What is Data Structure?

Whenever we want to work with large amount of data, then organizing that data is very important. If that data is not organized effectively, it is very difficult to perform any task on that data. If it is organized effectively then any operation can be performed easily on that data.

A data structure can be defined as follows...

Data structure is a method of organizing large amount of data more efficiently so that any operation on that data becomes easy

NOTE-
☀ Every data structure is used to organize the large amount of data
☀ Every data structure follows a particular principle
☀ The operations in a data structure should not violate the basic principle of that data structure.

Based on the organizing method of a data structure, data structures are divided into two types.

- Linear Data Structures
- Non - Linear Data Structures

Linear Data Structures
If a data structure is organizing the data in sequential order, then that data structure is called as Linear Data Structure.
i.e

Arrays
List (Linked List)
Stack
Queue

Non - Linear Data Structures

If a data structure is organizing the data in random order, then that data structure is called as Non-Linear Data Structure.
i.e

Tree
Graph
Dictionaries
Heaps
Tries, Etc.,

Operation used in Data Structure:

Data Structure Point Of View, Following Are Some Important Categories Of Algorithms

- **Search** − Algorithm To Search An Item In A Data Structure.
- **Sort** − Algorithm To Sort Items In A Certain Order.
- **Insert** − Algorithm To Insert Item In A Data Structure.
- **Update** − Algorithm To Update An Existing Item In A Data Structure.
- **Delete** − Algorithm To Delete An Existing Item From A Data Structure.

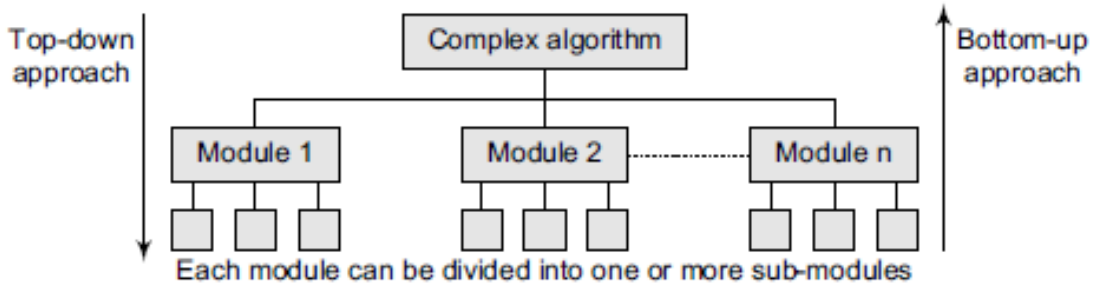
Algorithm Design

Algorithms are used to manipulate the data contained in data structures. When working with data structures, algorithms are used to perform operations on the stored data.
A complex algorithm is often divided into smaller units called modules. This process of dividing an algorithm into modules is called modularization. The key advantages of modularization are as
follows:

- It makes the complex algorithm simpler to design and implement.

- Each module can be designed independently. While designing one module, the details ofother modules can be ignored, thereby enhancing clarity in design which in turn simplifiesimplementation, debugging, testing, documenting, and maintenance of the overall algorithm.

There are two main approaches to design an algorithm—top-down approach and bottom-up approach:



CONTROL STRUCTURES USED IN ALGORITHMS

An algorithm has a finite number of steps. Some steps may involve decision-making and repetition.Broadly speaking, an algorithm may employ one of the following control structures:

(a) sequence,
(b) decision,
(c) repetition.

Sequence

By sequence, we mean that each step of an algorithm is executed in a specified order. Let us write an algorithm toadd two numbers. This algorithm performs the steps in a purely sequential order, as shown in Fig.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: SET SUM = A+B
Step 4: PRINT SUM
Step 5: END
```

Decision

Decision statements are used when the execution of a process depends on the outcome of some condition. For example, if x = y, then print EQUAL. So the general form of 'IF' construct can be given as:

IF condition Then process

A condition in this context is any statement that may evaluate to either a true value or a false value. In the above example, a variable x can be either equal to y or not equal to y. However, it cannot be both true and false. If the condition is true, then the process is executed.

A decision statement can also be stated in the following manner:

IF condition
Then process1
ELSE process2

This form is popularly known as the IF–ELSE construct. Here, if the condition is true, then process1 is executed, else process2 is executed. Figure 2.11 shows an algorithm to check if two numbers are equal.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: IF A = B
             PRINT "EQUAL"
        ELSE
             PRINT "NOT EQUAL"
        [END OF IF]
Step 4: END
```

Repetition

Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as while, do–while, and for loops. These loops execute one or more steps until some condition is true.

```
Step 1: [INITIALIZE] SET I = 1, N = 10
Step 2: Repeat Steps 3 and 4 while I<=N
Step 3: PRINT I
Step 4: SET I = I+1
        [END OF LOOP]
Step 5: END
```

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

- A Priori Analysis − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.

