- **Data -**
  Data can be name, email, country etc.

- **DataStructure -**
  - A way of organizing and storing data that can be accessed and modified efficiently.
  - Data structures define the layout(stack, array etc) of data memory and improve the performance of operations like search, sorting.
  - Ex :
    - Arrays
    - LinkedList
    - Graph
    - Tree

- **Algorithm**
  - A finite sequence of well-defined instructions to solve a problem or perform a task. It focuses on optimizing time and space.
  - Ex - Searching, Sorting

Why DSA:
- Optimization
- Coding Interview
- Real-world application

- **About Complexities**
  A. To determine the efficiency of program, we have
    - Time Complexity: Tells us the how much time our code takes to run
    - Space Complexity: Tells how much memory our codes use.

- **Asymptotic Notation**
  - We use the asymptotic notation to compare the efficiencies of algorithms.
  - It is a mathematical tool that estimates the time based on input size while running the code.
  - It gives the idea how the algorithm behaves as input size increases.

Type of Asymptotic
- Big O : Describe the worst case scenario as input size increase
- Omega: Describe the best case scenario
- Theta: Describe the avg-case scenario as input size increases.

Complexities

1. Big(10 - Constant Time -
- No matter how big element is, function will take constant time.
- For ex - array[1] - to find it, it will take constant time.

2. O(n) - Linear Time
- TIme Required is based on the size of element
- For ex - array{1,3,4,5} - For loop will take time - traversing

3. O(n2) - Quadratic Time
- Input grows, time taken grows quadratically.
- Ex - Bubble Sort, checking pairs in the array
  Suppose to find the largest element in array.

4. O(log`n) - Logarithmic Array
- Inuput size array divide into half.
- For ex - Binary search - suppose we want to find the largest number in array.
  - We will sort the array
  - Check if middle number is highest
  - Then will cut that array in half (to find largest)

5. O(nlogn0 - Linearamatic Array
- Divide the input in subproblem and process each subproblem linearly.
- Merge sort or Quick sort etc.

6. O(n!) - Factorial Time Complexity
- It involves generating all the permutation and combination of a set.

7. O(2`n) - Exponential Time Complexity
- Time grows with the size of the input, which means it double with each additional input.
- Recursion algorithm - fibonacci series.

Linear Search :

Linear search is a straightforward algorithm used to find the position of a target element within a list. It checks each element in the list sequentially until it finds the target element or reaches the end of the list.

How Linear Search Works :

1. Start from the beginning: Begin with the first element of the array or list.

2. Check each element: Compare the current element with the target element.

3. Move to the next element: If the current element does not match the target, move to the next element.

4. Stop if found or at the end: If a match is found, return the index of the element. If the end of the list is reached and no match is found, return an indication (e.g., -1) that the element is not present in the list.

Algorithm :

Here’s a step-by-step algorithm:

1. Input: An array arr of size n , and a target element x .

2. For each element arr[i] in arr : If arr[i] == x , return i .

3. If the loop completes without finding x , return 1 .

Binary Search :

Binary search is a highly efficient algorithm used to find an element's position in a sorted array. Unlike linear search, which checks each element one by one, binary search divides the search space in half with each step, making it faster.

Here's a detailed explanation:

1. Prerequisites Sorted Array: Binary search can only be applied to a sorted array. If the array is not sorted, the results will be unpredictable.

2. Access to Middle Element: The algorithm requires the ability to access the middle element directly, so random access is essential (like in arrays or lists).\

2. How It Works Binary search follows a divide-and-conquer approach.

Here are the steps:
1. Initialize Pointers: Start with two pointers: low (pointing to the first element of the array) and high (pointing to the last element of the array).

2. Find the Midpoint: Calculate the middle index: mid=low+2high−low
mid=low+high−low2\text{mid} = \text{low} + \frac{\text{high} - \text{low}}{2}

3. Compare the Middle Element: If the middle element equals the target, you've found the element, and the search ends. If the middle element is less than the target, move the low pointer to mid + 1 (discarding the left half). If the middle element is greater than the target, move the high pointer to mid - 1 (discarding the right half).

4. Repeat Until Found or Exceeded: Continue this process until the target is found or the low pointer exceeds the high pointer (meaning the target is not present in the array).

Linked List :

A **Linked List** is a linear data structure where each element (called a node) contains two parts: the data and a reference (or pointer) to the next node in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations, making insertions and deletions more efficient.

How Linked List Works :

1. **Node Structure**:
   Each node has two fields – one for storing data and the other for storing the address of the next node.

2. **Starting Point**:
   The list starts with a special node called the *head*, which points to the first node in the list. If the list is empty, the head is set to `null`.

