C language important topics:

The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.

It can be defined by the following ways:

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

1) C as a mother language

C language is considered as the mother language of all the modern programming languages because **most of the compilers, JVMs, Kernels, etc. are written in C language**, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc.

It provides the core concepts like the array

, strings
, functions
, file handling
, etc. that are being used in many languages like C++
, Java
, C#
, etc.

---

2) C as a system programming language

A system programming language is used to create system software. C language is a system programming language because it **can be used to do low-level programming (for example driver and kernel)**. It is generally used to create hardware devices, OS, drivers, kernels, etc. For example, Linux kernel is written in C.

It can't be used for internet programming like Java, .Net, PHP, etc.

---

3) C as a procedural language

A procedure is known as a function, method, routine, subroutine, etc. A procedural language **specifies a series of steps for the program to solve the problem**.

A procedural language breaks the program into functions, data structures, etc.

C is a procedural language. In C, variables and function prototypes must be declared before being used.

---

4) C as a structured programming language

A structured programming language is a subset of the procedural language. **Structure means to break a program into parts or blocks** so that it may be easy to understand.

In the C language, we break the program into parts using functions. It makes the program easier to understand and modify.

---

5) C as a mid-level programming language

C is considered as a middle-level language because it **supports the feature of both low-level and high-level languages**. C language program is converted into assembly code, it supports pointer arithmetic (low-level), but it is machine independent (a feature of high-level).

A **Low-level language** is specific to one machine, i.e., machine dependent. It is machine dependent, fast to run. But it is not easy to understand.

A **High-Level language** is not specific to one machine, i.e., machine independent. It is easy to understand.

C Program

All C programs are given with C compiler so that you can quickly change the C program code.

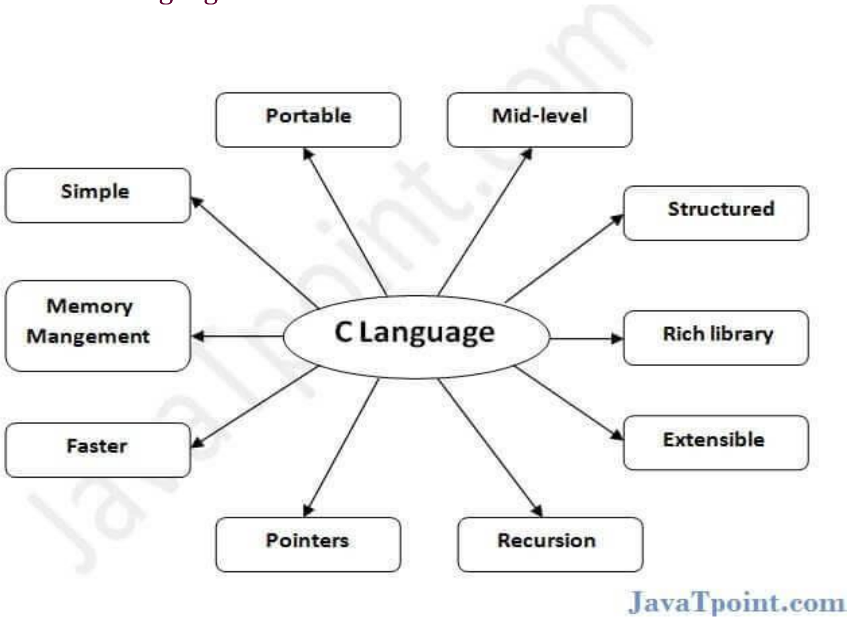
File: main.c

```c
#include <stdio.h>
int main()
{
   printf("Hello C Programming\n");
return 0;
}
```

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.

Dennis Ritchie is known as the **founder of the c language**.

Features of C Language



C is the widely used language. It provides many **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible

1) Simple

C is a simple language in the sense that it provides a **structured approach** (to break the problem into parts), **the rich set of library functions**, **data types**, etc.

2) Machine Independent or Portable

Unlike assembly language, c programs **can be executed on different machines** with some machine specific changes. Therefore, C is a machine independent language.

3) Mid-level programming language

Although, C is **intended to do low-level programming**. It is used to develop system applications such as kernel, driver, etc. It **also supports the features of a high-level language**. That is why it is known as mid-level language.

[redacted]

4) Structured programming language

C is a structured programming language in the sense that **we can break the program into parts using functions**. So, it is easy to understand and modify. Functions also provide code reusability.

5) Rich Library

C **provides a lot of inbuilt functions** that make the development fast.

6) Memory Management

It supports the feature of **dynamic memory allocation**. In C language, we can free the allocated memory at any time by calling the **free()** function.

7) Speed

The compilation and execution time of C language is fast since there are lesser inbuilt functions and hence the lesser overhead.

8) Pointer

C provides the feature of pointers. We can directly interact with the memory by using the pointers. We **can use pointers for memory, structures, functions, array**, etc.

9) Recursion

In C, we **can call the function within the function**. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

10) Extensible

C language is extensible because it **can easily adopt new features**.

First C Program

Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

To write the first c program, open the C console and write the following code:

```
#include <stdio.h>
int main(){
printf("Hello C Language");
return 0;
}
```

#include <stdio.h> includes the **standard input output** library functions. The printf() function is defined in stdio.h .

int main() The **main() function is the entry point of every program** in c language.

printf() The printf() function is **used to print data** on the console.

return 0 The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

How to compile and run the c program

There are 2 ways to compile and run the c program, by menu and by shortcut.

By menu

Now **click on the compile menu then compile sub menu** to compile the c program.

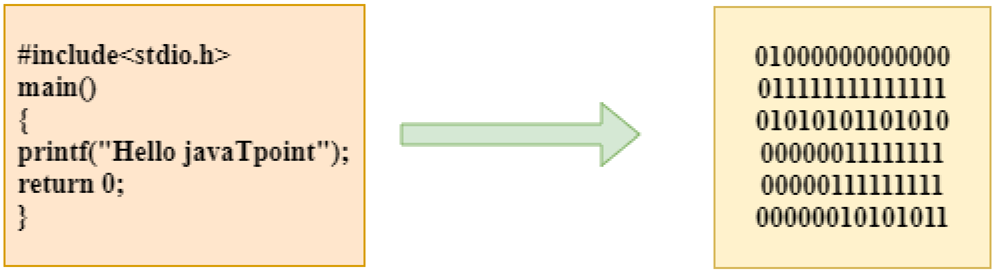
Then **click on the run menu then run sub menu** to run the c program

Or, press ctrl+f9 keys compile and run the program directly.

Compilation process in c

What is a compilation?

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.



```
#include<stdio.h>
main()
{
printf("Hello javaTpoint");
return 0;
}
```

```
01000000000000
011111111111111
01010101101010
00000011111111
00000111111111
00000010101011
```

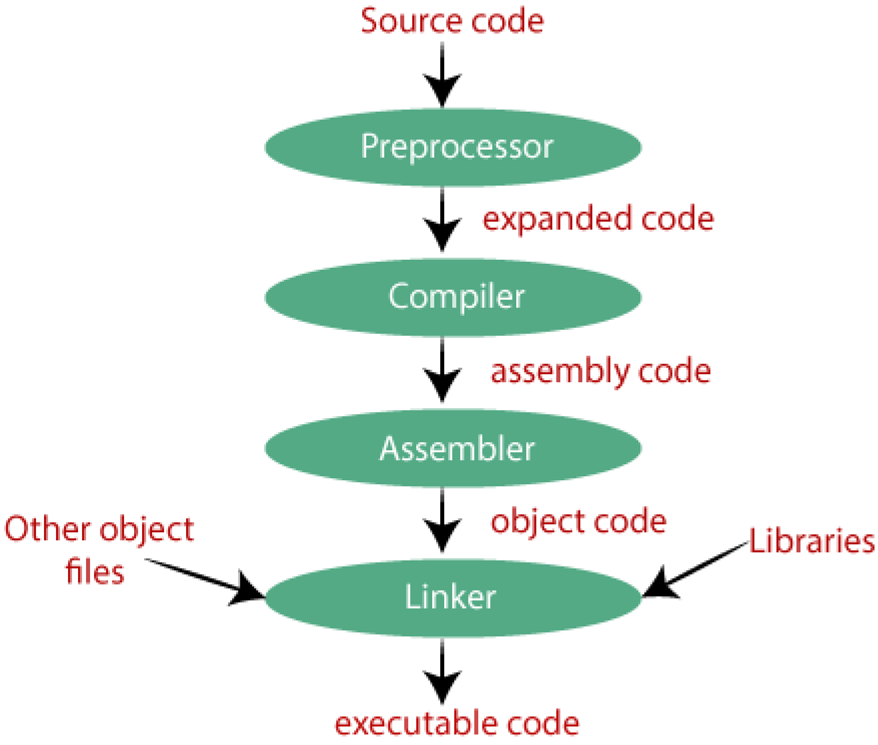
The c compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.

The preprocessor takes the source code as an input, and it removes all the comments from the source code. The preprocessor takes the preprocessor directive and interprets it. For example, if **<stdio.h>,** the directive is available in the program, then the preprocessor interprets the directive and replace this directive with the content of the **'stdio.h'** file.

The following are the phases through which our program passes before being transformed into an executable form:

- **Preprocessor**

- **Compiler**
- **Assembler**
- **Linker**



Preprocessor

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

Compiler

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

Assembler

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,' and in UNIX, the extension is 'o'. If the name of the source file is **'hello.c',** then the name of the object file would be 'hello.obj'.

Linker

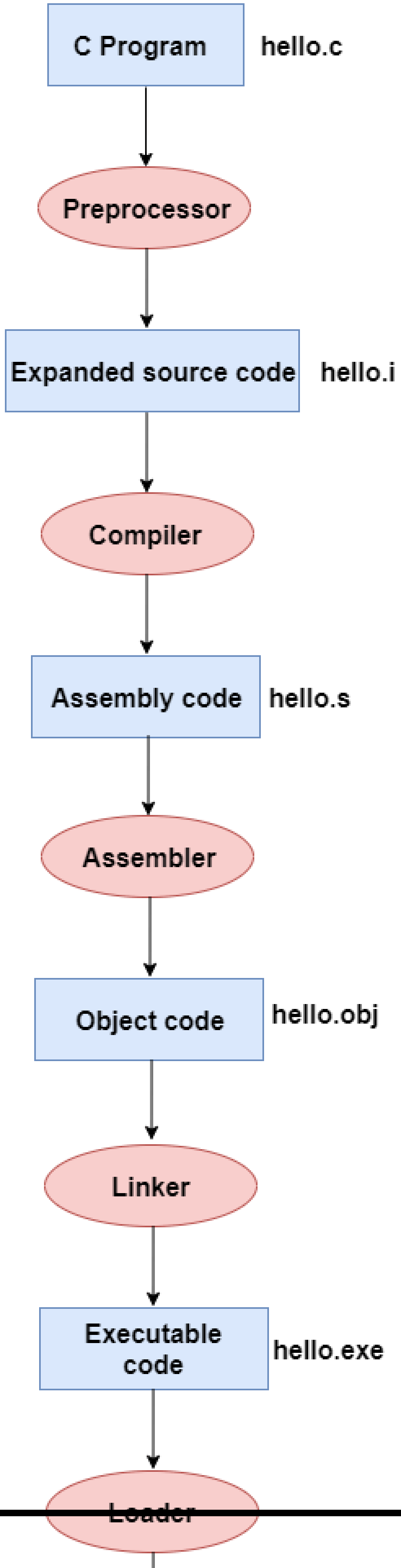
Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this. It links the object code of these files to our program. Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files. The output of the linker is the executable file. The name of the executable file is the same as the source file but differs only in their extensions. In DOS, the extension of the executable file is '.exe', and in UNIX, the executable file can be named as 'a.out'. For example, if we are using printf() function in a program, then the linker adds its associated code in an output file.

Let's understand through an example.

hello.c

```c
#include <stdio.h>
int main()
{
   printf("Hello java");
   return 0;
}
```

Now, we will create a flow diagram of the above program:


C Program
hello.c
Preprocessor
Expanded source code
hello.i
Compiler
Assembly code
hello.s
Assembler
Object code
hello.obj
Linker
Executable code
hello.exe
Loader

In the above flow diagram, the following steps are taken to execute a program:

- Firstly, the input file, i.e., **hello.c,** is passed to the preprocessor, and the preprocessor converts the source code into expanded source code. The extension of the expanded source code would be **hello.i.**
- The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code. The extension of the assembly code would be **hello.s.**
- This assembly code is then sent to the assembler, which converts the assembly code into object code.
- After the creation of an object code, the linker creates the executable file. The loader will then load the executable file for the execution.

printf() and scanf() in C:

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function

The **printf() function** is used for output. It prints the given statement to the console.

The syntax of printf() function is given below:

```
printf("format string",argument_list);
```

The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.

scanf() function

The **scanf() function** is used for input. It reads the input data from the console.

```
scanf("format string",argument_list);
```

Program to print cube of given number

Let's see a simple example of c language that gets input from the user and prints the cube of the given number.

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

```c
int number;
printf("enter a number:");
scanf("%d",&number);
printf("cube of number is:%d ",number*number*number);
return 0;
}
```

Output

```
enter a number:5
cube of number is:125
```

The **scanf("%d",&number)** statement reads integer number from the console and stores the given value in number variable.

The **printf("cube of number is:%d ",number*number*number)** statement prints the cube of number on the console.

Program to print sum of 2 numbers

Let's see a simple example of input and output in C language that prints addition of 2 numbers.

```c
#include<stdio.h>
int main(){
int x=0,y=0,result=0;

printf("enter first number:");
scanf("%d",&x);
printf("enter second number:");
scanf("%d",&y);

result=x+y;
printf("sum of 2 numbers:%d ",result);

return 0;
}
```

Output

```
enter first number:9
enter second number:9
sum of 2 numbers:18
```

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:

```c
type variable_list;
```

The example of declaring the variable is given below:

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

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

We can also provide values while declaring the variables as given below:

```c
int a=10,b=20;//declaring 2 variable of integer type
float f=20.8;
char c='A';
```

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, float, etc.

Valid variable names:

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

Invalid variable names:

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

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.

```
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.

```
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.

```
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
}
```

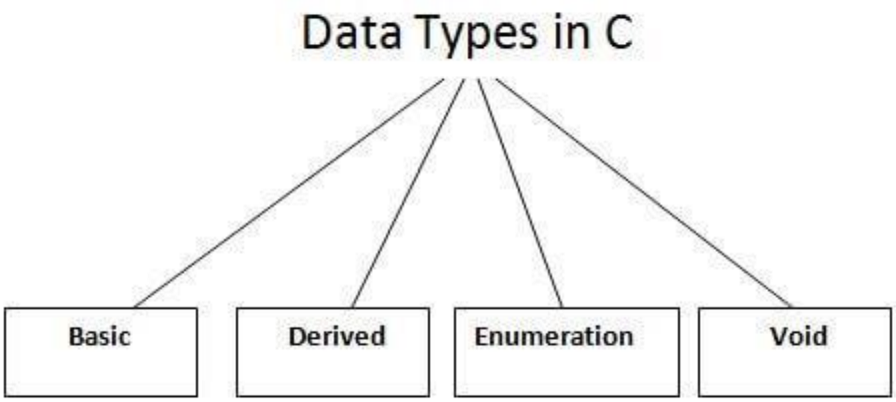
External Variable

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**.

```
extern int x=10;//external variable (also global)
```

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

Basic Data 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.

Let's see the basic data types. Its size is given **according to 32-bit architecture**.

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	10 byte	

Keywords in C:

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

auto	break	case	char	const	continue	default
double	else	enum	extern	float	for	goto

int	long	register	return	short	signed	sizeof
struct	switch	typedef	union	unsigned	void	volatile

C Identifiers:

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore. If the identifier is not used in the external linkage, then it is called as an internal identifier. If the identifier is used in the external linkage, then it is called as an external identifier.

We can say that an identifier is a collection of alphanumeric characters that begins either with an alphabetical character or an underscore, which are used to represent various programming elements such as variables, functions, arrays, structures, unions, labels, etc. There are 52 alphabetical characters (uppercase and lowercase), underscore character, and ten numerical digits (0-9) that represent the identifiers. There is a total of 63 alphanumerical characters that represent the identifiers.

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Types of identifiers :

- Internal identifier
- External identifier

Internal Identifier

If the identifier is not used in the external linkage, then it is known as an internal identifier. The internal identifiers can be local variables.

External Identifier

If the identifier is used in the external linkage, then it is known as an external identifier. The external identifiers can be function names, global variables.

Differences between Keyword and Identifier

Keyword	Identifier
Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

Let's understand through an example.

```
int main()

{
  int a=10;
  int A=20;
  printf("Value of a is : %d",a);
  printf("\nValue of A is :%d",A);
  return 0;
```

}

Output

```
Value of a is : 10
Value of A is :20
```

The above output shows that the values of both the variables, 'a' and 'A' are different. Therefore, we conclude that the identifiers are case sensitive.

C Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc.

There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators
- Bitwise Operators
- Ternary or Conditional Operators
- Assignment Operator
- Misc Operator

Precedence of Operators in C

The precedence of operator species that which operator will be evaluated first and next. The associatively specifies the operator direction to be evaluated; it may be left to right or right to left.

Let's understand the precedence by the example given below:

The value variable will contain **210** because * (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C operators is given below:

Category	**Operator**	**Associativity**

Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Comments in C :

Comments in C language are used to provide information about lines of code. It is widely used for documenting code. There are 2 types of comments in the C language.

1. Single Line Comments
2. Multi-Line Comments

Single Line Comments

Single line comments are represented by double slash \\. Let's see an example of a single line comment in C.

```
#include<stdio.h>
int main(){
   //printing information
   printf("Hello C");
return 0;
}
```

Output:

```
Hello C
```

Multiline Comments:

Multi-Line comments are represented by slash asterisk * ... *\. It can occupy many lines of code, but it can't be nested. Syntax:

```
/*
code
to be commented
*/
```

Let's see an example of a multi-Line comment in C.

```
#include<stdio.h>
int main(){
   /*printing information
     Multi-Line Comment*/
   printf("Hello C");
return 0;
}
```

Output:

```
Hello C
```

C Format Specifier :

The Format specifier is a string used in the formatted input and output functions. The format string determines the format of the input and output. The format string always starts with a '%' character.

The commonly used format specifiers in printf() function are:

Format specifier	Description
%d or %i	It is used to print the signed integer value where signed integer means that the variable can positive and negative values.
%u	It is used to print the unsigned integer value where the unsigned integer means that the variabl only positive value.
%o	It is used to print the octal unsigned integer where octal integer value always starts with a 0 value.
%x	It is used to print the hexadecimal unsigned integer where the hexadecimal integer value always s a 0x value. In this, alphabetical characters are printed in small letters such as a, b, c, etc.
%X	It is used to print the hexadecimal unsigned integer, but %X prints the alphabetical characters in u such as A, B, C, etc.
%f	It is used for printing the decimal floating-point values. By default, it prints the 6 values after '.'.
%e/%E	It is used for scientific notation. It is also known as Mantissa or Exponent.
%g	It is used to print the decimal floating-point values, and it uses the fixed precision, i.e., the value decimal in input would be exactly the same as the value in the output.
%p	It is used to print the address in a hexadecimal form.
%c	It is used to print the unsigned character.

%s	It is used to print the strings.
%ld	It is used to print the long-signed integer value.

Let's understand the format specifiers in detail through an example.

- **%d**

```
int main()
{
 int b=6;
 int c=8;
 printf("Value of b is:%d", b);
 printf("\nValue of c is:%d",c);

  return 0;
}
```

In the above code, we are printing the integer value of b and c by using the %d specifier.

Escape Sequence in C :

An escape sequence in C language is a sequence of characters that doesn't represent itself when used inside string literal or character.

It is composed of two or more characters starting with backslash \. For example: \n represents new line.

List of Escape Sequences in C :

Escape Sequence	Meaning

\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark
\nnn	octal number
\xhh	hexadecimal number
\0	Null

Escape Sequence Example

```
#include<stdio.h>
      int main(){
    int number=50;
   printf("You\nare\nlearning\n\'c\' language\n\"Do you know C language\"");
return 0;
}
```

Output:

```
You
are
learning
'c' language
"Do you know C language"
```

Constants in C :

A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3.4, "c programming" etc.

There are different types of constants in C programming.

List of Constants in C

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.

2 ways to define constant in C :

There are two ways to define constant in C programming

1.const keyword

2define preprocessor

1) C const keyword :

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

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

Now, the value of PI variable can't be changed.

```
#include<stdio.h>

int main(){
   const float PI=3.14;
   printf("The value of PI is: %f",PI);
   return 0;
}
```

Output:

```
The value of PI is: 3.140000
```

If you try to change the the value of PI, it will render compile time error.

```
#include<stdio.h>

int main(){
const float PI=3.14;
PI=4.5;
printf("The value of PI is: %f",PI);
   return 0;
}
```

Output:

```
Compile Time Error: Cannot modify a const object
```

What are literals :

Literals are the constant values assigned to the constant variables. We can say that the literals represent the fixed values that cannot be modified. It also contains memory but does not have references as variables. For example, const int =10; is a constant integer expression in which 10 is an integer literal.

Types of literals :

There are four types of literals that exist in C programming:

- **Integer literal**
- **Float literal**
- **Character literal**
- **String literal**

Integer literal

It is a numeric literal that represents only integer type values. It represents the value neither in fractional nor exponential part.

It can be specified in the following three ways:

Decimal number (base 10) :

It is defined by representing the digits between 0 to 9. For example, 45, 67, etc.

Octal number (base 8) :

It is defined as a number in which 0 is followed by digits such as 0,1,2,3,4,5,6,7. For example, 012, 034, 055, etc.

Hexadecimal number (base 16) :

It is defined as a number in which 0x or 0X is followed by the hexadecimal digits (i.e., digits from 0 to 9, alphabetical characters from (a-z) or (A-Z)).

An integer literal is suffixed by following two sign qualifiers:

L or l: It is a size qualifier that specifies the size of the integer type as long.

U or u: It is a sign qualifier that represents the type of the integer as unsigned. An unsigned qualifier contains only positive values.

```
#include <stdio.h>
int main()
{
   const int a=23;  // constant integer literal
   printf("Integer literal : %d", a);
   return 0;
}
```

Output

```
Integer literal : 23
```

Float literal :

It is a literal that contains only floating-point values or real numbers. These real numbers contain the number of parts such as integer part, real part, exponential part, and fractional part. The floating-point literal must be specified either in decimal or in exponential form. Let's understand these forms in brief.

Decimal form

The decimal form must contain either decimal point, exponential part, or both. If it does not contain either of these, then the compiler will throw an error. The decimal notation can be prefixed either by '+' or '-' symbol that specifies the positive and negative numbers.

Examples of float literal in decimal form are:

1. 1.2, +9.0, -4.5

Let's see a simple example of float literal in decimal form.

```
#include <stdio.h>
int main()
{
   const float a=4.5; // constant float literal
   const float b=5.6; // constant float literal
   float sum;
   sum=a+b;
   printf("%f", sum);
   return 0;
}
```

Output

```
10.100000
```

Exponential form :

The exponential form is useful when we want to represent the number, which is having a big magnitude. It contains two parts, i.e., mantissa and exponent. For example, the number is 2340000000000, and it can be expressed as 2.34e12 in an exponential form.

Character literal :

A character literal contains a single character enclosed within single quotes. If multiple characters are assigned to the variable, then we need to create a character array. If we try to store more than one character in a variable, then the warning of a **multi-character character constant** will be generated. Let's observe this scenario through an example.

```
#include <stdio.h>
int main()
{
   const char c='ak';
   printf("%c",c);
   return 0;
}
```

In the above code, we have used two characters, i.e., 'ak', within single quotes. So, this statement will generate a warning as shown below.

String literal :

A string literal represents multiple characters enclosed within double-quotes. It contains an additional character, i.e., '\0' (null character), which gets automatically inserted. This null character specifies the termination of the string. We can use the '+' symbol to concatenate two strings.

For example,

String1= "javatpoint";

String2= "family";

To concatenate the above two strings, we use '+' operator, as shown in the below statement:

"javatpoint " + "family"= javatpoint family

Tokens in C :

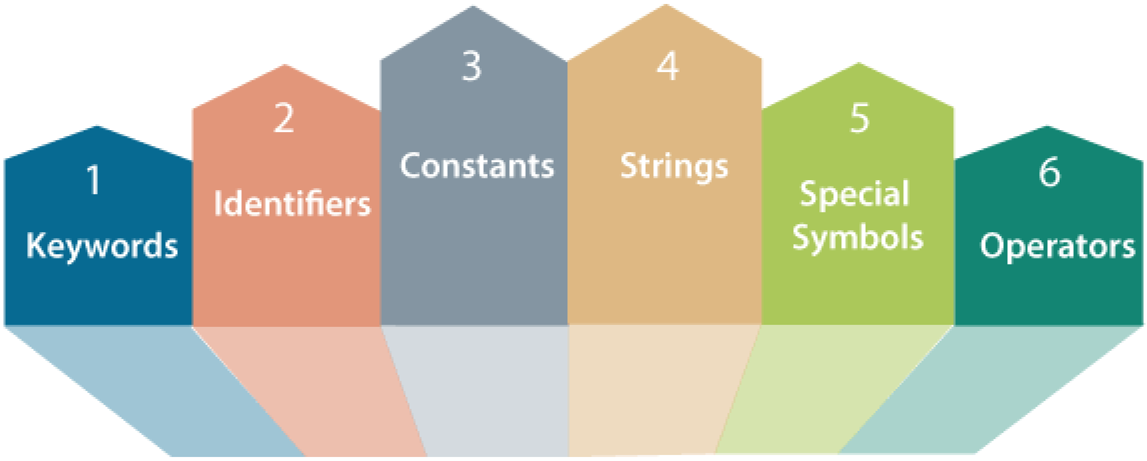
Tokens in C is the most important element to be used in creating a program in C. We can define the token as the smallest individual element in C. For `example, we cannot create a sentence without using words; similarly, we cannot create a program in C without using tokens in C. Therefore, we can say that tokens in C is the building block or the basic component for creating a program in C language

.

Classification of tokens in C :

Tokens in C language

can be divided into the following categories:



Classification of C Tokens

- Keywords in C
- Identifiers in C
- Strings in C
- Operators in C

- Constant in C
- Special Characters in C

Let's understand each token one by one.

Keywords in C

Keywords in C

can be defined as the **pre-defined** or the **reserved words** having its own importance, and each keyword has its own functionality. Since keywords are the pre-defined words used by the compiler, so they cannot be used as the variable names. If the keywords are used as the variable names, it means that we are assigning a different meaning to the keyword, which is not allowed. C language supports 32 keywords given below:

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

Identifiers in C :

Identifiers in C

are used for naming variables, functions, arrays, structures, etc. Identifiers in C are the user-defined words. It can be composed of uppercase letters, lowercase letters, underscore, or digits, but the starting letter should be either an underscore or an alphabet. Identifiers cannot be used as keywords. Rules for constructing identifiers in C are given below:

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Strings in C :

Strings in C

are always represented as an array of characters having null character '\0' at the end of the string. This null character denotes the end of the string. Strings in C are enclosed within double quotes, while characters are enclosed within single characters. The size of a string is a number of characters that the string contains.

Now, we describe the strings in different ways:

char a[10] = "javatpoint"; // The compiler allocates the 10 bytes to the 'a' array.

char a[] = "javatpoint"; // The compiler allocates the memory at the run time.

char a[10] = {'j','a','v','a','t','p','o','i','n','t','\0'}; // String is represented in the form of characters.

Operators in C

Operators in C

is a special symbol used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands. Depending on the number of operands, operators are classified as follows:

Unary Operator :

A unary operator is an operator applied to the single operand. For example: increment operator (++), decrement operator (--), sizeof, (type)*.

Binary Operator :

The binary operator is an operator applied between two operands. The following is the list of the binary operators:

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators
- Bitwise Operators
- Conditional Operators
- Assignment Operator
- Misc Operator

Constants in C :

A constant is a value assigned to the variable which will remain the same throughout the program, i.e., the constant value cannot be changed.

There are two ways of declaring constant:

- Using const keyword
- Using #define pre-processor

Types of constants in C

Constant	Example
Integer constant	**10, 11, 34, etc.**
Floating-point constant	**45.6, 67.8, 11.2, etc.**
Octal constant	**011, 088, 022, etc.**
Hexadecimal constant	**0x1a, 0x4b, 0x6b, etc.**

Character constant	**'a', 'b', 'c', etc.**
String constant	**"java", "c++", ".net", etc.**

Special characters in C

Some special characters are used in C, and they have a special meaning which cannot be used for another purpose.

- **Square brackets [ ]:** The opening and closing brackets represent the single and multidimensional subscripts.
- **Simple brackets ( ):** It is used in function declaration and function calling. For example, printf() is a pre-defined function.
- **Curly braces { }:** It is used in the opening and closing of the code. It is used in the opening and closing of the loops.
- **Comma (,):** It is used for separating for more than one statement and for example, separating function parameters in a function call, separating the variable when printing the value of more than one variable using a single printf statement.
- **Hash/pre-processor (#):** It is used for pre-processor directive. It basically denotes that we are using the header file.
- **Asterisk (*):** This symbol is used to represent pointers and also used as an operator for multiplication.
- **Tilde (~):** It is used as a destructor to free memory.
- **Period (.):** It is used to access a member of a structure or a union.

C if else Statement :

The if-else statement in C is used to perform the operations based on some specific condition. The operations specified in if block are executed if and only if the given condition is true.

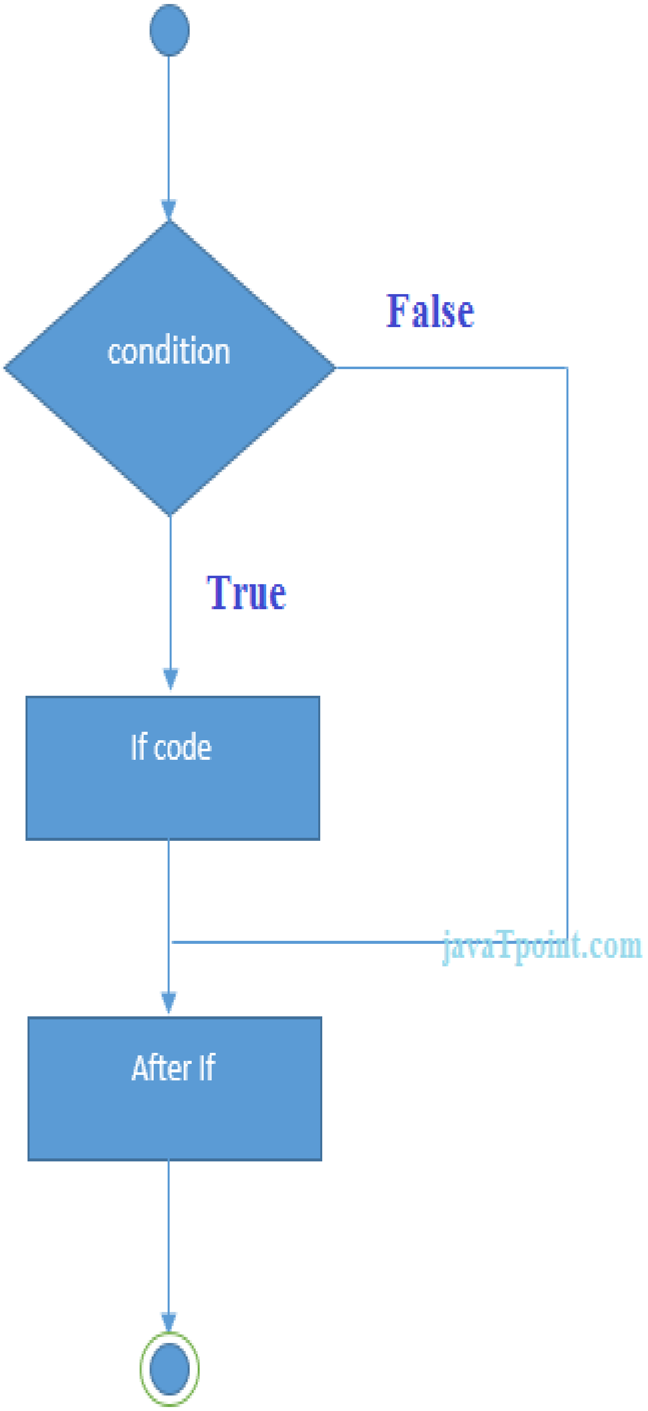
There are the following variants of if statement in C language.

- If statement
- If-else statement
- If else-if ladder
- Nested if

If Statement :

The if statement is used to check some given condition and perform some operations depending upon the correctness of that condition. It is mostly used in the scenario where we need to perform the different operations for the different conditions. The syntax of the if statement is given below.

```
if(expression){

//code to be executed
```



}

Flowchart of if statement in C :

Let's see a simple example of C language if statement.

```
#include<stdio.h>

int main(){
int number=0;
printf("Enter a number:");
scanf("%d",&number);
if(number%2==0){
printf("%d is even number",number);
}    return 0;
1.  }
```

Output

```
Enter a number:4
4 is even number
enter a number:5
```

Program to find the largest number of the three.\

```
#include <stdio.h>

int main()
{
   int a, b, c;
    printf("Enter three numbers?");
   scanf("%d %d %d",&a,&b,&c);
   if(a>b && a>c)
   {
      printf("%d is largest",a);
   }
   if(b>a  && b > c)
   {
      printf("%d is largest",b);
   }
   if(c>a && c>b)
```

```
  {
    printf("%d is largest",c);
  }
  if(a == b && a == c)
  {
    printf("All are equal");
  }
}
```

Output

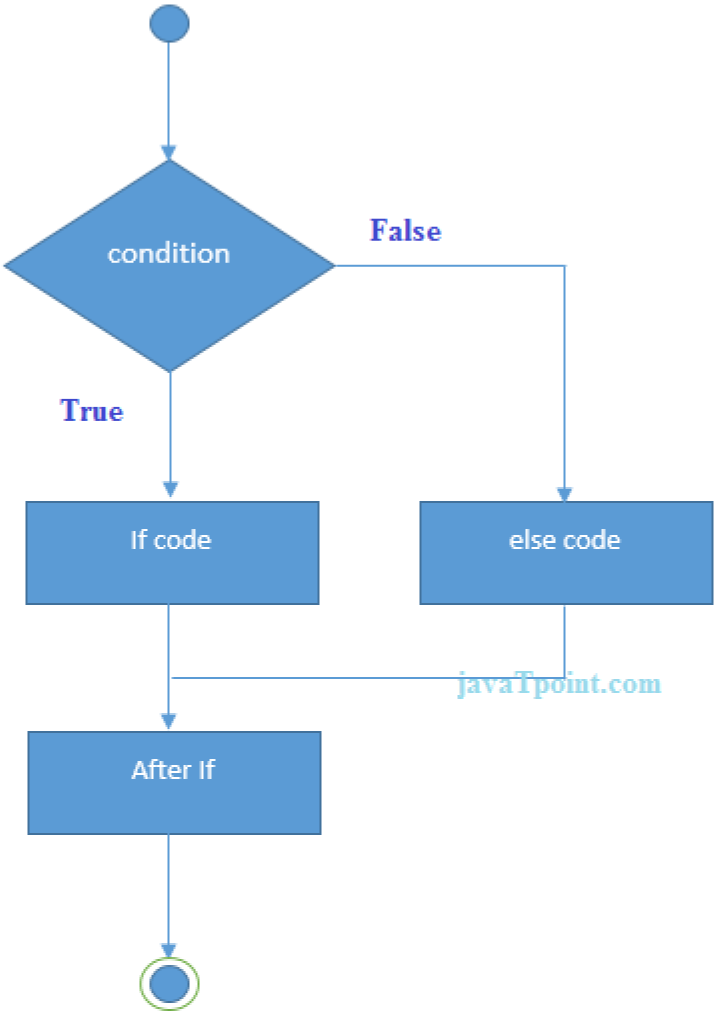
```
Enter three numbers?
12 23 34
34 is largest
```

If-else Statement :

The if-else statement is used to perform two operations for a single condition. The if-else statement is an extension to the if statement using which, we can perform two different operations, i.e., one is for the correctness of that condition, and the other is for the incorrectness of the condition. Here, we must notice that if and else block cannot be executed simiulteneously. Using if-else statement is always preferable since it always invokes an otherwise case with every if condition. The syntax of the if-else statement is given below.

```
if(expression){

//code to be executed if condition is true
}else{
//code to be executed if condition is false
}
```

Flowchart of the if-else statement in C



Let's see the simple example to check whether a number is even or odd using if-else statement in C language.

```
#include<stdio.h>

int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
if(number%2==0){
```

```
printf("%d is even number",number);
}
else{
printf("%d is odd number",number);
}
return 0;
}
```

Output

```
enter a number:4
4 is even number
enter a number:5
5 is odd number
```

Program to check whether a person is eligible to vote or not.:

```
#include <stdio.h>

int main()
{
   int age;
   printf("Enter your age?");
   scanf("%d",&age);
   if(age>=18)
   {
      printf("You are eligible to vote...");
   }
   else
   {
      printf("Sorry ... you can't vote");
   }
}
```

Output

```
Enter your age?18
You are eligible to vote...
Enter your age?13
```

```
Sorry ... you can't vote
```

If else-if ladder Statement

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. There are multiple else-if blocks possible. It is similar to the switch case statement where the default is executed instead of else block if none of the cases is matched.

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}

else{
//code to be executed if all the conditions are false
}
```

Flowchart of else-if ladder statement in C :

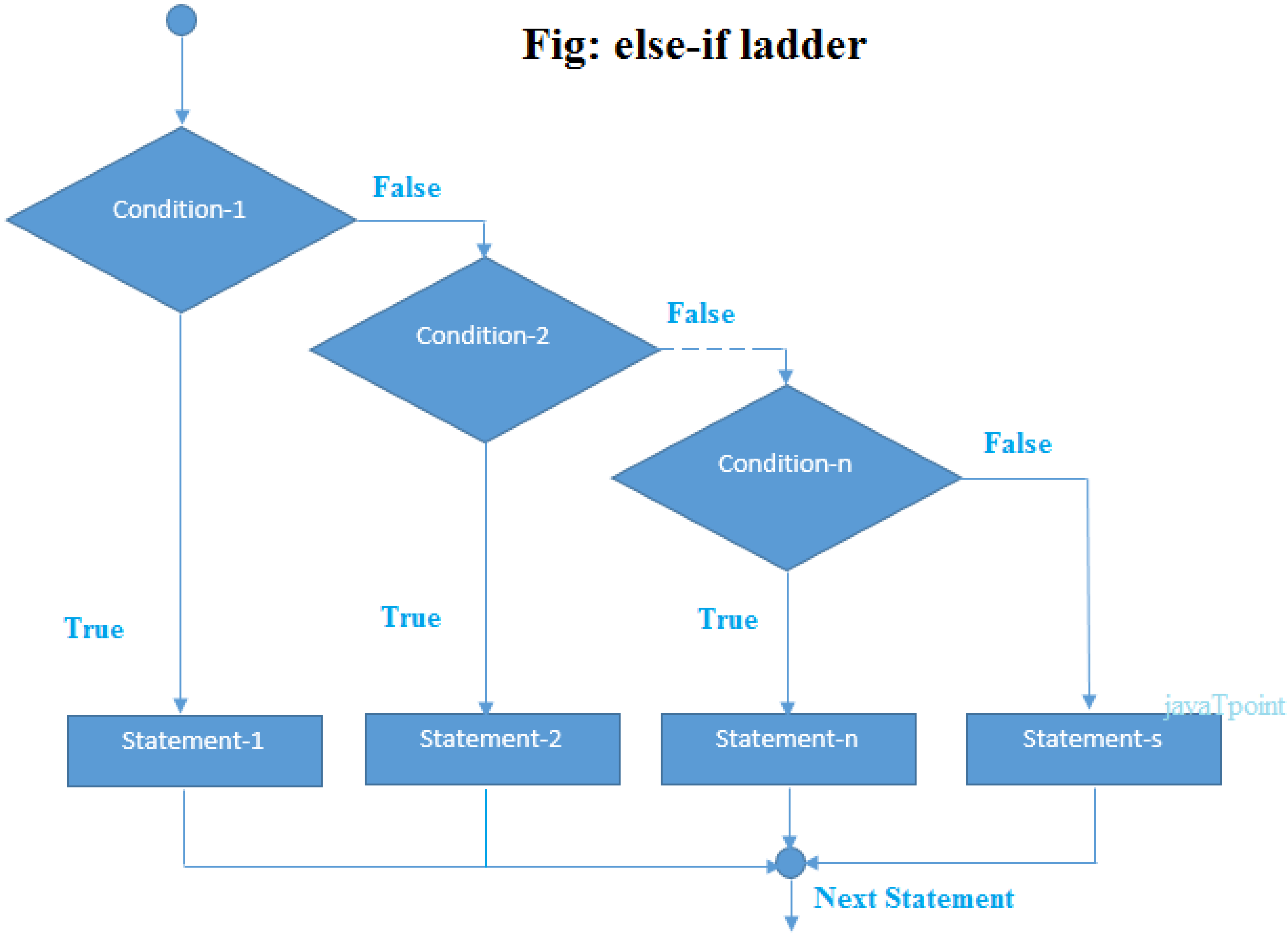


Fig: else-if ladder

The example of an if-else-if statement in C language is given below.

```
#include<stdio.h>

int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
if(number==10){
printf("number is equals to 10");
}
else if(number==50){
printf("number is equal to 50");
}
else if(number==100){
printf("number is equal to 100");
}
else{
printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

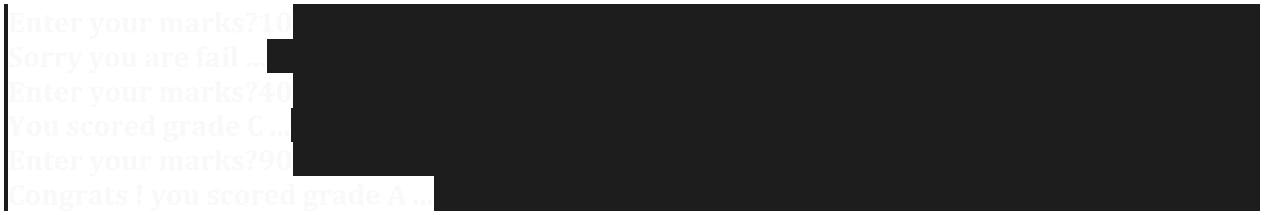
Output

```
enter a number:4
number is not equal to 10, 50 or 100
enter a number:50
number is equal to 50
```

Program to calculate the grade of the student according to the specified marks.

```
#include <stdio.h>

int main()
{
  int marks;
  printf("Enter your marks?");
  scanf("%d",&marks);
  if(marks > 85 && marks <= 100)
  {
    printf("Congrats ! you scored grade A ...");
  }
  else if (marks > 60 && marks <= 85)
  {
    printf("You scored grade B + ...");
  }
  else if (marks > 40 && marks <= 60)
  {
    printf("You scored grade B ...");
  }
else if (marks > 30 && marks <= 40)
{       printf("You scored grade C ...");
}
else
{
   printf("Sorry you are fail ...");
}
}
```

Output



```
Enter your marks?10
Sorry you are fail ...
Enter your marks?40
You scored grade C ...
Enter your marks?90
Congrats ! you scored grade A ...
```

Graphs Terminology

A graph consists of:

- A set, V, of vertices (nodes)
- A collection, E, of pairs of vertices from V called edges (arcs)

Edges, also called arcs, are represented by (u, v) and are either:

Directed if the pairs are ordered (u, v)

u the origin

v the destination

Undirected if the pairs are unordered

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and Eis the set of edges, connecting the pairs of vertices. Take a look at the following graph −

In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

Then a graph can be:

Directed graph (di-graph) if all the edges are directed

Undirected graph (graph) if all the edges are undirected

Mixed graph if edges are both directed or undirected

Illustrate terms on graphs

End-vertices of an edge are the endpoints of the edge.

Two vertices are adjacent if they are endpoints of the same edge.

An edge is incident on a vertex if the vertex is an endpoint of the edge.

Outgoing edges of a vertex are directed edges that the vertex is the origin.

Incoming edges of a vertex are directed edges that the vertex is the destination.

Degree of a vertex, v, denoted deg(v) is the number of incident edges.

Out-degree, outdeg(v), is the number of outgoing edges.

In-degree, indeg(v), is the number of incoming edges.

Parallel edges or multiple edges are edges of the same type and end-vertices

Self-loop is an edge with the end vertices the same vertex

Simple graphs have no parallel edges or self-loops

Properties

If graph, G, has m edges then $\Sigma v \in G \deg(v) = 2m$

If a di-graph, G, has m edges then

$\Sigma v \in G$ indeg(v) = m = $\Sigma v \in G$ outdeg(v)

If a simple graph, G, has m edges and n vertices:

If G is also directed then $m \leq n(n-1)$

If G is also undirected then $m \leq n(n-1)/2$

So a simple graph with n vertices has O(n

2

) edges at most

More Terminology

Path is a sequence of alternating vetches and edges such that each successive vertex is connected by the edge. Frequently only the vertices are listed especially if there are no parallel edges.

Cycle is a path that starts and end at the same vertex.

Simple path is a path with distinct vertices.

Directed path is a path of only directed edges

Directed cycle is a cycle of only directed edges.

Sub-graph is a subset of vertices and edges.

Spanning sub-graph contains all the vertices.

