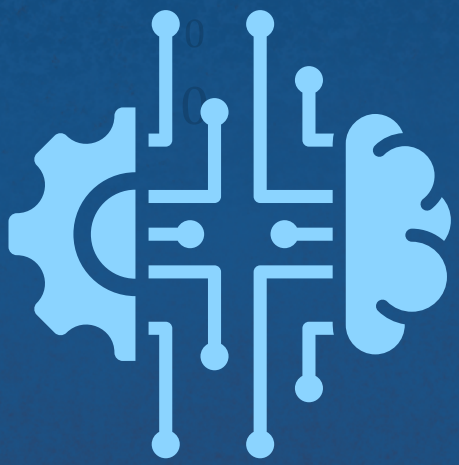
DSA NOTES

Part- 1



FOLLOW US!

www.topperworld.in

Introduction to Data Structure

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

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

A data structure is said to be **linear** if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be **non linear** if its elements form a hierarchical classification where, data items appear at various levels.

Trees and **Graphs** are widely used **non-linear data structures**. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Importance of Data Structure

Due to the complexity of applications and the daily growth in data, there may be issues with processing **speed, data searching, handling numerous requests,** etc. Data structure offers a method for effectively **managing, organising, and storing data.** Data structures make it simple to navigate through the data elements. Data structures offer **productivity, reuse, and abstraction**. Because storing and retrieving user data as quickly as feasible is a program's primary job, it plays a significant role in improving performance.

Types of Data Structure

Data structures are divided into two types:

- **Primitive data structures.**
- **Non-primitive data structures.**

➔ Primitive Data Structure :

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

➔ Non-Primitive Data Structure :

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure shows the classification of data structures.

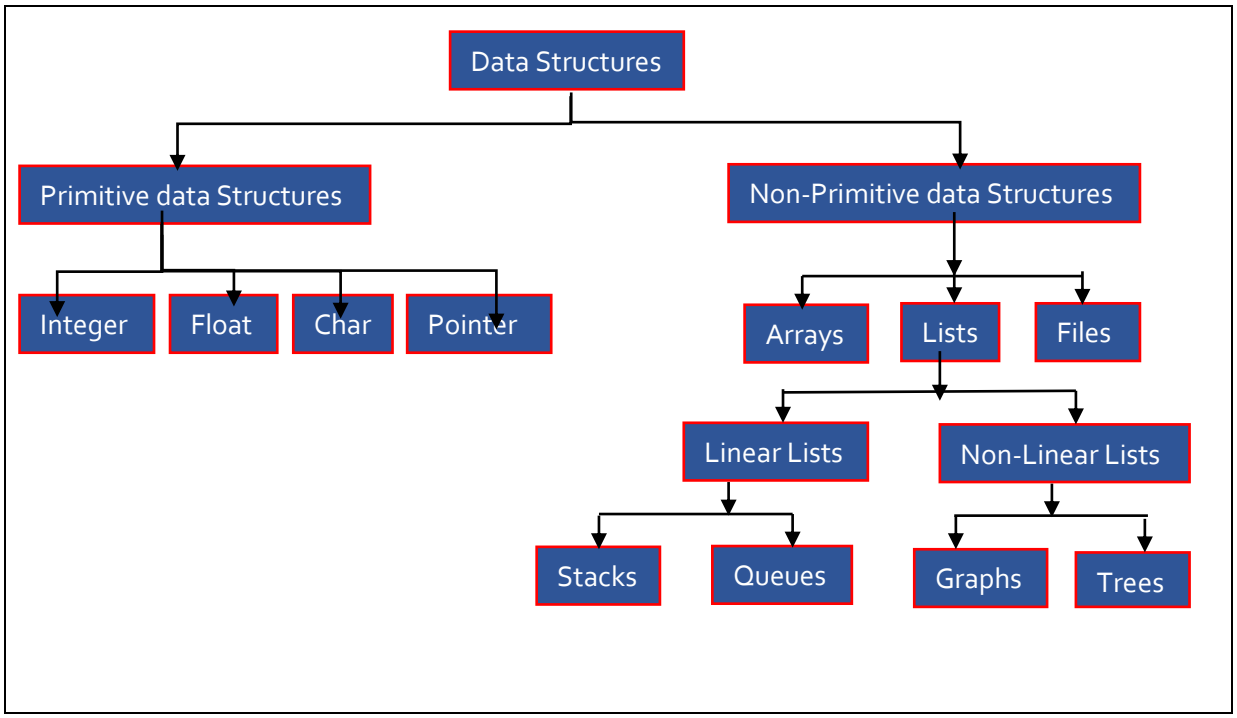


Figure : Classification of Data Structures.

The non-primitive data structure is divided into two types:

- **Linear data structure**
- **Non-linear data structure**

➔ Linear Data Structure

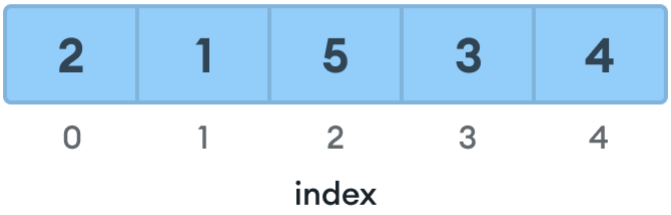
In **linear data structures**, the elements are arranged in sequence one after the other.

Elements are arranged in particular order, they are easy to implement.

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.

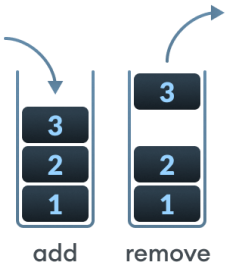


An array with each element represented by an index

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.



In a stack, operations can be perform only from one end (top here).

3. Queue Data Structure

Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first.

It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first.

In a queue, addition and removal are performed from separate ends.

4. Linked List Data Structure

In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.

