

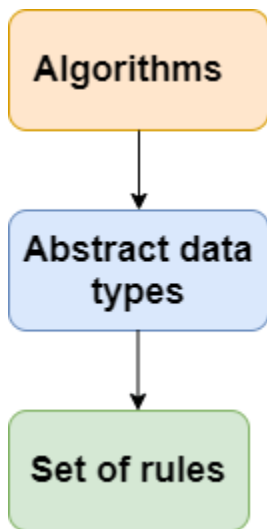
DATA STRUCTURE NOTES
BY
VINEET SINGH
GOVT POLYTECHNIC KANPUR
(2020-21)

What is Data Structure?

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory. Let's see the different types of data structures.

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.



Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

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

- Linear data structure
- Non-linear data structure

Linear Data Structure

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.

We will discuss the above data structures in brief in the coming topics. Now, we will see the common operations that we can perform on these data structures.

Data structures can also be classified as:

- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.
- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

Which Data Structure?

A data structure is a way of organizing the data so that it can be used efficiently. Here, we have used the word efficiently, which in terms of both the space and time. For example, a stack is an ADT (Abstract data type) which uses either arrays or linked list data structure for the implementation. Therefore, we conclude that we require some data structure to implement a particular ADT.

An ADT tells **what** is to be done and data structure tells **how** it is to be done. In other words, we can say that ADT gives us the blueprint while data structure provides the implementation part. Now the question arises: how can one get to know which data structure to be used for a particular ADT?.

As the different data structures can be implemented in a particular ADT, but the different implementations are compared for time and space. For example, the Stack ADT can be implemented by both Arrays and linked list. Suppose the array is providing time efficiency while the linked list is providing space efficiency, so the one which is the best suited for the current user's requirements will be selected.

Advantages of Data structures

The following are the advantages of a data structure:

- **Efficiency:** If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- **Reusability:** The data structures provide reusability means that multiple client programs can use the data structure.
- **Abstraction:** The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

Data Structure

Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising 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 i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

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 terminology is used as far as data structures are concerned.

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 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

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.

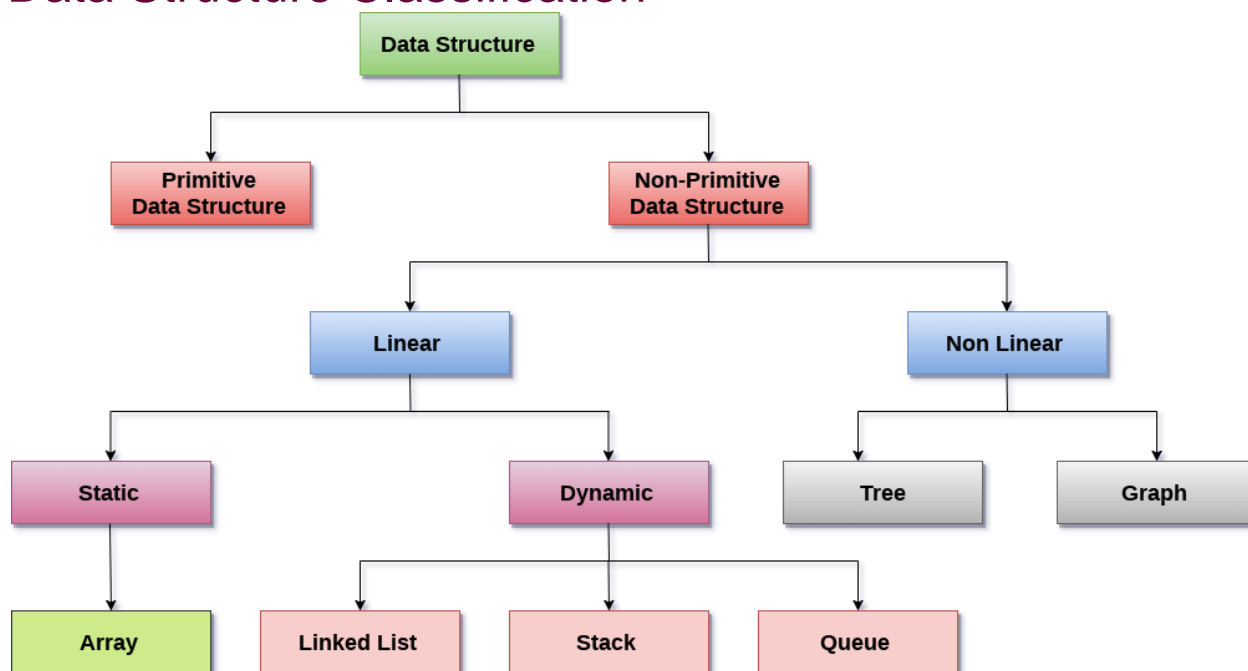
Advantages of Data Structures

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 which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Data Structure Classification



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.

Types of Linear Data Structures are given below:

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 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.

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.

Types of Non Linear Data Structures are given below:

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 children 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 can not 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 subject, 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

What is an Algorithm?

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

Characteristics of an Algorithm

The following are the characteristics of an algorithm:

- **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output:** We will get 1 or more output at the end of an algorithm.
- **Unambiguity:** An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- **Finiteness:** An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- **Effectiveness:** An algorithm should be effective as each instruction in an algorithm affects the overall process.
- **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step by step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

Why do we need Algorithms?

We need algorithms because of the following reasons:

- **Scalability:** It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- **Performance:** The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Let's understand the algorithm through a real-world example. Suppose we want to make a lemon juice, so following are the steps required to make a lemon juice:

Step 1: First, we will cut the lemon into half.

Step 2: Squeeze the lemon as much you can and take out its juice in a container.

Step 3: Add two tablespoon sugar in it.

Step 4: Stir the container until the sugar gets dissolved.

Step 5: When sugar gets dissolved, add some water and ice in it.

Step 6: Store the juice in a fridge for 5 to minutes.

Step 7: Now, it's ready to drink.

The above real-world can be directly compared to the definition of the algorithm. We cannot perform the step 3 before the step 2, we need to follow the specific order to make lemon juice. An algorithm also says that each and every instruction should be followed in a specific order to perform a specific task.

Now we will look an example of an algorithm in programming.

We will write an algorithm to add two numbers entered by the user.

The following are the steps required to add two numbers entered by the user:

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: Enter the values of a and b.

Step 4: Add the values of a and b and store the result in the sum variable, i.e., $\text{sum} = a + b$.

Step 5: Print sum

Step 6: Stop

Factors of an Algorithm

The following are the factors that we need to consider for designing an algorithm:

- **Modularity:** If any problem is given and we can break that problem into small-small modules or small-small steps, which is a basic definition of an algorithm, it means that this feature has been perfectly designed for the algorithm.
- **Correctness:** The correctness of an algorithm is defined as when the given inputs produce the desired output, which means that the algorithm has been designed algorithm. The analysis of an algorithm has been done correctly.
- **Maintainability:** Here, maintainability means that the algorithm should be designed in a very simple structured way so that when we redefine the algorithm, no major change will be done in the algorithm.
- **Functionality:** It considers various logical steps to solve the real-world problem.
- **Robustness:** Robustness means that how an algorithm can clearly define our problem.
- **User-friendly:** If the algorithm is not user-friendly, then the designer will not be able to explain it to the programmer.
- **Simplicity:** If the algorithm is simple then it is easy to understand.
- **Extensibility:** If any other algorithm designer or programmer wants to use your algorithm then it should be extensible.

Importance of Algorithms

1. **Theoretical importance:** When any real-world problem is given to us and we break the problem into small-small modules. To break down the problem, we should know all the theoretical aspects.
2. **Practical importance:** As we know that theory cannot be completed without the practical implementation. So, the importance of algorithm can be considered as both theoretical and practical.

Issues of Algorithms

The following are the issues that come while designing an algorithm:

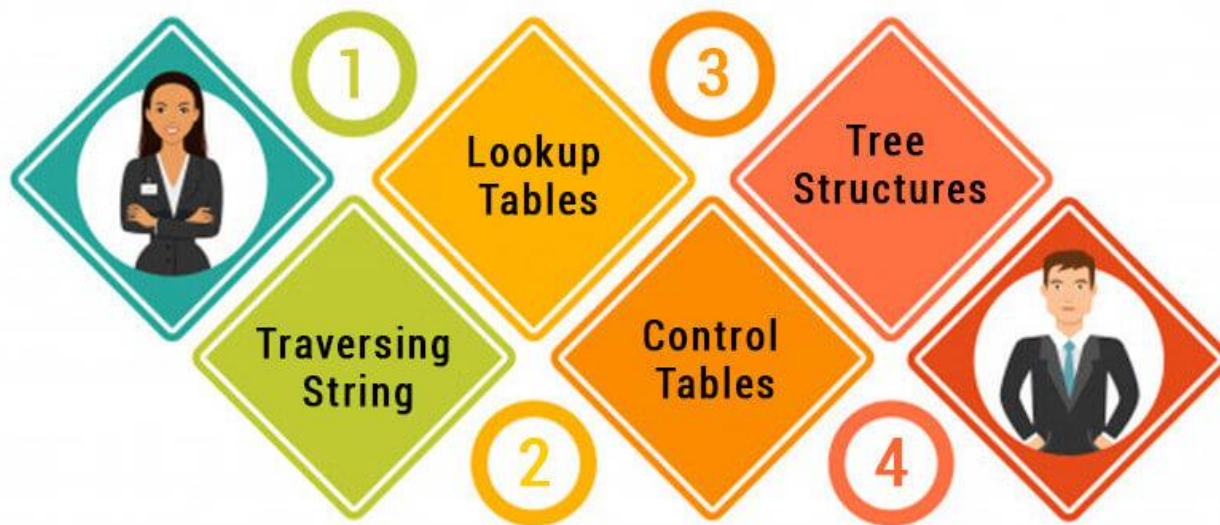
- **How to design algorithms:** As we know that an algorithm is a step-by-step procedure so we must follow some steps to design an algorithm.
- **How to analyze algorithm efficiency**

Pointer

Pointer is used to point 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:

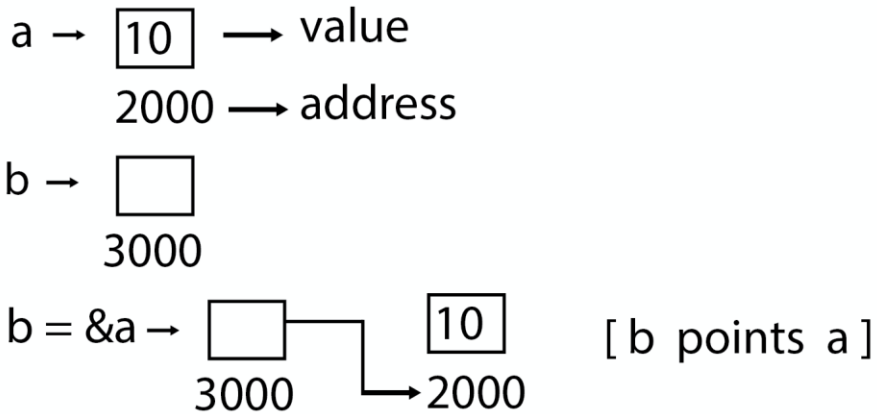
- Traversing String
- Lookup Tables
- Control Tables
- Tree Structures

Pointer Improves Performance for



Pointer Details

- **Pointer arithmetic:** There are four arithmetic operators that can be used in pointers: ++, --, +, -
- **Array of pointers:** You can define arrays to hold a number of pointers.
- **Pointer to pointer:** C allows you 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.



Program

Pointer

```
1. #include <stdio.h>
2.
3. int main( )
4. {
5.     int a = 5;
6.     int *b;
7.     b = &a;
8.
9.     printf ("value of a = %d\n", a);
10.    printf ("value of a = %d\n", *(&a));
11.    printf ("value of a = %d\n", *b);
12.    printf ("address of a = %u\n", &a);
13.    printf ("address of a = %d\n", b);
14.    printf ("address of b = %u\n", &b);
15.    printf ("value of b = address of a = %u", b);
16.    return 0;
17. }
```

Output

```
1. value of a = 5
2. value of a = 5
3. address of a = 3010494292
4. address of a = -1284473004
5. address of b = 3010494296
6. value of b = address of a = 3010494292
```

Program

Pointer to Pointer

```
1. #include <stdio.h>
2.
3. int main( )
4. {
5.     int a = 5;
6.     int *b;
7.     int **c;
8.     b = &a;
9.     c = &b;
10. printf ("value of a = %d\n", a);
11. printf ("value of a = %d\n", *(&a));
12. printf ("value of a = %d\n", *b);
13. printf ("value of a = %d\n", **c);
14. printf ("value of b = address of a = %u\n", b);
15. printf ("value of c = address of b = %u\n", c);
16. printf ("address of a = %u\n", &a);
17. printf ("address of a = %u\n", b);
18. printf ("address of a = %u\n", *c);
19. printf ("address of b = %u\n", &b);
20. printf ("address of b = %u\n", c);
21. printf ("address of c = %u\n", &c);
22. return 0;
23. }
```

Pointer to Pointer

```
1. value of a = 5
2. value of a = 5
3. value of a = 5
4. value of a = 5
5. value of b = address of a = 2831685116
6. value of c = address of b = 2831685120
7. address of a = 2831685116
8. address of a = 2831685116
9. address of a = 2831685116
10. address of b = 2831685120
11. address of b = 2831685120
12. address of c = 2831685128
```

Array

Definition

- Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in different subjects. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations.

The array **marks[10]** defines the marks of the student in 10 different subjects where each subject's marks are located at a particular subscript in the array i.e. **marks[0]** denotes the marks in the first subject, **marks[1]** denotes the marks in the 2nd subject and so on.

Properties of the Array

1. Each element is of the same data type and carries the same size i.e. int = 4 bytes.
2. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
3. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

For example, in C language, the syntax of declaring an array is like following:

1. **int** arr[10]; **char** arr[10]; **float** arr[5]

Need of using Array

In computer programming, the most of the cases requires to store the large number of data of similar type. To store such amount of data, we need to define a large number of variables. It would be very difficult to remember names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Following example illustrates, how array can be useful in writing code for a particular problem.

In the following example, we have marks of a student in six different subjects. The problem intends to calculate the average of all the marks of the student.

In order to illustrate the importance of array, we have created two programs, one is without using array and other involves the use of array to store marks.

Program without array:

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int marks_1 = 56, marks_2 = 78, marks_3 = 88, marks_4 = 76, marks_5 = 56, marks_6 = 89;
5.     float avg = (marks_1 + marks_2 + marks_3 + marks_4 + marks_5 + marks_6) / 6 ;
6.     printf(avg);
7. }
```

Program by using array:

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int marks[6] = {56,78,88,76,56,89};
5.     int i;
6.     float avg;
7.     for (i=0; i<6; i++ )
8.     {
9.         avg = avg + marks[i];
10.    }
11.    printf(avg);
12.}
```

Complexity of Array operations

Time and space complexity of various array operations are described in the following table.

Time Complexity

Algorithm	Average Case	Worst Case
Access	$O(1)$	$O(1)$
Search	$O(n)$	$O(n)$
Insertion	$O(n)$	$O(n)$
Deletion	$O(n)$	$O(n)$

Space Complexity

In array, space complexity for worst case is **$O(n)$** .

Advantages of Array

- Array provides the single name for the group of variables of the same type therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process, we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

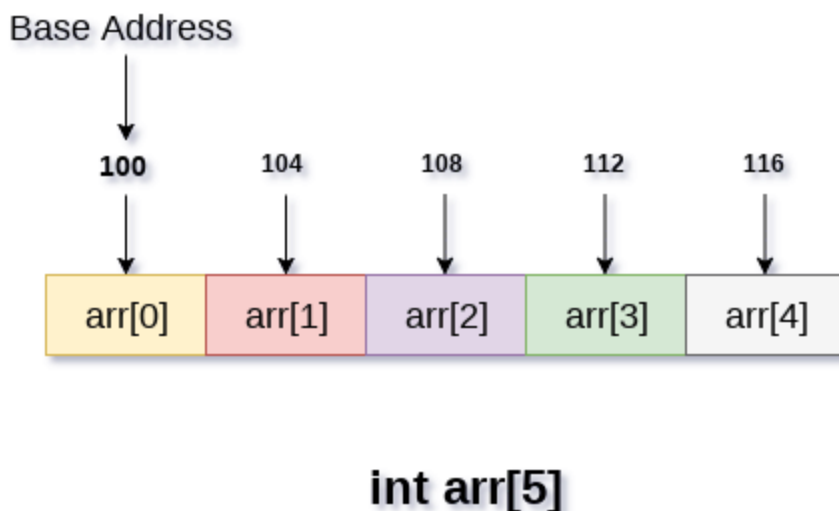
Memory Allocation of the array

As we have mentioned, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of first element in the main memory. Each element of the array is represented by a proper indexing.

The indexing of the array can be defined in three ways.

1. 0 (zero - based indexing) : The first element of the array will be `arr[0]`.
2. 1 (one - based indexing) : The first element of the array will be `arr[1]`.
3. n (n - based indexing) : The first element of the array can reside at any random index number.

In the following image, we have shown the memory allocation of an array `arr` of size 5. The array follows 0-based indexing approach. The base address of the array is 100th byte. This will be the address of `arr[0]`. Here, the size of `int` is 4 bytes therefore each element will take 4 bytes in the memory.



In 0 based indexing, If the size of an array is n then the maximum index number, an element can have is $n-1$. However, it will be n if we use 1 based indexing.

Accessing Elements of an array

To access any random element of an array we need the following information:

1. Base Address of the array.
2. Size of an element in bytes.
3. Which type of indexing, array follows.

Address of any element of a 1D array can be calculated by using the following formula:

1. Byte address of element $A[i] = \text{base address} + \text{size} * (i - \text{first index})$

Example :

1. In an array, $A[-10 \dots +2]$, Base address (BA) = 999, size of an element = 2 bytes,
2. find the location of $A[-1]$.
3. $L(A[-1]) = 999 + [(-1) - (-10)] \times 2$
4. $= 999 + 18$
5. $= 1017$

Passing array to the function :

As we have mentioned earlier that, the name of the array represents the starting address or the address of the first element of the array. All the elements of the array can be traversed by using the base address.

The following example illustrate, how the array can be passed to a function.

Example:

1. `#include <stdio.h>`
2. `int summation(int[]);`
3. `void main ()`
4. `{`
5. `int arr[5] = {0,1,2,3,4};`
6. `int sum = summation(arr);`
7. `printf("%d",sum);`
8. `}`
- 9.
10. `int summation (int arr[])`
11. `{`
12. `int sum=0,i;`
13. `for (i = 0; i<5; i++)`
14. `{`
15. `sum = sum + arr[i];`
16. `}`
17. `return sum;`
18. `}`

The above program defines a function named as summation which accepts an array as an argument. The function calculates the sum of all the elements of the array and returns it.

2D Array

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.

How to declare 2D Array

The syntax of declaring two dimensional array is very much similar to that of a one dimensional array, given as follows.

1. `int arr[max_rows][max_columns];`

however, It produces the data structure which looks like following.

	0	1	2	n-1
0	a[0][0]	a[0][1]	a[0][2]	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	a[2][n-1]
3	a[3][0]	a[3][1]	a[3][2]	a[3][n-1]
4	a[4][0]	a[4][1]	a[4][2]	a[4][n-1]
.
.
n-1	a[n-1][0]	a[n-1][1]	a[n-1][2]	a[n-1][n-1]

a[n][n]

Above image shows the two dimensional array, the elements are organized in the form of rows and columns. First element of the first row is represented by a[0][0] where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

How do we access data in a 2D array

Due to the fact that the elements of 2D arrays can be random accessed. Similar to one dimensional arrays, we can access the individual cells in a 2D array by using the indices of the cells. There are two indices attached to a particular cell, one is its row number while the other is its column number.

However, we can store the value stored in any particular cell of a 2D array to some variable x by using the following syntax.

```
1. int x = a[i][j];
```

where i and j is the row and column number of the cell respectively.

We can assign each cell of a 2D array to 0 by using the following code:

```
1. for ( int i=0; i<n ;i++)
2. {
3.     for (int j=0; j<n; j++)
4.     {
5.         a[i][j] = 0;
6.     }
7. }
```

Initializing 2D Arrays

We know that, when we declare and initialize one dimensional array in C programming simultaneously, we don't need to specify the size of the array. However this will not work with 2D arrays. We will have to define at least the second dimension of the array.

The syntax to declare and initialize the 2D array is given as follows.

```
1. int arr[2][2] = {0,1,2,3};
```

The number of elements that can be present in a 2D array will always be equal to (**number of rows * number of columns**).

Example : Storing User's data into a 2D array and printing it.

C Example :

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int arr[3][3],i,j;
5.     for (i=0;i<3;i++)
6.     {
7.         for (j=0;j<3;j++)
8.         {
9.             printf("Enter a[%d][%d]: ",i,j);
10.            scanf("%d",&arr[i][j]);
11.        }
12.    }
```

```

13.  printf("\n printing the elements ....\n");
14.  for(i=0;i<3;i++)
15.  {
16.      printf("\n");
17.      for (j=0;j<3;j++)
18.      {
19.          printf("%d\t",arr[i][j]);
20.      }
21.  }
22. }

```

C# Example

```

1.  using System;
2.
3.  public class Program
4.  {
5.      public static void Main()
6.      {
7.          int[,] arr = new int[3,3];
8.          for (int i=0;i<3;i++)
9.          {
10.             for (int j=0;j<3;j++)
11.             {
12.                 Console.WriteLine("Enter Element");
13.                 arr[i,j]= Convert.ToInt32(Console.ReadLine());
14.             }
15.         }
16.         Console.WriteLine("Printing Elements...");
17.         for (int i=0;i<3;i++)
18.         {
19.             Console.WriteLine();
20.             for (int j=0;j<3;j++)
21.             {
22.                 Console.Write(arr[i,j]+" ");
23.             }
24.         }
25.     }
26. }

```

Mapping 2D array to 1D array

When it comes to map a 2 dimensional array, most of us might think that why this mapping is required. However, 2 D arrays exists from the user point of view. 2D arrays are created to implement a relational

database table lookalike data structure, in computer memory, the storage technique for 2D array is similar to that of an one dimensional array.

The size of a two dimensional array is equal to the multiplication of number of rows and the number of columns present in the array. We do need to map two dimensional array to the one dimensional array in order to store them in the memory.

A 3 X 3 two dimensional array is shown in the following image. However, this array needs to be mapped to a one dimensional array in order to store it into the memory.

	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

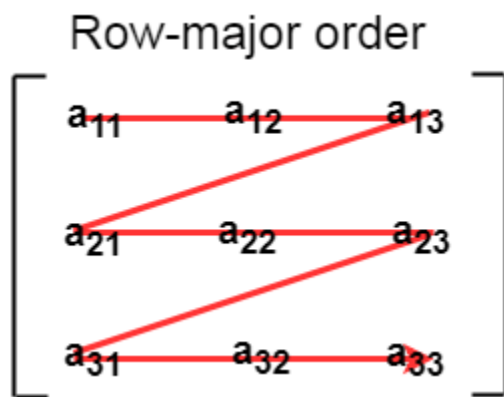
There are two main techniques of storing 2D array elements into memory

1. Row Major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is shown as follows.

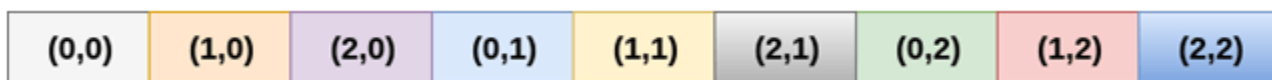
(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)
-------	-------	-------	-------	-------	-------	-------	-------	-------

first, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.

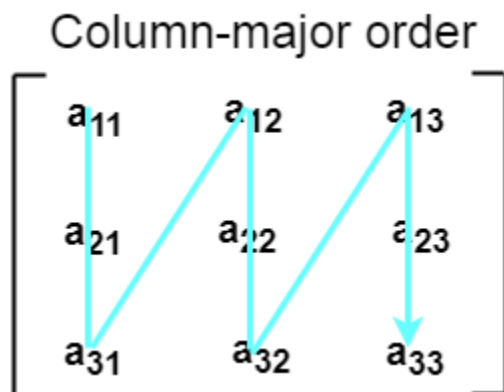


2. Column Major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in the above image is given as follows.



first, the 1st column of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.



Calculating the Address of the random element of a 2D array

Due to the fact that, there are two different techniques of storing the two dimensional array into the memory, there are two different formulas to calculate the address of a random element of the 2D array.

By Row Major Order

If array is declared by $a[m][n]$ where m is the number of rows while n is the number of columns, then address of an element $a[i][j]$ of the array stored in row major order is calculated as,

$$1. \text{Address}(a[i][j]) = B. A. + (i * n + j) * \text{size}$$

where, B. A. is the base address or the address of the first element of the array $a[0][0]$.

Example :

1. $a[10 \dots 30, 55 \dots 75]$, base address of the array (BA) = 0, size of an element = 4 bytes .
2. Find the location of $a[15][68]$.
- 3.
4. $\text{Address}(a[15][68]) = 0 +$
5. $((15 - 10) \times (68 - 55 + 1) + (68 - 55)) \times 4$
- 6.
7. $= (5 \times 14 + 13) \times 4$
8. $= 83 \times 4$
9. $= 332$ answer

By Column major order

If array is declared by $a[m][n]$ where m is the number of rows while n is the number of columns, then address of an element $a[i][j]$ of the array stored in row major order is calculated as,

$$1. \text{Address}(a[i][j]) = ((j * m) + i) * \text{Size} + BA$$

where BA is the base address of the array.

Example:

1. $A[-5 \dots +20][20 \dots 70]$, BA = 1020, Size of element = 8 bytes. Find the location of $a[0][30]$.
- 2.
3. $\text{Address } [A[0][30]) = ((30 - 20) \times 24 + 5) \times 8 + 1020 = 245 \times 8 + 1020 = 2980$ bytes

Linked List

Before understanding the linked list concept, we first look at ***why there is a need for a linked list.***

If we want to store the value in a memory, we need a memory manager that manages the memory for every variable. For example, if we want to create a variable of integer type like:

1. `int x;`

In the above example, we have created a variable 'x' of type integer. As we know that integer variable occupies 4 bytes, so 'x' variable will occupy 4 bytes to store the value.

Suppose we want to create an array of integer type like:

1. `int x[3];`

In the above example, we have declared an array of size 3. As we know, that all the values of an array are stored in a continuous manner, so all the three values of an array are stored in a sequential fashion. The total memory space occupied by the array would be **$3 * 4 = 12$ bytes.**

There are two major drawbacks of using array:

- We cannot insert more than 3 elements in the above example because only 3 spaces are allocated for 3 elements.
- In the case of an array, lots of wastage of memory can occur. For example, if we declare an array of 50 size but we insert only 10 elements in an array. So, in this case, the memory space for other 40 elements will get wasted and cannot be used by another variable as this whole space is occupied by an array.

In array, we are providing the fixed-size at the compile-time, due to which wastage of memory occurs. The solution to this problem is to use the **linked list**.

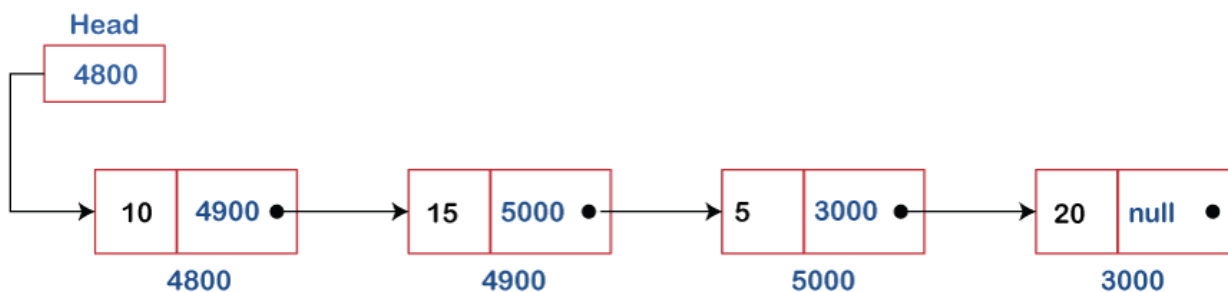
What is Linked List?

A linked list is also a collection of elements, but the elements are not stored in a consecutive location.

Suppose a programmer made a request for storing the integer value then size of 4-byte memory block is assigned to the integer value. The programmer made another request for storing 3 more integer elements; then, three different memory blocks are assigned to these three elements but the memory blocks are available in a random location. So, how are the elements connected?.

These elements are linked to each other by providing one additional information along with an element, i.e., the address of the next element. The variable that stores the address of the next element is known as a pointer. Therefore, we conclude that the linked list contains two parts, i.e., the first one is **the data element**, and the other is the **pointer**. The pointer variable will occupy 4 bytes which is pointing to the next element.

A linked list can also be defined as the collection of the nodes in which one node is connected to another node, and node consists of two parts, i.e., one is the data part and the second one is the address part, as shown in the below figure:



In the above figure, we can observe that each node contains the data and the address of the next node. The last node of the linked list contains the **NULL** value in the address part.

How can we declare the Linked list?

The declaration of an array is very simple as it is of single type. But the linked list contains two parts, which are of two different types, i.e., one is a simple variable, and the second one is a pointer variable. We can declare the linked list by using the user-defined data type known as structure.

The structure of a linked list can be defined as:

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
5. }
```

In the above declaration, we have defined a structure named as **a node** consisting of two variables: an integer variable (data), and the other one is the pointer (next), which contains the address of the next node.