- A Posterior Analysis − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- Time Factor − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.

- Space Factor − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)
Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop
Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.
For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

Pointers

A pointer is a programming language object, whose value refers to (or "points to") another value stored elsewhere in the computer memory using its memory address. A pointer references a location in memory, and obtaining the value stored at that location is known as dereferencing the pointer. Pointers-to-data significantly improve performance for repetitive operations such as traversing strings, lookup tables, control tables and tree structures. In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.

Its uses:

- Pointers increase the execution speed.
- A pointers enable us to access a variable that is defined outside the function.
- Pointers are more efficient in handling the data tables.
- Pointers reduce the length and complexity of a program .
- The use of pointer array to character strings results in saving of data storage space in memory.
- Pointers are used to pass information back and forth between functions.
- Pointers enable the programmers to return multiple data items from a function via functionarguments.
- Pointers provide an alternate way to access the individual elements of an array.
- Pointers are used to pass arrays and strings as function arguments.
- Pointers are used to create complex data structures, such as trees, linked lists, linked stacks,linked queues, and graphs.

Array Definition and Analysis

A linear array is a list of afinite number n of homogeneous data element (i.e., data elements of the same type) such that:

I. The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.

II. The elements of the array are stored respectively in successive memory locations.

The number n of elements is called the length or size of the array. If not explicitly stated, we will assume the index set consists of the integers 1, 2,……. n. In general, the length or the number of data elements of the array can be obtained from the index set by the formula

Length = UB - LB + 1

where UB is the largest index, called the upper bound, and LB is the smallest index, called the lower bound, of the array. Note that length =UB when LB=1.

The elements of an array A may be denoted by the subscript notation

A_1, A_2, A_3…….…… A_n

or by the parentheses notation (used in FORTRAN. PL/1 and BASIC)

A(1), A(2),…… A(N)

or by the bracket notation (used in Pascal)

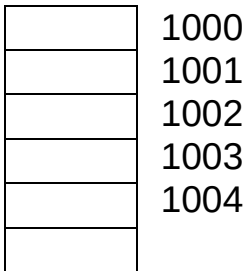
A[1], A[2]…….. A[N]

We will usually use the subscript notation or the bracket notation. Regardless of the notation, number K in A[K] is called a subscript or an index and A[K] is called a subscripted variable. Not that subscripts allow any element of A to be referenced by its relative position in A.

REPRESENTATION OF LINEAR ARRAYS IN MEMORY

Let A be a linear array in the memory of the computer. Recall that the memory of the computer is simply a sequence of addressed locations as pictured .Let us use the notation pictured

LOC(A[K])= address of the element A[K] of the array A



As previously noted, the elements of A are stored in successive memory cells. Accordingly, the computer does not need to keep track of the address of every element of A, but needs to keep track only of the address of the first element of A ,denoted by

Base(A)

And called the base address of A. Using this address Base(LA), the computer calculates the address of any element of A by the following formula:

$$LOC(A[K]) = Base(A) + w(K - \text{lower bound})$$

where w is the calculate number of words per memory cell for the array A. Observe that the time to calculate LOC(A[K]) is essentially the same for any value of K. Furthermore, given any subscripts K, one can locate and access the content value of A[K] without scanningany other element of A.

Traversing of Linear Arrays

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the content of each element of A or suppose we want to count the number of elements of A with given property. This can be accomplished by traversing A, that is, by accessing and processing (frequently called visiting) each element of An exactly once.

The following algorithm traverses a linear array:

Here, A is a linear array with lower bound LB and upper bound UB. This algorithm traverses A applying an operation PROCESS to each element of A.

Step I: [Initialize counter] Set K:= LB.
Step II: Repeat steps 3 and 4 while K<= UB:
Step III: Apply PROCESS to A[K]. [Visit element.]
Step IV: Set K:= K+1. [Increase counter.]
[End of step 2 loop.]
Step V: Exit.

Insertion And Deletion

Let A be a collection of data elements in the memory of the computer. “Inserting” refers to the operation of adding another element to the collection A, and “Deleting” refers to the operation of removing one of the elements from A.