Connected graph has all pairs of vertices connected by at least one path.

Connected component is the maximal connected sub-graph of a unconnected graph.

Forest is a graph without cycles.

Tree is a connected forest (previous type of trees are called rooted trees, these are free trees)

Spanning tree is a spanning subgraph that is also a tree.

More Properties

If G is an undirected graph with n vertices and m edges:

- If G is connected then m ≥ n - 1
- If G is a tree then m = n - 1
- If G is a forest then m ≤ n – 1

Graph Traversal:

1. Depth First Search
2. Breadth First Search

Lecture-20

Depth First Search:

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

 Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

 Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.

Step Traversal Description

1

Initialize the stack.

2

Mark

S as visited and put it

onto the stack. Explore any

unvisited adjacent node

from

S. We have three nodes

and we can pick any of them.

For this example, we shall

take the node in an

alphabetical order.

3

Mark

A as visited and put it

onto the stack. Explore any

unvisited adjacent node from

A. Both

Sand

D are adjacent

to

A but we are concerned for

unvisited nodes only.

4

Visit

D and mark it as visited

and put onto the stack. Here,

we have

B and

C nodes,

which are adjacent to

D and

both are unvisited. However,

we shall again choose in an

alphabetical order.

5

We choose

B, mark it as

visited and put onto the stack.

Here

Bdoes not have any

unvisited adjacent node. So,

we pop

Bfrom the stack.

6

We check the stack top for

return to the previous node

and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.

7

Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.

As C does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Lecture-21

Breadth First Search

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

 Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

 Rule 2 – If no adjacent vertex is found, remove the first vertex from the queue.

 Rule 3 – Repeat Rule 1 and Rule 2 until the queue is empty.

Step Traversal Description

1

Initialize the queue.

2

We start from visiting S(starting node), and mark it as visited.

3

We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.

4

Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.

5

Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.

6

Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.

7

From A we have D as unvisited adjacent node. We mark it as visited and enqueue

it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Lecture-22

Graph representation

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

Adjacency matrix

An adjacency matrix is a VxV binary matrix A. Element Ai,j is 1 if there is an edge from vertex i to vertex j else Ai,jis 0.

Note: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in Ai,j, the weight or cost of the edge will be stored.

In an undirected graph, if Ai,j = 1, then Aj,i = 1. In a directed graph, if Ai,j = 1, then Aj,i may or may not be 1.

Adjacency matrix provides constant time access (O(1)) to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is O(V2).

The adjacency matrix of the following graph is:

i/j : 1 2 3 4

1 : 0 1 0 1

2 : 1 0 1 0

3 : 0 1 0 1

4 : 1 0 1 0

The adjacency matrix of the following graph is:

i/j: 1 2 3 4

1 : 0 1 0 0

2 : 0 0 0 1

3 : 1 0 0 1

4 : 0 1 0 0

Adjacency list

The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array Ai

is a list, which contains all the

vertices that are adjacent to vertex i.

For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list Ai then vertex i will be in list Aj.

The space complexity of adjacency list is O(V + E) because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.

Note: A sparse matrix is a matrix in which most of the elements are zero, whereas a dense matrix is a matrix in which most of the elements are non-zero.

Consider the same undirected graph from an adjacency matrix. The adjacency list of the graph is as follows:

A1 → 2 → 4

A2 → 1 → 3

A3 → 2 → 4

A4 → 1 → 3

Consider the same directed graph from an adjacency matrix. The adjacency list of the graph is as follows:

A1 → 2

A2 → 4

A3 → 1 → 4

A4 → 2

Lecture-23

Topological Sorting:

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).

Algorithm to find Topological Sorting:

In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Topological Sorting vs Depth First Traversal (DFS):

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting

Dynamic Programming

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

Input:

```
 graph[][] = { {0, 5, INF, 10},
 {INF, 0, 3, INF},
 {INF, INF, 0, 1},
 {INF, INF, INF, 0} }
```

which represents the following graph

```
 10
 (0)------->(3)
 | /|\
 5 | |
 | | 1
 \|/ |
 (1)------->(2)
 3
```

Note that the value of graph[i][j] is 0 if i is equal to j

And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

Output:

Shortest distance matrix

```
 0 5 8 9
 INF 0 3 4
 INF INF 0 1
 INF INF INF 0
```

Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

The following figure shows the above optimal substructure prop

UNIT-II

Pointer Variables – Pointer Operators - Pointer Expressions – Pointers And Arrays – Multiple Indirection – Initializing Pointers – Pointers to Functions – C‟s Dynamic Allocation Functions –

Problems with Pointers.

Understanding the scope of Functions – Scope Rules – Type Qualifiers – Storage Class Specifiers-

Functions Arguments –The Return Statement.

POINTERS:

- Pointer is a variable which holds address of anothervariable of same data-type.
- The size of a pointer depends on the amount of memory to be addressed.
- By default pointer size is 4 bytes
- Pointer is used to access memory and manipulate addresses.
- A pointer comes under Derived data type.

Advantages of Pointers

a) Pointers are efficient in handling data and associated with array.

b) Pointers are used for saving memory space.

c) Pointers reduce length and complexity of the program

d) Use of pointers assigns the memory space and also releases it. It helps to make better use of the available memory (dynamic memory allocation).

e) Since the pointer data manipulation is done with address, the execution time is faster

f) The two dimensional and multi-dimensional array representation is easy in pointers.

g) Pointer allows for references to function, this may facilitating passing of function as arguments to other functions.

h) Pointers are more compact and efficient code

i) Pointers can be used to achieve clarity and simplicity

j) Pointers are used to pass information between function and its reference point

k) Pointers provides a way to return multiple data items from a function using its function arguments

l) Pointers also provide an alternative way to access an array element

m) Pointers enables us to access the memory directly.

Pointer Variables:

- As we know that, each variable has two attributes: address and value.
- A variable can take any value specified by its data type.
  For example, if the variable i is of the integer type, it can take any value permitted in therange specified by the integer data type.
- A pointer to an integer is a variable that can store the address of that integer.

Consider the declaration,

inti = 3 ;

This declaration tells the C compiler to:
(a) Reserve space in memory to hold the integer value.
(b) Associate the name i with this memory location.
(c) Store the value 3 at this location.

We can print this address number through the following program:

```
void main( )
{
inti = 3 ;
printf ( "\nAddress of i = %u", &i ) ; // “address of” operator
printf ( "\nValue of i = %d", i ) ;
printf(“\nValue of i=%d”,*(&i)); // “value at address” operator
}
```

Expected Output

```
Address of i = 65524
Value of i = 3
Value of i=3
```

<u>Declaring and Initializing pointers:</u>

Syntax:

<datatype> *<pointer name>

For Ex:

```
int *ptr;         /* pointer to int */
char *s;          /* pointer to char */
float *fp;        /* pointer to float */
char **s;         /* pointer to variable that is a pointer to char */
```

Initializing Pointers

- After a non static, local pointer is declared but before it has been assigned a value, it contains an unknown value. (Global and static local pointers are automatically initialized to null.) Should you try to use the pointer before giving it a valid value, you will probably crash your program— and possibly your computer's operating system as well— a very nasty type of error!
- There is an important convention that most C programmers follow when working withpointers: A pointer that does not currently point to a valid memory location is given the valuenull (which is zero).
- Null is used because C guarantees that no object will exist at the null address.
- Thus, any pointer that is null implies that it points to nothing and should not be used.One way to give a pointer a null value is to assign zero to it. For example, thefollowing initializes **p** to null.

```
char *p = 0;
```

Additionally, many of C's headers, such as **<stdio.h>**, define the macro **NULL**, which is a null pointer constant.

Therefore, you will often see a pointer assigned null using a statement such as this:

```
p = NULL;
```

Note: Always the Size of a pointer is 4 Bytes (default) because irrespective of the data type of the pointer , it occupies 4 bytes only as they are holding some unsigned integers called addresses.

```
int *a;
float *b;
char *c;
printf("%d %d %d",sizeof(a),sizeof(b),sizeof(c)); prints 4 4 4
```

Q. What is NULL pointer explain it?

NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

Important Points

1. **NULL vs Uninitialized pointer –** An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.
2. **NULL vs Void Pointer** – Null pointer is a value, while void pointer is a type

```
#include <stdio.h>
int main()
{
        // Null Pointer
        int *ptr = NULL;

        printf("The value of ptr is %p", ptr);
        return 0;
}
```

Output:

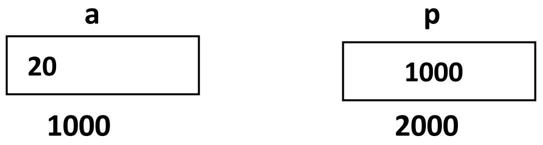
```
The value of ptr is (nil)
```

Q. Wild pointer in C?

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

```
Ex:main()
{
        int *p /* wild pointer */
        int a = 20;
        int *p = &a;// p is not a wild pointer now
        printf("%d%d%d%d",p,*p,*&p,&a);
    }
```

output: 1000 20 1000 1000.



Q. What is void pointer?

Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.

Important Points

1. void pointers **cannot be dereferenced**. It can however be done using typecasting the void pointer
2. Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

```
#include<stdlib.h>

int main()
{
        int x = 4;
        float y = 5.5;

        //A void pointer
        void *ptr;
        ptr = &x;

        // (int*)ptr - does type casting of void
        // *((int*)ptr) dereferences the typecasted
        // void pointer variable.
        printf("Integer variable is = %d", *( (int*) ptr) );

        // void pointer is now float
        ptr = &y;
        printf("\nFloat variable is= %f", *( (float*) ptr) );

        return 0;
}
```

Output:

```
Integer variable is  =  4

Float variable is= 5.500000
```

Q. Dangling pointer in C?

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are **three** different ways where Pointer acts as dangling pointer

1. **De-allocation of memory**

```
// Deallocating a memory pointed by ptr causes
// dangling pointer
#include <stdlib.h>
#include  <stdio.h>
int main()
{
    int *ptr = (int *)malloc(sizeof(int));

    // After below free call, ptr becomes a
    // dangling pointer
    free(ptr);

    // No more a dangling pointer
    ptr = NULL;
}
```

2. **Function Call**

```
// The pointer pointing to local variable becomes
// dangling when local variable is not static.
#include<stdio.h>

int *fun()
{
    // x is local variable and goes out of
    // scope after an execution of fun() is
    // over.
    int x = 5;

    return&x;
}

// Driver Code
int main()
{
    int *p = fun();
    fflush(stdin);

    // p points to something which is not
    // valid anymore
    printf("%d", *p);
    return 0;
}
```

Output: A garbage address

The above problem doesn’t appear (or p doesn’t become dangling) if x is a static variable.

```
// The pointer pointing to local variable doesn't
// become dangling when local variable is static.
#include<stdio.h>

int *fun()
{
    // x now has scope throughout the program
    static int x = 5;

    return&x;
}

int main()
{
    int *p = fun();
    fflush(stdin);

    // Not a dangling pointer as it points
    // to static variable.
    printf("%d",*p);
}
```

Output: 5

3. **Variable goes out of scope**

```
void main()
{
int *ptr;
  .....
  .....
  {
intch;
ptr = &ch;
  }
  .....
  // Here ptr is dangling pointer
}
```

Q. Explain about pointer operators? Or what are the different types of operators used for pointers representation

Pointer Operators:

- Look at the first **printf( )** statement carefully. **'&'**used in this statement is C's**'addressof'**operator. The expression **&i**returns the address of the variable **i**, which in this case happens to be 65524. We have been using the „&" operator all the time in the **scanf( )**statement.
- The other pointer operator available in C is **'*'**, called **'value at address'**operator. It gives the value stored at a particular address. The „value at address" operator is also called **'indirection'**operator.

Pointers are associated with two operators &,*.

& - address of operator / referencing operator

***- value at address operator / Dereferencing/indirection Operator.**

Consider the Statement

```
int *p;
intx=5;
  p=&x;
```

- Here '&' is called address of a variable. 'p' contains the address of a variable x.
- The operator &returns the memory address of variable on which it is operated, this is called Referencing.
- The * operator is called an indirection operator or dereferencing operator which issued to display the contents of the Pointer Variable.

Assume that x is stored at the memory address 2000. Then the output for thefollowing printf statements is :

Output

```
Printf("The Value of x is %d",x);       5
Printf("The Address of x is %u",&x); 2000
Printf("The Address of x is %u",p); 2000
Printf("The Value of x is %d",*p);       5
Printf("The Value of x is %d",*(&x));       5
```

Ex2:

```
main()
 {
void *a;
int n=2,*m;
double d=2.3,*c;
  a=&n;
  m=a;
printf("\n%d %d %d",a,*m,m);
  a=&d;
  c=a;
printf("\n%d %3.1f %d",a,*c,c);
 }
```

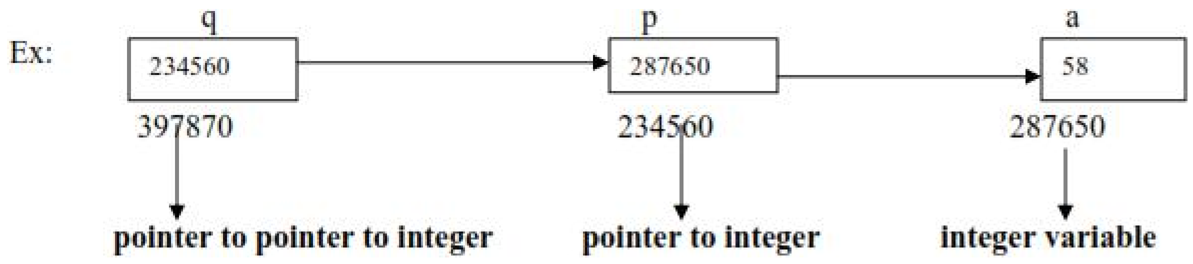
In the above program a is declared as a pointer to void which is used to carry the address of an int(a=&n)and to carry the address of a double(a=&d) and the original pointers are recovered with out any loss of information.

Q. What is multiple pointers or multi pointers or pointer to pointers

Multiple Indirection(Pointers To Pointers):

- Pointer is a variable that contains the address of the another variable. Similarly another pointer variable can store the address of this pointer variable, So we can say, this is a pointer to pointer variable.
- Pointers to Pointers– also considered as chain of Pointers. Because A single pointer can store the address of a variable.
- A double pointer can store the address of single pointer
- A Triple Pointer can store the address of double pointer.
- For example,we can have a pointer pointing to a pointer to an integer.This two levelindirection is seen as below:

```
 //Local declarations
int a;
int* p;
int **q;
```



```
//statements
a=58;
p=&a;
q=&p;
printf("%3d",a);
printf("%3d",*p);
printf("%3d",**q);
```

- There is no limit as to how many level of indirection we can use but practicallywe seldom use morethantwo.Each level of pointer indirection requires a separateindirection operator when it is dereferences .

- In the above figure to refer to 'a' using the pointer 'p', we have to dereference it as shown below.

  *p

- To refer to the variable 'a' using the pointer 'q' ,we have to dereference it twicetoget to the integer 'a' because there are two levels of indirection(pointers) involved.
- If wedereference it only once we are referring 'p' which is a pointer to an integer .Anotherway to say this is that 'q' is a pointer to a pointer to an integer.Thedouble dereference isshown below:

  **q

- In above example all the three references in the printf statement refer to the variable 'a'.The first printf statement prints the value of the variable 'a' directly,second uses thepointer 'p',third uses the pointer 'q'.The result is the value 58 printed 3 times as below

```
58        58         58
```

Ex2:

```
int a=10,*b,**c,***d;
     b = &a;
     c = &b;
     d = &c;
printf ("%d %d %d %d",a,*b,**c,***d);
```

prints 10 10 10 10

Q. Explain about pointer expressions ?

Pointer Expressions:

In general, expressions involving pointers conform to the same rules as other expressions. This section examines a few special aspects of pointer expressions, such as assignments,conversions and arithmetic.

1. **Pointer Assignments**
2. **Pointer Conversions**
3. **Pointer Arithmetic**
4. **Address Arithmetic**

(i) Pointer Assignments

You can use a pointer on the right-hand side of an assignment statement to assign its value to
another pointer. When both pointers are the same type, the situation is straightforward.

For example:

```
#include <stdio.h>
#include<conio.h>
void main()
{
int x = 99;
int *p1, *p2;
p1 = &x;
p2 = p1;
/* print the value of x twice */
printf("Values at p1 and p2: %d %
d\n", *p1, *p2);
/* print the address of x twice */
printf("Addresses pointed to by p1 and p2: %u %u", p1, p2);
getch();
}
```

After the assignment sequence

```
p1 = &x;
p2 = p1;
```

p1and **p2** both point to **x**.

Thus, both **p1** and **p2** refer to the same object.

Expected Output

```
Values at p1 and p2: 99 99
Addresses pointed to by p1 and p2: 65524 65524
```

⍰⍰Notice that the addresses are displayed by using the **%u printf( )** format specifier,which causes **printf( )** to display an address in the format used by the host computer.

⍰⍰It is also possible to assign a pointer of one type to a pointer of another type.
However, doing so involves a pointer conversion, which is the subject of the next section.

(ii) Pointer Conversions

One type of pointer can be converted into another type of pointer.

In C, it is permissible to assign a **void** * pointer to any other type of pointer.

It is also permissible to assign any other type of pointer to a **void** * pointer.

Generic Pointers:A void Pointer is also called as a Generic Pointer, which can store the address of any variable .

```
Ex: -  int a;
          float b;
          char c;
          void *ptr;  // void pointer
          ptr = &a;
```

```
            *(int *) ptr = 10;
            ptr = &b;
        *(float *) ptr = 3.14;
            ptr = &c;
            *(char *) ptr= ‘s’;
Printf(“%d  %f  %c”,a,b,c); prints 10 3.14 s
```

Through void pointer values can be assigned through proper type casting

(iii) Pointer Arithmetic

Like normal variables, pointers can also be used to perform various arithmetic operations like
addition, subtraction etc.

//Program to show the use Arithmetic operations on pointers

```
#include<stdio.h>
#include<conio.h>
void main()
{
inta,b,*res,*ptr1,*ptr2;
clrscr();
printf("\nEnter any two numbers:");
scanf("%d%d",&a,&b);
ptr1=&a;
ptr2=&b;
*res=*ptr1+*ptr2; // addition
printf("\nSum=%d",*res);
*res=*ptr1-*ptr2; // subtraction
printf("\nDifference=%d",*res);
*res=*ptr1**ptr2; // multiplication
printf("\nProduct=%d",*res);
*res=*ptr1/(*ptr2); // division
printf("\nDivision=%d",*res);
*res=*ptr1%*ptr2; // mod
printf("\nRemainder=%d",*res);
getch();
}
```

Expected Output

```
Enter any two numbers:
20
10
Sum=30
Difference=10
```

Product=200
Division=2
Remainder=0

Q. Explain what are the operations performed on pointers?

Q. Explain Valid and invalid operations performed on pointers

4 Address Arithmetic :

<u>Valid operations that can we do with pointers are</u>

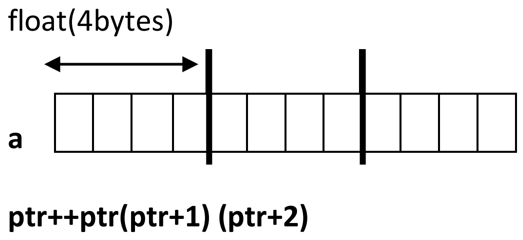
- we can increment/decrement a pointer
- we can add or subtract a integer value to/from a pointer
- we can subtract two pointer, which tells the number of elements in between these two addresses.

Operator	**Result**
++	Goes to thenextmemorylocation that the pointerispointingto.
--	Goes to thepreviousmemorylocation that the pointeris
-=or -	Subtracts value from pointer.
+=or+	Addingtothepointer

1. Incrementing/Decrementing a Pointer

Consider a float variable (a) and a pointer to a float (ptr) as shown below

float a=2.3;
float *ptr=&a;



Pointer Arithmetic Assume that **ptr** points to **a** then if we increment the pointer (++ptr) it moves to the position shown 4 bytes on. If on the other hand we added 2 to the pointer then it moves 2 **float positions***i.e* 8 bytes as shown in the Figure.

Ex2:

```
#include<stdio.h>
        main()
        {       inta,b,c,d,e,f,*p,*q;
                clrscr();
                printf("Enter a,b values:");
                scanf("%d%d",&a,&b);
                p=&a;
                q=&b;
                c=--q;
                d=q--;
                e=++p;
                f=p++;
                printf("\n pre decrement=%d", c);
                printf("\n post decrement=%d", d);
                printf("\n pre increment=%d", e);
                printf("\n post increment=%d", f);
                getch();
        }
```

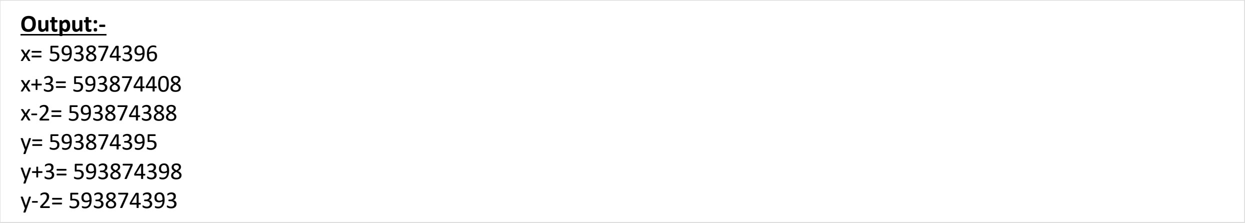
2. Addition/Subtraction of a constant number to a pointer

Addition or **subtraction** of a constant number to a pointer is allowed. The result is similar to the increment or decrement operator with the only difference being the increase or decrease in the memory location by the constant number given.

Also, not to forget the values get incremented or decremented according to the type of variable it stores.

```
#include <stdio.h>
int main()
{
  //declaring the pointer for integer variable
int a = 5, *x;

  //declaring the pointer for char variable
char b = 'z', *y;

  //storing the memory location of variable a in pointer variable x
  x = &a;

  /*The corresponding values of the addition and subtraction operations on pointer variable x are
given below*/
  //printing the actual value of x
printf("x= %d\n", x);

  //the value incremented by 3
printf("x+3= %d\n", x + 3);

  //the value decremented by 2
printf("x-2= %d\n", x - 2);

  //storing the memory location of variable b in pointer variable y
  y = &b;
  /*The corresponding values of the addition and subtraction operations on pointer variable y are
given below*/
  //printing the actual value of y
printf("y= %d\n", y);

  //the value incremented by 3
printf("y+3= %d\n", y + 3);

  //the value decremented by 2
printf("y-2= %d\n", y - 2);
return 0;
}
```

Output:



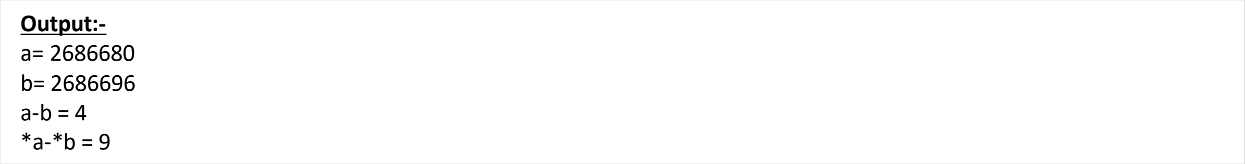
```
Output:-
x= 593874396
x+3= 593874408
x-2= 593874388
y= 593874395
y+3= 593874398
y-2= 593874393
```

3. Subtraction of one pointer from another

A **pointer** variable can be subtracted from another pointer variable only if they point to the elements of the same array. Also, subtraction of one pointer from another pointer that points to the elements of the same array gives the number of elements between the array elements that are indicated by the pointer.

```
#include <stdio.h>
int main()
{
int num[10] = {1, 5, 9, 4, 8, 3, 0, 2, 6, 7}, *a, *b;

   //storing the address of num[2] in variable a
   a = &num[2];

   //storing the address of num[6] in variable b
   b = &num[6];

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

   //prints the number of elements between the two elements indicated by the pointers
printf("a-b = %d\n", b - a);

   //prints the difference in value of the two elements
printf("*a-*b = %d\n", *a - *b);
return 0;
}
```

Output:



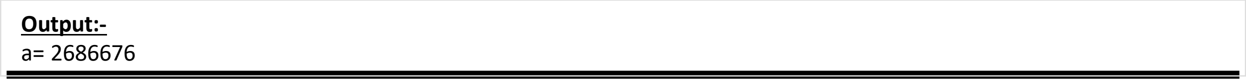
```
Output:-
a= 2686680
b= 2686696
a-b = 4
*a-*b = 9
```

4. Comparison of two pointers

Comparison of two pointer variables is possible only if the two pointer variables are of the same type. It becomes more convenient if they point to the elements of the same array. These comparisons are to check *equality* or *inequality*. The result is *true* if both the pointers point to the same location in the memory and *false* if they point to different locations in the memory.

```
#include <stdio.h>
int main()
{
int num[10] = {1, 5, 9, 4, 8, 3, 0, 2, 6, 7}, *a, *b, *c;

   //storing the address of num[2] in variable a
   a = &num[2];

   //base address plus 2 stores the address of num[2] in the variable b
   b = (num + 2);

   //storing the address of num[6] in variable b
   c = &num[6];

   //Print values of all the pointers
printf("a= %d\n", a);
printf("b= %d\n", b);
printf("c= %d\n", c);

   //comparing for equality
if (a == b)
printf("a and b point to the same location and the value is: %d\n", *a);

   //comparing for inequality
if (a != c)
printf("a and c do not point to the same location in the memory");
return 0;
}
```

Output:



```
Output:-
a= 2686676
```

```
b= 2686676
c= 2686692
a and b point to the same location and the value is: 9
a and c do not point to the same location in the memory
```

Invalid Operations with Pointers.

- **Two Pointers cannot be added, multiplied, divided**
- **A Pointer cannot be adding double, float values**
- **A Pointer cannot be multiplied by a integer.**
- **A Pointer cannot be divided by a integer.**
- **Masking or shifting of pointers**
- **Assigning a pointer of one to another type of pointer**
- Modulo operation on pointer.
- Cannot perform bitwise AND,OR,XOR operations on pointer.
- Cannot perform NOT operation or negation operation.

Q. Explain where we can use pointers ?

C uses pointers explicitly with:

- **Functions.**
- **Arrays,**
- **Structures,**

Pointers and Functions:

Parameter passing mechanism in 'C' is of two types.

1. Call by Value

2. Call by Reference.

- The process of passing the actual value of variables is known as Call by Value.The process of calling a function using pointers to pass the addresses of variablesis known as Call by Reference.The function which is called by reference can changethe value of the variable used in the call.

Comparison between call-by-value and call-by-reference

S.NO	Call-by-value(normal variables)	Call-by-reference(pointer variables)
1	Different memory locations are occupied by formal and actual arguments	Same memory location occupied by formal and actual arguments, so there is a saving of memory location.
2	Any alteration in the value of the arguments passed is local to the function and is not accepted in the calling program	Any alteration in the value of the arguments passed is accepted in the calling program
3	The usual method to call a function in which only the value of the variable is passed as an argument	The address of the variable is passed as an argument

4	When a new location is created it is very slow.	The existing memory location is used through its address, it is very fast.
5	There is no possibility of wrong data manipulation since the argument are directly used in an expression	There is a possibility of wrong data manipulation since the addresses are used in an expression. A good skill of programming is required.

Example of Call by Reference:

```
#include<stdio.h>
main()
 {
inta,b;
  a = 10;
  b = 20;
swap (&a, &b);
printf("After Swapping \n");
printf("a = %d \t b = %d", a,b);
 }
void swap(int *x, int *y)
  {
int temp;
temp = *x;
   *x = *y;
   *y = temp;
  }
```

Function Pointers:

Ex:

```
int add(intx,int y);
main()
{
int x=6,y=9;
int(*ptr)(int,int);  //pointer to function
ptr=add;
printf("%dplus%dequals%d\n",x,y,(*ptr)(x,y));
return 0;
}
int add(intx,int y)
{
returnx+y;
}
```

Pointers and Arrays :

- When an array is declared, elements of array are stored in contiguous locations.
- The address of the first element of an array is called its base address.

Consider the array

2000	2002	2004	2006	2008
a[0]	a[1]	a[2]	a[3]	a[4]

- The name of the array is called its base address. i.e., a and k&a[20] are equal.
- Now both a and a[0] points to location 2000. If we declare p as an integer pointer, thenwe can make the pointer P to point to the array a by following assignment

p = a;

We can access every value of array a by moving P from one element to another.

i.e., P points to 0th element
P+1 points to 1st element
P+2 points to 2nd element
P+3 points to 3rd element
P +4 points to 4th element

Ex1:Single dimensional array through pointers

```
main()
{
int a[]={1,2,3};
inti,*p=a; //p=a;
for(i=0;i<5;++i)
printf("%d",*(p+i));
}prints 1 2 3
```

Two Dimensional array through pointers

```
main()
{
inti,j,a[3][3]={1,2,3,4,5,6,7,8,9};
for(i=0;i<3;++i,printf("\n"))
for(j=0;j<3;++j)
printf("%d",*(*(p+i)+j));         //p++
}
o/p: 1 2 3
     4 5 6
     7 8 9
```

- **In one dimensional array, a[i] element is referred by (a+i) is the address of ith element. * (a+i) is the value at the I element.**
- **In two-dimensional array, a[i][j] element is represented as *(*(a+i)+j)**

- **<u>Note:</u> a[i][j][k][l] can be represented as *(*(*(*(a+i)+j)+k)+l)**

<u>Pointers and Structures</u>

These are fairly straight forward and are easily defined. Consider the following:

```
struct COORD {float x,y,z;} pt;
struct COORD *pt_ptr;
pt_ptr = &pt; /* assigns pointer to pt */
```

the**->**operator lets us access a member of the structure pointed to by a pointer.*i.e.*:

```
pt_ptr-> x = 1.0;
pt_ptr-> y = - 3.0;
```

Ex: Linked Lists

```
typedefstruct {  int value;
                  ELEMENT *next;
                 } ELEMENT;
                 ELEMENT n1, n2;
                 n1.next = &n2;
```

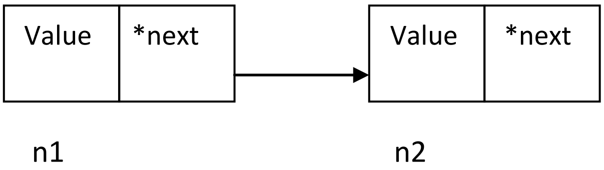


Fig. Linking Two Nodes

NOTE: We can only declare next as a pointer to ELEMENT. We cannot have a element of the variable type as this would set up a ***recursive*** definition which is **NOT ALLOWED**. We are allowed to set a pointer reference since 4 bytes are set aside for any pointer.

Q. explain what are the problems with pointers

Problems with Pointers

- Nothing will get you into more trouble than a wild pointer! Pointers are a mixedblessing. They give you tremendous power, but when a pointer is used incorrectly, orcontains the wrong value, it can be a very difficult bug to find.
- An erroneous pointer is difficult to find because the pointer, by itself, is not theproblem. The trouble starts when you access an object through that pointer. In short, whenyou attempt to use a bad pointer, you are reading or writing to some unknown piece ofmemory. If you read from it, you will get a garbage value, which will probably cause yourprogram to malfunction. If you write to it, you might be writing over other pieces of yourcode or data. In either case, the problem might not

show up until later in the execution ofyour program and may lead you to look for the bug in the wrong place. There may be little orno evidence to suggest that the pointer is the original cause of the problem. Programmers losesleep over this type of bug time and time again.

- Because pointer errors are so troublesome, you should, of course, do your best neverto generate one. To help you avoid them, a few of the more common errors are discussedhere.

(i) The classic example of a pointer error is the *uninitialized pointer*.

Consider this program:

```
/* This program is wrong. */
void main()
{
int x, *p;
x = 10;
*p = x; /* error, p not initialized */
}
```

Eventually, your program stops working. In this simple example, most compilers will issue a warning.

(ii) A second common error is caused by a simple misunderstanding of how to use a pointer.

Consider the following:

```
/* This program is wrong. */
#include <stdio.h>
void main()
{
int x, *p;
x = 10;
p = x;
printf("%d", *p);
}
```

- The call to **printf( )** does not print the value of **x**, which is 10, on the screen. It prints some unknown value because the assignment p = x; is wrong. That statement assigns the value 10 to the pointer **p**. However, **p** is supposed to contain an address, not a value. To correct the program, write p = &x;
- As with the earlier error, most compilers will issue at least a warning message when you attempt to assign **x** to **p**.

Dynamic Memory Allocation:

- Dynamic memory allocation uses predefined functions to allocate and releasememory for data while the program is running. It effectively postpones the data definition,but not the declaration to run time.

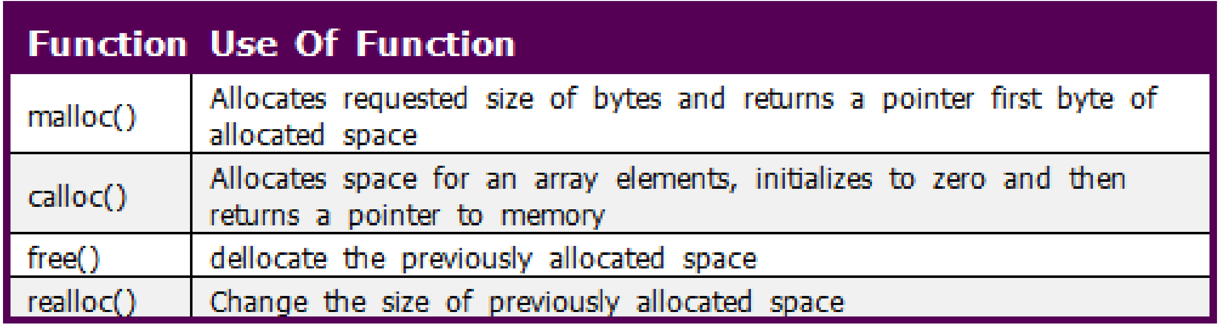
- To use dynamic memory allocation ,we use either standard data types or derivedtypes .To access data in dynamic memory we need pointers.

Advantages of Dynamic memory allocation

- It has the ability to reserve or allocate additional memory space during the program execution.
- It has the ability to release unwanted memory space during the program execution
- It is very useful to modify the size of the previously allocated memory.
- It is very useful to allocate memory space to an array of elements and initialize them to zero.

Memory Allocation Functions:

- Four memory management functions are used with dynamic memory. Three of them malloc,calloc,andreallocare used for memory allocation. The fourth, free is used toreturn memory when it is no longer needed.
- All the memory management functions arefound in standard library file(stdlib.h).



Function	Use Of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

Function	Syntax
malloc()	(datatype *) malloc (number *sizeof(int));
calloc()	(datatype *) calloc (number, sizeof(int));
realloc()	(datatype *) realloc (pointer_name, number * sizeof(int));
free()	free (pointer_name);

Block Memory Allocation (malloc) :

- malloc() is used to allocate memory space in bytes for variables of any valid C data type.
- It returns starting address of the allocated bytes.
- If sufficient memory is not available, then it returns the NULL value.
- It allocates a block a size bytes of consecutive memory from the heap of memory.

Declaration:

Syntax : -

pointer= (data_type*)malloc(number *sizeof(int));

int *p=(int*)malloc(6*sizeof(int);

Example:

```
#include  <stdio.h>
#include <stdlib.h>
int main()
{
        // This pointer will hold the
        // base address of the block created
        int* ptr;
        int n, i;

        // Get the number of elements for the array
        n = 5;
        printf("Enter number of elements: %d\n", n);

        // Dynamically allocate memory using malloc()
        ptr = (int*)malloc(n * sizeof(int));

        // Check if the memory has been successfully
        // allocated by malloc or not
        if (ptr == NULL) {
                printf("Memory not allocated.\n");
                exit(0);
        }
        else {

                // Memory has been successfully allocated
                printf("Memory successfully allocated using malloc.\n");

                // Get the elements of the array
                for (i = 0; i< n; ++i) {
                        ptr[i] = i + 1;
                }

                // Print the elements of the array
                printf("The elements of the array are: ");
                for (i = 0; i< n; ++i) {
                        printf("%d, ", ptr[i]);
                }
        }

        return 0;
}
```

Output:

```
Enter number of elements: 5
```

```
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5
```

Contagious Memory Allocation(calloc) :

“calloc” or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

(datatype *) calloc (number, sizeof(int));

Example:

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

int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

Output:

```
Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5
```

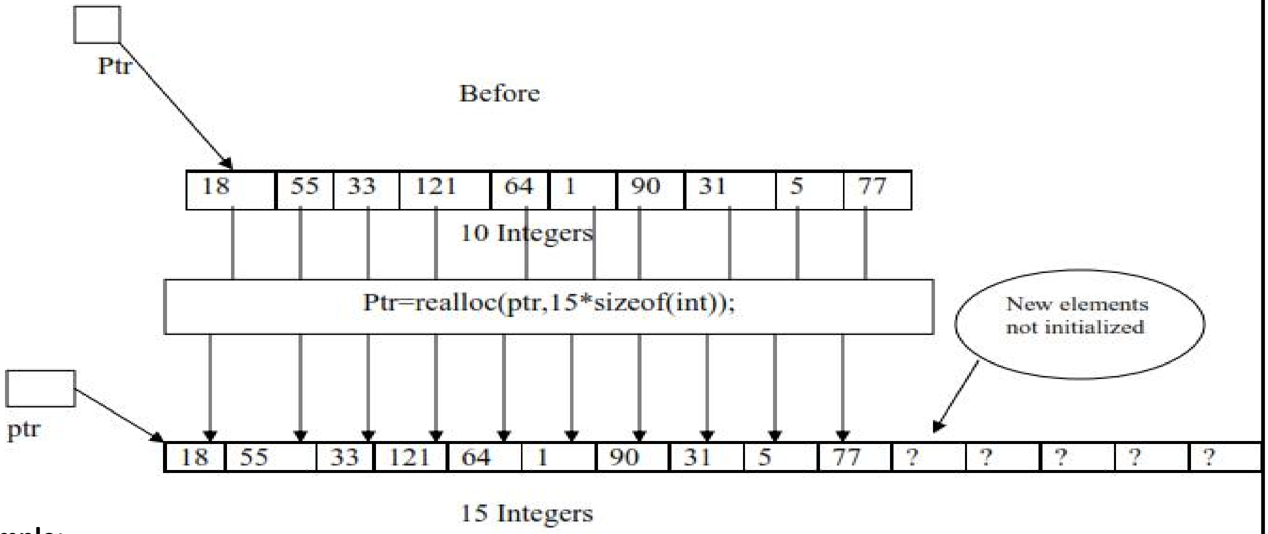
Reallocation Of Memory(realloc):

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

Syntax:

(datatype *) realloc (pointer_name, number * sizeof(int));

Ex: Ptr=realloc(ptr,15*sizeof(int);



Example:

```
#include  <stdio.h>
#include <stdlib.h>
int main()
{
        // This pointer will hold the
        // base address of the block created
        int* ptr;
```

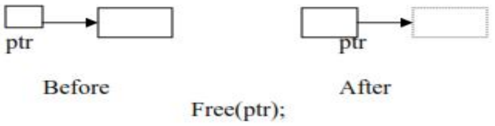
```
int n, i;

// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
}
else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i< n; ++i) {
                ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i< n; ++i) {
                printf("%d, ", ptr[i]);
        }

        // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the array: %d\n", n);

        // Dynamically re-allocate memory using realloc()
        ptr = realloc(ptr, n * sizeof(int));

        // Memory has been successfully allocated
        printf("Memory successfully re-allocated using realloc.\n");

        // Get the new elements of the array
        for (i = 5; i< n; ++i) {
                ptr[i] = i + 1;
        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i< n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }

    return 0;
}
```

output:

```
Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5

Enter the new size of the array: 10
Memory successfully re-allocated using realloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

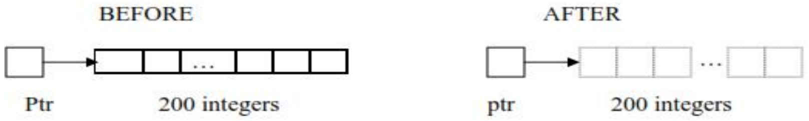
Releasing Memory(free):

- free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.
  void free(void *ptr);



Free(ptr);

In above example it releases a single element to allocated with a malloc,back to heap.



Free(ptr);

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Free the memory
        free(ptr);
        printf("Malloc Memory successfully freed.\n");

        // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");

        // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }

    return 0;
}
```

Output:

```
Enter number of elements: 5
Memory successfully allocated using malloc.
Malloc Memory successfully freed.

Memory successfully allocated using calloc.
```

Calloc Memory successfully freed.

Difference between static memory allocation and dynamic memory allocation in C:

S.no	Static memory allocation	Dynamic memory allocation
1	In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
2	Memory size can’t be modified while execution. Example: array	Memory size can be modified while execution. Example: Linked list

Difference between malloc() and calloc() functions in C:

S.no	malloc()	calloc()
1	It allocates only single block of requested memory	It allocates multiple blocks of requested memory
2	int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
3	malloc () doesn’t initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
4	type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));

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.

The array is indexed from 0 to (size-1)

Properties of array

The array contains the following properties.

Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.

Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.

Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantage of c array

1) Code optimization: less code to the access the data.

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

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

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

Disadvantage of c array

1) fixed size: whatever size, we define at the time of declaration of the array, we can't exceed the limit. so, it doesn't grow the size dynamically like linkedlist which we will learn later.

Declaration of c array

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

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

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

int marks[5];

Index →	**0**	**1**	**2**	**3**	**4**
Values →	**Garbage value**	**Garbage value**	**Garbage value**	**Garbage value**	**Garbage value**
Individual element →	**Marks[0]**	**Marks[1]**	**Marks[2]**	**Marks[3]**	**Marks[4]**

Address Locations→	**80820**	**80822**	**80824**	**80826**	**80828**

Array Memory Mapping

5 → means array size or subscript number or Dimension number
Marks → array name

int a[10];// an array of ten integers; a[0], a[1], …, a[9]
char name[20];// an array of 20 characters
floatnums[50];// an array of fifty floating numbers; nums[0], nums[1], …,nums[49]
int c[];// an array of an unknown number of integers; c[0], c[1], …, c[size-1]
int table[5][10];// a two dimensional array of integers

Initialization of c array

1. **Static type initialization**
2. **Runtime initialization**

1. **Static initialization: method1**

We manually allocate values to the array variables is called as static initialization

int marks[5];
The simplest way to initialize an array is by using the index of each element. we can initialize each element of the array by using the index. consider the following example.
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;

Index →	**0**	**1**	**2**	**3**	**4**	
Values →	**80**	**60**	**70**	**85**	**75**	
Individual element →	**Marks[0]**	**Marks[1]**	**Marks[2]**	**Marks[3]**	**Marks[4]**	
Address Locations →	**80820**	**80822**	**80824**	**80826**	**80828**	

Array Memory Mapping

Printing array values:	**By using loops:**

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

```
int i=0;
int marks[5];//declaration of array
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
Printf("%4dt",marks[0]);
Printf("%4d",marks[1]);
Printf("%4d",marks[2]);
Printf("%4d",marks[3]);
Printf("%4d",marks[4]);
Return 0;
}
Output: 80   60   70   85   75
```

```
int i=0;
int marks[5];//declaration of array
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
//traversal of array
for(i=0;i<5;i++){
printf("%4d \n",marks[i]);
}//end of for loop
return 0;
}
Output: 80   60   70   85   75
```

Static initialization: method2:
declaration with initialization

we can initialize the c array at the time of declaration. let's see the code.

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

Static initialization: method3:

we can initialize the c array at the time of declaration

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

in this declaration array size is 5 but we initialize only 3 values so, the remaining values are initialized with '0' (zeros).

Index →	0	1	2	3	4
Values →	20	30	40	0	0
Individual element →	Marks[0]	Marks[1]	Marks[2]	Marks[3]	Marks[4]
Address Locations →	80820	80822	80824	80826	80828

Array Memory Mapping

Static initialization: method4:

we can initialize the c array at the time of declaration

int marks[ ]={20,30,40,50,60.70,80};
in this declaration array size is not defined so, thecompilerfix the array size based on the max size of the initialization element . in this case max initialization element is 7. So the array size is fix with 7.

Index →	0	1	2	3	4	5	6
Values →	**20**	**30**	**40**	**50**	**60**	**70**	**80**
Individual element →	**Marks[0]**	**Marks[1]**	**Marks[2]**	**Marks[3]**	**Marks[4]**	**Marks[5]**	**Marks[6]**
Address Locations →	**80820**	**80822**	**80824**	**80826**	**80828**	**80830**	**80832**

Array Memory Mapping

2. Runtime initialization

Runtime initialization means we assign values through runtime by using keyboard.

```
#include<stdio.h>
Int main()
{
Int marks[5],I;
Printf("\nEnter 5 elements in to the array");
For(i=0;i<5;i++);
Scanf("%d",&marks[i]);
//print the array
For(i=0;i<5;i++)
Printf("\n marks[%d]=%d",I,marks[i]);
Return 0;
}
```

Using array elements in expressions

Example: int a[10];

declares 10 integers and can be accessed by the name a
each variable is assigned a unique location (location is also called as an index). the range of the location is 0 to (length – 1). for the said array range of the location is 0 to 9.

