# MODULE-I
# INTRODUCTION TO DATA STRUCTURES

## Basic Concepts: Introduction to Data Structures:

A data structure is a way of storing data in a computer so that it can be used efficiently and it will allow the most efficient algorithm to be used. The choice of the data structure begins from the choice of an abstract data type (ADT). A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structure introduction refers to a scheme for organizing data, or in other words it is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations.

A data structure should be seen as a logical concept that must address two fundamental concerns.

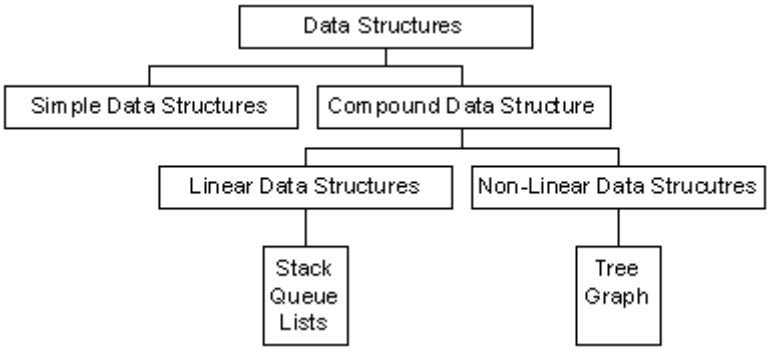
1. First, how the data will be stored, and
2. Second, what operations will be performed on it.

As data structure is a scheme for data organization so the functional definition of a data structure should be independent of its implementation. The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation. The way in which the data is organized affects the performance of a program for different tasks. Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data. Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs.

## Classification of Data Structures:

Data structures can be classified as

- Simple data structure
- Compound data structure
- Linear data structure
- Non linear data structure




[Fig 1.1 Classification of Data Structures]

**Simple Data Structure:**

Simple data structure can be constructed with the help of primitive data structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, arrays, pointers, structures, unions, etc. are examples of primitive data structures.

**Compound Data structure:**

Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as

- Linear data structure
- Non-linear data structure **Linear Data Structure:**

Linear data structures can be constructed as a continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear Data Structures the relationship of adjacency is maintained between the data elements.

**Operations applied on linear data structure:**

The following list of operations applied on linear data structures

1. Add an element
2. Delete an element
3. Traverse
4. Sort the list of elements
5. Search for a data element

For example Stack, Queue, Tables, List, and Linked Lists.

**Non-linear Data Structure:**

Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the data items.

**Operations applied on non-linear data structures:**

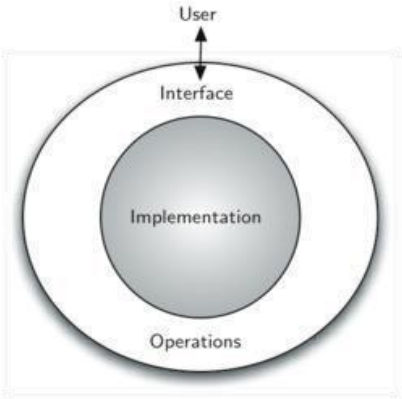
The following list of operations applied on non-linear data structures.

1. Add elements
2. Delete elements
3. Display the elements
4. Sort the list of elements
5. Search for a data element

For example Tree, Decision tree, Graph and Forest

## Abstract Data Type:

An abstract data type, sometimes abbreviated ADT**,** is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user's view. This is called information hiding**.** The implementation of an abstract data type, often referred to as a data structure**,** will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.




[Fig. 1.2: Abstract Data Type (ADT)]

## Algorithms:

### Structure and Properties of Algorithm:

An algorithm has the following structure

1. Input Step
2. Assignment Step
3. Decision Step
4. Repetitive Step
5. Output Step

An algorithm is endowed with the following properties:

1. **Finiteness:** An algorithm must terminate after a finite number of steps.
2. **Definiteness:** The steps of the algorithm must be precisely defined or unambiguously specified.
3. **Generality:** An algorithm must be generic enough to solve all problems of a particular class.
4. **Effectiveness:** the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation.

