C-PROGRAMMING & DATA STRUCTURES

LECTURE NOTES

(20A05201T)

I – BTECH

Prepared by

P.Khatija Khan & K.Khushboo

Department of Computer Science & Engineering

VEMU INSTITUTE OF TECHNOLOGY

(Approved by AICTE, New Delhi and Affiliated to JNTUA, Ananthapuramu)
Accredited by NAAC, NBA (EEE , ECE & CSE) & ISO 9001-2015 Certified Institution
Near Pakala. P. Kothakota, Chittoor-Tirupati Highway
Chittoor, Andhra Pradesh -517112
Website: www.vemu.org

R20 Regulations

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR
(Established by Govt. of A.P., ACT No.30 of 2008)
ANANTAPUR – 515 002 (A.P) INDIA

Course Code	**C&DS**	**L**	**T**	**P**	**C**
20A05201T		**3**	**0**	**0**	**3**

Course Objectives

- To illustrate the basic concepts of C programming language.
- To discuss the concepts of Functions, Arrays, Pointers and Structures.
- To familiarize with Stack, Queue and Linked lists data structures.
- To explain the concepts of non-linear data structures like graphs and trees.
- To learn different types of searching and sorting techniques.

Course outcomes (CO) : After completion of the course, the student can able to

CO-1: To illustrate the Basic concepts of C programming language.

CO-2: To apply the concepts of Functions, Arrays, Pointers and Structures in programming.

CO-3: To use concepts like Stack, Queue and Linked list in Data Structures.

CO-4: To apply the concepts of Non-Linear data Structures like graphs and Trees in organizing and storing the data.

CO-5: To use different types of Searching and Sorting techniques for data retrieving and arranging.

Syllabus

UNIT-1

Introduction to C Language - C language elements, variable declarations and data types, operators and expressions, decision statements - If and switch statements, loop control statements - while, for, do-while statements, arrays.

UNIT – 2

Functions, types of functions, Recursion and argument passing, pointers, storage allocation, pointers to functions, expressions involving pointers, Storage classes – auto, register, static, extern, Structures, Unions, Strings, string handling functions, and Command line arguments.

UNIT-3

Data Structures, Overview of data structures, stacks and queues, representation of a stack, stack related terms, operations on a stack, implementation of a stack, evaluation of arithmetic expressions, infix, prefix, and postfix notations, evaluation of postfix expression, conversion of expression from infix to postfix, recursion, queues - various positions of queue, representation of queue, insertion, deletion, searching operations.

UNIT – 4

Linked Lists – Singly linked list, dynamically linked stacks and queues, polynomials using singly linked lists, using circularly linked lists, insertion, deletion and searching operations, doubly linked lists and its operations, circular linked lists and its operations.

UNIT-5

Trees - Tree terminology, representation, Binary trees, representation, binary tree traversals. binary tree operations, Graphs - graph terminology, graph representation, elementary graph operations, Breadth

First Search (BFS) and Depth First Search (DFS), connected components, spanning trees. Searching and Sorting – sequential search, binary search, exchange (bubble) sort, selection sort, insertion sort.

Text Books:

1. The C Programming Language, Brian W Kernighan and Dennis M Ritchie, Second Edition, Prentice Hall Publication.
2. Fundamentals of Data Structures in C, Ellis Horowitz, SartajSahni, Susan Anderson-Freed, Computer Science Press.
3. Programming in C and Data Structures, J.R.Hanly, Ashok N. Kamthane and A. AnandaRao, Pearson Education.
4. B.A. Forouzon and R.F. Gilberg, "COMPUTER SCIENCE: A Structured Programming Approach Using C", Third edition, CENGAGE Learning, 2016.
5. Richard F. Gilberg & Behrouz A. Forouzan, "Data Structures: A Pseudocode Approach with C", Second Edition, CENGAGE Learning, 2011.

Reference Books:

1. Pradip Dey and Manas Ghosh, Programming in C, Oxford University Press, 2nd Edition 2011.
2. E. Balaguruswamy, "C and Data Structures", 4th Edition, Tata Mc Graw Hill.
3. A.K. Sharma, Computer Fundamentals and Programming in C, 2nd Edition, University Press.
4. M.T. Somashekara, "Problem Solving Using C", PHI, 2nd Edition 2009.

UNIT-1

UNIT-I

What is a Computer?

Definition : Computer is an advanced electronic device that takes raw data as an input from the user and processes it under the control of a set of instructions (called program), produces a result (output), and saves it for future use

Functionalities of a Computer:

If we look at it in a very broad sense, any digital computer carries out the following five functions −

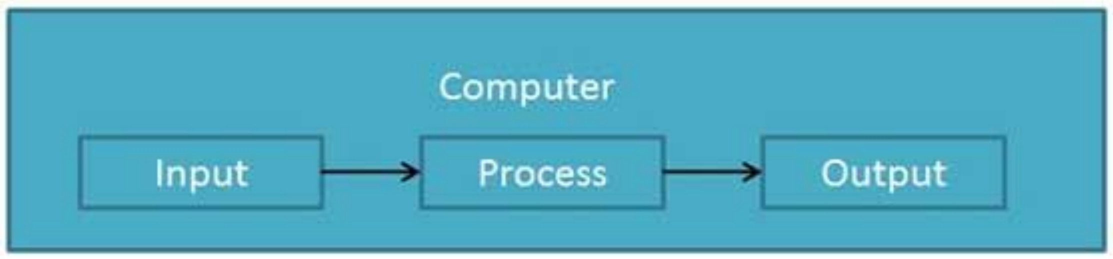
Step 1 − Takes data as input.

Step 2 − Stores the data/instructions in its memory and uses them as required.

Step 3 − Processes the data and converts it into useful information.

Step 4 − Generates the output.

Step 5 − Controls all the above four steps.



Block Diagram of a computer:

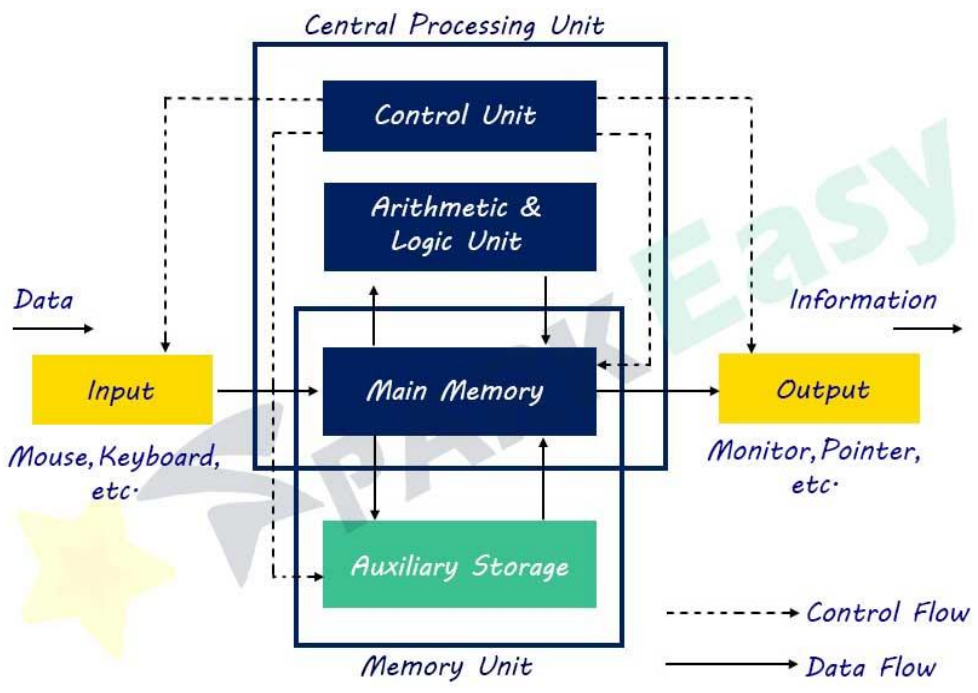


Fig: Block Diagram of Computer

Input Unit:

This unit contains devices with the help of which we enter data into the computer. This unit creates a link between the user and the computer. The input devices translate the information into a form understandable by the computer.

CPU (Central Processing Unit):

CPU is considered as the brain of the computer. CPU performs all types of data processing operations. It stores data, intermediate results, and instructions (program). It controls the operation of all parts of the computer.

CPU itself has the following three components −

- ALU (Arithmetic Logic Unit)
- Memory Unit
- Control Unit

Arithmetic Logic Unit (ALU):

Data entered into computer is sent to RAM, from where it is then sent to ALU, where rest of data processing takes place. All types of processing, such as comparisons, decision-making and processing of non-numeric information takes place here and once again data is moved to RAM.

Control Unit:

As name indicates, this part of CPU extracts instructions, performs execution, maintains and directs operations of entire system.

Functions of Control Unit

Control unit performs following functions −

- It controls all activities of computer
- Supervises flow of data within CPU
- Directs flow of data within CPU
- Transfers data to Arithmetic and Logic Unit
- Transfers results to memory
- Fetches results from memory to output devices

Memory Unit:

This is unit in which data and instructions given to computer as well as results given by computer are stored. Unit of memory is "Byte".

1 Byte = 8 Bits

Output Unit:

The output unit consists of devices with the help of which we get the information from the computer. This unit is a link between the computer and the users. Output devices translate the computer's output into a form understandable by the users.

Structure of C Program

The structure of a C program can be mainly divided into six parts, each having its purpose. It makes the program easy to read, easy to modify, easy to document, and makes it consistent in format.

Basic Structure of the C Program:

Section	Description
Documentation	Consists of the description of the program, programmer's name, and creation date. These are generally written in the form of comments.
Link	All header files are included in this section which contains different functions from the libraries. A copy of these header files is inserted into your code before compilation.
Definition	Includes preprocessor directive, which contains symbolic constants. E.g.: #define allows us to use constants in our code. It replaces all the constants with its value in the code.
Global Declaration	Includes declaration of global variables, function declarations, static global variables, and functions.
Main() Function	For every C program, the execution starts from the main() function. It is mandatory to include a main() function in every C program.
Subprograms	Includes all user-defined functions (functions the user provides). They can contain the inbuilt functions and the function definitions declared in the Global Declaration section. These are called in the main() function.

Let's look at an example to understand the structure of a C program:

Example: Write a program to calculate our age.

In the following example, we'll calculate age concerning a year.

Code:

```
/**                    //Documentation
 * file: age.c
 * author: you
 * description: program to find our age.
 */

#include <stdio.h>      //Link

#define BORN 2000       //Definition
```

```
int age(int current);   //Global Declaration

int main(void)          //Main() Function
{
  int current = 2021;
  printf("Age: %d", age(current));
  return 0;
}

int age(int current) {     //Subprograms
   return current - BORN;
}
```

Output

```
Age: 21
```

Let's explore the code:

Different sections of the above code

Documentation

In a C program, single-line comments can be written using two forward slashes i.e., //, and we can create multi-line comments using /* */. Here, we've used multi-line comments.

```
/**
 * file: age.c
 * author: you
 * description: program to find our age.
 */
```

Link

All header files are included in this section.

A header file is a file that consists of C declarations that can be used between different files. It helps us in using others' code in our files. A copy of these header files is inserted into your code before compilation.

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

Definition

A preprocessor directive in C is any statement that begins with the "#" symbol. The #define is a preprocessor compiler directive used to create constants. In simple terms, #define basically allows the macro definition, which allows the use of constants in our code.

```
#define BORN 2000
```

We've created a constant BORN which is assigned a value of 2000. Generally, uppercase letters are preferred for defining the constants. The above constant BORN will be replaced by 2000 throughout our code wherever used.

#define is typically used to make a source program easy to modify and compile in different execution environments.

The define statement **does not** ends with a semicolon.

Global Declaration

This section includes all global variables, function declarations, and static variables. The variables declared in this section can be used anywhere in the program. They're accessible to all the functions of the program. Hence, they are called global variables.

```
int age(int current);
```

We've declared our age function, which takes one integer argument and returns an integer.

Main() Function

In the structure of a C program, this section contains the main function of the code. The compiler starts execution from the main() function. It can use global variables, static variables, inbuilt functions, and user-defined functions. The return type of the main() function can be void and also not necessarily int.

```
int main(void)
{
  int current = 2022;
  printf("Age: %d", age(current));
  return 0;
}
```

Here, we've declared a variable named current and assigned the value as 2022. Then we've called the printf() function, with calls the age() function, which takes only one parameter.

Subprograms

This includes the user-defined functions called in the main() function. User-defined functions are generally written after the main() function irrespective of their order.

When the user-defined function is called from the main() function, the control of the program shifts to the called function, and when it encounters a return statement, it returns to the main() function. In this case, we've defined the age() function, which takes one parameter, i.e., the current year.

```
int age(int current) {
   return current - BORN;
}
```

This function is called in the main function. It returns an integer to the main function.

Conclusion

- To conclude, the structure of a C program can be divided into six sections, namely - Documentation, Link, Definition, Global Declaration, Main() Function, and Subprograms.
- The main() function is compulsory to include in every C program, whereas the rest are optional.
- A well-structured C program makes debugging easier and increases the readability and modularity of the code.

Elements of C Programming Language

As every language has some basic geometrical rules and elements, similarly C language has some elements and rules for building a program which has some meaning.

Character Set: In Real world to communicate with people we use language like Hindi English Urdu extra which is constructed and Defined by some characters, words extra. Similarly in C programming language we have various characters to communicate with the computer in order to produce a meaningful program and can produce an output.

Character Set in C Language

Type	Set
Lowercase	a-z
Uppercase	A-Z
Digits	0-9
special characters	!,@,#,$,%
White space	space, tab, and new lines

Keywords:

1) they are those elements of C language whose meaning has already being defined or explained.
2) keyword are called pre-defined words.

32 keywords in C language

Auto	Do	Goto	signed	unsigned	Break
Void	Else	Int	Case	static	Double
Sizeof	Enum	Long	struct	char	If
While	Const	extern	register	continue	Volatile
defualt	For	typeodef	Float	short	Return
Union	Const				

Data types: In every language we have to perform some operations means some task on some values of variables.

Example:

Like 'Suhaan' Here suhaan is a set of alphabets all characters.

like '290' here 290 is a **Numerical value.**

So here Ramesh and 290 are different types and they are allotted to some variables. So in order to recognize each variables and its type we have to specify their types and their types are called **'data types'** of that

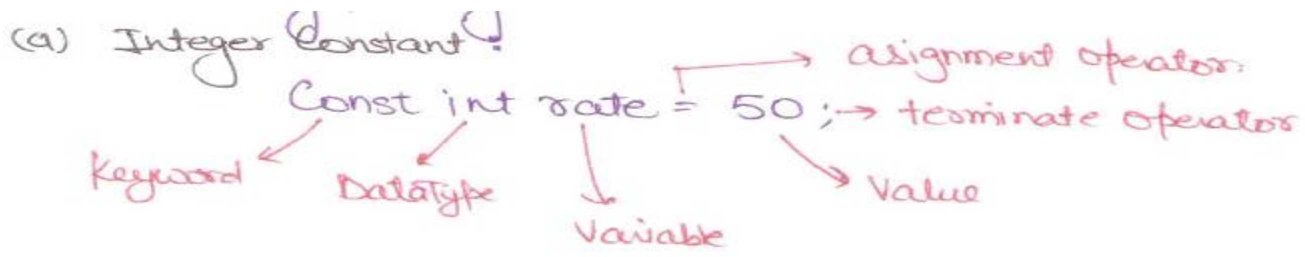
variable.
So the real Example is:
int a=6;
here a is a **variable** of **int** datatype
6 is a **value** assigned to the variable 'a' using **assignment operator**
Every data type in C language has Storage Size and Value Ranges.

Data types	Storage size	Value range
Char	1 byte	a-z, A-z, special characters (-128 to 127)
Unsigned char	1 byte	a-z, A-z, special character(0 to 255)
Int	2-4 bytes	-32768 to 32767 or -214748348 to 2147483647
unsigned int	2-4 bytes	0 to 65535 or 0 to 4294967295
Short	2 bytes	-32768 to 32767
unsigned short	2 bytes	0 to 65535
Long	4 bytes	-2147483648 to 2147483647
unsigned long	4 bytes	0 to 4294967295
Float	4 bytes	1.2E-38 to 3.4E+38 6 decimal place
Double	8 bytes	2.3E-308 to 1.7E+308 15 decimal place
long double	10 bytes	3.4E-4932 to 1.1E+4932 19 decimal place

Constants: Symbolic name or an entity that does not change its value during the execution of a program.
There are two major categories of a constant:
Constants are classified into two types they are:

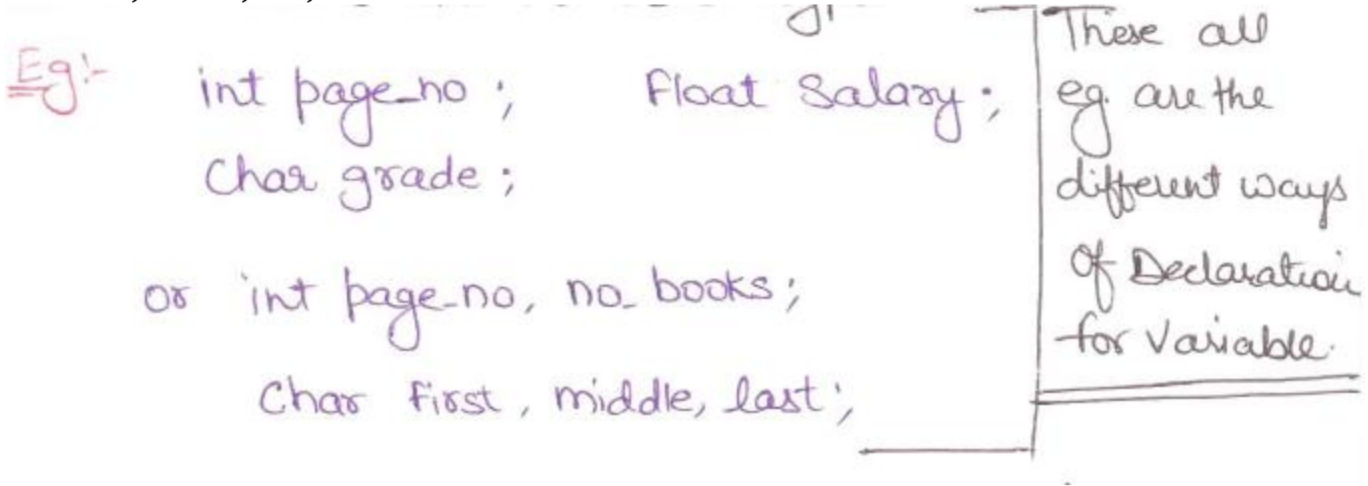
1. **Primary Constant**
2. **Secondary Constant**

Primary Constant: This category consist basic constants of C language like integer constant, float or real integer constant, character constant.
Secondary Constant: This category consist arrays, pointers, structure, union, enum etc.
We will discuss only primary constants :
a) Integer Constant:
const int rate= 50;



In this **const** is a **keyword**, **int** is a **data type**, **rate** is a **value** , **=** is a **assignment operator**, **50** is a **value**, **;** is a **terminate operator to terminate the statement**
In this value of variable **rate**'' will not be changed throughout the program .
b) const float or real integer:
const float pi= 3.1415;---> this is called the declaration with complete statement .
c) Character Constant:
const char ch= 'A';
a character data type value is always assigned with ''quotes.

Variables: It is the most fundamental aspect of any language. It is a location in the Computer memory which can store value and is a given symbolic name for easy reference. Variable declaration requires that you inform c of the variable's name and date data types.
Example:
int page_no; char grade; float salary;
or
int page-no, no-books;
char first, middle, last;


Eg:- int page_no ; Float Salary ;
Char grade ;
or int page-no, no_books ;
Char First , middle, last ;
These all eg are the different ways Of Declaration for Variable.

These all examples are the different ways of declaration for variable.
Initialisation of variables: When a variable is declared it contains and defined values for garbage values. So we can initially initialise a value to the variable.
Example:
int page-no=10;
char grade='A';
float salary= 1200.50;
int a=10, b=20, c=30;

Expressions: An expression consist of combination of operands, operators, variables and functions calls. An expression can be arithmetic, logical, or relational.
Example of Expressions:
a+b=c [arithmetic operation]
a>b [relational operation]
a==b [logical operations]
func(a,b) [function calls]

Statement: A statement is a complete instruction to the computer. In C, statements are indicated by semicolon ; at the end.
Example:
legs =4; **it is a statement.**
legs=4 **it is an expression.**

Other examples are
y=x+4;
x=90;
x=a+b/c+d*z;
x=16+(y+10-5);

What is algorithm?

An algorithm is a procedure or step-by-step instruction for solving a problem. They form the foundation of writing a program.

For writing any programs, the following has to be known:

- Input
- Tasks to be preformed
- Output expected

Algorithm 1: Add two numbers entered by the user

```
Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

        sum←num1+num2

Step 5: Display sum

Step 6: Stop
```

Algorithm 2: Find the largest number among three numbers

```
Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a > b

           If a > c

              Display a is the largest number.

           Else
```

```
              Display c is the largest number.

        Else

           If b > c

              Display b is the largest number.

           Else

              Display c is the greatest number.

Step 5: Stop
```

Algorithm 3: Find Roots of a Quadratic Equation $ax^2 + bx + c = 0$

```
Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

         D ← b2-4ac

Step 4: If D ≥ 0

         r1 ← (-b+√D)/2a
```

```
              r2 ← (-b-√D)/2a

              Display r1 and r2 as roots.

        Else

              Calculate real part and imaginary part

              rp ← -b/2a

              ip ← √(-D)/2a

              Display rp+j(ip) and rp-j(ip) as roots

Step 5: Stop
```

Algorithm 4: Find the factorial of a number

```
Step 1: Start

Step 2: Declare variables n, factorial and i.

Step 3: Initialize variables

          factorial ← 1

          i ← 1

Step 4: Read value of n

Step 5: Repeat the steps until i = n

     5.1: factorial ← factorial*i

     5.2: i ← i+1

Step 6: Display factorial

Step 7: Stop
```

Algorithm 5: Check whether a number is prime or not

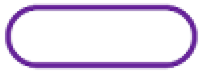
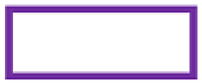
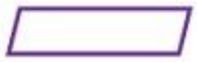
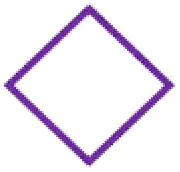
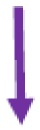
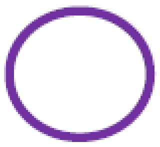
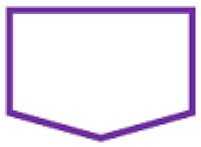
```
Step 1: Start

Step 2: Declare variables n, i, flag.

Step 3: Initialize variables

        flag ← 1

        i ← 2

Step 4: Read n from the user.

Step 5: Repeat the steps until i=(n/2)

     5.1 If remainder of n÷i equals 0

            flag ← 0

            Go to step 6

     5.2 i ← i+1

Step 6: If flag = 0

           Display n is not prime
        else
           Display n is prime
Step 7: Stop
```

Flowchart:

Flowchart is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

Flowchart Symbols:

Here is a chart for some of the common symbols used in drawing flowcharts.

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.
	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

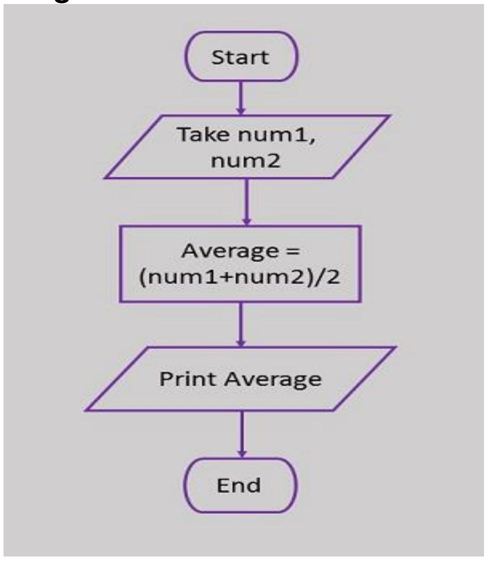
Guidelines for Developing Flowcharts:

These are some points to keep in mind while developing a flowchart –

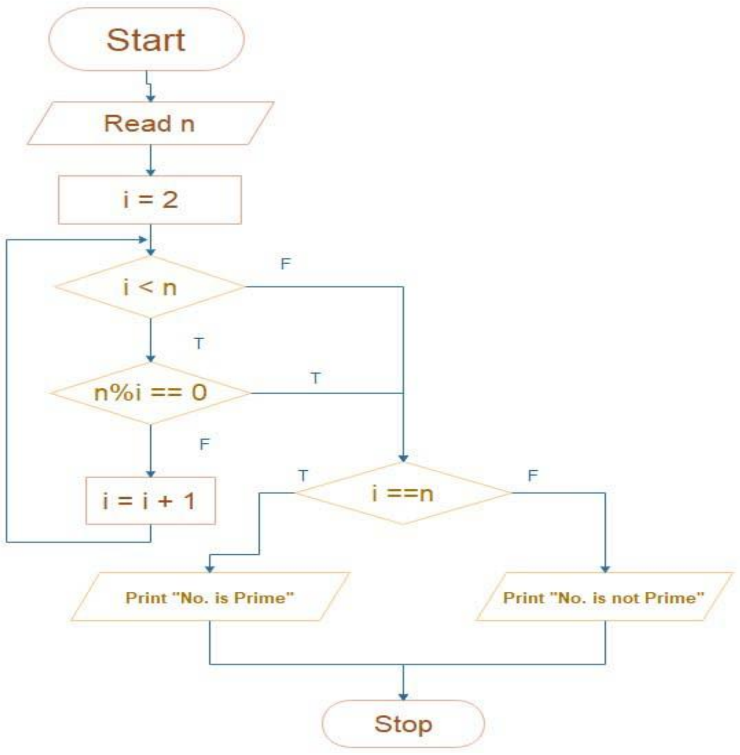
- Flowchart can have only one start and one stop symbol
- On-page connectors are referenced using numbers

- Off-page connectors are referenced using alphabets
- General flow of processes is top to bottom or left to right
- Arrows should not cross each other

Here is a flowchart to calculate the average of two numbers:



Flow chart for prime numbers:



l

C – Variables

- A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
- The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive

Variable Definition in C:

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows −
type variable_list;

Here, **type** must be a valid C data type including char, int, float, double, bool, or any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −

datatype variable_name = value;

Some examples are −

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

Variable Declaration in C:

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only, the compiler needs actual variable definition at the time of linking the program.

Example:

Try the following example, where variables have been declared at the top, but they have been defined and initialized inside the main function −

```
#include <stdio.h>

int main () {

   /* variable definition: */
   int a, b;
   int c;
   float f;

   /* actual initialization */
   a = 10;
   b = 20;

   c = a + b;
   printf("value of c : %d \n", c);

   f = 70.0/3.0;
   printf("value of f : %f \n", f);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
value of c : 30
value of f : 23.333334
```

Datatypes in C language:

Data Type	Memory (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	16	3.4E-4932 to 1.1E+4932	%Lf

Integer Types:

The integer data type in C is used to store the whole numbers without decimal values. Octal values, hexadecimal values, and decimal values can be stored in int data type in C. We can determine the size of the int data type by using the **sizeof operator** in C. Unsigned int data type in C is used to store the data values from zero to positive numbers but it can't store negative values like signed int. Unsigned int is larger in size than signed int and it uses "%u" as a format specifier in C programming language. Below is the programming implementation of the int data type in C.

- **Range:** -2,147,483,648 to 2,147,483,647
- **Size:** 2 bytes or 4 bytes
- **Format Specifier:** %d

Note: The size of an integer data type is compiler-dependent, when processors are 16-bit systems, then it shows the output of int as 2 bytes. And when processors are 32-bit then it shows 2 bytes as well as 4 bytes.

Character Types

Character data type allows its variable to store only a single character. The storage size of the character is 1. It is the most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

- **Range:** (-128 to 127) or (0 to 255)
- **Size:** 1 byte
- **Format Specifier:** %c

Floating-Point Types

In C programming float data type is used to store floating-point values. Float in C is used to store decimal and exponential values. It is used to store decimal numbers (numbers with floating point values) with single precision.

- **Range:** 1.2E-38 to 3.4E+38
- **Size:** 4 bytes
- **Format Specifier:** %f

Double Types:

A Double data type in C is used to store decimal numbers (numbers with floating point values) with double precision. It is used to define numeric values which hold numbers with decimal values in C. Double data type is basically a precision sort of data type that is capable of holding 64 bits of decimal numbers or floating points. Since double has more precision as compared to that float then it is much more obvious that it occupies twice the memory as occupied by the floating-point type. It can easily accommodate about 16 to 17 digits after or before a decimal point.

- **Range:** 1.7E-308 to 1.7E+308
- **Size:** 8 bytes
- **Format Specifier:** %lf

Void Data types:

The void data type in C is used to specify that **no value** is present. It does not provide a result value to its caller. It has no values and no operations. It is used to represent **nothing**. Void is used in multiple ways as function return type, function arguments as void, and pointers to void.

C Programming Operators

An operator is a symbol that operates on a value or a variable. For example: `+` is an operator to perform addition.

C has a wide range of operators to perform various operations.

Arithmetic Operators:

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	Multiplication
/	division
%	remainder after division (modulo division)

Example 1: Arithmetic Operators

```
// Working of arithmetic operators
#include <stdio.h>
int main()
{
   int a = 9,b = 4, c;

   c = a+b;
   printf("a+b = %d \n",c);
   c = a-b;
   printf("a-b = %d \n",c);
   c = a*b;
   printf("a*b = %d \n",c);
   c = a/b;
   printf("a/b = %d \n",c);
   c = a%b;
   printf("Remainder when a divided by b = %d \n",c);

   return 0;
}
```

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
```

```
Remainder when a divided by b=1
```

Increment and Decrement Operators:

C programming has two operators increment `++` and decrement `--` to change the value of an operand (constant or variable) by 1.
Increment `++` increases the value by 1 whereas decrement `--` decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators
#include <stdio.h>
int main()
{
   int a = 10, b = 100;
   float c = 10.5, d = 100.5;

   printf("++a = %d \n", ++a);
   printf("--b = %d \n", --b);
   printf("++c = %f \n", ++c);
   printf("--d = %f \n", --d);

   return 0;
}
```
Run Code

Output

```
++a = 11
--b = 99
++c = 11.500000
--d = 99.500000
```

Here, the operators `++` and `--` are used as prefixes. These two operators can also be used as postfixes like `a++` and `a--`. Visit this page to learn more about how increment and decrement operators work when used as postfix.

Assignment Operators:

An assignment operator is used for assigning a value to a variable. The most common assignment operator is `=`

Operator	Example	Same as
=	a = b	a = b

Operator	Example	Same as
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example 3: Assignment Operators

```
// Working of assignment operators
#include <stdio.h>
int main()
{
   int a = 5, c;

   c = a;      // c is 5
   printf("c = %d\n", c);
   c += a;     // c is 10
   printf("c = %d\n", c);
   c -= a;     // c is 5
   printf("c = %d\n", c);
   c *= a;     // c is 25
   printf("c = %d\n", c);
   c /= a;     // c is 5
   printf("c = %d\n", c);
   c %= a;     // c = 0
   printf("c = %d\n", c);

   return 0;
}
```

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1
<	Less than	5 < 3 is evaluated to 0
!=	Not equal to	5 != 3 is evaluated to 1
>=	Greater than or equal to	5 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0

Example 4: Relational Operators

```
// Working of relational operators
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10;

   printf("%d == %d is %d \n", a, b, a == b);
   printf("%d == %d is %d \n", a, c, a == c);
   printf("%d > %d is %d \n", a, b, a > b);
   printf("%d > %d is %d \n", a, c, a > c);
   printf("%d < %d is %d \n", a, b, a < b);
   printf("%d < %d is %d \n", a, c, a < c);
   printf("%d != %d is %d \n", a, b, a != b);
   printf("%d != %d is %d \n", a, c, a != c);
   printf("%d >= %d is %d \n", a, b, a >= b);
   printf("%d >= %d is %d \n", a, c, a >= c);
   printf("%d <= %d is %d \n", a, b, a <= b);
   printf("%d <= %d is %d \n", a, c, a <= c);

   return 0;
}
```

Output

```
5 == 5 is 1
5 == 10 is 0
```

```
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c==5) equals to 0.

Example 5: Logical Operators

```
// Working of logical operators

#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10, result;

   result = (a == b) && (c > b);
   printf("(a == b) && (c > b) is %d \n", result);

   result = (a == b) && (c < b);
   printf("(a == b) && (c < b) is %d \n", result);

   result = (a == b) || (c < b);
   printf("(a == b) || (c < b) is %d \n", result);

   result = (a != b) || (c < b);
   printf("(a != b) || (c < b) is %d \n", result);
```

```
    result = !(a != b);
    printf("!(a != b) is %d \n", result);

    result = !(a == b);
    printf("!(a == b) is %d \n", result);

    return 0;
}
```

Output

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

Explanation of logical operator program

- (a == b) && (c > 5) evaluates to 1 because both operands (a == b) and (c > b) is 1 (true).
- (a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
- (a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
- (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).
- !(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).
- !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

Bitwise Operators:

- During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

- Bitwise operators are used in C programming to perform bit-level operations.

Operators	**Meaning of operators**
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left

Operators	Meaning of operators
>>	Shift right

Special Operators:

Comma Operator:

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

The sizeof operator:

The sizeof is a unary operator that returns the size of data (constants, variables, array, structure, etc).
Example 6: sizeof Operator

```
#include <stdio.h>
int main()
{
    int a;
    float b;
    double c;
    char d;
    printf("Size of int=%lu bytes\n",sizeof(a));
    printf("Size of float=%lu bytes\n",sizeof(b));
    printf("Size of double=%lu bytes\n",sizeof(c));
    printf("Size of char=%lu byte\n",sizeof(d));

    return 0;
}
```

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

Other operators such as:

ternary operator	?:,
reference operator	&
dereference operator	*

Control Structures

Control structures are used to control the flow of execution in a program. These control statements are classified into the following:

1. Decision making/Conditional/Branching/sectional Statements
2. Looping Statements/Iterative Statements
3. Jump Statements/Loop control statements

Decision Making Statements:

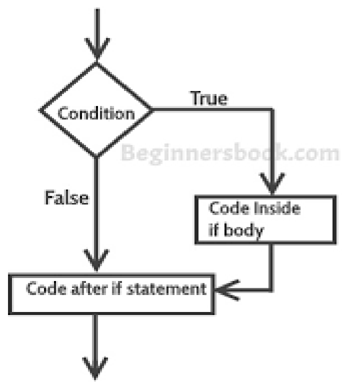
Decision making statements are executed based on the selection. If condition is true then one set of statements is executed otherwise another set of statements is executed. These statements are again classified into the following:

1. Simple if
2. if-else
3. if-else-if ladder
4. nested if
5. switch

1. simple if:

This condition occurs when a programmer can skip or execute a set of various instructions on the basis of the condition value. We select a one-way, simple statement. When the available condition gets evaluated as true, then a set of various statements will be carried out. In case the condition is false, then the control here will proceed ahead in the program with the declaration mentioned below, after the program's if declaration.

The syntax :	**Flowchart:**	**Program:**

If (condition1)
{
Body of if statement;
}
Next statement;
	 Condition
	True
	False
	Code Inside if body
	Code after if statement
	Beginnersbook.com
		include <stdio.h>
		int main () {
		int a = 10;
		if(a < 20) {
		printf("a is less than 20\n");
		}
		printf("value of a is : %d\n", a);
		return 0;
		}

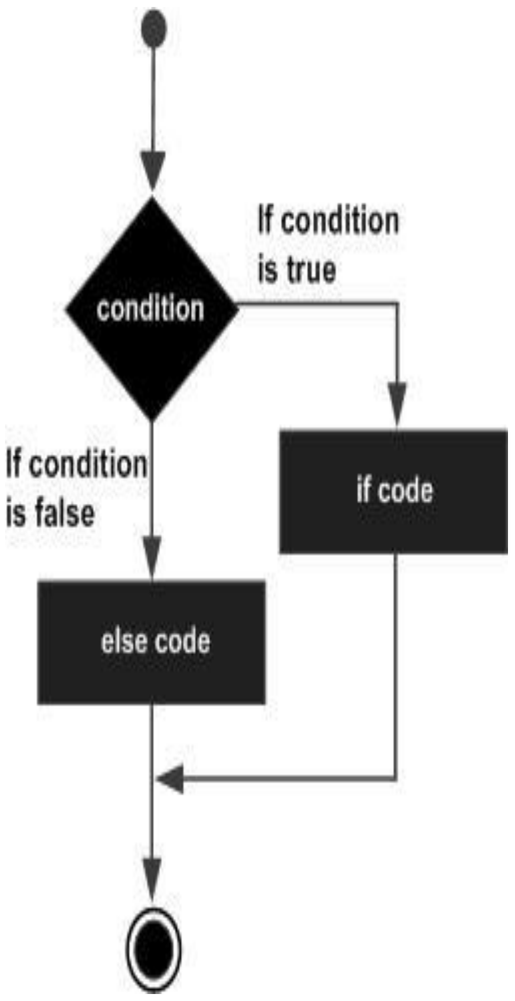
2.The If… Else Statement:

When we use the if… else statement, there occurs an execution of two different types of statements in a program. First, if the available condition in the program is true, then there will be an execution of the first statement. The execution of the **else** will only occur if the condition available to us is **false**

The syntax :

```
If (condition 1)
{
Statement 1 ;
}
else
{
Statement 2 ;
}
Next Statement;
```

Flowchart:



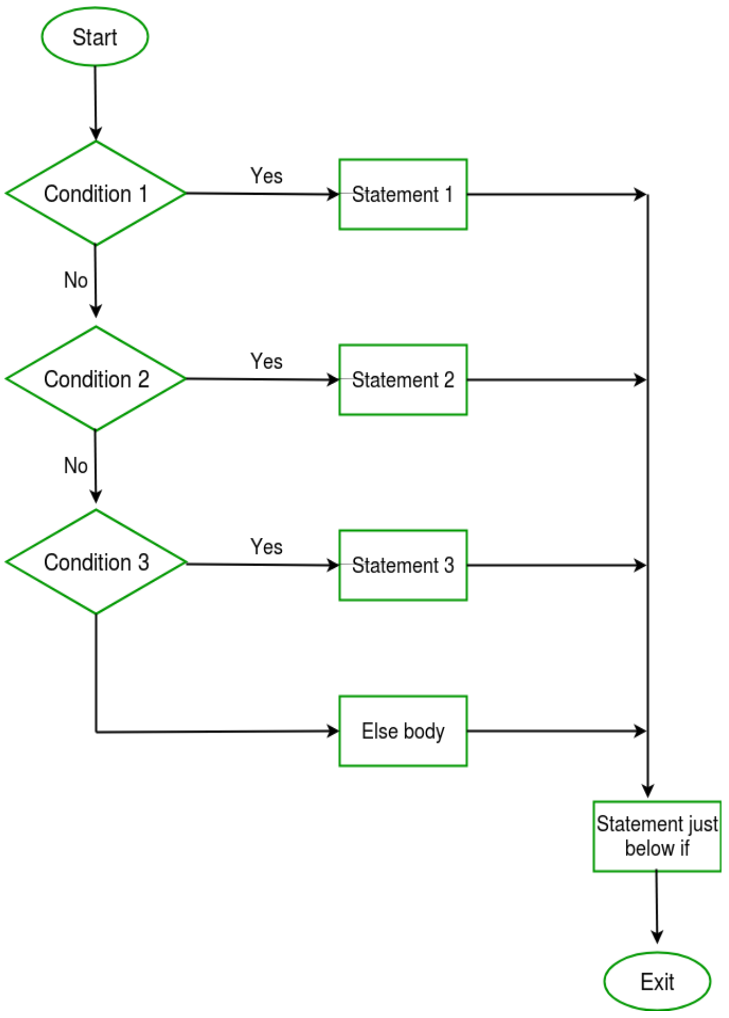
Program:

```
#include<stdio.h>
void main()
{
int num1, num2;
printf("Please Enter Two different values\n");
scanf("%d %d", &num1, &num2);
if(num1 > num2)
{
printf("%d is Largest\n", num1);
}
else
{
printf("%d is Largest\n", num2);
}
}
```

3.The else If Ladder:

In this statement, the execution of an array of instructions occurs only when the available condition is correct. The verification of the next condition occurs when this first condition is incorrect. In case all of the specifications fail even after the verification, then there will be an execution of the default block statements. The remainder of the program's ladder is shown below.

The syntax :	Flowchart:	Program:

If (condition 1)
{
Statement 1;
}
else if (condition 2)
{
Statement 2;
}
else if (condition 3)
{
Statement 3 ;
}
…………
…………
else
{
Default statement;
}
Next Statement (s);
	 Start
	Condition 1
	Yes
	Statement 1
	No
	Condition 2
	Yes
	Statement 2
	No
	Condition 3
	Yes
	Statement 3
	Else body
	Statement just below if
	Exit
		#include<stdio.h>
		void main()
		{
		int scores;
		float result;
		printf("enter your scores:");
		scanf("%d",&scores);
		if scores>=85:
		result='A+'
		else if scores>=65:
		result='B+'
		else if scores>=45:
		result='C+'
		else:
		result="FAIL"
		print("Result: ",result)

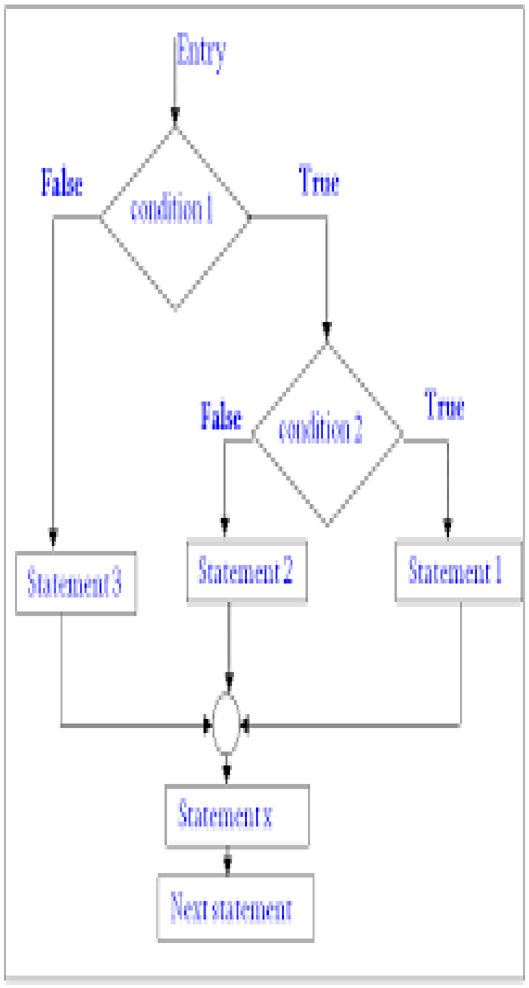
4.nested if statement:

nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

The syntax :

