Chapter 01
Basic of Data Structures

1. Data structure- Definition:

- Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a data structure.
- Data structure is the structural representation of logical relationships between elements of data. In other words a data structure is a way of organizing data items by considering its relationship to each other.
- Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of associativity. Data structure affects the design of both the structural and functional aspects of a program.

Algorithm + Data Structure = Program

- Data structures are the building blocks of a program; here the selection of a particular data structure will help the programmer to design more efficient programs as the complexity and volume of the problems solved by the computer is steadily increasing day by day.

The representation of a particular data structure in the memory of a computer is called a storage structure. That is, a data structure should be represented in such a way that it utilizes maximum efficiency. The data structure can be represented in both main and auxiliary memory of the computer. A storage structure representation in auxiliary memory is often called a file structure.

It is clear from the above discussion that the data structure and the operations on organized data items can integrally solve the problem using a computer

Data structure = Organized data + Operations

2. Types of data structures:

- Data structures are generally classified into primitive and non-primitive data structures. Basic data types such as integer, real, character and Boolean are known as primitive data structures. These data types consist of characters that cannot be divided, and hence they are also called simple data types.
- The non-primitive data structure is the processing of complex numbers, very few computer are capable of doing arithmetic on complex numbers, linked-

list, stacks, queue, trees and graphs are example of non-primitive data structures.

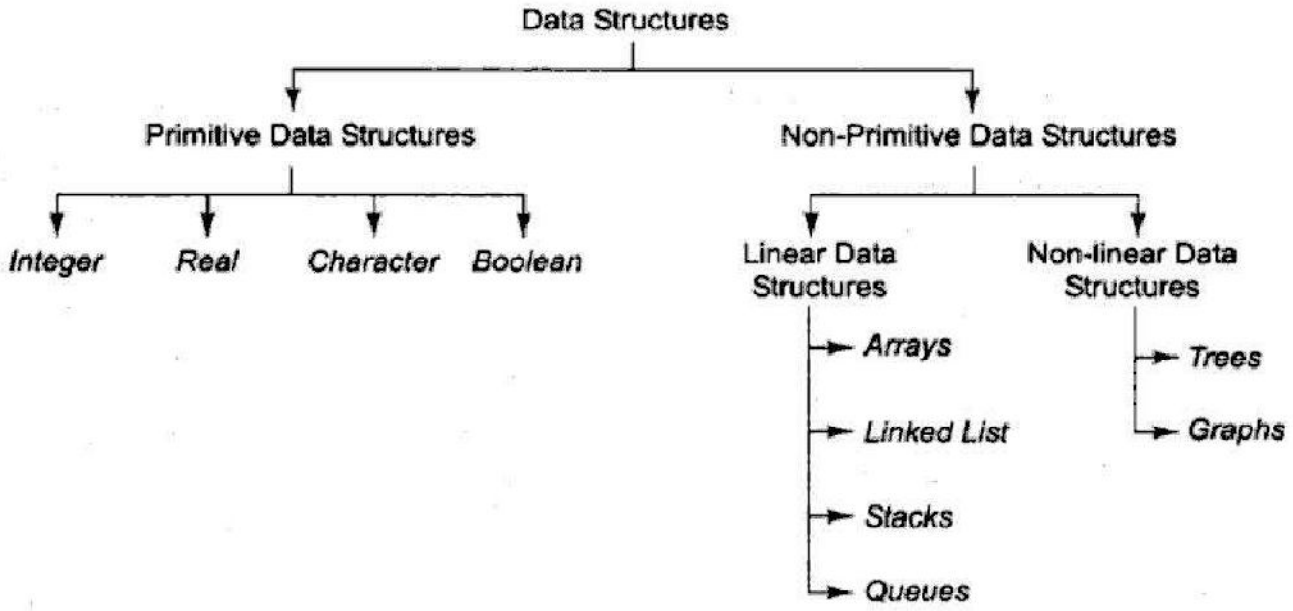


Fig. 1.1 Classification of Data Structures

- **Primitive data structures:**
  - **Integer:** An integer is the basic data types which is commonly used for storing negative as well as non-negative integer numbers. Example 38, -38

  - **Real Number:** The method used to represent real numbers in computer is floating-point notation. In this notation, the real number is represented by a number called a mantissa, times a base raised to an integer power called an exponent. Example 20952 X 10^{-2} mantissa is 20952 and exponent is -2.

