Introduction to Data Structures and Analysis of Algorithm

By Asst Prof Christopher Uz

Course: DSAA

Pillai College of Engineering

Bad programmers worry about the code. Good ones worry about data structure and their relationships.

-Linus Torvalds

What is Data Structure?

- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently
- Widely used in almost every aspect of Computer Science i.e., Operating System, Compiler Design, Artificial intelligence, Graphics and many more
- 2 types of data structure namely Primitive & Non-Primitive DS

Primitive Data Structure

- It is a type of data structure that stores the data of only one type
- Examples of primitive data structures are
  - Integer
  - Float
  - Char
  - Double
  - Pointer

Non-Primitive Data Structures

- It is a type of data structure which is a user-defined and it stores the data of different types in a single entity
- It is categorized into two parts such as linear data structure and non-linear data structure

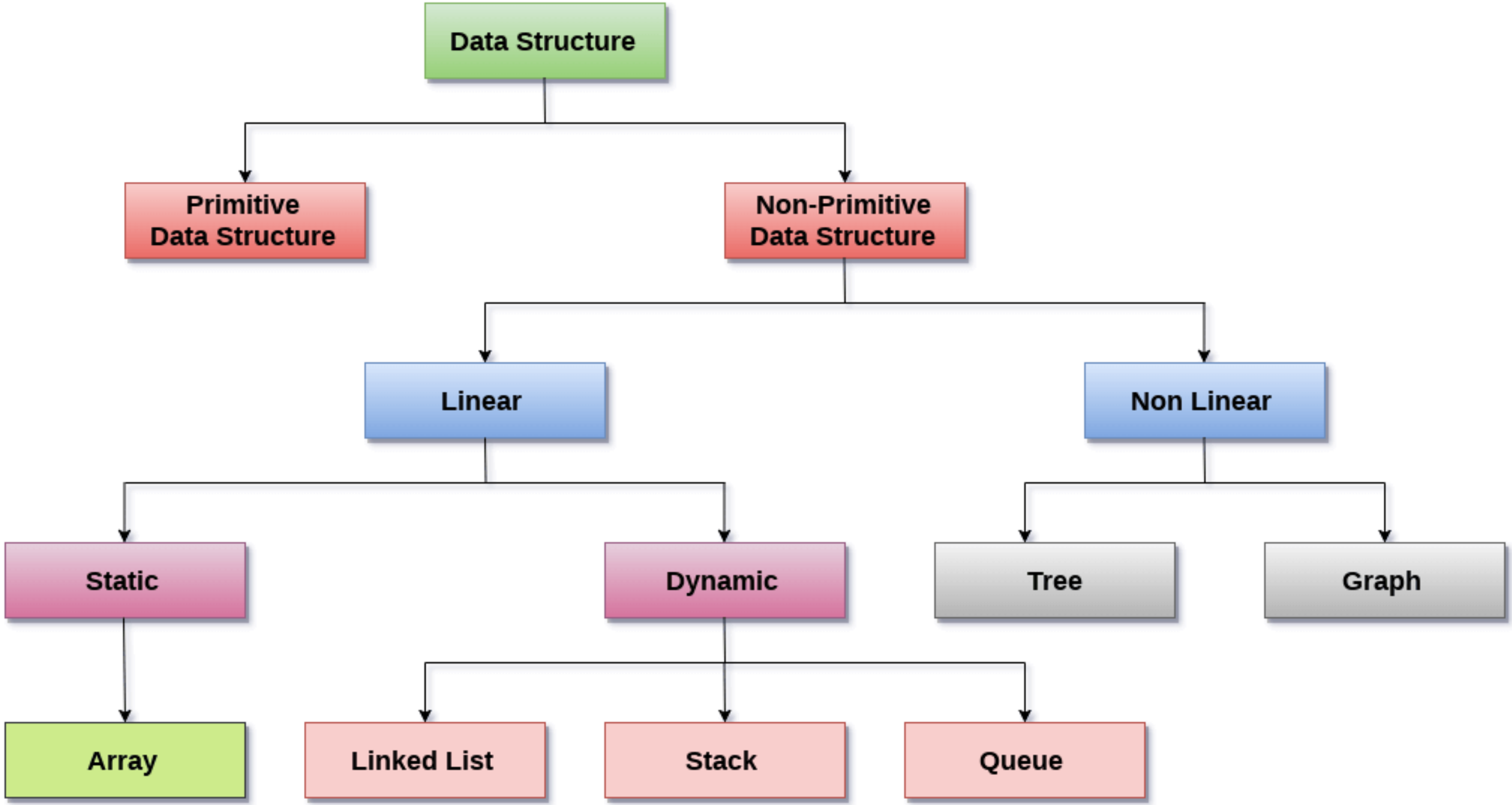
Non-Primitive Data Structures…

Linear DS

- Data elements are arranged in a sequential order
- Single level is involved
- Data elements can be traversed in a single run only
- E.g.: Array, Stack, Queue, Linked list
- Application: MS Word(Stack), Bitmap Images(Array)

Non Linear DS

- Data elements are attached in hierarchically manner
- Multiple levels are involved
- Data elements can’t be traversed in a single run only
- E.g.: Trees and Graphs
- Application: Social network friends(Graphs)


Data Structure
Primitive Data Structure
Non-Primitive Data Structure
Linear
Non Linear
Static
Dynamic
Tree
Graph
Array
Linked List
Stack
Queue

Static Data Structures

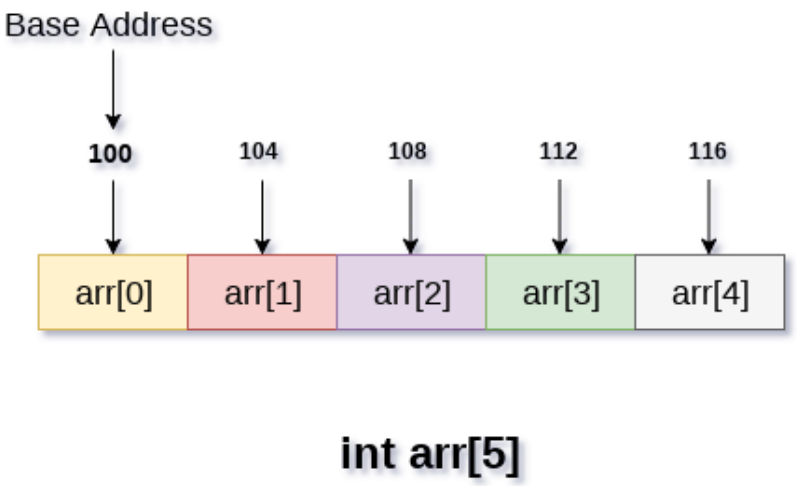
- It is a collection of data in memory that is fixed in size
- Max size need to known in advance as memory cannot be reallocated at a later point
- E.g. Arrays
- Advantages: Very efficient in terms of being able to access elements directly (index)
- Disadvantages: Can be inefficient in terms of memory usage

Note: While size of array is fixed, we can still change the assigned value of elements

Arrays

- Collection of similar data types
- Array is the simplest DS
- Data can be randomly accessed by using its index number
- Stored in consecutive memory locations

```
int arr[5] = {56,78,88,76,56);
```