3. **Traversal**:
   To access elements, traversal starts from the head node and continues by following the pointers until the end of the list (where the next pointer is `null`).

4. **Insertion & Deletion**:
   New nodes can be inserted or deleted by adjusting the next pointers of the adjacent nodes, making these operations efficient compared to arrays (no shifting of elements is required).

Types of Linked Lists :

1. **Singly Linked List**:
   Each node points to the next node only.

2. **Doubly Linked List**:
   Each node points to both the next and the previous nodes.

3. **Circular Linked List**:
   The last node points back to the head, forming a circle.

Basic Operations on Linked List :

1. **Insertion**:

   - At the beginning

   - At the end

   - After a given node

2. **Deletion**:

   - From the beginning

   - From the end

   - A specific node

3. **Search**:
   Traverse the list to find a node with the desired value.

4. **Traversal**:
   Visit each node starting from the head to process data or display the list.

Stack :

A **Stack** is a **Last In First Out (LIFO)** data structure — the last element added is the first one to be removed.

Think of it like a stack of plates:

- **Push**: Add plate on top
- **Pop**: Remove plate from top
- **Peek**: View top plate without removing

Use Cases:

- Undo feature in editors
- Expression evaluation
- Parentheses matching
- DFS in graphs

Queue :

A Queue is a linear data structure that follows the First In, First Out (FIFO) principle, meaning the element added first will be removed first. It is similar to a real-life queue (e.g., a line at a ticket counter).

Key Operations:
1. Enqueue: Adding an element to the end of the queue.
2. Dequeue: Removing an element from the front of the queue.
3. Peek/Front: Getting the front element without removing it.
4. IsEmpty: Checking if the queue is empty.
5. IsFull: (For fixed-size queues) Checking if the queue is full.

Types of Queues
Queues come in different variations based on their structure and behavior. Below are the main types of queues with their descriptions and examples:

1. Simple Queue (Linear Queue)
Definition: A basic queue that follows the First In, First Out (FIFO) principle. Elements are added at the rear and removed from the front.

Key Operations:
Enqueue: Add an element at the rear.
Dequeue: Remove an element from the front.
Limitation: Once the queue is full, no more elements can be added even if some are removed (unless implemented as circular).

Use Case: Customer service ticketing system.

```
Queue<Integer> queue = new LinkedList<>();
queue.add(10); // Enqueue
queue.add(20);
System.out.println(queue.poll()); // Dequeue: 10
```

2. Circular Queue

Definition: A queue where the rear pointer wraps around to the front when the the end of the array is reached, making better use of storage.

Key Difference: Unlike a simple queue, it efficiently utilizes the array by reusing freed spaces.

Use Case: CPU scheduling, memory management.

Enqueue: Rear → Index wraps back to 0
Dequeue: Front → Index wraps back to 0

3. Priority Queue

Definition: A queue where each element has a priority. Elements with higher priority are dequeued before those with lower priority, regardless of their order of arrival.

Behavior:
Highest Priority: Removed first
Same Priority: Follow FIFO for elements of equal priority.

Use Case: Task scheduling, network packet management.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(15); // Priority 1
pq.add(10); // Priority 2
pq.add(20); // Priority 0 (smallest value gets highest priority)
System.out.println(pq.poll()); // Output: 10
```

4. Deque (Double-Ended Queue)
Definition: A queue where elements can be added or removed from both ends (front and rear).

Variants:
Input-restricted deque: Insertion allowed at one end, deletion allowed at both ends.

Output-restricted deque: Deletion allowed at one end, insertion allowed at both ends.

Use Case: Palindrome checking, caching mechanisms (e.g., LRU Cache).

```
ArrayDeque deque = new ArrayDeque<>();
deque.addFirst(10); // Add at front
deque.addLast(20); // Add at rear

System.out.println(deque.removeFirst()); // Remove from front
System.out.println(deque.removeLast()); // Remove from rear
```

5. Double Priority Queue

Definition: A priority queue that allows removal of both the highest and lowest-priority elements.

Behavior:
Removal happens based on either maximum or minimum priority.

Use Case: Complex scheduling systems.

6. Concurrent Queue

Definition: A thread-safe queue designed for use in multi-threaded Environments.

Types in Java:
ConcurrentLinkedQueue : Non-blocking queue.
BlockingQueue : Blocks when trying to enqueue/dequeue in full/empty Conditions.

Use Case: Producer-consumer problems in multithreading.

7. Double-Ended Priority Queue (DEPQ)

Definition: A combination of deque and priority queue where elements can be added or removed from both ends with priority considerations.

Use Case: Advanced data management systems

—---TREE IS IN THE PPT—-------------------------------------