```
if (condition 1)
{
If (condition 2)
{
Statement 1 (s1);
}
else
{
Statement 2 (s2);
}
}
else
{
Statement 3
}

Next statement;
```

Flowchart:



Program:

```
#include <stdio.h>

int main() {

  double n1, n2, n3;

  printf("Enter three numbers: ");
  scanf("%lf %lf %lf", &n1, &n2, &n3);

  // outer if statement
  if (n1 >= n2) {

    // inner if...else
    if (n1 >= n3)
      printf("%.lf is the largest number.",
n1);
    else
      printf("%.lf is the largest number.",
n3);
  }

  // outer else statement
  else {

    // inner if...else
    if (n2 >= n3)
      printf("%.lf is the largest number.",
n2);
    else
      printf("%.lf is the largest number.",
n3);
  }

  return 0;
}
```

5.The Switch Statements:

A switch statement is a multi-way type of selection statement . The switch declaration comes into play when more than three alternatives (conditions) exist in a program. This command then switches between all the available blocks on the basis of the expression value. Then, each block has a corresponding value with it.

Every block is shown here with the use of the case keyword. As a matter of fact, the case keyword is also followed by the block label. Note that the break statement and default block statement are very optional in the case of the switch statement.

Syntax:

Switch (expression_A)

{

 case_A:

Statement 1;

break;

 case_B:

Statement 2 ;

break;

case_C;

Statement 3 ;

break;

….

 case_Z:

Statement N;

break;

Default:

   Default Statement;

break;

}

Next statement;

Flowchart:

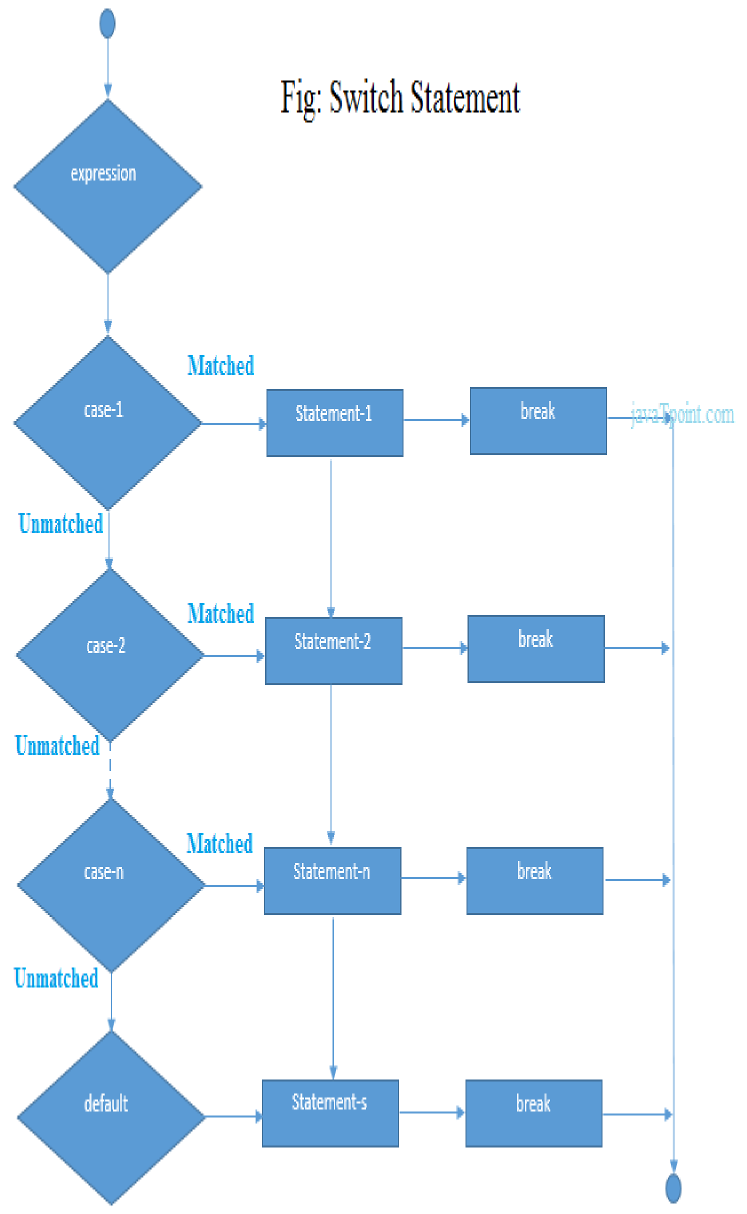


Fig: Switch Statement

Program:

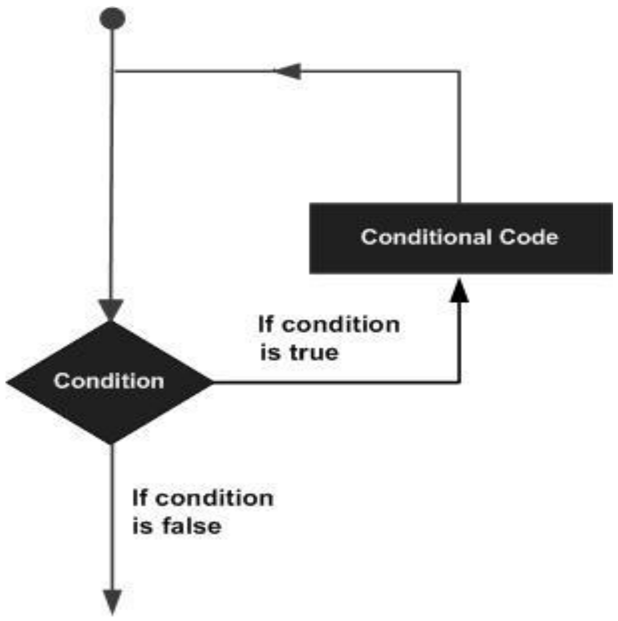
```
/ Program to create a simple
calculator
#include <stdio.h>

int main() {
    char operation;
    double n1, n2;

 printf("Enter (+, -, *, /):");
 scanf("%c", &operation);
 printf("Enter two operands:");
 scanf("%lf %lf",&n1, &n2);

switch(operation)
{
 case '+':
 printf("%.1lf + %.1lf =
%.1lf",n1, n2, n1+n2);
           break;

case '-':
 printf("%.1lf - %.1lf =
%.1lf",n1, n2, n1-n2);
           break;

case '*':
printf("%.1lf * %.1lf =
%.1lf",n1, n2, n1*n2);
      break;

 case '/':
 printf("%.1lf / %.1lf =
%.1lf",n1, n2, n1/n2);
break;

 // operator doesn't match any
case constant +, -, *, /
default:
   printf("Error! operator is
not correct");
    }
return 0;
}
```

Looping Statements:

when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −



C programming language provides the following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	**while loop** Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	**for loop** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**do...while loop** It is more like a while statement, except that it tests the condition at the end of the loop body.
4	**nested loops** You can use one or more loops inside any other while, for, or do..while loop.

While Loop:

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

Syntax:

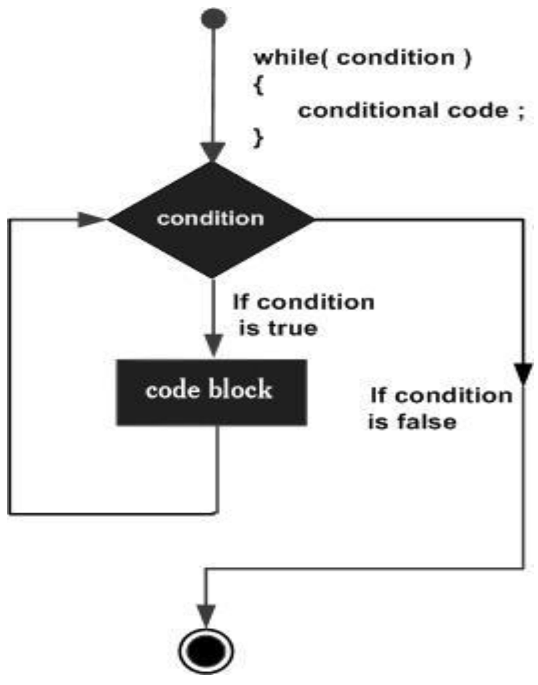
The syntax of a **while** loop in C programming language is −

```
while(condition)
{
  statement(s);
}

Next statement;
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

The syntax of a **while** loop in C programming language is −	**Flow Diagram**	**Program:**

Initialization;
 	
while(condition)
{
  statement(s);
}
 	
Next statement;
	 while(condition)
	{
	conditional code ;
	}
	condition
	If condition is true
	code block
	If condition is false
		`#include <stdio.h>`
		
		`int main () {`
		
		`  /* local variable definition */`
		`  int a = 10;`
		
		`  /* while loop execution */`
		`  while( a < 20 ) {`
		`    printf("value of a: %d\n", a);`
		`    a++;`
		`  }`
		
		`  return 0;`
		`}`

Note : while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

do...while loop in C:

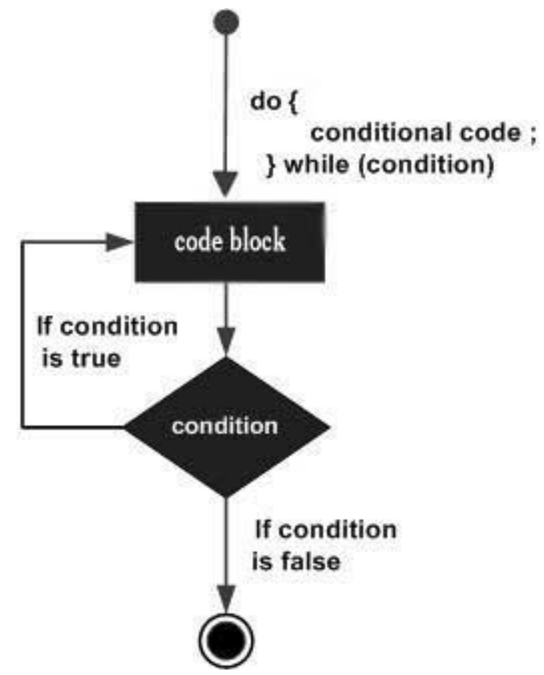
Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

The syntax of a **do...while** loop in C programming language is −

```
Initialization;
do {
   statement(s);
} while( condition );
Next statements;
```

Flow Diagram



Program:

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do {
      printf("value of a:
%d\n", a);
      a = a + 1;
   }while( a < 20 );

   return 0;
}
```

for loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

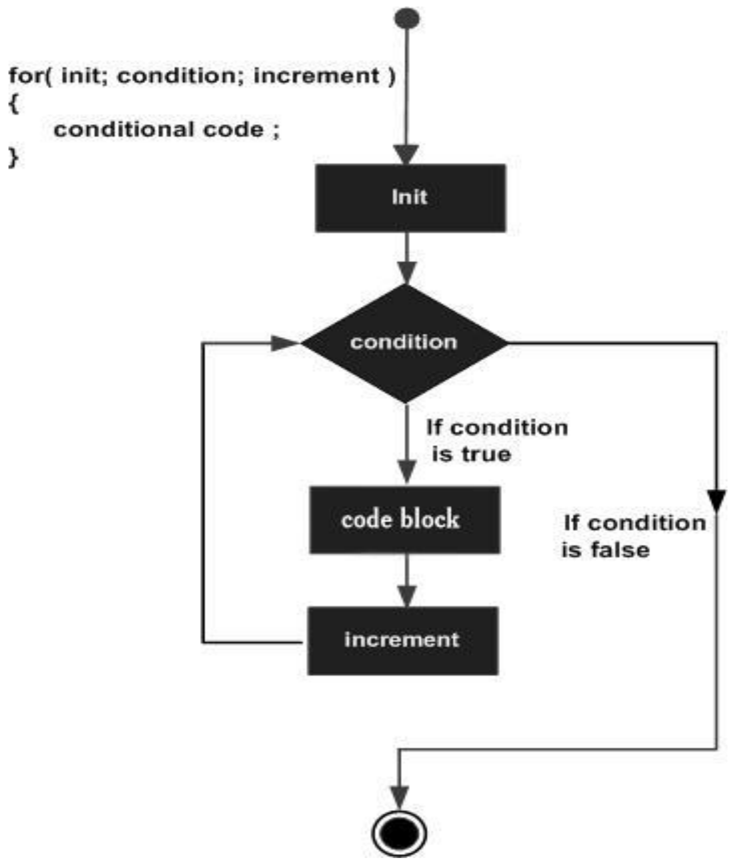
Here is the flow of control in a 'for' loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
- After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

The syntax of a **for** loop in C programming language is −

```
for ( init; condition; increment )
{
   statement(s);
}
```
Next statement;

Flow Diagram



Program:

```
#include <stdio.h>

int main () {

   int a;

   /* for loop execution */
   for( a = 10; a < 20; a = a + 1 ){
      printf("value of a: %d\n", a);
   }

   return 0;
}
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

Sr.No	Control Statement & Description
1	break statement Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	goto statement Transfers control to the labeled statement.

break statement:

The **break** statement in C programming has the following two usages −

- When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

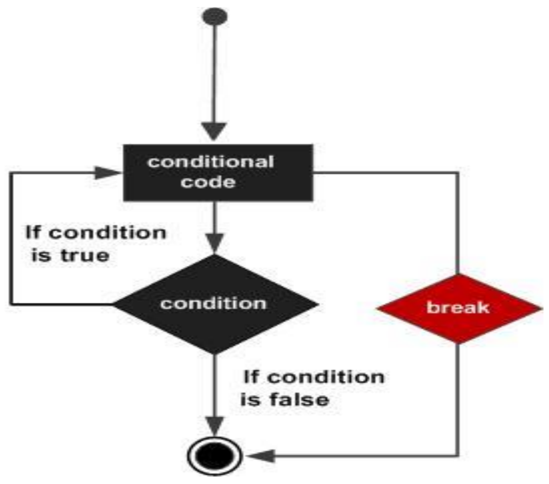
If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in C is as follows −

break;

Flow Diagram



Example

```
#include <stdio.h>

int main () {

  /* local variable definition */
  int a = 10;
  while( a < 20 ) {

    printf("value of a: %d\n", a);
    a++;

    if( a > 15) {
      /* terminate the loop using break statement */
      break;
    }
  }

  return 0;
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

continue statement:

The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

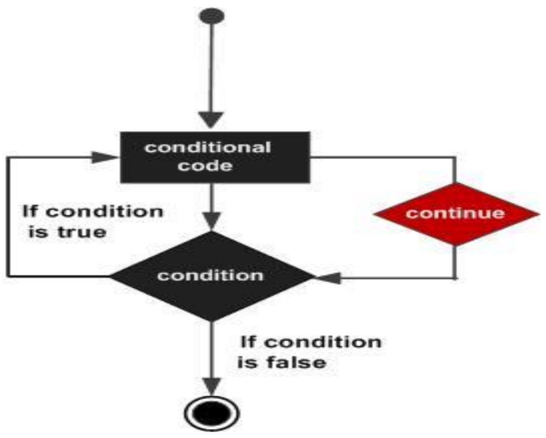
For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests.

Syntax

The syntax for a **continue** statement in C is as follows −

continue;

Flow Diagram



Example

```
#include <stdio.h>

int main () {
  int a = 10;

  /* do loop execution */
  do {

    if( a == 15) {
      /* skip the iteration */
      a = a + 1;
      continue;
    }

    printf("value of a: %d\n", a);
    a++;

  } while( a < 20 );

  return 0;
}
```

Output

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

When the above code is compiled and executed, it produces the following result −

goto statement:

A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

NOTE − Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

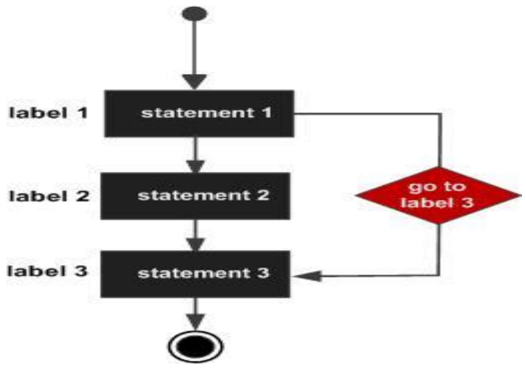
Syntax

The syntax for a **goto** statement in C is as follows −

```
goto label;
..
.
label: statement;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

Flow Diagram



Example

```
#include <stdio.h>

int main () {

  /* local variable definition */
  int a = 10;

  /* do loop execution */
  LOOP:do {

    if( a == 15) {
      /* skip the iteration */
      a = a + 1;
      goto LOOP;
    }

    printf("value of a: %d\n", a);
    a++;
```

```
Output:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
```

```
  }while( a < 20 );

  return 0;
}
```

C Array

- An array is 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. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.
- C array is beneficial if you have to store similar elements. 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 the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations.
- By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array.

Properties of Array:

The array contains the following properties.

- Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- 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.

Advantage of C Array:

1) Code Optimization: Less code to the access the data.

2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.

3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of C Array:

1) Fixed Size: Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Types of Arrays:

Arrays can be classified into types:

1.One dimensional array

2.Two dimensional array

One dimensional array:

The one-dimensional array can be defined as a sequential collection of elements. The 1D array is also called as linear array.

Declaration of C Array

We can declare an array in the c language in the following way.

```
data_type array_name[array_size];
```

Now, let us see the example to declare the array.

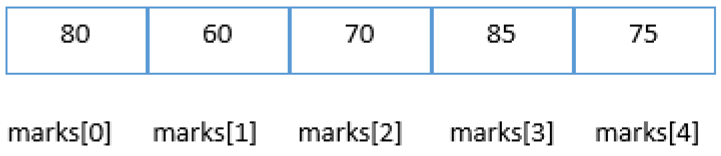
```
int marks[5];
```

Here, int is the *data_type*, marks are the *array_name*, and 5 is the *array_size*.

Initialization of Array

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index. Consider the following example.

```
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
```



Initialization of Array

```
#include<stdio.h>
void main ()
{
    int arr[100],i,n,largest,sec_largest;
    printf("Enter the size of the array?");
    scanf("%d",&n);
    printf("Enter the elements of the array?");
    for(i = 0; i<n; i++)
    {
        scanf("%d",&arr[i]);
    }
    largest = arr[0];
    sec_largest = arr[1];
    for(i=0;i<n;i++)
    {
        if(arr[i]>largest)
        {
            sec_largest = largest;
            largest = arr[i];
        }
        else if (arr[i]>sec_largest && arr[i]!=largest)
        {
            sec_largest=arr[i];
        }
    }
    printf("largest = %d, second largest = %d",largest,sec_largest);
  }
```

Declaration with Initialization

We can initialize the c array at the time of declaration. Let's see the code.

```
int marks[5]={20,30,40,50,60};
```

In such case, there is **no requirement to define the size**. So it may also be written as the following code.

```
int marks[]={20,30,40,50,60};
```

Let's see the C program to declare and initialize the array in C

```
#include<stdio.h>
int main(){
int i=0;
int marks[5]={20,30,40,50,60};//declaration and initialization of array
 //traversal of array
for(i=0;i<5;i++){
printf("%d \n",marks[i]);
}
return 0;
}
```

Two Dimensional Array:

The two-dimensional 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 lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

Declaration of two dimensional Array :

The syntax to declare the 2D array is given below.

```
data_type array_name[rows][columns];
```

Consider the following example.

```
int twodimen[4][3];
```

Here, 4 is the number of rows, and 3 is the number of columns.

Initialization of 2D Array in C:

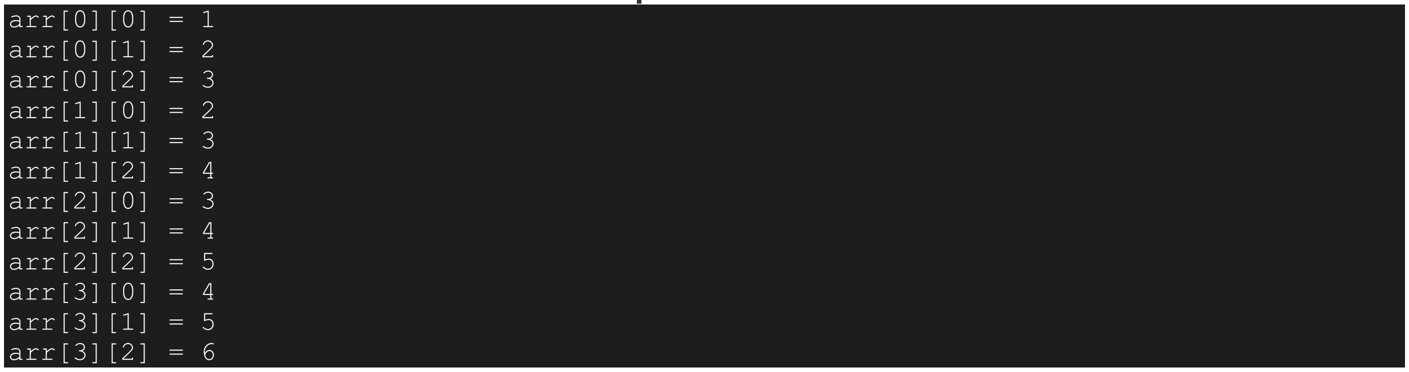
In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays. We will have to define at least the second dimension of the array. The two-dimensional array can be declared and defined in the following way.

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

Two-dimensional array example in C

```
#include<stdio.h>
int main(){
int i=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
//traversing 2D array
for(i=0;i<4;i++){
 for(j=0;j<3;j++){
   printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
 }//end of j
}//end of i
return 0;
}
```

Output



```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```

UNIT-2

UNIT-II

Functions

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as ***procedure* or *subroutine*** in other programming languages.

Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

SNo	C function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)

3	Function definition	return_type function_name (argument list)
		{
		function body;
		}

Types of Functions

There are two types of functions in C programming:

1. **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.

Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

Example without return value

```
  void hello()
{
    printf("hello c");
}
```

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

Example with return value:

```
int get()
{
    return 10;
 }
```

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

```
float get(){
return 10.2;
}
```

Now, you need to call the function, to get the value of the function.

Different aspects of function calling/Types of user defined functions:

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

Example for Function without argument and return value

Example 1

```
#include<stdio.h>
void printName();
void main ()
{
    printf("Hello ");
    printName();
}
void printName()
{
    printf("welcome");
}
```

Output

```
Hello welcome
```

Example 2

```
#include<stdio.h>
void sum();
void main()
{
    printf("\nGoing to calculate the sum of two numbers:");
    sum();
}
void sum()
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    printf("The sum is %d",a+b);
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example for Function without argument and with return value

Example 1

```
#include<stdio.h>
int sum();
void main()
{
    int result;
    printf("\nGoing to calculate the sum of two numbers:");
    result = sum();
    printf("%d",result);
}
int sum()
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    return a+b;
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example 2: program to calculate the area of the square

```
#include<stdio.h>
int sum();
void main()
{
```

```c
    printf("Going to calculate the area of the square\n");
    float area = square();
    printf("The area of the square: %f\n",area);
}
int square()
{
    float side;
    printf("Enter the length of the side in meters: ");
    scanf("%f",&side);
    return side * side;
}
```

Output

```
Going to calculate the area of the square
Enter the length of the side in meters: 10
The area of the square: 100.000000
```

Example for Function with argument and without return value

Example 1:

```c
#include<stdio.h>
void sum(int, int);
void main()
{
    int a,b,result;
    printf("\nGoing to calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    sum(a,b);
}
void sum(int a, int b)
{
    printf("\nThe sum is %d",a+b);
}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example 2: program to calculate the average of five numbers

```
#include<stdio.h>
void average(int, int, int, int, int);
void main()
{
   int a,b,c,d,e;
   printf("\nGoing to calculate the average of five numbers:");
   printf("\nEnter five numbers:");
   scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
   average(a,b,c,d,e);
}
void average(int a, int b, int c, int d, int e)
{
   float avg;
   avg = (a+b+c+d+e)/5;
   printf("The average of given five numbers : %f",avg);
}
```

Output

```
Going to calculate the average of five numbers:
Enter five numbers:10
20
30
40
50
The average of given five numbers : 30.000000
```

Example for Function with argument and with return value

Example:

```
#include<stdio.h>
int sum(int, int);
void main()
{
   int a,b,result;
   printf("\nGoing to calculate the sum of two numbers:");
   printf("\nEnter two numbers:");
   scanf("%d %d",&a,&b);
   result = sum(a,b);
   printf("\nThe sum is : %d",result);
}
int sum(int a, int b)
{
   return a+b;
}
```

Output

```
Going to calculate the sum of two numbers:
Enter two numbers:10
20
The sum is : 30
```

Example 2: Program to check whether a number is even or odd

```c
#include<stdio.h>
int even_odd(int);
void main()
{
 int n,flag=0;
 printf("\nGoing to check whether a number is even or odd");
 printf("\nEnter the number: ");
 scanf("%d",&n);
 flag = even_odd(n);
 if(flag == 0)
 {
    printf("\nThe number is odd");
 }
 else
 {
    printf("\nThe number is even");
 }
}
int even_odd(int n)
{
    if(n%2 == 0)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Output

```
Going to check whether a number is even or odd
Enter the number: 100
The number is even
```

Library Functions

Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension **.h**.

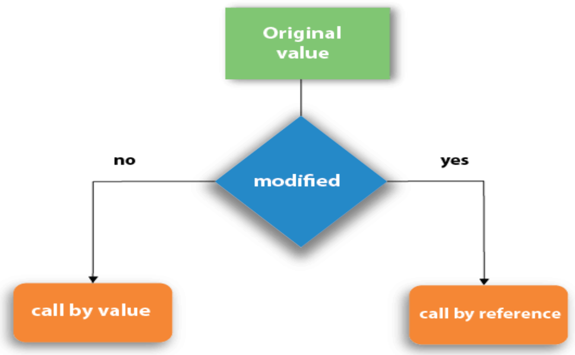
We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include stdio.h in our program which is a header file that contains all the library functions regarding standard input/output.

The list of mostly used header files is given in the following table.

S.No	Header file	Description
1	stdio.h	This is a standard input/output header file. It contains all the library functio regarding standard input/output.
2	conio.h	This is a console input/output header file.
3	string.h	It contains all string related library functions like gets(), puts(),etc.
4	stdlib.h	This header file contains all the general library functions like malloc calloc(), exit(), etc.
5	math.h	This header file contains all the math operations related functions like sqrt pow(), etc.
6	time.h	This header file contains all the time-related functions.
7	ctype.h	This header file contains all character handling functions.
8	stdarg.h	Variable argument functions are defined in this header file.
9	signal.h	All the signal handling functions are defined in this header file.
10	setjmp.h	This file contains all the jump functions.
11	locale.h	This file contains locale functions.
12	errno.h	This file contains error handling functions.
13	assert.h	This file contains diagnostics functions.

Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.



Let's understand call by value and call by reference in c language one by one.

Call by value in C:

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

```
#include<stdio.h>
void change(int num) {
   printf("Before adding value inside function num=%d \n",num);
   num=num+100;
   printf("After adding value inside function num=%d \n", num);
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(x);//passing value in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by Value Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int , int); //prototype of the function
```

```
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
   swap(a,b);
   printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in call by value, a = 10, b = 20
}
void swap (int a, int b)
{
   int temp;
   temp = a;
   a=b;
   b=temp;
   printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10
}
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```
#include<stdio.h>
void change(int *num) {
   printf("Before adding value inside function num=%d \n",*num);
   (*num) += 100;
   printf("After adding value inside function num=%d \n", *num);
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(&x);//passing reference in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

Call by reference Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
   swap(&a,&b);
   printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in c
all by reference, a = 10, b = 20
}
void swap (int *a, int *b)
{
   int temp;
   temp = *a;
   *a=*b;
   *b=temp;
   printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 20, b = 10
}
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Difference between call by value and call by reference in c

S.No	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

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.

```
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{
    if (n==0)
    {
        return 0;
    }
    else if ( n == 1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}
```

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:

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.

```
if (test_for_base)
{
    return some_value;
}
else if (test_for_another_base)
{
    return some_another_value;
}
else
{
    // Statements;
    recursive call;
}
```

Example of recursion in C

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

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
    int n,f;
    printf("Enter the value of n?");
    scanf("%d",&n);
    f = fibonacci(n);
    printf("%d",f);
}
int fibonacci (int n)
{
    if (n==0)
    {
    return 0;
    }
    else if (n == 1)
    {
       return 1;
    }
    else
    {          return fibonacci(n-1)+fibonacci(n-2);
    }
}
```

Output

```
Enter the value of n?12
144
```

Memory allocation of Recursive method:

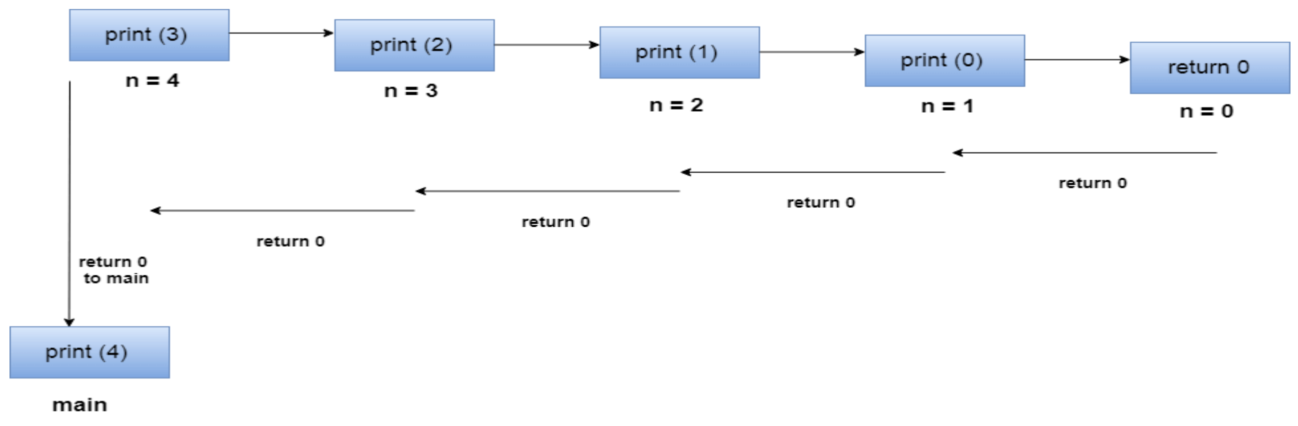
Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. 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.

```
int display (int n)
{
    if(n == 0)
        return 0; // terminating condition
    else
    {
        printf("%d",n);
        return display(n-1); // recursive call
    }
}
```

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

Storage Classes in C

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

- Automatic
- External
- Static
- Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
Auto	RAM	Garbage Value	Local	Within function
Extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
Static	RAM	Zero	Local/Global	Till the end of the main program, Retains value between multiple functions call

Register	Register	Garbage Value	Local	Within the function

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.

The scope of the automatic variables is limited to the block in which they are defined.

- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

Example 1

```
#include <stdio.h>
int main()
{
int a; //auto
char b;
float c;
printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
return 0;
}
```

Output:

```
garbage garbage garbage
```

Example 2

```
#include <stdio.h>
int main()
{
int a = 10,i;
printf("%d ",++a);
{
int a = 20;
for (i=0;i<3;i++)
{
printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
}
}
printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
}
```

Output:

```
11 20 20 20 11
```

Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

Example 1

```
#include<stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main ()
{
printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printed.
}
```

Output:

```
0 0 0.000000 (null)
```

Example 2

```
#include<stdio.h>
void sum()
{
static int a = 10;
static int b = 24;
printf("%d %d \n",a,b);
a++;
b++;
}
void main()
{
int i;
for(i = 0; i< 3; i++)
{
sum(); // The static variables holds their value between multiple function calls.
}
}
```

Output:

```
10 24
11 25
```

```
12 26
```

Register

- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables, i.e., we can not use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0.
- The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler?s choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Example 1

```c
#include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.
printf("%d",a);
}
```

Output:

```
0
```

Example 2

```c
#include <stdio.h>
int main()
{
register int a = 0;
printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.
}
```

Output:

```
main.c:5:5: error: address of register variable ?a? requested
printf("%u",&a);
^~~~~~
```

External

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.

- We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

```
#include <stdio.h>
int main()
{
extern int a;
printf("%d",a);
}
```

Output

```
main.c:(.text+0x6): undefined reference to `a'
collect2: error: ld returned 1 exit status
```

Example 2

```
#include <stdio.h>
int a;
int main()
{
extern int a; // variable a is defined globally, the memory will not be allocated to a
printf("%d",a);
}
```

Output

```
0
```

Example 3

```
#include <stdio.h>
int a;
int main()
{
extern int a = 0; // this will show a compiler error since we can not use extern and initializer at same time
printf("%d",a);
}
```

Output

```
compile time error
main.c: In function ?main?:
main.c:5:16: error: ?a? has both ?extern? and initializer
extern int a = 0;
```

Example 4

```
#include <stdio.h>
int main()
{
extern int a; // Compiler will search here for a variable a defined and initialized somewhere in the pogram or not.
printf("%d",a);
}
int a = 20;
```

Output

```
20
```

Example 5

```
extern int a;
int a = 10;
#include <stdio.h>
int main()
{
printf("%d",a);
}
int a = 20; // compiler will show an error at this line
```

Output

```
compile time error
```

Pointers:

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer.

Consider the following example to define a pointer which stores the address of an integer.

```
int n = 10;
int* p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.
```

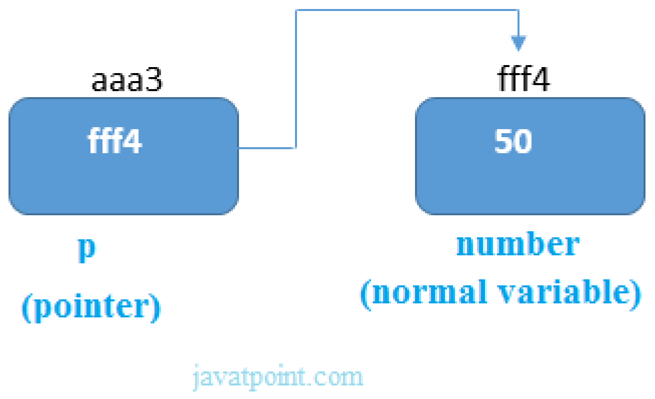
Declaring a pointer:

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

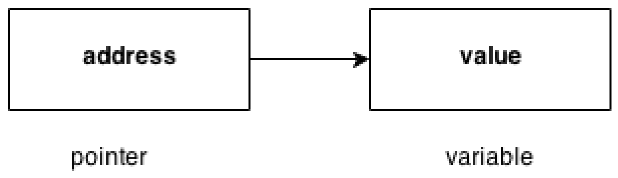
```
int *a;//pointer to int
char *c;//pointer to char
```

Pointer Example

An example of using pointers to print the address and value is given below.



68M



1.3K

As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

```
#include<stdio.h>
int main(){
int number=50;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the ad
dress of number.
printf("Value of p variable is %d \n",*p); /* As we know that * is used to dereference a pointer therefore if we
 print  *p, we will get the value stored at the address contained by p.   */
return 0;
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can **return multiple values from a function** using the pointer.

3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation:

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures:

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator:

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
#include<stdio.h>
int main(){
int number=50;
printf("value of number is %d, address of number is %u",number,&number);
return 0;
}
```

Output

```
value of number is 50, address of number is fff4
```

<u>Memory Allocation in C Programs</u>

The C language supports two kinds of memory allocation through the variables in C programs:

- ***Static allocation*** is what happens when you declare a static or global variable. Each static or global variable defines one block of space, of a fixed size. The space is allocated once, when your program is started (part of the exec operation), and is never freed.
- ***Automatic allocation*** happens when you declare an automatic variable, such as a function argument or a local variable. The space for an automatic variable is allocated when the compound statement containing the declaration is entered, and is freed when that compound statement is exited.

In GNU C, the size of the automatic storage can be an expression that varies. In other C implementations, it must be a constant.

A third important kind of memory allocation, *dynamic allocation*, is not supported by C variables but is available via GNU C Library functions.

- Dynamic Memory Allocation

Dynamic memory allocation is a technique in which programs determine as they are running where to store some information. You need dynamic allocation when the amount of memory you need, or how long you continue to need it, depends on factors that are not known before the program runs.

For example, you may need a block to store a line read from an input file; since there is no limit to how long a line can be, you must allocate the memory dynamically and make it dynamically larger as you read more of the line.

Dynamic memory allocation in C:

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1.malloc()
2.calloc()
3.realloc()
4.free()

Let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for **dynamic memory** allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.

realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C:

The malloc() function allocates single block of requested memory.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

```
ptr=(cast-type*)malloc(byte-size)
```

Let's see the example of malloc() function.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
   if(ptr==NULL)
   {
       printf("Sorry! unable to allocate memory");
       exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
       scanf("%d",ptr+i);
       sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
return 0;
}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
```

```
Sum=30
```

calloc() function in C:

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

```
ptr=(cast-type*)calloc(number, byte-size)
```

Let's see the example of calloc() function.

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

realloc() function in C:

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

1.ptr=realloc(ptr, **new**-size)

free() function in C:

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

1. free(ptr)

Reading complex pointers:

There are several things which must be taken into the consideration while reading the complex pointers in C. Lets see the precedence and associativity of the operators which are used regarding pointers.

Operator	Precedence	Associativity
(), []	1	Left to right
*, identifier	2	Right to left
Data type	3	-

Here,we must notice that,

- (): This operator is a bracket operator used to declare and define the function.
- []: This operator is an array subscript operator
- * : This operator is a pointer operator.
- Identifier: It is the name of the pointer. The priority will always be assigned to this.
- Data type: Data type is the type of the variable to which the pointer is intended to point. It also includes the modifier like signed int, long, etc).

C Function Pointer:

As we know that we can create a pointer of any data type such as int, char, float, we can also create a pointer pointing to a function. The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

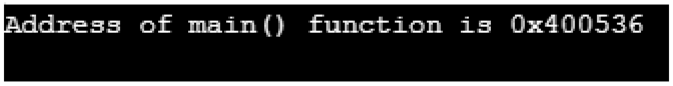
Let's see a simple example.

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

```
int main()
{
   printf("Address of main() function is %p",main);
   return 0;
}
```

The above code prints the address of **main()** function.

Output



```
Address of main() function is 0x400536
```

In the above output, we observe that the main() function has some address. Therefore, we conclude that every function has some address.

Declaration of a function pointer:

Till now, we have seen that the functions have addresses, so we can create pointers that can contain these addresses, and hence can point them.

Syntax of function pointer

```
return type (*ptr_name)(type1, type2…);
```

For example:

```
int (*ip) (int);
```

In the above declaration, *ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

```
float (*fp) (float);
```

In the above declaration, ***fp** is a pointer that points to a function that returns a float value and accepts a float value as an argument.

We can observe that the declaration of a function is similar to the declaration of a function pointer except that the pointer is preceded by a **'*'**. So, in the above declaration, fp is declared as a function rather than a pointer.

Till now, we have learnt how to declare the function pointer. Our next step is to assign the address of a function to the function pointer.

```
float (*fp) (int , int);    // Declaration of a function pointer.
float func( int , int );    // Declaration of  function.
fp = func;                  // Assigning address of func to the fp pointer.
```

In the above declaration, **'fp'** pointer contains the address of the **'func'** function.

Note: Declaration of a function is necessary before assigning the address of a function to the function pointer.

Calling a function through a function pointer:

We already know how to call a function in the usual way. Now, we will see how to call a function using a function pointer.

Suppose we declare a function as given below:

```
float func(int , int);      // Declaration of a function.
```

Calling an above function using a usual way is given below:

```
result = func(a , b);     // Calling a function using usual ways.
```

Calling a function using a function pointer is given below:

```
result = (*fp)( a , b);    // Calling a function using function pointer.
```

Or

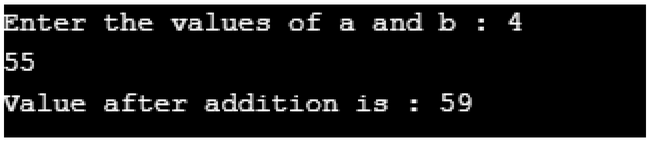
```
result = fp(a , b);         // Calling a function using function pointer, and indirection operator can be removed.
```

The effect of calling a function by its name or function pointer is the same. If we are using the function pointer, we can omit the indirection operator as we did in the second case. Still, we use the indirection operator as it makes it clear to the user that we are using a function pointer.

Let's understand the function pointer through an example

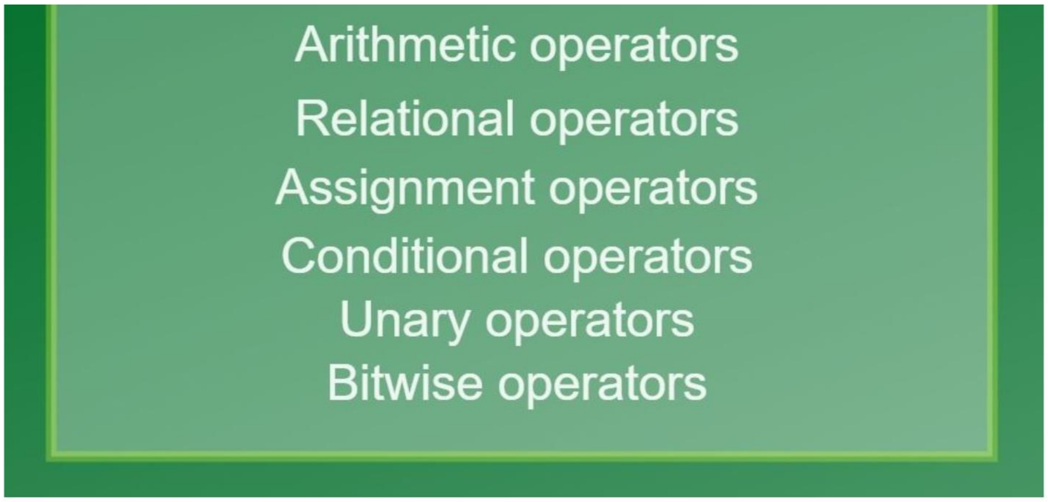
```
#include <stdio.h>
int add(int,int);
int main()
{
  int a,b;
  int (*ip)(int,int);
  int result;
  printf("Enter the values of a and b : ");
  scanf("%d %d",&a,&b);
  ip=add;
  result=(*ip)(a,b);
  printf("Value after addition is : %d",result);
   return 0;
}
int add(int a,int b)
{
   int c=a+b;
   return c;
}
```

Output



```
Enter the values of a and b : 4
55
Value after addition is : 59
```

Pointer Expressions


Arithmetic operators
Relational operators
Assignment operators
Conditional operators
Unary operators
Bitwise operators

Pointer Expressions in C

Arithmetic Operators:
We can perform arithmetic operations to pointer variables using arithmetic operators. We can add an integer or subtract an integer using a pointer pointing to that integer variable. The given table shows the arithmetic operators that can be performed on pointer variables:

Examples:
*ptr1 + *ptr2

*ptr1 * *ptr2

*ptr1 + *ptr2 - *ptr3

We can also directly perform arithmetic expression using integers. Lets look at the example given below where p1 and p2 are pointers.

p1+10, p2-5, p1-p2+10, p1/2
Below diagram represents how exactly the expression/operators work with pointers.

