**UNIT-I**

**What are Data Structures?**

Data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

Depending on your requirement and project, it is important to choose the right data structure for your project. For example, if you want to store data sequentially in the memory, then you can go for the Array data structure.

## Types of Data Structure

Basically, data structures are divided into two categories:

* Linear data structure
* Non-linear data structure

Let's learn about each type in detail.

## Linear data structures

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.

However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.

**Popular linear data structures are:**

### 1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language

## Types of Data Structure

Basically, data structures are divided into two categories:

* Linear data structure
* Non-linear data structure

Let's learn about each type in detail.

## Linear data structures

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.

However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.

**Popular linear data structures are:**

### 1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language

## Types of Data Structure

Basically, data structures are divided into two categories:

* Linear data structure
* Non-linear data structure

Let's learn about each type in detail.

## Linear data structures

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.

However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.

**Popular linear data structures are:**

### 1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language

### 2. Stack Data Structure

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.

It works just like a pile of plates where the last plate kept on the pile will be removed first

## Non linear data structures

Unlike linear data structures, elements in non-linear data structures are not in any sequence. Instead they are arranged in a hierarchical manner where one element will be connected to one or more elements.

Non-linear data structures are further divided into graph and tree based data structures.

### 1. Graph Data Structure

In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.

### 2. Trees Data Structure

Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.

**Analysis of Algorithms:**

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.

The Need for Analysis

In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

* **Worst-case** − The maximum number of steps taken on any instance of size **a**.
* **Best-case** − The minimum number of steps taken on any instance of size **a**.
* **Average case** − An average number of steps taken on any instance of size **a**.
* **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa. In this context, if we compare **bubble sort** and **merge sort**. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

**Data structure operations:**

There are 8 operations that can be performed on Data Structures. These are:
1. **Creation**: Creating a data structure according to the requirement.
                   eg: an integer array of 5 values.
                                  int ar[5];   //ar is the name of array
2. **Insertion**: Inserting values into data structure. There can be three ways to insert elements into data structure: at the beginning, at the end and at the desired locaiton.
3. **Traversal**: Visiting each element of the data structure at least once.
4. **Search**: Searching of an element in the given number of elements. The elements can be searched in two ways:
         a. Linear Search: Simplest way of searching an element.
         b. Binary Search: It works on divide and conquer rule.
5. **Sorting**: Rearranging the elements in a paritcular order, ascending or descending order. There are several sorting algorithms:
        a. Bubble Sort
        b. Selection Sort
        c. Quick Sort
        d. Merge Sort
        e. Heap Sort
6. **Merging**: Combining the data items of two sorted files into single file in the sorted form.
7. **Updation**: Updating the current value in the data structure with some new value.
8. **Deletion**: Deleting the undesired value from the data structure. There are 3 ways to delete a value from data structure. These are: from the beginning, from the end and from the given location.

**Array:**

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

* **Element** − Each item stored in an array is called an element.
* **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

* Index starts with 0.
* Array length is 10 which means it can store 10 elements.
* Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

An array is a homogeneous sequential collection of data items over a single variable name.

**For example,** int student[30];

Here, student is an array name holds 30 collection of data item, with a single variable name.

## Characteristics

The characteristics of arrays are as follows −

* An array is always stored in consecutive memory location.
* It can store multiple value of similar type, which can be referred with single name.
* The pointer points to the first location of memory block, which is allocated to the array name.
* An array can either be an integer, character, or float data type that can be initialized only during the declaration.
* The particular element of an array can be modified separately without changing the other elements.
* All elements of an array can be distinguishing with the help of index number.

## Basic Operations in Array:

Following are the basic operations supported by an array.

* **Traverse** − print all the array elements one by one.
* **Insertion** − Adds an element at the given index.
* **Deletion** − Deletes an element at the given index.
* **Search** − Searches an element using the given index or by the value.
* **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

|  |  |
| --- | --- |
| **Data Type** | **Default Value** |
| Bool | False |
| Char | 0 |
| Int | 0 |
| Float | 0.0 |
| Double | 0.0f |
| Void |  |
| wchar\_t | 0 |

## Traverse Operation

This operation is to traverse through the elements of an array.