A linked list

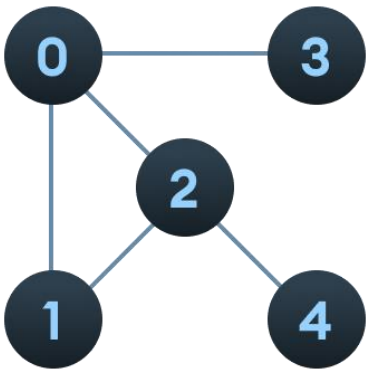
➔ Non-Linear Data Structure

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

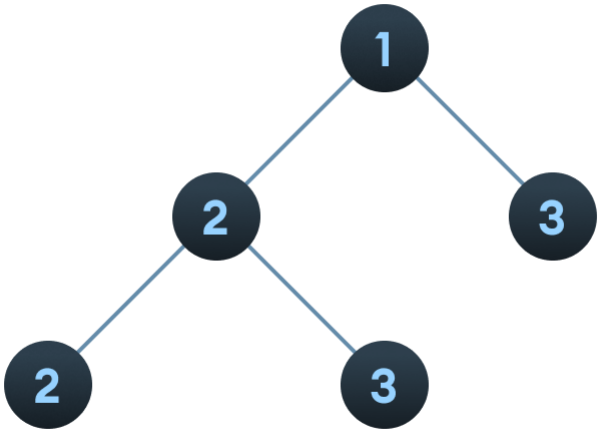
In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.



Graph data structure example

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.



Tree data structure example

➔ Linear Vs Non-linear Data Structures

Now that we know about linear and non-linear data structures, let's see the major differences between them.

Linear Data Structures	Non Linear Data Structures
• The data items are arranged in sequential order, one after the other.	The data items are arranged in non-sequential order (hierarchical manner).
• All the items are present on the single layer.	The data items are present at different layers.
• It can be traversed on a single run. That is, if we start from the first element, we can traverse all the elements sequentially in a single pass.	It requires multiple runs. That is, if we start from the first element it might not be possible to traverse all the elements in a single pass.

• The memory utilization is not efficient.	Different structures utilize memory in different efficient ways depending on the need.
• The time complexity increase with the data size.	Time complexity remains the same.
• Example: Arrays, Stack, Queue	Example: Tree, Graph, Map

Data structures: Organization of data

The collection of data you work with in a program have some kind of structure or organization. No matter how complex your data structures are they can be broken down into two fundamental types:

- **Contiguous**
- **Non-Contiguous**

➔Contiguous Structures

In **contiguous structures**, terms of data are kept together in memory (either RAM or in a file). An **array** is an example of a contiguous structure. Since each element in the array is located next to one or two other elements.

➔Non-Contiguous Structures

Items in a **non-contiguous structure** and scattered in memory, but we linked to each other in some way. A **linked list** is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node.

Figure below illustrates the difference between contiguous and non-contiguous structures.

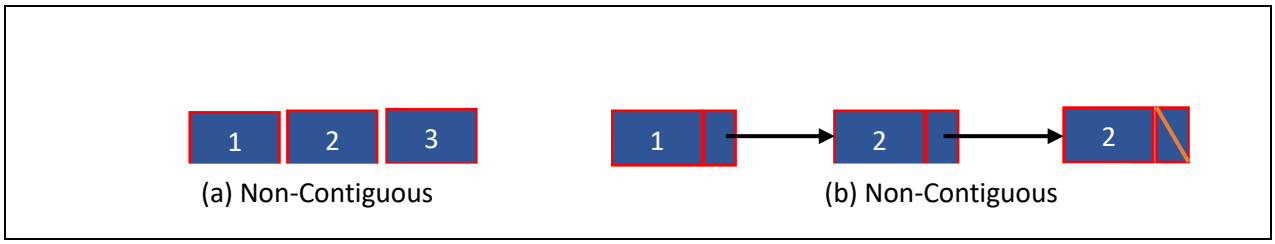


Figure : Contiguous and Non-contiguous structures compared

➔Hybrid structures

If two basic types of structures are mixed then it is a hybrid form. Then one part contiguous and another part non-contiguous.

For example, figure shows how to implement a double–linked list using three parallel arrays, possibly stored a past from each other in memory.