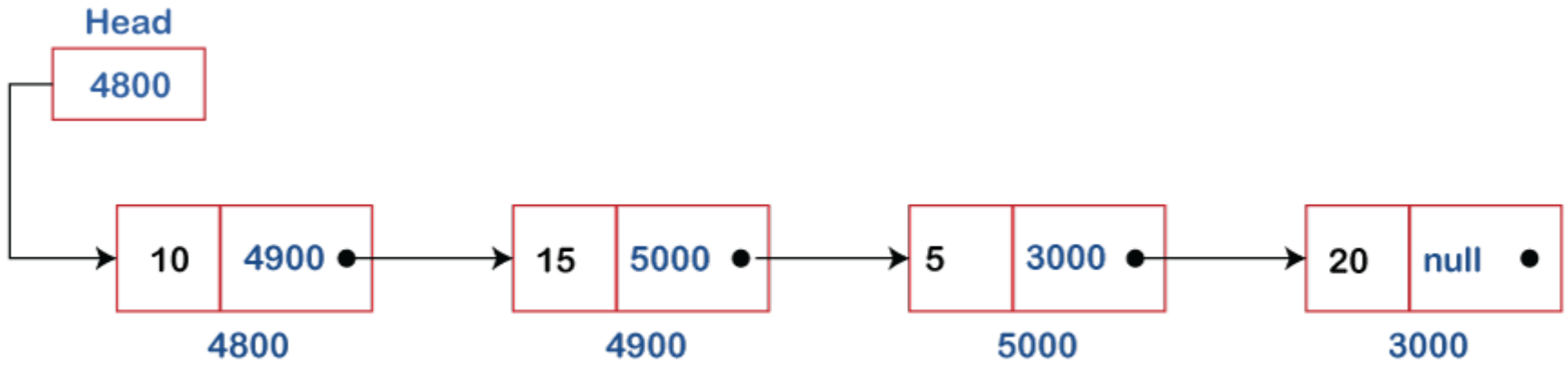
Dynamic Data structures

- The memory capacity of a dynamic data structure is not fixed
- The number of data items that it can hold is constrained only by the overall memory allocation to the program
- E.g. Linked List
- Advantages: Only uses the space needed at any time and makes efficient use of memory
- Disadvantages: Elements cannot be readily accessed directly as they are accessed by memory location rather than index

Note: A linked list only allows serial access therefore slow to implement searches

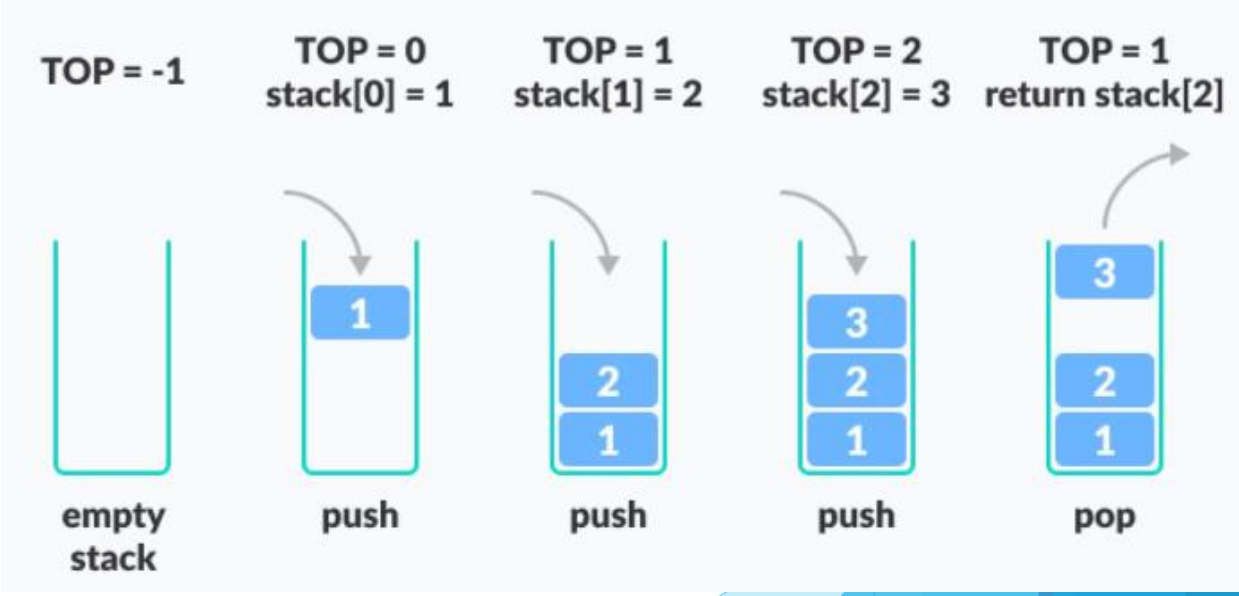
Linked Lists

- It is a very flexible, dynamic data structure in which elements (called nodes) form a sequential list
- Each node is allocated space as it is added to the list
- Every node in the list points to the next node in the list