  - **Character:** Some data are storing character or string form. There are different codes available to store data in character form such as BCD, EDCDIC and ASCII.

  - **Boolean:** Some data are represent in only two way like true-false, on-off or flag set-flag reset. These kinds of data are considering Boolean type.

Based on the structure and arrangement of data, non-primitive data structures are further classified into linear and non-linear.

- **Linear Data Structure:**

A data structure is said to be linear if its elements from a sequence or a linear list. In linear data structures, the data is arranged in a linear fashion although the way they stored in memory need not be sequential. Arrays, linked lists, stacks and queues are example of linear data structures.

Arrays: Linear array mean a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers, usually 0,1,2,…..,n.

Example 1.2

A linear array STUDENT consisting of the names of six students is pictured in Fig. 1.2. Here STUDENT[1] denotes John Brown, STUDENT[2] denotes Sandra Gold, and so on.

Linear arrays are called one-dimensional arrays because each element in such an array is referenced by one subscript. A *two-dimensional array* is a collection of similar data elements where each element is referenced by two subscripts. (Such arrays are called *matrices* in mathematics, and *tables* in business applications.) Multidimensional arrays are defined analogously.

	STUDENT
1	John Brown
2	Sandra Gold
3	Tom Jones
4	June Kelly
5	Mary Reed
6	Alan Smith

Example 1.3

A chain of 28 stores, each store having 4 departments, may list its weekly sales (to the nearest dollar) as in Fig. 1.3. Such data can be stored in the computer using a two-dimensional array in which the first subscript denotes the store and the second subscript the department. If SALES is the name given to the array, then

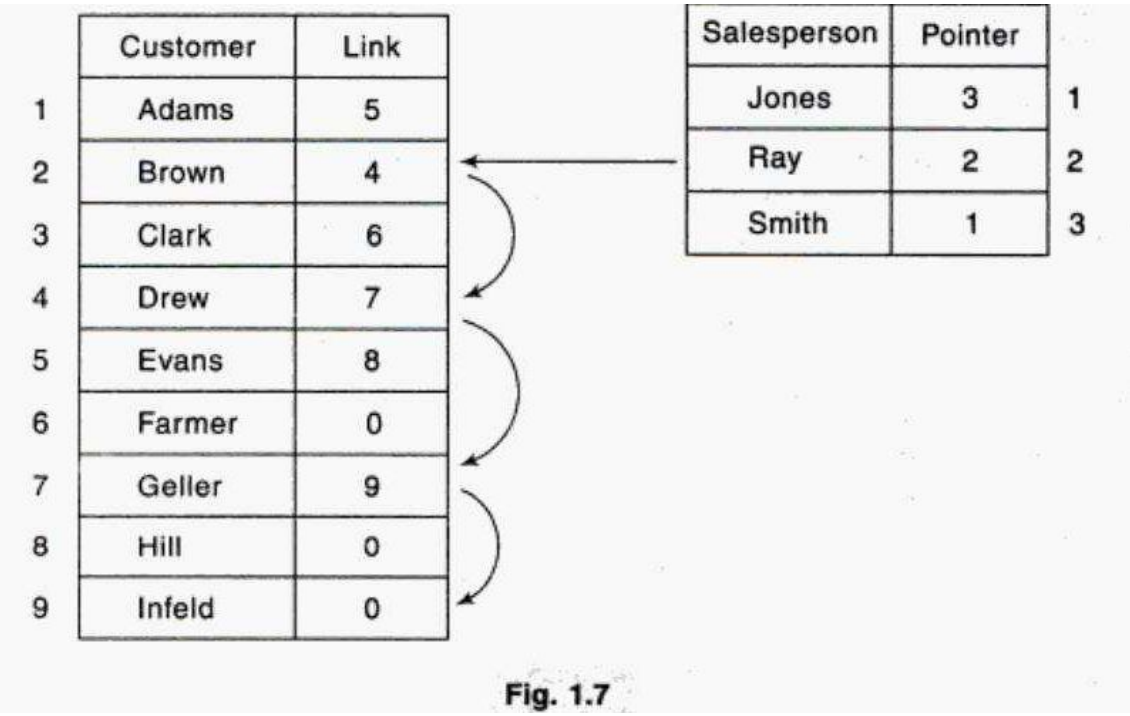
SALES[1, 1] = 2872, SALES[1, 2] = 805, SALES[1, 3] = 3211, ..., SALES[28, 4] = 982

Store \ Dept.	1	2	3	4
1	2872	805	3211	1560
2	2196	1223	2525	1744
3	3257	1017	3686	1951
. . .	. . .	. . .	. . .	. . .
28	2618	931	2333	982

Fig. 1.3

The size of this array is denoted by 28 × 4 (read 28 by 4), since it contains 28 *rows* (the horizontal lines of numbers) and 4 *columns* (the vertical lines of numbers).

