Unit-1: INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

1. **Algorithmic Notation: Format Conventions, Statement and Control Structures.**
2. **Time and Space Analysis.**
3. **Data types and Abstract data types.**
4. **Types of Data structures: Primitive, Non primitive - Linear and Nonlinear Data structures.**

- **Data structures** are methods for organizing and storing data in a computer so that it can be accessed and used efficiently. ***A data structure is a way to store data.***

  Basic Data Structures

  Arrays: Collections of elements stored in a contiguous block of memory, allowing for quick access based on an index.

  Linked Lists: A series of nodes, where each node contains data and a pointer to the next node in the sequence.

  Stacks: Data structures that follow a Last-In, First-Out (LIFO) principle, where the last element added is the first to be removed.

  Queues: Data structures that operate on a First-In, First-Out (FIFO) principle, like a waiting line.

  Trees: Hierarchical structures where data is organized into nodes with parent-child relationships.

  Graphs: A network of interconnected nodes (vertices) and edges that represent relationships between data points.

  Hash Tables: Structures that use a hash function to map keys to values, providing very fast average-case lookup times.

- **Algorithm designing is an important process of solving the problem.** The designing of an algorithm considers various aspects like space and time requirement for the execution of algorithm.

- **Algorithm**

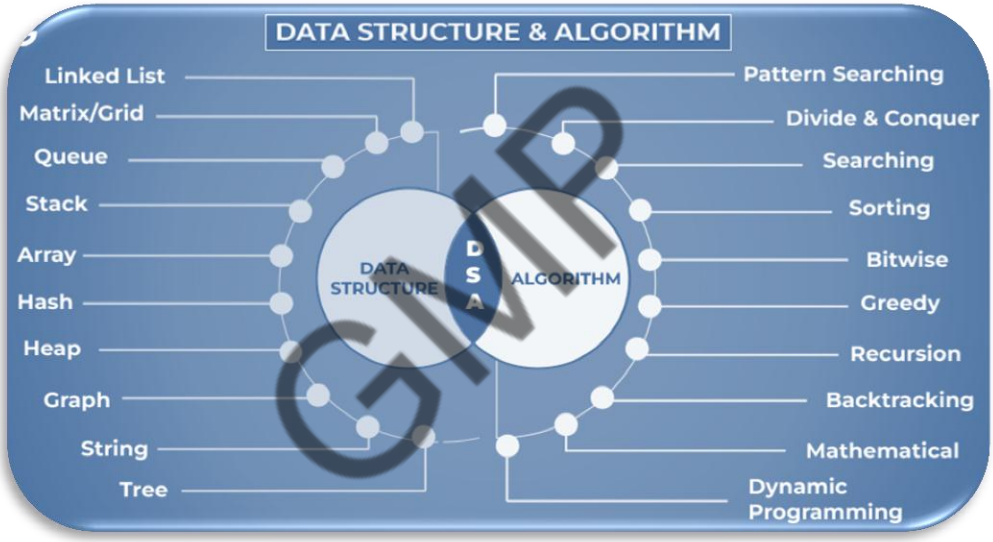
  The algorithm provides the way for solving the given problem in a systematic way. The term algorithm refers to the sequence of instructions that must be followed to solve a problem. Alternatively, an algorithm is a logical representation of the instructions which should be executed to perform a meaningful task.

  There are following characteristics of the algorithm:

- ✓ Each instruction should be unique and concise
- ✓ Each instruction should be relative in nature and should not repeat infinitely
- ✓ Result should be available to the user after the algorithm terminates.

Therefore, an algorithm is any defined computational procedure, along with a specified set of allowable inputs that produce some value or set values as output.

- ❖ ***Data Structures is about how data can be stored in different structures.***
- ❖ ***Algorithms is about how to solve different problems, often by searching through and manipulating data structures.***
- ❖ ***Theory about Data Structures and Algorithms (DSA) helps us to use large amounts of data to solve problems efficiently.***



1. **Algorithmic Notation: Format Conventions, Statement and Control Structures.**

Algorithmic Notation:

There are two basic approaches for designing the algorithm.