0	1	2	3	4	5	6	7	8	9
15	8	14	45	48	45	-57	45	1	0

```
x = a[1] * 2;    /* sets x to 16 */
a[4] = 88;   /* replaces 48 with 88 */
```

m = 6;
y = a[m]; /* sets y to –57 */
z = a[a[9]]; /* sets z to 15 */

Accessing Array Elements

Accessing array elements by using
1. Accessing Individual element
2. Accessing by using loops
3. Accessing by using pointers
4. Accessing by using array name with different ways

1. **Accessing Individual element**
#include<stdio.h>
int main(){
int i=0;
int marks[5];//declaration of array
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
Printf(“%4dt”,marks[0]);
Printf(“%4d”,marks[1]);
Printf(“%4d”,marks[2]);
Printf(“%4d”,marks[3]);
Printf(“%4d”,marks[4]);
Return 0;
}
Output: 80   60   70   85   75
	2. **Accessing by using loops**
	#include<stdio.h>
	int main(){
	int i=0;
	int marks[5];//declaration of array
	marks[0]=80;//initialization of array
	marks[1]=60;
	marks[2]=70;
	marks[3]=85;
	marks[4]=75;
	//traversal of array
	for(i=0;i<5;i++){
	printf("%4d \n",marks[i]);
	}//end of for loop
	return 0;
	}
	Output: 80 60 70 85 75

3. Accessing by using pointers

- **Array elements are always stored in contiguous memory locations.**
- **A pointer when incremented always points to the next location of its type.**

```
#include<stdio.h>
int main()
{
int num[]={24,34,12,44,56,17 };
inti,*j;
```

Pointer variable _ j__

```
j=&num[0]; //assign address of zeroth element
for(i=0;i<=5;i++)
{
printf("\n  address=%u  element  = %d",j,*j);
j++; //increment pointer point to next location
}
return (0);
}
```

Pointer value	65512
Pointer address	8500

Output:

```
Address=65512
element=24
Address=65516
element=34
Address=65520
element=12
Address=65524
element=44
Address=65528  element=56
Address=65532  element=17
```

Index →	0	1	2	3	4	5
Values →	24	34	12	44	56	17
Individual element →	num[0]	num [1]	num [2]	num [3]	num [4]	num [5]
Address Locations →	65512	65516	65520	65524	65528	65532

Array Memory Mapping

4. Accessing by using array name with different ways

int num[]={24,34,12,44,56,17 };

Mentioning the name of the array means we get the base address of the arrayso, Below all are point to the same base location

```
num[i]
*(num+i)
*(i+num)
i[num]
```

Example:

```
#include<stdio.h>
int main()
{
int num[]={24,34,12,44,56,17 };
inti;
for(i=0;i<-5;i++)
{
printf("\n address=%u",&num[i]);
printf("element=%3d %3d",num[i], i[num]);
printf("element=%3d %3d",*(num+i), *(i+num));
}
return 0;
}
```

Output:

```
Address=65512 element=24   24   24   24
Address=65516 element=34   34   34   34
Address=65520 element=12   12   12   12
Address=65524 element=44   44   44   44
Address=65528 element=56   56   56   56
Address=65532 element=17   17   17   17
```

No index out of bound checking:

there is no index out of bounds checking in c

```
#include <stdio.h>

int main()
{
   int  arr[2];
   printf("%d ", arr[3]);
   printf("%d ", arr[-2]);
     return 0;
}
```

Output

```
-449684907 4195777
```

Copying arrays:

we have two arrays list1 and list2

int list1[6] = {2, 4, 6, 8, 10, 12};

int list2[6];

and we want to copy the contents of list1 to list2. for general variables (e.g. int x=3, y=5) we use simple assignment statement (x=y or y=x). but for arrays the following statement is wrong.

list2 = list1;

we must copy between arrays element by element and the two arrays must have the same size. in the following example, we use a for loop which makes this easy.

```
#include <stdio.h>
main()
int list1[6] = {2, 4, 6, 8, 10, 12};
int list2[6];
for (int ctr = 0; ctr<6; ctr++)
{
list2[ctr] = list1[ctr];
}
printf("elements of list2 :\n");
for (int ctr = 0; ctr<6; ctr++)
 {
printf("%d ",list2[ctr]);
 }
}
copy
output:
elements of list2 :
2 4 6 8 10 12
```

Q. Write a c program for finding the no. of students passed in an examination

```
#include<stdio.h>
#include<conio.h>
int main()
  {
intn,a[40],count=0,i;
printf("Enter the number of students");
scanf("%d",&n);
printf("\n Enter marks of the students");
for(i=1;i<=n;i++)
 {
printf("\n Enter %d student marks :",i);
scanf("%d",&a[i]);
if(a[i]>=50)
count=count+1;
}
printf(" Number of students passed =%d ", count);
return 0;
 }
```

Output:

```
Enter the number of students7
 Enter marks of the students
 Enter 1 student marks :55
 Enter 2 student marks :42
 Enter 3 student marks :77
 Enter 4 student marks :63
 Enter 5 student marks :29
 Enter 6 student marks :57
 Enter 7 student marks :89
 Number of students passed =5
```

Q. Write a program to find Average of Student marks

```
#include<stdio.h>
 #include<conio.h>
void main()
 {
int marks[50],i,sum=0,average,count=0;
printf("Enter the Student marks");
for(i=0;i<5;i++)
 {
scanf("%d",&marks[i]);
sum=sum+marks[i];
 }
average=sum/5.0;
for(i=0;i<5;i++)
if(marks[i]>average)
count++;
printf("No of students who scored more than average marks:%d",count);
return 0;
} Output:
```

Enter the Student marks: 96 56 86 76 36 No of students who scored more than average marks: 3

Q. write a c program to reverse an array

Different arrays	**Within the array**

#include <stdio.h>
#define MAX_SIZE 100   // Maximum array size
int main()
{
   int arr[MAX_SIZE], reverse[MAX_SIZE];
   int size, i, arrIndex, revIndex;
 	
   /* Input size of the array */
	#include<stdio.h>
	int main()
	{
	int a[20],i,n,t;
	printf("\n Enter the array size: ");
	scanf("%d",&n);
	printf("\n Enter %d elements in to thearray:");
	for(i=1;i<=n;i++)

```
printf("\nFind reverse of an array:");
printf("\nEnter size of the array: ");
   scanf("%d", &size);

   /* Input array elements */
   printf("Enter %d elements in to the array: ",size);
   for(i=0; i<size; i++)
   {
      scanf("%d", &arr[i]);
   }
revIndex = 0;
arrIndex = size - 1;
   while(arrIndex>= 0)
   {
reverse[revIndex] = arr[arrIndex];
revIndex++;
arrIndex--;
   }
   printf("\nReversed array : ");
   for(i=0; i<size; i++)
   {
      printf("%d\t", reverse[i]);
   }
   return 0;
}
```

```
scanf("%d",&a[i]);
printf("\n Given elements are :" );
for(i=1;i<=n;i++)
printf("%4d",a[i]);
for(i=1;i<=n/2;i++)
{
t=a[i];
a[i]=a[n-i+1];
a[n-i+1]=t;
}
printf("\n Given elements in reverse order are :" );
for(i=1;i<=n;i++)
printf("%4d",a[i]);
 }
```

Output:

```
Enter the array size: 5
 Enter 5 elements in to thearray:100 -50 25 11 5
 Given elements are : 100 -50  25  11  5
Given elements in reverse order are : 5 11 25 -50 100
```

Two dimensional array

So far we have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. In such situations this concept is useful.

✓An array with two dimensions are called “Two dimensional array”

✓An array with two dimensions‟ are called matrix.

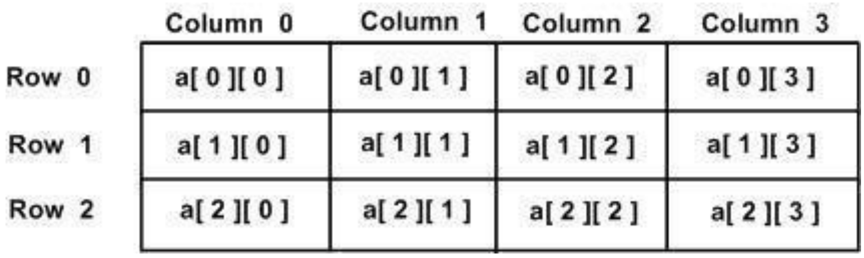
✓When the data must be stored in the form of a matrix we use two dimensional arrays.

Declaration of Two –Dimensional array:
Syntax:
datatype array-name [row- size] [column- size];

For example a two dimensional array consisting of 5 rows and 3 columns. So the total number of elements which can be stored in this array are 5*3 i.e. 15

Ex: int a[3][4];

a two-dimensional array **a**, which contains three rows and four columns can be shown as follows −



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initialization of 2-D array:
Two –D array can also be initialized at the place of declaration itself.
Syntax:
Data-type array-name [row-size] [column-size] = {{row1 list}, {row2 list}….{rown list}

We already know, when we initialize a normal array (or you can say one dimensional array) during declaration, we need not to specify the size of it. However that's not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration. Let's understand this with the help of few examples –

Method1:-
/* Valid declaration*/
intabc[2][2] = {1, 2, 3 ,4 } ;
Method2:-
/* Valid declaration*/
intabc[][2] = {1, 2, 3 ,4 } ;

NOTE:
/* Invalid declaration – you must specify second dimension*/
intabc[][] = {1, 2, 3 ,4 } ;
intabc[2][] = {1, 2, 3 ,4 };

Method3:-
intabc[2][2] = { {1, 2}, {3 ,4} } ;

```
intabc[2][2] = { {1, 2},
                 {3 ,4}
                 } ;
```

Method4:-

```
intabc[2][3] = { {1}, {3 ,4}, } ;
```

here, we declared abc array with row size = 2 and column size = 3. but we only assigned 1 column in the 1st row and 2 columns in the 2nd row. in these situations, the remaining values will assign to default values (0 in this case).

Storage Representation of two –Dimensional array:

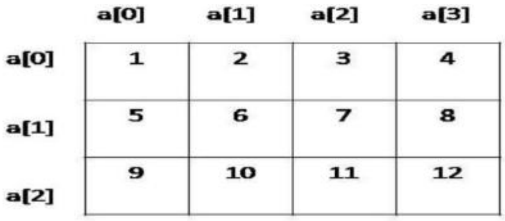
When speaking of two-dimensional arrays, we are logically saying that, it consists of two rows and columns but when it is stored in memory, the memory is linear.

✓Hence, the actual storage differs from our matrix representation.

Two major types of representation can be used for 2-D array

1. Row representation
2. Column representation

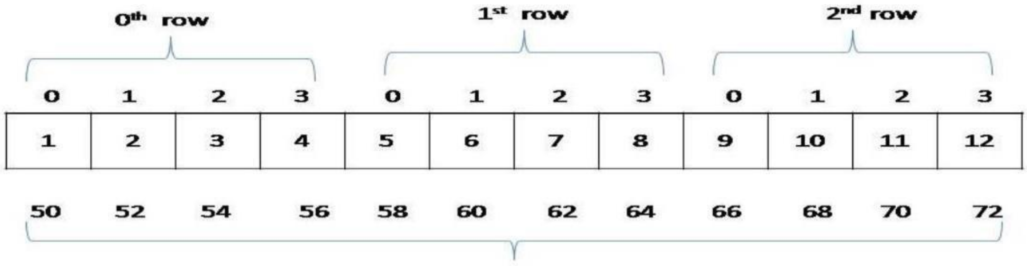
e.g. int a[3][3]



	a[0]	a[1]	a[2]	a[3]
a[0]	1	2	3	4
a[1]	5	6	7	8
a[2]	9	10	11	12

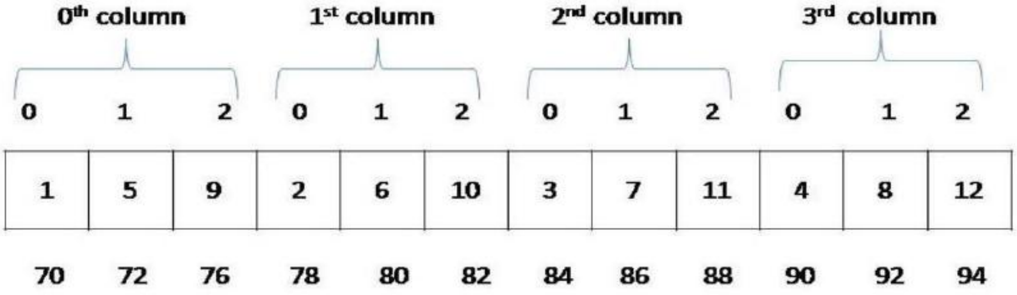
Logical view of a [3] [4];

Row Representation



0th row				1st row				2nd row			
0	1	2	3	0	1	2	3	0	1	2	3
1	2	3	4	5	6	7	8	9	10	11	12
50	52	54	56	58	60	62	64	66	68	70	72

Column Representation



0th column			1st column			2nd column			3rd column		
0	1	2	0	1	2	0	1	2	0	1	2
1	5	9	2	6	10	3	7	11	4	8	12
70	72	76	78	80	82	84	86	88	90	92	94

Q. Storing elements in a matrix and printing it.

```
#include <stdio.h>
#define MAX 10
void main ()
{
intarr[MAX][MAX],i,j;
            intm,n;
            printf("\n enter size of matrix1 and matrix2:");
            scanf("%d%d",&m,&n);

for (i=0;i<m;i++)  //row
    {
for (j=0;j<n;j++)    //column
        {
printf("Enter a[%d][%d]: ",i,j);
scanf("%d",&arr[i][j]);
        }
    }
printf("\n printing the elements in matrix form. .. \n");
for(i=0;i<m;i++) // row
    {
```

```
for (j=0;j<n;j++)  //column
        {
printf("%d\t",arr[i][j]);
        }
                          printf("\n");
    }
}
```

```
Output:
enter size of matrix1 and matrix2:3 3
Enter a[0][0]: 1
Enter a[0][1]: 2
Enter a[0][2]: 3
Enter a[1][0]: 4
Enter a[1][1]: 5
Enter a[1][2]: 6
Enter a[2][0]: 7
Enter a[2][1]: 8
Enter a[2][2]: 9
printing the elements in matrix form....
1       2       3
4       5       6
7       8       9
```

Q. Matrix Addition

```
#include<stdio.h>
#define  MAX  100
int main ()
{
    int a[MAX][MAX],b[MAX][MAX],c[MAX][MAX],i,j,k;
             int r1,c1,r2,c2;
             printf("\nColumn of first matrix and row of second matrix must be same.");
             printf("\n Enter the number of rows and columns for matrix A(between 1 and 50):");
             scanf("%d%d",&r1,&c1);
             printf("\n Enter the number of rows and columns for matrix B(between 1 and 50):");
             scanf("%d%d",&r2,&c2);

    if (r1!=r2 && c1!=c2)
    {
             printf("Matrices Addition is not possible");
             printf("\nRows and Columns of Both Matrices are must be same:");
             }
```

```
else
{
            printf("\n Enter matrix1 values:\n");
       for (i=0;i<r1;i++)  //row
         for (j=0;j<c1;j++)    //column
          scanf("%d",&a[i][j]);

            printf("\n Enter matrix2 values:\n");
       for (i=0;i<r2;i++)  //row
         for (j=0;j<c2;j++)    //column
          scanf("%d",&b[i][j]);

      printf("\nThe First matrix is :\n");
            for(i=0;i<r1;i++)
            {
            printf("\n");
            for(j=0;j<c1;j++)
      printf("%d\t",a[i][j]);
            }

      printf("\nThe Second matrix is :\n");
            for(i=0;i<r2;i++)
            {
            printf("\n");
            for(j=0;j<c2;j++)
            printf("%d\t",b[i][j]);
            }

            for ( i=0;i<r1;i++)//row of first matrix
       {
         for ( j=0;j<c2;j++) //column of second matrix
         {
           c[i][j] = a[i][j] + b[i][j];
         }

       }

       printf("\nAddition of two entered matrices are:-\n");

       for (i=0;i<r1;i++)  //row of first matrix
       {
```

```
                         for (j=0;j<c2;j++)    //column of second matrix
                    printf("%4d",c[i][j]);
                 printf("\n");
                }
  }

return(0);
}
```

Output:

```
Column of first matrix and row of second matrix must be same.
 Enter the number of rows and columns for matrix A(between 1 and 50):1 2

 Enter the number of rows and columns for matrix B(between 1 and 50):3 2
Matrices Addition is not possible
Rows and Columns of Both Matrices are must be same:
```

Output:

```
Column of first matrix and row of second matrix must be same.
 Enter the number of rows and columns for matrix A(between 1 and 50):2 2

 Enter the number of rows and columns for matrix B(between 1 and 50):2 2

 Enter matrix1 values:
1 2 3 4

 Enter matrix2 values:
1 1 1 1

The First matrix is :

1      2
3      4
The Second matrix is :

1      1
1      1
Addition of two entered matrices are:-
   2  3
   4   5
```

Q. Matrix Subtraction

```
#include<stdio.h>
#define  MAX  100
int main ()
{
   int a[MAX][MAX],b[MAX][MAX],c[MAX][MAX],i,j,k;
          int r1,c1,r2,c2;
          printf("\nColumn of first matrix and row of second matrix must be same.");
          printf("\n Enter the number of rows and columns for matrix A(between 1 and 50):");
          scanf("%d%d",&r1,&c1);
          printf("\n Enter the number of rows and columns for matrix B(between 1 and 50):");
          scanf("%d%d",&r2,&c2);

   if (r1!=r2 || c1!=c2)
   {
          printf("Matrices Subtraction is not possible");
          printf("\nRows and Columns of Both Matrices are must be same:");
          }
  else
  {
                    printf("\n Enter matrix1 values:\n");
             for (i=0;i<r1;i++)  //row
               for (j=0;j<c1;j++)    //column
                 scanf("%d",&a[i][j]);

                    printf("\n Enter matrix2 values:\n");
             for (i=0;i<r2;i++)  //row
               for (j=0;j<c2;j++)    //column
                 scanf("%d",&b[i][j]);

           printf("\nThe First matrix is :\n");
                    for(i=0;i<r1;i++)
                    {
                    printf("\n");
                    for(j=0;j<c1;j++)
          printf("%d\t",a[i][j]);
                    }

          printf("\nThe Second matrix is :\n");
                    for(i=0;i<r2;i++)
                    {
```

```
               printf("\n");
               for(j=0;j<c2;j++)
               printf("%d\t",b[i][j]);
               }

               for ( i=0;i<r1;i++)//row of first matrix
          {
            for ( j=0;j<c2;j++) //column of second matrix
            {
               c[i][j] = a[i][j] - b[i][j];
            }

          }

          printf("\nSubtraction of two entered matrices are:-\n");

          for (i=0;i<r1;i++)  //row of first matrix
          {
                for (j=0;j<c2;j++)    //column of second matrix
             printf("%4d",c[i][j]);
           printf("\n");
          }
 }

return(0);
}
```

Output:

```
Column of first matrix and row of second matrix must be same.
 Enter the number of rows and columns for matrix A(between 1 and 50):2 2

 Enter the number of rows and columns for matrix B(between 1 and 50):4 4
Matrices Subtraction is not possible
Rows and Columns of Both Matrices are must be same:
```

Output:

```
Column of first matrix and row of second matrix must be same.
 Enter the number of rows and columns for matrix A(between 1 and 50):2 2
 Enter the number of rows and columns for matrix B(between 1 and 50):2 2
 Enter matrix1 values:
1 2 3 4
 Enter matrix2 values:
```

```
1 1 1 1
The First matrix is :
1       2
3       4
The Second matrix is :
1       1
1       1
Subtraction of two entered matrices are:-
   0  1
   2  3
```

Q. Matrix multiplication

```
#include<stdio.h>
#define  MAX  100
int main ()
{
    int a[MAX][MAX],b[MAX][MAX],c[MAX][MAX],i,j,k;
            int r1,c1,r2,c2;
            printf("\nColumn of first matrix and row of second matrix must be same.");
            printf("\n Enter the number of rows and columns for matrix A(between 1 and 50):");
            scanf("%d%d",&r1,&c1);
            printf("\n Enter the number of rows and columns for matrix B(between 1 and 50):");
            scanf("%d%d",&r2,&c2);

    if (c1!=r2)
    {
            printf("Matrices multiplication is not possible");
            printf("\nColumn of first matrix and row of second matrix must be same.");
            }
  else
  {
                        printf("\n Enter matrix1 values:\n");
                for (i=0;i<r1;i++)  //row
                   for (j=0;j<c1;j++)    //column
                     scanf("%d",&a[i][j]);

                        printf("\n Enter matrix2 values:\n");
                for (i=0;i<r2;i++)  //row
                   for (j=0;j<c2;j++)    //column
                     scanf("%d",&b[i][j]);
```

```
             printf("\nThe First matrix is :\n");
                        for(i=0;i<r1;i++)
                        {
                        printf("\n");
                        for(j=0;j<c1;j++)
            printf("%d\t",a[i][j]);
                        }

            printf("\nThe Second matrix is :\n");
                        for(i=0;i<r2;i++)
                        {
                        printf("\n");
                        for(j=0;j<c2;j++)
                        printf("%d\t",b[i][j]);
                        }

            for ( i=0;i<r1;i++)//row of first matrix
                {
                  for ( j=0;j<c2;j++) //column of second matrix
                  {
                    c[i][j]=0;
                        for(k=0;k<c1;k++ )//column of first matrix
                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
                  }

                }

                printf("\nProduct of two entered matrices are:-\n");

                for (i=0;i<r1;i++)  //row of first matrix
                {
                          for (j=0;j<c2;j++)    //column of second matrix
                     printf("%4d",c[i][j]);
                 printf("\n");
                }
  }

return(0);
}
```

Output:

```
Column of first matrix and row of second matrix must be same.
 Enter the number of rows and columns for matrix A(between 1 and 50):2 2
 Enter the number of rows and columns for matrix B(between 1 and 50):1 2

Matrices multiplication is not possible
Column of first matrix and row of second matrix must be same.
```

Output:

```
Column of first matrix and row of second matrix must be same.
 Enter the number of rows and columns for matrix A(between 1 and 50):2 2

 Enter the number of rows and columns for matrix B(between 1 and 50):2 2
 Enter matrix1 values:
1 2 3 4
 Enter matrix2 values:
1 1 1 1
The First matrix is :
1       2
3       4
The Second matrix is :
1       1
1       1
Product of two entered matrices are:-
   3  3
   7   7
```

Q. Transpose of the matrix

```
#include<stdio.h>
#define  MAX  100
int main ()
{
    int a[MAX][MAX],b[MAX][MAX],i,j;
             int r1,c1;
             printf("\nColumn of first matrix and row of second matrix must be same.");
             printf("\n Enter the number of rows and columns for matrix A(between 1 and 50):");
             scanf("%d%d",&r1,&c1);

    printf("\n Enter matrix values:\n");
                 for (i=0;i<r1;i++)  //row
                    for (j=0;j<c1;j++)    //column
                      scanf("%d",&a[i][j]);
```

```
          printf("\nThe Given matrix is :\n");
                   for(i=0;i<r1;i++)
                   {
                   printf("\n");
                   for(j=0;j<c1;j++)
          printf("%d\t",a[i][j]);
                   }

          for ( i=0;i<r1;i++)//row
                   for ( j=0;j<c1;j++) //column
                b[j][i] = a[i][j];

          printf("\nTranspose Matrix is:-\n");
            for (i=0;i<r1;i++)  //row
             {
                    for (j=0;j<c1;j++)    //column
                printf("%4d",b[i][j]);
             printf("\n");
             }
return(0);
}
```

Output:

```
Column of first matrix and row of second matrix must be same.
 Enter the number of rows and columns for matrix A(between 1 and 50):2 2
 Enter matrix values:
1 2 3 4
The Given matrix is :
1      2
3      4
Transpose Matrix is:-
   1  3
   2   4
```

How to declare a multidimensional array in c

a multidimensional array is declared using the following syntax:

type array_name[d1][d2][d3][d4]………[dn];

where each **d** is a dimension, and **dn** is the size of final dimension.

examples:

1. **int table[5][5][20];**
2. **float arr[5][6][5][6][5];**

in example 1:

- **int** designates the array type integer.
- **table** is the name of our 3d array.
- our array can hold 500 integer-type elements. this number is reached by multiplying the value of each dimension. in this case: **5x5x20=500**.

In example 2:

- array **arr** is a five-dimensional array.
- It can hold 4500 floating-point elements (**5x6x5x6x5=4500**).

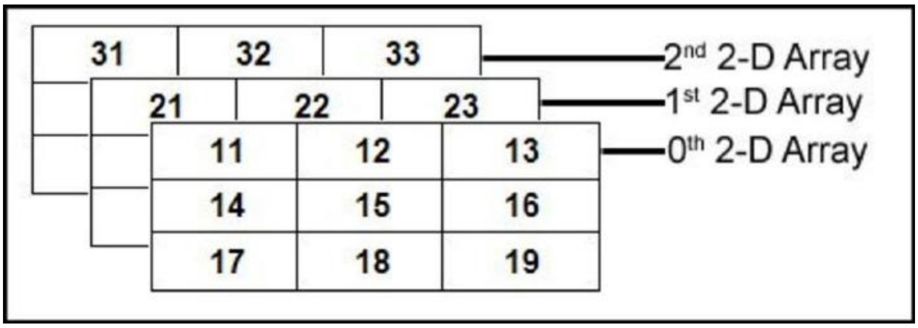
Can you see the power of declaring an array over variables? when it comes to holding multiple values in c programming, we would need to declare several variables. but a single array can hold thousands of values.

Explanation of a 3d array

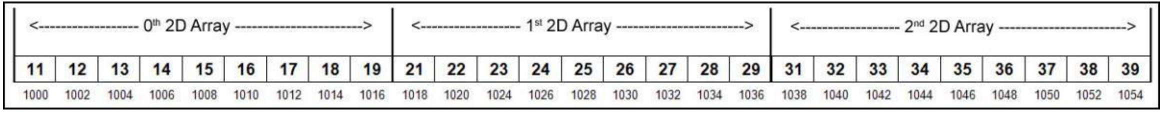
let's take a closer look at a 3d array. a 3d array is essentially an array of arrays of arrays: it's an array or collection of 2d arrays, and a 2d array is an array of 1d array.

it may sound a bit confusing, but don't worry. as you practice working with multidimensional arrays, you start to grasp the logic.

the diagram below may help you understand:



3d array conceptual view



<------------------ 0th 2D Array ---------------------->									<------------------ 1st 2D Array ---------------------->									<------------------ 2nd 2D Array ---------------------->								
11	12	13	14	15	16	17	18	19	21	22	23	24	25	26	27	28	29	31	32	33	34	35	36	37	38	39
1000	1002	1004	1006	1008	1010	1012	1014	1016	1018	1020	1024	1026	1028	1030	1032	1034	1036	1038	1040	1042	1044	1046	1048	1050	1052	1054

3d array memory map.

initializing a 3d array in c

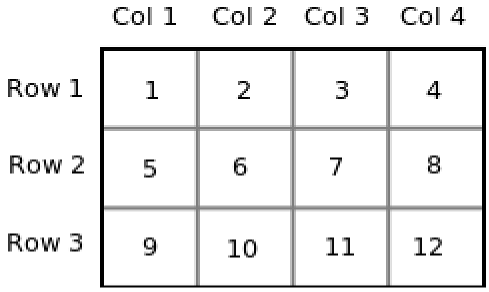
like any other variable or array, a 3d array can be initialized at the time of compilation. by default, in c, an uninitialized 3d array contains “garbage” values, not valid for the intended use.

Pointer to Multidimensional Arrays

1. Pointers and two dimensional Arrays: In a two dimensional array, we can access each element by using two subscripts, where first subscript represents the row number and second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, we can access any element *arr[i][j]* of the array using the pointer expression ***(*(arr + i) + j)**. Now we’ll see how this expression can be derived.

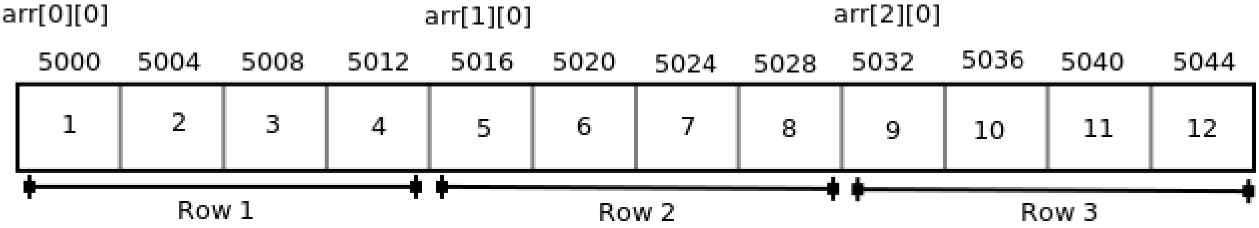
Let us take a two dimensional array *arr[3][4]*:

intarr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };



	Col 1	Col 2	Col 3	Col 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

Since memory in a computer is organized linearly it is not possible to store the 2-D array in rows and columns. The concept of rows and columns is only theoretical, actually, a 2-D array is stored in row-major order i.e rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory.

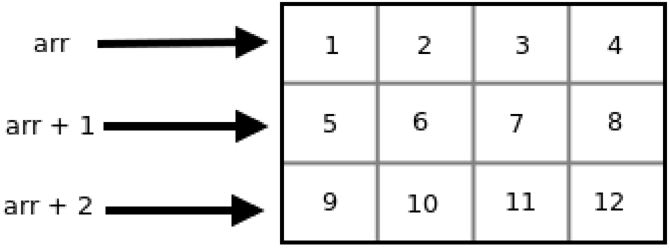


arr[0][0]				arr[1][0]				arr[2][0]			
5000	5004	5008	5012	5016	5020	5024	5028	5032	5036	5040	5044
1	2	3	4	5	6	7	8	9	10	11	12
Row 1				Row 2				Row 3			

Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another. In other words, we can say that 2-D dimensional arrays that are placed one after another. So here *arr* is an array of 3 elements where each element is a 1-D array of 4 integers.

We know that the name of an array is a constant pointer that points to 0th 1-D array and contains address 5000. Since *arr* is a ‘pointer to an array of 4 integers’, according to pointer arithmetic the expression arr + 1 will represent the address 5016 and expression arr + 2 will represent address 5032.

So we can say that *arr* points to the 0th 1-D array,
arr + 1 points to the 1st 1-D array and
arr + 2 points to the 2nd 1-D array.



arr - Points to 0th element of arr - Points to 0th 1-D array - 5000
arr + 1 - Points to 1th element of arr - Points to 1nd 1-D array - 5016
arr + 2 - Points to 2th element of arr - Points to 2nd 1-D array - 5032

In general we can write:

arr + i Points to ith element of arr ->Points to ith 1-D array

- Since arr + i points to ith element of *arr*, on dereferencing it will get ith element of *arr* which is of course a 1-D array. Thus the expression **(arr + i)* gives us the base address of ith 1-D array.
- We know, the pointer expression **(arr + i)* is equivalent to the subscript expression *arr[i]*. So **(arr + i)* which is same as *arr[i]* gives us the base address of ith 1-D array.

***(arr + 0) - arr[0] - Base address of 0th 1-D array - Points to 0th element of 0th 1-D array - 5000**
***(arr + 1) - arr[1] - Base address of 1st 1-D array - Points to 0th element of 1st 1-D array - 5016**
***(arr + 2) - arr[2] - Base address of 2nd 1-D array - Points to 0th element of 2nd 1-D array - 5032**

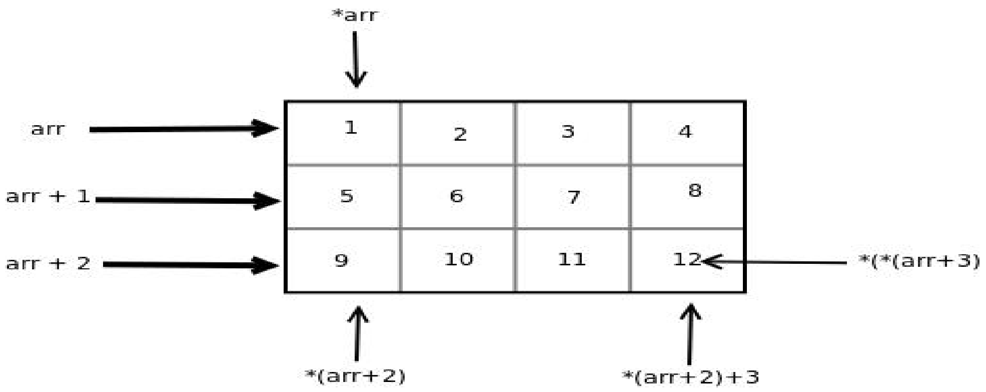
In general we can write:
*(arr + i) -arr[i] - Base address of ith 1-D array -> Points to 0th element of ith 1-D array

Note: Both the expressions *(arr + i)* and **(arr + i)* are pointers, but their base type are different. The base type of *(arr + i)* is ‘an array of 4 units’ while the base type of *(arr + i) or arr[i] is int.

- To access an individual element of our 2-D array, we should be able to access any jth element of ith 1-D array.

- Since the base type of **(arr + i)* is *int* and it contains the address of 0th element of ith 1-D array, we can get the addresses of subsequent elements in the ith 1-D array by adding integer values to **(arr + i)*.
- For example **(arr + i) + 1* will represent the address of 1st element of 1stelement of ith 1-D array and **(arr+i)+2* will represent the address of 2nd element of ith 1-D array.
-
- Similarly *(arr + i) + j will represent the address of jth element of ith 1-D array. On dereferencing this expression we can get the jth element of the ith 1-D array.

arr	**Points to 0th 1-D array**
***arr**	**Points to 0th element of 0th 1-D array**
(arr + i)	**Points to ith 1-D array**
***(arr + i)**	**Points to 0th element of ith 1-D array**
***(arr + i) + j)**	**Points to jth element of ith 1-D array**
***(*(arr + i) + j)**	**Reprents the value of jth element of ith 1-D array**



```
// C program to print the values and
// address of elements of a 2-D array
#include<stdio.h>

int main()
{
intarr[3][4] = {
                                    { 10, 11, 12, 13 },
                                    { 20, 21, 22, 23 },
                                    { 30, 31, 32, 33 }
                            };
inti, j;
for (i = 0; i< 3; i++)
{
    printf("Address of %dth array = %p %p\n",
                                    i, arr[i], *(arr + i));

    for (j = 0; j < 4; j++)
    printf("%d %d ", arr[i][j], *(*(arr + i) + j));
    printf("\n");
}

return 0;
}
```

Output:

```
Address of 0th array = 0x7ffe50edd580 0x7ffe50edd580

10 10 11 11 12 12 13 13

Address of 1th array = 0x7ffe50edd590 0x7ffe50edd590

20 20 21 21 22 22 23 23

Address of 2th array = 0x7ffe50edd5a0 0x7ffe50edd5a0

30 30 31 31 32 32 33 33
```

Strings

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

There are two ways to declare a string in c language.

- By char array
- By string literal

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

charch[13]={'a', 'n', 'n', 'a', 'm', 'a', 'c', 'h', 'a', 'r', “y', ‘a’, \0'};

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

C Strings

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

char ch[13]={'a', 'n', 'n', 'a', 'm', 'a', 'c', 'h', 'a', 'r', ‘’y', ‘a’, \0'};

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

charch[]="annamacharya";

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

Q. Difference between char array and string literal

There are two main differences between char array and literal.

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

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

String Example in C

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

char ch[13]={'a', 'n', 'n', 'a', 'm', 'a', 'c', 'h', 'a', 'r', ‘’y', ‘a’, \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	11	12
a	n	n	a	m	a	c	h	a	r	y	a	\0

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

char ch[]={'a', 'n', 'n', 'a', 'm', 'a', 'c', 'h', 'a', 'r', ‘’y', ‘a’, \0'};

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

char ch[]="annamacharya";

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

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(){
charch[13]={'a', 'n', 'n', 'a', 'm', 'a', 'c', 'h', 'a', 'r', ‘’y', ‘a’, \0'};
char ch2[13]="annamacharya";

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

Output

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

Q. Explain about traversing methods in strings OR Explain how to traverse string array

Traversing String

Traversing the string is one of the most important aspects in any of the programming languages. We may need to manipulate a very large text which can be done by traversing the text. Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are **two ways** to traverse a string.

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

Let's discuss each one of them.

1. Using the length of string

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

```
#include<stdio.h>
void main ()
{
char s[13] = " annamacharya ";
inti = 0;
int count = 0;
while(i<13)
   {
      if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
      {
count ++;
      }
i++;
   }
printf("The number of vowels %d",count);
}
```

Output

```
The number of vowels 5
```

2. Using the null character

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

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

```
{
char s[13] = "annmacharya";
inti = 0;
int count = 0;
while(s[i] != NULL)
   {
       if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
       {
count ++;
       }
i++;
   }
printf("The number of vowels %d",count);
}
```

Output

```
The number of vowels 5
```

Q. what are the limitations of the scanf() function explain it

Accepting string as the input

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

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

Output

```
Enter the string?annamacharya is the best
You entered annamacharya
```

It is clear from the output that, the above code will not work for space separated strings. To make this code working for the space separated strings, the minor changed required in the scanf function,

i.e., instead of writing scanf("%s",s), we must write: scanf("%[^\n]s",s) which instructs the compiler to store the string s while the new line (\n) is encountered. Let's consider the following example to store the space-separated strings.

```
#include<stdio.h>

void main ()
{     char s[20];
printf("Enter the string?");
scanf("%[^\n]s",s);
printf("You entered %s",s);
}
```

Output

```
Enter the string?annamacharya is the best
You entered annamacharya is the best
```

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

Q. What are the limitations of array? OR Explain about array bounds checking

Some important points

However, there are the following points which must be noticed while entering the strings by using scanf.

The compiler doesn't perform bounds checking on the character array. Hence, there can be a case where the length of the string can exceed the dimension of the character array which may always overwrite some important data.

Instead of using scanf, we may use gets() which is an inbuilt function defined in a header file string.h. The gets() is capable of receiving only one string at a time.

Q, Discuss about pointers and strings?

Pointers with strings

We have used pointers with the array, functions, and primitive data types so far. However, pointers can be used to point to the strings. There are various advantages of using pointers to point strings. Let us consider the following example to access the string via the pointer.

```
#include<stdio.h>
void main ()
{
char s[13] = "annamacharya ";
char *p = s; // pointer p is pointing to string s.
printf("%s",p); // the string annamacharya is printed if we print p.
}
```

Output

```
annamacharya
```

Q. Explain about string copying

Wecannot change the content of s or copy the content of s into another string directly. For this purpose, we need to use the pointers to store the strings. In the following example, we have shown the use of pointers to copy the content of a string into another.

```
#include<stdio.h>
void main ()
{
char *p = "hello annamacharya";
printf("String p: %s\n",p);
char *q;
printf("copying the content of p into q...\n");
   q = p;
printf("String q: %s\n",q);
}
```

Output

```
String p: hello annamacharya
copying the content of p into q...
String q: hello annamacharya
```

Once a string is defined, it cannot be reassigned to another set of characters. However, using pointers, we can assign the set of characters to the string. Consider the following example.

```
#include<stdio.h>
void main ()
{
char *p = "hello annamacharya ";
printf("Before assigning: %s\n",p);
   p = "hello";
printf("After assigning: %s\n",p);
}
```

Output

```
Before assigning: hello annamacharya
After assigning: hello
```

Q. How to copy the contents of one array in to another array by using pointers

C gets() function

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

Declaration

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

Reading string using gets()

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

Output

```
Enter the string?
annamacharya is the best
You entered annamacharya is the best
```

C puts() function

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

Declaration

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

Output:

```
Enter your name: Sreenivas reddy
Your name is: Sreenivas reddy
```

C String Functions

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

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

Q. Write a c program to check whether a given string is palindrome or not?

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

int main(){
char string1[20];
inti, length;
int flag =0;

   printf("Enter a string:");
   scanf("%s", string1);

   length =strlen(string1);

for(i=0;i < length ;i++){
if(string1[i]!= string1[length-i-1]){
        flag =1;
break;
}
}

if(flag){
     printf("%s is not a palindrome", string1);
}
else{
     printf("%s is a palindrome", string1);
}
return0;
}
OUTPUT:
Enter a string: MADAM
MADAM is a palindrome
```

Q. Implement string length function as User defined function?

```
#include<stdio.h>

// Prototype Declaration
int FindLength(char str[]);

int main() {
  char str[100];
  int length;

  printf("\nEnter the String : ");
  gets(str);

  length = FindLength(str);

  printf("\nLength of the String is : %d", length);
  return(0);
}

int FindLength(char str[]) {
  int len = 0;
  while (str[len] != '\0')
len++;
  return (len);
}
```

Q. Implement string copy function as User defined function?

```
#include<stdio.h>

int main() {
   char s1[100], s2[100];
   int i;

   printf("\nEnter the string :");
   gets(s1);

i = 0;
   while (s1[i] != '\0') {
      s2[i] = s1[i];
i++;
   }

   s2[i] = '\0';
   printf("\nCopied String is %s ", s2);

   return (0);
}
```

Q. Implement string concatenation function as User defined function?

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

void concat(char[], char[]);

int main() {
   char s1[50], s2[30];

   printf("\nEnter String 1 :");
   gets(s1);
   printf("\nEnter String 2 :");
   gets(s2);

concat(s1, s2);
   printf("nConcated string is :%s", s1);
```

```
  return (0);
}

void concat(char s1[], char s2[]) {
  int i, j;

i = strlen(s1);

  for (j = 0; s2[j] != '\0'; i++, j++) {
    s1[i] = s2[j];
  }

  s1[i] = '\0';
}
```

Q. Implement string reverse function as User defined function?

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

int main() {
  char str[100], temp;
  int i, j = 0;

  printf("\nEnter the string :");
  gets(str);

i = 0;
  j = strlen(str) - 1;

  while  (i<  j)  {
    temp = str[i];
str[i]  =  str[j];
str[j] = temp;
i++;
    j--;
  }

  printf("\nReverse string is :%s", str);
  return (0);
```

Q. Implement string comparison function as User defined function?

```
#include<stdio.h>

int main() {
   char str1[30], str2[30];
   int i;

   printf("\nEnter two strings :");
   gets(str1);
   gets(str2);

i = 0;
   while (str1[i] == str2[i] && str1[i] != '\0')
i++;
   if (str1[i] > str2[i])
      printf("str1 > str2");
   else if (str1[i] < str2[i])
      printf("str1 < str2");
   else
      printf("str1 = str2");

   return (0);
}
```

Array of Strings

A string is a 1-D array of characters, so an array of strings is a 2-D array of characters. Just like we can create a 2-D array of int, float etc; we can also create a 2-D array of character or array of strings. Here is how we can declare a 2-D array of characters.

Declaration of the array of strings

Syntax:-

char string-array-name[row-size][column-size];

Here the first index (row-size) specifies the maximum number of strings in the array, and the second index (column-size) specifies the maximum length of every individual string.

For example, char language[5][10]; In the “language” array we can store a maximum of 5 Strings and each String can have a maximum of 10 characters.

In C language, each character take 1 byte of memory. For the “language” array it will allocate 50 bytes (1*5*10) of memory. Where each String will have 10 bytes (1*10) of memory space.

Initialization of array of strings

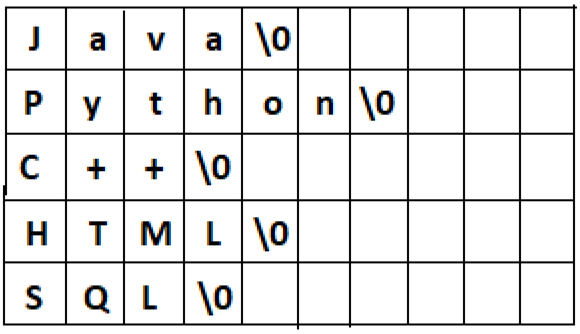
Two dimensional (2D) strings in C language can be directly initialized as shown below,

```
char language[5][10] = {"Java", "Python", "C++", "HTML", "SQL"};

charlargestcity[6][15] =

   {"Tokyo", "Delhi", "Shanghai", "Mumbai", "Beijing", "Dhaka"};
```

The two dimensional (2D) array of Strings in C also can be initialized as,

```
char language[5][10] =

{

  {'J','a','v','a','\0'},

  {'P','y','t','h','o','n','\0'},

  {'C','+','+','\0'},

  {'H','T','M','L','\0'},

  {'S','Q','L','\0'}

};
```

Since it is a two-dimension of characters, so each String (1-D array of characters) must end with null character i.e. ‘\0’



J	a	v	a	\0					
P	y	t	h	o	n	\0			
C	+	+	\0						
H	T	M	L	\0					
S	Q	L	\0						

Note1:- the number of characters (column-size) must be declared at the time of the initialization of the two-dimensional array of strings.

```
// it is valid
char language[ ][10] = {"Java", "Python", "C++", "HTML", "SQL"};
```

But that the following declarations are **invalid**.

```
// invalid
char language[ ][ ] = {"Java", "Python", "C++", "HTML", "SQL"};

// invalid
char language[5][ ] = {"Java", "Python", "C++", "HTML", "SQL"};
```

Note2:- Once we initialize the array of String then we can’t directly assign a new String.