5. **Input-Output:** The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

## Different Approaches to Design an Algorithm:

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties.

**Practical Algorithm Design Issues**:

1. **To save time (Time Complexity):** A program that runs faster is a better program.
2. **To save space (Space Complexity):** A program that saves space over a competing program is considerable desirable.

### Efficiency of Algorithms:

The performances of algorithms can be measured on the scales of **time** and **space**. The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

**Time Complexity:** The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.

**Space Complexity:** The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.

The time complexity of an algorithm can be computed either by an **empirical** or **theoretical** approach. The **empirical** or **posteriori testing** approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

### Analyzing Algorithms

Suppose M is an algorithm, and suppose n is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) with some standard functions. The most common computing times are

$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$

Example:

**Program Segment A**

```
-------------------------
      x =x + 2;
-------------------------
```

**Program Segment B**

```
-------------------------
for k =1 to n do
      x =x + 2;
end;
-------------------------
```

**Program Segment C**

```
-------------------------
for j =1 to n do
  for x = 1 to n do
       x =x + 2;
  end
end;
-------------------------
```

**Total Frequency Count of Program Segment A**

| Program Statements | Frequency Count |
|---|---|
| ---------- | |
| x =x + 2; | 1 |
| ---------- | |
| Total Frequency Count | 1 |

**Total Frequency Count of Program Segment B**

| Program Statements | Frequency Count |
|---|---|
| ---------- | |
| for k =1 to n do | (n+1) |
| x =x + 2; | n |
| end; | n |
| ---------- | |
| Total Frequency Count | 3n+1 |

**Total Frequency Count of Program Segment C**

| Program Statements | Frequency Count |
|---|---|
| ---------- | |
| for j =1 to n do | (n+1) |
| for x = 1 to n do | n(n+1) |
| x =x + 2; | $n^2$ |
| end | $n^2$ |
| end; | n |
| ---------- | |
| Total Frequency Count | $3n^2+3n+1$ |

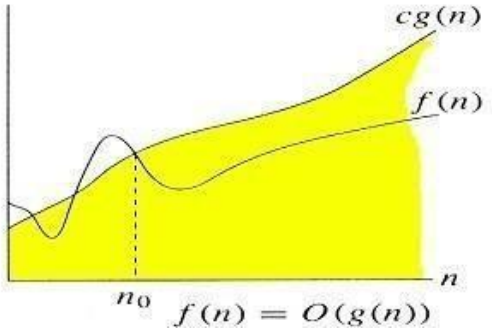
The total frequency counts of the program segments A, B and C given by 1, (3n+1) and ($3n^2$+3n+1) respectively are expressed as O(1), O(n) and O($n^2$). These are referred to as the time complexities of the program segments since they are indicative of the running times of the program segments. In a similar manner space complexities of a program can also be expressed in terms of mathematical notations, which is nothing but the amount of memory they require for their execution.

**Asymptotic Notations:**

It is often used to describe how the size of the input data affects an algorithm's usage of computational resources. Running time of an algorithm is described as a function of input size n for large n.

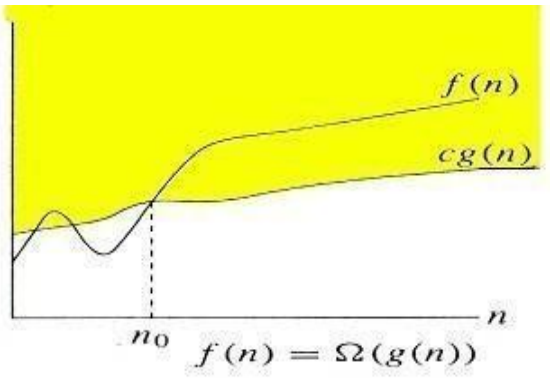
**Big oh(O):** Definition: f(n) = O(g(n)) (read as f of n is big oh of g of n) if there exist a positive integer $n_0$ and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$. Here g(n) is the upper bound of the function f(n).

