Course Details

Course Title : Data Structure and Algorithms (3 Cr.)

Course Code : CACS201

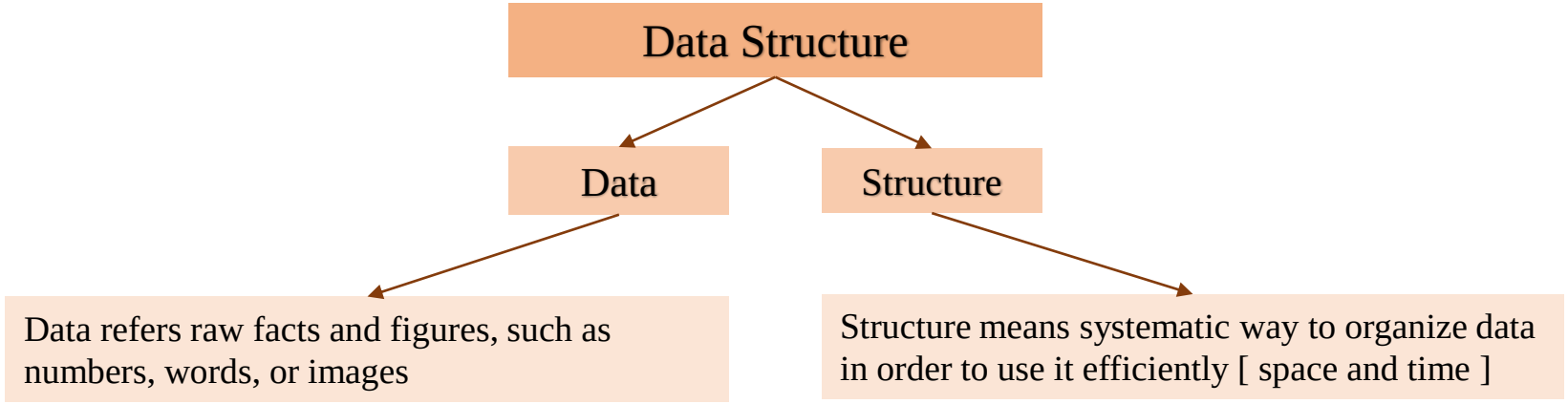
Why should we learn DSA?

1. Data Structures allow us to organize and store data, whereas Algorithms allow us to process that data meaningfully.
2. Learning Data Structures and Algorithms will help us become better Programmers.
3. We will be able to write code that is more effective and reliable.
4. We will also be able to solve problems more quickly and efficiently.

Unit 1

Introduction to Data Structure

Data Structure



- A data structure is a way of organizing and managing data in a systematic and efficient manner in memory.
- A well-designed data structure can help optimize performance, reduce memory usage, and improve the overall efficiency of computer programs and systems that rely on data.
- Some common examples of data structures include arrays, linked lists, stacks, queues, trees, and graphs, each of which has its own unique characteristics and uses.
- Interface and Implementation are foundation of a data structure. Interface represents the set of operations that a data structure supports. Implementation provides the internal representation of a data structure and definition of the algorithms used in the operations of the data structure.

Basic Terminologies related to Data Structure

- **Data** − Data are values or set of values. For example, the Employee's name and ID are the data related to the Employee.
- **Data Item** − Data item refers to single unit of values.
- **Group Items** − Data items that are divided into sub items are called as Group Items. For example, an employee's name can have a first, middle, and last name.
- **Elementary Items** − Data items that cannot be divided are called as Elementary Items. For example, the ID of an Employee.
- **Attribute and Entity** − An entity is that which contains certain attributes or properties, which may be assigned values.

Attributes	ID	First Name	Last Name	Gender	Job Title
Values	01	Sujan	Poudel	Male	Instructor

- **Entity Set** − Entities of similar attributes form an entity set.
- **Field** − Field is a single elementary unit of information representing an attribute of an entity.
- **Record** − Record is a collection of field values of a given entity.
- **File** − File is a collection of records of the entities in a given entity set.

Characterstics of Data Structure

- **Correctness :** Data structure implementation should implement its interface correctly.
- **Time Complexity :** Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity :** Memory usage of a data structure operation should be as little as possible.

