Data Structure

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science, for example: Operating System, Compiler Design, Artificial intelligence, Graphics etc.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminologies are used in data structures:

Data

Data can be defined as an elementary value or the collection of values. ***for example***: student's **name** and its **id** are the data about the student.

Group Items

Data items which have subordinate data items are called Group item, ***for example***: name of a student can have first name and the last name.

Record

Record can be defined as the collection of various data items, ***for example***: if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File

A File is a collection of various records of one type of entity, ***for example***: if there are 60 students in the class, then there will be 20 records in the related file where each record contains the data about each student.

Attribute and Entity

An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field

Field is a single elementary unit of information representing the attribute of an entity.

Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:

Processor speed

To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search
Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests
If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process.

In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

Advantages of Data Structures
Efficiency
Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability
Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction
Data structure is specified by the ADT (Abstract Data Type) which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

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

1) Linear Data Structures
2) Non - Linear Data Structures

1) Linear Data Structures
A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

In other words, we can say that If a data structure organizes the data in sequential order, then that data structure is called a Linear Data Structure. Example:

- Arrays
- List (Linked List)
- Stack
- Queue

Arrays
An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional. The individual elements of the array age are:
age[0], age[1], age[2], age[3],......... age[98], age[99].

Linked List

Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack

Stack is a linear list in which insertion and deletions are allowed only at one end, called top. A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue

Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front.
It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

2) Non - Linear Data Structures

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure. Simply we can say that If a data structure organizes the data in random order, then that data structure is called as Non-Linear Data Structure. Example:

- Tree
- Graph
- Dictionaries
- Heaps
- Tries, Etc.

Trees

Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called leaf node while the topmost node is called root node. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

Graphs

Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Operations on Data Structure

1. Traversing

Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subjects, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

2. Insertion

Insertion can be defined as the process of adding the elements to the data structure at any location. If the size of data structure is n then we can only insert n-1 data elements into it.

3. Deletion

The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then underflow occurs.

4. Searching

The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5. Sorting

The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6. Merging

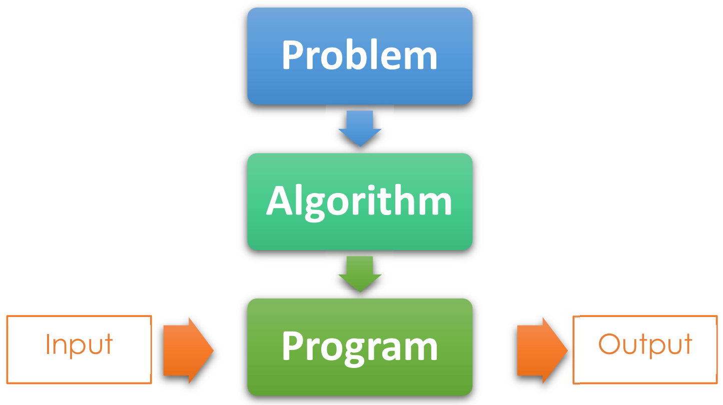
When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

Algorithm

An algorithm is a step by step procedure to solve a problem. In normal language, the algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which form a program. This program is executed by a computer to produce a solution. Here, the program takes required data as input, processes data according to the program instructions and finally produces a result as shown in the following picture.



Categories of Algorithm

The major categories of algorithms are given below:

- **Sort**: Algorithm developed for sorting the items in certain order.
- **Search**: Algorithm developed for searching the items inside a data structure.
- **Delete**: Algorithm developed for deleting the existing element from the data structure.
- **Insert**: Algorithm developed for inserting an item inside a data structure.
- **Update**: Algorithm developed for updating the existing element inside a data structure.

Performance of Algorithm

The performance of algorithm is measured on the basis of following properties:

- **Time complexity**: It is a way of representing the amount of time needed by a program to run to the completion.
- **Space complexity**: It is the amount of memory space required by an algorithm, during a course of its execution. Space complexity is required in situations when limited memory is available and for the multi user system.

Each algorithm must have:

- **Specification**: Description of the computational procedure.
- **Pre-conditions**: The condition(s) on input.
- **Body of the Algorithm**: A sequence of clear and unambiguous instructions.
- **Post-conditions**: The condition(s) on output.

Example: Design an algorithm to multiply the two numbers **x** and **y** and display the result in **z**.

```
Step 1 START
Step 2 declare three integers x, y & z
Step 3 define values of x & y
Step 4 multiply values of x & y
Step 5 store the output of step 4 in z
Step 6 print z
Step 7 STOP
```

Characteristics of an Algorithm

An algorithm must follow the mentioned below characteristics:

- **Input:** Every algorithm must take zero or more number of input values from external.
- **Output**: Every algorithm must produce an output as result.
- **Definiteness**: Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation).
- **Finiteness**: For all different cases, the algorithm must produce result within a finite number of steps.
- **Effectiveness**: Every instruction must be basic enough to be carried out and it also must be feasible.

Performance Analysis of an algorithm

If we want to go from city "A" to city "B", there can be many ways of doing this. ***We can go by flight, by bus, by train and also by bicycle***. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows:

Performance of an algorithm is a process of making evaluative judgement about algorithms.

It can also be defined as follows:

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.

We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.

Generally, the performance of an algorithm depends on the following elements:

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.

When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.

Based on this information, performance analysis of an algorithm can also be defined as follows:

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures:

1) Space Complexity
2) Time Complexity

1) Space Complexity

Space required to complete the task of that algorithm. It includes program space and data space. When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes:

- To store program instructions.
- To store constant values.
- To store variable values.
- And for few other things like function calls, jumping statements etc.

