**MODULE -1**

**Introduction to Data Structures:**

**What is Data:**

* Data refers to raw information that consists of basic facts and figures.
* A data item refers to a single unit of values.
* Data can come in the form of text, figures, images, numbers, graphs, or symbols.
* For example, data might include individual prices, weights, addresses, ages, names, temperatures, dates, or distances, etc.,.

**What is Information:**

* Information is defined as classified or organized data that has some meaningful value for the user.
* Essentially, information is the result of analyzing and interpreting pieces of data.
* Information is also the processed data used to make decisions and take action.

**The Key Differences Between Data vs Information**

* Data is a collection of facts, while information puts those facts into context.
* While data is raw and unorganized, information is organized.
* Data points are individual and sometimes unrelated. Information maps out that data to provide a big-picture view of how it all fits together.
* Data, on its own, is meaningless. When it’s analyzed and interpreted, it becomes meaningful information.
* Data does not depend on information; however, information depends on data.
* Data isn’t sufficient for decision-making, but you can make decisions based on information.

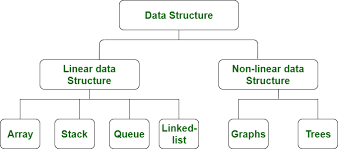
**Data Structure:**

* In computer science, a ***data structure*** is a particular way of storing and organizing data in a computer’s memory so that it can be used efficiently.
* Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a ***data structure***.
* The choice of a particular data model depends on the two considerations:
  + First- it must be rich enough in structure to mirror the actual relationships of the data in the real world.
  + On the other hand, the structure should be simple enough that one can effectively process the data whenever necessary.
  + To develop a program of an algorithm we should select an appropriate data structure for that algorithm.

**Categories of Data Structure:**

The data structure can be classified in to major types:

1. Linear Data Structure – for Example Arrays, Stacks, Queues and Linked list
2. Non-linear Data Structure- for example Trees and Graphs



*Fig 1.1 Categories of Data Structure*

***1. Linear Data Structure:***

* A linear data structure is a type of data structure that stores the data linearly or sequentially.
* In the linear data structure, data is arranged in such a way that one element is adjacent to its previous and the next element.
* It includes the data at a single level such that we can traverse all data into a single run.
* The implementation of the linear data structure is always easy as it stores the data linearly.
* The common examples of linear data types are Array , Stack, Queue, and Linked List etc.

1. **Arrays:** An array is a collection of elements or values of same type.

**Example:** int mark[5] = {40, 60, 80, 70, 90}

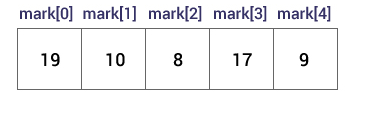


Fig1.2 Array Representation

1. **Stack:** Stack is a linear data structure which follows a particular order in which the operations are performed. The order is Last In First Out(LIFO). The following are the basic operations in stack:

* **Push:**Adds an item in the stack.
* **Pop:** Removes an item from the stack.

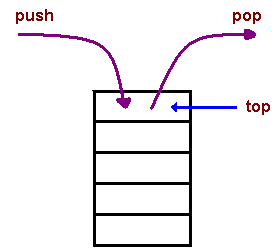


Fig 1.3 Stack Representation

1. **Queue:** Like [Stack](http://quiz.geeksforgeeks.org/stack-set-1/), [Queue](http://en.wikipedia.org/wiki/Queue_%28data_structure%29)is a linear structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO).

The following are the four basic operations are performed on queue:

**Enqueue:**Adds an item to the queue.

**Dequeue:** Removes an item from the queue

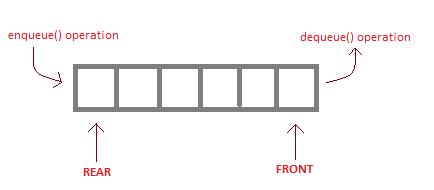
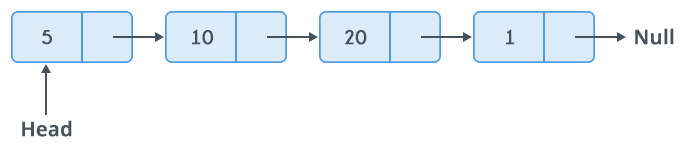


Fig 1.4 Queue Representation

1. **Linked list:** A **linked list** is a way to store a collection of elements. Each element in a linked list is stored in the form of a **node**. A **data** part stores the element and a **next** part stores the link to the next node.



Fig 1.5 Node Representation



**Fig 1.6 Linked List** Representation

**Importance of Linear Data Structure:**

* **Sequential Organization:** In linear data structures, data elements are arranged sequentially, one after the other. Each element has a unique predecessor (except for the first element) and a unique successor (except for the last element)
* **Fixed or Dynamic Size**: Linear data structures can have either fixed or dynamic sizes. Arrays typically have a fixed size when they are created, while other structures like linked lists, stacks, and queues can dynamically grow or shrink as elements are added or removed.
* **Simple implementation:** Their sequential nature makes them easy to understand and implement in various programming languages. Linear data structures work well mainly in the development of application software.
* **Efficient access:** Accessing specific elements is often quick and efficient, especially when using the element's index or position in the sequence. A user can find all of the data elements at a single level in a linear data structure. You can traverse a linear data structure in a single run.