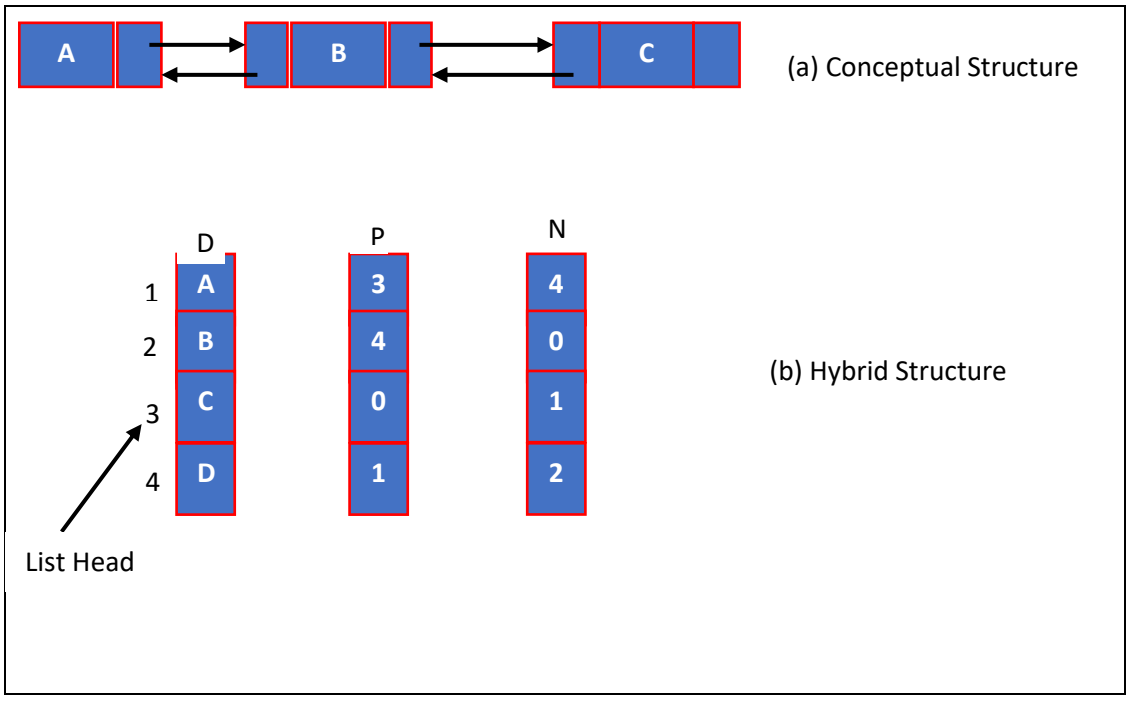


Figure : A double linked list via a hybrid data structure

The array D contains the data for the list, whereas the array P and N hold the previous and next "pointers". The pointers are actually nothing more than indexes into the D array. For instance, D[i] holds the data for node i and p[i] holds the index to the node previous to i, where may or may not reside at position i–1. Like wise, N[i] holds the index to the next node in the list.

Operation of Data Structure

Data structure operations are the methods used to manipulate the data in a data structure. The most common data structure operations are:

- **Traversal :** Traversal operations are used to visit each node in a data structure in a specific order. This technique is typically employed for **printing**, **searching**, **displaying**, and **reading** the data stored in a data structure.
- **Insertion :** Insertion operations add **new data elements** to a data structure. You can do this at the data structure's beginning, middle, or end.
- **Deletion :** Deletion operations remove data elements from a data structure. These operations are typically performed on nodes that are no longer needed.
- **Search :** Search operations are used to find a **specific data element** in a data structure. These operations typically employ a compare function to determine if two data elements are equal.
- **Sort :** Sort operations are used to **arrange** the data elements in a data structure in a specific order. This can be done using various sorting algorithms, such as insertion sort, bubble sort, merge sort, and quick sort.
- **Merge :** Merge operations are used to **combine** two data structures into one. This operation is typically used when two data structures need to be combined into a single structure.

- **Copy :** Copy operations are used to create a **duplicate** of a data structure. This can be done by copying each element in the original data structure to the new one.

Application of Data Structure

Data structures have many applications, such as:

➔ **Data Storage:**

Data structures facilitate efficient data persistence, like specifying attribute collections and corresponding structures used in database management systems to store records.

➔ **Data Exchange:**

Organized information, defined by data structures, can be shared between applications like TCP/IP packets.

➔ **Resource and Service Management:**

Data structures such as linked lists can enable core operating systems resources and services to perform functions like file directory management, memory allocation, and processing scheduling queues.

➔ **Scalability:**

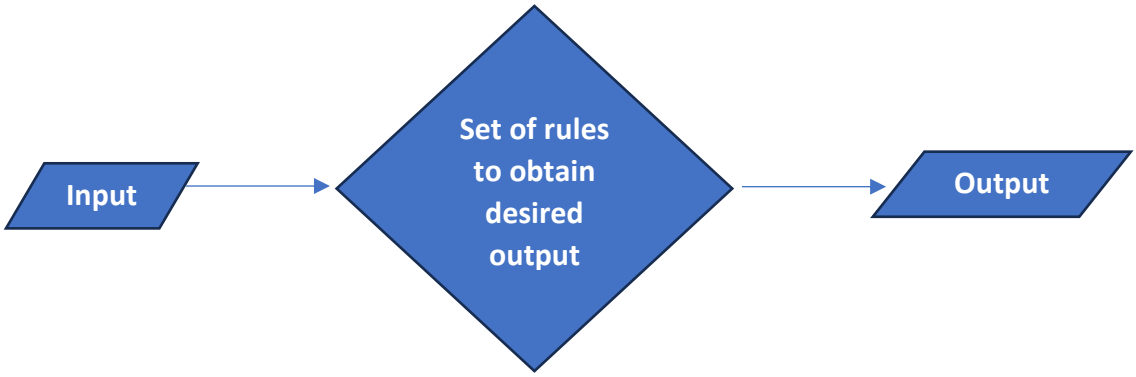
Big data applications rely on data structures to manage and allocate data storage across many distributed storage locations. This function guarantees scalability and high performance.

Advantages of Data structures

- Data structure facilitates effective data storage in storage devices.
- The use of data structures makes it easier to retrieve data from a storage device.
- The data structure allows for the effective and efficient processing of both little and big amounts of data.
- Manipulation of vast amounts of data is simple when a proper data structure technique is used.
- The use of a good data structure may assist a programmer to save a lot of time or processing time while performing tasks such as data storage, retrieval, or processing.
- Most well-organized data structures, including stacks, arrays, graphs, queues, trees, and linked lists, have well-built and pre-planned approaches for operations such as storage, addition, retrieval, modification, and deletion. The programmer may totally rely on these facts while utilising them.
- Data structures such as arrays, trees, linked lists, stacks, graphs, and so on are thoroughly verified and proved, so anybody may use them directly without the need for study and development. If you opt to design your own data structure, you may need to do some study, but it will almost certainly be to answer a problem that is more sophisticated than what these can supply.
- In the long term, data structure utilization might merely encourage reusability.

Introduction to Algorithms

- An **algorithm** is a finite set of instructions or logic, written in order, to accomplish a certain predefined task.



Algorithm

- **Algorithm** is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as pseudocode or using a flowchart.

Every Algorithm must satisfy the following properties:

- **Input-** There should be 0 or more inputs supplied externally to the algorithm.
- **Output-** There should be atleast 1 output obtained.
- **Definiteness-** Every step of the algorithm should be clear and well defined.
- **Finiteness-** The algorithm should have finite number of steps.

