KVSB - Dr.K.V.SUBBA REDDY

SCHOOL OF BUSINESS MANAGEMENT

MCA & MBA COLLEGES

Institute Name:	Dr. K. V. Subba Reddy School of Business Management
College Code:	JJ
ICET Code:	KVSB
Name of Programmes:	MBA & MCA

SUBJECT: DATA STRUCTURES

Data Structures

(Course code: 21F00104, for MCA - Regulations: R21)

Course Code	DATA STRUCTURES	L	T	P	C
21F00104		4	0	0	4
	Semester	I			

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 the different types of searching and sorting techniques.

Course Outcomes (CO): Student will be able to

- Use C basic concepts to write simple C programs
- Explain the different notations of arithmetic express
- Analyze various operations on linked list
- Develop the representation of Tress
- Design the different sorting technique

UNIT – I	Lecture Hrs:

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.
Introduction to Functions, Storage classes, Arrays, Structures, Unions, Pointers, Strings and Command line arguments.

UNIT – II	Lecture Hrs:

Data Structures, Stacks and Queues- Overview of Data Structure, 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 - III	Lecture Hrs:

Linked Lists–Pointers, 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 – IV	Lecture Hrs:

Trees- Tree terminology, representation, Binary tress, 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.

UNIT – V	

Searching and Sorting–Sequential, Binary, Exchange (Bubble) Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort. Searching- Linear and Binary Search Methods.

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.

Introduction to C Language:

C programming is a general-purpose, procedural programming language developed in **1972** by **Dennis M.Ritchie** at the Bell Telephone Laboratories (AT&T Bell Laboratories) to develop the UNIX operating system. C is the most widely used computer language. It has the popularity along with Java programming language, which is also equally popular and most widely used among modern software programmers.

Why to learn C Programming:

C programming language is a must for students and working professionals to become a great Software Engineer especially when they are working in Software Development Domain. C has some advantages to learn:

- Easy to learn
- Structured language
- It produces efficient programs
- It can handle low-level activities
- It can be compiled on a variety of computer platforms

- C was invented to write an operating system called UNIX.
- C is a successor of B language which was introduced around the early 1970s.
- The language was formalized in 1988 by the American National Standard Institute (ANSI).
- The UNIX OS was totally written in C.
- Today C is the most widely used and popular System Programming Language.

C Language Elements:

The basic C Language elements are used to create a C program. These elements are - the valid

- **character set**,
- **identifiers**,
- **keywords**,
- basic **data types** and their representation,
- **Constants** and
- **Variables**.

The C Character Set

C uses the uppercase English alphabets **A to Z**, the lowercase letters **a to z**, the digits **0 to 9**, and certain special characters as building blocks to form basic program elements viz. constants, variables, operators, expressions and statements.

The special characters are listed below:

!	*	+	\	"	<
#	(	=	|	{	>
%	)	~	;	}	/
^	-	[	:	,	?
&	-	]	'	.	(blank)

In addition, certain combinations of these characters, such as **'\b'**, **'\n'** and **'\t'**, are used to represent special condition such as **backspace**, **newline** and **horizontal tab**, respectively. These character combinations are known as <u>escape sequences</u>.

Identifiers

Identifiers are names given to various items in the program, such as variables, functions and arrays. An identifier consists of letters and digits, in any order, except that the first character must be a letter. Both upper and lowercase letters are permitted. Upper and lowercase letters are not interchangeable (i.e., an uppercase letter is not equivalent to the corresponding lowercase letter). The underscore character (_) can also be included, and it is treated as a letter.

Keywords like **if**, **else**, **int**, **float**, etc., have special meaning and they cannot be used as identifier names.

The following are examples of **valid identifier names**:

A, ab123, velocity, stud_name,
circumference, Average, TOTAL

The following names are **not valid identifiers**:

1st	- the first character must be a letter
"Jamshedpur"	- illegal characters (")
stud-name	- illegal character (-)
stud name	- illegal character (blank space)

Although an identifier can be long, most implementations recognize typically 31 characters.

Keywords (Reserved Words)

Keywords/ Reserved words given for specific purpose; we can't use them for other purpose. The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

Data Types: means type / classification of the data which we are using in the program.

There are only few basic data types in C. These are listed in the table below:

Data type	Description	Size	Range
char	single character	1 byte	0 - 255
int	integer number	4 bytes	-2147483648 to 2147483647
float	single precision floating point number (number containing fraction & or an exponent)	4 bytes	3.4E-38 to 3.4E+38
double	double precision floating point number	8 bytes	1.7E-308 to 1.7E+308

The list of data types can be increased by using the data type qualifiers such as - ***short, long****, and* ***unsigned****.* For example, an integer quantity can be defined as long, short, signed or unsigned integer.

The memory requirement of an integer data varies depending on the compilers used. The qualified basic data types and their sizes are shown in table below.

Data type in C	Size	Range
short int	2 bytes	-32768 to 32767
long int	4 bytes	-2147483648 to 2147483647
unsigned short int	2 bytes	0 to 65535
unsigned int	4 bytes	0 to 4294967295
unsigned long int	4 bytes	0 to 4294967295
long double (extended precision)	8 bytes	1.7E-308 to1.7E+308

Constants:

Constants refer to **fixed values** that the program may not alter / modified during its execution. These fixed values are also called literals.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Examples:

Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point Constant	10.3, 20.2, 450.6 etc.
Octal Constant	021, 033, 046 etc.
Hexadecimal Constant	0x2a, 0x7b, 0xaa etc.
Character Constant	'a', 'b', 'x' etc.
String Constant	"c", "c program", "c in javatpoint" etc.

There are two ways to define constant in C programming.

1. **const** keyword
2. **#define** preprocessor

1) The **const** keyword is used to define constant in C programming.

```
const  float  PI=3.14;
```

If we declare like this, then value of the PI variable can't be changed thought the program execution.

```
#include<stdio.h>
int main( )
{  const float PI=3.14;
    printf("The value of PI is: %f",PI);
    return 0;
}
```

Output:

```
The value of PIis: 3.140000
```

If you try to change the value of the PI in this program, it will give you compile time error.
PI=4.5;   if you add like this, program will give compile time error.

2) **#define** preprocessor

The #define preprocessor is also used to define constant. We will learn about #define preprocessor directive later.

See here for: #define preprocessor directive.

```
#include <stdio.h>
#define PI 3.14
main( ) {
   printf("%f",PI);
}
```

```
Output: 3.14
```

Variables in C:
A variable is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.
It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the **syntax** to declare a variable: datatype variable; or
datatype variable_list;

Example:

```
int a;
float b;
char  c;
```

Here, a, b, c are variables. The int, float, char are the data types.

Rules for defining variables

- A variable can have **alphabets**, **digits**, and **underscore**.
- A variable name **can start with** the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, floats, etc.

Valid variable names:

```
int a;
int _ab;
int a30;
```

Invalid variable names:

```
int 2;
int a b;
int long;
```

```
// C program to add two numbers
#include<stdio.h>
 int main( )
{
   int A, B, sum = 0;
   printf("Enter value for A: \n");
   scanf("%d", &A);
   printf("Enter value for B: \n");
   scanf("%d", &B);
   sum = A + B;
   // Print the sum
   printf("Sum of A and B is: %d", sum);
   return 0;
}
```

Output:

```
Enter value for A: 5
Enter value for B: 3
Sum of A and B is: 8
```

Variable Declarations and Data Types

- Variables should be declared in the C program before to use.
- Memory space is not allocated for a variable while declaration. It happens only on variable definition.
- Variable initialization means assigning a value to the variable.

Syntax:

Variable declaration `data_type variable_name;`

Example: int x, y, z;
char gender;

Variable initialization `data_type variable_name = value;`

Example: int x = 52, y = 30;
char gender = ‘F’;

Based on the declaration place in the program we have the following types of variables in C.

Types of Variables in C

There are many types of variables in c:

1. local variable
2. global variable
3. static variable
4. automatic variable
5. external variable

Local Variable

A variable that is declared inside the function or block is called a local variable.It must be declared at the start of the block.

Example:
```
void function1( )
{
int x=10;  //local variable
}
```

You must have to initialize the local variable before it is used.

Global Variable

A variable that is declared outside the function or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions.
It must be declared at the start of the block.

Example:
```
int value=20;//global variable
void function1( )
{
int x=10;//local variable
}
```

Static Variable

A variable that is declared with the **static** keyword is called static variable. It retains its value between multiple function calls.

Example:
```
void function1( )
{
int x=10;//local variable
static int y=10;//static variable
x=x+1;
y=y+1;
printf("%d,%d",x,y);
}
```

If you call this function many times, the **local variable will print the same value** for each function call, e.g, 11,11,11 and so on. But the **static variable will print the incremented value** in each function call, e.g. 11, 12, 13 and so on.

Automatic Variable

All variables in C that are declared inside the block, are automatic variables by default. We can explicitly declare an automatic variable using **auto keyword**.

```
void  main( ) {
int x=10;
//local variable (also automatic)
auto int y=20;  //automatic variable
}
```

External Variable

We can share a variable in multiple C source files by using an external variable. To declare an external variable, you need to use **extern keyword**.

myfile.h

Syntax: **extern int** x=10;   //external variable (also global)

```
include  "myfile.h"
#include <stdio.h>
void printValue()
{
   printf("Global variable: %d", global_variable);
}
```

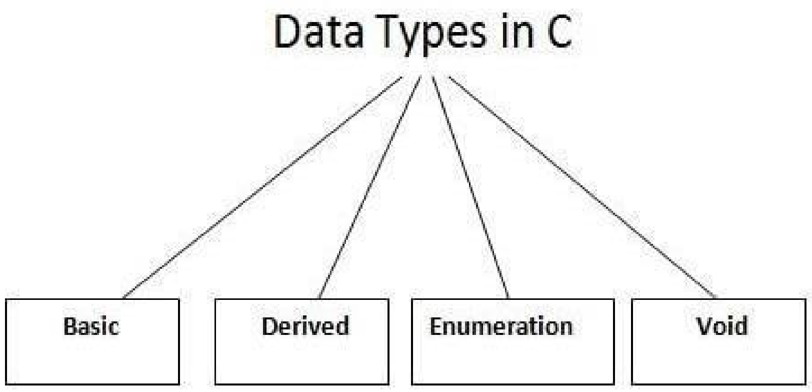
```
#include <stdio.h>
extern int x = 32;
int b = 8;
int main() {
  auto int a = 28;
  extern int b;
  printf("The value of auto variable : %d", a);
  printf("The value of extern variables x and b : %d,%d",x,b);
  x = 15;
  printf("The value of modified extern variable x : %d",x);
  return 0;
}
```

Output:

```
The value of auto variable : 28
The value of extern variables x and b : 32,8
The value of modified extern variable x : 15
```

Data Types in C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.



There are the following data types in C language.

Types	Data Types
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void

Types & Description
Basic Types They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.
Enumerated types They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
The type void The type specifier *void* indicates that no value is available.
Derived types They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals. The memory size of the basic data types may change according to 32 or 64-bit operating system.

Data Types	**Memory Size**	**Range**
char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255
short	2 byte	−32,768 to 32,767
signed short	2 byte	−32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	2 byte	−32,768 to 32,767
signed int	2 byte	−32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	−32,768 to 32,767
signed short int	2 byte	−32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
float	4 byte	
double	8 byte	
long double	10byte	

Operators and Expressions:

- The **symbols** which are used to perform logical and mathematical operations in a C program are called C operators.
- These C operators join individual constants and variables to form expressions.
- Operators, functions, constants and variables are combined together to form expressions.
- Consider the expression A + B * 5. Where, +, * are operators, A, B are variables, 5 is constant and A + B * 5 is an expression.

Types of C Operators:

C language offers many types of operators. They are,

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators (ternary operators)
7. Increment/decrement operators
8. Special operators

1) Arithmetic operators:

These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus.

Arithmetic Operators/Operation	Example
+ (Addition)	A+B
– (Subtraction)	A-B
* (multiplication)	A*B
/ (Division)	A/B
% (Modulus)	A%B

2) Assignment operators:

These are used to assign the values for the variables in C programs.

- For example, if the value "10" is to be assigned for the variable "sum", it can be assigned as "sum = 10;"
- There are 2 categories of assignment operators in C language. They are,
  1. Simple assignment operator (Example: =)
  2. Compound assignment operators(Example: +=, -=, *=, /=, %=, &=, ^=)

Operators	Example / Description
=	sum = 10; 10 is assigned to variable sum
+=	sum += 10; This is same as sum = sum + 10
-=	sum -= 10; This is same as sum = sum – 10
*=	sum *= 10; This is same as sum = sum * 10
/=	sum /= 10; This is same as sum = sum / 10
%=	sum %= 10; This is same as sum = sum % 10
&=	sum&=10; This is same as sum = sum & 10
^=	sum ^= 10; This is same as sum = sum ^ 10

3) Relational operators:

These operators are used to compare the value of two variables.

- Relational operators are used to find the relation between two variables. i.e. to compare the values of two variables in a C program.

Operators	Example/Description
>	x > y (x is greater than y)
<	x < y (x is less than y)
>=	x >= y (x is greater than or equal to y)
<=	x <= y (x is less than or equal to y)
==	x == y (x is equal to y)
!=	x != y (x is not equal to y)

4) Logical operators:

These operators are used to perform logical operations on the given expressions.

- There are 3 logical operators in C language. They are, logical AND (&&), logical OR (\|\|) and logical NOT (!).

Operators	Example/Description
&& (logical AND)	(x>5)&&(y<5) It returns true when both conditions are true
|| (logical OR)	(x>=10)||(y>=10) It returns true when at-least one of the condition is true
! (logical NOT)	!((x>5)&&(y<5)) It reverses the state of the operand "((x>5) && (y<5))" If "((x>5) && (y<5))" is true, logical NOT operator makes it false

5) Bit wise operators:

These operators are used to perform bit operations on given two variables.

- These operators are used to perform bit operations. Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits.
- Bit wise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise NOT), ^ (XOR), << (left shift) and >> (right shift).

Truth table for bit wise operation & bit wise operators:

x	y	x|y	x&y	x^y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Below are the bit-wise operators and their name in c language.

1. & – Bitwise AND
2. | – Bitwise OR
3. ~ – Bitwise NOT
4. ^ – XOR
5. << – Left Shift
   >> – Right Shift

6) Conditional operators (ternary operators):

- Conditional operators return one value if condition is true and returns another value is condition is false.
- This operator is also called as ternary operator.

  Syntax : (Condition? true_value: false_value);
  Example : (A > 100 ? 0 : 1);
- In above example, if A is greater than 100, 0 is returned else 1 is returned. This is equal to if else conditional statements.

7) Increment/decrement operators:

- Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.
- Syntax:
  Increment operator: ++var_name; (or) var_name++;
  Decrement operator: – -var_name; (or) var_name – -;
- Example:
  Increment operator : ++ i ; i ++ ;
  Decrement operator : – – i ; i – – ;
  Example program for increment operators

```
#include <stdio.h>
int main( )
{
int i=1;
while(i<10)
{
        printf("%d ",i);
        i++;
      }
}
```

OUTPUT:

```
1 2 3 4 5 6 7 8 9
```

8) Special operators:

Below are some of the special operators that the C programming language offers.

Operators	Description
&	This is used to get the address of the variable. Example :&a will give address of a.
*****	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
sizeof ()	This gives the size of the variable. Example : size of (char) will give us 1.

Expressions:

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.

Example:

```
a+b
c
s-1/7*f
.
.
etc
```

Types of Expressions:

Expressions may be of the following types:

- **Constant expressions**: Constant Expressions consists of only constant values. A constant value is one that doesn't change.
  Examples:
  5, 10 + 5 / 6.0, 'x'
- **Integral expressions**: Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions.

  Examples:

x, x * y, x + int(5.0)
where x and y are integer variables.

- **Floating expressions**: Float Expressions are which produce floating point results after implementing all the automatic and explicit type conversions.
  Examples:
  x + y, 10.75
  where x and y are floating point variables.
- **Relational expressions**: Relational Expressions yield results of type **bool** which takes a value **true** or **false**. When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as Boolean expressions.
  Examples:
  x <= y, x + y > 2
- **Logical expressions**: Logical Expressions combine two or more relational expressions and produces bool type results.
  Examples:
  x > y && x == 10, x == 10 || y == 5
- **Pointer expressions**: Pointer Expressions produce address values.
  Examples:
  &x, ptr, ptr++
  where x is a variable and ptr is a pointer.
- **Bitwise expressions**: Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits.
  Examples:
  x << 3
  shifts three bit position to left
  y >> 1
  shifts one bit position to right.
  Shift operators are often used for multiplication and division by powers of two.

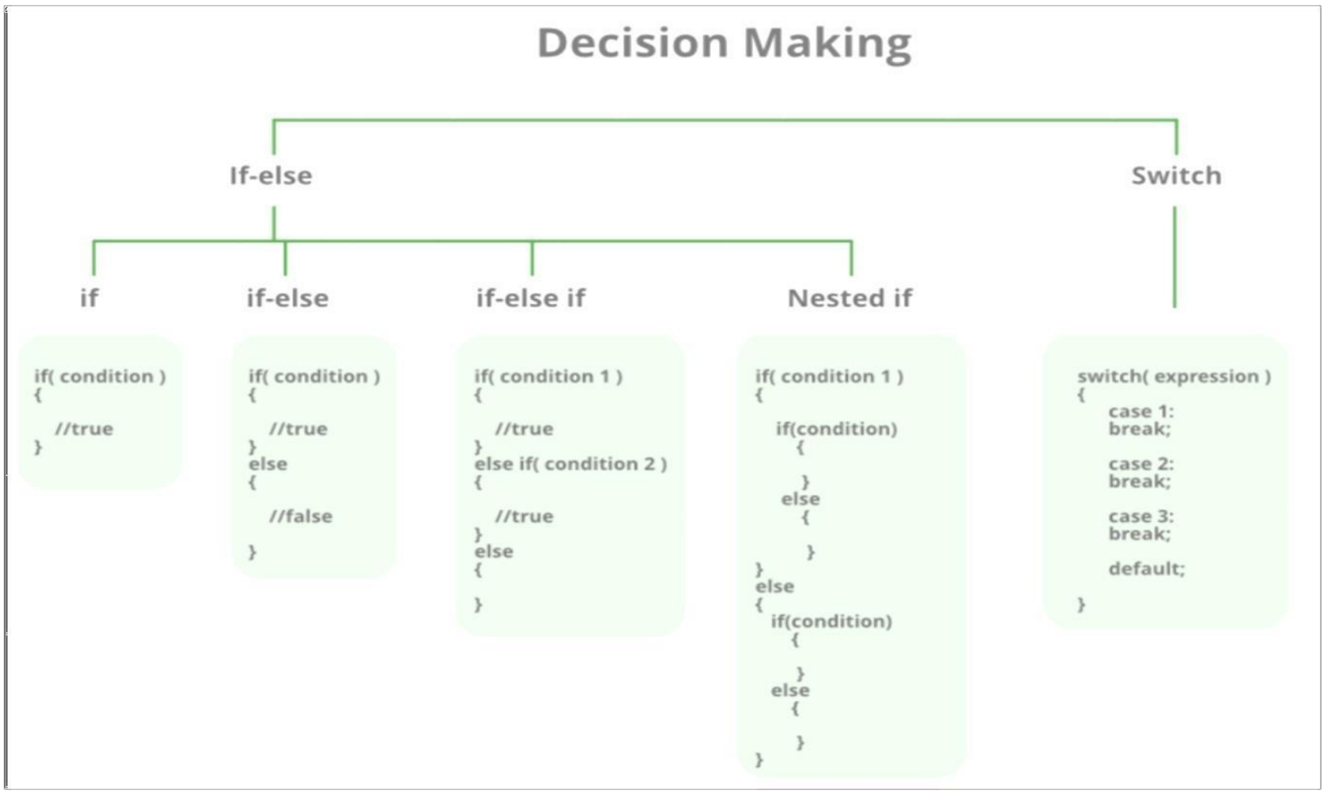
Note: An expression may also use combinations of the above expressions. Such expressions are known as **compound expressions**.

Decision Statements - If and Switch Statements:

There come situations in real life when we need to make some decisions and based on these decisions, we decide what we should do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.

For **example**, in C if x occurs then execute y else execute z. There can also be multiple conditions like in C if x occurs then execute p, else if condition y occurs execute q, else execute r.

This condition of C else-if is one of the many ways of importing multiple conditions.



```
if( condition )
{
   //true
}
```

```
if( condition )
{
   //true
}
else
{
   //false
}
```

```
if( condition 1 )
{
   //true
}
else if( condition 2 )
{
   //true
}
else
{
}
```

```
if( condition 1 )
{
  if(condition)
    {
    }
  else
    {
    }
}
else
{
  if(condition)
    {
    }
  else
    {
    }
}
```

```
switch( expression )
{
    case 1:
    break;

    case 2:
    break;

    case 3:
    break;

    default;
}
```

Decision-making statements in programming languages decide the direction of the flow of program execution. Decision-making statements available in C or C++ are:

1. **if statement**
2. **if..else statements**
3. **nested if statements**
4. **if-else-if ladder**
5. **switch statements**
6. **Jump Statements:**
   a. **break**
   b. **continue**
   c. **goto**
   d. **return**

if statement in C:

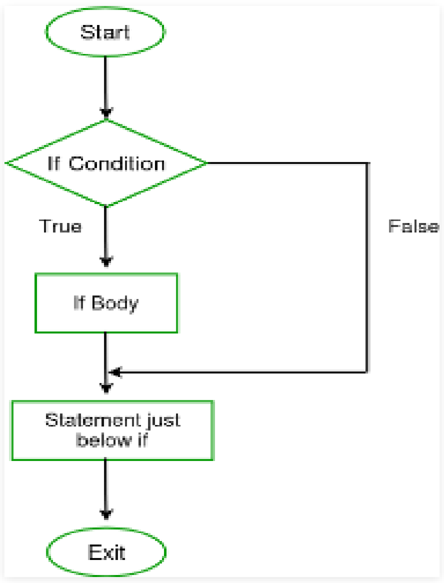
if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
  // Statements to execute if
  // condition is true
}
```

Here, the **condition** after evaluation will be either true or false. C if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not.

If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.



Example:

```
if(condition)
   statement1;
   statement2;

// Here if the condition is true, if block
// will consider only statement1 to be inside
// its block.
```

Flowchart

```
// C program to illustrate If statement
#include <stdio.h>
 int main( )
{
   int i = 10;
 if(i> 15)
    {
      printf("10 is less than 15");
    }
       printf("I am Not in if");
}
```

Output:

```
I am not in if
```

The condition present in the if statement: is false. So, the block below the if statement is not executed.

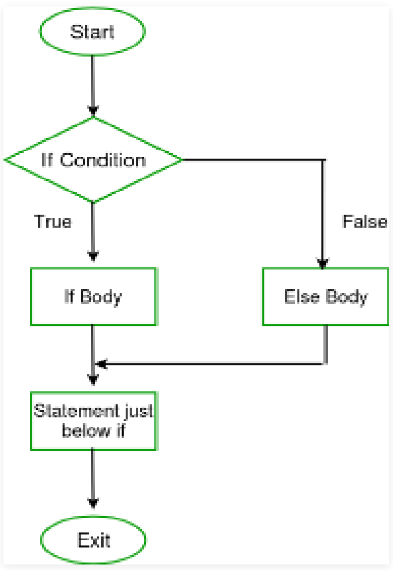
if-else:

The *if* statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the C *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
    {
    // Executes this block if
    // condition is true
    }
    else
    {
    // Executes this block if
    // condition is false
    }
```

Flowchart:



Example:

```
// C program to illustrate If statement
#include <stdio.h>
 int main( ) {
   int  i = 20;
   if(i< 15)
{
            printf("i is smaller than 15");
   }
   else
{
            printf("i is greater than 15");
   }
   return 0;
}
```

Output:

```
i is greater than 15
```

The block of code following the *else* statement is executed as the condition present in the *if* statement is false.

Nested-if in C:

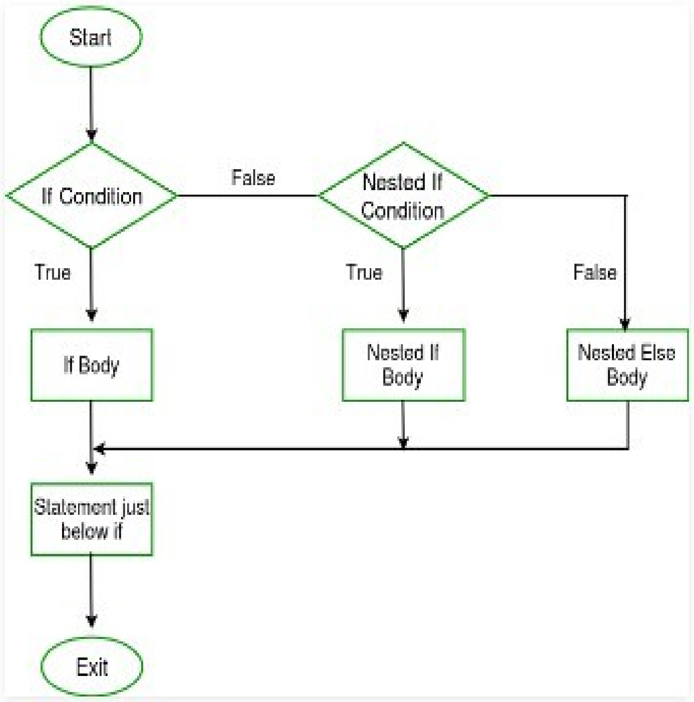
A nested if in C is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, both C and C++ allow us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
    {
     // Executes when condition1 is true
     if (condition2)
     {
       // Executes when condition2 is true
```

```
            }
        }
```

Flowchart



Example:

```
 // C program to illustrate nested-if statement
#include <stdio.h>
 int main( ) {
    int  i  =  10;
    if(i = =10)
    {
       // First if statement
       if(i< 15)
          printf("i is smaller than 15\n");
        // Nested - if statement
       // Will only be executed if statement above
       // is true
       if(i< 12)
           printf("i is smaller than 12 too\n");
       else
           printf("i is greater than 15");
    }
     return0;
}
```

Output:

```
i is smaller than 15
i is smaller than 12 too
```

if-else-if ladder in C:

Here, a user can decide among multiple options. The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none of the conditions are true, then the final else statement will be executed.

Syntax:

```
if (condition)
     statement;
  else if (condition)
     statement;
  .
  .
  else
     statement;
```

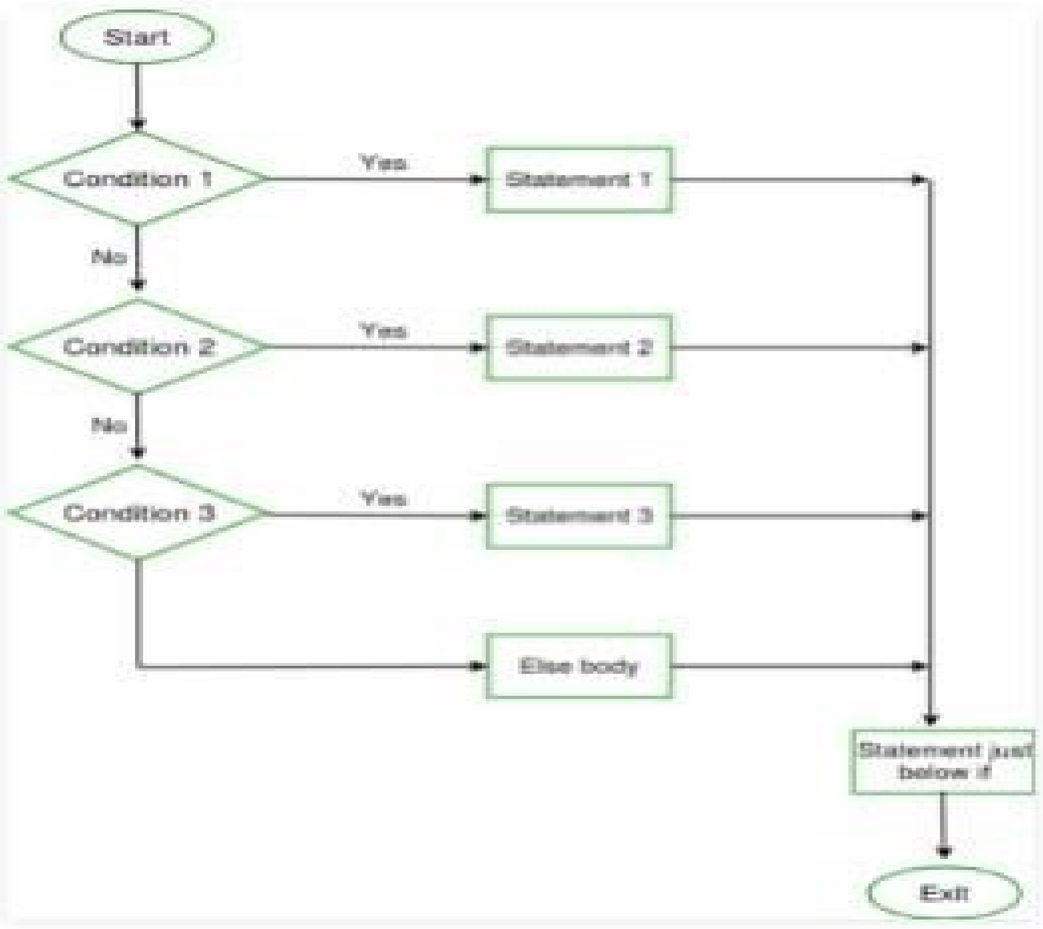
Example:

```
// C program to illustrate nested-if statement
#include <stdio.h>
int main( ) {
   int i = 20;

   if(i == 10)
      printf("i is 10");
   elseif(i == 15)
      printf("i is 15");
   elseif(i == 20)
      printf("i is 20");
   else
      printf("i is not present");
}
```

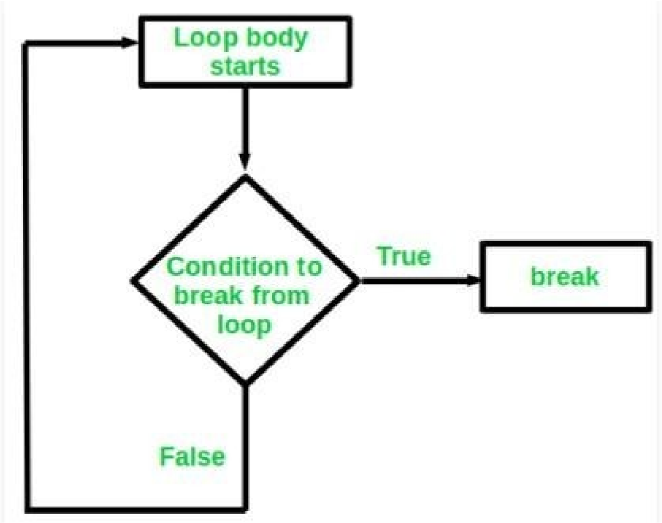
Output:

```
i is 20
```



Jump Statements in C
These statements are used in C orC++ for the unconditional flow of control throughout the functions in a program. They support four types of jump statements:

1. **C break:** This loop control statement is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.
   Syntax:
   break;
   1. Basically, break statements are used in situations when we are not sure about the actual number of iterations for the loop or we want to terminate the loop based on some condition.

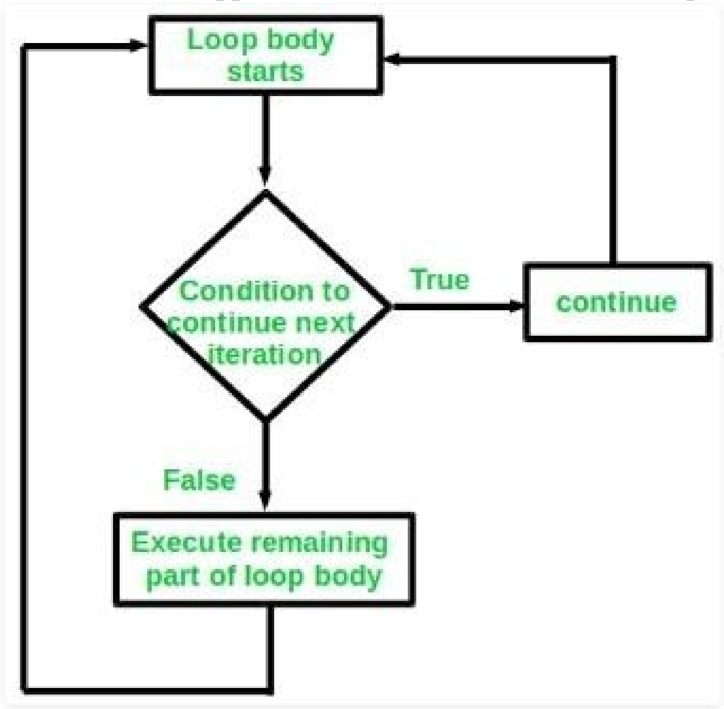


Example:
```
 // C program to illustrate
// Linear Search
 #include <stdio.h>
 voidfindElement(int arr[ ], int size, int key)
{
   // loop to traverse array and search for key
   for(int i = 0; i< size; i++) {
      if(arr[i] == key) {
         printf("Element found at position: %d", (i + 1));
         break;
      }
   }
}

int main( ) {
   int arr[ ] = { 1, 2, 3, 4, 5, 6 };
    // no of elements
   int n = 6;
    // key to be searched
   int key = 3;
   // Calling function to find the key
   findElement(arr, n, key);
    return 0;
}
```

Output:
```
Element found at position: 3
```

continue: This loop control statement is just like the break statement. The *continue* statement is opposite to that of the break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop.
As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.


Loop body starts
Condition to continue next iteration
True
continue
False
Execute remaining part of loop body

Example:

```
// C program to explain the use
// of continue statement
#include <stdio.h>
 int main( ) {
   // loop from 1 to 10
   for(int i = 1; i<= 10; i++) {
       // If i is equals to 6,
      // continue to next iteration
      // without printing
      if(i = = 6)
          continue;
       else
          // otherwise print the value of i
          printf("%d ", i);
   }
    return 0;
}
```

Output:

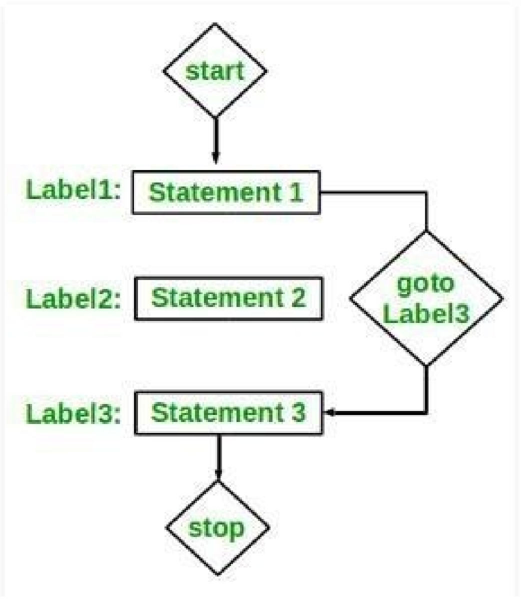
```
1 2 3 4 5 7 8 9 10
```

goto: The goto statement in C/C++ also referred to as unconditional jump statement can be used to jump from one point to another within a function.

Syntax:

```
Syntax1       |   Syntax2
----------------------------
goto label;   |    label:
.             |    .
.             |    .
.             |    .
label:        |    goto label;
```

1. In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here label is a user-defined identifier that indicates the target statement. The statement immediately followed after 'label:' is the destination statement. The 'label:' can also appear before the 'goto label;' statement in the above syntax.



examples to use goto statement:

Examples:

```
 // C program to print numbers
// from 1 to 10 using goto statement
#include <stdio.h>
 // function to print numbers from 1 to 10
voidprintNumbers()
{
   int n = 1;
label:
   printf("%d ",n);
   n++;
   if(n <= 10)
      goto label;
}

// Driver program to test above function
int main( ) {
   printNumbers( );
   return 0;
}
```

Output:

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

return: The return in C or C++ returns the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and return the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void function, a return value is must be returned.

Syntax: return[expression];

Example:

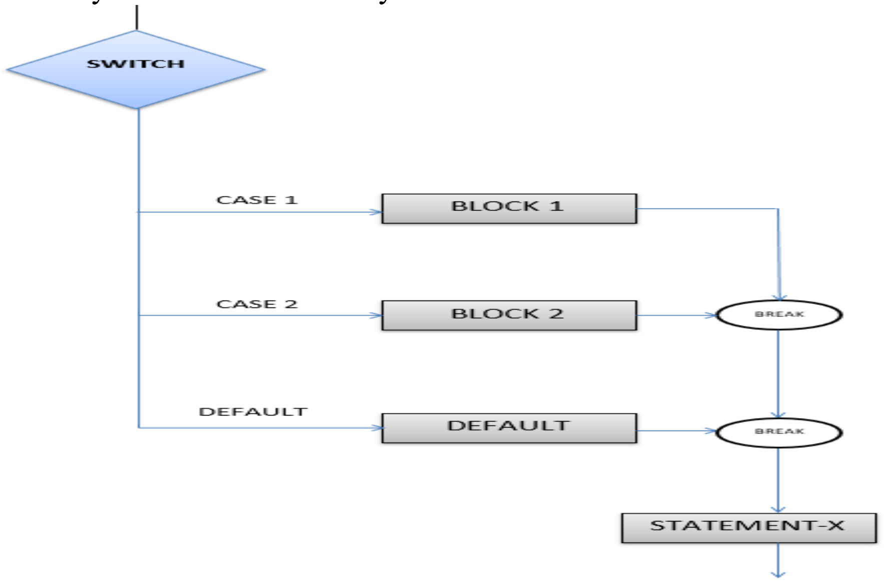
```
 // C code to illustrate return
#include <stdio.h>
 // non-void return type
// function to calculate sum
int SUM(int a, int b)
{
   ints1 = a + b;
   returns 1;
}
 // returns void// function to print
void print(int s2)
{
   printf("The sum is %d", s2);
   return;
}
int main( )
{
   int num1 = 10;
   int num2 = 10;
   int sum_of = SUM(num1, num2);
   printf(sum_of);
   return 0;
}
```

Output:

```
The sum is 20
```

Switch Statement:

C switch statement is used when you have multiple possibilities for the if statement. Switch case will allow you to choose from multiple options. When we compare it to a general electric switchboard, you will have many switches in the switchboard but you will only select the required switch, similarly, the switch case allows you to set the necessary statements for the user.



Syntax:

```
switch(variable)
{
case 1:
   //execute your code
break;

case n:
   //execute your code
break;

default:
   //execute your code
break;
}
```

Example program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int day;
    printf("Enter the day of the week (1 to 7) :");
    scanf("%d", &day);
    switch(day)
   {
        case1 :printf("Today is SUNDAY");break;
        case2 :printf("Today is MONDAY");break;
        case3 :printf("Today is TUESDAY");break;
        case4 :printf("Today is WEDNESDAY");break;
        case5 :printf("Today is THURSDAY");break;
        case6 :printf("Today is FRIDAY");break;
        case7 :printf("Today is SATURDAY");break;
        default:printf("Enter a valid choice(l to 7 only)");
   }
   getch();
}
```

OUTPUT:

```
Enter a valid choice(l to 7 only): 5
Today is THURSDAY
```

The following rules apply to a switch statement −

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Application of Switch Case:

The switch statement is convenient to be used if there are a large number of alternative paths. It is particularly used in menu-driven programs. It is efficient and faster than if statement. But it can only be used in case of equality (= =) operator but not with other relational operators and it cannot be used with multiple variables.

Loop Control Statements:

Looping Statements in C execute the sequence of statements or block of statementsmany times until the stated condition becomes false. A loop in C consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop toexecute until the specified condition becomes false. The purpose of the C loop is to repeat the same code a number of times.

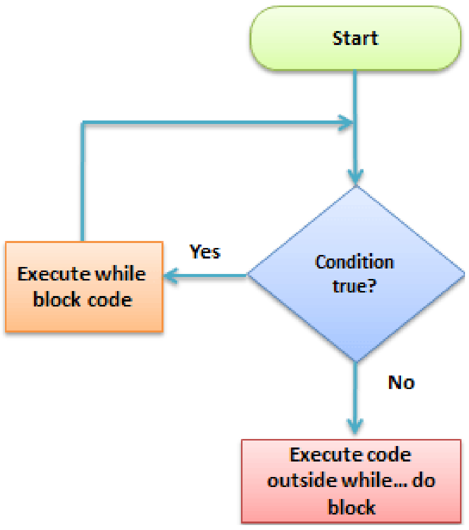
Types of Loops in C:

Depending upon the position of a control statement in a program, looping statement in C is classified into two types:

1. Entry controlled loop
2. Exit controlled loop

In an **entry control loop**, a condition is checked before executing the body of a loop. It is also called as a pre-checking loop.

In an **exit controlled loop**, a condition is checked after executing the body of a loop. It is also called as a post-checking loop.



The control conditions must be well defined and specified otherwise the loop will execute an infinite number of times. The loop that does not stop executing and processes the statements number of times is called as an infinite loop. An **infinite loop** is also called as an "**Endless loop**." Following are some characteristics of an infinite loop:

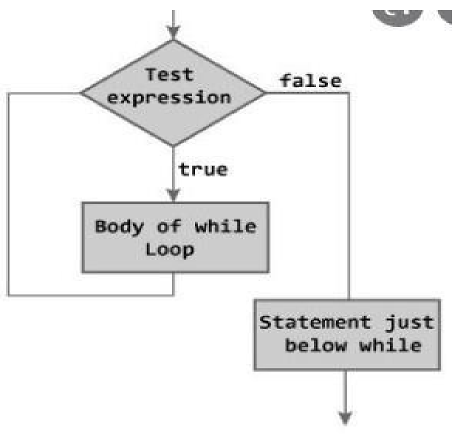
1. No termination condition is specified.
2. The specified conditions never meet.

The specified condition determines whether to execute the loop body or not.
'C' programming language provides us with three types of loop constructs:

1. The while loop
2. The do-while loop
3. The for loop

While Loop: In **while** loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only the body of a loop is executed.

Syntax:
```
while(condition)
{
//code to be executed
}
```



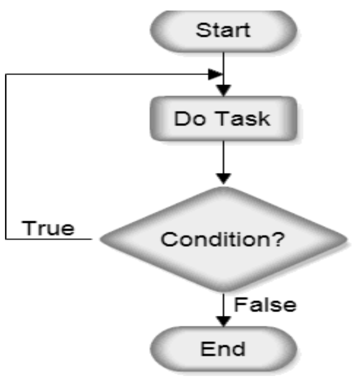
```
#include<stdio.h>
int main()
{
int count=1;
while(count <=4)
{
printf("%d ", count);
   count++;
}
return 0;
}
```

Output:
```
1234
```

step1: The variable count is initialized with value 1 and then it has been tested for the condition.
step2: If the condition returns true then the statements inside the body of while loop are executed else control comes out of the loop.
step3: The value of count is incremented using ++ operator then it has been tested again for the loop condition.

Do-While Loop: In a **do…while** loop, the condition is always executed after the body of a loop. It is also called an exit-controlled loop.
The do-while loop is mainly used in the case where we need to execute the loop at least once. The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user.

Syntax:
```
do
{
//code to be executed
}while(condition);
```



Example:
```
#include<stdio.h>
int main()
{
        int j=0;
        do
        {
        printf("Value of variable j is: %d\n", j);
                j++;
        }while(j<=3);
        return 0;
}
```
Output:
```
Value of variable j is:0
Value of variable j is:1
Value of variable j is:2
Value of variable j is:3
```

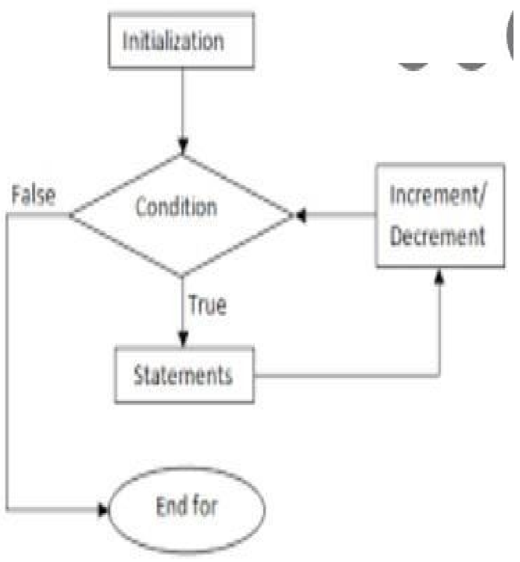
The do-while runs at least once even if the condition is false because the condition is evaluated, after the execution of the body of loop.

For Loop:In a **for** loop, the initial value is performed only once, then the condition tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.

This is one of the most frequently used loop in C programming.

Syntax of for loop:

```
for (initialization; condition test; increment or decrement)
{
      //Statements to be executed repeatedly
}
```



Example:

```
#include<stdio.h>
int main()
{
  int i;
 for(i=1;i<=3;i++)
  {
   printf("%d\n",i);
  }
  return 0;
}
```

Output:

```
1
2
3
```

Step 1: First initialization happens and the counter variable gets initialized.
Step 2: In the second step the condition is checked, where the counter variable is tested for the given condition, if the condition returns true then the C statements inside the body of for loop gets executed, if the condition returns false then the for loop gets terminated and the control comes out of the loop.
Step 3: After successful execution of statements inside the body of loop, the counter variable is incremented or decremented, depending on the operation (++ or –).

Various forms of for loop in C

I am using variable num as the counter in all the following examples –

1) Here instead of num++, I’m using num=num+1 which is same as num++.

```
for(num=10; num<20; num=num+1)
```

2) Initialization part can be skipped from loop as shown below, the counter variable is declared before the loop.

```
int num=10;
for(;num<20;num++)
```

Note: Even though we can skip initialization part but semicolon (;) before condition is must, without which you will get compilation error.

3) Like initialization, you can also skip the increment part as we did below. In this case semicolon (;) is must after condition logic. In this case the increment or decrement part is done inside the loop.

```
for(num=10; num<20;)
{
//Statements
```

```
    num++;
}
```
4) This is also possible. The counter variable is initialized before the loop and incremented inside the loop.
```
int num=10;
for(;num<20;)
{
//Statements
    num++;
}
```
5) As mentioned above, the counter variable can be decremented as well. In the below example the variable gets decremented each time the loop runs until the condition num>10 returns false.
```
for(num=20; num>10; num--)
```

Introduction to functions in C:

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main( )**, and all the most trivial programs can define additional functions.
You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.
A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.
The C standard library provides numerous built-in functions that your program can call. For example, **strcat( )** to concatenate two strings, **memcpy( )** to copy one memory location to another location, and many more functions.
A function can also be referred as a method or a sub-routine or a procedure, etc.
Defining a Function
The general form of a function definition in C programming language is as follows −
```
return_typefunction_name( parameter list ) {
  body of the function
}
```
A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example
Given below is the source code for a function called **max()**. This function takes two parameters **num1** and **num2** and returns the maximum value between the two −
```
/* function returning the max between two numbers */
                int max(int num1, int num2)
                {
                  /* local variable declaration */
                  int result;
                   if (num1 > num2)
```

```
        result = num1;
      else
        result = num2;
       return result;
    }
```
Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following **parts** −

```
return_typefunction_name( parameter list );
```
For the above defined function max( ), the function declaration is as follows −

```
int max(int num1, int num2);
```
Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

```
int max(int, int);
```
Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Storage classes in C:

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.

A storage class in C is used to describe the following things:

- The variable scope.
- The location where the variable will be stored.
- The initialized value of a variable.
- A lifetime of a variable.
- Who can access a variable?

There are four types of storage classes in C

1) Automatic
2) External
3) Static
4) 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	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

1) Automatic:

- Automatic variables are allocated memory automatically at runtime.auto keyword will be used.
- 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.

```
#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 garbagegarbage
```

2) **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:-

```
void test(); //Function declaration (discussed in next topic)main()
{
  test( );
  test( );
  test( );
}
void test( )
{
  staticinta=0;  //Static variablea=a+1;
  printf("%d\t",a);
}
```

output:

```
        1        2        3
```

3) 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 cannot dereference the register variables, i.e., we cannot 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 cannot be stored into the register since we cannot use more than one storage specifier for the same variable.

```
#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

```
#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);
^~~~~~
```

4) 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
```

Arrays:

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.

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.

Different Types of Arrays:

1) One Dimensional Array.
2) Two Dimensional Arrays.
3) Multi Dimensional Arrays. (Two or Three etc..)

Declaration of C Array

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

Syntax: typearrayName[size];

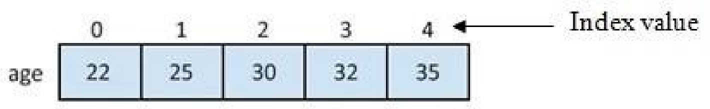
This is called a one-dimensional array. An array type can be any valid C data types, and array size must be an integer constant greater than zero.

Example:double amount[5];

Initialize an array:

Arrays can be initialized at declaration time:
int age[5]={22,25,30,32,35};

A pictorial representation of the Array:



Example Program:

```
#include<stdio.h>
int main( )
{
  int a[5]={10,20,30,40,50};
  int i;
  printf("elements of the array are");
  for(i=0;i<5;i++)
    printf("%d", a[i]);
}
```

Output:

```
Elements of the array are
10 20 30 40 50
```

Two multidimensional arrays

These are used in situations where a table of values have to be stored (or) in matrices applications.

Syntax

The syntax is given below −

datatype array_ name[row size] [column size];

For example: int a[5] [5];

a[0][0] 10	a[0][1] 20	a[0][2] 30
a[1][0] 40	a[1][1] 50	a[1][2] 60
a[2][0] 70	a[2][1] 80	a[2][2] 90

Following is the C Program for **compile time initialization** −

```
#include<stdio.h>
main()
{
   int a[3][3]={10,20,30,40,50,60,70,80,90};
   int i,j;
   printf("Elements of the array are");
   for(i=0;i<3;i++){
      for(j=0;j<3;j++){
         printf("%d \t", a[i][j]);
      }
      printf("\n");
   }
}
```

OUTPUT:

```
Elements of the array are:
10 20 30
40 50 60
70 80 90
```

Structure (struct):

Structure is a user-defined data type in a programming language that stores different data types' values together. The **struct** keyword is used to define a structure data type in a program. The struct data type stores one or more than one data element of different kinds in a variable.

Suppose that if we want to store the data of employee in our C/C++ project, where we have to store the following different parameters:

- Id
- Name
- Department
- Email Address

One way to store 4 different data by creating 4 different arrays for each parameter, such as **id[] name[]**, **department[]**, and **email[]**. Using array id[i] represents the id of the **i**th employee. Similarly, name[i] represents the name of **i**th employee (name). Array element department[i] and email[i] represent the **i**th employee's department and email address.

Syntax of struct

```
struct [structure_name]
{
   type member_1;
   type member_2;
   . . .
   type member_n;
};
```

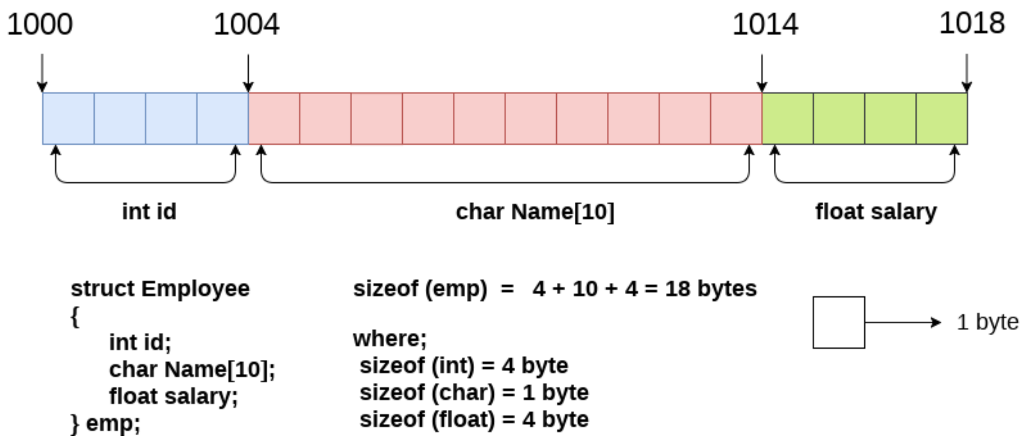
Example:

```
struct employee
{
   int id;
   char name[50];
   float  salary;
   char email[30];
};
```

Let's see the another example to define a structure for an entity employee in c.

```
struct employee
{   int id;
   char name[20];
   float salary;
};
```

The following image shows the memory allocation of the structure employee that is defined in the above example.



```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

```
sizeof (emp)  =   4 + 10 + 4 = 18 bytes

where;
 sizeof (int) = 4 byte
 sizeof (char) = 1 byte
 sizeof (float) = 4 byte
```

Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure. Let's understand it by the diagram given below:

Declaring structure variable:

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

```
struct employee
{   int id;
   char name[50];
   float salary;
};
```

Now write given code inside the main() function.

```
struct employee e1, e2;
```

The variables **e1** and **e2** can be used to access the values stored in the structure. Here, **e1** and **e2** can be treated in the same way as the objects in C++ and Java.

2nd way:

Let's see another way to declare variable at the time of defining the structure.

```
struct employee
{   int id;
   char name[50];
   float salary;
}e1,e2;
```

Which approach is good

If the number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

Accessing members of the structure

There are two ways to access structure members:

1. By **.** (member or dot operator)
2. By **->** (structure pointer operator)

Let's see the code to access the *id* member of *p1* variable by. (member) operator.

```
p1.id
```

Let's see another example of the structure in C language to store many employees information.

1. #include<stdio.h>

```
#include <string.h>
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;  //declaring e1 and e2 variables for structure
int main( )
{
  //store first employee information
  e1.id=101;
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
  e1.salary=56000;

 //store second employee information
  e2.id=102;
  strcpy(e2.name, "James Bond");
  e2.salary=126000;

  //printing first employee information
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
  printf( "employee 1 salary : %f\n", e1.salary);

  //printing second employee information
  printf( "employee 2 id : %d\n", e2.id);
  printf( "employee 2 name : %s\n", e2.name);
  printf( "employee 2 salary : %f\n", e2.salary);
  return 0;
}
```

OUTPUT:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : James Bond
employee 2 salary : 126000.000000
```

Union in C:

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

Union is a user-defined data type, but unlike structures, they share the same memory location.

Syntax of declaring Union

```
union [name of union]
  {
    type member1;
    type member2;
    type member3;
  };
```

Union is declared using the "union" keyword and name of union. Number 1, number 2, number 3 are individual members of union. The body part is terminated with a semicolon**;**.

Example of Union in C Programming

```
#include <stdio.h>
union item
{
   int x;
   float y;
   char ch;
};
```

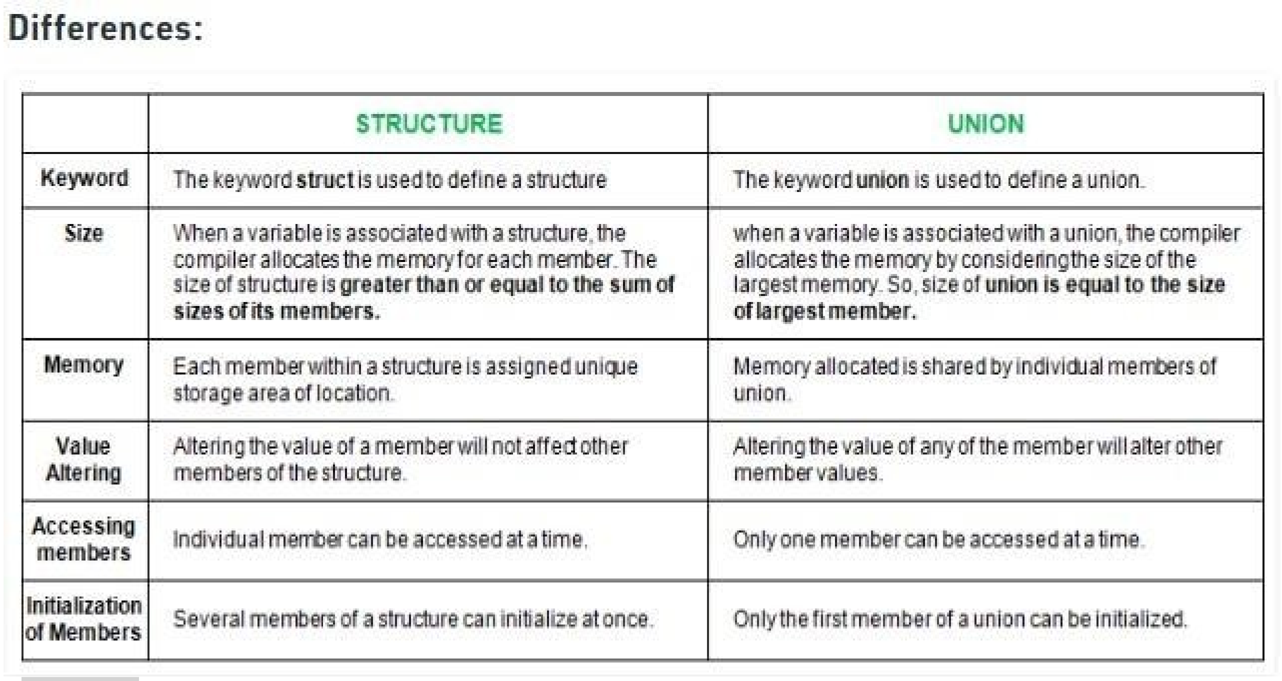
```
int main( )
{
   union item it;
it.x = 12;
it.y = 20.2;
   it.ch = 'a';
printf("%d\n", it.x);
printf("%f\n", it.y);
printf("%c\n", it.ch);
   return 0;
}
```

Output:

```
1101109601
20.199892
a
```

In the above program, you can see that the values of **x** and **y** gets corrupted. Only variable **ch** prints the expected result. It is because, in union, the memory location is shared among all member data types.

Therefore, the only data member whose value is currently stored, will occupy memory space. The value of the variable ch was stored at last, so the value of the rest of the variables is lost.



Differences:

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

What is Pointer in C?

The **Pointer** in C, is a variable that stores address of another variable. A pointer can also be used to refer to another pointer function. A pointer can be incremented/decremented, i.e., to point to the next/ previous memory location. The purpose of pointer is to save memory space and achieve faster execution time.

How to Use Pointers in C

If we declare a variable **v** of type **int**, **v** will actually store a value.

int v = 5;

v is equal to **5** now.

However, each variable, apart from value, also has its address (or, simply put, where it is located in the memory). The address can be retrieved by putting an ampersand (&) before the variable name.

&v

If you print the address of a variable on the screen, it will look like a totally random number (moreover, it can be different from run to run).

Now, what is a pointer? Instead of storing a value, a pointer will y store the address of a variable.

Pointer Variable:

syntax datatype *pointer_variablename =&variablename;

int *y = &v;

VARIABLE	POINTER
A **value** stored in a **named** storage/memory address	A **variable** that **points to** the storage/memory address of **another** variable

Declaring a Pointer

Like variables, pointers in C programming have to be declared before they can be used in your program. Pointers can be named anything you want as long as they obey C's naming rules. A pointer declaration has the following form.

data_type *pointer_variable_name;

Example: int *ptr3;

A simple program for pointer illustration is given below:

```
#include <stdio.h>
int main( )
{
  int a=10;    //variable declaration
  int *p;      //pointer variable declaration
  p=&a;        //store address of variable a in pointer p
printf("Address stored in a variable p is:%x\n",p);  //accessing the address
printf("Value stored in a variable p is:%d\n",*p);   //accessing the value
  return 0;
}
```

Output:

```
Address stored in a variable p is:60ff08
Value stored in a variable p is:10
```

Extra Information:

Types of Pointers in C

Following are the different **Types of Pointers in C**:

Null Pointer

We can create a null pointer by assigning null value during the pointer declaration. This method is useful when you do not have any address assigned to the pointer. A null pointer always contains value 0.

Following program illustrates the use of a null pointer:

```
#include <stdio.h>
int main()
{
        int *p = NULL;      //null pointer
        printf("The value inside variable p is:\n%x",p);
        return 0;
}
```

Output:

```
The value inside variable p is:
0
```

Void Pointer

In C programming, a void pointer is also called as a generic pointer. It does not have any standard data type. A void pointer is created by using the keyword void. It can be used to store an address of any variable.

Following program illustrates the use of a void pointer:

```
#include <stdio.h>
int main()
{
void *p = NULL;    //void pointer
printf("The size of pointer is:%d\n",sizeof(p));
return 0;
}
```

Output:

```
The size of pointer is:4
```

Wild pointer

A pointer is said to be a wild pointer if it is not being initialized to anything. These types of C pointers are not efficient because they may point to some unknown memory location which may cause problems in our program and it may lead to crashing of the program. One should always be careful while working with wild pointers.

Following program illustrates the use of wild pointer:

Following program illustrates the use of wild pointer:

```
#include <stdio.h>
int main()
{
int *p;     //wild pointer
printf("\n%d",*p);
return 0;
}
```

Output:

```
timeout: the monitored command dumped core
sh: line 1: 95298 Segmentation fault     timeout 10s main
```

Strings in C:

A String in C is nothing but a collection of characters in a linear sequence. 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.

char ch[10]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\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	8	9	10
j	a	v	a	t	p	o	i	n	t	\0

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

char ch[ ]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};

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

char ch[ ]="javatpoint";

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.