Advantages of using a Linked list over Array

The following are the advantages of using a linked list over an array:

- **Dynamic data structure:**
The size of the linked list is not fixed as it can vary according to our requirements.
- **Insertion and Deletion:**
Insertion and deletion in linked list are easier than array as the elements in an array are stored in a consecutive location. In contrast, in the case of a linked list, the elements are stored in a random location. The complexity for insertion and deletion of elements from the beginning is $O(1)$ in the linked list, while in the case of an array, the complexity would be $O(n)$. If we want to insert or delete the element in an array, then we need to shift the elements for creating the space. On the other hand, in the linked list, we do not have to shift the elements. In the linked list, we just need to update the address of the pointer in the node.
- **Memory efficient**
Its memory consumption is efficient as the size of the linked list can grow or shrink according to our requirements.
- **Implementation**
Both the stacks and queues can be implemented using a linked list.

Disadvantages of Linked list

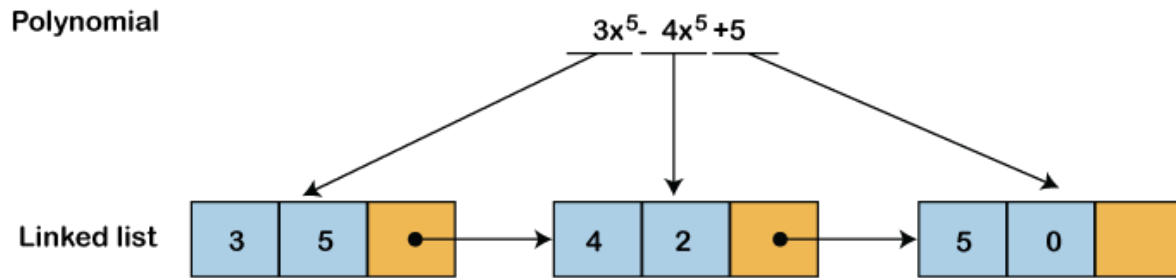
The following are the disadvantages of linked list:

- **Memory usage**
The node in a linked list occupies more memory than array as each node occupies two types of variables, i.e., one is a simple variable, and another is a pointer variable that occupies 4 bytes in the memory.
- **Traversal**
In a linked list, the traversal is not easy. If we want to access the element in a linked list, we cannot access the element randomly, but in the case of an array, we can randomly access the element by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- **Reverse traversing**
In a linked list, backtracking or reverse traversing is difficult. In a doubly linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked List

The applications of the linked list are given below:

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial. We know that polynomial is a collection of terms in which each term contains coefficient and power. The coefficients and power of each term are stored as node and link pointer points to the next element in a linked list, so linked list can be used to create, delete and display the polynomial.



- A sparse matrix is used in scientific computation and numerical analysis. So, a linked list is used to represent the sparse matrix.
- The various operations like student's details, employee's details or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Stack, Queue, tree and various other data structures can be implemented using a linked list.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- To implement hashing, we require hash tables. The hash table contains entries that are implemented using linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Types of Linked List

Before knowing about the types of a linked list, we should know what is **linked list**. So, to know about the linked list, click on the link given below:

Types of Linked list

The following are the types of linked list:

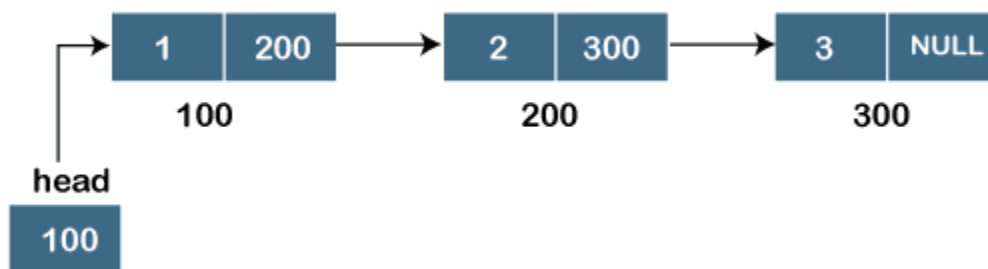
- [Singly Linked list](#)

- [Doubly Linked list](#)
- [Circular Linked list](#)
- [Doubly Circular Linked list](#)

Singly Linked list

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a **pointer**.

Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:



We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a **head pointer**.

The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

Representation of the node in a singly linked list

```

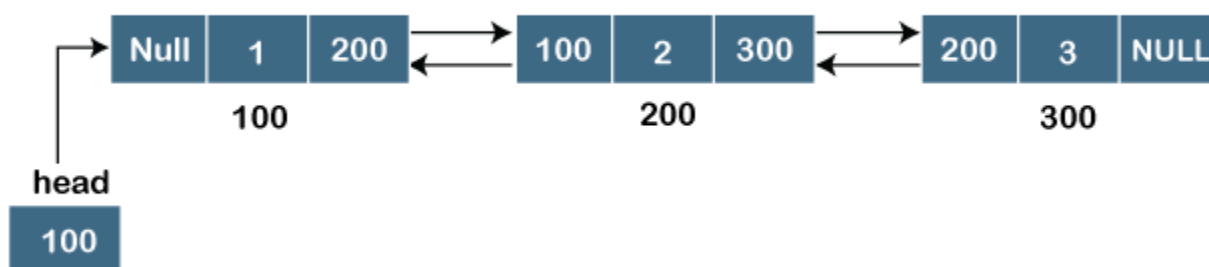
1. struct node
2. {
3.     int data;
4.     struct node *next;
5. }
  
```

In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

Doubly linked list

As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.

Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below:



As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the **address of the next** while the other part of the node stores the **previous node's address**. The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node.

Representation of the node in a doubly linked list

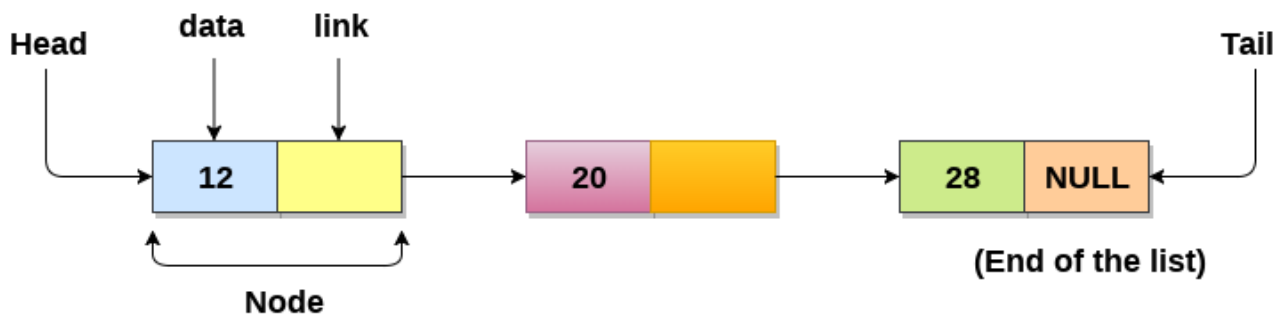
1. struct node
2. {
3. int data;
4. struct node *next;
5. struct node *prev;
6. }

In the above representation, we have defined a user-defined structure named **a node** with three members, one is **data** of integer type, and the other two are the pointers, i.e., **next** and **prev** of the node type. The **next pointer** variable holds the address of the next node, and the **prev pointer** holds the address of the previous node. The type of both the pointers, i.e., **next** and **prev** is **struct node** as both the pointers are storing the address of the node of the **struct node** type.

Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.

- The last node of the list contains pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and be linked together to form a list. This achieves optimized utilization of space.
- List size is limited to the memory size and doesn't need to be declared in advance.
- Empty nodes cannot be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains the following limitations:

1. The size of the array must be known in advance before using it in the program.
2. Increasing the size of the array is a time-consuming process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using a linked list is useful because,

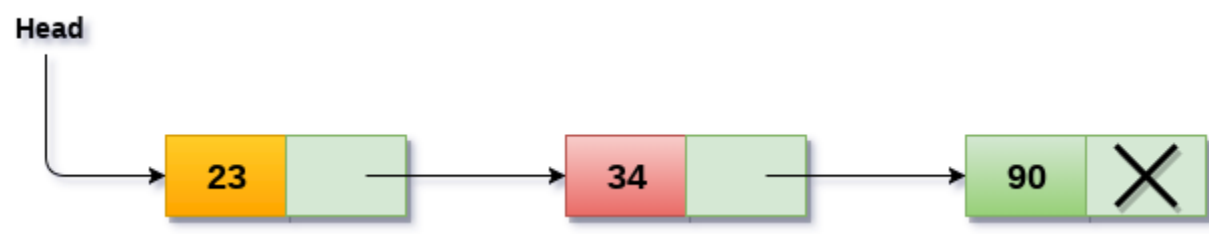
1. It allocates memory dynamically. All the nodes of a linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. The list grows as per the program's demand and is limited to the available memory space.

Singly linked list or One way chain

A singly linked list can be defined as a collection of an ordered set of elements. The number of elements may vary according to the need of the program. A node in the singly linked list consists of two parts: a data part and a link part. The data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Complexity

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Singly Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

1. struct node
2. {
3. int data;
4. struct node *next;
5. };
6. struct node *head, *ptr;
7. ptr = (struct node *)malloc(sizeof(struct node *));

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after	It involves deleting the node after the specified node in the list. we need

	<u>specified node</u>	to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	<u>Traversing</u>	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	<u>Searching</u>	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Linked List in C: Menu Driven Program

```

1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     int data;
6.     struct node *next;
7. };
8. struct node *head;
9.
10. void begininsert ();
11. void lastinsert ();
12. void randominsert();
13. void begin_delete();
14. void last_delete();
15. void random_delete();
16. void display();
17. void search();
18. void main ()
19. {
20.     int choice =0;
21.     while(choice != 9)
22.     {
23.         printf("\n\n*****Main Menu*****\n");
24.         printf("\nChoose one option from the following list ...\n");
25.         printf("\n===== \n");

```

```

26.     printf("\n1.Insert in beginning\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n
27.     5.Delete from last\n6.Delete node after specified location\n7.Search for an element\n8.Show\n9.Exit\n");
28.     printf("\nEnter your choice?\n");
29.     scanf("\n%d",&choice);
30.     switch(choice)
31.     {
32.         case 1:
33.             begininsert();
34.             break;
35.         case 2:
36.             lastinsert();
37.             break;
38.         case 3:
39.             randominsert();
40.             break;
41.         case 4:
42.             begin_delete();
43.             break;
44.         case 5:
45.             last_delete();
46.             break;
47.         case 6:
48.             random_delete();
49.             break;
50.         case 7:
51.             search();
52.             break;
53.         case 8:
54.             display();
55.             break;
56.         case 9:
57.             exit(0);
58.             break;
59.         default:
60.             printf("Please enter valid choice..");
61.     }
62. }
63. }
64. void begininsert()
65. {
66.     struct node *ptr;
67.     int item;
68.     ptr = (struct node *) malloc(sizeof(struct node *));

```



```
69.  if(ptr == NULL)
70.  {
71.      printf("\nOVERFLOW");
72.  }
73.  else
74.  {
75.      printf("\nEnter value\n");
76.      scanf("%d",&item);
77.      ptr->data = item;
78.      ptr->next = head;
79.      head = ptr;
80.      printf("\nNode inserted");
81.  }
82.
83.}
84. void lastinsert()
85.{
86.    struct node *ptr,*temp;
87.    int item;
88.    ptr = (struct node*)malloc(sizeof(struct node));
89.    if(ptr == NULL)
90.    {
91.        printf("\nOVERFLOW");
92.    }
93.    else
94.    {
95.        printf("\nEnter value?\n");
96.        scanf("%d",&item);
97.        ptr->data = item;
98.        if(head == NULL)
99.        {
100.            ptr -> next = NULL;
101.            head = ptr;
102.            printf("\nNode inserted");
103.        }
104.        else
105.        {
106.            temp = head;
107.            while (temp -> next != NULL)
108.            {
109.                temp = temp -> next;
110.            }
111.            temp->next = ptr;
112.            ptr->next = NULL;
113.            printf("\nNode inserted");
```

```

114.
115.     }
116. }
117. }
118. void randominsert()
119. {
120.     int i,loc,item;
121.     struct node *ptr, *temp;
122.     ptr = (struct node *) malloc (sizeof(struct node));
123.     if(ptr == NULL)
124.     {
125.         printf("\nOVERFLOW");
126.     }
127.     else
128.     {
129.         printf("\nEnter element value");
130.         scanf("%d",&item);
131.         ptr->data = item;
132.         printf("\nEnter the location after which you want to insert ");
133.         scanf("\n%d",&loc);
134.         temp=head;
135.         for(i=0;i<loc;i++)
136.         {
137.             temp = temp->next;
138.             if(temp == NULL)
139.             {
140.                 printf("\ncan't insert\n");
141.                 return;
142.             }
143.
144.         }
145.         ptr ->next = temp ->next;
146.         temp ->next = ptr;
147.         printf("\nNode inserted");
148.     }
149. }
150. void begin_delete()
151. {
152.     struct node *ptr;
153.     if(head == NULL)
154.     {
155.         printf("\nList is empty\n");
156.     }
157.     else
158.     {

```

```

159.         ptr = head;
160.         head = ptr->next;
161.         free(ptr);
162.         printf("\nNode deleted from the begining ...\n");
163.     }
164. }
165. void last_delete()
166. {
167.     struct node *ptr,*ptr1;
168.     if(head == NULL)
169.     {
170.         printf("\nlist is empty");
171.     }
172.     else if(head -> next == NULL)
173.     {
174.         head = NULL;
175.         free(head);
176.         printf("\nOnly node of the list deleted ...\n");
177.     }
178.
179.     else
180.     {
181.         ptr = head;
182.         while(ptr->next != NULL)
183.         {
184.             ptr1 = ptr;
185.             ptr = ptr ->next;
186.         }
187.         ptr1->next = NULL;
188.         free(ptr);
189.         printf("\nDeleted Node from the last ...\n");
190.     }
191. }
192. void random_delete()
193. {
194.     struct node *ptr,*ptr1;
195.     int loc,i;
196.     printf("\n Enter the location of the node after which you want to perform deletion \n");
197.     scanf("%d",&loc);
198.     ptr=head;
199.     for(i=0;i<loc;i++)
200.     {
201.         ptr1 = ptr;
202.         ptr = ptr->next;
203.

```

```

204.         if(ptr == NULL)
205.         {
206.             printf("\nCan't delete");
207.             return;
208.         }
209.     }
210.     ptr1 ->next = ptr ->next;
211.     free(ptr);
212.     printf("\nDeleted node %d ",loc+1);
213. }
214. void search()
215. {
216.     struct node *ptr;
217.     int item,i=0,flag;
218.     ptr = head;
219.     if(ptr == NULL)
220.     {
221.         printf("\nEmpty List\n");
222.     }
223.     else
224.     {
225.         printf("\nEnter item which you want to search?\n");
226.         scanf("%d",&item);
227.         while (ptr!=NULL)
228.         {
229.             if(ptr->data == item)
230.             {
231.                 printf("item found at location %d ",i+1);
232.                 flag=0;
233.             }
234.             else
235.             {
236.                 flag=1;
237.             }
238.             i++;
239.             ptr = ptr -> next;
240.         }
241.         if(flag==1)
242.         {
243.             printf("Item not found\n");
244.         }
245.     }
246.
247. }
248.

```

```

249.     void display()
250.     {
251.         struct node *ptr;
252.         ptr = head;
253.         if(ptr == NULL)
254.         {
255.             printf("Nothing to print");
256.         }
257.         else
258.         {
259.             printf("\nprinting values . . . .\n");
260.             while (ptr!=NULL)
261.             {
262.                 printf("\n%d",ptr->data);
263.                 ptr = ptr -> next;
264.             }
265.         }
266.     }
267.

```

Output:

```

*****Main Menu*****

Choose one option from the following list ...

=====

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
1

Enter value
1

Node inserted

*****Main Menu*****

Choose one option from the following list ...

=====

1.Insert in begining

```

```
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
```

Enter your choice?

2

Enter value?

2

Node inserted

*****Main Menu*****

Choose one option from the following list ...

=====

```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
```

Enter your choice?

3

Enter element value1

Enter the location after which you want to insert 1

Node inserted

*****Main Menu*****

Choose one option from the following list ...

=====

```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
```

Enter your choice?

8

printing values

1
2
1

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete node after specified location
- 7.Search for an element
- 8.Show
- 9.Exit

Enter your choice?

2

Enter value?

123

Node inserted

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete node after specified location
- 7.Search for an element
- 8.Show
- 9.Exit

Enter your choice?

1

Enter value

1234

Node inserted

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last

```
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
```

Enter your choice?

4

Node deleted from the begining ...

*****Main Menu*****

Choose one option from the following list ...

=====

```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
```

Enter your choice?

5

Deleted Node from the last ...

*****Main Menu*****

Choose one option from the following list ...

=====

```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
```

Enter your choice?

6

Enter the location of the node after which you want to perform deletion

1

Deleted node 2

*****Main Menu*****

Choose one option from the following list ...

=====

```
1.Insert in begining
2.Insert at last
```



```
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
```

Enter your choice?

8

printing values

1

1

*****Main Menu*****

Choose one option from the following list ...

=====

```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
```

Enter your choice?

7

Enter item which you want to search?

1

item found at location 1

item found at location 2

*****Main Menu*****

Choose one option from the following list ...

=====

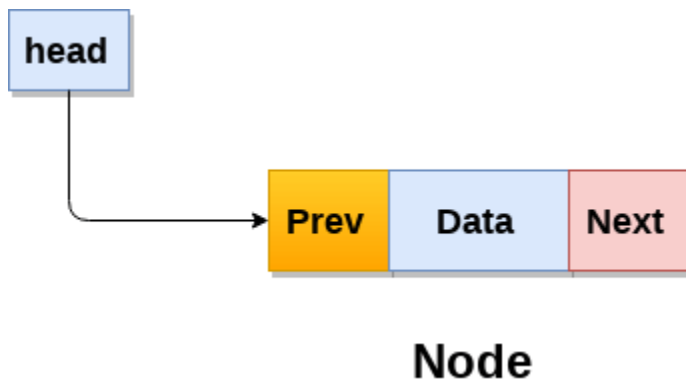
```
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
```

Enter your choice?

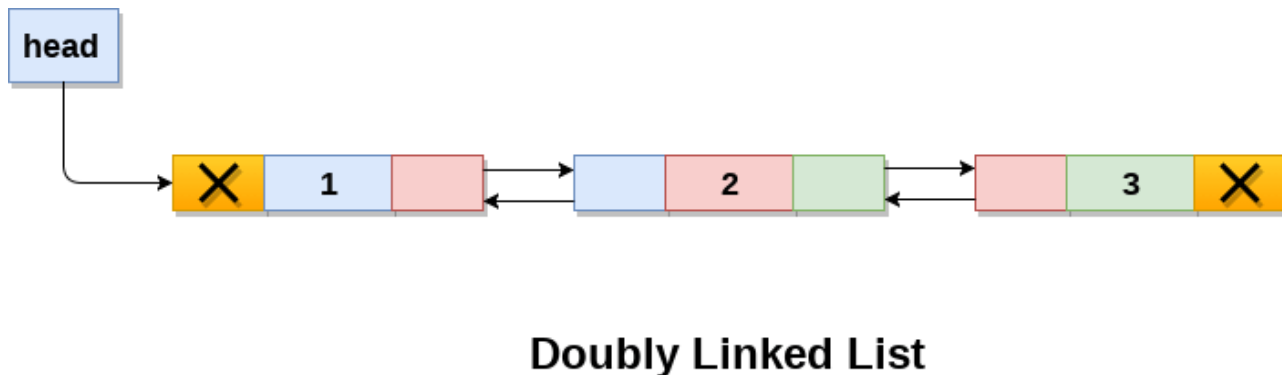
9

Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



In C, structure of a node in doubly linked list can be given as :

1. struct node
2. {
3. struct node *prev;
4. int data;
5. struct node *next;
6. }

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this

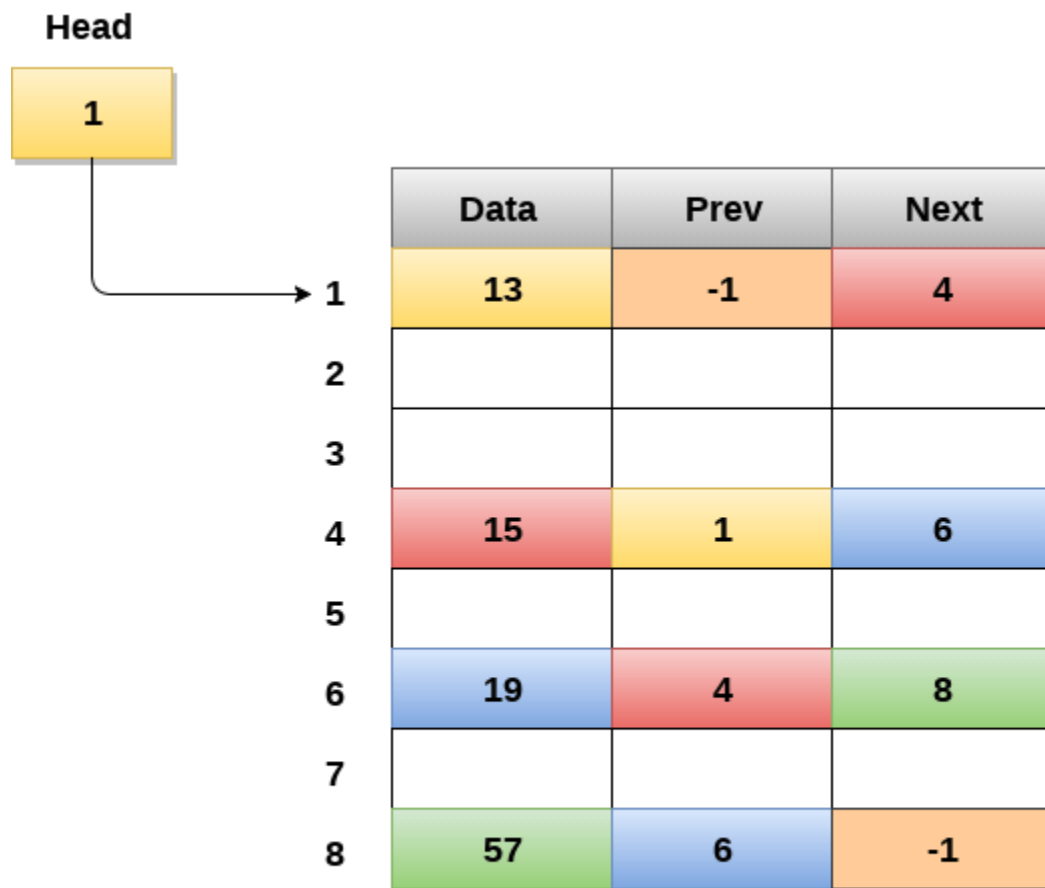
limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

Operations on doubly linked list

Node Creation

```

1. struct node
2. {
3.     struct node *prev;
4.     int data;
5.     struct node *next;
6. };
7. struct node *head;

```

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description
1	<u>Insertion at beginning</u>	Adding the node into the linked list at beginning.
2	<u>Insertion at end</u>	Adding the node into the linked list to the end.
3	<u>Insertion after specified node</u>	Adding the node into the linked list after the specified node.
4	<u>Deletion at beginning</u>	Removing the node from beginning of the list
5	<u>Deletion at the end</u>	Removing the node from end of the list.
6	<u>Deletion of the node having given data</u>	Removing the node which is present just after the node containing the given data.
7	<u>Searching</u>	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	<u>Traversing</u>	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Menu Driven Program in C to implement all the operations of doubly linked list

```

1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     struct node *prev;
6.     struct node *next;
7.     int data;
8. };
9. struct node *head;
10. void insertion_beginning();
11. void insertion_last();
12. void insertion_specified();
13. void deletion_beginning();
14. void deletion_last();
15. void deletion_specified();
16. void display();
17. void search();
18. void main ()
19. {
20.     int choice =0;
21.     while(choice != 9)
22.     {
23.         printf("\n*****Main Menu*****\n");
24.         printf("\nChoose one option from the following list ...\n");
25.         printf("\n===== \n");
26.         printf("\n1.Insert in beginning\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n
27.         5.Delete from last\n6.Delete the node after the given data\n7.Search\n8.Show\n9.Exit\n");
28.         printf("\nEnter your choice?\n");
29.         scanf("\n%d",&choice);
30.         switch(choice)
31.         {
32.             case 1:
33.                 insertion_beginning();
34.                 break;
35.             case 2:
36.                 insertion_last();
37.                 break;
38.             case 3:
39.                 insertion_specified();
40.                 break;
41.             case 4:
42.                 deletion_beginning();
43.                 break;
44.             case 5:

```

```

45.     deletion_last();
46.     break;
47.     case 6:
48.     deletion_specified();
49.     break;
50.     case 7:
51.     search();
52.     break;
53.     case 8:
54.     display();
55.     break;
56.     case 9:
57.     exit(0);
58.     break;
59.     default:
60.     printf("Please enter valid choice..");
61. }
62. }
63.}

64. void insertion_beginning()
65. {
66.     struct node *ptr;
67.     int item;
68.     ptr = (struct node *)malloc(sizeof(struct node));
69.     if(ptr == NULL)
70.     {
71.         printf("\nOVERFLOW");
72.     }
73.     else
74.     {
75.         printf("\nEnter Item value");
76.         scanf("%d",&item);
77.
78.         if(head==NULL)
79.         {
80.             ptr->next = NULL;
81.             ptr->prev=NULL;
82.             ptr->data=item;
83.             head=ptr;
84.         }
85.         else
86.         {
87.             ptr->data=item;
88.             ptr->prev=NULL;
89.             ptr->next = head;

```

```

90.     head->prev=ptr;
91.     head=ptr;
92. }
93. printf("\nNode inserted\n");
94.}
95.
96.}
97. void insertion_last()
98.{
99. struct node *ptr,*temp;
100.     int item;
101.     ptr = (struct node *) malloc(sizeof(struct node));
102.     if(ptr == NULL)
103.     {
104.         printf("\nOVERFLOW");
105.     }
106.     else
107.     {
108.         printf("\nEnter value");
109.         scanf("%d",&item);
110.         ptr->data=item;
111.         if(head == NULL)
112.         {
113.             ptr->next = NULL;
114.             ptr->prev = NULL;
115.             head = ptr;
116.         }
117.         else
118.         {
119.             temp = head;
120.             while(temp->next!=NULL)
121.             {
122.                 temp = temp->next;
123.             }
124.             temp->next = ptr;
125.             ptr ->prev=temp;
126.             ptr->next = NULL;
127.         }
128.
129.     }
130.     printf("\nnode inserted\n");
131. }
132. void insertion_specified()
133. {
134.     struct node *ptr,*temp;

```

```

135.     int item,loc,i;
136.     ptr = (struct node *)malloc(sizeof(struct node));
137.     if(ptr == NULL)
138.     {
139.         printf("\n OVERFLOW");
140.     }
141.     else
142.     {
143.         temp=head;
144.         printf("Enter the location");
145.         scanf("%d",&loc);
146.         for(i=0;i<loc;i++)
147.         {
148.             temp = temp->next;
149.             if(temp == NULL)
150.             {
151.                 printf("\n There are less than %d elements", loc);
152.                 return;
153.             }
154.         }
155.         printf("Enter value");
156.         scanf("%d",&item);
157.         ptr->data = item;
158.         ptr->next = temp->next;
159.         ptr -> prev = temp;
160.         temp->next = ptr;
161.         temp->next->prev=ptr;
162.         printf("\nnode inserted\n");
163.     }
164. }
165. void deletion_beginning()
166. {
167.     struct node *ptr;
168.     if(head == NULL)
169.     {
170.         printf("\n UNDERFLOW");
171.     }
172.     else if(head->next == NULL)
173.     {
174.         head = NULL;
175.         free(head);
176.         printf("\nnode deleted\n");
177.     }
178.     else
179.     {

```



```

180.         ptr = head;
181.         head = head -> next;
182.         head -> prev = NULL;
183.         free(ptr);
184.         printf("\nnode deleted\n");
185.     }
186.
187. }
188. void deletion_last()
189. {
190.     struct node *ptr;
191.     if(head == NULL)
192.     {
193.         printf("\n UNDERFLOW");
194.     }
195.     else if(head->next == NULL)
196.     {
197.         head = NULL;
198.         free(head);
199.         printf("\nnode deleted\n");
200.     }
201.     else
202.     {
203.         ptr = head;
204.         if(ptr->next != NULL)
205.         {
206.             ptr = ptr -> next;
207.         }
208.         ptr -> prev -> next = NULL;
209.         free(ptr);
210.         printf("\nnode deleted\n");
211.     }
212. }
213. void deletion_specified()
214. {
215.     struct node *ptr, *temp;
216.     int val;
217.     printf("\n Enter the data after which the node is to be deleted : ");
218.     scanf("%d", &val);
219.     ptr = head;
220.     while(ptr -> data != val)
221.         ptr = ptr -> next;
222.     if(ptr -> next == NULL)
223.     {
224.         printf("\nCan't delete\n");

```

```

225.     }
226.     else if(ptr -> next -> next == NULL)
227.     {
228.         ptr ->next = NULL;
229.     }
230.     else
231.     {
232.         temp = ptr -> next;
233.         ptr -> next = temp -> next;
234.         temp -> next -> prev = ptr;
235.         free(temp);
236.         printf("\nnode deleted\n");
237.     }
238. }
239. void display()
240. {
241.     struct node *ptr;
242.     printf("\n printing values...\n");
243.     ptr = head;
244.     while(ptr != NULL)
245.     {
246.         printf("%d\n",ptr->data);
247.         ptr=ptr->next;
248.     }
249. }
250. void search()
251. {
252.     struct node *ptr;
253.     int item,i=0,flag;
254.     ptr = head;
255.     if(ptr == NULL)
256.     {
257.         printf("\nEmpty List\n");
258.     }
259.     else
260.     {
261.         printf("\nEnter item which you want to search?\n");
262.         scanf("%d",&item);
263.         while (ptr!=NULL)
264.         {
265.             if(ptr->data == item)
266.             {
267.                 printf("\nitem found at location %d ",i+1);
268.                 flag=0;
269.                 break;

```

```

270.         }
271.         else
272.         {
273.             flag=1;
274.         }
275.         i++;
276.         ptr = ptr -> next;
277.     }
278.     if(flag==1)
279.     {
280.         printf("\nItem not found\n");
281.     }
282. }
283.
284. }

```

Output

```
*****Main Menu*****
```

Choose one option from the following list ...

```
=====
```

```

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

```

Enter your choice?

```
8
```

printing values...

```
*****Main Menu*****
```

Choose one option from the following list ...

```
=====
```

```

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

```

Enter your choice?

```
1
```

Enter Item value12

Node inserted

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

1

Enter Item value123

Node inserted

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

1

Enter Item value1234

Node inserted

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data

7.Search
8.Show
9.Exit

Enter your choice?