- **Correctness-** Every step of the algorithm must generate a correct output.

- An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

  - **Time Complexity**
  - **Space Complexity**

➔ Space Complexity

Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.

Sometime Auxiliary Space is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = Auxiliary Space + Input space

Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

   It's the amount of memory used to save the compiled version of instructions.

2. **Environmental Stack**

   Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

For example, If a function A() calls function B() inside it, then all th variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the funciton A().

3. **Data Space**

   Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack**.

Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	**Size**
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

Now let's learn how to compute space complexity by taking a few examples:

```
{

    int z = a + b + c;

    return(z);

}
```

In the above expression, variables a, b, c and z are all integer types, hence they will take up 4 bytes each, so total memory requirement will be (4(4) + 4) = 20 bytes, this additional 4 bytes is for **return value**. And because this space requirement is fixed for the above example, hence it is called **Constant Space Complexity**.

Let's have another example, this time a bit complex one,

```
// n is the length of array a[]

int sum(int a[], int n)

{

      int x = 0;      // 4 bytes for x

      for(int i = 0; i < n; i++)     // 4 bytes for
i

      {

          x  = x + a[i];

      }

      return(x);

}
```

- In the above code, 4*n bytes of space is required for the array a[] elements.
- 4 bytes each for x, n, i and the return value.

Hence the total memory requirement will be (4n + 12), which is increasing linearly with the increase in the input value n, hence it is called as **Linear Space Complexity.**

Similarly, we can have quadratic and other complex space complexity as well, as the complexity of an algorithm increases.

But we should always focus on writing algorithm code in such a way that we keep the space complexity **minimum**.

➔ Time Complexity

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity. We will study about it in detail in the next tutorial.

Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution. Like in the example above, for the first code the loop will run n number of times, so the time complexity will be n atleast and as the value of n will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of n, it will always give the result in 1 step.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

```
statement;
```

- Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
    statement;
}
```

- The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)
{
    for(j=0; j < N;j++)
    {
```

```
    statement;

    }

}
```

- This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

Asymptotic Analysis

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. **The efficiency is measured with the help of asymptotic notations**.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

- For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

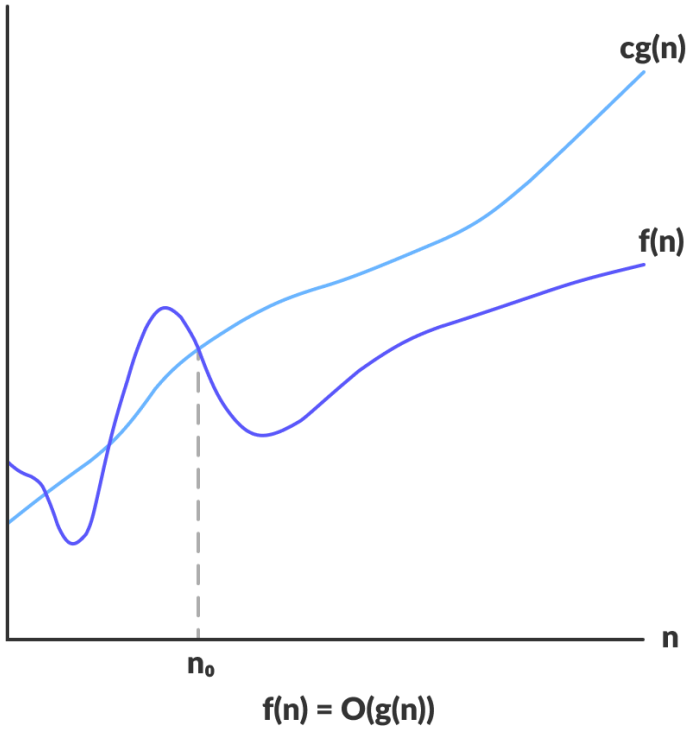
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- **Big-O notation**
- **Omega notation**
- **Theta notation**

➔ Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



Big-O gives the upper bound of a function

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0 \}$

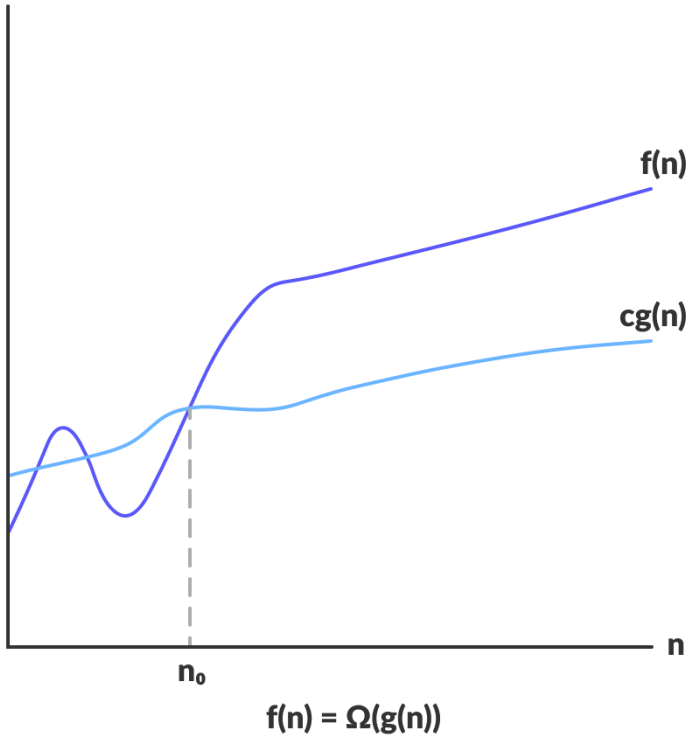
- The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

  For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

  Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

➔ Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.



