DATA STRUCTURES

UNIT-1:

Data Structure:

It is a particular way of organizing and storing data in a computer. So that it can be accessed and modified efficiently. A data structure will have a collection of data and functions or operations that can be applied on the data.

TYPES OF DATA STRUCTURES

1. Primitive and Non-Primitive Data Structure

Primitive Data Structure defines a set of primitive elements that do not involve any other elements as its subparts. These are generally built-in data type in programming languages.

E.g.:- Integers, Characters etc.

Non-Primitive Data Structures are that defines a set of derived elements such as Arrays, Structures and Classes.

2. Linear and Non-Linear Data Structure

A Data Structure is said to be linear, if its elements from a sequence and each element have a unique successor and predecessor.

E.g.:- Stalk, Queue etc.

Non-Linear Data Structures are used to represent data that have a hierarchical relationship among the elements. In non-linear Data Structure every element has more than one predecessor and one successor.

E.g.:- Trees, Graphs

3. Static and Dynamic Data Structure

A Data Structure is referred as Static Data Structure, if it is created before program execution i.e., during compilation time. The variables of Static Data Structure have user specified name.

E.g.:- Array

Data Structures that are created at run time are called as Dynamic Data Structure. The variables of this type are known always referred by user defined name instead using their addresses through pointers.

E.g.:- Linked List

4. Sequential and Direct Data Structure

This classification is with respect to access operation associated with the data type. Sequential access means the locations are accessed in a sequential form.

E.g.:- To access $\boldsymbol{n^{th}}$ element, it must access preceding $\boldsymbol{n-1}$ data elements

E.g.:- Linked List

Direct Access means any element can access directly without accessing its predecessor or successor i.e., $\boldsymbol{n^{th}}$ element can be accessed directly.

E.g.:- Array

OPERATIONS ON DATA STRUCTURE

The basic operations that can be performed on a Data Structure are:

i) Insertion: - Operation of storing a new data element in a Data Structure.
ii) Deletion: - The process of removal of data element from a Data Structure.
iii) Traversal: - It involves processing of all data elements present in a Data Structure.
iv) Merging: - It is a process of compiling the elements of two similar Data Structures to form a new Data Structure of the same type.
v) Sorting: - It involves arranging data element in a Data Structure in a specified order.
vi) Searching: - It involves searching for the specified data element in a Data Structure.

COMPLEXITY OF ALGORITHMS

Generally algorithms are measured in terms of time complexity and space complexity.

1. Time Complexity

Time Complexity of an algorithm is a measure of how much time is required to execute an algorithm for a given number of inputs. And it is measured by its rate of growth relative to a standard function. Time Complexity can be calculated by adding compilation time and execution time. Or it can do by counting the number of steps in an algorithm.

2. Space Complexity

Space Complexity of an algorithm is a measure of how much storage is required by the algorithm. Thus space complexity is the amount of computer memory required during program execution as a function of input elements. The space requirement of algorithm can be performed at compilation time and run time.

ASYMPTOTIC ANALYSIS

For analysis of algorithm it needs to calculate the complexity of algorithms in terms of resources, such as time and space. But when complexity is calculated, it does not provide

the exact amount of resources required. Therefore instead of taking exact amount of resources, the complexity of algorithm is represented in a general mathematical form which will give the basic nature of algorithm.

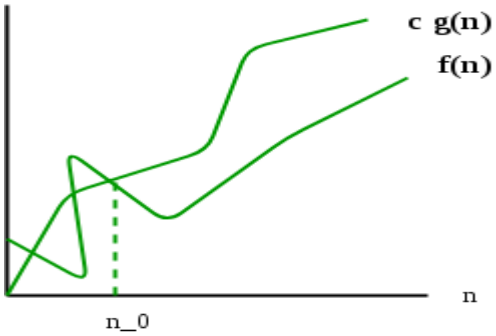
Thus on analyzing an algorithm, it derives a mathematical formula to represent amount of time and space required for the execution. The Asymptotic analysis of algorithm evaluates the performance of an algorithm in terms of input size It calculates how does the time taken by an algorithm increases with input size. It focuses on:

i) Analyzing the problem with large input size.
ii) Considers only leading terms of formula. Since the lower order term contracts lesser to the overall complexity as input grows larger.
iii) Ignores the coefficient of leading term.

ASYMPTOTIC NOTATION

The commonly used Asymptotic Notations to calculate the complexity of algorithms are:

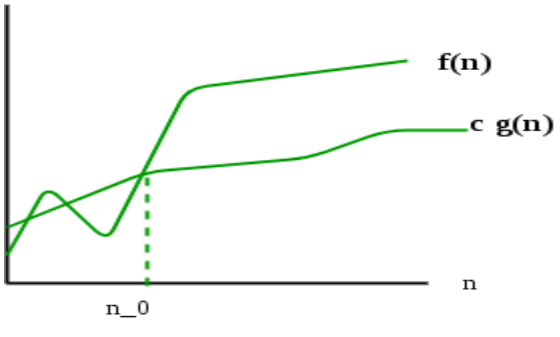
1. **Big Oh – O: -** The notation **O**(n) is the formal way to express the upper bound of an algorithm's complexity.



f(n) = O(g(n))

```
O(g(n)) = { f(n): there exist positive constants c and
                  n0 such that 0 <= f(n) <= c*g(n) for
                  all n >= n0}
```

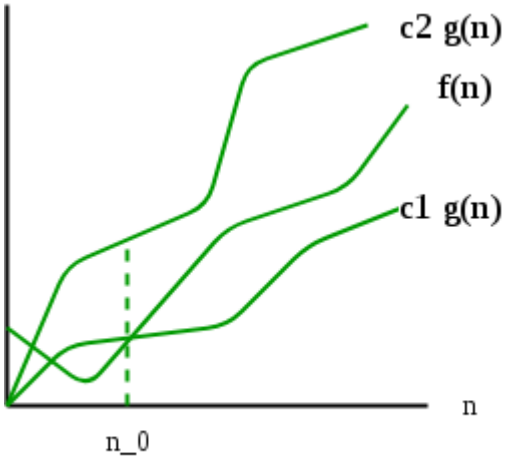
2. **Big Omega – Ω: -** The notation **Ω**(n) is the formal way of representing lower bound of an algorithm's complexity.



f(n) = Omega(g(n))

```
Ω (g(n)) = {f(n): there exist positive constants c and
                  n0 such that 0 <= c*g(n) <= f(n) for
                  all n >= n0}.
```

3. **Big Theta** – **Θ**: - The notation **Θ**($\boldsymbol{n}$) is the formal way to representing lower bound and upper bound of an algorithm's complexity.



f(n) = theta(g(n))

```
Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such
                 that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}
```

STACK

Stack is a linear Data Structure in which all the insertion and deletion operations are done at one end, called top of the stack and operations are performed in Last In First Out [**LIFO**] order. A stack can be implemented by using an array or by using linked list.

The array representation method needs to set the amount memory needed initially. So they are limited in size, i.e. it can't shrink or expand. Linked List representation uses Dynamic memory management method. So they are not limited in size i.e. they can shrink or expand.

The class declaration for stack using array is represented as:

```
class stack
 {
        int a[10], st;
        public:
        stack()
        {
        st =-1;
        }
        void push(int);
        void pop();
};
```