with links between elements.
Traversal	Can be traversed in a single run (sequentially).	Requires multiple paths to traverse all elements.
Complexity	Simple to implement and manage.	More complex due to multiple connections.
Examples	Array, Linked List, Stack, Queue	Tree, Graph

Algorithm

An **algorithm** is a step-by-step procedure or a set of rules that takes some values or set of values as input and produce some values or set values as output to solve a specific problem. It is like a recipe that tells a computer what to do, in what order, and how to process data to get the desired output.

Characteristics of an Algorithm:

1. **Input** – Takes zero or more inputs.
2. **Output** – Produces at least one output.
3. **Definiteness** – Each step must be clear and unambiguous.
4. **Finiteness** – Must have a limited number of steps.
5. **Effectiveness** – Each step must be basic enough to be executed.

Complexity of an Algorithm

The **complexity of an algorithm** refers to how much time and space (memory) it requires as the input size grows. It helps in analyzing the efficiency of an algorithm.

Types of Complexity:

1. **Time Complexity** – Measures the time an algorithm takes to run based on input size.
2. **Space Complexity** – Measures the memory required by an algorithm during execution.

Asymptotic Efficiency of an Algorithm

Asymptotic efficiency refers to the performance of an algorithm as the input size (**n**) grows very large. It helps compare different algorithms based on their **time complexity** and **space complexity** without focusing on machine-specific details.

Asymptotic Notations

To describe an algorithm's efficiency, we use three main asymptotic notations:

1. **Big-O (O) – Upper bound (worst case)**
   - Represents the maximum time an algorithm can take.
   - Example: **$O(n^2)$** means the time taken will not exceed **n^2** operations in the worst case.
   - **Example Algorithm: Bubble Sort ($O(n^2)$)**
2. **Omega (Ω) – Lower bound (best case)**
   - Represents the minimum time an algorithm takes in the best case.
   - Example: **$\Omega(n)$** means the algorithm runs at least in **n** operations.
   - **Example Algorithm: Insertion Sort ($\Omega(n)$ for best case when already sorted).**
3. **Theta (Θ) – Tight bound (average case)**
   - Represents both the upper and lower bound, meaning it defines the exact time complexity.

- Example: **Θ(n log n)** means the algorithm runs in **n log n** time in most cases.
- **Example Algorithm: Merge Sort (Θ(n log n))**

Array (Linear Array)

An **array** is a **collection of elements of the same data type**, stored in **contiguous memory locations**. It is a **linear data structure**, meaning elements are arranged in a sequence, and each element is accessed using an **index**.

1. Row-Major Order

- **Stores elements row by row** in memory.
- **Left to right, one row at a time**.

2. Column-Major Order

- **Stores elements column by column** in memory.
- **Top to bottom, one column at a time**.

Bubble Sort Algorithm

Bubble Sort is a simple sorting algorithm that repeatedly **compares and swaps** adjacent elements if they are in the wrong order. This process continues until the array is sorted.

Sorting Algorithms: Selection Sort, Bubble Sort, and Insertion Sort

1. Selection Sort

- **Idea**: Repeatedly find the **smallest** element and place it in the correct position.
- **Steps**:
  1. Find the **minimum element** in the unsorted part.
  2. Swap it with the first element of the unsorted part.
  3. Repeat until the array is sorted.
- **Time Complexity**:
  - **Best case (already sorted):** $O(n^2)$
  - **Worst case:** $O(n^2)$
  - **Average case:** $O(n^2)$
- **Space Complexity**: $O(1)$ (In-place sorting, no extra space needed)

---

2. Bubble Sort

- **Idea**: Repeatedly compare adjacent elements and **swap if needed** to move the largest element to the end.

- **Steps**:
  1. Compare and swap adjacent elements if they are in the wrong order.
  2. After each pass, the largest element gets placed at the correct position.
  3. Repeat until sorted.
- **Time Complexity**:
  - **Best case (already sorted):** $O(n)$ (optimized version)
  - **Worst case:** $O(n^2)$
  - **Average case:** $O(n^2)$
- **Space Complexity**: $O(1)$ (In-place sorting)

---

3. Insertion Sort

- **Idea**: Take elements one by one and **insert them into the correct position** in the sorted part.
- **Steps**:
  1. Start from index 1 (assuming the first element is sorted).
  2. Pick an element and shift all larger elements to the right.
  3. Insert the element in its correct position.
  4. Repeat until sorted.
- **Time Complexity**:
  - **Best case (already sorted):** $O(n)$
  - **Worst case:** $O(n^2)$
  - **Average case:** $O(n^2)$
- **Space Complexity**: $O(1)$ (In-place sorting)

Comparison of Sorting Algorithms

Algorithm	Best Case	Worst Case	Average Case	Space Complexity	Stability
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes

Conclusion:

- **Insertion Sort** is best for nearly sorted data.
- **Bubble Sort** is simple but inefficient.
- **Selection Sort** performs the same in all cases but is not stable.

Let me know if you need code for any of these! ☺