Let us understand pointer arithmetic expression better with given code:

```
#include <stdio.h>
int main()
```

```
{
   // Integer variables
   int a = 20, b = 10;

   // Variables for storing arithmetic
   // operations solution
   int add, sub, div, mul, mod;

   // Pointer variables for variables
   // a and b
   int *ptr_a, *ptr_b;

   // Initialization of pointers
   ptr_a = &a;
   ptr_b = &b;

   // Performing arithmetic Operations
   // on pointers
   add = *ptr_a + *ptr_b;
   sub = *ptr_a - *ptr_b;
   mul = *ptr_a * *ptr_b;
   div = *ptr_a / *ptr_b;
   mod = *ptr_a % *ptr_b;

   // Printing values
   printf("Addition = %d\n", add);
   printf("Subtraction = %d\n", sub);
   printf("Multiplication = %d\n", mul);
   printf("Division = %d\n", div);
   printf("Modulo = %d\n", mod);
   return 0;
}
```

Output:

```
Addition = 30

Subtraction = 10

Multiplication = 200

Division = 2

Modulo = 0
```

Note: While performing division, make sure you put a blank space between '/' and '*' of the pointer as together it would make a multi-line comment('/*').

Example:

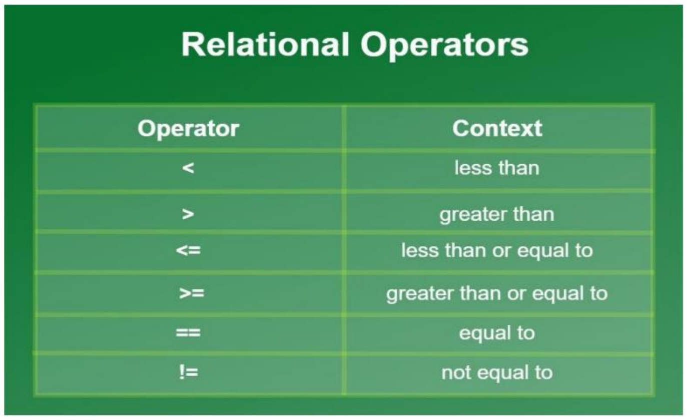
Incorrect: *ptr_a/*ptr_b;

Correct: *ptr_a / *ptr_b;

Correct: (*ptr_a)/(*ptr_b);

Relational Operators:

Relational operations are often used to compare the values of the variable based on which we can take decisions. The given table shows the relational operators that can be performed on pointer variables.



Relational Operators

Operator	Context
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

Relational Operations on Pointers in C

Example:

*ptr1 > *ptr2

*ptr1 < *ptr2

The value of the relational expression is either 0 or 1 that is false or true. The expression will return value 1 if the expression is true and it'll return value 0 if false.

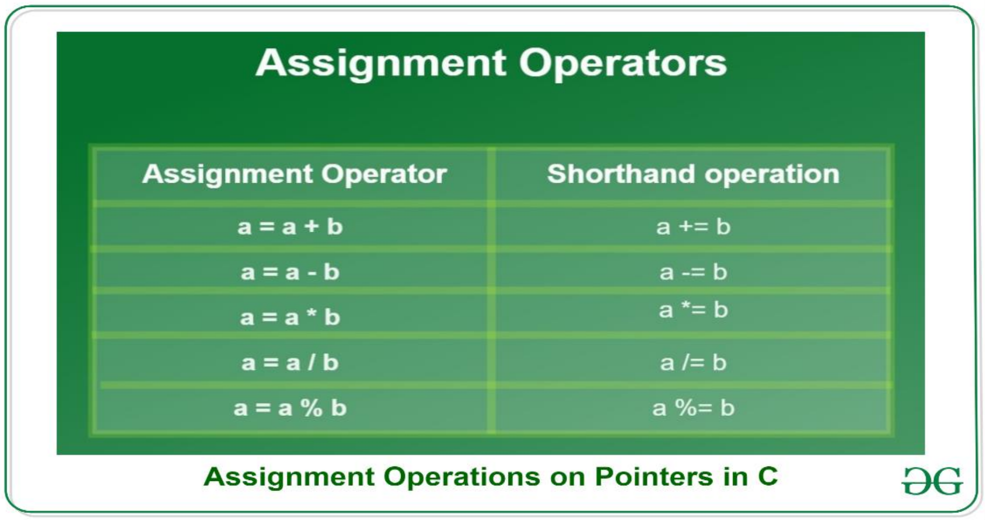
Let us understand relational expression on pointer better with the code given below:

```
// Program showing pointer expressions
// during Relational Operations
#include <stdio.h>
int main()
{
    // Initializing integer variables
    int a = 20, b = 10;
    // Declaring pointer variables
    int* ptr_a;
    int* ptr_b;
    // Initializing pointer variables
    ptr_a = &a;
    ptr_b = &b;
    // Performing relational operations
    // less than operator
    if (*ptr_a < *ptr_b)
    {
        printf( "%d is less than %d.", *ptr_a, *ptr_b);
    }
    // Greater than operator
    if (*ptr_a > *ptr_b)
    {
        printf("%d is greater than %d.", *ptr_a, *ptr_b);
    }

    // Equal to
    if (*ptr_a == *ptr_b) {
        printf(
            "%d is equal to %d.", *ptr_a, *ptr_b);
    }
    return 0;
}
```

Output:
20 is greater than 10.

Output:
20 is greater than 10.

Assignment Operators:
Assignment operators are used to assign values to the identifiers. There are multiple shorthand operations available. A table is given below showing the actual assignment statement with its shorthand statement.



Assignment Operators

Assignment Operator	Shorthand operation
a = a + b	a += b
a = a - b	a -= b
a = a * b	a *= b
a = a / b	a /= b
a = a % b	a %= b

Assignment Operations on Pointers in C

Examples:
*a=10

*b+=20

*z=3.5

*s=4.56743

Let us understand assignment operator in better way with the help of code given below:

```
  // Program showing pointer expressions
  // during Assignment Operations
  #include <stdio.h>
  int main()
  {
      // Initializing integer variable
      int a = 30;

      // Declaring pointer variable
```

```
    int* ptr_a;

    // Initializing pointer using
    // assignment operator
    ptr_a = &a;

    // Changing the variable's value using
    // assignment operator
    *ptr_a = 50;

    // Printing value of 'a' after
    // updating its value
    printf("Value of variable a = %d", *ptr_a);

    return 0;
}
```

Output:

```
Value of variable a = 50
```

Conditional Operators

There is only one mostly used conditional operator in C known as Ternary operator. Ternary operator first checks the expression and depending on its return value returns true or false, which triggers/selects another expression.

Syntax:

expression1 ? expression2 : expression3;

Example:

c = (*ptr1 > *ptr2) ? *ptr1 : *ptr2;

- As shown in example, assuming *ptr1=20 and *ptr2=10 then the condition here becomes true for the expression, so it'll return value of true expression i.e. *ptr1, so variable 'c' will now contain value of 20.
- Considering same example, assume *ptr1=30 and *ptr2=50 then the condition is false for the expression, so it'll return value of false expression i.e. *ptr2, so variable 'c' will now contain value 50.

Let us understand the concept through the given code:

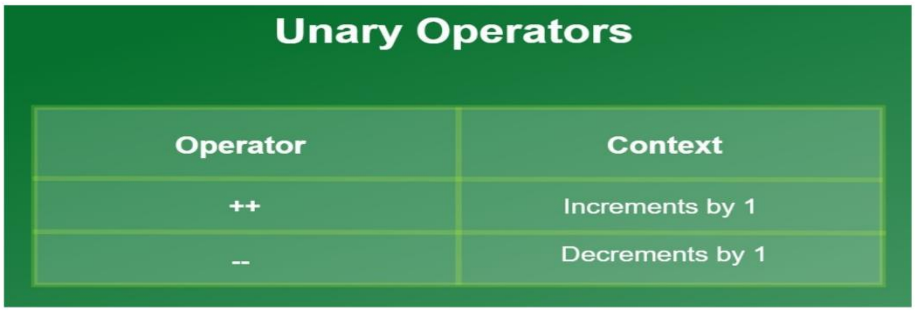
```
// Program showing pointer expressions
// during Conditional Operations
#include <stdio.h>
int main()
{
    // Initializing integer variables
    int a = 15, b = 20, result = 0;

    // Declaring pointer variables
    int *ptr_a, *ptr_b;

    // Initializing pointer variables
    ptr_a = &a;
    ptr_b = &b;

    // Performing ternary operator
    result = ((*ptr_a > *ptr_b) ? *ptr_a : *ptr_b);

    // Printing result of ternary operator
    printf("%d is the greatest.", result);
    return 0;
}
```

Output:

```
20 is the greatest.
```

Unary Operators:

There are mainly two operators which are given as follows.



Unary Operators

Operator	Context
++	Increments by 1
--	Decrements by 1

Unary Operations on Pointers in C

Examples:

(*ptr1)++

(*ptr1)--

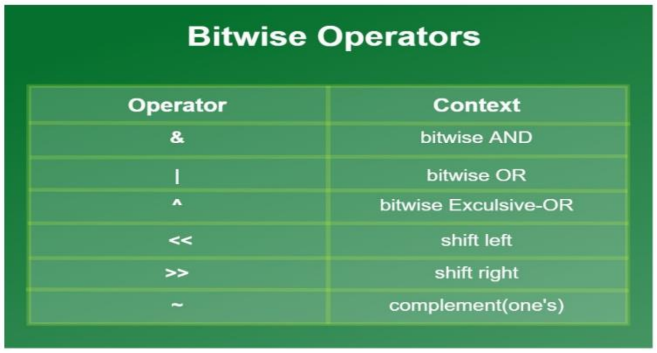
Let us understand the use of the unary operator through the given code:

```
// Program showing pointer expressions
// during Unary Operations
#include <stdio.h>
int main()
{
    // Initializing integer variable
    int a = 34;

    // Declaring pointer variable
    int* ptr_a;

    // Initializing pointer variable
    ptr_a = &a;

    // Value of a before increment
    printf("Increment:\n");
    printf(
        "Before increment a = %d\n", *ptr_a);

    // Unary increment operation
    (*ptr_a)++;

    // Value of a after increment
    printf(
        "After increment a = %d", *ptr_a);

    // Value before decrement
    printf("\n\nDecrement:\n");
    printf(
        "Before decrement a = %d\n", *ptr_a);

    // unary decrement operation
    (*ptr_a)--;

    // Value after decrement
    printf("After decrement a=%d", *ptr_a);

    return 0;
}
```

Output:

```
Increment:

Before increment a = 34

After increment a = 35


Decrement:

Before decrement a = 35

After decrement a=34
```

Bitwise Operators:
Binary operators are also known as bitwise operators. It is used to manipulate data at bit level. Bitwise operators can’t be used for float and double datatype. A table is shown below with all bitwise operators:



Bitwise Operators

Operator	Context
&	bitwise AND
|	bitwise OR
^	bitwise Exculsive-OR
<<	shift left
>>	shift right
~	complement(one's)

Bitwise Operations on Pointers in C

Examples:

*ptr1 & *ptr2

*ptr1 | *ptr2

*ptr1 ^ *ptr2

Let us understand the concept through the given code:

```
// Program showing pointer expressions
// during Bitwise Operations
#include <stdio.h>
int main()
{
    // Declaring integer variable for
    // storing result
    int and, or, ex_or;

    // Initializing integer variable
    int a = 1, b = 2;

    // Performing bitwise operations
    // AND operation
    and = a & b;

    // OR operation
    or = a | b;

    // EX-OR operation
    ex_or = a ^ b;

    // Printing result of operations
    printf("\na AND b = %d", and);
    printf("\na OR b = %d", or);
    printf("\na Exclusive-OR b = %d", ex_or);
    return 0;
}
```

Output:

```
a AND b = 0

a OR b = 3
```

a Exclusive-OR b = 3

Structures

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

Defining a Structure:

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows −

```
struct [structure tag] {

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Accessing Structure Members:

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program −

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

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Union

A **union** is a user defined data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union:

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union .

```

#include <stdio.h>
#include <string.h>
 union Data {
   int i;
   float f;
   char str[20];
};
```

```
 int main( )
 {
  union Data data;
printf( "Memory size occupied by data : %d\n", sizeof(data));
 return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Memory size occupied by data : 20
```

Accessing Union Members:

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program −

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

union collecton {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union collection data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

```
#include <stdio.h>
#include <string.h>
 union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

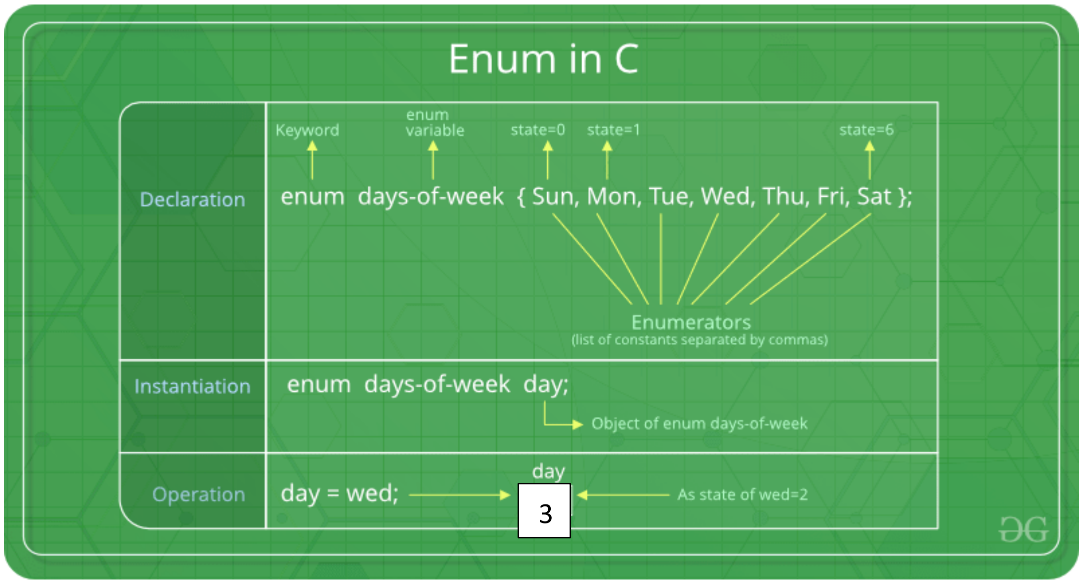
When the above code is compiled and executed, it produces the following result −

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

Enumeration

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.



enum State {Working = 1, Failed = 0};

The keyword ‘enum’ is used to declare new enumeration types in C and C++. Following is an example of enum declaration.

```
// An example program to demonstrate working of enum in C
#include<stdio.h>

enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};

int main()
{
    enum week day;
    day = Wed;
    printf("%d",day);
    return 0;
}
```

Output:

In the above example, we declared “day” as the variable and the value of “Wed” is allocated to day, which is 2. So as a result, 2 is printed.
Another example of enumeration is:

```
// Another example program to demonstrate working
// of enum in C
#include<stdio.h>

enum year{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
```

```
int main()
{
  int i;
  for (i=Jan; i<=Dec; i++)
    printf("%d ", i);

  return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

In this example, the for loop will run from i = 0 to i = 11, as initially the value of i is Jan which is 0 and the value of Dec is 11.

C Strings

- The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences.
- Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where the string ends.
- When we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.
- There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring **string by char array** in C language.

1. **char** ch[10]={'p', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};

As we know, array index starts from 0, so it will be represented as in the figure given below.

0	1	2	3	4	5	6	7
p	**r**	**o**	**g**	**r**	**a**	**m**	**\0**

While declaring string, size is not mandatory. So we can write the above code as given below:

1. **char** ch[]={'p', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};

We can also define the **string by the string literal** in C language. For example:

1. **char** ch[]="program";

In such case, '\0' will be appended at the end of the string by the compiler.

Difference between char array and string literal:

There are two main differences between char array and literal.

- We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.

- The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

String Example in C:

Let's see a simple example where a string is declared and being printed. The '%s' is used as a format specifier for the string in c language.

```
#include<stdio.h>
#include <string.h>
int main(){
  char ch[11]= {'p', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
   char ch2[11]="program";

   printf("Char Array Value is: %s\n", ch);
   printf("String Literal Value is: %s\n", ch2);
 return 0;
}
```

Output

```
Char Array Value is: program
String Literal Value is: program
```

Traversing String:

Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are two ways to traverse a string.

- By using the length of string
- By using the null character.

Let's discuss each one of them.

Using the length of string:

Let's see an example of counting the number of vowels in a string.

```
#include<stdio.h>
void main ()
{
    char s[11] = "javatpoint";
    int i = 0;
    int count = 0;
    while(i<11)
    {
        if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
        {
            count ++;
        }
        i++;
    }
    printf("The number of vowels %d",count);
}
```

Output

```
The number of vowels 4
```

Using the null character:

Let's see the same example of counting the number of vowels by using the null character.

```
#include<stdio.h>
void main ()
```

```
{
    char s[11] = "javatpoint";
    int i = 0;
    int count = 0;
    while(s[i] != NULL)
    {
        if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
        {
            count ++;
        }
        i++;
    }
    printf("The number of vowels %d",count);
}
```

Output

```
The number of vowels 4
```

Accepting string as the input:

Till now, we have used scanf to accept the input from the user. However, it can also be used in the case of strings but with a different scenario. Consider the below code which stores the string while space is encountered.

```
#include<stdio.h>
void main ()
{
    char s[20];
    printf("Enter the string?");
    scanf("%s",s);
    printf("You entered %s",s);
}
```

Output

```
Enter the string?programing language
You entered programing
```

It is clear from the output that, the above code will not work for space separated strings. To make this code working for the space separated strings, the minor changed required in the scanf function, i.e., instead of writing scanf("%s",s), we must write: scanf("%[^\n]s",s) which instructs the compiler to store the string s while the new line (\n) is encountered. Let's consider the following example to store the space-separated strings.

```
#include<stdio.h>
void main ()
{
    char s[20];
    printf("Enter the string?");
    scanf("%[^\n]s",s);
    printf("You entered %s",s);
}
```

Output

```
Enter the string? programing language
You entered programing language
```

Here we must also notice that we do not need to use address of (&) operator in scanf to store a string since string s is an array of characters and the name of the array, i.e., s indicates the base address of the string (character array) therefore we need not use & with it.

gets() and puts() functions:

The gets() and puts() are declared in the header file stdio.h. Both the functions are involved in the input/output operations of the strings.

gets() function:

The gets() function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The gets() allows the user to enter the space-separated strings. It returns the string entered by the user.

Declaration

```
char[] gets(char[]);
```

Reading string using gets()

```
#include<stdio.h>
void main ()
{
   char s[30];
   printf("Enter the string? ");
   gets(s);
   printf("You entered %s",s);
}
```

Output

```
Enter the string? programing language
You entered programing language
```

The gets() function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered. It suffers from buffer overflow, which can be avoided by using fgets(). The fgets() makes sure that not more than the maximum limit of characters are read. Consider the following example.

puts() function:

The puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.

Declaration

```
int puts(char[])
```

Let's see an example to read a string using gets() and print it on the console using puts().

```
#include<stdio.h>
#include <string.h>
int main(){
char name[50];
printf("Enter your name: ");
gets(name); //reads string from user
printf("Your name is: ");
puts(name);  //displays string
return 0;
}
```

Output:

```
Enter your name: Sonoo Jaiswal
```

```
Your name is: Sonoo Jaiswal
```

String Functions:

There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

String Length: strlen() function:

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

```
#include<stdio.h>
#include <string.h>
int main(){
char ch[20]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
   printf("Length of string is: %d",strlen(ch));
 return 0;
}
```

Output:

```
Length of string is: 10
```

Copy String: strcpy():

The strcpy(destination, source) function copies the source string in destination.

```
#include<stdio.h>
#include <string.h>
int main(){
 char ch[20]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
   char ch2[20];
   strcpy(ch2,ch);
   printf("Value of second string is: %s",ch2);
 return 0;
}
```

Output:

```
Value of second string is: javatpoint
```

String Concatenation: strcat():

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

```
#include<stdio.h>
#include <string.h>
int main(){
  char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
   char ch2[10]={'c', '\0'};
   strcat(ch,ch2);
   printf("Value of first string is: %s",ch);
 return 0;
}
```

Output:

```
Value of first string is: helloc
```

Compare String: strcmp():

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using *gets()* function which reads string from the console.

```
#include<stdio.h>
#include <string.h>
int main(){
  char str1[20],str2[20];
  printf("Enter 1st string: ");
  gets(str1);//reads string from console
  printf("Enter 2nd string: ");
  gets(str2);
  if(strcmp(str1,str2)==0)
      printf("Strings are equal");
  else
      printf("Strings are not equal");
 return 0;
}
```

Output:

```
Enter 1st string: hello
Enter 2nd string: hello
Strings are equal
```

Reverse String: strrev():

The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function.

```
#include<stdio.h>
#include <string.h>
int main(){
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nReverse String is: %s",strrev(str));
 return 0;
}
```

Output:

```
Enter string: javatpoint
String is: javatpoint
Reverse String is: tnioptavaj
```

String Lowercase: strlwr():

The strlwr(string) function returns string characters in lowercase. Let's see a simple example of strlwr() function.

```
#include<stdio.h>
#include <string.h>
int main(){
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nLower String is: %s",strlwr(str));
 return 0;
}
```

Output:

```
Enter string: JAVATpoint
String is: JAVATpoint
Lower String is: javatpoint
```

String Uppercase: strupr():

The strupr(string) function returns string characters in uppercase. Let's see a simple example of strupr() function.

```
#include<stdio.h>
#include <string.h>
int main(){
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nUpper String is: %s",strupr(str));
 return 0;
}
```

Output:

```
Enter string: javatpoint
String is: javatpoint
Upper String is: JAVATPOINT
```

String strstr():

The strstr() function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

Syntax:

```
char *strstr(const char *string, const char *match)
```

String strstr() parameters:

string: It represents the full string from where substring will be searched.

match: It represents the substring to be searched in the full string.

String strstr() example:

```
#include<stdio.h>
#include <string.h>
int main(){
  char str[100]="this is javatpoint with c and java";
  char *sub;
  sub=strstr(str,"java");
  printf("\nSubstring is: %s",sub);
 return 0;
}
```

Output:

```
javatpoint with c and java
```

UNIT-3

UNIT-III

What are Data Structures?

- ❖ 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, Artifical 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 for Data Structures

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

Processor speed: To handle very large amout 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.

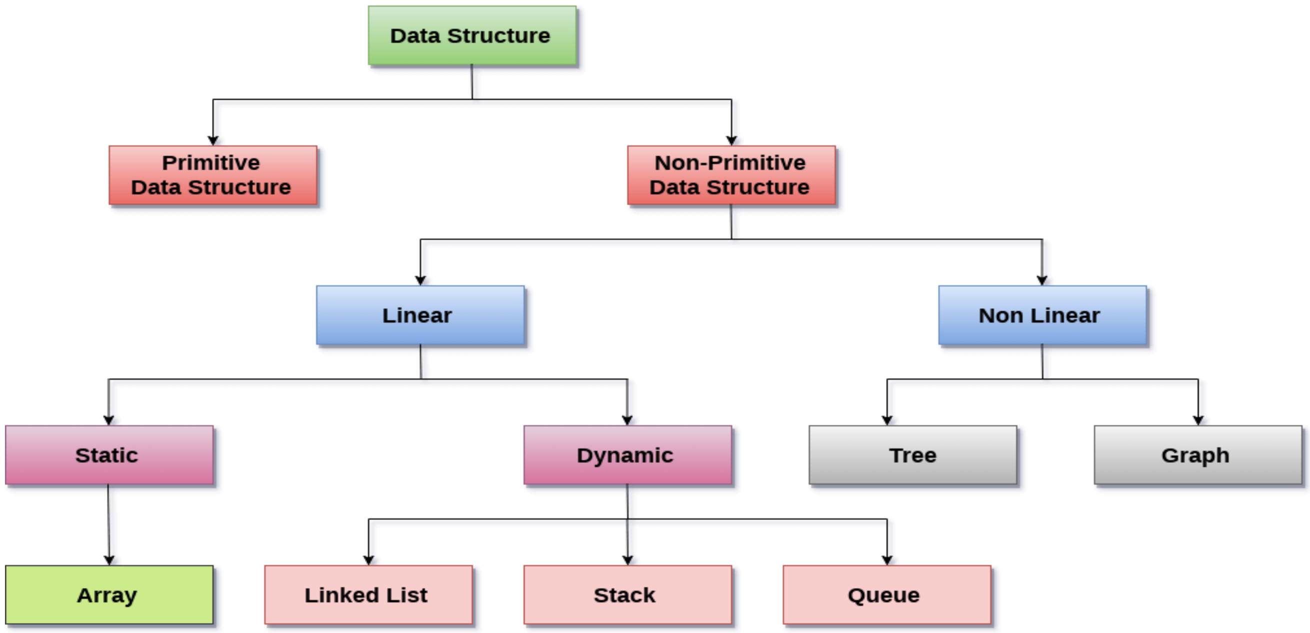
Advantages of Data Structures

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 perticular 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.

Types of Data Structures



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 age are:

age[0], age[1], age[2], age[3],......... age[98], age[99].

What is a Stack?

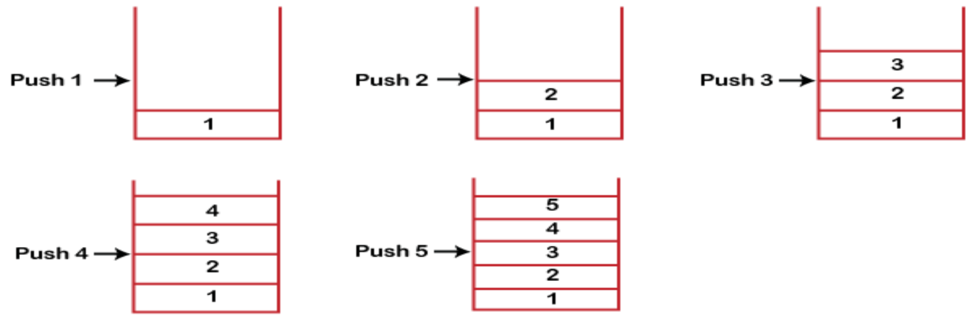
- ❖ A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle.
- ❖ Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack.
- ❖ Whenever an element is added in the 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

- o It is called as stack because it behaves like a real-world stack, piles of books, etc.
- o A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- o 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

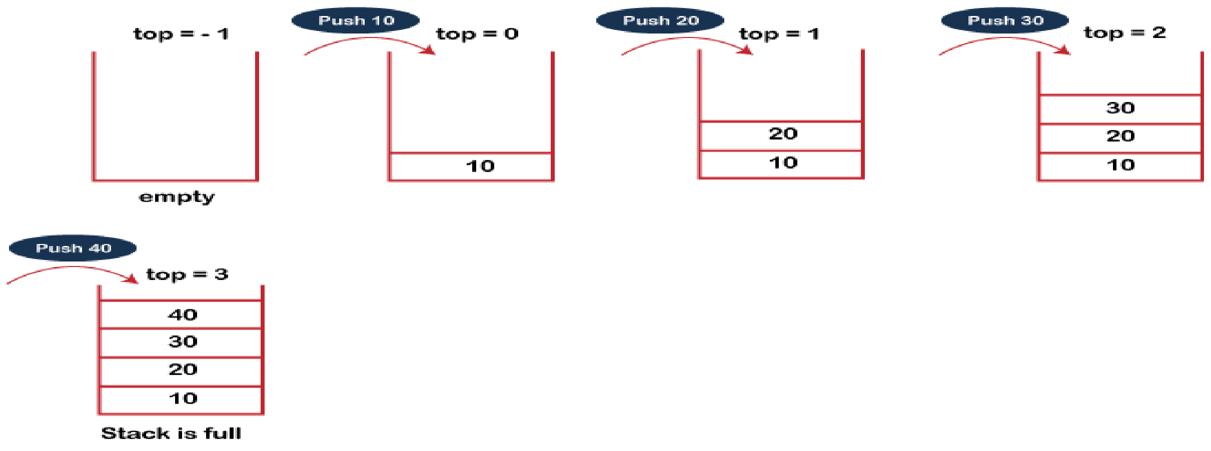
The following are some common operations implemented on the stack:

- **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:

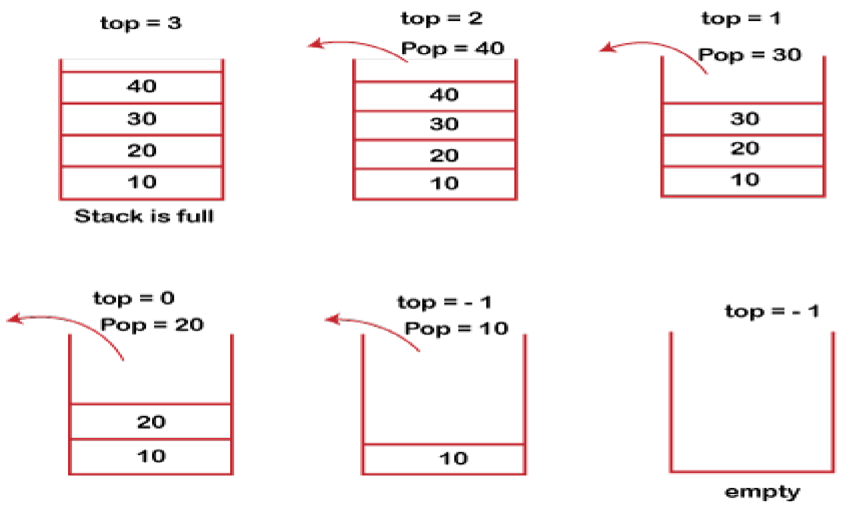
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as **-1 or NULL** to check that the stack is empty.
- When 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:

```
int main()
{
   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.

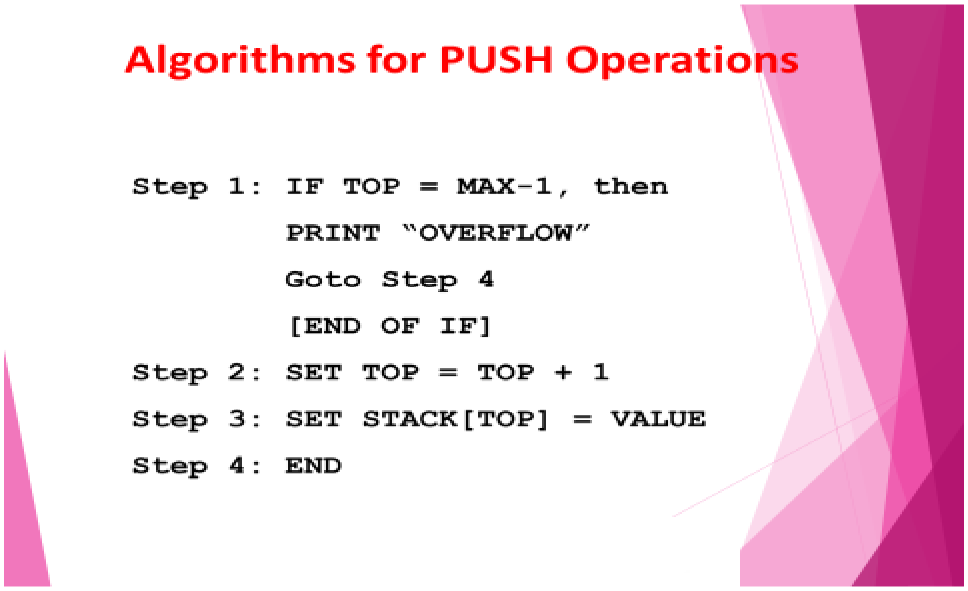
Adding an element onto the stack (push operation):

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 refere 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:



Algorithms for PUSH Operations

```
Step 1: IF TOP = MAX-1, then
            PRINT "OVERFLOW"
            Goto Step 4
            [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Time Complexity : o(1)

implementation of push algorithm in C language

```
void push (int val,int MAX) //n is size of the stack
{
   if (top == MAX-1)
   printf("\n Overflow");
   else
   {
   top = top +1;
```

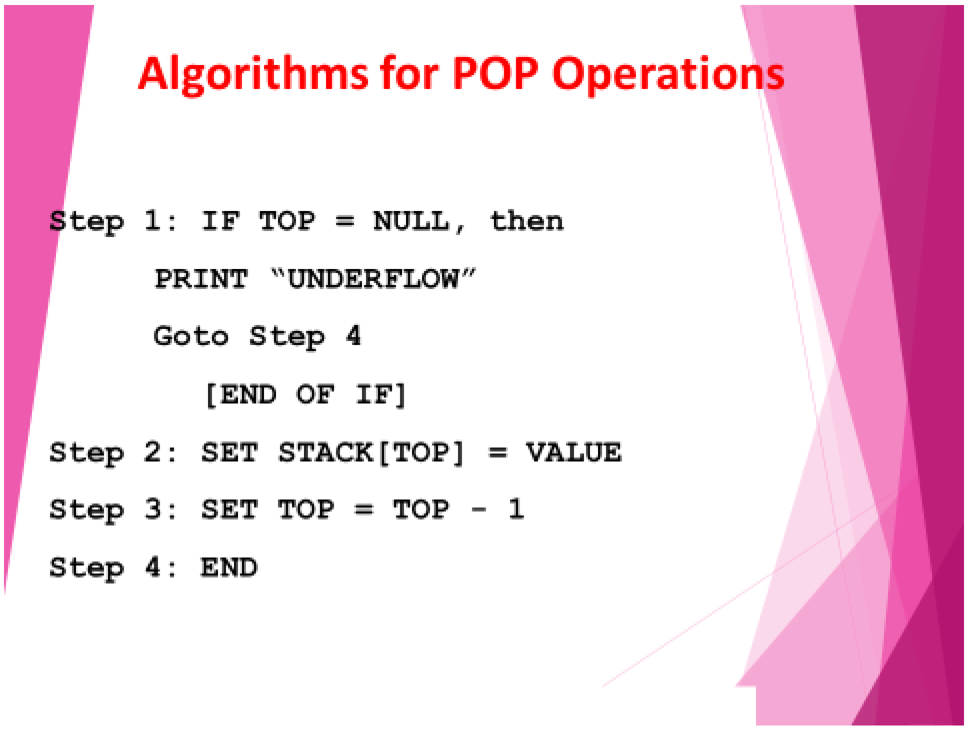
```
    stack[top] = val;
    }
}
```

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 :



Algorithms for POP Operations

```
Step 1: IF TOP = NULL, then
             PRINT "UNDERFLOW"
             Goto Step 4
                [END OF IF]
Step 2: SET STACK[TOP] = VALUE
Step 3: SET TOP = TOP - 1
Step 4: END
```

Time Complexity : o(1)

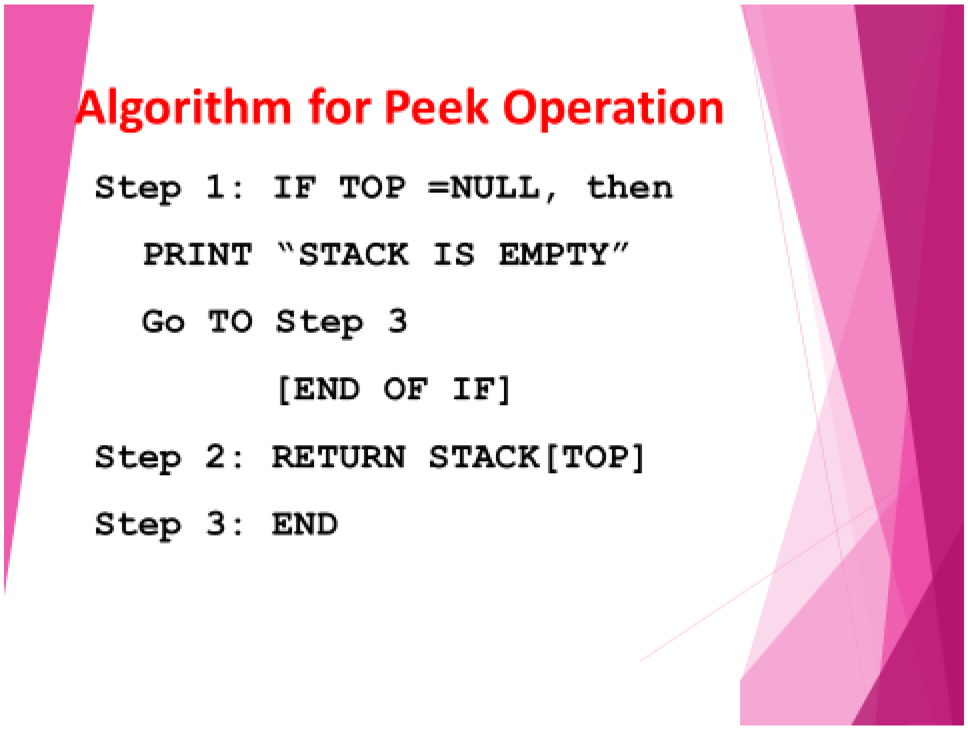
Implementation of POP algorithm using C language

```
int pop ()
{
    if(top == NULL)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack[top - - ];
    }
}
```

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 :



Algorithm for Peek Operation

```
Step 1: IF TOP =NULL, then
   PRINT "STACK IS EMPTY"
   Go TO Step 3
         [END OF IF]
Step 2: RETURN STACK[TOP]
Step 3: END
```

Time complexity: o(n)

Implementation of Peek algorithm in C language

```
int peek()
{
    if (top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack [top];
    }
}
```

C program for implantation stack using arrays

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();

void main ()
{

    printf("Enter the number of elements in the stack ");
    scanf("%d",&n);
    printf("*********Stack operations using array*********");

printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("Chose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
```

```
            {
                pop();
                break;
            }
            case 3:
            {
                show();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice ");
            }
        };
    }
}

void push ()
{
    int val;
    if (top == n )
    printf("\n Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}

```

```
void pop ()
{
    if(top == -1)
    printf("Underflow");
    else
    top = top -1;
}
void show()
{
    for (i=top;i>=0;i--)
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
        printf("Stack is empty");
    }
}
```

Arithmetic Expression Evaluation:

The stack organization is very effective in evaluating arithmetic expressions. Expressions are usually represented in what is known as **Infix notation**, in which each operator is written between two operands (i.e., A + B). With this notation, we must distinguish between (A + B)*C and A + (B * C) by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

1. Polish notation (prefix notation) –

It refers to the notation in which the operator is placed before its two operands. Here no parentheses are required, i.e.,

+AB

2. Reverse Polish notation(postfix notation) –

It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses is required in Reverse Polish notation, i.e.,

AB+

Stack-organized computers are better suited for post-fix notation than the traditional infix notation. Thus, the infix notation must be converted to the postfix notation. The conversion from infix notation to postfix notation must take into consideration the operational hierarchy.

There are 3 levels of precedence for 5 binary operators as given below:

Highest: Exponentiation (^)

Next highest: Multiplication (*) and division (/)

Lowest: Addition (+) and Subtraction (-)

For example –

Infix notation: (A-B)*[C/(D+E)+F]

Post-fix notation: AB- CDE +/F +*

Here, we first perform the arithmetic inside the parentheses (A-B) and (D+E). The division of C/(D+E) must be done prior to the addition with F. After that multiply the two terms inside the parentheses and bracket.

Now we need to calculate the value of these arithmetic operations by using a stack.

Evaluating postfix expression

The procedure for getting the result is:

1. Convert the expression in Reverse Polish notation(post-fix notation).
2. Push the operands into the stack in the order they appear.
3. When any operator encounters then pop two topmost operands for executing the operation.
4. After execution push the result obtained into the stack.
5. After the complete execution of expression, the final result remains on the top of the stack.

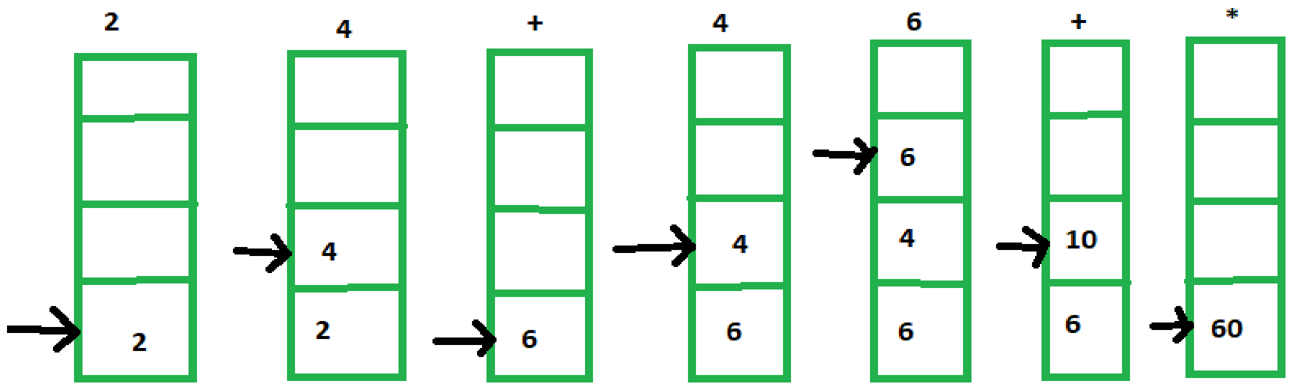
For example –

Infix notation: (2+4) * (4+6)

Post-fix notation: 2 4 + 4 6 + *

Result: 60

The stack operations for this expression evaluation is shown below:



Stack operations to evaluate (2+4)*(4+6)

Evaluation of postfix expression

Algorithm

Scan the input string from left to right.

For each input symbol,

- If it is a digit then, push it on to the stack.
- If it is an operator then, pop out the top most two contents from the stack and apply the operator on them. Later on, push the result on to stack.
- If the input symbol is '\0', empty the stack.

