What are Data Structures and Algorithms Terminology

- **Data structure** — a way to store and organize data so you can use it efficiently (examples: arrays, linked lists, trees, hash maps).
- **Algorithm** — a step-by-step procedure to solve a problem (examples: sorting, searching, shortest path).
- **Interplay:** choosing the right data structure often makes the algorithm simpler/faster (e.g., using a hash map for fast lookup).

What is a Data Structure

- A **data structure** is a way of **organizing, storing, and managing data** so that it can be used efficiently.
- It provides operations like **insertion, deletion, searching, updating, and traversal**.
- Example: Imagine a library – books can be organized in different ways (alphabetically, by subject, by author). The method you choose to organize them is like a data structure.

Why are Data Structures important

- Helps in writing **efficient programs**.
- Saves **time** (faster access, searching, processing).
- Saves **memory** (proper arrangement of data).
- Forms the **base of algorithms** used in real-world applications (databases, OS, AI, compilers, etc.).

Types of Data Structures

(A) Primitive Data Structures

- Basic data types provided by programming languages.
- Examples in Java:
  - int, float, char, boolean, double.
- These store single values (atomic).

(B) Non-Primitive Data Structures

Abstract Data Types (ADT)

- These define **what operations** can be performed but **not how** they are implemented.
- Examples:
  - **List** (ordered collection of items).
  - **Stack** (push, pop).
  - **Queue** (enqueue, dequeue).
  - **Tree**, **Graph**.
- These are more advanced structures built using primitive types.
- They are divided into two categories:

1. Linear Data Structures

- Data elements are arranged **sequentially**, one after another.
- Easy to traverse (using loops).
- Examples:
    1. **Array** – fixed-size, continuous memory, fast access.
    2. **Linked List** – dynamic memory, nodes connected by pointers.
    3. **Stack** – LIFO (Last In First Out), e.g., undo operation in Word.
    4. **Queue** – FIFO (First In First Out), e.g., printer queue, ticket line.

2. Non-Linear Data Structures

- Data elements are arranged **hierarchically or as a network**.
- One element can connect to multiple elements.
- Examples:
    1. **Tree** – hierarchical structure (root → children → leaves).
        - Binary Tree, Binary Search Tree, AVL, Heap, etc.
    2. **Graph** – set of nodes (vertices) and edges (connections).
        - Social networks, maps, routes.

Basic Operations on Data Structures

Almost all DS support these fundamental operations:

Operation	Description
Traversal	Accessing each data element exactly once
Insertion	Adding a new element
Deletion	Removing an element
Searching	Finding a specific element
Sorting	Arranging elements in order (ascending/descending)
Merging	Combining two or more data structures

Some Basics of Data Structures

1. Decimal to Binary

1. Take the decimal number.
2. Divide it by 2.
3. Write down the remainder (0 or 1).
4. Update the number = quotient of division.
5. Repeat steps 2–4 until the number becomes 0.
6. Write all remainders in **reverse order** → That's the binary.

Example:
Decimal = 13

Step	Divide by 2	Quotient	Remainder
1	13 ÷ 2	6	1
2	6 ÷ 2	3	0
3	3 ÷ 2	1	1
4	1 ÷ 2	0	1

Write remainders in reverse → **1101** → Binary

2. Binary to Decimal

1. Take the binary number.
2. Start from the **rightmost bit**, assign position 0.
3. Multiply each bit by **2^position**.
4. Add all the results → That's the decimal number.

Example
Binary = 1101

Bit (from right)	Position	Multiply by 2^position	Value
1	0	1 × 2^0	1
0	1	0 × 2^1	0
1	2	1 × 2^2	4
1	3	1 × 2^3	8

Add all → 1 + 0 + 4 + 8 = **13** → Decimal

Bitwise Operators in Java

1. Overview

- **Definition:** Bitwise operators work **on individual bits** of integer types (`int`, `long`, `byte`, `short`).
- **Purpose:** Efficient manipulation of bits, fast computations, and low-level operations.
- **Common Uses:** Masking, checking bits, setting bits, toggling, swapping, competitive programming, encryption.

List of Bitwise Operators

Operator	Symbol	Description
AND	&	Bitwise AND
OR	|	Bitwise OR
XOR	^	Bitwise Exclusive OR
NOT	~	Bitwise Complement (NOT)
Left Shift	<<	Shift bits left, fill 0 on right
Right Shift	>>	Shift bits right, sign-extended
Unsigned Right Shift	>>>	Shift bits right, fill 0 on left

Bitwise AND (&)

- **Definition:** 1 if **both bits are 1**, else 0
- **Truth Table:**

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

- **Java Example:**

```
int a = 5; // 101
int b = 3; // 011
System.out.println(a & b); // 1
```

Bitwise OR (|)

- **Definition:** 1 if **any bit is 1**, else 0
- **Truth Table:**

A	B	A | B
0	0	0

A	B	A | B
0	1	1
1	0	1
1	1	1

- **Java Example:**

```
int a = 5; // 101
int b = 3; // 011
System.out.println(a | b); // 111 = 7
```

- **Applications:** Setting bits, combining masks, flag operations.

5. Bitwise XOR (^)

- **Definition:** 1 if **bits are different**, else 0
- **Truth Table:**

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

- **Java Example:**

```
int a = 5; // 101
int b = 3; // 011
System.out.println(a ^ b); // 110 = 6
```

- **Applications:** Swapping numbers without temp, finding unique elements.

6. Bitwise NOT (~)

- **Definition:** Flips all bits (0 → 1, 1 → 0)
- **Example:**

```
a = 5 → 00000101
~a = 11111010 → -6 (Two's complement)
```

- **Java Example:**

```
int a = 5;
System.out.println(~a); // -6
```

- **Applications:** Complement operations, negative numbers handling, masking.

7. Shift Operators

Left Shift (<<)

- Shifts bits left by specified positions, **fills 0 on the right**
- Equivalent to multiplying by 2^n
- **Example:**

```
int a = 5; // 101
System.out.println(a << 1); // 1010 = 10
```

Right Shift (>>)

- Shifts bits right, **sign bit (MSB) is preserved**
- Equivalent to dividing by 2^n (floor for positive, ceil for negative)
- **Example:**

```
int a = 10; // 1010
System.out.println(a >> 1); // 0101 = 5
int n = 5; // 101
int pos = 0;
n = n ^ (1 << pos); // 100 = 4
```

Quick Tips / Summary

Operator	Function	Example
&	AND	Masking, checking even
|	OR	Setting bits, flags
^	XOR	Swap without temp, unique element
~	NOT	Bit complement
<<	Left Shift	Multiply by 2^n
>>	Right Shift	Divide by 2^n (signed)
>>>	Unsigned Right Shift	Divide by 2^n (fill 0)

Key Points:

1. Bitwise ops are **fast (O(1))**
2. Widely used in **competitive programming and system-level tasks**
3. Mastery of **masking, toggling, setting/clearing bits** is essential