- ❖ **Top-Down approach:** In this approach we start from the main component of the program and decomposing it into sub problem or components. This process continues until all the sub modules do not solve. Top-down design method takes the form of **stepwise refinement**. In this, we start with the topmost module and incrementally add modules that it calls.
- ❖ **Bottom – Up approach:** In this approach of designing we start with the most basic or primitive components and proceeds to higher-level components. Bottom-up method works with **layer of abstraction**.

Here a simple example of the algorithm to demonstrate the various algorithmic notations.

Example:

Algorithm Greatest:

This algorithm finds the largest algebraic element of vector A which contains N elements and places the result in MAX. I is used to subscript A.

1. [Is the vector empty?]
   If N < 1
   Then print(„Empty Vector")
   Exit
2. [Initialize]
   MAX=A[1] [We assume initially that A[1] is the greatest element]
   I=2
3. [Examine all elements of vector]
   Do while I<=N
   3.1 [Change MAX if it is smaller than the next element]
   If MAX < A[I]
   Then MAX= A[I]
   3.2 [Prepare to examine next element in vector]
   I = I+1
4. [Finished]
   Exit

1.1 Format Convention of algorithm

Hence from this example we can consider the following format conventions for writing the algorithm. These conventions are so general that these may be used for writing any algorithm.

Name of Algorithm: Every algorithm is given an identify name

Introductory Comment: The algorithm name is followed by a brief description of the tasks the algorithm performs.

Steps: The algorithm is made up of a sequence of numbered steps, each beginning with a phrase enclosed in square brackets which gives an abbreviated description of that step.

Comments: Every step of the algorithm is explained for better understanding of it. These comments are expressed in brackets. Comments specify no action and are included only for clarity.

1.2 Statements and Control Structures.

It includes the various operators and looping methods those are required for logical and arithmetical operations.
Example: Assignment statement, If-statement, Case statement and looping methods.

1.3 Variable names: An entity that possesses a value and its name is chosen to reflect the meaning of the value it holds. For example The MAX is considered as the variable in our previous example of algorithm Greatest.

1.4 Data structures: various data structures including static and dynamic structures are used for the implementation of algorithm.

1.5 Arithmetic operations and expressions: The algorithm notation includes the standard binary and unary operators according to their standard mathematical order of precedence as follows:

Sr. No.	**Operation**	**Symbol**	**Order of Evaluation**
1.	Parentheses	()	Inner to outer
2.	Exponentiation, Unary plus, minus	∧, ++ , _ _	Right to left
3.	Multiplication, Division	*, /	Left to right
4.	Addition, Subtraction	=, -	Left to right

1.6 Relations and Relational Operators: There are standard relational operators (<, <=, >=, ≠, =, ==) are used with their usual meaning in the implementation of algorithm. A relation evaluates to a logical expression that is, it has one of two possible values, *True* or *False*.

1.7 Logical operations and Expressions: The algorithmic notation also includes the standard logical operators like NOT, OR & AND with their usual meaning. These may be used to connect relations to form compound relations whose only values are *True* or *False*. In order that logical expressions be clear, we consider that operators precedence is as follows:

Precedence	**Operator**
1	Parentheses
2	Arithmetic
3	Relational
4	Logical

1.8 Input and output: The algorithm notation must include the notation for input and output. The input is obtained and placed in a variable and output is obtained by getting the value from variable.

1.9 Subalgorithms-Functions and Procedures:

Functions: A function is used when we want a single value returned to the calling routine. Transfer of control and returning of the value are accomplished by "Return (value)".

Procedures: A procedure is similar to a function but there is no value returned explicitly. A procedure is also invoked differently, where there are parameters, a procedure returns its results through the parameters.

2. Time and Space Analysis.

After an algorithm has been designed its efficiency must be analysed. This involves determining whether the algorithm is economical in the use of computer resources, i.e. **CPU time** and **memory requirement.** The term used to refer to the memory required by an algorithm is **memory space** and the term used to refer to the computational time is the **execution time**. The importance of efficiency of an algorithm is the **correctness**. Thus, it always produces the correct result and **algorithm complexity** which considers both the difficulty of implementing an algorithm along with its efficiency.

Therefore the requirement for implementation of an algorithm with correctness considers many aspects. The fundamental question arises is that "How can we judge how useful a certain combination of data structures and algorithm is?" Of course the answer of this question depends that how can we evaluate the effort that arises from performing a computation using the certain combination of data structures and algorithms. There may be many algorithms devised for an application and we must analyse and validate the algorithms to judge the suitable one.

