Short Notes:-

Abstract Data Type :-

Abstract Data Type is a useful tool for specifying the logical properties of a data type. Basically a data type is a collection of values and a set of operations on those values. That collection and those operations form a mathematical construct that may be implemented using a particular hardware or software data structure. The term ADT refers to the basic mathematical concept that defines the data type.

ADT thus refers to the mathematical concept that defines the data type and is not concerned with implementation i.e. space & time efficiency. ADT is a useful guideline to implementors and a useful tool to programmers who wish to use the data type correctly.

An ADT consists of 2 parts. 1) A Value definition 2) An Operator definition.

The value definition defines the collection of values for the ADT and consists of 2 parts
a) Abstract clause b) Condition clause.

In Operator definition, each operator is defined as an abstract function with 3 parts
a) Header b) Preconditions c) Postconditions

Definitions :-

1) **Null List –** A list with no nodes in it is called as empty list or null list.
2) **Multilist –** Multilist are the lists in which nodes can be shared among several lists.
3) **Shared List –** Same as above
4) **Dangling Pointers –** These are pointers that do not point to a valid object. They arise when an object is deleted or deallocated & the pointer value is not modified so that it still points to the memory location of the deallocated memory.
5) **Sub Lists** – A Generalized Link List is a special type of link list which not only contains links to individual nodes but also links (pointers) to other lists. These lists are called as sublists of GLL.
6) **Compound Node –** This node represents a set of compounds.
7) **3 Applications of Hash Table –** Address Book, Credit Card Authorization & Mapping Host Names

8) **Applications of Queue – a)** Queues are used in Computers for scheduling of resources to applications. These resources are CPU, Printer, etc. **b)** Multiple print jobs are arranged in FIFO manner in a print queue & given to the printer. **c)** In batch programming, multiple jobs are combined into a batch & executed in sequential manner (exception is priority queue).
9) **Applications of Stack –** a) Subroutine calls, recursion b) Interrupt Handling c) Interconversion between Infix, Postfix & Prefix expressions d) Matching parenthesis in an expression

Performance Analysis of Algorithms :-

In Computer Science, an algorithm is a finite set of instructions that, if followed, accomplishes a particular task. The vital criteria for judging/evaluating the performance of an algorithm are :-

1. ***Space Complexity*** – Deals with storage requirements
2. ***Time Complexity*** – Deals with computing time

Definition of Space Complexity – The space complexity of an algorithm is the amount of memory it needs to run till completion.

The space needed by an algorithm is the sum of the following 2 components :-

1. *Fixed Part* – This includes the space for simple variables, constants, etc.
2. *Variable Part* – This includes the space needed by variables whose size depends upon a particular problem.

e.g.

```
int sum(int a[], int n)
{
        int s = 0, i;

        for (i=0; i<n; i++)
                s = s + a[i];

        return (s);
}
```

Here i, s and n are fixed part but size of a[] is a variable part.

Time Complexity – This deals with the amount of time taken by a program for execution. To calculate the rough estimate of time required, we have to identify the *Active Operations* and *Book Keeping Operations*. Active operations are the one which are executed often e.g. statements in loop. Other operations like initializations, single print command (outside a loop), etc. are called as book keeping operations and as they do not execute as many times as the active operations, need not be computed.

The total number of times a statement gets executed is defined as its frequency count. *eg*.

```
for (i=1;i<=m;i++)
{
        for (j=1;j<=n;j++)
                a = a * b;
}
```

Here a=a*b executes m * n times which is it's frequency count.
If there are more than one active operations, the sum of frequency count yields a polynomial. However for determining the time complexity, only the largest magnitude is of concern & can be expressed using *"Big-O"* notation. *eg*.

$$\mathbf{f(n) = n^4 + 20n^3 + n + 40}$$

Here the term with the highest order n^4 determines the time complexity of the f(n) which is noted as $\mathbf{O(n^4)}$.

Other cases to consider during analysis of an algorithm :-

1) ***Performance of algorithm can also be analyzed on the basis of input***. For *eg*. in sorting, if the input list is already sorted, some sorting algorithms will perform very well but others may perform very poorly. The opposite may be true if the list is randomly arranged. Hence multiple input sets must be considered while analyzing an algorithm. These include:

a) *Best Case Input* b) *Worst Case Input* c) *Average Case Input*

2) ***Rate of Growth*** – While analyzing an algorithm, other than just considering the exact no. of operations performed by the algorithm, it is also important to analyze the rate of increase in operations as the size of the input increases.

To check this, some of the common *complexity functions* (running time functions) are :-

1.	Constant	f(n) = c
2.	Linear	f(n) = n
3.	Logarithmic	f(n) = log n
4.	Linear-Log	f(n) = n log n
5.	Quadratic	$f(n) = n^2$

6.	Cubic	$f(n) = n^3$
7.	Exponential	$f(n) = 2^n$

Following table shows the rate of growth for some of the common classes of algorithms for a wide range of input sizes.

n	**$\log_2 n$**	**$n\log_2 n$**	**n^2**	**n^3**	**2^n**
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	4294967296
64	6	384	4096	262144	Too Large

Thus the time and space complexity of an algorithm with "n" operations can be expressed as follows:-

1. The algorithms running time (growth) is at least as fast as some function or even faster.
2. The algorithms running time is of order of some function of n.
3. The algorithms running time is less than some function of n.

The above categories can be expressed using Asymptotic notations *Big Omega* **Ω**, *Big Theta* **θ** *and Big O* respectively. This notation is called "asymptotic" because it deals with the behaviour of functions in the limit i.e. for sufficiently large values of *"n"*.

A. <u>O Notation</u> (Big – Oh Notation)

This notation is used to denote upper bounds. It is expressed as O(f(n)) which means that the time taken never exceeds roughly f(n) operations. It is the most commonly used notation in calculating efficiency of an algorithm.

Defn :- The function f(n) = O(g(n)) [read as f of n is big oh of g of n] if there exists +ve constants c and n_0 such that

f(n) <= c * g(n) for all n, where $n >= n_0$

B. <u>Omega Notation</u> - Ω

This notation is used to denote lower bounds. It is written as

$T_n = \Omega(f(n))$

which indicates that T_n takes at least about f(n) operations.

Defn :- The function f(n) = Ω (g(n)) if there exists +ve constants c and n_0 such that

f(n) >= c * g(n) for all n, where $n >= n_0$

C. <u>Theta Notation</u> - θ

This notation denotes both upper and lower bounds of f(n). It is written as

$T_n = \theta(f(n))$

Defn :- The function f(n) = θ(g(n)) if there exists +ve constants c1, c2 and n_0 such that

c_1 * g(n) <= f(n) <= c_2 * g(n) for all n, where $n >= n_0$