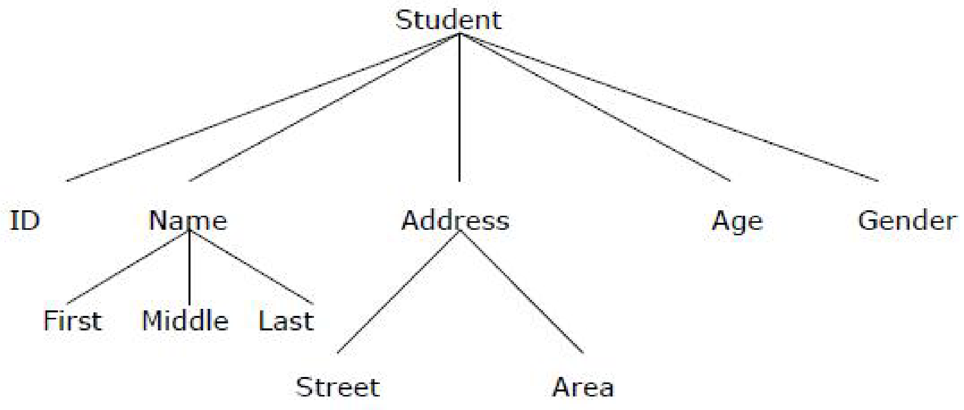
UNIT-1

1.1 BASIC TERMINOLOGY: ELEMENTARY DATA ORGANIZATION

1.1.1 Data and Data Item

Data are simply collection of facts and figures. Data are values or set of values. A data item refers to a single unit of values. Data items that are divided into sub items are group items; those that are not are called elementary items. For example, a student's name may be divided into three sub items – [first name, middle name and last name] but the ID of a student would normally be treated as a single item.



In the above example (ID, Age, Gender, First, Middle, Last, Street, Area) are elementary data items, whereas (Name, Address) are group data items.

1.1.2 Data Type

Data type is a classification identifying one of various types of data, such as floating-point, integer, or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; and the way values of that type can be stored. It is of two types: Primitive and non-primitive data type. Primitive data type is the basic data type that is provided by the programming language with built-in support. This data type is native to the language and is supported by machine directly while non-primitive data type is derived from primitive data type. For example- array, structure etc.

1.1.3 Variable

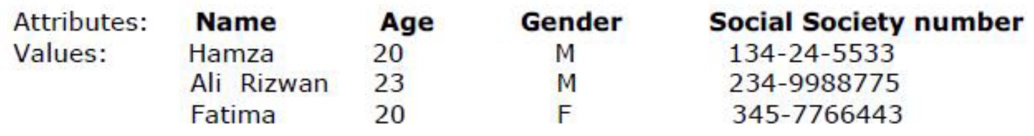
It is a symbolic name given to some known or unknown quantity or information, for the purpose of allowing the name to be used independently of the information it represents. A variable name in computer source code is usually associated with a data storage location and thus also its contents and these may change during the course of program execution.

1.1.4 Record

Collection of related data items is known as record. The elements of records are usually called fields or members. Records are distinguished from arrays by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type.

1.1.5 Program
A sequence of instructions that a computer can interpret and execute is termed as program.

1.1.6 Entity
An entity is something that has certain attributes or properties which may be assigned some values. The values themselves may be either numeric or non-numeric.
Example:



Attributes:	**Name**	**Age**	**Gender**	**Social Society number**
Values:	Hamza	20	M	134-24-5533
	Ali Rizwan	23	M	234-9988775
	Fatima	20	F	345-7766443

1.1.7 Entity Set
An entity set is a group of or set of similar entities. For example, employees of an organization, students of a class etc. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute. The term *"information"* is sometimes used for data with given attributes, of, in other words meaningful or processed data.

1.1.8 Field
A field is a single elementary unit of information representing an attribute of an entity, a record is the collection of field values of a given entity and a file is the collection of records of the entities in a given entity set.