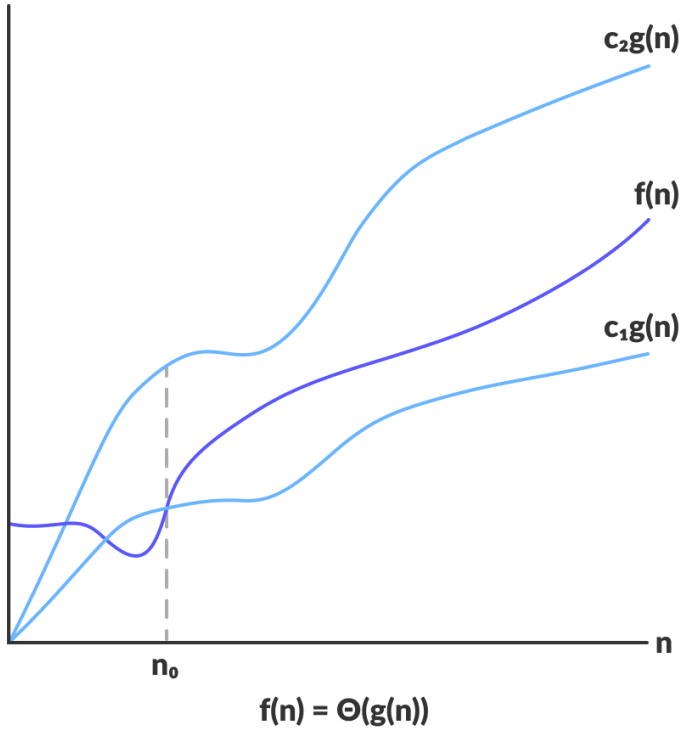
Omega gives the lower bound of a function

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

- The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.
  For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

➔ Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



Theta bounds the function within constants factors

For a function $g(n)$, $\Theta(g(n))$ is given by the relation:

$$\Theta(g(n)) = \{ f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$$

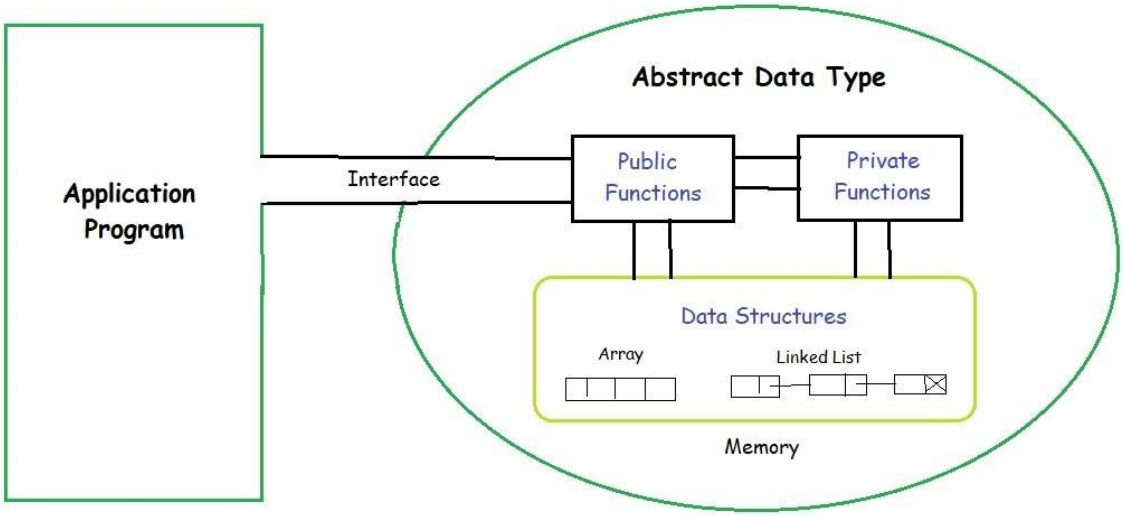
- The above expression can be described as a function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.

Abstract Data Types

An Abstract Data Type in data structure is a kind of a data type whose behaviour is defined with the help of some attributes and some functions. Generally, we write these attributes and functions inside a class or a structure so that we can use an object of the class to use that particular abstract data type.

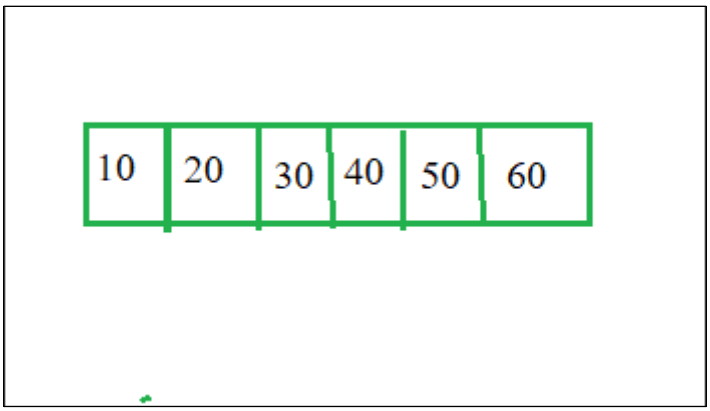
Examples of Abstract Data Type in Data Structure are list, stack, queue etc.

The process of providing only the essentials and hiding the details is known as abstraction.



The user of data-types does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.
So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely **List ADT**, **Stack ADT**, **Queue ADT**.

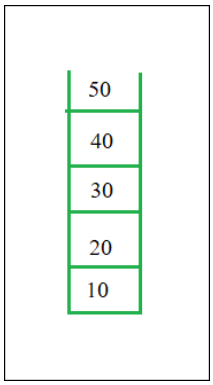
1. List ADT



View of list

- The data is generally stored in key sequence in a list which has a head structure consisting of **count, pointers** and **address of compare function** needed to compare the data in the list.
- The data node contains the *pointer* to a data structure and a self-referential pointer which points to the next node in the list.
- The **List ADT Functions** is given below:
- **get()** – Return an element from the list at any given position.
- **insert()** – Insert an element at any position of the list.
- **remove()** – Remove the first occurrence of any element from a non-empty list.
- **removeAt()** – Remove the element at a specified location from a non-empty list.
- **replace()** – Replace an element at any position by another element.
- **size()** – Return the number of elements in the list.
- **isEmpty()** – Return true if the list is empty, otherwise return false.
- **isFull()** – Return true if the list is full, otherwise return false.