```
char language[5][10] = {"Java", "Python", "C++", "HTML", "SQL"};

// now, we can't directly assign a new String
language[0] = "Kotlin"; // invalid

// we must copy the String
strcpy(language[0], "Kotlin"); // valid
// Or,
scanf(language[0], "Kotlin"); // valid
```

Reading and displaying 2d array of strings in C

```
#include<stdio.h>
int main()
{
char name[10][20];
inti,n;
printf("Enter the number of names (<10): ");
scanf("%d",&n);

    // reading string from user
printf("Enter %d names:\n",n);
for(i=0; i<n; i++)
gets(name[i]);

    // dispaying strings
printf("\nEntered names are:\n");
for(i=0;i<n;i++)
puts(name[i]);

return 0;
}
```

```
Enter the number of names (<10): 5
Enter 5 names:
c
c plus plus
java
python
angular 7

Entered names are:
c
c plus plus
java
python
angular 7Enter the number of names (<10): 5
Enter 5 names:
c
c plus plus
java
python
angular 7

Entered names are:
c
c plus plus
java
python
```

angular 7

Command line arguments in C

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

int main() { /* ... */ }

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.
To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

int main(int argc, char *argv[]) { /* ... */ }

or

int main(int argc, char **argv) { /* ... */ }

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.
- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

Properties of Command Line Arguments:

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[n] points last argument.

Note : You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes “” or single quotes ”.

```
#include<stdio.h>

intmain(intargc,char* argv[])
```

```
{
   intcounter;
   printf("Program Name Is: %s",argv[0]);
   if(argc==1)
      printf("\nNo Extra Command Line Argument Passed Other Than Program Name");
   if(argc>=2)
   {
      printf("\nNumber Of Arguments Passed: %d",argc);
      printf("\n----Following Are The Command Line Arguments Passed ----");
      for(counter=0;counter<argc;counter++)
         printf("\nargv[%d]: %s",counter,argv[counter]);
   }
   return0;
}
```

Output in different scenarios:

Without argument: When the above code is compiled and executed without passing any argument, it produces following output.

```
$ ./a.out

Program Name Is: ./a.out

No Extra Command Line Argument Passed Other Than Program Name
```

Three arguments : When the above code is compiled and executed with a three arguments, it produces the following output.

```
$ ./a.out First Second Third

Program Name Is: ./a.out

Number Of Arguments Passed: 4

----Following Are The Command Line Arguments Passed----

argv[0]: ./a.out

argv[1]: First

argv[2]: Second

argv[3]: Third
```

Single Argument : When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following output.

```
$ ./a.out "First Second Third"

Program Name Is: ./a.out

Number Of Arguments Passed: 2

----Following Are The Command Line Arguments Passed----

argv[0]: ./a.out

argv[1]: First Second Third
```

Single argument in quotes separated by space : When the above code is compiled and executed with a single argument separated by space but inside single quotes, it produces the following output.

```
$ ./a.out 'First Second Third'

Program Name Is: ./a.out

Number Of Arguments Passed: 2

----Following Are The Command Line Arguments Passed----

argv[0]: ./a.out

argv[1]: First Second Third
```

Q. **C Program to Add two numbers using Command Line Arguments**

```
#include<stdio.h>
void main(int argc, char * argv[]) {
  int i, sum = 0;
  if (argc != 3) {
printf("You have forgot to type numbers.");
    exit(1);
  }
  printf("The sum is : ");
  for (i = 1; i<argc; i++)
    sum = sum + atoi(argv[i]);
  printf("%d", sum);

}
```

Output:

```
add1020
The sum is:30
```

Q. **C Program to find factorial of a number using Command Line Arguments**

```
// C program to find factorial of a number
// using command line arguments

#include <stdio.h>
#include <stdlib.h> /* atoi */

// Function to find factorial of given number
unsigned int factorial(unsigned int n)
{
        int f = 1, i;
        for (i = 2; i<= n; i++)
                f =f*i;
        return f;
}

// Driver code
int main(int argc, char* argv[])
{

        int num, res = 0;

        // Check if the length of args array is 1
        if (argc == 1)
        {
                printf("No command line arguments found.\n");
                printf("\n Ex: cmd_factorial 5");
        }
        else {

                // Get the command line argument and
                // Convert it from string type to integer type
                // using function "atoi( argument)"
                num = atoi(argv[1]);

                // Find the factorial
                printf("%d\n", factorial(num));
        }
        return 0;
}
```

OUTPUT:

```
No command line arguments found.
 Ex: cmd_factorial 5
```

OUTPUT:

```
G:\Record_Observation\cprgs>cmd_factorial 6
720
```

Q. **C Program to find LCM, GCM of two numbers using Command Line Arguments**

```
#include<stdio.h>
#include<conio.h>
int main(int argc,char *argv[])
{
        int a,b,hcf=1,lcm,i,j;

        if(argc!=3)
        {
                printf("\n please use programname value1 value2\n");
                return-1;
        }
        a=atoi(argv[1]);
        b=atoi(argv[2]);
        j=(a<b)?a:b;
        for(i=1;i<=j;i++)
        {
                if(a%i==0 &&b%i==0)
                {
                        hcf=i;
                }
        }
        lcm=(a*b)/hcf;
        printf("\nThe LCM of %d and %d is %d",a,b,lcm);
        printf("\nThe HCF is %d",hcf);
return(0);
}
```

OUTPUT:

```
G:\Record_Observation\cprgs>cmd_factors 4
please use program name value1 value2
```

OUTPUT:

```
G:\Record_Observation\cprgs>cmd_factors 4 5
The LCM of 4 and 5 is 20
The HCF is 1
```

Q. **C Program to find sum of n numbers using Command Line Arguments**

```
#include <stdio.h>

int main(int argc, char *argv[])
{
	int a,b,sum;
	int i; //for loop counter

	if(argc<2)
	{	printf("No command line arguments found.\n");
		printf("please use \"prg_name value1 value2 ... \"\n");
		return -1;
	}
	sum=0;
	for(i=1; i<argc; i++)
	{
		printf("arg[%2d]: %d\n",i,atoi(argv[i]));
		sum += atoi(argv[i]);
	}
	printf("SUM of all values: %d\n",sum);
	return 0;
}
```

OUTPUT:

```
No command line arguments found.
please use "prg_name value1 value2 ... "
```

OUTPUT:

```
G:\Record_Observation\cprgs>cmd_sum_of_n 10 5 20 -5 30
arg[ 1]: 10
arg[ 2]: 5
arg[ 3]: 20
arg[ 4]: -5
arg[ 5]: 30
SUM of all values: 60
```

Unit 3

Sorting and Searching: Sorting by selection, sorting by exchange, sorting by insertion, sorting by partitioning, binary search.

Structures: Basics of structures, structures and functions, arrays of structures, pointers to structures, self-referential structures, table lookup, typedef, unions, bit-fields.

Some other Features: Variable-length argument lists, formatted input-Scanf, file access, Error handling-stderr and exit, Line Input and Output, Miscellaneous Functions.

SORTING AND SEARCHING

Sorting By Exchange: Bubble Sort, Quick Sort
Sorting By Selection: Selection Sort, Heap sort
Sorting By Insertion: Insertion Sort, Shell Sort
Sorting By Partition: Quick Sort, Merge Sort

Bubble Sort-

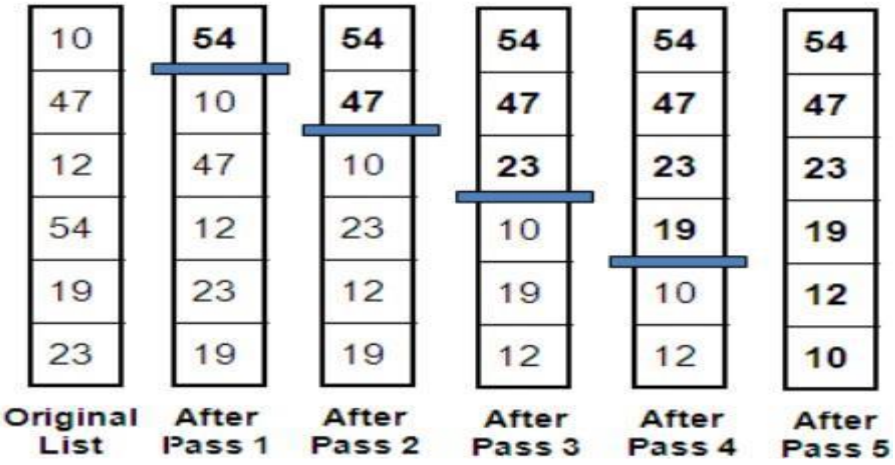
- Bubble sort is the easiest sorting algorithm to implement.
- It is inspired by observing the behavior of air bubbles over foam.
- It is an in-place sorting algorithm.
- It uses no auxiliary data structures (extra space) while sorting.

How Bubble Sort Works?

- Bubble sort uses multiple passes (scans) through an array.
- In each pass, bubble sort compares the adjacent elements of the array.
- It then swaps the two elements if they are in the wrong order.
- In each pass, bubble sort places the next largest element to its proper position.
- In short, it bubbles down the largest element to its correct position.

Ex:- A list of unsorted elements are: 10 47 12 54 19 23

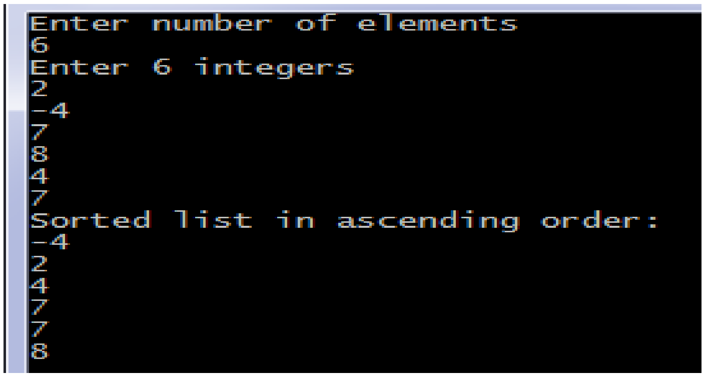
(Bubble up for highest value shown here)



Original List	After Pass 1	After Pass 2	After Pass 3	After Pass 4	After Pass 5
10	54	54	54	54	54
47	10	47	47	47	47
12	47	10	23	23	23
54	12	23	10	19	19
19	23	12	19	10	12
23	19	19	12	12	10

Program:

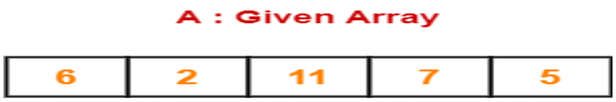
```
/* Bubble sort code */
#include  <stdio.h>
int main()
{
  int array[100], n, c, d, swap;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 0 ; c < n - 1; c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (array[d] > array[d+1]) /* For decreasing order use '<' instead of '>' */
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
     printf("%d\n", array[c]);
  return 0;
}
```



```
Enter number of elements
6
Enter 6 integers
2
-4
7
8
4
7
Sorted list in ascending order:
-4
2
4
7
7
8
```

Bubble Sort Example-

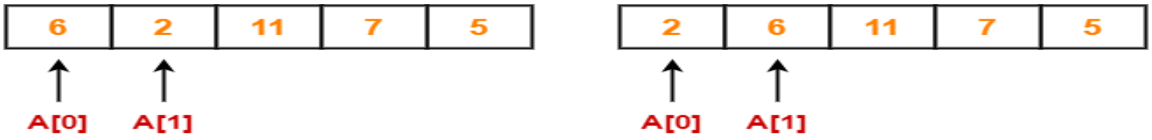
Consider the following array A-



Now, we shall implement the above bubble sort algorithm on this array.

Step-01:

- **We have pass=1 and i=0.**
- **We perform the comparison A[0] > A[1] and swaps if the 0th element is greater than the 1th element.**
- **Since 6 > 2, so we swap the two elements.**



Before Swapping

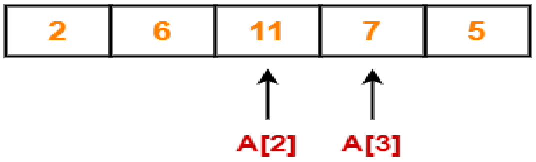
After Swapping

Step-02:

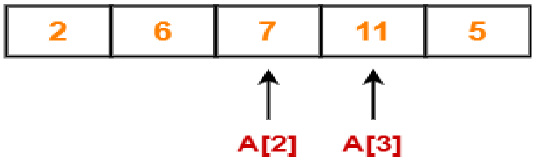
- **We have pass=1 and i=1.**
- **We perform the comparison A[1] > A[2] and swaps if the 1th element is greater than the 2th element.**
- **Since 6 < 11, so no swapping is required.**

Step-03:

- We have pass=1 and i=2.
- We perform the comparison A[2] > A[3] and swaps if the 2nd element is greater than the 3rd element.
- Since 11 > 7, so we swap the two elements.



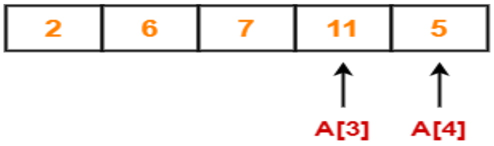
Before Swapping



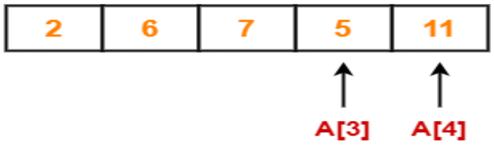
After Swapping

Step-04:

- We have pass=1 and i=3.
- We perform the comparison A[3] > A[4] and swaps if the 3rd element is greater than the 4th element.
- Since 11 > 5, so we swap the two elements.



Before Swapping



After Swapping

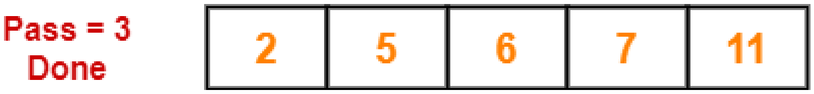
Finally after the first pass, we see that the largest element 11 reaches its correct position.

Step-05:

- Similarly after pass=2, element 7 reaches its correct position.
- The modified array after pass=2 is shown below-

Step-06:

- Similarly after pass=3, element 6 reaches its correct position.
- The modified array after pass=3 is shown below-

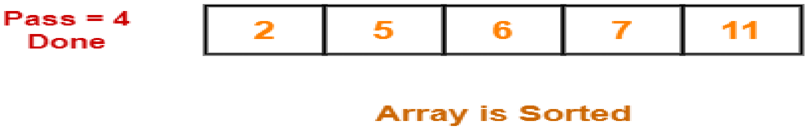


Pass = 3 Done

2	5	6	7	11

Step-07:

- No further improvement is done in pass=4.
- This is because at this point, elements 2 and 5 are already present at their correct positions.
- The loop terminates after pass=4.
- Finally, the array after pass=4 is shown below-



Pass = 4 Done

2	5	6	7	11

Array is Sorted

Time Complexity Analysis-

- Bubble sort uses two loops- inner loop and outer loop.
- The inner loop deterministically performs O(n) comparisons.

The following table summarizes the time complexities of bubble sort in each case-

	Time Complexity
Best Case	O(n)
Average Case	$\Theta(n^2)$
Worst Case	$O(n^2)$

Problem-01:

The number of swapping needed to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order using bubble sort is- (ISRO CS 2017)

1. 11
2. 12
3. 13
4. 10

Solution-

In bubble sort, Number of swaps required = Number of inversion pairs.
Here, there are 10 inversion pairs present which are-

1. (8,7)
2. (22,7)
3. (22,9)
4. (8,5)
5. (22,5)
6. (7,5)
7. (9,5)
8. (31,5)
9. (22,13)
10. (31,13)

Thus, Option (D) is correct.

Selection Sort-

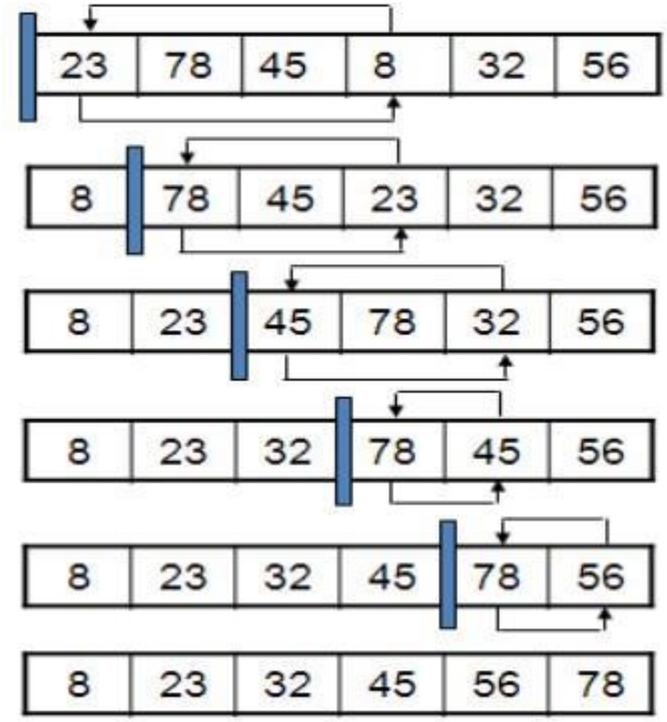
- Selection sort is one of the easiest approaches to sorting.
- It is inspired from the way in which we sort things out in day to day life.
- It is an in-place sorting algorithm because it uses no auxiliary data structures while sorting.

How Selection Sort Works?

Consider the following elements are to be sorted in ascending order using selection sort-

- It finds the first smallest element (2).
- It swaps it with the first element of the unordered list.
- It finds the second smallest element (5).
- It swaps it with the second element of the unordered list.
- Similarly, it continues to sort the given elements.

Ex:- A list of unsorted elements are: 23 78 45 8 32 56



23 78 45 8 32 56

8 78 45 23 32 56

8 23 45 78 32 56

8 23 32 78 45 56

8 23 32 45 78 56

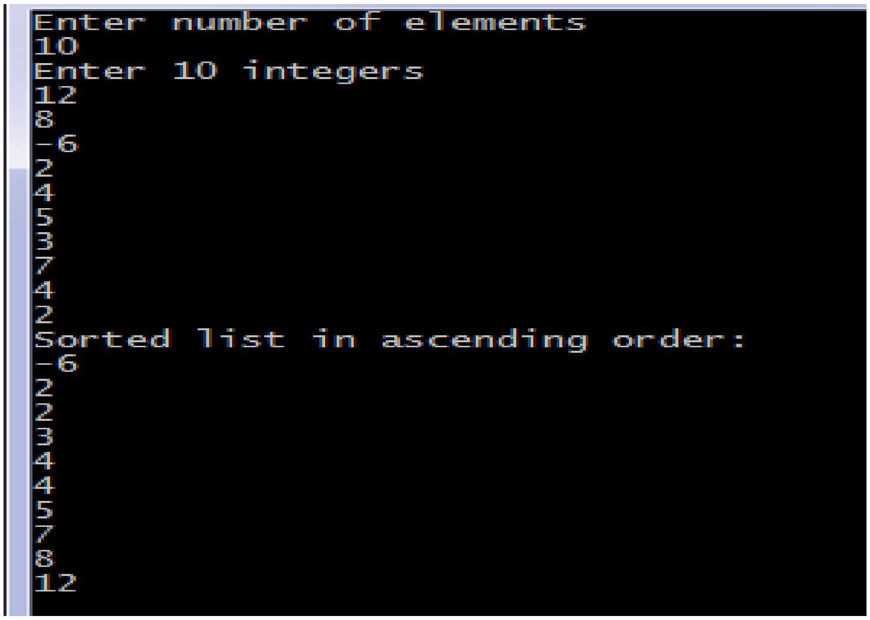
8 23 32 45 56 78

A list of sorted elements now : 8 23 32 45 56 78

As a result, sorted elements in ascending order are-
2, 5, 6, 7, 11

Program:

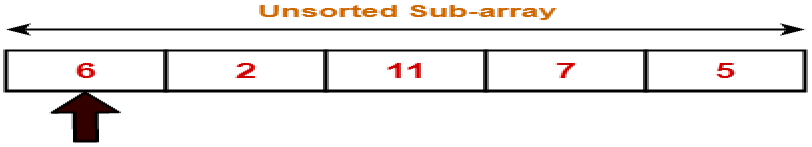
```
#include <stdio.h>
int main()
{
  int array[100], n, c, d, position, t;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 0; c < (n - 1); c++) // finding minimum element (n-1) times
  {
    position = c;
    for (d = c + 1; d < n; d++)
    {
      if (array[position] > array[d])
        position = d;
    }
    if (position != c)
    {
      t = array[c];
      array[c] = array[position];
      array[position] = t;
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
    printf("%d\n", array[c]);
  return 0;
}
```



```
Enter number of elements
10
Enter 10 integers
12
8
-6
2
4
5
3
7
4
2
Sorted list in ascending order:
-6
2
2
3
4
4
5
7
8
12
```

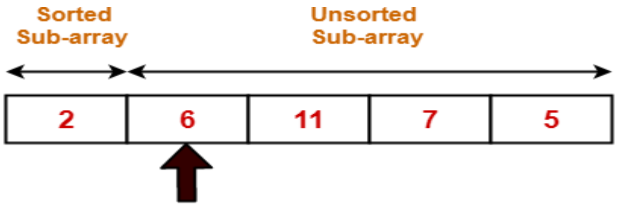
The above selection sort algorithm works as illustrated below-

Step-01: For i = 0



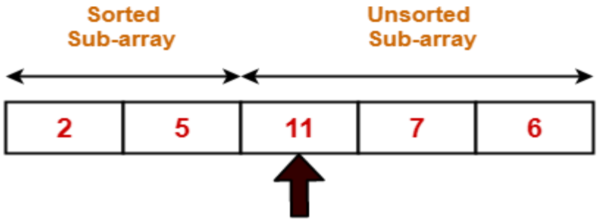
We start here, find the minimum element and swap it with the 1st element of array

Step-02: For i = 1



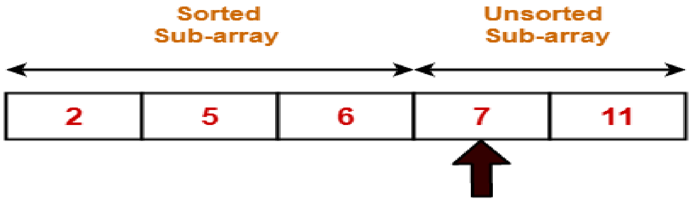
We start here, find the minimum element and swap it with the 2nd element of array

Step-03: For i = 2



We start here, find the minimum element and swap it with the 3rd element of array

Step-04: For i = 3

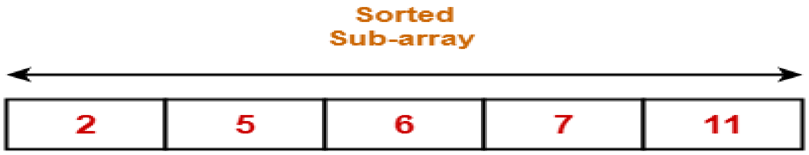


We start here, find the minimum element but there is no need to swap
(4th element is itself the minimum)

Step-05: For i = 4

Loop gets terminated as 'i' becomes 4.

The state of array after the loops are finished is as shown-



With each loop cycle,

- The minimum element in unsorted sub-array is selected.
- It is then placed at the correct location in the sorted sub-array until array A is completely sorted.

Time Complexity Analysis-

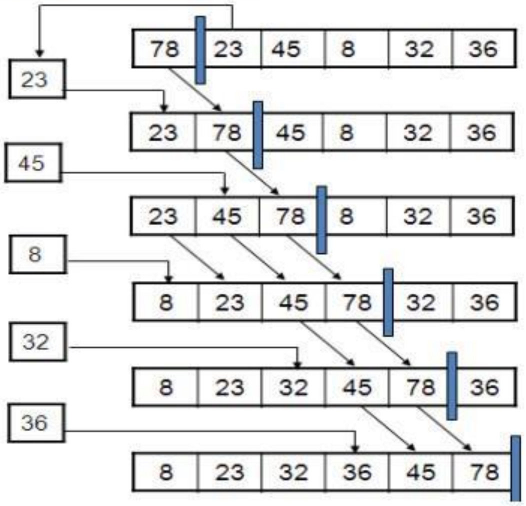
- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has O(n^2) time complexity.

	Time Complexity
Best Case	n^2
Average Case	n^2
Worst Case	n^2

Insertion Sort-

- Insertion sort is an in-place sorting algorithm.
- It uses no auxiliary data structures while sorting.
- It is inspired from the way in which we sort playing cards.

Ex:- A list of unsorted elements are: 78 23 45 8 32 36 . The results of insertion sort for each pass is as follows:-



A list of sorted elements now : 8 23 32 36 45 78

Program:

/ Insertion sort ascending order */*

```
#include <stdio.h>
int main()
{
  int n, array[1000], c, d, t, flag = 0;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
  scanf("%d", &array[c]);
  for (c = 1 ; c <= n - 1; c++) {
    t = array[c];
    for (d = c - 1 ; d >= 0; d--) {
      if (array[d] > t) {
      array[d+1] = array[d];
      flag = 1;
      }
      else
        break;
    }
    if (flag)
      array[d+1] = t;
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c <= n - 1; c++) {
    printf("%d\n", array[c]);
  }
  return 0;
}
```

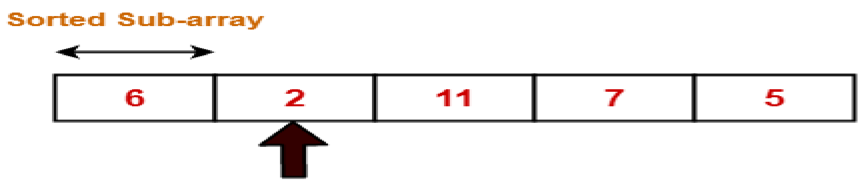
Insertion Sort Example-

Consider the following elements are to be sorted in ascending order-

6, 2, 11, 7, 5

The above insertion sort algorithm works as illustrated below-

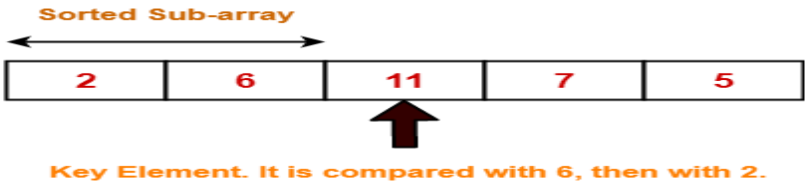
Step-01: For i = 1



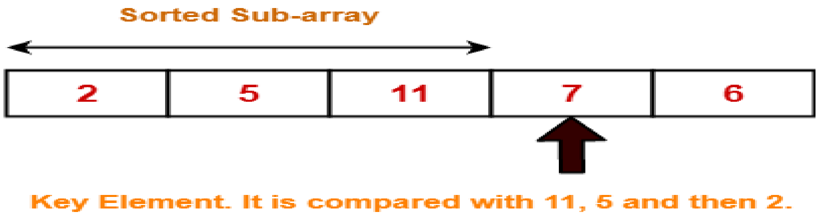
6	2	11	7	5

Key Element. We compare it with all elements of sorted sub-array.

Step-02: For i = 2



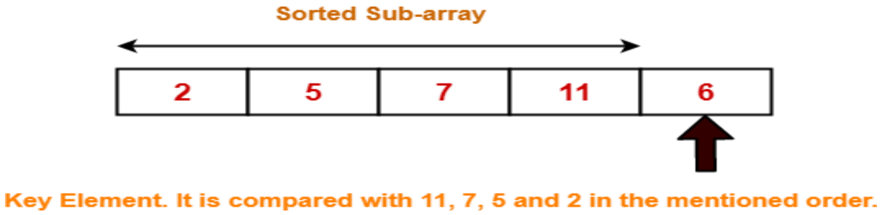
Step-03: For i = 3



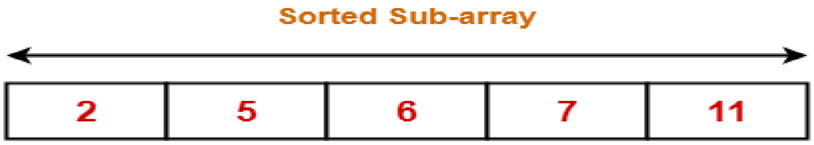
2	5	11	7	6		For j = 2; 11 > 7 so A[3] = 11
2	5	11	11	6		For j = 1; 5 < 7 so loop stops and A[2] = 7
2	5	7	11	6		After inner loop ends

Working of inner loop when i = 3

Step-04: For i = 4


Sorted Sub-array
2
5
7
11
6
Key Element. It is compared with 11, 7, 5 and 2 in the mentioned order.

Loop gets terminated as 'i' becomes 5. The state of array after the loops are finished-



With each loop cycle,

- One element is placed at the correct location in the sorted sub-array until array A is completely sorted.

Time Complexity Analysis-

- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has O(n^2) time complexity.

	Time Complexity
Best Case	n
Average Case	n2
Worst Case	n2

Quick Sort-

- Quick Sort is a famous sorting algorithm.
- It sorts the given data items in ascending order.
- It uses the idea of divide and conquer approach.
- It follows a recursive algorithm.

How Does Quick Sort Works?

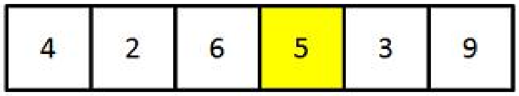
- Quick Sort follows a recursive algorithm.
- It divides the given array into two sections using a partitioning element called as pivot.

The division performed is such that-

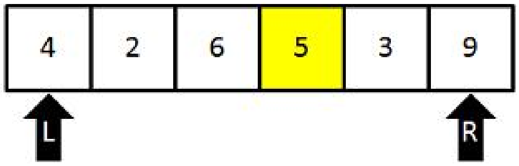
- All the elements to the left side of pivot are smaller than pivot.
- All the elements to the right side of pivot are greater than pivot.

After dividing the array into two sections, the pivot is set at its correct position.
Then, sub arrays are sorted separately by applying quick sort algorithm recursively.

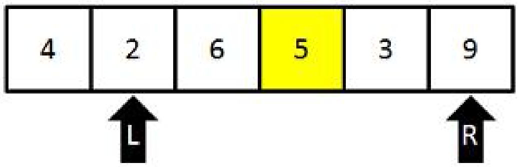
Step 1
Determine pivot



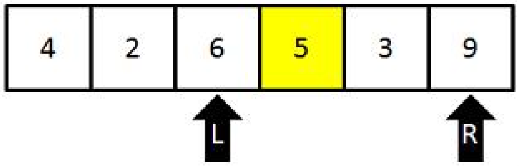
Step 2
Start pointers at left and right



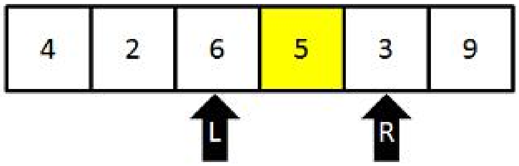
Step 3
Since 4 < 5, shift left pointer



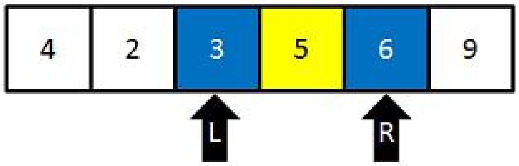
Step 4
Since 2 < 5, shift left pointer
Since 6 > 5, stop



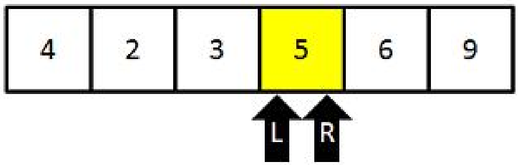
Step 5
Since 9 > 5, shift right pointer
Since 3 < 5, stop



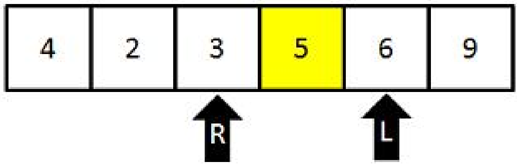
Step 6
Swap values at pointers



Step 7
Move pointers one more step



Step 8
Since 5 == 5,
move pointers one more step
Stop



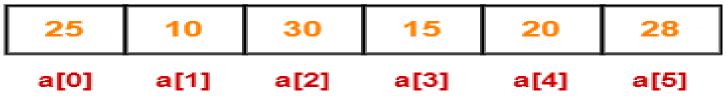
Program:

```
// Quick sort in C

#include <stdio.h>

// function to swap elements
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}

// function to find the partition position
int partition(int array[], int low, int high) {

  // select the rightmost element as pivot
  int pivot = array[high];

  // pointer for greater element
  int i = (low - 1);

  // traverse each element of the array
  // compare them with the pivot
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {

      // if element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;

      // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
  }

  // swap the pivot element with the greater element at i
  swap(&array[i + 1], &array[high]);

  // return the partition point
  return (i + 1);
}
```

```
void quickSort(int array[], int low, int high) {
  if (low < high) {

    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot
    int pi = partition(array, low, high);

    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);

    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}

// function to print array elements
void printArray(int array[], int size) {
for (int i = 0; i < size; ++i) {
    printf("%d ", array[i]);
  }
  printf("\n");
}

// main function
int main() {
  int data[] = {8, 7, 2, 1, 0, 9, 6};

  int n = sizeof(data) / sizeof(data[0]);

  printf("Unsorted Array\n");
  printArray(data, n);

  // perform quicksort on data
  quickSort(data, 0, n - 1);

  printf("Sorted array in ascending order: \n");
  printArray(data, n);
}
```

Quick Sort Example-

Consider the following array has to be sorted in ascending order using quick sort algorithm-



Quick Sort Example

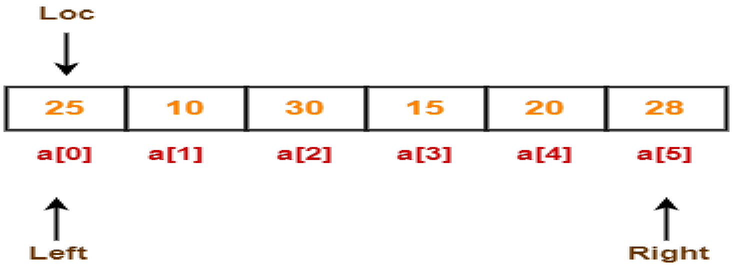
Quick Sort Algorithm works in the following steps-

Step-01:

Initially-

- **Left** and **Loc** (pivot) points to the first element of the array.
- **Right** points to the last element of the array.

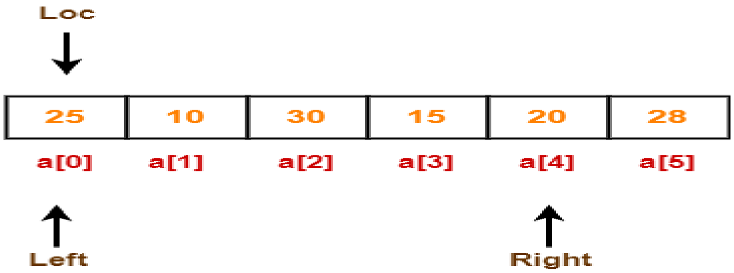
So to begin with, we set **loc** = 0, **left** = 0 and **right** = 5 as-



Step-02:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] < a[right], so algorithm moves **right** one position towards left as-

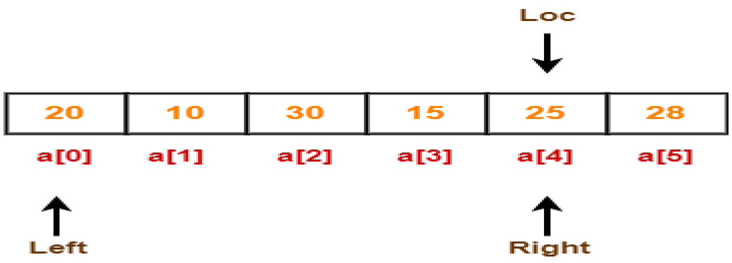


Now, **loc** = 0, **left** = 0 and **right** = 4.

Step-03:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as-

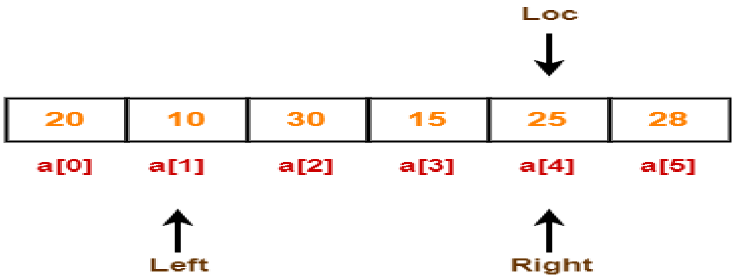


Now, **loc** = 4, **left** = 0 and **right** = 4.

Step-04:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

As a[loc] > a[left], so algorithm moves **left** one position towards right as-

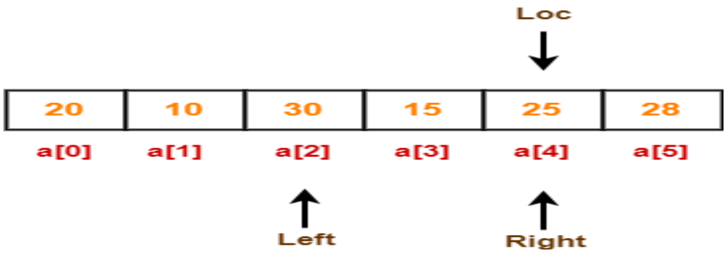


Now, **loc** = 4, **left** = 1 and **right** = 4.

Step-05:

Since **loc** points at right, so algorithm starts from **left** and move towards right.

As a[loc] > a[left], so algorithm moves **left** one position towards right as-

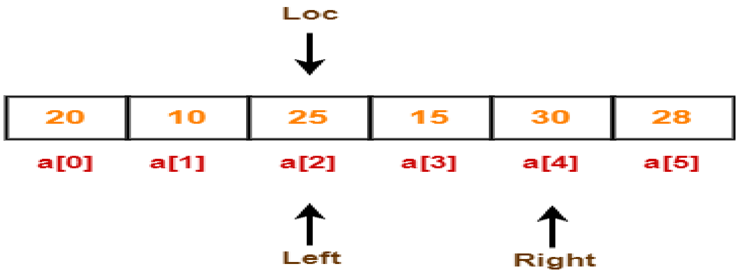


Now, **loc** = 4, **left** = 2 and **right** = 4.

Step-06:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

As a[loc] < a[left], so we algorithm swaps a[loc] and a[left] and **loc** points at **left** as-

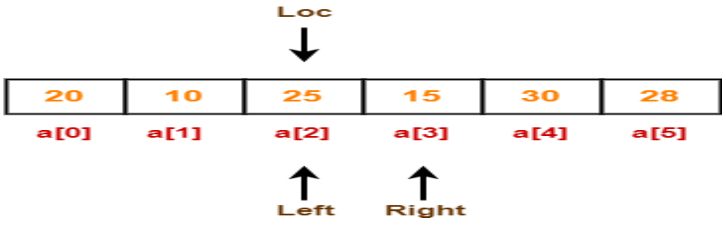


Now, **loc** = 2, **left** = 2 and **right** = 4.

Step-07:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] < a[right], so algorithm moves **right** one position towards left as-

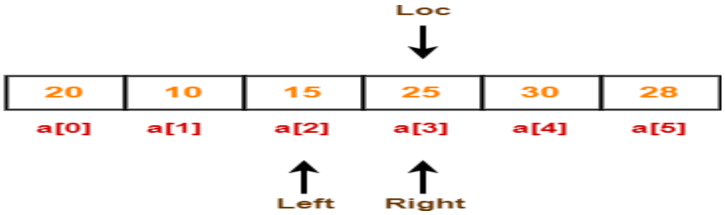


Now, **loc** = 2, **left** = 2 and **right** = 3.

Step-08:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as-

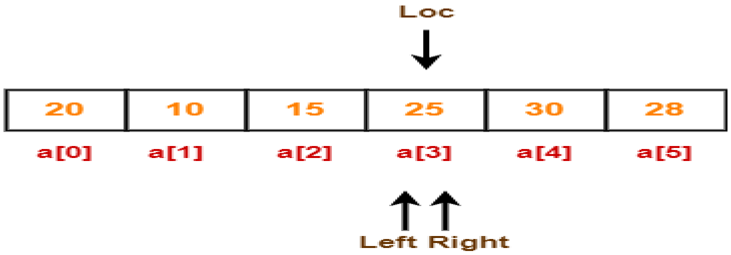


Now, **loc** = 3, **left** = 2 and **right** = 3.

Step-09:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

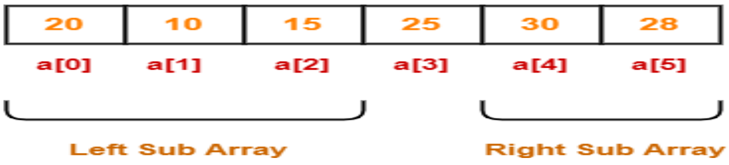
As a[loc] > a[left], so algorithm moves **left** one position towards right as-



Now, **loc** = 3, **left** = 3 and **right** = 3.

Now,

- **loc**, **left** and **right** points at the same element.
- This indicates the termination of procedure.
- The pivot element 25 is placed in its final position.
- All elements to the right side of element 25 are greater than it.
- All elements to the left side of element 25 are smaller than it.



Now, quick sort algorithm is applied on the left and right sub arrays separately in the similar manner.

Advantages of Quick Sort-

The advantages of quick sort algorithm are-

- Quick Sort is an in-place sort, so it requires no temporary memory.
- Quick Sort is typically faster than other algorithms.

(because its inner loop can be efficiently implemented on most architectures)

- Quick Sort tends to make excellent usage of the memory hierarchy like virtual memory or caches.
- Quick Sort can be easily parallelized due to its divide and conquer nature.

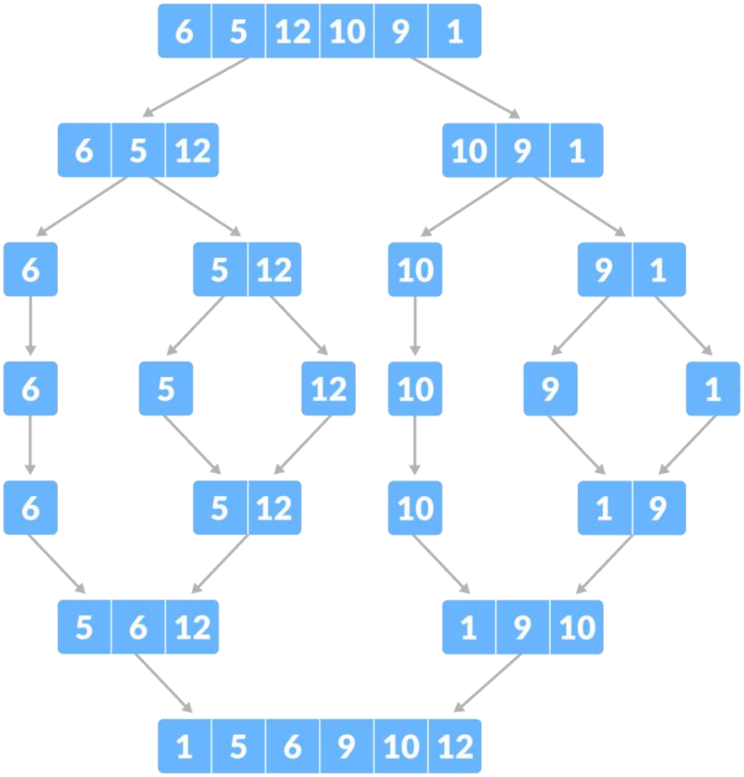
Merge Sort-

- Merge sort is a famous sorting algorithm.
- It uses a divide and conquer paradigm for sorting.
- It divides the problem into sub problems and solves them individually.
- It then combines the results of sub problems to get the solution of the original problem.

Merge Sort Algorithm-

Merge Sort Algorithm works in the following steps-

- It divides the given unsorted array into two halves- left and right sub arrays.
- The sub arrays are divided recursively.
- This division continues until the size of each sub array becomes 1.
- After each sub array contains only a single element, each sub array is sorted trivially.
- Then, the above discussed merge procedure is called.
- The merge procedure combines these trivially sorted arrays to produce a final sorted array.



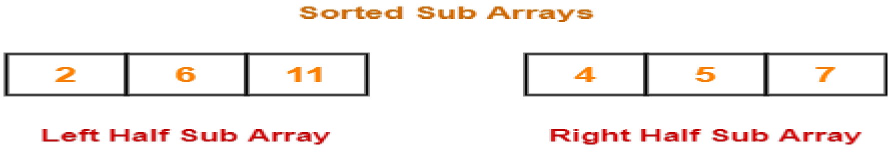
Program:

```
// Merge sort in C

#include <stdio.h>

// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {

  // Create L ← A*p..q+ and M ← A*q+1..r+
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  // Maintain current index of sub-arrays and main array
  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  // Until we reach either end of either L or M, pick larger among
  // elements L and M and place them in the correct position at A[p..r]
  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}

// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r) {
  if (l < r) {

    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}

// Print the array
void printArray(int arr[], int size) {
  for (int i = 0; i < size; i++)
  printf("%d ", arr[i]);
  printf("\n");
}