1.1.9 File
File is a collection of records of the entities in a given entity set. For example, file containing records of students of a particular class.

1.1.10 Key
A key is one or more field(s) in a record that take(s) unique values and can be used to distinguish one record from the others.

1.2 ALGORITHM
A well-defined computational procedure that takes some value, or a set of values, as input and produces some value, or a set of values, as output. It can also be defined as sequence of computational steps that transform the input into the output.

An algorithm can be expressed in three ways:-
(i) in any natural language such as English, called pseudo code.
(ii) in a programming language or
(iii) in the form of a flowchart.

1.3 EFFICIENCY OF AN ALGORITHM
In computer science, algorithmic efficiency are the properties of an algorithm which relate to the amount of resources used by the algorithm. An algorithm must be analyzed to determine its

resource usage. Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process.

For maximum efficiency we wish to minimize resource usage. However, the various resources (e.g. time, space) can not be compared directly, so which of two algorithms is considered to be more efficient often depends on which measure of efficiency is being considered as the most important, e.g. is the requirement for high speed, or for minimum memory usage, or for some other measure. It can be of various types:

- Worst case efficiency: It is the maximum number of steps that an algorithm can take for any collection of data values.
- Best case efficiency: It is the minimum number of steps that an algorithm can take any collection of data values.
- Average case efficiency: It can be defined as
  - the efficiency averaged on all possible inputs
  - must assume a distribution of the input
  - We normally assume uniform distribution (all keys are equally probable)

If the input has size *n*, efficiency will be a function *of n*

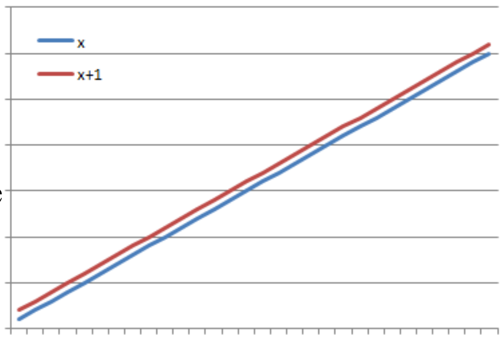
1.4 TIME AND SPACE COMPLEXITY

Complexity of algorithm is a function of size of input of a given problem instance which determines how much running time/memory space is needed by the algorithm in order to run to completion.
Time Complexity: Time complexity of an algorithm is the amount of time it needs in order to run to completion.
Space Complexity: Space Complexity of an algorithm is the amount of space it needs in order to run to completion.
There are two points which we should consider about computer programming:-
(i) An appropriate data structure and
(ii) An appropriate algorithm.

1.5 ASYMPTOTIC NOTATIONS

1.5.1 Asymptotic

It means a line that continually approaches a given curve but does not meet it at any finite distance.



Example

x is asymptotic with x + 1 as shown in graph.

Asymptotic may also be defined as a way to describe the behavior of functions in the limit or without bounds.

Let f(x) and g(x) be functions from the set of real numbers to the set of real numbers.

We say that f and g are asymptotic and write f(x) ≈ g(x) if

$\lim_{x \to \infty}$ f(x) / g(x) = *c* (constant)

1.5.2 Asymptotic Notations

1.7.2.1 Big-Oh Notation (O)

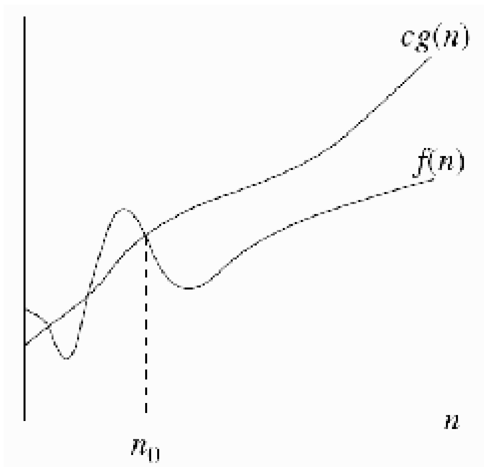
It provides possibly asymptotically tight **upper** bound for f(n) and it does not give best case complexity but can give worst case complexity.