Linked List: Consider following fig 1.7 is example of linked lists. Although the terms "pointer" and "link" are usually used synonymously. The term "pointer" when an element in one list points to an element in a different list and to reverse the term "link" for the case when an element in a list points to an element in that same list.

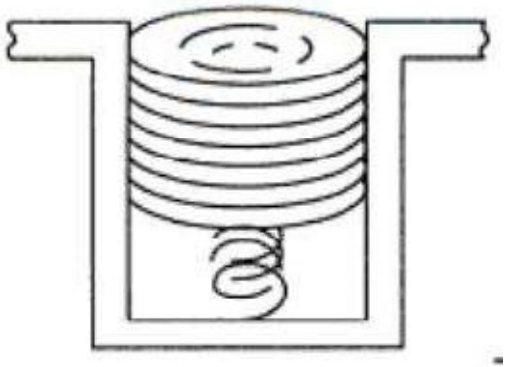


	Customer	Link
1	Adams	5
2	Brown	4
3	Clark	6
4	Drew	7
5	Evans	8
6	Farmer	0
7	Geller	9
8	Hill	0
9	Infeld	0

Salesperson	Pointer	
Jones	3	1
Ray	2	2
Smith	1	3

Fig. 1.7

Stacks: A stack also called last-in first-out (LIFO) system, is a linear list in which insertions and deletions can be take place only at one end, called the top. This structure is similar in its operation to a stack of dishes on a spring system. Note that new dishes are inserted only at the stack and dishes can be deleted only from the top of the stack.



(a) Stack of dishes

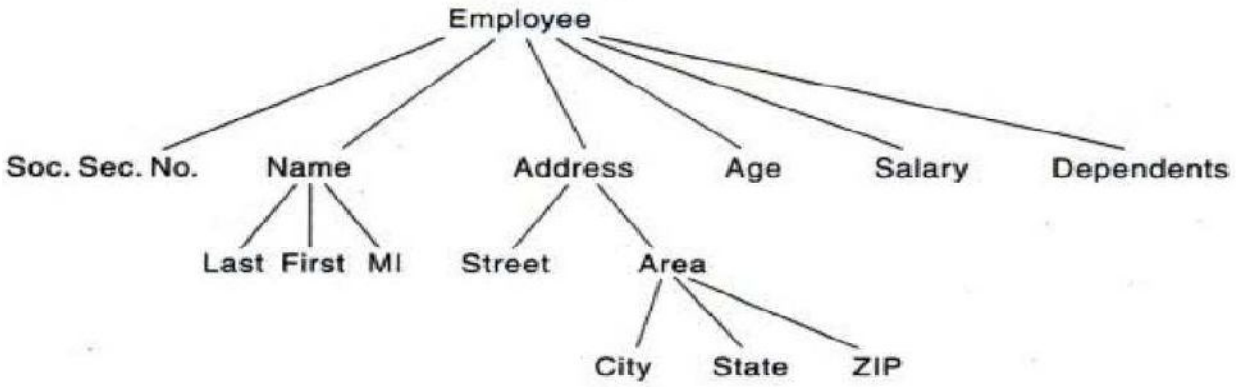
Queue: A queue also called a first-in first-out (FIFO) system, is a linear list in which deletion can be take place only at one end of the list, the "front" of the list and insertions can take place only at the other end of the list, the "rear" of the list. This structure operates in much the same way as a line of people waiting at the bus stop. Another analogy is which automobiles waiting to pass through an insertion-the first car in the first car through.

(b) Queue waiting for a bus

- **Non-linear data structure:**

A non-linear data structure means they not arranged in sequence. The insertion and deletion of data is therefore not possible in a linear fashion. Trees and graphs are examples of non-linear data structures.

Trees: 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.