Following is the C program for an evaluation of postfix expression

```
#include<stdio.h>
int top = -1, stack [100];

main ( ){
   char a[50], ch;
   int i,op1,op2,res,x;
   void push (int);
   int pop( );
   int eval (char, int, int);
   printf("enter a postfix expression:");
   gets (a);
```

```
  for(i=0; a[i]!='\0'; i++){
    ch = a[i];
    if (ch>='0' && ch<='9')
      push('0');
    else{
      op2 = pop ( );
      op1 = pop ( );
      res = eval (ch, op1, op2);
      push (res);
    }
  }
  x = pop ( );
  printf("evaluated value = %d", x);
  getch ( );
}
void push (int n){
  top++;
  stack [top] = n;
}
int pop ( ){
  int res ;
  res = stack [top];
  top--;
  return res;
}
int eval (char ch, int op1, int op2){
  switch (ch){
    case '+' : return (op1+op2);
    case '-' : return (op1-op2);
    case '*' : return (op1*op2);
    case '/' : return (op1/op2);
  }
}
```

Output:

When the above program is executed, it produces the following result −

Run 1:

```
enter a postfix expression:45+
evaluated value = 9
```

Run 2:

```
enter a postfix expression: 3 5 2 * +
evaluated value = 13
```

Convert Infix to Postfix Expression

Infix expressions are readable and solvable by humans. We can easily distinguish the order of operators, and also can use the parenthesis to solve that part first during solving mathematical expressions. The computer cannot differentiate the operators and parenthesis easily, that's why postfix conversion is needed.

To convert infix expression to postfix expression, we will use the stack data structure. By scanning the infix expression from left to right, when we will get any operand, simply add them to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

Algorithm to convert Infix To Postfix

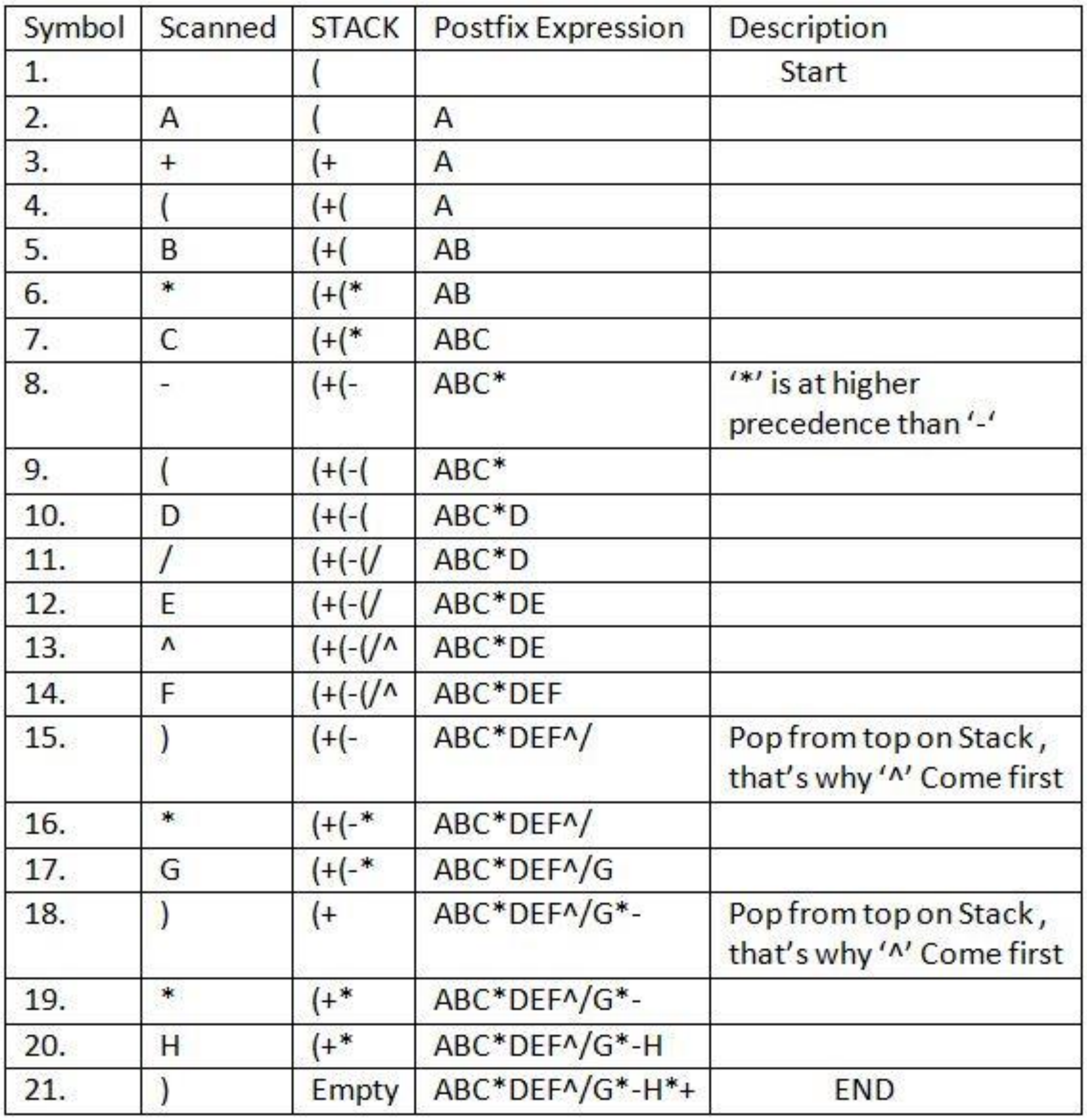
Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

```
1. Push "("onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as
   or higher precedence than operator.
2. Add operator to Stack.
   [End of If]
6. If a right parenthesis is encountered ,then:
1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is
   encountered.
2. Remove the left Parenthesis.
   [End of If]
   [End of If]
7. END.
```

ADVERTISEMENT

Let's take an example to better understand the algorithm

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where ^ is an exponential operator.



Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: ABC*DEF^/G*-H*+

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

C Program to Convert Infix to Postfix using Stack

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

char stack[100];
int top = -1;
```

```
void push(char x)
{
    stack[++top] = x;
}

char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}

int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
    return 0;
}

int main()
{
    char exp[100];
    char *e, x;
    printf("Enter the expression : ");
    scanf("%s",exp);
    printf("\n");
    e = exp;

    while(*e != '\0')
    {
        if(isalnum(*e))
            printf("%c ",*e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
            while((x = pop()) != '(')
                printf("%c ", x);
        }
        else
        {
            while(priority(stack[top]) >= priority(*e))
                printf("%c ",pop());
            push(*e);
        }
        e++;
    }

    while(top != -1)
    {
        printf("%c ",pop());
    }return 0;
}
```

Output Test Case 1:

```
Enter the expression : a+b*c

a b c * +
```

Output Test Case 2:

```
Enter the expression : (a+b)*c+(d-a)

a b + c * d a - +
```

Output Test Case 3:

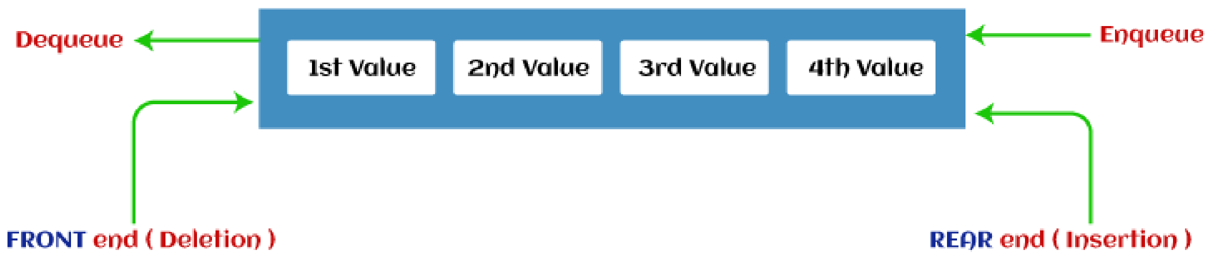
```
Enter the expression : ((4+8)(6-5))/((3-2)(2+2))
4 8 + 6 5 - 3 2 - 2 2 + /
```

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.



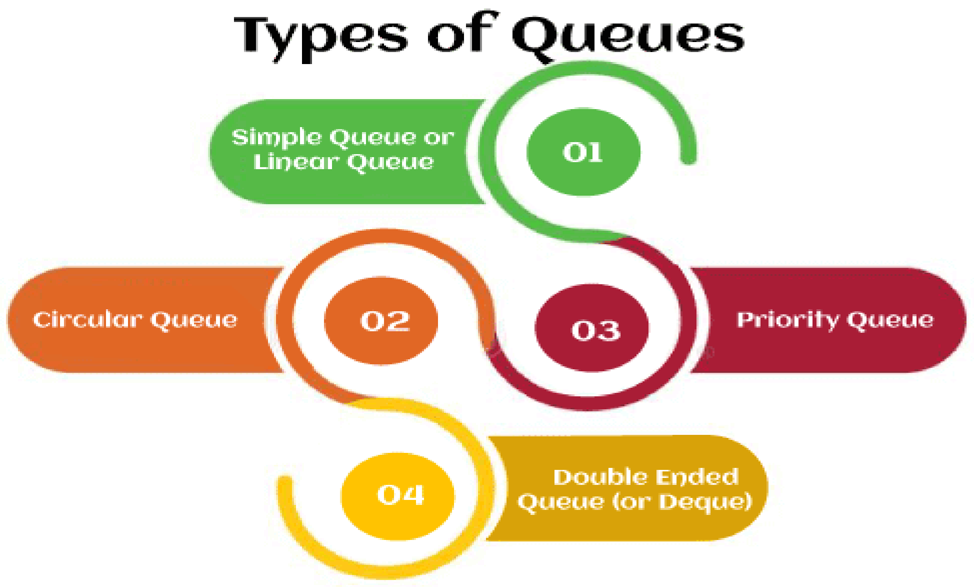
Applications of Queue

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 Queue

There are four different types of queue that are listed as follows -

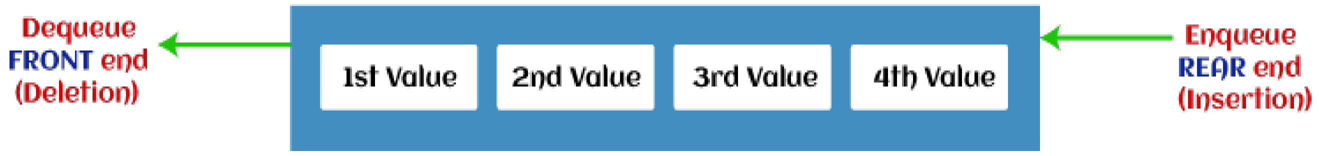


- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Let's discuss each of the type of queue.

Simple Queue or Linear Queue

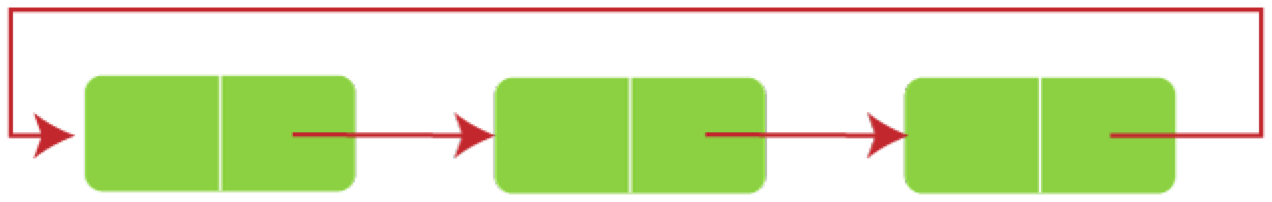
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



The 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 representation of circular queue is shown in the below image -

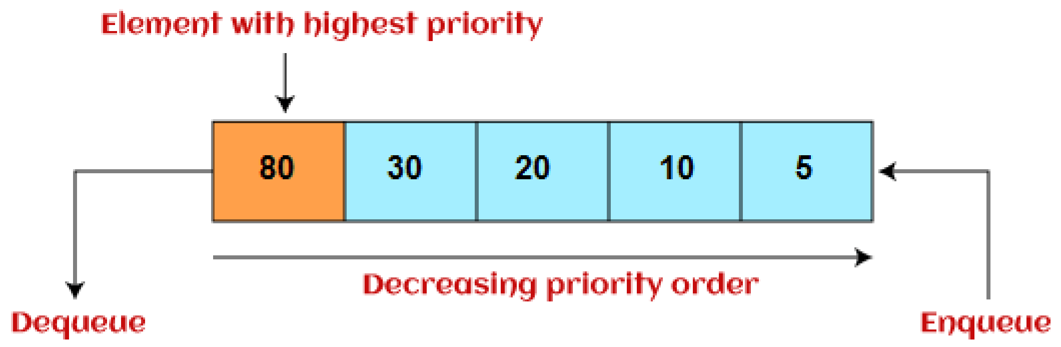


Circular Queue

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. The main advantage of using the circular queue is better memory utilization.

Priority Queue

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

There are two types of priority queue that are discussed as follows -

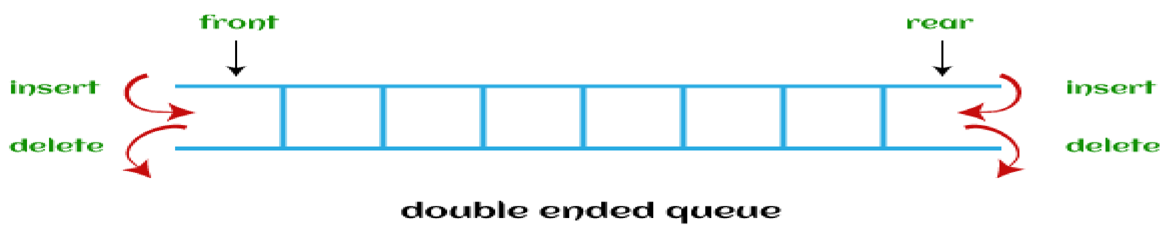
- **Ascending priority queue -** In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
- **Descending priority queue -** In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

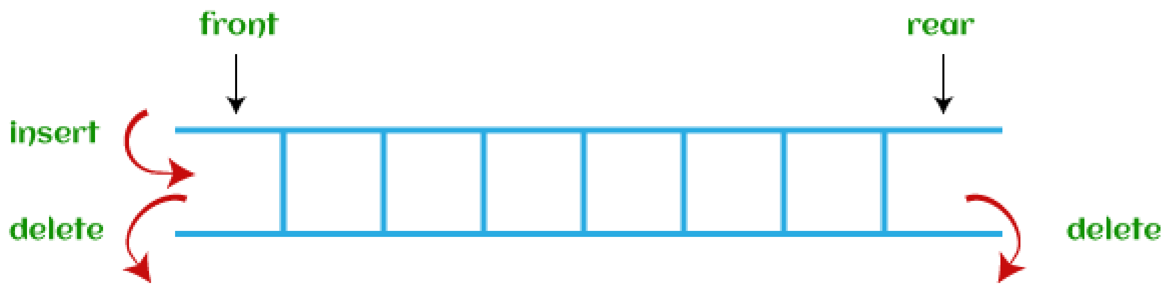
Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

The representation of the deque is shown in the below image -



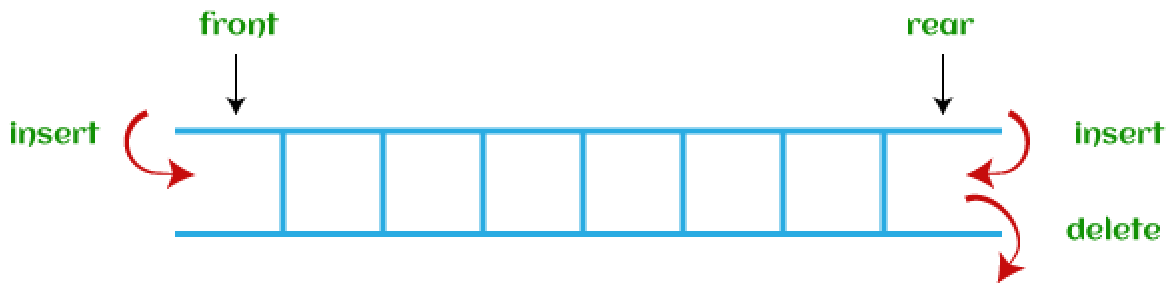
There are two types of deque that are discussed as follows -

- **Input restricted deque -** As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

- **Output restricted deque -** As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Now, let's see the operations performed on the queue.

Operations performed on queue:

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
```

```
    return true
  else
    return false
  endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

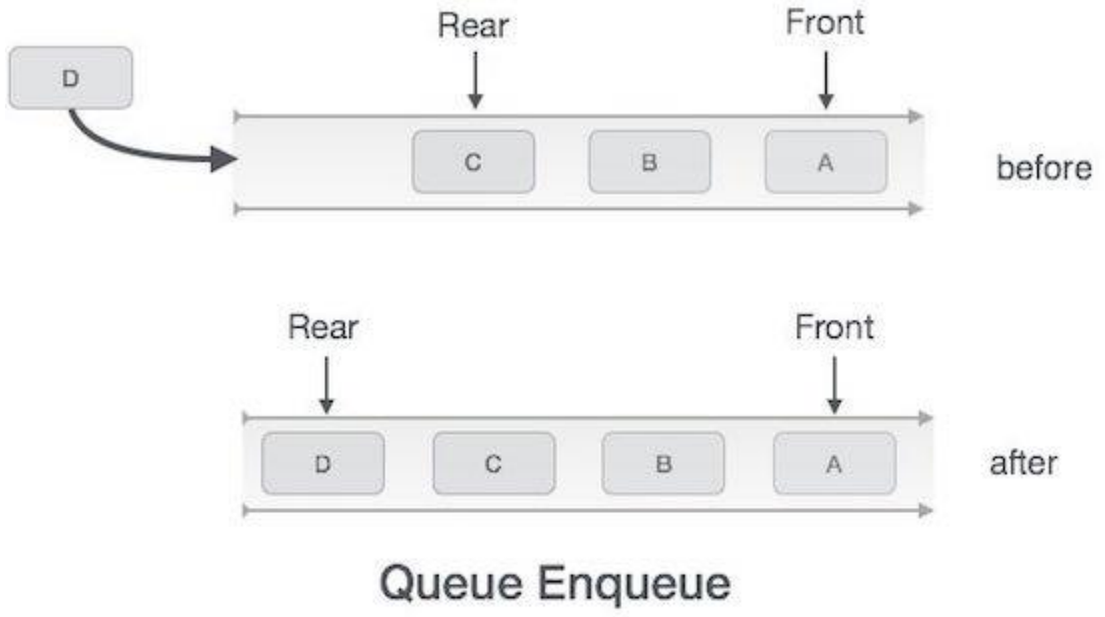
```
bool isempty() {
  if(front < 0 || front > rear)
    return true;
  else
    return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.

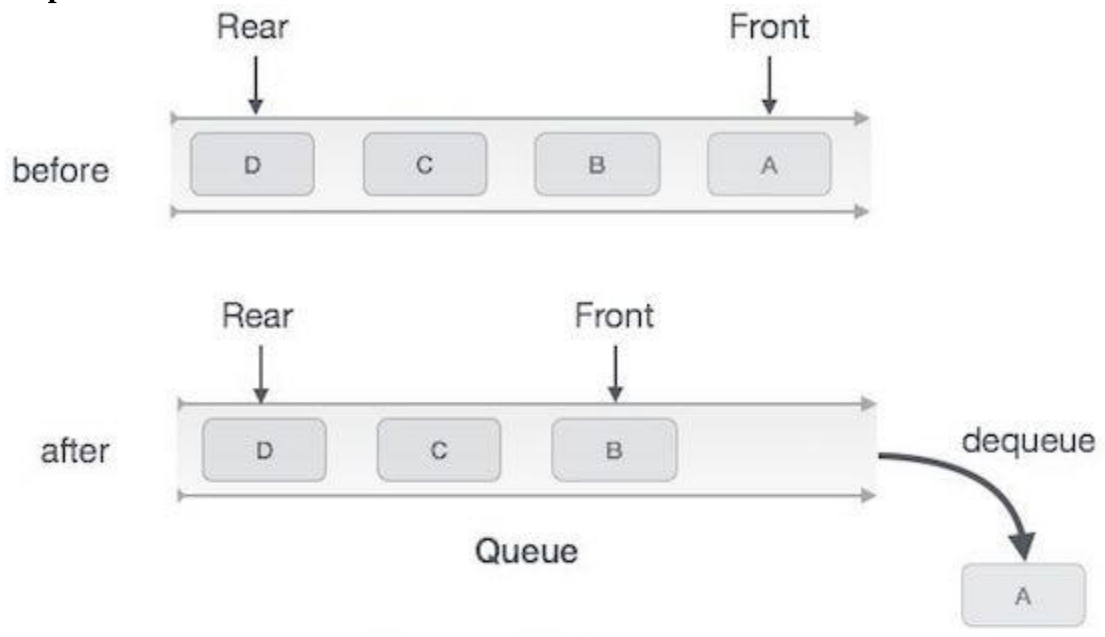


Implementation of enqueue() in C programming language −

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

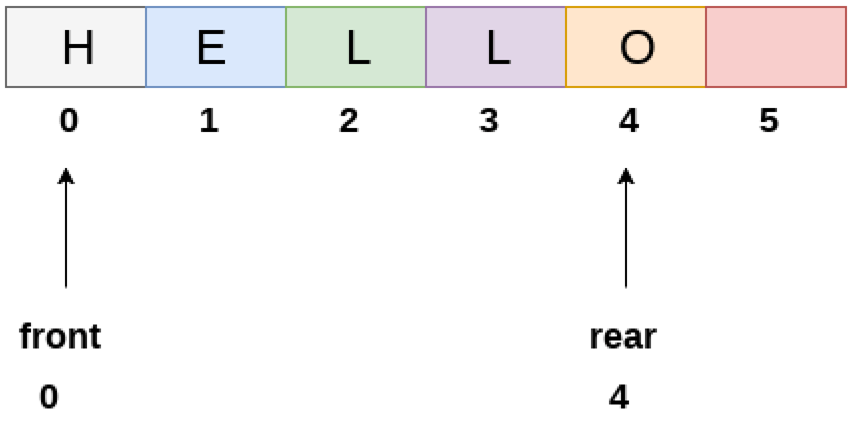
- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

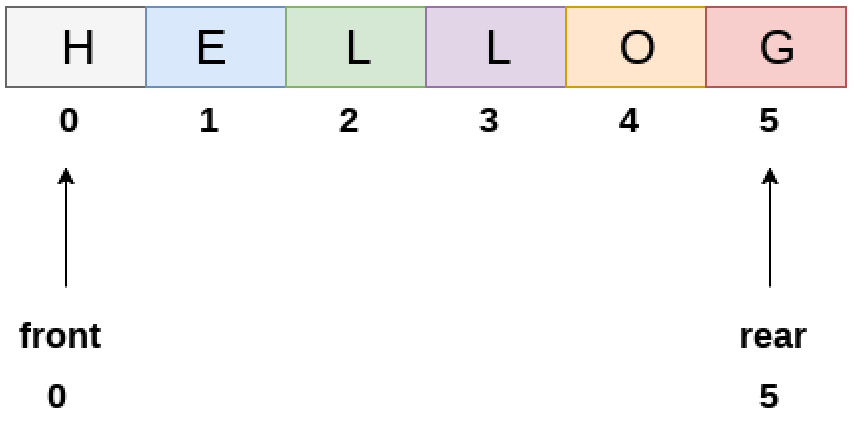
Array representation of Queue:

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.



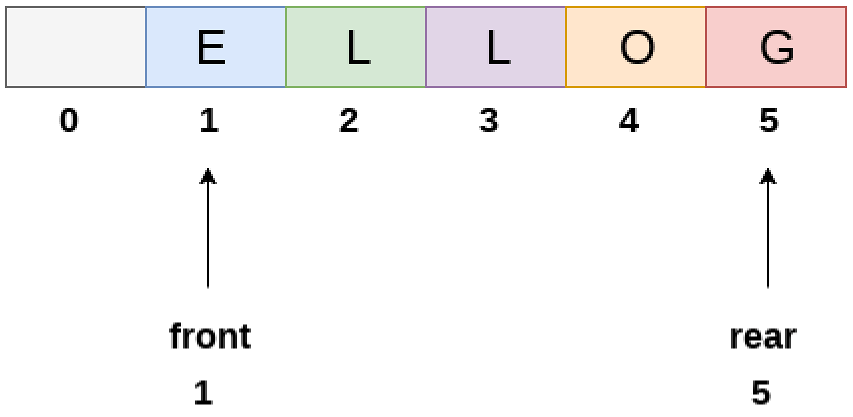
Queue

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(Dequeue)

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

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

C implementation:

```
void insert (int queue[], int max, int front, int rear, int item)
{
    if (rear + 1 == max)
    {
        printf("overflow");
    }
    else
    {
        if(front == -1 && rear == -1)
        {
            front = 0;
            rear = 0;
        }
        else
        {
            rear = rear + 1;
        }
        queue[rear]=item;
    }
}
```

Algorithm to delete an element from the queue(Dequeue):

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 implementation

```
int delete (int queue[], int max, int front, int rear)
{
    int y;
    if (front == -1 || front > rear)

    {
        printf("underflow");
    }
    else
    {
        y = queue[front];
        if(front == rear)
        {
            front = rear = -1;
            else
            front = front + 1;

        }
        return y;
    }
}
```

Program to implement queue using array:

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n*************************Main Menu*****************************\n");
        printf("\n=================================================================\n");

        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
    }
}
void insert()
{
    int item;
    printf("\nEnter the element\n");
    scanf("\n%d",&item);
    if(rear == maxsize-1)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    if(front == -1 && rear == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        rear = rear+1;
    }
    queue[rear] = item;
    printf("\nValue inserted ");

}
void delete()
{
    int item;
    if (front == -1 || front > rear)
    {
        printf("\nUNDERFLOW\n");
        return;

    }
    else
    {
        item = queue[front];
        if(front == rear)
        {
            front = -1;
            rear = -1 ;
        }
        else
        {
            front = front + 1;
        }
        printf("\nvalue deleted ");
    }
```

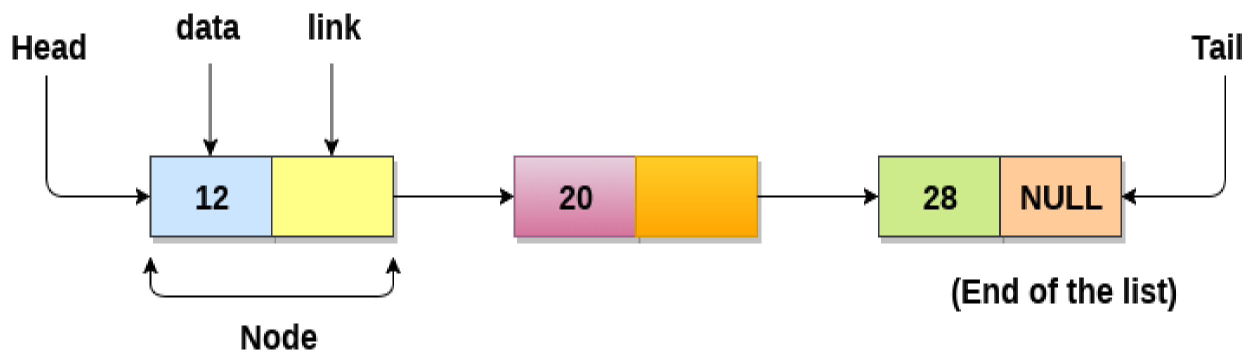
```


}

void display()
{
    int i;
    if(rear == -1)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values .....\n");
        for(i=front;i<=rear;i++)
        {
            printf("\n%d\n",queue[i]);
                        }
                    }
                }
```

UNIT-4

UNIT-IV

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 any where in the memory and linked together to make 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 node can not 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 following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking 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 linked list is useful because,

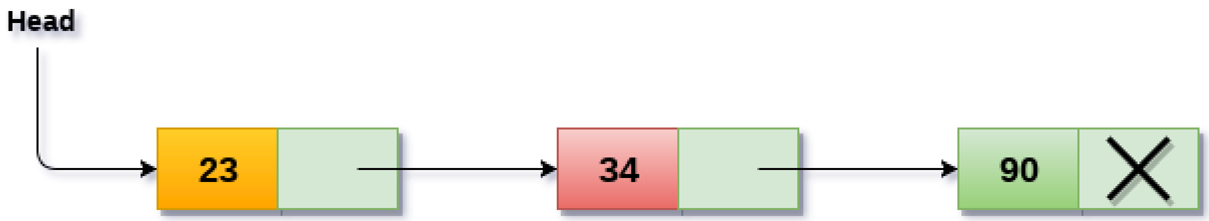
1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and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

Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. 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								Space Compleity
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

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

```
struct node
{
    int data;
    struct node *next;
};
struct node *head, *ptr;
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.

S.No	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.

S.No	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 specified node	It involves deleting the node after the specified node in the list. we need 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	Traversing	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	Searching	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. .

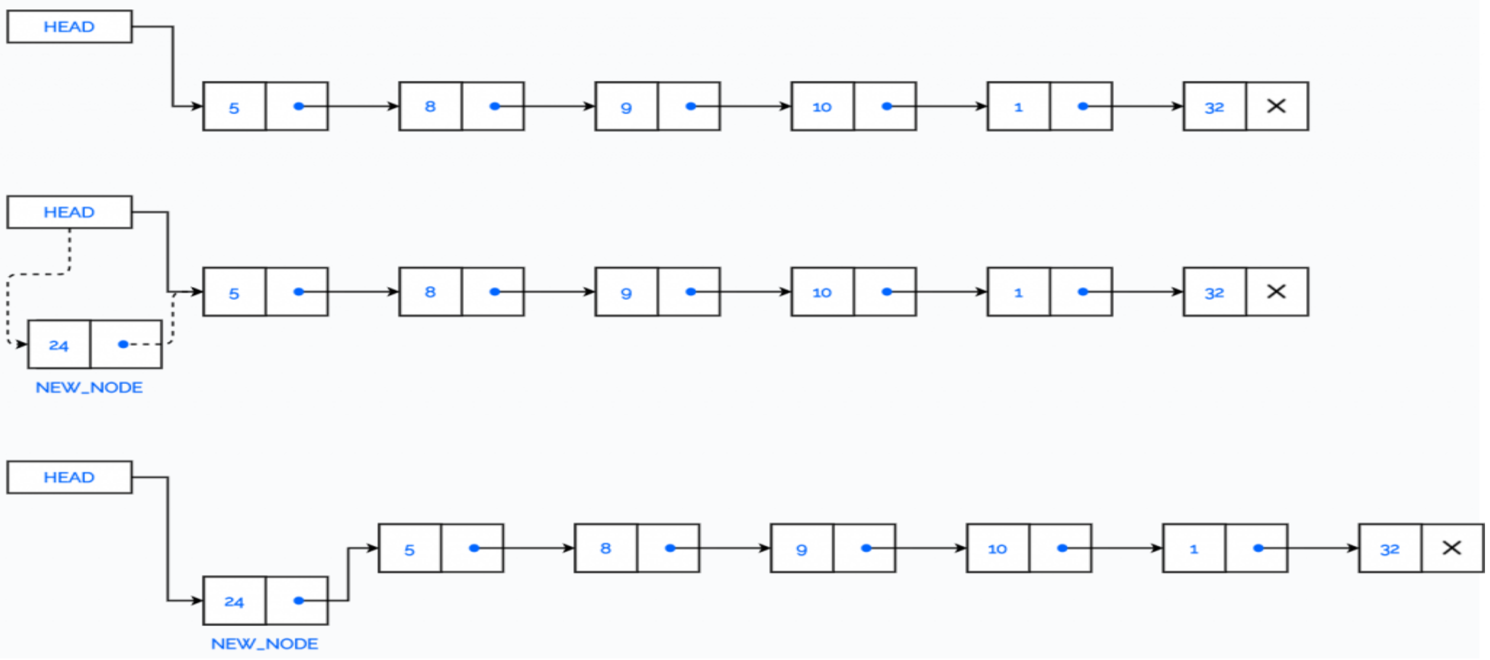
Inserting Elements to a Linked List

We will see how a new node can be added to an existing linked list in the following cases.

1. The new node is inserted at the beginning.
2. The new node is inserted at the end.
3. The new node is inserted after a given node.

Insert a Node at the beginning of a Linked list:

Consider the linked list shown in the figure. Suppose we want to create a new node with data 24 and add it as the first node of the list. The linked list will be modified as follows.



- Allocate memory for new node and initialize its DATA part to 24.
- Add the new node as the first node of the list by pointing the NEXT part of the new node to HEAD.
- Make HEAD to point to the first node of the list.

Algorithm: InsertAtBeginning

```
Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 7
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = HEAD
Step 6: SET HEAD = NEW_NODE
Step 7: EXIT
```

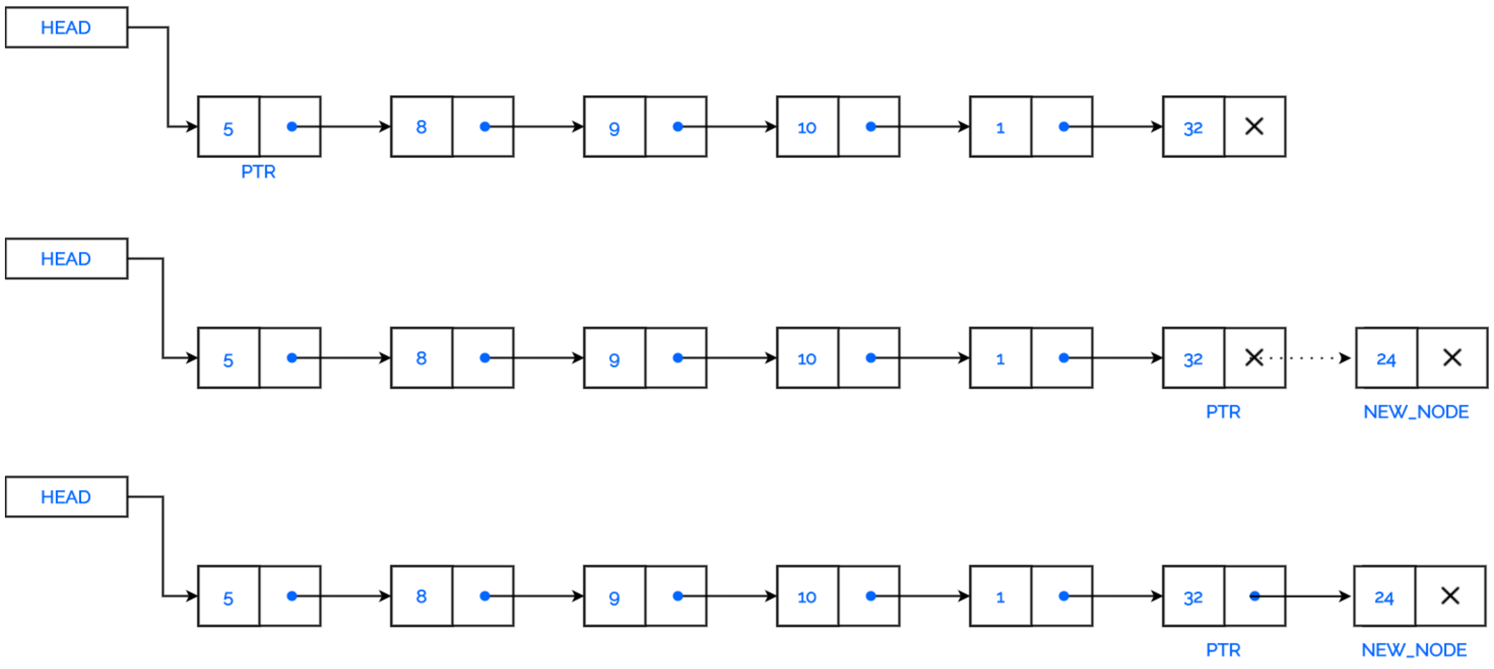
Note that the first step of the algorithm checks if there is enough memory available to create a new node. The second, and third steps allocate memory for the new node.

This algorithm can be implemented in C as follows:

```
struct node *new_node;
new_node = (struct node*) malloc(sizeof(struct node));
new_node->data = 24;
new_node->next = head;
head = new_node;
```

Insert a Node at the end of a Linked list:

Take a look at the linked list in the figure. Suppose we want to add a new node with data 24 as the last node of the list. Then the linked list will be modified as follows.



- Allocate memory for new node and initialize its DATA part to 24.
- Traverse to last node.
- Point the NEXT part of the last node to the newly created node.
- Make the value of next part of last node to NULL.

Algorithm: InsertAtEnd

```
Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 10
[END OF IF]
```

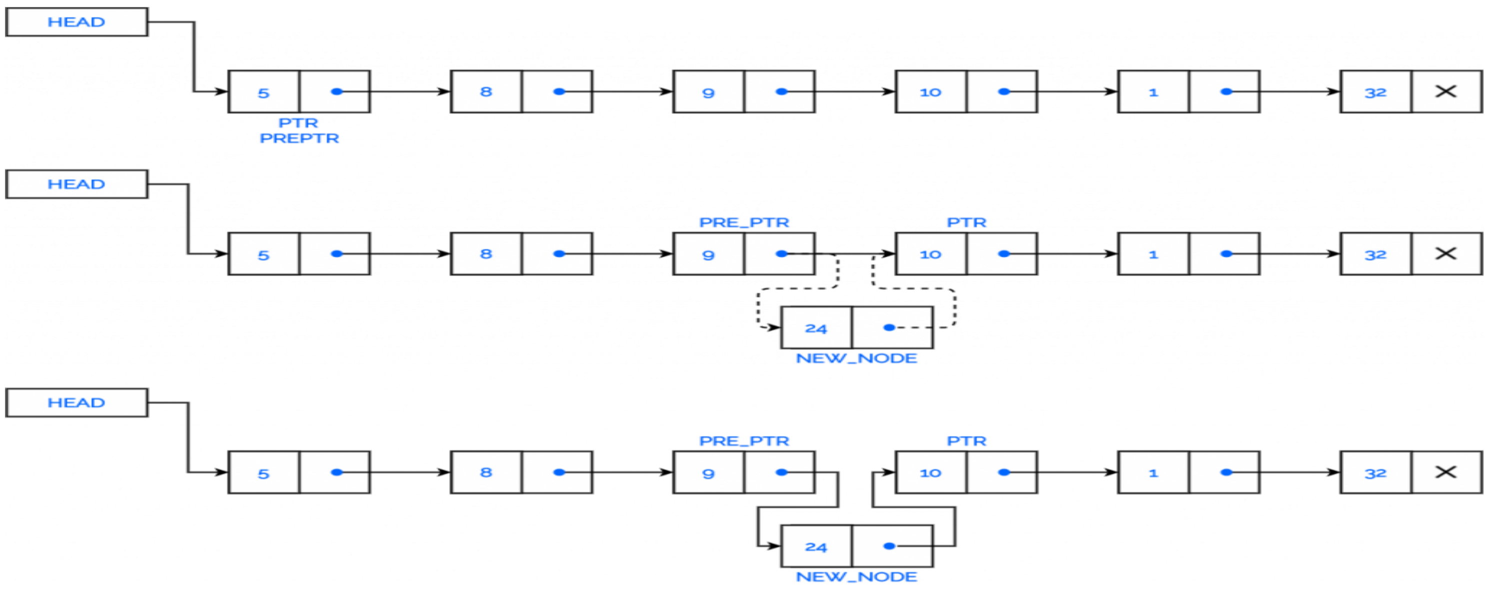
```
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = HEAD
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

This can be implemented in C as follows,

```
struct node *new_node;
new_node = (struct node*) malloc(sizeof(struct node));
new_node->data = 24;
new_node->next = NULL;
struct node *ptr = head;
while(ptr->next != NULL){
ptr = ptr->next;
}
ptr->next = new_node;
```

Insert a Node after a given Node in a Linked list:

The last case is when we want to add a new node after a given node. Suppose we want to add a new node with value 24 after the node having data 9. These changes will be done in the linked list.



- Allocate memory for new node and initialize its DATA part to 24.
- Traverse the list until the specified node is reached.
- Change NEXT pointers accordingly.

Algorithm: InsertAfterAnElement

```
Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 12
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = HEAD
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA != NUM
Step 8: SET PREPTR = PTR
Step 9: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 1 : PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

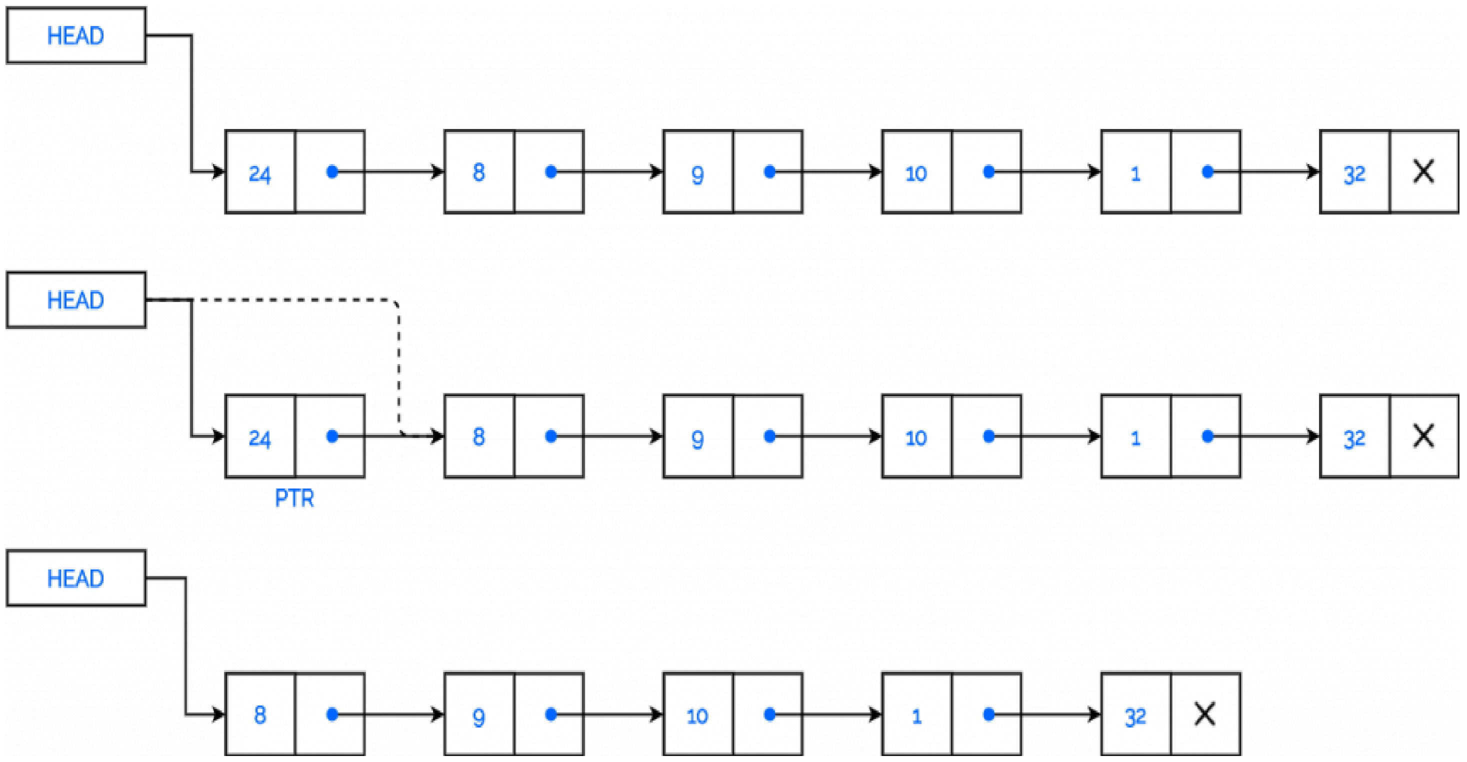
Deleting Elements from a Linked List

Let's discuss how a node can be deleted from a linked listed in the following cases.