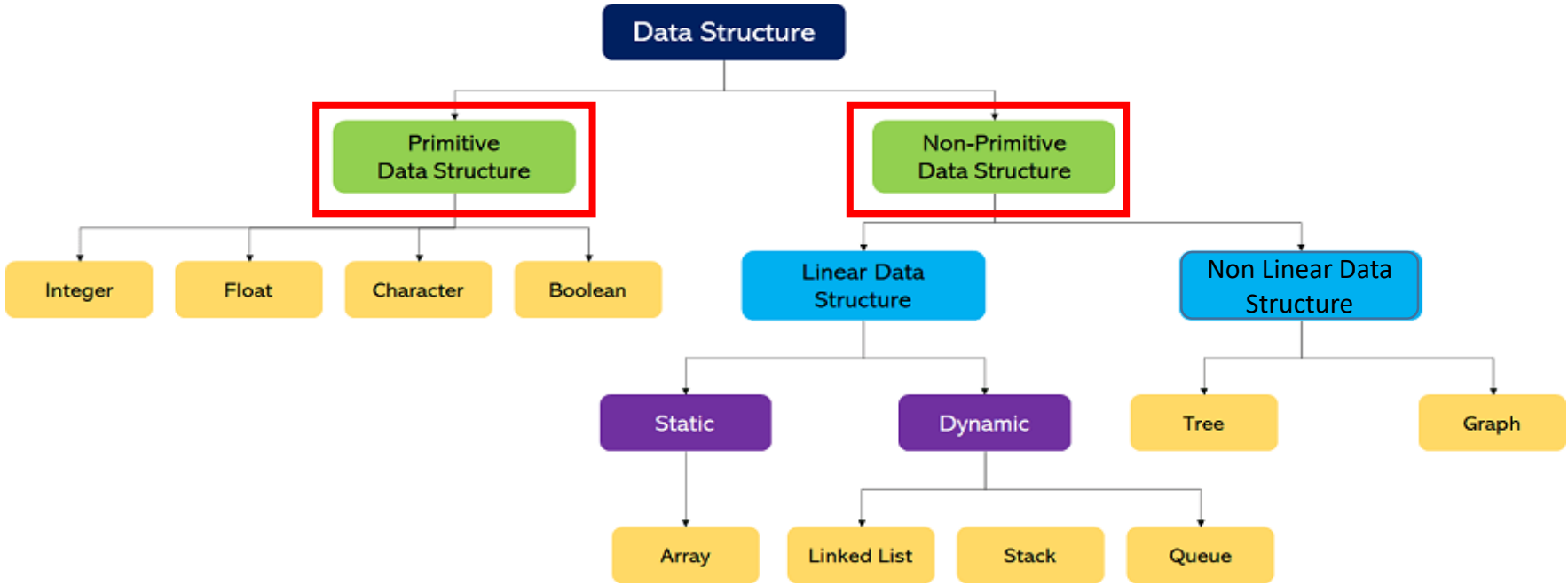
Need of Data Structure

- As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

1. **Data Search :** Consider an inventory of 1 million(106) items of a store. If the application is to search an item, it has to search an item in 1 million(106) items every time slowing down the search. As data grows, search will become slower.
2. **Processor speed :** Processor speed although being very high, falls limited if the data grows to billion records.
3. **Multiple requests :** As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

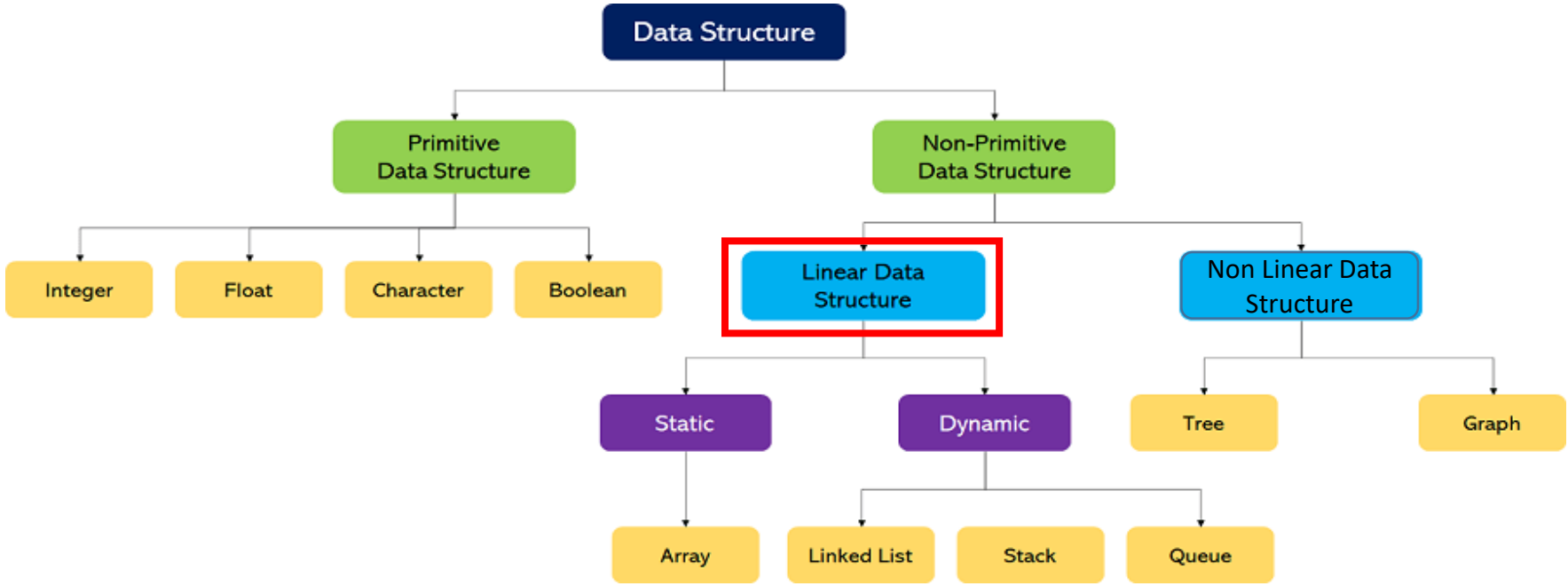
- To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Classification of Data Structures



- ***Primitive Data Structures*** are the data structures consisting of the numbers and the characters that come in-built into programs. These data structures can be manipulated or operated directly by machine-level instructions.
- ***Non-Primitive Data Structures*** are those data structures derived from Primitive Data Structures and can't be manipulated or operated directly by machine-level instructions. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).