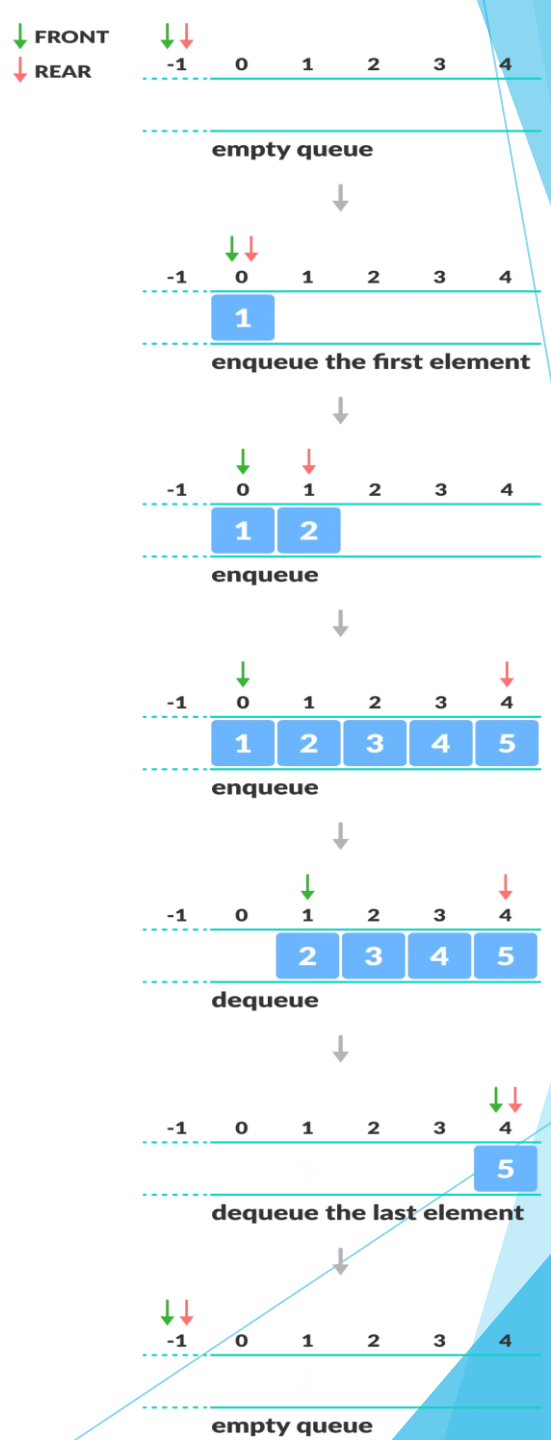
Stacks

- A Stack is a linear DS that follows the LIFO (Last-In-First-Out) principle
- It contains only one pointer ‘top pointer’ pointing to the topmost element of the stack
- Insertion and Deletion of elements are done at only one end, which is known as the top of the stack
- Implemented using array/linked list



Queues

- A queue is a FIFO DS in which the element that is inserted first is the first one to be taken out
- Elements in a queue are added at one end called the **rear** and removed from the other end called the **front**
- Implemented using array/linked list



Concept of Abstract Data Type(ADT)

- An Abstract Data Type (ADT) is the specification of a data type within some language, independent of an implementation
- An ADT does not specify how the data type is implemented
- The implementation details are hidden from the user of the ADT and protected from outside access; a concept referred to as encapsulation
- Real life example: Book is Abstract(Telephone Book is an implementation)

Concept of Abstract Data Type(ADT)…

- A data structure is the implementation for an ADT
- In an object-oriented language, an ADT and its implementation together make up a class
- Each operation associated with the ADT is implemented by a member function or method
- Array, List, Map, Queue, Set, Stack, Table, Tree, and Vector are examples of ADTs

Operations on Data Structures

- There are different types of operations that can be performed for the manipulation of data in every data structure. Some operations are