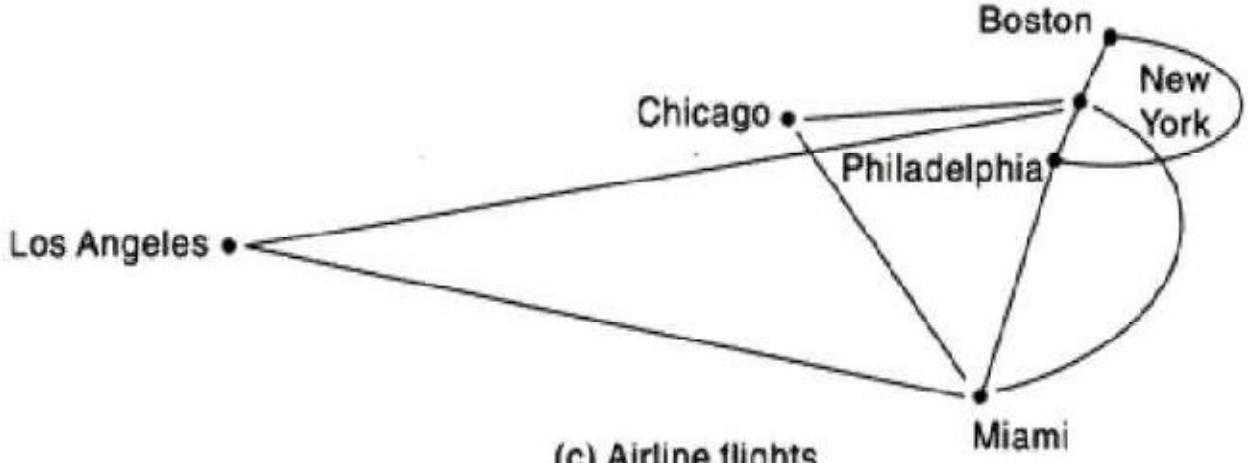
(a)

```
01 Employee
     02 Social Security Number
     02 Name
          03 Last
          03 First
          03 Middle Initial
     02 Address
          03 Street
          03 Area
               04 City
               04 State
               04 ZIP
     02 Age
     02 Salary
     02 Dependents
```

(b)

Fig. 1.8

Graph: Data sometimes contain a relationship between pair of element which is not necessarily hierarchical in nature. For example, suppose an airline flies only between the cities connected by lines. The data structure which reflects this type of relationship is called a graph.



(c) Airline flights

3. Data Structure 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 on the frequency with which specific operations are performed. This section introduces the reader to some of the most frequently used of these operations.

The following four operations play a major role in this text:

1. *Traversing:* Accessing each record exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "visiting" the record.)
2. *Searching:* Finding the location of the record with a given key value, or finding the locations of all records which satisfy one or more conditions.
3. *Inserting:* Adding a new record to the structure.
4. *Deleting:* Removing a record from the structure.

Sometimes two or more of the operations may be used in a given situation; e.g., we may want to delete the record with a given key, which may mean we first need to search for the location of the record.

The following two operations, which are used in special situations, will also be considered:

1. *Sorting:* Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as social security number or account number)
2. *Merging:* Combining the records in two different sorted files into a single sorted file

4. Algorithms: Complexity, Time and Space complexity:

The term '**algorithm**' refers to the sequence of instructions that must be followed to solve a problem. In other words, an algorithm is a logical representation of the instructions which should be executed to perform a meaningful task.

An algorithm has certain characteristics. These are as follows:

- Each instruction should be unique and concise.
- Each instruction should be relative in nature and should not be repeated infinitely.
- Repetition of same task(s) should be avoided.
- The result should be available to the user after the algorithm terminates.

Thus, an algorithm is any well defined computational procedure, along with a specified set of allowable inputs, that produce some value or set of values as output.

After an algorithm has been designed, its **efficiency** must be analyzed. This involves determining whether the algorithm is economical in the use of computer resources, i.e. CPU time and memory. The term used to refer to the memory required by an algorithm is **memory space** and the term used to refer to the computational time is the **running time**.

The importance of efficiency of an algorithm is in the **correctness**—that is, does it always produce the correct result, and **program complexity** which considers both the difficulty of implementing an algorithm along with its efficiency.

Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computational problem.

That is, in practice to solve any complex real life problems; first we have to define the problems. Second step is to design the algorithm to solve that problem.