Hence this effort is measured normally with following two important factors those have the direct relationship with the performance of the algorithm:

- **Space complexity:** Memory space used i.e. **Space complexity.** The space complexity of an algorithm is the amount of memory it needs to run.

- **Time Complexity:** CPU time involves runtime or execution time for the program based on the algorithm i.e. **Time Complexity.** The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function it was written for.

- **Complexity of Algorithms**

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.
The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data.

- ***The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.***

Algorithms can be evaluated by the rate of growth of the time or space required to solve larger and larger instances of a problem.

Rate of Growth:
The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big–OH (O) 1
2. Big–OMEGA (□)
3. Big–THETA (□) and
4. Little–OH (o)

Big-OH O (Upper Bound)

f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal (<) that of g(n).

Big-OMEGA Ω (Lower Bound)

f(n) = Ω (g(n)) (pronounced omega), says that the growth rate of f(n) is greater than or equal to (>) that of g(n).

Big-THETA Θ (Same order)

f(n) = Θ (g(n)) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = □ (g(n)].

Little-OH (o)

T(n) = o(p(n)) (pronounced little oh), says that the growth rate of T(n) is less than the growth rate of p(n) [if T(n) = O(p(n)) and T(n) □ □(p(n))].

Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions.

The most common computing times are:
O(1), O(log2 n), O(n), O(n. log2 n), O(n2), O(n3), O(2n), n! and nn

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

n	**log2 n**	**n*log2n**	**n2**	**n3**	**2n**
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

2.1 Time Complexity

The time complexity of an algorithm is the amount of computer time that it needs to run to completion.

In order to compute the time complexity of an algorithm we consider only the frequency count of the important steps or instructions.

This efficiency is measured using the following two methods:

- Asymptotic Analysis
- Big-O analysis

It is very general that the actual time (wall-clock time) of a program is affected by:

- Size of the input
- Programming language
- Programming tricks
- Compiler
- CPU Speed

- Multiprogramming level

Hence instead of wall clock time for the program if we consider the pattern of the program‟s behaviour as the program size increases. This is called the **Asymptotic Analysis with asymptotic notations.**

Big- O Analysis

If *f(n)* represent the computing time of some algorithm and *g(n)* represents a known standard function like *n, n2, n log n etc* then to write: *f(n) is O g(n)* means that *f(n)* of *n* is equal to biggest order of function *g(n).*

This implies only when:

$$f(n) \leq C \log(n)$$

for all sufficiently large integers *n*, where *C* is the constant. Thus from the above statements we can say that the computing time of an algorithm is *O(g(n)),* we mean that its execution time is no more than a constant time *g(n),* n is the parameter which characterizes the input and / or outputs.

From the practical point of view, we get the Big-O notation for a function by:

1. Ignoring multiplicative constants (these are due to pesky difference in compiler, CPU, etc.)
2. Discarding the lower order terms (as *n* gets larger, the largest term has the biggest impact). Like;

- *8410 = O(1)*
- *100n3+nlogn+67n7+4n = O(n7)*

The **Big-O notation** helps to determine the time as well as space complexity of the algorithms. The Bog-O notation has been extremely useful to classify algorithms by their performances.

Example: consider the three simple algorithms with different number of sequences or steps:

Algorithm 1:

```
a=a+1
```

Algorithm 2:

```
For i= 1 to n do:
    a=a+1
end loop
```

Algorithm 3:

```
For i=1 to n do
```

```
   For j= 1to n do
      a=a+1
   end loop
end loop
```

Now we do the analysis of these three algorithms and can see their performance with Big-O notation.

In the algorithm 1 we may find that the execution statement *a=a+1* is the independent and is not constrained within any loop. Therefore, the number of times this will execute is 1. Thus, the frequency count of this algorithm is 1. Hence its **Time Complexity** is *O(1).*

In the algorithm 2, the execution statement *a=a+1* is inside the loop. The number of times it is executed is *n* as the loop runs for *n* times. The frequency count for this algorithm is *n.* Hence its **Time Complexity** is *O(n).*

In the algorithm 3, the frequency count for the execution statement *a=a+1* is *n2* as the inner loop runs *n* times, each time the outer loop runs, the outer loop also runs for n times. Hence its **Time Complexity** is *O(n2).*

Therefore during the analysis of algorithm we have the concern to determine the order of magnitude of an algorithm. Thus, we consider only those statements which may have the greatest frequency count.