Inserting an element at the ‘end’ of a linear array can be easily done. On the other hand, suppose we need to insert an element in the middle of the array. Then on the average, half of the elements must be moved downward to new locations to accommodate the new element and keep an order of the order of the other elements.

Similarly, deleting an element at the ‘end’ of an array presents no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element is moved one location upward in order to ‘fill up’ the array.

Algorithm: (Inserting into a linear array) INSERT (A, N, K, ITEM).

Here, A is a linear array with N elements and K is a positive integer such that K <= N. This algorithm inserts an element ITEM into the Kth position in A.

Step I: Set J:= N. [Initialize counter.]
Step II: Repeat steps 3 and 4 while J >= K:
Step III: Set A[J+1] := A[J]. [Move element downward.]
Step IV: J:= J-1. [Decrease counter.]
[End of step 2 loop.]
Step V: Set A[K] := ITEM. [Insert element.]
Step VI: Set N:= N+1. [Reset N.]
Step VII: Exit.

Algorithm: (Deletion from a linear array.) DELETE (A, K, N, ITEM).
Here, A is a linear array with N elements and K is a positive integer such that K<= N. This algorithm deletes the Kth element from A.

Step I: Set ITEM:= A[K] and J:= K. [Initialize counter.]
Step II: Repeat steps 3 and 4 while J>N:
Step III: A[J] := A[J+1]. [Move element upward.]
Step IV: Set J:= J+1. [Increase counter.]
[End of step 2 loop.]
Step V: Set N:= N-1. [Reset N.]
Step VI: Exit.

Searching in Array

Linear Search

Suppose A is a linear array with n elements. Given no other information about A, the most intuitive way to search for a given ITEM in A is to compute ITEM with each element of A one by one. That is, first we test whether A[1] = ITEM, and then we test whether A[2] = ITEM, and so on. This method, which traverses A sequentially to locate ITEM, is called 'linear search' or 'sequential search'.

Algorithm: (Linear Search.)LINEAR(A, N, ITEM, LOC).

Here, A is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in A, or sets LOC := 0 if the search is unsuccessful.

Step I: [Insert ITEM at the end of A.]
Set A[N+1] := ITEM.
Step II:[Initialize counter.]
Set LOC := 1.
Step III:[Search for ITEM.]
Repeat while A[LOC] != ITEM.
Set := LOC := LOC+1.
[End of step 3 loop.]
Step IV:[Successful?]
If LOC := N+1, then:
Print LOC := NULL.
Else:
Print LOC.
Step V:Exit.

Binary Search-

Suppose A is an array which is sorted in increasing numerical order or, equivalently alphabetically. Then, there is an extremely efficient searching algorithm, called 'binary search', which can be used to find the location LOC of a given ITEM of information in A.

Algorithm: (Binary Search.)BINARY(A, LB, UB, ITEM, LOC).
Here, A is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variable BEG, END, and MID denote respectively, the beginning, end and middle locations of a segment of elements of A. This algorithm finds the location LOC of ITEM in A or sets LOC = NULL.
Step I:[Initialize segment variable.]
Set BEG := LB, END := UB and MID := INT((BEG+END)/2).
Step II: Repeat steps 3 and 4 while BEG <= END and A[MID] != ITEM:
Step III: If A[MID] > ITEM, then:
Set END := MID-1.
Else:
Set BEG := MID+1.
[End of If structure.]
Step IV: Set MID := INT((BEG+END)/2).
[End of step 2 loop.]
Step V: If BEG > END, then:
Set LOC := NULL.
Else:
Set LOC := MID.
Step VI: Exit.

Limitations of Binary Search:

1.The list must be sorted.

2.One must have direct access to the middle element in a sub list.

Sorting in Arrays:

Bubble Sorting

Let A be a list of ‘n’ numbers. Sorting A refers to the operation of re-arranging the elements of A, so they are in increasing order, i.e. so that

A[1]< A[2]< A[3]< …….. <A[N]

For example, Suppose A originally is the list 8, 4, 19, 2, 7, 13, 5, 16. After sorting, A is the list 2, 4, 5, 7, 8, 13, 16, 19.

Algorithm: (Bubble Sort.)

BUBBLE(A, N). Here, A is an array with N elements. This algorithm sorts the elements in A.