### Example

Following program traverses and prints the elements of an array:

#include <stdio.h>

main() {

 int LA[] = {1,3,5,7,8};

 int item = 10, k = 3, n = 5;

 int i = 0, j = n;

 printf("The original array elements are :\n");

 for(i = 0; i<n; i++) {

 printf("LA[%d] = %d \n", i, LA[i]);

 }

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

## Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

### Example

Following is the implementation of the above algorithm −

include <stdio.h>

main() {

 int LA[] = {1,3,5,7,8};

 int item = 10, k = 3, n = 5;

 int i = 0, j = n;

 printf("The original array elements are :\n");

 for(i = 0; i<n; i++) {

 printf("LA[%d] = %d \n", i, LA[i]);

 }

 n = n + 1;

 while( j >= k) {

 LA[j+1] = LA[j];

 j = j - 1;

 }

 LA[k] = item;

 printf("The array elements after insertion :\n");

 for(i = 0; i<n; i++) {

 printf("LA[%d] = %d \n", i, LA[i]);

 }

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after insertion :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 10

LA[4] = 7

LA[5] = 8

For other variations of array insertion operation [click here](https://www.tutorialspoint.com/data_structures_algorithms/array_insertion_algorithm.htm)

## Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

### Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of LA.

1. Start

2. Set J = K

3. Repeat steps 4 and 5 while J < N

4. Set LA[J] = LA[J + 1]

5. Set J = J+1

6. Set N = N-1

7. Stop

### Example

Following is the implementation of the above algorithm −

#include <stdio.h>

void main() {

 int LA[] = {1,3,5,7,8};

 int k = 3, n = 5;

 int i, j;

 printf("The original array elements are :\n");

 for(i = 0; i<n; i++) {

 printf("LA[%d] = %d \n", i, LA[i]);

 }

 j = k;

 while( j < n) {

 LA[j-1] = LA[j];

 j = j + 1;

 }

 n = n -1;

 printf("The array elements after deletion :\n");

 for(i = 0; i<n; i++) {

 printf("LA[%d] = %d \n", i, LA[i]);

 }

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after deletion :

LA[0] = 1

LA[1] = 3

LA[2] = 7

LA[3] = 8

## Search Operation

You can perform a search for an array element based on its value or its index.

### Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

1. Start

2. Set J = 0

3. Repeat steps 4 and 5 while J < N

4. IF LA[J] is equal ITEM THEN GOTO STEP 6

5. Set J = J +1

6. PRINT J, ITEM

7. Stop

### Example

Following is the implementation of the above algorithm −

#include <stdio.h>

void main() {

 int LA[] = {1,3,5,7,8};

 int item = 5, n = 5;

 int i = 0, j = 0;

 printf("The original array elements are :\n");

 for(i = 0; i<n; i++) {

 printf("LA[%d] = %d \n", i, LA[i]);

 }

 while( j < n){

 if( LA[j] == item ) {

 break;

 }

 j = j + 1;

 }

 printf("Found element %d at position %d\n", item, j+1);

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

Found element 5 at position 3

## Update Operation

Update operation refers to updating an existing element from the array at a given index.

### Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the Kth position of LA.

1. Start

2. Set LA[K-1] = ITEM

3. Stop

### Example

Following is the implementation of the above algorithm −

#include <stdio.h>

void main() {

 int LA[] = {1,3,5,7,8};

 int k = 3, n = 5, item = 10;

 int i, j;

 printf("The original array elements are :\n");

 for(i = 0; i<n; i++) {

 printf("LA[%d] = %d \n", i, LA[i]);

 }

 LA[k-1] = item;

 printf("The array elements after updation :\n");

 for(i = 0; i<n; i++) {

 printf("LA[%d] = %d \n", i, LA[i]);

 }

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after updation :

LA[0] = 1

LA[1] = 3

LA[2] = 10

LA[3] = 7

LA[4] = 8

Amortized Analysis:

Amortized Analysis is **used for algorithms where an occasional operation is very slow, but most of the other operations are faster**. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a particularly expensive operation.