Space complexity of an algorithm can be defined as follows:

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows:

Instruction Space: It is the amount of memory used to store compiled version of instructions.

Environmental Stack: It is the amount of memory used to store information of partially executed functions at the time of function call.

Data Space: It is the amount of memory used to store all the variables and constants.

Note - *When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack.* ***That means we calculate only the memory required to store Variables, Constants, Structures, etc.***

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following:

- 2 bytes to store Integer value.
- 4 bytes to store Floating Point value.
- 1 byte to store Character value.
- 6 (OR) 8 bytes to store double value.

Consider the following piece of code:

Example 1

```
int square(int a)
{
      return a*a;
}
```

In the above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.

That means, totally it requires 4 bytes of memory to complete its execution. And these 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be ***Constant Space Complexity***.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.

Consider the following piece of code:

Example 2

```
int sum(int A[ ], int n)
{
      int sum = 0, i;
      for(i=0; i<n; i++)
            sum=sum+A[i];
      return sum;
}
```

In the above piece of code, it requires:

- 'n*2' bytes of memory to store array variable 'a[ ]'
- 2 bytes of memory for integer parameter 'n'
- 4 bytes of memory for local integer variables 'sum' and 'i' (2 bytes each)
- 2 bytes of memory for return value.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the total amount of memory required depends on the value of 'n'. As 'n' value increases the space required also increases proportionately. This type of space complexity is said to be ***Linear Space Complexity***.

*If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be **Linear Space Complexity.***

2) Time Complexity

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity. The time complexity of an algorithm can be defined as follows:

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, the running time of an algorithm depends upon the following:

- Whether it is running on Single processor machine or Multi processor machine.
- Whether it is a 32-bit machine or 64-bit machine.
- Read and Write speed of the machine.
- The amount of time required by an algorithm to perform Arithmetic operations, logical operations, return value and assignment operations etc.
- Input data

Note - When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent. We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because the configuration changes from one system to another system. To solve this problem, we must assume a model machine with a specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

To calculate the time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration:

- It is a Single processor machine
- It is a 32-bit Operating System machine
- It performs sequential execution
- It requires 1 unit of time for Arithmetic and Logical operations
- It requires 1 unit of time for Assignment and Return value
- It requires 1 unit of time for Read and Write operations

Now, we calculate the time complexity of following example code by using the above-defined model machine. Consider the following piece of code:

Example 1

```
int sum(int a, int b)
{
    return a+b;
}
```

In the above sample code, it requires 1 unit of time to calculate **a+b** and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of **a** and **b**. That means for all input values, it requires the same amount of time i.e. 2 units.

If any program requires a fixed amount of time for all input values then its time complexity is said to be ***Constant Time Complexity.***

Consider the following piece of code:

Example 2

```
int sum(int A[], int n)
{
int sum=0,i;
    for(i=0;i<n;i++)
        sum=sum+A[i];
        return sum;
}
```
For the above code, time complexity can be calculated as follows:

- Cost is the amount of computer time required for a single operation in each line.
- Repetition is the amount of computer time required by each operation for all its repetitions.
- Total is the amount of computer time required by each operation to execute.
- So above code requires **'4n+4'** Units of computer time to complete the task. Here the exact time is not fixed. And it changes based on the n value. If we increase the n value then the time required also increases linearly.

Totally it takes **'4n+4'** units of time to complete its execution and it is ***Linear Time Complexity***.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be ***Linear Time Complexity***.

Time Space Trade Off

In computer science, a space-time or time-memory tradeoff is a way of solving a problem in:

- Less time by using more memory) or,
- By solving a problem in very little space by spending a long time.

Types of Trade Off

1) Compressed / Uncompressed Data
2) Re-Rendering / Stored Images
3) Smaller Code / Loop Unrolling
4) Lookup Table / Recalculation

Compressed / Uncompressed Data

A space -time trade off can be applied to the problem of data storage, if data is stored uncompressed, it takes more space but less time. And if the data is stored compressed, it takes less space but more time to run the decompression algorithm.

Re-Rendering / Stored Images

Storing only the source and rendering it as an image every time the page is requested would be trading time for space. **It will take more time but less space.** Storing the images would be trading space for time. **It will take more space but less time.**

Smaller Code (with loop) / Larger Code (without loop)

Smaller code occupies **less space** in memory but it requires **high computation** time which is required for jumping back to the beginning of the loop at the end of each iteration.

Larger code or loop unrolling can be traded for higher program speed. It occupies **more space** in memory but requires **less computation time**.

Lookup Table / Recalculation

In lookup table, an implementation can include the entire table which reduces computing time but increases the amount of memory needed. It can recalculate i.e. compute table entries as needed, increasing computing time but reducing memory requirements.

Example: More time, Less space

```
    int a,b;
    printf("enter value of a \n");
    scanf("%d",&a);
    printf("enter value of b \n");
    scanf("%d",&b);
    b=a+b;
    printf("output is:%d",b);
```

Example: More Space, Less time

```
    int a,b,c;
    printf("enter value of a,b and c");
    scanf("%d%d%d",&a,&b,&c);
    printf("output is:%d",c=a+b);
```

Pointer

Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at the location is known as dereferencing the pointer. Pointer improves the performance for repetitive process such as:

1) Traversing String
2) Lookup Tables
3) Control Tables
4) Tree Structures

A pointer variable takes 2 Bytes in memory, because the pointer variable holds the address of a variable and the address of memory location always in unsigned integer. An unsigned integer takes 2 Bytes; therefore, pointer occupies 2 Bytes.