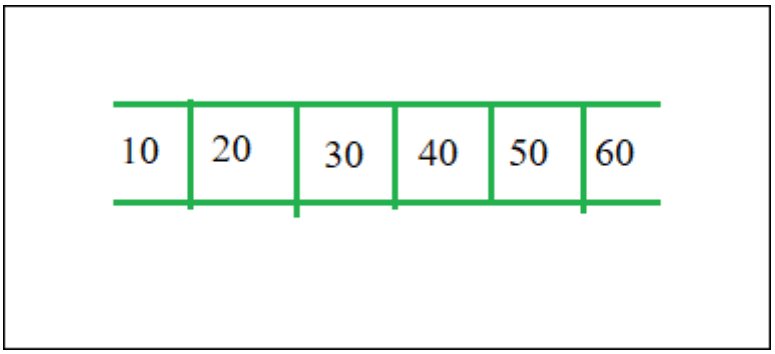
2. Stack ADT



View of stack

- In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the **data** and **address** is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to **top** and **count** of number of entries currently in stack.
- **push()** – Insert an element at one end of the stack called top.
- **pop()** – Remove and return the element at the top of the stack, if it is not empty.
- **peek()** – Return the element at the top of the stack without removing it, if the stack is not empty.
- **size()** – Return the number of elements in the stack.
- **isEmpty()** – Return true if the stack is empty, otherwise return false.
- **isFull()** – Return true if the stack is full, otherwise return false.

3. Queue ADT



View of Queue

- The queue abstract data type (ADT) follows the basic design of the stack abstract data type.
- Each node contains a void pointer to the **data** and the **link pointer** to the next element in the queue. The program's responsibility is to allocate memory for storing the data.
- **enqueue()** – Insert an element at the end of the queue.
- **dequeue()** – Remove and return the first element of the queue, if the queue is not empty.
- **peek()** – Return the element of the queue without removing it, if the queue is not empty.
- **size()** – Return the number of elements in the queue.
- **isEmpty()** – Return true if the queue is empty, otherwise return false.
- **isFull()** – Return true if the queue is full, otherwise return false.

➔Features of ADT:

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.
- **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
- **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organizing and manipulating data in a structured and efficient manner.

Abstract data types (ADTs) have several advantages and disadvantages that should be considered when deciding to use them in software development. Here are some of the main advantages and disadvantages of using ADTs:

➔Advantages:

- **Encapsulation**: ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
- **Abstraction**: ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
- **Data Structure Independence**: ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
- **Information Hiding**: ADTs can protect the integrity of data by controlling access and preventing unauthorized modifications.
- **Modularity**: ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

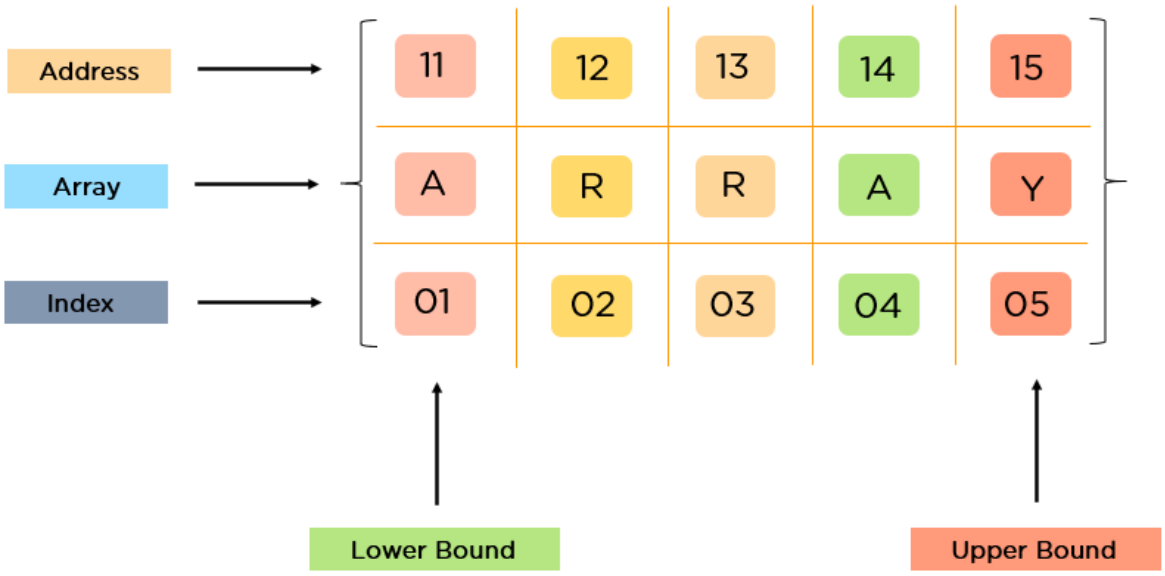
➔Disadvantages:

- **Overhead**: Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
- **Complexity**: ADTs can be complex to implement, especially for large and complex data structures.
- **Learning** Curve: Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
- **Limited Flexibility:** Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
- **Cost**: Implementing ADTs may require additional resources and investment, which can increase the cost of development

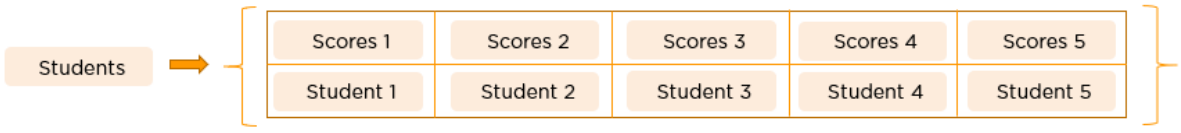
Arrays

An **array** is a linear data structure that collects elements of the **same data** type and stores them in contiguous and adjacent memory locations.