1) 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.
2) 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[10]={'a', 's', 'h', 'o', 'k', 'a', '\0'};
   char ch2[10]="ashoka";
printf("Char Array Value is: %s\n", ch);
printf("String Literal Value is: %s\n", ch2);
return 0;
}
```

Output

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

Command Line Arguments:

Command line means command prompt / command user interface. Arguments mean input values.So at the time of execution passing the input values from the command prompt to the program is called Command Line Arguments.

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using **main()** function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example:

```
#include<stdio.h>
intmain(intargc,char *argv[ ])
{printf(" \n Name of my Program %s \t",argv[0]);
if(argc==2)
{
printf("\n Value given by user is: %s \t",argv[1]);
}
elseif(argc>2)
{
printf("\n Many values given by users.\n");
}
else
{printf(" \n Single value expected.\n");
}
}
```

Output:

```
Name of the Program Example.c  Hello
Value given by user is: Hello
```

End of the UNIT-1

Data Structures

(Course code: 21F00104, for MCA - Regulations: R21)

UNIT – II

Overview of Data Structures: The data structure name indicates itself that organizing the data in memory in efficient manner. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. **Array** is a collection of memory elements in which data is stored sequentially, i.e., one after another.

In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory. Let's see the different types of data structures.

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

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

Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure:

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

Non-Primitive Data structure:

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

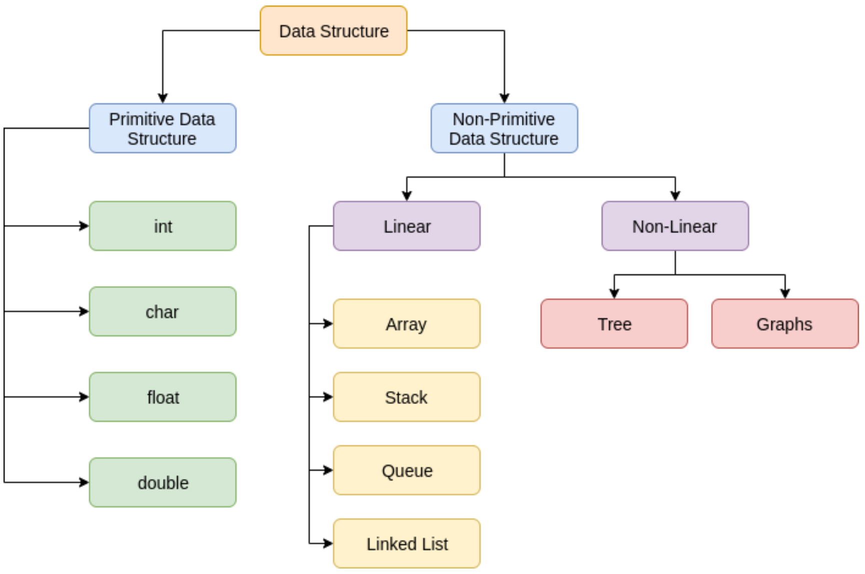
- Linear data structure
- Non-linear data structure

Linear Data Structure:

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

Non Linear Data Structure:

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



Data structures can also be classified as:

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

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

Major Operations we can perform using Data Structures are:

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

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

Representation of 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.

A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only.

Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.

In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

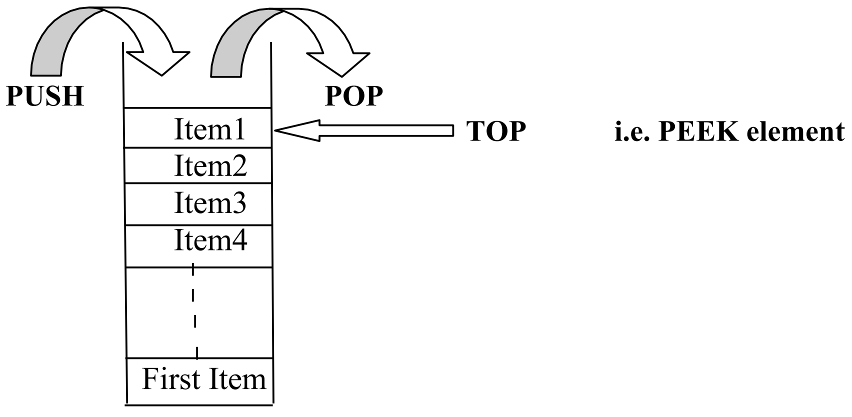
Some key points related to stack

It is called as stack because it behaves like a real-world stack, piles of books, etc.

A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.

It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Typical view of a stack data structure:



Stack

Stack Related Terms:

- The insertion and deletions takes place at one position is called "**TOP**" of the stack.
- An element in the stack is termed as "**ITEM**".
- Inserting an element into the stack is called "**PUSH**".
- Deleting the element from the stack is called "**POP**".
- Finding the TOP element in the stack is called "**PEEK**".

A stack may be represented in the memory in various ways. There are two main ways:
using a one-dimensional array and
using a single linked list.

Array Representation of Stacks: First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, the
items of the stack can be stored in a sequential fashion.

In Figure, Item **i** denotes the ***ith*** item in the stack; **l** and ***u*** denote the index range of the array in use; usually the values of these indices are 1 and SIZE respectively. TOP is a pointer to point the position of the array up to which it is filled with the items of the stack. With this representation, the following two ways can be stated:

EMPTY: $TOP < 1$
FULL: $TOP \geq u$

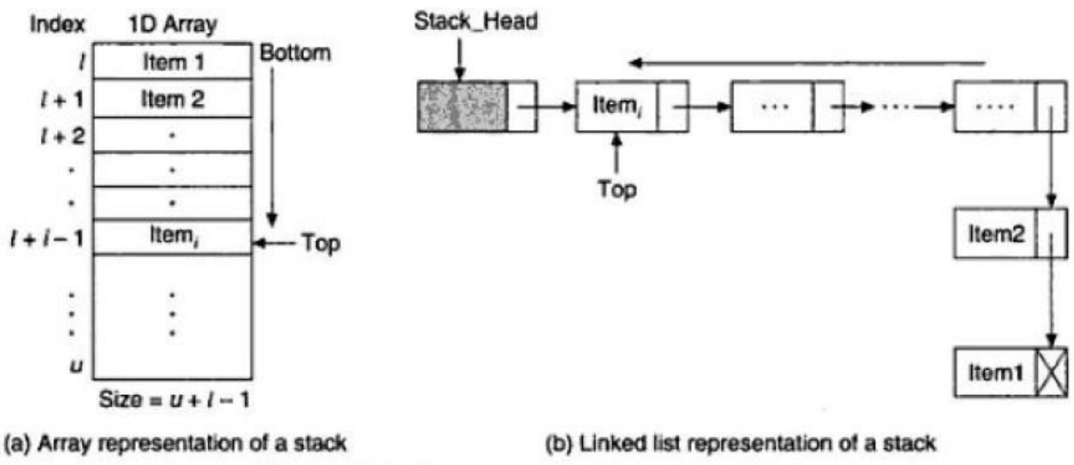


Figure 4.3 Two ways of representing stacks.

Linked List Representation of Stacks :

Although array representation of stacks is very easy and convenient but it allows the representation of only fixed sized stacks. In several applications, the size of the stack may vary during program execution. An obvious solution to this problem is to represent a stack using a linked list.

A single linked list structure is sufficient to represent any stack. Here, the DATA field is for the ITEM, and the LINK field is, as usual, to point to the next' item. Above Figure **b** depicts such a stack using a single linked list.

In the linked list representation, the **first node** on the list is the **current item** that is the item at the top of the stack and the **last node** is the node containing the bottom-most item. Thus, a **PUSH** operation will add a new node in the front and a **POP** operation will remove a node from the front of the list.

Standard Stack Operations:

The following are some common operations implemented on the stack:

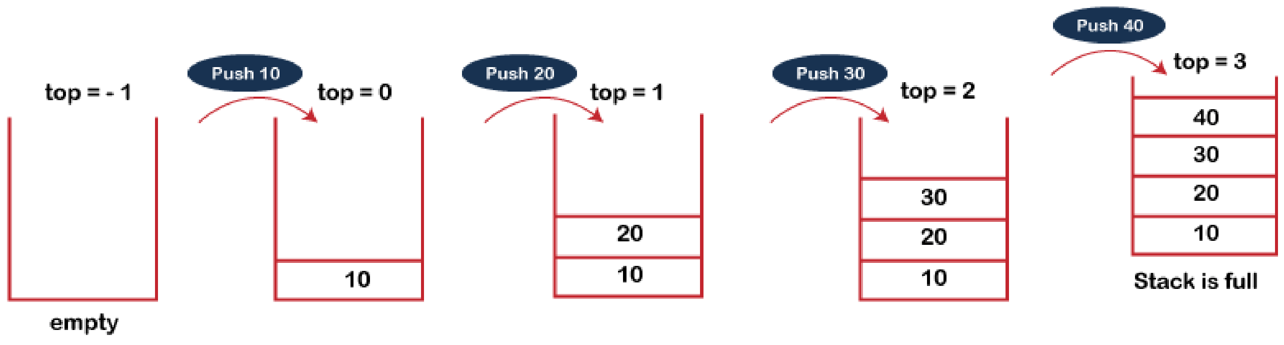
- **push( ):** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop( ):** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, 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 are given below:

1) Before inserting an element in a stack, we check whether the stack is full or not.
2) If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
3) When we initialize a stack, we set the value of top as -1 to check that the stack is empty.

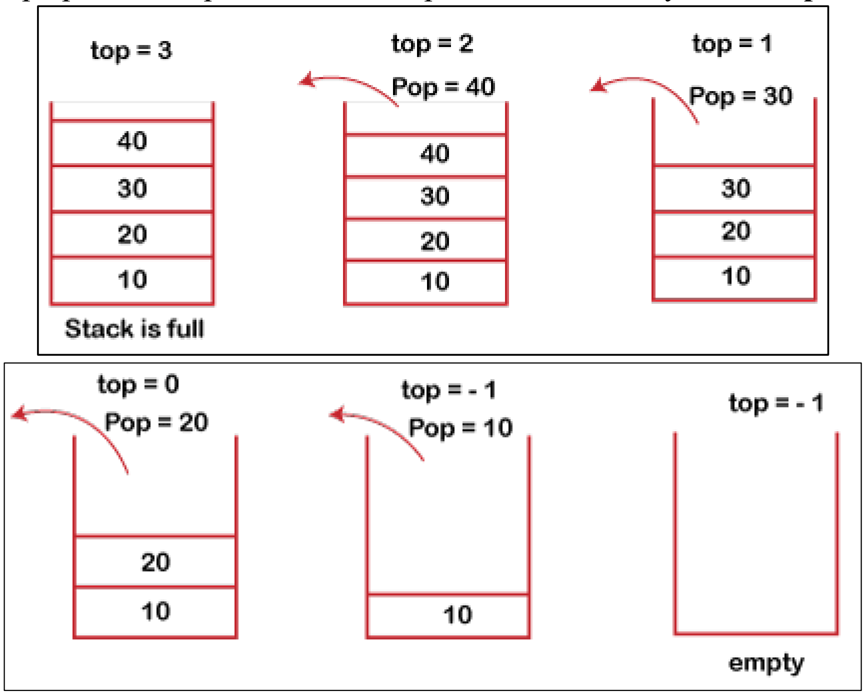
4) When the new element is **pushed** into a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
5) The elements will be inserted until we reach the ***max*** size of the stack.



POP operation:

The steps involved in the POP operation are given below:

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



Algorithm for PUSH:

This procedure pushes an item into Stack.

PUSH (STACK, TOP, MAX_SIZE, ITEM)

```
step 1.   IF   TOP = =MAX_SIZE, THEN        // test stack is full or not.
               PIRNT "OVERFLOW"
                    RETURN
          ELSE
step 2.   SET  TOP = TOP + 1;               // incrementing top or moving top
step 3.   SET  STACK [ TOP] = ITEM;         // placing item into stack at top
step 4.   RETURN
```

Algorithm for POP:

This procedure pops an item from the Stack.

```
POP (STACK, TOP, ITEM)
step 1.   IF  TOP = = -1, THEN                 //testing stack is empty or not
              PRINT  "UNDER FLOW"              // if true stack is empty
                  RETURN;
step 2.   SET  ITEM = STACK [TOP];             // deleteing element which is on top
step 3.   SET  TOP = TOP – 1;                  // next top must be decremented
step 4.   RETURN;
```

Algorithm for PEEK:

This procedure find the element in the top position of stack.

```
  PEEK (STACK, TOP, ITEM)
step 1.    IF  TOP = = -1, THEN                       //testing stack is empty or not
        PRINT  "UNDERFLOW"                            // printing stack is empty
        RETURN;
    ELSE
step 2.    RETURN   STACK [TOP]                       // element which is on top
```

Alogorithm for DISPLAY:

This procedure display the elements in the stack of array.

```
    Display(STACK, TOP, i)
step 1.     IF  TOP = -1,  THEN                       // testing stack is empty or not
            PRINT  "STACK IS EMPTY"        //if true printing stack is empty.
            RETURN;
step 2.     ELSE
   FOR (i= MAX_SIZE; i>-1; i- -)        //if else printing all elements in the stack by
         PRINT(STACK[ i ]);             // by using for loop, decrementing i value.
```

Representation of Stack:

Stack data structure can be **implementing in two ways**. They are as follows...

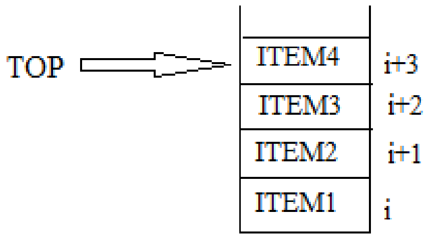
1. **Using Array**
2. **Using Linked List**

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

Stack Representation by Using Array:

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert(PUSH) or delete(POP) the values into that array by using **LIFO principle** with the help of a variable **'Top'**.

Initially Top is set to **-1**. Whenever we want to **PUSH** / insert a value into the stack, increment the top value by one and then insert. Whenever we want to **POP** / delete a value from the stack, then delete the top value and decrement the top value by one.

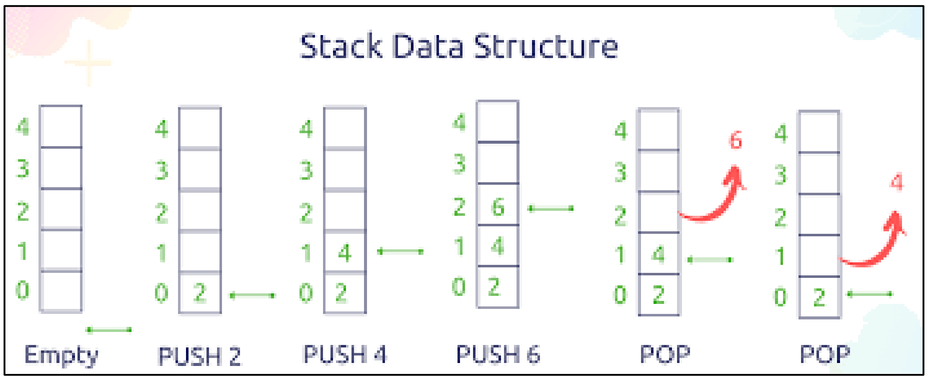


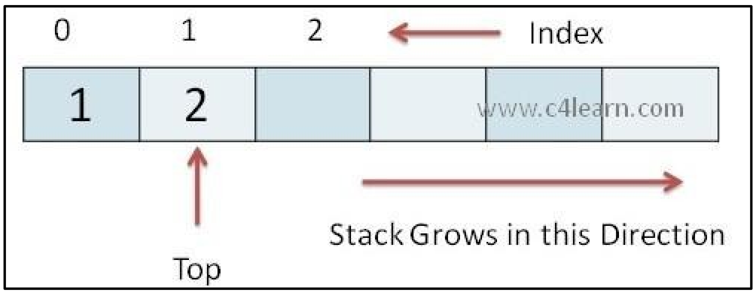
a [6] Stack using Array

Operations on Stacks:

Basic operations to manipulate a stack are as follows:

1. **Push** : This operation is used to insert an element on to the stack.
2. **Pop** : This operation is used to delete an element from the stack.
3. **Peek** : This operation is used to find out the Top element of the stack.
4. **Diplay** : This operation is used to dipslay all the elements in the stack**.**





Algorithm for PUSH:

This procedure pushes an item into Stack.

```
     PUSH (STACK, TOP, MAX_SIZE, ITEM)
step 5.   IF   TOP = =MAX_SIZE, THEN        // test stack is full or not.
             PIRNT "OVERFLOW"
                RETURN
          ELSE
step 6.   SET  TOP =  TOP + 1;             // incrementing top or moving top
step 7.   SET  STACK [ TOP] = ITEM;        // placing item into stack at top
step 8.   RETURN
```

Algorithm for POP:

This procedure pops an item from the Stack.

```
 POP (STACK, TOP, ITEM)
 step 5.   IF  TOP = = -1, THEN            //testing stack is empty or not
              PRINT  "UNDER FLOW"          // if true stack is empty
                  RETURN;
 step 6.   SET  ITEM = STACK [TOP];        // deleteing element which is on top
 step 7.   SET  TOP = TOP – 1;             // next top must be decremented
 step 8.   RETURN;
```

Algorithm for PEEK:

This procedure finds the element in the top position of stack.

```
    PEEK (STACK, TOP, ITEM)
step 3.    IF  TOP = = -1, THEN                  //testing stack is empty or not
        PRINT  "UNDERFLOW"                       // printing stack is empty
        RETURN;
    ELSE
step 4.    RETURN   STACK [TOP]                  // element which is on top
```

Alogorithm for DISPLAY:

This procedure displays the elements in the stack of array.

```
        Display(STACK, TOP, i)
step 3.     IF  TOP = -1,  THEN                       // testing stack is empty or not
            PRINT  "STACK IS EMPTY"           //if true printing stack is empty.
            RETURN;
step 4.     ELSE
        FOR (i= MAX_SIZE; i>-1; i- -)          //if else printing all elements in the stack by
           PRINT(STACK[ i ]);                  // by using for loop, decrementing i value.
```

Stack Representation by using Linked List:

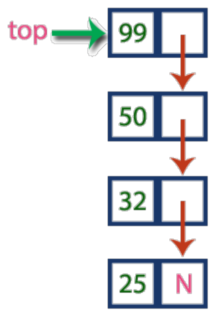
The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going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

Definition:

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

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

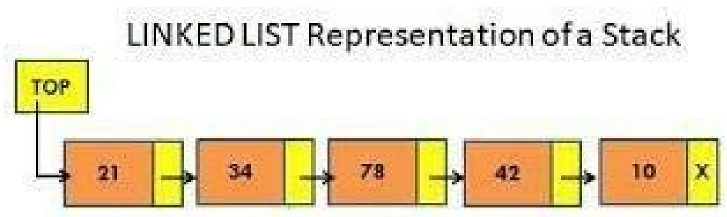
Example



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

Operations: Basic operations to manipulate a stack are as follows:

1. **Push** : This operation is used to insert an element in to the stack.
2. **Pop** : This operation is used to delete an element from the stack.
3. **Peek** : This operation is used to find out the Top element of the stack.
4. **Diplay** : This operation is used to display all the elements in the stack.



Stack Operations using Linked List

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

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode** → **next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode** → **next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

pop( ) - Deleting an Element from a Stack

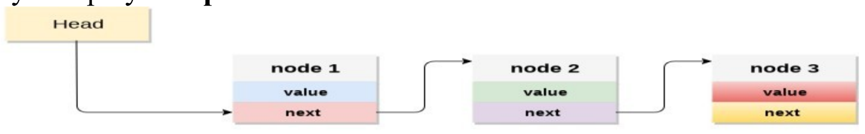
We can use the following steps to delete a node from the stack...

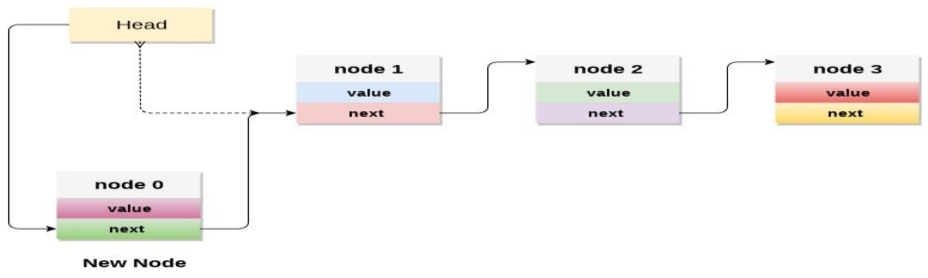
- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top** = **top** → **next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

display( ) - Displaying stack of elements

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

- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2 -** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp** → **data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp** → **next** != **NULL**).
- **Step 5 -** Finally! Display '**temp** → **data** ---> **NULL**'.





Program: **Note:** Write a Program to implement the Stack operations using an Single Linked List? (Dont forget if required you have to write example program if they asked for 12 marks. See the Lab Program)

Applications of STACK:

In our computers Both hardware and software stacks have been used to support four major computing areas in computing requirements: here we will learn two major applications.

1) A classical application of stack is "**Evaluation of Arithmetic Expression**".
   Here compiler uses a stack to translate input arithmetic expression into their corresponding Object code.

2) Another important application of stack is during the execution of "**Recursive Program**". It uses run time stacks for storing recursive functions.

What is an Expression?

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

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

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

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

Expression Types

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

1. **Infix Expression**
2. **Prefix Expression**
3. **Postfix Expression**

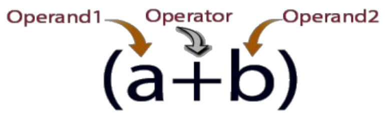
1) Infix Expression: (or **Infix Notation**)

In infix expression, operator is used in between operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example



2) Prefix Expression: (or **Polish Notation**)

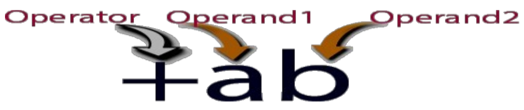
In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**".

The general structure of Prefix expression is as follows...

Next page….

Operator Operand1 Operand2

Example

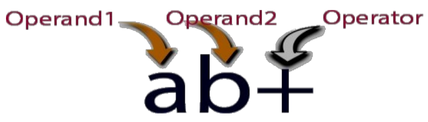


3) Postfix Expression: (or **Reverse Polish Notation** or **Sufix Notation**)

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

Operand1 Operand2 Operator

Example



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

Evaluation of arithmetic expression in a computer took two-step process.

1. First convert the infix to postfix expression.
2. Then evaluate the postfix expression by using STACK.

Infix to Postfix Conversion using STACK Data Structure:

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

Let ‘Q’ be an arithmetic expression written in “INFIX notation”. ‘Q’ contains operands, operators (Like **+, -, *,** / etc) and left & right paranthesis.

Here a stack follows the operator precedency & associations of an operators. For example as below.

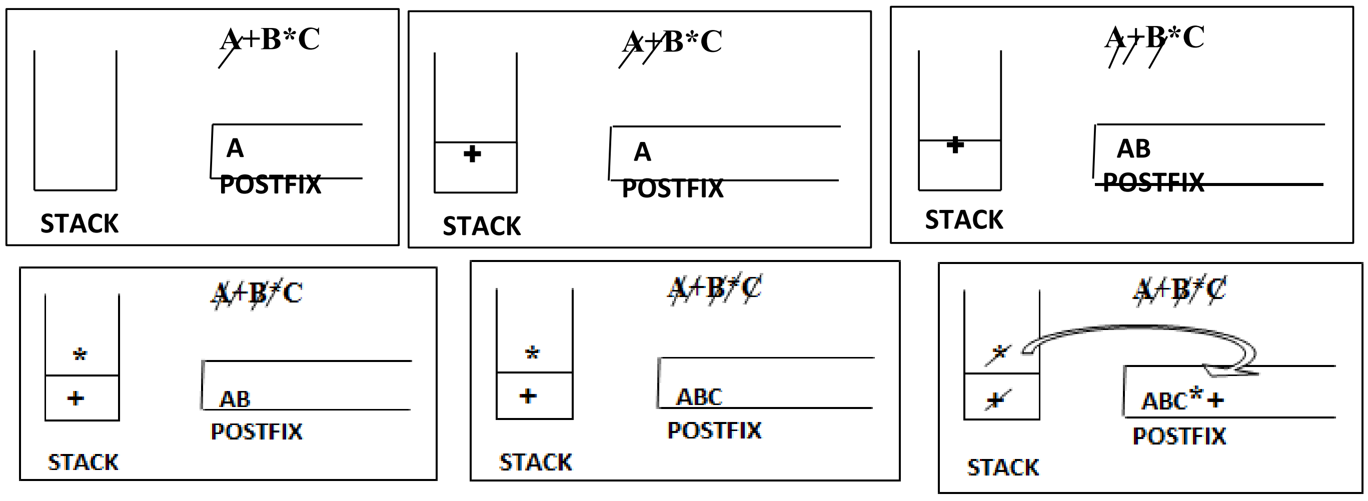
- **( )** 1st highest precedence (left to right associativity).
- ***** / 2nd Precedence
- **+ -** 3rd Precedence

Algorithm to convert the Infix expression into Postfix expression:

Read /scan the given infix expression from left side to right side, observe the scanned character.

1) If the character is LEFT Parenthesis, then PUSH it into the STACK.
2) If the character is an OPERAND, then ADD it to the POSTFIX Expression.
3) If the character is an OPERATOR, then check whether STACK is Empty or not.
   a) If the STACK is empty, then PUSH operator into the STACK.
   b) If the STACK is not empty, then check the priority of the operator.
      i) If the priority of the operator **>** operator present at TOP of the STACK, then PUSH operator into STACK.
      ii) If the priority of operator **<=** operator present at TOP of the STACK, then POP the operator from the STACK and ADD it to the POSTFIX expression and go to step **i)**
4) If the character is RIGHT Parenthesis, then POP all the operators from the STACK until it reaches LEFT Parenthesis and ADD to the POSTFIX expression.
5) After reading the all characters in INFIX, if STACK is not empty then POP all and ADD to POSTFIX.

Example1: The given Infix expression is: **A+B*C**



Finally we get POSTFIX expression **ABC*+**

Like this we can convert the INFIX to POSTFIX expression by using STACK. This process will be done by the computer also.

Example2: Given expression is: **A + B * (C – D)** **need to convert into POSTFIX**

By using above algorithm steps we will get POSTFIX notation as below.

See the table in next page.

INFIX	POSTFIX	STACK
A	A	(
A+	A	+
A+B	AB	+

A+B*	AB	+*
A+B*(	AB	+*(
A+B*(C	ABC	+*(
A+B*(C-	ABC	+*(-
A+B*(C-D	ABCD	+*(-
A+B*(C-D)	ABCD-	+*(
	ABCD-*	+
	ABCD-*+	POSTFIX expression

Evaluation of POSTFIX Expression:

After converting from Infix expression to Postfix expression. We can easily evaluate the Postfix expression by using STACK.

ALGORITHM:

Item Read from POSTFIX expression	Action
IF OPERAND encountered	PUSH it onto the stack
IF OPERATOR encountered	POP the two OPERANDS from the stack and apply the OPERATOR to them. Then PUSH the result into stack.
OUTPUT	POP the result from the stack after completion of Postfix expression.

Example:

With the previous POSTFIX expression ABCD-*+

Let consider simple values like A = 1, B = 2, C = 3, D = 4

ABCD-*+ or 1234 -*+

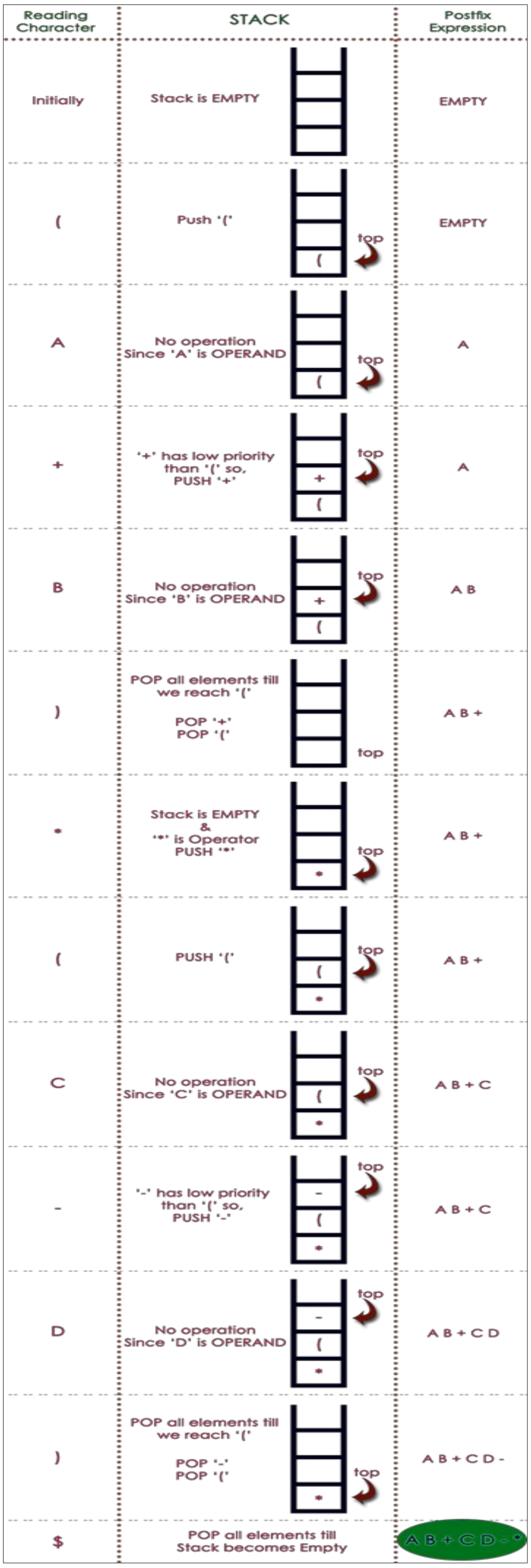
Character scanned from POSTFIX expression.	OPERATION	STACK
1		1
2		1, 2
3		1, 2, 3
4		1, 2, **3, 4**
-	3 - 4	1, 2, **-1**
*	2 * - 1	1, -2
+	1 + (-2)	-1

Rough work:
To solve manually:
A+B*(C-D)
1+2*(3-4)
1+2*(-1)
1+(-2)
1-2
-1 is the result

Example for Infix to Postfix:

Consider the following Infix Expression... (A + B) * (C - D)
The given infix expression can be converted into postfix expression using Stack data Structure as follows...

In next page



Reading Character	STACK		Postfix Expression
Initially	Stack is EMPTY		EMPTY
(	Push '('	(	EMPTY
A	No operation Since 'A' is OPERAND	(	A
+	'+' has low priority than '(' so, PUSH '+'	+ (	A
B	No operation Since 'B' is OPERAND	+ (	A B
)	POP all elements till we reach '(' POP '+' POP '('		A B +
*	Stack is EMPTY & '*' is Operator PUSH '*'	*	A B +
(	PUSH '('	(*	A B +
C	No operation Since 'C' is OPERAND	(*	A B + C
-	'-' has low priority than '(' so, PUSH '-'	- (*	A B + C
D	No operation Since 'D' is OPERAND	- (*	A B + C D
)	POP all elements till we reach '(' POP '-' POP '('	*	A B + C D -
$	POP all elements till Stack becomes Empty		A B + C D - *

The final Postfix Expression is... A B + C D - *

Another application of STACK

RECURSION:

A function that calls itself is called a **recursive function** and this technique is called recursion. A recursive function will call itself until a final call that does not require a call to itself is made.

Example: Finding factorial of a number.

n! = n x (n-1) x (n-2) x - - - - - - 3 x 2 x 1

n! = n x (n -1)!

ALGORITHM:

```
        FACTORIAL( N)
step 1.    IF (N = = 0) THEN
              RETURN  1;
step 2.    ELSE
              Fact = N x FACTORIAL (N – 1)
step 3.    END IF
step 4.    RETURN (Fact)
step 5.    STOP
```

```c
#include <stdio.h>
int Fact(int);
int main( )
{
 int num, val;
 //read a number from the user
 printf("Enter the number: ");
 scanf("%d", &num);
 //call fact function
 val = Fact(num);
 //print result
 printf("Factorial of %d = %d", num, val);
 return 0;
}
//function to calculate factorial
int Fact(int n)
{
 if (n == 1)
   return 1;
 else
   return (n * Fact(n-1));
}
```

OUTPUT:

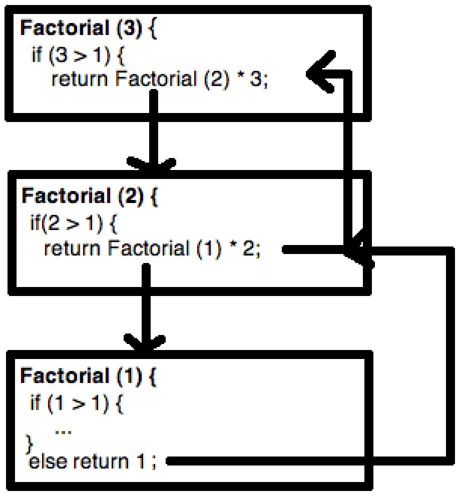
```
Enter the number: 5
Factorial of 5 = 120
```

To implement the above, we require two stacks.

1. One for storing the parameter N and
2. Another to hold the return address.

   Example:

   FACTORIAL(3)



```
Factorial (3) {
  if (3 > 1) {
    return Factorial (2) * 3;

Factorial (2) {
  if(2 > 1) {
    return Factorial (1) * 2;

Factorial (1) {
  if (1 > 1) {
    ...
  }
  else return 1 ;
```

So **2 x 3 = 6** final results is **6**

So **1 x 2 = 2** Result returns **2**

Result returns **1**

Implementation of a Stack:

Stack is a linear data structure. Which follows **L**ast **I**n **F**irst **O**ut principle. We can implement the stack by using C program. Here we can implement in two ways.

1) Using Arrays 2) Using Linked List.

Implementation of Stack Using Array in C:

The C Program is written for implementation of STACK using Array, the basic operations of stack are PUSH( ) and POP( ).

PUSH function in the code is used to insert an element to the top of stack, POP function used to remove the element from the top of stack. The display function in the code is used to print the values. All stack functions are implemented in C Code.

C Program for STACK Using Arrays:

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main( )
{
   //clrscr( );
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t--------------------");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
   do
   {
      printf("\n Enter the Choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push( );
            break;
         }
         case 2:
         {
            pop( );
            break;
         }
         case 3:
         {
            display( );
            break;
         }
         case 4:
         {
            printf("\n\t EXIT POINT ");
            break;
         }
         default:
         {
            printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
            }
        }
    }
    while(choice!=4);
    return 0;
}
void push( )
{
    if(top>=n-1)
    {
        printf("\n\tSTACK is over flow");
    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
        top++;
        stack[top]=x;
    }
}
void pop( )
{
    if(top<=-1)
    {
        printf("\n\t Stack is under flow");
    }
    else
    {
        printf("\n\t The popped elements is
                                        %d",stack[top]);
        top--;
    }
}
void display( )
{
    if(top>=0)
    {
        printf("\n The elements in STACK \n");
        for(i=top; i>=0; i--)
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }

}
```

```
OUTPUT:
Enter the size of STACK[MAX=100]:10

         STACK OPERATIONS USING ARRAY
        --------------------------------
         1.PUSH
         2.POP
         3.DISPLAY
         4.EXIT
 Enter the Choice:1
 Enter a value to be pushed:12

 Enter the Choice:1
 Enter a value to be pushed:24

 Enter the Choice:1
 Enter a value to be pushed:98

 Enter the Choice:3

 The elements in STACK

98
24
12
 Press Next Choice
 Enter the Choice:2

         The popped elements is 98
 Enter the Choice:3

 The elements in STACK

24
12
 Press Next Choice
 Enter the Choice:4

         EXIT POINT
```

C Program to Implement the STACK Operations Using Linked List:
The same implementation of stack using c is written using pointers:

```
#include<stdio.h>
#include<conio.h>
struct Node
```

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

OUTPUT:

```
:: Stack using Linked List ::
******** MENU  ********
    1. Push
    2. Pop
    3. Display
    4. Exit
     Enter your choice: 1
     Enter the value to be insert: 20
     Insertion is Success!!!
******** MENU  ********
    1. Push
    2. Pop
    3. Display
    4. Exit
     Enter your choice: 1
     Enter the value to be insert: 30
     Insertion is Success!!!
******** MENU  ********
    1. Push
    2. Pop
    3. Display
    4. Exit
     Enter your choice: 1
     Enter the value to be insert: 40
     Insertion is Success!!!
******** MENU  ********
    1. Push
    2. Pop
    3. Display
    4. Exit
     Enter your choice: 3
     40--->30--->20--->NULL
******** MENU  ********
    1. Push
```

```
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else{
    struct Node *temp = top;
    while(temp->next != NULL){
           printf("%d--->",temp->data);
           temp = temp -> next;
    }
    printf("%d--->NULL",temp->data);
  }
}
```

Queues:

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

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. Means in a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end.

