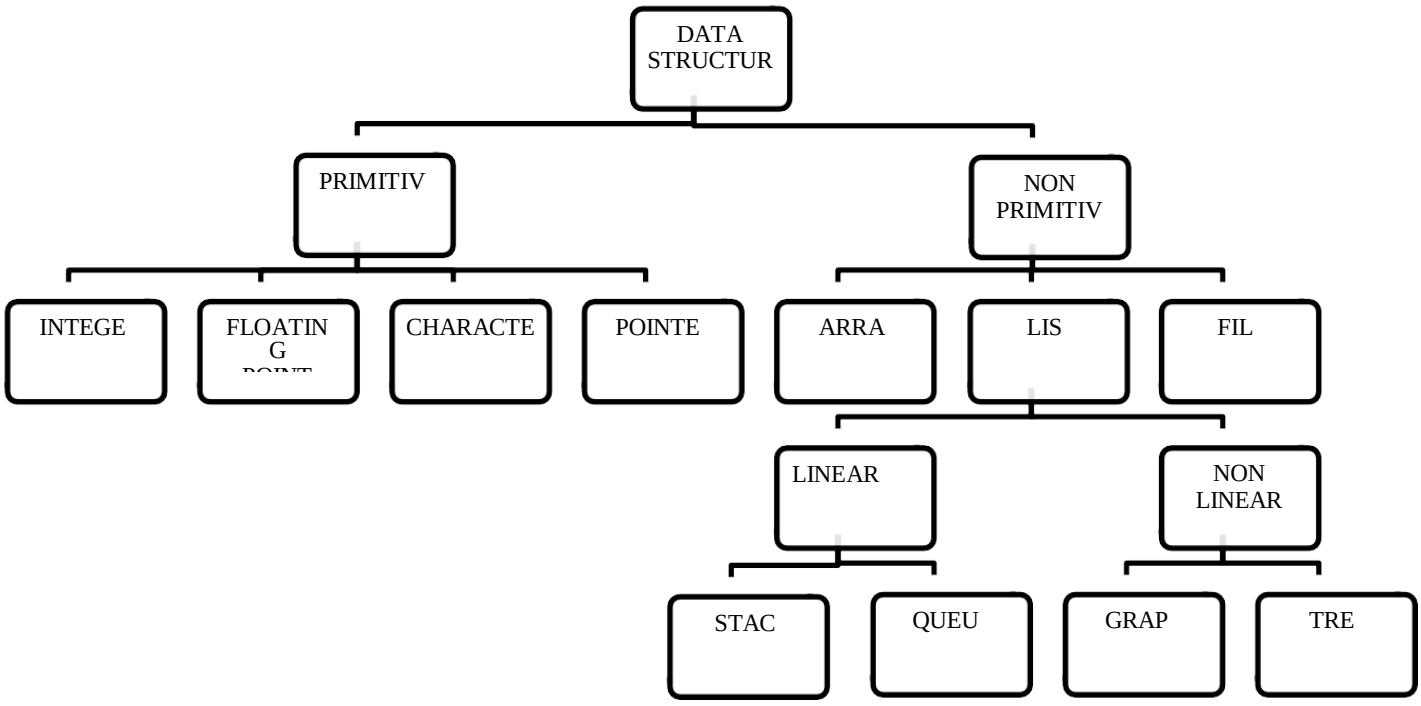
UNIT – I : Data Structures and Algorithms

What is Data Structure?

- **Data structure** is a representation of the logical relationship existing between individual elements ofdata.
- Data Structure is a way of organizing all data items that considers not only the elements stored but alsotheir relationship to each other.
- We can also define data structure as a mathematical or logical model of a particular organization ofdata items.
- The representation of particular data structure in the main memory of a computer is called as **storagestructure.**
- The storage structure representation in auxiliary memory is called as **file structure**.
- It is defined as the way of storing and manipulating data in organized form so that it can be usedefficiently.
- Data Structure mainly specifies the following four things
  - Organization of Data
  - Accessing methods
  - Degree of associativity
  - Processing alternatives for information
- Algorithm + Data Structure = Program
- Data structure study covers the following points
  - Amount of memory require to store.
  - Amount of time require to process.
  - Representation of data in memory.
  - Operations performed on that data.