Arrays work on an index system starting from 0 to (n-1), where n is the size of the array.



➔Need of Array :



Let's suppose a class consists of ten students, and the class has to publish their results. If you had declared all ten variables individually, it would be challenging to manipulate and maintain the data.

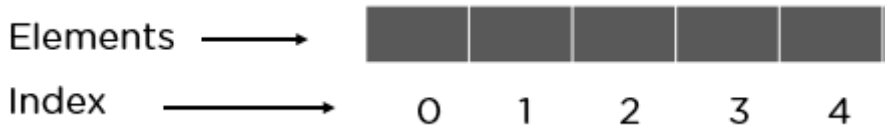
If more students were to join, it would become more difficult to declare all the variables and keep track of it. To overcome this problem, arrays came into the picture.

➔Types of Arrays :

There are majorly two types of arrays, they are:

- **One-Dimensional Arrays**
- **Multi-Dimensional Arrays**

➔ One-Dimensional Arrays:

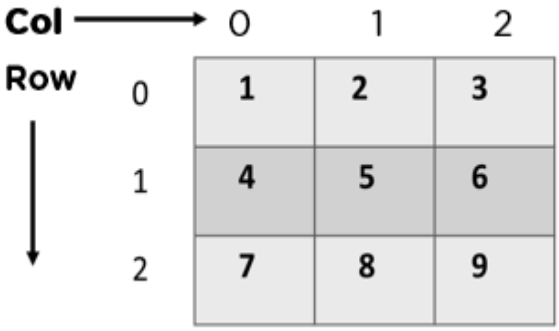


You can imagine a 1d array as a row, where elements are stored one after another.

➔Multi-Dimensional Arrays:

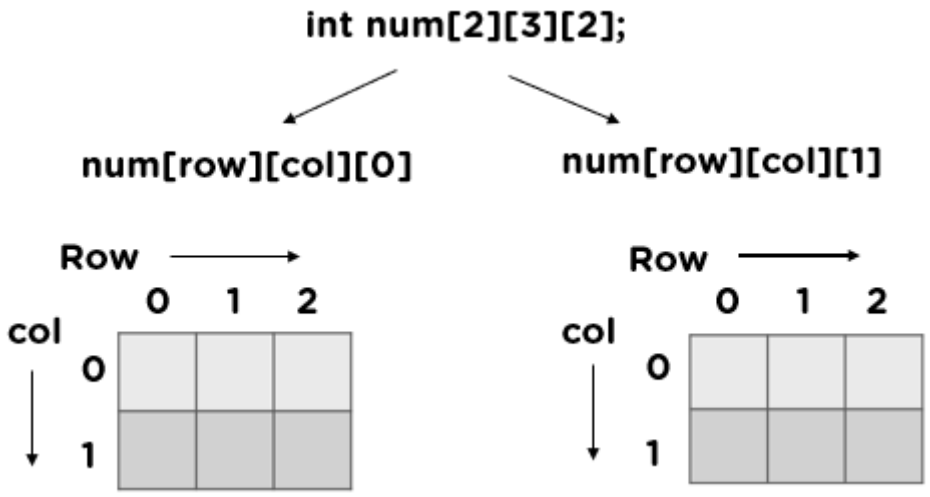
These multi-dimensional arrays are again of two types. They are:

1. Two-Dimensional Arrays :



You can imagine it like a table where each cell contains elements.

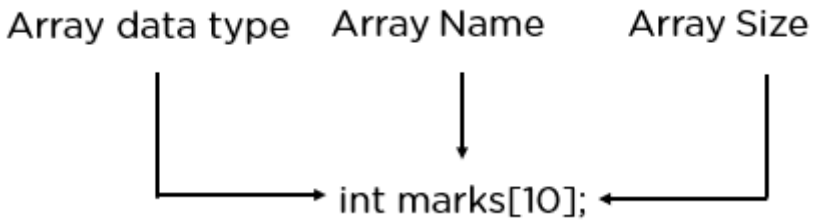
2. Three-Dimensional Arrays:



You can imagine it like a cuboid made up of smaller cuboids where each cuboid can contain an element.

In this "arrays in data structures" tutorial, you will work around one-dimensional arrays.

➔Declaration of Array



Arrays are typically defined with square brackets with the size of the arrays as its argument.

Here is the syntax for arrays:

- **1D Arrays:** **int arr[n];**

- **2D Arrays: int arr[m][n];**
- **3D Arrays: int arr[m][n][o];**

➔Initialization of an Array

You can initialize an array in four different ways:

- **Method 1:**

```
int a[6] = {2, 3, 5, 7, 11, 13};
```

- **Method 2:**

```
int arr[]= {2, 3, 5, 7, 11};
```

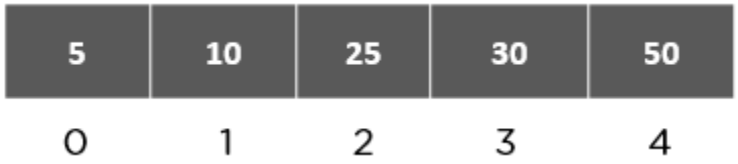
- **Method 3:**

```
int n;
scanf("%d",&n);
int arr[n];
for(int i=0;i<5;i++)
{
scanf("%d",&arr[i]);
}
```

- **Method 4:**

```
int arr[4];
arr[0]=1;
arr[1]=2;
arr[2]=3;
arr[3]=4;
```

➔Accessing Elements of Arrays in Data Structures



You can access elements with the help of the index at which you stored them. Let's discuss it with a code:

```c
#include<stdio.h>
int main()
{
int a[5] = {2, 3, 5, 7, 11};
printf("%d\n",a[0]); // we are accessing
printf("%d\n",a[1]);
printf("%d\n",a[2]);
printf("%d\n",a[3]);
printf("%d",a[4]);
return 0;
}
```

OUTPUT :-

```
2
3
5
7
11
--------------------------------
Process exited after 0.1476 seconds with return value 0
Press any key to continue . . .
```

➔Advantages of Arrays

- Arrays store multiple elements of the same type with the same name.
- You can randomly access elements in the array using an index number.
- Array memory is predefined, so there is no extra memory loss.
- Arrays avoid memory overflow.
- 2D arrays can efficiently represent the tabular data.

