DATA STRUCTURE

- In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure.
- Data structure is the structural representation of logical relationship between data elements.
- A data structure is a special way of organizing and storing data in a computer so that it can be used efficiently.
- This means that a data structure organizes data items based on the relationship between the data elements.
- **Example:**

  A house can be identified by the house name, location, number of floors and so on. These structured set of variables depend on each other to identify the exact house.
- *Array, Linked List* , *Stack, Queue, Tree, Graph* etc are all data structures that stores the data in a special way so that we can access and use the data efficiently efficiently.

Need for Data Structure:

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

Need for Data Structure:

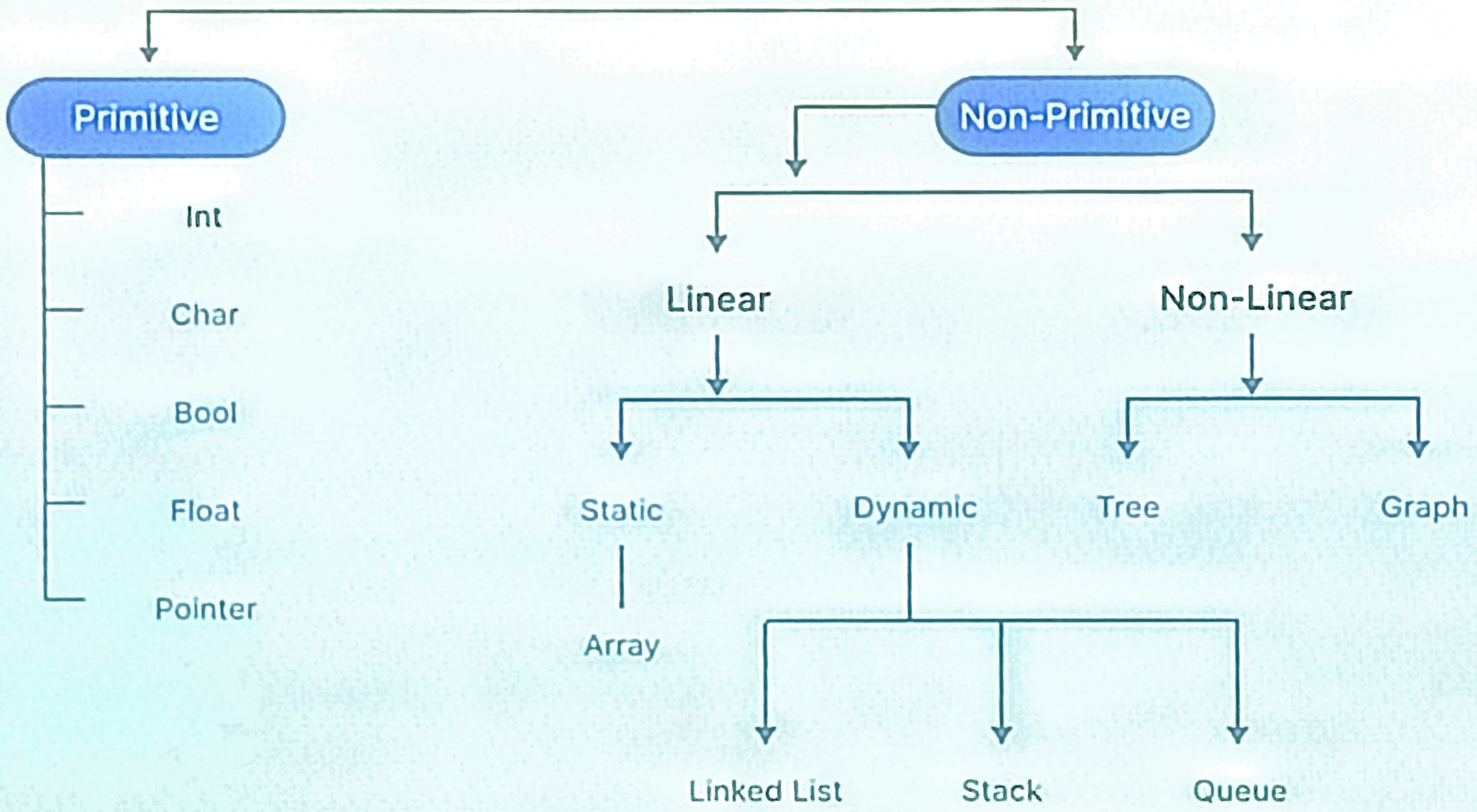
- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

Advantages of Data Structures

- **Efficient Memory use:** With efficient use of data structure memory usage can be optimized, for e.g we can use linked list vs arrays when we are not sure about the size of data. When there is no more use of memory, it can be released.