Classification of Data Structures

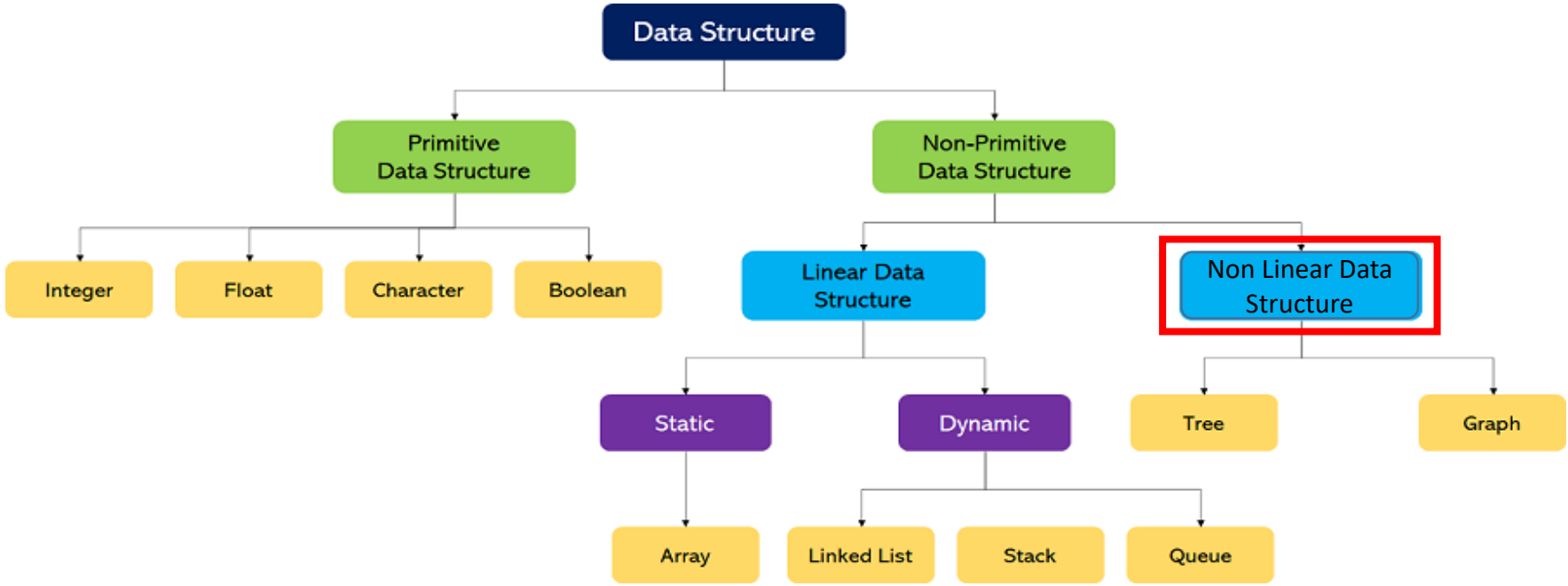


- Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -

1. Linear Data Structures:

A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure. The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element. However, it is not necessarily true in the case of memory, as the arrangement may not be sequential.