**2. Non-linear Data Structure:**

* Non-Linear Data Structure is another important type Data structure in which data elements are not arranged linearly or sequentially.
* Data elements are present at the multilevel, for example, trees.
* In trees, the data elements are arranged in the hierarchical form, whereas in graphs, the data elements are arranged in random order, using the edges and vertex.
* Since the data structure is non-linear, it does not involve a single level.
* Therefore, a user can't traverse all of its elements in a single run. Multiple runs are required to traverse through all the elements completely.
* The data elements are present at different levels. In Non-linear data structures, there are different paths for an element to reach the other element.
* This structure is mainly used to represent data containing a hierarchical relationship between elements. **e.g. Trees,Graphs**,and T**able of contents.**

**Tree:**

* A tree data structure consists of various nodes linked together. The structure of a tree is hierarchical that forms a relationship like that of the parent and a child.
* A tree is a non-linear data structure that consists of a root node and potentially many levels of additional nodes that form a hierarchy.
* A tree can be empty with no nodes called the **null** or empty tree or a tree is a structure consisting of one node called the **root** and one or more subtrees.

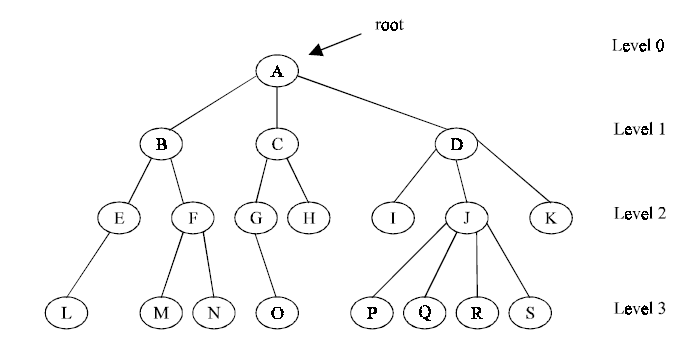


Fig1.7: Representation of Tree

**Graph :**

* A *graph* is a collection of nodes called *vertices*, and the connections between them, called *edges*.
* A graph data structure consists of a finite (and possibly mutable) [set](http://en.wikipedia.org/wiki/Set_(computer_science)) of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**
* For Example :The following diagram shows a graph with 5 vertices and 7 edges. The edges between A and D and B and C are pairs that make a bidirectional connection, represented here by a double-headed arrow.

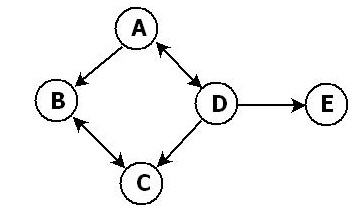
****

Fig 1.8 :Representation of Graph

***Difference between Linear Data Structure and Non-Linear Data Structure:***

| **S.NO** | **Linear Data Structure** | **Non-linear Data Structure** |
| --- | --- | --- |
| 1. | In a linear data structure, data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent. | In a non-linear data structure, data elements are attached in hierarchically manner. |
| 2. | In linear data structure, single level is involved. | Whereas in non-linear data structure, multiple levels are involved. |
| 3. | Its implementation is easy in comparison to non-linear data structure. | While its implementation is complex in comparison to linear data structure. |
| 4. | In linear data structure, data elements can be traversed in a single run only. | While in non-linear data structure, data elements can’t be traversed in a single run . |
| 5. | In a linear data structure, memory is not utilized in an efficient way. | While in a non-linear data structure, memory is utilized in an efficient way. |
| 6. | Its examples are: array, stack, queue, linked list, etc. | While its examples are: trees and graphs. |
| 8. | Linear data structures are useful for simple data storage and manipulation. | Non-linear data structures are useful for representing complex relationships and data hierarchies, such as in social networks, file systems, or computer networks. |

**Operations on Data Structures:**

* 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 in the frequency with which specific operations are performed.

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

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

**Abstract Data Type model:**

* The word Abstract is the process of hiding the internal details of an application from the outer world and only show the needed information.
* The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
* Abstract data types, commonly abbreviated ADTs, are a way of classifying data structures based on how they are used and the behaviours they provide.
* It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

**ADT model:**

A representation of the ADT model is shown in the following figure.

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

A diagram of data structure

Description automatically generated

Fig 1.9 ADT Model

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

# **Specifications for ADTs**

Several criteria can be used to judge the fitness of an ADT specification.

* **Formal.** Specifications must be formal. The meaning of each element in the specification must be defined in enough detail.
* **Applicable.** ADTs should be widely applicable. An ADT should be generally reusable for many different concrete use-cases. ADTs are best suited to describe the behavior of common data structures, library components, modules, programming language features, etc.
* **Minimal.** ADT specifications should be minimal. The specification should include the interesting and widely applicable parts of the behavior and nothing more. Each behavior should be described precisely and unambiguously, but in as little specific or concrete detail as possible.
* **Extensible.** ADTs should be extensible. A small change in a requirement should lead to only a small change in the specification.
* **Declarative.** Declarative specifications describe what, not how. ADTs should be described in terms of what things are, and relationship mappings between inputs and outputs.