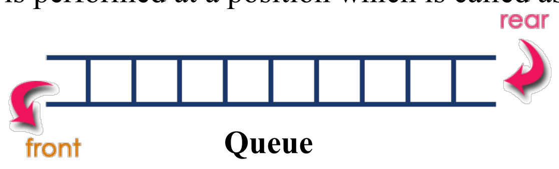
In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.

Various Positions of Queue:

When we see the positions of queue, we found them as below.

Front: The deletion operation is performed at a position which is known as '**front**'

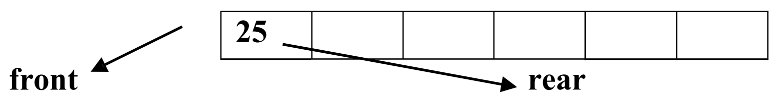
Rear: The insertion operation is performed at a position which is called as **'rear'**.



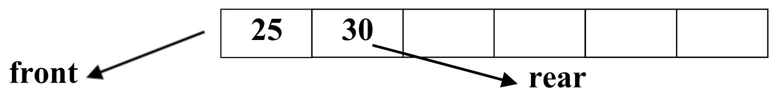
Example:

Inserting 25, 30, 51, 60 and 85 into QUEUE.

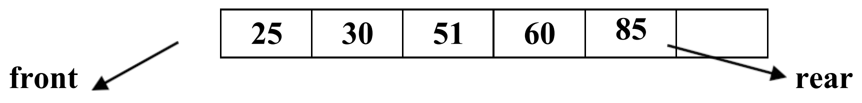
Inserting 25 into queue…



Inserting 30 into queue…



Similarly Inserting 51, 60 and 85 into queue…



Representation of Queue:

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

1. Queue Representing by **Using Array**
2. Queue Representing by **Using Linked List**

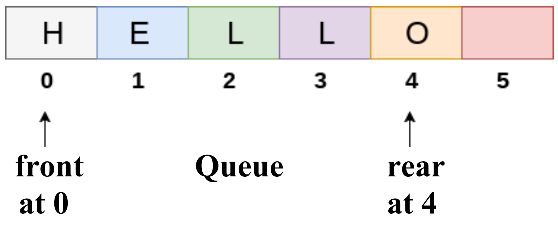
When a queue is implemented using an array, that queue can organize an only limited number of elements.

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

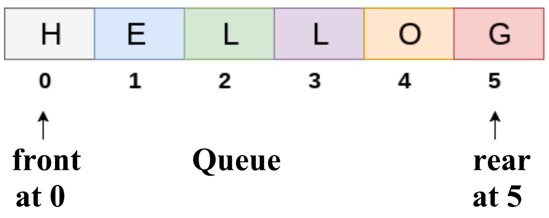
1) Queue Representing by Using Array:

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.

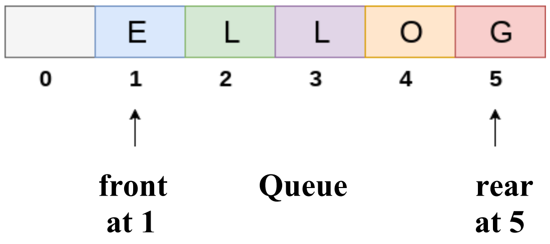


After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting the new element

After deleting an element, the value of front will increase from -1 to 0. However, the queue will look something like following. Deleting the element will be done only at front.



Queue after deleting the element

Different operations in the Queue:

In the queue concept we can perform the following operations:

1) Insertion,
2) Deletion,
3) Searching Operations.

Insertion and deletion operations will be termed as follows.

The following operations are performed on a queue data structure...

1. **enQueue(value) -** To insert an element into the queue
2. **deQueue( ) -** To delete an element from the queue

display( ) - To display the elements of the queue

1) Insertion operation: (enqueue operation)

Algorithm to insert any element in a queue:

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

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

Otherwise keep increasing the index 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 stop
      [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
```

2) Deletion operation: (dequeue operation)

Algorithm to Deleting the element from the queue:

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

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

Algorithm

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

3) Searching operation in queue:

Algorithm to search the element in the queue:

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

Otherwise, compare the key element value from front to rear by incrementing the front index value, until it founds.

Algorithm

```
o Step 1: IF FRONT = = -1 or FRONT > REAR
        Write UNDERFLOW
      ELSE
          SET INDEX = FRONT
      Step 2:  IF KEY =  = QUEUE[INDEX]
               PRINT Element found at Index value
          ELSE
              INDEX = INDEX+1
              REPEAT STEP 2
              STOP
       ELSE
              SET FRONT = FRONT + 1
      REPEAT THE COMPARISION UNTIL FRONT = = REAR
      [END OF IF]
Step 2: EXIT
```

Queue Using Linked List

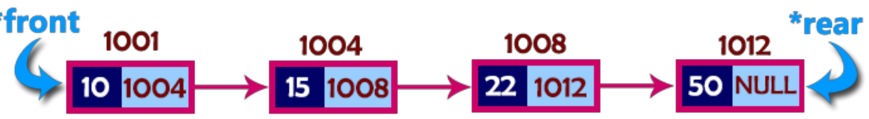
The major problem with the queue implemented using an array is it will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use.

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

The Queue implemented using linked list can organize as many data values as we want.

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

Example



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

Operations

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

- **Step 1** - Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2** - Define a '**Node**' structure with two members **data** and **next**.
- **Step 3** - Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4** - Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

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

- **Step 1** - Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- **Step 2** - Check whether queue is **Empty** (**rear** == **NULL**)
- **Step 3** - If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- **Step 4** - If it is **Not Empty** then, set **rear** → **next** = **newNode** and **rear** = **newNode**.

deQueue( ) - Deleting an Element from Queue

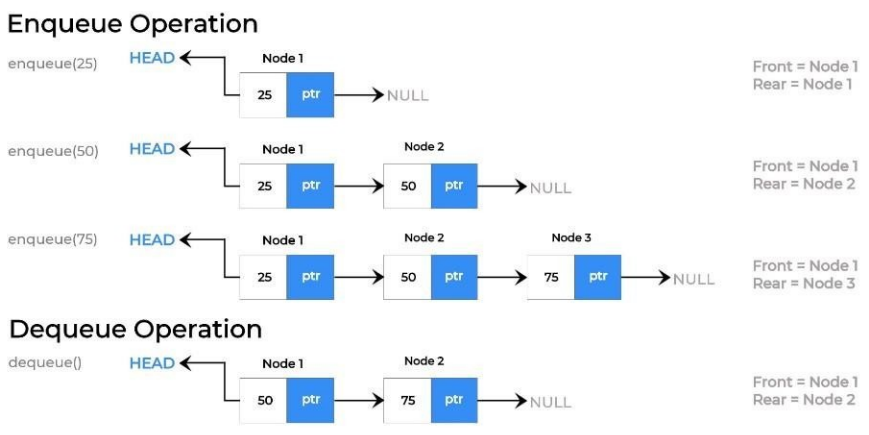
We can use the following steps to delete a node from the queue...

- **Step 1** - Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2** - If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3** - If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4** - Then set '**front** = **front** → **next**' and delete '**temp**' (**free(temp)**).

display( ) - Displaying the elements of Queue

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

- **Step 1** - Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2** - If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3** - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4** - Display '**temp** → **data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp** → **next** != **NULL**).
- **Step 5** - Finally! Display '**temp** → **data** ---> **NULL**'.



Front of the queue after enqueue and dequeue operation is 50

C 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( )
```

```c
{
    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]);
     }
   }
 }
```

OUTPUT:

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

Enter your choice ?1
Enter the element
123
Value inserted
*************Main Menu**************
==============================================
1.insert an element
2.Delete an element
3.Display the queue
4.Exit
Enter your choice ?1
Enter the element
90
Value inserted
*************Main Menu**************
===================================
1.insert an element
2.Delete an element
3.Display the queue
4.Exit
Enter your choice ?2
value deleted
*************Main Menu**************
==============================================
1.insert an element
2.Delete an element
3.Display the queue
4.Exit
Enter your choice ?3
printing values .....
90
*************Main Menu**************
==============================================
1.insert an element
```

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

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

void main( )
{
  int choice, value;
  clrscr();
  printf("\n:: Queue Implementation using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
           case 1: printf("Enter the value to be insert: ");
                   scanf("%d", &value);
                   insert(value);
                   break;
           case 2: delete( ); break;
           case 3: display( ); break;
           case 4: exit(0);
           default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}
void insert(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode -> next = NULL;
  if(front == NULL)
    front = rear = newNode;
  else{
    rear -> next = newNode;
    rear = newNode;
  }
  printf("\nInsertion is Success!!!\n");
}
void delete( )
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    front = front -> next;
    printf("\nDeleted element: %d\n", temp->data);
    free(temp);
  }
}
void display( )
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    while(temp->next != NULL){
           printf("%d--->",temp->data);
           temp = temp -> next;
    }
    printf("%d--->NULL\n",temp->data);
  }
}
```

OUTPUT:

```
*********** MENU **********
1.  Insert
2.  Delete
3.  Display
4.  Exit
Enter your choice: 1
Enter the value to be insert: 8
Insertion is Success!!!

*********** MENU **********
1.  Insert
2.  Delete
3.  Display
4.  Exit
Enter your choice: 1
Enter the value to be insert: 12
Insertion is Success!!!

*********** MENU **********
1.  Insert
2.  Delete
3.  Display
4.  Exit
Enter your choice: 1
Enter the value to be insert: 6
Insertion is Success!!!
*********** MENU **********
1.  Insert
2.  Delete
3.  Display
4.  Exit
Enter your choice: 3
8--->12--->6--->NULL
```

Tower of Hanoi:

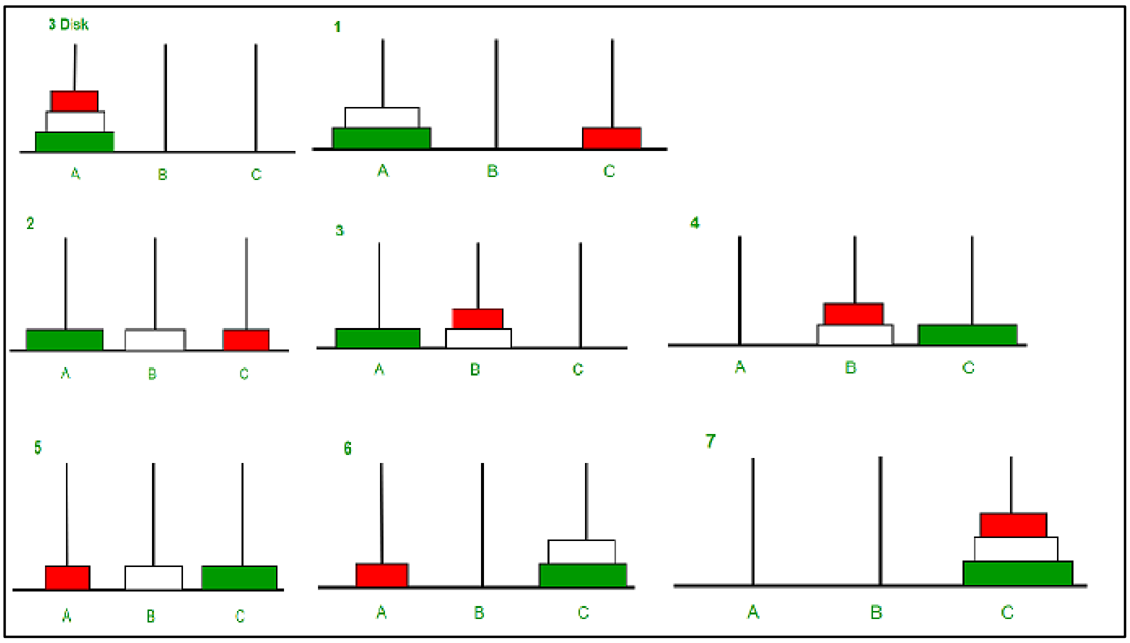
Tower of Hanoi is a mathematical puzzle where we have three rods (pegs) and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

Procedure / Approach :

Take an example for 2 disks :
Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.
Step 1 : Shift first disk from 'A' to 'B'.
Step 2 : Shift second disk from 'A' to 'C'.
Step 3 : Shift first disk from 'B' to 'C'.
The pattern here is :
Shift 'n-1' disks from 'A' to 'B'.
Shift last disk from 'A' to 'C'.
Shift 'n-1' disks from 'B' to 'C'.

Image illustration for 3 disks:



Examples:

Input : 2
Output : Disk 1 moved from A to B
Disk 2 moved from A to C
Disk 1 moved from B to C
Input : 3
Output : Disk 1 moved from A to C
Disk 2 moved from A to B
Disk 1 moved from C to B
Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C

Towers of Hanoi program in 'C':

This C Program uses recursive function & solves the tower of hanoi. The tower of hanoi is a mathematical puzzle. It consists of threerods, and a number of disks of different sizes which can slideonto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top. We have to obtain the same stack on the third rod.
Here is the source code of the C program for solving towers of hanoi. The C Program is successfully compiled and run on a Linux system. The program output is also shown below.

```
/*
 * C program for Tower of Hanoi using Recursion
 */
#include <stdio.h>

void towers(int, char, char, char);

int main( )
{
    int num;

    printf("Enter the number of disks : ");
    scanf("%d", &num);
    printf("The sequence of moves involved in the Tower of Hanoi are :\n");
    towers(num, 'A', 'C', 'B');
    return 0;
}
void towers(int num, char frompeg, char topeg, char auxpeg)
{
    if (num == 1)
    {
        printf("\n Move disk 1 from peg %c to peg %c", frompeg, topeg);
        return;
    }
    towers(num - 1, frompeg, auxpeg, topeg);
    printf("\n Move disk %d from peg %c to peg %c", num, frompeg, topeg);
    towers(num - 1, auxpeg, topeg, frompeg);
}
```

OUTPUT:

```
Enter the number of disks: 3
The sequence of moves involved in the Tower of Hanoi are:

Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg C
```

You prepare any of one of the program (another program given in next page)

Another program for Towers of Hanoi:

```
#include<stdio.h>
void TOH(int n,char x,char y,char z)
{
  if(n>0) {
    TOH(n-1,x,z,y);
    printf("\n%c to %c",x,y);
    TOH(n-1,z,y,x);
  }
}
int main() {
  int n=3;
  TOH(n,'A','B','C');
}
```

Output

```
A to B
A to C
B to C
A to B
C to A
C to B
A to B
```

END OF THE UNIT-2

Data Structures

(Course code: 21F00104, for MCA - Regulations: R21)

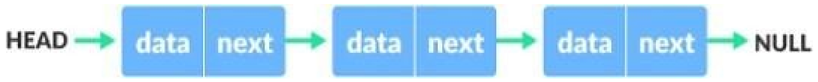
UNIT – III

Topics: Linked Lists–Pointers, 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.

******

Linked Lists:

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted but in Linked list if we have the head node then we can traverse to any node through it and insert new node at the required position.

For **example**, in a system, if we maintain a sorted list of IDs in an array id[ ].

id[ ] = [1000, 1010, 1050, 2000, 2040].

And if we want to **insert** a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[ ], everything after 1010 has to be moved due to this so much work is being done which affects the efficiency of the code.

Advantages over arrays: (advantages of linked list)

1) Dynamic size: Linked List will allow to increase the size whenever required and as well as decrease the size when not required at run time.

2) Ease of insertion/deletion: Linked List will allow insertion and deletion whenever we required.

Drawbacks in Linked List:

1) Random access is not allowed. We have to access elements sequentially starting from the first node (head node). So we cannot do binary search with linked lists efficiently with its default implementation.

2) Extra memory space for a pointer is required with each element of the list.

3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Linked List Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head points to NULL.

Each node in a list consists of at least two parts:

1) **data** (we can store integer, strings or any type of data).

2) **Pointer** (Or Reference) to the next node (connects one node to another)

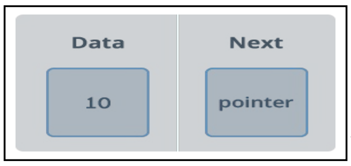
In C, we can represent a node using structures. Below is an example of a linked list node with integer data.

```
// A linked list node
struct Node
{
  int data;
  struct Node* next;
};
```

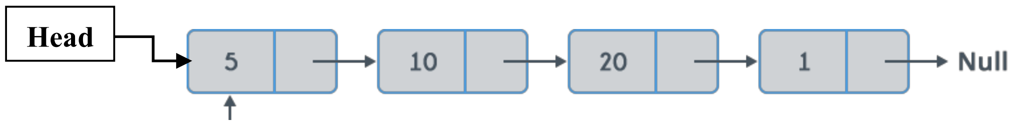
Linked Lists–Pointers:

A linked list is a list constructed using pointers. A linked list is not fixed in size but can grow and shrink while your program is running. Here we will learn how to define and manipulate linked lists, which will serve to introduce you to a new way of using pointers.

Nodes A structure like the one shown in belowdiagram: it consists of items that we have drawn as boxes connected by arrows. The **boxes** are called **nodes** and the **arrows** represent **pointers**. Each of the nodes in the following diagram contains an integer, and a pointer that can point to other nodes of the same type. Note that pointers point to the entire node, not to the individual items (such as 10 or "rolls") that are inside the node.



Node



A node is a collection of two sub-elements or parts. A data part that stores the element and a next part that stores the link to the next node.

A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.

Declaring a Linked list:

In C language, a linked list can be implemented using structure and pointers.

```
struct LinkedList
{
   int data;
   struct LinkedList  *next;
 };
```

OR

```
struct node
{
  int data;
  struct node *next;
};
```

The above definition is used to create every node in the list. The **data** field stores the element and the **next** is a pointer to store the address of the next node.

Here in place of a data type, **struct LinkedList** / struct node is written before next. That's because it's a **self-referencing pointer**. It means a pointer that points to whatever it is a part of. Here **next** is a part of a node and it will point to the next node.

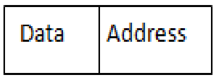
Types of Linked List:

Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.

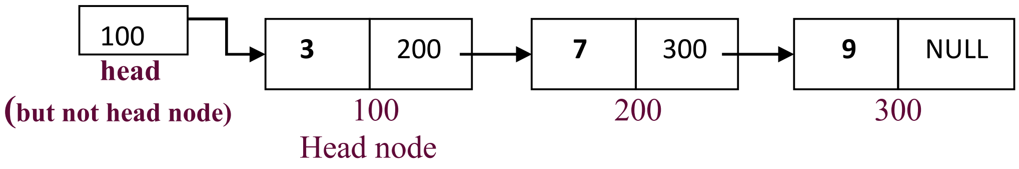
There are different types of Linked List. They are as follows:

1) Singly linked list
2) Doubly linked list
3) Circular linked list
4) Doubly Circular linked list.

1) **Singly linked list:** Singly linked list is the most common used linked list. When we say Linked list it means we are studying singly linked list. In singly linked list each node consist of two parts, one is Data part and another one is Address part. Data part contains value, and address part contains address of next node. Address part also called pointer part.



Node



Here in singly linked list head will point the head node address. Head will not a head node, because head will not hold any data. First node which contains data part and address part will call it as head node.

Suppose first node contains value: 3, second node contains value:7, like third node value 9. Then first node address part / next contains the address of the second node i.e., 200. Second node next hold the address of the next node i.e., 300. As there is no 4th node here, third node next (last node next) will hold NULL. At the beginning **head** will be there without data part. This head hold only address of the first node / head node.

Here in this list we have only single link between each node, so that we are calling it as singly linked list. Here in singly linked list forward traversal only possible, backward direction or reverse traversal is not possible.

We can represent this singly linked list node with the following code.

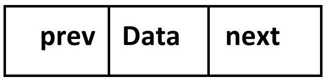
```
struct  node
{
   int   data;
   struct node *next;
};
```

2) Doubly linked list:

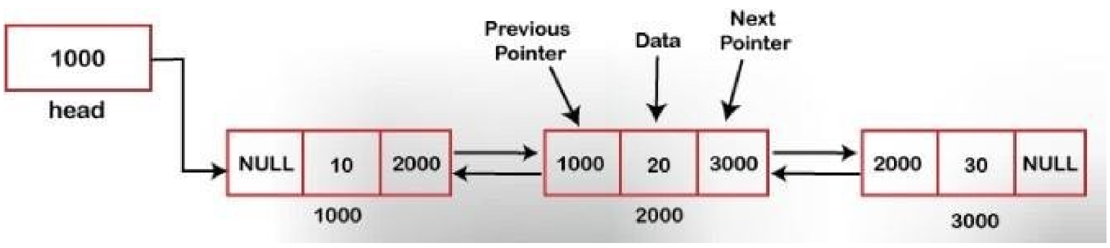
A doubly linked list is another type of the linked list . As name suggest doubly linked list means, a list which contains double links. Here a node contains three parts **Data** part, **Next** part and **Previous** part.

Therefore, we can say that list has two references, i.e., forward and backward reference to traverse in either direction.

- Data part holds the value,
- Next part holds the address of next node using pointer.
- Previous part holds the address of previous node.



Node



Even in double linked list the **head** will store the address of first node. As per above diagram head pointer stored the address of first node i.e., 1000.

Here in doubly linked list forward and backward traversals both possible with the help of next and previous address parts.

We can represent this doubly linked list node with the following code.

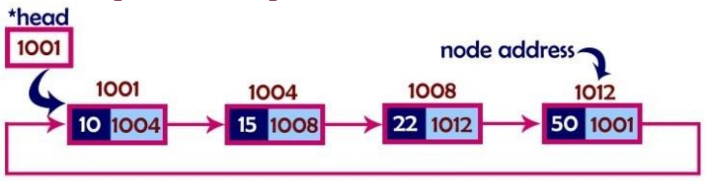
```
struct  node
{
   int   data;
   struct node *next;
  struct node *prev;
};
```

3) Circular linked list:

A circular linked list is another type of the linked list . In this circular linked list it contains single links. Here a node contains TWO parts **Data** part, **Next** part. It is almost similar to doubly linked list, but last node next part will point or hold the address of first node. First node previous part holds the address of last node. So that it will look like circular form. This type will be called doubly circular linked list.

Therefore, we can say that list has only one reference, i.e., forward reference to traverse in forward direction only.

- ➢ Data part holds the value,
- ➢ Next part holds the address of next node using pointer.
- ➢ Last node address part or next part hold the address of first node.



Circular Linked List

Here first node data part will hold the value 10, first node next part hold the address of next node like 1004 stored. Like this it will continue up to last node. When come to last node, next part will point or store the address of first node. So that a circular like reference / link will create. This is called circular linked list. Here head will point the first node address like single linked list. Only the difference with single list is: in single linked list last node next part will hold NULL because no next node.

We can represent this singly linked list node with the following code.

```
struct  node
{
   int   data;
   struct node *next;
};
```

Even while representing the code for circular linked list, there is no mojor difference with singly linked list. Only here in circular linked list the last node next part we need to store the first node address.

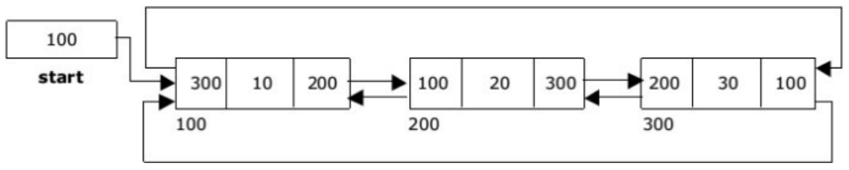
lastNode->next = firstnode;

4) Doubly Circular linked list:

A doubly circular linked list is another type of the linked list . In this doubly circular linked list it contains double links. Here a node contains THREE parts **Data** part, **Next** part and **Previous** part. It is almost similar to double linked list, but here last node next part will point or hold the address of first node. And first node previous part will hold the address of the last node. So that it will look like circular form. This type will be called circular linked list.

Here in doubly circular linked list, it contains TWO references, i.e., forward reference to traverse in forward direction and backward reference to traverse in backward direction. So here bidirectional traversing is possible.

- Data part holds the value,
- Next part holds the address of next node using pointer.
- Previous part holds the address of previous node using pointer.
- Last node address part or next part hold the address of first node.



Doubly Circular Linked List

Here first node data part will hold the value 10, first node next part hold the address of next node like 200 stored. Like this it will continue up to last node. When come to last node, next part will point or store the address of first node. Similarly first node previous part will store the address of last node. So that double circular will be creating. This is called doubly circular linked list.

We can represent this doubly circular linked list node with the following code. It is similar to doubly linked list node structure.

```
struct  node
{
    int   data;
    struct node *next;
    struct node *prev;
};
```

Even while representing the code for doubly circular linked list, there is no major difference with doubly linked list. Only here in doubly circular linked list the last node next part will store the first node address and first node previous address part will store the address of last node.

last->next = firstnode;
first->prev=lastnode;

Differences between the singly-linked list and doubly linked list:

- **Definition**

The **singly-linked** is a linear data structure that consists of a collection of nodes in which one node consists of two parts, i.e., one is the data part, and another one is the address part. In contrast, a **doubly-linked** list is also a linear data structure in which the node consists of three parts, i.e., one is the data part, and the other two are the address parts.

- **Direction**

As we know that in a **singly linked list**, a node contains the address of the next node, so the elements can be traversed in only one direction, i.e., forward direction. In contrast, in a **doubly-linked list**, the node contains two pointers (previous pointer and next pointer) that hold the ***address of the next node*** and the ***address of the previous node, respectively*** so elements can be traversed in both directions.

- **Memory space**

The **singly linked list** occupies less memory space as it contains a single address. We know that the pointer variable stores the address, and the pointer variable occupies 4 bytes; therefore, the memory space occupied by the pointer variable in the singly linked list is also 4 bytes. The **doubly linked** list holds two addresses in a node, one is of the next node and the other one is of the previous node; therefore, the space occupied by the two pointer variables is 8 bytes.

- **Insertion and Deletion**

The insertion and deletion in a singly-linked list are less complex than a doubly linked list. If we insert an element in a singly linked list then we need to update the address of only next node. On the other hand, in the doubly linked list, we need to update the address of both the next and the previous node.

Let's look at the differences in a tabular form.

Basis of comparison	Singly linked list	Doubly linked list
Definition	A single linked list is a list of nodes in which node has two parts, the first part is the data part, and the next part is the pointer pointing to the next node in the sequence of nodes.	A doubly linked list is also a collection of nodes in which node has three fields, the first field is the pointer containing the address of the previous node, the second is the data field, and the third is the pointer containing the address of the next node.
Access	The singly linked list can be traversed only in the forward direction.	The doubly linked list can be accessed in both directions.
List pointer	It requires only one list pointer variable, i.e., the head pointer pointing to the first node.	It requires two list pointer variables, **head** and **last** . The head pointer points to the first node, and the last pointer points to the last node of the list.
Memory space	It utilizes less memory space.	It utilizes more memory space.
Efficiency	It is less efficient as compared to a doubly-linked list.	It is more efficient.
Implementation	It can be implemented on the stack.	It can be implemented on stack, heap and binary tree.
Complexity	In a singly linked list, the time complexity for inserting and deleting an element from the list is **O(n)** .	In a doubly-linked list, the time complexity for inserting and deleting an element is **O(1)** .

Dynamically Linked Stacks:

Dynamic Stack, is a stack data structure whose the length or capacity (maximum number of elements that can be stored) **increases** or **decreases** in real time based on the operations (like insertion or deletion) performed on it.

Stack is one of the most popular used data structures which have multiple applications in real life. So we must be familiar with its structure and implementation, so that we can use stack in our program with ease.

When we are working on stack with arrays, already we know arrays are fixed size. Arrays will not support dynamic memory allocation. Up to fixed size only we can insert the elements, beyond that overflow will be occurred in arrays. To overcome this drawback i.e., to make stack dynamic we implement stack using linked list and such stack is known as dynamic stack and in this section we are going to learn dynamic stack.

Dynamic linked stack:

1) It will be done at run time.
2) The size of the data structure is not fixed.
3) Size can be modified during the operations performed on it. Means it can be shrink or grown whenever need.
4) More flexible.
5) Here it allocates the memory dynamically.
6) In stacks there is no overflow, because any number of elements we can add or insert into the stack dynamically.
7) But here underflow may be occurred, because if there is no single node in the stack but user trying to access will give underflow.

We can perform the following operations on the stack using linked list:

PUSH/insertion, POP/deletion operations we can perform. When we are performing push operation the memory will be allocated dynamically with the help of **malloc( )** function.

PUSH / Insertion in singly linked list at beginning:

Pushing / Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links. There are the following steps which need to be followed in order to inser a new node in the list at beginning.

- Allocate the space for the new node and store data into the data part of the node. This will be done by the following statements.

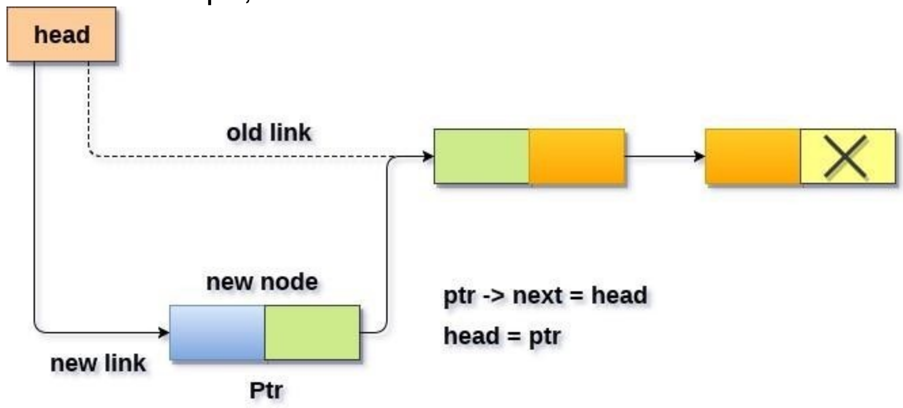
```
ptr = (struct node *) malloc(sizeof(struct node *));
ptr → data = item
```

- Make the link part of the new node pointing to the existing first node of the list. This will be done by using the following statement.

```
ptr->next = head;
```

- At the last, we need to make the new node as the first node of the list this will be done by using the following statement.

```
head = ptr;
```



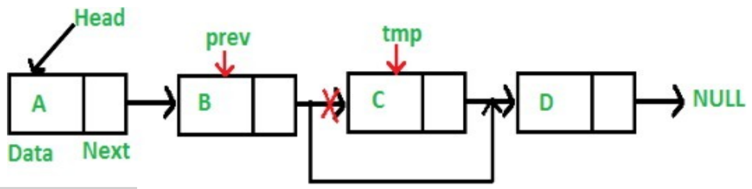
POP operation /Deleting a node from the stack:

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)

After performing pop operation the stack will be shrink dynamically.