1. The first node is deleted.
2. The last node is deleted.
3. The node after a given node is deleted

Delete a Node from the beginning of a Linked list:

Suppose we want to delete a node from the beginning of the linked list. The list has to be modified as follows:



- Check if the linked list is empty or not. Exit if the list is empty.
- Make HEAD points to the second node.
- Free the first node from memory.

Algorithm: DeleteFromBeginning

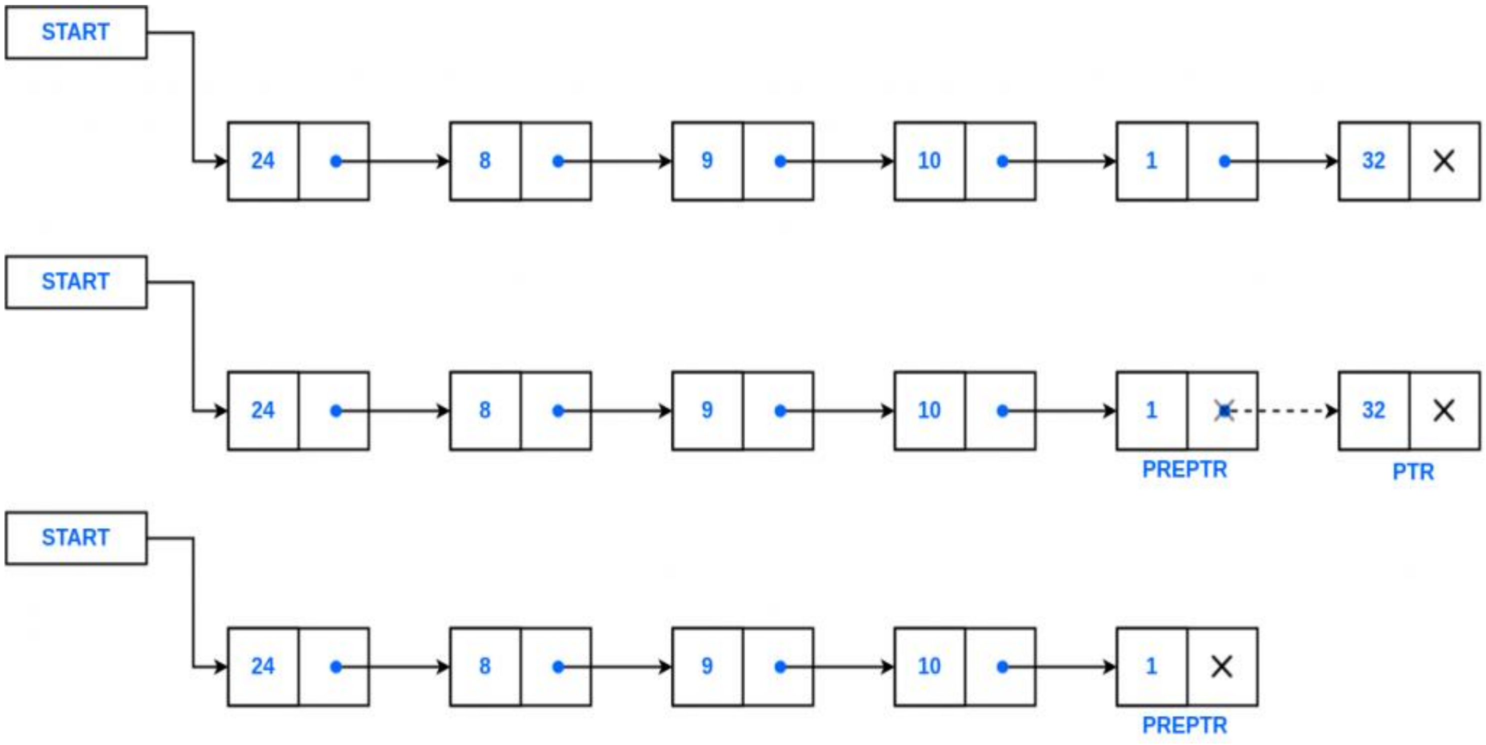
```
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 5
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: SET HEAD = HEAD -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

This can be implemented in C as follows,

```
if(head == NULL)
{
printf("Underflow");
}
else
{
ptr = head;
head = head -> next;
free(ptr);
}
```

Delete last Node from a Linked list:

Suppose we want to delete the last node from the linked list. The linked list has to be modified as follows:



- Traverse to the end of the list.
- Change value of next pointer of second last node to NULL.
- Free last node from memory.

Algorithm: DeleteFromEnd

```
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4: SET PREPTR = PTR
Step 5: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

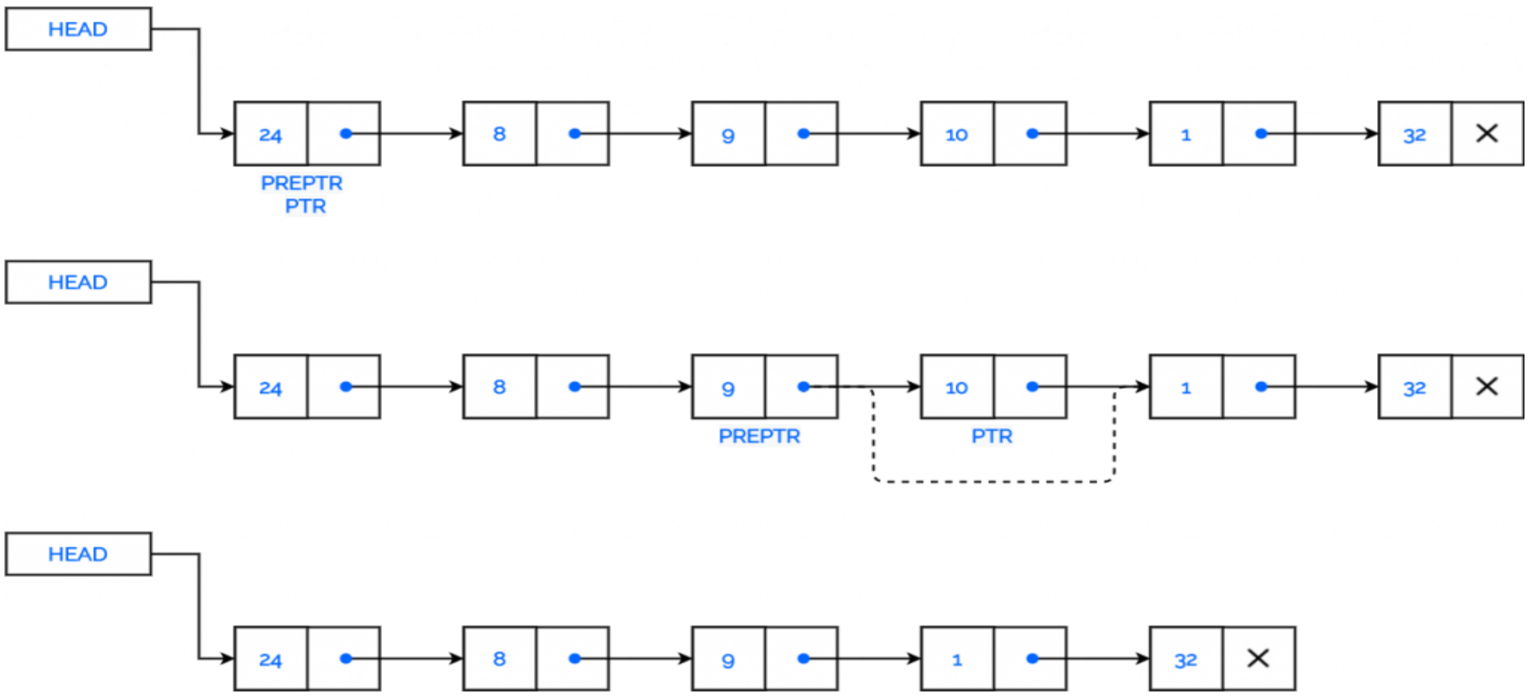
Here we use two pointers PTR and PREPTR to access the last node and the second last node.

This can be implemented in C as follows:

```
if(head == NULL)
{
printf("Underflow");
}
else
{
struct node* ptr = head;
struct node* preptr = NULL;
while(ptr->next!=NULL){
preptr = ptr;
ptr = ptr->next;
}
preptr->next = NULL;
free(ptr);
}
```

Delete the Node after a given Node in a Linked list:

Suppose we want to delete the that comes after the node which contains data 9.



- Traverse the list upto the specified node.
- Change value of next pointer of previous node(9) to next pointer of current node(10).

Algorithm: DeleteAfterANode

```
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 10
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5: SET PREPTR = PTR
Step 6: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10 : EXIT
```

Implementation in C takes the following form:

```
if(head == NULL)
{
printf("Underflow");
}
else
{
struct node* ptr = head;
struct node* preptr = ptr;
while(ptr->data!=num){
preptr = ptr;
ptr = ptr->next;
}
struct node* temp = ptr;
preptr -> next = ptr -> next;
free(temp);
}
```

Search

Finding an element is similar to a traversal operation. Instead of displaying data, we have to check whether the data matches with the **item** to find.

- Initialize PTR with the address of HEAD. Now the PTR points to the first node of the linked list.
- A while loop is executed which will compare data of every node with item.
- If item has been found then control goes to last step.

Algorithm: Search

```
Step 1: [INITIALIZE] SET PTR = HEAD
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: If ITEM = PTR -> DATA
SET POS = PTR
Go To Step 5
ELSE
SET PTR = PTR -> NEXT
[END OF IF]
[END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Search operation can be implemented in C as follows:

```
struct node* ptr = head;
struct node* pos = NULL;
while (ptr != NULL) {
if (ptr->data == item)
pos = ptr
printf("Element Found");
break;
else
ptr = ptr -> next;
}
```

Linked List in C: Menu Driven Program

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
    int choice =0;
    while(choice != 9)
    {
        printf("\n\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
        printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n
        5.Delete from last\n6.Delete node after specified location\n7.Search for an element\n8.Show\n9.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            beginsert();
            break;
            case 2:
            lastinsert();
            break;
            case 3:
            randominsert();
            break;
            case 4:
            begin_delete();
            break;
            case 5:
            last_delete();
            break;
            case 6:
            random_delete();
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
```

```
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void beginsert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node *));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value\n");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nNode inserted");
    }

}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
                        ptr -> next = NULL;
                        head = ptr;
                        printf("\nNode inserted");
                    }
                    else
                    {
                        temp = head;
                        while (temp -> next != NULL)
                        {
                            temp = temp -> next;
                        }
            temp->next = ptr;
            ptr->next = NULL;
            printf("\nNode inserted");

        }
    }
}
void randominsert()
{
    int i,loc,item;
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }

        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
    }
}
void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nList is empty\n");
    }
    else
    {
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\nNode deleted from the begining ...\n");
    }
}
```

```
void last_delete()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nOnly node of the list deleted ...\n");
    }

    else
    {
        ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
        ptr1->next = NULL;
        free(ptr);
        printf("\nDeleted Node from the last ...\n");
    }
}
void random_delete()
{
    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which you want to perform deletion \n");
    scanf("%d",&loc);
    ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node %d ",loc+1);
}
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }

}

void display()
{
    struct node *ptr;
    ptr = head;
    if(ptr == NULL)
    {
        printf("Nothing to print");
    }
    else
    {
        printf("\nprinting values . . . . .\n");
        while (ptr!=NULL)
        {
            printf("\n%d",ptr->data);
            ptr = ptr -> next;
        }
    }
}

```

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
```

Traverse a Linked List

Accessing the nodes of a linked list in order to process it is called **traversing** a linked list. Normally we use the traverse operation to display the contents or to search for an element in the linked list. The algorithm for traversing a linked list is given below.

Algorithm: Traverse

```
Step 1: [INITIALIZE] SET PTR = HEAD
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: Apply process to PTR -> DATA
Step 4: SET PTR = PTR->NEXT
[END OF LOOP]
Step 5: EXIT
```

- We first initialize PTR with the address of HEAD. Now the PTR points to the first node of the linked list.
- A while loop is executed, and the operation is continued until PTR reaches the last node (PTR = NULL).
- Apply the process(display) to the current node.
- Move to the next node by making the value of PTR to the address of next node.

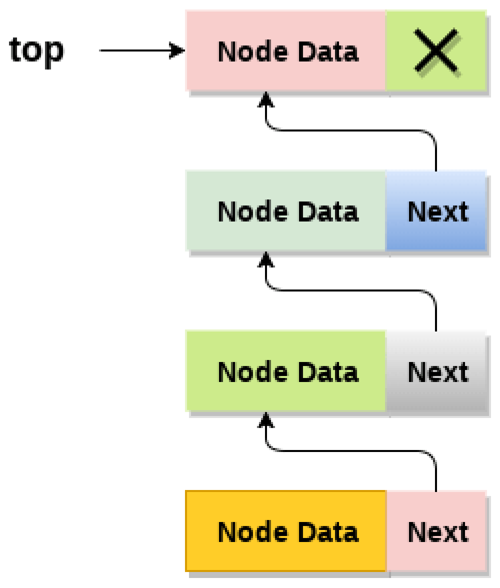
The following block of code prints all elements in a linked list in C.

```
struct node *ptr = head;
printf("Elements in the linked list are : ");
while (ptr != NULL)
{
printf("%d ", ptr->data);
ptr = ptr->next;
}
```

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.



Stack

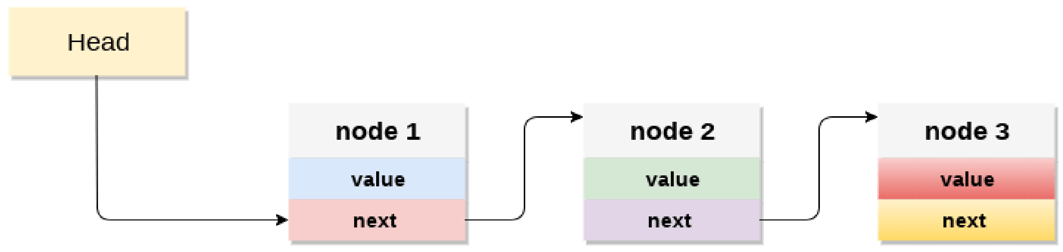
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

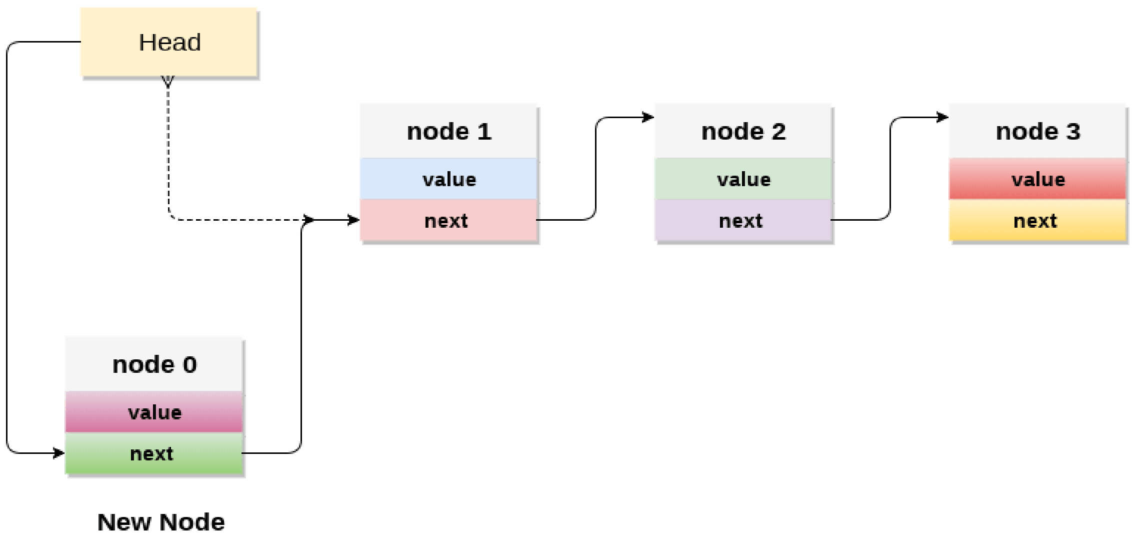
Adding a node to the stack (Push operation):

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 :

```
void push ()
{
    int val;
    struct node *ptr =(struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
            head=ptr;
        }
        else
```

```c
        {
            ptr->val = val;
            ptr->next = head;
            head=ptr;

        }
        printf("Item pushed");

    }
}
```

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 :

1. **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.
2. **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

```c
void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");

    }
}
```

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.

- Copy the head pointer into a temporary pointer.
- 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

```c
void display()
{
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
        printf("Stack is empty\n");
    }
    else
    {
        printf("Printing Stack elements \n");
        while(ptr!=NULL)
        {
            printf("%d\n",ptr->val);
            ptr = ptr->next;
        }
    }
}
```

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

```c
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
```

```
struct node
{
int val;
struct node *next;
};
struct node *head;

void main ()
{
    int choice=0;
    printf("\n*********Stack operations using linked list*********\n");
    printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("\n\nChose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
        default:
        {
            printf("Please Enter valid choice ");
        }
    };
}
}
void push ()
{
    int val;
    struct node *ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
            head=ptr;
        }
        else
        {
            ptr->val = val;
            ptr->next = head;
            head=ptr;

        }
        printf("Item pushed");

    }
}

void pop()
{
    int item;
```

```
struct node *ptr;
if (head == NULL)
{
    printf("Underflow");
}
else
{
    item = head->val;
    ptr = head;
    head = head->next;
    free(ptr);
    printf("Item popped");

}
        }
        void display()
        {
            int i;
            struct node *ptr;
            ptr=head;
            if(ptr == NULL)
            {
                printf("Stack is empty\n");
            }
            else
            {
                printf("Printing Stack elements \n");
                while(ptr!=NULL)
                {
                    printf("%d\n",ptr->val);
                    ptr = ptr->next;
                }
            }
        }
```

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.

1. 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. 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

```
o Step 1: Allocate the space for the new node PTR
o Step 2: SET PTR -> DATA = VAL
o 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]
o Step 4: END
```

C Function

```c
void insert(struct node *ptr, int item; )
{


    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
```

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.

```
ptr = front;
        front = front -> next;
        free(ptr);
```

The algorithm and C function is given as follows.

Algorithm

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

C Function

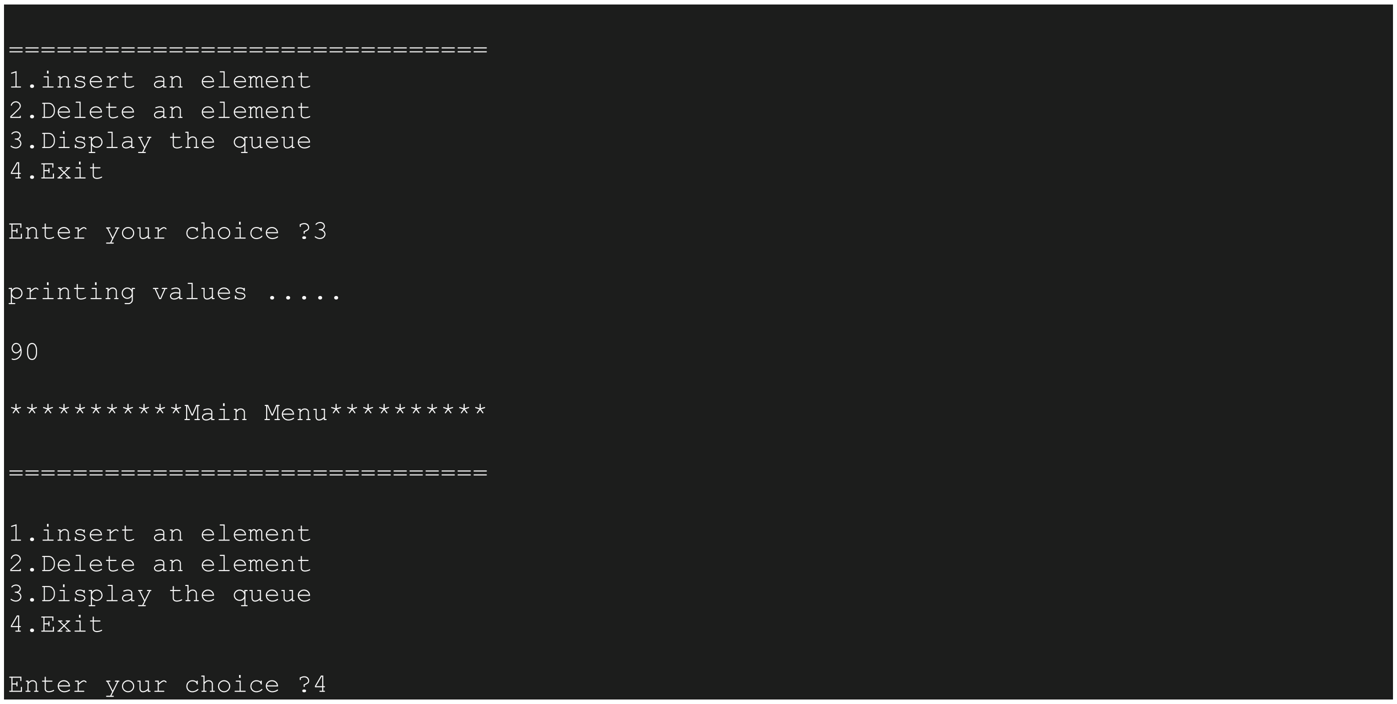
```c
void delete (struct node *ptr)
{
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
```

Menu-Driven Program implementing all the operations on Linked Queue

```c
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n************************Main Menu****************************\n");
        printf("\n==================================================================\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",& choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
    }
}
void insert()
{
    struct node *ptr;
    int item;

    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
void delete ()
{
    struct node *ptr;
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
void display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values .....\n");
        while(ptr != NULL)
        {
                printf("\n%d\n",ptr -> data);
                ptr = ptr -> next;
            }
        }
    }
```

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
```

Polynomials using linked list

linked list is a data structure that stores each element as an object in a node of the list. every note contains two parts data han and links to the next node.

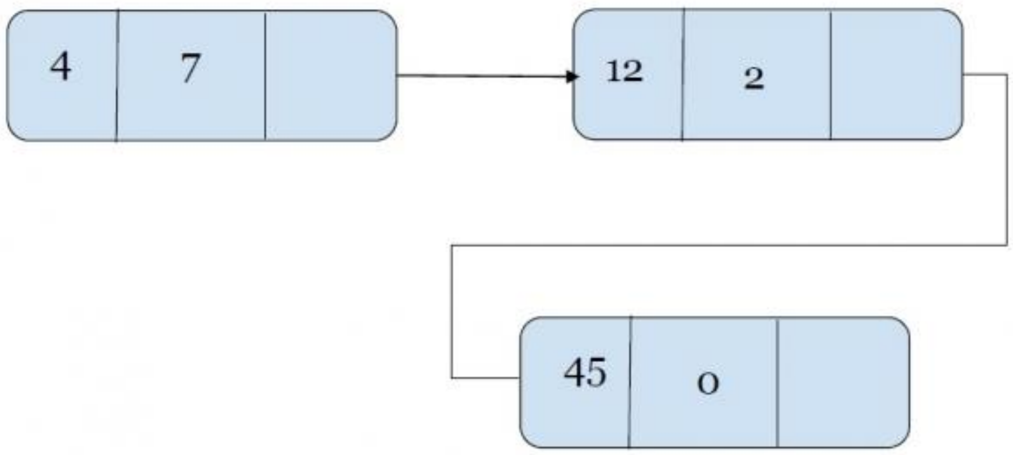
Polynomial is a mathematical expression that consists of variables and coefficients. for example x^2 - 4x + 7

In the **Polynomial linked list**, the coefficients and exponents of the polynomial are defined as the data node of the list.

For adding two polynomials that are stored as a linked list. We need to add the coefficients of variables with the same power. In a linked list node contains 3 members, coefficient value link to the next node.

a linked list that is used to store Polynomial looks like −

Polynomial : $4x^7 + 12x^2 + 45$



This is how a linked list represented polynomial looks like.

Adding two polynomials that are represented by a linked list. We check values at the exponent value of the node. For the same values of exponent, we will add the coefficients.

Example,

Input :

p1= $13x^8 + 7x^5 + 32x^2 + 54$

p2= $3x^{12} + 17x^5 + 3x^3 + 98$

Output : $3x^{12} + 13x^8 + 24x^5 + 3x3 + 32x^2 + 152$

Explanation − For all power, we will check for the coefficients of the exponents that have the same value of exponents and add them. The return the final polynomial.

Algorithm

Input − polynomial p1 and p2 represented as a linked list.

Step 1: loop around all values of linked list and follow step 2& 3.

Step 2: if the value of a node's exponent. is greater copy this node to result node and head towards the next node.

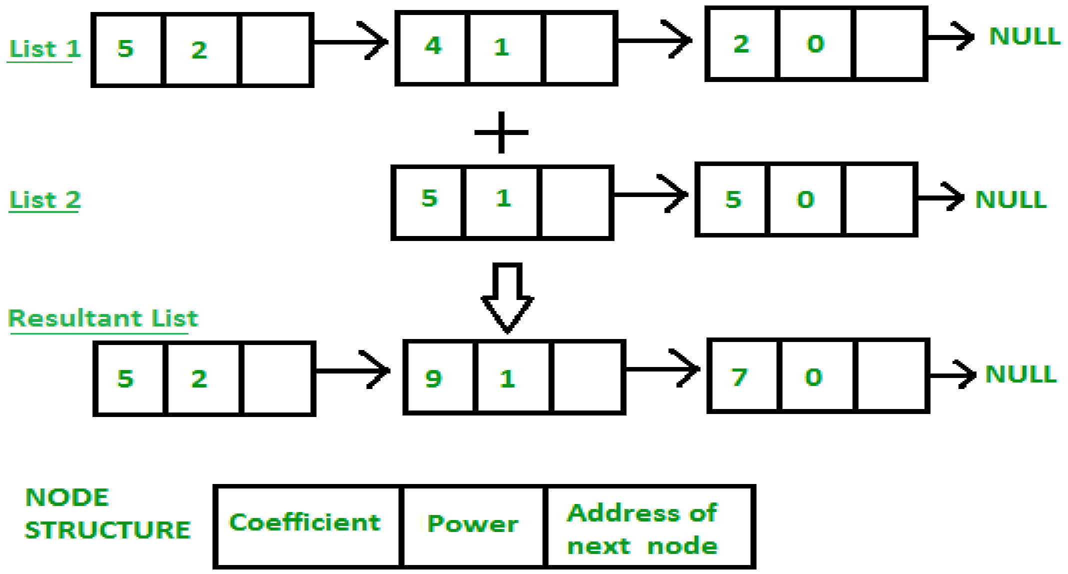
Step 3: if the values of both node's exponent is same add the coefficients and then copy the added value with node to the result.

Step 4: Print the resultant node

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

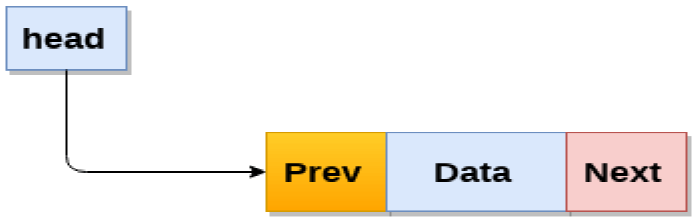
Example:

```
Input:
     1st number = 5x² + 4x¹ + 2x⁰
     2nd number = -5x¹ - 5x⁰
Output:
        5x²-1x¹-3x⁰
Input:
     1st number = 5x³ + 4x² + 2x⁰
     2nd number = 5x^1 - 5x^0
Output:
        5x³ + 4x² + 5x¹ - 3x⁰
```



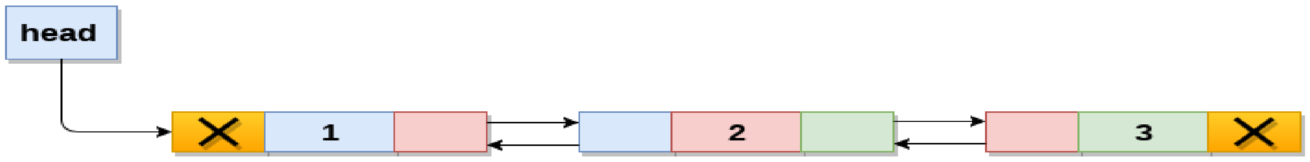
Doubly linked list

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.



Node

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



Doubly Linked List

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

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
}
```

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

Operations on doubly linked list

Node Creation

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
};
struct node *head;
```

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

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	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	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Insertion in doubly linked list at beginning

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

- Allocate the space for the new node in the memory. This will be done by using the following statement.

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

- Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

```
1. ptr->next = NULL;
2.      ptr->prev=NULL;
3.      ptr->data=item;
4.      head=ptr;
```

- In the second scenario, the condition **head == NULL** become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.
- This will be done by using the following statements.

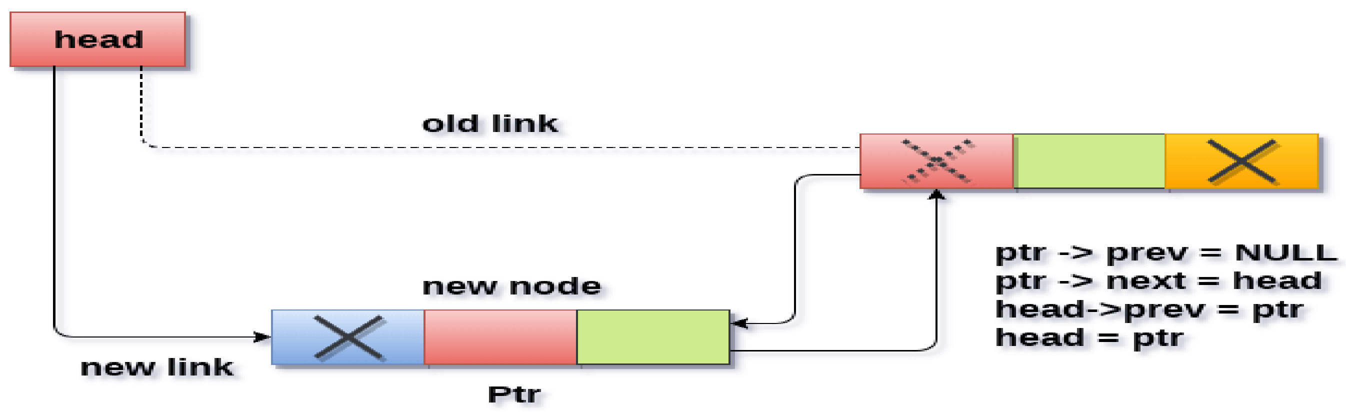
```
1. ptr->next = head;
2.   head→prev=ptr;
```

Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

```
1. ptr→prev =NULL
2. head = ptr
```

Algorithm :

- **Step 1:** IF ptr = NULL
- Write OVERFLOW
  Go to Step 9
  [END OF IF]
- **Step 2:** SET NEW_NODE = ptr
- **Step 3:** SET ptr = ptr -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> PREV = NULL
- **Step 6:** SET NEW_NODE -> NEXT = START
- **Step 7:** SET head -> PREV = NEW_NODE
- **Step 8:** SET head = NEW_NODE
- **Step 9:** EXIT



Insertion into doubly linked list at beginning

Insertion in doubly linked list at the end

In order to insert a node in doubly linked list at the end, we must make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.

- Allocate the memory for the new node. Make the pointer **ptr** point to the new node being inserted.

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

- Check whether the list is empty or not. The list is empty if the condition **head == NULL** holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

```
1. ptr->next = NULL;
   2.          ptr->prev=NULL;
3.   ptr->data=item;
4.   head=ptr;
```

- In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer **temp** to head and traverse the list by using this pointer.

```
1. Temp = head;
2. while (temp != NULL)
3. {
4. temp = temp → next;
5.   }
```

the pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e. ptr.

```
1. temp→next =ptr;
```

make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.

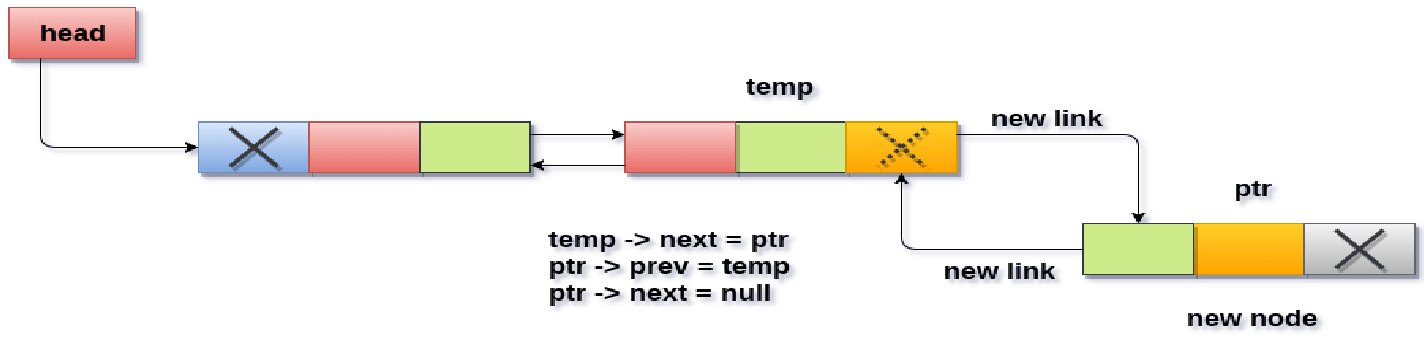
```
1. ptr → prev = temp;
```

make the next pointer of the node ptr point to the null as it will be the new last node of the list.

```
1.          ptr → next = NULL
```

Algorithm

```
• Step 1: IF PTR = NULL
• Write OVERFLOW
    Go to Step 11
   [END OF IF]
• Step 2: SET NEW_NODE = PTR
• Step 3: SET PTR = PTR -> NEXT
• Step 4: SET NEW_NODE -> DATA = VAL
• Step 5: SET NEW_NODE -> NEXT = NULL
• Step 6: SET TEMP = START
• Step 7: Repeat Step 8 while TEMP -> NEXT != NULL
• Step 8: SET TEMP = TEMP -> NEXT
        ▪ [END OF LOOP]
• Step 9: SET TEMP -> NEXT = NEW_NODE
• Step 10C: SET NEW_NODE -> PREV = TEMP
• Step 11: EXIT
```



Insertion into doubly linked list at the end

Insertion in doubly linked list after Specified node

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Use the following steps for this purpose.

- Allocate the memory for the new node. Use the following statements for this.

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

- Traverse the list by using the pointer **temp** to skip the required number of nodes in order to reach the specified node.

```
1.  temp=head;
2.      for(i=0;i<loc;i++)
```

```
{
    temp = temp->next;
    if(temp == NULL) // the temp will be //null if the list doesn't last long //up to mentioned location
    {
        return;
    }
}
```

- The temp would point to the specified node at the end of the **for** loop. The new node needs to be inserted after this node therefore we need to make a fer pointer adjustments here. Make the next pointer of **ptr** point to the next node of temp.

1. ptr → next = temp → next;

make the **prev** of the new node ptr point to temp.

1. ptr → prev = temp;

make the **next** pointer of temp point to the new node ptr.

1. temp → next = ptr;

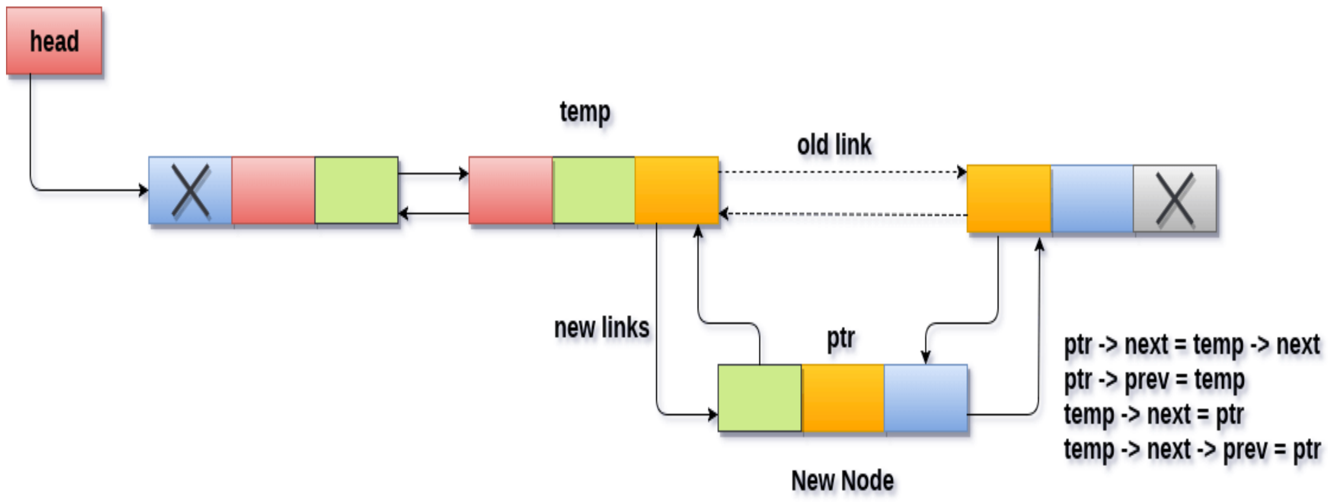
make the **previous** pointer of the next node of temp point to the new node.

1. temp → next → prev = ptr;

Algorithm

```
o Step 1: IF PTR = NULL
    Write OVERFLOW
    Go to Step 15
  [END OF IF]
o Step 2: SET NEW_NODE = PTR
o Step 3: SET PTR = PTR -> NEXT
o Step 4: SET NEW_NODE -> DATA = VAL
o Step 5: SET TEMP = START
o Step 6: SET I = 0
o Step 7: REPEAT 8 to 10 until I<="" li="">
o Step 8: SET TEMP = TEMP -> NEXT
o STEP 9: IF TEMP = NULL
o STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
    GOTO STEP 15
    [END OF IF]
  [END OF LOOP]
```

- **Step 11:** SET NEW_NODE -> NEXT = TEMP -> NEXT
- **Step 12:** SET NEW_NODE -> PREV = TEMP
- **Step 13 :** SET TEMP -> NEXT = NEW_NODE
- **Step 14:** SET TEMP -> NEXT -> PREV = NEW_NODE
- **Step 15:** EXIT



Insertion into doubly linked list after specified node

Deletion at beginning

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

1. Ptr = head;
2. head = head → next;

now make the prev of this new head node point to NULL. This will be done by using the following statements.

1. head → prev = NULL

Now free the pointer ptr by using the **free** function.

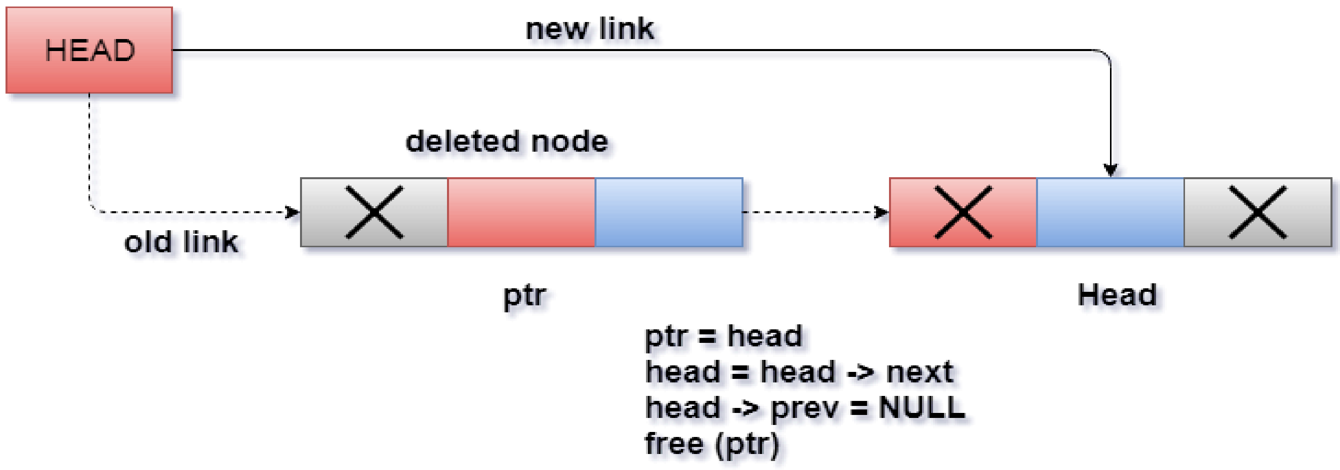
1. free(ptr)

Algorithm

- **STEP 1:** IF HEAD = NULL

  WRITE UNDERFLOW
  GOTO STEP 6

- **STEP 2:** SET PTR = HEAD
- **STEP 3:** SET HEAD = HEAD → NEXT
- **STEP 4:** SET HEAD → PREV = NULL
- **STEP 5:** FREE PTR
- **STEP 6:** EXIT



Deletion in doubly linked list from beginning

Deletion in doubly linked list at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

- If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.
- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
    if(ptr->next != NULL)
    {
        ptr = ptr -> next;
    }
```

- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of **ptr** to **NULL**.

1. ptr → prev → next = NULL

free the pointer as this the node which is to be deleted.

1. free(ptr)

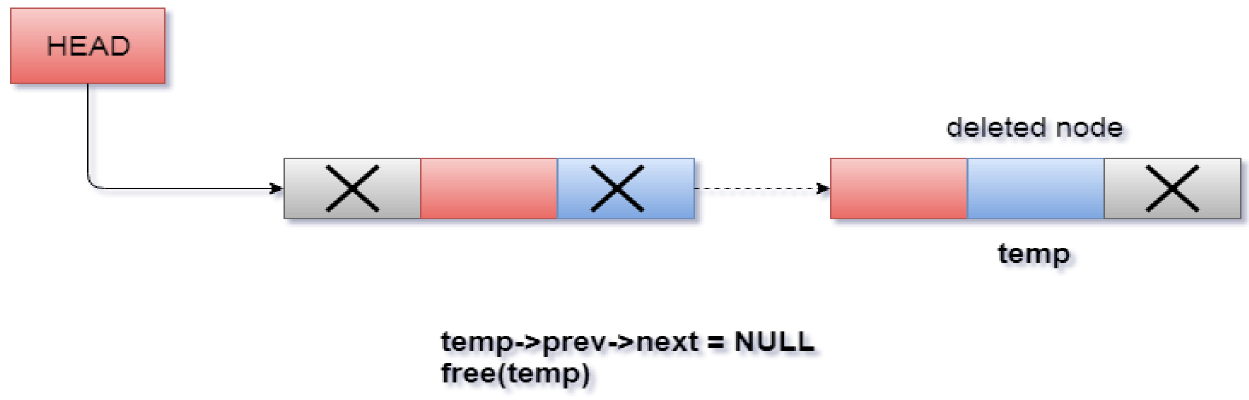
- **Step 1:** IF HEAD = NULL

Write UNDERFLOW
Go to Step 7
[END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** REPEAT STEP 4 WHILE TEMP->NEXT != NULL
- **Step 4:** SET TEMP = TEMP->NEXT

[END OF LOOP]

- **Step 5:** SET TEMP ->PREV-> NEXT = NULL
- **Step 6:** FREE TEMP
- **Step 7:** EXIT



Deletion in doubly linked list at the end

Deletion in doubly linked list after the specified node

In order to delete the node after the specified data, we need to perform the following steps.

- Copy the head pointer into a temporary pointer temp.

