[DATA STRUCTURES]

Chapter - 01 : "Introduction to Data Structures"

- **INTRODUCTION TO DATA STRUCTURES**

A **Data type** refers to a named group of data which share similar properties or characteristics and which have common behavior among them. Three fundamental data types used in C programming are **int** for integer values, **float** for floating-point numbers and **char** for character values.

But, sometimes a need arises to treat a group of different data types as a single unit. For example, a record in a file can have several fields of different data types and entire record may be required to be processed in one unit. In such a case, **Data structures** can be beneficial as data structures let you combine data of different types and process them together.

Def : ***A Data structure is a named group of data of different data types which can be processed as a single unit.***

Data structure is representation of the logical relationship existing between individual elements of data. In other words, a **Data structure** is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

Data structures are the building blocks of a program. And hence the selection of particular data structure stresses on the following two things :

1. The data structures must be rich enough in structure to reflect the relationship existing between the data.
2. And the structure should be simple so that we can process data effectively whenever required.

The identification of the inbuilt data structure is very important in nature. And the structure of input and output data can be use to derive the structure of a program. Data structures affects the design of both structural and functional aspects of a program.

Algorithm + Data structure = Program

We know that an ***algorithm*** *is a step-by-step procedure to solve a particular function i.e., it is a set of instructions written to carry out certain tasks and the data structure is the way of organizing the data with their logical relationship maintained.*

To develop a program of an algorithm, we should select an appropriate data structure for that algorithm. Therefore, algorithm and its associated data structures form a program.

CLASSIFICATION OF DATA STRUCTURE

Data structures are normally divided into two broad categories.

(i) Primitive Data Structures(built-in)

(ii) Non-Primitive Data Structures(user defined)