// Driver program
int main() {
  int arr[] = {6, 5, 12, 10, 9, 1};
  int size = sizeof(arr) / sizeof(arr[0]);

  mergeSort(arr, 0, size - 1);

  printf("Sorted array: \n");
  printArray(arr, size);
}
```

How Merge Sort Works?

The merge procedure of merge sort algorithm is used to merge two sorted arrays into a third array in sorted order.

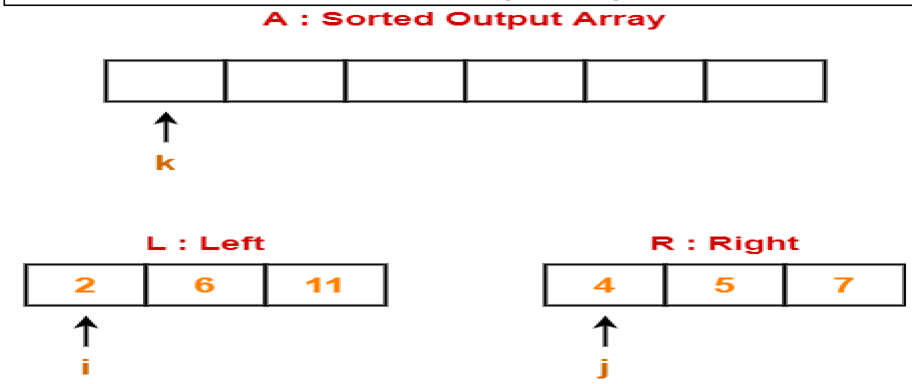
Consider we want to merge the following two sorted sub arrays into a third array in sorted order-



The above merge procedure of merge sort algorithm is explained in the following steps-

Step-01:

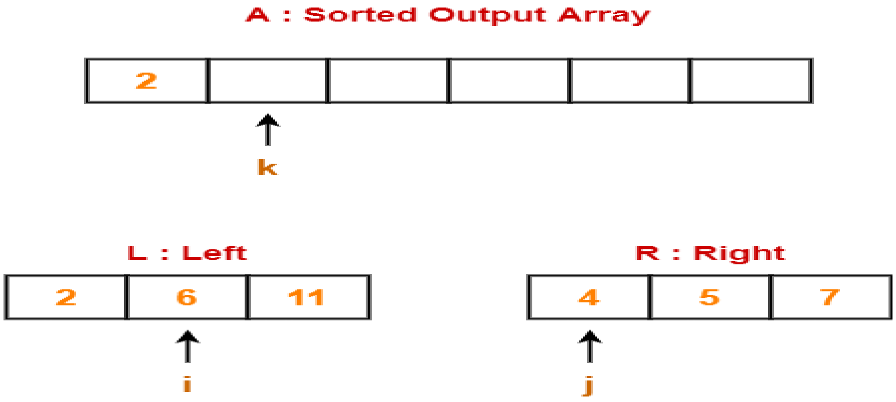
- Create two variables i and j for left and right sub arrays.
- Create variable k for sorted output array.



Step-02:

- We have i = 0, j = 0, k = 0.
- Since L[0] < R[0], so we perform A[0] = L[0] i.e. we copy the first element from left sub array to our sorted output array.
- Then, we increment i and k by 1.

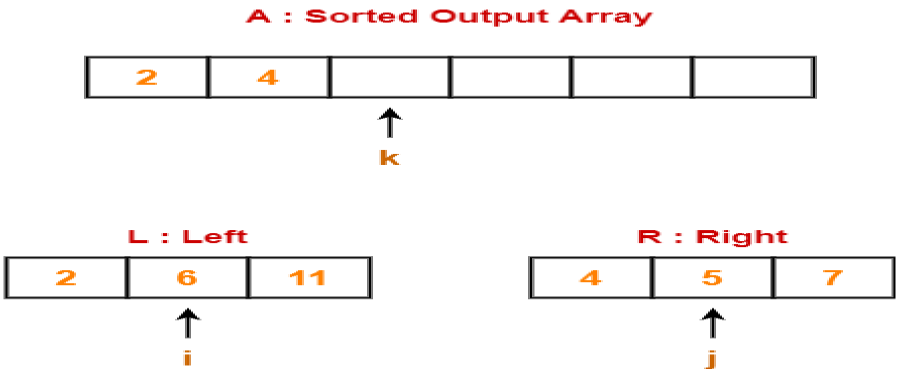
Then, we have-



Step-03:

- We have i = 1, j = 0, k = 1.
- Since L[1] > R[0], so we perform A[1] = R[0] i.e. we copy the first element from right sub array to our sorted output array.
- Then, we increment j and k by 1.

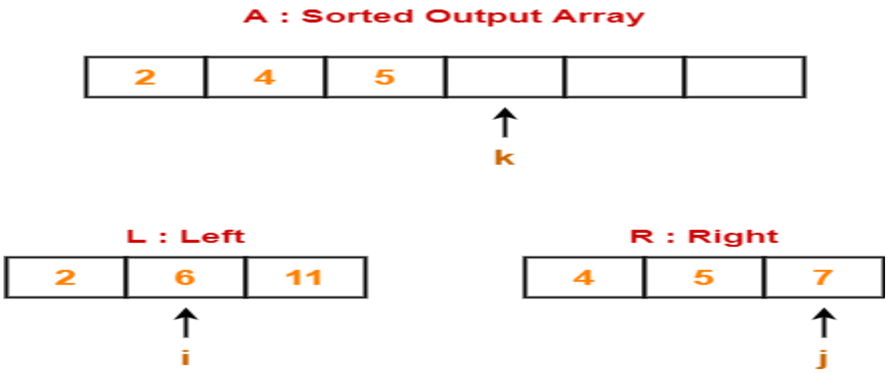
Then, we have-



Step-04:

- We have i = 1, j = 1, k = 2.
- Since L[1] > R[1], so we perform A[2] = R[1].
- Then, we increment j and k by 1.

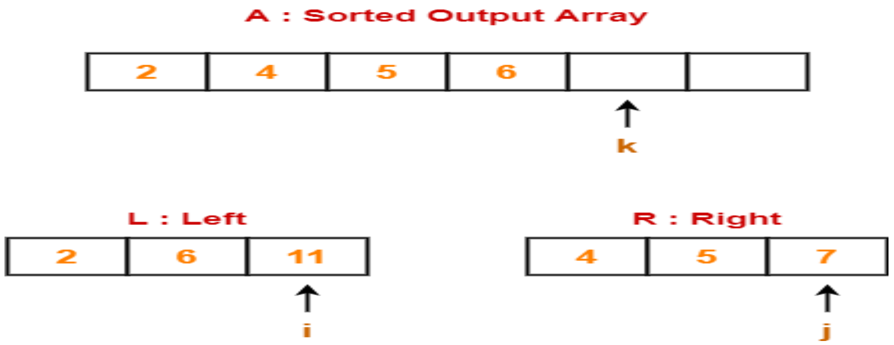
Then, we have-



Step-05:

- We have i = 1, j = 2, k = 3.
- Since L[1] < R[2], so we perform A[3] = L[1].
- Then, we increment i and k by 1.

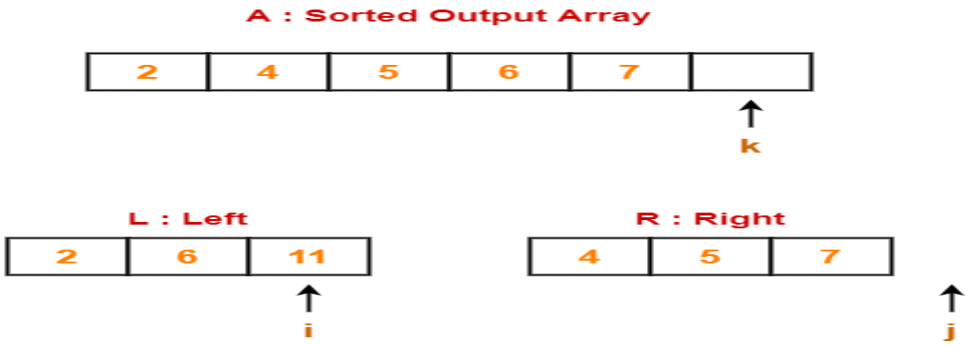
Then, we have-



Step-06:

- We have i = 2, j = 2, k = 4.
- Since L[2] > R[2], so we perform A[4] = R[2].
- Then, we increment j and k by 1.

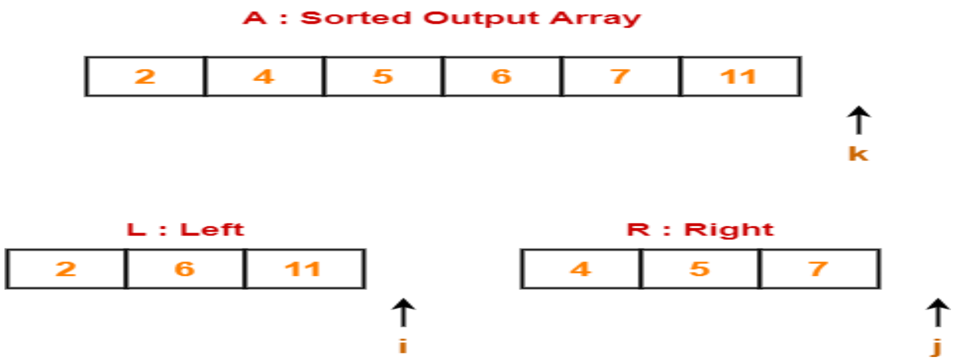
Then, we have-



Step-07:

- Clearly, all the elements from right sub array have been added to the sorted output array.
- So, we exit the first while loop with the condition while(i<nL && j<nR) since now j>nR.
- Then, we add remaining elements from the left sub array to the sorted output array using next while loop.

Finally, our sorted output array is-



Basically,

- After finishing elements from any of the sub arrays, we can add the remaining elements from the other sub array to our sorted output array as it is.
- This is because left and right sub arrays are already sorted.

Time Complexity

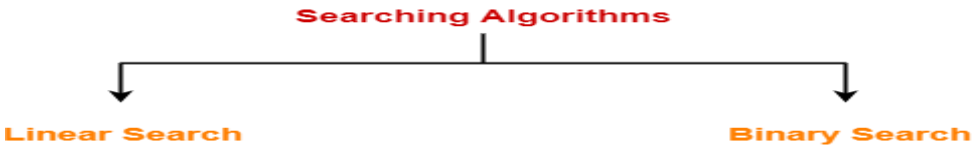
The above mentioned merge procedure takes Θ(n) time.
This is because we are just filling an array of size n from left & right sub arrays by incrementing i and j at most Θ(n) times.

Searching Algorithms

- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.

Searching Algorithms-

Searching Algorithms are a family of algorithms used for the purpose of searching.
The searching of an element in the given array may be carried out in the following two ways-



1. Linear Search
2. Binary Search

Linear Search-

-

- **Linear Search is the simplest searching algorithm.**
- **It traverses the array sequentially to locate the required element.**

- **It searches for an element by comparing it with each element of the array one by one.**
- **So, it is also called as Sequential Search.**

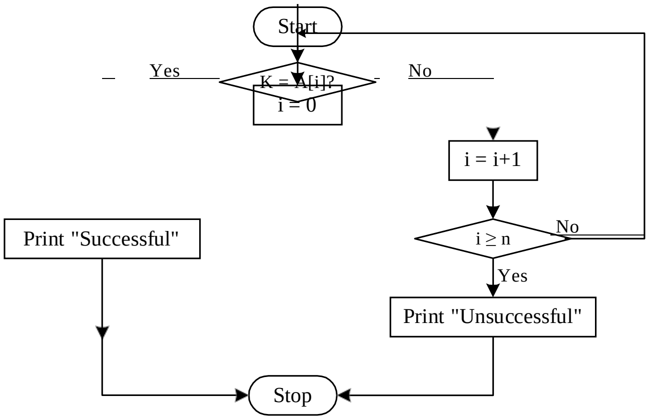
-

Linear Search Algorithm is applied when-

- **No information is given about the array.**
- **The given array is unsorted or the elements are unordered.**
- **The list of data items is smaller.**

-

Linear Search Algorithm-



-

Consider-

- **There is a linear array 'a' of size 'n'.**
- **Linear search algorithm is being used to search an element 'item' in this linear array.**
- **If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.**

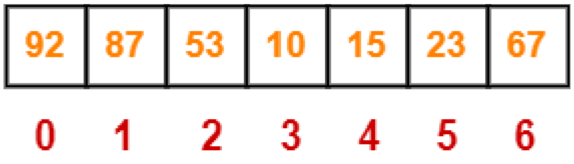
Linear Search Example-

-

Consider-

- **We are given the following linear array.**
- **Element 15 has to be searched in it using Linear Search Algorithm.**

-



92	87	53	10	15	23	67
0	1	2	3	4	5	6

LInear Search Example

Now,

- Linear Search algorithm compares element 15 with all the elements of the array one by one.
- It continues searching until either the element 15 is found or all the elements are searched.

Linear Search Algorithm works in the following steps-

Step-01:

- It compares element 15 with the 1st element 92.
- Since 15 ≠ 92, so required element is not found.
- So, it moves to the next element.

Step-02:

- It compares element 15 with the 2nd element 87.
- Since 15 ≠ 87, so required element is not found.
- So, it moves to the next element.

Step-03:

- It compares element 15 with the 3rd element 53.
- Since 15 ≠ 53, so required element is not found.
- So, it moves to the next element.

Step-04:

- It compares element 15 with the 4th element 10.
- Since 15 ≠ 10, so required element is not found.
- So, it moves to the next element.

Step-05:

- It compares element 15 with the 5th element 15.

- **Since 15 = 15, so required element is found.**
- **Now, it stops the comparison and returns index 4 at which element 15 is present.**

Time Complexity of Linear Search Algorithm is O(n).

Binary Search-

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.

Binary Search Algorithm can be applied only on **Sorted arrays**.

So, the elements must be arranged in-

- Either ascending order if the elements are numbers.
- Or dictionary order if the elements are strings.

To apply binary search on an unsorted array,

- First, sort the array using some sorting technique.
- Then, use binary search algorithm.

Binary Search Algorithm-

Consider-

- There is a linear array ‘a’ of size ‘n’.
- Binary search algorithm is being used to search an element ‘item’ in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.
- Variables beg and end keeps track of the index of the first and last element of the array or sub array in which the element is being searched at that instant.
- Variable mid keeps track of the index of the middle element of that array or sub array in which the element is being searched at that instant.

Explanation

Binary Search Algorithm searches an element by comparing it with the middle most element of the array.
Then, following three cases are possible-

Case-01

If the element being searched is found to be the middle most element, its index is returned.

Case-02

If the element being searched is found to be greater than the middle most element,
then its search is further continued in the right sub array of the middle most element.

Case-03

If the element being searched is found to be smaller than the middle most element,
then its search is further continued in the left sub array of the middle most element.

This iteration keeps on repeating on the sub arrays until the desired element is found
or size of the sub array reduces to zero.

Time Complexity Analysis-

Binary Search time complexity analysis is done below-

- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

Thus, we have-

Time Complexity of Binary Search Algorithm is O($\log_2 n$).
Here, n is the number of elements in the sorted linear array.

This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

BINARY SEARCH EXAMPLE: Perform Binary Search on below list for finding value 98.

4	21	36	44	62	75	89	92	95	97	98

Step 1: Mark low, high and index numbers on the list

4	**21**	**36**	**44**	**62**	**75**	**89**	**92**	**95**	**97**	**98**	
0	1	2	3	4	5	6	7	8	9	10	← indexes or pointers
↑ low										↑ high	

Step 2: $Find\ mid = \frac{low+high}{2} = \frac{0+10}{2} = 5. Note\ that\ 5\ is\ the\ index\ value\ of\ element\ 75$

4	**21**	**36**	**44**	**62**	**75**	**89**	**92**	**95**	**97**	**98**	
0	1	2	3	4	5	6	7	8	9	10	← indexes or pointers
↑ low					↑ mid					↑ high	

Step 3: Compare 98 (Number to be searched) & 75. If they are equal search is completed.

98 ≠ 75.

Also 98 > 75, so cancel 75 and below elements from original list. The new list is as follows:

89	**92**	**95**	**97**	**98**
6	7	8	9	10

Step 4: Find new values of low, high and mid.

89	**92**	**95**	**97**	**98**
6	7	8	9	10
↑ low		↑ mid		↑ high

low = 6, high = 10

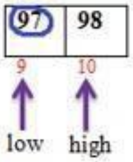
$mid = \frac{low + high}{2} = \frac{6 + 10}{2} = \frac{16}{2} = 8. Note\ that\ 8\ is\ the\ index\ value\ of\ element\ 95$

Step 5: Compare 98 (Number to be searched) & 95.

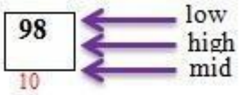
98 > 95. So cancel 95 and below elements. New list will be as follows:

97	**98**
9	10

Step 6: Find low, high and mid. $mid = \frac{9+10}{2} = 9.5 \approx 9. \mathit{Note\ that\ 9\ is\ the\ index\ value\ of\ element\ 97}$



Step 7: Compare target value 98 with 97. 98 > 97. So cancel 97 and below. New list is as follows:



Step 8: Find low = high = mid = 10. Compare target value 98 with 98. 98 = 98. The target value is FOUND. **DONE**.

SUMMARY OF ABOVE STEPS IS GIVEN IN BELOW TABLE

Pass	low	high	mid-point	mid value
1	0	10	5	**75**
2	6	10	8	**95**
3	9	10	9	**97**
4	10	10	10	**98**

Structures

Scalar variables can hold one piece of information and arrays can hold a number of pieces of information of the same datatype. These two data types can handle a great variety of situations. But quite often we deal with entities that are collection of dissimilar data types.

For example, suppose you want to store data about a book.
You might want to store its name (a string), its price (a float) and number of pages in it(an int). If data about say3 such books is to be stored, then we can follow **two approaches:**

(a) Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages.
(b) Use a structure variable.

In the first approach, the program becomes more difficult to handle as the number of items relating to the book go on increasing. For example, we would be required to use a number of arrays, if we also decide to store name of the publisher, date of purchase of book, etc. To solve this problem, C provides a special data type—the structure.

Defining a structure
`struct` keyword is used to define a structure. `struct` defines a new data type which is a collection of primary and derived data types.
Syntax:

```
struct[structure_tag or structure_name]
{
//member variable 1
//member variable 2
//member variable 3
...
}[structure_variables];
```

As you can see in the syntax above, we start with the `struct` keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like `int`, `float`, `array` etc.
After the closing curly brace, we can specify one or more structure variables, again this is optional.
Note: The closing curly brace in the structure type declaration must be followed by a semicolon(`;`).
Example of Structure

```
struct Student
{
char name[25];
int age;
char branch[10];
// F for female and M for male
char gender;
};
```

Here `struct Student` declares a structure to hold the details of a student which consists of 4 data fields, namely `name`, `age`, `branch` and `gender`. These fields are called structure elements or members.

Each member can have different data type, like in this case, `name` is an array of `char` type and `age` is of `int` type etc. **Student** is the name of the structure and is called as the **structure tag**.

Declaring Structure Variables

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined. **Structure** variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following **two ways:**

1) Declaring Structure variables separately

```
struct Student
{
char name[25];
int age;
char branch[10];
//F for female and M for male
char gender;
};

struct Student S1, S2;//declaring variables of struct Student
```

2) Declaring Structure variables with structure definition

```
struct Student
{
char name[25];
int age;
char branch[10];
//F for female and M for male
char gender;
}S1, S2;
```

Here `S1` and `S2` are variables of structure `Student`. However this approach is not much recommended.

Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

```
struct Patient
{
float height;
int weight;
int age;
};

struct Patient p1 = { 180.75 , 73, 23 };    //initialization
```

OR,

```
struct Patient p1;
p1.height = 180.75;     //initialization of each member separately
p1.weight = 73;
p1.age = 23;
```

Accessing Structure Members:

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot **.** Operator also called **period** or **member access** operator.

For example:

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

struct Student
{
char name[25];
int age;
char branch[10];
//F for female and M for male
char gender;
};

Int main()
{
struct Student s1;
/*s1 is a variable of Student type and age is a member of Student    */
       s1.age  =18;
/*using string function to add name    */
strcpy(s1.name,"seenu");

/*displaying the stored values    */
printf("Name of Student 1: %s\n", s1.name);
printf("Age of Student 1: %d\n", s1.age);

return(0);
}
```

```
Name of Student 1: seenu
Age of Student 1: 18
```

We can also use scanf() to give values to structure members through terminal.

```
scanf(" %s ", s1.name);
scanf(" %d ",&s1.age);
```

DIFFERENCE BETWEEN C VARIABLE, C ARRAY AND C STRUCTURE:

A normal C variable can hold only one data of one data type at a time.

An array can hold group of data of same data type.

A structure can hold group of data of different data types and Data types can be int, char, float, double and long double etc.

C Structure:

Syntax	struct student
	{
	int a;
	char b[10];
	}s1,s2;

Example	S1.a = 10;
	strcpy(s1.b, “Hello”);

C Variable:

int	Syntax: int a;
	Example: a = 20;

char	Syntax: char b; Example: b=’Z’;

C Array:

int	**Syntax:** int a[3];
	Example:
	a[0] = 10;
	a[1] = 20;
	a[2] = 30;

char	**Syntax:** char b[10];
	Example:
	b=”Hello”;

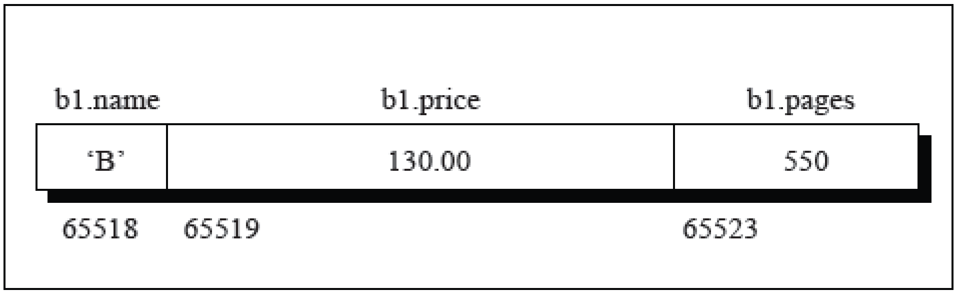
How Structure Elements are Stored

Whatever be the elements of a structure, they are always stored in contiguous memory locations. The following program would illustrate this:

```
main()
{
struct book
{
char name;
float price;

intpages ;
} ;
struct book b1 = {'B', 130.00, 550 } ;
printf ("\nAddress of name =%u",&b1.name );
printf ("\nAddress of price=%u",&b1.price);
printf ("\nAddress of pages=%u",&b1.pages) ;
getch();
}
```

Memory Representation

Actually the structure elements are stored in memory as shown below



<u>ExpectedOutput</u>

Address of name=65518

```
Address of price=65519
Address of pages =65523
```

ExampleProgram

```
voidmain()
{
        struct book
        {
        char name[20];
        float price;
        intpage;
        }b1;

        printf("\nEnter the name, price, pages of book\n");
        scanf("%s%f%d",&b1.name,&b1.price,&b1.page);

        printf("\nBook Name=%s",b1.name);
        printf("\nPrice=%.2f",b1.price);
        printf("\nPages=%d",b1.page);

getch();
}
```

Expected Output

```
Enter the name, price, pages of book

CP
275.50
370

Book Name=CP
Price=275.50
Pages=370
```

C STRUCTURE DECLARATION IN SEPARATE HEADER FILE:

In above structure programs, C structure is declared in main source file. Instead of declaring C structure in main source file, we can have this structure declaration in another file called “header file” and we can include that header file in main source file as shown below.

HEADER FILE NAME – STRUCTURE.H

Before compiling and executing below C program, create a file named “structure.h” and declare the below structure.

```
struct student
{
int id;
char name[20];
float percentage;
} record;
```

MAIN FILE NAME – STRUCTURE.C:

In this program, above created header file is included in “structure.c” source file as #include “structure.h”. So, the structure declared in “structure.h” file can be used in “structure.c” source file.

```
// File name - structure.c
#include <stdio.h>
#include <string.h>
#include "structure.h"  /* header file where C structure is
declared */

int main()
{

   record.id=1;
strcpy(record.name, "Raju");
record.percentage = 86.5;

printf(" Id is: %d \n", record.id);
printf(" Name is: %s \n", record.name);
printf(" Percentage is: %f \n", record.percentage);
return 0;
}
```

OUTPUT:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

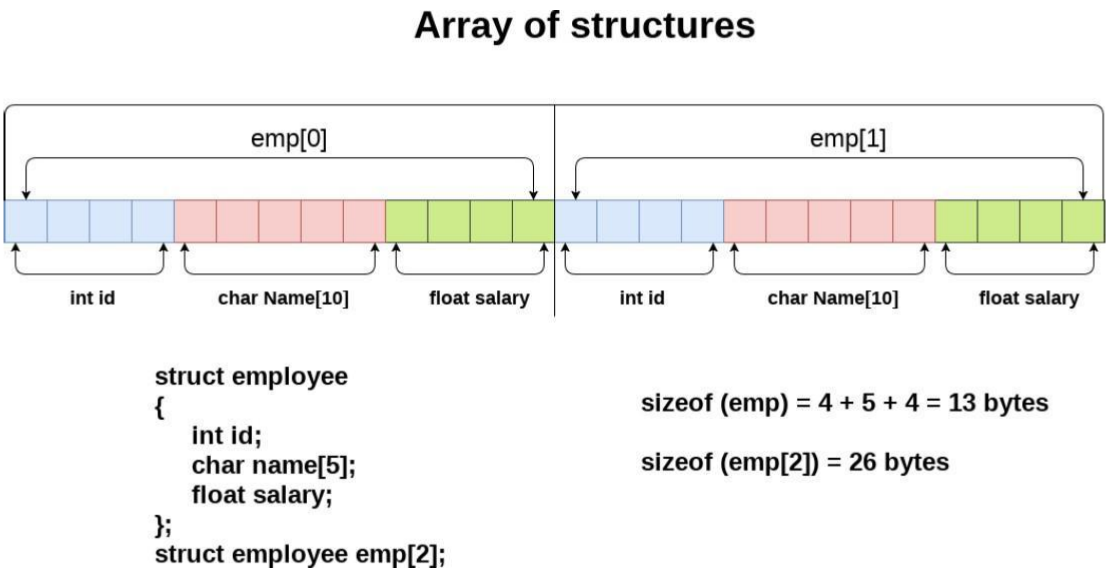
Array of Structures

- If we want to store data of 100 books we would be required to use100 different structure variables from**b1** to **b100**, which is definitely not very convenient.
- A better approach would be to use an array of structures.

- The syntax we use to reference each element of the array **b** is similar to the syntax used for arrays of **int**s and **char**s.
- For example,we refer to zeroth books price as **b[0]. price**. Similarly, we refer first books pages as **b[1].pages**.

ExampleProgram1

```
//Arrayof structures
struct student
{
Int rno;
char name[20];
};
int main()
```



Array of structures

```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

sizeof (emp) = 4 + 5 + 4 = 13 bytes

sizeof (emp[2]) = 26 bytes

```
{
struct student s[2]={{25,"Ritchie"},{20,"Babbage"}};       //structure array initialization
int i;
printf("\n***Students Details***");
for(i=0;i<2;i++)
{
printf("\nRoll no=%d\nName=%s",s[i].rno,s[i].name);
printf("\n ---------------- \n");
getch();
}
```

Expected Output

```
***Students Details***
Rollno=25
Name=Ritchie
-------------------
Rollno=20
Name=Babbage
-------------------
```

ExampleProgram2

```
//Array of structures elements accepting from keyboard
struct student
{
Int rno;
char name[20];
char branch[10];
};

int main()
{
struct student s[2];
int i;
printf("\nEnter Rollno, Name & Branch of two students:\n");
for(i=0;i<2;i++)
{ scanf("%d%s%s",&s[i].rno,&s[i].name,&s[i].branch);
printf("\n");
}

printf("\n***Students Details***");
for(i=0;i<2;i++)
{
printf("\nRoll no=%d\nName=%s\nBranch=%s",s[i].rno,s[i].name,s[i].branch);
printf("\n..........................\n");
}
getch();

}
```

ExpectedOutput

```
Enter Rollno, Name & Branch of two students:
25
Ritchie
CSE

20
Babbage
CSE

***Students Details***
Rollno=25
Name=Ritchie
Branch=CSE
----------------------
Rollno=20
Name=Babbage
Branch=CSE
----------------------
```

- In an array of structures, all elements of the array are stored in adjacent memory locations.
- Since each element of this array is a structure, and since all structure elements are always stored in adjacent locations you can very well visualize the arrangement of array of structures in memory.
- In our example, **s[0]**’s **rno**, **name** and **branch** in memory would be immediately followed by **s[1]**’s **rno**, **name** and **branch**, and soon.

Arrays and Structures within Structures (Nested Structures)

A member of a structure can be either a simple variable, such as an **int** or **double**, or an aggregate type. InC, aggregate types are arrays and structures. You have already seen one type of aggregate element: the character arrays used in**s[0]**.

For example, consider this structure:

```
struct x{
inta[10][10]; /* 10x10arrayof ints */
float b;
}y;
```

To reference integer 3,7in**a** of structure **y**, write y.a[3][7]

One structure can be nested within another structure. Using this facility complex data types can be created.

ExampleProgram

```
//Structure within Structure
voidmain()
{
        struct student          //outer structure
        {
        char name[20];
        intrno;
                struct address  //inner structure
                {
                charcity[20];
                char state[20];
                }addr;
}stud;
printf("Enter Student name, Roll no,City,State\n");
scanf("%s%d%s%s",&stud.name,&stud.rno,&stud.aaddr.city,&stud.addr.state);
printf("\nName:%s\nRoll no:%d",stud.name,stud.rno);
printf("\nCity:%s\nState:%s",stud.addr.city,stud.addr.state)
;
getch();
}
```

ExpectedOutput

```
Enter Student name, Roll no, City, State
Sreenivas
25
Kurnool
AP
Name:
Sreenivas
Rollno:25
City:Kurnool
State:AP
```

In the above program, the method used to access the element of a structure that is part of another structure. For this the dot operator is used twice, as in the expression, **stud.addr.city** and **stud.addr.state**

Structure Pointers

The way we can have a pointer pointing to an **int**,or a pointer pointing to a **char**, similarly we can have a pointer pointing to a **struct**.

Such pointers are known as „structure pointers‟.

```
//Pointer to structure
int main()
{
        struct student
        {
        char name[20];
        int rno;
        float per;
        };
        struct students={"Sreenivas",25,71.00};
        struct student *ptr;
        ptr=&s;             //pointer "ptr"pointingto structure variable „s‟
        printf("\nName:%s\nRollno:%d\nPercentage:%.2f",ptr->name,ptr->rno,ptr->per);
getch();
}
```

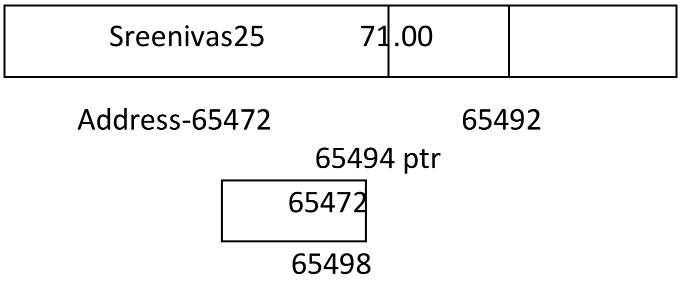
ExpectedOutput

```
Name: Sreenivas
Rollno:25
Percentage:71.00
```

- We can't use **ptr.name** or **ptr.rno** because **ptr** is not a structure variable but a pointer to a structure, and the dot operator requires a structure variable on its left.
- In such cases C provides an **operator->, called an arrow operator** to refer to the structure elements.
- **Remember that on the left hand side of the'.' Structure operator, there must alwaysbe a structure variable, where as on the left hand side of the'->' operator there must always be a pointer to a structure.**

Memory Representation

b1.name b1.rno b1.per



Passing Structures to Functions

Like an ordinary variable, a structure variable can also be passed to afunction. The following are the three different ways of passing structure arguments to functions.

- ***a) Passing individual structure elements to a function***
- ***b) Passing an entire structure to a function***
- ***c) Passing address of a structure to a function***

(a)Passing individual structure elements to a function

Example Program

```
//to pass an individual structure element
void display(char *n,char*a,intp);        //function prototype

int main()
{
        struct book
        {
        char name[20];
        char author[40];
        intprice;
        };
        struct book b={"Computer","Sreenivas Reddy",250};
        display(b.name, b.author, b.price); //passing individual structure elements to display()
        getch();
        return 0;
}
void display(char *n,char*a,intp)
{
```

```
printf("\nBook Name:%s\nAuthor:%s\nPrice:%d",n,a,p);
}
```

ExpectedOutput

```
Book Name:Computer
Author: Sreenivas
Reddy
Price:250
```

In the above program, we are passing the base addresses of the arrays **name** and **author**, and the value stored in price.
Thus, this is a mixed call—a **call by reference** as well as a **call by value**.

(b) Passing an entire structure to a function

It can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing.
A better way would be to pass the entire structure variable at a time.

ExampleProgram

```
//to pass an entirestructure
void display();     //function prototype

struct book          //structure named "book"
{
char name[20]; char author[20]; intprice;

};

int main()
{
        struct book b={"Computer","Sreenivas Reddy",250};
display(b);        //passing entirestructure variable „b‟
getch();
}
void display(struct bookb1)
{
printf("\nName=%s\nAuthor=%s\nPrice=%d",b1.name,b1.author,b1.price);
}
```

ExpectedOutput

```
Book Name:Computer
Author: Sreenivas
Reddy
Price:250
```

(c) Passing address of a structure to a function

//to pass the address ofstructure

Like variables, the address of structure variable can be passed to a function.

ExampleProgram

```
void display();         //function prototype
struct book              //structure definition outsidethe main()
{
char name[20];
char author[20];
intprice;
};

voidmain()
{
struct book b={"Computer","Sreenivas Reddy",250};
display(&b);        //passing address of structure variable „b‟todisplay()
getch();
}
void display(struct book*b1)
{
printf("\nName=%s\nAuthor=%s\nPrice=%d",b1->name,b1->author,b1->price);
}
```

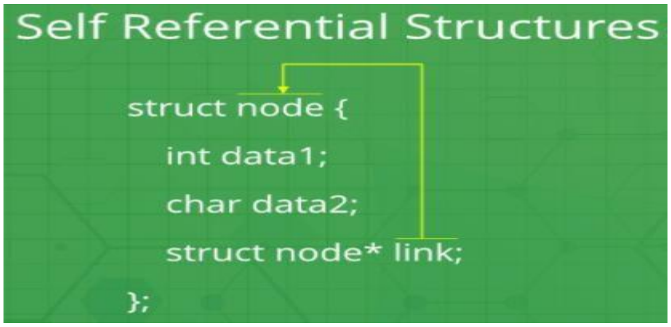
<u>ExpectedOutput</u>

```
Book Name:Computer Author: Sreenivas Reddy

Price:250
```

Self Referential Structures

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

In other words, structures pointing to the same type of structures are self-referential in nature.



Self Referential Structures

```
struct node {
    int data1;
    char data2;
    struct node* link;
};
```

```
struct node {
	int data1;
	char data2;
	struct node* link;
};
int main()
{	struct node ob;
	return 0;
}
```

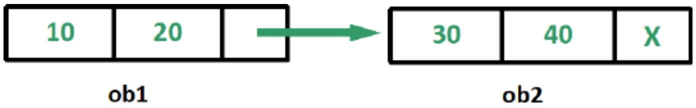
In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.

An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

Types of Self Referential Structures

1. Self Referential Structure with Single Link
2. Self Referential Structure with Multiple Links

1. Self Referential Structure with Single Link: These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.

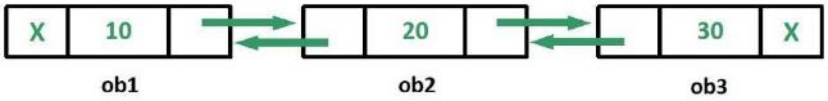


```
#include <stdio.h>
struct node {
	int data1;
char data2;
        struct node* link;
};
int main()
{       struct node ob1; // Node1
        // Initialization
        ob1.link = NULL;
        ob1.data1 = 10;
        ob1.data2 = 20;

struct node ob2; // Node2
        // Initialization
        ob2.link = NULL;
        ob2.data1 = 30;
        ob2.data2 = 40;

        // Linking ob1 and ob2
        ob1.link = &ob2;

        // Accessing data members of ob2 using ob1
        printf("%d",  ob1.link->data1);
        printf("\n%d", ob1.link->data2);
        return 0;
}
```

Output:

```
30
40
```

2. Self Referential Structure with Multiple Links: Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links. The connections made in the above example can be understood using the following figure.



```
#include <stdio.h>

struct node {
        int data;
        struct node* prev_link;
        struct node* next_link;
};

int main()

{
        struct node ob1; // Node1

        // Initialization
        ob1.prev_link = NULL;
        ob1.next_link = NULL;
        ob1.data = 10;

        struct node ob2; // Node2

        // Initialization
        ob2.prev_link = NULL;
        ob2.next_link = NULL;
        ob2.data = 20;

        struct node ob3; // Node3

        // Initialization
        ob3.prev_link = NULL;
        ob3.next_link = NULL;
        ob3.data = 30;

        // Forward links
        ob1.next_link = &ob2;
        ob2.next_link = &ob3;

        // Backward links
        ob2.prev_link = &ob1;
        ob3.prev_link = &ob2;

        // Accessing data of ob1, ob2 and ob3 by ob1
        printf("%d\t", ob1.data);
        printf("%d\t", ob1.next_link->data);
        printf("%d\n", ob1.next_link->next_link->data);

        // Accessing data of ob1, ob2 and ob3 by ob2
        printf("%d\t", ob2.prev_link->data);
        printf("%d\t", ob2.data);
        printf("%d\n", ob2.next_link->data);

        // Accessing data of ob1, ob2 and ob3 by ob3
        printf("%d\t", ob3.prev_link->prev_link->data);
        printf("%d\t", ob3.prev_link->data);
        printf("%d", ob3.data);
        return 0;
}
```

Output:

```
10    20    30
10    20    30
10    20    30
```

In the above example we can see that ‘ob1’, ‘ob2’ and ‘ob3’ are three objects of the self referential structure ‘node’. And they are connected using their links in such a way that any of them can easily access each other’s data. This is the beauty of the self referential structures. The connections can be manipulated according to the requirements of the programmer.

Applications:

Self referential structures are very useful in creation of other complex data structures like:

Linked Lists
Stacks
Queues
Trees
Graphs etc

C – Typedef

- Typedef is a keyword that is used to give a new symbolic name for the existing name in a C program. This is same like defining alias for the commands.
- Consider the below structure.

typedef<existing_name><alias_name>

Lets take an example and see how typedef actually works.

typedef unsigned long ulong;

The above statement define a term ulong for an unsigned long datatype. Now this ulong identifier can be used to define unsigned long type variables.

ulongi, j;

Structure definition using typedef

- Variable for the above structure can be **declared in two ways.**

1st way :

```
struct student record;       /* for normal variable */
struct student *record;     /* for pointer variable */
```

2nd way :

```
typedef struct student status;
```

- When we use “typedef” keyword before struct <tag_name> like above, after that we can simply use type definition “status” in the C program to declare structure variable.
- Now, structure variable declaration will be, “status record”.
- This is equal to “struct student record”. Type definition for “struct student” is status. i.e. status = “struct student”

Example:

```
#include<stdio.h>

#include<string.h>
typedef struct employee
{
char name[50];
int salary;
}emp;

void main( )
{
emp e1;
printf("\nEnter Employee record:\n");
printf("\nEmployee name:\t");
scanf("%s", e1.name);
printf("\nEnter Employee salary: \t");
scanf("%d", &e1.salary);
printf("\nstudent name is %s", e1.name);
printf("\nroll is %d", e1.salary);
}
```

C – Union

- C Union is also like structure, i.e. collection of different data types which are grouped together. Each element in a union is called member.
- Union and structure in C are same in concepts, except allocating memory for their members.

Structure allocates storage space for all its members separately.

- Whereas, Union allocates one common storage space for all its members
- We can access only one member of union at a time. We can’t access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Where as Structure allocates storage space for all its members separately.
- Many union variables can be created in a program and memory will be allocated for each union variable separately.

Below table will help you how to form a C union, declare a union, initializing and accessing the members of the union.

Using normal variable	Using pointer variable

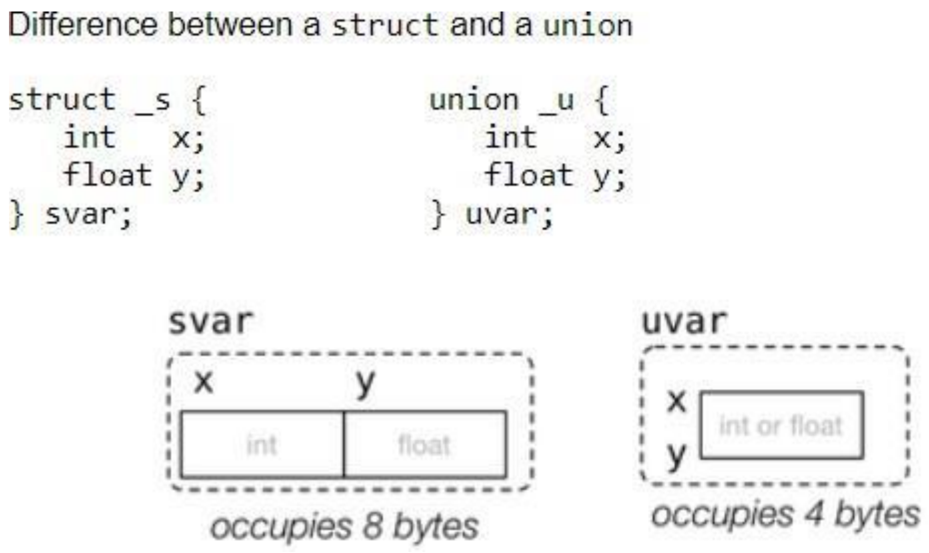
Syntax:
union tag_name
{
data type var_name1;
data type var_name2;
data type var_name3;
};
	Syntax:
	union tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};

Example:
union student
{
int mark;
char name[10];
float average;
};
	Example:
	union student
	{
	int mark;
	char name[10];
	float average;
	};

Declaring union using normal variable: union student report;	**Declaring union using pointer variable:** union student *report, rep;
Initializing union using normal variable: union student report = ,100, “Mani”, 99.5-;	**Initializing union using pointer variable:** union student rep = {100, “Mani”, 99.5-; report = &rep;

Accessing union members using normal variable:
report.mark;
report.name;
report.average;
	Accessing union members using pointer variable:
	report -> mark;
	report -> name;
	report -> average;

MEMORY REPRESENTATION:



Difference between a struct and a union

```
struct _s {              union _u {
   int   x;                 int   x;
   float y;                 float y;
} svar;                  } uvar;
```

EXAMPLE PROGRAM FOR C UNION:

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

union student
{
char name[20];
char subject[20];
float percentage;
};

int main()
{
union student record1;
union student record2;

   // assigning values to record1 union variable
strcpy(record1.name, "Raju");
strcpy(record1.subject, "Maths");
      record1.percentage = 86.50;

printf("Union  record1  values  example\n");
printf("   Name   :   %s   \n",   record1.name);
printf(" Subject  : %s \n", record1.subject);
printf(" Percentage : %f \n\n", record1.percentage);

   // assigning values to record2 union variable
printf("Union record2 values example\n");

strcpy(record2.name, "Mani");
printf(" Name       : %s \n", record2.name);

strcpy(record2.subject, "Physics");
printf(" Subject    : %s \n", record2.subject);

      record2.percentage = 99.50;
printf(" Percentage : %f \n", record2.percentage);
return 0;
}
```

OUTPUT:

```
Union record1 values example
Name :
Subject :
Percentage : 86.500000;
Union record2 values example
Name : Mani
Subject : Physics
Percentage : 99.500000
```

EXAMPLE PROGRAM – ANOTHER WAY OF DECLARING C UNION:

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

union student
{
char name[20];
char subject[20];
float percentage;
}record;

int main()
{

strcpy(record.name, "Raju");
strcpy(record.subject, "Maths");
record.percentage = 86.50;

printf(" Name       : %s \n", record.name);
printf(" Subject    : %s \n", record.subject);
printf(" Percentage : %f \n", record.percentage);
return 0;
}
```

OUTPUT:

```
Name :

Subject :
Percentage : 86.500000
```

NOTE:

We can access only one member of union at a time. We can’t access all member values at the same time in union.

But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Where as Structure allocates storage space for all its members separately.

C Structure	C Union
Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
Structure occupies higher memory space.	Union occupies lower memory space over structure.
We can access all members of structure at a time.	We can access only one member of union at a time.

Structure example:
struct student
{
int mark;
char name[6];
double average;
};
	Union example:
	union student
	{
	int mark;
	char name[6];
	double average;
	};

For above structure, memory allocation	For above union, only 8 bytes of memory

will be like below. int mark – 2B char name[6] – 6B double average – 8B Total memory allocation = 2+6+8 = 16 Bytes	will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

Enum in C

The enum in C is also known as the enumerated type. It is a user-defined data type that consists of integer values, and it provides meaningful names to these values. The use of enum in C makes the program easy to understand and maintain. The enum is defined by using the enum keyword.

The following is the way to define the enum in C:

enum flag{integer_const1, integer_const2, ... integter_constN};

In the above declaration, we define the enum named as flag containing 'N' integer constants. The default value of integer_const1 is 0, integer_const2 is 1, and so on. We can also change the default value of the integer constants at the time of the declaration.

For example:

1. **enum** fruits{mango, apple, strawberry, papaya};

The default value of mango is 0, apple is 1, strawberry is 2, and papaya is 3. If we want to change these default values, then we can do as given below:

1. **enum** fruits{
2. mango=2,
3. apple=1,
4. strawberry=5,
5. papaya=7,
6. };

Enumerated type declaration

As we know that in C language, we need to declare the variable of a pre-defined type such as int, float, char, etc. Similarly, we can declare the variable of a user-defined data type, such as enum. Let's see how we can declare the variable of an enum type.

Suppose we create the enum of type status as shown below:

enum status{**false**,**true**};

Now, we create the variable of status type:

enum status s; // creating a variable of the status type.

In the above statement, we have declared the 's' variable of type status.

To create a variable, the above two statements can be written as:

```
          enum status{false,true} s;
```

In this case, the default value of false will be equal to 0, and the value of true will be equal to 1.

Let's create a simple program of enum.