Let f be a nonnegative function. Then we define the three most common asymptotic bounds as follows.

We say that f(n) is Big-O of g(n), written as f(n) = O(g(n)), iff there are positive constants c and n0 such that

$$0 \le f(n) \le c\, g(n) \text{ for all } n \ge n0$$

If f(n) = O(g(n)), we say that g(n) is an upper bound on f(n).



Example - $n^2 + 50n = O(n^2)$

$0 \le h(n) \le cg(n)$

$0 \le n^2 + 50n \le cn^2$

$0/n^2 \le n^2/n^2 + 50n/n^2 \le cn^2/n^2$ Divide by n^2

$0 \le 1 + 50/n \le c$ Note that $50/n \to 0$ as $n \to \infty$

	Pick n = 50
$0 \leq 1 + 50/50 = 2 \leq c = 2$	With c=2
$0 \leq 1 + 50/n_0 \leq 2$	Find n_0
$-1 \leq 50/n_0 \leq 1$	
$-20n_0 \leq 50 \leq n_0 = 50$	n_0=50
$0 \leq n^2 + 50n \leq 2n^2$	$\forall\ n \geq n_0=50, c=2$

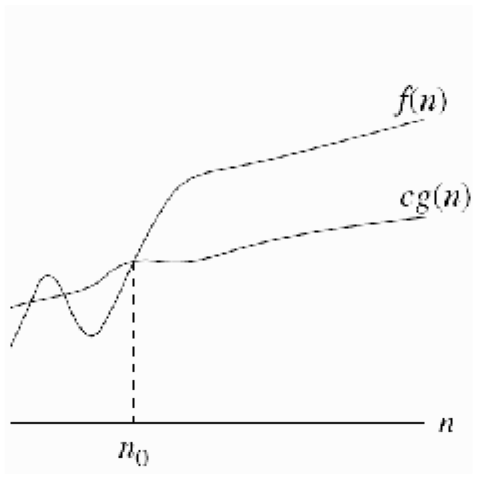
1.7.2.2 Big-Omega Notation (Ω)

It provides possibly asymptotically tight **lower** bound for f(n) and it does not give worst case complexity but can give best case complexity

f(n) is said to be Big-Omega of g(n), written as f(n) = Ω(g(n)), iff there are positive constants c and n0 such that

$$0 \leq c\ g(n) \leq f(n) \text{ for all } n \geq n0$$

If f(n) = Ω(g(n)), we say that g(n) is a lower bound on f(n).



Example - $n^3 = \Omega(n^2)$ with c=1 and n_0=1

$0 \leq cg(n) \leq h(n)$

$0 \leq 1*1^2 = 1 \leq 1 = 1^3$

$0 \leq cg(n) \leq h(n)$

$0 \le cn^2 \le n^3$

$0/n^2 \le cn^2/n^2 \le n^3/n^2$

$0 \le c \le n$

$0 \le 1 \le 1$ with c=1 and n_0=1 since n increases.

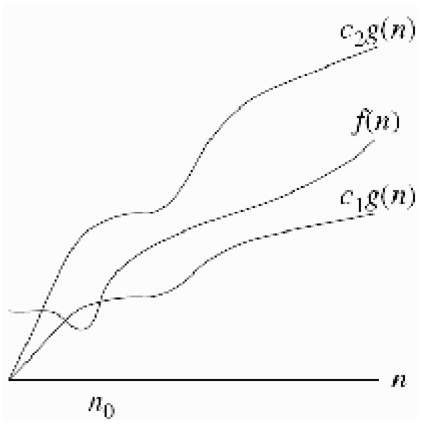
$$\lim_{n \to \infty} n = \infty$$

1.7.2.3 Big-Theta Notation (Θ)

• We say that f(n) is Big-Theta of g(n), written as f(n) = Θ(g(n)), iff there are positive constants c1, c2 and n0 such that

$$0 \le c1\ g(n) \le f(n) \le c2\ g(n) \text{ for all } n \ge n0$$

Equivalently, f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)). If f(n) = Θ(g(n)), we say that g(n) is a tight bound on f(n).