Dynamically Linked Queues:

Queue is a linear data structure which follows the **First in, First out principle** (FIFO). Queue supports operations like **enqueue** and **dequeue**. It can be implemented using array and linked list. The benefit of implementing queue using linked list over arrays is that it allows to grow the queue as per the requirements, i.e., **memory can be allocated dynamically**.

A **good example** of a queue is a queue of customers purchasing a train ticket, where the customer who comes first will be served first.

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

The Queue implemented using linked list can organize as many data values we want. At run time we can add any number of elements into the queue at rear. So that queue may be grown. Even when we are deleting the elements or nodes from the queue then automatically queue may be shrink dynamically.

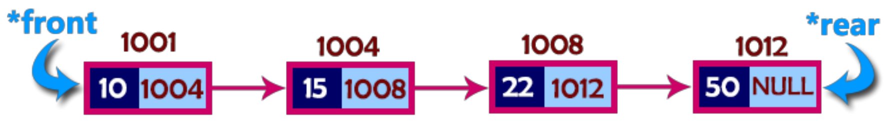
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

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

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

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

Example: A linked queue is shown here:



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

Operations on Linked Queues:

Each node of a linked queue consists of two fields: data and next (storing address of next node). The data field of each node contains the assigned value and the next points to the node containing the next item in the queue.

A linked queue consists of two pointers i.e. front pointer and rear pointer. The front pointer stores the address of the first element of the queue and the rear pointer stores the address of the last element of the queue.

Insertion is performed at the rear end whereas deletion is performed at the front end of the queue. If front and rear both points to NULL, it signifies that the queue is empty.

The two main operations performed on linked queue are:

- Insertion / enqueue
- Deletion dequeue

Insertion (enqueue)

Insert operation or insertion on a linked queue adds an element to the end of queue. The new element which is added becomes the last element of the queue.

Algorithm to perform Insertion on a linked queue:

1. Create a new node pointer.
   ptr = (struct node *) malloc (sizeof(struct node));
2. Now, two conditions arises, i.e, either the queue is empty or queue contains at least one element.
3. If queue is empty, then the new node added will be both front and rear, and the next pointer of front and rear will point to NULL.
```
*ptr->data = val;

if (front == NULL) {
   front = ptr;
   rear = ptr;
   front -> next = NULL;
   rear -> next = NULL;
}
```
4. If queue contains at least one element, then the condition front == NULL becomes false. So, make the next pointer of rear point to new node ptr and point rear pointer to the newly created node ptr
```
rear -> next = ptr;
rear = ptr;
```

Hence, a new node(element) is added to the queue.

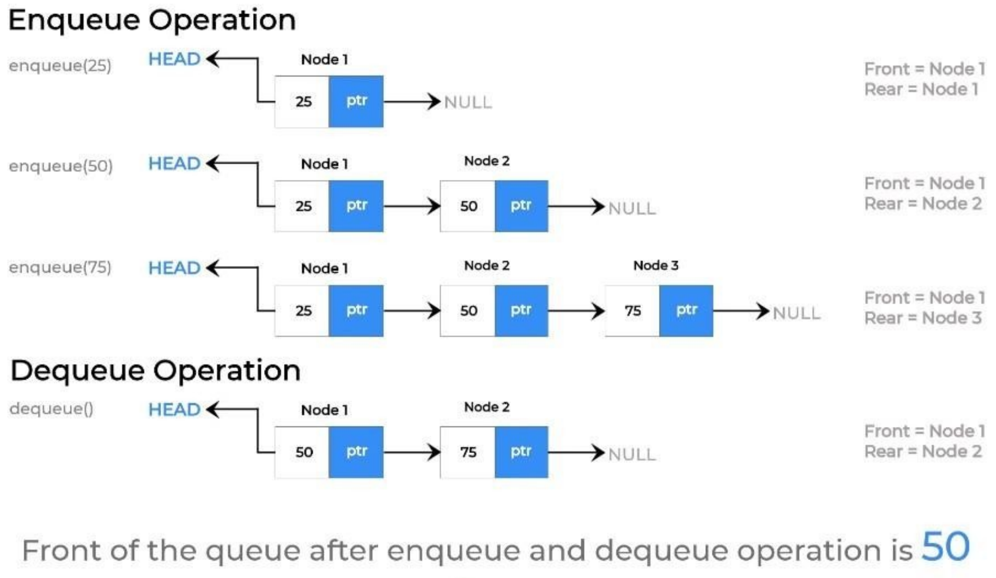
Deletion: (dequeue)

Deletion or delete operation on a linked queue removes the element which was first inserted in the queue, i.e., always the first element of the queue is removed.

Steps to perform Deletion on a linked queue:

1. Check if the queue is empty or not.
2. If the queue is empty, i.e, front==NULL, so we just print 'underflow' on the screen and exit.
3. If the queue is not empty, delete the element at which the front pointer is pointing. For deleting a node, copy the node which is pointed by the front pointer into the pointer ptr and make the front pointer point to the front's next node and free the node pointed by the node ptr. This can be done using the following statement:

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



Polynomials Using Singly Linked Lists

Polynomials and Sparse Matrix are two important applications of arrays and linked lists. A polynomial is composed of different terms where each of them holds a **coefficient** and an **exponent**.
Here we will learn the polynomials using linked list.

Polynomial means:

A polynomial p(x) is the expression in variable x which is in the form $(ax^n + bx^{n-1} + ... + jx + k)$, where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

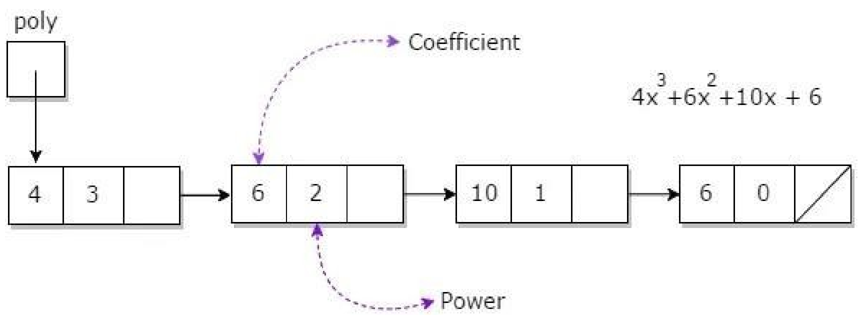
- one is the **coefficient**
- other is the **exponent**

Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



<u>Representation of Polynomial:</u>

<u>Polynomial can be represented in the various ways. These are:</u>

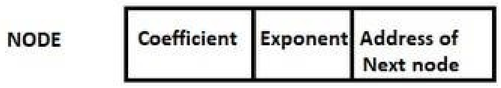
- By the use of arrays
- By the use of Linked List

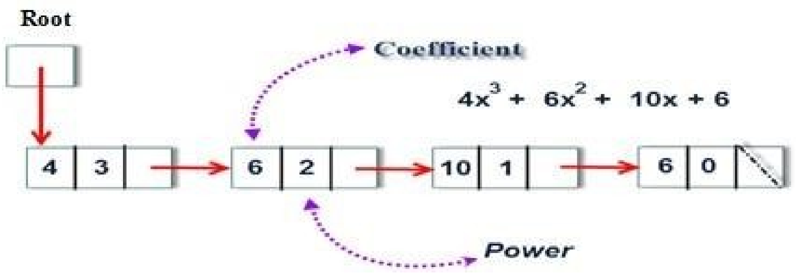
Polynomial representation using Linked List:

The <u>linked list can be used to represent a polynomial of any degree</u>. Simply the information field is changed according to the number of variables used in the polynomial. <u>If a single variable is used in the polynomial the information field of the node contains **two parts**: one for **coefficient**</u> of variable and the <u>other for power / **exponent** of variable</u>. Let us consider an example to represent a polynomial using linked list as follows:

Polynomial: $4x^3-6x^2+10x+6$

Linked List:





In the above linked list, the e<u>xternal pointer 'ROOT' point to the first node of the linked list.</u> The first node of the linked list contains the information about the variable with the highest degree/power. The first node points to the next node with next lowest degree /power of the variable.

<u>Representation of a polynomial using the linked list is beneficial when the operations on the polynomial like **addition** and **subtractions** are performed</u>. The resulting polynomial can also be traversed very easily to display the polynomial.

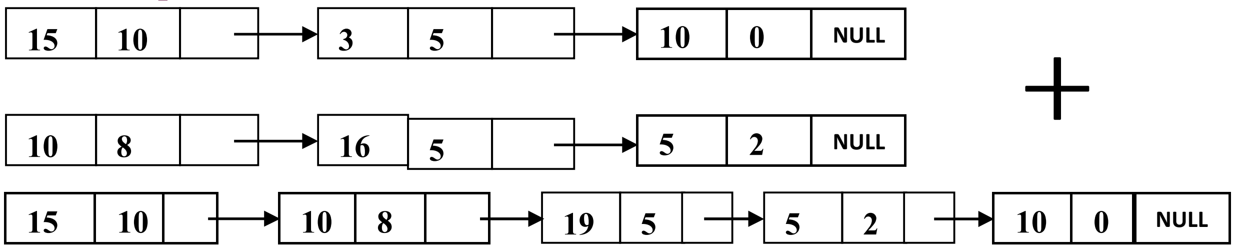
<u>Algorithm for Polynomial Addition using Linkedlist :</u>

- <u>Step 1</u>: loop around all values of linked list and follow step 2 & 3.
- <u>Step 2</u>: if the value of a node's exponent is greater copy this node to result node and head towards the next node.
- <u>Step 3</u>: if the values of both node's exponent is same add the coefficients and then copy the added value with node to the result.
- <u>Step 4</u>: Print the resultant node.

Another Example to represent polynomial as well as adding two polynomials:

$P(x) = 15x^{10}+3x^5+10$ **$P(x)+P(q) = 15x^{10} + 10x^8 + 19x^5 + 5x^2 +10$**

$P(q) = 10x^8+16x^5+5x^2$



<u>Results</u>

Polynomials Using Circularly Linked Lists:

Polynomial means:

A polynomial p(x) is the expression in variable x which is in the form $(ax^n + bx^{n-1} + \dots + jx + k)$, where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the **coefficient**
- other is the **exponent**

Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Given two polynomial numbers represented by a circular linked list, the task is to add these two polynomials by adding the coefficients of the powers of the same variable.

Note: In given polynomials, the term containing the higher power of **x** will come first.

Examples:

Input:
*1st Number = 5x^2 * y^1 + 4x^1 * y^2 + 3x^1 * y^1 + 2x^1*
*2nd Number = 3x^1 * y^2 + 4x^1*
Output:
*5x^2 * y^1 + 7x^1 * y^2 + 3x^1 * y^1 + 6x^1*

Explanation:
*The coefficient of x^2 * y^1 in 1st numbers is 5 and 0 in the 2nd number. Therefore, sum of the coefficient of x^2 * Y^1 is 5.*
*The coefficient of x^1 * y^2 in 1st numbers is 4 and 3 in the 2nd number. Therefore, sum of the coefficient of x^1 * Y^2 is 7.*
*The coefficient of x^1 * y^1 in 1st numbers is 3 and 0 in the 2nd number. Therefore, sum of the coefficient of x^1 * Y^1 is 2.*
*The coefficient of x^1 * Y^0 in 1st numbers is 2 and 4 in the 2nd number. Therefore, sum of the coefficient of x^1 * Y^0 is 6.*
Input:
*1st Number = 3x^3 * y^2 + 2x^2 + 5x^1 * y^1 + 9y^1 + 2*
*2nd Number = 4x^3 * y^3 + 2x^3 * y^2 + 1y^2 + 3*
Output:
*4x^3 * y^3 + 5x^3 * y^2 + 2x^2 + 5x^1 * y^1 + 1y^2 + 9y^1 + 5*

Approach: Follow the below steps to solve the problem:

1. Create two circular linked lists, where each node will consist of the coefficient, power of **x**, power of **y** and pointer to the next node.
2. Traverse both the polynomials and check the following conditions:
   - If power of **x** of 1st polynomial is greater than power of **x** of second polynomial then store node of first polynomial in resultant polynomial and increase counter of polynomial **1**.
   - If power of **x** of 1st polynomial is less than power of **x** of second polynomial then store the node of second polynomial in resultant polynomial and increase counter of polynomial **2**.
   - If power of **x** of 1st polynomial is equal to power of **x** of second polynomial and power of **y** of 1st polynomial is greater than power of **y** of 2nd polynomial then store the node of first polynomial in resultant polynomial and increase counter of polynomial **1**.
   - If power of **x** of 1st polynomial is equal to power of **x** of second polynomial and power of **y** of 1st polynomial is equal to power of **y** of 2nd polynomial then store the sum of coefficient of both polynomial in resultant polynomial and increase counter of both polynomial **1** and polynomial **2**.
3. If there are nodes left to be traversed in 1st polynomial or in 2nd polynomial then append them in resultant polynomial.
4. Finally, print the resultant polynomial.

Circular Linked List and its operations:

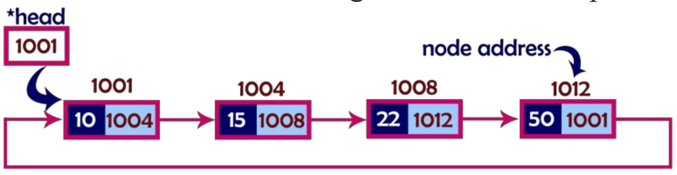
Circular Linked List:

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

Example



Operations on Circular Linked List:

In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined** functions.
- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer '**head**' and set it to **NULL**.
- **Step 5 -** Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

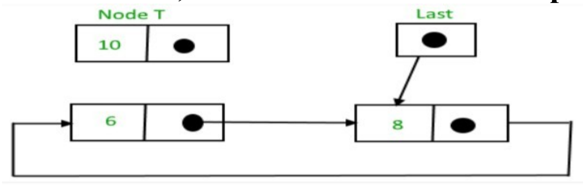
In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

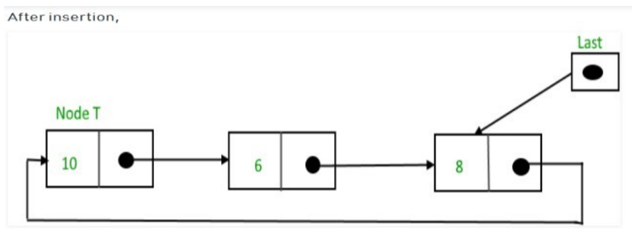
1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** = = **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .
- **Step 4 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
- **Step 5 -** Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** = = **head**').
- **Step 6 -** Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.



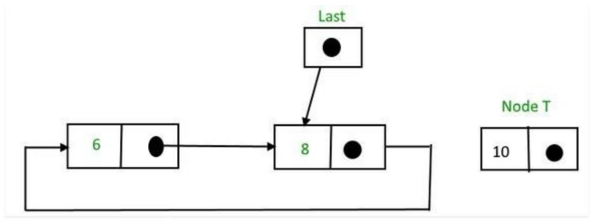


Inserting At End of the list

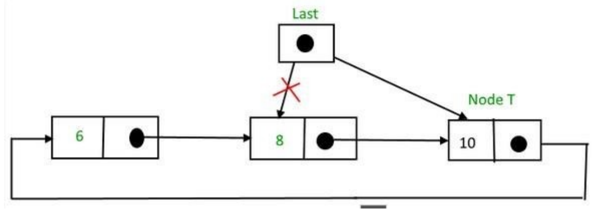
We can use the following steps to insert a new node at end of the circular linked list...

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

- **Step 2 -** Check whether list is **Empty** (**head** = = **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** = = **head**).
- **Step 6 -** Set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.



After insertion,

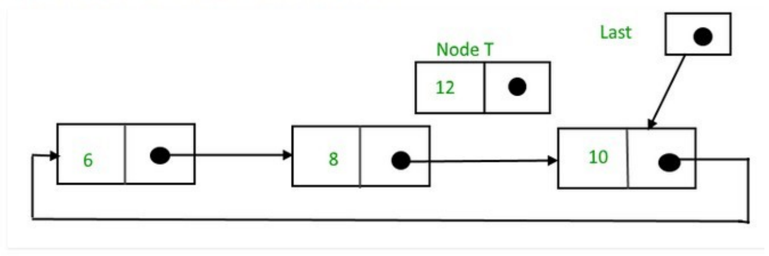


Inserting At Specific location in the list (After a Node)

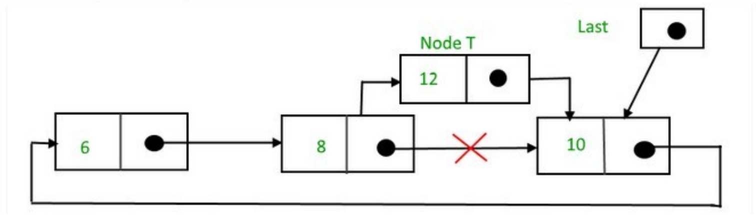
We can use the following steps to insert a new node after a node in the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** = = **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5** - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6** - Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8 -** If **temp** is last node then set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.
- **Step 8 -** If **temp** is not last node then set **newNode** → **next** = **temp** → **next** and **temp** → **next** = **newNode**.

Suppose 12 needs to be inserted after the node has the value 10,



After searching and insertion,



Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

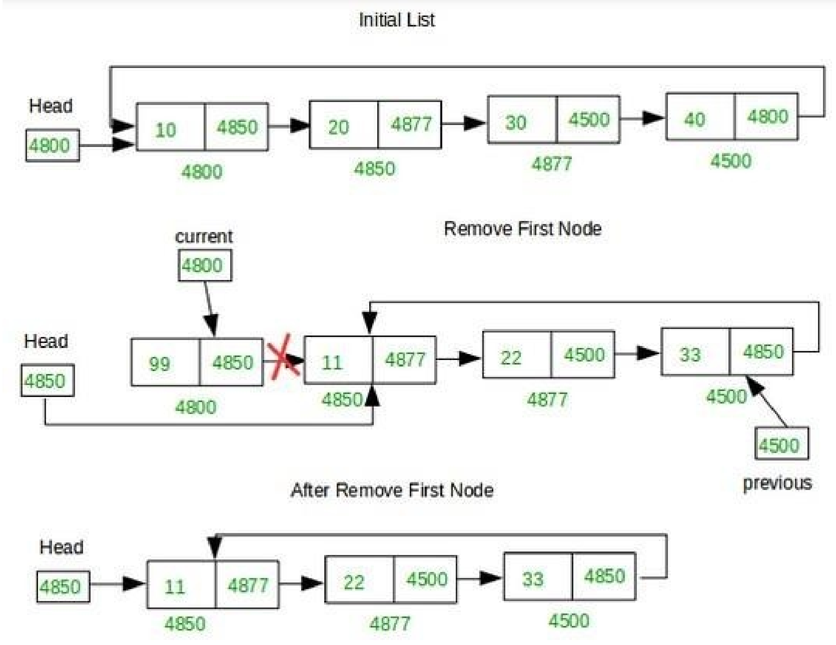
1. Deleting from Beginning of the list
2. Deleting from End of the list

3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** = = **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- **Step 4 -** Check whether list is having only one node (**temp1 → next** == **head**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** = = **head**)
- **Step 7 -** Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

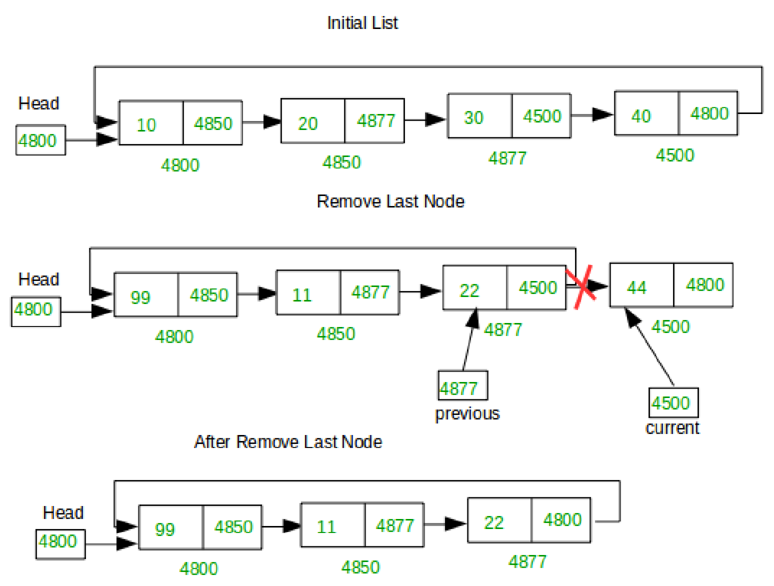


Deleting First Node from Circular Linked List

Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1 → next** == **head**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)
- **Step 7 -** Set **temp2 → next** = **head** and delete **temp1**.

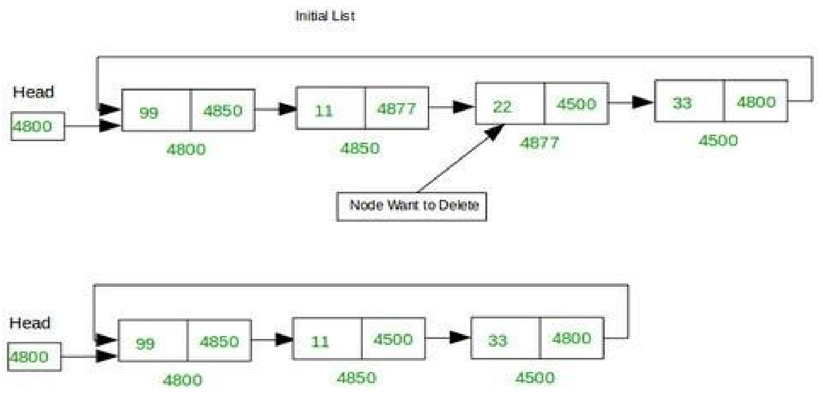


Deleting Last node from Circular Linked List

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)
- **Step 7 -** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → next** = **head** and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 1 1-** If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).



After deleting the node which contains 22

Searching in the Linked List:
To search any value in the linked list, we can traverse the linked list and compares the value present in the node.

```
bool searchLL(Node head, int val)
{
   Node temp = head // creating a temp variable pointing to the head of the linked list
   while( temp != NULL) // traversing the list
   {
      if( temp.data == val )
         return true
      temp = temp.next
   }
   return false
}
```

Displaying a circular Linked List
We can use the following steps to display the elements of a circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep displaying **temp** → **data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5 -** Finally display **temp** → **data** with arrow pointing to **head** → **data**.

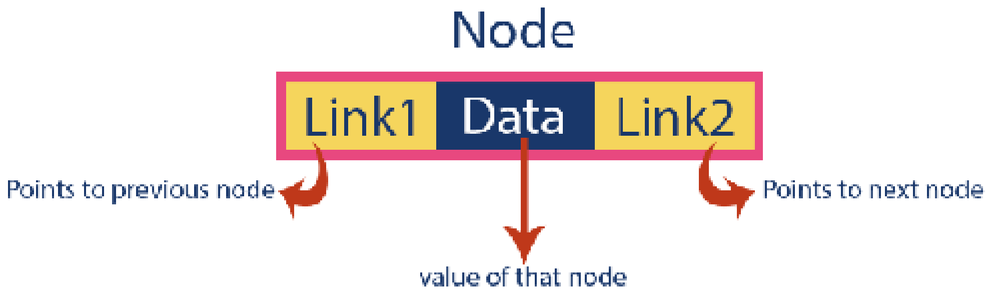
******

Doubly Linked List:

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we cannot traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

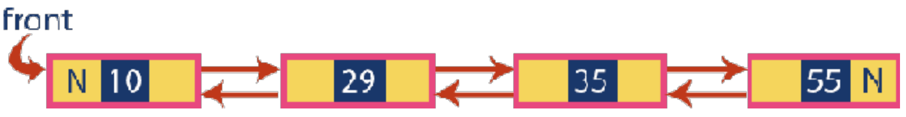
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example



Important Points to be Remembered

1) In double linked list, the first node must be always pointed by head.
2) Always the previous field of the first node must be NULL.
3) Always the next field of the last node must be NULL.

Operations on Doubly Linked List

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Insertion

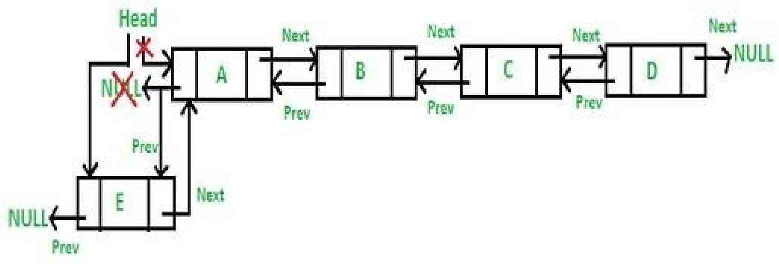
In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

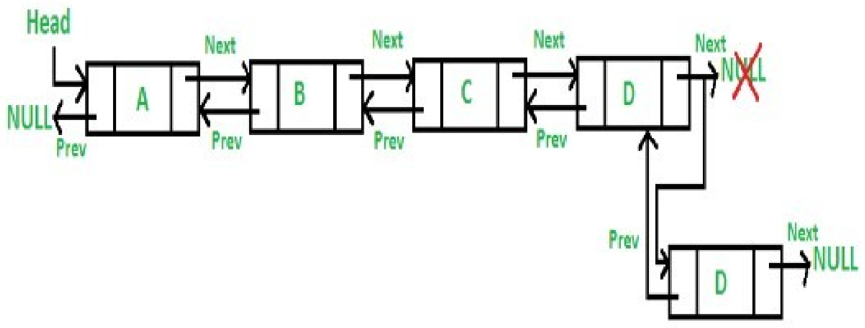
- **Step 1 -** Create a **newNode** with given value and **newNode → previous** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** = = **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.



Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** = = **NULL**)
- **Step 3 -** If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

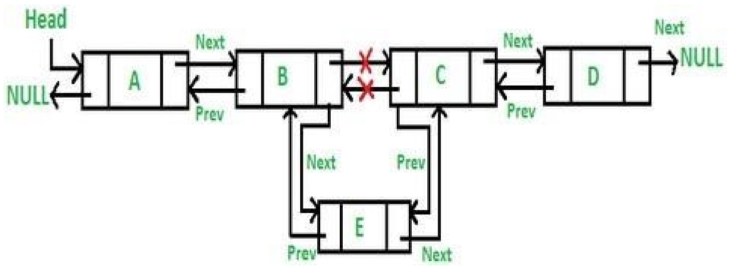


Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** = = **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5 -** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

- **Step 6 -** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7 -** Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.



Deletion

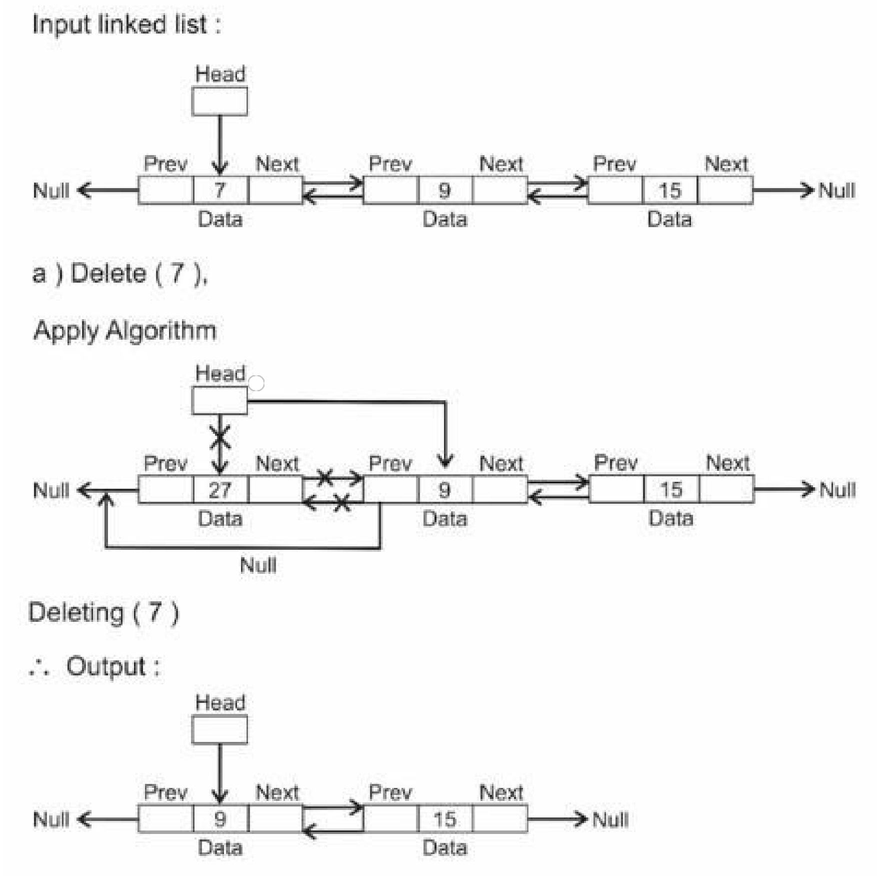
In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** = = **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp → previous** is equal to **temp → next**)
- **Step 5 -** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.



Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** = = **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)
- **Step 5 -** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)
- **Step 7 -** Assign **NULL** to **temp → previous → next** and delete **temp**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.
- **Step 5 -** If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the fuction.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- **Step 9 -** If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.
- **Step 10 -** If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).
- **Step 11 -** If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).
- **Step 12 -** If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Display **'NULL <--- '**.
- **Step 5 -** Keep displaying **temp → data** with an arrow (**<===>**) until **temp** reaches to the last node
- **Step 6 -** Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

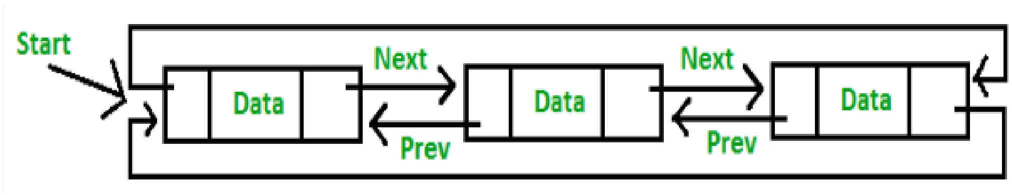
******

Doubly Circular Linked List:

Circular Doubly Linked List has **properties of both doubly linked list and circular linked list** in which two consecutive elements are linked or connected by previous and next pointer and the last node points to first node by next pointer and also the first node points to last node by the previous pointer.

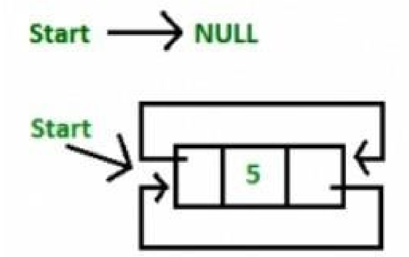
Following is representation of a Circular doubly linked list node in C/C++:

```
// Structure of the node
struct node
{
   int data;
   struct node *next; // Pointer to next node
   struct node *prev; // Pointer to previous node
};
```

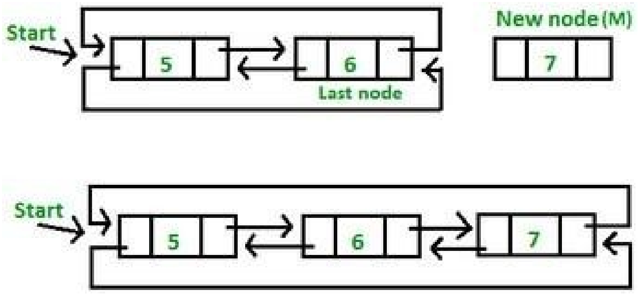


Insertion in Circular Doubly Linked List:

- **Insertion at the end of list or in an empty list**
  - **Empty List (start = NULL):** A node(Say N) is inserted with data = 5, so previous pointer of N points to N and next pointer of N also points to N. But now start pointer points to the first node the list.