- **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting

- **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location

- **Deletion:** The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location

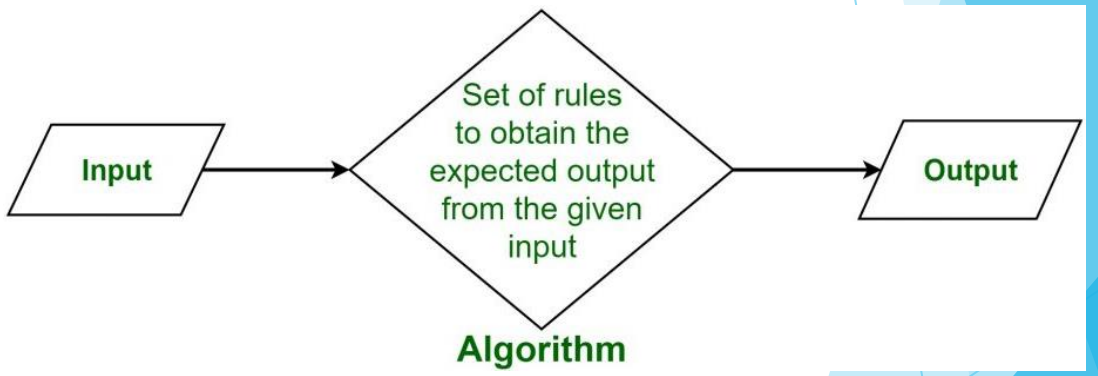
Operations on Data Structures…

- **Searching:** The process of finding the location of an element within the data structure is called Searching
- **Sorting:** The process of arranging the data structure in a specific order is known as Sorting
- **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