**Implementation Of ADT**

* An implementation of ADT consists of data structures to store the data items and algorithms for basic operation. All the data structures i.e. List, stack, queue etc are examples of ADT.

**1. Array ADT**

* An array is probably the most versatile or fundamental Abstract Data Type.. An **array** is a finite sequence of storage cells, for which the following operations are defined:
* create(A,N) -> creates an array A with storage for N items;
* A[i]=item -> stores item in theithposition in the arrayA; and
* Item = A[i] - > returns the value of the item stored in the ith position in the array A.

A green line with numbers

Description automatically generated

*Fig 1.10 View of Array*

**Function in Array ADT**

* create(A,N) : creates an array ‘A’ with storage for ‘N’ items;
* bool isEmpty(); - Retrurns true if array is empty
* bool isFull(); - Retrurns true if array is full
* int length(); - Returns the length of the array
* void add(int item,int pos) – Adds the element at the given position
* void del(int item); - Deletes the given item from the array.
* void display(); - Prints the elements of array.

***2.Linked List ADT***

* Linked List is an Abstract Data Type (ADT) that holds a collection of Nodes, the nodes can be accessed in a sequential way.
* When the Nodes are connected with only the next pointer the list is called Singly Linke List. A linked list is a series of connected nodes, where each node is a data structure.



Fig 1.11 Representation of Node in Linked List

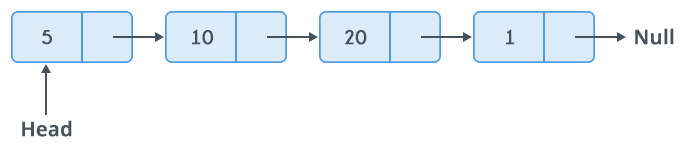


Fig 1.12 View of Linked List

* The data node contains the *pointer* to a data structure and a *self-referential pointer* which points to the next node in the list.

**Functions in Linked Lis ADT**

* 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.

1. **Stack ADT**

* A stack is an Abstract Data Type(ADT) that serves as a collection of elements with two main operations: Push, which adds an element to the collection, and. Pop, which removes the most recently added element.

A green rectangular object with numbers

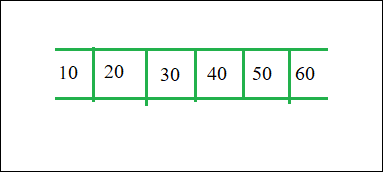
Description automatically generated

*Fig 1.13 View of stack*

**Function in Stack ADT**

* 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.

**4. Queue ADT**



*Fig 1.14 View of Queue*

* The queue abstract data type (ADT) follows the basic design of the stack abstract data type.
* A queue is an object (an abstract data structure - ADT) that allows the following operations: Enqueue: Add an element to the end of the queue. Dequeue: Remove an element from the front of the queue.

**Function in Queue ADT**

* 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(or Advantages) 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.

### **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.

### **Overview of time and space complexity**

* Generally, there is always more than one way to solve a problem in computer science with different algorithms.
* Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal.
* There are two such methods used, [**time complexity**](https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/) and [**space complexity**](https://www.geeksforgeeks.org/g-fact-86/)

**Time complexity**

* Time complexity refers to the amount of time required by an algorithm to run as a function of the input size.
* The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called **time complexity** of the algorithm.
* Time complexity is very useful measure in algorithm analysis.
* To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

**How to Calculate Time Complexity?**

* Analysing the growth rate of an algorithm's running time as input size grows is necessary to determine how time-complex it is.
* It gives an estimate of how the algorithm performs as the input size increases. Here are the general steps to calculate time complexity:
* ***Identify the algorithm:*** Determine the specific algorithm for which you want to calculate the time complexity. It could consist of a series of operations combined with a loop or a recursive function.
* ***Identify the input size:*** Identify the elements that make up the algorithm's input size. For example, if the algorithm operates on an array, the input size could be the length of the array.
* ***Determine the basic operations:*** Identify the fundamental operations that contribute to the running time of the algorithm. These operations could be comparisons, assignments, arithmetic operations, function calls, or any other significant actions.
* ***Count the operations:*** Analyze how many times each basic operation is executed as a function of the input size. You may need to consider different scenarios or branches within the algorithm.
* ***Express the count as a function of the input size***: Create a mathematical expression that represents the count of basic operations as a function of the input size.
* ***Simplify the expression:*** Simplify the mathematical expression by removing constants, lower-order terms, and insignificant factors. Focus on the most dominant term that represents the growth rate of the algorithm.
* ***Determine the time complexity notation:*** Use Big O notation to express the condensed expression.

***ASYMPTOTIC NOTATIONS***

* + **Asymptotic notations**  are used to analyze the performance of an algorithm for some large data set.
  + **Asymptotic notations** are used to write fastest and slowest possible running time for an algorithm.

***Why is Asymptotic Notation Important?***

1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

Usually, the time required by an algorithm falls under three types −

* **Best Case** − Minimum time required for program execution.
* **Average Case** − Average time required for program execution.
* **Worst Case** − Maximum time required for program execution.