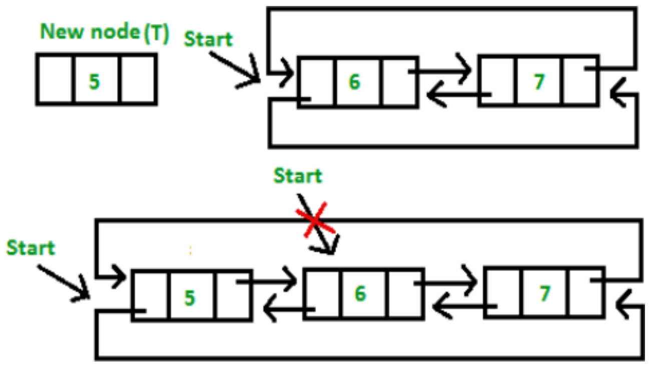


- **List initially contains some nodes, start points to first node of the List:** A node(newNode M) is inserted with data = 7, so previous pointer of M points to last node, next pointer of M points to first node and last node's next pointer points to this M node and first node's previous pointer points to this M node.



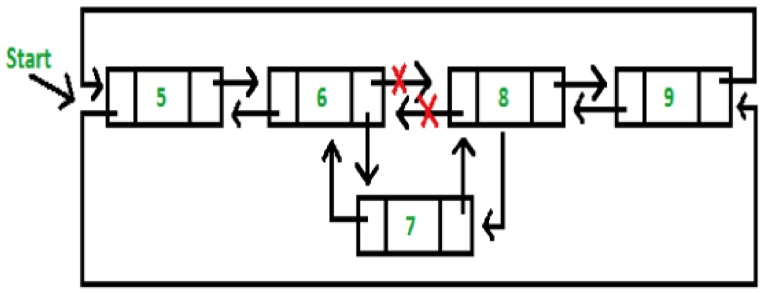
Insertion at the beginning of the list:

To insert a node at the beginning of the list, create a node(newNode T) with data = 5, T next pointer points to first node of the list, T previous pointer points to last node the list, last node's next pointer points to this T node, first node's previous pointer also points this T node and at last don't forget to shift 'Start' pointer to this T node.



Insertion in between the nodes of the list:

To insert a node in between the list, two data values are required one after which new node will be inserted and another is the data of the new node.



****

Following are the advantages and disadvantages of a circular doubly linked list:

Advantages:

- List can be traversed from both directions i.e. from head to tail or from tail to head.
- Jumping from head to tail or from tail to head is done in constant time O(1).
- Circular Doubly Linked Lists are used for the implementation of advanced data structures like Fibonacci Heap.

Disadvantages

- It takes slightly extra memory in each node to accommodate the previous pointer.
- Lots of pointers involved while implementing or doing operations on a list. So, pointers should be handled carefully otherwise data of the list may get lost.

Applications of Circular doubly linked list

- Managing songs playlist in media player applications.
- Managing shopping cart in online shopping.

End of the Unit-3

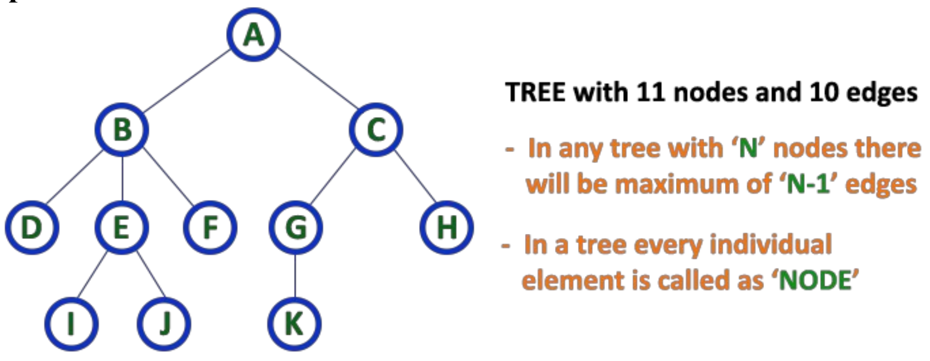
UNIT-4

Trees

A tree is a data structure consisting of nodes organized as a hierarchy. **Tree** is a widely used data structure that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

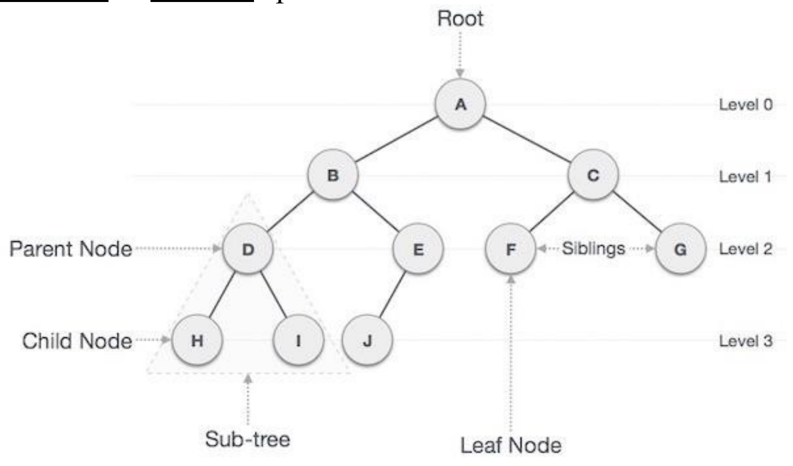
In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

Example Tree


A
B
C
D
E
F
G
H
I
J
K
TREE with 11 nodes and 10 edges
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Binary Tree : Binary Tree is a special used for data storage purposes. Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as left child and the other is known as right child.

A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Edge** : a connector between one node to another
- **Depth**: The depth of a node is the number of edges from the node to the tree's root node.
- **Height** of node – The height of a node is the number of edges on the longest downward path between that node and a leaf.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Tree properties :

1) A tree with n nodes has exactly (n-1) edges or branches.

2) In a tree every node except the root has exactly one parent (and the root node does not have a parent)

3) There is exactly one path connecting any two nodes in a tree.

4) The maximum number of nodes in a binary tree of height K is $2^{K+1}-1$ where K>=0.

(or you can also write the below steps)

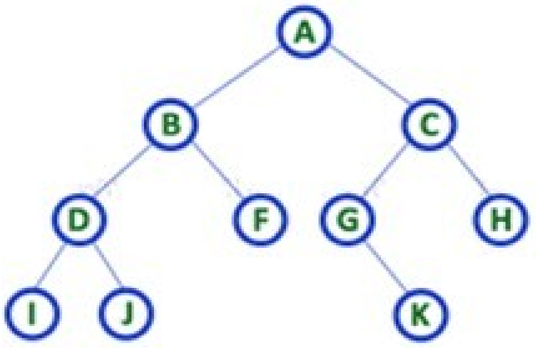
1. The maximum number of nodes at level 'l' of a binary tree is 2^{l-1}.
   Here level is number of nodes on path from root to the node (including root and node). Level of root is 1. This can be proved by induction.
   For root, $l = 1$, number of nodes = $2^{1-1} = 1$
   Assume that maximum number of nodes on level l is 2^{l-1}
   Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^{l-1}$
2. Maximum number of nodes in a binary tree of height 'h' is $2^h – 1$.
   Here height of a tree is maximum number of nodes on root to leaf path. Height of a leaf node is considered as 1.
   This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + .. + 2^{h-1}$ This is a simple geometric series with h terms and sum of this series is $2^h – 1$.
   In some books, height of a leaf is considered as 0. In this convention, the above formula becomes $2^{h+1} – 1$
3. In a Binary Tree with **N** nodes, minimum possible height or minimum number of levels is $[\ \mathrm{Log}_2(N+1)\ ]$
   This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as **0**, then above formula for minimum possible height becomes $[\ \mathrm{Log}_2(N+1)\ ] – 1$
4. A Binary Tree with **L** leaves has at least $[\ \mathrm{Log}_2 L\ ] + 1$ levels
   A Binary tree has maximum number of leaves when all levels are fully filled.
5. In Binary tree, number of leaf nodes is always one more than nodes with two children.

Binary Tree Representations

A binary tree data structure is represented by using two methods. Those methods are as follows...

1. Array Representation
2. Linked List Representation

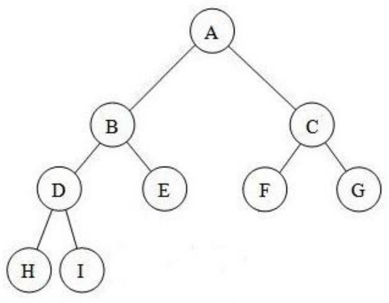
Consider the following binary tree...



1. **Array Representation / Sequential Representation:**

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. Consider the above example of binary tree and it is represented as follows...

To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of $\mathbf{2^{n+1} - 1}$.



Case-I:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
Index ⟶	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

If you representing binary tree by using arrays that too from index 0 (zero) then

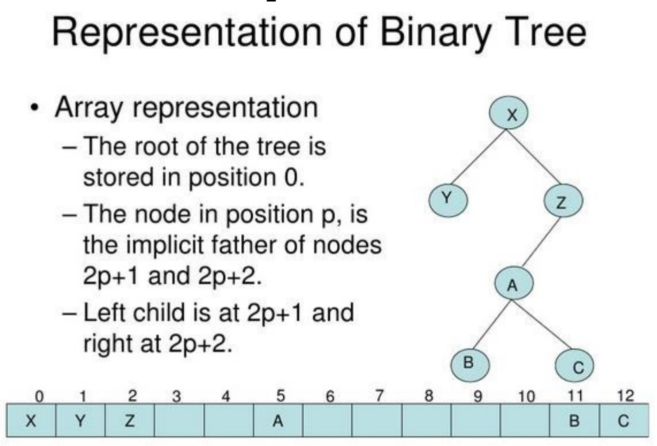
5) If a node is at i th index:-
   - ➢ Left child would be at:- [(2*i)+1]
   - ➢ Right child would be at:- [(2*i)*2]
   - ➢ Parent would be at:- $\lfloor\frac{(i-1)}{2}\rfloor$

Case-II:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
Index ⟶	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

If you representing binary tree by using arrays that too from index 1 (one) then

1) If a node is at i th index:-
   - ➢ Left child would be at:- (2*i)
   - ➢ Right child would be at:- [(2*i)*1]
   - ➢ Parent would be at:- $\lfloor\frac{i}{2}\rfloor$

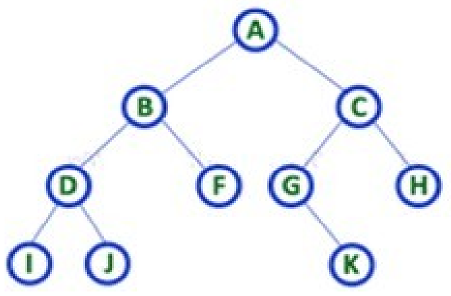


Representation of Binary Tree

- Array representation
  - The root of the tree is stored in position 0.
  - The node in position p, is the implicit father of nodes 2p+1 and 2p+2.
  - Left child is at 2p+1 and right at 2p+2.

0	1	2	3	4	5	6	7	8	9	10	11	12
X	Y	Z			A						B	C

Another Example:



Array Representation

A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

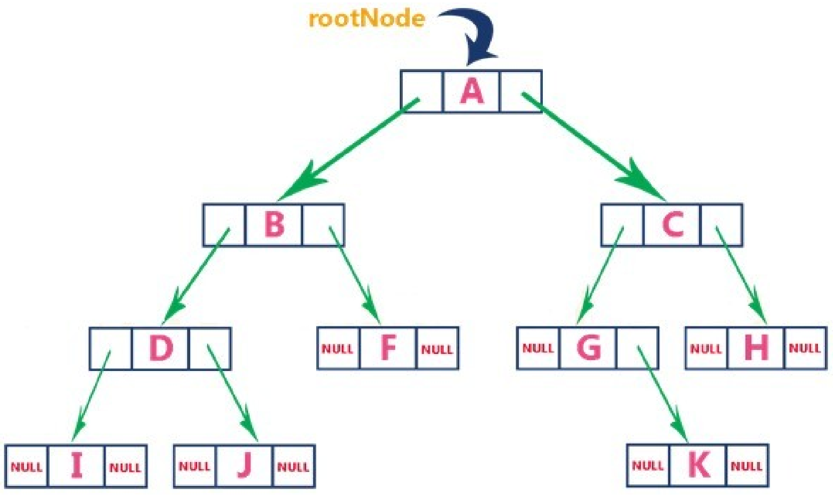
2. Linked List Representation

We use **double linked list** to represent a binary tree. In a double linked list, every node consists of three fields. **First** field for storing left child address, **second** for storing actual data and **third** for storing right child address.

In this linked list representation, a node has the following structure...

Left Child Address	Data	Right Child Address

The above example of binary tree represented using Linked list representation is shown as follows...



Binary trees in linked representation are stored in the memory as linked lists. These lists have nodes that aren't stored at adjacent or neighboring memory locations and are linked to each other through the parent-child relationship associated with trees.

In this representation, each node has three different parts –

1) pointer that points towards the right node,
2) pointer that points towards the left node,
3) data element.

This is the more common representation. All binary trees consist of a root pointer that point in the direction of the root node. When you see a root node pointing towards null or 0, you should know that you are dealing with an empty binary tree. The right and left pointers store the address of the right and left children of the tree.

Applications of Binary Trees:

- Binary Search Tree - Used in *many* search applications where data is constantly entering/leaving, such as the map and set objects in many languages libraries.
- Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered.
- Binary Tries - Used in almost every high-bandwidth router for storing router-tables.
- Hash Trees - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.
- Heaps - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality-of-Service in routers, and A* *(path-finding algorithm used in AI applications, including robotics and video games).* Also used in heap-sort.
- Huffman Coding Tree (Chip Uni) - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
- GGM Trees - Used in cryptographic applications to generate a tree of pseudo-random numbers.
- Syntax Tree - Constructed by compilers and (implicitly) calculators to parse expressions.
- Treap - Randomized data structure used in wireless networking and memory allocation.
- T-tree - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so.

Binary Tree Traversals:

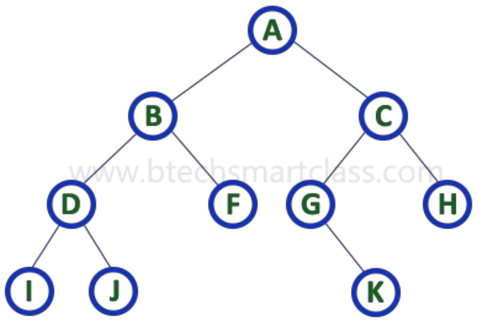
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

Consider the following binary tree...



1. In - Order Traversal (leftChild - root - rightChild) Left-Root-Right

In In-Order traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited **first**, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of a binary tree, first we try to visit left child of root node '**A**', but A's left child '**B**' is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit **'I'** then go for its root node **'D'** and later we visit D's right child **'J'**. With this we have completed the left part of node B. Then visit **'B'** and next B's right child **'F'** is visited. With this we have completed left part of node A. Then visit root node **'A'**. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit **'G'** and then visit G's right child K. With this we have completed the left part of node C. Then visit root node **'C'** and next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (root - leftChild - rightChild) Root, Left, Right

In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node **'A'** then visit its left child **'B'** which is a root for D and F. So we visit B's left child **'D'** and again D is a root for I and J. So we visit D's left child **'I'** which is the leftmost child. So next we go for visiting D's right child **'J'**. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child **'F'**. With this we have completed root and left parts of node A. So we go for A's right child **'C'** which is a root node for G and H. After visiting C, we go for its left child **'G'** which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child **'K'**. With this, we have completed node C's root and left parts. Next visit C's right child **'H'** which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

3. Post - Order Traversal (leftChild - rightChild - root) Left, Right, Root

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Binary Search Tree (BST): or for Binary Tree Operations **(Binary Tree will not follow the greater or lesser formula)**

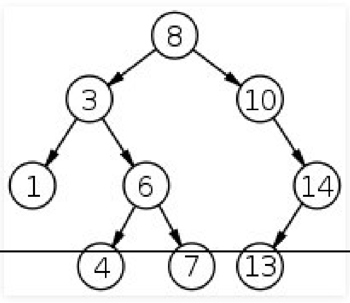
A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties.

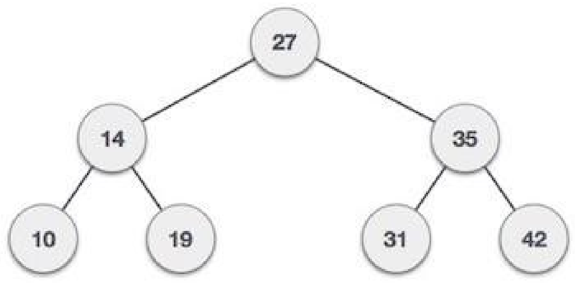
- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.
- The left and right subtree each must also be a binary search tree.
  There must be no duplicate nodes.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree .

Representation : BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following example representations of BST −





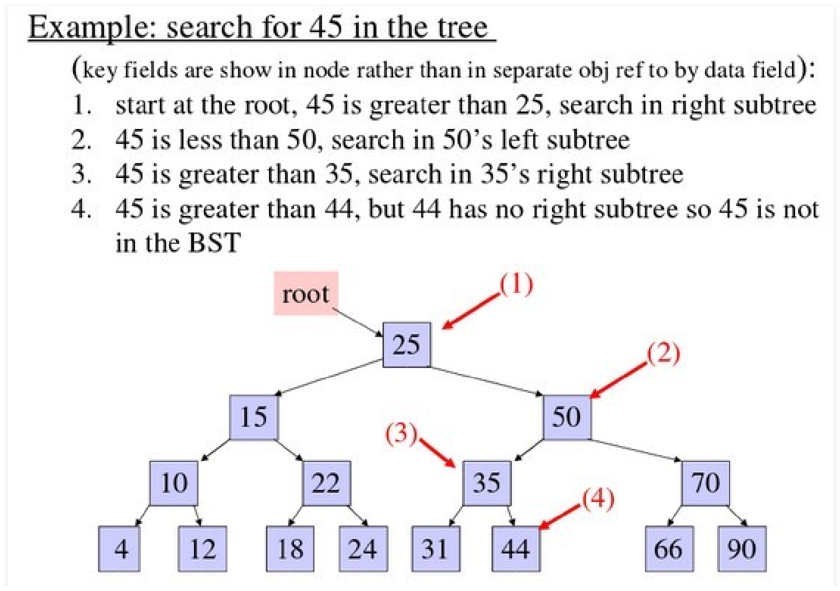
Basic Operations in BST:

Following are the basic operations of a Binary Search tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Deletion**- Delete an element from the tree.

Search Operation : Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

- **item == root.val:** We terminate the search as the **item** is found
- **item > root.val:** We just check the right subtree because all the values in the left subtree are lesser than **root.val**
- **item < root.val:** Now we just check the left subtree as all values in the right subtree are greater than **root.val**


Example: search for 45 in the tree

(key fields are show in node rather than in separate obj ref to by data field):

1. start at the root, 45 is greater than 25, search in right subtree
2. 45 is less than 50, search in 50's left subtree
3. 45 is greater than 35, search in 35's right subtree
4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST

Insert Operation: Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

1. If the root is NULL, create a new node with value **item** and return it.
2. Else, Compare **item** with **root.val**
   - If **root.val < item**, recurse for right subtree
   - If **root.val > item**, recurse for left subtree

Example:

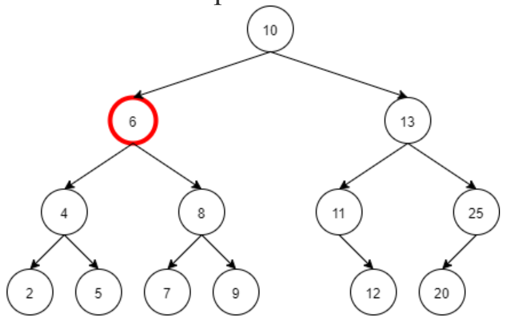
```
         100                              100
        /   \        Insert 40           /   \
      20     500     --------->        20     500
     /  \                             /  \
   10    30                         10    30
                                            \
                                             40
```

Deletion Operation:

When we delete a node in BST, we may encounter three cases →

1. The node to be deleted **is a leaf node**: Easiest case, simply remove the node from the tree

2. The node to be deleted **has only one child**: Replace the node by its child
3. The node to be deleted **has both children**: The node still needs to be replaced to maintain BST properties, but which node should replace this deleted node?



Suppose we have to delete the node with value 6, which node should be selected ideally to replace this node?
→ The inorder successor of this node would be the aptest choice to replace this node as its inorder successor is the smallest element that is greater than this node.
Note: *The inorder predecessor can be used to replace this node too. But conventionally, we use the inorder successor to replace the node.*
Solution Steps
You need to delete the node with value **item** and then return the **root** of the modified tree. First, we need to find the node to be deleted and then replace it by the appropriate node if needed.

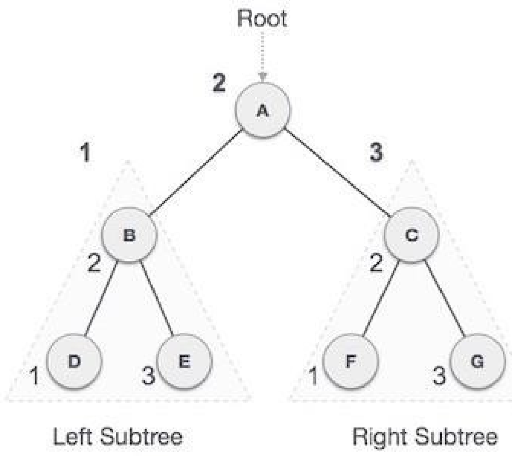
1. Check if the root is NULL, if it is, just return the root itself. It's an empty tree!
2. If **root.val < item**, recurse the right subtree.
3. If **root.val > item**, recurse the left subtree.
4. If both above conditions above false, this means **root.val == item**.
5. Now we first need to check how many child did **root** have.
6. **CASE 1: No Child** → Just delete **root** or deallocate space occupied by it
7. **CASE 2: One Child** →Replace **root** by its child
8. **CASE 3: Two Children**
   - Find the inorder successor of the **root** (Its the smallest element of its right subtree). Let's call it **new_root.**
   - Replace **root** by its inorder successor
   - Now recurse to the right subtree and delete **new_root.**

9. Return the **root**.

<u>**Tree Traversals in Binary Search Tree:**</u>
Tree Traversals:
Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal. There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

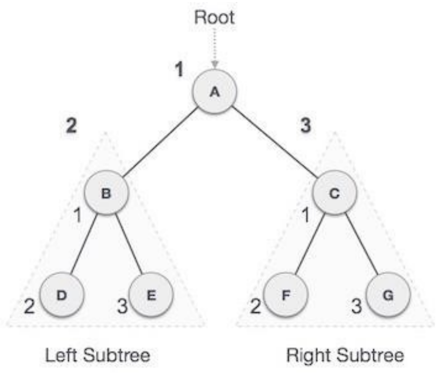
1. In-order Traversal : In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.
If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

D → B → E → A → F → C → G

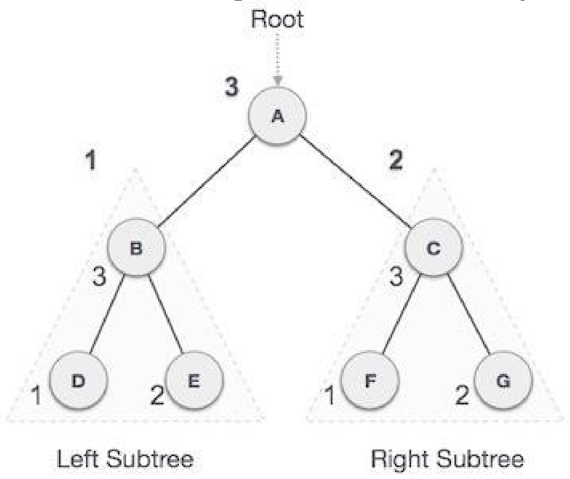
2. Pre-order Traversal: In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

3. Post- Order Traversal : In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from A, and following pre-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

****

Graphs:

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of linkes known as edges (or Arcs) which connets the vertices. A graph is defined as follows...

Graph is a collection of vertices and arcs which connects vertices in the graph.

Graph is a collection of nodes and edges which connects nodes in the graph.

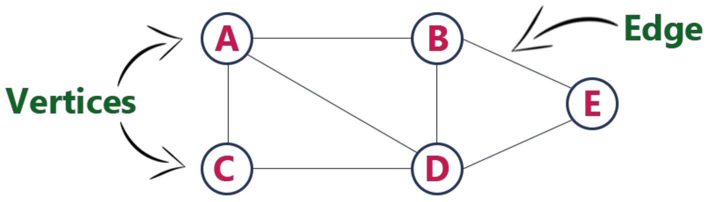
Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example

The following is a graph with **5** vertices and **6** edges.

This graph **G** can be defined as G = (V , E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology:

We use the following terms in graph data structure...

Vertex

A individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge

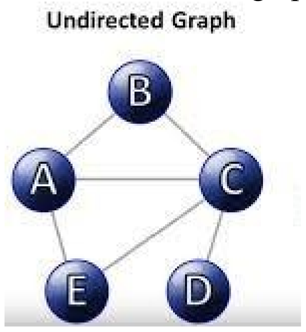
An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are **7** edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed egde is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is an edge with cost on it.

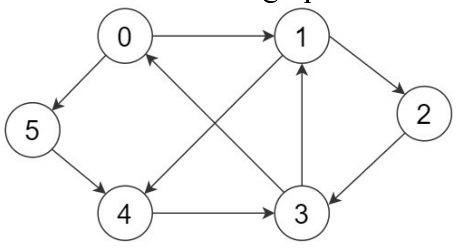
Undirected Graph

A graph with only undirected edges is said to be undirected graph.



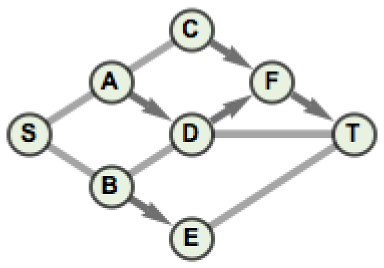
Directed Graph

A graph with only directed edges is said to be directed graph.



Mixed Graph

A graph with undirected and directed edges is said to be mixed graph.



End vertices or Endpoints
The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.
Origin If an edge is directed, its first endpoint is said to be origin of it.
Destination
If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.
Adjacent
If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.
Incident An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.
Outgoing Edge A directed edge is said to be outgoing edge on its orign vertex.
Incoming Edge A directed edge is said to be incoming edge on its destination vertex.
Degree Total number of edges connected to a vertex is said to be degree of that vertex.
Indegree Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
Outdegree Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.
Parallel edges or Multiple edges
If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.
Self-loop An edge (undirected or directed) is a self-loop if its two endpoints coincide.
Simple Graph A graph is said to be simple if there are no parallel and self-loop edges.
Path:
A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations

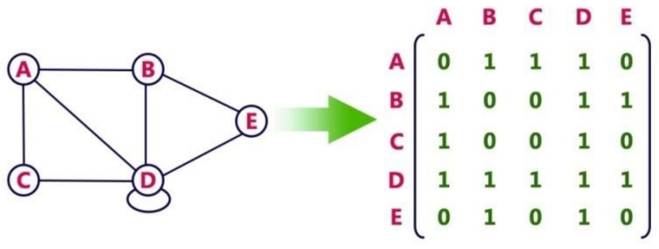
Graph data structure is represented using following representations...

1. Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

1) Adjacency Matrix

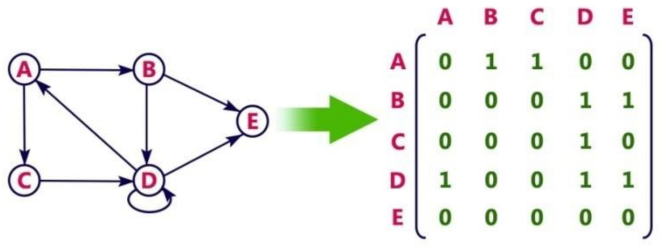
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



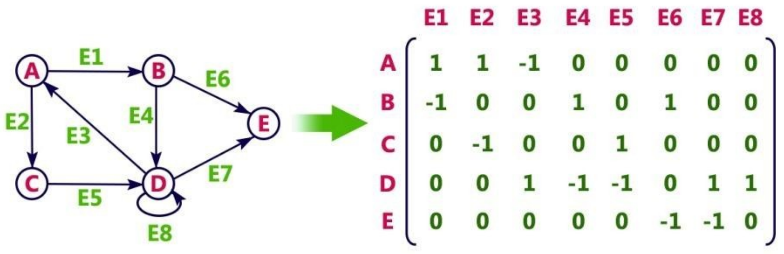
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph representation...



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

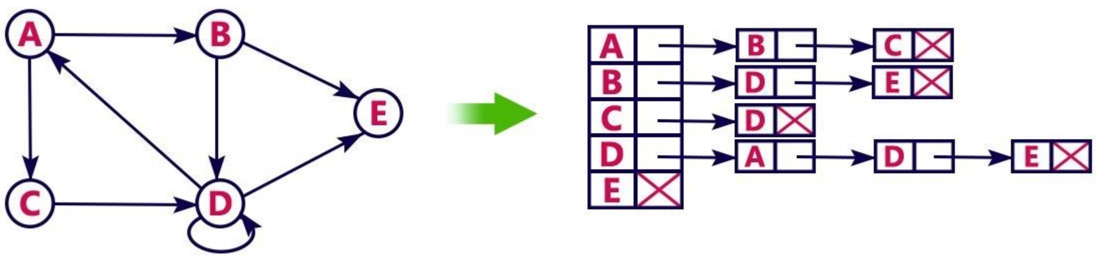
2) Incidence Matrix
In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represent vertices and columns represent edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.
For example, consider the following directed graph representation...



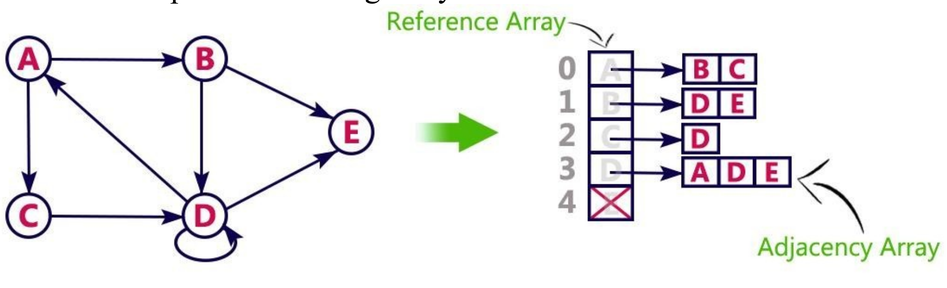
	E1	E2	E3	E4	E5	E6	E7	E8
A	1	1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	1
E	0	0	0	0	0	-1	-1	0

3) Adjacency List
In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



****

ELEMENTARY GRAPH OPERATIONS:
Given a graph G = (V E) and a vertex v in V(G) we wish to visit all vertices in G that are reachable from v (i.e., all vertices that are connected to v). We shall look at **two ways** of doing this: **depth-first search** and **breadth-first search**.
Although these methods work on both directed and undirected graphs the following discussion assumes that the graphs are undirected.

Graph Traversals:

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

- DFS (Depth First Search)
- BFS (Breadth First Search)

DFS (Depth First Search)

DFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use **Stack** data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

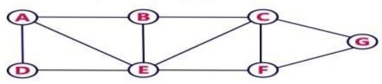
We use the following steps to implement DFS traversal...