Declaration:

data_type *variable_name;

For example:

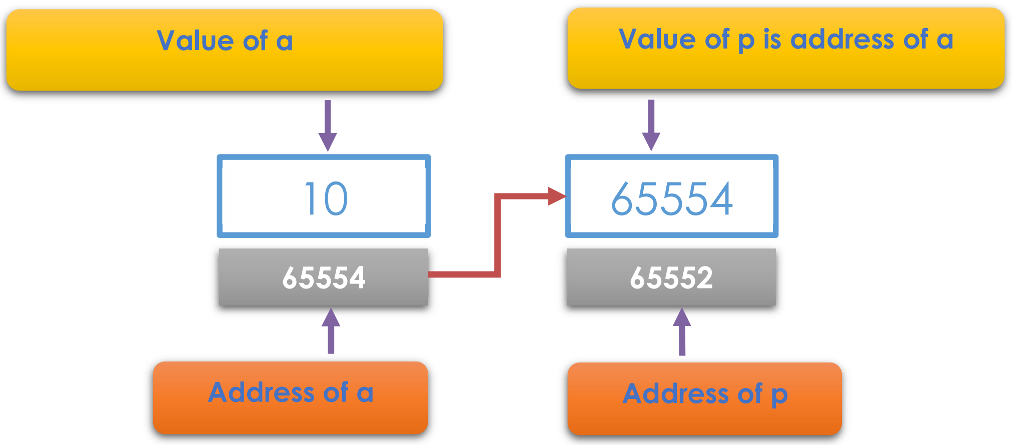
```
int *p, *q;
```

While learning about pointer we have to concentrate on two main things about pointer. These are –

```
&     -     address of <variablename>
*     -     value of <variablename>
```

For example: Consider there we have an integer variable and another is pointer variable. Let see following explanation –

```
int a=10;
int *p;
p=&a;
```



a	=	10
&a	=	65554
p	=	65554
&p	=	65552
***p**	=	10

For Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a;
    int *p;
    clrscr();
    printf("Enter a number for variable a: ");
    scanf("%d",&a);
    p=&a;
    printf("The value of a is: %d",a);
    printf("\nThe address of a is: %u",&a);
    printf("\n The value of p(holds the address of a) is: %u",p);
    printf("\n The address of p is: %u",&p);
    printf("\n The value of *p(value on address that holds by p) is: %d",*p);
    getch();
}
```

OUTPUT:

```
Enter a number for variable a: 25
The value of a is: 25
The address of a is: 65524
The value of p (holds the address of a) is: 65524
The address of p is: 65522
The value of *p(value on address that holds by p) is: 25
```

Pointer Details

Pointer arithmetic

There are four arithmetic operators that can be used in pointers: ++, --, +, -

Array of pointers

We can define arrays to hold a number of pointers.

Pointer to pointer

C allows us to have pointer on a pointer and so on.

Passing pointers to functions in C

Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.

Return pointer from functions in C

C allows a function to return a pointer to the local variable, static variable and dynamically allocated memory as well.

Structure

A structure is a composite data type (User Defined Data Type) that defines a grouped list of variables that are to be placed under one name in a block of memory. It allows different variables to be accessed by using a single pointer to the structure. In other words, we can say that Structure is a group of different data items.

In 'C' programming arrays allow us to define variables that can hold several data items of the same kind but **structure** is another **user defined data type** available in C programming, which allows us to combine data items of different kinds.

As we mention above Structures are used to represent a record. Suppose we want to keep track of our books in a library. For example –

- Title
- Author
- Subject
- Book ID

So, store these kinds of information we use structure.

Advantages

- It can hold variables of different data types.
- We can create objects containing different types of attributes.
- It allows us to re-use the data layout across programs.
- It is used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

Defining a Structure

To define a structure, we use the **struct** statement. The **struct** statement defines a new data type, with more than one member for your program. Each member definition is a normal variable definition, such as `int` i; or `float` f; or any other valid variable definition. The syntax of the **struct** statement is following –

Syntax:

```
struct <structure tag name>
{
      member definition(s);
} <one or more structure Objects>;
```

defining an object at this scope is treated as global

Defining a Structure Variable/Object

At the end of the structure's definition, **before the final semicolon, we can specify one or more structure variables but it is optional.** We can declare structure variable according to following methods –

1) **As Global Scope**
2) **As Local Scope**

1) As Global Scope

When we are defining variable of a structure as global scope then it is accessible to all functions (main() and other user-defined function) equally.

Syntax:

Outside of all functions	**Before the final semicolon of structure –**

`struct <tag name>`
`{`
`      members;`
`};`
`struct **<tag name> <variable name>;**`
	`struct <tag name>`
	`{`
	`      members;`
	`}struct **<tag name> <variable name>;**`

2) As Local Scope

If we define a structure variable as local scope or in a function body then it will not be accessible outside of that function.

Syntax:

```
struct <tag name>
{
      members;
}
void main()
{
      struct <tag name> <variable name>;
}
```

Accessing Structure Data Member

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure:

For Example:

Que 118: WAP to define structure for book information and input the information about book then display it.