8

printing values...

1234

123

12

*****Main Menu*****

Choose one option from the following list ...

=====

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?

2

Enter value89

node inserted

*****Main Menu*****

Choose one option from the following list ...

=====

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?

3

Enter the location1

Enter value12345

node inserted

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

8

printing values...

1234

123

12345

12

89

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

4

node deleted

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

5

node deleted

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

8

printing values...

123

12345

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

6

Enter the data after which the node is to be deleted : 123

*****Main Menu*****

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

8

```
printing values...
123
```

```
*****Main Menu*****
```

```
Choose one option from the following list ...
```

```
=====
```

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

```
Enter your choice?
```

```
7
```

```
Enter item which you want to search?
```

```
123
```

```
item found at location 1
```

```
*****Main Menu*****
```

```
Choose one option from the following list ...
```

```
=====
```

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

```
Enter your choice?
```

```
6
```

```
Enter the data after which the node is to be deleted : 123
```

```
Can't delete
```

```
*****Main Menu*****
```

```
Choose one option from the following list ...
```

```
=====
```

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show


```
9.Exit
```

```
Enter your choice?
```

```
9
```

```
Exited..
```

What is a Stack?

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a **stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.**

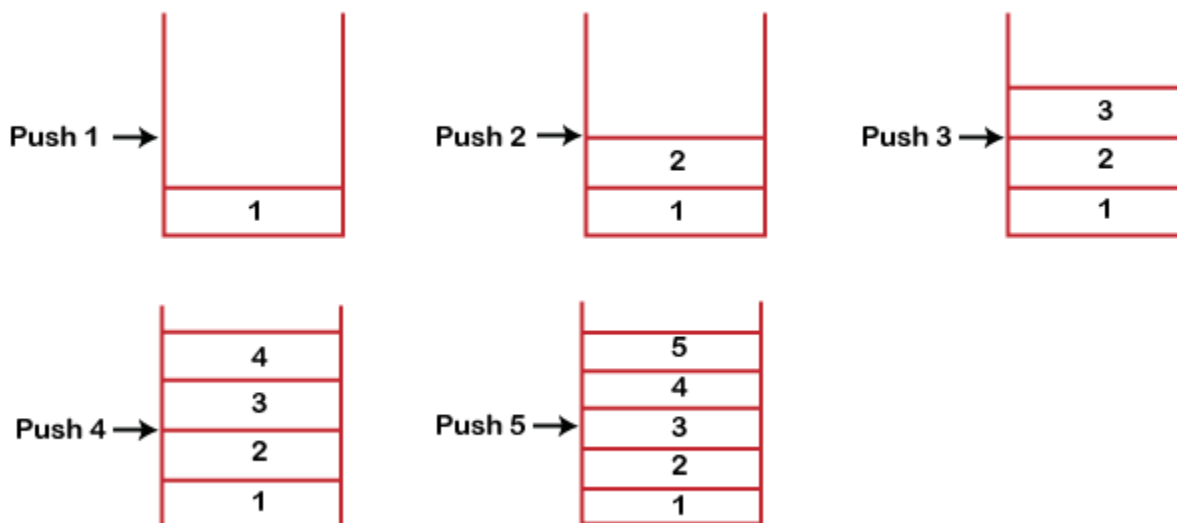
Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

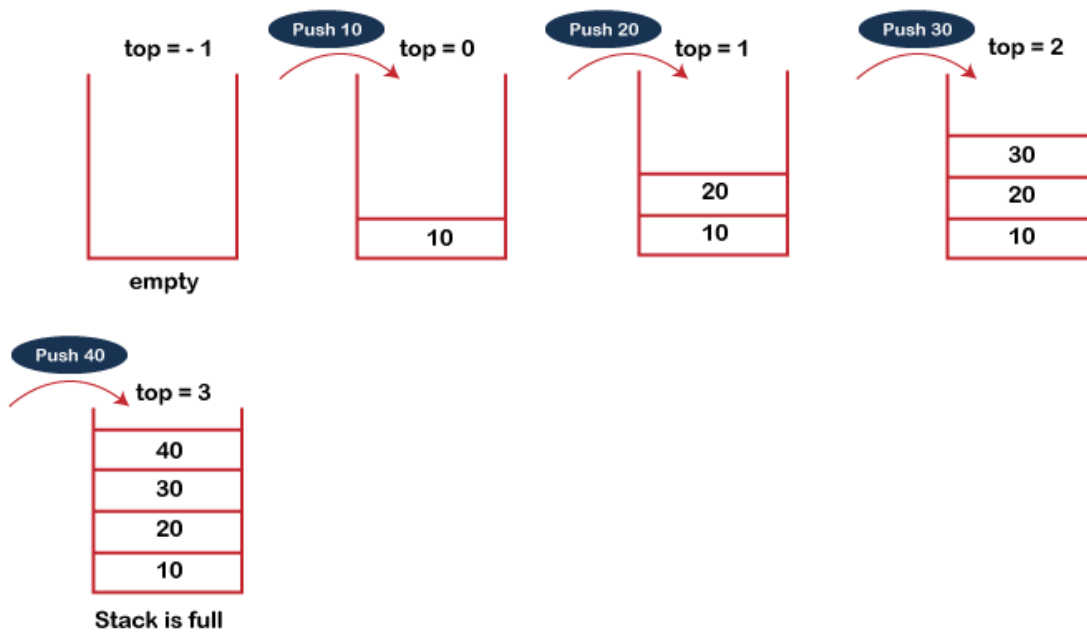
The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

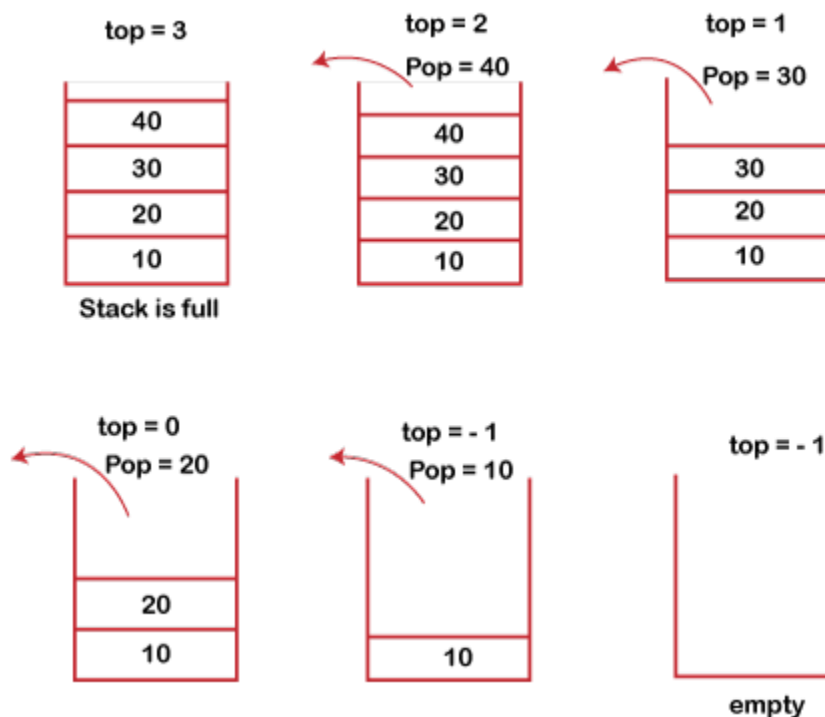
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the **overflow** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1**, and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the **max** size of the stack.



POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the **underflow** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the **top**
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Applications of Stack

The following are the applications of the stack:

- **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:

```
1. int main()  
2. {  
3.     cout<<"Hello";  
4.     cout<<"javaTpoint";  
5. }
```

As we know, each program has an *opening* and *closing* braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "javaTpoint" string, so we can achieve this with the help of a stack.
First, we push all the characters of the string in a stack until we reach the null character.
After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
- **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
 - Infix to prefix
 - Infix to postfix
 - Prefix to infix
 - Prefix to postfix
 - Postfix to infix
- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are

assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refer to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

1. begin
2. **if** top = n then stack full
3. top = top + 1
4. stack (top) : = item;
5. end

Time Complexity : $O(1)$

implementation of push algorithm in C language

1. **void** push (**int** val,**int** n) //n is size of the stack
2. {
3. **if** (top == n)
4. printf("\n Overflow");
5. **else**
6. {
7. top = top +1;
8. stack[top] = val;
9. }
10. }

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is

stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

1. begin
2. **if** top = 0 then stack empty;
3. item := stack(top);
4. top = top - 1;
5. end;

Time Complexity : $O(1)$

Implementation of POP algorithm using C language

1. **int** pop ()
2. {
3. **if**(top == -1)
4. {
5. printf("Underflow");
6. **return** 0;
7. }
8. **else**
9. {
10. **return** stack[top - -];
11. }
12. }

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

PEEK (STACK, TOP)

1. Begin
2. **if** top = -1 then stack empty
3. item = stack[top]
4. **return** item
5. End

Time complexity: $O(n)$

Implementation of Peek algorithm in C language

1. **int** peek()
2. {

```

3.  if (top == -1)
4.  {
5.      printf("Underflow");
6.      return 0;
7.  }
8.  else
9.  {
10.     return stack [top];
11. }
12.}

```

C program

```

1. #include <stdio.h>
2. int stack[100],i,j,choice=0,n,top=-1;
3. void push();
4. void pop();
5. void show();
6. void main ()
7. {
8.
9.     printf("Enter the number of elements in the stack ");
10.    scanf("%d",&n);
11.    printf("*****Stack operations using array*****");
12.
13.    printf("\n-----\n");
14.    while(choice != 4)
15.    {
16.        printf("Chose one from the below options...\n");
17.        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
18.        printf("\n Enter your choice \n");
19.        scanf("%d",&choice);
20.        switch(choice)
21.        {
22.            case 1:
23.            {
24.                push();
25.                break;
26.            }
27.            case 2:
28.            {
29.                pop();
30.                break;
31.            }
32.            case 3:
33.            {

```

```
34.         show();
35.         break;
36.     }
37.     case 4:
38.     {
39.         printf("Exiting....");
40.         break;
41.     }
42.     default:
43.     {
44.         printf("Please Enter valid choice ");
45.     }
46. };
47. }
48.}
49.
50. void push ()
51. {
52.     int val;
53.     if (top == n )
54.         printf("\n Overflow");
55.     else
56.     {
57.         printf("Enter the value?");
58.         scanf("%d",&val);
59.         top = top +1;
60.         stack[top] = val;
61.     }
62.}
63.
64. void pop ()
65. {
66.     if(top == -1)
67.         printf("Underflow");
68.     else
69.         top = top -1;
70.}
71. void show()
72. {
73.     for (i=top;i>=0;i--)
74.     {
75.         printf("%d\n",stack[i]);
76.     }
77.     if(top == -1)
78.     {
```



```
79.     printf("Stack is empty");
80. }
81. }
```

Java Program

```
1. import java.util.Scanner;
2. class Stack
3. {
4.     int top;
5.     int maxsize = 10;
6.     int[] arr = new int[maxsize];
7.
8.
9.     boolean isEmpty()
10.    {
11.        return (top < 0);
12.    }
13.    Stack()
14.    {
15.        top = -1;
16.    }
17.    boolean push (Scanner sc)
18.    {
19.        if(top == maxsize-1)
20.        {
21.            System.out.println("Overflow !!");
22.            return false;
23.        }
24.        else
25.        {
26.            System.out.println("Enter Value");
27.            int val = sc.nextInt();
28.            top++;
29.            arr[top]=val;
30.            System.out.println("Item pushed");
31.            return true;
32.        }
33.    }
34.    boolean pop ()
35.    {
36.        if (top == -1)
37.        {
38.            System.out.println("Underflow !!");
39.            return false;
40.        }
```

```

41.     else
42.     {
43.         top --;
44.         System.out.println("Item popped");
45.         return true;
46.     }
47. }
48. void display ()
49. {
50.     System.out.println("Printing stack elements .....");
51.     for(int i = top; i>=0;i--)
52.     {
53.         System.out.println(arr[i]);
54.     }
55. }
56. }
57. public class Stack_Operations {
58. public static void main(String[] args) {
59.     int choice=0;
60.     Scanner sc = new Scanner(System.in);
61.     Stack s = new Stack();
62.     System.out.println("*****Stack operations using array*****\n");
63.     System.out.println("\n-----\n");
64.     while(choice != 4)
65.     {
66.         System.out.println("\nChose one from the below options...\n");
67.         System.out.println("\n1.Push\n2.Pop\n3.Show\n4.Exit");
68.         System.out.println("\n Enter your choice \n");
69.         choice = sc.nextInt();
70.         switch(choice)
71.         {
72.             case 1:
73.             {
74.                 s.push(sc);
75.                 break;
76.             }
77.             case 2:
78.             {
79.                 s.pop();
80.                 break;
81.             }
82.             case 3:
83.             {
84.                 s.display();
85.                 break;

```

```

86.     }
87.     case 4:
88.     {
89.         System.out.println("Exiting....");
90.         System.exit(0);
91.         break;
92.     }
93.     default:
94.     {
95.         System.out.println("Please Enter valid choice ");
96.     }
97. };
98. }
99. }
100. }

```

C# Program

```

1. using System;
2.
3. public class Stack
4. {
5.     int top;
6.     int maxsize=10;
7.     int[] arr = new int[10];
8.     public static void Main()
9.     {
10.        Stack st = new Stack();
11.        st.top=-1;
12.        int choice=0;
13.        Console.WriteLine("*****Stack operations using array*****");
14.        Console.WriteLine("\n-----\n");
15.        while(choice != 4)
16.        {
17.            Console.WriteLine("Chose one from the below options...\n");
18.            Console.WriteLine("\n1.Push\n2.Pop\n3.Show\n4.Exit");
19.            Console.WriteLine("\n Enter your choice \n");
20.            choice = Convert.ToInt32(Console.ReadLine());
21.            switch(choice)
22.            {
23.                case 1:
24.                {
25.                    st.push();
26.                    break;
27.                }
28.                case 2:

```

```
29.     {
30.         st.pop();
31.         break;
32.     }
33.     case 3:
34.     {
35.         st.show();
36.         break;
37.     }
38.     case 4:
39.     {
40.         Console.WriteLine("Exiting....");
41.         break;
42.     }
43.     default:
44.     {
45.         Console.WriteLine("Please Enter valid choice ");
46.         break;
47.     }
48. };
49. }
50. }
51.
52. Boolean push ()
53. {
54.     int val;
55.     if(top == maxsize-1)
56.     {
57.
58.         Console.WriteLine("\n Overflow");
59.         return false;
60.     }
61.     else
62.     {
63.         Console.WriteLine("Enter the value?");
64.         val = Convert.ToInt32(Console.ReadLine());
65.         top = top +1;
66.         arr[top] = val;
67.         Console.WriteLine("Item pushed");
68.         return true;
69.     }
70. }
71.
72. Boolean pop ()
73. {
```

```

74.  if (top == -1)
75.  {
76.      Console.WriteLine("Underflow");
77.      return false;
78.  }
79.
80.  else
81.  {
82.      {
83.          top = top - 1;
84.          Console.WriteLine("Item popped");
85.          return true;
86.      }
87. }
88. void show()
89. {
90.
91.     for (int i=top;i>=0;i--)
92.     {
93.         Console.WriteLine(arr[i]);
94.     }
95.     if(top == -1)
96.     {
97.         Console.WriteLine("Stack is empty");
98.     }
99. }
100.     }

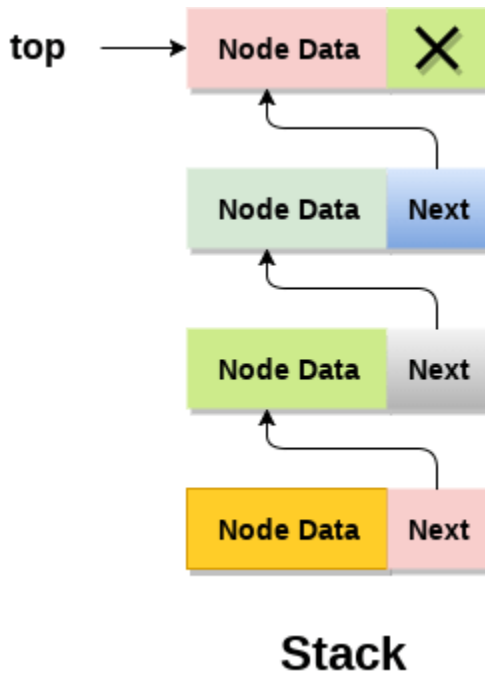
```

○

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



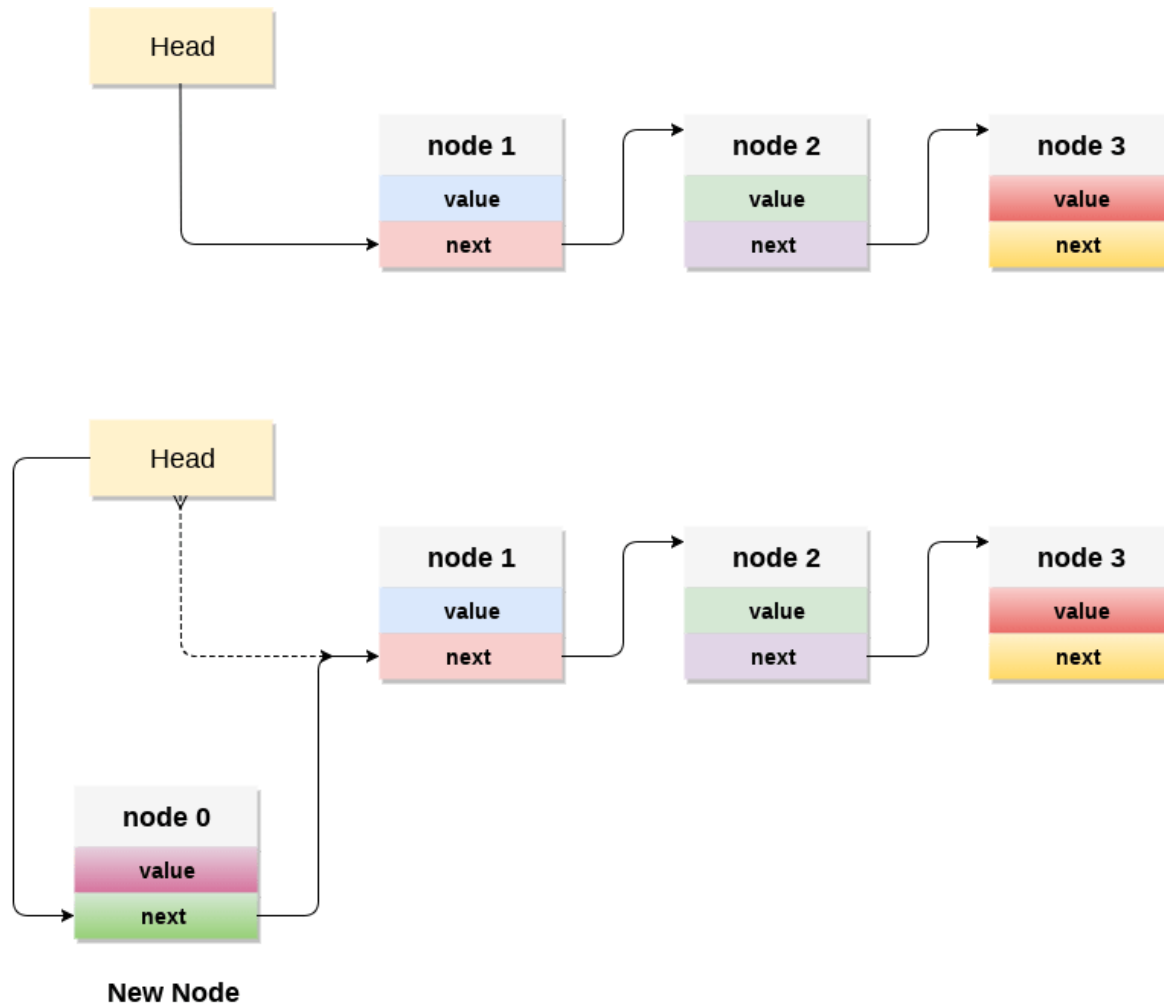
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : $O(1)$



C implementation :

```

1. void push ()
2. {
3.     int val;
4.     struct node *ptr =(struct node*)malloc(sizeof(struct node));
5.     if(ptr == NULL)
6.     {
7.         printf("not able to push the element");
8.     }
9.     else
10.    {
11.        printf("Enter the value");
12.        scanf("%d",&val);
13.        if(head==NULL)
14.        {
15.            ptr->val = val;
16.            ptr -> next = NULL;
17.            head=ptr;
18.        }
19.        else

```

```

20.    {
21.        ptr->val = val;
22.        ptr->next = head;
23.        head=ptr;
24.
25.    }
26.    printf("Item pushed");
27.
28. }
29.}

```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

30. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
31. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : $O(n)$

C implementation

```

1. void pop()
2. {
3.     int item;
4.     struct node *ptr;
5.     if (head == NULL)
6.     {
7.         printf("Underflow");
8.     }
9.     else
10.    {
11.        item = head->val;
12.        ptr = head;
13.        head = head->next;
14.        free(ptr);
15.        printf("Item popped");
16.
17.    }
18.}

```


Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

19. Copy the head pointer into a temporary pointer.

20. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : $O(n)$

C Implementation

```
1. void display()
2. {
3.     int i;
4.     struct node *ptr;
5.     ptr=head;
6.     if(ptr == NULL)
7.     {
8.         printf("Stack is empty\n");
9.     }
10.    else
11.    {
12.        printf("Printing Stack elements \n");
13.        while(ptr!=NULL)
14.        {
15.            printf("%d\n",ptr->val);
16.            ptr = ptr->next;
17.        }
18.    }
19.}
```

Menu Driven program in C implementing all the stack operations using linked list :

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. void push();
4. void pop();
5. void display();
6. struct node
7. {
8.     int val;
9.     struct node *next;
10.};
11. struct node *head;
12.
```

```

13. void main ()
14. {
15.     int choice=0;
16.     printf("\n*****Stack operations using linked list*****\n");
17.     printf("\n-----\n");
18.     while(choice != 4)
19.     {
20.         printf("\n\nChose one from the below options...\n");
21.         printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
22.         printf("\n Enter your choice \n");
23.         scanf("%d",&choice);
24.         switch(choice)
25.         {
26.             case 1:
27.             {
28.                 push();
29.                 break;
30.             }
31.             case 2:
32.             {
33.                 pop();
34.                 break;
35.             }
36.             case 3:
37.             {
38.                 display();
39.                 break;
40.             }
41.             case 4:
42.             {
43.                 printf("Exiting....");
44.                 break;
45.             }
46.             default:
47.             {
48.                 printf("Please Enter valid choice ");
49.             }
50.         };
51.     }
52. }
53. void push ()
54. {
55.     int val;
56.     struct node *ptr = (struct node*)malloc(sizeof(struct node));
57.     if(ptr == NULL)

```

```

58. {
59.     printf("not able to push the element");
60. }
61. else
62. {
63.     printf("Enter the value");
64.     scanf("%d",&val);
65.     if(head==NULL)
66.     {
67.         ptr->val = val;
68.         ptr -> next = NULL;
69.         head=ptr;
70.     }
71.     else
72.     {
73.         ptr->val = val;
74.         ptr->next = head;
75.         head=ptr;
76.
77.     }
78.     printf("Item pushed");
79.
80. }
81.}
82.
83. void pop()
84. {
85.     int item;
86.     struct node *ptr;
87.     if (head == NULL)
88.     {
89.         printf("Underflow");
90.     }
91.     else
92.     {
93.         item = head->val;
94.         ptr = head;
95.         head = head->next;
96.         free(ptr);
97.         printf("Item popped");
98.
99.     }
100. }
101. void display()
102. {

```

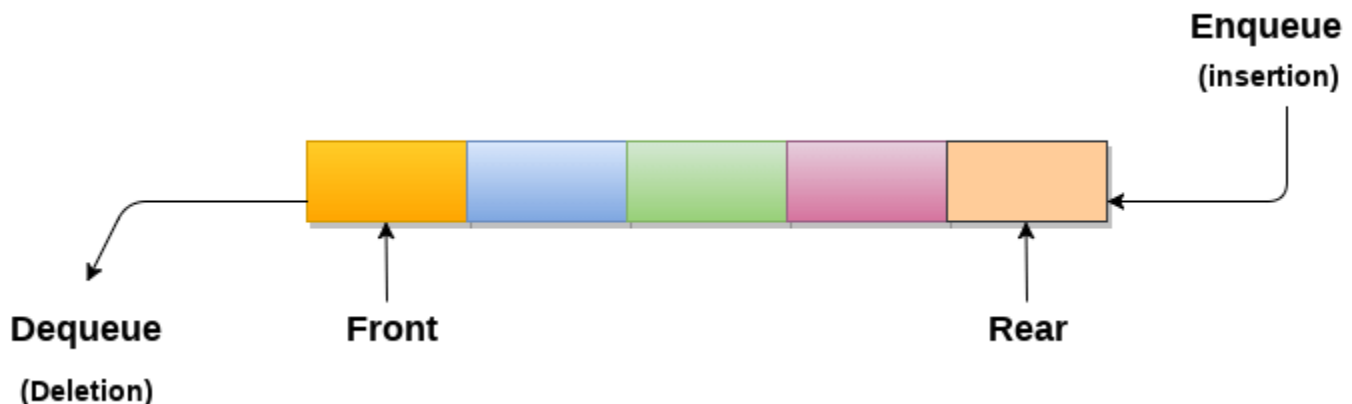
```