Classification of Data Structure



Data Structures are normally classified into two broad categories

1. Primitive Data Structure
2. Non-primitive data Structure

Data types

A particular kind of data item, as defined by the values it can take, the programming language used, orthe operations that can be performed on it.

Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.
  - Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
  - Float: It is a data type which use for storing fractional numbers.
  - Character: It is a data type which is used for character values.

Pointer: A variable that holds memory address of another variable are called pointer.

Non primitive Data Type

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneousdata items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure
  - **Array:** An array is a fixed-size sequenced collection of elements of the same data type.
  - **List:** An ordered set containing variable number of elements is called as Lists.
  - **File:** A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Linear data structures

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
  - Static memory allocation
  - Dynamic memory allocation
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sortingand Merging.
- Examples of Linear Data Structure are Stack and Queue.
- Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only.
  - The insertion operation is referred to as ‘PUSH’ and deletion operation is referred to as ‘POP’ operation.
  - Stack is also called as Last in First out (LIFO) data structure.
- Queue: The data structure which permits the insertion at one end and Deletion at another end, known asQueue.

- End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
- Queue is also called as First in First out (FIFO) data structure.

Nonlinear data structures

- Nonlinear data structures are those data structure in which data items are not arranged in a sequence.
- Examples of Non-linear Data Structure are Tree and Graph.

- **Tree:** A tree can be defined as finite set of data items (nodes) in which data items are arranged in branchesand sub branches according to requirement.
  - Trees represent the hierarchical relationship between various elements.
  - Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting tocircle.

- **Graph:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.
  - A tree can be viewed as restricted graph.

Difference between Linear and Non Linear Data Structure

Linear Data Structure	**Non-Linear Data Structure**
Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Eg. Array, Stacks, linked list, queue.	Eg. tree, graph.
Implementation is easy.	Implementation is difficult.

ALGORITHMS:

INTRODUCTION:

- The concept of an algorithm is fundamental to computer science. Algorithms exist for many common problems, and designing efficient algorithms plays a crucial role in developing large-scale computer applications.
- Algorithm is a step-by-step procedure for solving a computational problem, which defines a set of instructions to be executed in a certain order to get the desired output.
- A program is also a step by step procedure for solving a problem.
- At design time we write the steps for solving the given problem, this is not program but algorithm (simple English like statements without using any syntax. This will not be written on a computer but on a paper.)
- Algorithms are generally created independent of any language, to write algorithms, any natural language can be used. Ex: English.
- An algorithm can be implemented in more than one programming language.

Algorithm	**Program**
1. Written at Design Time. 2. Any person who is having the domain knowledge can write an algorithm. 3. Any language can be used with optional mathematical notations. 4. Software & Hardware independent. 5. After writing the algorithm, it'll be analyzed for best results and measure it according to time and space complexities.	1. Written at Implementation. 2. Programmers will write programs. 3. Only programming languages has to be used. Ex: C, C++, JAVA. 4. Software & Hardware dependent. Ex: Operating system and processor. 5. After coding a program it'll be tested.

Characteristics / Specifications of Algorithms:

- **Input:** An algorithm has zero or more inputs, taken from a specified set of objects.
- **Output:** An algorithm has one or more outputs, which have a specified relation to the inputs.
- **Definiteness:** Each step must be precisely defined; Each instruction is clear and unambiguous.
- **Finiteness:** The algorithm must always terminate after a finite number of steps.
- **Effectiveness:** All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.