```c
#include<stdio.h>
#include<conio.h>
struct Books
{
    char t[50];
    char au[50];
    char s[100];
    int id;
};
void main()
{
    struct Books B;
    clrscr();
    printf("Enter book Title: ");
    gets(B.t);
    fflush(stdin);
    printf("Enter Author: ");
    scanf("%s",B.au);
    printf("Enter Subject: ");
    scanf("%s",B.s);
    printf("Enter Book ID: ");
    scanf("%d",&B.id);
    printf("Book Details are -");
    printf("Book Title: %s\n", B.t);
    printf("Book Author: %s\n", B.au);
    printf("Book Subject: %s\n", B.s);
    printf("Book ID: %d\n", B.id);
    getch();
}
```

OUTPUT:

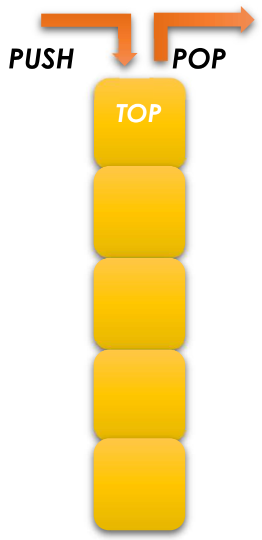
```
Enter Book Title: C Language
Enter Author: Unknown
Enter Subject: C
Enter Book ID: 231

Book Details are following -
Book Title: C Language
Book Author: Unknown
Book Subject: C
Book ID: 231
```

Stack

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at a single position which is known as "top". That means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on ***LIFO (Last In First Out)*** principle.

Stack Representation

In a stack, the insertion operation is performed using a function called "***push***" and deletion operation is performed using a function called "***pop***".



In the figure, **PUSH** and **POP** operations are performed at a top position in the stack. *That means, both the insertion and deletion operations are performed at one end (at Top).*

A stack data structure can be defined as follows:

Stack is a linear data structure in which the operations are performed based on ***LIFO*** *principle.*

OR

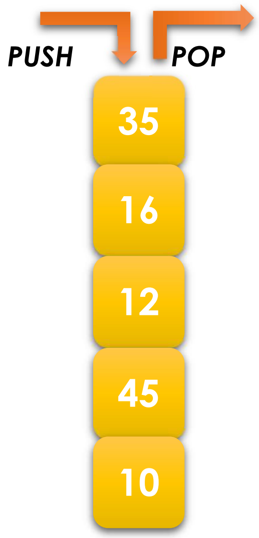
A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle.

Example: If we want to create a stack by inserting 10,45,12,16 and 35. Then 10 becomes the bottom-most element and 35 is the topmost element. The last inserted element 35 is at Top of the stack as shown in the image below.

Primitive Operations on a Stack

The following operations are performed on the stack...

1) **Push (To insert an element on to the stack)**
2) **Pop (To delete an element from the stack)**
3) **Display (To display elements of the stack)**



Implementation of Stack

Stack data structure can be implemented in two ways. They are as follows:

1) Using Array
2) Using Linked List

When a stack is implemented using an array, that stack can organize an only limited number of elements. When a stack is implemented using a linked list, that stack can organize an unlimited number of elements.

1) Stack Using Array

A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple. Just define a one-dimensional array of specific size and insert or delete the values into that array by using ***LIFO*** principle with the help of a variable called **'*top*'**. Initially, the top is set to ***-1***. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

A stack can be implemented using array as follows:

Before implementing actual operations, first follow the below steps to create an empty stack.

Step 1 - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

Step 2 - Declare all the **functions** used in stack implementation.

Step 3 - Create a one-dimensional array with fixed size **(int stack[SIZE])**

Step 4 - Define an integer variable **'top'** and initialize with **'-1'**. **(int top = -1)**

Step 5 - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

push(value) - Inserting value into the stack

In a stack, **push()** is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack:

Step 1 - Check whether **stack** is **FULL. (top == SIZE-1)**

Step 2 - If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT FULL**, then increment **top** value by one **(top++)** and set **stack[top]** to value **(stack[top] = value**).

pop() - Delete a value from the Stack

In a stack, **pop()** is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack:

Step 1 - Check whether **stack** is **EMPTY**. **(top == -1)**

Step 2 - If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one **(top--)**.

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack:

Step 1 - Check whether stack is **EMPTY. (top == -1)**

Step 2 - If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then define a variable **'i'** and initialize with **top**. Display **stack[i]** value and decrement **i** value by one **(i--)**.

Step 4 - Repeat above step until **i** value becomes **'0'**.

C Program: Implementation of Stack using Array

```c
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define SIZE 10
void push(int);
void pop();
void display();
int stack[SIZE], top = -1;
void main()
```

```c
{
    int value, choice;
    while (1)
    {
        printf("\n***** Menu *****\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            printf("Enter the value to be insert: ");
            scanf("%d", &value);
            push(value);
            break;
        case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            exit(0);
            break;
        default:
            printf("\nWrong selection!!! Try again!!!");
        }
    }
}

void push(int value)
{
    if (top == SIZE - 1)
        printf("\nStack is Full!!! Insertion is not possible!!!");
    else
    {
        top++;
        stack[top] = value;
        printf("\nInsertion success!!!");
    }
}
void pop()
{
    if (top == -1)
    {
        printf("\nStack is Empty!!!\nPress Enter to continue\n");
        getch();
    }
    else
    {
        printf("\nDeleted : %d", stack[top]);
        top--;
    }
}
void display()
{
    if (top == -1)
    {
        printf("\nStack is Empty!!!\nPress Enter to continue\n");
        getch();
    }
    else
    {
        int i;
        printf("\nStack elements are:\n");
        for (i = top; i >= 0; i--)
            printf("%d\n", stack[i]);
    }
}
```

OUTPUT:

```
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 23
Insertion Success!!!
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12
Insertion Success!!!
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements are:
23    12
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
```

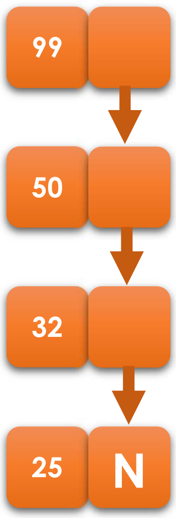
2) Stack Using Linked List

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use.

A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as **'top'** element. That means every newly inserted element is pointed by **'top'**. Whenever we want to remove an element from the stack, simply remove the node which is pointed by **'top'** by moving **'top'** to its previous node in the list. The **next** field of the first element must be always **NULL**.

Example:



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the header files which are used in the program. And declare all the user defined functions.

Step 2 - Define a 'Node' structure with two members data and next.

Step 3 - Define a Node pointer 'top' and set it to NULL.

Step 4 - Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

push(value) - Inserting an element into the Stack
We can use the following steps to insert a new node into the stack...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether stack is **Empty (top == NULL)**

Step 3 - If it is **Empty**, then set **newNode → next = NULL.**

Step 4 - If it is **Not Empty**, then set **newNode → next = top**.

Step 5 - Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack
We can use the following steps to delete a node from the stack:

Step 1 - Check whether **stack** is **Empty (top == NULL)**.

Step 2 - If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

Step 3 - If it is **Not Empty**, then define a Node pointer **'*temp*'** and set it to **'*top*'**.

Step 4 - Then set **'top = top → next'.**

Step 5 - Finally, delete **'temp'. (free(temp))**.

display() - Displaying stack of elements
We can use the following steps to display the elements (nodes) of a stack:

Step 1 - Check whether **stack** is **Empty (top == NULL)**.

Step 2 - If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.

Step 4 - Display **'temp → data --->'** and move it to the **next** node. Repeat the same until temp reaches to the first node in the **stack**. **(temp → next != NULL).**

Step 5 - Finally! Display **'temp → data ---> NULL'**.

C Program: Implementation of Stack using Linked List

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *top = NULL;
void push(int);
void pop();
void display();
void main()
```

```c
{
    int choice, value;
    printf("\n:: Stack using Linked List ::\n");
    while (1)
    {
        printf("\n****** Menu ******\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
              case 1:
                  printf("Enter the value to be insert: ");
                  scanf("%d", &value);
                  push(value);
                  break;
              case 2:
                  pop();
                  break;
              case 3:
                  display();
                  break;
              case 4:
                  exit(0);
              default:
                  printf("\nWrong selection!!! Please try again!!!\n");
        }
    }
}
void push(int value)
{
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = value;
    if (top == NULL)
    {
        newNode->next = NULL;
    }
    else
    {
        newNode->next = top;
    }
    top = newNode;
    printf("\nInsertion is Success!!!\n");
}
void pop()
{
    if (top == NULL)
    {
        printf("\nStack is Empty!!!\nPress Enter to continue\n");
        getch();
    }
    else
    {
        struct node *temp = top;
        printf("\nDeleted element: %d", temp->data);
        top = temp->next;
        free(temp);
    }
}
void display()
{
    if (top == NULL)
    {
        printf("\nStack is Empty!!!\n Press Enter to continue\n");
        getch();
    }
    else
    {
        struct node *temp = top;
        while (temp->next != NULL)
        {
            printf("%d--->", temp->data);
            temp = temp->next;
        }
        printf("%d--->NULL", temp->data);
    }
}
```

OUTPUT:

```
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 23
Insertion Success!!!
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12
Insertion Success!!!
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements are:
23--->12--->NULL
***** Menu *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
```

Applications of Stack

There are following some important application of stacks are given following:

Expression Conversion

An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

Infix to Postfix or Infix to Prefix Conversion −

The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent. These postfix or prefix notations are used in computers to express some expressions. These expressions are not so much familiar to the infix expression, but they have some great advantages also. We do not need to maintain operator ordering, and parenthesis.

Expression Evaluation

Stack is used to evaluate prefix, postfix and infix expressions.

Postfix or Prefix Evaluation −

After converting into prefix or postfix notations, we have to evaluate the expression to get the result. For that purpose, also we need the help of stack data structure.

Syntax Parsing

Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Backtracking

Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

Parenthesis Checking

Stack is used to check the proper opening and closing of parenthesis.

String Reversal

Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

Function Call

Stack is used to keep information about the active functions or subroutines.

Expressions

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression can be defined as follows:

An expression is a collection of operators and operands that represents a specific value.

In above definition, ***operator*** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the ***operators*** can perform the task. Here operand can be a direct value or variable or address of memory location.

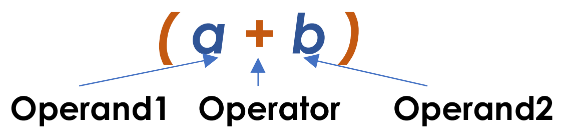
Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows:

1) Infix Expression
2) Postfix Expression
3) Prefix Expression

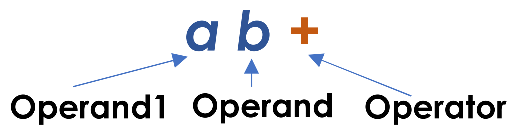
1) Infix Expression

In infix expression, operator is used in between the operands. The general structure of an Infix expression is as follows:



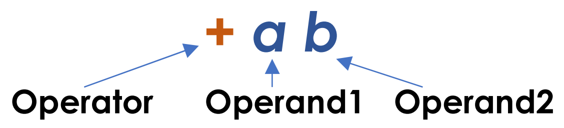
2) Postfix Expression

In postfix expression, operator is used after operands. We can say that **"Operator follows the Operands"**. The general structure of Postfix expression is as follows:



3) Prefix Expression

In prefix expression, operator is used before operands. We can say that **"Operands follows the Operator"**. The general structure of Prefix expression is as follows:



Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like **Infix to Postfix, Infix to Prefix, Prefix to Postfix** and vice versa.