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

* Ο Notation
* Ω Notation
* θ Notation

***Big Oh Notation, Ο***

* The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

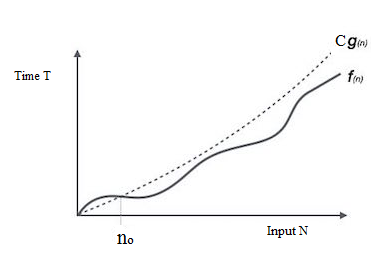


Fig 1.15 Big O Notation

* If **f(n) <= C g(n)**for all **n >= n0**, **C > 0** . Then we can represent **f(n)** as **O(g(n))**.

# **f(n) = O(g(n))** Hence, function g (n) is an upper bound for function f (n), as g (n) grows faster than f (n)

* + ***Examples:*** 
    - *(i). 3n+2 = 0(n) as 3n+2 < 4n for all n > 2.  
      (ii). 3n + 3 = O(n) as 3n + 3 < 4n for all n > 3.*

***Omega Notation, Ω***

* The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

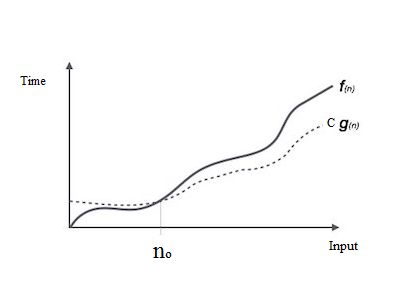


Fig 1.16 Omega **Ω** Notation

* If **f(n) >= C g(n)**for all **n >= n0**, **C > 0**. Then we can represent **f(n)** as **Ω(g(n))**.
* **f(n) = Ω(g(n)) ,** Hence, function g (n) is an lower bound for function f (n).
  + - Examples:
      * 
      * 

***Theta Notation, θ***

* The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

A graph of a curve

Description automatically generated

Fig 1.17 Theta **Θ** Notation

* If **C1 g(n) <= f(n) >= C2 g(n)**for all **n >= n0**, **C1, C2 > 0**. Then we can represent **f(n)** as **Θ(g(n))**.
* **f(n) = Θ(g(n))**

***Common Time Complexities***

Understanding the common time and space complexities of various data structures and algorithms is essential for efficient problem-solving. Here are some examples:

### ***Time Complexities:***

|  |  |
| --- | --- |
| **Name** | **Time complexity** |
| Constant Time | O(1) |
| Logarithmic Time | O(log n) |
| Linear Time | O (n) |
| Quadratic Time | O(n2) |
| Cubic Time | O(n3) |
| Exponential Time | O(2n) |

**Example1: Time Complexity Analysis of SUM algorithm**

A white rectangular object with black text

Description automatically generated

* Here 2n+3 is a Linear Equation , in this the dominant term is 2n,
* So, the Time Complexity of Sum of 1 to N numbers is O(n).

**Space complexity**

* The **space complexity** is the measurement of total space required by an algorithm to execute properly. It also includes memory required by input variables. Basically, it's the sum of auxiliary space and the memory used by input variables.
* Auxiliary Space is the extra space or temporary space used by an algorithm.
  + **Note:**Space complexity = Auxiliary space + Memory used by input variables
* It is good programming practice o know in advance whether the available memory space is sufficient for the program to b executed

***Example1: Space complexity for SUM algorithm***

*Sum(A,n)*

*{*

*S=0*

*For i=0 to n do*

*{*

*S = s + a[i]*

*}*

*Return S*

*}*

* Here array of length A[N] is  **N**, and variables s,i**, and n** are used in the algorithm
* so, the total space used is **N \* 2 + 1 \*2 +1 \* 2+1 \* 2 = 2N + 6**, (here 2 is a unit space taken by int data type).
* For many inputs, constant  is insignificant, and it can be said that the space complexity is **O(N)**.

## *Time Complexity of Different Data Structures:*

Here are the time complexities associated with common data structures:

### **Arrays:**

* Access: O(1)
* Search: O(n)
* Insertion (at the end): O(1)
* Insertion (at the beginning or middle): O(n)
* Deletion (from the end): O(1)
* Deletion (from the beginning or middle): O(n)

### **Linked Lists:**

* Access: O(n)
* Search: O(n)
* Insertion (at the beginning): O(1)
* Insertion (at the end, with a tail pointer): O(1)
* Insertion (at the end, without a tail pointer): O(n)
* Insertion (in the middle): O(n)
* Deletion (from the beginning): O(1)
* Deletion (from the end, with a tail pointer): O(1)
* Deletion (from the end, without a tail pointer): O(n)
* Deletion (from the middle): O(n)

### **Stacks:**

* Push: O(1)
* Pop: O(1)
* Peek: O(1)

### **Queues:**

* Enqueue: O(1)
* Dequeue: O(1)
* Peek: O(1)

### ***Importance of Analyzing Time and Space Complexity***

Analyzing the time and space complexity of data structures and algorithms is crucial for several reasons:

1. **Performance Evaluation:** Time and space complexity analysis helps in comparing different algorithms and choosing the most efficient one for a given problem.
2. **Scalability:** Understanding the growth rate of algorithms allows us to predict their behavior when dealing with large input sizes.
3. **Optimization:** By identifying the bottlenecks in an algorithm, we can optimize it to improve its efficiency and reduce resource consumption.
4. **Resource Planning:** Analyzing space complexity helps in estimating the memory requirements for executing an algorithm, allowing for better resource allocation.

