1. Basic Terminologies in Data Structures

Data structures organize data efficiently for processing. Basic terms include **data**, **information**, and **data items** (individual values). A **data type** defines the nature of data (int, float, etc.), while an **abstract data type (ADT)** specifies operations without implementation details. **Data organization** refers to arrangement methods like arrays or lists. Structures can be **linear** (arrays, stacks) or **non-linear** (trees, graphs). Understanding these terms helps in selecting appropriate structures for efficient storage, retrieval, and manipulation of data in algorithms.

2. Data Structure Operations

Common operations include **insertion**, **deletion**, **traversal**, **searching**, and **sorting**. Insertion adds an element, deletion removes it, and traversal visits all elements. Searching finds a specific element, while sorting arranges data in order. Each operation has a cost measured in time and space complexity. Efficient data structures minimize these costs. For example, arrays allow fast access but slow insertion, while linked lists provide dynamic insertion but slower access. Choosing the right structure depends on the operation frequency and application requirements.

3. Analysis of Algorithms

Algorithm analysis evaluates performance in terms of **time and space complexity**. It helps compare different approaches to solve a problem. Analysis can be **best case**, **worst case**, or **average case**. Time complexity measures execution steps, while space complexity measures memory usage. For example, searching in an array may take O(1) (best) or O(n) (worst). Efficient algorithms reduce computational cost and improve scalability. Algorithm analysis is essential for designing optimized programs, especially for large datasets.

4. Asymptotic Notations

Asymptotic notations describe algorithm efficiency for large inputs. The main types are **Big O (O)**, **Omega (Ω)**, and **Theta (Θ)**. Big O gives the upper bound (worst case), Omega gives the lower bound (best case), and Theta gives the exact bound. For example, linear search has O(n) complexity, while binary search has O(log n). These notations ignore constants and focus on growth rate. They help in comparing algorithms and predicting performance without running the code.

5. Time-Space Trade-off

Time-space trade-off means optimizing either execution time or memory usage. Faster algorithms may require more memory, while memory-efficient algorithms may take more time. For example, using hashing improves search speed but consumes extra space. Similarly, recursion may use more stack space compared to iteration. Choosing the right balance depends on system constraints like memory availability and processing speed. This concept is crucial in real-world applications like databases, operating systems, and embedded systems.

📘 Module 2: Stacks and Queues

1. Stack ADT and Operations

A stack is a **LIFO (Last In First Out)** structure. Basic operations include **push** (insert), **pop** (remove), and **peek** (view top element). It can be implemented using arrays or linked lists. Stack overflow occurs when it is full, and underflow when empty. Time complexity of operations is O(1). Stacks are used in recursion, expression evaluation, and function calls. They are simple but powerful structures in algorithm design.

2. Applications of Stack (Expression Conversion)

Stacks are used in converting expressions like **infix to postfix or prefix**. In infix expressions, operators are placed between operands, while postfix places operators after operands. Stack helps manage operator precedence and parentheses. The algorithm scans the expression and uses stack to temporarily store operators. This conversion simplifies evaluation since postfix expressions don't need parentheses. The time complexity is O(n), making it efficient for compilers and calculators.

3. Queue ADT and Types

A queue follows **FIFO (First In First Out)** principle. Basic operations include **enqueue** (insert) and **dequeue** (remove). Types include **simple queue**, **circular queue**, and **priority queue**. Circular queues overcome space wastage, while priority queues process elements based on priority. Queues are used in scheduling, buffering, and simulations. Operations typically take O(1) time. They are essential for managing sequential processing tasks.

4. Circular Queue

A circular queue connects the last position to the first, forming a circle. It avoids unused space in linear queues. Two pointers, **front** and **rear**, manage operations. When rear reaches the end, it wraps around. Overflow occurs when the queue is full, and underflow when empty. Circular queues are efficient in memory utilization and used in applications like CPU scheduling and buffering. Operations like enqueue and dequeue have O(1) complexity.

5. Priority Queue

In a priority queue, elements are processed based on priority rather than order. Higher priority elements are removed first. It can be implemented using arrays, linked lists, or heaps. Operations include insertion and deletion, typically O(log n) using heaps. Priority queues are used in scheduling algorithms, shortest path algorithms like Dijkstra, and simulations. They are useful where importance matters more than order of arrival.

📘 Module 3: Linked Lists

1. Singly Linked List

A singly linked list consists of nodes where each node contains data and a pointer to the next node. It allows dynamic memory allocation. Operations include traversal, insertion, deletion, and searching. Unlike arrays, linked lists do not require contiguous memory. However, access time is slower (O(n)). They are useful when frequent insertion and deletion are required. It is one of the fundamental dynamic data structures.

2. Insertion and Deletion in Linked List

Insertion can occur at the beginning, middle, or end. It involves adjusting pointers. Deletion removes a node by linking the previous node to the next. Both operations take O(1) time if position is known. However, searching for position takes O(n). Proper pointer handling is crucial to avoid memory issues. These operations make linked lists flexible compared to arrays.

3. Doubly Linked List

A doubly linked list has two pointers: one to the next node and one to the previous node. It allows traversal in both directions. Operations like insertion and deletion are easier compared to singly linked lists. However, it requires more memory due to extra pointer. Time complexity remains O(1) for insertion/deletion (if position known). It is used in navigation systems and undo operations.

4. Circular Linked List

In circular linked list, the last node points back to the first node. There is no NULL pointer. It allows continuous traversal. Operations are similar to singly linked list but require careful handling of loops. It is useful in applications like round-robin scheduling. Time complexity remains O(n) for traversal and O(1) for insertion at known position.

5. Linked Representation of Stack and Queue

Stacks and queues can be implemented using linked lists. In stack, insertion and deletion occur at the head. In queue, insertion occurs at rear and deletion at front. This avoids overflow issues of arrays. Memory is allocated dynamically. Operations take O(1) time. Linked implementation is flexible and efficient for dynamic data sizes.

📘 Module 4: Searching, Sorting & Hashing

1. Linear and Binary Search

Linear search checks each element sequentially (O(n)). Binary search works on sorted arrays and divides the search space in half (O(log n)). Binary search is faster but requires sorted data. Linear search is simple but inefficient for large datasets. Choosing the method depends on data organization.

2. Bubble, Selection, and Insertion Sort

These are simple sorting algorithms. Bubble sort repeatedly swaps adjacent elements ($O(n^2)$). Selection sort selects the minimum element each pass ($O(n^2)$). Insertion sort builds sorted array step by step ($O(n^2)$, best case O(n)). They are easy to implement but inefficient for large data.

3. Quick Sort

Quick sort is a divide-and-conquer algorithm. It selects a pivot and partitions the array. Average time complexity is O(n log n), worst case $O(n^2)$. It is faster in practice and widely used. It requires less memory compared to merge sort.

4. Merge Sort and Heap Sort

Merge sort divides array into halves and merges them (O(n log n)). It is stable but requires extra space. Heap sort uses heap structure and has O(n log n) complexity. It is in-place but not stable. Both are efficient for large datasets.

5. Hashing

Hashing maps data to a fixed index using a hash function. It provides O(1) average time for search, insertion, and deletion. Collisions occur when two keys map to same index. Techniques like chaining and open addressing resolve collisions. Hashing is widely used in databases and caching systems.