103.     int i;
104.     struct node *ptr;
105.     ptr=head;
106.     if(ptr == NULL)
107.     {
108.         printf("Stack is empty\n");
109.     }
110.     else
111.     {
112.         printf("Printing Stack elements \n");
113.         while(ptr!=NULL)
114.         {
115.             printf("%d\n",ptr->val);
116.             ptr = ptr->next;
117.         }
118.     } }

```

Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
2. Queue is referred to be as First In First Out list.
3. For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.

3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Types of Queues

Before understanding the types of queues, we first look at '**what is Queue**'.

What is the Queue?

A queue in the data structure can be considered similar to the queue in the real-world. A queue is a data structure in which whatever comes first will go out first. It follows the FIFO (First-In-First-Out) policy. In Queue, the insertion is done from one end known as the rear end or the tail of the queue, whereas the deletion is done from another end known as the front end or the head of the queue. In other words, it can be defined as a list or a collection with a constraint that the insertion can be performed at one end called as the rear end or tail of the queue and deletion is performed on another end called as the front end or the head of the queue.



Operations on Queue

There are two fundamental operations performed on a Queue:

- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value. The dequeue operation can also be designed to void.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** When the Queue is completely full, then it shows the overflow condition.
- **Queue underflow (isempty):** When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow condition.

A Queue can be represented as a container opened from both the sides in which the element can be enqueued from one side and dequeued from another side as shown in the below figure:

Implementation of Queue

There are two ways of implementing the Queue:

- **Sequential allocation:** The sequential allocation in a Queue can be implemented using an array.
For more details, click on the below link: <https://www.javatpoint.com/array-representation-of-queue>
- **Linked list allocation:** The linked list allocation in a Queue can be implemented using a linked list.
For more details, click on the below link: <https://www.javatpoint.com/linked-list-implementation-of-queue>

What are the use cases of Queue?

Here, we will see the real-world scenarios where we can use the Queue data structure. The Queue data structure is mainly used where there is a shared resource that has to serve the multiple requests but can serve a single request at a time. In such cases, we need to use the Queue data structure for queuing up the requests. The request that arrives first in the queue will be served first. The following are the real-world scenarios in which the Queue concept is used:

- Suppose we have a printer shared between various machines in a network, and any machine or computer in a network can send a print request to the printer. But, the printer can serve a single request at a time, i.e., a printer can print a single document at a time. When any print request comes from the network, and if the printer is busy, the printer's program will put the print request in a queue.
- . If the requests are available in the Queue, the printer takes a request from the front of the queue, and serves it.
- The processor in a computer is also used as a shared resource. There are multiple requests that the processor must execute, but the processor can serve a single request or execute a single process at a time. Therefore, the processes are kept in a Queue for execution.

Types of Queue

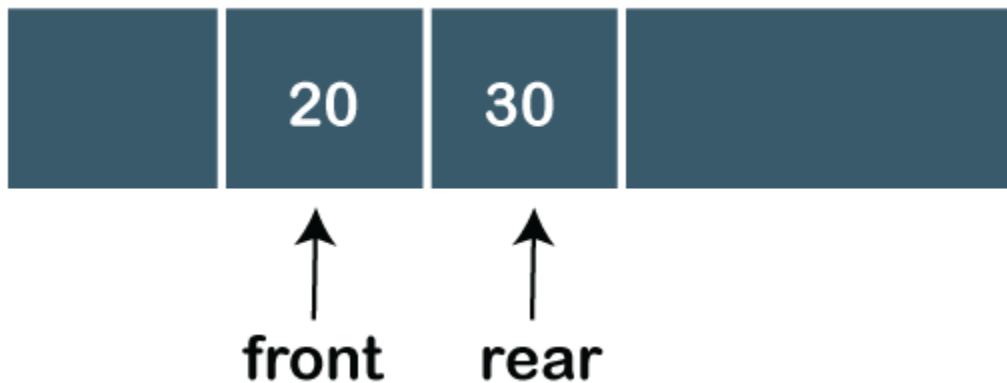
There are four types of Queues:

- **Linear Queue**

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule. The linear Queue can be represented, as shown in the below figure:



The above figure shows that the elements are inserted from the rear end, and if we insert more elements in a Queue, then the rear value gets incremented on every insertion. If we want to show the deletion, then it can be represented as:

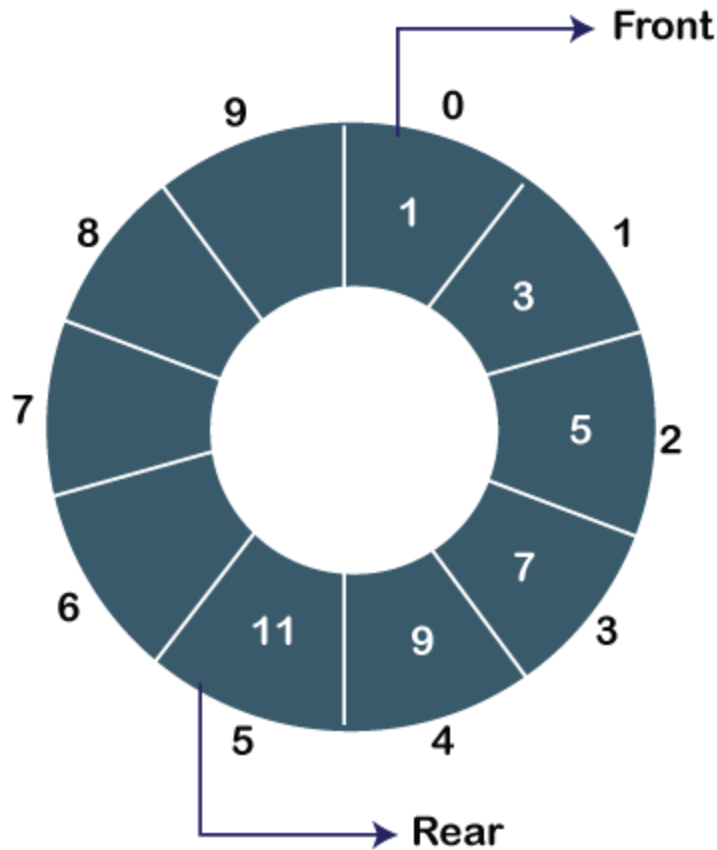


In the above figure, we can observe that the front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.

The major drawback of using a **linear Queue** is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the **overflow** condition as the rear is pointing to the last element of the Queue.

- **Circular Queue**

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as **Ring Buffer** as all the ends are connected to another end. The circular queue can be represented as:



The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

To know more about circular queue, click on the below link: <https://www.javatpoint.com/circular-queue>

- **Priority Queue**

A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.

In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority. The priority Queue can be shown as:

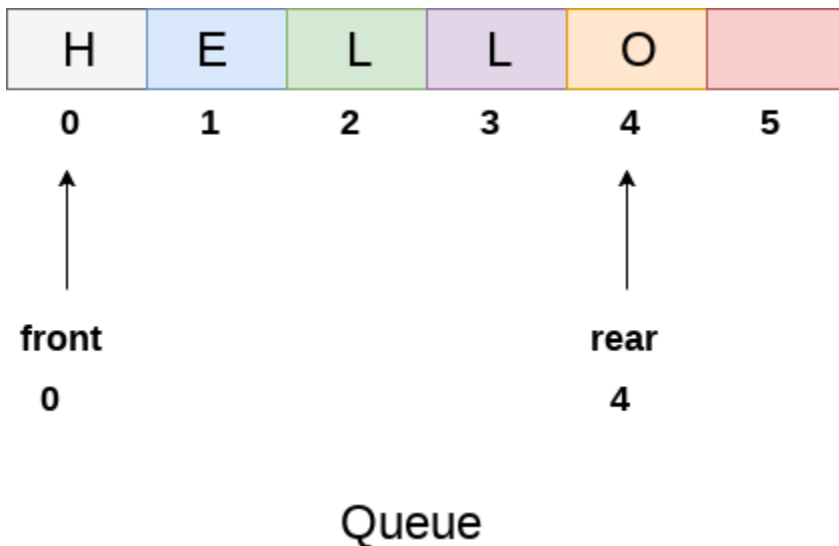
The above figure shows that the highest priority element comes first and the elements of the same priority are arranged based on FIFO structure.

- **Deque**

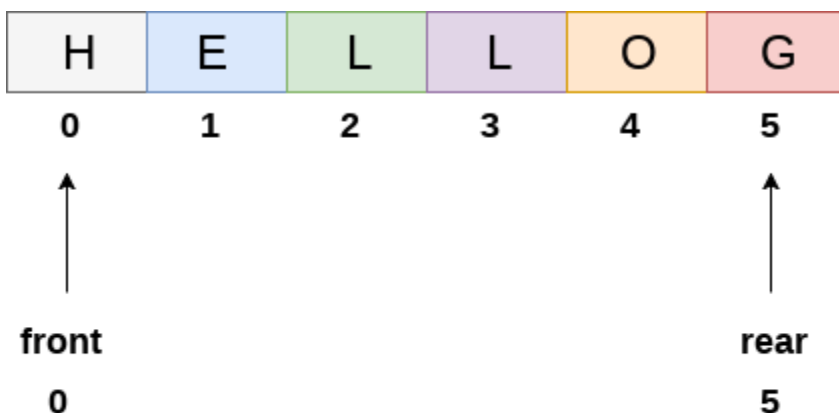
Both the Linear Queue and Deque are different as the linear queue follows the FIFO principle whereas, deque does not follow the FIFO principle. In Deque, the insertion and deletion can occur from both ends.

Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.

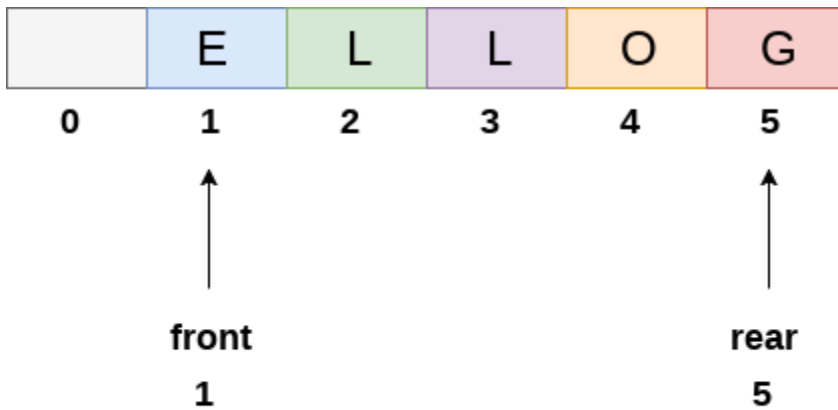


The above figure shows the queue of characters forming the English word "**HELLO**". Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

- **Step 1:** IF REAR = MAX - 1
 Write OVERFLOW
 Go to step
 [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
 SET FRONT = REAR = 0
 ELSE
 SET REAR = REAR + 1
 [END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

C Function

```

1. void insert (int queue[], int max, int front, int rear, int item)
2. {
3.     if (rear + 1 == max)
4.     {
5.         printf("overflow");

```

```

6.  }
7.  else
8.  {
9.      if(front == -1 && rear == -1)
10.     {
11.         front = 0;
12.         rear = 0;
13.     }
14.     else
15.     {
16.         rear = rear + 1;
17.     }
18.     queue[rear]=item;
19. }
20.}

```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

- **Step 1:** IF FRONT = -1 or FRONT > REAR
Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]
SET FRONT = FRONT + 1
[END OF IF]
- **Step 2:** EXIT

C Function

```

1. int delete (int queue[], int max, int front, int rear)
2. {
3.     int y;
4.     if (front == -1 || front > rear)
5.
6.     {
7.         printf("underflow");
8.     }
9.     else
10.    {
11.        y = queue[front];

```

```

12.     if(front == rear)
13.     {
14.         front = rear = -1;
15.     else
16.         front = front + 1;
17.
18.     }
19.     return y;
20. }
21.}

```

Menu driven program to implement queue using array

```

1. #include<stdio.h>
2. #include<stdlib.h>
3. #define maxsize 5
4. void insert();
5. void delete();
6. void display();
7. int front = -1, rear = -1;
8. int queue[maxsize];
9. void main ()
10. {
11.     int choice;
12.     while(choice != 4)
13.     {
14.         printf("\n*****Main Menu*****\n");
15.         printf("\n=====
=====
1. insert an element\n2. Delete an element\n3. Display the queue\n4. Exit\n")
;
16.         printf("\nEnter your choice ?");
17.         scanf("%d",&choice);
18.         switch(choice)
19.         {
20.             case 1:
21.                 insert();
22.                 break;
23.             case 2:
24.                 delete();
25.                 break;
26.             case 3:
27.                 display();
28.                 break;
29.

```

```
30.     case 4:
31.         exit(0);
32.         break;
33.     default:
34.         printf("\nEnter valid choice??\n");
35.     }
36. }
37.}
38. void insert()
39. {
40.     int item;
41.     printf("\nEnter the element\n");
42.     scanf("\n%d",&item);
43.     if(rear == maxsize-1)
44.     {
45.         printf("\nOVERFLOW\n");
46.         return;
47.     }
48.     if(front == -1 && rear == -1)
49.     {
50.         front = 0;
51.         rear = 0;
52.     }
53.     else
54.     {
55.         rear = rear+1;
56.     }
57.     queue[rear] = item;
58.     printf("\nValue inserted ");
59.
60. }
61. void delete()
62. {
63.     int item;
64.     if (front == -1 || front > rear)
65.     {
66.         printf("\nUNDERFLOW\n");
67.         return;
68.
69.     }
70.     else
71.     {
72.         item = queue[front];
73.         if(front == rear)
74.         {
```

```

75.         front = -1;
76.         rear = -1 ;
77.     }
78.     else
79.     {
80.         front = front + 1;
81.     }
82.     printf("\nvalue deleted ");
83. }
84.
85.
86.}
87.
88.void display()
89.{
90.    int i;
91.    if(rear == -1)
92.    {
93.        printf("\nEmpty queue\n");
94.    }
95.    else
96.    { printf("\nprinting values ..... \n");
97.        for(i=front;i<=rear;i++)
98.        {
99.            printf("\n%d\n",queue[i]);
100.        }
101.    }
102.    }

```

Output:

```

*****Main Menu*****

```

```

=====

```

```

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

```

```

Enter your choice ?1

```

```

Enter the element
123

```

```

Value inserted

```

```

*****Main Menu*****

```

```

=====

```

```

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
90

Value inserted

*****Main Menu*****

=====

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2

value deleted

*****Main Menu*****

=====

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

90

*****Main Menu*****

=====

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

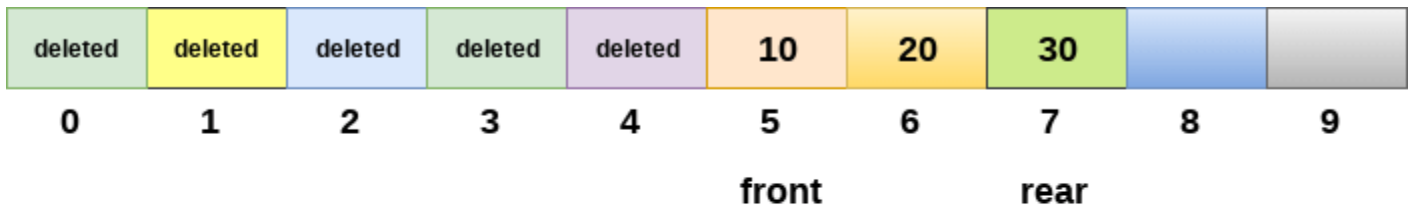
Enter your choice ?4

```

Drawback of array implementation

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



limitation of array representation of queue

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

- **Deciding the array size**

One of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

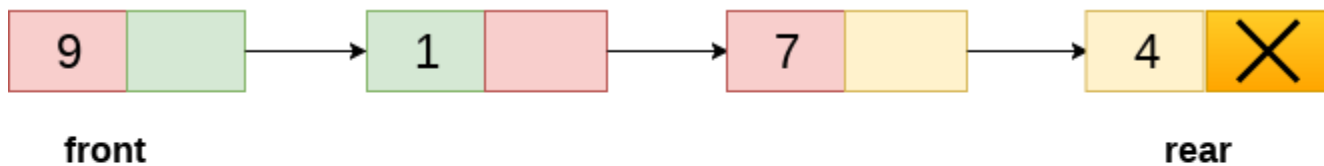
The storage requirement of linked representation of a queue with n elements is $O(n)$ while the time requirement for operations is $O(1)$.

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

1. `Ptr = (struct node *) malloc (sizeof(struct node));`

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

1. `ptr -> data = item;`
2. `if(front == NULL)`
3. `{`
4. `front = ptr;`
5. `rear = ptr;`
6. `front -> next = NULL;`
7. `rear -> next = NULL;`
8. `}`

In the second case, the queue contains more than one element. The condition `front = NULL` becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

1. `rear -> next = ptr;`
2. `rear = ptr;`
3. `rear->next = NULL;`

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

- **Step 1:** Allocate the space for the new node PTR
- **Step 2:** SET PTR -> DATA = VAL
- **Step 3:** IF FRONT = NULL
SET FRONT = REAR = PTR
SET FRONT -> NEXT = REAR -> NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR
SET REAR -> NEXT = NULL
[END OF IF]
- **Step 4:** END

C Function

```
1. void insert(struct node *ptr, int item; )
2. {
3.
4.
5.     ptr = (struct node *) malloc (sizeof(struct node));
6.     if(ptr == NULL)
7.     {
8.         printf("\nOVERFLOW\n");
9.         return;
10.    }
11.    else
12.    {
13.        ptr -> data = item;
14.        if(front == NULL)
15.        {
16.            front = ptr;
17.            rear = ptr;
18.            front -> next = NULL;
19.            rear -> next = NULL;
20.        }
21.        else
22.        {
23.            rear -> next = ptr;
24.            rear = ptr;
25.            rear->next = NULL;
26.        }
```

```
27.  }
28. }
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition `front == NULL` becomes true if the list is empty, in this case, we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer `front`. For this purpose, copy the node pointed by the `front` pointer into the pointer `ptr`. Now, shift the `front` pointer, point to its next node and free the node pointed by the node `ptr`. This is done by using the following statements.

1. `ptr = front;`
2. `front = front -> next;`
3. `free(ptr);`

The algorithm and C function is given as follows.

Algorithm

- **Step 1:** IF `FRONT == NULL`
Write " Underflow "
Go to Step 5
[END OF IF]
- **Step 2:** SET `PTR = FRONT`
- **Step 3:** SET `FRONT = FRONT -> NEXT`
- **Step 4:** FREE `PTR`
- **Step 5:** END

C Function

```
1. void delete (struct node *ptr)
2. {
3.     if(front == NULL)
4.     {
5.         printf("\nUNDERFLOW\n");
6.         return;
7.     }
8.     else
9.     {
10.        ptr = front;
11.        front = front -> next;
12.        free(ptr);
13.    }
14. }
```

Menu-Driven Program implementing all the operations on Linked Queue

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     int data;
6.     struct node *next;
7. };
8. struct node *front;
9. struct node *rear;
10. void insert();
11. void delete();
12. void display();
13. void main ()
14. {
15.     int choice;
16.     while(choice != 4)
17.     {
18.         printf("\n*****Main Menu*****\n");
19.         printf("\n=====
=====\\n");
20.         printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\\n");
21.         printf("\nEnter your choice ?");
22.         scanf("%d",& choice);
23.         switch(choice)
24.         {
25.             case 1:
26.                 insert();
27.                 break;
28.             case 2:
29.                 delete();
30.                 break;
31.             case 3:
32.                 display();
33.                 break;
34.             case 4:
35.                 exit(0);
36.                 break;
37.             default:
38.                 printf("\nEnter valid choice??\\n");
39.         }
40.     }
41. }
```

```
42. void insert()
43. {
44.     struct node *ptr;
45.     int item;
46.
47.     ptr = (struct node *) malloc (sizeof(struct node));
48.     if(ptr == NULL)
49.     {
50.         printf("\nOVERFLOW\n");
51.         return;
52.     }
53.     else
54.     {
55.         printf("\nEnter value?\n");
56.         scanf("%d",&item);
57.         ptr -> data = item;
58.         if(front == NULL)
59.         {
60.             front = ptr;
61.             rear = ptr;
62.             front -> next = NULL;
63.             rear -> next = NULL;
64.         }
65.         else
66.         {
67.             rear -> next = ptr;
68.             rear = ptr;
69.             rear->next = NULL;
70.         }
71.     }
72. }
73. void delete ()
74. {
75.     struct node *ptr;
76.     if(front == NULL)
77.     {
78.         printf("\nUNDERFLOW\n");
79.         return;
80.     }
81.     else
82.     {
83.         ptr = front;
84.         front = front -> next;
85.         free(ptr);
86.     }
```

```

87.}
88.void display()
89.{
90.    struct node *ptr;
91.    ptr = front;
92.    if(front == NULL)
93.    {
94.        printf("\nEmpty queue\n");
95.    }
96.    else
97.    { printf("\nprinting values ..... \n");
98.        while(ptr != NULL)
99.        {
100.            printf("\n%d\n",ptr -> data);
101.            ptr = ptr -> next;
102.        }
103.    }
104.    }

```

Output:

```

*****Main Menu*****
=====

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
123

*****Main Menu*****
=====

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
90

*****Main Menu*****
=====

1.insert an element
2.Delete an element

```

```

3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

123

90

*****Main Menu*****

=====

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2

*****Main Menu*****

=====

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

90

*****Main Menu*****

=====

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

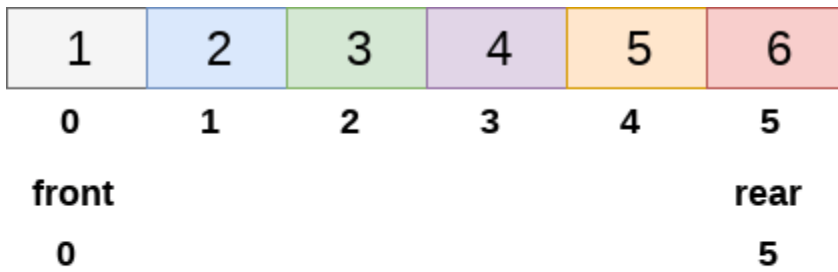
Enter your choice ?4

```

Circular Queue

Deletions and insertions can only be performed at front and rear end respectively, as far as linear queue is concerned.

Consider the queue shown in the following figure.

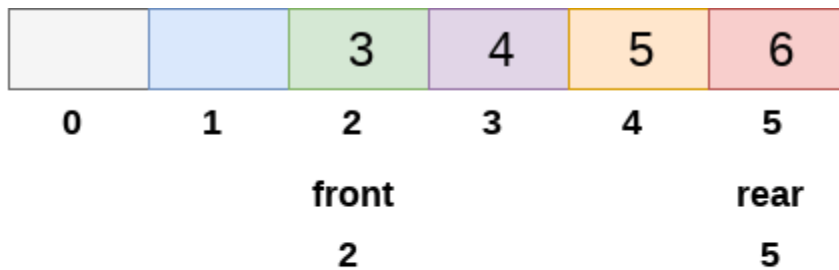


Queue

The Queue shown in above figure is completely filled and there can't be inserted any more element due to the condition **rear == max - 1 becomes true**.

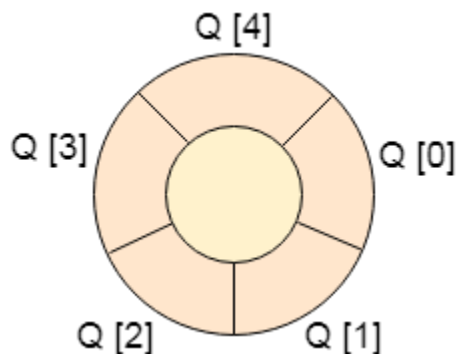
However, if we delete 2 elements at the front end of the queue, we still can not insert any element since the condition **rear = max -1 still holds**.

This is the main problem with the linear queue, although we have space available in the array, but we can not insert any more element in the queue. This is simply the memory wastage and we need to overcome this problem.



Queue after deletion of first 2 elements

One of the solution of this problem is circular queue. In the circular queue, the first index comes right after the last index. You can think of a circular queue as shown in the following figure.



Circular queue will be full when **front = -1** and **rear = max-1**. Implementation of circular queue is similar to that of a linear queue. Only the logic part that is implemented in the case of insertion and deletion is different from that in a linear queue.

Complexity

Time Complexity

Front	O(1)
Rear	O(1)
enQueue()	O(1)
deQueue()	O(1)

Insertion in Circular queue

There are three scenario of inserting an element in a queue.

1. **If $(\text{rear} + 1) \% \text{maxsize} = \text{front}$** , the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.
2. **If $\text{rear} \neq \text{max} - 1$** , then rear will be incremented to the **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
3. **If $\text{front} \neq 0$ and $\text{rear} = \text{max} - 1$** , then it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.

Algorithm to insert an element in circular queue

- **Step 1:** IF $(\text{REAR} + 1) \% \text{MAX} = \text{FRONT}$
Write " OVERFLOW "
Goto step 4
[End OF IF]
- **Step 2:** IF $\text{FRONT} = -1$ and $\text{REAR} = -1$
SET $\text{FRONT} = \text{REAR} = 0$
ELSE IF $\text{REAR} = \text{MAX} - 1$ and $\text{FRONT} \neq 0$
SET $\text{REAR} = 0$
ELSE
SET $\text{REAR} = (\text{REAR} + 1) \% \text{MAX}$
[END OF IF]
- **Step 3:** SET $\text{QUEUE}[\text{REAR}] = \text{VAL}$
- **Step 4:** EXIT

C Function

```

1. void insert(int item, int queue[])
2. {
3.     if((rear+1)%maxsize == front)
4.     {
5.         printf("\nOVERFLOW");
6.         return;
7.     }
8.     else if(front == -1 && rear == -1)
9.     {
10.        front = 0;
11.        rear = 0;
12.    }
13.    else if(rear == maxsize -1 && front != 0)
14.    {
15.        rear = 0;
16.    }
17.    else
18.    {
19.        rear = (rear+1)%maxsize;
20.    }
21.    queue[rear] = item;
22.}

```

Algorithm to delete an element from a circular queue

To delete an element from the circular queue, we must check for the three following conditions.

1. If front = -1, then there are no elements in the queue and therefore this will be the case of an underflow condition.
2. If there is only one element in the queue, in this case, the condition rear = front holds and therefore, both are set to -1 and the queue is deleted completely.
3. If front = max -1 then, the value is deleted from the front end the value of front is set to 0.
4. Otherwise, the value of front is incremented by 1 and then delete the element at the front end.

Algorithm

- **Step 1:** IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]
- **Step 2:** SET VAL = QUEUE[FRONT]
- **Step 3:** IF FRONT = REAR
SET FRONT = REAR = -1
ELSE

```
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]
```

- **Step 4:** EXIT

C Function

```
1. void delete()
2. {
3.     if(front == -1 & rear == -1)
4.     {
5.         printf("\nUNDERFLOW\n");
6.         return;
7.     }
8.     else if(front == rear)
9.     {
10.        front = -1;
11.        rear = -1;
12.    }
13.    else if(front == maxsize -1)
14.    {
15.        front = 0;
16.    }
17.    else
18.        front = front + 1;
19.}
```

Menu-Driven program implementing all the operations on a circular queue

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. #define maxsize 5
4. void insert();
5. void delete();
6. void display();
7. int front = -1, rear = -1;
8. int queue[maxsize];
9. void main ()
10.{
11.    int choice;
```

```

12.  while(choice != 4)
13.  {
14.      printf("\n*****Main Menu*****\n");
15.      printf("\n=====
=====\\n");
16.      printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\\n")
;
17.      printf("\nEnter your choice ?");
18.      scanf("%d",&choice);
19.      switch(choice)
20.      {
21.          case 1:
22.              insert();
23.              break;
24.          case 2:
25.              delete();
26.              break;
27.          case 3:
28.              display();
29.              break;
30.          case 4:
31.              exit(0);
32.              break;
33.          default:
34.              printf("\nEnter valid choice??\\n");
35.      }
36.  }
37. }
38. void insert()
39. {
40.     int item;
41.     printf("\nEnter the element\\n");
42.     scanf("%d",&item);
43.     if((rear+1)%maxsize == front)
44.     {
45.         printf("\nOVERFLOW");
46.         return;
47.     }
48.     else if(front == -1 && rear == -1)
49.     {
50.         front = 0;
51.         rear = 0;
52.     }
53.     else if(rear == maxsize -1 && front != 0)

```

```
54. {
55.     rear = 0;
56. }
57. else
58. {
59.     rear = (rear+1)%maxsize;
60. }
61. queue[rear] = item;
62. printf("\nValue inserted ");
63.}
64. void delete()
65. {
66.     int item;
67.     if(front == -1 & rear == -1)
68.     {
69.         printf("\nUNDERFLOW\n");
70.         return;
71.     }
72.     else if(front == rear)
73.     {
74.         front = -1;
75.         rear = -1;
76.     }
77.     else if(front == maxsize -1)
78.     {
79.         front = 0;
80.     }
81.     else
82.         front = front + 1;
83. }
84.
85. void display()
86. {
87.     int i;
88.     if(front == -1)
89.         printf("\nCircular Queue is Empty!!!\n");
90.     else
91.     {
92.         i = front;
93.         printf("\nCircular Queue Elements are : \n");
94.         if(front <= rear){
95.             while(i <= rear)
96.                 printf("%d %d %d\n",queue[i++],front,rear);
97.         }
98.         else{
```

```

99.     while(i <= maxsize - 1)
100.         printf("%d %d %d\n", queue[i++],front,rear);
101.         i = 0;
102.         while(i <= rear)
103.             printf("%d %d %d\n",queue[i++],front,rear);
104.         }
105.     }
106. }

```

Output:

```

*****Main Menu*****
=====
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
1

Value inserted
*****Main Menu*****
=====
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
2

Value inserted
*****Main Menu*****
=====
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
3

Value inserted
*****Main Menu*****
=====
1.insert an element
2.Delete an element

```

3.Display the queue
4.Exit

Enter your choice ?3

Circular Queue Elements are :
1
2
3

*****Main Menu*****

=====
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2

*****Main Menu*****

=====
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
4

Value inserted
*****Main Menu*****

=====
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

Circular Queue Elements are :
2
3
4

*****Main Menu*****

=====
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
1

OVERFLOW

```
*****Main Menu*****
```

```
=====
```

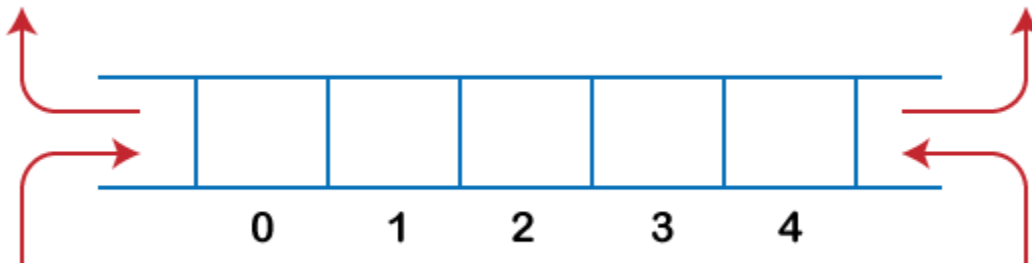
```
1.insert an element  
2.Delete an element  
3.Display the queue  
4.Exit
```

```
Enter your choice ?
```

```
4
```

Deque

The dequeue stands for **Double Ended Queue**. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.