***Arrays :***

* An array is a collection of elements of the same type that are referenced by a common name.
* An array elements are placed in contiguous memory locations that can be individually referenced by adding an index (or subscript) to a unique identifier.
* The lowest address corresponds to the first element, and the highest address to the last element.
* Usually, the array of characters is called a ‘string’, whereas an array of int or float is called simply an array. Arrays may have from one to several dimensions.
* A one-dimensional array is like a list;  A two dimensional array is like a table;

***One Dimensional Arrays :***

* A list of items or elements can be given one variable name using only one subscript and such a variable is called single subscripted variable or a ***one-dimensional array.***

***Array Declaration :***

* Like a regular variable, an array must be declared before it is used.
* The general form of single-dimension array declaration is:

***Type variable-name[size];***

* Here, ***type*** declares the base data type of the array (like int, float...), ***size*** (which is always enclosed in square brackets []), defines how many elements the array will hold. Variable-name is a valid identifier.
* for example, the following declares as integer array named sample that is ten elements long:

***int arr[5];***

* The range of index of an array is from 0 to size-1.
* In arrays 0is the index of their first element and siz-1 is the last element of an array. From the above example , arr array has ten elements, ie., arr[0] through arr[4].

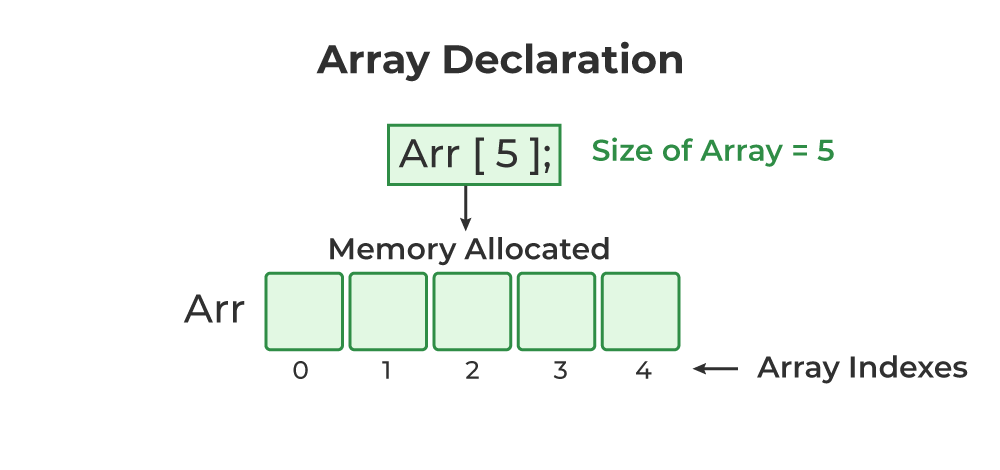


Fig 1.19 Array Representation

***Array Initialization :***

* Arrays may be initialized when they are declared, just as any other variables.
* Place the initialization data in curly {} braces following the equals sign.
* An array may be partially initialized, by providing fewer data items than the size of the array.  The remaining array elements will be automatically initialized to zero.

Following are a few examples that demonstrate this.

2 4 12 5 45

A[0]]

A[1]]

A[2]]

A[3]]

A[4]]

int arr[5] = { 2, 4, 812, 5, 45} ;

10 20 30 40 50 60

b[0]]

b[1]]

b[2]]

b[3]]

b[4]]

b[5]]

int b[ ] = { 10,20,30,40,50,60} ;

c[0]]

12.3 34.2 -23.4 0 0

c[1]]

c[2]]

c[3]]

c[4]]

float c[5] = { 12.3, 34.2 -23.4} ;

***Accessing the values of an array:***

* Elements of an array are accessed by specifying the index ( or subscript ) of the desired element within square [ ] brackets after the array name.
* Array Index/subscripts must be of integer type.
* Array indices start at zero in C, and go to one less than the size of the array(size-1).  For example, a five element array will have indices zero through four.
* Arrays are commonly used in conjunction with loops, in order to perform the same calculations on all ( or some part ) of  the data items in the array.

*The general format is :*

***Array-name[index]***

* Following the previous examples in which **‘a’** had 5 elements and each of those elements was of type int. For example, to store the value 75 in the third element of **‘a’**, we could write the following statement:

a[2] = 75;

* for example, to pass the value of the third element of ‘a’ to a variable called ‘x’, we could write:

x = a[2];

**Example 1:**

*/\* Read and display the 10 elements of array \*/*

*main()*

*{*

*int a[5],i;*

*clrscr();*

*printf(“\n Enter 5 Elements of an array….:”);*

*for (i=0; i<5; i++)*

*{*

*Scanf(“%d”,&a [i] );*

*}*

*printf(“\n The Elements of an array are ….:”);*

*for (i=0; i<5; i++)*

*{*

*Printf(“\na[%d] = %4d”,i,a[i]);*

*}*

*getch();*

*}*

***Sample Output :***

Enter 5 Elements of an array…: 5 10 15 20 25

The elements of an array are :

A[0]= 5 A[1]=10 A[2]=15 A[3]=20 A[4]=25

***Example: Reversing the Elements of Array***

* Suppose we have an array with *n* elements. We shall have to reverse the elements present in the [array](https://www.tutorialspoint.com/cprogramming/c_arrays.htm) and display them.