Writing and executing programs and then optimizing them may be effective for small programs. Optimization of a program is directly concerned with algorithm design. But for a large program, each part of the program must be well organized before writing the program. There are few steps of refinement involved when a problem is converted to program; this method is called *stepwise refinement method.* There are two approaches for algorithm design; they are *top-down* and *bottom-up* algorithm design.

We can write an informal algorithm, if we have an appropriate mathematical model for a problem. The initial version of the algorithm will contain general statements, *i.e.*,; informal instructions. Then we convert this informal algorithm to formal algorithm, that is, more definite instructions by applying any programming language syntax and semantics partially. Finally a program can be developed by converting the formal algorithm by a programming language manual.

From the above discussion we have understood that there are several steps to reach a program from a mathematical model. In every step there is a refinement (or conversion). That is to convert an informal algorithm to a program, we must go through several stages of formalization until we arrive at a program — whose meaning is formally defined by a programming language manual — is called stepwise refinement techniques.

There are three steps in refinement process, which is illustrated in Fig. 1.1.

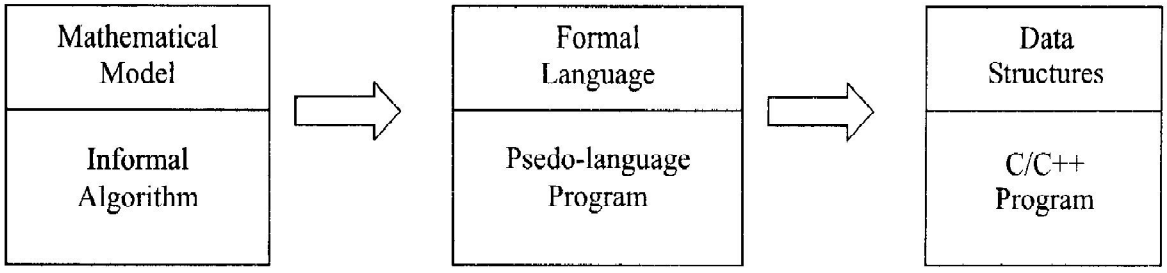


Fig. 1.1

1. In the first stage, modeling, we try to represent the problem using an appropriate mathematical model such as a graph, tree etc. At this stage, the solution to the problem is an algorithm expressed very informally.

2. At the next stage, the algorithm is written in pseudo-language (or formal algorithm) that is, a mixture of any programming language constructs and less formal English statements. The operations to be performed on the various types of data become fixed.

3. In the final stage we choose an implementation for each abstract data type and

write the procedures for the various operations on that type. The remaining informal statements in the pseudo-language algorithm are replaced by (or any programming language) C/C++ code.

Complexity:

When we talk of complexity in context of computers, we call it **computational complexity.** Computational complexity is a characterization of the time or space requirements for solving a problem by a particular algorithm. These requirements are expressed in terms of a single parameter that represents the size of the problem.

Given a particular problem, say one of the simplification problems. Let 'n' denote its size. The time required of a specific algorithm for solving this problem is expressed by a function:

$$f : R \rightarrow R$$

such that f(n) is the largest amount of time needed by the algorithm to solve the problem of size n. Function 'f' is usually called the time complexity function.

Thus, we conclude that the analysis of the program requires two main considerations:

- Time complexity
- Space complexity

The time complexity of a program/algorithm is the amount of computer time that it needs to run to completion. The space complexity of a program/algorithm is the amount of memory that it needs to run to completion.

Time Complexity

While measuring the time complexity of an algorithm, we concentrate on developing only the frequency count for all key statements (statements that are important and are the basic instructions of an algorithm). This is because, it is often difficult to get reliable timing figure because of clock limitations and the multiprogramming or the sharing environment.

Consider the algorithm given below:

ALGORITHM A	a = a + 1

ALGORITHM B	for x = 1 to n step 1 a = a + 1 Loop

ALGORITHM C	for x = 1 to n step 1 for y = 1 to n step 1 a = a + 1 Loop

In the algorithm A we may find that the statement a=a+1 is independent and is not contained within any loop. Therefore, the number of times this shall be executed is 1. We can say that the **frequency count** of algorithm A is **1**.

In the second algorithm, i.e. B, the key statement out of three statements is the assignment operation a = a+1. Because this statement is contained within a loop, the number of times it is executed is n, as the loop runs for n times. The frequency count for this algorithm is **n**.