Priority and Associativity of Operators

Before performing any expression conversion, we should know about the priority and associativity of operators. Please have a look at following table to understand it:

Precedence	Operator	Description	Associativity
1	`++ --`	Suffix/postfix increment and decrement	Left-to-right
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Structure and union member access	
	`->`	Structure and union member access through pointer	
2	`++ --`	Prefix increment and decrement	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(type)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of	
3	`* / %`	**Multiplication, division, and remainder**	Left-to-right
4	`+ -`	**Addition and subtraction**	
5	`<< >>`	Bitwise left shift and right shift	
6	`< <=`	For relational operators < and ≤ respectively	
	`> >=`	For relational operators > and ≥ respectively	
7	`== !=`	For relational = and ≠ respectively	
8	`&`	Bitwise AND	
9	`^`	Bitwise XOR (exclusive or)	
10	`|`	Bitwise OR (inclusive or)	
11	`&&`	Logical AND	
12	`||`	Logical OR	
13	`?:`	Ternary conditional	Right-to-Left
14	`=`	Simple assignment	
	`+= -=`	Assignment by sum and difference	
	`*= /= %=`	Assignment by product, quotient, and remainder	
	`<<= >>=`	Assignment by bitwise left shift and right shift	
	`&= ^= |=`	Assignment by bitwise AND, XOR, and OR	
15	`,`	Comma	Left-to-right

Table: Operator precedence & Associativity

Infix to Postfix Conversion

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure:

- Find all the operators in the given Infix Expression.
- Find the order of operators evaluated according to their Operator precedence.

- Convert each operator into required type of expression (Postfix or Prefix) in the same order.

To convert Infix Expression into Postfix Expression using a stack data structure, we can use the following rules/steps:

1) Read all the symbols one by one from **left to right** in the given Infix Expression.
2) **If the reading symbol is operand, then directly print it to the result (Output).**
3) **If the reading symbol is operator, then Push it on to the top of stack.**
4) **If reading symbol is operator and If it has same priority in context of previous operator then we POP out the previous operator first then we will PUSH last scanned operator. For example: if - comes after + then + will POP out then – will PUSHED.**
5) **If reading symbol is operator and If it has higher priority than previous operator that has lower priority then it can stay with it. For example: if * comes after + then it can stay with +.**
6) **If reading symbol is operator and If it has lower priority than previous operator that has higher priority, then we will POP the existing higher priority operator from stack. For example: if - comes after / then / will POPPED from stack than – will Pushed.**
7) If the reading symbol is left parenthesis '(', then Push it on to the Stack.
8) **If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.**
9) **If the reading symbol is operator (+ , - , * , / etc.), then Push it on to the Stack.** However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example: Let's consider following expression for conversion of INFIX to POSTFIX

Infix Expression: A+B*C-D/E+F-G

Step	Character Scanned	Stack	Postfix
1	*A*	Empty	A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6	-	-	A B C * +
7	D	-	A B C * + D
8	/	- /	A B C * + D
9	E	- /	A B C * + D E
10	+	+	A B C * + D E / -
11	F	+	A B C * + D E / -
12	-	-	A B C * + D E / - +
13	G	-	A B C * + D E / - + G
POSTFIX EXPRESSION			**A B C * + D E / - + G -**

Table: Infix to Postfix Conversion

Postfix to Infix Conversion

Now let us reverse the above conversion to verify that we are on right track:

Step	Character Scanned (Postfix)	Infix
1	A	A
2	B	A B
3	C	A B C
4	*	A B * C
5	+	A + B * C
6	D	A + B * C D
7	E	A + B * C D E
8	/	A + B * C D / E
9	-	A + B * C - D / E
10	F	A + B * C - D / E F
11	+	A + B * C - D / E + F
12	G	A + B * C - D / E + F G
13	-	A + B * C - D / E + F – G

Table: Postfix to Infix Conversion

Infix to Prefix Conversion

As we convert infix to postfix, we can also convert infix to prefix also. To do so we have to follow some rules/steps, that are given following:

1) First reverse the given infix expression.
2) Scan all the Symbols and operands one by one from **left to the right.**
3) **If the reading symbol is operand, then directly print it to the result (Output).**
4) **If the reading symbol is operator, then Push it on to the top of stack.**
5) **If reading symbol is operator and If it has same priority in context of previous operator then it can stay with existing operator. For example: if - comes after + then + will be PUSHED at the top of stack.**
6) **If reading symbol is operator and If it has higher priority than previous operator then all the lower priority operator will POP out one by one then higher priority operator will PUSED at top of the stack. For example: if * comes after + then + will POP out from stack and then * will PUSHED.**
7) **If reading symbol is operator and If it has lower priority than previous operator then it will PUSHED at the top of the stack. For example: if + comes after / then + will be PUSHED at the top of the stack.**
8) If the reading symbol is left parenthesis '(', then Push it on to the Stack.
9) **If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.**
10) **If the reading symbol is operator (+ , - , * , / etc.), then Push it on to the Stack.** However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example: Let's consider following expression for conversion of INFIX to PREFIX