| f(n) | g(n) | |
|---|---|---|
| $16n^3 + 45n^2 + 12n$ | $n^3$ | $f(n) = O(n^3)$ |
| $34n - 40$ | n | f(n) = O(n) |
| 50 | 1 | f(n) = O(1) |




**Omega(Ω):** Definition: f(n) = Ω(g(n)) ( read as f of n is omega of g of n), if there exists a positive integer $n_0$ and a positive number c such that $|f(n)| \geq c\ |g(n)|$ for all $n \geq n_0$. Here g(n) is the lower bound of the function f(n).

| f(n) | g(n) | |
|---|---|---|
| $16n^3 + 8n^2 + 2$ | $n^3$ | $f(n) = \Omega(n^3)$ |
| $24n + 9$ | n | $f(n) = \Omega(n)$ |



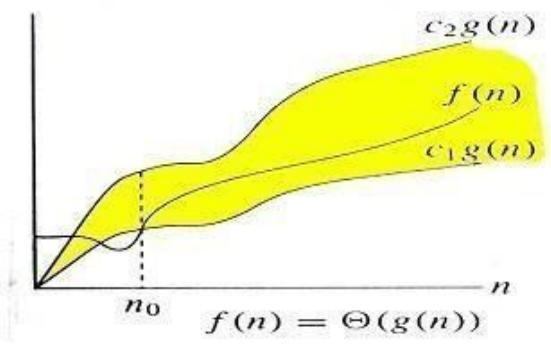

**Theta(Θ):** For a function g(n), Θ(g(n)) is given by the relation:

$\Theta(g(n)) = \{ f(n)$: there exist positive constants $c_1$, $c_2$ and $n_0$
such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants $c_1$ and $c_2$ such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.

If a function f(n) lies anywhere in between $c_1 g(n)$ and $c_2 > g(n)$ for all $n \geq n0$, then f(n) is said to be asymptotically tight bound.

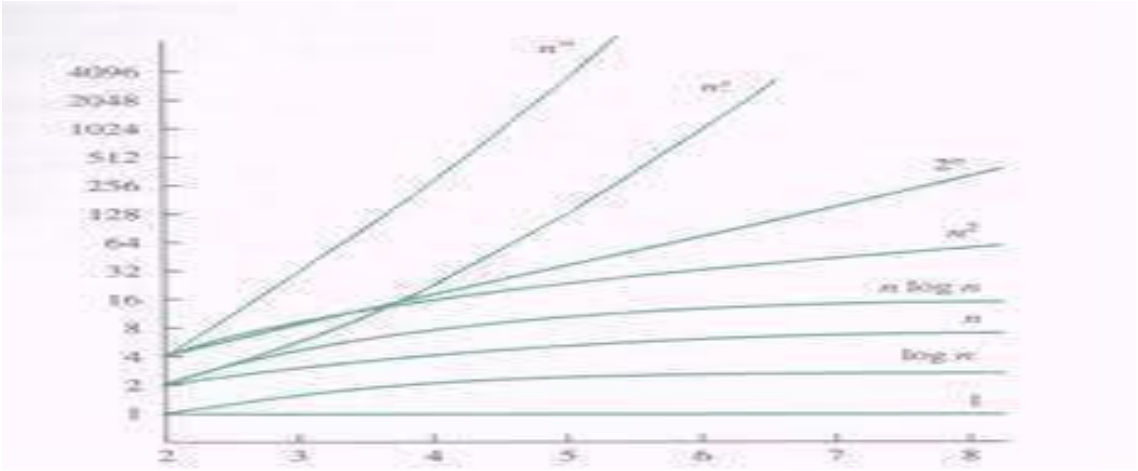
| f(n) | g(n) | |
|---|---|---|
| $16n^3 + 30n^2 - 90$ | $n^2$ | $f(n) = \Theta(n^2)$ |
| $7.2^n + 30n$ | $2^n$ | $f(n) = \Theta(2^n)$ |




**Little oh(o):** Definition: f(n) = O(g(n)) ( read as f of n is little oh of g of n), if f(n) = O(g(n)) and f(n) ≠ Ω(g(n)).