```
#include <stdio.h>
enum weekdays{Sunday=1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
int main()
{
enum weekdays w; // variable declaration of weekdays type
 w=Monday; // assigning value of Monday to w.
printf("The value of w is %d",w);
return 0;
}
```

Output:

```
The value of w is 2
```

example

```
#include <stdio.h>
enum months{jan=1, feb, march, april, may, june, july, august, september, october, november,
december};
int main()

{
// printing the values of months
for(int i=jan;i<=december;i++)
 {
printf("%d, ",i);
 }
return 0;
}
```

Output:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

use an enum in a switch case statement

```
#include <stdio.h>
enum days{sunday=1, monday, tuesday, wednesday, thursday, friday, saturday};
int main()
{
enum days d;
   d=monday;
switch(d)
   {
casesunday:
```

```
printf("Today is sunday");
break;
casemonday:
printf("Today is monday");
break;
casetuesday:
printf("Today is tuesday");
break;
casewednesday:
printf("Today is wednesday");
break;
casethursday:
printf("Today is thursday");
break;
casefriday:
printf("Today is friday");
break;
casesaturday:
printf("Today is saturday");
break;
   }

return 0;
}
```

Output:

```
Today is Monday
```

Some important points related to enum

- If we do not provide any value to the enum names, then the compiler will automatically assign the default values to the enum names starting from 0.
- We can also provide the values to the enum name in any order, and the unassigned names will get the default value as the previous one plus one.
- The values assigned to the enum names must be integral constant, i.e., it should not be of other types such string, float, etc.

```
#include <stdio.h>

int main(void) {
enum fruits{mango = 1, strawberry=0, apple=1};
printf("The value of mango is %d", mango);
printf("\nThe value of apple is %d", apple);
return 0;
}
```

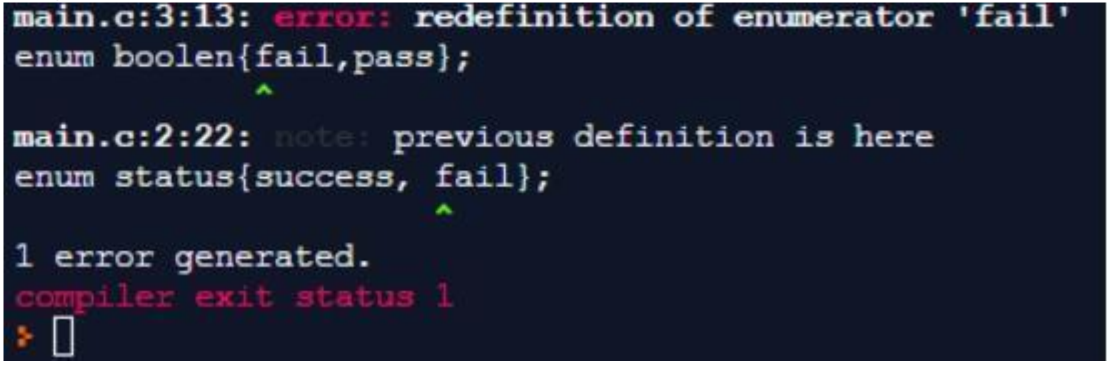
Output:

The value of mango is 1
The value of apple is 1

- All the enum names must be unique in their scope, i.e., if we define two enum having same scope, then these two enums should have different enum names otherwise compiler will throw an error.

```
#include <stdio.h>
enum status{success, fail};
enumboolen{fail,pass};
int main(void) {

printf("The value of success is %d", success);
return 0;
}
```

Output:



```
main.c:3:13: error: redefinition of enumerator 'fail'
enum boolen{fail,pass};
            ^
main.c:2:22: note: previous definition is here
enum status{success, fail};
                     ^
1 error generated.
compiler exit status 1
> 
```

- In enumeration, we can define an enumerated data type without the name also.

```
#include <stdio.h>
enum {success, fail} status;
int main(void) {
status=success;
printf("The value of status is %d", status);
return 0;
   }
```

Output

The value of status is 0

Enum vs. Macro in C

Macro can also be used to define the name constants, but in case of an enum, all the name constants can be grouped together in a single statement.
For example,
# define pass 0;
# define success 1;

The above two statements can be written in a single statement by using the enum type.
enum status{pass, success};

- The enum type follows the scope rules while macro does not follow the scope rules.
- In Enum, if we do not assign the values to the enum names, then the compiler will automatically assign the default value to the enum names. But, in the case of macro, the values need to be explicitly assigned.
- The type of enum in C is an integer, but the type of macro can be of any type.

Bit Fields in C

When we use structures in the c programming language, the memory required by structure variable is the sum of memory required by all individual members of that structure. To save memory or to restrict memory of members of structure we use bitfield concept. Using bitfield we can specify the memory to be allocated for individual members of a structure. To understand the bitfields,

Declaring Bit Fields:

Variables that are defined using a predefined width or size are called bit fields. This bit field can leave more than a single bit. The format and syntax of bit-field declaration inside a structure is something like this:

```
struct{

  data_typemember_name: width;

};
```

Date structure in C

```
struct Date
{
unsigned int day;

unsigned int month;
unsigned int year;
} ;
```

Here, the variable of Date structure allocates 6 bytes of memory.

In the above example structure the members day and month both does not requires 2 bytes of memory for each. Becuase member day stores values from 1 to 31 only which requires 5 bits of memory, and the member month stores values from 1 to 12 only which required 4 bits of memory. So, to save the memory we use the bitfields.
Consider the following structure with bitfields...

Date structure in C

```
struct Date
{
unsigned int day : 5;
unsigned int month : 4;
unsigned int year;
} ;
```

Here, the variable of Date structure allocates 4 bytes of memory.
Example:

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

struct {
unsigned int age : 3;
} Age;

int main( ) {

Age.age = 4;
printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
printf( "Age.age : %d\n", Age.age );

Age.age = 7;
printf( "Age.age : %d\n", Age.age );

Age.age = 8;
printf( "Age.age : %d\n", Age.age );

return 0;
}
```

Output:

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Important points about bit fields in C

1) A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the following program.

```
#include <stdio.h>

// A structure without forced alignment
struct test1 {
        unsigned int x : 5;
        unsigned int y : 8;
};

// A structure with forced alignment
struct test2 {
        unsigned int x : 5;
        unsigned int : 0;
        unsigned int y : 8;
};

int main()
{
        printf("Size of test1 is %lu bytes\n",
                sizeof(struct test1));
        printf("Size of test2 is %lu bytes\n",
                sizeof(struct test2));
        return 0;
}
```

Output:

```
Size of test1 is 4 bytes
Size of test2 is 8 bytes
```

2) We cannot have pointers to bit field members as they may not start at a byte boundary.

```
#include <stdio.h>
struct test {
        unsigned int x : 5;
        unsigned int y : 5;
        unsigned int z;
};
int main()
```

```
{
        struct test t;

        // Uncommenting the following line will make
        // the program compile and run
        printf("Address of t.x is %p", &t.x);

        // The below line works fine as z is not a
        // bit field member
        printf("Address of t.z is %p", &t.z);
        return 0;
}
```

Output:

```
prog.c: In function 'main':
prog.c:14:1: error: cannot take address of bit-field 'x'
printf("Address of t.x is %p", &t.x);
 ^
```

3) It is implementation defined to assign an out-of-range value to a bit field member.

```
#include <stdio.h>
struct test {
        unsigned int x : 2;
        unsigned int y : 2;
        unsigned int z : 2;
};
int main()
{
        struct test t;
        t.x = 5;
        printf("%d", t.x);
        return 0;
}
```

Output:

```
Implementation-Dependent
```

4) In C++, we can have static members in a structure/class, but bit fields cannot be static.

```
// The below C++ program compiles and runs fine
struct test1 {
        static unsigned int x;
};
```

```
int main() {}
```

Output:

```
// But below C++ program fails in the compilation
// as bit fields cannot be static
struct test1 {
        static unsigned int x : 5;
};
int main() {}
```

Output:

```
 prog.cpp:5:29: error: static member 'x' cannot be a bit-field
static unsigned int x : 5;
                       ^
```

5) Array of bit fields is not allowed. For example, the below program fails in the compilation.

```
struct test {
        unsigned int x[10] : 5;
};

int main()
{
}
```

Output:

```
prog.c:3:1: error: bit-field 'x' has invalid type

unsigned int x[10]: 5;

 ^
```

6) Use bit fields in C to figure out a way whether a machine is little-endian or big-endian.

Applications –

- If storage is limited, we can go for bit-field.
- When devices transmit status or information encoded into multiple bits for this type of situation bit-fiels is most effiecient.
- Encryption routines need to access the bits within a byte in that situation bit-field is quite usefull.

C - Variable Length Arguments

Sometimes, you may come across a situation, when you want to have a function, which can take variable number of arguments, i.e., parameters, instead of predefined number of parameters. The C programming language provides a solution for this situation and you are allowed to define a function which can accept variable number of parameters based on your requirement. The following example shows the definition of such a function.

```
intfunc(int, ... ) {
  .
  .
  .
}

int main() {
func(1, 2, 3);
func(1, 2, 3, 4);
}
```

It should be noted that the function **func()** has its last argument as ellipses, i.e. three dotes (**...**) and the one just before the ellipses is always an **int** which will represent the total number variable arguments passed. To use such functionality, you need to make use of **stdarg.h** header file which provides the functions and macros to implement the functionality of variable arguments and follow the given steps −

- Define a function with its last parameter as ellipses and the one just before the ellipses is always an **int** which will represent the number of arguments.
- Create a **va_list** type variable in the function definition. This type is defined in stdarg.h header file.
- Use **int** parameter and **va_start** macro to initialize the **va_list** variable to an argument list. The macro va_start is defined in stdarg.h header file.
- Use **va_arg** macro and **va_list** variable to access each item in argument list.
- Use a macro **va_end** to clean up the memory assigned to **va_list** variable.

Now let us follow the above steps and write down a simple function which can take the variable number of parameters and return their average −

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

double average(int num,...) {

  va_list valist;
double sum = 0.0;
inti;

  /* initialize valist for num number of arguments */
va_start(valist, num);

   /* access all the arguments assigned to valist */
for (i = 0; i< num; i++) {
sum += va_arg(valist, int);
   }

   /* clean memory reserved for valist */
va_end(valist);

return sum/num;
}

int main() {
printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

Output:

```
Average of 2, 3, 4, 5 = 3.500000
Average of 5, 10, 15 = 10.000000
```

Q. find minimum of given set of integers.

```
#include <stdarg.h>
#include <stdio.h>
int min(int arg_count, ...)
{
inti;
int min, a; 159

// va_list is a type to hold information about variable arguments
va_listap;

// va_start must be called before accessing variable argument list
va_start(ap, arg_count);

// Now arguments can be accessed one by one
// using va_arg macro. Initialize min as first argument in list
min = va_arg(ap, int);

// traverse rest of the arguments to find out minimum
for (i = 2; i<= arg_count; i++)
if ((a = va_arg(ap, int)) < min)
        min = a;

// va_end should be executed before the function returns whenever va_start has been
previously
// used in that function
va_end(ap);

return min;
}
// Driver code
int main()
{
int count = 5;
printf("Minimum value is %d", min(count, 12, 67, 6, 7, 100));
return 0;
}
```

Output:

```
Minimum value is 6
```

Linked List

Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.

A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.

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

DS Linked List

Uses of Linked List

The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.

list size is limited to the memory size and doesn't need to be declared in advance.

Empty node can not be present in the linked list.

We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

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

Array contains following limitations:

The size of array must be known in advance before using it in the program.

Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.

All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

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

It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.

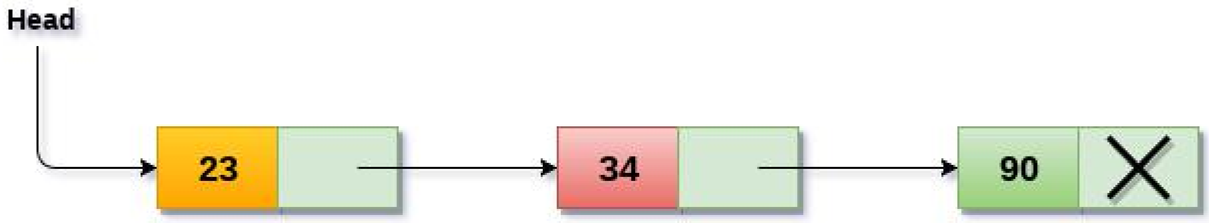
Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Singly linked list or One way chain

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

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

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



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

Complexity

Data Structure	Time Complexity								Space Compleity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Operations on Singly Linked List

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

Node Creation

```
struct node
{
   int data;
   struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

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

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

Deletion and Traversing

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

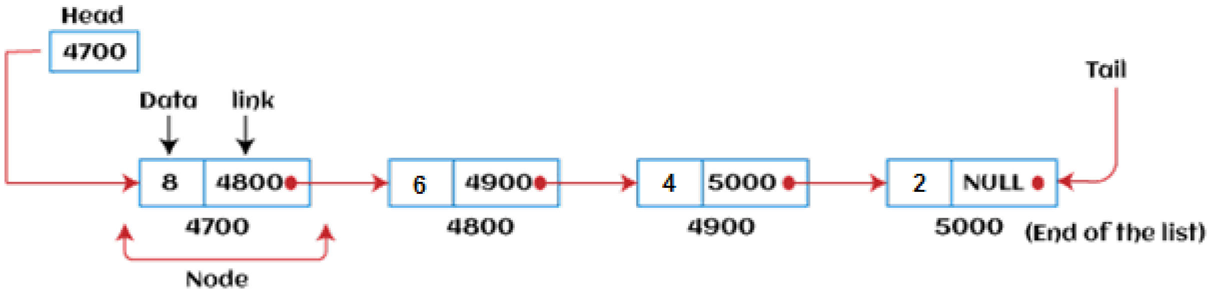
SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can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

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

linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory. A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.

Representation of a Linked list

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below -



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

Now, the question arises why we should use linked list over array?

Why use linked list over array?

Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -

- The size of the array must be known in advance before using it in the program.
- Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.

- All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because -

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

How to declare a linked list?

It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different types, i.e., one is the simple variable, while another is the pointer variable. We can declare the linked list by using the user-defined data type **structure.**

The declaration of linked list is given as follows -

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

In the above declaration, we have defined a structure named as **node** that contains two variables, one is **data** that is of integer type, and another one is **next** that is a pointer which contains the address of next node.

Now, let's move towards the types of linked list.

Types of Linked list

Linked list is classified into the following types -

- **Singly-linked list -** Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.
- **Doubly linked list -** Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).

- **Circular singly linked list -** In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
- **Circular doubly linked list -** Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

Now, let's see the benefits and limitations of using the Linked list.

Advantages of Linked list

The advantages of using the Linked list are given as follows -

- **Dynamic data structure -** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
- **Insertion and deletion -** Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.
- **Memory efficient -** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- **Implementation -** We can implement both stacks and queues using linked list.

Disadvantages of Linked list

The limitations of using the Linked list are given as follows -

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

Applications of Linked list

The applications of the Linked list are given as follows -

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Operations performed on Linked list

The basic operations that are supported by a list are mentioned as follows -

- **Insertion -** This operation is performed to add an element into the list.
- **Deletion -** It is performed to delete an operation from the list.
- **Display -** It is performed to display the elements of the list.
- **Search -** It is performed to search an element from the list using the given key.

Complexity of Linked list

Now, let's see the time and space complexity of the linked list for the operations search, insert, and delete.

1. Time Complexity

Operations	Average case time complexity	Worst-case time complexity
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)
Search	O(n)	O(n)

Types of Linked List

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

Types of Linked list

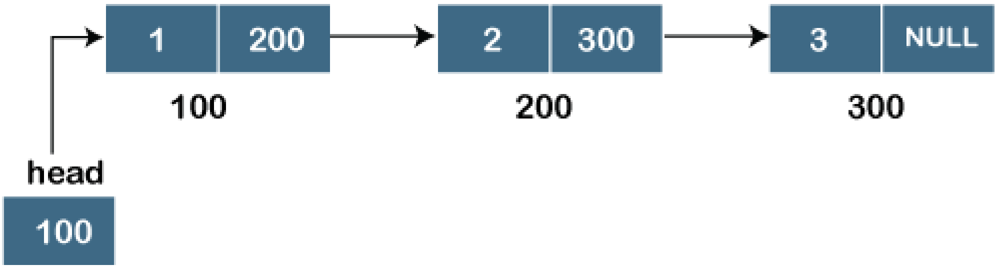
The following are the types of linked list:

- Singly Linked list
- Doubly Linked list
- Circular Linked list
- Doubly Circular Linked list

Singly Linked list

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

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



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

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

Representation of the node in a singly linked list

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

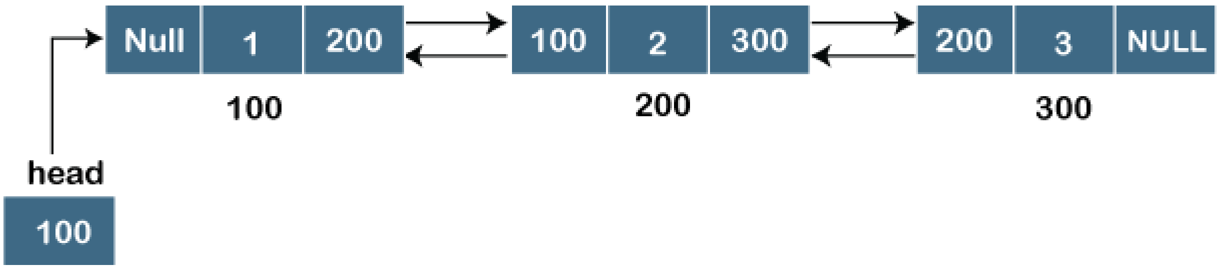
In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

To know more about a singly linked list, click on the link given below:

Doubly linked list

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

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



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

Representation of the node in a doubly linked list

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

```
 struct node *prev;
}
```

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

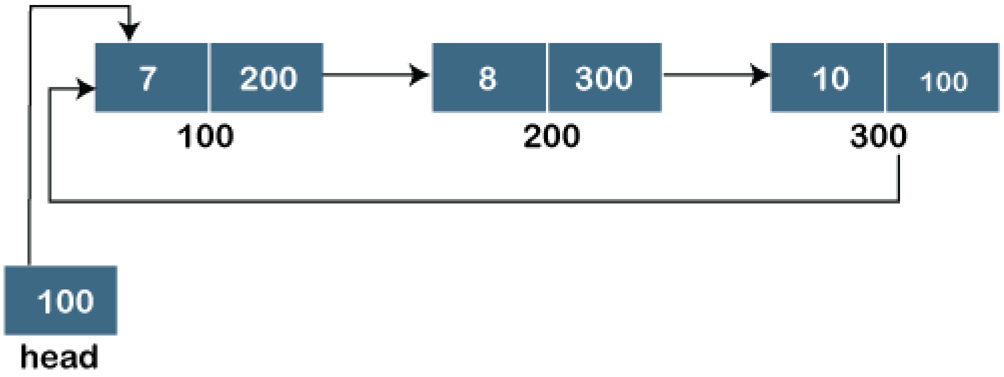
To know more about doubly linked list, click on the link given below:

Circular linked list

A circular linked list is a variation of a singly linked list. The only difference between the ***singly linked list*** and a ***circular linked*** list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:

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

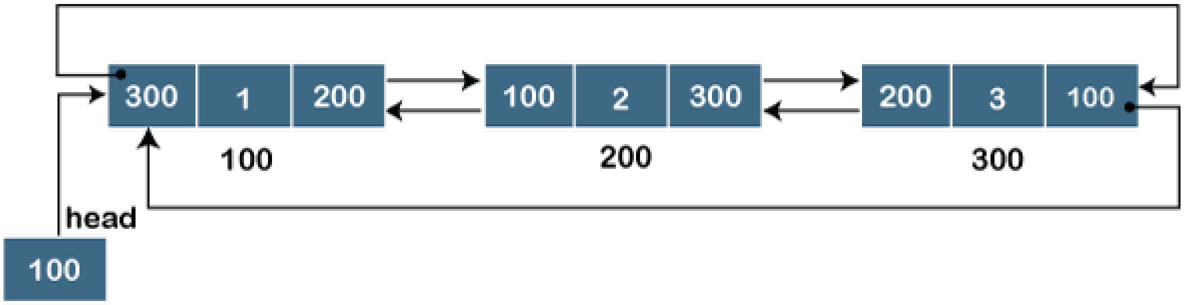
A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:



To know more about the circular linked list, click on the link given below:

Doubly Circular linked list

The doubly circular linked list has the features of both the ***circular linked list*** and ***doubly linked list***.

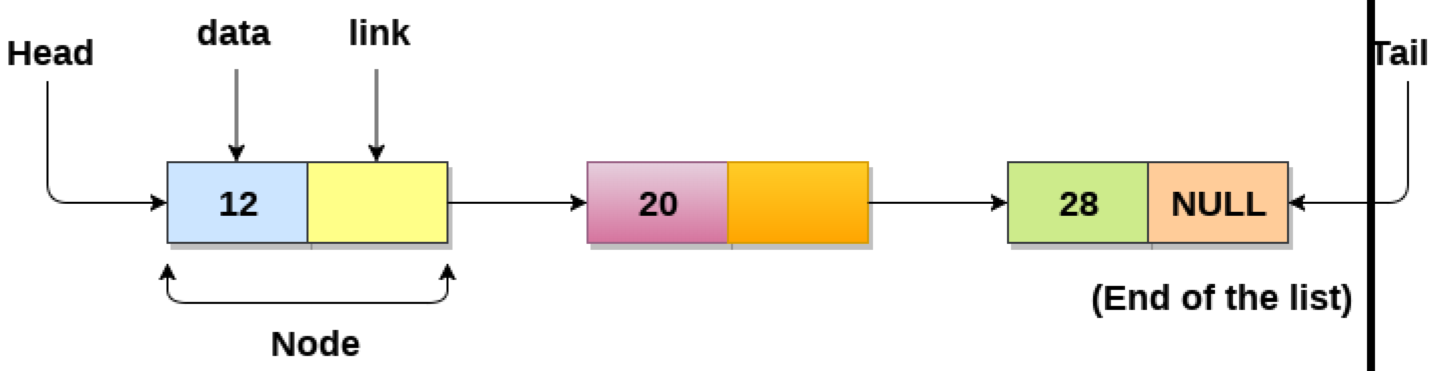


The above figure shows the representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle. It is a doubly linked list also because each node holds the address of the previous node also. The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node. As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.

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

Linked List

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



Uses of Linked List

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

Why use linked list over array?

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

Array contains following limitations:

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

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

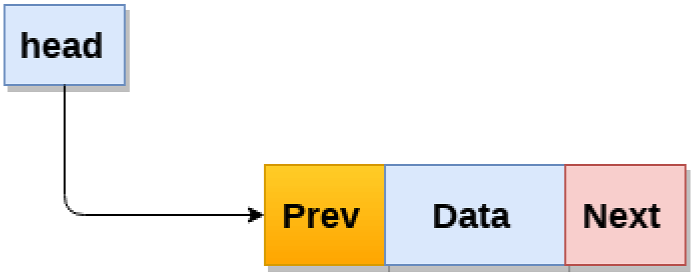
1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and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

Singly linked list or One way chain

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

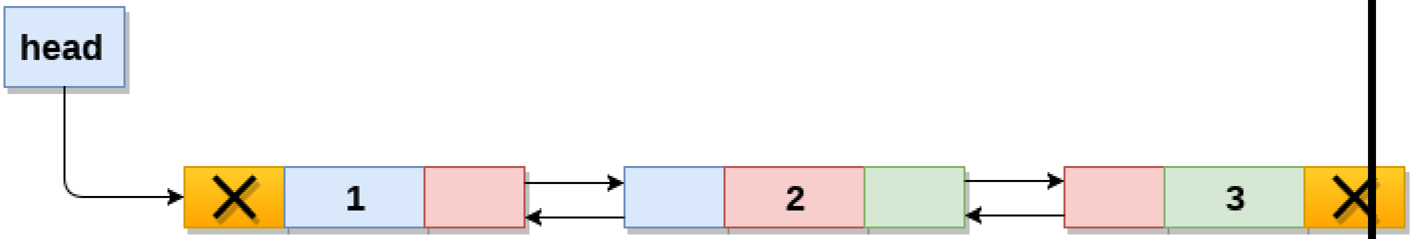
Doubly linked list

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



Node

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



Doubly Linked List

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

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

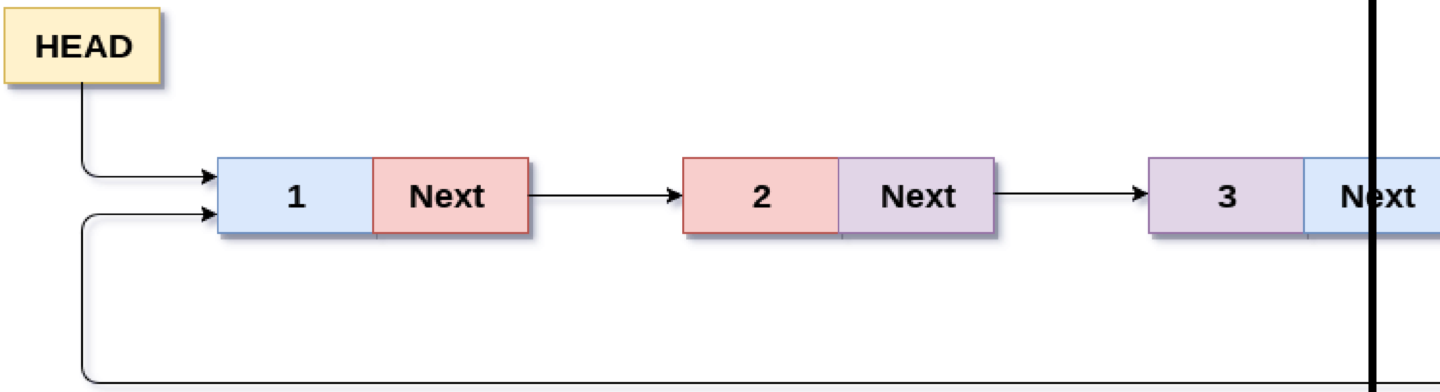
The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



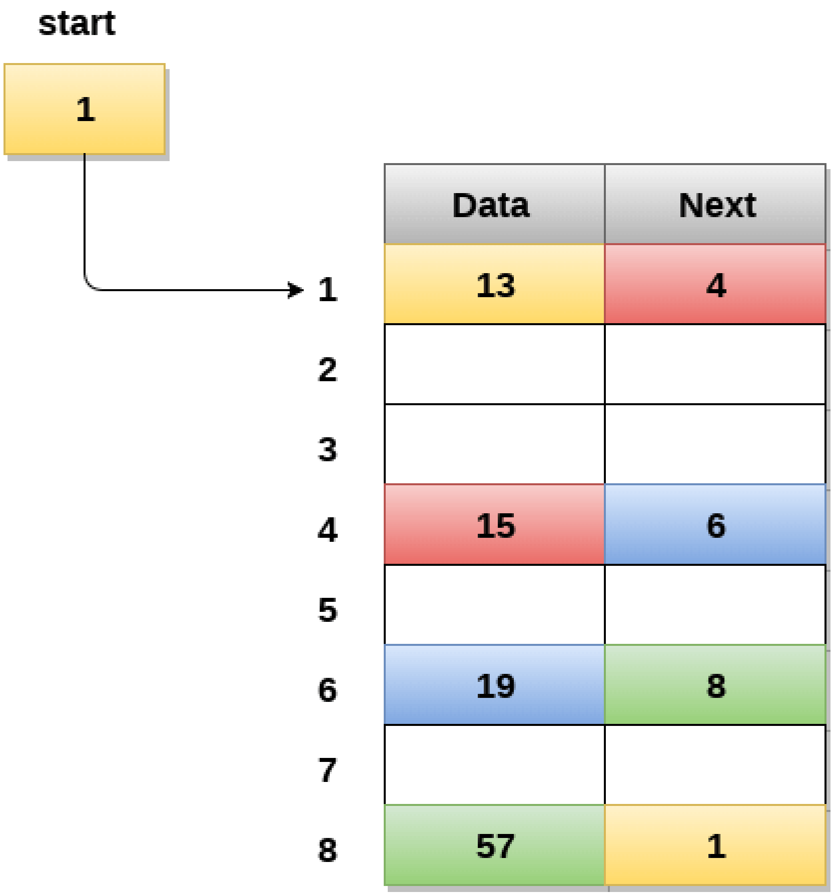
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



	Data	Next
1	13	4
2		
3		
4	15	6
5		
6	19	8
7		
8	57	1

Memory Representation of a circular linked list

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

Operations on Circular Singly linked list:

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

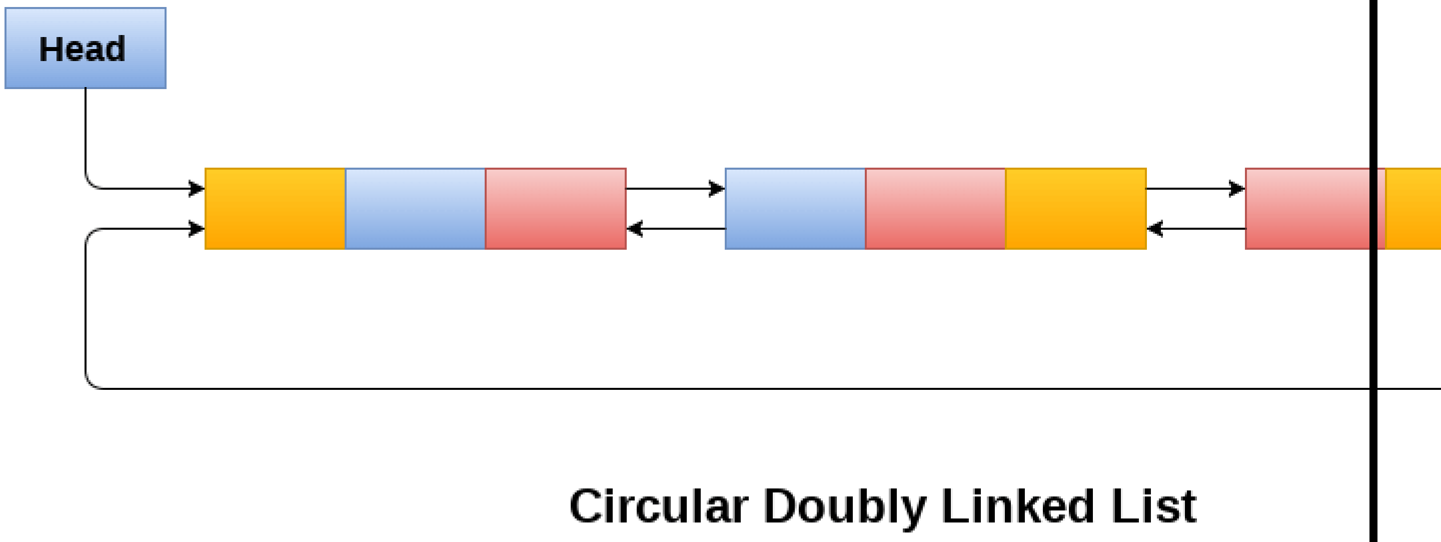
Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Circular Doubly Linked List

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.

A circular doubly linked list is shown in the following figure.



Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

Memory Management of Circular Doubly linked list

The following figure shows the way in which the memory is allocated for a circular doubly linked list. The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1. Since, each node of the list is supposed to have three parts therefore, the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4. The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image i.e. 1. In circular doubly linked list, the last node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.

Tree Data Structure

We read the linear data structures like an array, linked list, stack and queue in which all the elements are arranged in a sequential manner. The different data structures are used for different kinds of data.

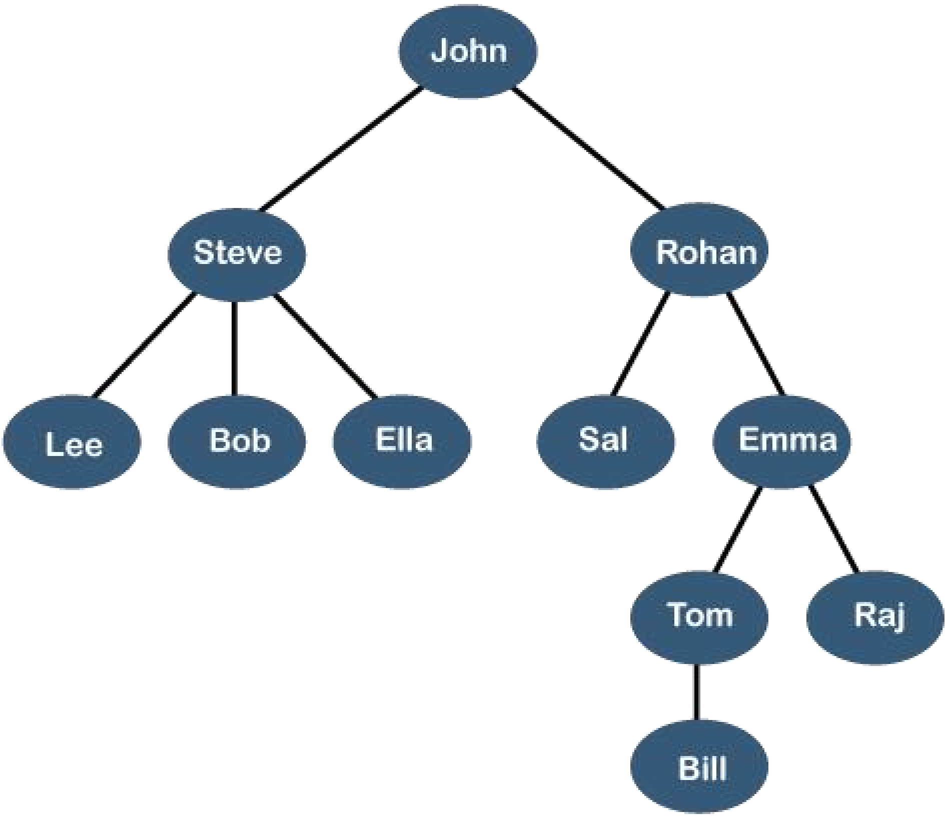
Some factors are considered for choosing the data structure:

- **What type of data needs to be stored**?: It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:** If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to

perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the ***binary search***. The binary search works very fast for the simple list as it divides the search space into half.

- **Memory usage:** Sometimes, we want a data structure that utilizes less memory.

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



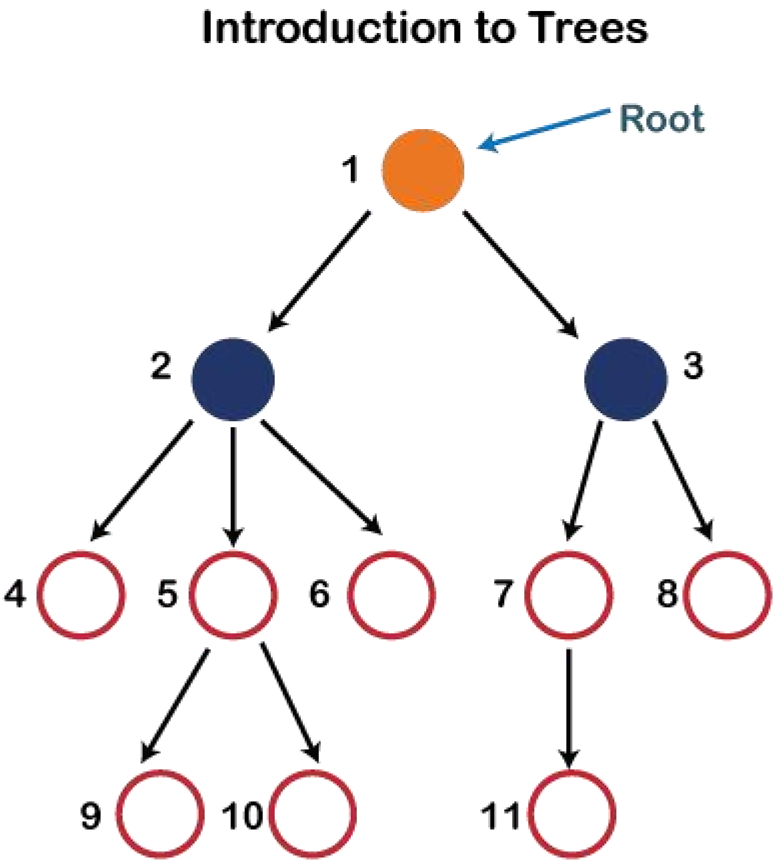
The above tree shows the **organization hierarchy** of some company. In the above structure, ***john*** is the **CEO** of the company, and John has two direct reports named as ***Steve*** and ***Rohan***. Steve has three direct reports named ***Lee, Bob, Ella*** where ***Steve*** is a manager. Bob has two direct reports named ***Sal*** and ***Emma***. **Emma** has two direct reports named ***Tom*** and ***Raj***. Tom has one direct report named ***Bill***. This particular logical structure is known as a ***Tree***. Its structure is similar to the real tree, so it is named a ***Tree***. In this structure, the ***root*** is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data in a hierarchical way.

Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.

Some basic terms used in Tree data structure.

Let's consider the tree structure, which is shown below:

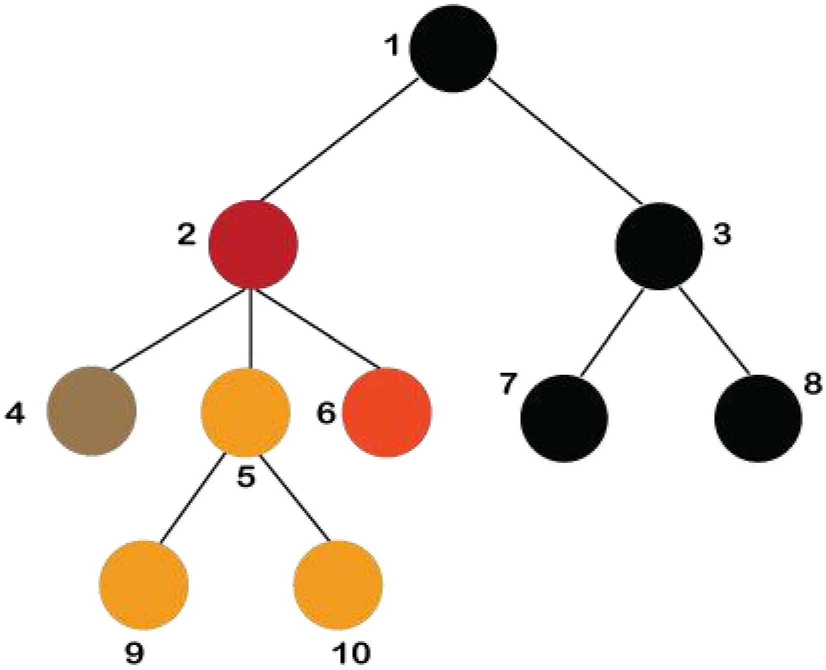


In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.

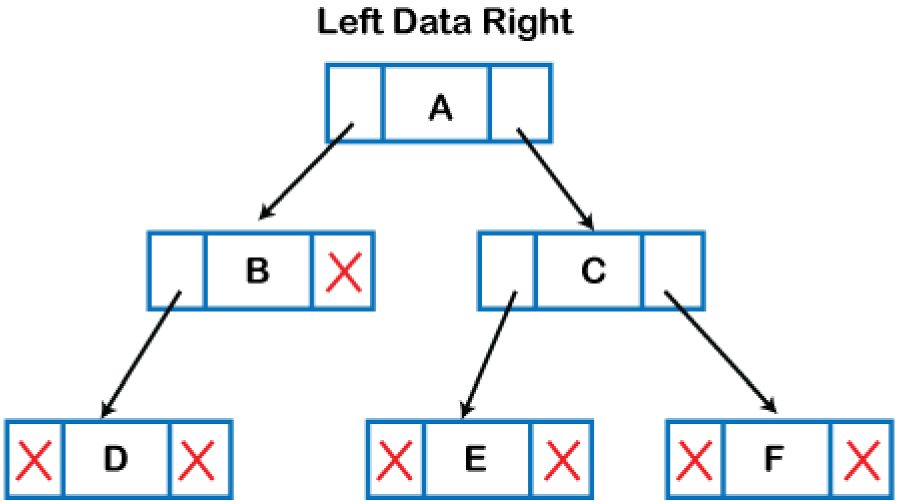


- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

```
struct node
{
 int data;
struct node *left;
struct node *right;
}
```

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Applications of trees

The following are the applications of trees:

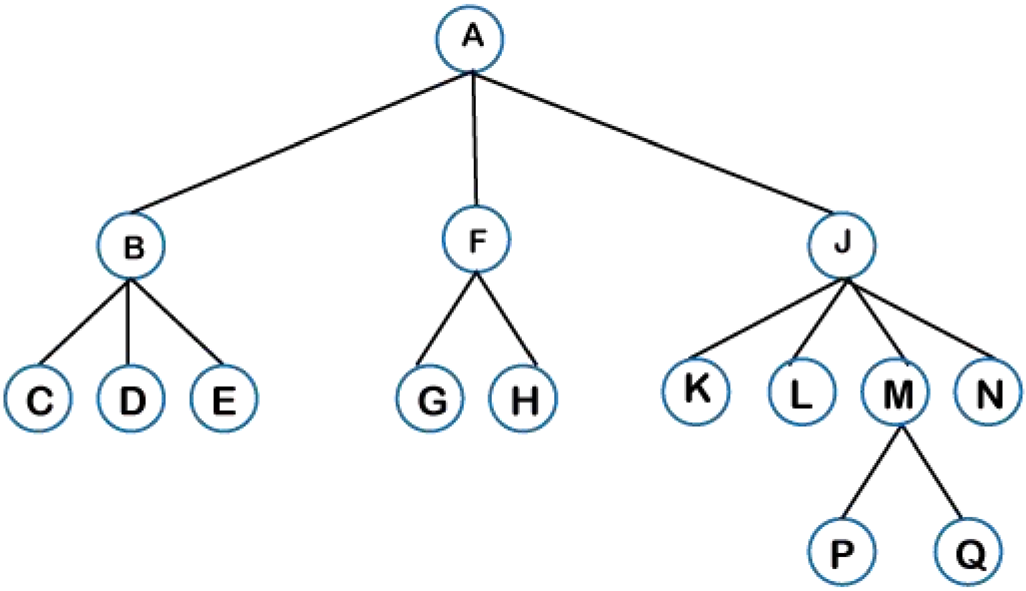
- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.

- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

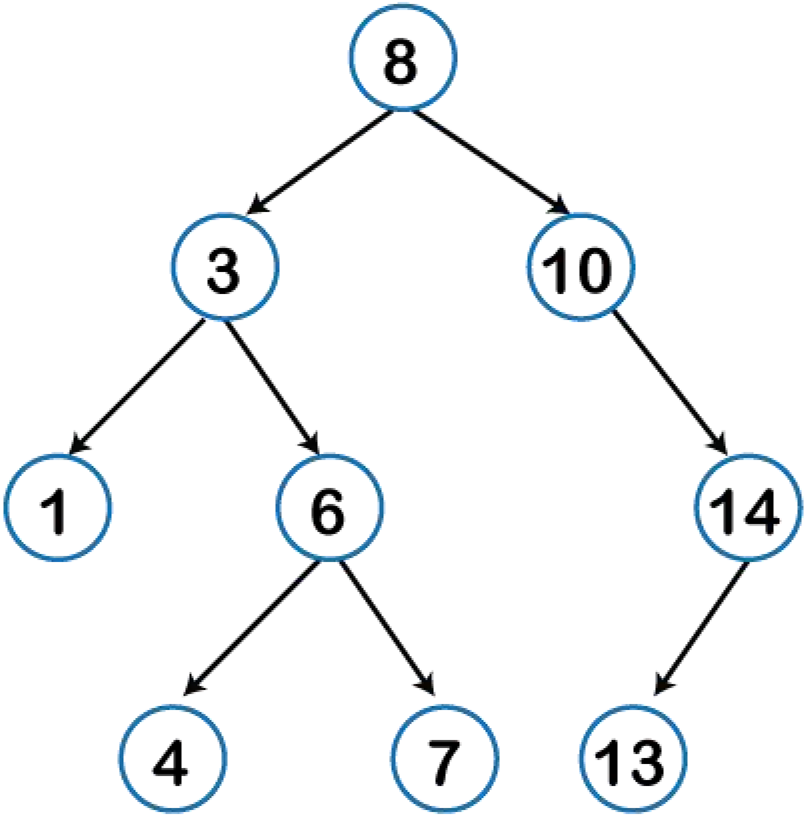
Types of Tree data structure

The following are the types of a tree data structure:

- **General tree:** The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.



  There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered. Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.
- **Binary tree:** Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



To know more about the binary tree, click on the link given below: https://www.javatpoint.com/binary-tree

- **Binary Search tree:** Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer). Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

A node can be created with the help of a user-defined data type known as ***struct,*** as shown below:

```
struct node
{
  int data;
  struct node *left;
struct node *right;
}
```

The above is the node structure with three fields: data field, the second field is the left pointer of the node type, and the third field is the right pointer of the node type.

- **AVL tree**

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the ***binary tree*** as well as of the ***binary search tree***. It is a self-balancing binary search tree that was invented by ***Adelson Velsky Lindas***. Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the ***balancing factor***.

We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the ***difference between the height of the left subtree and the height of the right subtree***. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.

To know more about the AVL tree, click on the link given below:

- **Red-Black Tree**

The red-Black tree is the binary search tree. The prerequisite of the Red-Black tree is that we should know about the binary search tree. In a binary search tree, the value of the left-subtree should be less than the value of that node, and the value of the right-subtree should be greater than the value of that node. As we know that the time complexity of binary search in the average case is $\log_2 n$, the best case is O(1), and the worst case is O(n).

When any operation is performed on the tree, we want our tree to be balanced so that all the operations like searching, insertion, deletion, etc., take less time, and all these operations will have the time complexity of ***$\log_2 n$.***

The red-black tree is a self-balancing binary search tree. AVL tree is also a height balancing binary search tree then **why do we require a Red-Black tree**. In the AVL tree, we do not know how many rotations would be required to balance the tree, but in the Red-black tree, a maximum of 2 rotations are required to balance the tree. It contains one extra bit that represents either the red or black color of a node to ensure the balancing of the tree.

- **Splay tree**

The splay tree data structure is also binary search tree in which recently accessed element is placed at the root position of tree by performing some rotation operations. Here, ***splaying*** means the recently accessed node. It is a ***self-balancing*** binary search tree having no explicit balance condition like **AVL** tree.

It might be a possibility that height of the splay tree is not balanced, i.e., height of both left and right subtrees may differ, but the operations in splay tree takes order of **logN** time where **n** is the number of nodes.

Splay tree is a balanced tree but it cannot be considered as a height balanced tree because after each operation, rotation is performed which leads to a balanced tree.

- **Treap**

Treap data structure came from the Tree and Heap data structure. So, it comprises the properties of both Tree and Heap data structures. In Binary search tree, each node on the left subtree must be equal or less than the value of the root node and each node on the right subtree must be equal or greater than the value of the root node. In heap data structure, both right and left subtrees contain larger keys than the root; therefore, we can say that the root node contains the lowest value.

In treap data structure, each node has both ***key*** and ***priority*** where key is derived from the Binary search tree and priority is derived from the heap data structure.

The **Treap** data structure follows two properties which are given below:

- Right child of a node>=current node and left child of a node <=current node (binary tree)
- Children of any subtree must be greater than the node (heap)

- **B-tree**

B-tree is a balanced **m-way** tree where **m** defines the order of the tree. Till now, we read that the node contains only one key but b-tree can have more than one key, and more than 2 children. It always maintains the sorted data. In binary tree, it is possible that leaf nodes can be at different levels, but in b-tree, all the leaf nodes must be at the same level.

If order is m then node has the following properties:

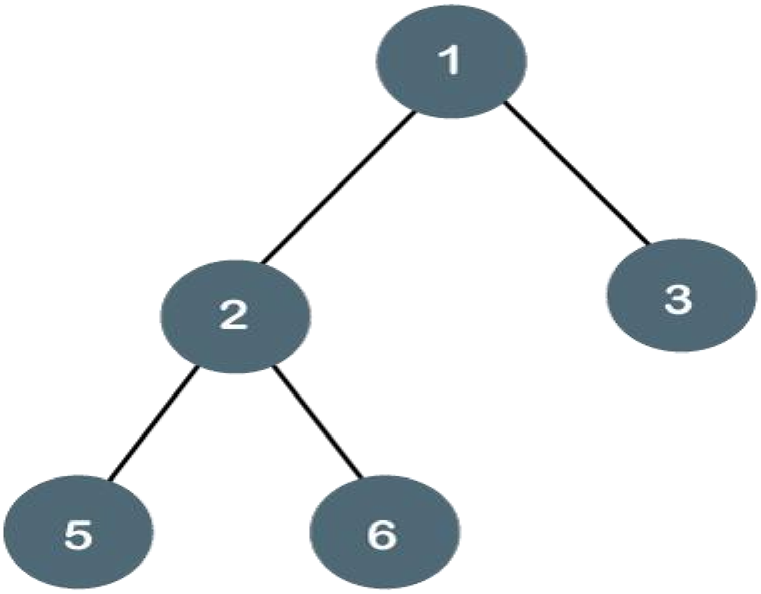
- Each node in a b-tree can have maximum **m** children
- For minimum children, a leaf node has 0 children, root node has minimum 2 children and internal node has minimum ceiling of m/2 children. For example, the value of m is 5 which means that a node can have 5 children and internal nodes can contain maximum 3 children.
- Each node has maximum (m-1) keys.

The root node must contain minimum 1 key and all other nodes must contain atleast **ceiling of m/2 minus 1** keys.

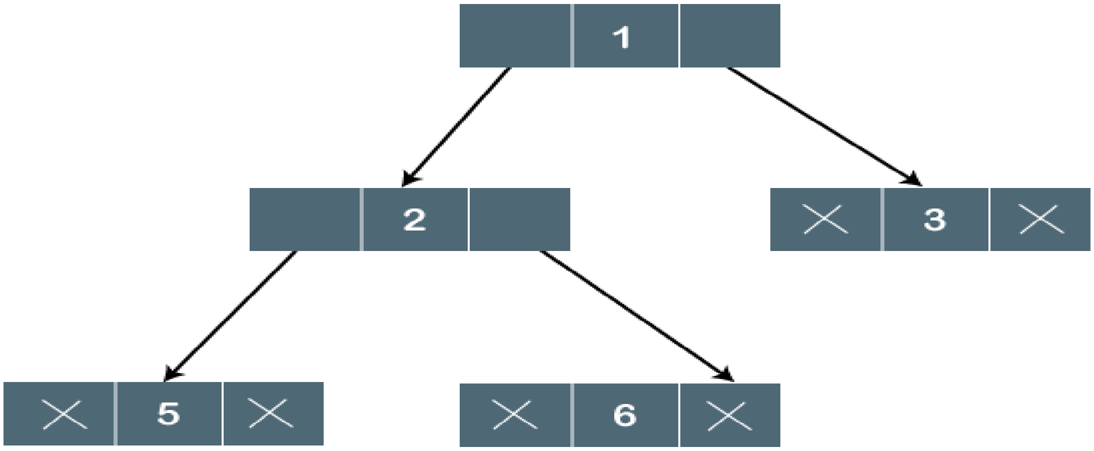
Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of

nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is ($2^0 + 2^1 + 2^2 + \ldots 2^h$) = 2^{h+1} -1.

- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$\log_2(n+1) = \log2(2^{h+1})$

$\log_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

The maximum height can be computed as:

As we know that,

n = h+1

h= n-1

Types of Binary Tree

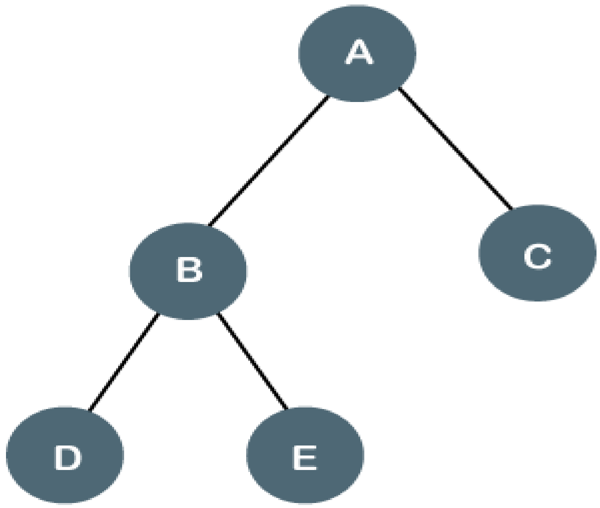
There are four types of Binary tree:

- **Full/ proper/ strict Binary tree**
- **Complete Binary tree**
- **Perfect Binary tree**
- **Degenerate Binary tree**
- **Balanced Binary tree**

1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.



In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2^{h+1} -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of the full binary tree can be computed as:

n= 2*h - 1

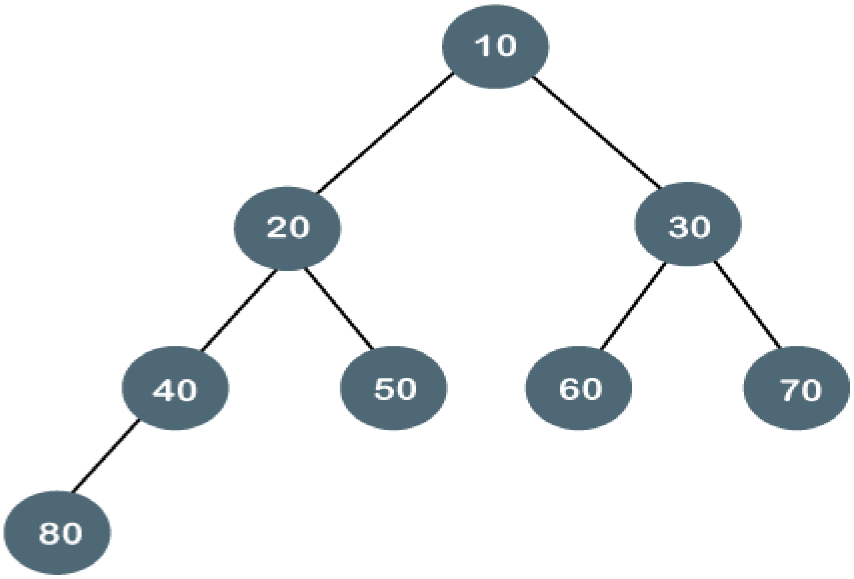
n+1 = 2*h

h = n+1/2

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.



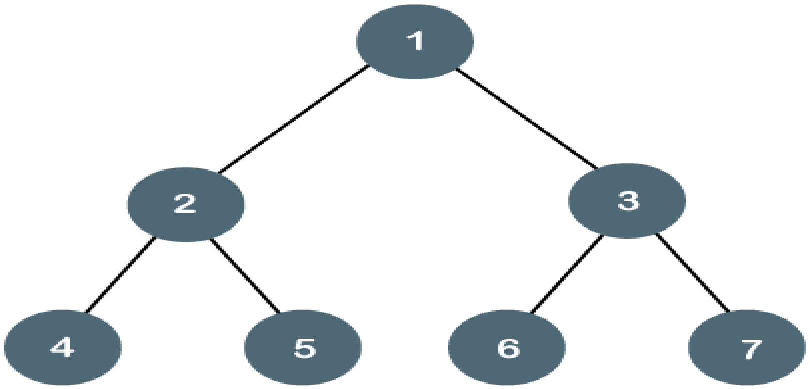
The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is 2^{h+1} - 1.
- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of a complete binary tree is

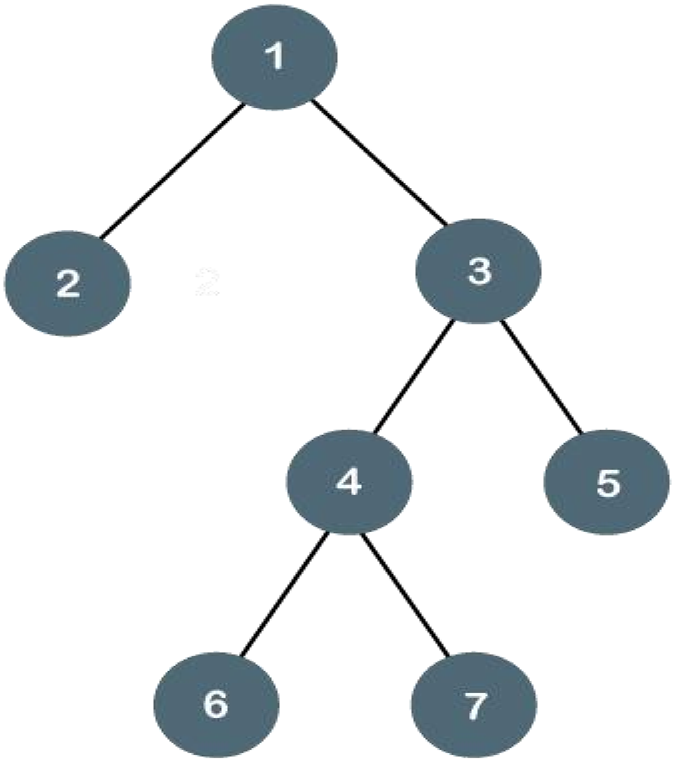
Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



Let's look at a simple example of a perfect binary tree.

The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.

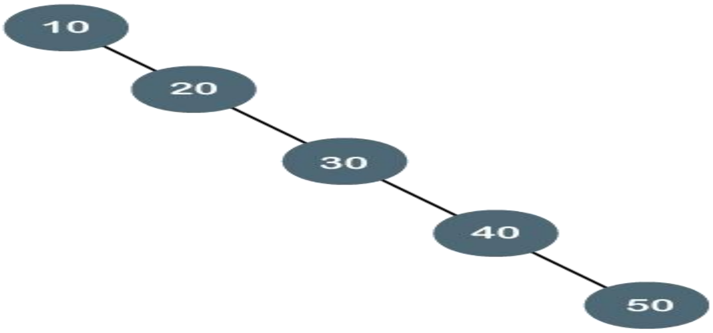


> *Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.*

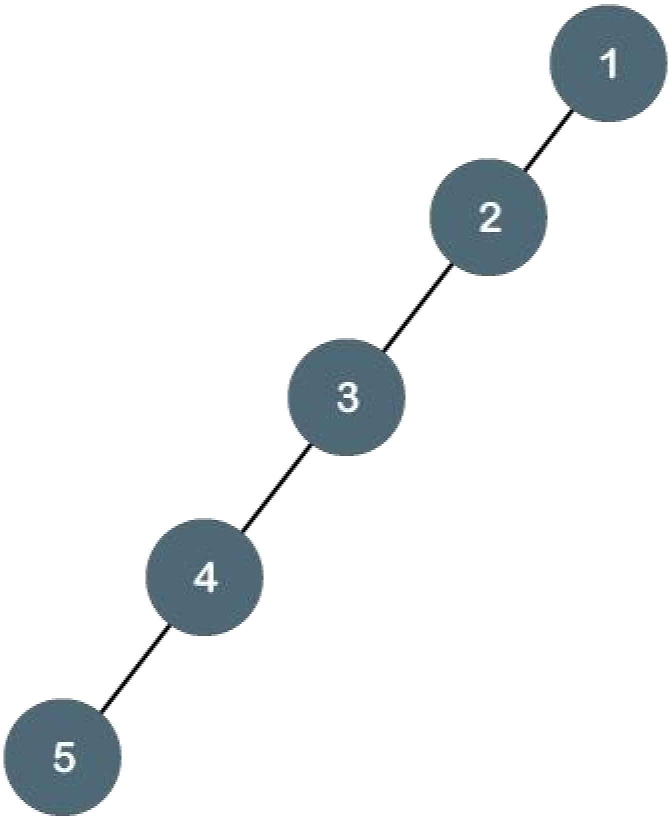
Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.

Let's understand the Degenerate binary tree through examples.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

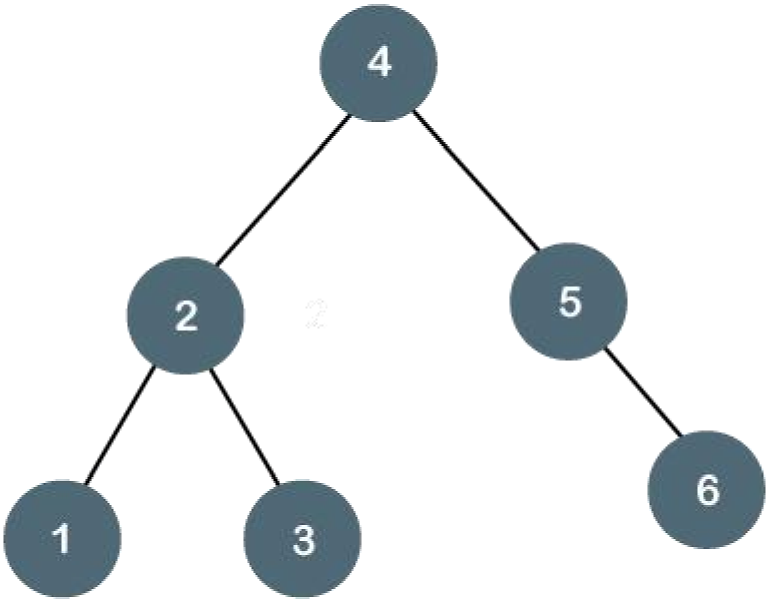


The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

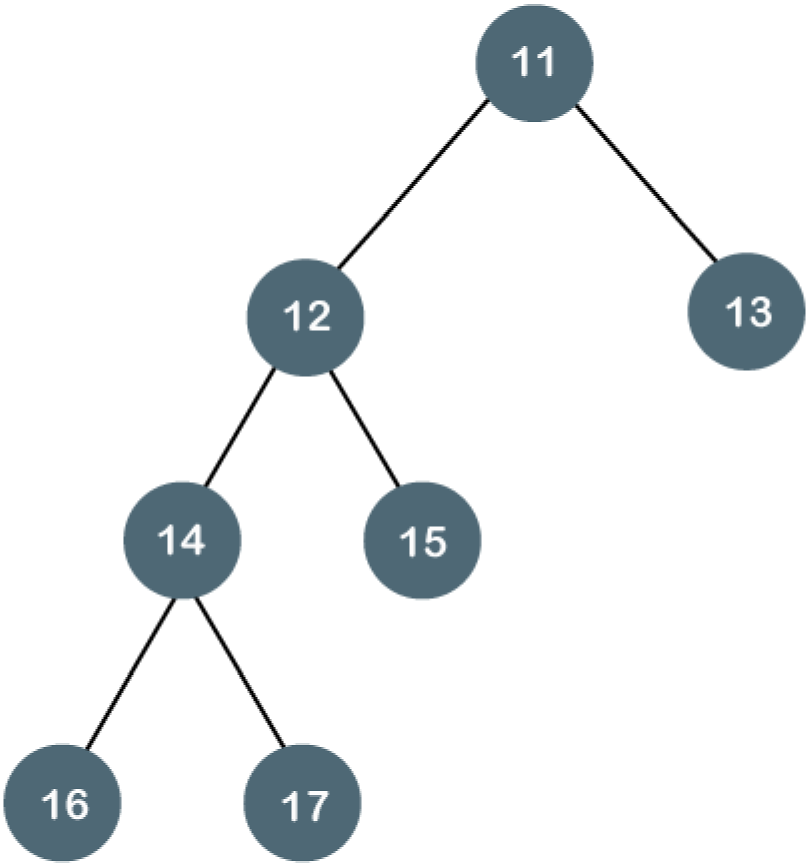
Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

Let's understand the balanced binary tree through examples.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.



The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

Binary Tree Implementation

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

```
struct node
{
  int data,
  struct node *left, *right;
}
```

In the above structure, **data** is the value, **left pointer** contains the address of the left node, and **right pointer** contains the address of the right node.

Binary Tree program in C

```
#include<stdio.h>
    struct node
   {
       int data;
       struct node *left, *right;
    }
    void main()
   {
     struct node *root;
     root = create();
   }
struct node *create()
{
  struct node *temp;
  int data;
  temp = (struct node *)malloc(sizeof(struct node));
  printf("Press 0 to exit");
  printf("\nPress 1 for new node");
  printf("Enter your choice : ");
  scanf("%d", &choice);
  if(choice==0)
{
```

```
return 0;
}
else
{
  printf("Enter the data:");
  scanf("%d", &data);
  temp->data = data;
  printf("Enter the left child of %d", data);
  temp->left = create();
printf("Enter the right child of %d", data);
temp->right = create();
return temp;
}
}
```

The above code is calling the create() function recursively and creating new node on each recursive call. When all the nodes are created, then it forms a binary tree structure. The process of visiting the nodes is known as tree traversal. There are three types traversals used to visit a node:

- Inorder traversal
- Preorder traversal
- Postorder traversal

Binary Search tree

In this article, we will discuss the Binary search tree. This article will be very helpful and informative to the students with technical background as it is an important topic of their course.

Before moving directly to the binary search tree, let's first see a brief description of the tree.

What is a tree?

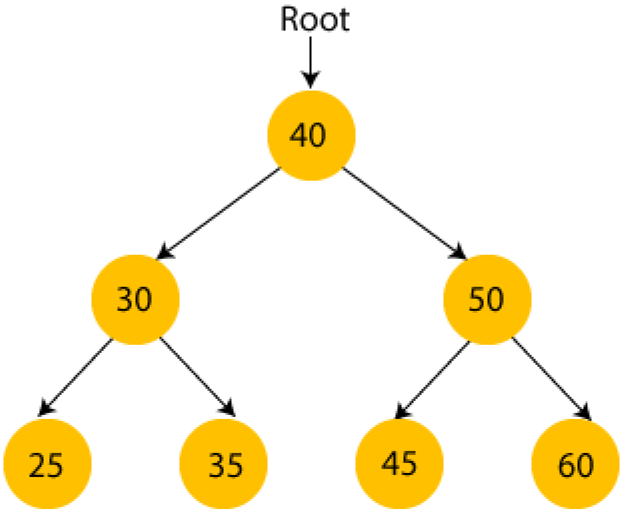
A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called as nodes that are linked together to simulate a hierarchy. Tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially.

Now, let's start the topic, the Binary Search tree.

What is a Binary Search tree?

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

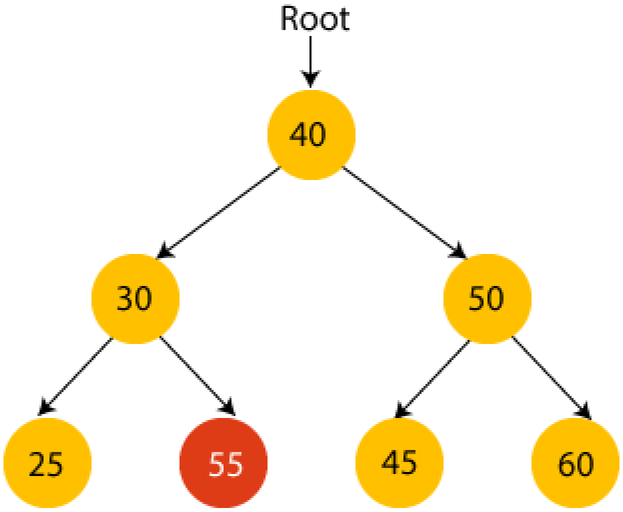
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a binary search tree

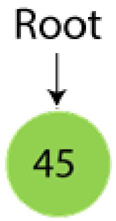
Now, let's see the creation of binary search tree using an example.

Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

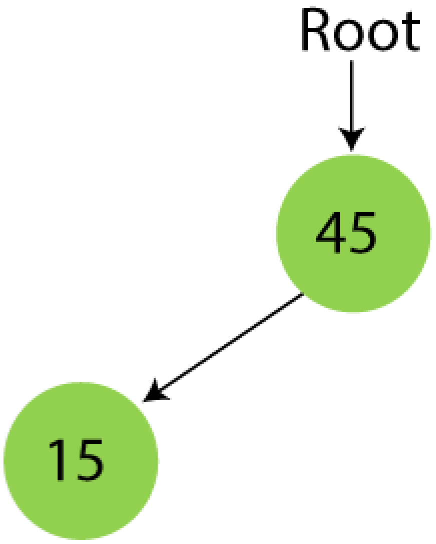
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.



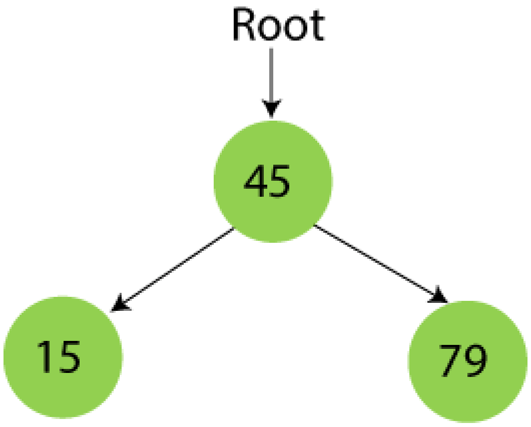
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



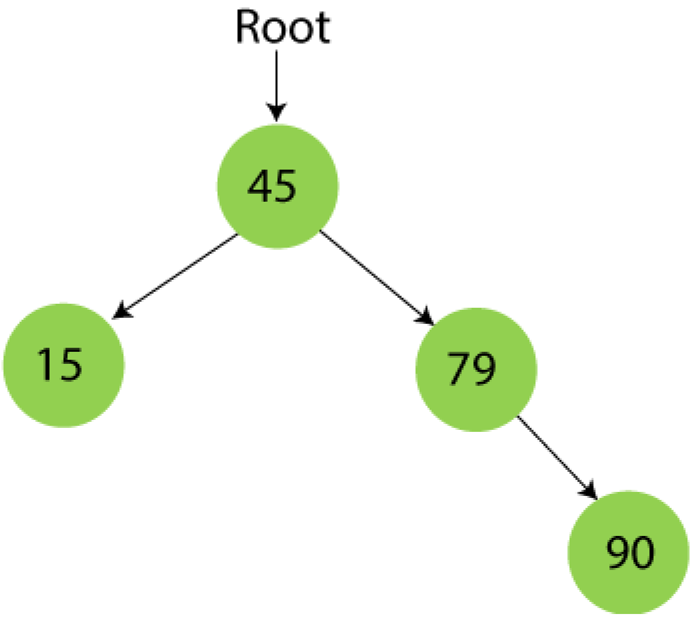
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



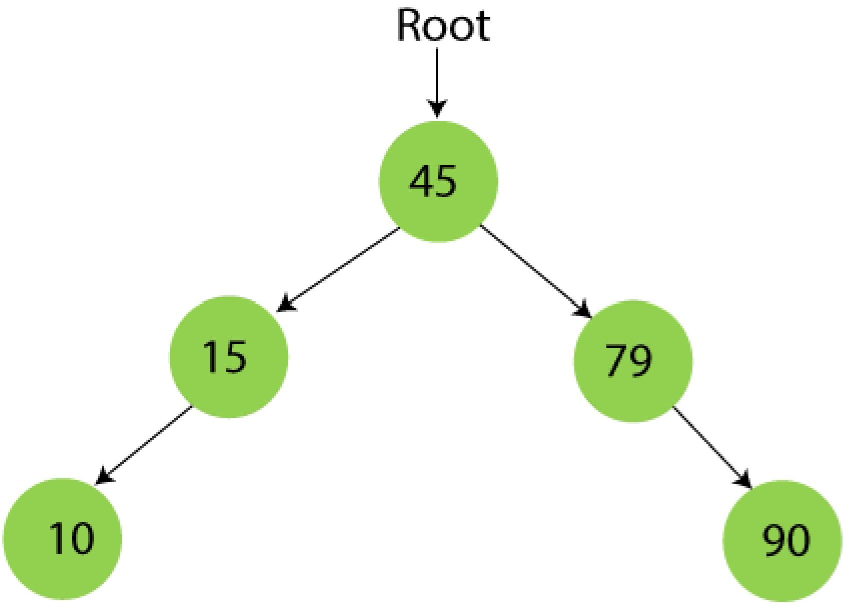
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



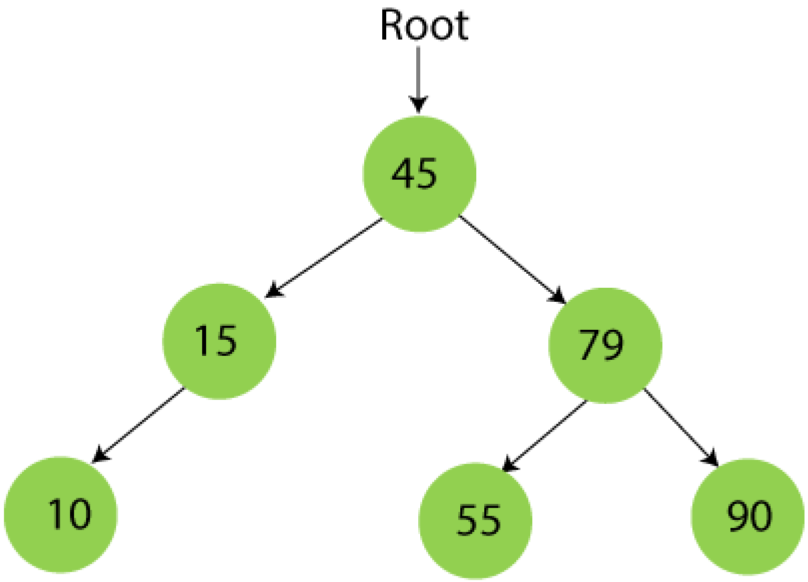
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



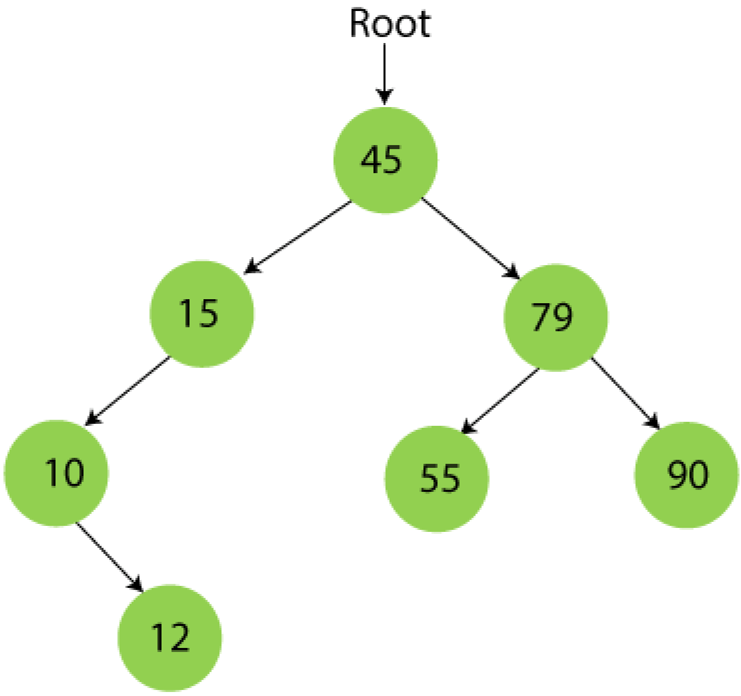
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



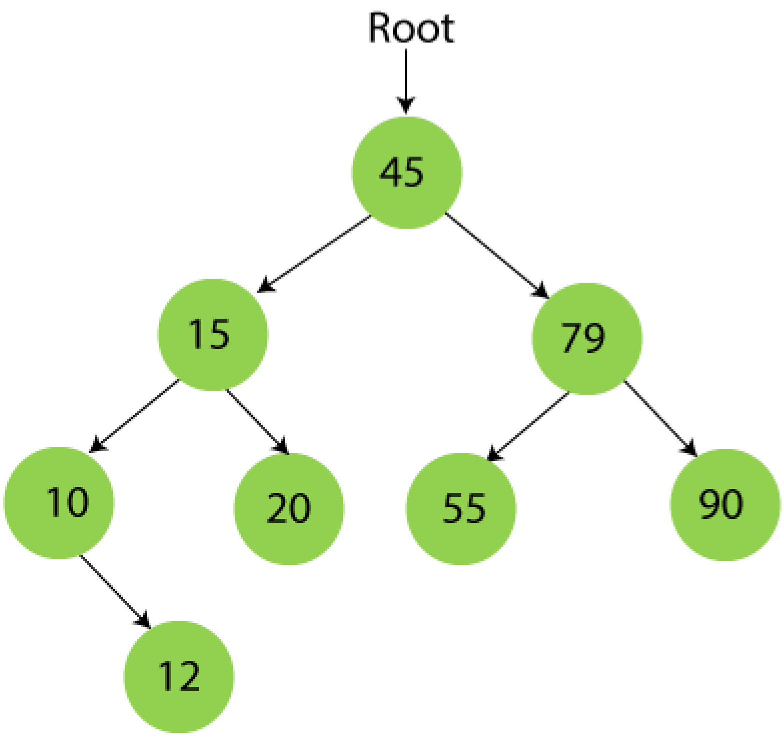
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



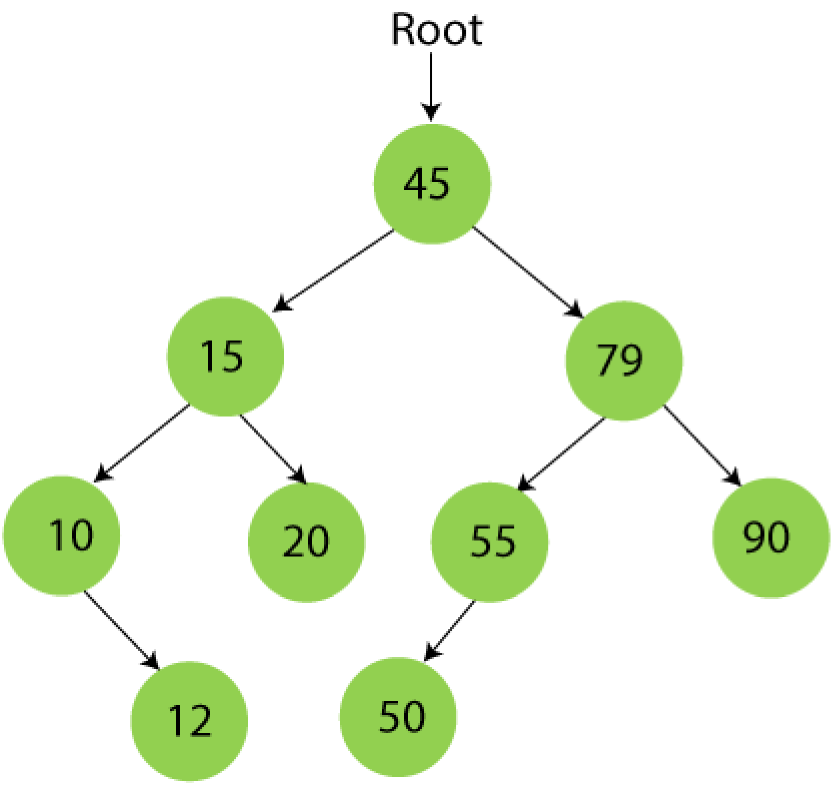
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

Let's understand how a search is performed on a binary search tree.

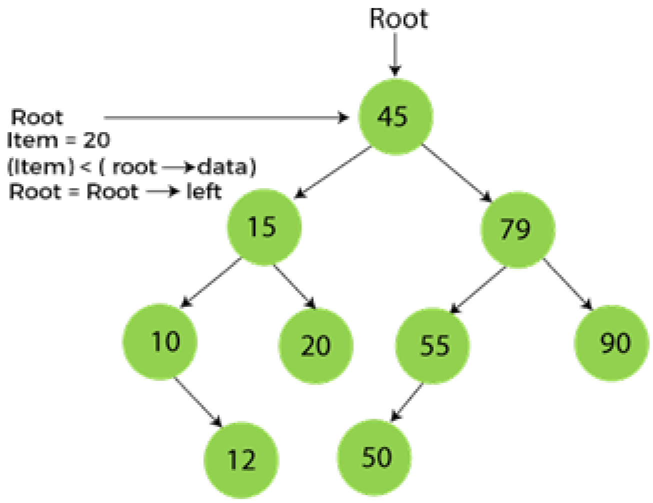
Searching in Binary search tree

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

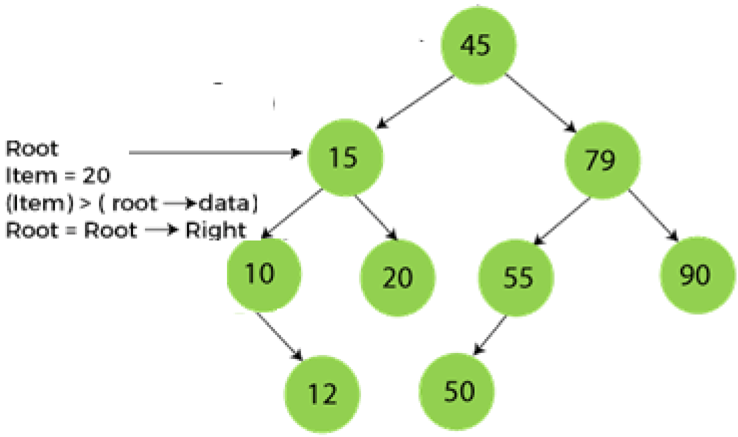
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

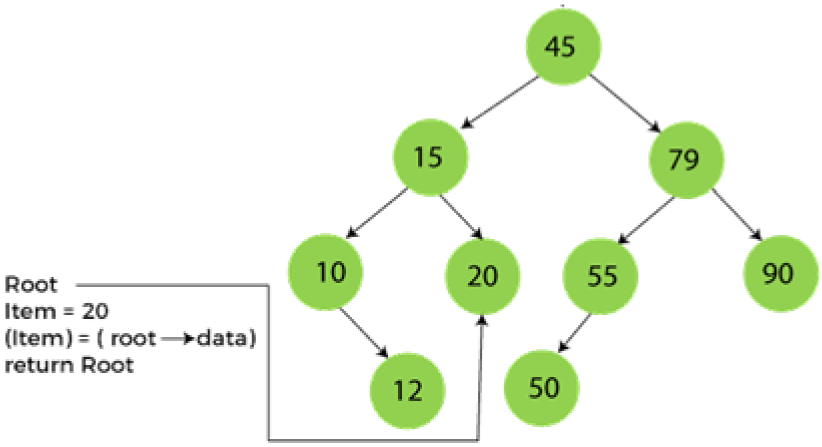
Step1:



Step2:



Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

Algorithm to search an element in Binary search tree