How to write an algorithm:

- While writing algorithm any intelligible or understandable statements can be used, as long as it is understandable by the concerned parties.
- No need to follow any programming language syntax or notations specifically in the algorithms. There is no fixed syntax for writing the algorithms.
- Any language can be used to write the algorithms with optional mathematical notations. Ex: English with

mathematical notations.

- An algorithm can be designed to get a solution of a given problem. A problem can be solved in more than one ways. Hence, many solution algorithms can be derived for a given problem.

Example:

Problem − Design an algorithm to add two numbers and display the result.	
Step 1 – START Step 2 − declare three integers a, b & c	Step 1 – START Step 2 − get values of a & b
Step 3 − define values of a & b Step 4 − add values of a & b Step 5 − store output of step 4 to c Step 6 − print c Step 7 − STOP	Step 3 − c ← a + b Step 4 − display c Step 5 – STOP

Advantages of Algorithms:

- It is easy to understand.
- An algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Understanding complex logic through algorithms can be very difficult.
- Branching and Looping statements are difficult to show in Algorithms**(imp)**.

PERFORMANCE ANALYSIS:

- The most important attribute of a program is, its correctness.
- If a program does not perform the intended task, then it is of no use.
- However, a correct program may also be of little use if it takes more memory than the available memory in a computer where it runs, as well as if it takes more time than the use is willing to wait.
- The word program performance is used to refer the memory and time requirements of a program.
- In straight approach, program performance mean the amount of computer memory and time needed to run a program.
- There are two approaches to determine the performance of a program. One is analytical, where analytical methods are used to analyze the performance. Second is experimental, where performance is measured by conducting experiments.

Space Complexity:

- The space complexity of a program is the amount of memory needed to the program for completion.
- The following are the reasons for the importance of space complexity:
    1. If the program is for multi user computer system, then the amount of memory needed for the program must be specified.
    2. It is always advantageous to know whether or not sufficient memory is available to run the program.
    3. A problem may have several solutions with different space requirements. The users will always prefer the solution which consumes less space in the computer. The solution which consumes less memory leaves the user with more memory for other tasks.

Components of Space Complexity:

The space required by the program has the following components:

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

- The amount of instruction space needed by the program depends on the following factors:
  1. The compiler used to compile the program into machine code.
  2. The compiler options in effect at the time of compilation.
  3. The target computer.

Data Space: The space needed to store all constants and variable values.

- This has the following to components.
  1. Space needed by constants and simple variables.
  2. Space needed by dynamically allocated objects such as arrays and class objects.

Environment stack space: This is used to save information needed to stop and resume execution of partially completed functions.

Each time a function is invoked the following data are saved on environment stack:

1. The return address.
2. The values of all local variables and formal parameters in the function being invoked (in case of recursive function).

Time complexity:

- The time complexity of a program is the amount of computer time it needs to run the program.
- The following are the reasons for the importance of space complexity:
  1. Computer systems require the user to provide a valid time limit for the execution of a program, so as to avoid the program from entering into infinite loops.
  2. The program must need to provide a satisfactory real-time response.
     For instance: A text editor that takes a minute to move the cursor for page down and page up will not be acceptable.

Components of Time Complexity:

- The time complexity of a program depends on all the factors that the space complexity depends.
  1. A program will run faster on a computer capable of executing 10^9 instructions per second than on a computer 10^7 instructions per second.
  2. A solution with less number of instructions always require less execution time. Small problem solutions take less time compared to lagers problem solution.
  3. Some compilers will take less time than others to generate the machine code.

- The time taken by a program P is the sum of the compile time and the runtime. As a compiled program can be executed several times without recompilation. Only the runtime will be considered. This runtime is denoted by t_p.
- As the factors of t_p are not known, when the program is created. The t_p can only be estimated.
- t_p can be determined as the sum of number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on that the code for P performs.
  $t_p(n) = c_aADD(n) + c_sSUB(n) + c_mMUL(n) + c_dDIV(n) + \ldots$

  Where ca, cs, cm, and cd are the time needed for addition, subtraction, multiplication and division. And ADD, SUB, MUL and DIV are functions whose value is the number of addition, subtraction, multiplication and divisions performed in code P.
  Two more approaches for estimating runtime are:
  1. Operation Count: Identifying the key operations and determining the number of times these operations are performed.
  2. Step Count: Determining the total number of steps executed by the program.

