Unit I
Basic Concepts:

Algorithm: Definition and characteristics

An algorithm is a step-by-step procedure used to solve a problem or complete a task. It is a finite sequence of clear instructions that takes some input, processes it, and produces the required output.

In simple words, an algorithm is a plan or method to solve a problem correctly and systematically.

Before writing a computer program, we first design an algorithm. It helps in understanding the logic of the problem and finding the correct solution.

Characteristics of an Algorithm

Every algorithm must satisfy the following important characteristics:
1. Input :
An algorithm must accept zero or more input values.
These inputs are the data given to solve the problem.
Example:
To add two numbers, the inputs are the two numbers.
2. Output:
An algorithm must produce at least one output.
The output is the result obtained after processing the input.
Example:
For addition, the output is the sum of the two numbers.
3. Definiteness:
Each step of the algorithm must be clear and unambiguous.
There should be no confusion in any instruction.
Every instruction must have only one meaning.
4. Finiteness:
An algorithm must terminate after a finite number of steps.
It should not run forever.
This ensures that the solution is obtained in a limited time.
5. Effectiveness:
Each step must be simple and executable.
The operations should be basic enough to be performed exactly in a finite amount of time.
6. Generality:
An algorithm should solve all valid instances of a problem, not just one specific case.
For example, an addition algorithm should work for any two numbers, not only for 2 and 3.
Example of an Algorithm
Problem: Add two numbers.
Algorithm:

Start
Read two numbers A and B
Compute Sum = A + B
Display Sum
Stop

This algorithm clearly shows input, processing, and output.

Importance of Algorithms in Data Structures: Algorithms are the foundation of computer programming and data structures.

They help in:
Writing correct programs
Improving efficiency
Reducing time and memory usage
Solving complex problems step by step
A good algorithm leads to an efficient program.

Q. What is an algorithm and explain its characteristics.

Complexity analysis

Space complexity refers to the amount of memory an algorithm requires during execution. It includes memory needed for input data, output data, and temporary variables. Space complexity depends on the size of input and the number of extra variables used. Some algorithms need additional memory, while others work with minimal space. For example, using an extra array to store results increases space complexity. Recursive algorithms usually require more space due to function call stacks. Efficient algorithms try to reduce unnecessary memory usage. Space complexity is important when working with limited memory systems. A good algorithm balances both time and space requirements. Space analysis helps in designing memory-efficient programs

Time complexity refers to the amount of time an algorithm takes to run as a function of the input size. It helps us compare algorithms based on their efficiency rather than actual execution time. Time complexity is usually measured by counting the number of basic operations performed by the algorithm. The execution time depends on factors such as input size and the type of operations used. Algorithms may have different time complexities for best, average, and worst cases. For example, searching an element in an array using linear search takes more time as the array size increases. If an array has *n* elements, linear search may take up to *n* comparisons. Time complexity helps programmers select better algorithms for large data sets. In data structures, efficient time complexity is very important for performance

---

Q. Explain about complexity analysis.
OR there can be 5 marks questions about each topic

Asymptotic Notations

Asymptotic analysis is a method used to study the performance of an algorithm as the size of input increases. It helps in understanding how the time or memory required by an algorithm grows for large inputs. Instead of measuring exact execution time, this analysis focuses on the general behavior of the algorithm. Small constants and machine-dependent factors are ignored so that algorithms can be compared fairly.

The main purpose of asymptotic analysis is to compare algorithms and select an efficient one before implementation. It helps programmers predict how an algorithm will behave when data size becomes very large. This is important because an algorithm that works

well for small input may perform poorly for large input. Using asymptotic analysis, such problems can be avoided in advance.

A good example to explain asymptotic analysis is **linear search and binary search**. In linear search, elements are checked one by one until the required element is found. If the list has 10 elements, it may take up to 10 comparisons. If the list has 1,000 elements, it may take up to 1,000 comparisons. As the input size increases, the number of operations increases proportionally, making linear search slow for large data.

In binary search, the list is first arranged in sorted order. The search space is divided into half in every step. Even if the list contains a large number of elements, only a small number of comparisons are needed. As the input size increases, the number of operations increases very slowly. Therefore, binary search is much faster than linear search for large data sets.

By analyzing the growth behavior of these two algorithms, asymptotic analysis clearly shows that binary search is more efficient than linear search for large inputs. This analysis helps in choosing the correct algorithm based on data size. Hence, asymptotic analysis plays a vital role in designing efficient algorithms in data structures.

Q. What are asymptotic notations and what are their purposes?

Introduction to Data structures

Definition: A **data structure** is a way of collecting and organizing data in memory so that various operations can be performed efficiently. It represents the knowledge of how data should be organized and stored in the computer. Data structures are not programming languages like C, C++ or Java, but they are a **set of algorithms** that can be implemented using any programming language. They help in rendering data elements in terms of some relationship for better organization and storage. A well-designed data structure reduces complexity and increases the efficiency of programs. Hence, data structures play an important role in effective data management in computer systems.

Types of Data structures: Data structures are classified based on how data elements are organized and related to each other in memory. The data structures are broadly classified into **primitive data structures** and **non-primitive (structured) data structures**.