According to the third algorithm, the frequency count for the statement a=a+1 is n^2 as the inner loop runs n times, each time the outer loop runs, the outer loop also runs for n times. n^2 is said to be different in increasing order of magnitude just like 1, 10, 100 depending upon the n. During the analysis of algorithm we shall be concerned with determining the order of magnitude of an algorithm. This means that we will determine only those statements which may have the greatest frequency count.

The following formulas are useful in counting the steps executed by an algorithm:

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

$$1^1 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

If an algorithm performs f(n) basic operations when the size of its input is n, then its total running time will be cf(n), where c is a constant that depends upon the algorithm, on the way it is programmed, and on the way the computer is used, but c does not depend on the size of the input.

Space Complexity

The space needed by the program is the sum of the following components:

Fixed space requirement This includes the instruction space, for simple variables, fixed size structured variables, and constants.

Variable space requirement This consists of space needed by structured variables whose size depends on particular instance of variables. It also includes the additional space required when the function uses recursion.

Abstract Data Type:

An abstract data type (ADT) refers to a set of data values and associated operations that are specified accurately, independent of any particular implementation. With an ADT, we know what a specific data type can do, but how it actually does it is hidden. In broader terms, the ADT consists of a set of definitions that allow us to use the functions while hiding the implementation.

The properties of an abstract data type are emphasized through the following examples.

An abstract data type can thus be further defined as a data declaration packaged together with the operations that are meaningful for the data type. In other words, we encapsulate the data and the operations on the data, and then we hide them from the user.

The user need not know the data structure to use the ADT. Considering Example 1.8, the application program should have no knowledge of the data structure. All references to and manipulation of the data in the queue must be handled through defined interfaces in the structure. Allowing the application program to directly reference the data structure is a common fault in many implementations. This prevents the ADT from being fully portable to other applications.

Abstract Data Type Model

A representation of the ADT model is shown in Fig. 1.13. Notice that there are two different parts of the ADT model—functions (public and private) and data structures. Both are contained within the ADT model itself, and do not come within the scope of the application program. On the other hand, data structures are available to all of the ADTs functions as required, and a function may call on any other function to accomplish its task. This means that data structures and functions are within the scope of each other.

Data are entered, accessed, modified and deleted through the external

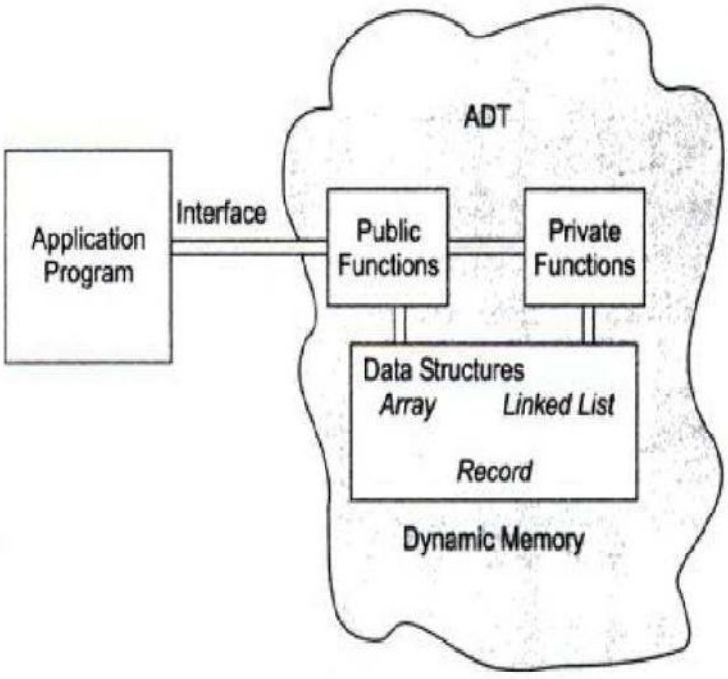


Fig. 1.13 ADT Model

application programming interface. This interface can only access the public functions. For each ADT operation, there is an algorithm that performs its specific task. The operation name and parameters are available to the application, and they provide the only interface to the application.

When a list is controlled entirely by the program, it is implemented using simple structures. Note that it is not enough if we just encapsulate the structure in an ADT, it is also necessary for multiple versions of the structure to coexist. Therefore, we must hide the implementation from the user, while being able to store different data at the same time.

Two basic structures, namely array and linked list, can be used to implement an ADT list.