## Case 1: Reversing the Elements of the Array with Additional Array

## *#include <stdio.h>*

## *void main()*

## *{*

## *int a[50],rev[50], n,i,j;*

## *printf(" Enter the Size of the Array...:");*

## *scanf("%d",&n);*

## 

## *printf("\nEnter the Array Elements..:");*

## *for(i=0;i<n;i++)*

## *{*

## *scanf("%d", &a[i]);*

## *}*

## *// Array Elements Before Reversing*

## *printf("\nElements of Array - A..:\n");*

## *for(i=0;i<n;i++)*

## *{*

## *printf("%5d", a[i]);*

## *}*

## *// reversing the Array Elements*

## *for(j=0,i=n-1;j<n;i--,j++)*

## *{*

## *rev[j] = a[i];*

## *}*

## *// Array Elements After Reversing*

## *printf("\nReverse the Array-A Elements..:\n");*

## *for(i=0;i<n;i++)*

## *{*

## *printf("%5d", rev[i]);*

## *}*

## *getch();*

## *}*

## *Output:*

## *Enter the Size of the Array...:5*

## *Enter the Array Elements..:*

## *10 20 30 40 50*

## *Elements of Array - A..:*

## *10 20 30 40 50*

## *Reverse the Array-A Elements..:*

## *50 40 30 20 10*

## Case 2: Reversing the Elements of the Array with out Additional Array

## *#include <stdio.h>*

## *#include <stdlib.h>*

## *#define n 6*

## *int main()*

## *{*

## *int a[50], n,I,temp;*

## *printf(" Enter the Size of the Array...:");*

## *scanf("%d",&n);*

## 

## *printf("\nEnter the Array Elements..:");*

## *for(i=0;i<n;i++)*

## *{*

## *scanf("%d", &a[i]);*

## *}*

## *// Array Elements Before Reversing*

## *printf("\nElements of Array - A..:\n");*

## *for(i=0;i<n;i++)*

## *{*

## *printf("%5d", a[i]);*

## *}*

## *for(int i = 0; i<n/2; i++)*

## *{*

## *temp = a[i];*

## *arr[i] = a[n-i-1];*

## *a[n-i-1] = temp;*

## *}*

## 

## *// Array Elements After Reversing*

## *printf("\n\nReverse Elements of an Array...:\n");*

## *for(int i = 0; i < n; i++)*

## *{*

## *printf("%5d", a[i]);*

## *}*

## *return 0;*

## *}*

## Output:

## *Elements of an Array...:*

## *10 20 30 40 50*

## *Reverse Elements of an Array...:*

## *50 40 30 20 10*

## Multidimensional Array in C

* Multi-dimensional Arrays in C are those arrays that have more than one dimension.
* Some of the popular multidimensional arrays are 2D arrays and 3D arrays.
* We can declare arrays with more dimensions than 3d arrays but they are avoided as they get very complex and occupy a large amount of space.

***Two Dimensional Arrays :***

* Two dimensional arrays are arrays with two dimensions, namely rows and colums.
* Each element of the array is accessed using two subscripts, the first subscript indicating the row number and the second indicating the column number.
* A 2-D array can be thought of as a collection of a number of one dimensional arrays, each indicating a row.

***Declaration of two dimensional arrays:***

Two dimensional arrays can be declared as follows :

***Type variable\_name[row\_size][column\_size];***

Here, ***type*** declares the base type of the array (like int, float...), row\_size (which is always enclosed in square brackets []) defines the number of rows in an array and column\_size (which is always enclosed in square brackets []) defines the number of columns in an array. Variable-name is a valid identifier.

Example:

To declare two-dimensional integer array num of size (3,4),we write:

***int num[3][4];***

Left Index determines row Right index determines column

* Two dimensional arrays are stored in a row-column matrix where the first index indicates the row and the second indicates the column.
* This means that the right most index changes faster than the leftmost when accessing the elements in the array in the order in which they are actually stored in memory.

***Initializing a 2-Dimensional Array:***

* Like the one-dimensional array, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces as shown below:

Type variable-name[row\_size][colum\_size]={List of values};

For example:

*Int a[2][3]={1,2,3,4,5,6};*

This statement initializes the elements of first row to 1,2 and 3 and second row to 4,5 and 6. The inilization is done row by row.

* We can also intilize a two-dimensional array in the form of a matrix as shown below :

Int a[2][3]={ {1,2,3},{4,5,6} };

* It is important to remember that while initializing a 2-D array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Thus the declarations:

int a[2][3] = { 1,2,3,4,5,6} ;

int a[ ][3] = { 1,2,3,4,5,6} ; are perfectly acceptable, whereas,

int a[2][ ] = { 12, 34, 23, 45, 56, 45 } ;

int a[ ][ ] = { 12, 34, 23, 45, 56, 45 } ; would never work.