Common Computing Times of Algorithm

The common computing times of algorithms in the order of their performance are as follows:

- *O(1)*: It means that the computing time of the algorithm is constant
- *O(log n)*: It means that the computing time of the algorithm is logarithmic
- *O(n)*: It means that the computing time of the algorithm is directly proportional to n. It is known as the linear time.
- *O(n log n)*: It means that the computing time of the algorithm is logarithmic
- *O(n2)*: It is known as the quadratic time
- *O(n3)*: It is known as the cubic time
- *O(2n)*: It is known as the exponential time. Generally the algorithm with exponential time has no practical use.

There are **different types of time complexities** which can analyse for an algorithm:

Best case time complexity: The best case complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size '*n'*. The running time of many algorithm varies not only for the input of different size but for the different inputs of the same size.

Average case time complexity: The time that an algorithm will require to execute input data of size '*n'* is known as average case time complexity. We can say that the value that is

obtained by averaging the running time of an algorithm for all possible inputs of size '*n'* can determine average-case time complexity.

Worst case time complexity: The worst time complexity of an algorithm is a measure of the maximum time that the algorithm will require for an input of size '*n'*. The worst case complexity is useful for a number of reasons. After knowing the worst case time complexity, we can guarantee that the algorithm will never take more than this time.

Hence the computation of exact time taken by the algorithm for its execution is very difficult. Thus, the work done by an algorithm for the execution of the input of size '*n'* defines the time analysis as function *f(n)* of the input data items.

2.2 Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion.

The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

3. Data types and Abstract data types

A **data type** is a concrete, specific kind of data (like an integer or string) with its own set of values and allowed operations, whereas an **Abstract Data Type (ADT)** is a mathematical model or specification of a data type, focusing on *what* operations can be performed and *what* behaviors are expected, but not how it is implemented. In essence, ADTs provide a blueprint for data types, defining the interface without exposing the underlying implementation details.

3.1 Data Types are a classification of data that specifies the type of data a variable can hold and the set of operations that can be performed on it.

Examples:

Primitive Types: Integers (int), floating-point numbers (float), characters (char), booleans (bool).

Composite Types: Strings, arrays.

3.2 Abstract Data Types (ADTs) are a mathematical and logical model that defines a data type in terms of its behavior and the operations that can be performed on it.

Examples:

Stack: A linear data structure where elements are added and removed from the top in a "last-in, first-out" (LIFO) manner.

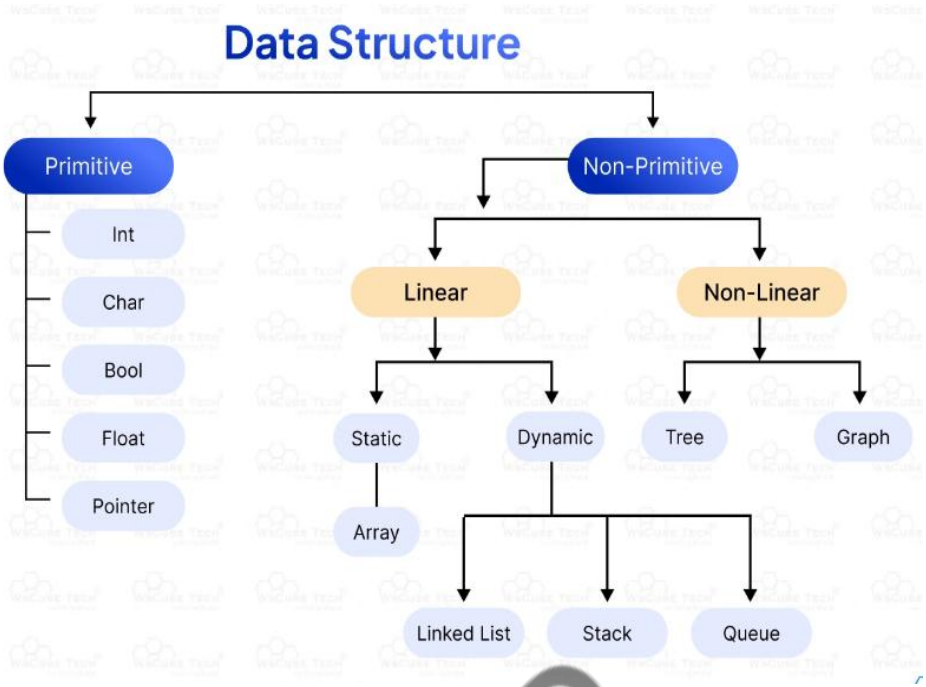
Queue: A linear data structure where elements are added at one end and removed from the other in a "first-in, first-out" (FIFO) manner.