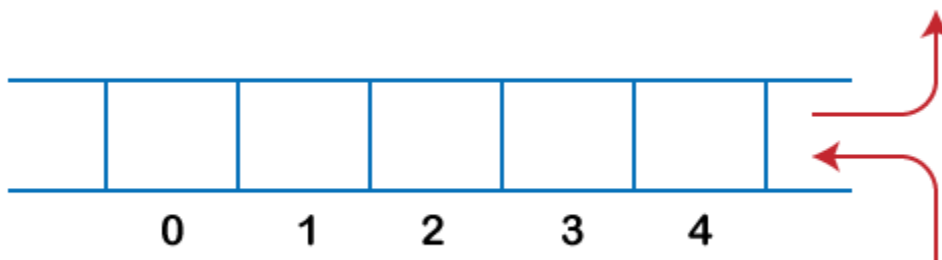


Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

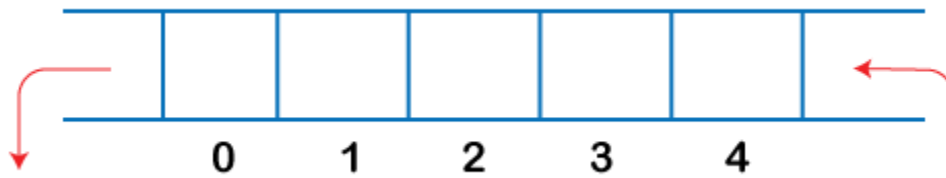
Let's look at some properties of deque.

- Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.

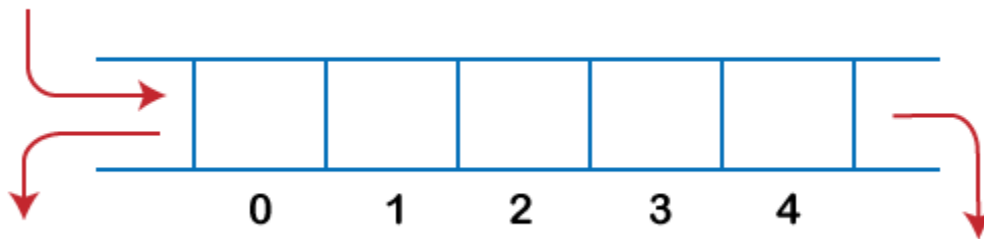


In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.

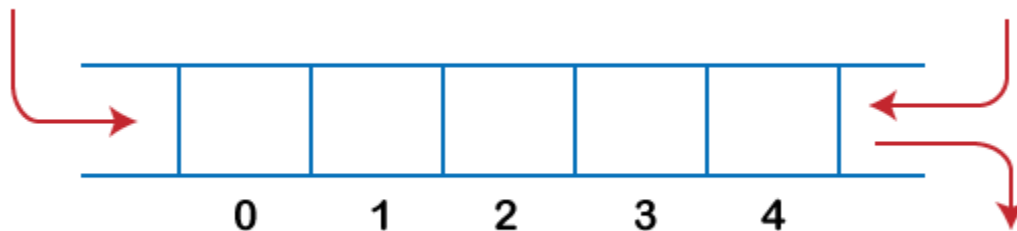


There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.



2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



Operations on Deque

The following are the operations applied on deque:

- **Insert at front**
- **Delete from end**
- **insert at rear**
- **delete from rear**

Other than insertion and deletion, we can also perform **peek** operation in deque. Through **peek** operation, we can get the **front** and the **rear** element of the dequeue.

We can perform two more operations on dequeue:

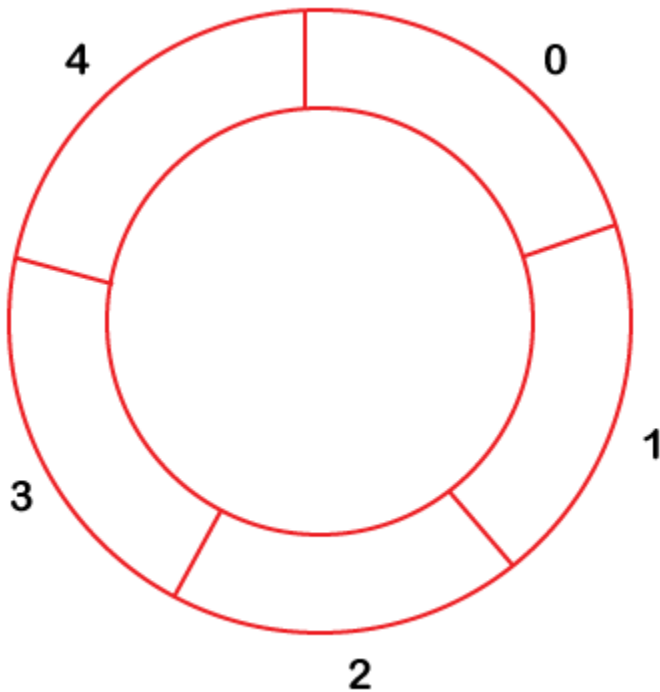
- **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.
- **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

Memory Representation

The deque can be implemented using two data structures, i.e., **circular array**, and **doubly linked list**. To implement the deque using circular array, we first should know **what is circular array**.

What is a circular array?

An array is said to be **circular** if the last element of the array is connected to the first element of the array. Suppose the size of the array is 4, and the array is full but the first location of the array is empty. If we want to insert the array element, it will not show any overflow condition as the last element is connected to the first element. The value which we want to insert will be added in the first location of the array.



Applications of Deque

- The deque can be used as a **stack** and **queue**; therefore, it can perform both redo and undo operations.
- It can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.
- It can be used for multiprocessor scheduling. Suppose we have two processors, and each processor has one process to execute. Each processor is assigned with a process or a job, and each process

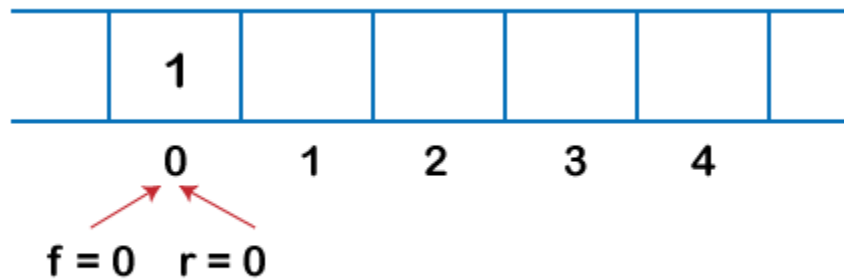
contains multiple threads. Each processor maintains a deque that contains threads that are ready to execute. The processor executes a process, and if a process creates a child process then that process will be inserted at the front of the deque of the parent process. Suppose the processor P_2 has completed the execution of all its threads then it steals the thread from the rear end of the processor P_1 and adds to the front end of the processor P_2 . The processor P_2 will take the thread from the front end; therefore, the deletion takes from both the ends, i.e., front and rear end. This is known as the **A-steal algorithm** for scheduling.

Implementation of Deque using a circular array

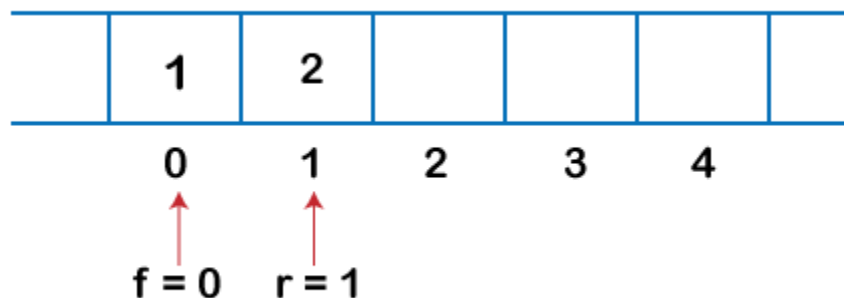
The following are the steps to perform the operations on the Deque:

Enqueue operation

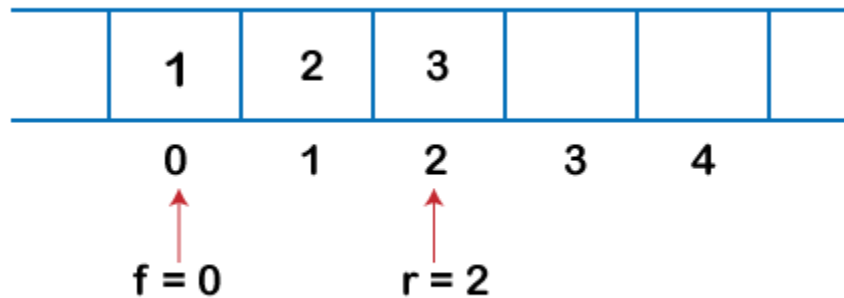
- Initially, we are considering that the deque is empty, so both front and rear are set to -1, i.e., **f = -1** and **r = -1**.
- As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then **front is equal to 0**, and the **rear is also equal to 0**.



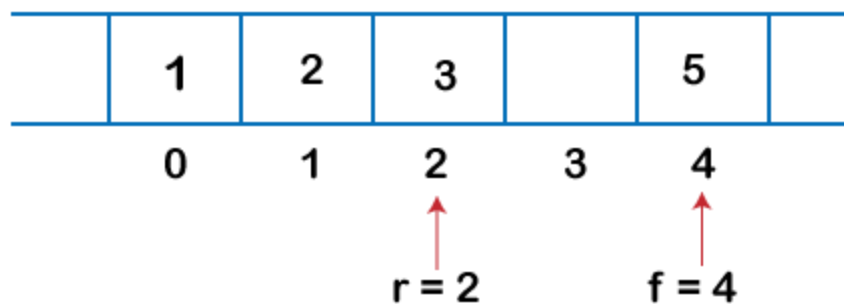
- Suppose we want to insert the next element from the rear. To insert the element from the rear end, we first need to increment the rear, i.e., **rear=rear+1**. Now, the rear is pointing to the second element, and the front is pointing to the first element.



4. Suppose we are again inserting the element from the rear end. To insert the element, we will first increment the rear, and now rear points to the third element.



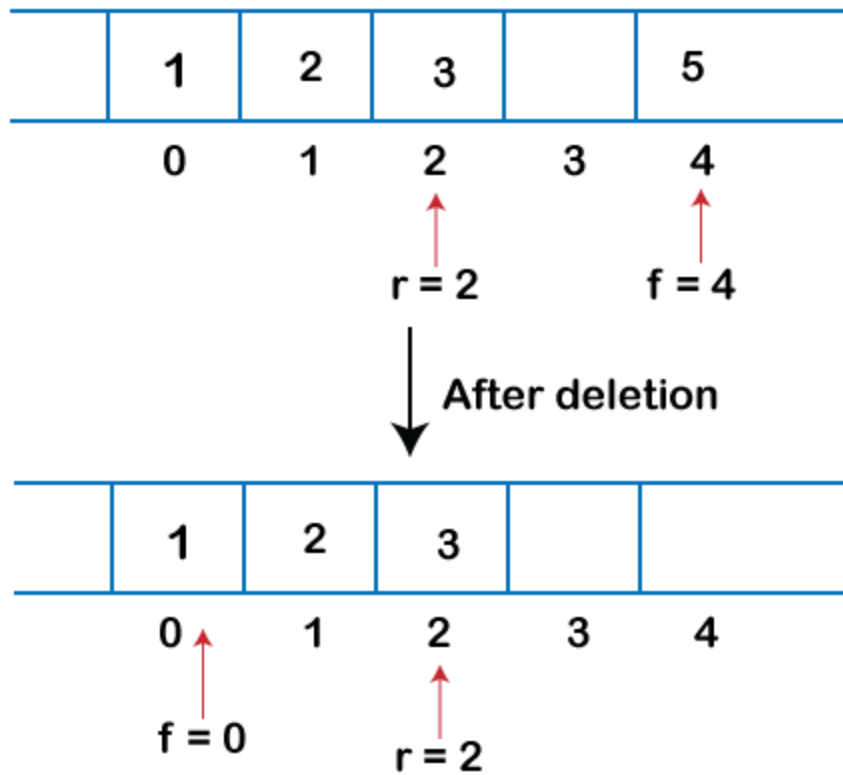
5. If we want to insert the element from the front end, and insert an element from the front, we have to decrement the value of front by 1. If we decrement the front by 1, then the front points to -1 location, which is not any valid location in an array. So, we set the front as **(n - 1)**, which is equal to 4 as n is 5. Once the front is set, we will insert the value as shown in the below figure:



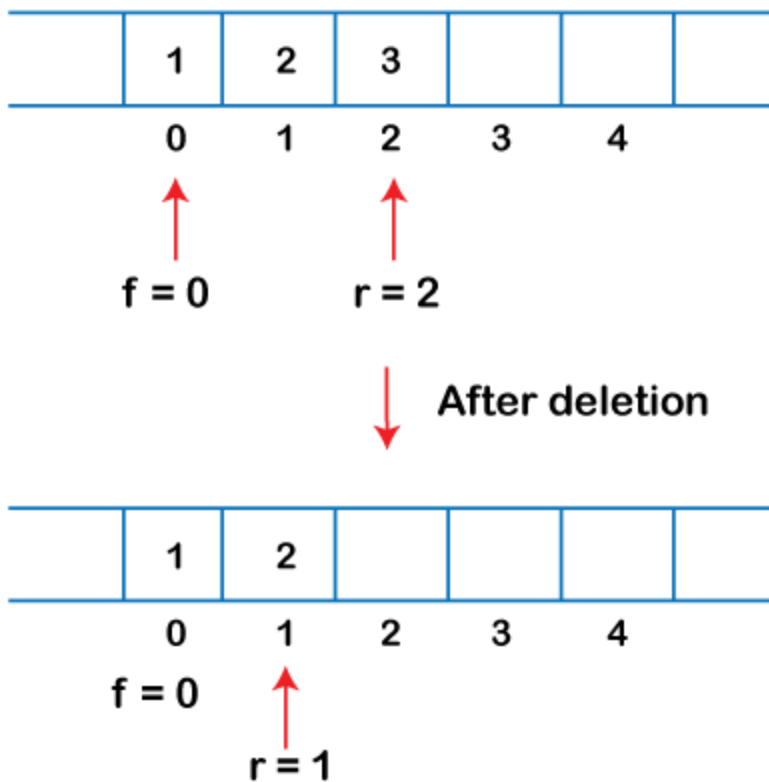
Deque Operation

1. If the front is pointing to the last element of the array, and we want to perform the delete operation from the front. To delete any element from the front, we need to set **front=front+1**. Currently, the value of the front is equal to 4, and if we increment the value of front, it becomes 5 which is not a valid index. Therefore, we conclude that if front points to the last element, then front

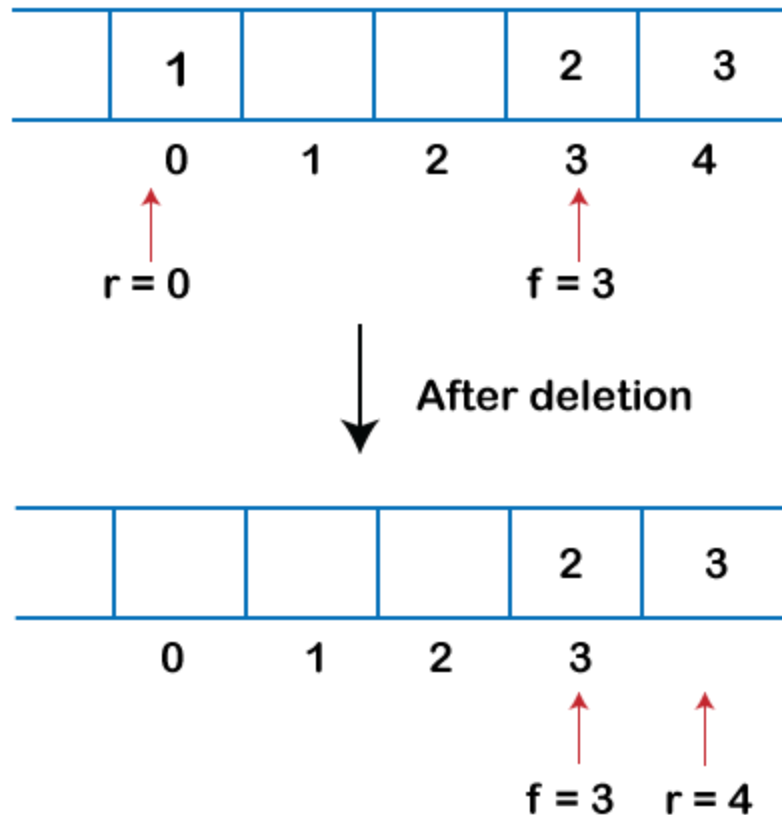
is set to 0 in case of delete operation.



2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., **rear=rear-1** as shown in the below figure:



3. If the rear is pointing to the first element, and we want to delete the element from the rear end then we have to set **rear=n-1** where **n** is the size of the array as shown in the below figure:



Let's create a program of deque.

The following are the six functions that we have used in the below program:

- **enqueue_front():** It is used to insert the element from the front end.
- **enqueue_rear():** It is used to insert the element from the rear end.
- **dequeue_front():** It is used to delete the element from the front end.
- **dequeue_rear():** It is used to delete the element from the rear end.
- **getfront():** It is used to return the front element of the deque.
- **getrear():** It is used to return the rear element of the deque.

```
1. #define size 5
2. #include <stdio.h>
3. int deque[size];
4. int f=-1, r=-1;
5. // enqueue_front function will insert the value from the front
6. void enqueue_front(int x)
7. {
8.     if((f==0 && r==size-1) || (f==r+1))
9.     {
10.         printf("deque is full");
```

```

11. }
12. else if((f== -1) && (r== -1))
13. {
14.     f=r=0;
15.     deque[f]=x;
16. }
17. else if(f==0)
18. {
19.     f=size-1;
20.     deque[f]=x;
21. }
22. else
23. {
24.     f=f-1;
25.     deque[f]=x;
26. }
27.}
28.
29.// enqueue_rear function will insert the value from the rear
30.void enqueue_rear(int x)
31.{
32.    if((f==0 && r==size-1) || (f==r+1))
33.    {
34.        printf("deque is full");
35.    }
36.    else if((f== -1) && (r== -1))
37.    {
38.        r=0;
39.        deque[r]=x;
40.    }
41.    else if(r==size-1)
42.    {
43.        r=0;
44.        deque[r]=x;
45.    }
46.    else
47.    {
48.        r++;
49.        deque[r]=x;
50.    }
51.
52.}
53.
54.// display function prints all the value of deque.
55.void display()

```

```

56. {
57.     int i=f;
58.     printf("\n Elements in a deque : ");
59.
60.     while(i!=r)
61.     {
62.         printf("%d ",deque[i]);
63.         i=(i+1)%size;
64.     }
65.     printf("%d",deque[r]);
66. }
67.
68. // getfront function retrieves the first value of the deque.
69. void getfront()
70. {
71.     if((f==-1) && (r==-1))
72.     {
73.         printf("Deque is empty");
74.     }
75.     else
76.     {
77.         printf("\nThe value of the front is: %d", deque[f]);
78.     }
79.
80. }
81.
82. // getrear function retrieves the last value of the deque.
83. void getrear()
84. {
85.     if((f==-1) && (r==-1))
86.     {
87.         printf("Deque is empty");
88.     }
89.     else
90.     {
91.         printf("\nThe value of the rear is: %d", deque[r]);
92.     }
93.
94. }
95.
96. // dequeue_front() function deletes the element from the front
97. void dequeue_front()
98. {
99.     if((f==-1) && (r==-1))
100.    {

```



```

101.     printf("Deque is empty");
102. }
103. else if(f==r)
104. {
105.     printf("\nThe deleted element is %d", deque[f]);
106.     f=-1;
107.     r=-1;
108.
109. }
110. else if(f==(size-1))
111. {
112.     printf("\nThe deleted element is %d", deque[f]);
113.     f=0;
114. }
115. else
116. {
117.     printf("\nThe deleted element is %d", deque[f]);
118.     f=f+1;
119. }
120. }
121.
122. // dequeue_rear() function deletes the element from the rear
123. void dequeue_rear()
124. {
125.     if((f==-1) && (r==-1))
126.     {
127.         printf("Deque is empty");
128.     }
129.     else if(f==r)
130.     {
131.         printf("\nThe deleted element is %d", deque[r]);
132.         f=-1;
133.         r=-1;
134.
135.     }
136.     else if(r==0)
137.     {
138.         printf("\nThe deleted element is %d", deque[r]);
139.         r=size-1;
140.     }
141.     else
142.     {
143.         printf("\nThe deleted element is %d", deque[r]);
144.         r=r-1;
145.     }

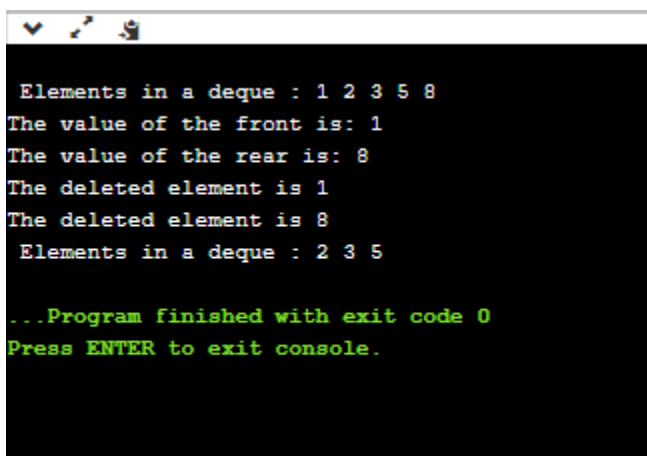
```

```

146.     }
147.
148.     int main()
149.     {
150.         // inserting a value from the front.
151.         enqueue_front(2);
152.         // inserting a value from the front.
153.         enqueue_front(1);
154.         // inserting a value from the rear.
155.         enqueue_rear(3);
156.         // inserting a value from the rear.
157.         enqueue_rear(5);
158.         // inserting a value from the rear.
159.         enqueue_rear(8);
160.         // Calling the display function to retrieve the values of deque
161.         display();
162.         // Retrieve the front value
163.         getfront();
164.         // Retrieve the rear value.
165.         getrear();
166.         // deleting a value from the front
167.         dequeue_front();
168.         //deleting a value from the rear
169.         dequeue_rear();
170.         // Calling the display function to retrieve the values of deque
171.         display();
172.         return 0;
173.     }

```

Output:



```

Elements in a deque : 1 2 3 5 8
The value of the front is: 1
The value of the rear is: 8
The deleted element is 1
The deleted element is 8
Elements in a deque : 2 3 5
...Program finished with exit code 0
Press ENTER to exit console.

```

What is a priority queue?