* If the values are missing in an initialize, they are automatically set to zero.

For example

Int a[2][3]={ {1,2},{4}};

* + This statement initializes the first two elements of the first row to 1 and 2, the first element of the second row to 4, and all other elements to zero.

## *Access the Elements of a 2D Array*

* To access an element of a two-dimensional array, you must specify the index number of both the row and column.
* This statement accesses the value of the element in the **first row (0)** and **third column (2)** of the **matrix** array.

### Example

int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };  
  
printf("%d", matrix[0][2]);  // Outputs 2

## *Loop Through a 2D Array*

To loop through a multi-dimensional array, you need one loop for each of the array's dimensions.

The following example outputs all elements in the **matrix** array:

### *Example*

*main()*

*{*

*int matrix[2][3],I,j;*

*printf(“Enter Elements of the Matrix…:”);*

*for (i = 0; i < 2; i++) {  
   for (j = 0; j < 3; j++) {  
     scanf("%d", &matrix[i][j]);*

*}*

*}*

*for (i = 0; i < 2; i++) {  
  for (j = 0; j < 3; j++) {  
    printf("%d", matrix[i][j]);*

*}*

*Printf(“\n”);*

*}*

*}*

***Storage representation of 2-D dimensional arrays in memory:***

* When speaking of 2-D array, we are logically saying that it consists of rows and columns but when it is stored in memory , no facility for two dimensional storage is available.
* All the elements of two dimensional array are stored in leaner or sequential memory locations.
* The elements of an array can be referred using two indices, one for attaining row number and another for obtaining column number.

There are two major types of representations can be used for 2-D arrays.

1. ***Row major representation***
2. ***Column major representation***

***1.Row major representation:***

In Row major representation, all the elements are stored in contiguous memory locations ie., the elements of 1st row are stored in continuous memory locations followed by elements of 2nd row and so on.

The row major representation of 2-D array elements is illustrated in following example.

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| 0 |
| 1 |
| 2 |
| 3 |
| 0 |
| 1 |
| 2 |
| 3 |

|  |
| --- |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
| 10 |
| 11 |
| 12 |

|  |
| --- |
| 500 |
| 502 |
| 504 |
| 506 |
| 508 |
| 510 |
| 512 |
| 514 |
| 516 |
| 518 |
| 520 |
| 522 |

For example, consider the array a[3][4]:

0 1 2 3

Row1

0

1

2

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |

Row2

Row3

The Row major representation of 2-D array elements

Here 500 is the base address of the array and each element is stored at a distance of 2 bytes because each integer element requires 2 bytes of storage.

***Accessing address of 2-D array elements :***

If we know the base address( ie.,1st element address) of an array, we can access any element of that array.

For example the array a[m][n], then address of a[i][j] can be calculated as shown below :

***Base address + (i\*n + j) \* size of an array element***

From the above example, the address of a[1][3] is :

500 + (1\*4+3) \* 2 = 514 where n=4

***2.Column major representation:***

In Column major representation, all the elements are stored in contiguous memory locations ie., the elements of 1st column are stored in continuous memory locations followed by elements of 2nd column and so on.

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 0 |
| 1 |
| 2 |
| 0 |
| 1 |
| 2 |
| 0 |
| 1 |
| 2 |

|  |
| --- |
| 1 |
| 5 |
| 9 |
| 2 |
| 6 |
| 10 |
| 3 |
| 7 |
| 11 |
| 4 |
| 8 |
| 12 |

|  |
| --- |
| 500 |
| 502 |
| 504 |
| 506 |
| 508 |
| 510 |
| 512 |
| 514 |
| 516 |
| 518 |
| 520 |
| 522 |

For example, consider the array a[3][4]:

Column1

0 1 2 3

0

1

2

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |

Column2

Column3

Column4

The Column major representation of 2-D array elements

Here 500 is the base address of the array and each element is stored at a distance of 2 bytes because each integer element requires 2 bytes of storage.

***Accessing address of 2-D array elements :***

If we know the base address( ie.,1st element address) of an array, we can access any element of that array.

For example the array a[m][n], then address of a[i][j] can be calculated as shown below :

Base address +(i + j \*m) \* size of an array element

From the above example, the address of a[1][3] is :