**Time Complexity:**

**Time Complexities of various Algorithms:**




**Numerical Comparision of Different Algorithms:**

| S.No. | $\log_2 n$ | n | $n\log_2 n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|
| 1. | 0 | 1 | 1 | 1 | 1 | 2 |
| 2. | 1 | 2 | 2 | 4 | 8 | 4 |
| 3. | 2 | 4 | 8 | 16 | 64 | 16 |
| 4. | 3 | 8 | 24 | 64 | 512 | 256 |
| 5. | 4 | 16 | 64 | 256 | 4096 | 65536 |

**Reasons for analyzing algorithms:**

1. To predict the resources that the algorithm requires • Computational Time(CPU consumption).
   - Memory Space(RAM consumption).
   - Communication bandwidth consumption.
2. To predict the running time of an algorithm

   Total number of primitive operations executed.

## Recursive Algorithms:

**GCD Design:** Given two integers a and b, the greatest common divisor is recursively found using the formula

$$gcd(a,b) = \begin{cases} a & \text{if } b=0 \\ b & \text{if } a=0 \\ gcd(b, a \bmod b) & \end{cases}$$

Base case

General case

**Fibonacci Design:** To start a fibonacci series, we need to know the first two numbers.

$$Fibonacci(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ Fibonacci(n-1) + fibonacci(n-2) & \end{cases}$$

Base case

General case

## Difference between Recursion and Iteration:

1. A function is said to be recursive if it calls itself again and again within its body whereas iterative functions are loop based imperative functions.
2. Reursion uses stack whereas iteration does not use stack.
3. Recursion uses more memory than iteration as its concept is based on stacks.
4. Recursion is comparatively slower than iteration due to overhead condition of maintaining stacks.
5. Recursion makes code smaller and iteration makes code longer.
6. Iteration terminates when the loop-continuation condition fails whereas recursion terminates when a base case is recognized.
7. While using recursion multiple activation records are created on stack for each call where as in iteration everything is done in one activation record.
8. Infinite recursion can crash the system whereas infinite looping uses CPU cycles repeatedly.
9. Recursion uses selection structure whereas iteration uses repetetion structure.

### Types of Recursion:

Recursion is of two types depending on whether a function calls itself from within itself or whether two functions call one another mutually. The former is called **direct recursion** and the later is called **indirect recursion**. Thus there are two types of recursion:

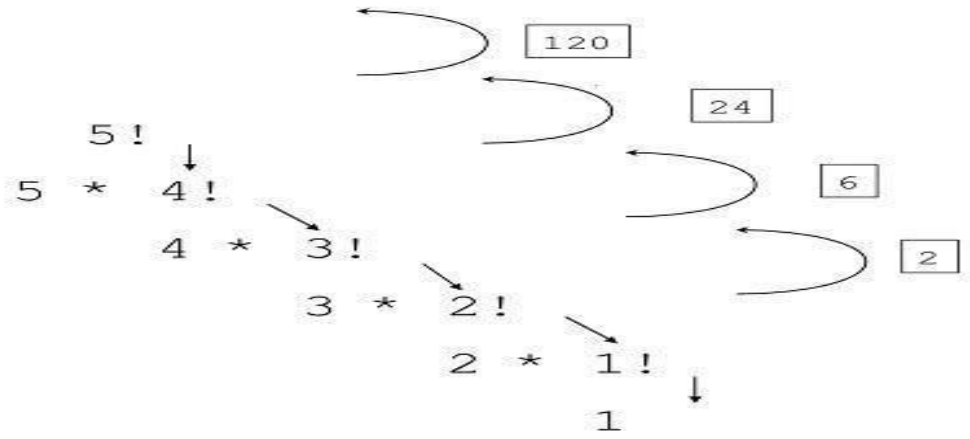
- Direct Recursion
- Indirect Recursion

Recursion may be further categorized as:

- Linear Recursion
- Binary Recursion
- Multiple Recursion

### Linear Recursion:

It is the most common type of Recursion in which function calls itself repeatedly until base condition [termination case] is reached. Once the base case is reached the results are return to the caller function. If a recursive function is called only once then it is called a linear recursion.