➔Disadvantages of Arrays

- The number of elements in an array should be predefined
- An array is static. It cannot alter its size after declaration.
- Insertion and deletion operation in an array is quite tricky as the array stores elements in continuous form.
- Allocating excess memory than required may lead to memory wastage

➔Basic Operations in the Arrays

The basic operations in the Arrays are insertion, deletion, searching, display, traverse, and update. These operations are usually performed to either modify the data in the array or to report the status of the 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.
- **Display** – Displays the contents of the array.

➔Traversal Operation

This operation traverses through all the elements of an array. We use loop statements to carry this out.

Example:-

```
public class ArrayDemo {
   public static void main(String []args) {
      int A[] = new int[5];
      System.out.println("The array elements are: ");
      for(int i = 0; i < 5; i++) {
         A[i] = i + 2;
         System.out.println("A[" + i + "] = " + A[i]);
      }
   }
}
```

OUTPUT:-

```
The array elements are:
A[0] = 2
A[1] = 3
A[2] = 4
A[3] = 5
A[4] = 6
```

➔Insertion Operation

In the insertion operation, we are adding one or more elements to the array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. This is done using input statements of the programming languages.

Example:-

```
public class ArrayDemo {
   public static void main(String []args) {
      int A[] = new int[3];
      System.out.println("Array Before Insertion:");
      for(int i = 0; i < 3; i++)
         System.out.println("A[" + i + "] = " + A[i]); //prints
empty array
      System.out.println("Inserting Elements..");
```

```
    // Printing Array after Insertion
    System.out.println("Array After Insertion:");
    for(int i = 0; i < 3; i++) {
      A[i] = i+3;
      System.out.println("A[" + i + "] = " + A[i]);
    }
  }
}
```

OUTPUT:-

```
Array Before Insertion:
A[0] = 0
A[1] = 0
A[2] = 0
Inserting Elements..
Array After Insertion:
A[0] = 3
A[1] = 4
A[2] = 5
```

➔Deletion Operation

In this array operation, we delete an element from the particular index of an array. This deletion operation takes place as we assign the value in the consequent index to the current index.

Example:-

```
public class ArrayDemo {
  public static void main(String []args) {
    int A[] = new int[3];
    int n = A.length;
    System.out.println("Array Before Deletion:");
    for(int i = 0; i < n; i++) {
      A[i] = i + 3;
      System.out.println("A[" + i + "] = " + A[i]);
    }
    for(int i = 1; i<n-1; i++) {
      A[i] = A[i+1];
      n = n - 1;
    }
    System.out.println("Array After Deletion:");
    for(int i = 0; i < n; i++) {
      System.out.println("A[" + i + "] = " + A[i]);
    }
  }
}
```

OUTPUT:-

```
Array Before Deletion:
A[0] = 3
A[1] = 4
A[2] = 5
Array After Deletion:
A[0] = 3
A[1] = 5
```

➔Search Operation

Searching an element in the array using a key; The key element sequentially compares every value in the array to check if the key is present in the array or not.

Example:-

```
public class ArrayDemo{
   public static void main(String []args){
      int A[] = new int[5];
      System.out.println("Array:");
      for(int i = 0; i < 5; i++) {
         A[i] = i + 3;
         System.out.println("A[" + i + "] = " + A[i]);
      }
      for(int i = 0; i < 5; i++) {
         if(A[i] == 6)
            System.out.println("Element " + 6 + " is found at index " + i);
      }
   }
}
```

OUTPUT:-

```
Array:
A[0] = 3
A[1] = 4
A[2] = 5
A[3] = 6
A[4] = 7
Element 6 is found at index 3
```

➔Update Operation

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

Example:-

```
public class ArrayDemo{
   public static void main(String []args) {
      int A[] = new int[5];
      int item = 15;
      System.out.println("The array elements are: ");
      for(int i = 0; i < 5; i++) {
         A[i] = i + 2;
         System.out.println("A[" + i + "] = " + A[i]);
      }
      A[3] = item;
      System.out.println("The array elements after updation are: ");
      for(int i = 0; i < 5; i++)
         System.out.println("A[" + i + "] = " + A[i]);
   }
}
```

OUTPUT:-

```
The array elements are:
A[0] = 2
A[1] = 3
A[2] = 4
A[3] = 5
A[4] = 6
The array elements after updation are:
A[0] = 2
A[1] = 3
A[2] = 4
A[3] = 15
A[4] = 6
```

➔Display Operation

This operation displays all the elements in the entire array using a print statement.

Example:-

```
public class ArrayDemo {
   public static void main(String []args) {
      int A[] = new int[5];
      System.out.println("The array elements are: ");
      for(int i = 0; i < 5; i++) {
         A[i] = i + 2;
         System.out.println("A[" + i + "] = " + A[i]);
      }
   }
}
```

OUTPUT:-

```
The array elements are:
A[0] = 2
A[1] = 3
A[2] = 4
A[3] = 5
A[4] = 6
```

Unlock Your Potential With Topperworld

LEARN MORE

www. topperworld.in

FOLLOW US!