Infix Expression: A – B * C + D / E + F - G

G – F + E / D + C * B - A

Reversing the Infix Expression

Step	Character Scanned	Stack	Postfix
1	G	Empty	G
2	-	-	G
3	F	-	F G
4	+	- +	E F G
5	E	- +	E F G
6	/	/	- + E F G
7	D	/	D - + E F G
8	+	/ +	D - + E F G
9	C	/ +	C D - + E F G
10	*	/ *	+ C D - + E F G
11	B	/ *	B + C D - + E F G
12	-	/ * -	B + C D - + E F G
13	A	/ * -	A B + C D - + E F G
	INFIX TO PREFIX		/ * - A B + C D - + E F G

Table: Infix to Postfix Conversion

Prefix to Infix Conversion

Now let us reverse the above conversion to verify that we are on right track:

Step	Character Scanned (Prefix)	Infix (Reversed)
1	G	G
2	F	G F
3	E	G F E
4	+	G F + E
5	-	G - F + E
6	D	G - F + E D
7	C	G - F + E D C
8	+	G - F + E D + C
9	B	G - F + E D + C B
10	A	G - F + E D + C B A
11	-	G - F + E D + C B - A
12	*	G - F + E D + C * B - A
13	/	G - F + E / D + C * B - A

Table: Postfix to Infix Conversion

Now reverse the expression to get infix Expression:

G – F + E / D + C * B – A

A – B * C + D / E + F - G

Reversing Back to get the Infix Expression

Recursion

In C programming language, function calls can be made from the main() function, other functions or from the same function itself. The recursive function is defined as follows:

*A function called by itself is called **recursive** function.*

The recursive functions should be used very carefully because, when a function called by itself it enters into the infinite loop. And when a function enters into the infinite loop, the function execution never gets completed. We should define the condition to exit from the function call so that the recursive function gets terminated.

When a function is called by itself, the first call remains under execution till the last call gets invoked. Every time when a function call is invoked, the function returns the execution control to the previous function call.

Example:

```
#include <stdio.h>
#include <conio.h>
int factorial(int);
void main()
{
    int fact, n;
    printf("Enter any positive integer: ");
    scanf("%d", &n);
    fact = factorial(n);
    printf("\nFactorial of %d is %d\n", n, fact);
}
int factorial(int n)
{
    int t = 1;
    if (t < n)
    {
        t = n * factorial(n - 1); // recursive function call
    }
    return t;
}
```

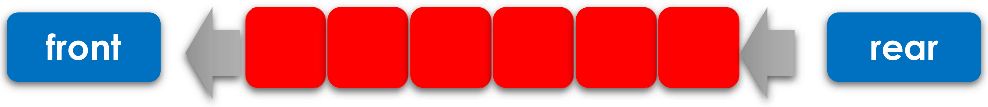
Output:

```
Enter any positive integer: 5
Factorial of 5 is 120
```

In the above example program, the **factoria**l() function call is initiated from main() function with the value 5. Inside the **factorial()** function, the function calls **factorial(5)**, **factorial(4)**, **factorial(3)**, **factorial(2)**, **factorial(1)** and **factorial(0)** are called recursively. In this program execution process, the function call **factorial(5)** remains under execution till the execution of function calls **factorial(4)**, **factorial(3)**, **factorial(2)**, **factorial(1)** and **factorial(0)** gets completed. Similarly the function call **factorial(4)** remains under execution till the execution of function calls **factorial(3)**, **factorial(2)**, **factorial(1)** and **factorial(0)** gets completed. In the same way the function call **factorial(3)** remains under execution till the execution of function call **factorial(2)**, **factorial(1)** and **factorial(0)** gets completed. And this process will remain calling function recursively until the final call is made and the condition get false.

Queue

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end. In a queue data structure, the insertion operation is performed at a position which is known as ***'rear'*** and the deletion operation is performed at a position which is known as ***'front'***. In queue data structure, the insertion and deletion operations are performed based on ***FIFO (First In First Out)*** principle.



In a queue data structure, the insertion operation is performed using a function called "enQueue()" and deletion operation is performed using a function called "deQueue()".

Queue data structure can be defined as follows:

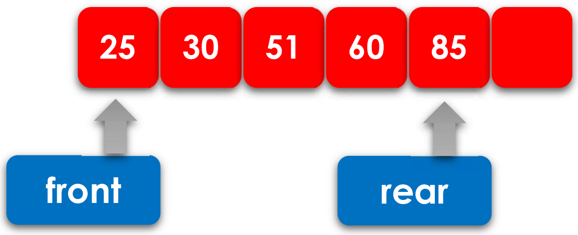
Queue data structure is a linear data structure in which the operations are performed based on ***FIFO*** *principle.*

A queue data structure can also be defined as

Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on ***FIFO*** *principle.*

Example: Queue after inserting 25, 30, 51, 60 and 85.

After inserting five elements:



Primitive Operations on a Queue

The following operations are performed on a queue data structure:

1) **enQueue(value) - (To insert an element into the queue)**
2) **deQueue() - (To delete an element from the queue)**
3) **display() - (To display the elements of the queue)**

Implementation of Queue Data Structure

Queue data structure can be implemented in two ways. They are as follows:

1) Using Array
2) Using Linked List

When a queue is implemented using an array, that queue can organize an only limited number of elements. When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

1) Queue Using Array

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple. Just define a one-dimensional array of specific size and insert or delete the values into that array by using ***FIFO (First In First Out)*** principle with the help of variables ***'front'*** and ***'rear'***.