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

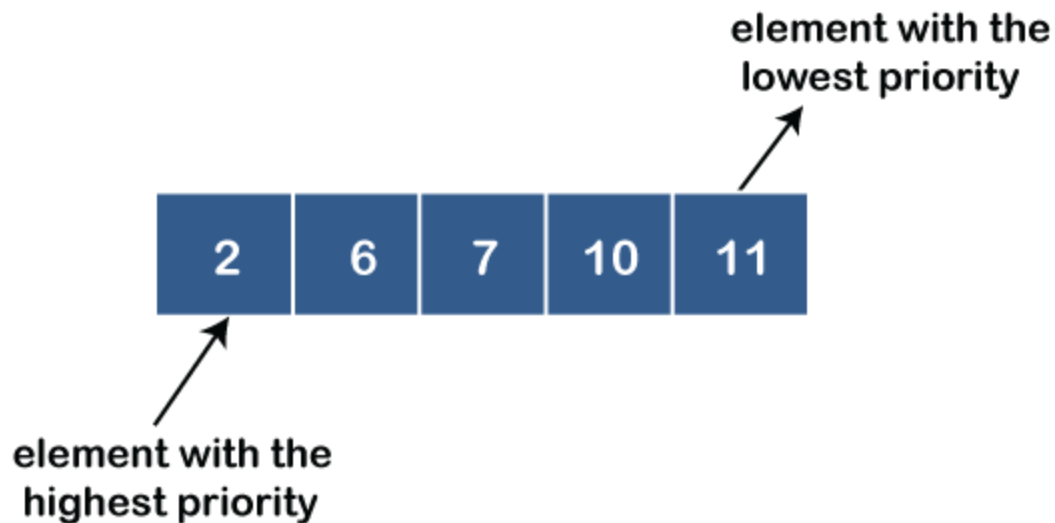
- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

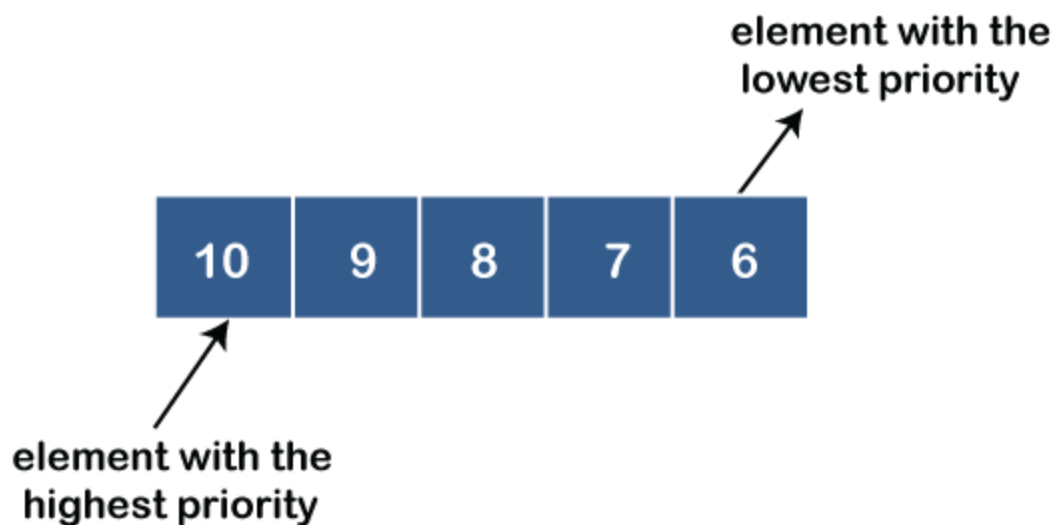
There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest

priority in a priority queue.



- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and **LINK** basically contains the address of the next node.

	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step.

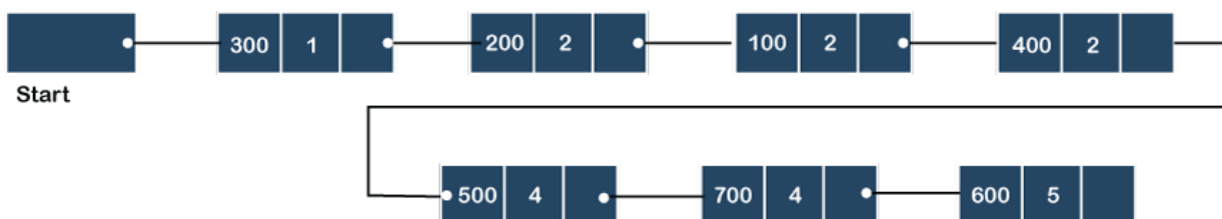
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority

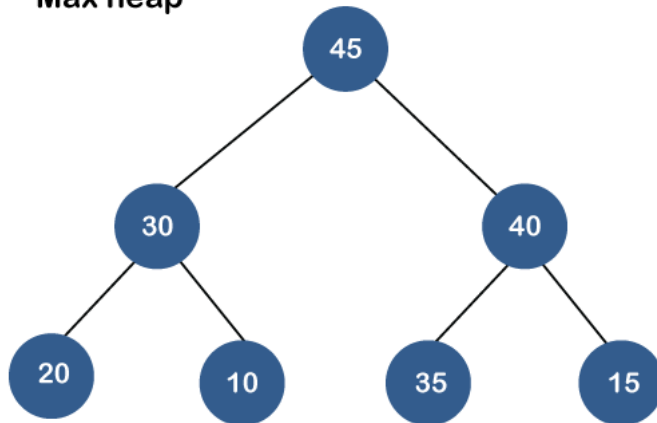
queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

What is Heap?

A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

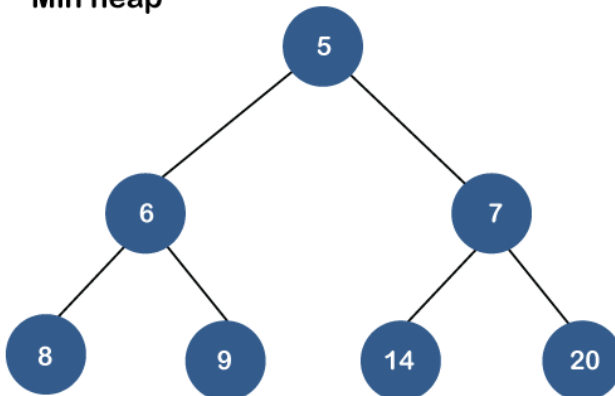
- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.

Max heap



- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.

Min heap



Both the heaps are the binary heap, as each has exactly two child nodes.

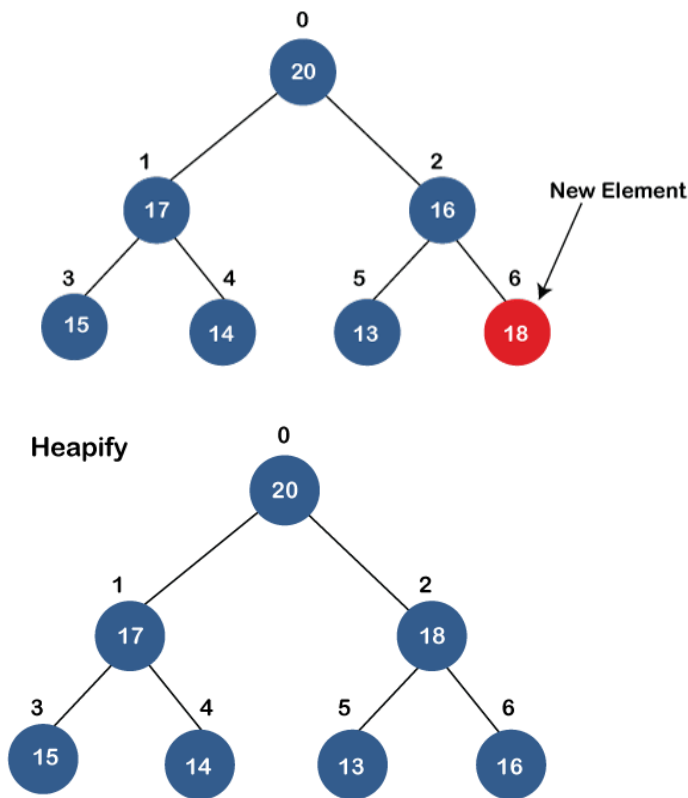
Priority Queue Operations

The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

- **Inserting the element in a priority queue (max heap)**

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.



- **Removing the minimum element from the priority queue**

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

Program to create the priority queue using the binary max heap.

```
1. #include <stdio.h>
```

```
2. #include <stdio.h>
3. int heap[40];
4. int size=-1;
5.
6. // retrieving the parent node of the child node
7. int parent(int i)
8. {
9.
10.     return (i - 1) / 2;
11. }
12.
13. // retrieving the left child of the parent node.
14. int left_child(int i)
15. {
16.     return i+1;
17. }
18. // retrieving the right child of the parent
19. int right_child(int i)
20. {
21.     return i+2;
22. }
23. // Returning the element having the highest priority
24. int get_Max()
25. {
26.     return heap[0];
27. }
28. //Returning the element having the minimum priority
29. int get_Min()
30. {
31.     return heap[size];
32. }
33. // function to move the node up the tree in order to restore the heap property.
34. void moveUp(int i)
35. {
36.     while (i > 0)
37.     {
38.         // swapping parent node with a child node
39.         if(heap[parent(i)] < heap[i]) {
40.
41.             int temp;
42.             temp=heap[parent(i)];
43.             heap[parent(i)]=heap[i];
44.             heap[i]=temp;
45.
46.
```



```

47. }
48.     // updating the value of i to i/2
49.     i=i/2;
50. }
51.}
52.
53.//function to move the node down the tree in order to restore the heap property.
54.void moveDown(int k)
55.{
56.    int index = k;
57.
58.    // getting the location of the Left Child
59.    int left = left_child(k);
60.
61.    if (left <= size && heap[left] > heap[index]) {
62.        index = left;
63.    }
64.
65.    // getting the location of the Right Child
66.    int right = right_child(k);
67.
68.    if (right <= size && heap[right] > heap[index]) {
69.        index = right;
70.    }
71.
72.    // If k is not equal to index
73.    if (k != index) {
74.        int temp;
75.        temp=heap[index];
76.        heap[index]=heap[k];
77.        heap[k]=temp;
78.        moveDown(index);
79.    }
80.}
81.
82.// Removing the element of maximum priority
83.void removeMax()
84.{
85.    int r= heap[0];
86.    heap[0]=heap[size];
87.    size=size-1;
88.    moveDown(0);
89.}
90.//inserting the element in a priority queue
91.void insert(int p)

```

```

92. {
93.     size = size + 1;
94.     heap[size] = p;
95.
96.     // move Up to maintain heap property
97.     moveUp(size);
98. }
99.
100.    //Removing the element from the priority queue at a given index i.
101.    void delete(int i)
102.    {
103.        heap[i] = heap[0] + 1;
104.
105.        // move the node stored at ith location is shifted to the root node
106.        moveUp(i);
107.
108.        // Removing the node having maximum priority
109.        removeMax();
110.    }
111.    int main()
112.    {
113.        // Inserting the elements in a priority queue
114.
115.        insert(20);
116.        insert(19);
117.        insert(21);
118.        insert(18);
119.        insert(12);
120.        insert(17);
121.        insert(15);
122.        insert(16);
123.        insert(14);
124.        int i=0;
125.
126.        printf("Elements in a priority queue are : ");
127.        for(int i=0;i<=size;i++)
128.        {
129.            printf("%d ",heap[i]);
130.        }
131.        delete(2); // deleting the element whose index is 2.
132.        printf("\nElements in a priority queue after deleting the element are : ");
133.        for(int i=0;i<=size;i++)
134.        {
135.            printf("%d ",heap[i]);
136.        }

```

```

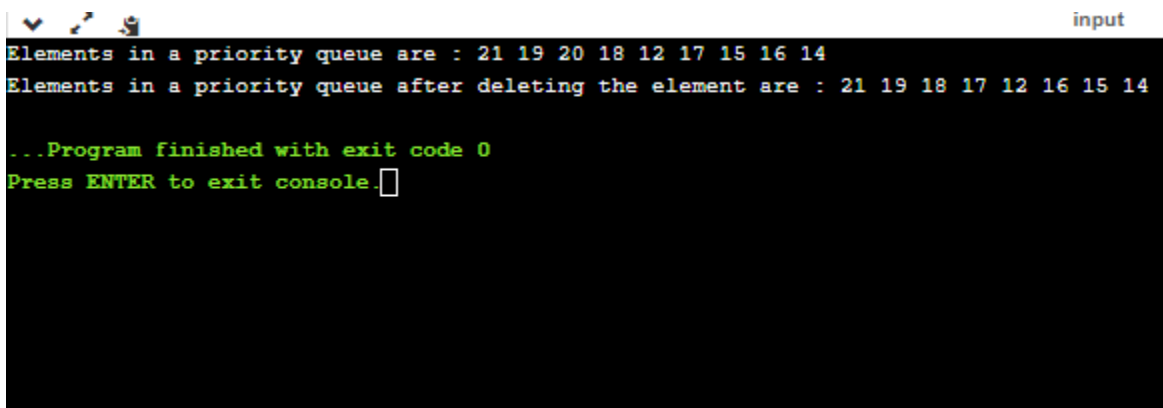
137.     int max=get_Max();
138.         printf("\nThe element which is having the highest priority is %d: ",max);
139.
140.
141.     int min=get_Min();
142.         printf("\nThe element which is having the minimum priority is : %d",min);
143.     return 0;
144. }

```

In the above program, we have created the following functions:

- **int parent(int i):** This function returns the index of the parent node of a child node, i.e., i.
- **int left_child(int i):** This function returns the index of the left child of a given index, i.e., i.
- **int right_child(int i):** This function returns the index of the right child of a given index, i.e., i.
- **void moveUp(int i):** This function will keep moving the node up the tree until the heap property is restored.
- **void moveDown(int i):** This function will keep moving the node down the tree until the heap property is restored.
- **void removeMax():** This function removes the element which is having the highest priority.
- **void insert(int p):** It inserts the element in a priority queue which is passed as an argument in a function.
- **void delete(int i):** It deletes the element from a priority queue at a given index.
- **int get_Max():** It returns the element which is having the highest priority, and we know that in max heap, the root node contains the element which has the largest value, and highest priority.
- **int get_Min():** It returns the element which is having the minimum priority, and we know that in max heap, the last node contains the element which has the smallest value, and lowest priority.

Output



```

input
Elements in a priority queue are : 21 19 20 18 12 17 15 16 14
Elements in a priority queue after deleting the element are : 21 19 18 17 12 16 15 14
...Program finished with exit code 0
Press ENTER to exit console.

```

Recursion in C

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to

impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

In the following example, recursion is used to calculate the factorial of a number.

```
1. #include <stdio.h>
2. int fact (int);
3. int main()
4. {
5.     int n,f;
6.     printf("Enter the number whose factorial you want to calculate?");
7.     scanf("%d",&n);
8.     f = fact(n);
9.     printf("factorial = %d",f);
10.}
11.int fact(int n)
12.{
13.    if (n==0)
14.    {
15.        return 0;
16.    }
17.    else if ( n == 1)
18.    {
19.        return 1;
20.    }
21.    else
22.    {
23.        return n*fact(n-1);
24.    }
25.}
```

Output

```
Enter the number whose factorial you want to calculate?5
factorial = 120
```

We can understand the above program of the recursive method call by the figure given below:

```

return 5 * factorial(4) = 120
└─ return 4 * factorial(3) = 24
    └─ return 3 * factorial(2) = 6
        └─ return 2 * factorial(1) = 2
            └─ return 1 * factorial(0) = 1

```

javaTpoint.com

$1 * 2 * 3 * 4 * 5 = 120$

Fig: Recursion

Recursive Function

A recursive function performs the tasks by dividing it into the subtasks. There is a termination condition defined in the function which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the function.

The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case. All the recursive functions can be written using this format.

Pseudocode for writing any recursive function is given below.

```

1. if (test_for_base)
2. {
3.     return some_value;
4. }
5. else if (test_for_another_base)
6. {
7.     return some_another_value;
8. }
9. else
10. {
11.     // Statements;
12.     recursive call;
13. }

```

Example of recursion in C

Let's see an example to find the nth term of the Fibonacci series.

```

1. #include<stdio.h>
2. int fibonacci(int);
3. void main ()
4. {
5.     int n,f;
6.     printf("Enter the value of n?");
7.     scanf("%d",&n);
8.     f = fibonacci(n);
9.     printf("%d",f);
10.}
11.int fibonacci (int n)
12.{
13.    if (n==0)
14.    {
15.        return 0;
16.    }
17.    else if (n == 1)
18.    {
19.        return 1;
20.    }
21.    else
22.    {
23.        return fibonacci(n-1)+fibonacci(n-2);
24.    }
25.}

```

Output

```

Enter the value of n?12
144

```

Memory allocation of Recursive method

Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

Let us consider the following example to understand the memory allocation of the recursive functions.

```

1. int display (int n)
2. {
3.     if(n == 0)
4.         return 0; // terminating condition
5.     else
6.     {
7.         printf("%d",n);

```

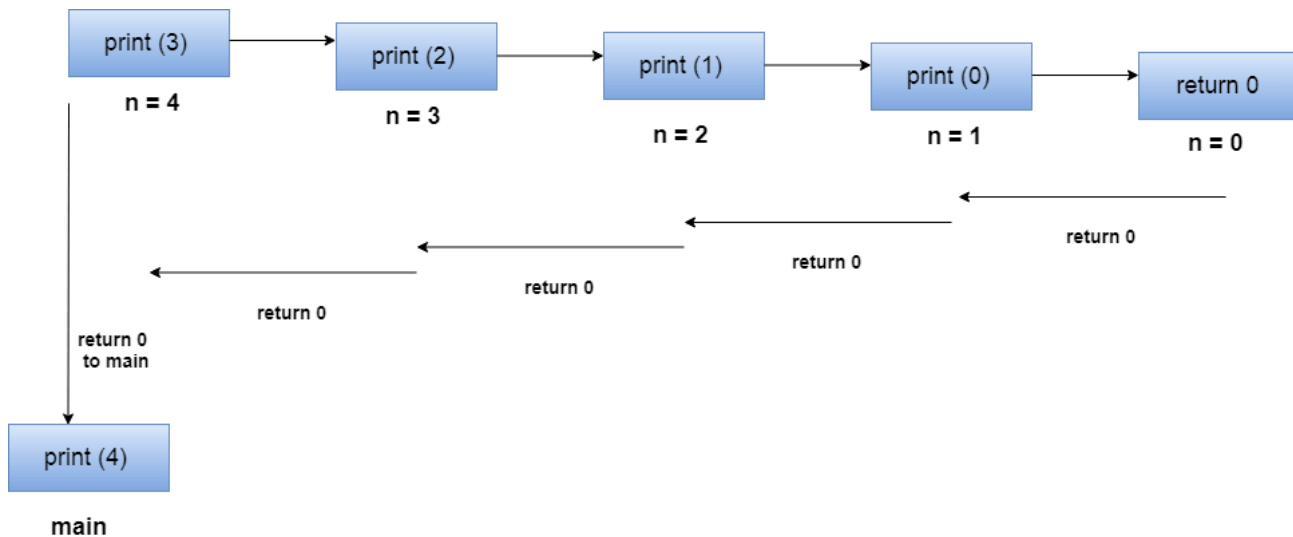
```

8.     return display(n-1); // recursive call
9.     }
10. }

```

Explanation

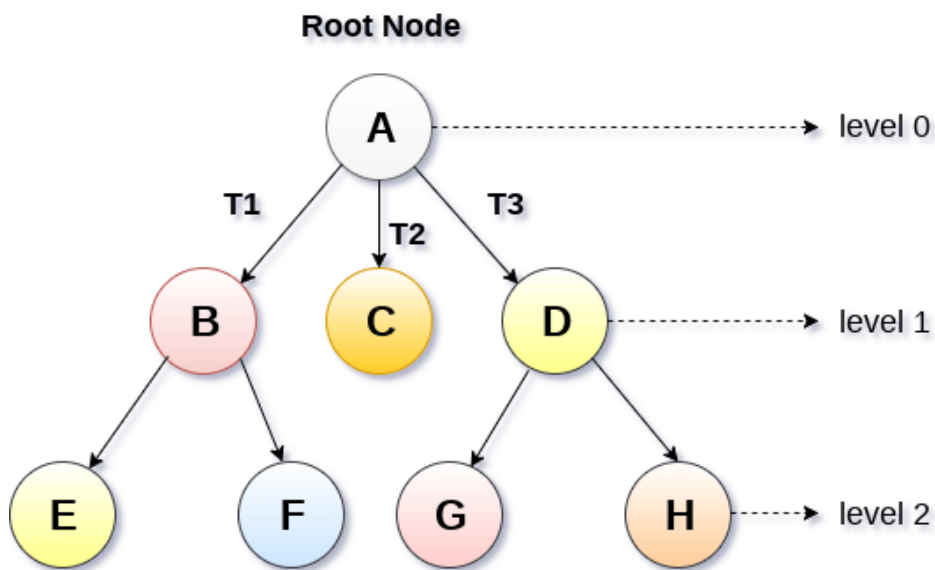
Let us examine this recursive function for $n = 4$. First, all the stacks are maintained which prints the corresponding value of n until n becomes 0, Once the termination condition is reached, the stacks get destroyed one by one by returning 0 to its calling stack. Consider the following image for more information regarding the stack trace for the recursive functions.



Stack tracing for recursive function call

Tree

- A Tree is a recursive data structure containing the set of one or more data nodes where one node is designated as the root of the tree while the remaining nodes are called as the children of the root.
- The nodes other than the root node are partitioned into the non empty sets where each one of them is to be called sub-tree.
- Nodes of a tree either maintain a parent-child relationship between them or they are sister nodes.
- In a general tree, A node can have any number of children nodes but it can have only a single parent.
- The following image shows a tree, where the node A is the root node of the tree while the other nodes can be seen as the children of A.



Tree

Basic terminology

- **Root Node** :- The root node is the topmost node in the tree hierarchy. In other words, the root node is the one which doesn't have any parent.
- **Sub Tree** :- If the root node is not null, the tree T1, T2 and T3 is called sub-trees of the root node.
- **Leaf Node** :- The node of tree, which doesn't have any child node, is called leaf node. Leaf node is the bottom most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Path** :- The sequence of consecutive edges is called path. In the tree shown in the above image, path to the node E is $A \rightarrow B \rightarrow E$.
- **Ancestor node** :- An ancestor of a node is any predecessor node on a path from root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, the node F have the ancestors, B and A.
- **Degree** :- Degree of a node is equal to number of children, a node have. In the tree shown in the above image, the degree of node B is 2. Degree of a leaf node is always 0 while in a complete binary tree, degree of each node is equal to 2.
- **Level Number** :- Each node of the tree is assigned a level number in such a way that each node is present at one level higher than its parent. Root node of the tree is always present at level 0.

Static representation of tree

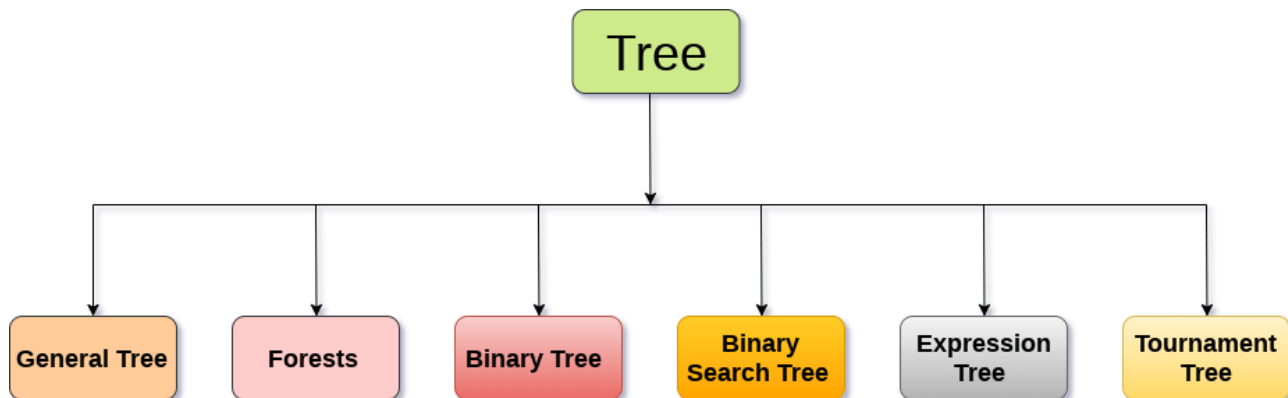

```
1. #define MAXNODE 500
2. struct treenode {
3.     int root;
4.     int father;
5.     int son;
6.     int next;
7. }
```

Dynamic representation of tree

```
1. struct treenode
2. {
3.     int root;
4.     struct treenode *father;
5.     struct treenode *son
6.     struct treenode *next;
7. }
```

Types of Tree

The tree data structure can be classified into six different categories.

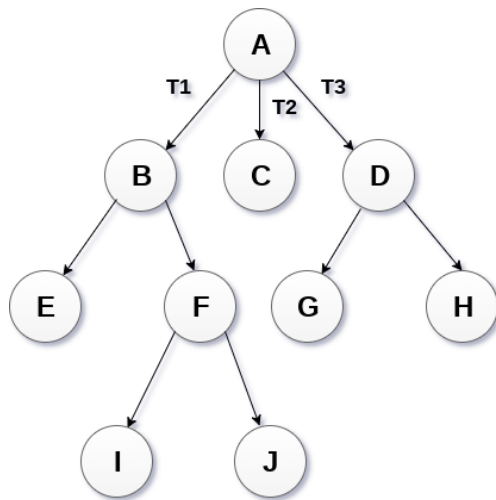


General Tree

General Tree stores the elements in a hierarchical order in which the top level element is always present at level 0 as the root element. All the nodes except the root node are present at number of levels. The nodes which are present on the same level are called siblings while the nodes which are present on the different levels exhibit the parent-child relationship among them. A node may contain any number of sub-trees. The tree in which each node contain 3 sub-tree, is called ternary tree.

Forests

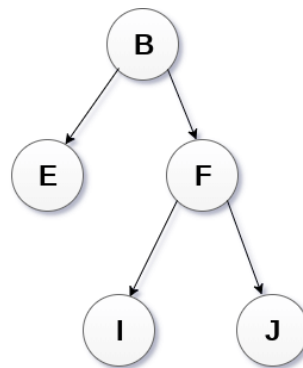
Forest can be defined as the set of disjoint trees which can be obtained by deleting the root node and the edges which connects root node to the first level node.



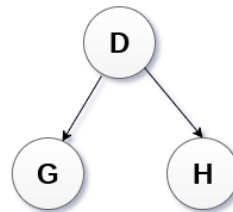
General Tree



Forest 2



Forest 1



Forest 3

Binary Tree

Binary tree is a data structure in which each node can have at most 2 children. The node present at the top most level is called the root node. A node with the 0 children is called leaf node. Binary Trees are used in the applications like expression evaluation and many more. We will discuss binary tree in detail, later in this tutorial.

Binary Search Tree

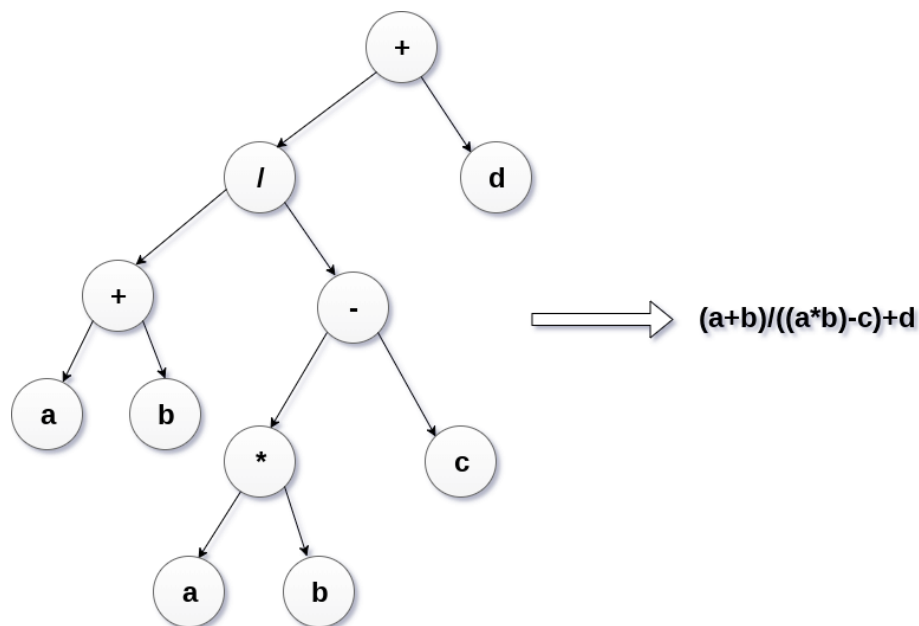
Binary search tree is an ordered binary tree. All the elements in the left sub-tree are less than the root while elements present in the right sub-tree are greater than or equal to the root node element. Binary search trees are used in most of the applications of computer science domain like searching, sorting, etc.

Expression Tree

Expression trees are used to evaluate the simple arithmetic expressions. Expression tree is basically a binary tree where internal nodes are represented by operators while the leaf nodes are represented by operands. Expression trees are widely used to solve algebraic expressions like $(a+b)*(a-b)$. Consider the following example.

Q. Construct an expression tree by using the following algebraic expression.

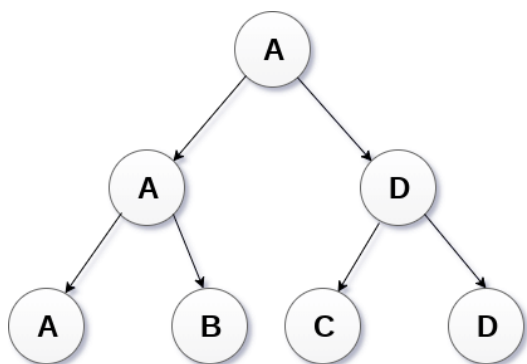
$$(a + b) / (a * b - c) + d$$



Tournament Tree

Tournament tree are used to record the winner of the match in each round being played between two players. Tournament tree can also be called as selection tree or winner tree. External nodes represent the players among which a match is being played while the internal nodes represent the winner of the match played. At the top most level, the winner of the tournament is present as the root node of the tree.