- **Reusability:** Data structures can be reused, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Classification of Data Structures



Primitive Data Structure

- Primitive data structures consist of the numbers and the characters which are built in programs. These can be manipulated or operated directly by the machine level instructions.
- Basic data types such as integer, real, character, and Boolean come under primitive data structures.
- These data types are also known as simple data types because they consist of characters that cannot be divided.

Non-primitive Data Structure

- Non-primitive data structures are those that are derived from primitive data structures.
- These data structures cannot be operated or manipulated directly by the machine level instructions. They focus on formation of a set of data elements that is either homogeneous (same data type) or heterogeneous (different data type).
- These are further divided into linear and non-linear data structure based on the structure and arrangement of data.

Linear Data Structure

- A data structure that maintains a linear relationship among its elements is called a linear data structure.
- Here, the data is arranged in a linear fashion.
- But in the memory, the arrangement may not be sequential. Ex: Arrays, linked lists, stacks, queues.

Non-linear Data Structure

- Non-linear data structure is a kind of data structure in which data elements are not arranged in a sequential order.
- There is a hierarchical relationship between individual data items.
- Here, the insertion and deletion of data is not possible in a linear fashion. Trees and graphs are examples of non-linear data structures.

ALGORITHM

Def^n: Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Characteristics of an Algorithm

i. Clear and Unambiguous

ii. Well-Defined Inputs

iii. Well-Defined Outputs

iv. Finite-ness

v. Feasible

vi. Language Independent

Advantages and Disadvantages of Algorithm

Advantages:

- It is easy to understand.
- 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 :

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.

Different approach to design an algorithm

Top-Down Approach:

- A top-down approach starts with identifying major components of system or program decomposing them into their lower level components & iterating until desired level of module complexity is achieved .
- In this we start with topmost module & incrementally add modules that is calls.

Bottom-Up Approach:

- A bottom-up approach starts with designing most basic or primitive component & proceeds to higher level components.
- Starting from very bottom, operations that provide layer of abstraction are implemented

Ways of writing an algorithm algorithms

Generally, there are 3 ways of writing an algorithm.

1. Step by step English-Like Algorithm

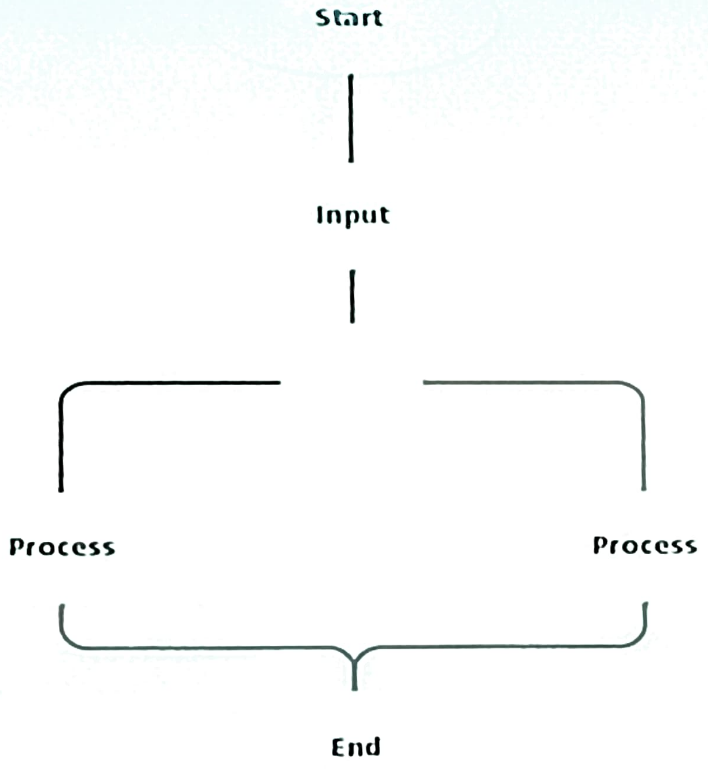
- An algorithm can be written in simple English but this method also has some demerits.
- Natural language can be ambiguous and therefore lack the characteristic of being definite.
- English language-like algorithms are not considered good for most of the tasks.