### Binary Recursion:

Some recursive functions don't just have one call to themselves; they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.

Example1: The Fibonacci function fib provides a classic example of binary recursion. The Fibonacci numbers can be defined by the rule: fib(n) = 0 if n is 0,

= 1 if n is 1,

= fib(n-1) + fib(n-2) otherwise

For example, the first seven Fibonacci numbers are

Fib(0) = 0

Fib(1) = 1

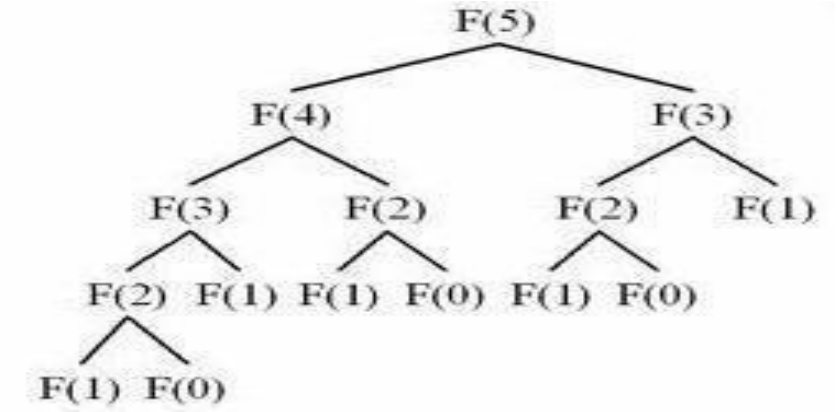
Fib(2) =Fib(1)+Fib(0) = 1

Fib(3) = Fib(2) + Fib(1) = 2

Fib(4) = Fib(3) + Fib(2) = 3

Fib(5) = Fib(4) + Fib(3) = 5

Fib(6) = Fib(5) + Fib(4) = 8




**Tail Recursion:**

Tail recursion is a form of linear recursion. In tail recursion, the recursive call is the last thing the function does. Often, the value of the recursive call is returned. As such, tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop, the same effect can generally be achieved. In fact, a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code.

A good example of a tail recursive function is a function to compute the GCD, or Greatest Common Denominator, of two numbers: def factorial(n): if n == 0: return 1 else: return factorial(n-1) * n def tail_factorial(n, accumulator=1):

if n == 0: return 1 else: return tail_factorial(n- 1,

accumulator * n)

**Recursive algorithms for Factorial, GCD, Fibonacci Series and Towers of Hanoi:**

**Factorial(n)**

Input: integer $n \geq 0$

Output: n!

1. If n = 0 then return (1)
2. else return prod(n, factorial(n − 1))

**GCD(m, n)**

Input: integers $m > 0$, $n \geq 0$

Output: gcd (m, n)

1. If n = 0 then return (m)
2. else return gcd(n,m mod n)

Time-Complexity: O(ln n)

**Fibonacci(n)**

Input: integer $n \geq 0$

Output: Fibonacci Series: 1 1 2 3 5 8 13……..

1. if n=1 or n=2
2. then Fibonacci(n)=1
3. else Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

**Towers of Hanoi**

Input: The aim of the tower of Hanoi problem is to move the initial n different sized disks from needle A to needle C using a temporary needle B. The rule is that no larger disk is to be placed above the smaller disk in any of the needle while moving or at any time, and only the top of the disk is to be moved at a time from any needle to any needle.

Output:

1. If n=1, move the single disk from A to C and return,
2. If n>1, move the top n-1 disks from A to B using C as temporary.
3. Move the remaining disk from A to C.
4. Move the n-1 disk disks from B to C, using A as temporary.