Classification of Data Structures

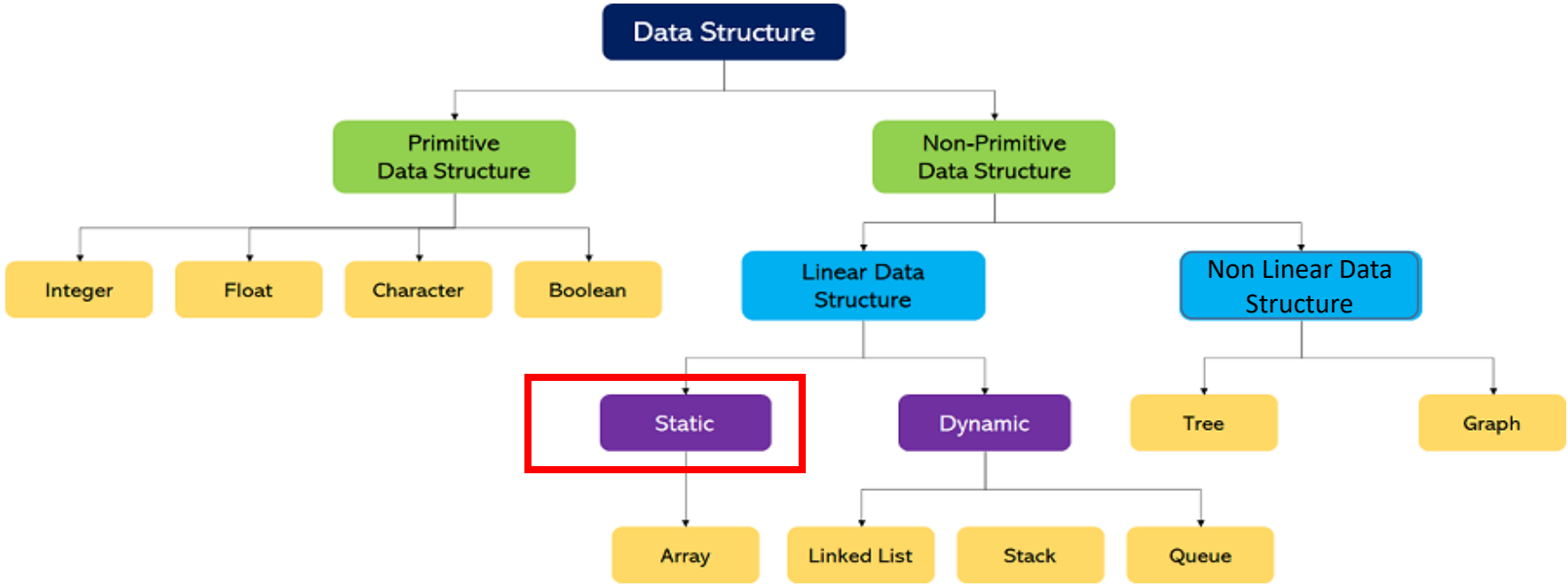


2. Non-Linear Data Structures:

Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

- Examples: Tree and Graph

Classification of Data Structures

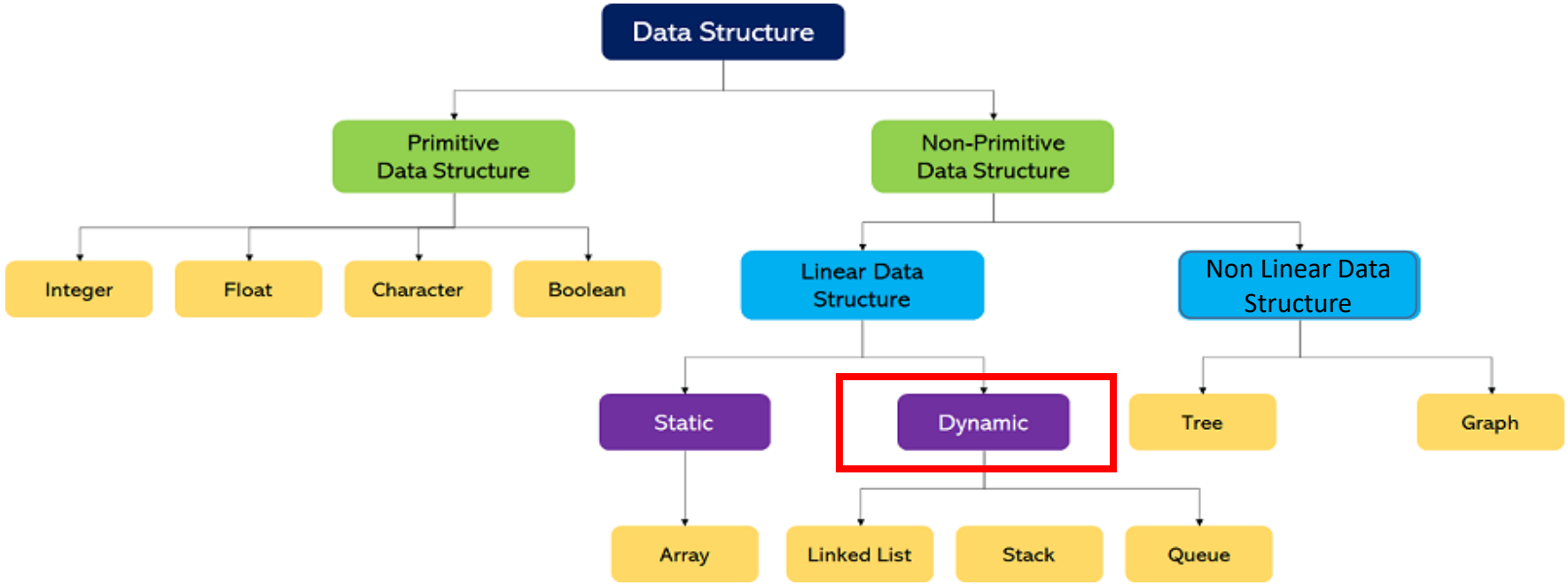


Based on memory allocation, the Linear Data Structures are further classified into two types:

1. Static Data Structures: The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered.

- The **Array** is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.

Classification of Data Structures



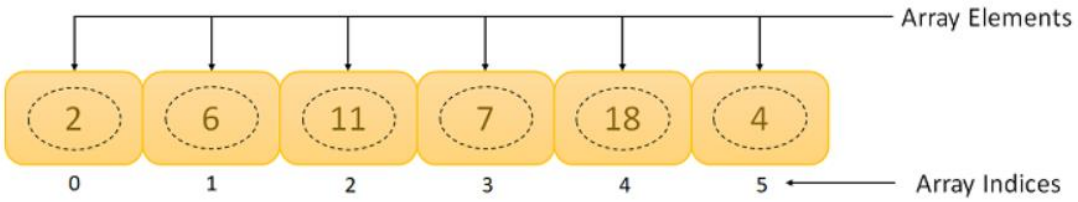
2. Dynamic Data Structures: The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code.

- Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the code.
- **Linked Lists**, **Stacks**, and **Queues** are common examples of dynamic data structures/

Linear Data Structures

1. Array

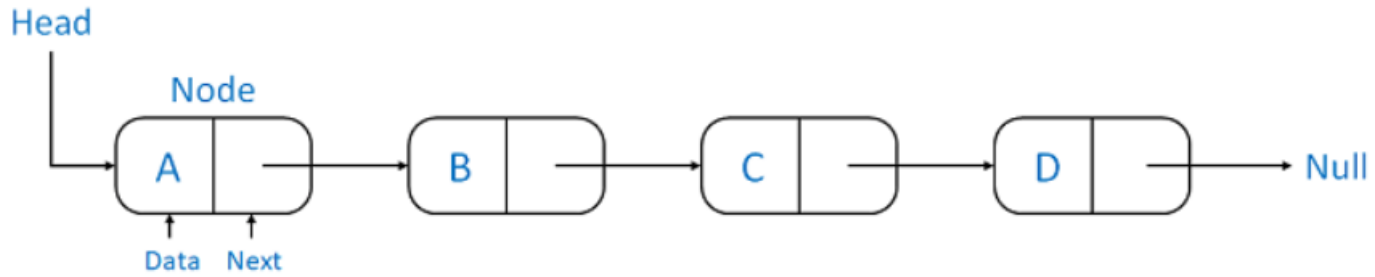
- An **Array** is a data structure used to collect multiple data elements of the same data type into one variable.
- Instead of storing multiple values of the same data types in separate variable names, we could store all of them together into one variable.
- Classification of Array: One-Dimensional Array, Two-Dimensional Array and Multidimensional Array.



Linear Data Structures

2. Linked List

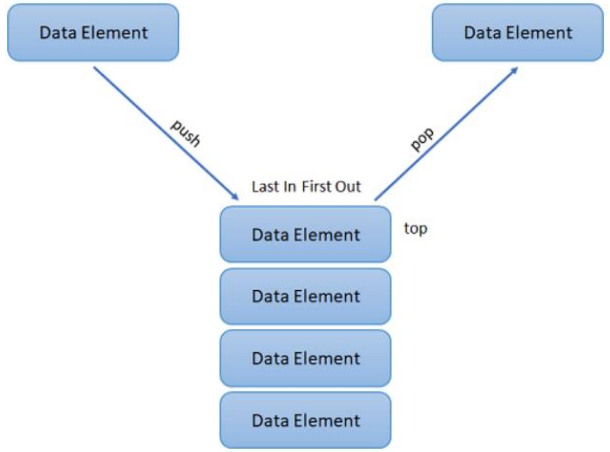
- A Linked List is another example of a linear data structure used to store a collection of data elements dynamically.
- Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list.
- The pointer of the last node of the linked list consists of a null pointer, as it points to nothing.
- Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.
- Classification of Linked List: Singly-Linked List, Doubly-Linked List and Circular Linked List.



Linear Data Structures

3. Stacks

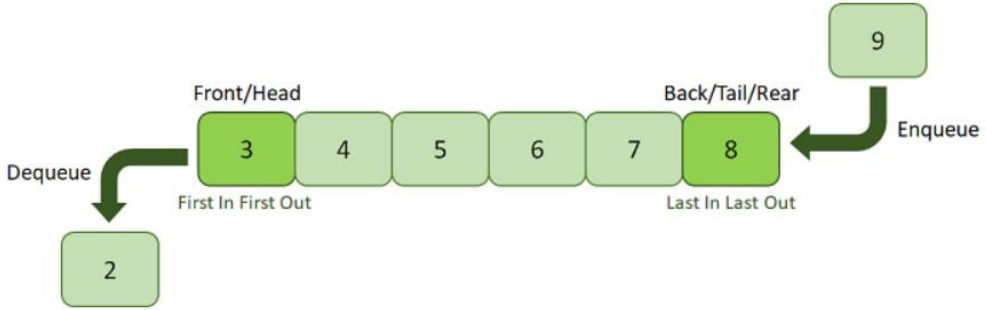
- A Stack is a Linear Data Structure that follows the LIFO (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top.
- Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List.
- Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.
- Two primary operations in the stack: PUSH and POP.