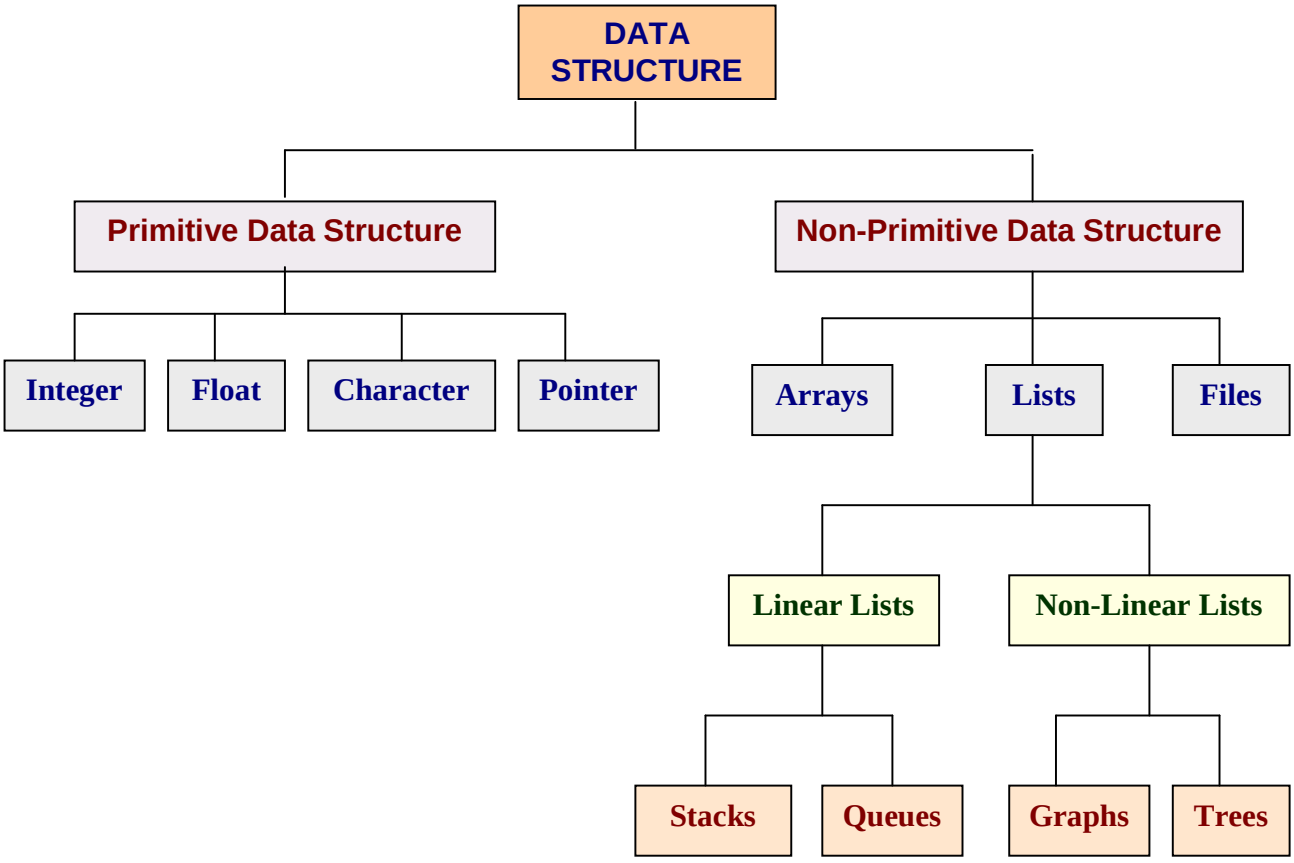

Fig.(1) : Classification of Data structure.

PRIMITIVE DATA STRUCTURES (BUILT-IN) :

These are basic structures and are directly operated upon by the machine instructions. These, in general, have different representations on different computers. **Integers, floating-point numbers, character constants, string constants, pointers** etc. fall in this category.

NON-PRIMITIVE DATA STRUCTURES (USER-DEFINED)

These are more complicated data structures. These are derived from the primitive data structures. The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. **Arrays, structures, lists and files** are examples.

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.

OPERATIONS OF DATA STRUCTURES

The basic **operations** that are performed on data structures are as follows :

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 :** Searching operation finds the presence of the desired data item in the list of data item. It may also find the locations of all elements that satisfy certain conditions.
3. **Inserting :** Inserting means addition of a new data element in a data structure.
4. **Deleting :** Deleting means removal of a data element from a data structure.

Sometimes, two or more of the operations may be used in a given situation. For e.g, we may want to delete a data element from a data structure, 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 :** Sorting is the process of arranging all data items in a data structure in a particular order say for example, either in ascending order or in descending order.
(2) **Merging :** Combining the records of two different sorted files into a single sorted file.

EXAMPLES & REAL LIFE APPLICATIONS

A company contains employee file in which each record contain the following data for a given employee :

Name, Address, Telephone number, Employee age, sex , employ number.

1. Suppose the company wants to announce a meeting through a mailing. Then one would traverse the file to obtain employee name and address for each member.

2. Suppose one wants to find the name of all members living in a certain area. Again one would traverse the file to obtain the data.
3. Suppose one wants to obtain address for the given employee name. Then one would search the file for the record containing employee name.
4. Suppose a new person joins the company. Then one would insert his or her record into the file.
5. Suppose an employee dies. Then one would delete his or her record from the file.
6. Suppose an employee has moved and has a new address and telephone number. Given the name of the member, one would first need to search for the record in the file. Then one would perform the "update"- i..e., change items in the record with the new data.
7. Suppose one wants to find the number of members 65 or older. Again one would traverse the file, counting such members.

■ DESCRIPTION OF VARIOUS DATA STRUCTURES

1. ARRAYS

An **array** is defined as a set of finite number of homogeneous elements or data items. It means an array can contain one type of data only, either all integers, all floating-point numbers, or all characters. Declaration of arrays is as follows :

```
int A[10];
```

where **int** specifies the data type or type of elements array stores. "**A**" is the name of the array, and the number specified inside the square brackets is the number of elements an array can store, this is also called **size** or length of array.

Some important concepts of **arrays** are :

(1) The individual element of an array can be accessed by specifying name of the array, followed by index or subscript (which is an integer number specifying the location of element in the array) inside square brackets. For example to access the fifth element of array a, we have to give the following statement :

```
A[4];
```

(2) The first element of the array has index zero [0]. It means the first element and last element of the above array will be specified as :

A[0], and A[9] respectively.

(3) The elements of array will always be stored in consecutive memory locations.

(4) The number of elements that can be stored in an array i.e., the size of array or its length is given by the following equation :

(upperbound – lowerbound) + 1

For the above array, it would be (9-0) + 1 = 10. Where 0 is the lower bound of array, and 9 is the upper bound of array.

(5) Arrays can always be read or written through loop. If we read a one-dimensional array, it requires one loop for reading and other for writing (printing) the array. For example :

(a) For **reading** the array

```
for ( i = 0; i < = 9 ; i++)
{
   scanf("%d", & A [ i ] );
}
```

(b) For **writing** the array

```
for ( i = 0; i < = 9 ; i++)
{
   printf("%d ", A [ i ] );
}
```

If we are reading or writing two-dimensional array it would require two loops. And similarly the array of n dimension would required **n** loops.