For example, tree .of a chess tournament being played among 4 players is shown as follows. However, the winner in the left sub-tree will play against the winner of right sub-tree.

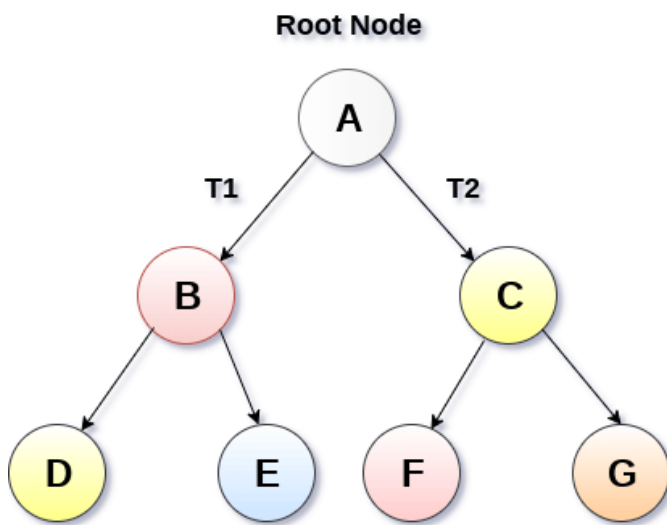


Binary Tree

Binary Tree is a special type of generic tree in which, each node can have at most two children. Binary tree is generally partitioned into three disjoint subsets.

1. Root of the node
2. left sub-tree which is also a binary tree.
3. Right binary sub-tree

A binary Tree is shown in the following image.



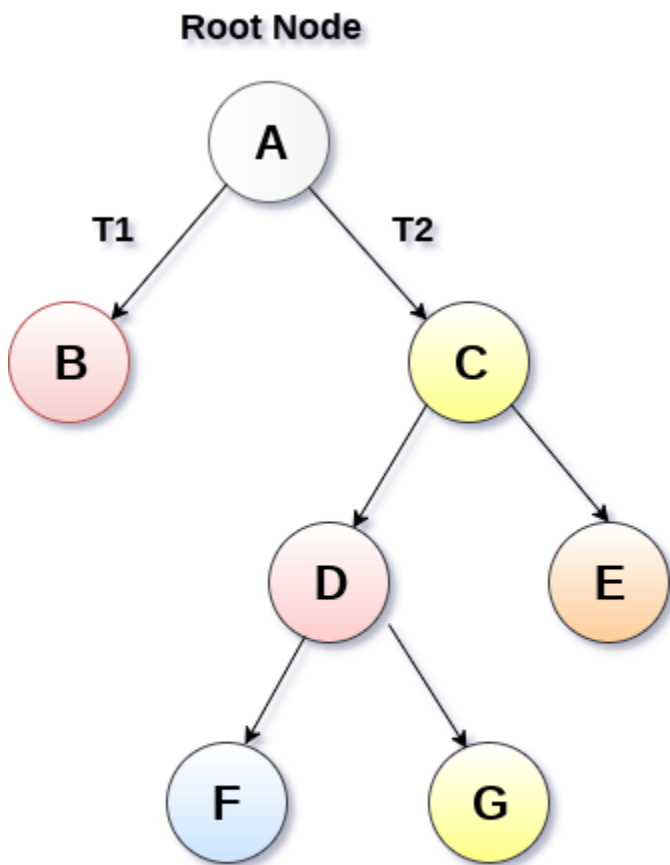
Binary Tree

Types of Binary Tree

1. Strictly Binary Tree

In Strictly Binary Tree, every non-leaf node contain non-empty left and right sub-trees. In other words, the degree of every non-leaf node will always be 2. A strictly binary tree with n leaves, will have $(2n - 1)$ nodes.

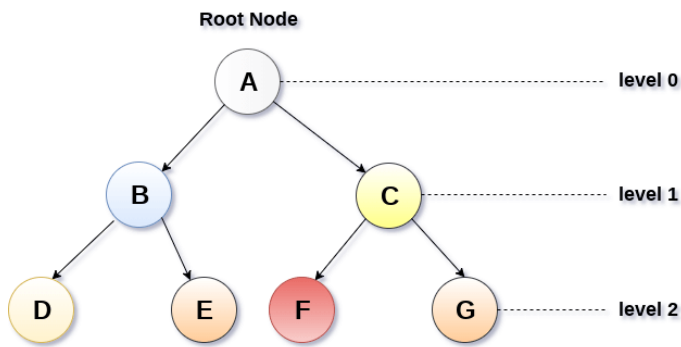
A strictly binary tree is shown in the following figure.



Strictly Binary Tree

2. Complete Binary Tree

A Binary Tree is said to be a complete binary tree if all of the leaves are located at the same level d . A complete binary tree is a binary tree that contains exactly 2^i nodes at each level between level 0 and d . The total number of nodes in a complete binary tree with depth d is $2^{d+1}-1$ where leaf nodes are 2^d while non-leaf nodes are 2^d-1 .



Complete Binary Tree

Binary Tree Traversal

SN	Traversal	Description
1	Pre-order Traversal	Traverse the root first then traverse into the left sub-tree and right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
2	In-order Traversal	Traverse the left sub-tree first, and then traverse the root and the right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
3	Post-order Traversal	Traverse the left sub-tree and then traverse the right sub-tree and root respectively. This procedure will be applied to each sub-tree of the tree recursively.

Binary Tree representation

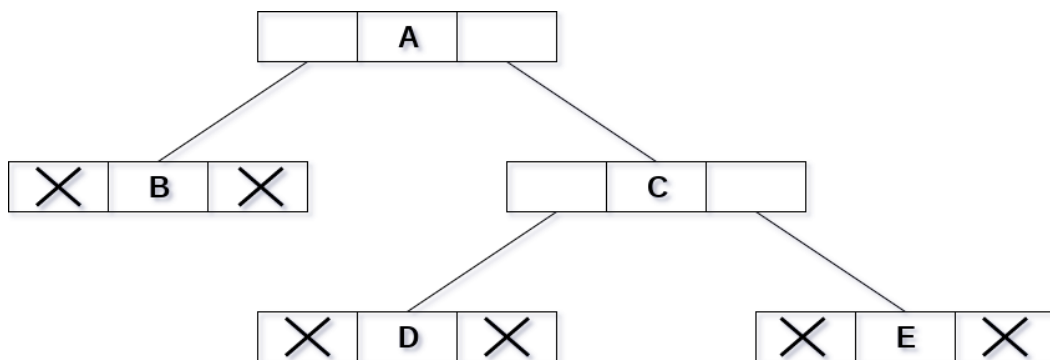
There are two types of representation of a binary tree:

1. Linked Representation

In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non-contiguous memory locations and linked together by inheriting parent

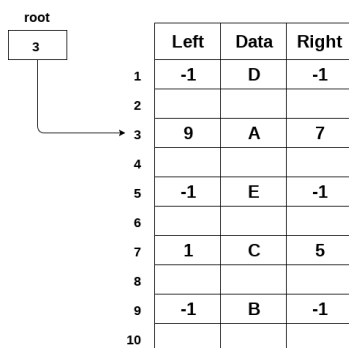
child relationship like a tree. every node contains three parts : pointer to the left node, data element and pointer to the right node. Each binary tree has a root pointer which points to the root node of the binary tree. In an empty binary tree, the root pointer will point to null.

Consider the binary tree given in the figure below.



In the above figure, a tree is seen as the collection of nodes where each node contains three parts : left pointer, data element and right pointer. Left pointer stores the address of the left child while the right pointer stores the address of the right child. The leaf node contains **null** in its left and right pointers.

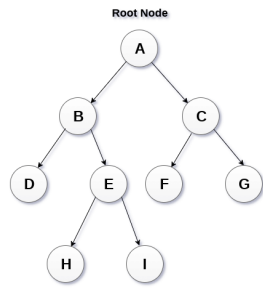
The following image shows about how the memory will be allocated for the binary tree by using linked representation. There is a special pointer maintained in the memory which points to the root node of the tree. Every node in the tree contains the address of its left and right child. Leaf node contains null in its left and right pointers.



Memory Allocation of Binary Tree using linked Representation

2. Sequential Representation

This is the simplest memory allocation technique to store the tree elements but it is an inefficient technique since it requires a lot of space to store the tree elements. A binary tree is shown in the following figure along with its memory allocation.



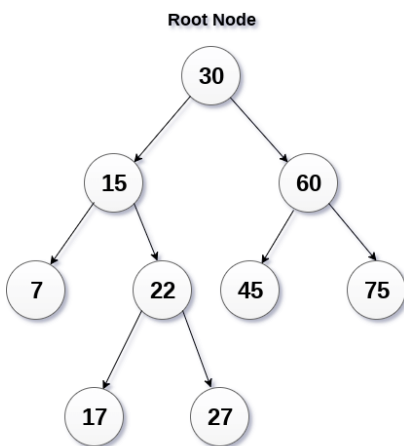
A	B	C	D	E	F	G			H	I
1	2	3	4	5	6	7	8	9	10	11

Sequential Representation of Binary Tree

In this representation, an array is used to store the tree elements. Size of the array will be equal to the number of nodes present in the tree. The root node of the tree will be present at the 1st index of the array. If a node is stored at i th index then its left and right children will be stored at $2i$ and $2i+1$ location. If the 1st index of the array i.e. `tree[1]` is 0, it means that the tree is empty.

Binary Search Tree

1. Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
2. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
3. Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
4. This rule will be recursively applied to all the left and right sub-trees of the root.



Binary Search Tree

A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.

Advantages of using binary search tree

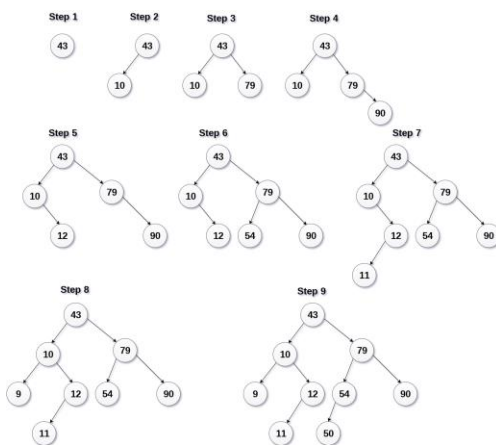
1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes $O(\log_2 n)$ time. In worst case, the time it takes to search an element is $O(n)$.
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

Q. Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.



Binary search Tree Creation

Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree.

SN	Operation	Description
----	-----------	-------------

1	<u>Searching in BST</u>	Finding the location of some specific element in a binary search tree.
2	<u>Insertion in BST</u>	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	<u>Deletion in BST</u>	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

Program to implement BST operations

```

1. #include <iostream>
2. #include <stdlib.h>
3. using namespace std;
4. struct Node {
5.     int data;
6.     Node *left;
7.     Node *right;
8. };
9. Node* create(int item)
10. {
11.     Node* node = new Node;
12.     node->data = item;
13.     node->left = node->right = NULL;
14.     return node;
15. }
16.
17. void inorder(Node *root)
18. {
19.     if (root == NULL)
20.         return;
21.
22.     inorder(root->left);
23.     cout<< root->data << " ";
24.     inorder(root->right);
25. }
26. Node* findMinimum(Node* cur)

```

```
27. {
28.     while(cur->left != NULL) {
29.         cur = cur->left;
30.     }
31.     return cur;
32. }
33. Node* insertion(Node* root, int item)
34. {
35.     if (root == NULL)
36.         return create(item);
37.     if (item < root->data)
38.         root->left = insertion(root->left, item);
39.     else
40.         root->right = insertion(root->right, item);
41.
42.     return root;
43. }
44.
45. void search(Node* &cur, int item, Node* &parent)
46. {
47.     while (cur != NULL && cur->data != item)
48.     {
49.         parent = cur;
50.
51.         if (item < cur->data)
52.             cur = cur->left;
53.         else
54.             cur = cur->right;
55.     }
56. }
57.
58. void deletion(Node*& root, int item)
59. {
60.     Node* parent = NULL;
61.     Node* cur = root;
62.
63.     search(cur, item, parent);
64.     if (cur == NULL)
65.         return;
66.
67.     if (cur->left == NULL && cur->right == NULL)
68.     {
69.         if (cur != root)
70.         {
71.             if (parent->left == cur)
```

```

72.         parent->left = NULL;
73.     else
74.         parent->right = NULL;
75. }
76. else
77.     root = NULL;
78.
79. free(cur);
80. }
81. else if (cur->left && cur->right)
82. {
83.     Node* succ = findMinimum(cur->right);
84.
85.     int val = succ->data;
86.
87.     deletion(root, succ->data);
88.
89.     cur->data = val;
90. }
91.
92. else
93. {
94.     Node* child = (cur->left)? cur->left: cur->right;
95.
96.     if (cur != root)
97.     {
98.         if (cur == parent->left)
99.             parent->left = child;
100.        else
101.            parent->right = child;
102.    }
103.
104.    else
105.        root = child;
106.    free(cur);
107. }
108. }
109.
110. int main()
111. {
112.     Node* root = NULL;
113.     int keys[8];
114.     for(int i=0;i<8;i++)
115.     {
116.         cout << "Enter value to be inserted";

```

```

117.      cin>>keys[i];
118.      root = insertion(root, keys[i]);
119.  }
120.
121.      inorder(root);
122.      cout<<"\n";
123.      deletion(root, 10);
124.      inorder(root);
125.      return 0;
126.  }

```

Output:

```

Enter value to be inserted? 10
Enter value to be inserted? 20
Enter value to be inserted? 30
Enter value to be inserted? 40
Enter value to be inserted? 5
Enter value to be inserted? 25
Enter value to be inserted? 15
Enter value to be inserted? 5

```

```

5      5      10      15      20      25      30      40
5      5      15      20      25      30      40

```

Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted. The algorithm of linear search is given as follows.

Algorithm

- LINEAR_SEARCH(A, N, VAL)
- **Step 1:** [INITIALIZE] SET POS = -1
- **Step 2:** [INITIALIZE] SET I = 1
- **Step 3:** Repeat Step 4 while I <= N
- **Step 4:** IF A[I] = VAL

SET POS = I

PRINT POS

Go to Step 6

[END OF IF]

SET I = I + 1

[END OF LOOP]

- **Step 5:** IF POS = -1

PRINT " VALUE IS NOT PRESENTIN THE ARRAY "

[END OF IF]

- **Step 6:** EXIT

Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

C Program

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
5.     int item, i, flag;
6.     printf("\nEnter Item which is to be searched\n");
7.     scanf("%d",&item);
8.     for (i = 0; i < 10; i++)
9.     {
10.        if(a[i] == item)
11.        {
12.            flag = i+1;
13.            break;
14.        }
15.        else
16.            flag = 0;
17.    }
18.    if(flag != 0)
19.    {
20.        printf("\nItem found at location %d\n",flag);
21.    }
22.    else
23.    {
24.        printf("\nItem not found\n");
25.    }
26. }
```

Output:

```
Enter Item which is to be searched
20
Item not found
Enter Item which is to be searched
23
Item found at location 2
```

Binary Search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search algorithm is given below.

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

- **Step 1:** [INITIALIZE] SET BEG = lower_bound
END = upper_bound, POS = - 1
- **Step 2:** Repeat Steps 3 and 4 while BEG <=END
- **Step 3:** SET MID = (BEG + END)/2
- **Step 4:** IF A[MID] = VAL
SET POS = MID
PRINT POS
Go to Step 6
ELSE IF A[MID] > VAL
SET END = MID - 1
ELSE
SET BEG = MID + 1
[END OF IF]
[END OF LOOP]
- **Step 5:** IF POS = -1
PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
[END OF IF]
- **Step 6:** EXIT

Complexity

SN	Performance	Complexity
1	Worst case	$O(\log n)$
2	Best case	$O(1)$
3	Average Case	$O(\log n)$
4	Worst case space complexity	$O(1)$

Example

Let us consider an array $arr = \{1, 5, 7, 8, 13, 19, 20, 23, 29\}$. Find the location of the item 23 in the array.

In 1st step :

1. $BEG = 0$
2. $END = 8$
3. $MID = 4$
4. $a[mid] = a[4] = 13 < 23$, therefore

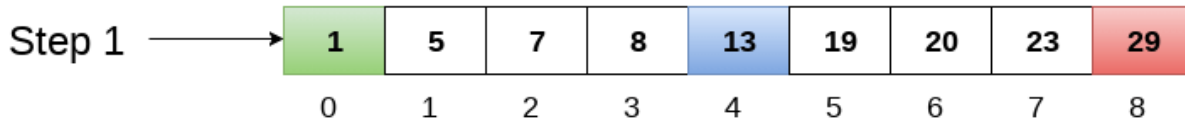
in Second step:

1. $Beg = mid + 1 = 5$
2. $End = 8$
3. $mid = 13/2 = 6$
4. $a[mid] = a[6] = 20 < 23$, therefore;

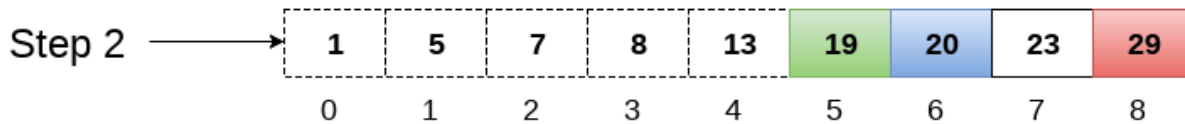
in third step:

1. $beg = mid + 1 = 7$
2. $End = 8$
3. $mid = 15/2 = 7$
4. $a[mid] = a[7]$
5. $a[7] = 23 = \text{item};$
6. therefore, set location = mid;
7. The location of the item will be 7.

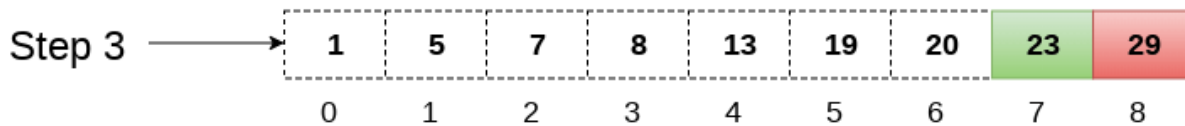
Item to be searched = 23



$a[mid] = 13$
 $13 < 23$
 $beg = mid + 1 = 5$
 $end = 8$
 $mid = (beg + end)/2 = 13 / 2 = 6$



$a[mid] = 20$
 $20 < 23$
 $beg = mid + 1 = 7$
 $end = 8$
 $mid = (beg + end)/2 = 15 / 2 = 7$



$a[mid] = 23$
 $23 = 23$
 $loc = mid$

Return location 7

Binary Search Program using Recursion

C program

1. `#include<stdio.h>`
2. `int binarySearch(int[], int, int, int);`
3. `void main ()`
4. `{`
5. `int arr[10] = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};`
6. `int item, location=-1;`
7. `printf("Enter the item which you want to search ");`

```

8.  scanf("%d",&item);
9.  location = binarySearch(arr, 0, 9, item);
10. if(location != -1)
11. {
12.     printf("Item found at location %d",location);
13. }
14. else
15. {
16.     printf("Item not found");
17. }
18.}
19.int binarySearch(int a[], int beg, int end, int item)
20.{
21.    int mid;
22.    if(end >= beg)
23.    {
24.        mid = (beg + end)/2;
25.        if(a[mid] == item)
26.        {
27.            return mid+1;
28.        }
29.        else if(a[mid] < item)
30.        {
31.            return binarySearch(a,mid+1,end,item);
32.        }
33.        else
34.        {
35.            return binarySearch(a,beg,mid-1,item);
36.        }
37.    }
38. }
39. return -1;
40.}

```

Output:

```

Enter the item which you want to search
19
Item found at location 2

```

Bubble Sort

In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Consider an array A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.

1. In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.
3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of this pass. The smallest element of the list is placed at the first index of the list.

Algorithm :

- **Step 1:** Repeat Step 2 For i = 0 to N-1
- **Step 2:** Repeat For J = i + 1 to N - 1
- **Step 3:** IF A[J] > A[i]
SWAP A[J] and A[i]
[END OF INNER LOOP]
[END OF OUTER LOOP]
- **Step 4:** EXIT

Complexity

Scenario	Complexity
Space	O(1)
Worst case running time	O(n ²)
Average case running time	O(n)
Best case running time	O(n ²)

C Program

```

1. #include<stdio.h>
2. void main ()
3. {
4.     int i, j,temp;
5.     int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
6.     for(i = 0; i<10; i++)
7.     {
8.         for(j = i+1; j<10; j++)
9.         {
10.            if(a[j] > a[i])
11.            {
12.                temp = a[i];

```

```

13.         a[i] = a[j];
14.         a[j] = temp;
15.     }
16. }
17. }
18. printf("Printing Sorted Element List ...\n");
19. for(i = 0; i<10; i++)
20. {
21.     printf("%d\n",a[i]);
22. }
23.}

```

Output:

```

Printing Sorted Element List . . .
7
9
10
12
23
34
34
44
78
101

```

Insertion Sort

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

Technique

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k-1], A[k-2], and so on until we find an element A[j] such that, $A[j] \leq A[k]$.

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

Complexity

Complexity	Best Case	Average Case	Worst Case
------------	-----------	--------------	------------

Time	$\Omega(n)$	$\theta(n^2)$	$o(n^2)$
Space			$o(1)$

Algorithm

- **Step 1:** Repeat Steps 2 to 5 for $K = 1$ to $N-1$
- **Step 2:** SET TEMP = ARR[K]
- **Step 3:** SET J = K - 1
- **Step 4:** Repeat while TEMP <= ARR[J]
SET ARR[J + 1] = ARR[J]
SET J = J - 1
[END OF INNER LOOP]
- **Step 5:** SET ARR[J + 1] = TEMP
[END OF LOOP]
- **Step 6:** EXIT

C Program

```

1. #include<stdio.h>
2. void main ()
3. {
4.     int i,j, k,temp;
5.     int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
6.     printf("\nprinting sorted elements...\n");
7.     for(k=1; k<10; k++)
8.     {
9.         temp = a[k];
10.        j= k-1;
11.        while(j>=0 && temp <= a[j])
12.        {
13.            a[j+1] = a[j];
14.            j = j-1;
15.        }
16.        a[j+1] = temp;
17.    }
18.    for(i=0;i<10;i++)
19.    {
20.        printf("\n%d\n",a[i]);
21.    }
22.}

```

Output:

```
Printing Sorted Elements . . .
7
9
10
12
23
23
34
44
78
101
```

Quick Sort

Quick sort is the widely used sorting algorithm that makes $n \log n$ comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach. The algorithm processes the array in the following way.

1. Set the first index of the array to left and loc variable. Set the last index of the array to right variable. i.e. $left = 0$, $loc = 0$, $end = n - 1$, where n is the length of the array.
2. Start from the right of the array and scan the complete array from right to beginning comparing each element of the array with the element pointed by loc.

Ensure that, $a[loc]$ is less than $a[right]$.

1. If this is the case, then continue with the comparison until right becomes equal to the loc.
 2. If $a[loc] > a[right]$, then swap the two values. And go to step 3.
 3. Set, $loc = right$
1. start from element pointed by left and compare each element in its way with the element pointed by the variable loc. Ensure that $a[loc] > a[left]$
 1. if this is the case, then continue with the comparison until loc becomes equal to left.
 2. $[loc] < a[right]$, then swap the two values and go to step 2.
 3. Set, $loc = left$.

Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	$O(n)$ for 3 way partition or $O(n \log n)$ simple partition	$O(n \log n)$	$O(n^2)$
Space Complexity	$O(1)$	$O(1)$	$O(1)$

Algorithm

PARTITION (ARR, BEG, END, LOC)

- **Step 1:** [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =
- **Step 2:** Repeat Steps 3 to 6 while FLAG =
- **Step 3:** Repeat while ARR[LOC] <=ARR[RIGHT]
AND LOC != RIGHT
SET RIGHT = RIGHT - 1
[END OF LOOP]
- **Step 4:** IF LOC = RIGHT
SET FLAG = 1
ELSE IF ARR[LOC] > ARR[RIGHT]
SWAP ARR[LOC] with ARR[RIGHT]
SET LOC = RIGHT
[END OF IF]
- **Step 5:** IF FLAG = 0
Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
SET LEFT = LEFT + 1
[END OF LOOP]
- **Step 6:**IF LOC = LEFT
SET FLAG = 1
ELSE IF ARR[LOC] < ARR[LEFT]
SWAP ARR[LOC] with ARR[LEFT]
SET LOC = LEFT
[END OF IF]
[END OF IF]
- **Step 7:** [END OF LOOP]
- **Step 8:** END

QUICK_SORT (ARR, BEG, END)

- **Step 1:** IF (BEG < END)
CALL PARTITION (ARR, BEG, END, LOC)
CALL QUICKSORT(ARR, BEG, LOC - 1)
CALL QUICKSORT(ARR, LOC + 1, END)
[END OF IF]
- **Step 2:** END

C Program

1. #include <stdio.h>
2. **int** partition(**int** a[], **int** beg, **int** end);
3. **void** quickSort(**int** a[], **int** beg, **int** end);
4. **void** main()
5. {

```
6.  int i;
7.  int arr[10]={90,23,101,45,65,23,67,89,34,23};
8.  quickSort(arr, 0, 9);
9.  printf("\n The sorted array is: \n");
10. for(i=0;i<10;i++)
11.  printf(" %d\t", arr[i]);
12. }
13. int partition(int a[], int beg, int end)
14. {
15.
16.  int left, right, temp, loc, flag;
17.  loc = left = beg;
18.  right = end;
19.  flag = 0;
20.  while(flag != 1)
21.  {
22.      while((a[loc] <= a[right]) && (loc!=right))
23.          right--;
24.      if(loc==right)
25.          flag = 1;
26.      else if(a[loc]>a[right])
27.      {
28.          temp = a[loc];
29.          a[loc] = a[right];
30.          a[right] = temp;
31.          loc = right;
32.      }
33.      if(flag!=1)
34.      {
35.          while((a[loc] >= a[left]) && (loc!=left))
36.              left++;
37.          if(loc==left)
38.              flag = 1;
39.          else if(a[loc] <a[left])
40.          {
41.              temp = a[loc];
42.              a[loc] = a[left];
43.              a[left] = temp;
44.              loc = left;
45.          }
46.      }
47.  }
48.  return loc;
49. }
50. void quickSort(int a[], int beg, int end)
```



```

51.{
52.
53.  int loc;
54.  if(beg<end)
55.  {
56.      loc = partition(a, beg, end);
57.      quickSort(a, beg, loc-1);
58.      quickSort(a, loc+1, end);
59.  }
60.}

```

Output:

```

The sorted array is:
23
23
23
34
45
65
67
89
90
101

```

Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using $n-1$ pass of selection sort algorithm.

- In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap $A[0]$ and $A[pos]$. Thus $A[0]$ is sorted, we now have $n-1$ elements which are to be sorted.
- In 2nd pas, position pos of the smallest element present in the sub-array $A[n-1]$ is found. Then, swap, $A[1]$ and $A[pos]$. Thus $A[0]$ and $A[1]$ are sorted, we now left with $n-2$ unsorted elements.
- In $n-1$ th pass, position pos of the smaller element between $A[n-1]$ and $A[n-2]$ is to be found. Then, swap, $A[pos]$ and $A[n-1]$.

Therefore, by following the above explained process, the elements $A[0]$, $A[1]$, $A[2]$, ..., $A[n-1]$ are sorted.

Example

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

A = {10, 2, 3, 90, 43, 56}.

Pass	Pos	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
1	1	2	10	3	90	43	56
2	2	2	3	10	90	43	56
3	2	2	3	10	90	43	56
4	4	2	3	10	43	90	56
5	5	2	3	10	43	56	90

Sorted A = {2, 3, 10, 43, 56, 90}

Complexity

Complexity	Best Case	Average Case	Worst Case
Time	$\Omega(n)$	$\theta(n^2)$	$o(n^2)$
Space			$o(1)$

Algorithm

SELECTION SORT(ARR, N)

- **Step 1:** Repeat Steps 2 and 3 for K = 1 to N-1
- **Step 2:** CALL SMALLEST(ARR, K, N, POS)
- **Step 3:** SWAP A[K] with ARR[POS]
[END OF LOOP]
- **Step 4:** EXIT

SMALLEST (ARR, K, N, POS)

- **Step 1:** [INITIALIZE] SET SMALL = ARR[K]
- **Step 2:** [INITIALIZE] SET POS = K
- **Step 3:** Repeat for J = K+1 to N -1
IF SMALL > ARR[J]
SET SMALL = ARR[J]
SET POS = J
[END OF IF]
[END OF LOOP]

- **Step 4:** RETURN POS

C Program