Linear Data Structures

4. Queue

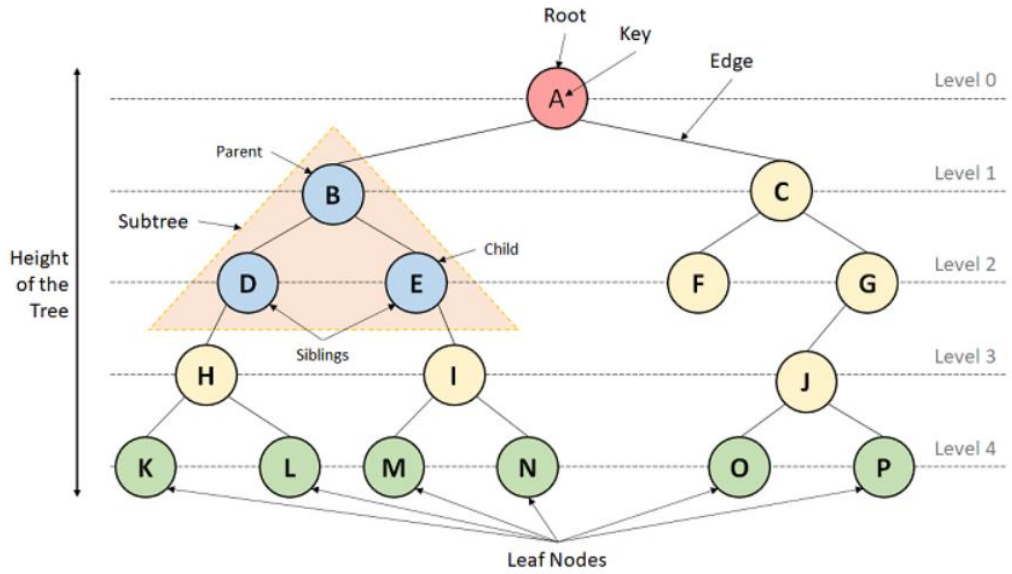
- A Queue is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements.
- The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end.
- Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements.
- Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.
- Two primary operations in the stack: Enqueue and Dequeue.



Non Linear Data Structures

1. Tree

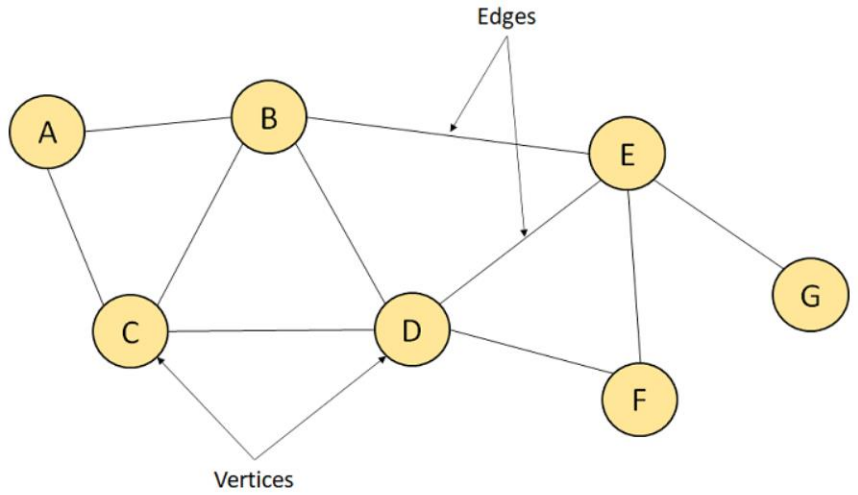
- A Tree is a Non-Linear Data Structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").
- The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively.
- It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.



Non Linear Data Structures

2. Graph

- A Graph is another example of a Non-Linear Data Structure comprising a finite number of nodes or vertices and the edges connecting them. The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks. For instance, the nodes or vertices of a Graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.
- The Graph data structure, G is considered a mathematical structure comprised of a set of vertices, V and a set of edges, E as shown below:
- G = (V,E)



Basic Operations of Data Structures

1. Traversal: Traversing a data structure means accessing each data element exactly once so it can be administered. For example, traversing is required while printing the names of all the employees in a department.

2. Search: Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints. Such a data element may or may not be present in the given set of data elements. For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.

3. Insertion: Insertion means inserting or adding new data elements to the collection. For example, we can use the insertion operation to add the details of a new employee the company has recently hired.

4. Deletion: Deletion means to remove or delete a specific data element from the given list of data elements. For example, we can use the deleting operation to delete the name of an employee who has left the job.

5. Sorting: Sorting means to arrange the data elements in either Ascending or Descending order depending on the type of application. For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.

Basic Operations of Data Structures

6. Merge: Merge means to combine data elements of two sorted lists in order to form a single list of sorted data elements.

7. Create: Create is an operation used to reserve memory for the data elements of the program. We can perform this operation using a declaration statement. The creation of data structure can take place either during the following:

a. Compile-time

b. Run-time

For example, the malloc() function [used to allocate memory dynamically during runtime] is used in C Language to create data structure.