Some common **operations** performed on arrays are :

1. Creation of an array.
2. Traversing an array (accessing array elements).
3. Insertion of new elements.
4. Deletion of required element.
5. Modification of an element.
6. Merging of arrays.

2. LINKED LISTS

A linked list is a linear collection of data elements, called node pointing to the next nodes by means of pointers. Each **node** is divided into two parts : the first part containing the information of the element, and the second part called the link or next pointer containing the address of the next node in the list. Technically, each such element is referred to as a node, therefore a list can be defined as a collection of nodes as shown in Fig. (2) below :

START (START stores the address of first node)

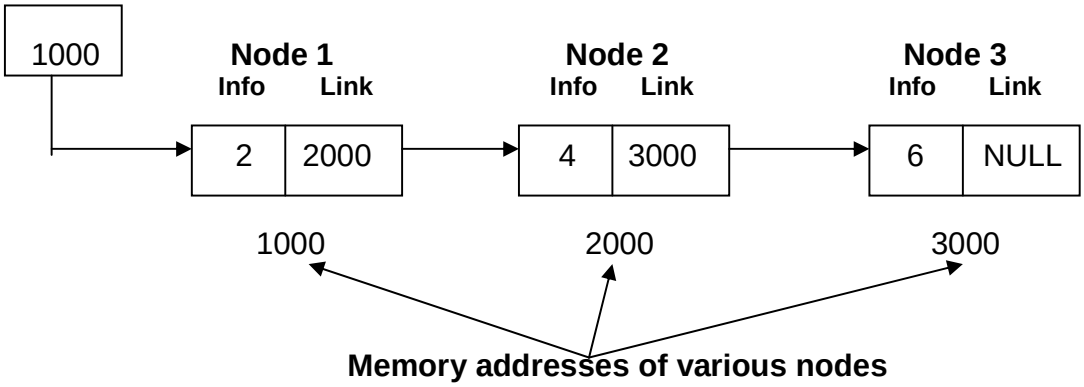

Fig. (2) Linked Lists

3. STACKS

A **stack** is a non-primitive linear data structure. It is an ordered list in which addition of new data item and deletion of already existing data item is done from only one end, known as *Top of Stack (TOS)*. As all the deletion and insertion in a stack is done from top of the stack, the last added element will be the first to be removed from the stack. Due to this reason, the stack is also called **Last-In-First-Out (LIFO)** type of list. Consider some examples,

- A common model of a stack is plates in a marriage party. Fresh plates are *"pushed"* onto the top and *"popped"* off the top.
- Some of you may eat biscuits. If you assume only one side of the cover is torn and biscuits are taken off one by one. This is called popping and similarly, if you want to preserve some biscuits for some time later, you will put them back into the pack through the same torn end called pushing.

4. QUEUES

Queue is a linear data structure that permits insertion of an element at one end and deletion of an element at the other end. The end at which the deletion of an element take place is called **front**, and the end at which insertion of a new element can take place is called **rear**. The deletion or insertion of elements can take place only at the front and rear end of the list respectively.

The first element that gets added into the queue is the first one to get removed from the list. Hence, Queue is also referred to as ***First-In-First-Out (FIFO)*** list. The name '*Queue*' comes from the everyday use of the term. Consider a railway reservation booth, at which we have to get into the reservation queue. New customers got into the queue from the rear end, whereas the customers who get their seats reserved leave the queue from the front end. It means the customers are serviced in the order in which they arrive the service center (i.e. first come first serve type of service). The same characteristics apply to our *Queue*. Fig. (3) shows the pictorial representation of a Queue.

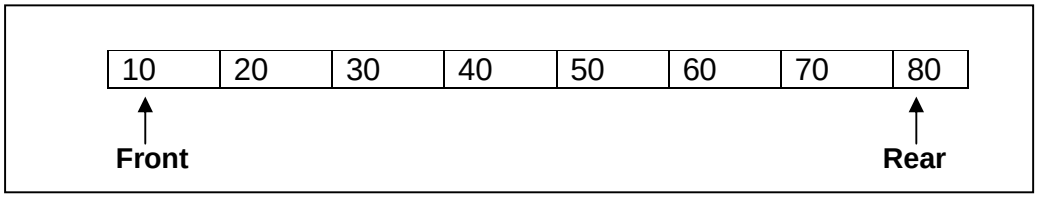

Fig. (3) : Pictorial representation of a Queue

In Fig (3), **10** is the first element and **80** is the last element added to the Queue. Similarly, **10** would be the first element to get removed and **80** would be the last element to get removed.