Step I: Repeat steps 2 to 4 for K=1 to N-1.
[This loop is for number of passes.]
Step II: Set PTR:= 1. [Initializes pass pointer PTR.]
Step III: Repeat step 4 while PTR <= N-K: [Execute pass.]
[This step 3 loop is for comparisons.]
Step IV: If A[PTR] > A[PTR+1], then:
[Interchange the elements.]
a) TEMP:= A[PTR]
b) A[PTR]:= A[PTR+1]
c) A[PTR+1] := TEMP
[End of If structure.]
Step V: Set PTR:= PTR+1. [Increase pointer.]
[End of step 3 loop.]
[End of step 1 loop.]
Step VI: Exit.

TWO-DIMENSIONAL ARRAYS

One-dimensional arrays are organized linearly in only one direction. But at times, we need to store data in the form of grids or tables.
Here, the concept of single-dimension arrays is extended to incorporate two-dimensional data structures. A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second denotes the column. The C compiler treats a two-dimensional array as an array of one-dimensional arrays.
Syntax:

```
data_typearray_name[row_size][column_size];
```

If the array elements are stored in ***column major*** order,
Address(A[J,K]) = Base + w{M (K – 1) + (J – 1)}

And if the array elements are stored in ***row major*** order,
Address(A[J,K]) = Base + w{N (J – 1) + (K – 1)}

where,
Base=Base Address
w= Word per memory cell/size of an element
M is the number of rows,
N is the number of columns,
and J and K are the subscripts of the array element.

Q-Consider a 20 X 5 two-dimensional array marks which has its base address = 1000 and the size of an element = 2. Now compute the address of the element, marks[18][4] assuming that the elements are stored in row major order.?
Solution:
Address(A[J][K]) = Base + w{N (I – 1) + (J – 1)}
Address(marks[18][4]) = 1000 + 2 {5(18 – 1) + (4 – 1)}
= 1000 + 2 {5(17) + 3}
= 1000 + 2 (88)
= 1000 + 176 = 1176

Multi-Dimensional Arrays

A multi-dimensional array in simple terms is an array of arrays. As we have one index in a one dimensional array, two indices in a two-dimensional array, in the same way, we have n indices in an n-dimensional array or multi-dimensional array.

Let C be an n-dimensional array the Length of dimension i of C can be find with

$$L_i = \text{Upper Bound} - \text{Lower Bound} + 1$$

For a given subscript K. the effective index E of is the number of indices preceding K in the L set, and E can be calculated from

$$E_i = K_i - \text{Lower bound}$$

The address LOC(C[K_1, K_2,….,K_N]) of an arbitrary element of C can be obtained from the formula

$LOC(C[K_1, K_2,..,K_N]) = Base(C)+w[(((\ldots(E_N L_{N-1}+E_{N-1})L_{N-2})+\ldots+.E_3)L_2+E_2)L_1+E_1]$

according to whether C is stored in column-major or row-major order. Once again, Base(C) denotes the address of the first element of C. and w denotes the number of words per memory location.

SPARSE MATRICES

Sparse matrix is a matrix that has large number of elements with a zero value. In order to efficiently utilize the memory, specialized algorithms and data structures that take advantage of the sparse structure should be used. If we apply the operations using standard matrix structures and algorithms to sparse matrices, then the execution will slow down and the matrix will consume large amount of memory.Sparse data can be easily compressed, which in turn can significantly reduce memory usage.

There are two types of sparse matrices.
In the first type of sparse matrix, all elements above the main diagonal have a zero value. This type of sparse matrix is also called a ***(lower) triagonal matrix*** because if you see it pictorially, all the elements with a non-zero value appear below the diagonal.

$$\begin{bmatrix} 1 & & & & \\ 5 & 3 & & & \\ 2 & 7 & -1 & & \\ 3 & 1 & 4 & 2 & \\ -9 & 2 & -8 & 1 & 7 \end{bmatrix}$$

In an upper-triangular matrix, all the elements main diagonal have non-zero value this type of sparse matrix is called ***(Upper) triagonal matrix.***

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ & 3 & 6 & 7 & 8 \\ & & -1 & 9 & 1 \\ & & & 9 & 2 \\ & & & & 7 \end{bmatrix}$$

There is another variant of a sparse matrix, in which elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of matrix is also called a tri-diagonal matrix.

$$\begin{bmatrix} 4 & 1 & & & & \\ 5 & 1 & 2 & & & \\ & 9 & 3 & 1 & & \\ & & 4 & 2 & 2 & \\ & & & 5 & 1 & 9 \\ & & & & 8 & 7 \end{bmatrix}$$