Example - $n^2/2-2n = \Theta(n^2)$

$c_1g(n) \le h(n) \le c_2g(n)$

$c_1n^2 \le n^2/2-2n \le c_2n^2$

$c_1n^2/n^2 \le n^2/2n^2-2n/n^2 \le c_2n^2/n^2$ Divide by n^2

$c_1 \le 1/2-2/n \le c_2$

O	Determine $c_2 = ½$	
	$½-2/n \le c_2 = ½$	$\lim_{n\to\infty} ½-2/n = ½$ maximum of $½-2/n$
Ω	Determine $c_1 = 1/10$	
	$0 < c_1 \le ½-2/n$	$0 < c_1$ minimum when $n=5$
	$0 < c_1 \le ½-2/5$	
	$0 < c_1 \le 5/10-4/10 = 1/10$	
n_0	Determine $n_0 = 5$	
	$c_1 \le ½-2/n_0 \le c_2$	
	$1/10 \le ½-2/n_0 \le ½$	
	$1/10-½ \le -2/n_0 \le 0$	Subtract ½
	$-4/10 \le -2/n_0 \le 0$	
	$-4/10\ n_0 \le -2 \le 0$	Multiply by n_0
	$-n_0 \le -2*10/4 \le 0$	Multiply by 10/4
	$n_0 \ge 2*10/4 \ge 0$	Multiply by -1
	$n_0 \ge 5 \ge 0$	
	$n_0 \ge 5$	$n_0 = 5$ satisfies
Θ	$0 < c_1n^2 \le n^2/2-2n \le c_2n^2 \quad \forall n \ge n_0$	with $c_1=1/10$, $c_2=½$ and $n_0=5$

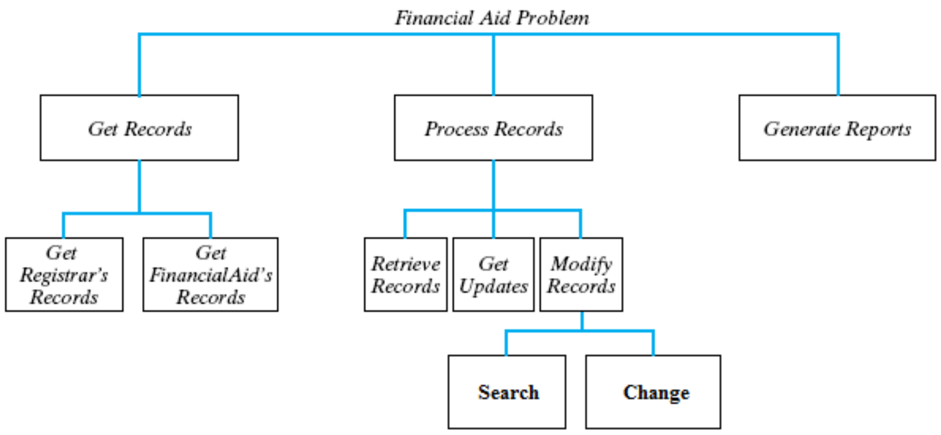
1.5.3 Time Space Trade-off

The best algorithm to solve a given problem is one that requires less memory space and less time to run to completion. But in practice, it is not always possible to obtain both of these objectives. One algorithm may require less memory space but may take more time to complete its execution. On the other hand, the other algorithm may require more memory space but may take less time to run to completion. Thus, we have to sacrifice one at the cost of other. In other words, there is Space-Time trade-off between algorithms.

If we need an algorithm that requires less memory space, then we choose the first algorithm at the cost of more execution time. On the other hand if we need an algorithm that requires less time for execution, then we choose the second algorithm at the cost of more memory space.