5. TREES

A **Tree** can be defined as a finite set of data items (nodes). **Tree** is a non-linear type of data structure in which data items are arranged of stored in a sorted sequence. Trees represent the hierarchical relationship between various elements. In trees :

1. There is a special data item at the top of hierarchy called the Root of the Tree.
2. The remaining data items are partitioned into number of mutually exclusive (i.e. disjoint) subsets, each of which is itself, a tree, which is called the subtree.
3. The tree always grows in length towards bottom in data structures, unlike natural trees which grows upwards.

The tree structure organizes the data into branches, which relate the information. A tree is shown in Fig. (4).

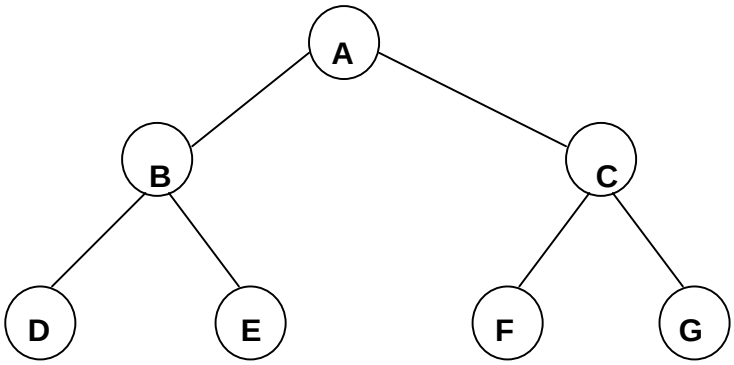

Fig. (4) : A Tree

6. GRAPHS

Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. Geometrical arrangement are very important while working with real life projects. For example, let us suppose there are five villages separated by a river. Now we want to construct bridges to connect these villages as shown in Fig. (5)

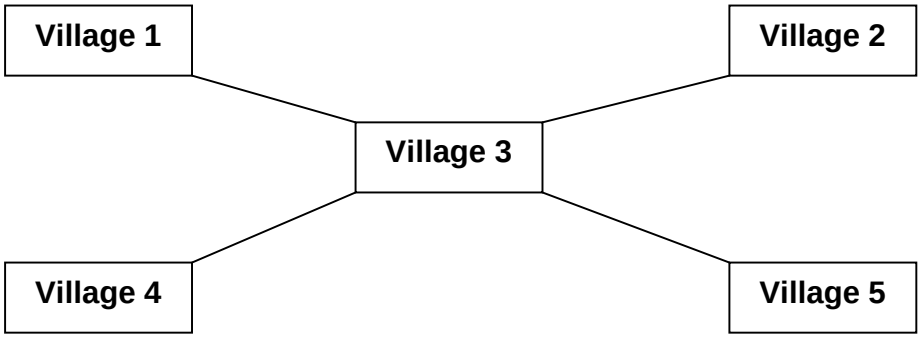

Fig. (5)

We can reduce the landmass of villages to a dot and we can change the shape of bridges. This will not change geometric arrangement of paths to connect different villages. We can draw a similar geometry of our project as follows :

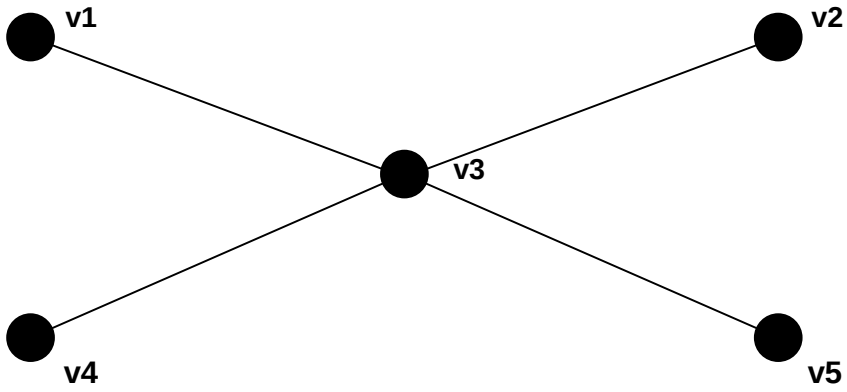

Fig. (6)

We have represented villages by dots (which are called vertex or node) and bridges by lines which are called edges. This type of drawing is called graph. Hence a graph can be defined as a ordered set (V,E), where V(G) represents the set of all elements called vertices and E(G) represents the edges between these vertices.

Fig. (6) shows a graph, for which V(G)={ v1, v2, v3, v4, v5 } and E(G) = { b1, b2, b3, b4 }.