List: A collection of items in a specific order.

4 Types of Data structures: Primitive, Non primitive - Linear and Nonlinear Data structures.

A data structure is a specialized and systematic format of storing, processing and organizing data in the memory of a computer. It is stored in a way that it can be accessed and used efficiently. It is essential for algorithms because of several data structure types. Various types of data structures allow storing data in the computer memory in such a way that the data can be accessed quickly for the required calculations.

Data structures types based on the type of operations they are required to perform are **primitive** and **non-primitive data structures.**



4.1 Primitive Data Structure

Primitive data structures are the basic building blocks of data manipulation. The primitive data structures are also known as built-in data structures as they are basic data structures in many high-level programming languages. They operate directly according to the machine's instructions.

The following are the **various types of data structures that are primitive**.

Int: Integers (int) are used to store whole numbers without any decimal points. They can represent a range of values depending on the system, allowing both positive and negative values.

Integers are commonly used in programming for counting, indexing arrays, and performing arithmetic operations.

Example: Counting the number of students in a class.

int studentCount = 30;

Char: Characters (char) store single characters from the Unicode (or ASCII in simpler systems) set, which includes not only alphabetical and numeric characters but also special symbols.

Each char requires one byte of memory, and they are used in programming to handle text processing.

Example: Storing the first letter of a name.

char initial = 'A';

Float: Floating-point numbers (float) are used to store decimal numbers and are particularly useful when precision is important, such as in scientific calculations, measurements, and financial computations.

Floating-point representation allows for a wide range of values but can introduce rounding errors.

Example: Storing a person's height.

```
float height = 5.9; // Height in feet
```

Bool: Booleans (bool) represent truth values and can hold one of two values: true or false. This data structure is fundamental in control structures for making decisions, performing loop checks, and managing binary states.

Example: Checking if a user is logged in.

```
bool isLoggedIn = true;
```

Pointer: Pointers store memory addresses of other variables. They are powerful because they allow for the manipulation of memory and the creation of complex data structures like **linked lists**, **trees**, and **graphs**.

Pointers are fundamental to understanding dynamic memory management in languages like C and C++.

Example: Pointing to an integer variable.

```
int x = 10;
int* ptr = &x; // Pointer to x
```


4.2 Non-Primitive Data Structure

Non-primitive data structures are complex data structures that use primitive data structures to form a structure of their own. This is why they are also known as derived types. Though in-built support is provided for these data structures by many programming languages, they are mostly user-defined. **Non-primitive data structures** can be further **classified** into **linear and non-linear data structures**.

4.2.1 Linear Data Structures

Linear data structures organize data in a sequential manner, meaning that elements are arranged in a specific order, and each element is connected to its previous and next element. This linear arrangement facilitates easy traversal, insertion, and deletion.

These data structures can be broadly divided into two categories based on how they manage memory: Static and Dynamic.

4.2.1.1 Static Data Structures

Static data structures have a fixed size, which means the amount of memory they require is determined at compile time before the program runs.

This fixed size cannot be changed once allocated, making static data structures more suited to situations where the maximum data is known beforehand.

Example: Array

Array: In a fundamental data structure

- Array stores a collection of data items in a linear sequence and contiguous memory location.
- Each data item is called an "element". Retrieval of data is easy with this data structure due to the storage of elements in contiguous memory locations.
- Each array element has a unique associated index value with which it can be accessed.
- The size of an array can be expressed mathematically as Length*Element size.
- Here length is the number of elements in the array and the element size is the memory size of the individual element.
- Arrays can be used to store data in tabular format, in financial analysis, in scientific computing, to represent graphs, CPU scheduling, in management systems in libraries, in online ticket booking systems, etc.