  Operation Counts: To estimate the time complexity of a program or a function, select one or more

operations such as add, multiply and compare and determine how many times these operations are performed. The success of this method depends on the ability of the programmer in identifying the operations that contributes to the time complexity. Operation count is a function of instance characteristics (size of input and output, system performance).

Best, Worst and Average Operation Counts:

- In the above example to find the desired element in the list, a compare operation is performed between the desired value and the elements in the list.
- Search process will begin at the start of the list and continue until it finds the desired element.
- The function returns the index value of element where it found the desired element.

Best Operation Count:

- Finding the element at the first location with n-elements in the list will lead in performing only one compare operation.
- This will be considered as the best case scenario for our example. And also need only one compare operation to lead to the success of the program.

Worst Operation Count:

- Finding the element at the last location with n-elements in the list, will lead in performing n-comparisons within the list.
- This will be considered as the worst case scenario for our example. And also need n-compares before finding the desired value.

Average Operation Count:

- Finding average operation count deals with performing the operation for some k-number of times, adding the total number of compare operations resulted, dividing with k will give the average case.
- The average operation count is often quite difficult to determine. As a result operation count will deal with only best and worst cases and average case is ignored.

How to Analyze an Algorithm:

The criteria for analyzing an algorithm:

1. **Time:**
   - The algorithm must be time efficient means it must be faster.
   - In the analysis process the time taken by the algorithm is measured, whether the algorithm is lengthy and time consuming or it is smaller and faster.
   - The time consumed by the algorithm will be in the form of a function (Time function) and not in the form of a watch/clock time.

2. **Space:**
   - The algorithm will be converted into a program and going to be executed on a computer.
   - One has to know how much memory space it is going to consume during its execution.

3. **Network Consumption:**
   - The algorithm, if it is doing network transactions sending and receiving the data to and from the network, then the consumption of Network resources need be considered as a criterion of analysis.

4. **Power Consumption:**
   - Now a day's most of the programs are coded not only for desktop computers but also for mobile phones, tablets and laptops, where the power consumption is an important issue.
   - If the algorithm falls under this category then power consumption is also an important criterion of analysis.

5. **CPU registers:**
   - The CPU will also have memory known as registers.
   - The amount of CPU register consumption by the algorithm need to be considered as a criterion while designing the algorithm.

Measuring Time and Space Complexities:

Time Analysis:
- Every simple statement (not nested) in an algorithm will take one unit of time.
- If an algorithm is calling or using another algorithm in it, then the called algorithm must have to be analyzed.

Space Analysis:
- Every simple variable in the algorithm will consume one word of memory.
- If it is a complex variable like an array then the total number of elements in that array need to considered.

Examples:

Ex1:

```
Algorithm swap(a,b)        Time                          Space
{                          ----------                    ----------
        temp <- a;  ----------  1          a ----------1
        a <- b; ----------  1              b ----------1
        b <- temp; ---------- 1        temp ----------1
}                  ----------             ----------
                        f(n) = 3 (a constant value)      s(n) = 3 words
                              O(1)                       O(1)
```

Recursive Algorithms:
- A function is a set of instructions that perform a logical operation, perhaps a very complex and long operation, can be grouped together as a function.
- Recursion is a process in which functions can call themselves (direct recursion) before they are done or they may call other functions that again invoke the calling function (indirect recursion).
- A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.
- In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

Example:	
Step 1: start Step 2: read number n Step 3: call factorial(n) Step 4: print factorial f Step 5: stop	factorial(n) Step 1: if n==1 then return 1 Step 2: else f=n*factorial(n-1) Step 3: return f