```
temp = head
```

- Traverse the list until we find the desired data value.

```
while(temp -> data != val)
temp = temp -> next;
```

- Check if this is the last node of the list. If it is so then we can't perform deletion.

```
if(temp -> next == NULL)
   {
return;
   }
```

- Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.

```
if(temp -> next -> next == NULL)
   {
       temp ->next = NULL;
   }
```

- Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

```
ptr = temp -> next;
       temp -> next = ptr -> next;
       ptr -> next -> prev = temp;
       free(ptr);
```

Algorithm

- **Step 1:** IF HEAD = NULL

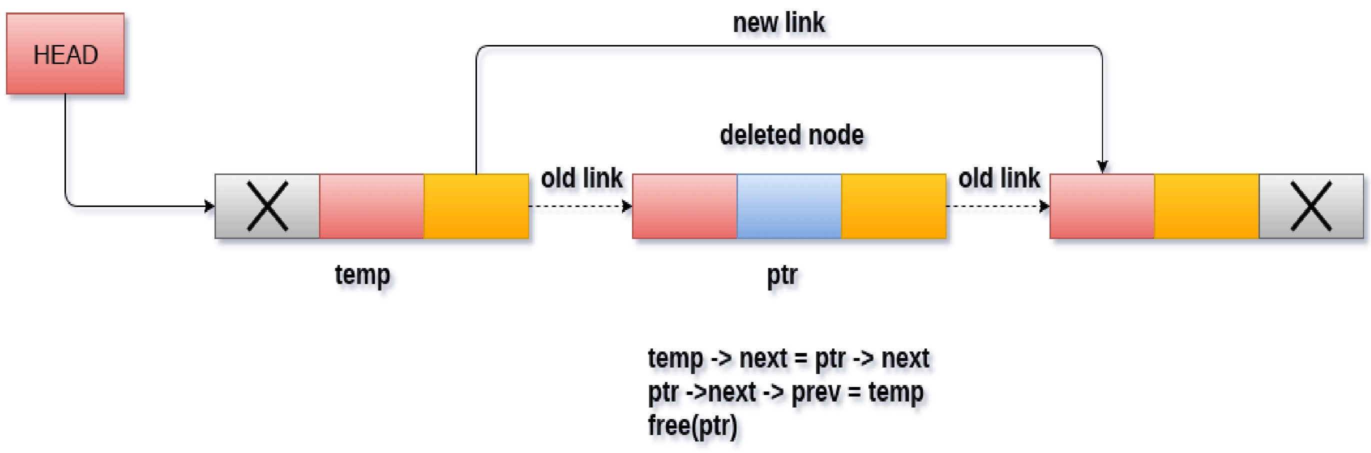
  Write UNDERFLOW
  Go to Step 9
  [END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** Repeat Step 4 while TEMP -> DATA != ITEM
- **Step 4:** SET TEMP = TEMP -> NEXT

  [END OF LOOP]

- **Step 5:** SET PTR = TEMP -> NEXT
- **Step 6:** SET TEMP -> NEXT = PTR -> NEXT

- **Step 7:** SET PTR -> NEXT -> PREV = TEMP
- **Step 8:** FREE PTR
- **Step 9:** EXIT



Deletion of a specified node in doubly linked list

Searching for a specific node in Doubly Linked List

We just need traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

- Copy head pointer into a temporary pointer variable ptr.

1. ptr = head

- declare a local variable I and assign it to 0.

1. i=0

- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm

- **Step 1:** IF HEAD == NULL

```
 WRITE "UNDERFLOW"
GOTO STEP 8
[END OF IF]
```

```
o Step 2: Set PTR = HEAD
o Step 3: Set i = 0
o Step 4: Repeat step 5 to 7 while PTR != NULL
o Step 5: IF PTR → data = item
      return i
      [END OF IF]
o Step 6: i = i + 1
o Step 7: PTR = PTR → next
o Step 8: Exit
```

Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

```
Ptr = head
```

then, traverse through the list by using while loop. Keep shifting value of pointer variable **ptr** until we find the last node. The last node contains **null** in its next part.

```
while(ptr != NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
```

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm

```
o Step 1: IF HEAD == NULL
       WRITE "UNDERFLOW"
      GOTO STEP 6
      [END OF IF]
o Step 2: Set PTR = HEAD
o Step 3: Repeat step 4 and 5 while PTR != NULL
o Step 4: Write PTR → data
o Step 5: PTR = PTR → next
```

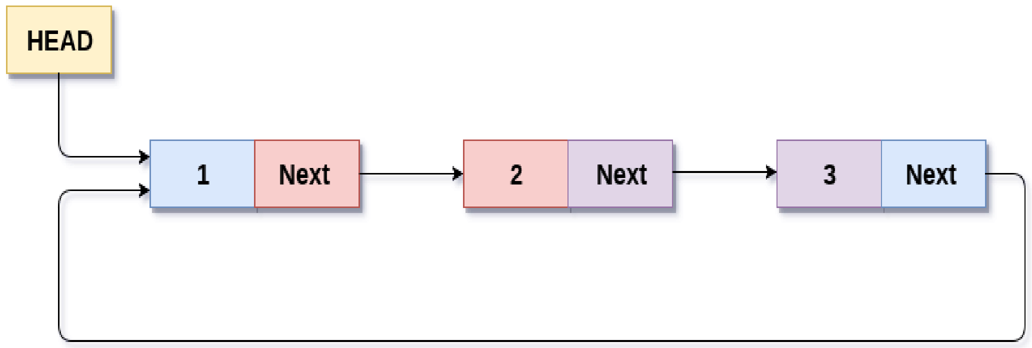
- **Step 6:** Exit

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular Singly Linked List

Operations on Circular Singly linked list:

Insertion

SN	Operation	Description
1	**Insertion at beginning**	Adding a node into circular singly linked list at the beginning.
2	**Insertion at the end**	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.

3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Insertion into circular singly linked list at beginning

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

Firstly, allocate the memory space for the new node by using the malloc method of C language.

```
struct node *ptr = (struct node *)malloc(sizeof(struct node));
```

In the first scenario, the condition **head == NULL** will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

```
if(head == NULL)
    {
        head = ptr;
        ptr -> next = head;
    }
```

In the second scenario, the condition head == NULL will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

```
temp = head;
        while(temp->next != head)
            temp = temp->next;
```

At the end of the loop, the pointer temp would point to the last node of the list. Since, in a circular singly linked list, the last node of the list contains a pointer to the first node of the list. Therefore, we need to make the next pointer of the last node point to the head node of the list and the new node which is being inserted into the list will be the new head node of the list therefore the next pointer of temp will point to the new node ptr.

This will be done by using the following statements.

```
temp -> next = ptr;
```

the next pointer of temp will point to the existing head node of the list.

```
1.  ptr->next = head;
```

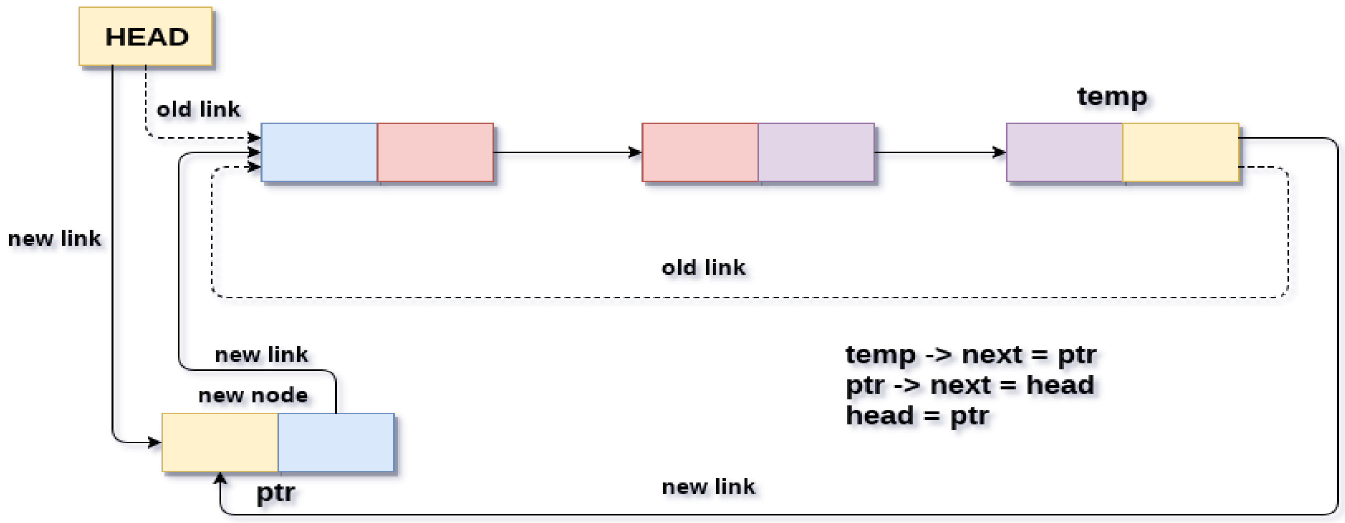
Now, make the new node ptr, the new head node of the circular singly linked list.

```
1.  head = ptr;
```

in this way, the node ptr has been inserted into the circular singly linked list at beginning.

Algorithm

```
o  Step 1: IF PTR = NULL

      Write OVERFLOW
     Go to Step 11
     [END OF IF]

o  Step 2: SET NEW_NODE = PTR
o  Step 3: SET PTR = PTR -> NEXT
o  Step 4: SET NEW_NODE -> DATA = VAL
o  Step 5: SET TEMP = HEAD
o  Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD
o  Step 7: SET TEMP = TEMP -> NEXT

   [END OF LOOP]

o  Step 8: SET NEW_NODE -> NEXT = HEAD
o  Step 9: SET TEMP → NEXT = NEW_NODE
o  Step 10: SET HEAD = NEW_NODE
o  Step 11: EXIT
```



Insertion into circular singly linked list at beginning

Insertion into circular singly linked list at the end

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

- Firstly, allocate the memory space for the new node by using the malloc method of C language.

```
struct node *ptr = (struct node *)malloc(sizeof(struct node));
```

In the first scenario, the condition **head == NULL** will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

```
if(head == NULL)
    {
       head = ptr;
       ptr -> next = head;
    }
```

In the second scenario, the condition **head == NULL** will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

```
temp = head;
       while(temp->next != head)
         temp = temp->next;
```

At the end of the loop, the pointer temp would point to the last node of the list. Since, the new node which is being inserted into the list will be the new last node of the list. Therefore the existing last node i.e. **temp** must point to the new node **ptr**. This is done by using the following statement.

```
temp -> next = ptr;
```

The new last node of the list i.e. ptr will point to the head node of the list.

```
ptr -> next = head;
```

In this way, a new node will be inserted in a circular singly linked list at the beginning.

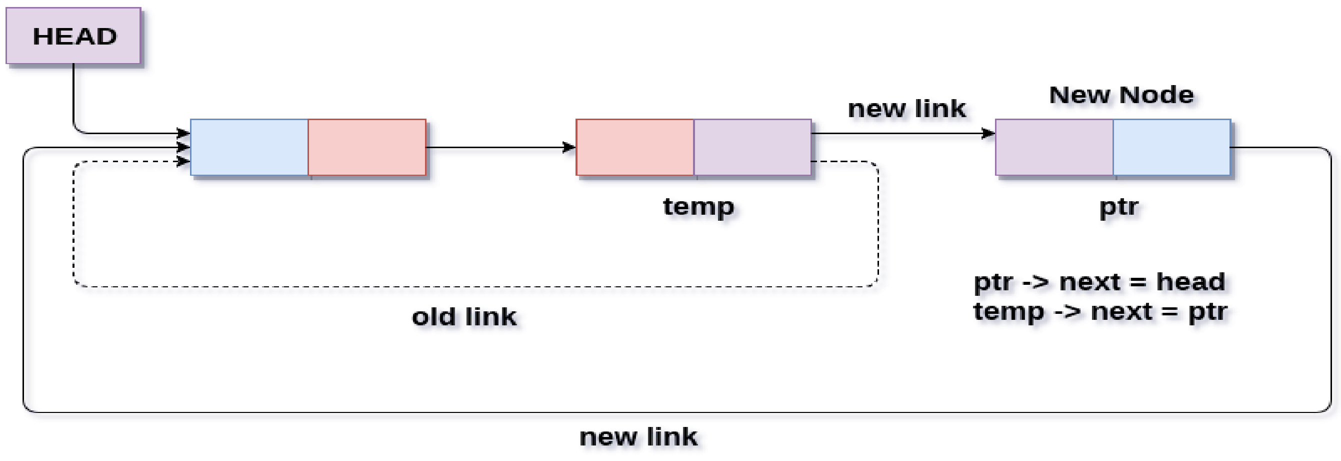
Algorithm

```
○ Step 1: IF PTR = NULL

      Write OVERFLOW
       Go to Step 1
      [END OF IF]
```

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> NEXT = HEAD
- **Step 6:** SET TEMP = HEAD
- **Step 7:** Repeat Step 8 while TEMP -> NEXT != HEAD
- **Step 8:** SET TEMP = TEMP -> NEXT

  [END OF LOOP]

- **Step 9:** SET TEMP -> NEXT = NEW_NODE
- **Step 10:** EXIT



Insertion into circular singly linked list at end

Deletion in circular singly linked list at beginning

In order to delete a node in circular singly linked list, we need to make a few pointer adjustments.

There are three scenarios of deleting a node from circular singly linked list at beginning.

Scenario 1: (The list is Empty)

If the list is empty then the condition **head == NULL** will become true, in this case, we just need to print **underflow** on the screen and make exit.

```
if(head == NULL)
  {
     printf("\nUNDERFLOW");
     return;
}
```

Scenario 2: (The list contains single node)

If the list contains single node then, the condition **head → next == head** will become true. In this case, we need to delete the entire list and make the head pointer free. This will be done by using the following statements.

```
if(head->next == head)
{
    head = NULL;
    free(head);
}
```

Scenario 3: (The list contains more than one node)

If the list contains more than one node then, in that case, we need to traverse the list by using the pointer **ptr** to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
        while(ptr -> next != head)
            ptr = ptr -> next;
```

At the end of the loop, the pointer ptr point to the last node of the list. Since, the last node of the list points to the head node of the list. Therefore this will be changed as now, the last node of the list will point to the next of the head node.

```
ptr->next = head->next;
```

Now, free the head pointer by using the free() method in C language.

```
free(head);
```

Make the node pointed by the next of the last node, the new head of the list.

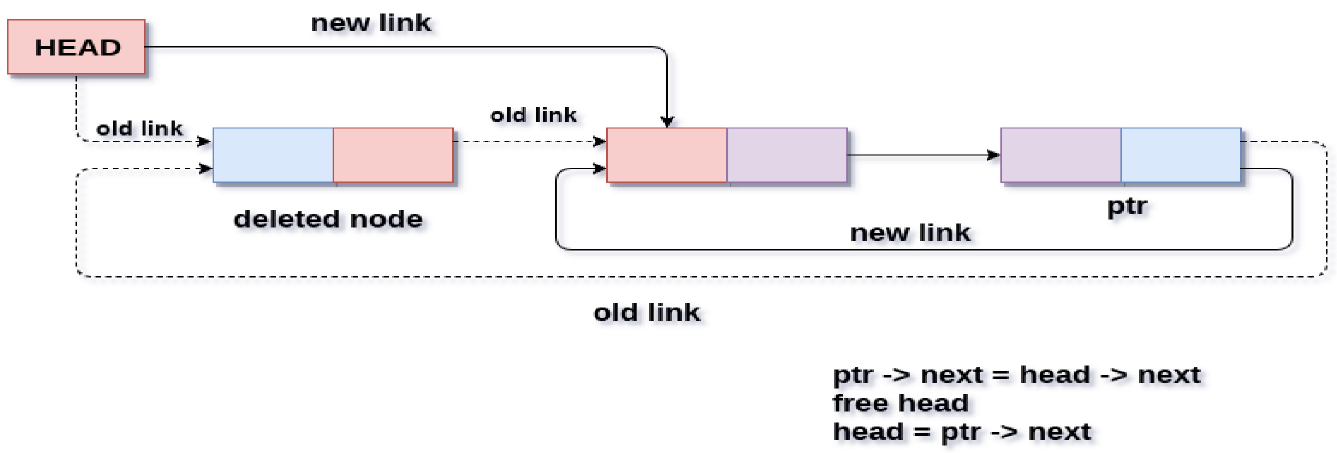
```
head = ptr->next;
```

In this way, the node will be deleted from the circular singly linked list from the beginning.

Algorithm

```
o   Step 1: IF HEAD = NULL
        Write UNDERFLOW
          Go to Step 8
         [END OF IF]
```

```
o Step 2: SET PTR = HEAD
o Step 3: Repeat Step 4 while PTR → NEXT != HEAD
o Step 4: SET PTR = PTR → next
      [END OF LOOP]
o Step 5: SET PTR → NEXT = HEAD → NEXT
o Step 6: FREE HEAD
o Step 7: SET HEAD = PTR → NEXT
o Step 8: EXIT
```



Deletion in circular singly linked list at beginning

Deletion in Circular singly linked list at the end

There are three scenarios of deleting a node in circular singly linked list at the end.

Scenario 1 (the list is empty)

If the list is empty then the condition **head == NULL** will become true, in this case, we just need to print **underflow** on the screen and make exit.

```
if(head == NULL)
  {
     printf("\nUNDERFLOW");
     return;
  }
```

Scenario 2(the list contains single element)

If the list contains single node then, the condition **head → next == head** will become true. In this case, we need to delete the entire list and make the head pointer free. This will be done by using the following statements.

```
if(head->next == head)
{
   head = NULL;
   free(head);
}
```

Scenario 3(the list contains more than one element)

If the list contains more than one element, then in order to delete the last element, we need to reach the last node. We also need to keep track of the second last node of the list. For this purpose, the two pointers ptr and preptr are defined. The following sequence of code is used for this purpose.

```
while(ptr ->next != head)
      {
         preptr=ptr;
         ptr = ptr->next;
      }
```

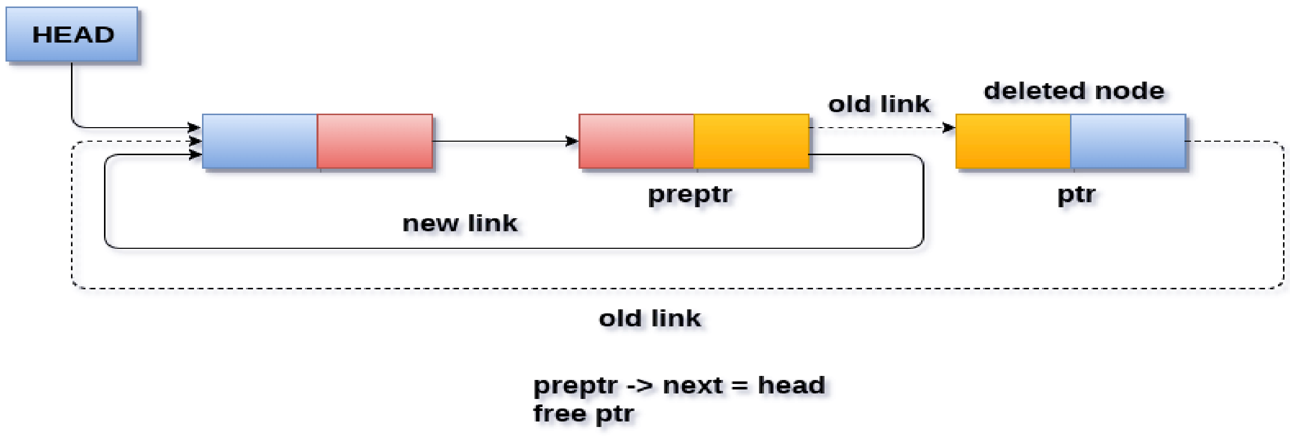
now, we need to make just one more pointer adjustment. We need to make the next pointer of preptr point to the next of ptr (i.e. head) and then make pointer ptr free.

```
preptr->next = ptr -> next;
      free(ptr);
```

Algorithm

```
○ Step 1: IF HEAD = NULL

     Write UNDERFLOW
      Go to Step 8
     [END OF IF]

○ Step 2: SET PTR = HEAD
○ Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != HEAD
○ Step 4: SET PREPTR = PTR
○ Step 5: SET PTR = PTR -> NEXT

   [END OF LOOP]
```

- **Step 6:** SET PREPTR -> NEXT = HEAD
- **Step 7:** FREE PTR
- **Step 8:** EXIT



Deletion in circular singly linked list at end

Searching in circular singly linked list

Searching in circular singly linked list needs traversing across the list. The item which is to be searched in the list is matched with each node data of the list once and if the match found then the location of that item is returned otherwise -1 is returned.

The algorithm and its implementation in C is given as follows.

Algorithm

```
o Step 1: SET PTR = HEAD
o Step 2: Set I = 0
o STEP 3: IF PTR = NULL

      WRITE "EMPTY LIST"
      GOTO STEP 8
      END OF IF

o STEP 4: IF HEAD → DATA = ITEM

      WRITE i+1 RETURN [END OF IF]
o STEP 5: REPEAT STEP 5 TO 7 UNTIL PTR->next != head
o STEP 6: if ptr → data = item
```

```
        write i+1
        RETURN
        End of IF

  o STEP 7: I = I + 1
  o STEP 8: PTR = PTR → NEXT

        [END OF LOOP]

  o STEP 9: EXIT
```

Traversing in Circular Singly linked list

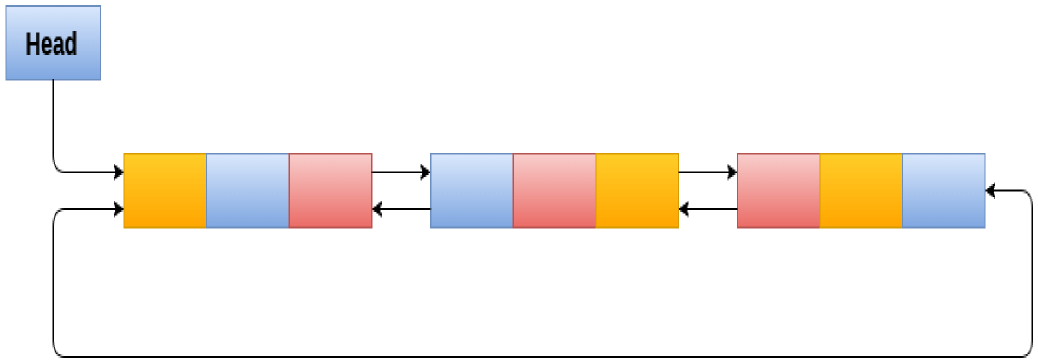
Traversing in circular singly linked list can be done through a loop. Initialize the temporary pointer variable **temp** to head pointer and run the while loop until the next pointer of temp becomes **head**. The algorithm and the c function implementing the algorithm is described as follows.

Algorithm

```
  o STEP 1: SET PTR = HEAD
  o STEP 2: IF PTR = NULL

        WRITE "EMPTY LIST"
        GOTO STEP 8
        END OF IF

  o STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR → NEXT != HEAD
  o STEP 5: PRINT PTR → DATA
  o STEP 6: PTR = PTR → NEXT

        [END OF LOOP]

  o STEP 7: PRINT PTR→ DATA
  o STEP 8: EXIT
```

Circular Doubly Linked List

- Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.
- A circular doubly linked list is shown in the following figure.



Circular Doubly Linked List

Operations on circular doubly linked list :

There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list. However, the operations on circular doubly linked list is described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

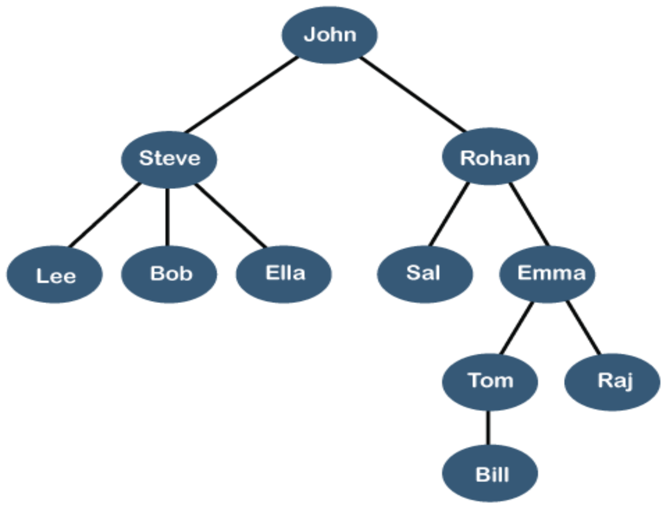
Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.

UNIT-5

UNIT-V

Tree Data Structure

Tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



The above tree shows the **organization hierarchy** of some company. In the above structure, ***john*** is the **CEO** of the company, and John has two direct reports named as ***Steve*** and ***Rohan***. Steve has three direct reports named ***Lee, Bob, Ella*** where ***Steve*** is a manager. Bob has two direct reports named ***Sal*** and ***Emma***. **Emma** has two direct reports named ***Tom*** and ***Raj***. Tom has one direct report named ***Bill***. This particular logical structure is known as a ***Tree***. Its structure is similar to the real tree, so it is named a ***Tree***. In this structure, the ***root*** is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data in a hierarchical way.

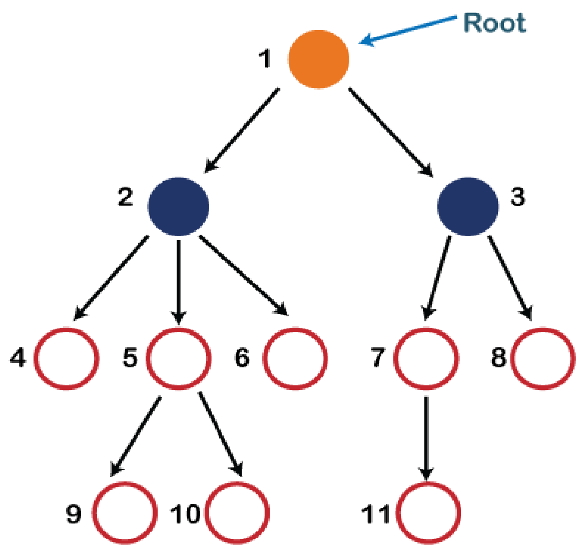
Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.

Some basic terms used in Tree data structure

Let's consider the tree structure, which is shown below:

Introduction to Trees



In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

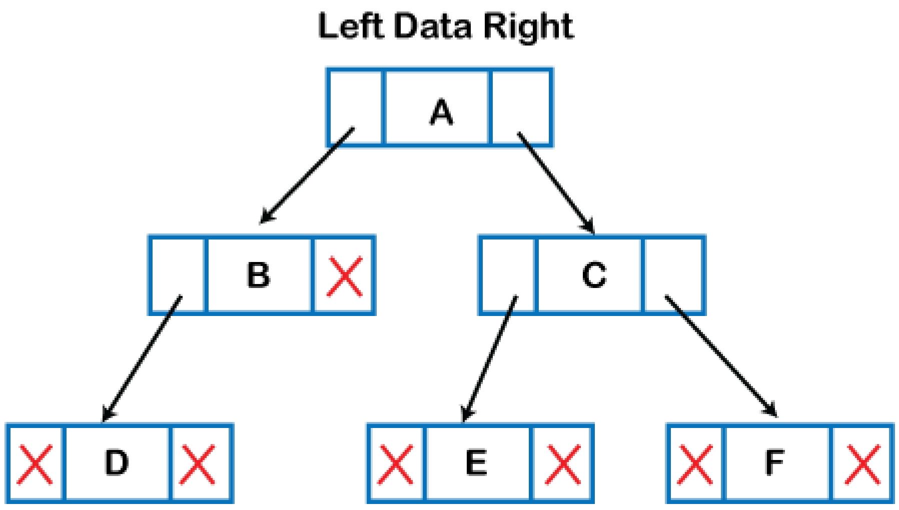
- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A 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.
- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

```
struct node
{
  int data;
struct node *left;
struct node *right;
} ;
```

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Applications of trees

The following are the applications of trees:

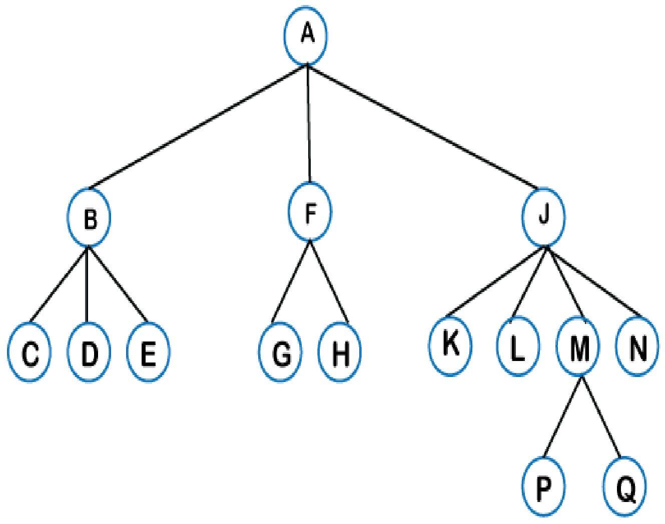
- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.

- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

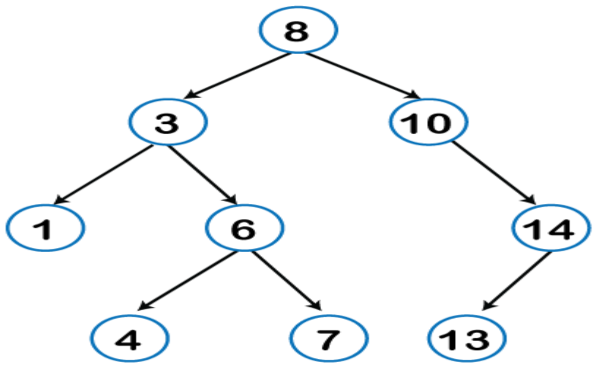
The following are the types of a tree data structure:

- **General tree:** The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.



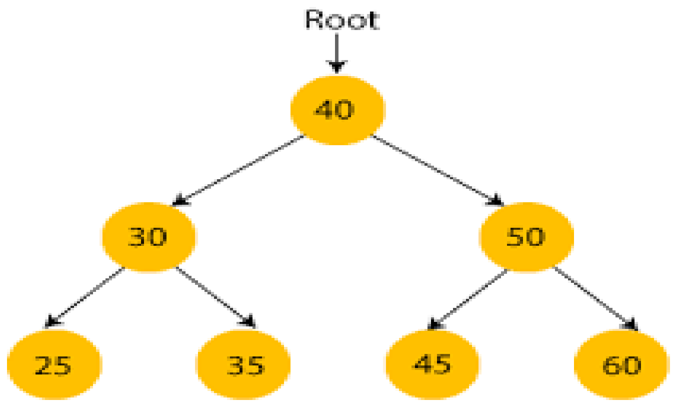
There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.
Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

- **Binary tree:** Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



- **Binary Search tree:**

Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer).
Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.



A node can be created with the help of a user-defined data type known as ***struct,*** as shown below:

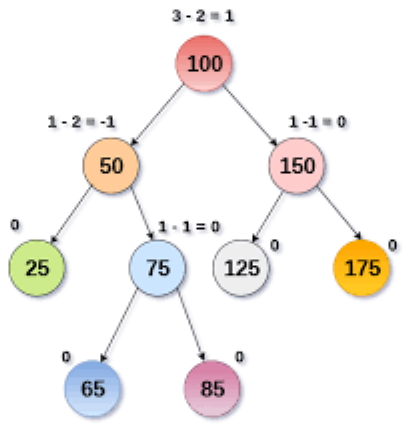
```
1.struct node
2.{
3.    int data;
4.    struct node *left;
5.struct node *right;
6.}
```

The above is the node structure with three fields: data field, the second field is the left pointer of the node type, and the third field is the right pointer of the node type.

AVL tree

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the ***binary tree*** as well as of the ***binary search tree***. It is a self-balancing binary search tree that was invented by ***Adelson Velsky Lindas***. Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the ***balancing factor***.

Balance Factor (k) = height (left(k)) - height (right(k))



AVL Tree

We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the ***difference between the height of the left subtree and the height of the right subtree***. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.

- ## Red-Black Tree

The red-Black tree is the binary search tree. The prerequisite of the Red-Black tree is that we should know about the binary search tree. In a binary search tree, the value of the left-subtree should be less than the value of that node, and the value of the right-subtree should be greater than the value of that node. As we know that the time complexity of binary search in the average case is $\log_2 n$, the best case is O(1), and the worst case is O(n).

When any operation is performed on the tree, we want our tree to be balanced so that all the operations like searching, insertion, deletion, etc., take less time, and all these operations will have the time complexity of ***$\log_2 n$*.**

The red-black tree is a self-balancing binary search tree. AVL tree is also a height balancing binary search tree then **why do we require a Red-Black tree**. In the AVL tree, we do not know how many rotations would be required to balance the tree, but in the Red-black tree, a maximum of 2 rotations are required to balance the tree. It contains one extra bit that represents either the red or black color of a node to ensure the balancing of the tree.

- ## Splay tree

The splay tree data structure is also binary search tree in which recently accessed element is placed at the root position of tree by performing some rotation operations. Here, ***splaying*** means the recently accessed node. It is a ***self-balancing*** binary search tree having no explicit balance condition like **AVL** tree.

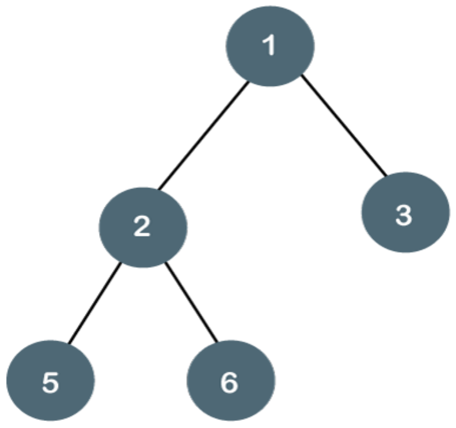
It might be a possibility that height of the splay tree is not balanced, i.e., height of both left and right subtrees may differ, but the operations in splay tree takes order of **logN** time where **n** is the number of nodes.

Splay tree is a balanced tree but it cannot be considered as a height balanced tree because after each operation, rotation is performed which leads to a balanced tree.

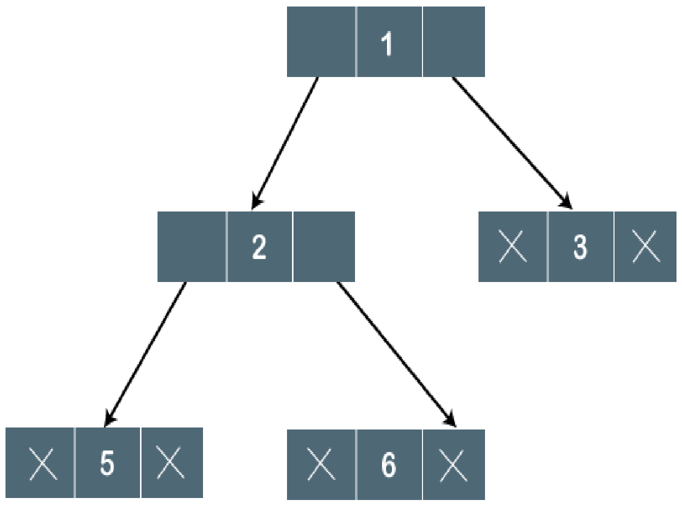
Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots . 2^h) = 2^{h+1} -1$.
- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

Types of Binary Tree

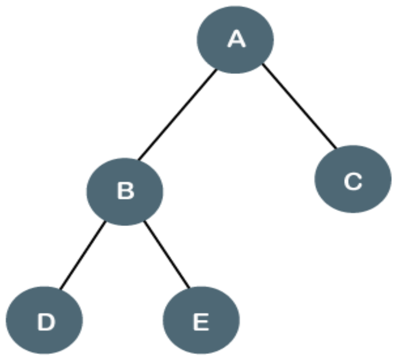
There are four types of Binary tree:

- **Full/ proper/ strict Binary tree**
- **Complete Binary tree**
- **Perfect Binary tree**
- **Degenerate Binary tree**
- **Balanced Binary tree**

1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.



In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

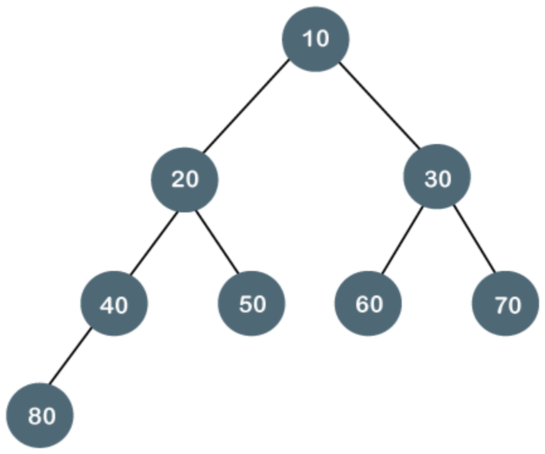
Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2^{h+1} -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is **$\log_2(n+1) - 1$.**

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.



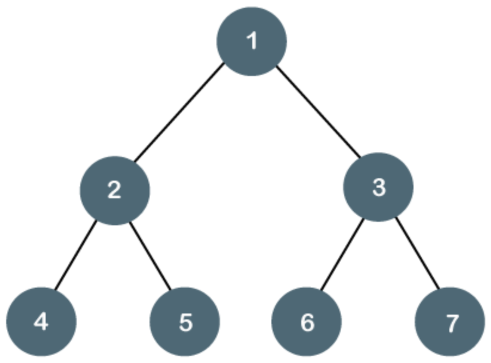
The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree:

- The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.
- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of a complete binary tree is

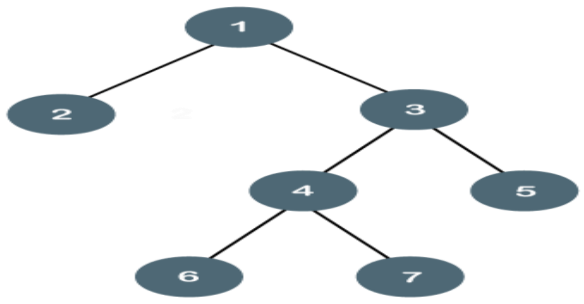
Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



Let's look at a simple example of a perfect binary tree.

The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.

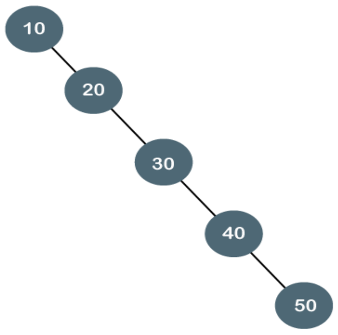


Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.

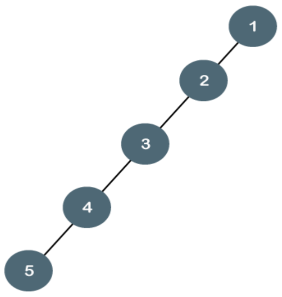
Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.

Let's understand the Degenerate binary tree through examples.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

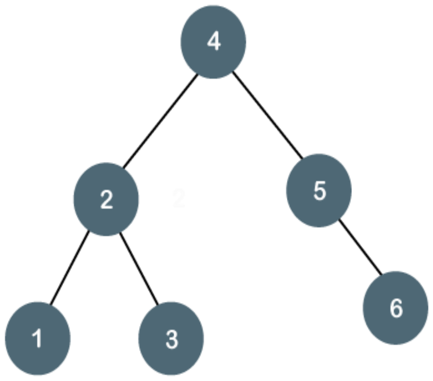


The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

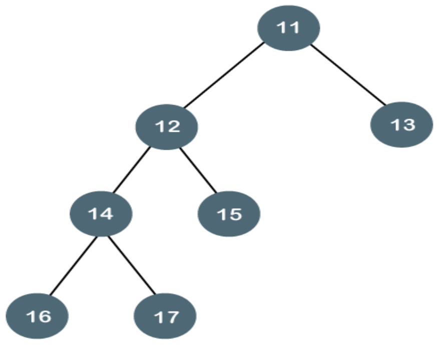
Balanced Binary Tree:

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

Let's understand the balanced binary tree through examples.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is 0.



The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

Binary Tree Implementation

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

```
struct node
{
  int data,
  struct node *left, *right;
}
```

In the above structure, **data** is the value, **left pointer** contains the address of the left node, and **right pointer** contains the address of the right node.

Binary Tree program in C

```
#include<stdio.h>
     struct node
    {
         int data;
         struct node *left, *right;
     }
     void main()
   {
      struct node *root;
      root = create();
   }
struct node *create()
{
   struct node *temp;
```

```
  int data;
  temp = (struct node *)malloc(sizeof(struct node));
  printf("Press 0 to exit");
  printf("\nPress 1 for new node");
  printf("Enter your choice : ");
  scanf("%d", &choice);
  if(choice==0)
{
return 0;
}
else
{
  printf("Enter the data:");
  scanf("%d", &data);
  temp->data = data;
  printf("Enter the left child of %d", data);
  temp->left = create();
printf("Enter the right child of %d", data);
temp->right = create();
return temp;
}
}
```

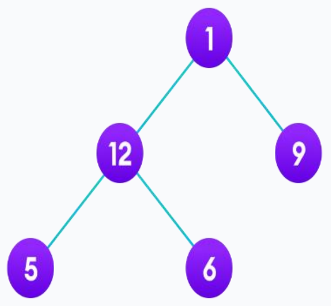
The above code is calling the create() function recursively and creating new node on each recursive call. When all the nodes are created, then it forms a binary tree structure. The process of visiting the nodes is known as tree traversal. There are three types traversals used to visit a node:

- **Inorder traversal**
- **Preorder traversal**
- **Postorder traversal**

Binary Tree Traversal - inorder, preorder and postorder

Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.