Initially both ***'front'*** and ***'rear'*** are set to **-1**. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Before we implement actual operations, first follow the below steps to create an empty queue:

Step 1 - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

Step 2 - Declare all the **user defined functions** which are used in queue implementation.

Step 3 - Create a one dimensional array with above defined **SIZE (int queue[SIZE])**

Step 4 - Define two integer variables **'front'** and **'rear'** and initialize both with **'-1'**. **(int front = -1, rear = -1)**

Step 5 - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue(value) - Inserting value into the queue

In a queue data structure, **enQueue()** is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The **enQueue()** function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue:

Step 1 - Check whether queue is **FULL**. **(rear == SIZE-1)**

Step 2 - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT FULL**, then increment **rear** value by one **(rear++)** and set **queue[rear] = value**.

deQueue() - Deleting a value from the Queue

In a queue data structure, **deQueue()** is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The **deQueue()** function does not take any value as parameter. We can use the following steps to delete an element from the queue:

Step 1 - Check whether queue is **EMPTY**. **(front == rear)**

Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then increment the **front** value by one **(front ++)**. Then display **queue[front]** as deleted element. Then check whether both front and rear are equal **(front == rear)**, if it **TRUE**, then set both front and rear to **'-1' (front = rear = -1)**.

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue:

Step 1 - Check whether queue is **EMPTY**. **(front == rear)**

Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then define an integer variable **'i'** and set **'i = front+1'**.

Step 4 - Display **'queue[i]'** value and increment **'i'** value by one **(i++)**. Repeat the same until **'i'** value reaches to **rear (i <= rear).**

C Program: Implementation of Queue using Array

OUTPUT:

```
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 23
Insertion Success!!!
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12
Insertion Success!!!
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3
Queue elements are:
23    12
***** Menu *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 4
```

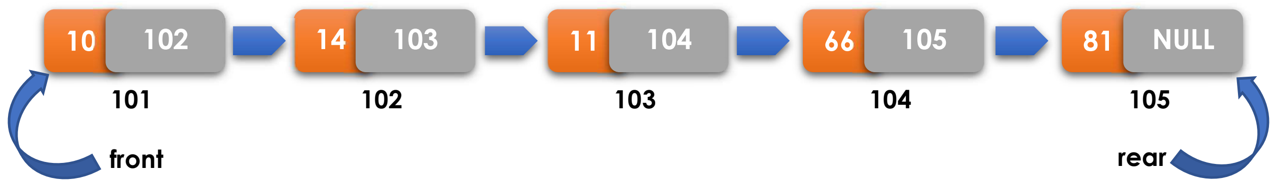
2) Queue Using Linked List

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use.

A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by **'rear'** and the first node is always pointed by **'front'**.

Example



In above example, the last inserted node is 50 and it is pointed by **'rear'** and the first inserted node is 10 and it is pointed by **'front'**. The order of elements inserted is 10, 15, 22 and 50.

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program. And declare all the **user defined functions**.

Step 2 - Define a **'Node'** structure with two members **data** and **next**.

Step 3 - Define two Node pointers **'front'** and **'rear'** and set both to **NULL**.

Step 4 - Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue:

Step 1 - Create a **newNode** with given value and set **'newNode → next'** to **NULL**.

Step 2 - Check whether queue is **Empty (rear == NULL)**

Step 3 - If it is **Empty** then, set **front = newNode** and **rear = newNode**.

Step 4 - If it is Not **Empty** then, set **rear → next = newNode** and **rear = newNode**.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue:

Step 1 - Check whether queue is **Empty (front == NULL).**

Step 2 - If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function

Step 3 - If it is Not **Empty** then, define a **Node** pointer **'temp'** and set it to **'front'**.

Step 4 - Then set **'front = front → next'** and delete **'temp' (free(temp))**.

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

Step 1 - Check whether queue is **Empty (front == NULL)**.

Step 2 - If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.

Step 3 - If it is Not **Empty** then, define a Node pointer **'temp'** and initialize with **front**.

Step 4 - Display **'temp → data --->'** and move it to the **next** node. Repeat the same until **'temp'** reaches to **'rear' (temp → next != NULL)**.

Step 5 - Finally! Display **'temp → data ---> NULL'**.

C Program: Implementation of Queue using Linked List

```
#include<stdio.h>
#include<conio.h>
struct Node
{
      int data;
      struct Node *next;
}*front=NULL,*rear=NULL;

void insert(int);
void delete();
void display();

void main()
{
      int choice, value;
      clrscr();
      while(1)
      {
            printf("\n****** MENU ******\n");
            printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
            printf("Enter your choice: ");
            scanf("%d",&choice);
            switch(choice)
            {
                  case 1:
                        printf("Enter the value to be insert: ");
                        scanf("%d", &value);
                        insert(value);
                  break;
                  case 2:
                        delete();
                  break;
                  case 3:
                        display();
                  break;
                  case 4:
                        exit(0);
                  break;
                  default:
                        printf("\nWrong selection!!! Please try again!!!\n");
            }
      }
}

void insert(int value)
{
      struct Node *newNode;
      newNode=(struct Node*)malloc(sizeof(struct Node));
      newNode->data=value;
      newNode->next=NULL;
      if(front==NULL)
      {
            front=rear=newNode;
      }
```