Step 1: Define a **Stack of size** total number of vertices in the graph.
Step 2: Select any vertex as **starting point** for traversal. Visit that vertex and **push** it on to the Stack.
Step 3: Visit any one of the **adjacent** vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
Step 5: When there is no new vertex to be visit then use **back tracking**(coming back to the vertex from which we came to current vertex) and **pop** one vertex from the stack.
Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.
Step 7: When stack becomes Empty, then produce final spanning tree *by removing unused edges* from the graph

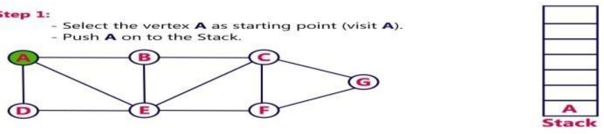
Example

In next page

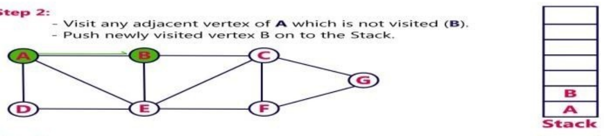
Consider the following example graph to perform DFS traversal



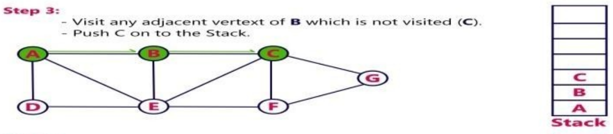
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



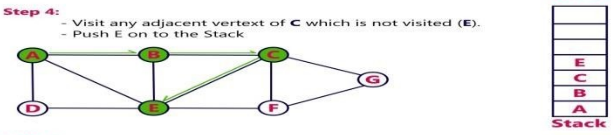
Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



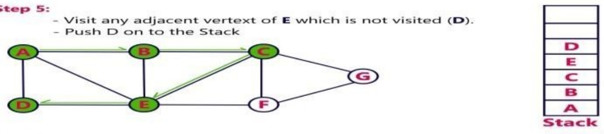
Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



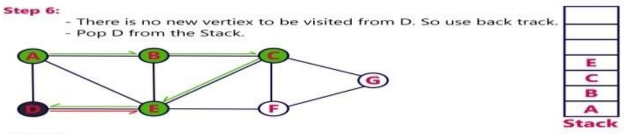
Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



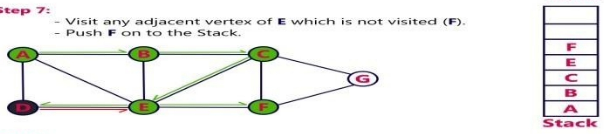
Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



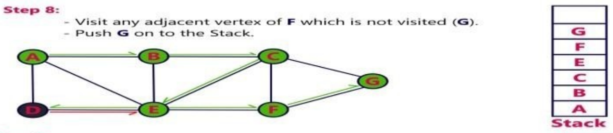
Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



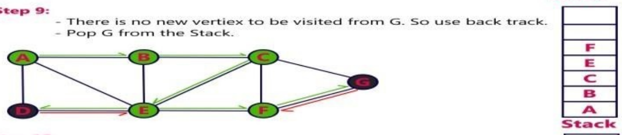
Step 7:
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



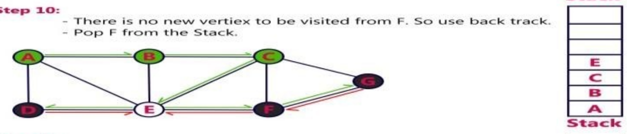
Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



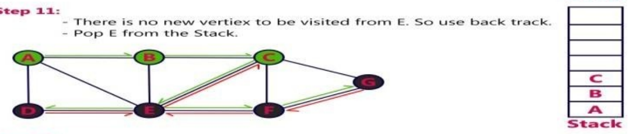
Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



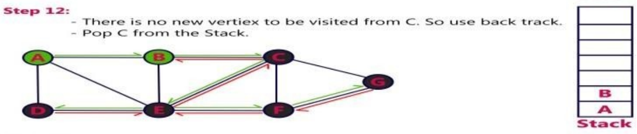
Step 10:
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



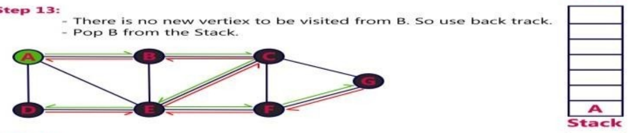
Step 11:
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



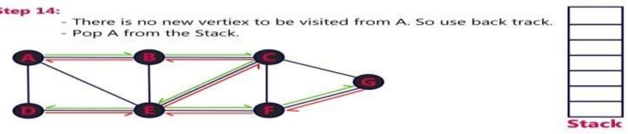
Step 12:
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



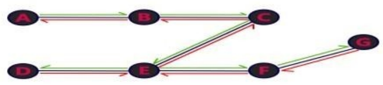
Step 13:
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



Step 14:
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



BFS (Breadth First Search)

BFS traversal of a graph, produces a **spanning tree** as final result. Spanning Tree is a graph without any loops. We use **Queue** data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

Step 1: Define a **Queue of size** total number of vertices in the graph.

Step 2: Select any vertex as **starting point** for traversal. Visit that vertex and **insert** it into the Queue.

Step 3: Visit all the **adjacent** vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

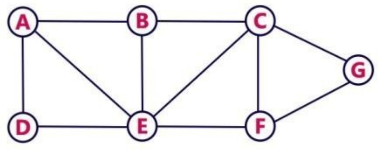
Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph.

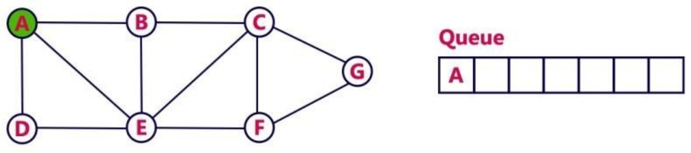
Example:

Consider the following example graph to perform BFS traversal



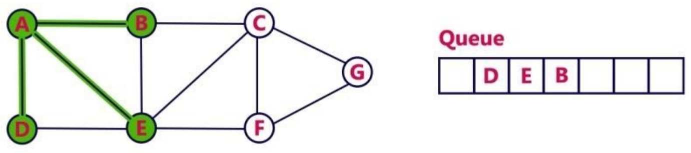
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



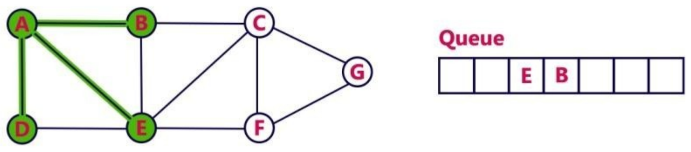
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



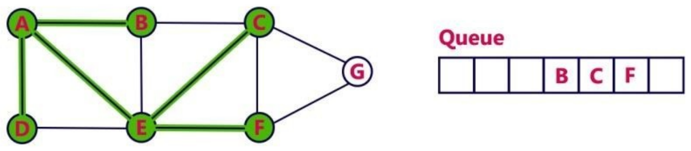
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



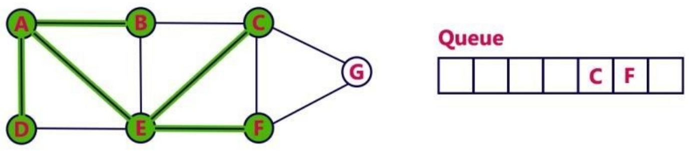
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



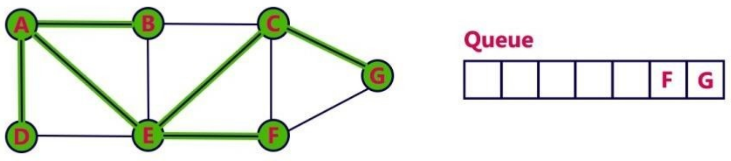
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



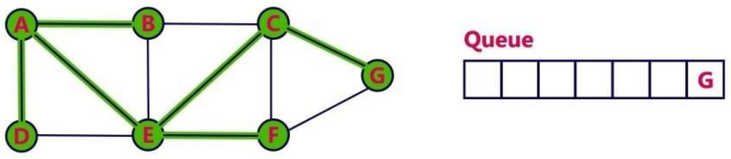
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



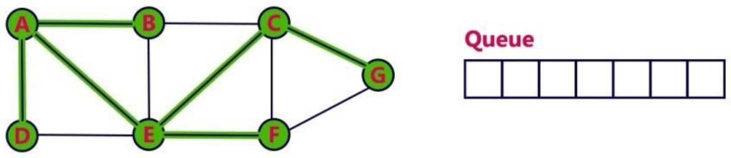
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

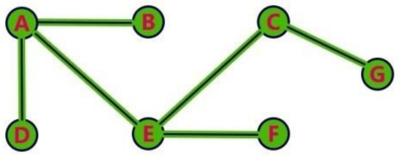


Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



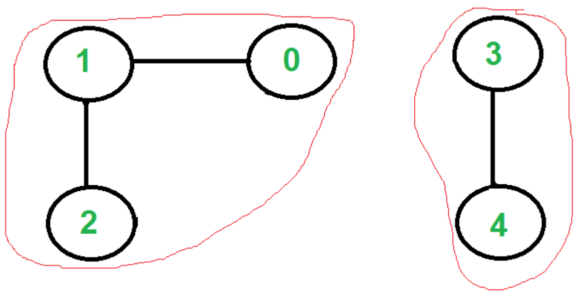
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Connected Components :

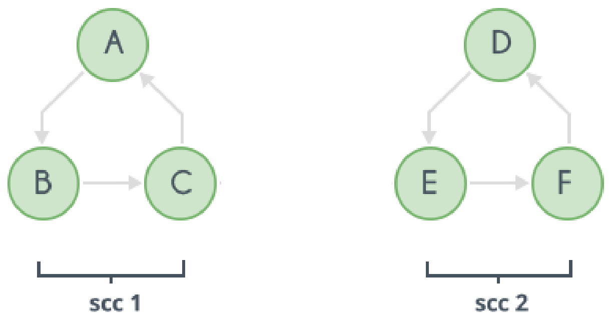
Connectivity in an undirected graph means that every vertex can reach every other vertex via any path. If the graph is not connected the graph can be broken down into Connected Components.

In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph. For **example**, the graph shown below has two connected components. A vertex with no incident edges is itself a connected component. A graph that is itself connected has exactly one connected component, consisting of the whole graph.



There are two connected components in above undirected graph
0 1 2
3 4

Strong Connectivity applies only to directed graphs. A directed graph is **strongly connected** if there is a directed path from any vertex to every other vertex. This is same as connectivity in an undirected graph, the only difference being strong connectivity applies to directed graphs and there should be directed paths instead of just paths. Similar to connected components, a directed graph can be broken down into Strongly Connected Components.



Finding connected components for an undirected graph is an easier task. We simple need to do either BFS or DFS starting from every unvisited vertex, and we get all strongly connected components. Below are steps based on DFS.

```
1) Initialize all vertices as not visited.
2) Do following for every vertex 'v'.
       (a) If 'v' is not visited before, call DFSUtil(v)
       (b) Print new line character
```

```
Method: DFSUtil(v)
    1) Mark 'v' as visited.
    2) Print 'v'
    3) Do following for every adjacent 'u' of 'v'.
        If 'u' is not visited, then recursively call DFSUtil(u)
```

****

What is a spanning tree?

A spanning tree can be defined as the <u>subgraph of an undirected connected graph.</u> It includes a<u>ll the vertices along with the least possible number of edges.</u> 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.

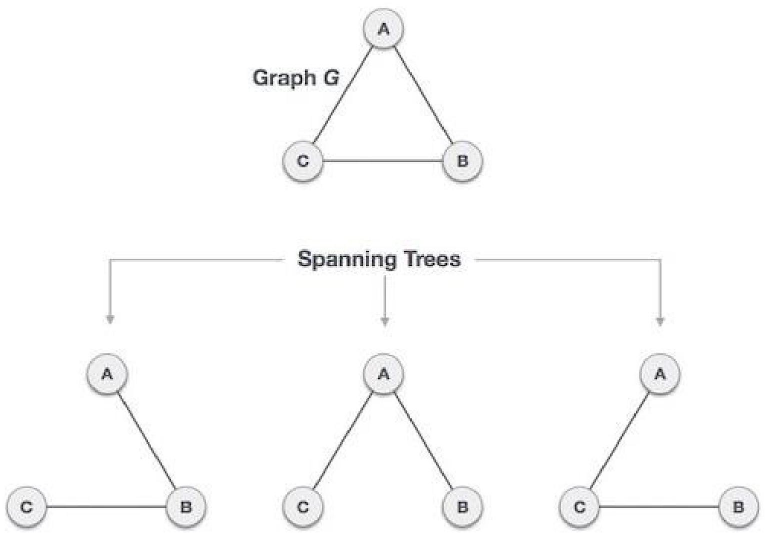
A complete undirected graph can have $\mathbf{n}^{\mathbf{n-2}}$ number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be $\mathbf{5^{5-2} = 125}$.

<u>Applications of the spanning tree:</u>

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**.



Properties of Spanning Tree:

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Minimum Spanning Tree (MST):

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

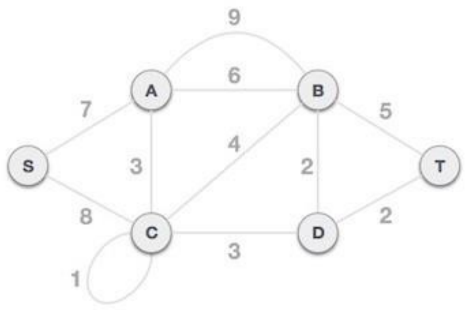
We shall learn about two most important spanning tree algorithms here −

- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

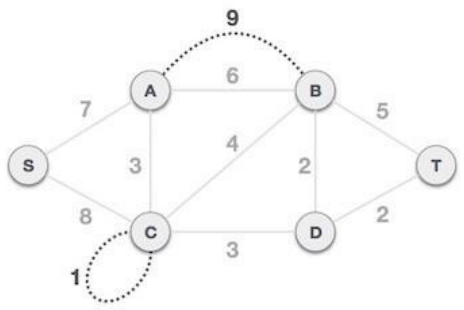
1) **Kruskal's algorithm** is used to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example −

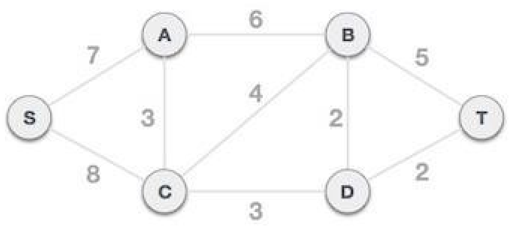


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



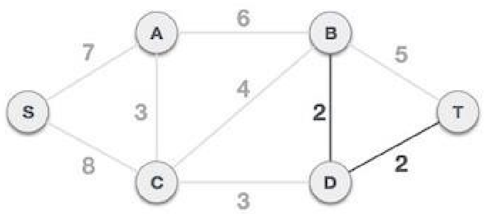
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

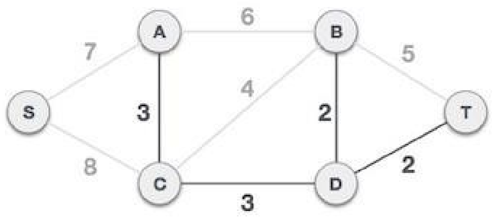
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 - Add the edge which has the least weightage

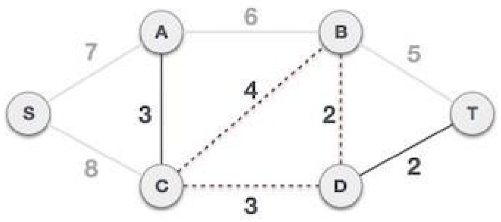
Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



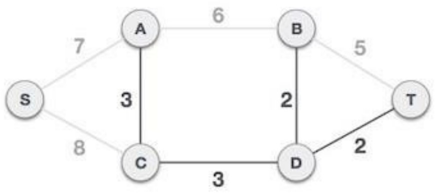
The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.
Next cost is 3, and associated edges are A,C and C,D. We add them again −



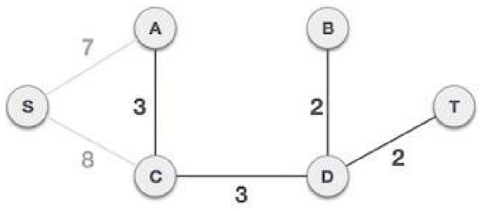
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



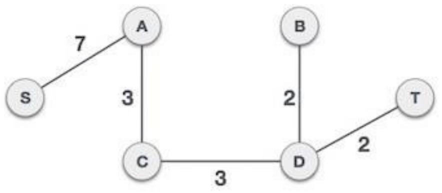
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



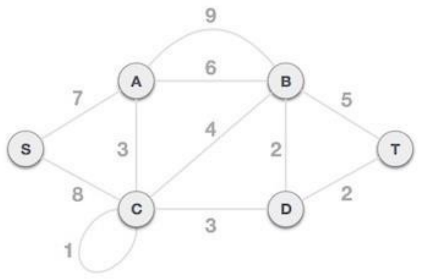
Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



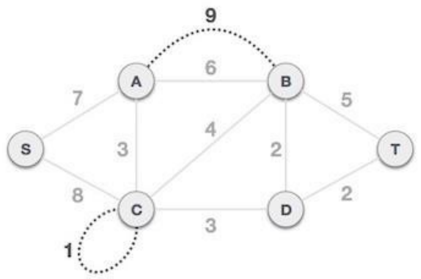
By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

2) **Prim's algorithm** is used to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms. Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

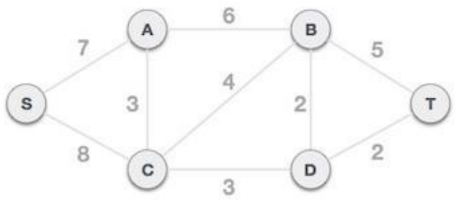
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

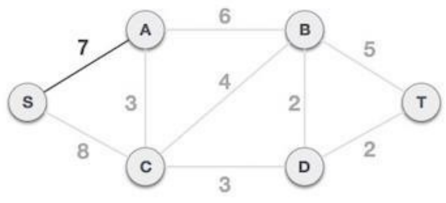


Step 2 - Choose any arbitrary node as root node

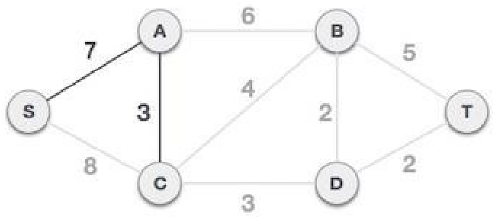
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

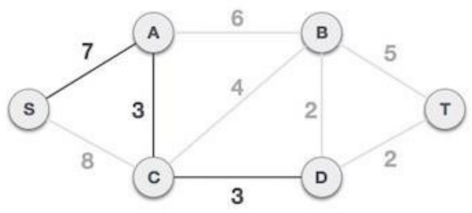
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



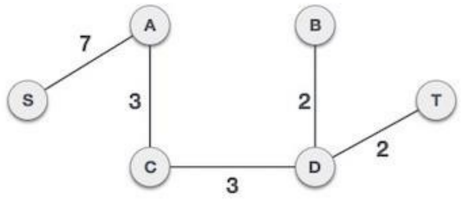
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

*******End of The - UNIT-4*******

Extra material for TREE related concepts:

Tree Terminology

In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows...

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

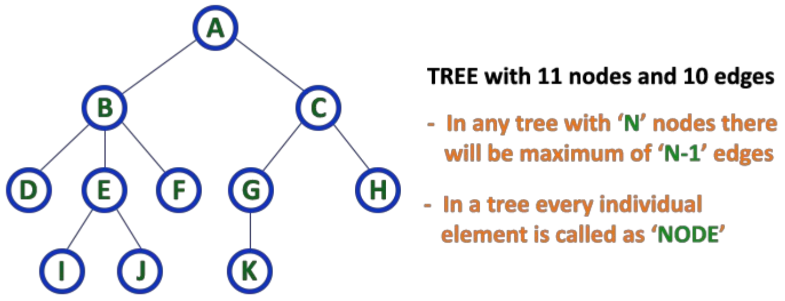
A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

In tree data structure, every individual element is called as **Node**. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

Example



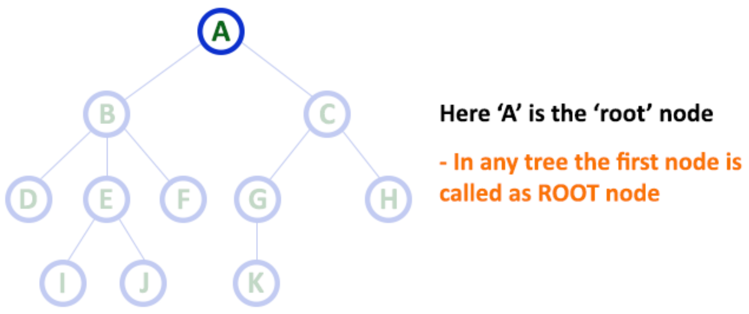
A
B
C
D
E
F
G
H
I
J
K
TREE with 11 nodes and 10 edges
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Terminology

In a tree data structure, we use the following terminology...

1. Root

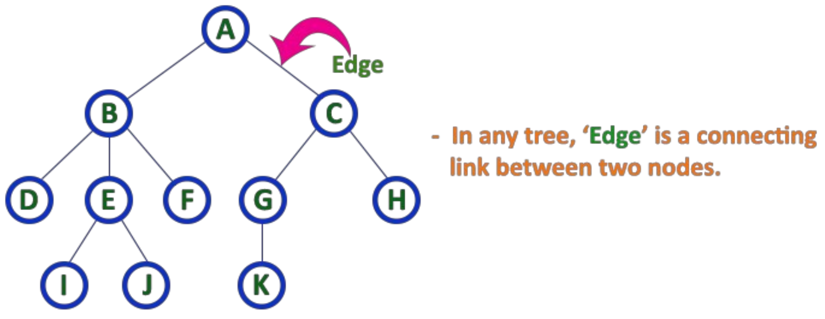
In a tree data structure, the first node is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. Edge

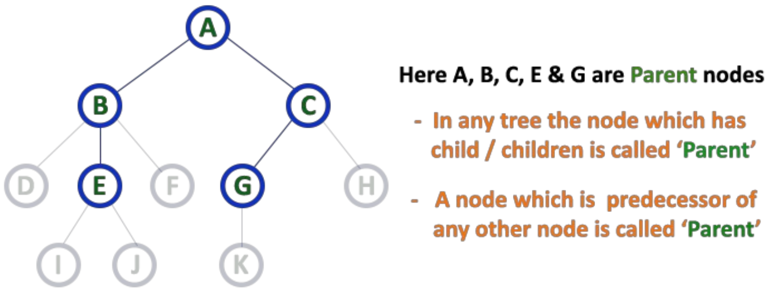
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

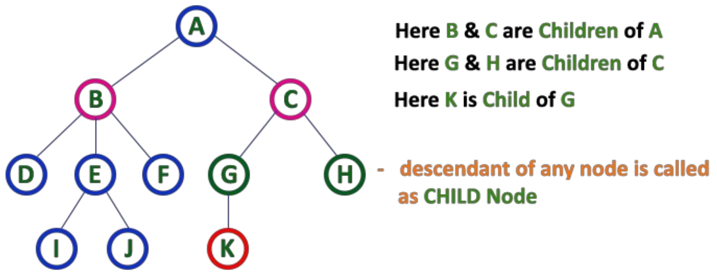
In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.


Here B & C are Children of A

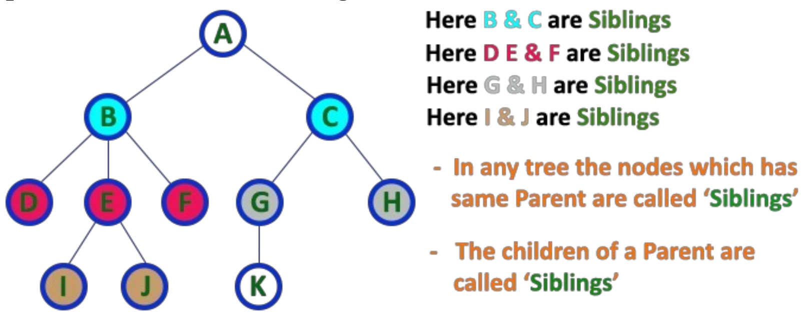
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.


Here B & C are Siblings

Here D E & F are Siblings

Here G & H are Siblings

Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

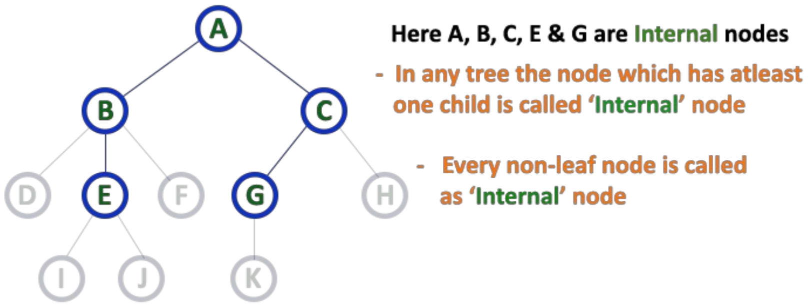
In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.

7. Internal Nodes

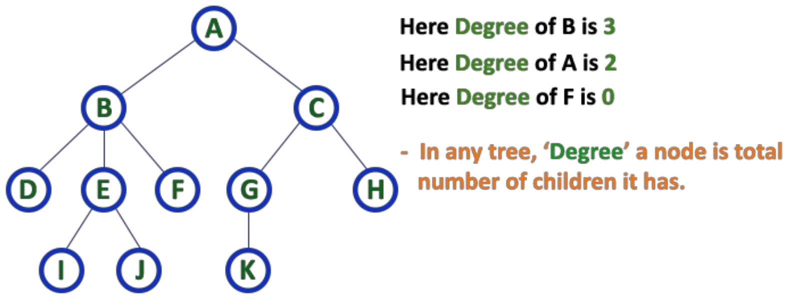
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.


A
B
C
D
E
F
G
H
I
J
K
Here A, B, C, E & G are Internal nodes
- In any tree the node which has atleast one child is called 'Internal' node
- Every non-leaf node is called as 'Internal' node

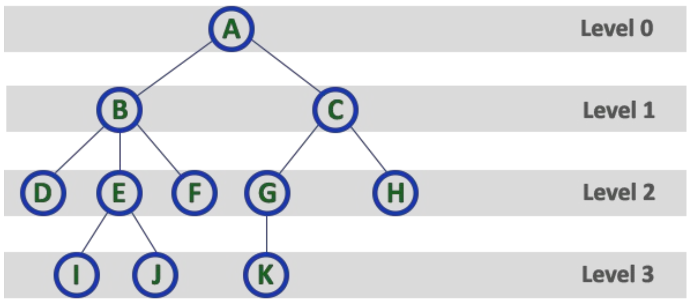
8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



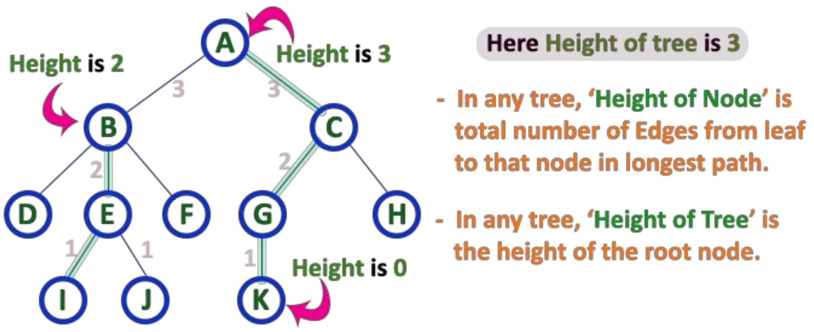
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



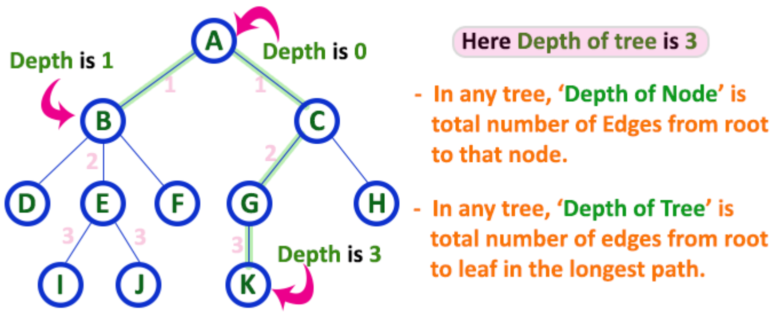
10. Height

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree.** In a tree, **height of all leaf nodes is '0'.**


Height is 2
Height is 3
Height is 0
Here Height of tree is 3
- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

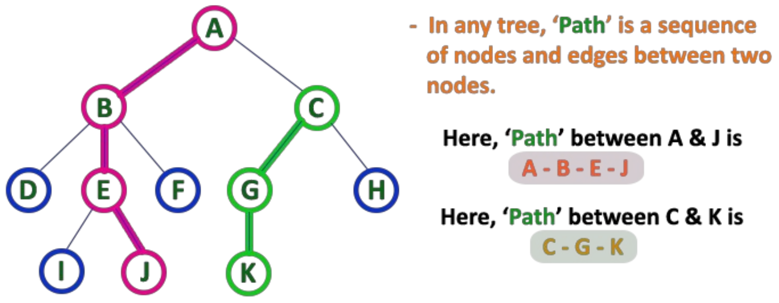
11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**


Depth is 1
Depth is 0
Depth is 3
Here Depth of tree is 3
- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

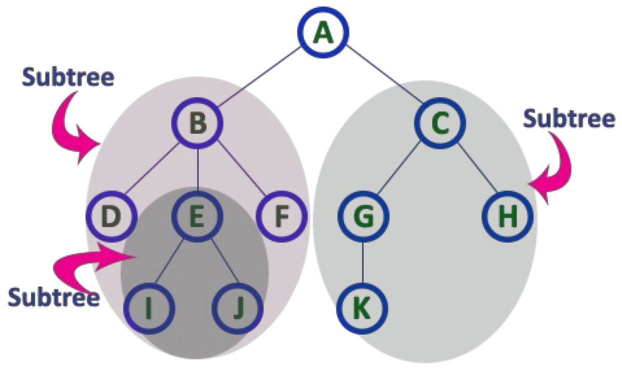
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.


- In any tree, 'Path' is a sequence of nodes and edges between two nodes.
Here, 'Path' between A & J is
A - B - E - J
Here, 'Path' between C & K is
C - G - K

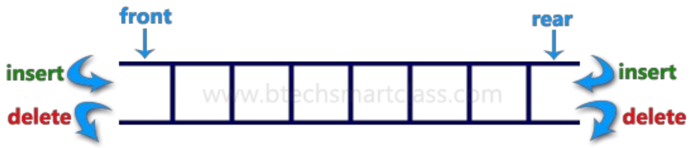
13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



Double Ended Queue:

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

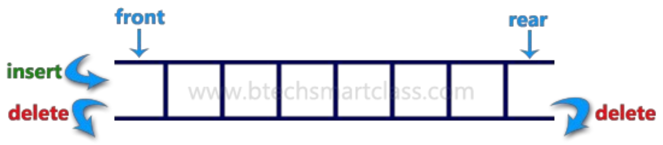


Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

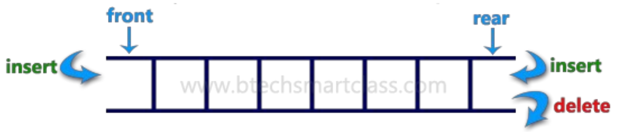
Input Restricted Double Ended Queue:

In input restricted double-ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue:

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



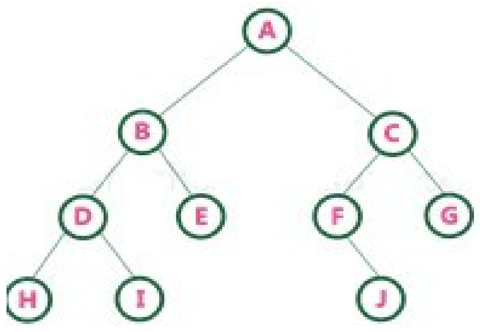
****

Threaded Binary Tree: (This topic not given in your syllabus. extra material)

A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child we use NULL pointer in that position. In any binary tree linked list representation, there are more number of NULL pointer than actual pointers. Generally, in any binary tree linked list representation, if there are 2N number of reference fields, then N+1 number of reference fields are filled with NULL (N+1 are NULL out of 2N). This NULL pointer does not play any role except indicating there is no link (no child).

Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.

Consider the following binary tree...



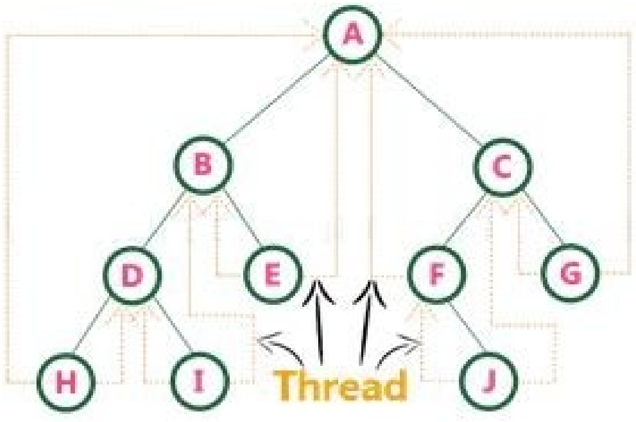
To convert above binary tree into threaded binary tree, first find the in-order traversal of that tree...