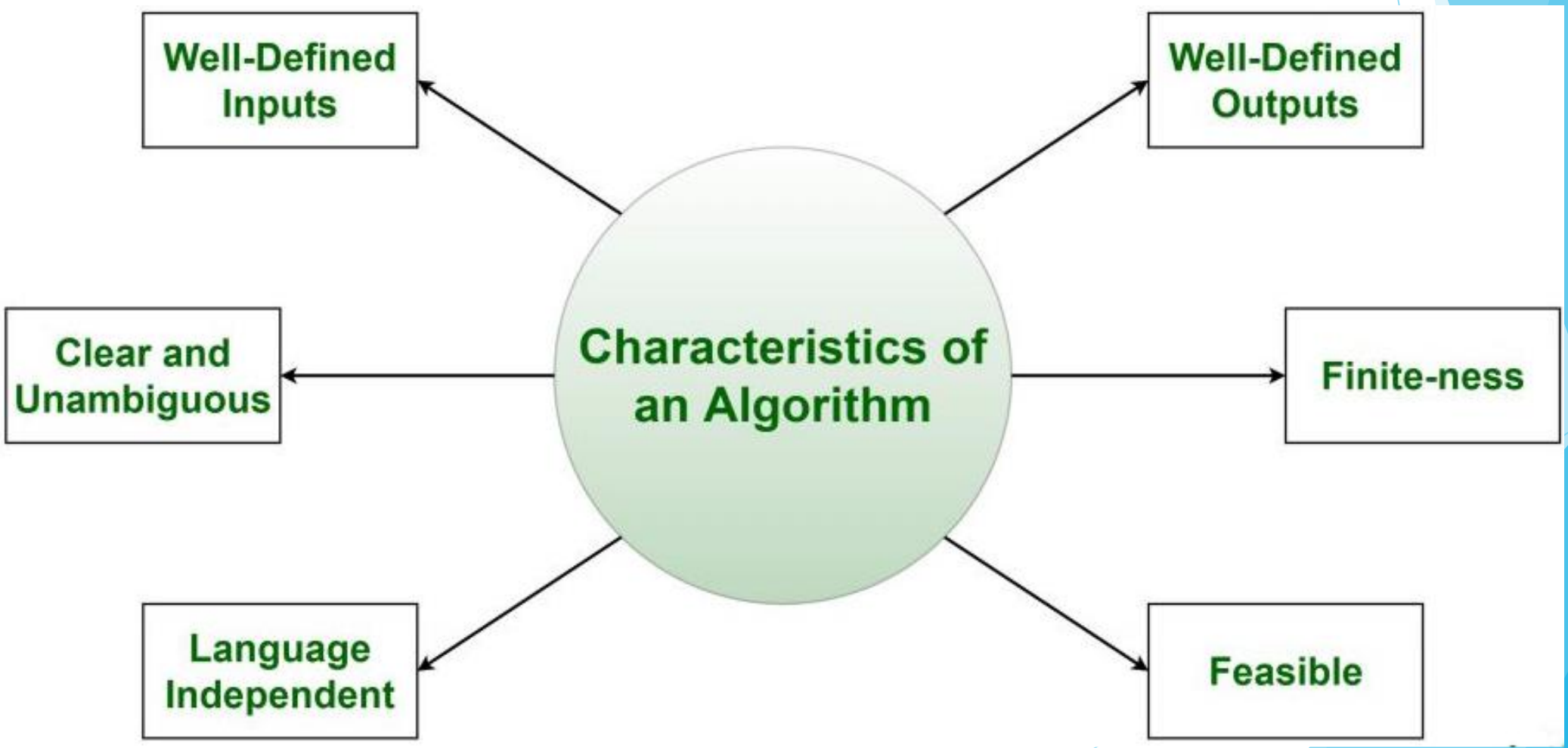
Introduction to Algorithms

- 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



Algorithm Complexity

- For an algorithm A, Algorithm Complexity is judged based on two parameters for an input of size n:

- **Time Complexity:** Time taken by the algorithm to solve the problem. It is measured by calculating the iteration of loops, number of comparisons, etc.

- **Space Complexity:** Space taken by the algorithm to solve the problem. It includes space used by necessary input variables and any extra space (excluding the space taken by inputs) that is used by the algorithm

Algorithm Analysis

- Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem
- Analysis of algorithms is the determination of the *amount of time and space resources required* to execute it

- Why is Algorithm Analysis important?
- To predict the behavior of an algorithm without implementing it on a specific computer
- The analysis is thus only an approximation; it is not perfect
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose

Types of Algorithm Analysis

- **Best case:** Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs

- **Worst case:** Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is not present at all then the worst case occurs

- **Average case:** In the average case take all random inputs and calculate the computation time for all inputs. And then we divide it by the total number of inputs

  Average case = all random case time / total no of case

Asymptotic Analysis (Growth of functions)

- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of *input size* (we don't measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size
- Using asymptotic analysis, we can very well conclude the *best case*, *average case*, and *worst case* scenario of an algorithm
- Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant
- Asymptotic notations are used to calculate the running time complexity of an algorithm

Asymptotic Notations

- Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes
- Three notations are used to calculate the running time complexity of an algorithm
  - Big O Notation (O)
  - Omega Notation (Ω)
  - Theta Notation (θ)

Asymptotic Notations…

- **Big O Notation (O)**
- The Big Oh notation O(n) is the formal way to express the *upper bound of an algorithm's running time*
- It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete

- **Omega Notation (Ω)**