1.6 ABSTRACT DATA TYPE

It can be defined as a collection of data items together with the operations on the data. The word "abstract" refers to the fact that the data and the basic operations defined on it are being studied independently of how they are implemented. It involves **what** can be done with the data, not **how** has to be done. For ex, in the below figure the user would be involved in checking that what can be done with the data collected not how it has to be done.



An implementation of ADT consists of storage structures to store the data items and algorithms for basic operation. All the data structures i.e. array, linked list, stack, queue etc are examples of ADT.

1.7 DATA STRUCTURE

In computer science, a data structure is a particular way of storing and organizing data in a computer's memory so that it can be used efficiently. Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a data structure. The choice of a particular data model depends on the two considerations first; it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data whenever necessary.

1.7.1 Need of data structure

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.

- Provide fast searching and sorting of data.

1.7.2 Selecting a data structure
Selection of suitable data structure involve following steps –

- Analyze the problem to determine the resource constraints a solution must meet.
- Determine basic operation that must be supported. Quantify resource constraint for each operation
- Select the data structure that best meets these requirements.
- Each data structure has cost and benefits. Rarely is one data structure better than other in all situations. A data structure require :
    - ➢ Space for each item it stores
    - ➢ Time to perform each basic operation
    - ➢ Programming effort.

Each problem has constraints on available time and space. Best data structure for the task requires careful analysis of problem characteristics.

1.7.3 Type of data structure

1.7.3.1 Static data structure
A data structure whose organizational characteristics are invariant throughout its lifetime. Such structures are well supported by high-level languages and familiar examples are arrays and records. The prime features of static structures are
(a) None of the structural information need be stored explicitly within the elements – it is often held in a distinct logical/physical header;
(b) The elements of an allocated structure are physically contiguous, held in a single segment of memory;
(c) All descriptive information, other than the physical location of the allocated structure, is determined by the structure definition;
(d) Relationships between elements do not change during the lifetime of the structure.

1.7.3.2 Dynamic data structure
A data structure whose organizational characteristics may change during its lifetime. The adaptability afforded by such structures, e.g. linked lists, is often at the expense of decreased efficiency in accessing elements of the structure. Two main features distinguish dynamic structures from static data structures. Firstly, it is no longer possible to infer all structural information from a header; each data element will have to contain information relating it logically to other elements of the structure. Secondly, using a single block of contiguous storage is often not appropriate, and hence it is necessary to provide some storage management scheme at run-time.

1.7.3.3 Linear Data Structure
A data structure is said to be linear if its elements form any sequence. There are basically two ways of representing such linear structure in memory.
a) One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays**.**

b) The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists**.**
The common examples of linear data structure are arrays, queues, stacks and linked lists.

1.7.3.4 Non-linear Data Structure
This structure is mainly used to represent data containing a hierarchical relationship between elements. E.g. graphs, family trees and table of contents.

1.9 A BRIEF DESCRIPTION OF DATA STRUCTURES

1.8.1 Array

The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number *n* of similar data referenced respectively by a set of *n* consecutive numbers, usually 1, 2, 3 . . . . *n*. if we choose the name **A** for the array, then the elements of **A** are denoted by subscript notation

$$\mathbf{A}\ 1, \mathbf{A}\ 2, \mathbf{A}\ 3 \ldots \mathbf{A}\ n$$

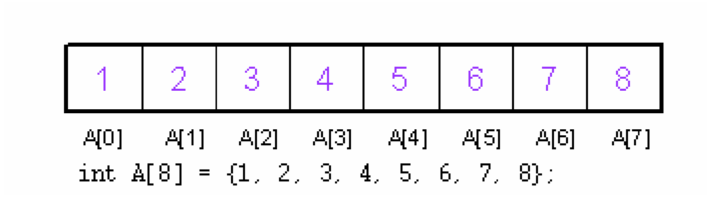
or by the parenthesis notation

$$A\ (1), A\ (2), A\ (3) \ldots\ldots A\ (n)$$

or by the bracket notation

$$A\ [1], A\ [2], A\ [3] \ldots\ldots A\ [n]$$

Example:
A linear array **A[8]** consisting of numbers is pictured in following figure.