Stack: The stack data structure is:

- One of the types of data structure that uses LIFO or Last In First Out to store and retrieve data elements. It simply means that the element stored last in the stack will be retrieved first.
- Stack uses two operations for implementing the LIFO principle, push and pop.
- The push() method inserts the element in the stack and the pop() method retrieves one or more elements from the stack.
- Stacks are used to maintain the call logs in mobile phones, to keep track of visited pages on a browser, etc.

Queue: The queue data structure:

- Uses the FIFO or First In First Out principle.
- Here the element stored first or the oldest element in the structure is retrieved first. It is retrieved by the dequeue() method.
- It is used for handling website traffic, asynchronous transfer of data, switching to multiple applications on an operating system, etc.

Linked List: The linked list data structure:

- Uses data memory allocation to store its elements in different locations. Despite being in different locations, these elements are arranged in a sequence and linked with one another.
- The element contains a data item and a link or reference to the subsequent item on the same list. This is why they are most preferred when one has to handle dynamic data elements.
- There are four major types of lists in data structures, singly linked lists, doubly linked lists, circular linked lists, and circular doubly linked lists.
- Linked lists can be used to implement stacks, graphs, etc., to perform arithmetic operations on long integers, for representation of sparse metrics, in memory management.

4.2.1.2 Dynamic Data Structures

Dynamic data structures can grow and shrink at runtime, making them more flexible and suitable for applications where the amount of data isn't known in advance or can change frequently.

They generally use pointers to manage elements, which adds a level of complexity but provides greater flexibility in memory management.

Example: Linked List, Stack, and **Queue**

4.2.2 Non-Linear Data Structures

Non-linear data structures the elements are stored in a non-linear or non-sequential manner. It is complicated since values are arranged at multiple levels(hierarchy), that is, each value is connected to one or more values. The following are the types of non-linear data structures.

- **Graph:** The graph is a node-based type of DS that has a list of other nodes that are linked in the graph. It has two components, vertices, and edges. The edges are used to connect vertices. This data structure is used for network representation. The graph data structure is used to represent the flow of computation, in modeling graphs, in Google Maps, to study molecules of physics and chemistry, etc.
- **Tree:** A subtype of the graph that represents its elements in a hierarchical order. A tree uses a node-like structure to make a hierarchical form and each node here represents a value. The root node is the uppermost node of the tree and the leaf node is the node at the bottom of the tree. This data structure imposes a rule that nodes of a tree do not create loops in the data structure. A tree is used for indexing databases, scanning, parsing, and generating code in compiler design, in routers in computer networks, social networking sites, etc.

Difference between Linear and Non-linear Data Structures:

S.NO	Linear Data Structure	Non-linear Data Structure
1.	In a linear data structure, data elements are arranged in a linear order where each and every element is attached to its	In a non-linear data structure, data elements are attached in hierarchically manner.

	previous and next adjacent.	
2.	In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
3.	Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
4.	In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
5.	In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
6.	Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
7.	Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.
8.	Linear data structures are useful for simple data storage and manipulation.	Non-linear data structures are useful for representing complex relationships and data hierarchies, such as in social networks, file systems, or computer networks.
9.	Performance is usually good for simple operations like adding or removing at the ends, but slower for operations like searching or removing elements in the middle.	Performance can vary depending on the structure and the operation, but can be optimized for specific operations.

Bibliography

Horowitz, E., S. Sahni: "Fundamental of computer Algorithms", Computer Science Press, 1978

J. P. Tremblay, P. G. Sorenson "An Introduction to Data Structures with Applications", Tata McGraw-Hill, 1984

M. Allen Weiss: "Data structures and Problem solving using C++", Pearson Addison Wesley, 2003

Ulrich Klehmet: "Introduction to Data Structures and Algorithms", URL: http://www7 . Informatik.uni-erlangen.de/~klehmet/teaching/SoSem/dsa

Markus Blaner: "Introduction to Algorithms and Data Structures", Saarland University, 2011

V. Abo. Hopcropft, Ullaman, "data Structure and Algorithms", I.P.E.

Seymour Lipschutz, "Data Structure", Schaum‟s outline Series.