- The Omega notation Ω(n) is the formal way to express the *lower bound of an algorithm's running time*
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete

Asymptotic Notations…

- **Theta Notation (θ)**
- The Theta notation θ(n) is the formal way to express both the *lower bound and the upper bound of an algorithm's running time*
- It measures the average case of an algorithm’s time complexity
- It represents the realistic time complexity of an algorithm
- Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm

Algorithm Strategies

- **Algorithm:** An algorithm is a sequence of computational steps that transform the input to an output. It is a tool for *solving* a well-specified computational *problem*
- **Strategy:** A strategy is an approach (or a series of approaches) devised to *solve* a computational *problem*
- *An algorithm is a strategy that always* **guarantees** *the correct answer*
- The Algorithm Strategy is a collection of methods or techniques to solve problems
- Some of the well-known strategies are Divide and Conquer, Greedy Method, Backtracking

Divide and Conquer

- The Divide and Conquer strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for the final answer
- Example: Merge sort, Quick sort
- This technique can be divided into the following three parts:
  - Divide: This involves dividing the problem into smaller sub-problems
  - Conquer: Solve sub-problems by calling recursively until solved
  - Combine: Combine the sub-problems to get the final solution of the whole problem

Greedy Method

- In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen

- In the greedy method, at each step, a decision is made to choose the local optimum, without thinking about the future consequences

- Example: Prim's, Kruskal's, Dijkstra's (Minimal Spanning Tree Algorithm)

Backtracking

- Backtracking technique is very useful in solving combinatorial problems that have a *single unique solution* where we have to find the correct combination of steps that lead to fulfillment of the task
- Such problems have multiple stages and there are multiple options at each stage
- This approach is based on exploring each available option at every stage *one-by-one*
- While exploring an option if a point is reached that doesn't seem to lead to the solution, the program control backtracks one step, and starts exploring the next option
- In this way, the program explores all possible course of actions and finds the route that leads to the solution.
- Example: N-queen problem, maize problem

Any Questions?