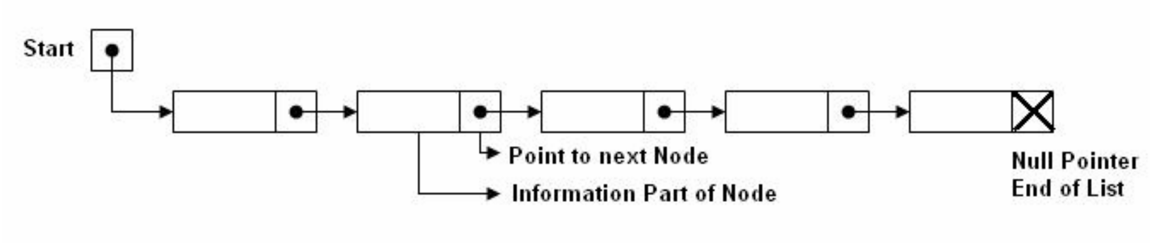


1.8.2 Linked List

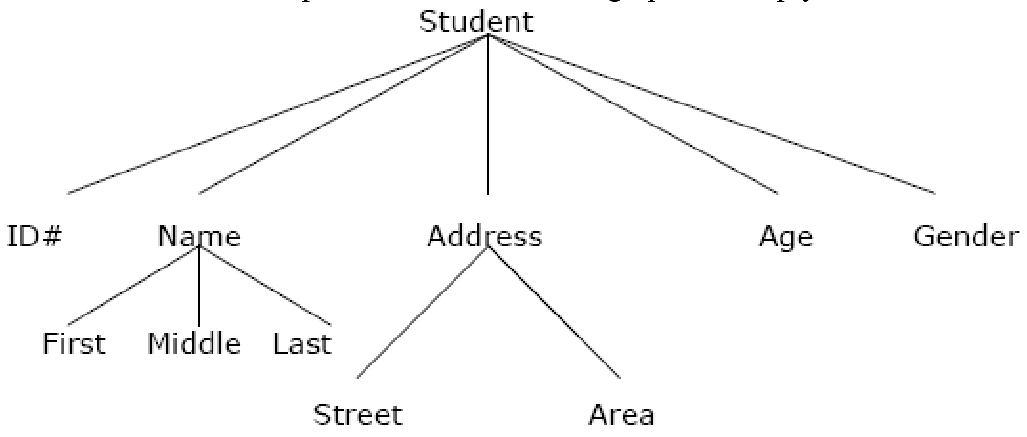
A linked list or one way list is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. Each node is divided into two parts:

- The first part contains the information of the element/node
- The second part contains the address of the next node (link /next pointer field) in the list.

There is a special pointer Start/List contains the address of first node in the list. If this special pointer contains null, means that List is empty.
Example:



1.8.3 Tree
Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree graph or, simply, a tree.



1.8.4 Graph
Data sometimes contains a relationship between pairs of elements which is not necessarily hierarchical in nature, e.g. an airline flights only between the cities connected by lines. This data structure is called Graph.

1.8.5 Queue
A queue, also called FIFO system, is a linear list in which deletions can take place only at one end of the list, the Font of the list and insertion can take place only at the other end Rear.

1.8.6 Stack
It is an ordered group of homogeneous items of elements. Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack). The last element to be added is the first to be removed (LIFO: Last In, First Out).

1.9 DATA STRUCTURES OPERATIONS
The data appearing in our data structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely in the frequency with which specific operations are performed.
The following four operations play a major role in this text:

- **Traversing:** accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called “visiting” the record.)
- **Searching:** Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.
- **Inserting:** Adding a new node/record to the structure.
- **Deleting:** Removing a node/record from the structure.

[redacted]