8. Selection: Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.

9. Update: The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.

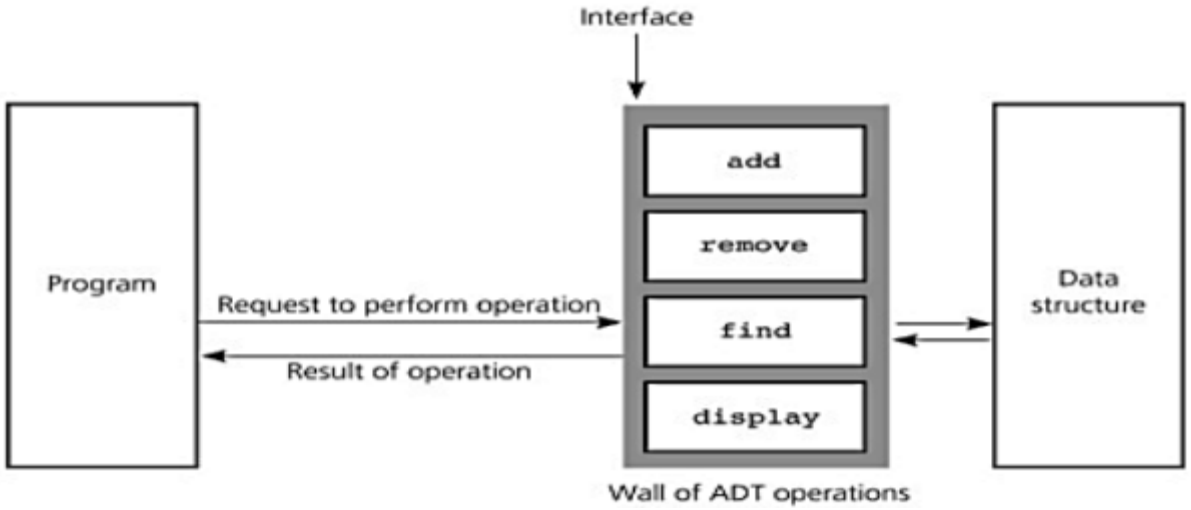
10. Splitting: The Splitting operation allows us to divide data into various subparts decreasing the overall process completion time.

Abstract Data Type

- Abstract data type (ADT) is a type of data that is defined by what it can do and what operations can be performed on it, rather than how it is made or how it works internally.
- It provides a general idea of what a particular data structure can do and how it can be used, without worrying about the specific details of how it is implemented.
- They provide a conceptual representation of a data structure, separating the implementation from the interface or behavior of the data structure.
- ADTs are useful because they allow programmers to focus on the functionality of the data structure rather than its implementation details, making code more modular, easier to maintain, and more reusable.
- Common examples of ADTs include stacks, queues, lists, trees, and graphs, which can be used to represent real-world problems in a logical and organized way.
- For example, a stack ADT can be used to represent a pile of plates, where the last plate put on the pile is the first plate to be removed. Similarly, a queue ADT can be used to represent a line of people waiting to buy tickets, where the first person in line is the first to buy a ticket.

Abstract Data Type

- It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.
- The user of data type does not need to know how that data type is implemented, for example: we have been using primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.



- So, a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type.

Importance of Data Structure

1. **Efficient data storage and retrieval:** Data structures provide a systematic and efficient way of storing and retrieving data from memory, enabling faster and more effective access to data.
2. **Algorithm design:** Data structures form the backbone of algorithm design, and the choice of data structure often determines the efficiency and complexity of an algorithm.
3. **Resource management:** Data structures help to manage limited resources such as memory and storage space, allowing programs to use resources more efficiently and effectively.
4. **Better organization:** By structuring data in a logical and organized way, data structures make it easier for programmers to understand and manage large datasets, reducing the likelihood of errors and improving code quality.
5. **Reusability and modularity:** Data structures are reusable and modular, meaning they can be used across multiple programs and applications. This saves time and effort for programmers, as they do not need to create new data structures for each project.
6. **Problem-solving:** Data structures provide a powerful way to represent and solve real-world problems, such as sorting, searching, and graph traversal, which are important in a wide range of applications.

Algorithm

- 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.
- From the data structure point of view, following are some important categories of algorithms:
  - **Search:** Algorithm to search an item in a data structure.
  - **Sort:** Algorithm to sort items in a certain order.
  - **Insert:** Algorithm to insert item in a data structure.
  - **Update:** Algorithm to update an existing item in a data structure.
  - **Delete:** Algorithm to delete an existing item in a data structure.

Characteristics of Algorithm

- Not all procedures can be called an algorithm. An algorithm should have the following characteristics:
  - **Unambiguous:** Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
  - **Input:** An algorithm should have 0 or more well-defined inputs.
  - **Output:** An algorithm should have 1 or more well-defined outputs, and should match the desired output.
  - **Finiteness:** Algorithms must terminate after a finite number of steps.
  - **Feasibility:** Should be feasible with the available resources.
  - **Independent:** An algorithm should have step-by-step directions, which should be independent of any programming code.

How to write an Algorithm?

- There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.
- As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control [if-else], etc. These common constructs can be used to write an algorithm.
- We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

✓ ***Example:*** *Design An Algorithm To Add Two Numbers And Display The Result.*

- ***Step 1*** *- START ADD*
- ***Step 2*** *- get values of a & b*
- ***Step 3*** *- c «a +b*
- ***Step 4*** *- display c*
- ***Step 5*** *- STOP*

Algorithm Analysis:

- Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following:
  - **Priori Analysis:** This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
  - **Posterior Analysis:** This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language.

- This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.
- We shall learn about a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved.
- The running time of an operation can be defined as the number of computer instructions executed per operation.