In-order traversal of above binary tree...

H - D - I - B - E - A - F - J - C - G

When we represent above binary tree using linked list representation, nodes H, I, E, F, J and G left child pointers are NULL. This NULL is replaced by address of its in-order predecessor, respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A. And nodes H, I, E, J and G right child pointers are NULL. This NULL ponters are replaced by address of its in-order successor, respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

Above example binary tree become as follows after converting into threaded binary tree.



Heap Data Structure: (This topic not given in your syllabus. extra material)

Heap data structure is a specialized binary tree based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on their value. A heap data structure, sometime called as Binary Heap.

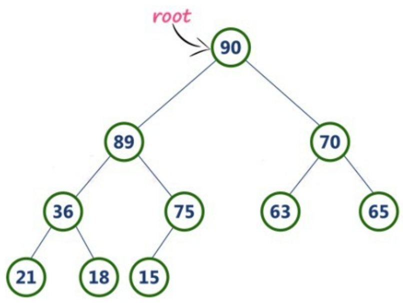
There are two types of heap data structures and they are as follows...

- Max Heap
- Min Heap

Every heap data structure has the following properties...

- 1 (Ordering): Nodes must be arranged in a order according to values based on Max heap or Min heap.
- 2 (Structural): All levels in a heap must full, except last level and nodes must be filled from left to right strictly.

Max Heep: Max heap data structure is a specialized full binary tree data structure except last leaf node can be alone. In a max heap nodes are arranged based on node value. Here, the value of the root node is greater than or equal to either of its children.



Max Heap Construction Algorithm

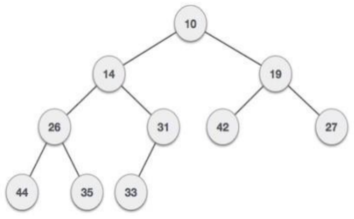
At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.

Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.



Min Heap –It is similar to Max Heap except that the value of the root node is less than or equal to either of its children.

Connected Components:

A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.

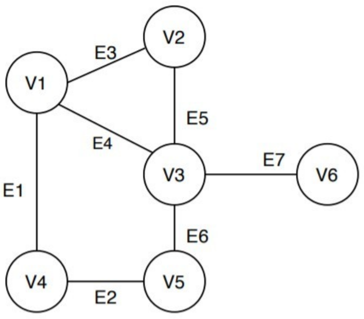
Let’s try to simplify it further, though. A set of nodes forms a connected component in an undirected graph if any node from the set of nodes can reach any other node by traversing edges. **The main point here is reachability.**
In connected components, all the nodes are always reachable from each other.

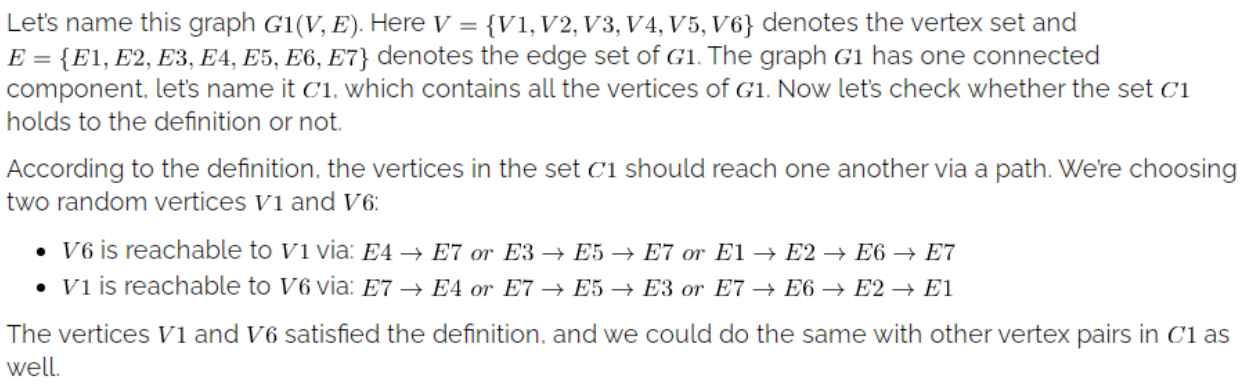
Few Examples

In this section, we’ll discuss a couple of simple examples. We’ll try to relate the examples with the definition given above.

3.1. One Connected Component

In this example, the given undirected graph has one connected component:





Let’s name this graph $G1(V, E)$. Here $V = \{V1, V2, V3, V4, V5, V6\}$ denotes the vertex set and $E = \{E1, E2, E3, E4, E5, E6, E7\}$ denotes the edge set of $G1$. The graph $G1$ has one connected component, let’s name it $C1$, which contains all the vertices of $G1$. Now let’s check whether the set $C1$ holds to the definition or not.

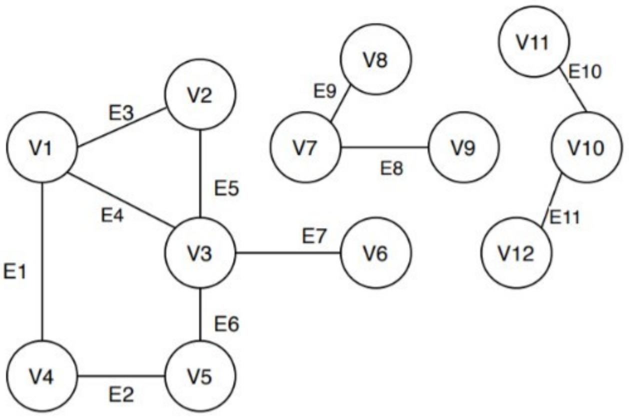
According to the definition, the vertices in the set $C1$ should reach one another via a path. We’re choosing two random vertices $V1$ and $V6$:

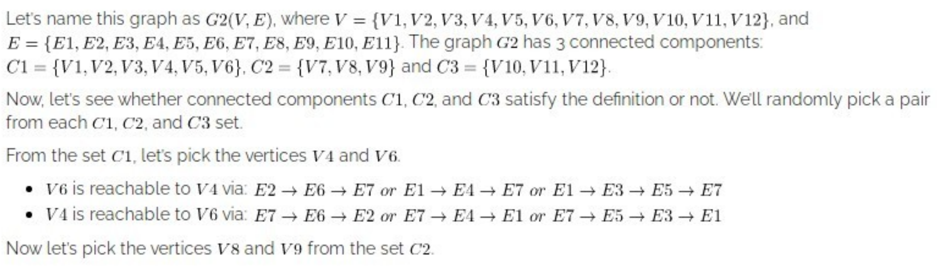
- $V6$ is reachable to $V1$ via: $E4 \rightarrow E7$ *or* $E3 \rightarrow E5 \rightarrow E7$ *or* $E1 \rightarrow E2 \rightarrow E6 \rightarrow E7$
- $V1$ is reachable to $V6$ via: $E7 \rightarrow E4$ *or* $E7 \rightarrow E5 \rightarrow E3$ *or* $E7 \rightarrow E6 \rightarrow E2 \rightarrow E1$

The vertices $V1$ and $V6$ satisfied the definition, and we could do the same with other vertex pairs in $C1$ as well.

More Than One Connected Component

In this example, the undirected graph has three connected components:





Let’s name this graph as $G2(V, E)$, where $V = \{V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12\}$, and $E = \{E1, E2, E3, E4, E5, E6, E7, E8, E9, E10, E11\}$. The graph $G2$ has 3 connected components: $C1 = \{V1, V2, V3, V4, V5, V6\}$, $C2 = \{V7, V8, V9\}$ and $C3 = \{V10, V11, V12\}$.

Now, let’s see whether connected components $C1$, $C2$, and $C3$ satisfy the definition or not. We’ll randomly pick a pair from each $C1$, $C2$, and $C3$ set.

From the set $C1$, let’s pick the vertices $V4$ and $V6$.

- $V6$ is reachable to $V4$ via: $E2 \rightarrow E6 \rightarrow E7$ *or* $E1 \rightarrow E4 \rightarrow E7$ *or* $E1 \rightarrow E3 \rightarrow E5 \rightarrow E7$
- $V4$ is reachable to $V6$ via: $E7 \rightarrow E6 \rightarrow E2$ *or* $E7 \rightarrow E4 \rightarrow E1$ *or* $E7 \rightarrow E5 \rightarrow E3 \rightarrow E1$

Now let’s pick the vertices $V8$ and $V9$ from the set $C2$.

Unit-5

Selection Sort:

Selection sort is a simple sorting algorithm. The selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending).

In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

This sorting algorithm is an comparison-based algorithm in which the list is divided into two parts, the **sorted part** at the left end and the **unsorted part** at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

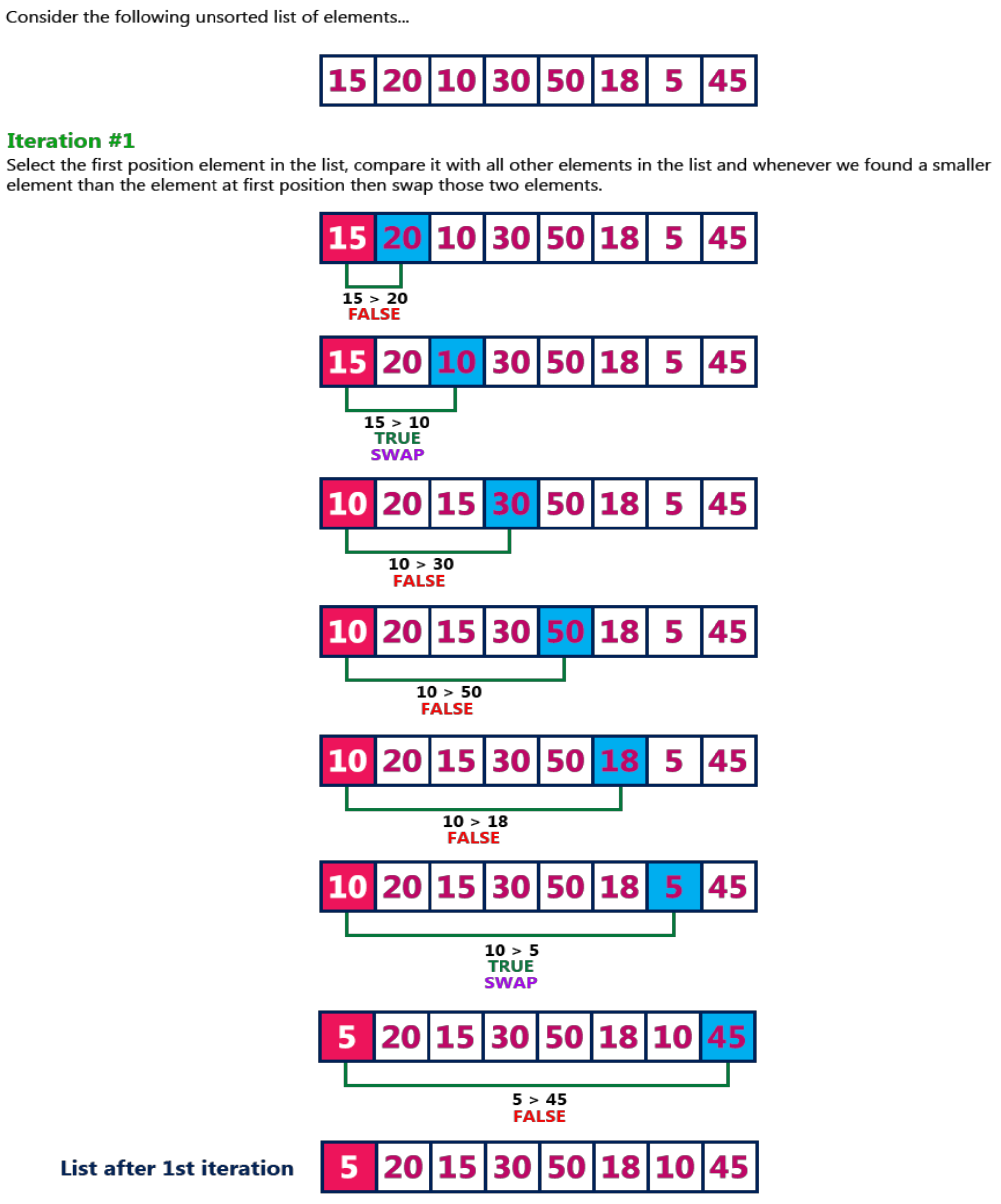
This process will be started from left most element and continues moving unsorted array boundary by one element to the right.

Step by Step Process

The selection sort algorithm is performed using following steps...

- **Step 1:** Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all other elements in the list.
- **Step 3:** For every comparison, if any element is smaller than selected element (for Ascending order), then these two are swapped.
- **Step 4:** Repeat the same procedure with next position in the list till the entire list is sorted.

Example:



Consider the following unsorted list of elements...

Iteration #1

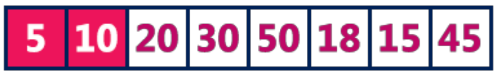
Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Continued in next page...

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

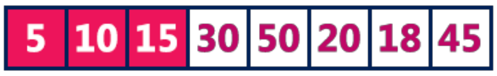
List after 2nd iteration



Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

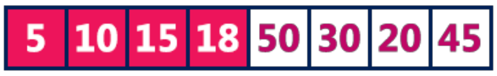
List after 3rd iteration



Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

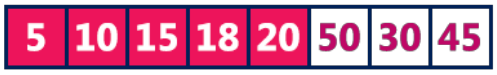
List after 4th iteration



Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

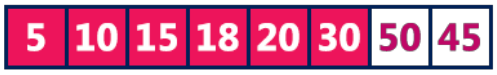
List after 5th iteration



Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

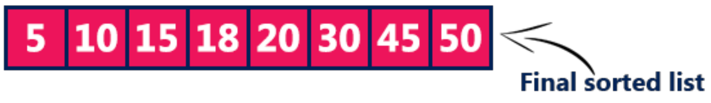
List after 6th iteration



Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration



Insertion Sort:

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

<u>Step by Step Process</u>:

The insertion sort algorithm is performed using following steps...

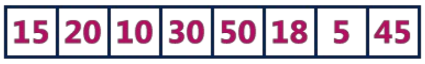
- **Step 1:** Assume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.
- **Step 2:** Consider first element from the unsorted list and insert that element into the sorted list in order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted list are moved into the sorted list.

(or)

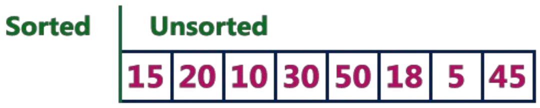
```
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater
         than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted
```

Example: in next page....

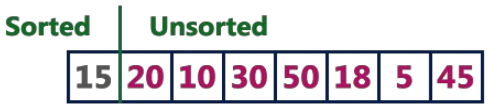
Consider the following unsorted list of elements...



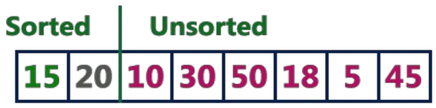
Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...



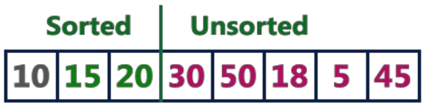
Move the first element 15 from unsorted portion to sorted portion of the list.



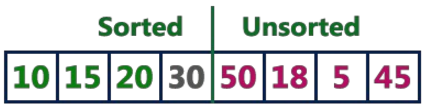
To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position



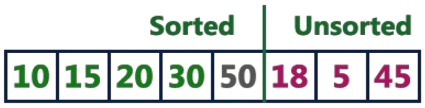
To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.



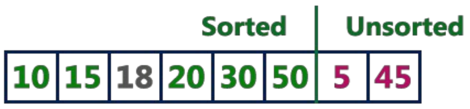
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



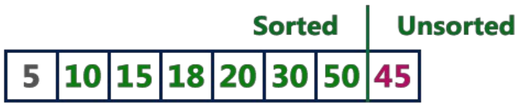
To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.



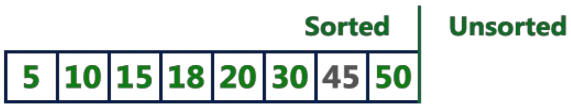
To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.



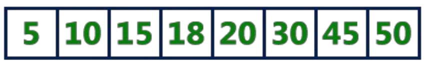
To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.



To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...



Bubble Sort: (Exchange Sort):

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2) where **n** is the number of items.

How Bubble Sort Works?

Now we should look into some practical aspects of bubble sort.

Algorithm:

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

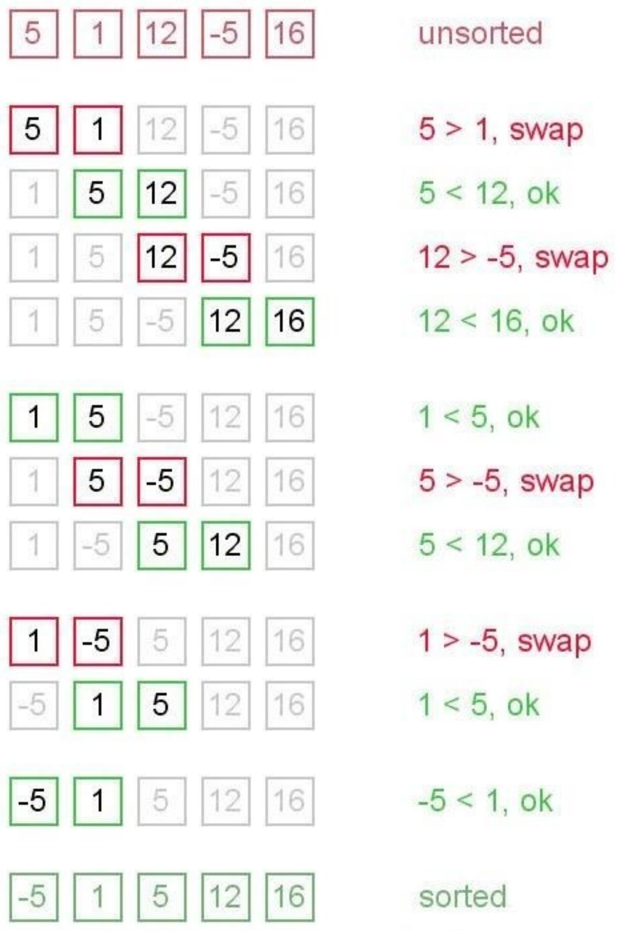
```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for
   return list

end BubbleSort
```

Step 1: Compare list[i] and list[i+1] and arrange them in the desired order.
Step 1 continued until we compare list[n-1] and list[n] to Arrange.
Step 2: In the fisrt iteration, the higher element in the list is sorted out.
So, we will repeat Step 1 with one less comparision.
....
....
Step n-1: Here, we compare only two elements called list[1] and list[2] and arrange them.
Then finally list will be in increasing order.

We take an unsorted array for our example.



Like this the given array will be sorted.

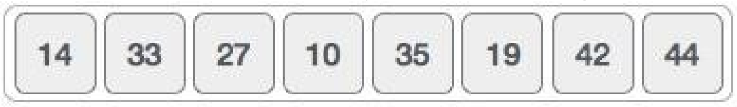
Merge Sort:

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

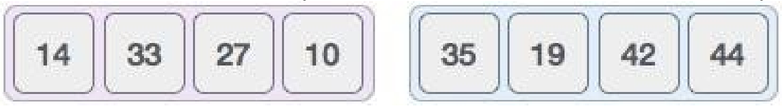
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

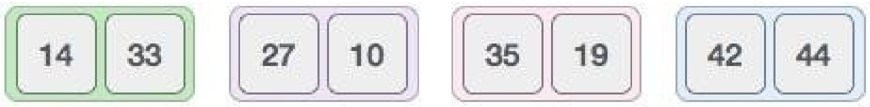
To understand merge sort, we take an unsorted array as the following −



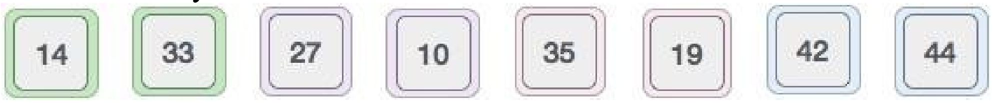
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of **8** items is divided into two arrays of size **4**.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

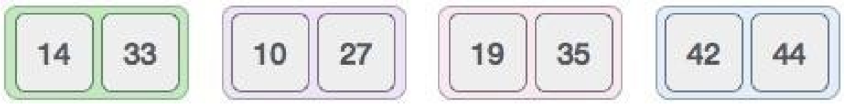


We further divide these arrays and we achieve atomic value which can no more be divided.

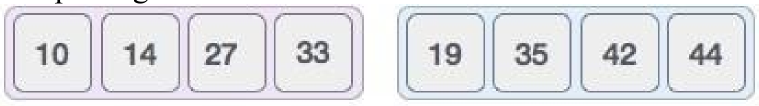


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

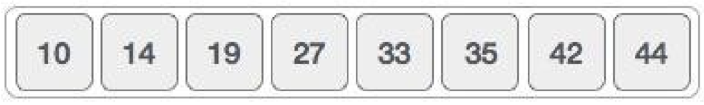
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −



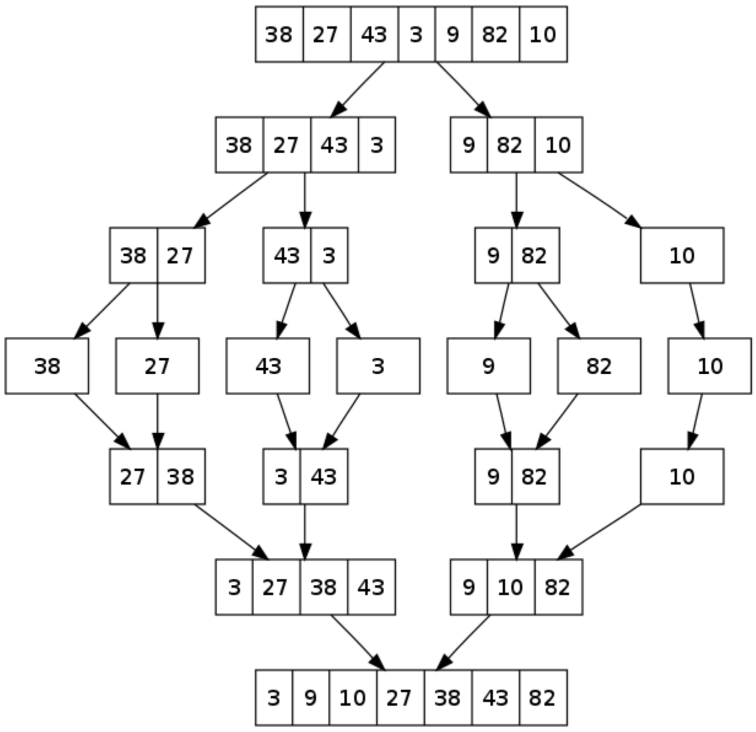
Now we should learn some programming aspects of merge sorting.

Algorithm:

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

Another example for Merge Sort
(For understanding Purpose Only)



Heap Sort:

A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called a *heap*.

Definition: The largest value at the top of the tree (Max Heap) or The least value at the top of the tree (Min Heap), so the heap sort algorithm must also reverse the order. It does this with the following steps:

1. Remove the topmost item (the largest) and replace it with the rightmost leaf. The topmost item Is stored in an array.

2. Re-establish the heap. (move the last leaf to the root)

3. Repeat steps 1 and 2 until there are no more items left in the heap.

The sorted elements are now stored in an array.

A heap sort is especially efficient for data that is already stored in a binary tree. In most cases, however, the *quick sort* algorithm is more efficient.

```
Input data: 4, 10, 3, 5, 1
                         4(0)
                        /   \
                     10(1)   3(2)
                      /   \
                   5(3)    1(4)

The numbers in bracket represent the indexes in the array representation of data.

Applying heapify procedure to index 1:
                         4(0)
                        /   \
                     10(1)    3(2)
                      /   \
                   5(3)    1(4)

Applying heapify procedure to index 0:
                        10(0)
                         /  \
                      5(1)   3(2)
                      /   \
                   4(3)    1(4)
```

As now root element is highest / max element so delete it and store in array.

```
The heapify procedure calls itself recursively to build heap in top down manner.
```

Each time root element will become max, then delete and store. This procedure for descending order. For ascending order only the difference is root element will be min element.

Quick Sort:

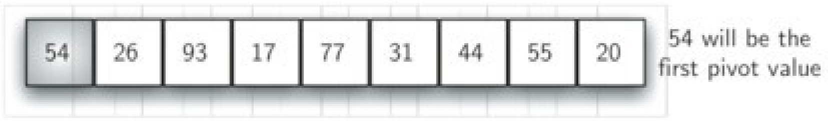
Quick sort or partition-exchange sort, is a fast sorting algorithm, which is using divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

The **quick sort** uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

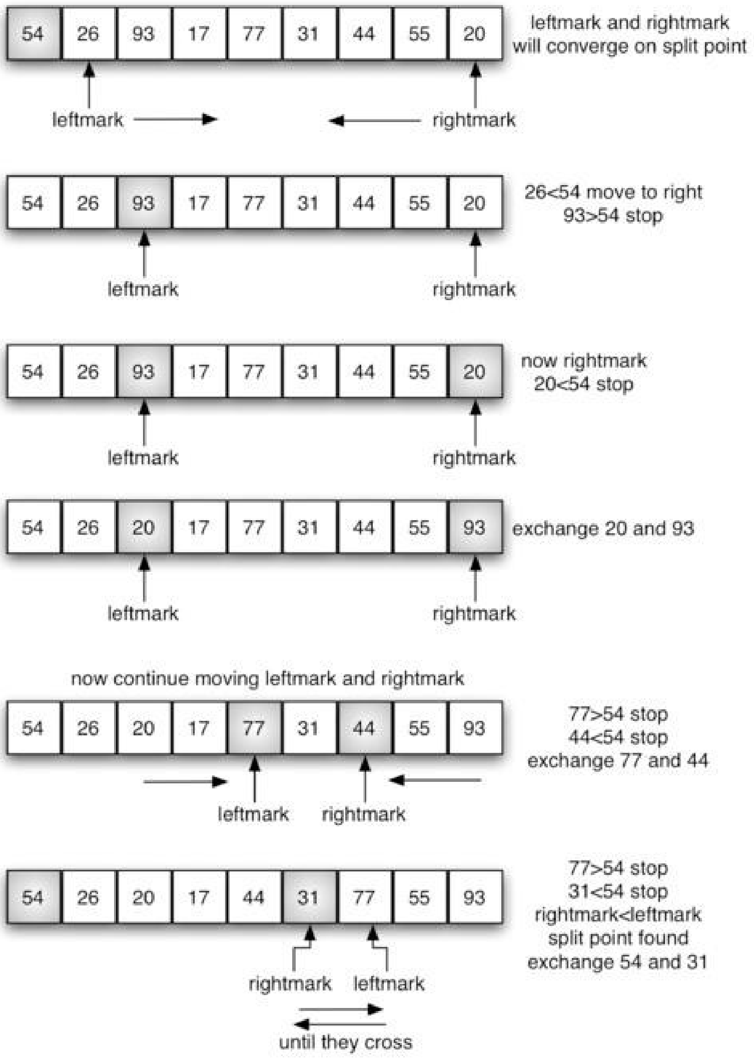
A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list.

The role of the pivot value is to assist with **splitting** the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

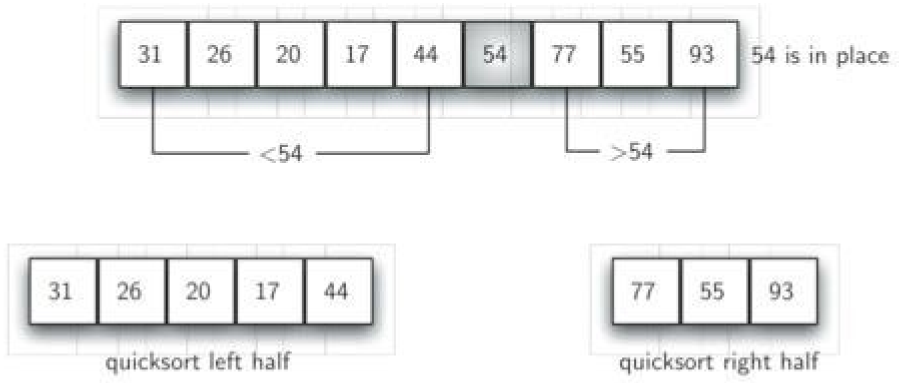
Below example shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The **partition** process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.



Partitioning begins by locating two position markers—let's call them leftmark and rightmark—at the beginning and end of the remaining items in the list (positions 1 and 8 in Figure 13). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. Figure 13 shows this process as we locate the position of 54.



We begin by incrementing leftmark until we locate a value that is greater than the pivot value. We then decrement rightmark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again. At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place (Figure 14). In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.



Other Steps for the Algorithm.
Steps to implement Quick sort:
1) Choose an element, called **pivot**, from the list. Generally pivot can be the middle index element. **2)** Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation. **3)** Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

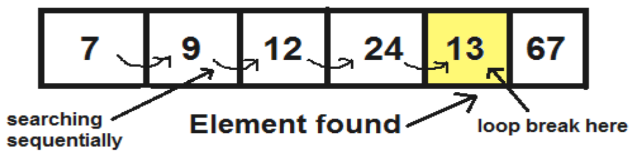
Linear Search. (or Sequential Search)

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection. If the element is not matched to any one of the list then it says not found in the list.

Linear search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the first element in the list.
- **Step 3:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 4:** If both are not matching, then compare search element with the next element in the list.
- **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- **Step 6:** If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Example: If we are searching for 13 element in the given array, it will compare sequentially....



Determine whether an array contains a particular item and, if so, the index at which the item is stored.

```
Algorithm Search(A,n)
Input: An array A[n], where n ≥ 1; an item x
Output: Index where x occurs in A, or -1
for i ← 0 to n − 1 do
    if A[i] = x then return(i);
return(-1);
```

Binary Search (Recursive Binary Search for sorted list)

The binary search algorithm can be used with only sorted list of element. The binary search can not be used for list of element which are in random order.

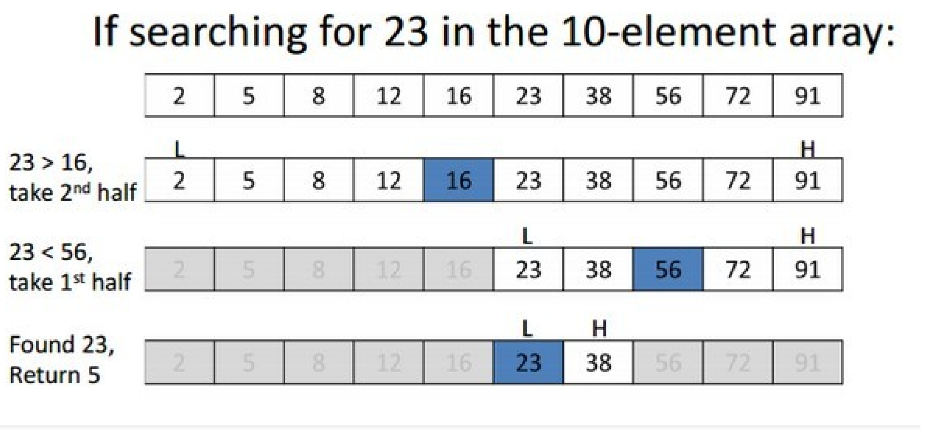
This search process starts comparing of the search element with the **middle** element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is **smaller** or **larger** than the middle element in the list.

If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element.

And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Find the middle element in the sorted list
- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 5:** If both are not matching, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8:** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.



Algorithm of binary search

```
Step 1: low = 0
Step 2: high = n-1
Step 3: while (low <= high)   repeat step 4 through step 6
    Step 4:  mid  = (low + high) / 2
    Step 5:  Is ( ele = =  a[mid])  then
                Loc = mid
                goto step 7
      Otherwise
                Step 6:  is ( ele < a[mid])?  then
                                  high = mid – 1
                              otherwise
                                  low = mid + 1
          [end while – step 3]
 Step 7:  Is ( Loc >= 0 ) ? then
         Print "search element found at location ", loc
            Otherwise
         Print "search element not found"
```

All The Best