Starting from top, Left to right

```
1 -> 12 -> 5 -> 6 -> 9
```

Starting from bottom, Left to right

```
5 -> 6 -> 12 -> 9 -> 1
```

Although this process is somewhat easy, it doesn't respect the hierarchy of the tree, only the depth of the nodes.

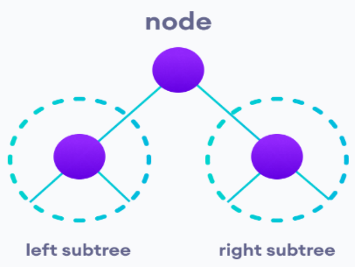
Instead, we use traversal methods that take into account the basic structure of a tree i.e.

```
struct node {
    int data;
    struct node* left;
    struct node* right;
}
```

The struct node pointed to by `left` and `right` might have other left and right children so we should think of them as sub-trees instead of sub-nodes.

According to this structure, every tree is a combination of

- A node carrying data
- Two subtrees



Left and Right Subtree

Remember that our goal is to visit each node, so we need to visit all the nodes in the subtree, visit the root node and visit all the nodes in the right subtree as well.

Depending on the order in which we do this, there can be three types of traversal

Inorder traversal:

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

```
inorder(root->left)
display(root->data)
inorder(root->right)
```

Preorder traversal

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

```
display(root->data)
preorder(root->left)
preorder(root->right)
```

Postorder traversal

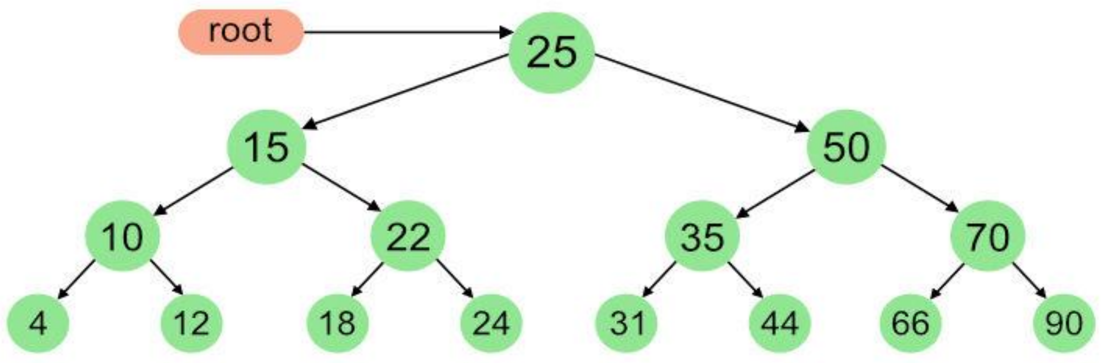
1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

```
postorder(root->left)
postorder(root->right)
display(root->data)
```

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



```
/ Tree traversal in C

#include <stdio.h>
#include <stdlib.h>

struct node {
  int item;
  struct node* left;
  struct node* right;
};

// Inorder traversal
void inorderTraversal(struct node* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);
}

// preorderTraversal traversal
void preorderTraversal(struct node* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}
```

```
// postorderTraversal traversal
void postorderTraversal(struct node* root) {
  if (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ->", root->item);
}

// Create a new Node
struct node* createNode(value) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}

// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
  root->left = createNode(value);
  return root->left;
}

// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
  root->right = createNode(value);
  return root->right;
}

int main() {
  struct node* root = createNode(1);
  insertLeft(root, 12);
  insertRight(root, 9);

  insertLeft(root->left, 5);
  insertRight(root->left, 6);

  printf("Inorder traversal \n");
  inorderTraversal(root);

  printf("\nPreorder traversal \n");
  preorderTraversal(root);

  printf("\nPostorder traversal \n");
  postorderTraversal(root);
}
```

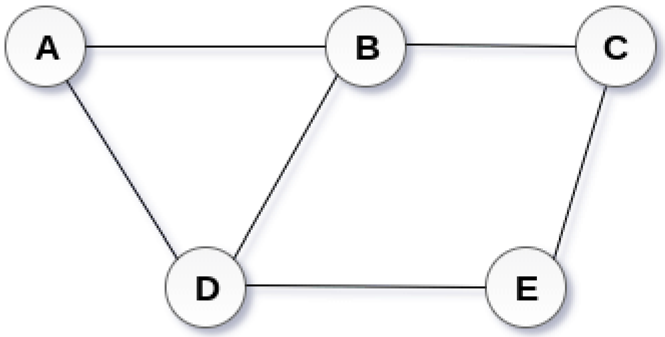
Graph

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.



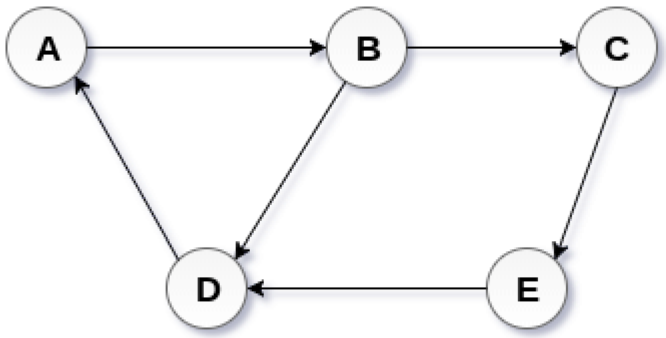
Undirected Graph

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.

Graph representation

In this article, we will discuss the ways to represent the graph. By Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.

A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

- **Sequential representation** (or, Adjacency matrix representation)
- **Linked list representation** (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

In this tutorial, we will discuss each one of them in detail.

ow, let's start discussing the ways of representing a graph in the data structure.

Sequential representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

An entry A_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between V_i and V_j. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

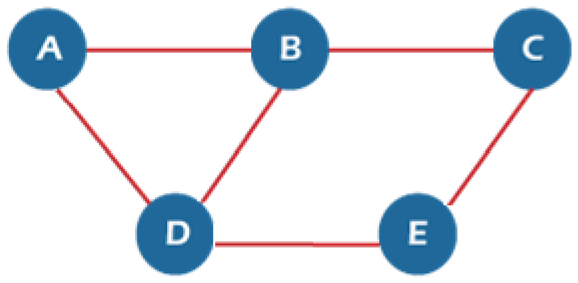
a_{ij} = 1 {if there is a path exists from V_i to V_j}

a_{ij} = 0 {Otherwise}

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0

Now, let's see the adjacency matrix representation of an undirected graph.



Undirected Graph

$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

Adjacency Matrix

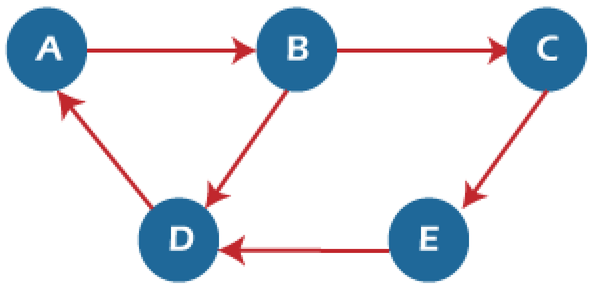
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

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

Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

Consider the below-directed graph and try to construct the adjacency matrix of it.



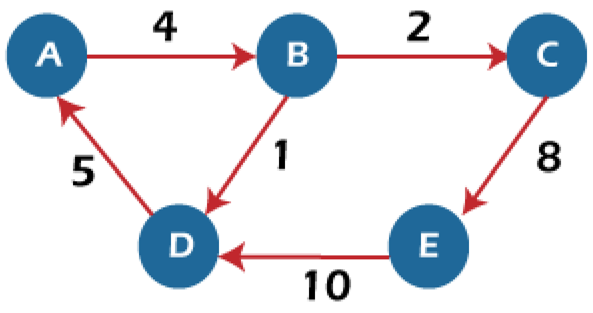
$$\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 0 & 0 & 0 \\
B & 0 & 0 & 1 & 1 & 0 \\
C & 0 & 0 & 0 & 0 & 1 \\
D & 1 & 0 & 0 & 0 & 0 \\
E & 0 & 0 & 0 & 1 & 0
\end{array}$$

Directed Graph **Adjacency Matrix**

In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

Adjacency matrix for a weighted directed graph

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.



$$\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 4 & 0 & 0 & 0 \\
B & 0 & 0 & 2 & 1 & 0 \\
C & 0 & 0 & 0 & 0 & 8 \\
D & 5 & 0 & 0 & 0 & 0 \\
E & 0 & 0 & 0 & 10 & 0
\end{array}$$

weighted Directed Graph **Adjacency Matrix**

n the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

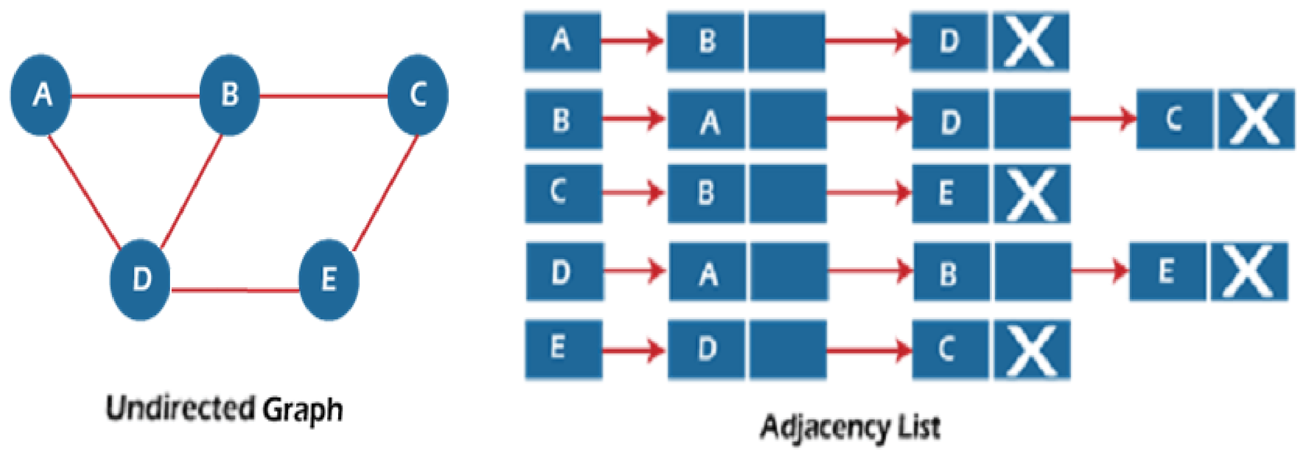
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

Linked list representation

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.

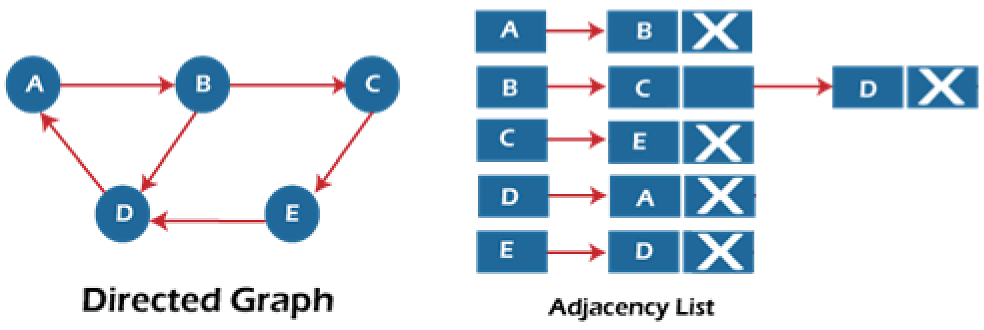


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

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 the last node of the list.

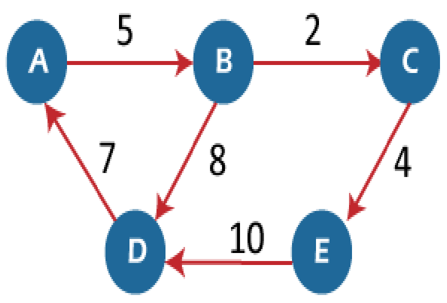
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph.

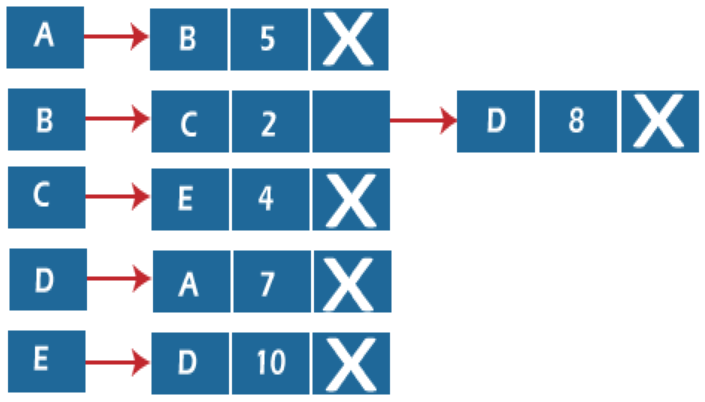


For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



Weighted Directed Graph



Adjacency List

In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

Implementation of adjacency matrix representation of Graph

Now, let's see the implementation of adjacency matrix representation of graph in C.

In this program, there is an adjacency matrix representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

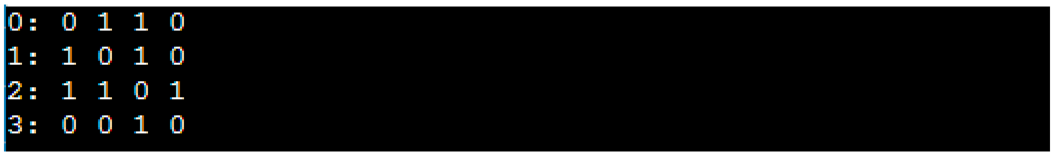
Here, there are four vertices and five edges in the graph that are non-directed.

```
/* Adjacency Matrix representation of an undirected graph in C */

#include <stdio.h>
#define V 4 /* number of vertices in the graph */

/* function to initialize the matrix to zero */
void init(int arr[][V]) {
  int i, j;
  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
      arr[i][j] = 0;
}

/* function to add edges to the graph */
void insertEdge(int arr[][V], int i, int j) {
  arr[i][j] = 1;
  arr[j][i] = 1;
```

```c
}

/* function to print the matrix elements */
void printAdjMatrix(int arr[][V]) {
  int i, j;
  for (i = 0; i < V; i++) {
    printf("%d: ", i);
    for (j = 0; j < V; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
}

int main() {
  int adjMatrix[V][V];

  init(adjMatrix);
  insertEdge(adjMatrix, 0, 1);
  insertEdge(adjMatrix, 0, 2);
  insertEdge(adjMatrix, 1, 2);
  insertEdge(adjMatrix, 2, 0);
  insertEdge(adjMatrix, 2, 3);

  printAdjMatrix(adjMatrix);

  return 0;
}
```

Output:

After the execution of the above code, the output will be -



```
0: 0 1 1 0
1: 1 0 1 0
2: 1 1 0 1
3: 0 0 1 0
```

Implementation of adjacency list representation of Graph

Now, let's see the implementation of adjacency list representation of graph in C.

In this program, there is an adjacency list representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

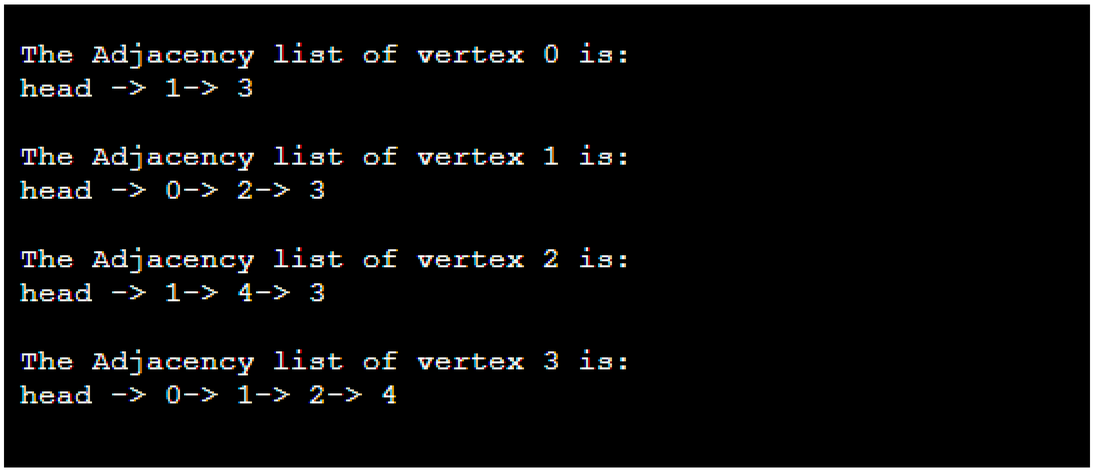
```c
/* Adjacency list representation of a graph in C */
#include <stdio.h>
#include <stdlib.h>

/* structure to represent a node of adjacency list */
struct AdjNode {
    int dest;
    struct AdjNode* next;
};

/* structure to represent an adjacency list */
struct AdjList {
    struct AdjNode* head;
};

/* structure to represent the graph */
struct Graph {
    int V; /*number of vertices in the graph*/
    struct AdjList* array;
};


struct AdjNode* newAdjNode(int dest)
{
    struct AdjNode* newNode = (struct AdjNode*)malloc(sizeof(struct AdjNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int V)
```

```
{
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));

    /* Initialize each adjacency list as empty by making head as NULL */
    int i;
    for (i = 0; i < V; ++i)
        graph->array[i].head = NULL;
    return graph;
}

/* function to add an edge to an undirected graph */
void addEdge(struct Graph* graph, int src, int dest)
{
    /* Add an edge from src to dest. The node is added at the beginning */
    struct AdjNode* check = NULL;
    struct AdjNode* newNode = newAdjNode(dest);

    if (graph->array[src].head == NULL) {
        newNode->next = graph->array[src].head;
        graph->array[src].head = newNode;
    }
    else {

        check = graph->array[src].head;
        while (check->next != NULL) {
            check = check->next;
        }
        // graph->array[src].head = newNode;
        check->next = newNode;
    }

    /* Since graph is undirected, add an edge from dest to src also */
    newNode = newAdjNode(src);
    if (graph->array[dest].head == NULL) {
        newNode->next = graph->array[dest].head;
        graph->array[dest].head = newNode;
    }
```

```c
    else {
        check = graph->array[dest].head;
        while (check->next != NULL) {
            check = check->next;
        }
        check->next = newNode;
    }
}
/* function to print the adjacency list representation of graph*/
void print(struct Graph* graph)
{
    int v;
    for (v = 0; v < graph->V; ++v) {
        struct AdjNode* pCrawl = graph->array[v].head;
        printf("\n The Adjacency list of vertex %d is: \n head ", v);
        while (pCrawl) {
            printf("-> %d", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}

int main()
{

    int V = 4;
    struct Graph* g = createGraph(V);
    addEdge(g, 0, 1);
    addEdge(g, 0, 3);
    addEdge(g, 1, 2);
    addEdge(g, 1, 3);
    addEdge(g, 2, 4);
    addEdge(g, 2, 3);
    addEdge(g, 3, 4);
    print(g);
    return 0;
}
```

Output:

In the output, we will see the adjacency list representation of all the vertices of the graph. After the execution of the above code, the output will be -



```
The Adjacency list of vertex 0 is:
head -> 1-> 3

The Adjacency list of vertex 1 is:
head -> 0-> 2-> 3

The Adjacency list of vertex 2 is:
head -> 1-> 4-> 3

The Adjacency list of vertex 3 is:
head -> 0-> 1-> 2-> 4
```

Graph Traversals:

Graph can be traversal by using two algorithms. They are:

1. BFS algorithm
2. DFS algorithm

BFS algorithm

In this article, we will discuss the BFS algorithm in the data structure. Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Applications of BFS algorithm

The applications of breadth-first-algorithm are given as follows -

- BFS can be used to find the neighboring locations from a given source location.
- In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.

- BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
- BFS is used to determine the shortest path and minimum spanning tree.
- BFS is also used in Cheney's technique to duplicate the garbage collection.
- It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Algorithm

The steps involved in the BFS algorithm to explore a graph are given as follows -

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

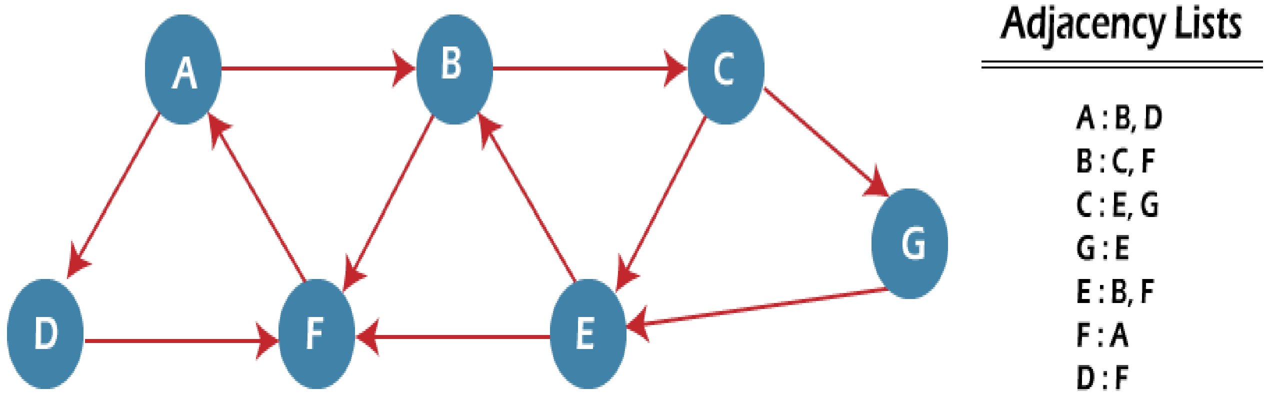
their STATUS = 2

(waiting state)

[END OF LOOP]

Example of BFS algorithm

Now, let's understand the working of BFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node 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.

Now, let's start examining the graph starting from Node A.

Step 1 - First, add A to queue1 and NULL to queue2.

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

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

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

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

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

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

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

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

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

Step 5 - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.

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

Step 6 - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

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

Complexity of BFS algorithm

Time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of BFS algorithm is **O(V+E)**, since in the worst case, BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).

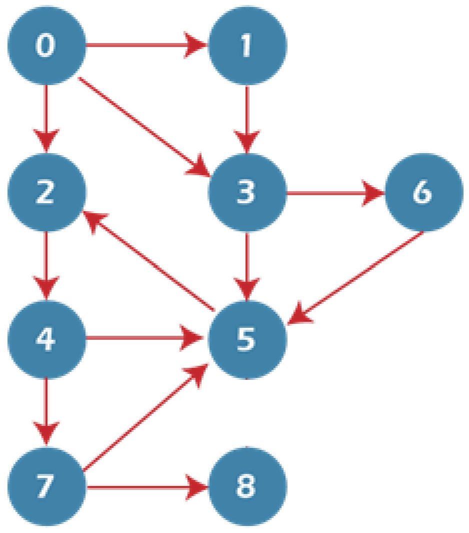
The space complexity of BFS can be expressed as **O(V)**, where V is the number of vertices.

Implementation of BFS algorithm

Now, let's see the implementation of BFS algorithm in java.

In this code, we are using the adjacency list to represent our graph. Implementing the Breadth-First Search algorithm in Java makes it much easier to deal with the adjacency list since we only have to travel through the list of nodes attached to each node once the node is dequeued from the head (or start) of the queue.

In this example, the graph that we are using to demonstrate the code is given as follows -

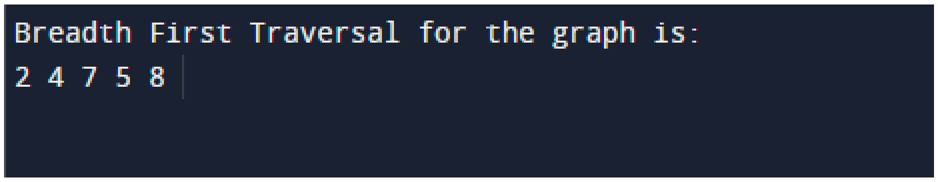


```
import java.io.*;
import java.util.*;
public class BFSTraversal
{
    private int vertex;       /* total number number of vertices in the graph */
    private LinkedList<Integer> adj[];      /* adjacency list */
    private Queue<Integer> que;           /* maintaining a queue */
    BFSTraversal(int v)
    {
        vertex = v;
        adj = new LinkedList[vertex];
        for (int i=0; i<v; i++)
        {
            adj[i] = new LinkedList<>();
        }
        que = new LinkedList<Integer>();
    }
    void insertEdge(int v,int w)
    {
        adj[v].add(w);      /* adding an edge to the adjacency list (edges are bidirectional in this example) */
```

```
}
void BFS(int n)
{
    boolean nodes[] = new boolean[vertex];      /* initialize boolean array for holding the data */
    int a = 0;
    nodes[n]=true;
    que.add(n);       /* root node is added to the top of the queue */
    while (que.size() != 0)
    {
        n = que.poll();        /* remove the top element of the queue */
        System.out.print(n+" ");    /* print the top element of the queue */
        for (int i = 0; i < adj[n].size(); i++)  /* iterate through the linked list and push all neighbors into queue */
        {
            a = adj[n].get(i);
            if (!nodes[a])      /* only insert nodes into queue if they have not been explored already */
            {
                nodes[a] = true;
                que.add(a);
            }
        }
    }
}
public static void main(String args[])
{
    BFSTraversal graph = new BFSTraversal(10);
    graph.insertEdge(0, 1);
    graph.insertEdge(0, 2);
    graph.insertEdge(0, 3);
    graph.insertEdge(1, 3);
    graph.insertEdge(2, 4);
    graph.insertEdge(3, 5);
    graph.insertEdge(3, 6);
    graph.insertEdge(4, 7);
    graph.insertEdge(4, 5);
    graph.insertEdge(5, 2);
    graph.insertEdge(6, 5);
    graph.insertEdge(7, 5);
    graph.insertEdge(7, 8);
    System.out.println("Breadth First Traversal for the graph is:");
```

```
        graph.BFS(2);
    }
}
```

Output



```
Breadth First Traversal for the graph is:
2 4 7 5 8
```

DFS (Depth First Search) algorithm

In this article, we will discuss the DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

The applications of using the DFS algorithm are given as follows -

- DFS algorithm can be used to implement the topological sorting.
- It can be used to find the paths between two vertices.
- It can also be used to detect cycles in the graph.
- DFS algorithm is also used for one solution puzzles.
- DFS is used to determine if a graph is bipartite or not.

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)

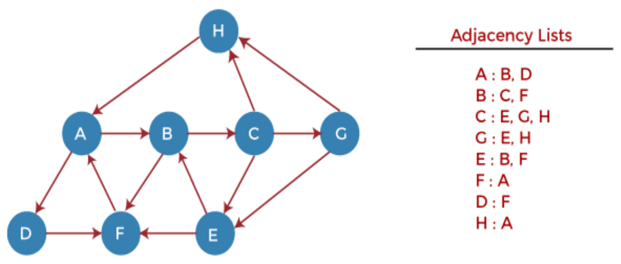
Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Example of DFS algorithm

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

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

Now, let's start examining the graph starting from Node H.

Step 1 - First, push H onto the stack.

1. STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H
2. STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A
2. STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1. Print: D
2. STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

1. Print: G
2. STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Complexity of Depth-first search algorithm

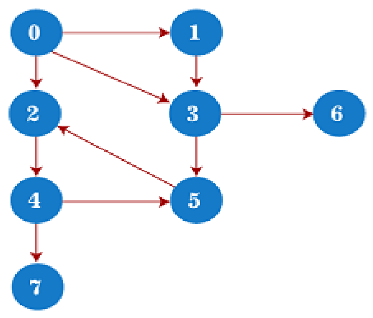
The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the DFS algorithm is O(V).

Implementation of DFS algorithm

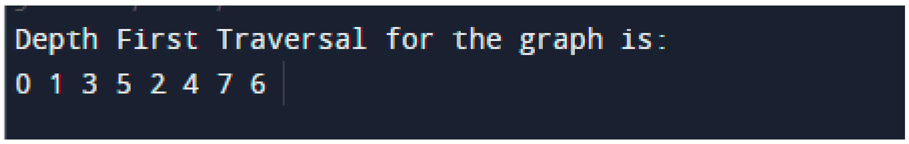
Now, let's see the implementation of DFS algorithm in Java.

In this example, the graph that we are using to demonstrate the code is given as follows -



```java
/*A sample java program to implement the DFS algorithm*/

import java.util.*;

class DFSTraversal {
  private LinkedList<Integer> adj[]; /*adjacency list representation*/
  private boolean visited[];

  /* Creation of the graph */
  DFSTraversal(int V) /*'V' is the number of vertices in the graph*/
  {
    adj = new LinkedList[V];
    visited = new boolean[V];

    for (int i = 0; i < V; i++)
      adj[i] = new LinkedList<Integer>();
  }

  /* Adding an edge to the graph */
  void insertEdge(int src, int dest) {
    adj[src].add(dest);
  }

  void DFS(int vertex) {
    visited[vertex] = true; /*Mark the current node as visited*/
    System.out.print(vertex + " ");

    Iterator<Integer> it = adj[vertex].listIterator();
    while (it.hasNext()) {
      int n = it.next();
      if (!visited[n])
        DFS(n);
    }
  }

  public static void main(String args[]) {
    DFSTraversal graph = new DFSTraversal(8);

        graph.insertEdge(0, 1);
        graph.insertEdge(0, 2);
        graph.insertEdge(0, 3);
        graph.insertEdge(1, 3);
        graph.insertEdge(2, 4);
        graph.insertEdge(3, 5);
        graph.insertEdge(3, 6);
        graph.insertEdge(4, 7);
        graph.insertEdge(4, 5);
        graph.insertEdge(5, 2);

        System.out.println("Depth First Traversal for the graph is:");
        graph.DFS(0);
  }
}
```

Output



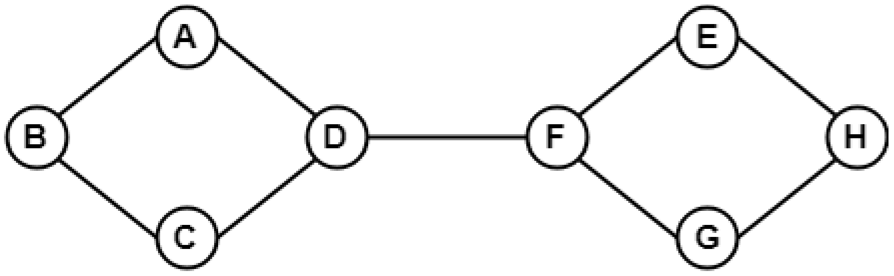
```
Depth First Traversal for the graph is:
0 1 3 5 2 4 7 6
```

Connected Components

- **Connectivity** is a basic concept of graph theory. It defines whether a graph is connected or disconnected. Without connectivity, it is not possible to traverse a graph from one vertex to another vertex.
- A graph is said to be **connected graph if there is a path between every pair of vertex**. From every vertex to any other vertex there must be some path to traverse. This is called the connectivity of a graph.
- A graph is said to be **disconnected, if there exists multiple disconnected vertices and edges**.

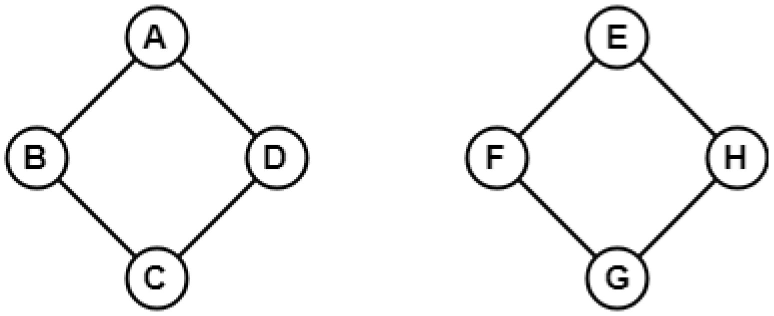
- Graph connectivity theories are essential in network applications, routing transportation networks, network tolerance etc.

Example



In the above example, it is possible to travel from one vertex to another vertex. Here, we can traverse from vertex B to H using the path B -> A -> D -> F -> E -> H. Hence it is a connected graph.

Example



In the above example, it is not possible to traverse from vertex B to H because there is no path between them directly or indirectly. Hence, it is a disconnected graph.

Let's see some basic concepts of Connectivity.

1. Cut Vertex

A single vertex whose removal disconnects a graph is called a cut-vertex.

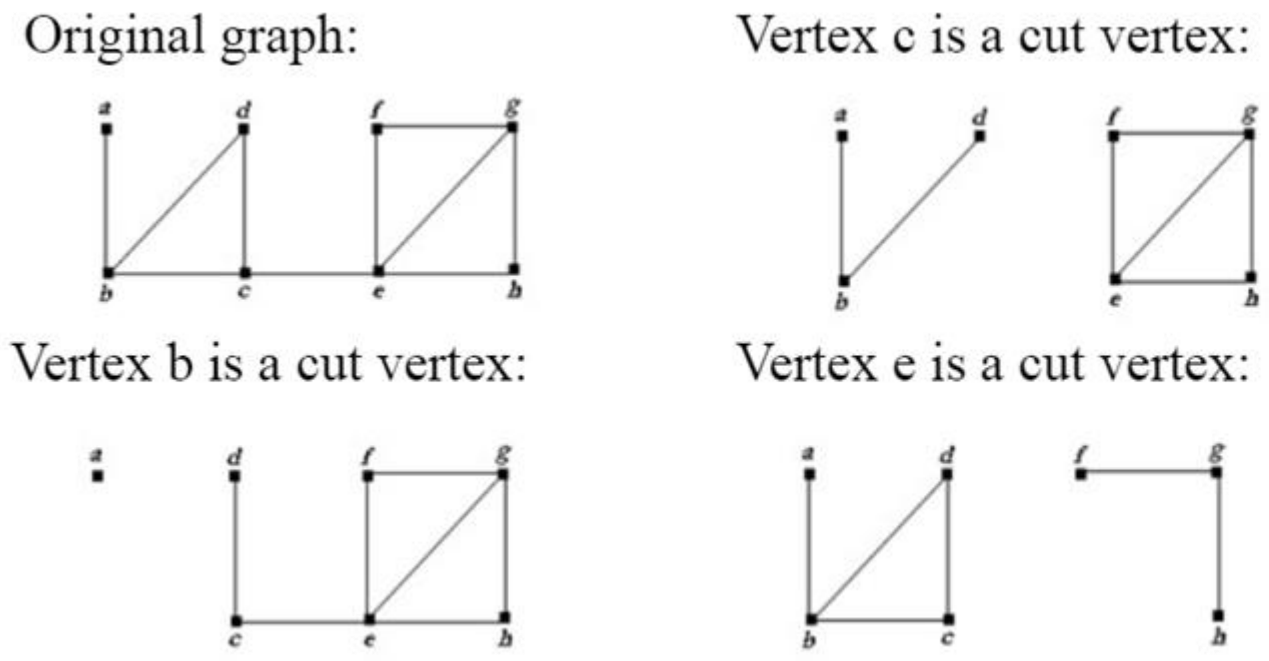
Let G be a connected graph. A vertex v of G is called a cut vertex of G, if G-v (Remove v from G) results a disconnected graph.

When we remove a vertex from a graph then graph will break into two or more graphs. This vertex is called a cut vertex.

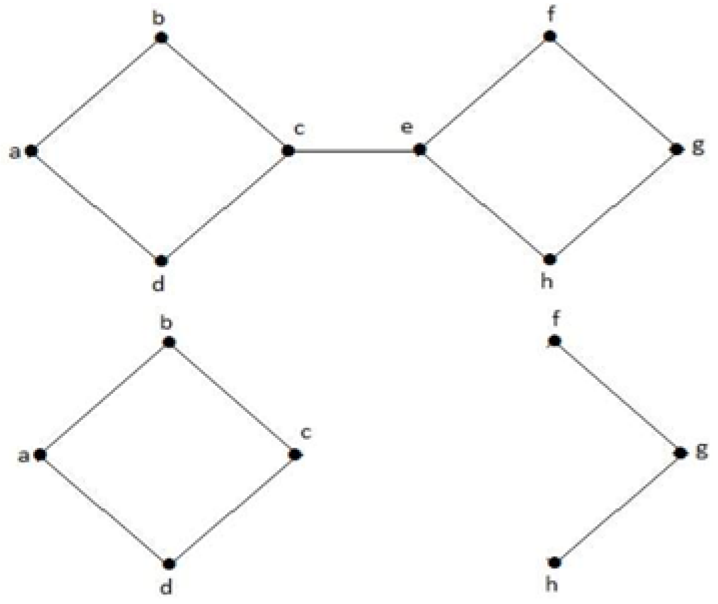
Note: Let G be a graph with n vertices:

- A connected graph G may have maximum (n-2) cut vertices.
- Removing a cut vertex may leave a graph disconnected.
- Removing a vertex may increase the number of components in a graph by at least one.
- Every non-pendant vertex of a tree is a cut vertex.

Example 1


Original graph:
Vertex c is a cut vertex:
Vertex b is a cut vertex:
Vertex e is a cut vertex:
a
d
f
g
b
c
e
h

Example 2



In the above graph, vertex 'e' is a cut-vertex. After removing vertex 'e' from the above graph the graph will become a disconnected graph.

2. Cut Edge (Bridge)

A **cut- Edge or bridge** is a single edge whose removal disconnects a graph.

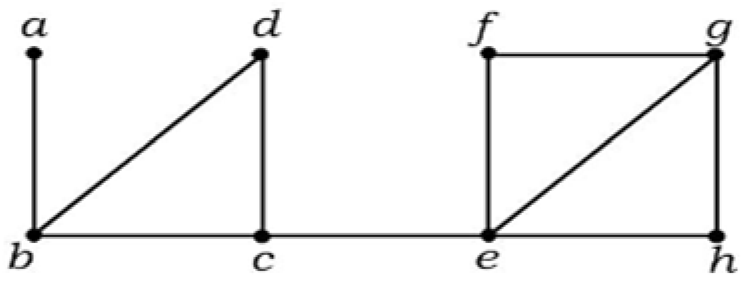
Let G be a connected graph. An edge e of G is called a cut edge of G, if G-e (Remove e from G) results a disconnected graph.

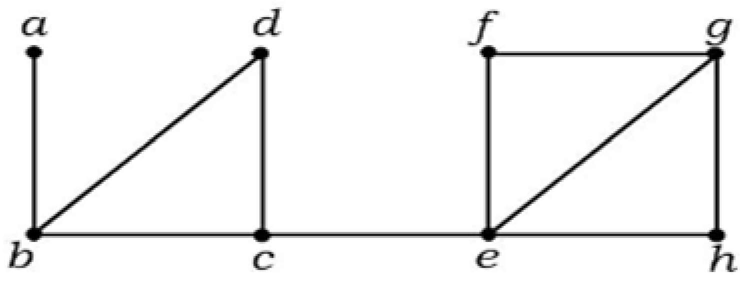
When we remove an edge from a graph then graph will break into two or more graphs. This removal edge is called a cut edge or bridge.

Note: Let G be a graph with n vertices:

- A connected graph G may have at most (n-1) cut edges.
- Removing a cut edge may leave a graph disconnected.
- Removal of an edge may increase the number of components in a graph by at most one.
- A cut edge 'e' must not be the part of any cycle in G.
- If a cut edge exists, then a cut vertex must also exist because at least one vertex of a cut edge is a cut vertex.
- If a cut vertex exists, then the existence of any cut edge is not necessary.

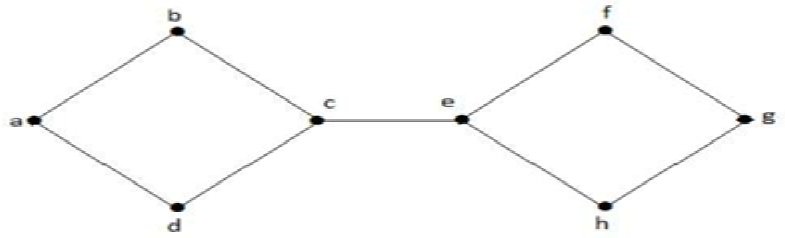
Example 1

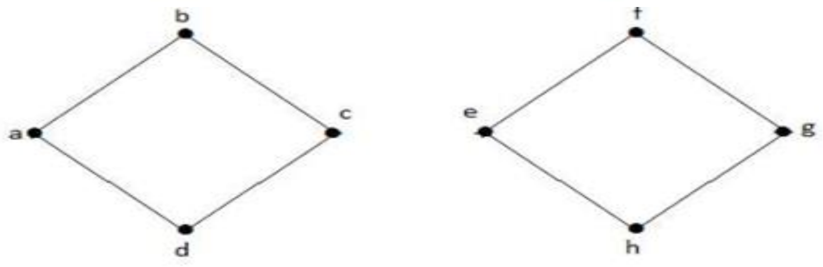




In the above graph, edge (c, e) is a cut-edge. After removing this edge from the above graph the graph will become a disconnected graph.

Example 2





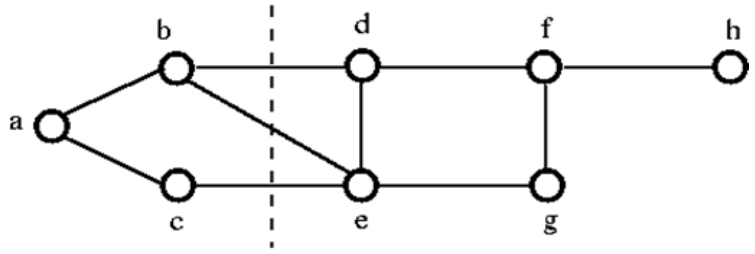
In the above graph, edge (c, e) is a cut-edge. After removing this edge from the above graph the graph will become a disconnected graph.

3. Cut Set

In a connected graph G, a cut set is a set S of edges with the following properties:

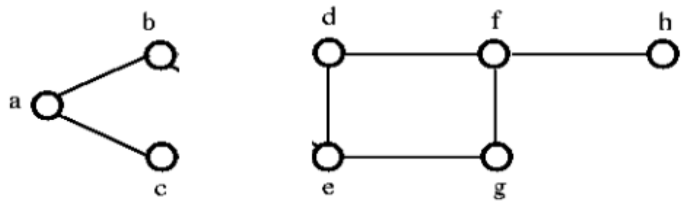
- The removal of all the edges in S disconnects G.
- The removal of some of edges (but not all) in S does not disconnect G.

Example 1



To disconnect the above graph G, we have to remove the three edges. i.e. bd, be and ce. We cannot disconnect it by removing just two of three edges. Hence, {bd, be, ce} is a cut set.