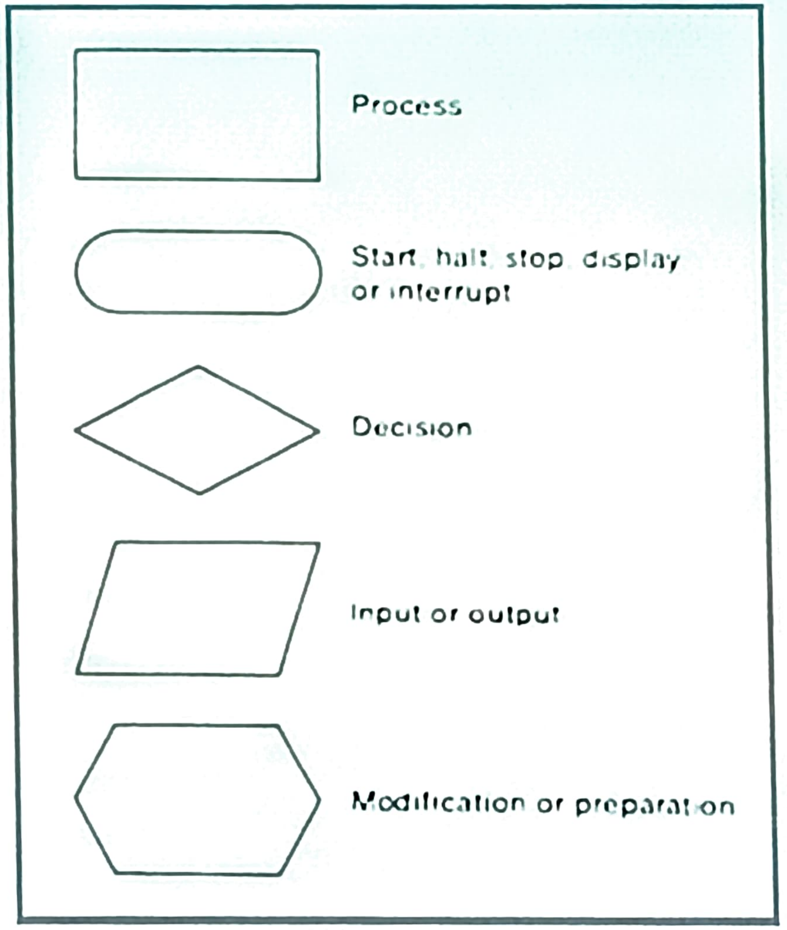
Example: *Write algorithm to find factorial of a number*

1. Start with an integer *n*.
2. Initialize a variable *fact = 1* to store the factorial result.
3. Set *i = 1* for iteration.
4. Multiplying *fact* by *i* in each iteration i.e. *fact <- fact*i*.
5. Update *i* value i.e. *i++*.
6. Repeat step 4-5 until *i* less than equal to *n*.
7. Return fact as the result.