500 + (1+3\*3) \* 2 = 520 where m=3

***Example 1 :*** *Reading and printing 2-D array elements :*

*#include <stdio.h>*

*#include<conio.h>*

*void main()*

*{*

*int a[3][3], i,j;*

*clrscr();*

*printf("\nEnter Elements of 2-D Array....");*

*for(i=0;i<3;i++)*

*{*

*for(j=0;j<3;j++)*

*{*

*scanf("%d",&a[i][j]);*

*}*

*}*

*printf("\nThe Elements of 2-D Array are …:\n");*

*for(i=0;i<3;i++)*

*{*

*for(j=0;j<3;j++)*

*{*

*printf("%4d",a[i][j]);*

*}*

*printf("\n");*

*}*

*getch();*

*}*

***Output:***

*Enter Elements of 2-D Array....1 2 3 4 5 6 7 8 9*

*The Elements of 2-D Array are …:*

*1 2 3*

*4 5 6*

*7 8 9*

***Example 2:*** *program to read the elements of two matrices and find addition of two matrices*

*#include <stdio.h>*

*void main()*

*{*

*int a[5][5], b[5][5],c[5][5];*

*int m,n,i,j;*

*clrscr();*

*printf(“\nEnter no. of Rows and No. of Columns (Order) of First Matrix…:”);*

*scanf("%d%d",&m,&n);*

*printf("\n Enter Elements of A Matrix…:");*

*for(i=0;i<m;i++)*

*{*

*for(j=0;j<n;j++)*

*{*

*scanf("%d",&a[i][j]);*

*}*

*}*

*printf("\n Enter Elements of B Matrix…:");*

*for(i=0;i<m;i++)*

*{*

*for(j=0;j<n;j++)*

*{*

*scanf("%d",&b[i][j]);*

*}*

*}*

*for(i=0;i<m;i++)*

*{*

*for(j=0;j<n;j++)*

*{*

*c[i][j]=a[i][j]+b[i][j];*

*}*

*}*

*printf("\nThe Addition of Two Matrices is …:\n");*

*for(i=0;i<m;i++)*

*{*

*for(j=0;j<n;j++)*

*{*

*printf("%4d",c[i][j]);*

*}*

*printf("\n");*

*}*

*getch();*

*}*

***Output:***

*Enter no. of Rows and No. of Columns (Order) of First Matrix…:2 2*

*Enter Elements of A Matrix…:1 0 2 3*

*Enter Elements of B Matrix…:2 1 0 2*

*The Addition of Two Matrices is …:*

*3 1*

*2 5*

***Example 3:*** *program to read the elements of two matrices and find Multiplication of two matrices*

*#include <stdio.h>*

*void main()*

*{*

*int a[5][5], b[5][5],c[5][5],m,n,p,q,i,j,k;*

*//clrscr();*

*printf(“\nEnter no. of Rows and No. of Columns (Order) of A Matrix…:”);*

*scanf("%d%d",&m,&n);*

*printf("\nEnter no. of Rows and No. of Columns (Order) of A Matrix…:");*

*scanf("%d%d",&p,&q);*

*if(n==p)*

*{*

*printf("\n Enter Elements of A Matrix…:");*

*for(i=0;i<m;i++)*

*{*

*for(j=0;j<n;j++)*

*{*

*scanf("%d",&a[i][j]);*

*}*

*}*

*printf("\n Enter Elements of B Matrix…:");*

*for(i=0;i<p;i++)*

*{*

*for(j=0;j<q;j++)*

*{*

*scanf("%d",&b[i][j]);*

*}*

*}*

*for(i=0;i<m;i++)*

*{*

*for(j=0;j<q;j++)*

*{*

*c[i][j]=0;*

*for(k=0;k<p;k++)*

*{*

*c[i][j]=c[i][j]+ (a[i][k]\*b[k][j]);*

*}*

*}*

*}*

*printf("\n The Multiplication of Two Matrices is …:\n");*

*for(i=0;i<m;i++)*

*{*

*for(j=0;j<q;j++)*

*{*

*printf("%4d",c[i][j]);*

*}*

*printf("\n");*

*}*

*}*

*else*

*{*

*printf("\nMultiplication of two matrices is not possible…:");*

*}*

*//getch();*

*}*

***Output:***

*Enter no. of Rows and No. of Columns (Order) of A Matrix…:2 2*

*Enter no. of Rows and No. of Columns (Order) of A Matrix…:2 2*

*Enter Elements of A Matrix…:1 2 2 1*

*Enter Elements of B Matrix…:1 0 2 1*

*The Multiplication of Two Matrices is …:*

*5 2*

*4 1*

***Example 4: Transpose matrix of given Matrix***

*#include <stdio.h>*

*#define N 4*

*// This function stores transpose of A[][] in B[][]*

*void transpose(int A[][N], int B[][N])*

*{*

*int i, j;*

*for (i = 0; i < N; i++)*

*for (j = 0; j < N; j++)*

*B[i][j] = A[j][i];*

*}*

*// Driver code*

*int main()*

*{*

*int a[N][N] = { { 1, 1, 1, 1 },*

*{ 2, 2, 2, 2 },*

*{ 3, 3, 3, 3 },*

*{ 4, 4, 4, 4 } };*

*// Note dimensions of B[][]*

*int b[N][N], i, j;*

*// Function call*

*printf("A matrix is \n");*

*for (i = 0; i < N; i++) {*

*for (j = 0; j < N; j++)*

*printf("%d ", a[i][j]);*

*printf("\n");*

*}*

*transpose(a, b);*

*printf("Transpose matrix of A is..: \n");*

*for (i = 0; i < N; i++) {*

*for (j = 0; j < N; j++)*

*printf("%d ", b[i][j]);*

*printf("\n");*

*}*

*return 0;*

*}*

***Output:***

*A matrix is*

*1 1 1 1*

*2 2 2 2*

*3 3 3 3*

*4 4 4 4*

*Transpose matrix of A is..:*

*1 2 3 4*

*1 2 3 4*

*1 2 3 4*

*1 2 3 4*