After removing the cut set from the above graph, it would look like as follows:



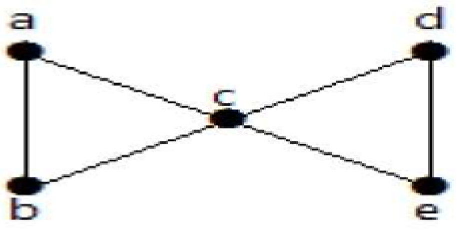
4. Edge Connectivity

The **edge connectivity** of a connected graph G is the minimum number of edges whose removal makes G disconnected. It is denoted by **λ(G)**.

When λ(G) ≥ k, then graph G is said to be **k-edge-connected**.

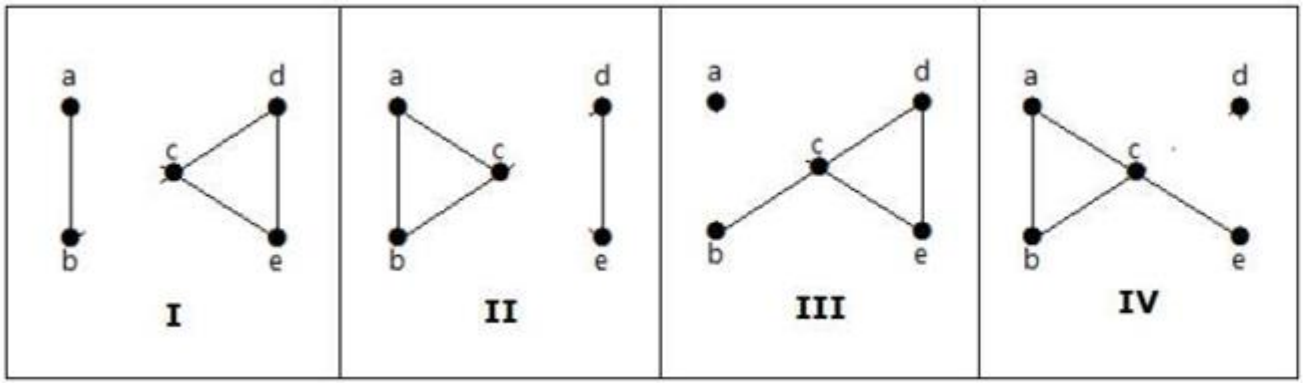
Example

Let's see an example,



From the above graph, by removing two minimum edges, the connected graph becomes disconnected graph. Hence, its edge connectivity is 2. Therefore the above graph is a **2-edge-connected graph**.

Here are the following four ways to disconnect the graph by removing two edges:



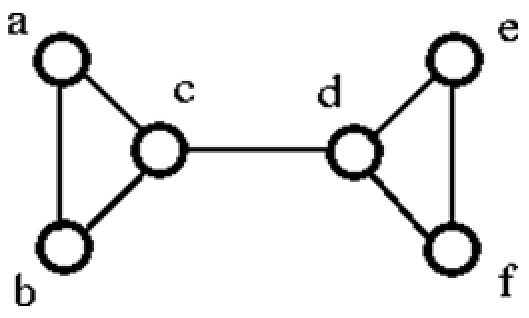
5. Vertex Connectivity

The connectivity (or vertex connectivity) of a connected graph G is the minimum number of vertices whose removal makes G disconnects or reduces to a trivial graph. It is denoted by K(G).

The graph is said to be k- connected or k-vertex connected when K(G) ≥ k. To remove a vertex we must also remove the edges incident to it.

Example

Let's see an example:



The above graph G can be disconnected by removal of the single vertex either 'c' or 'd'. Hence, its vertex connectivity is 1. Therefore, it is a 1-connected graph.

What is a spanning tree?

A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

A complete undirected graph can have $\mathbf{n^{n-2}}$ number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be $\mathbf{5^{5-2} = 125}$**.**

Applications of the spanning tree

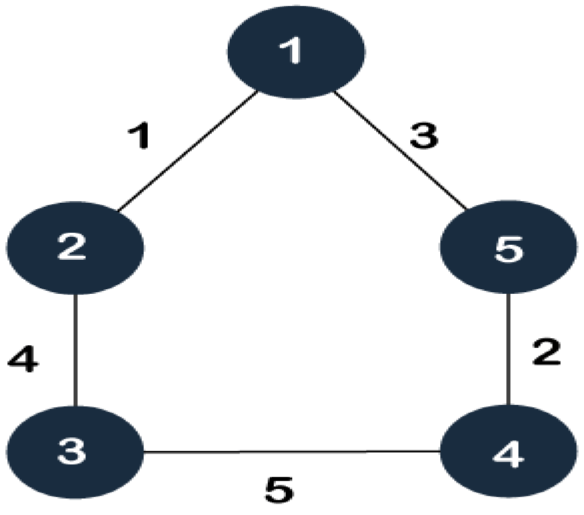
Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

- Cluster Analysis
- Civil network planning
- Computer network routing protocol

Now, let's understand the spanning tree with the help of an example.

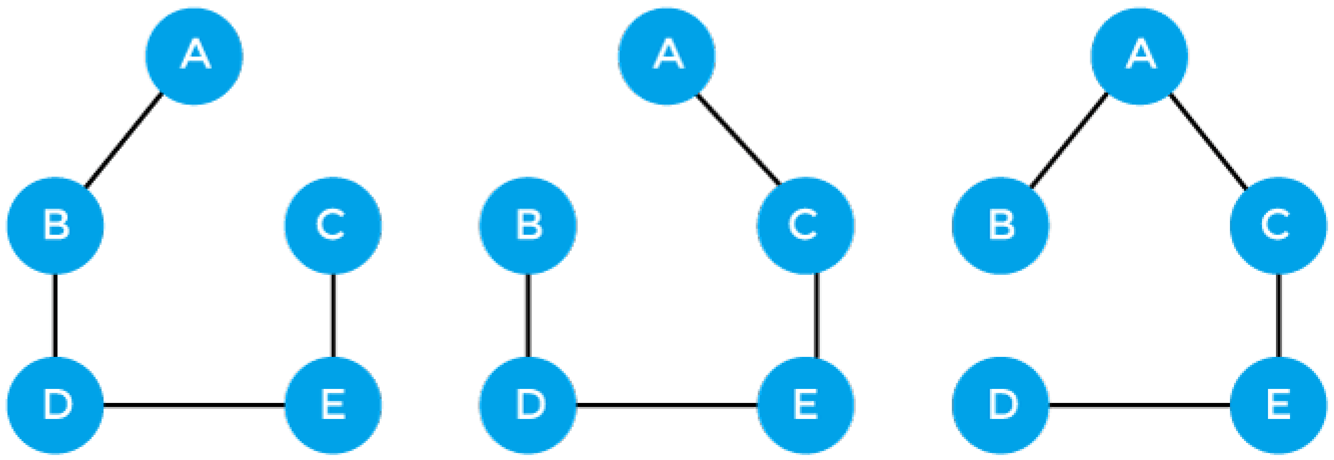
Example of Spanning tree

Suppose the graph be -



As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices. The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.

Some of the possible spanning trees that will be created from the above graph are given as follows -



Spanning tree 1 Spanning tree 2 Spanning tree 3

Properties of spanning-tree

Some of the properties of the spanning tree are given as follows -

- There can be more than one spanning tree of a connected graph G.
- A spanning tree does not have any cycles or loop.
- A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
- A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
- There can be a maximum $\mathbf{n^{n-2}}$ number of spanning trees that can be created from a complete graph.
- A spanning tree has **n-1** edges, where 'n' is the number of nodes.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

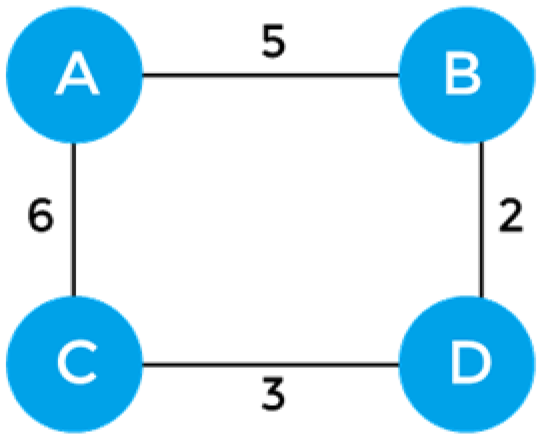
So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

Minimum Spanning tree

A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

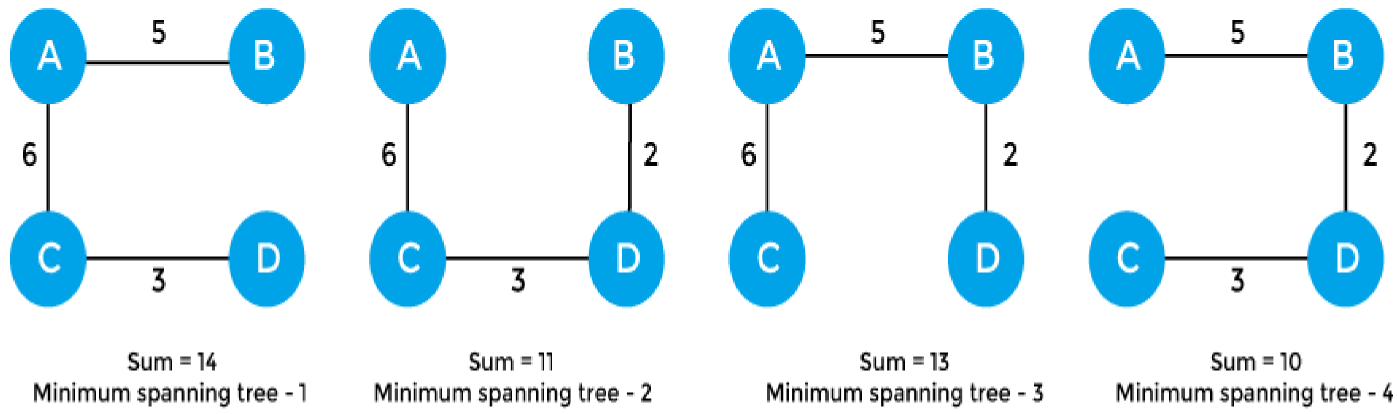
Example of minimum spanning tree

Let's understand the minimum spanning tree with the help of an example.

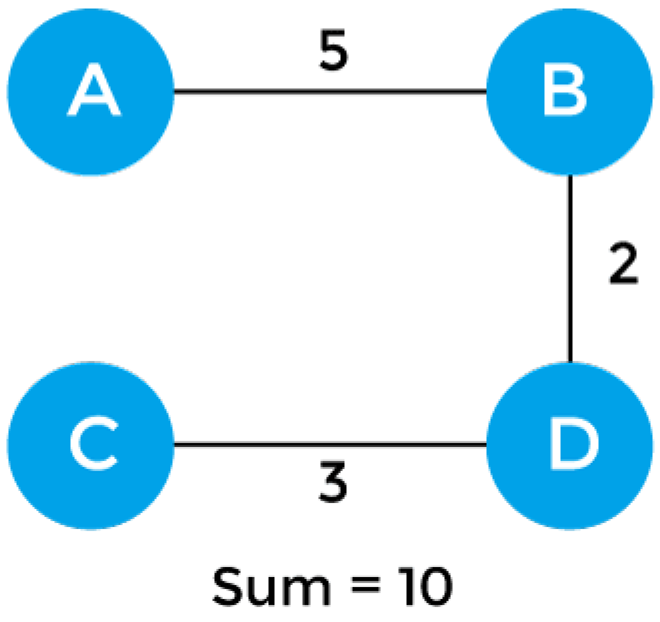


Weighted graph

The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -


A
B
C
D
5
6
3
Sum = 14
Minimum spanning tree - 1
6
2
3
Sum = 11
Minimum spanning tree - 2
5
6
2
Sum = 13
Minimum spanning tree - 3
5
2
3
Sum = 10
Minimum spanning tree - 4

So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



Applications of minimum spanning tree

The applications of the minimum spanning tree are given as follows -

- Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.
- It can be used to find paths in the map.

Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

- Prim's Algorithm
- Kruskal's Algorithm

Let's see a brief description of both of the algorithms listed above.

Prim's algorithm - It is a greedy algorithm that starts with an empty spanning tree. It is used to find the minimum spanning tree from the graph. This algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Kruskal's algorithm - This algorithm is also used to find the minimum spanning tree for a connected weighted graph. Kruskal's algorithm also follows greedy approach, which finds an optimum solution at every stage instead of focusing on a global optimum.

Linear Search Algorithm

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.

Two popular search methods are **Linear Search and Binary Search**. So, here we will discuss the popular searching technique, i.e., Linear Search Algorithm.

Linear search is also called as **sequential search algorithm.** It is the simplest searching algorithm. In Linear search, 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 is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is **O(n).**

The steps used in the implementation of Linear Search are listed as follows -

- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop,** compare the search element with the current array element, and -

- If the element matches, then return the index of the corresponding array element.
- If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return **-1.**

Now, let's see the algorithm of linear search.

Algorithm

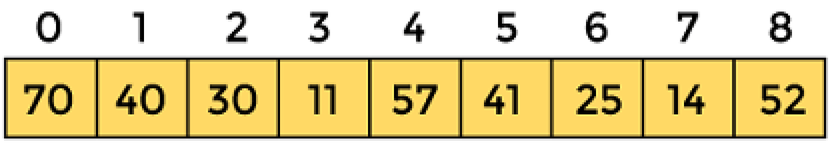
```
Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search
Step 1: set pos = -1
Step 2: set i = 1
Step 3: repeat step 4 while i <= n
Step 4: if a[i] == val
            Set pos = i
            print pos
            go to step 6
            [end of if]
        Set i = i + 1
    [end of loop]
Step 5: if pos = -1
        print "value is not present in the array "
        [end of if]
Step 6: exit
```

Working of Linear search:

Now, let's see the working of the linear search Algorithm.

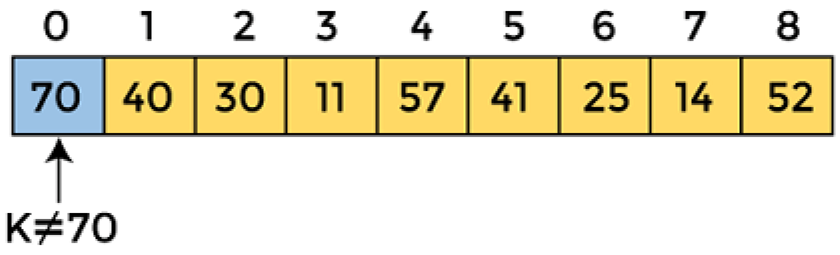
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

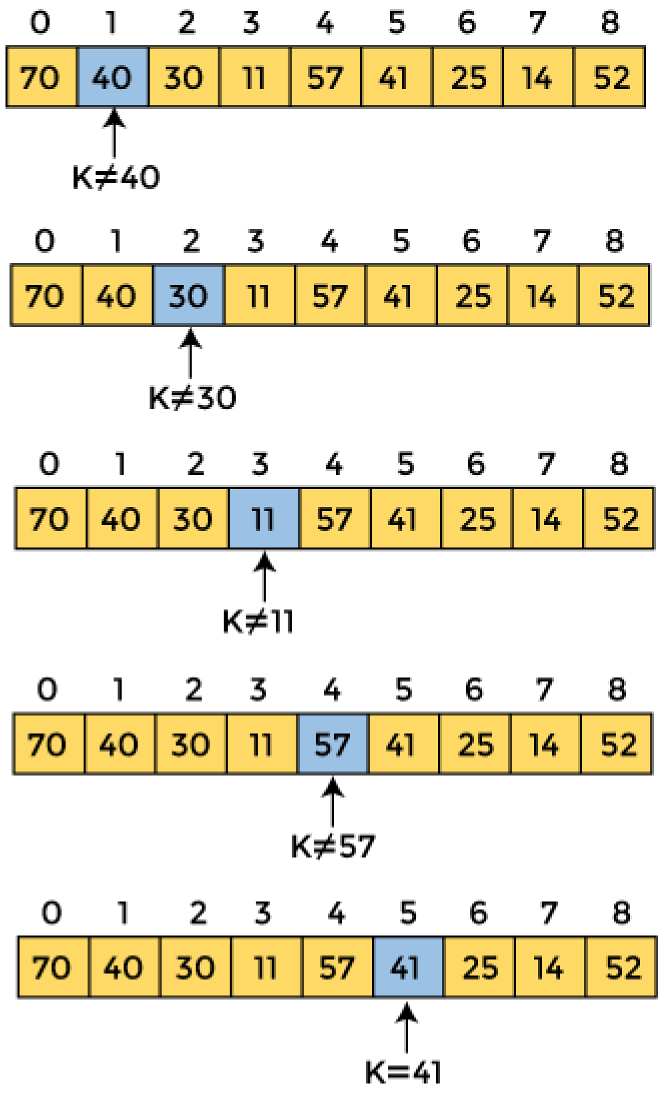


Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.



The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



Now, the element to be searched is found. So algorithm will return the index of the element matched.

Linear Search complexity

Now, let's see the time complexity of linear search in the best case, average case, and worst case. We will also see the space complexity of linear search.

1. Time Complexity

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

- **Best Case Complexity -** In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1).**
- **Average Case Complexity -** The average case time complexity of linear search is **O(n).**
- **Worst Case Complexity -** In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n).**

The time complexity of linear search is **O(n)** because every element in the array is compared only once.

2. Space Complexity

Space Complexity	O(1)

- The space complexity of linear search is O(1).

Implementation of Linear Search

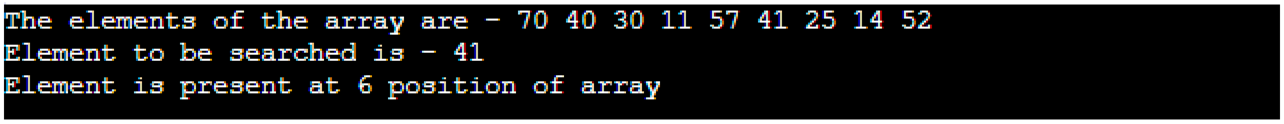
Now, let's see the programs of linear search in different programming languages.

Program: Write a program to implement linear search in C language.

```
#include <stdio.h>
int linearSearch(int a[], int n, int val) {
  // Going through array sequencially
  for (int i = 0; i < n; i++)
    {
        if (a[i] == val)
        return i+1;
    }
  return -1;
}
int main() {
```

```
int a[] = {70, 40, 30, 11, 57, 41, 25, 14, 52}; // given array
int val = 41; // value to be searched
int n = sizeof(a) / sizeof(a[0]); // size of array
int res = linearSearch(a, n, val); // Store result
printf("The elements of the array are - ");
for (int i = 0; i < n; i++)
printf("%d ", a[i]);
printf("\nElement to be searched is - %d", val);
if (res == -1)
printf("\nElement is not present in the array");
else
printf("\nElement is present at %d position of array", res);
return 0;
}
```

Output



```
The elements of the array are - 70 40 30 11 57 41 25 14 52
Element to be searched is - 41
Element is present at 6 position of array
```

Binary Search Algorithm:

Linear Search and Binary Search are the two popular searching techniques. Here we will discuss the Binary Search Algorithm.

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

Binary search follows **the 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 the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

NOTE: Binary search can be implemented on sorted array elements. If the list elements are not arranged in a sorted manner, we have first to sort them.

Now, let's see the algorithm of Binary Search.

Algorithm

```
Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search
Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <=end
```

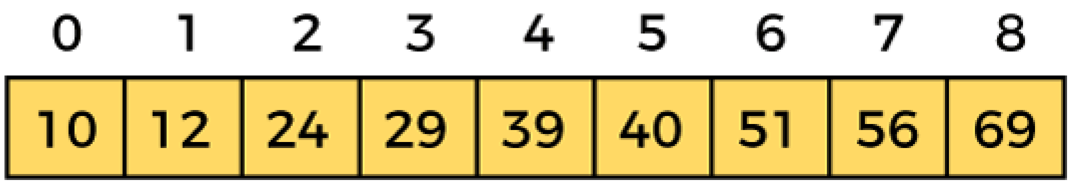
```
4.  Step 3: set mid = (beg + end)/2
5.  Step 4: if a[mid] = val
6.  set pos = mid
7.  print pos
8.  go to step 6
9.  else if a[mid] > val
10. set end = mid - 1
11. else
12. set beg = mid + 1
13. [end of if]
14. [end of loop]
15. Step 5: if pos = -1
16. print "value is not present in the array"
17. [end of if]
18. Step 6: exit
```

Working of Binary search

Now, let's see the working of the Binary Search Algorithm.

To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

Let the elements of array are -



Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

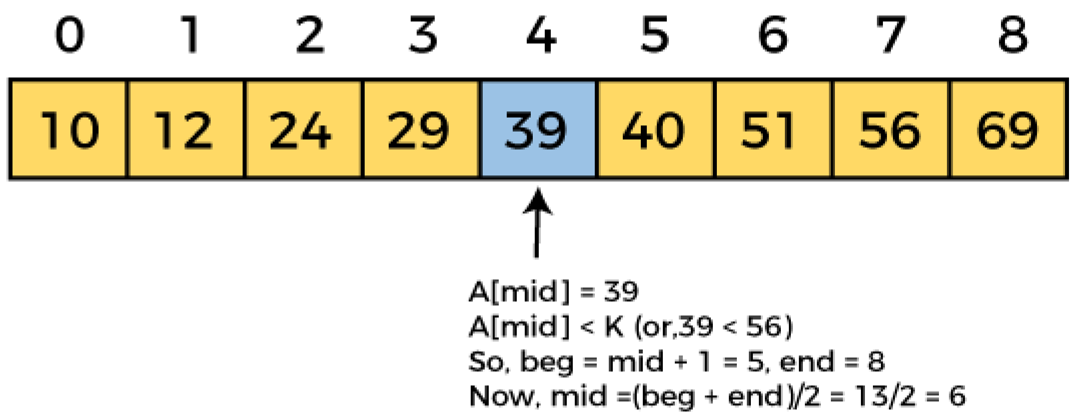
```
1.  mid = (beg + end)/2
```

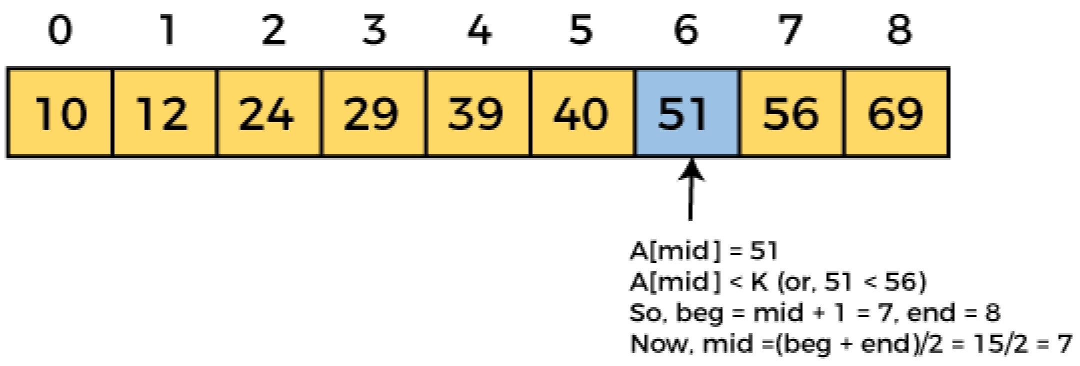
So, in the given array -

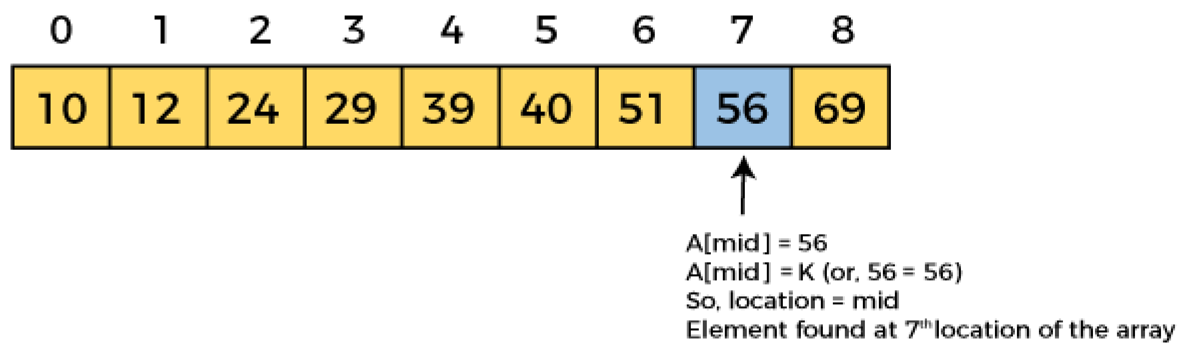
beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.







Now, the element to search is found. So algorithm will return the index of the element matched.

Binary Search complexity

Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

- **Best Case Complexity -** In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1).**
- **Average Case Complexity -** The average case time complexity of Binary search is **O(logn).**
- **Worst Case Complexity -** In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn).**

2. Space Complexity

Space Complexity	O(1)

- The space complexity of binary search is O(1).

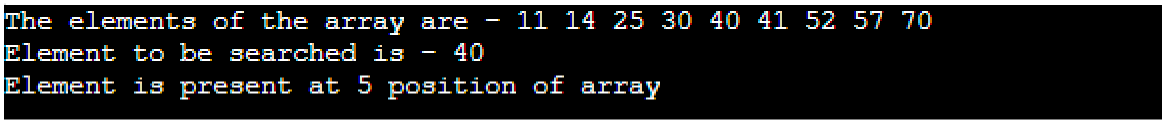
Implementation of Binary Search:

Now, let's see the programs of Binary search in different programming languages.

Program: Write a program to implement Binary search in C language.

```c
#include <stdio.h>
int binarySearch(int a[], int beg, int end, int val)
{
   int mid;
   if(end >= beg)
   {        mid = (beg + end)/2;
/* if the item to be searched is present at middle */
      if(a[mid] == val)
      {
         return mid+1;
      }
         /* if the item to be searched is smaller than middle, then it can only be in left subarray */
      else if(a[mid] < val)
      {
         return binarySearch(a, mid+1, end, val);
      }
         /* if the item to be searched is greater than middle, then it can only be in right subarray */
      else
      {
         return binarySearch(a, beg, mid-1, val);
      }
   }
   return -1;
}
int main() {
 int a[] = {11, 14, 25, 30, 40, 41, 52, 57, 70}; // given array
 int val = 40; // value to be searched
 int n = sizeof(a) / sizeof(a[0]); // size of array
 int res = binarySearch(a, 0, n-1, val); // Store result
 printf("The elements of the array are - ");
 for (int i = 0; i < n; i++)
 printf("%d ", a[i]);
 printf("\nElement to be searched is - %d", val);
 if (res == -1)
 printf("\nElement is not present in the array");
 else
 printf("\nElement is present at %d position of array", res);
 return 0;
}
```

Output



```
The elements of the array are - 11 14 25 30 40 41 52 57 70
Element to be searched is - 40
Element is present at 5 position of array
```

Bubble sort Algorithm

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is **$O(n^2)$,** where **n** is a number of items.

Bubble short is majorly used where -

- complexity does not matter
- simple and shortcode is preferred

Algorithm

In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

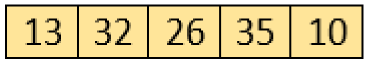
```
begin BubbleSort(arr)
  for all array elements
    if arr[i] > arr[i+1]
      swap(arr[i], arr[i+1])
    end if
  end for
  return arr
end BubbleSort
```

Working of Bubble sort Algorithm

Now, let's see the working of Bubble sort Algorithm.

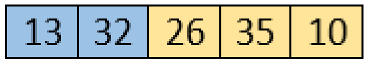
To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **O(n^2).**

Let the elements of array are -

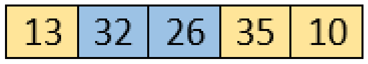


First Pass

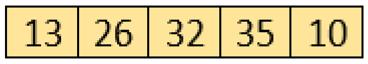
Sorting will start from the initial two elements. Let compare them to check which is greater.



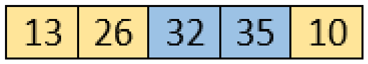
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

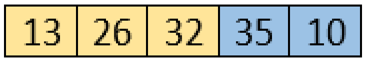


Now, compare 32 and 35.

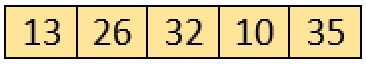


Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.



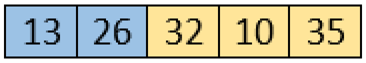
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

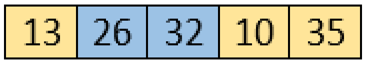


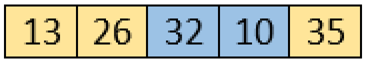
Now, move to the second iteration.

Second Pass

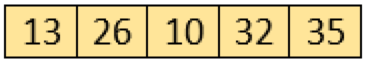
The same process will be followed for second iteration.

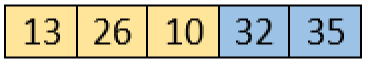






Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

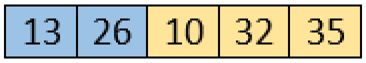


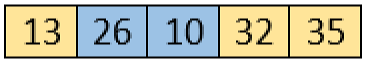


Now, move to the third iteration.

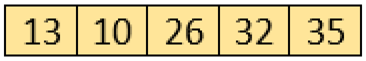
Third Pass

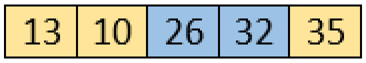
The same process will be followed for third iteration.

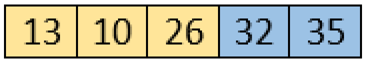




Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



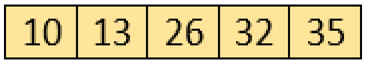




Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	O(n^2)
Worst Case	O(n^2)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **O(n^2).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **O(n^2).**

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.
- The space complexity of optimized bubble sort is O(2). It is because two extra variables are required in optimized bubble sort.

Now, let's discuss the optimized bubble sort algorithm.

Optimized Bubble sort Algorithm:

In the bubble sort algorithm, comparisons are made even when the array is already sorted. Because of that, the execution time increases.

To solve it, we can use an extra variable ***swapped.*** It is set to **true** if swapping requires; otherwise, it is set to **false.**

It will be helpful, as suppose after an iteration, if there is no swapping required, the value of variable **swapped** will be **false.** It means that the elements are already sorted, and no further iterations are required.

This method will reduce the execution time and also optimizes the bubble sort.

Algorithm for optimized bubble sort

The basic bubble sort algorithm can be explained as follows:

```
1. bubbleSort(array)
2.   for i <- 1 to indexOfLastUnsortedElement-1
3.     if leftElement > rightElement
4.       swap leftElement and rightElement
5. end bubbleSort
```

Implementation of Bubble sort

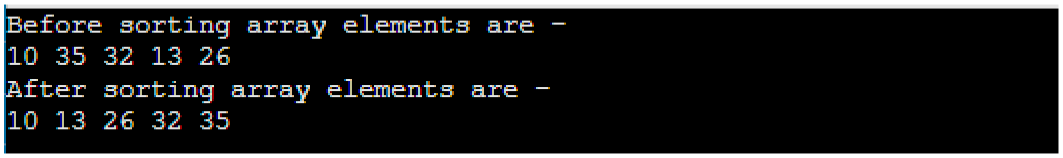
Now, let's see the programs of Bubble sort in different programming languages.

Program: Write a program to implement bubble sort in C language.

```
#include<stdio.h>
 void print(int a[], int n) //function to print array elements
   {
   int i;
   for(i = 0; i < n; i++)
   {
       printf("%d ",a[i]);
   }
   }
 void bubble(int a[], int n) // function to implement bubble sort
 {
   int i, j, temp;
   for(i = 0; i < n; i++)
   {
     for(j = i+1; j < n; j++)
       {
           if(a[j] < a[i])
           {
               temp = a[i];
               a[i] = a[j];
               a[j] = temp;
           }
       }
   }
 }
void main ()
{
   int i, j,temp;
   int a[5] = { 10, 35, 32, 13, 26};
   int n = sizeof(a)/sizeof(a[0]);
   printf("Before sorting array elements are - \n");
   print(a, n);
```

```
    bubble(a, n);
    printf("\nAfter sorting array elements are - \n");
    print(a, n);
}
```

Output



```
Before sorting array elements are -
10 35 32 13 26
After sorting array elements are -
10 13 26 32 35
```

Selection Sort Algorithm

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. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is **O(n^2)**, where **n** is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

Now, let's see the algorithm of selection sort.

Algorithm

```
SELECTION SORT(arr, n)

Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
Step 2: CALL SMALLEST(arr, i, n, pos)
Step 3: SWAP arr[i] with arr[pos]
[END OF LOOP]
Step 4: EXIT

```

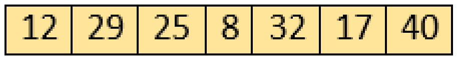
```
SMALLEST (arr, i, n, pos)
Step 1: [INITIALIZE] SET SMALL = arr[i]
Step 2: [INITIALIZE] SET pos = i
Step 3: Repeat for j = i+1 to n
if (SMALL > arr[j])
     SET SMALL = arr[j]
SET pos = j
[END OF if]
[END OF LOOP]
Step 4: RETURN pos
```

Working of Selection sort Algorithm

Now, let's see the working of the Selection sort Algorithm.

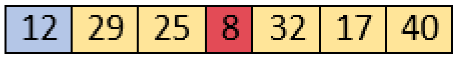
To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

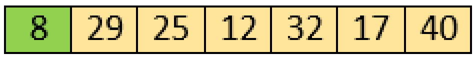


Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

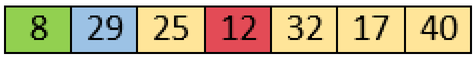
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.



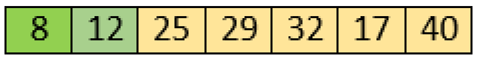
So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.



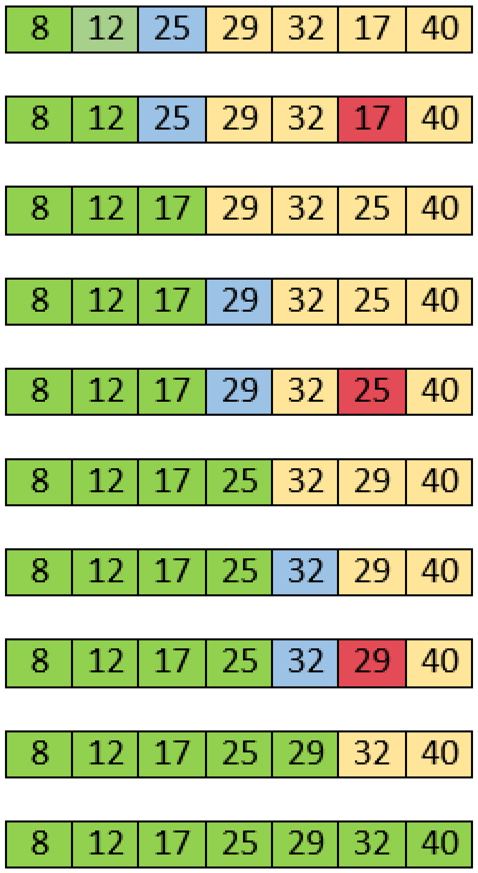
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.



Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.



The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

Selection sort complexity

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

1. Time Complexity

Case	**Time Complexity**
Best Case	O(n^2)
Average Case	O(n^2)
Worst Case	O(n^2)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **O(n^2)**.

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **O(n^2)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **O(n^2)**.

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

Implementation of selection sort

Now, let's see the programs of selection sort in different programming languages.

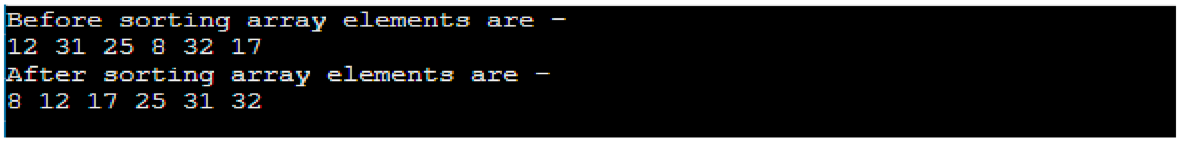
Program: Write a program to implement selection sort in C language.

```
#include <stdio.h>

void selection(int arr[], int n)
{
    int i, j, small;

    for (i = 0; i < n-1; i++)    // One by one move boundary of unsorted subarray
    {
        small = i; //minimum element in unsorted array

        for (j = i+1; j < n; j++)
        if (arr[j] < arr[small])
            small = j;
// Swap the minimum element with the first element
    int temp = arr[small];
    arr[small] = arr[i];
    arr[i] = temp;
    }
}

void printArr(int a[], int n) /* function to print the array */
```

```c
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}

int main()
{
    int a[] = { 12, 31, 25, 8, 32, 17 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    selection(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

Output:

After the execution of above code, the output will be -



```
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32
```

Insertion Sort Algorithm

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **$O(n^2)$**, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Now, let's see the algorithm of insertion sort.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

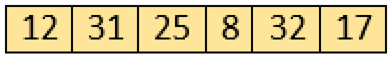
Step 6 - Repeat until the array is sorted.

Working of Insertion sort Algorithm

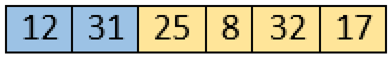
Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

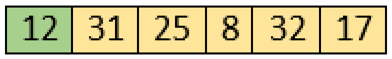
Let the elements of array are -



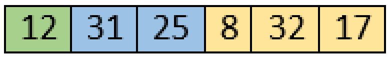
Initially, the first two elements are compared in insertion sort.

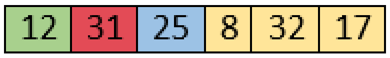


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



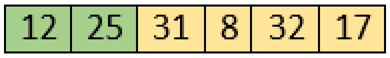
Now, move to the next two elements and compare them.



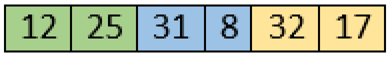


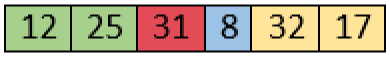
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

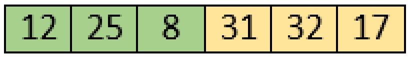


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

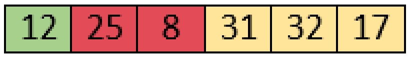




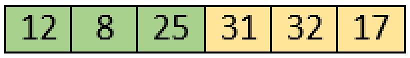
Both 31 and 8 are not sorted. So, swap them.



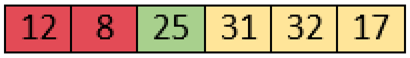
After swapping, elements 25 and 8 are unsorted.



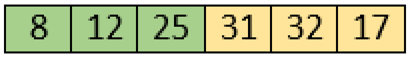
So, swap them.



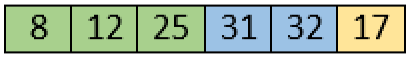
Now, elements 12 and 8 are unsorted.



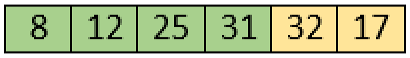
So, swap them too.



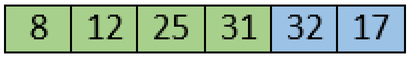
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



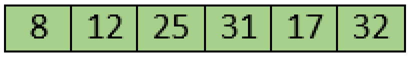
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

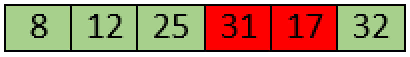


Move to the next elements that are 32 and 17.

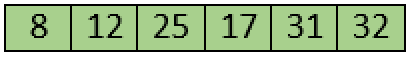


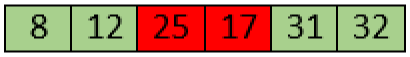
17 is smaller than 32. So, swap them.



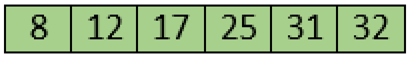


Swapping makes 31 and 17 unsorted. So, swap them too.





Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

Insertion sort complexity

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$**.

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

Implementation of insertion sort

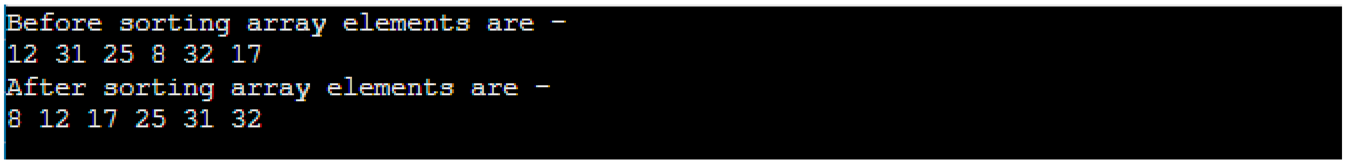
Now, let's see the programs of insertion sort in different programming languages.

Program: Write a program to implement insertion sort in C language.

```
#include <stdio.h>

void insert(int a[], int n) /* function to sort an aay with insertion sort */
{
    int i, j, temp;
    for (i = 1; i < n; i++) {
        temp = a[i];
        j = i - 1;

        while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to one position ahead from th
eir current position*/
```

```
        {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = temp;
    }
}

void printArr(int a[], int n) /* function to print the array */
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}

int main()
{
    int a[] = { 12, 31, 25, 8, 32, 17 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    insert(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);

    return 0;
}
```

Output:

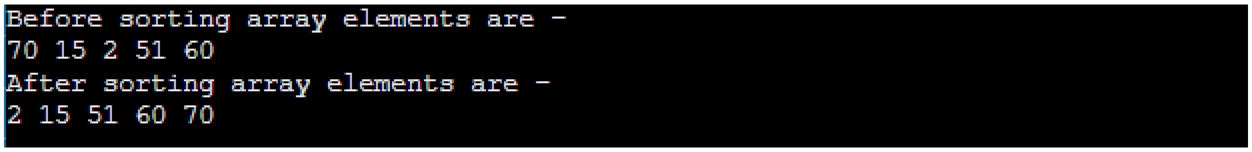


```
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32
```

Program: Write a program to implement insertion sort in python.

```
def insertionSort(a): # Function to implement insertion sort
    for i in range(1, len(a)):
        temp = a[i]
        # Move the elements greater than temp to one position
        #ahead from their current position
        j = i-1
        while j >= 0 and temp < a[j] :
                a[j + 1] = a[j]
                j = j-1
        a[j + 1] = temp

def printArr(a): # function to print the array

    for i in range(len(a)):
        print (a[i], end = " ")

a = [70, 15, 2, 51, 60]
print("Before sorting array elements are - ")
printArr(a)
insertionSort(a)
print("\nAfter sorting array elements are - ")
printArr(a)
```

Output:



```
Before sorting array elements are -
70 15 2 51 60
After sorting array elements are -
2 15 51 60 70
```