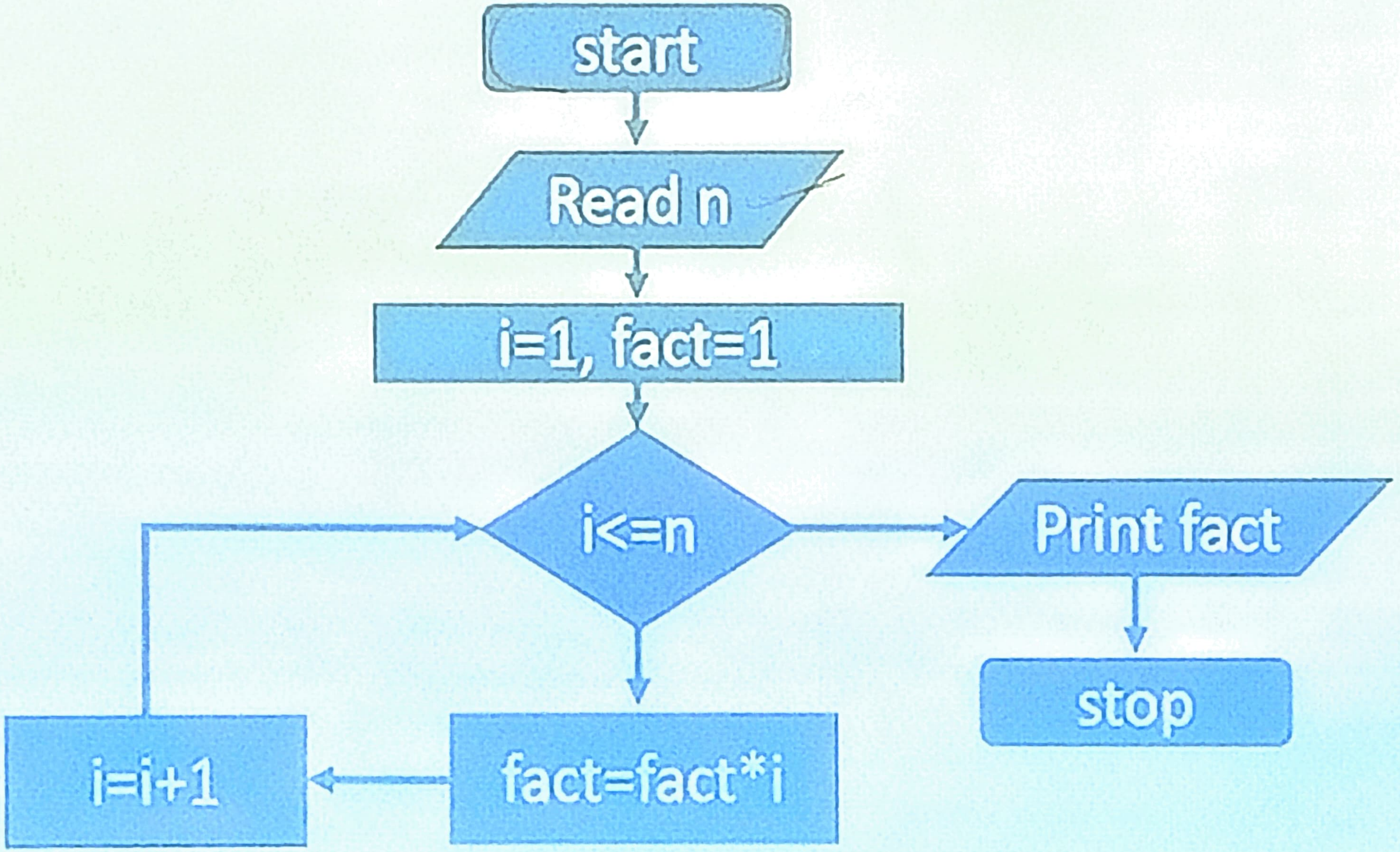
2. Flowchart

- Flowcharts pictorially depict a process. They are easy to understand and are commonly used in the case of simple problems.





Process
Start, halt, stop, display or interrupt
Decision
Input or output
Modification or preparation


start
Read n
i=1, fact=1
i<=n
Print fact
stop
fact=fact*i
i=i+1

3. Pseudocode

- The pseudocode has an advantage of being easily converted into any programming language.
- This way of writing algorithm is most acceptable and most widely used.
- In order to write a pseudocode, one must be familiar with the conventions of writing it.

Example: Write algorithm to find factorial of a number

```
1.  Start with an integer n.
2.  Initialize a variable fact = 1 to store the factorial result.
3.  Set i = 1 for iteration.
4.  For i = 1 to n
        fact = fact * n
    End For
5.  Display fact
```

Algorithmic Complexity

- Algorithmic complexity is concerned about how fast or slow particular algorithm performs.
- We define complexity as a numerical function *T(n)* - time versus the input size *n*.
- We want to define time taken by an algorithm without depending on the implementation details.
- But you agree that *T(n)* does depend on the implementation! A given algorithm will take different amounts of time on the same inputs depending on such factors as: processor speed; instruction set, disk speed, brand of compiler and etc.
- The complexity of an algorithm can be divided into two types. The time complexity and the space complexity.

Time Complexity:

- The time complexity of an algorithm is a metric that may be used to quantify the amount of time required for an algorithm to run as just a function of the length of the input.

Space complexity:

- Space complexity refers to the amount of storage required by the program/algorithm including the amount of space required by each variable.

Space Complexity = $\underline{\text{memory required by variable}}$ + extra space

Asymptotic Notation

- Asymptotic Notation in Data Structure is a way to describe how the time or space needed by an algorithm grows as the size of the input grows.
- It helps us understand how efficient an algorithm is when we have very large inputs.
- There are mainly three asymptotic notations:
  1. Big-O notation
  2. Omega notation
  3. Theta notation