# What is a one-dimensional array in C language?

[C](https://www.tutorialspoint.com/questions/category/C)[Server Side Programming](https://www.tutorialspoint.com/questions/category/Server-Side-Programming)[Programming](https://www.tutorialspoint.com/questions/category/Programming)

An array is a group of related items that store with a common name.

## Syntax

The syntax is as follows for declaring an array −

datatype array\_name [size];

## Types of arrays

Arrays are broadly classified into three types. They are as follows −

* One – dimensional arrays
* Two – dimensional arrays
* Multi – dimensional arrays

### One – dimensional array

The Syntax is as follows −

datatype array name [size]

For example, int a[5]

**Initialization**

An array can be initialized in two ways, which are as follows −

* Compile time initialization
* Runtime initialization

## Example

Following is the C program on compile time initialization −

#include<stdio.h>

int main ( ){

   int a[5] = {10,20,30,40,50};

   int i;

   printf ("elements of the array are");

   for ( i=0; i<5; i++)

      printf ("%d", a[i]);

}

## Output

Upon execution, you will receive the following output −

Elements of the array are

10 20 30 40 50

## Two multidimensional arrays

These are used in situations where a table of values have to be stored (or) in matrices applications.

**Syntax**

The syntax is given below −

datatype array\_ name [rowsize] [column size];

**For example** int a[5] [5];

|  |  |  |
| --- | --- | --- |
| a[0][0]10 | a[0][1]20 | a[0][2]30 |
| a[1][0]40 | a[1][1]50 | a[1][2]60 |
| a[2][0] | a[2][1] | a[2][2] |

Following is the C Program for **compile time initialization** −

## Example

Write a C program In MATRIX(6 mark question)

#include<stdio.h>

main ( ){

   int a[3][3] = {10,20,30,40,50,60,70,80,90};

   int i,j;

   printf ("elements of the array are");

   for ( i=0; i<3; i++){

      for (j=0;j<3; j++){

         printf("%d \t", a[i] [j]);

      }

      printf("\n");

   }

}

## Output

The output is stated below −

elements of the array are:

10 20 30

40 50 60

70 80 90

MULTI DIMENSIONAL ARRAY:

C language allows arrays of three (or) more dimensions. This is a multidimensional array.

The exact limit is determined by the compiler.

The syntax is as follows −

datatype arrayname [size1] [size2] ----- [sizen];

For example, for three – dimensional array −

int a[3] [3] [3];

Number of elements = 3\*3\*3 = 27 elements

## Example

## Write a matrix multiplication prg using C(10 mark question)

1. #include<stdio.h>
2. #include<stdlib.h>
3. **int** main(){
4. **int** a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
5. system("cls");
6. printf("enter the number of row=");
7. scanf("%d",&r);
8. printf("enter the number of column=");
9. scanf("%d",&c);
10. printf("enter the first matrix element=\n");
11. **for**(i=0;i<r;i++)
12. {
13. **for**(j=0;j<c;j++)
14. {
15. scanf("%d",&a[i][j]);
16. }
17. }
18. printf("enter the second matrix element=\n");
19. **for**(i=0;i<r;i++)
20. {
21. **for**(j=0;j<c;j++)
22. {
23. scanf("%d",&b[i][j]);
24. }
25. }
26.
27. printf("multiply of the matrix=\n");
28. **for**(i=0;i<r;i++)
29. {
30. **for**(j=0;j<c;j++)
31. {
32. mul[i][j]=0;
33. **for**(k=0;k<c;k++)
34. {
35. mul[i][j]+=a[i][k]\*b[k][j];
36. }
37. }
38. }
39. //for printing result
40. **for**(i=0;i<r;i++)
41. {
42. **for**(j=0;j<c;j++)
43. {
44. printf("%d\t",mul[i][j]);
45. }
46. printf("\n");
47. }
48. **return** 0;
49. }

**Output:**

enter the number of row=3

enter the number of column=3

enter the first matrix element=

1 1 1

2 2 2

3 3 3

enter the second matrix element=

1 1 1

2 2 2

3 3 3

multiply of the matrix=

6 6 6

12 12 12

18 18 18