Primitive data structures are the basic data types provided by the programming language and system. These data types store a single value at a time and are directly controlled by the system. Examples of primitive data structures include integer, character, float, double, and string. These data types are used to represent simple data values and form the building blocks for complex data structures.

Non-primitive data structures are those in which data elements are organized as a collection of related items. These data structures are used to store large amounts of data efficiently. Non-primitive data structures are further classified into **linear data structures** and **non-linear data structures**.

Linear data structures are those in which data elements are arranged in a sequential manner. In this type, each element is connected to its previous and next element except the first and last elements. Examples of linear data structures include arrays, linked lists, stacks, and queues. These data structures are easy to implement and are commonly used for sequential data processing.

Non-linear data structures are those in which data elements are not arranged in a sequential order. In this type, one element may be connected to multiple elements. Examples of non-linear data structures include trees and graphs. These data structures are used to represent hierarchical and network-based relationships.

Q. Define Data Structures and explain types of DS

Abstract Data Types (ADT): An **Abstract Data Type (ADT)** is a mathematical and logical model that defines a data type along with the operations that can be performed on it. It specifies **what data is stored** and **what operations are allowed**, without specifying how these operations are implemented. The ADT acts as a guideline for implementing a data type correctly in a programming language.

An ADT describes two important aspects:

1. how the data elements are related to each other, and
2. what operations can be performed on the data.

For example, the **INTEGER ADT** defines a set of whole numbers ranging from negative infinity to positive infinity and allows operations such as addition and subtraction. Programming languages like C, C++ and Java implement this ADT using data types such as int. The actual implementation details like memory size and range depend on the language and system.

Another example, Stack is one of the data structures, where data can be inserted and removed only from one end. The new element joining the set is kept on TOP and only the top most can be removed from it. In other words, only one element at TOP is accessible.

The informal ADT of STACK is as follows :

```
STACK: {1,2,3,4…n} , TOP
Condition : TOP=null

Insert operation: push()
        Precondition: top!=n
                insert (data)
                TOP - > data
Removal operation: pop()
        Precondition: TOP!=null
                remove(top)
                top - > current data
```

Q. Explain about ADT with an example

Introduction to Linked Lists

Representation of linked lists in Memory:

A linked list is a linear data structure used to store a collection of elements called nodes. Each node contains two parts: data and a link (pointer) to the next node in the list. Unlike arrays, linked lists do not store elements in contiguous memory locations. The order of elements is maintained using links between nodes. The first node of a linked list is called the head, and the last node points to NULL. Linked lists allow dynamic memory allocation, meaning memory is allocated as needed. Insertion and deletion operations are easier in linked lists compared to arrays. Linked lists are commonly used to implement stacks, queues, and other dynamic data structures.

In memory, a linked list is represented as a set of nodes stored at different memory locations. Each node consists of two fields: a data field that stores the actual value and a link field that stores the address of the next node. The address of the first node is stored in a pointer called HEAD. The link field of the last node contains NULL, indicating the end of the list. Nodes are not stored sequentially in memory; they may be scattered throughout memory. The linking of nodes using pointers maintains the logical order of data. Because of this representation, linked lists can grow or shrink dynamically during program execution. This memory structure makes linked lists flexible but slightly complex to manage.

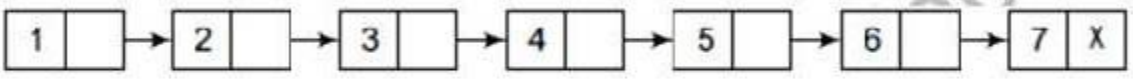

Figure 2.2 Simple linked list

Advantage: Easier to insert or delete data elements

Disadvantage: Slow search operation and requires more memory space

A linked list, in simple terms, is a linear collection of data elements. These data elements are called nodes. Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations. A linked list can be perceived as a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

Q. Explain Linked Lists with memory management.

Comparison between Linked List and Array:

An array stores its elements in contiguous memory locations, whereas a linked list stores its elements in non-contiguous memory locations. The size of an array is fixed at the time of declaration, while the size of a linked list is dynamic and can grow or shrink during program execution. In arrays, insertion and deletion operations are difficult and time-consuming because elements need to be shifted, whereas in linked lists insertion and deletion are easier as they only require updating pointers. Arrays allow direct access to elements using index values, but linked lists do not support direct access and require sequential traversal. Arrays may lead to memory wastage due to pre-allocation of space, whereas linked lists

allocate memory as needed, reducing wastage. Arrays are simpler to implement and manage, while linked lists are more complex due to the use of pointers. Thus, arrays are suitable for applications requiring fast access, whereas linked lists are preferred when frequent insertion and deletion operations are required.

Array	Linked List
Stores elements in **contiguous memory locations**	Stores elements in **non-contiguous memory locations**
Size is **fixed** at the time of declaration	Size is **dynamic** and can change at runtime
Insertion and deletion are **difficult and time-consuming**	Insertion and deletion are **easy and efficient**
Memory wastage may occur due to fixed size	Memory is allocated as needed, so less wastage
Direct access using index is possible	Direct access is not possible; traversal is required
Simple to implement	More complex due to use of pointers

Q. Compare between Arrays and Linked Lists.