Algorithm Complexity:

- Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.
  - **Time Factor:** Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
  - **Space Factor:** Space is measured by counting the maximum memory space required by the algorithm.
- The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

Space Complexity:

- Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components:
  1. **Fixed part :** It is a space required to store certain data and variables that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
  2. **Variable part:** It is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.
- Space complexity **S(P)** of any algorithm **P** is,

  S(P) = C + SP(I), where **C** is the fixed part and **S(I)** is the variable part of the algorithm
- Suppose, Algorithm: SUM (A, B)
  - **Step 1 :** START
  - **Step 2 :** C «~~ A +B + 10
  - **Step 3 :** Stop
- Here we have three variables A, B, and C and one constant. Hence 5(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity:

- Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion.
- Time requirements can be defined as a numerical function **T(n)**,where

T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-hit integers takes n steps.

Consequently, the total computational time is **T(n) = c * n**, where

c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

Trade-off between time and space complexity:

In many cases, there is a trade-off between time and space complexity, where an algorithm may use more memory to run faster or use less memory to run slower. Balancing these factors is an important consideration when designing algorithms for real-world applications.

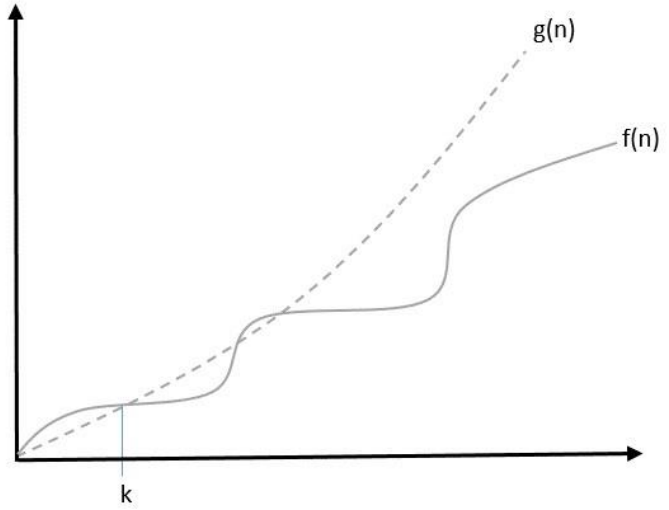
Asymptotic analysis

- Asymptotic analysis of an algorithm refers to defining the mathematical foundation,/framing of its run-time performance. 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 analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as **f(n)** and may be for another operation it is computed as **g(n2)**.
  - This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases.
  - Similarly, the running time of both operations will be nearly the same if **n** is significantly small.
- Usually, the time required by an algorithm falls under three types:
  - **Best Case:** Minimum time required for program execution. It is represented by Big O notation [O].
  - **Average Case:** Average time required for program execution. It is represented by Omega notation [Ω].
  - **Worst Case:** Maximum time required for program execution. It is represented by Theta notation [θ]

Asymptotic Notations

1. Big Oh Notation, O

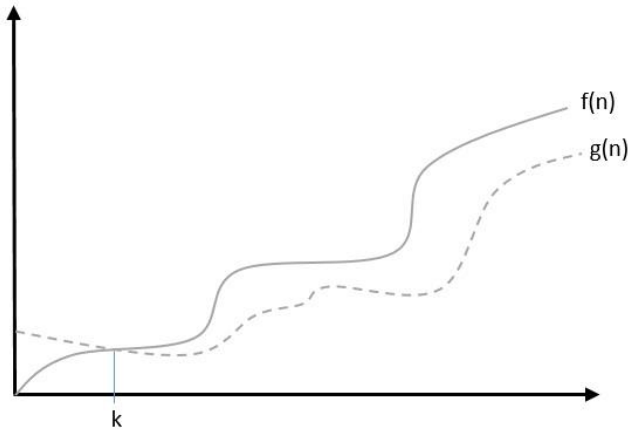
- The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



Asymptotic Notations

2. Big Omega Notation, Ω

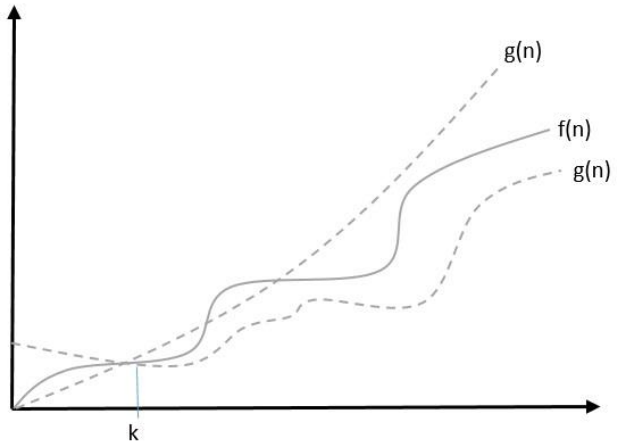
- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



Asymptotic Notations

3. Theta Notation, θ

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. Some may confuse the theta notation as the average case time complexity; while big theta notation could be almost accurately used to describe the average case, other notations could be used as well. It is represented as follows −



Common Asymptotic Notations

Following is a list of some common asymptotic notations:

Name	Notations
constant	O(1)
Logarithmic	O(log n)
linear	O(n)
n log n	O(n log n)
quadratic	$O(n^2)$
cubic	$O(n^3)$
polynomial	$n^{O(1)}$
exponential	$2^{O(n)}$