```
1. #include<stdio.h>
2. int smallest(int[],int,int);
3. void main ()
4. {
5.     int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
6.     int i,j,k,pos,temp;
7.     for(i=0;i<10;i++)
8.     {
9.         pos = smallest(a,10,i);
10.        temp = a[i];
11.        a[i]=a[pos];
12.        a[pos] = temp;
13.    }
14.    printf("\nprinting sorted elements...\n");
15.    for(i=0;i<10;i++)
16.    {
17.        printf("%d\n",a[i]);
18.    }
19.}
20.int smallest(int a[], int n, int i)
21.{
22.    int small,pos,j;
23.    small = a[i];
24.    pos = i;
25.    for(j=i+1;j<10;j++)
26.    {
27.        if(a[j]<small)
28.        {
29.            small = a[j];
30.            pos=j;
31.        }
32.    }
33.    return pos;
34.}
```

Output:

```
printing sorted elements...
7
9
10
12
23
```

23
34
44
78
101

Merge sort

Merge sort is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.

1. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
2. Conquer means sort the two sub-arrays recursively using the merge sort.
3. Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

The main idea behind merge sort is that, the short list takes less time to be sorted.

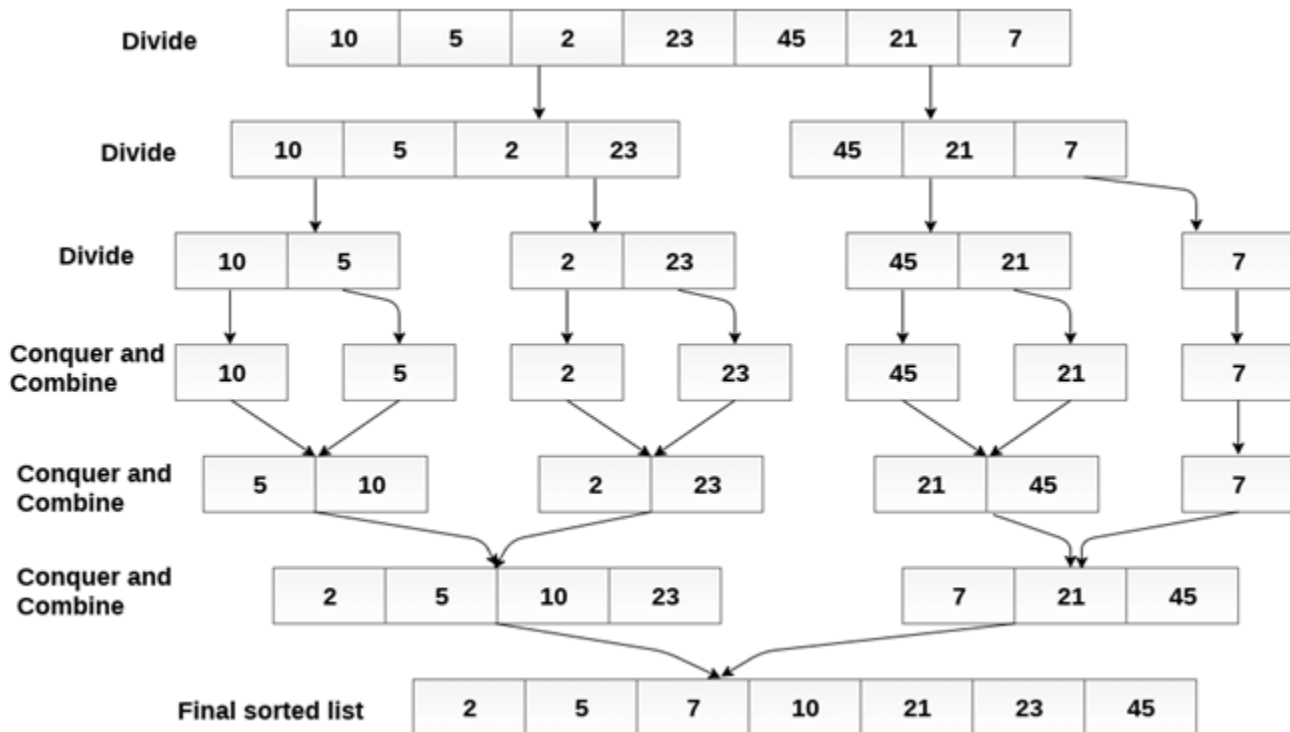
Complexity

Complexity	Best case	Average Case	Worst Case
Time Complexity	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Space Complexity			$O(n)$

Example :

Consider the following array of 7 elements. Sort the array by using merge sort.

1. $A = \{10, 5, 2, 23, 45, 21, 7\}$



Algorithm

- **Step 1:** [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
- **Step 2:** Repeat while (I <= MID) AND (J <= END)
 - IF ARR[I] < ARR[J]
 - SET TEMP[INDEX] = ARR[I]
 - SET I = I + 1
 - ELSE
 - SET TEMP[INDEX] = ARR[J]
 - SET J = J + 1
 - [END OF IF]
 - SET INDEX = INDEX + 1
 - [END OF LOOP]
- Step 3: [Copy the remaining elements of right sub-array, if any]
- IF I > MID
- Repeat while J <= END
- SET TEMP[INDEX] = ARR[J]
- SET INDEX = INDEX + 1, SET J = J + 1
- [END OF LOOP]
- [Copy the remaining elements of left sub-array, if any]
- ELSE
- Repeat while I <= MID

```

SET TEMP[INDEX] = ARR[I]
SET INDEX = INDEX + 1, SET I = I + 1
[END OF LOOP]
[END OF IF]

```

- **Step 4:** [Copy the contents of TEMP back to ARR] SET K = 0
- **Step 5:** Repeat while K < INDEX
 SET ARR[K] = TEMP[K]
 SET K = K + 1
 [END OF LOOP]
- **Step 6:** Exit

MERGE_SORT(ARR, BEG, END)

- **Step 1:** IF BEG < END
 SET MID = (BEG + END)/2
 CALL MERGE_SORT (ARR, BEG, MID)
 CALL MERGE_SORT (ARR, MID + 1, END)
 MERGE (ARR, BEG, MID, END)
 [END OF IF]
- **Step 2:** END

C Program

```

1. #include<stdio.h>
2. void mergeSort(int[],int,int);
3. void merge(int[],int,int,int);
4. void main ()
5. {
6.     int a[10]= {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
7.     int i;
8.     mergeSort(a,0,9);
9.     printf("printing the sorted elements");
10.    for(i=0;i<10;i++)
11.    {
12.        printf("\n%d\n",a[i]);
13.    }
14.
15.}
16.void mergeSort(int a[], int beg, int end)
17.{
18.    int mid;
19.    if(beg<end)
20.    {
21.        mid = (beg+end)/2;
22.        mergeSort(a,beg,mid);

```

```
23.     mergeSort(a,mid+1,end);
24.     merge(a,beg,mid,end);
25. }
26.}
27. void merge(int a[], int beg, int mid, int end)
28. {
29.     int i=beg,j=mid+1,k,index = beg;
30.     int temp[10];
31.     while(i<=mid && j<=end)
32.     {
33.         if(a[i]<a[j])
34.         {
35.             temp[index] = a[i];
36.             i = i+1;
37.         }
38.         else
39.         {
40.             temp[index] = a[j];
41.             j = j+1;
42.         }
43.         index++;
44.     }
45.     if(i>mid)
46.     {
47.         while(j<=end)
48.         {
49.             temp[index] = a[j];
50.             index++;
51.             j++;
52.         }
53.     }
54.     else
55.     {
56.         while(i<=mid)
57.         {
58.             temp[index] = a[i];
59.             index++;
60.             i++;
61.         }
62.     }
63.     k = beg;
64.     while(k<index)
65.     {
66.         a[k]=temp[k];
67.         k++;
```

```
68.  }  
69. }
```

Output:

```
printing the sorted elements  
7  
9  
10  
12  
23  
23  
34  
44  
78  
101
```

Heap Sort

Heap sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

- Build a heap H, using the elements of ARR.
- Repeatedly delete the root element of the heap formed in phase 1.

Complexity

Complexity	Best Case	Average Case	Worst case
Time Complexity	$\Omega(n \log (n))$	$\theta(n \log (n))$	$O(n \log (n))$
Space Complexity			$O(1)$

Algorithm

HEAP_SORT(ARR, N)

- **Step 1:** [Build Heap H]
Repeat for i=0 to N-1
CALL INSERT_HEAP(ARR, N, ARR[i])
[END OF LOOP]

- **Step 2:** Repeatedly Delete the root element
Repeat while $N > 0$
CALL Delete_Heap(ARR,N,VAL)
SET $N = N+1$
[END OF LOOP]
- **Step 3:** END

C Program

```

1. #include<stdio.h>
2. int temp;
3.
4. void heapify(int arr[], int size, int i)
5. {
6.     int largest = i;
7.     int left = 2*i + 1;
8.     int right = 2*i + 2;
9.
10.    if (left < size && arr[left] > arr[largest])
11.        largest = left;
12.
13.    if (right < size && arr[right] > arr[largest])
14.        largest = right;
15.
16.    if (largest != i)
17.    {
18.        temp = arr[i];
19.        arr[i] = arr[largest];
20.        arr[largest] = temp;
21.        heapify(arr, size, largest);
22.    }
23. }
24.
25. void heapSort(int arr[], int size)
26. {
27.     int i;
28.     for (i = size / 2 - 1; i >= 0; i--)
29.         heapify(arr, size, i);
30.     for (i = size - 1; i >= 0; i--)
31.     {
32.         temp = arr[0];
33.         arr[0] = arr[i];
34.         arr[i] = temp;
35.         heapify(arr, i, 0);
36.     }

```

```
37.}
38.
39. void main()
40.{
41. int arr[] = {1, 10, 2, 3, 4, 1, 2, 100, 23, 2};
42. int i;
43. int size = sizeof(arr)/sizeof(arr[0]);
44.
45. heapSort(arr, size);
46.
47. printf("printing sorted elements\n");
48. for (i=0; i<size; ++i)
49. printf("%d\n",arr[i]);
50.}
```

Output:

```
printing sorted elements
1
1
2
2
2
3
4
10
23
100
```

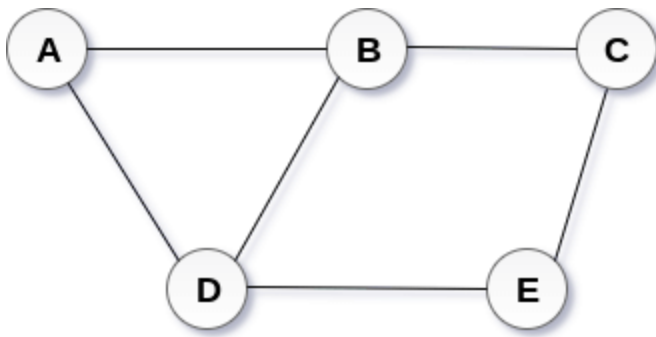
Graph

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set $G(V, E)$ where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges which are used to connect these vertices.

A Graph $G(V, E)$ with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



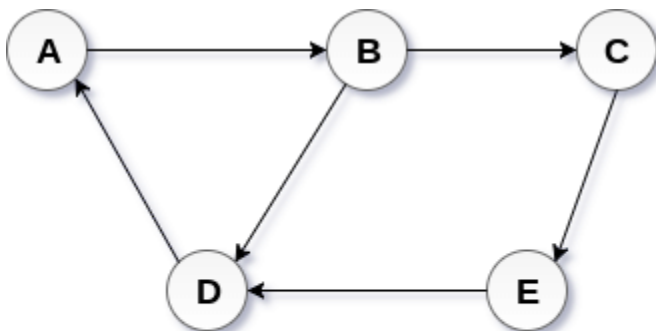
Undirected Graph

Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



Directed Graph

Graph Terminology

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0 = V_N$.

Simple Path

If all the nodes of the graph are distinct with an exception $V_0 = V_N$, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain $n(n-1)/2$ edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as $w(e)$ which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

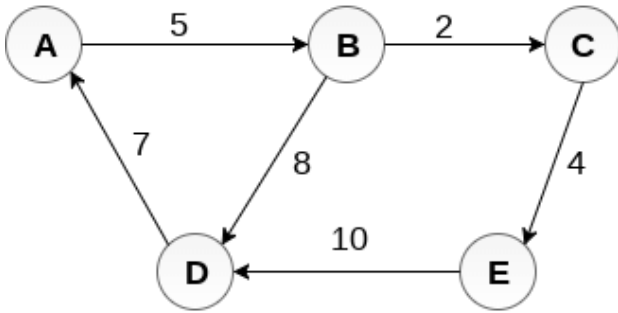
An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes

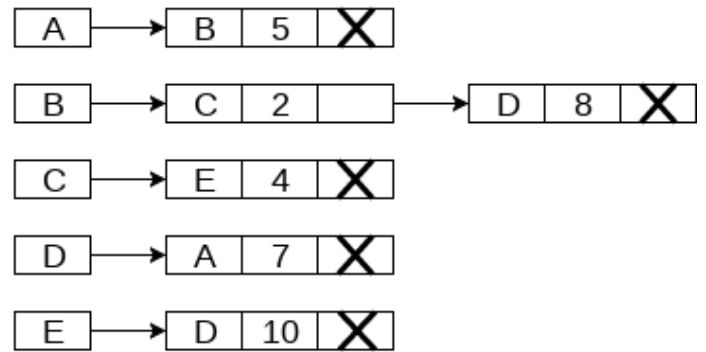
If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.



Weighted Directed Graph



Adjacency List

Graph Representation

By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.

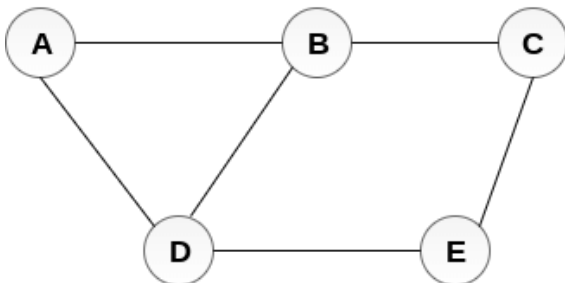
There are two ways to store Graph into the computer's memory. In this part of this tutorial, we discuss each one of them in detail.

1. Sequential Representation

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices, will have a dimension $n \times n$.

An entry M_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between V_i and V_j .

An undirected graph and its adjacency matrix representation is shown in the following figure.



Undirected Graph

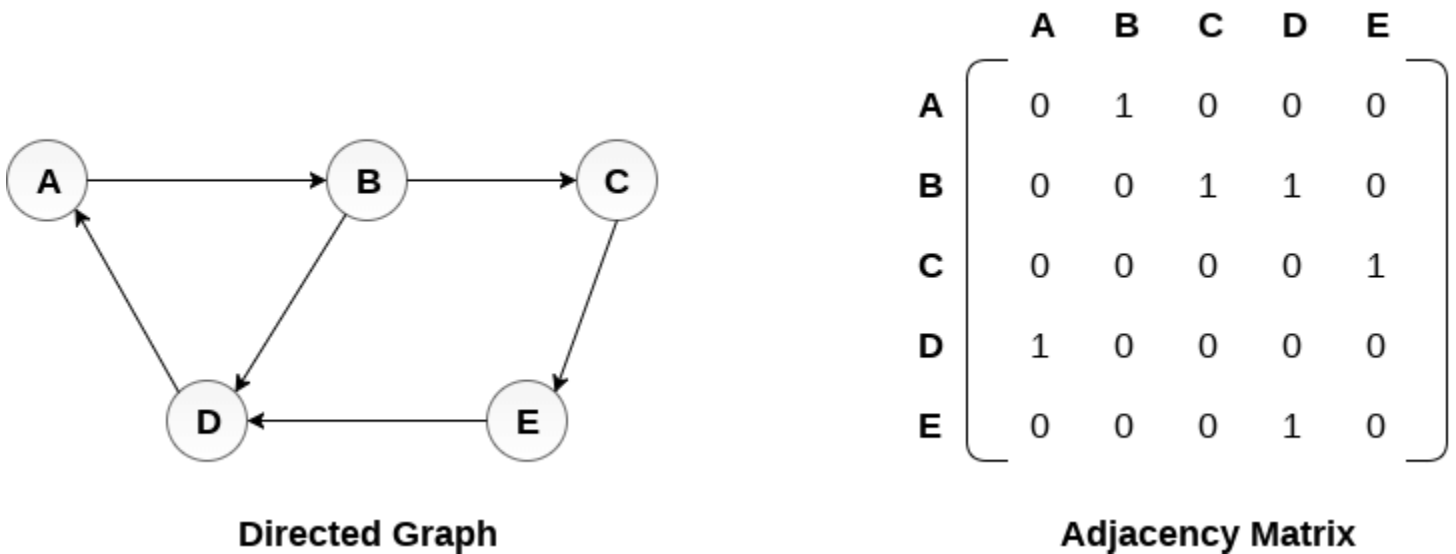
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

in the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.

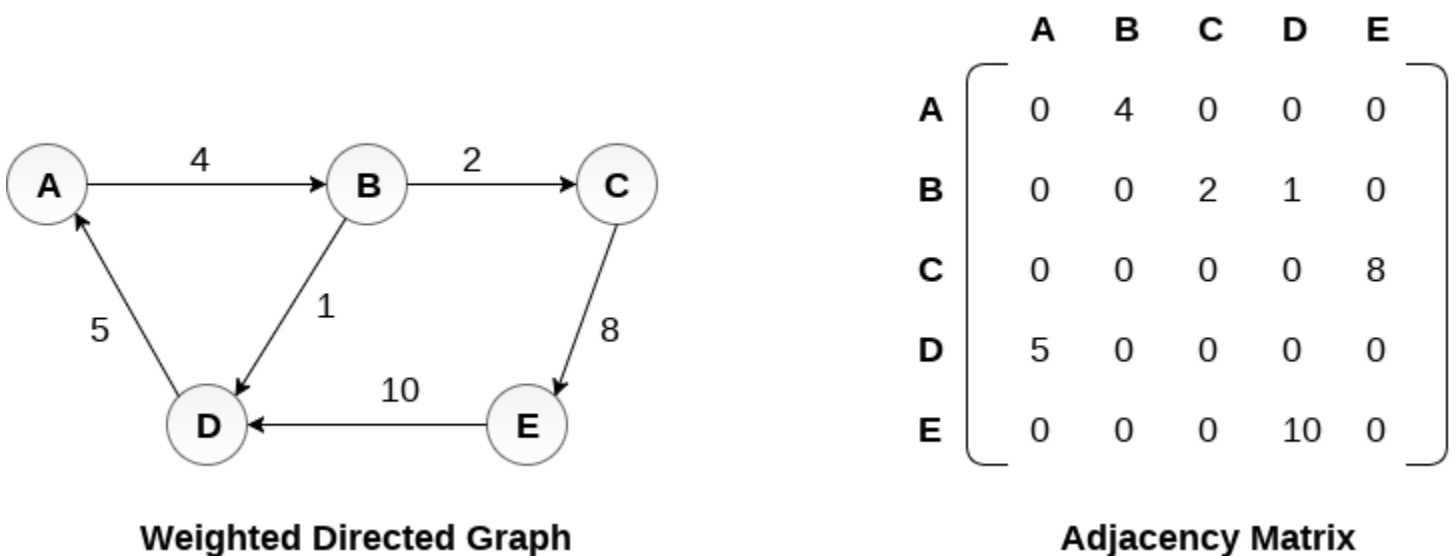
There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j .

A directed graph and its adjacency matrix representation is shown in the following figure.



Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges.

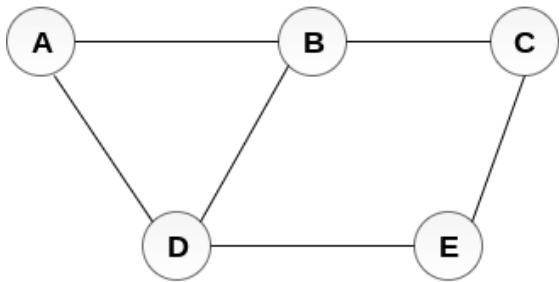
The weighted directed graph along with the adjacency matrix representation is shown in the following figure.



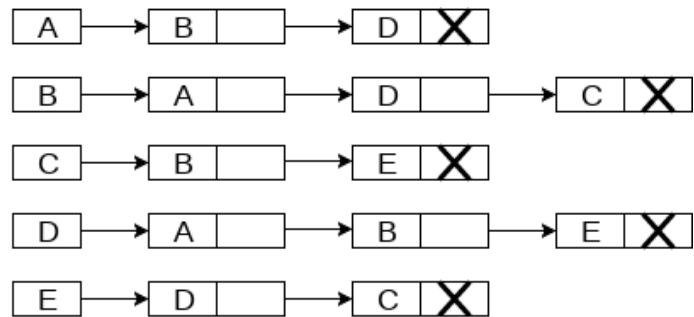
Linked Representation

In the linked representation, an adjacency list is used to store the Graph into the computer's memory.

Consider the undirected graph shown in the following figure and check the adjacency list representation.



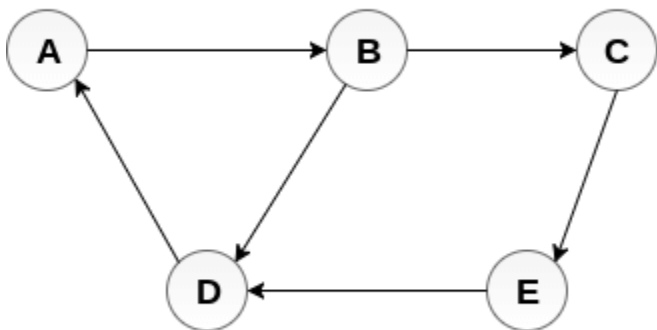
Undirected Graph



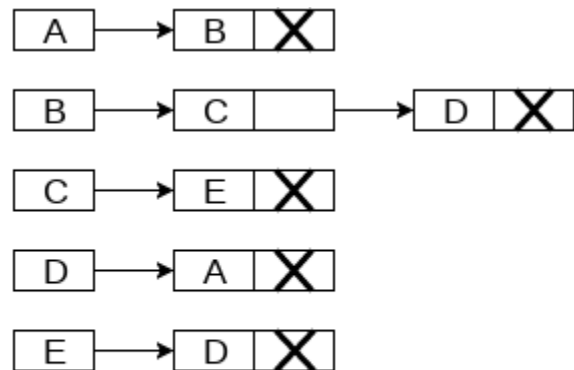
Adjacency List

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.



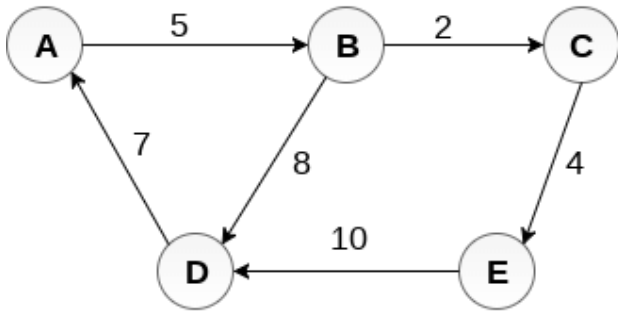
Directed Graph



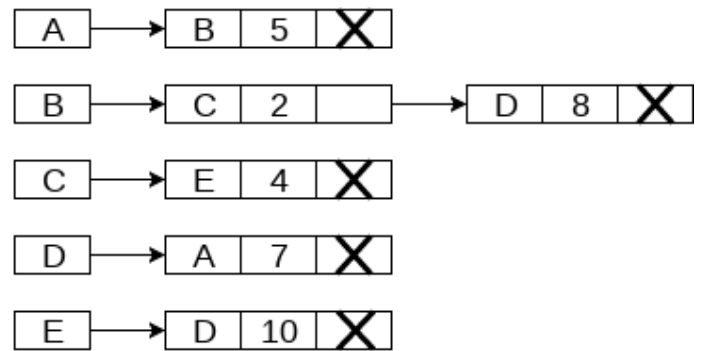
Adjacency List

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.



Weighted Directed Graph



Adjacency List

Graph Traversal Algorithm

In this part of the tutorial we will discuss the techniques by using which, we can traverse all the vertices of the graph.

Traversing the graph means examining all the nodes and vertices of the graph. There are two standard methods by using which, we can traverse the graphs. Lets discuss each one of them in detail.

- Breadth First Search
- Depth First Search

Breadth First Search (BFS) Algorithm

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

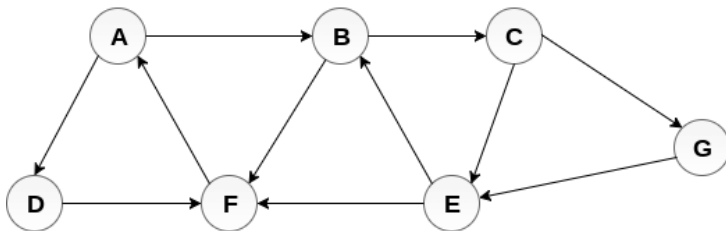
Algorithm

- **Step 1:** SET STATUS = 1 (ready state)
for each node in G
- **Step 2:** Enqueue the starting node A
and set its STATUS = 2
(waiting state)
- **Step 3:** Repeat Steps 4 and 5 until
QUEUE is empty
- **Step 4:** Dequeue a node N. Process it
and set its STATUS = 3
(processed state).

- **Step 5:** Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
[END OF LOOP]
- **Step 6:** EXIT

Example

Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.



Adjacency Lists

```

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F
  
```

Solution:

Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely **QUEUE1** and **QUEUE2**. **QUEUE1** holds all the nodes that are to be processed while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**.

Lets start examining the graph from Node A.

1. Add A to QUEUE1 and NULL to QUEUE2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

1. QUEUE1 = {C, F}
2. QUEUE2 = {A, B, D}

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

Now, backtrack from E to A, using the nodes available in QUEUE2.

The minimum path will be **A** → **B** → **C** → **E**.

Depth First Search (DFS) Algorithm

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

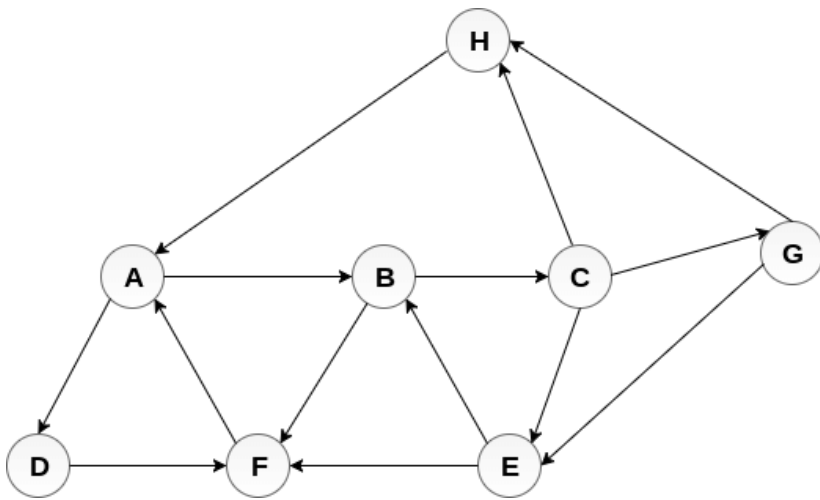
The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until STACK is empty
- **Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)
- **Step 5:** Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
[END OF LOOP]
- **Step 6:** EXIT

Example :

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Solution :

Push H onto the stack

1. STACK : H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are in ready state.

1. Print H
2. STACK : A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

1. Print A
2. Stack : B, D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

1. Print D
2. Stack : B, F

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

1. Print F
2. Stack : B

Pop the top of the stack i.e. B and push all the neighbours

1. Print B
2. Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

1. Print C
2. Stack : E, G

Pop the top of the stack i.e. G and push all its neighbours.

1. Print G
2. Stack : E

Pop the top of the stack i.e. E and push all its neighbours.

1. Print E
2. Stack :

Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be :

1. $H \rightarrow A \rightarrow D \rightarrow F \rightarrow B \rightarrow C \rightarrow G \rightarrow E$