```
Search (root, item)
Step 1 - if (item = root → data) or (root = NULL)
return root
else if (item < root → data)
return Search(root → left, item)
else
return Search(root → right, item)
END if
Step 2 - END
```

Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.

Deletion in Binary Search tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

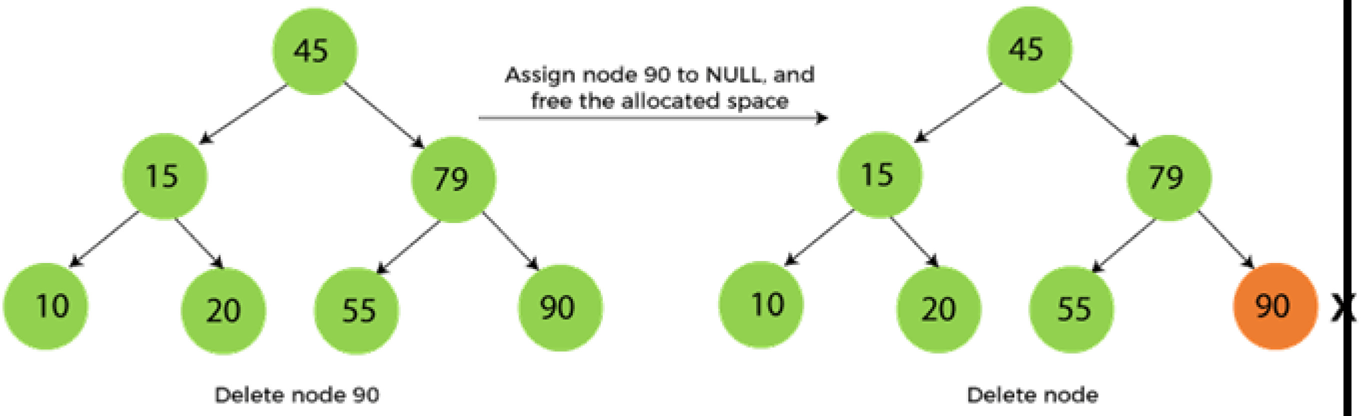
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

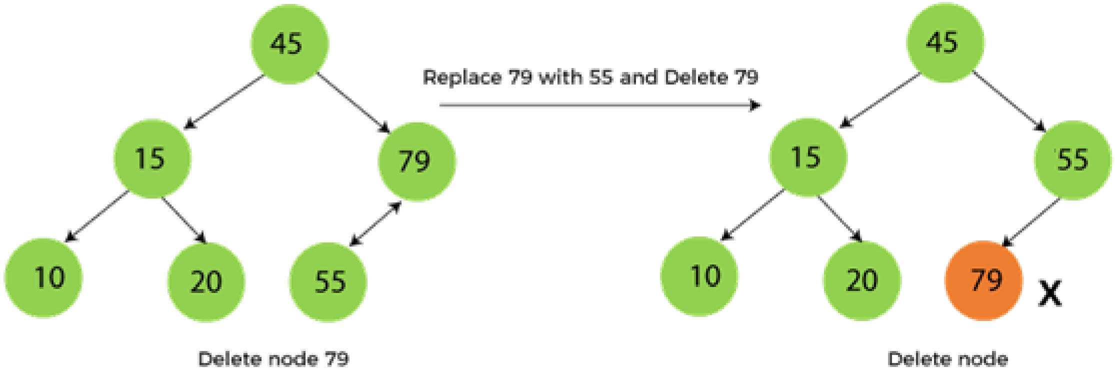


When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



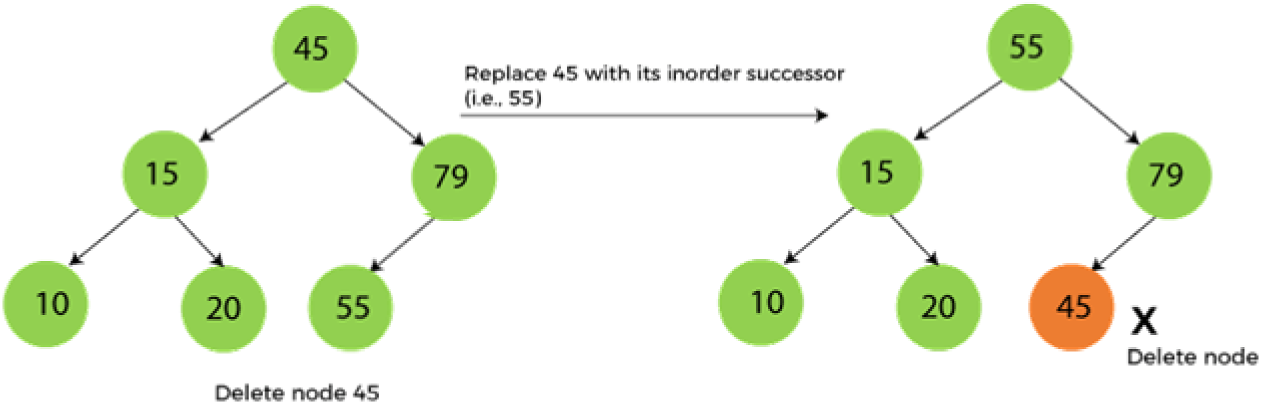
When the node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



Now let's understand how insertion is performed on a binary search tree.

Insertion in Binary Search tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

B+ Tree

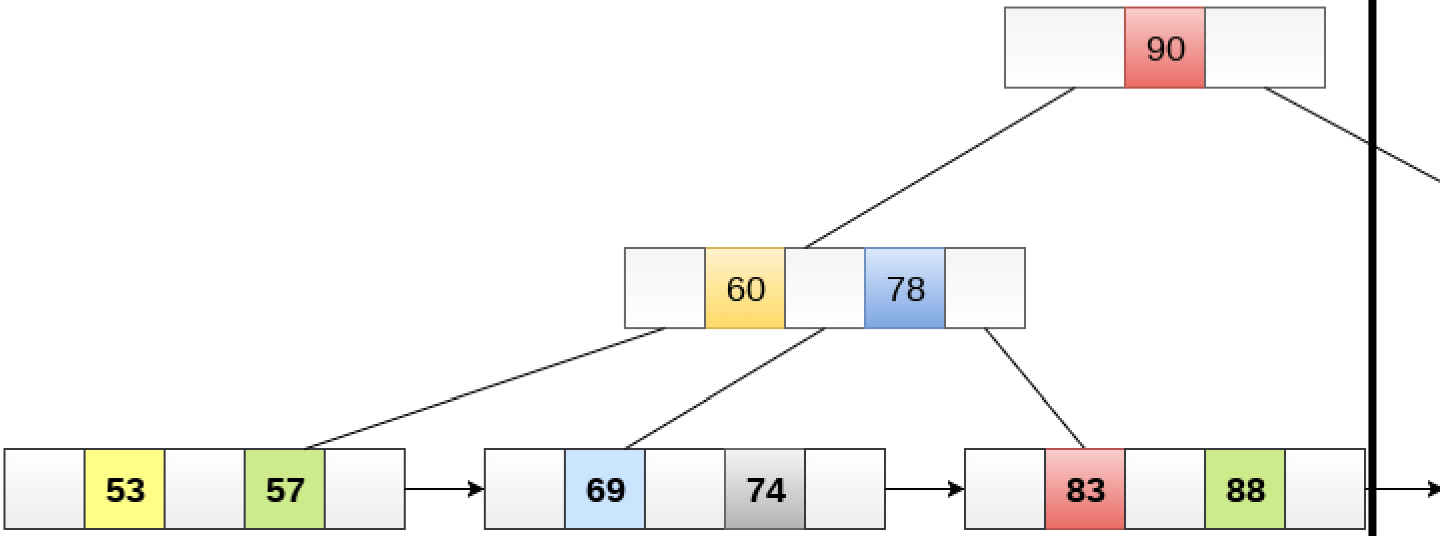
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Insertion in B+ Tree

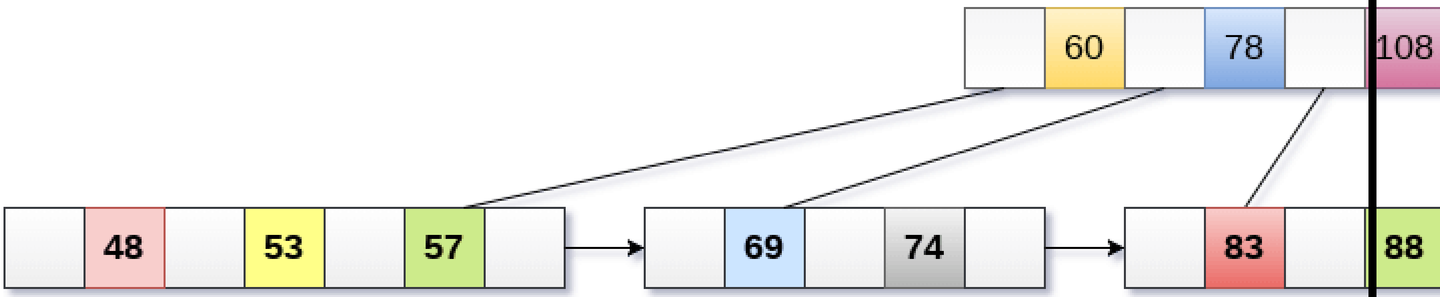
Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

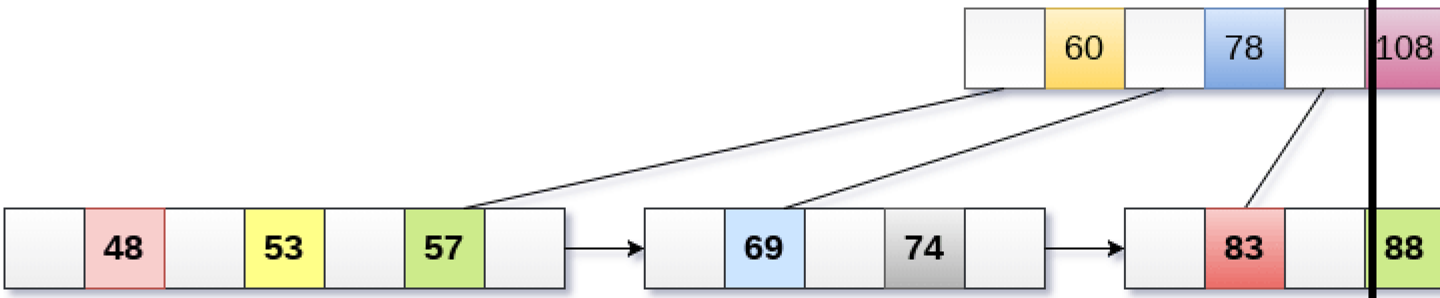
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example :

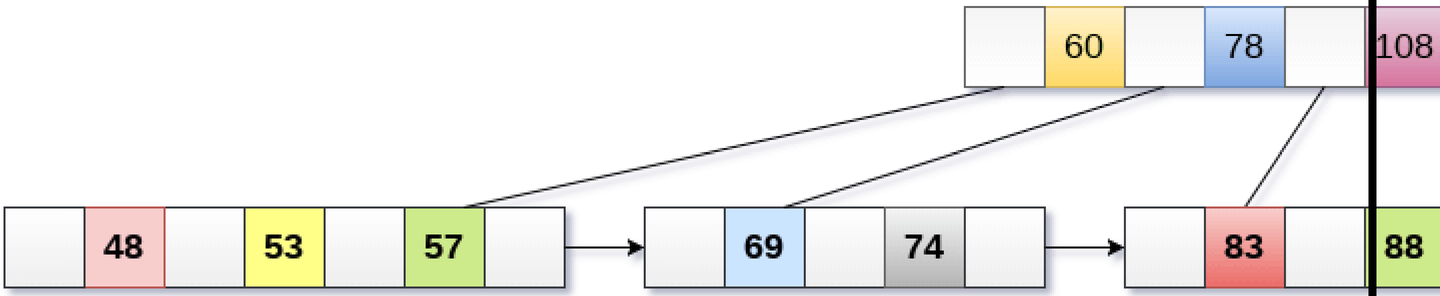
Insert the value 195 into the B+ tree of order 5 shown in the following figure.



195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.

Graph Introduction:

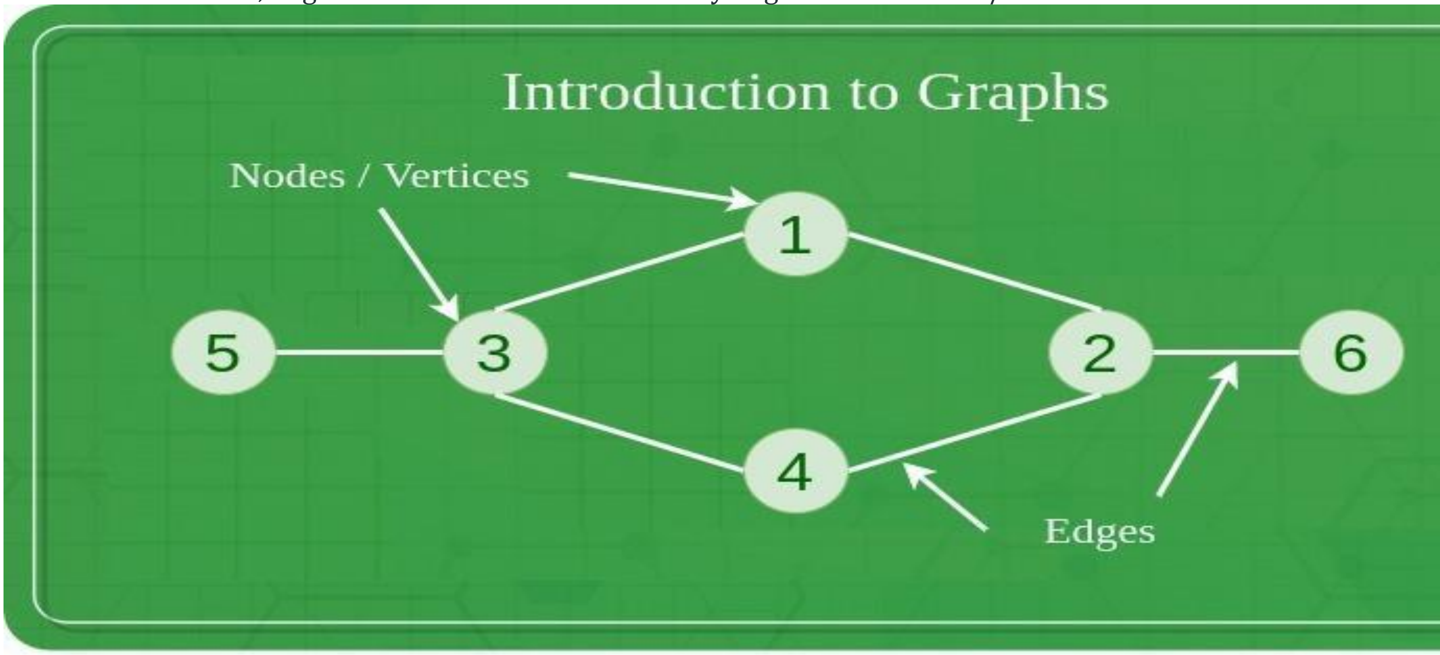
A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The graph is denoted by **G(V, E).**

Graph data structures are a powerful tool for representing and analyzing complex relationships between objects or entities. They are particularly useful in fields such as social network analysis, recommendation systems, and computer networks. In the field of sports data science, graph data structures can be used to analyze and understand the dynamics of team performance and player interactions on the field.

Imagine a game of football as a web of connections, where players are the nodes and their interactions on the field are the edges. This web of connections is exactly what a graph data structure represents, and it's the key to unlocking insights into team performance and player dynamics in sports.

Components of a Graph

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.



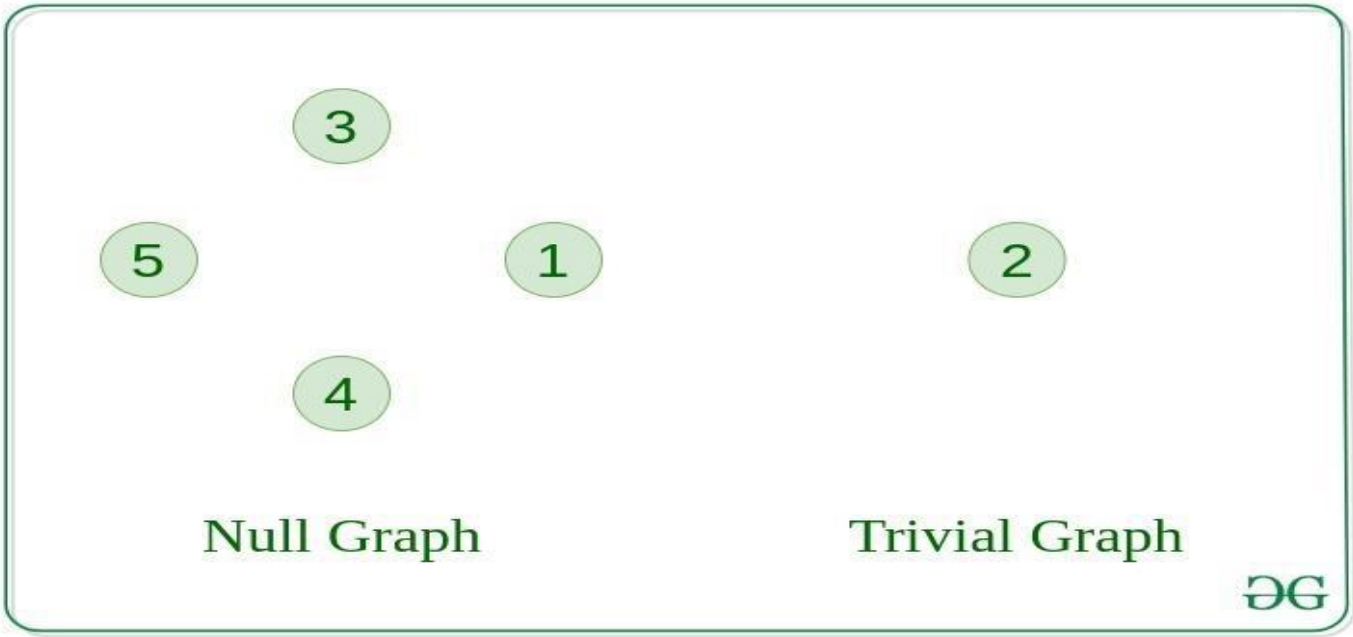
Types Of Graph

1. Null Graph

A graph is known as a null graph if there are no edges in the graph.

2. Trivial Graph

Graph having only a single vertex, it is also the smallest graph possible.

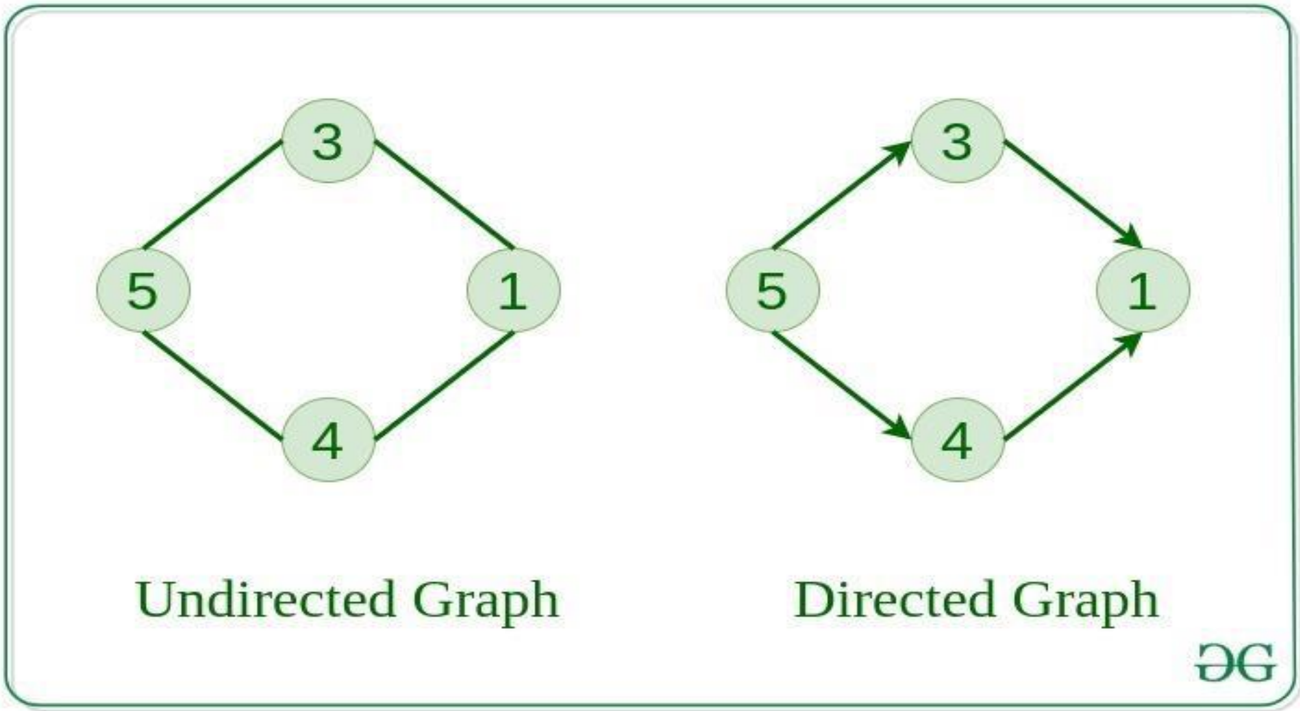


3. Undirected Graph

A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

4. Directed Graph

A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.

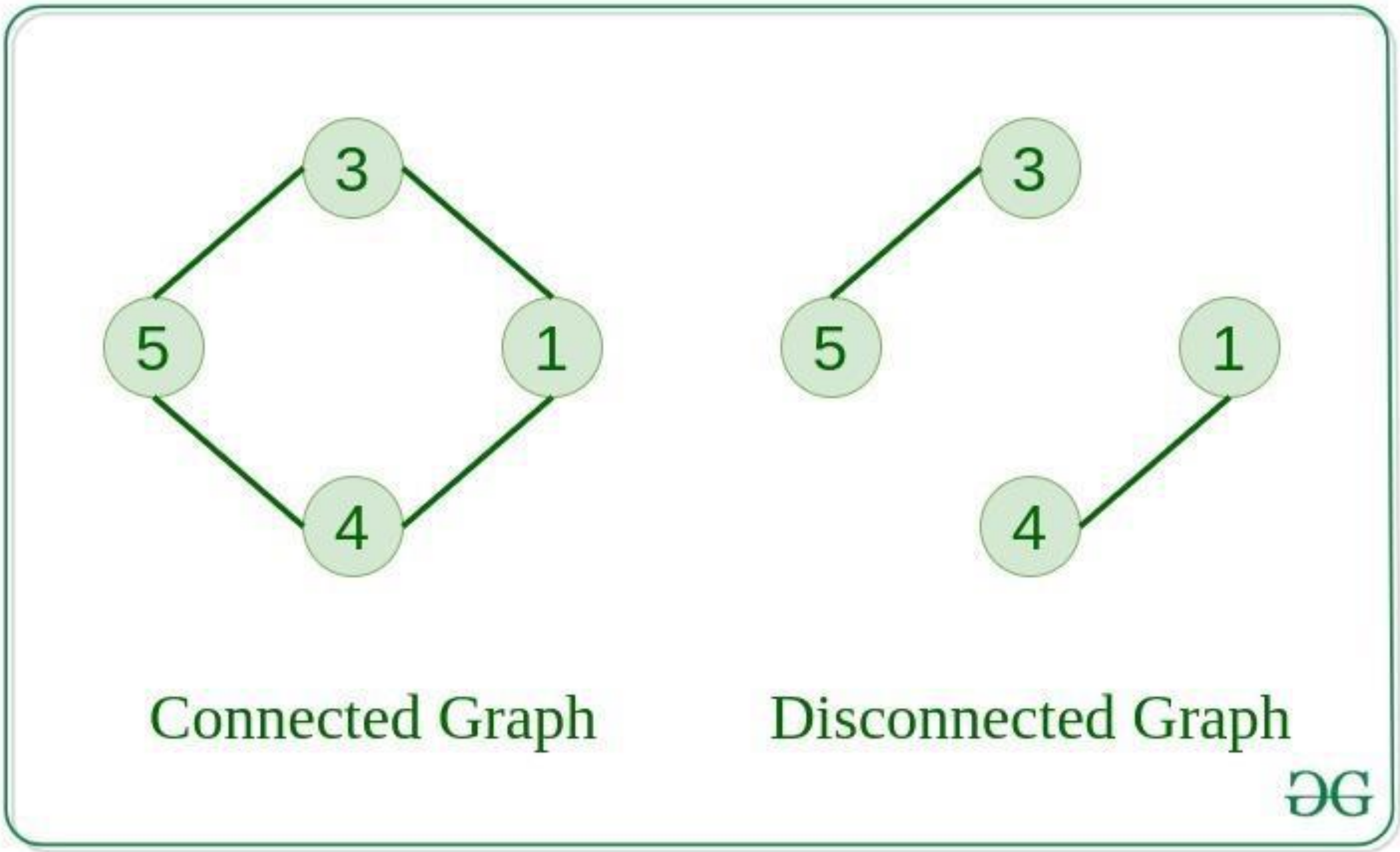


5. Connected Graph

The graph in which from one node we can visit any other node in the graph is known as a connected graph.

6. Disconnected Graph

The graph in which at least one node is not reachable from a node is known as a disconnected graph.



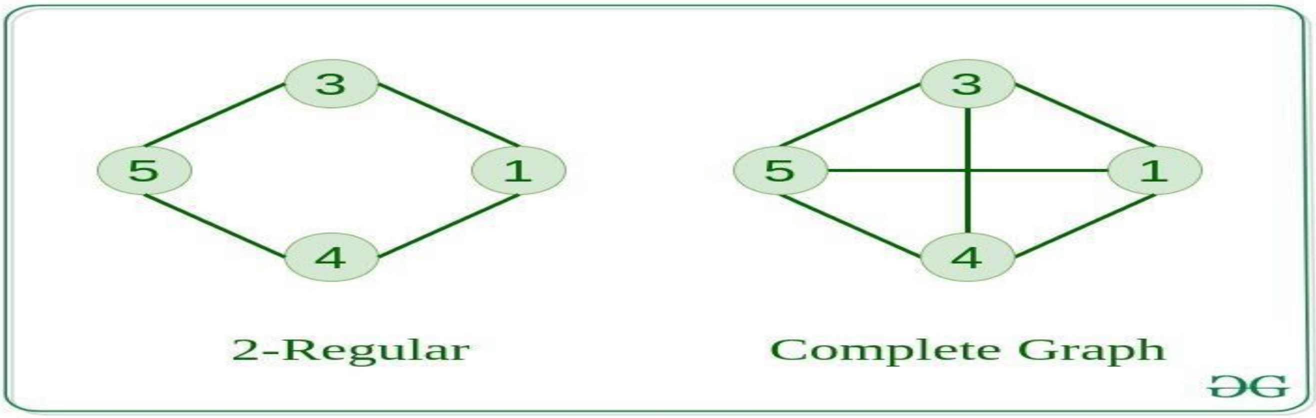
7. Regular Graph

The graph in which the degree of every vertex is equal to K is called K regular graph.

8. Complete Graph

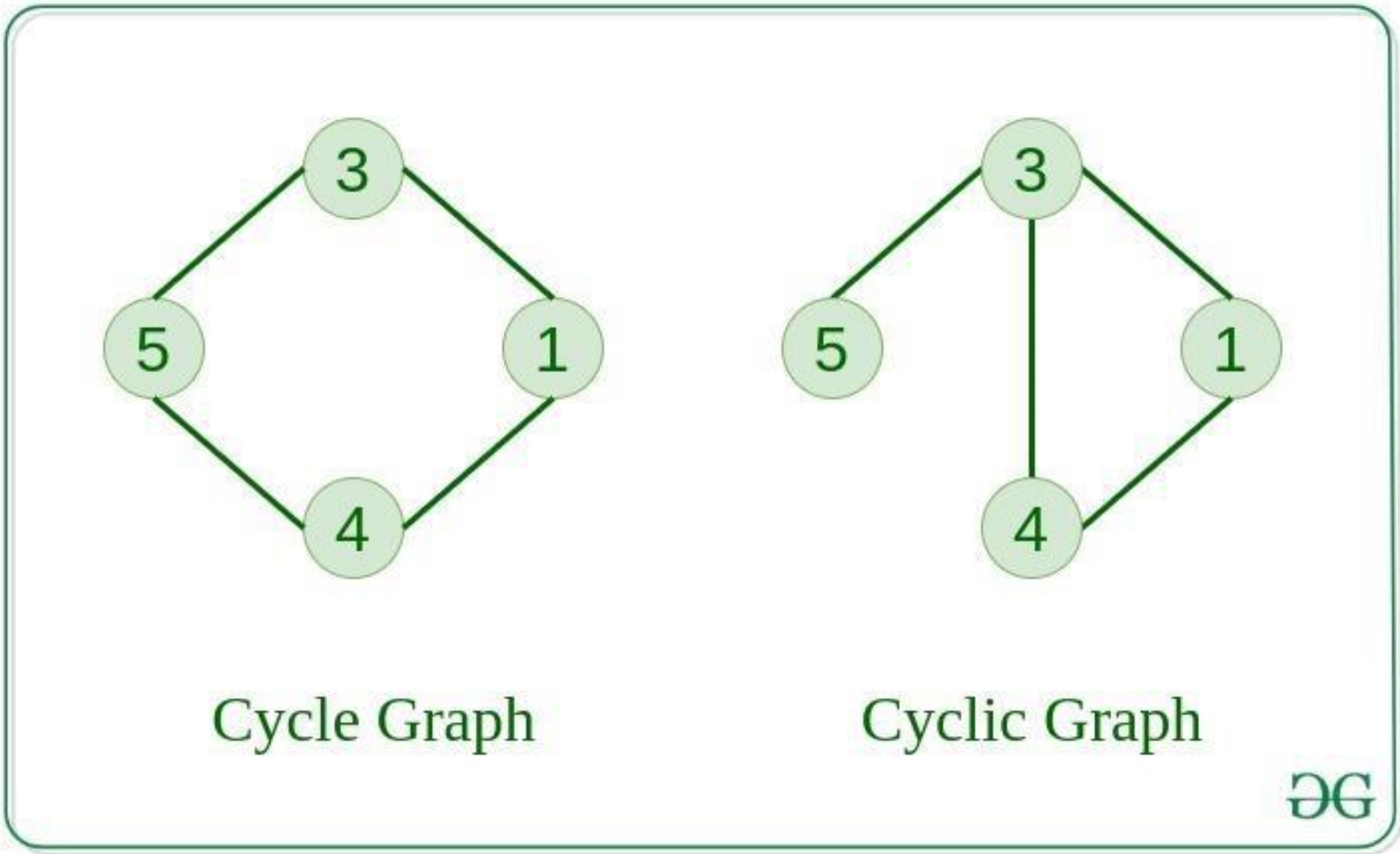
The graph in which from each node there is an edge to each other node.

.



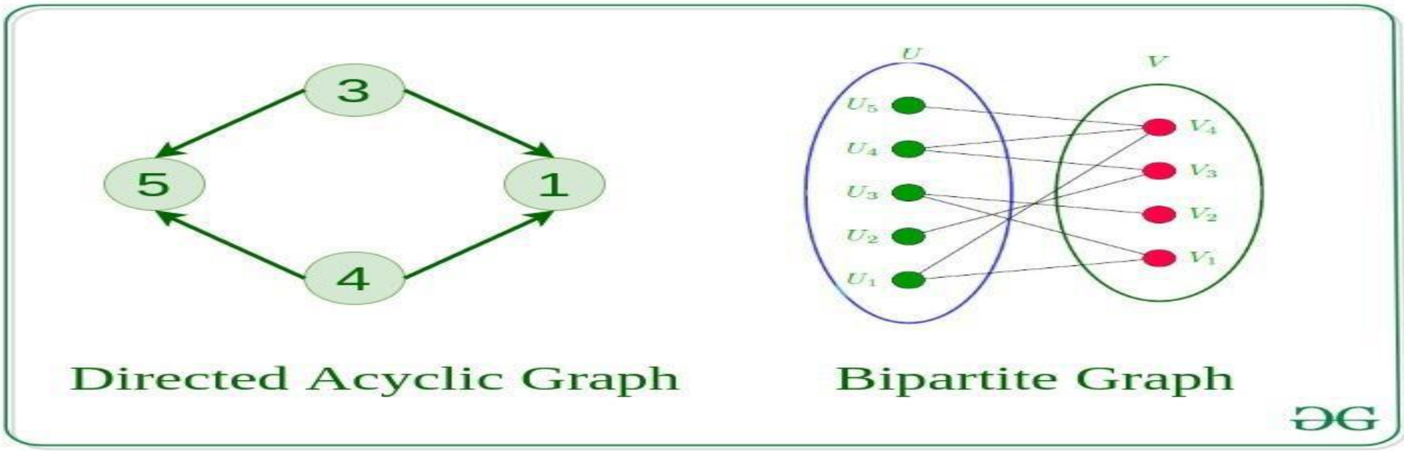
9. Cycle Graph
The graph in which the graph is a cycle in itself, the degree of each vertex is 2.

10. Cyclic Graph
A graph containing at least one cycle is known as a Cyclic graph.



11. Directed Acyclic Graph
A Directed Graph that does not contain any cycle.

12. Bipartite Graph
A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.

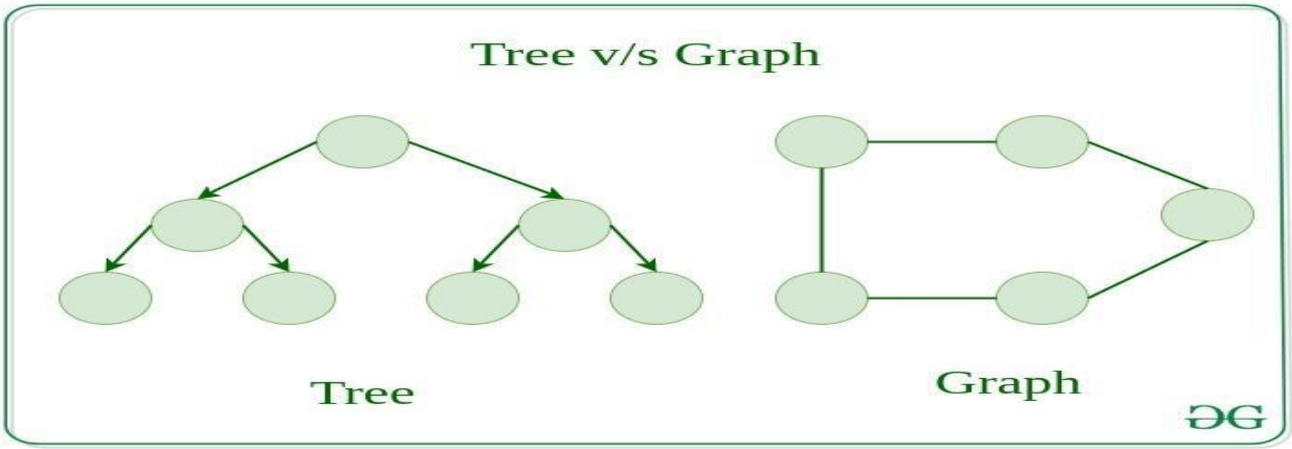


13. Weighted Graph

- A graph in which the edges are already specified with suitable weight is known as a weighted graph.
- Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

Tree v/s Graph

Trees are the restricted types of graphs, just with some more rules. Every tree will always be a graph but not all graphs will be trees. Linked List, Trees, and Heaps all are special cases of graphs.



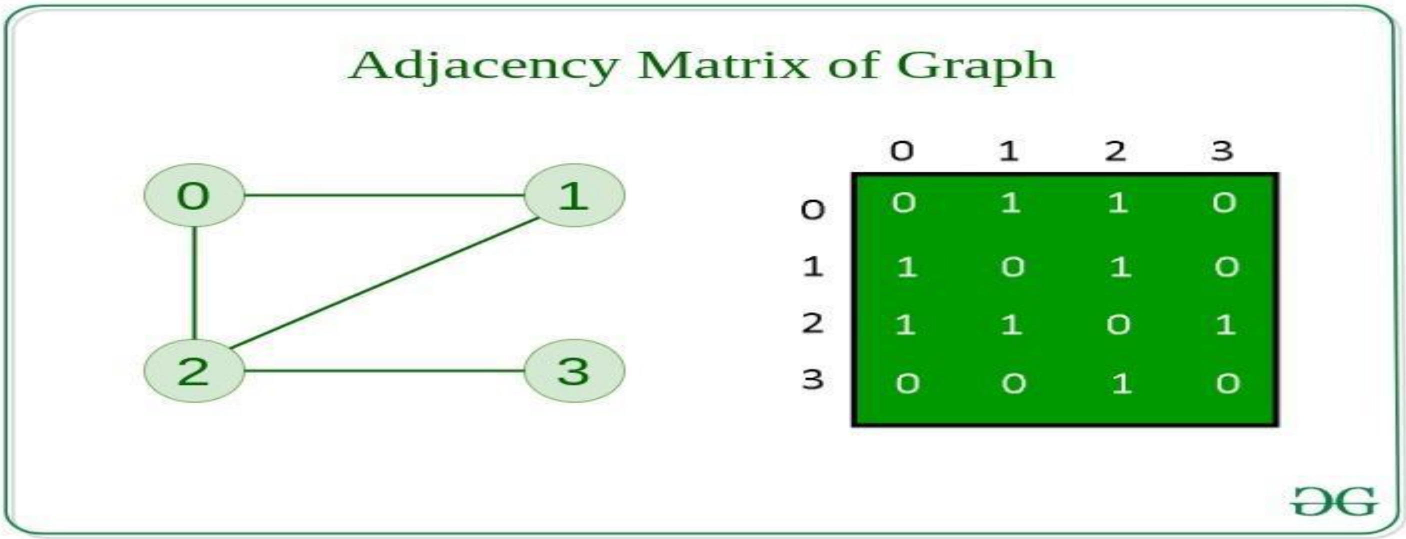
Representation of Graphs

There are two ways to store a graph:

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

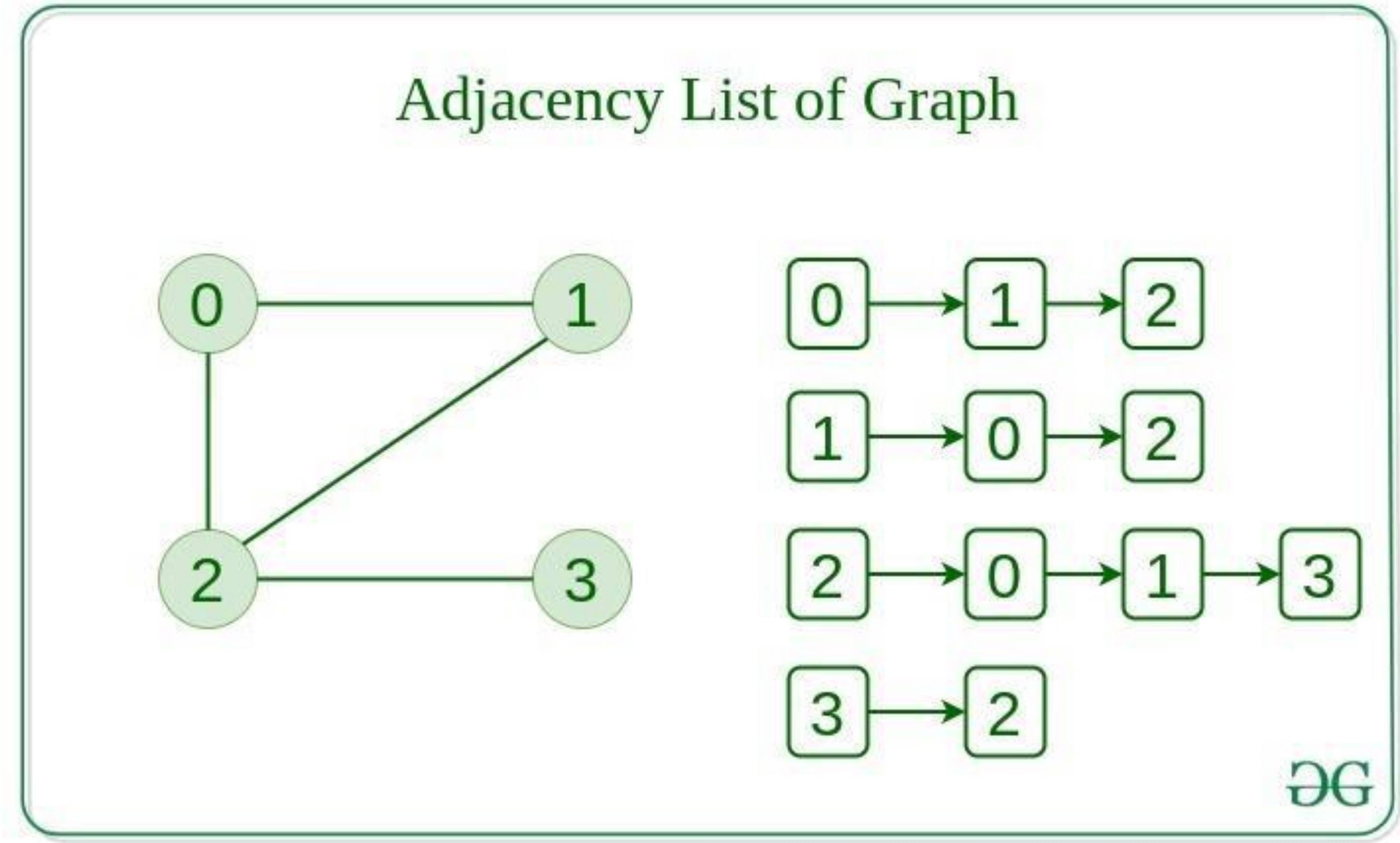
In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.



	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

Adjacency List

This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.



Comparison between Adjacency Matrix and Adjacency List

When the graph contains a large number of edges then it is good to store it as a matrix because only some entries in the matrix will be empty. An algorithm such as Prim's and Dijkstra adjacency matrix is used to have less complexity.

Action	Adjacency Matrix	Adjacency List
Adding Edge	O(1)	O(1)
Removing an edge	O(1)	O(N)
Initializing	O(N*N)	O(N)

Basic Operations on Graphs

Below are the basic operations on the graph:

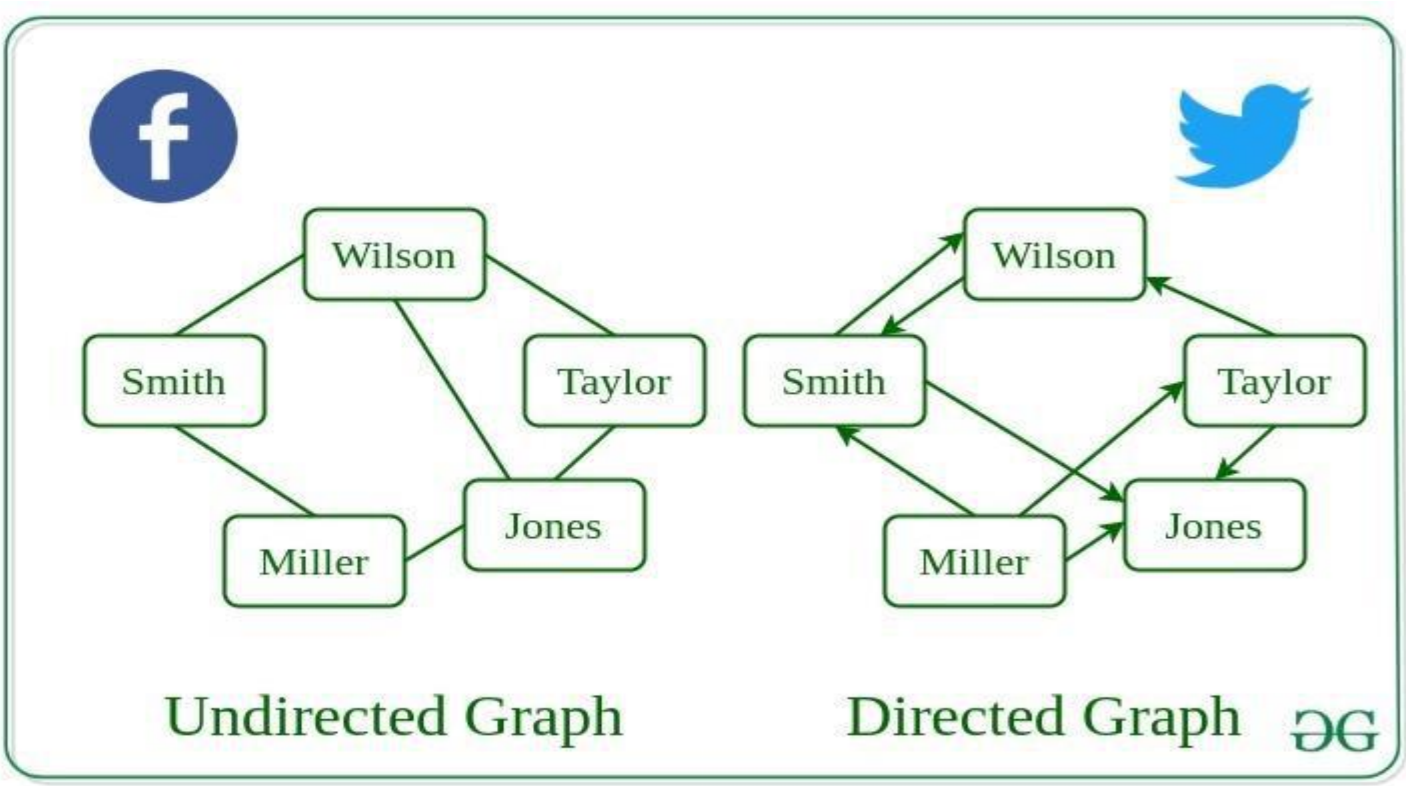
- Insertion of Nodes/Edges in the graph – Insert a node into the graph.
- Deletion of Nodes/Edges in the graph – Delete a node from the graph.

- Searching on Graphs – Search an entity in the graph.
- Traversal of Graphs – Traversing all the nodes in the graph.

Usage of graphs

- Maps can be represented using graphs and then can be used by computers to provide various services like the shortest path between two cities.
- When various tasks depend on each other then this situation can be represented using a Directed Acyclic graph and we can find the order in which tasks can be performed using topological sort.
- State Transition Diagram represents what can be the legal moves from current states. In-game of tic tac toe this can be used.

Real-Life Applications of Graph



Following are the real-life applications:

- Graph data structures can be used to represent the interactions between players on a team, such as passes, shots, and tackles. Analyzing these interactions can provide insights into team dynamics and areas for improvement.
- Commonly used to represent social networks, such as networks of friends on social media.
- Graphs can be used to represent the topology of computer networks, such as the connections between routers and switches.
- Graphs are used to represent the connections between different places in a transportation network, such as roads and airports.
- **Neural Networks:** Vertices represent neurons and edges represent the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^11 neurons and close to 10^15 synapses.

- **Compilers:** Graphs are used extensively in compilers. They can be used for type inference, for so-called data flow analysis, register allocation, and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
- **Robot planning:** Vertices represent states the robot can be in and the edges the possible transitions between the states. Such graph plans are used, for example, in planning paths for autonomous vehicles.

When to use Graphs:

- When you need to represent and analyze the relationships between different objects or entities.
- When you need to perform network analysis.
- When you need to identify key players, influencers or bottlenecks in a system.
- When you need to make predictions or recommendations.
- Modeling networks: Graphs are commonly used to model various types of networks, such as social networks, transportation networks, and computer networks. In these cases, vertices represent nodes in the network, and edges represent the connections between them.
- Finding paths: Graphs are often used in algorithms for finding paths between two vertices in a graph, such as shortest path algorithms. For example, graphs can be used to find the fastest route between two cities on a map or the most efficient way to travel between multiple destinations.
- Representing data relationships: Graphs can be used to represent relationships between data objects, such as in a database or data structure. In these cases, vertices represent data objects, and edges represent the relationships between them.
- Analyzing data: Graphs can be used to analyze and visualize complex data, such as in data clustering algorithms or machine learning models. In these cases, vertices represent data points, and edges represent the similarities or differences between them.

However, there are also some scenarios where using a graph may not be the best approach. For example, if the data being represented is very simple or structured, a graph may be overkill and a simpler data structure may suffice. Additionally, if the graph is very large or complex, it may be difficult or computationally expensive to analyze or traverse, which could make using a graph less desirable.

Advantages and Disadvantages:

Advantages:

1. Graphs are a versatile data structure that can be used to represent a wide range of relationships and data structures.
2. They can be used to model and solve a wide range of problems, including pathfinding, data clustering, network analysis, and machine learning.
3. Graph algorithms are often very efficient and can be used to solve complex problems quickly and effectively.
4. Graphs can be used to represent complex data structures in a simple and intuitive way, making them easier to understand and analyze.

Disadvantages:

1. Graphs can be complex and difficult to understand, especially for people who are not familiar with graph theory or related algorithms.
2. Creating and manipulating graphs can be computationally expensive, especially for very large or complex graphs.

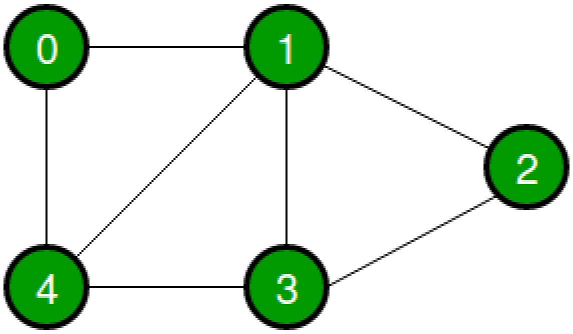
3. Graph algorithms can be difficult to design and implement correctly, and can be prone to bugs and errors.
4. Graphs can be difficult to visualize and analyze, especially for very large or complex graphs, which can make it challenging to extract meaningful insights from the data.

Graph and its representations

Graph is a data structure that consists of the following two components:

- A finite set of vertices also called nodes.
- A finite set of ordered pair of the form (u, v) called edge. The pair is ordered because (u, v) is not the same as (v, u) in the case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Following is an example of an undirected graph with 5 vertices.



Example of undirected graph with 5 vertices

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale. See this for more applications of graph.
In computer science, a graph is a data structure that is used to represent connections or relationships between objects. A graph consists of a set of vertices (also known as nodes) and a set of edges (also known as arcs) that connect the vertices. The vertices can represent anything from cities in a map to web pages in a network, and the edges can represent the relationships between them, such as roads or links.

A graph can be visualized as a collection of points (vertices) connected by lines (edges), where each vertex represents a point of interest and each edge represents a connection between two points. The edges can be directed or undirected, meaning they can either have a specific direction or be bidirectional. For example, a map of a city may have directed edges that represent the direction of one-way streets, while a social network may have undirected edges that represent friendships between individuals.

Representations of Graphs:

The following two are the most commonly used representations of a graph.

- Adjacency Matrix
- Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and the ease of use.

Adjacency List:

An adjacency list is a simple way to represent a graph as a list of vertices, where each vertex has a list of its adjacent vertices. Here's an example of an adjacency list for an undirected graph with 4 vertices:

makefile

Copy code

```
0: 1 3

1: 0 2

2: 1 3

3: 0 2
```

In this example, vertex 0 is adjacent to vertices 1 and 3, vertex 1 is adjacent to vertices 0 and 2, and so on.

Adjacency Matrix:

An adjacency matrix is a two-dimensional array that represents the graph by storing a 1 at position (i,j) if there is an edge from vertex i to vertex j, and 0 otherwise. Here's an example of an adjacency matrix for the same undirected graph:

Copy code

```
  0 1 2 3

0 0 1 0 1

1 1 0 1 0

2 0 1 0 1

3 1 0 1 0
```

In this example, there is an edge from vertex 0 to vertex 1 (represented by a 1 in position (0,1)), an edge from vertex 1 to vertex 0 (represented by a 1 in position (1,0)), and so on.

Incidence Matrix:

An incidence matrix is a two-dimensional array that represents the graph by storing a 1 at position (i,j) if vertex i is incident on edge j, and 0 otherwise. Here's an example of an incidence matrix for the same undirected graph:

Copy code

```
  0 1 2 3

0 1 1 0 1

1 1 0 1 0
```

2 0 1 1 0

3 1 0 0 1

In this example, vertex 0 is incident on edges 0, 1, and 3 (represented by a 1 in positions (0,0), (0,1), and (0,3)), vertex 1 is incident on edges 0, 2 (represented by a 1 in positions (1,0) and (1,2)), and so on.

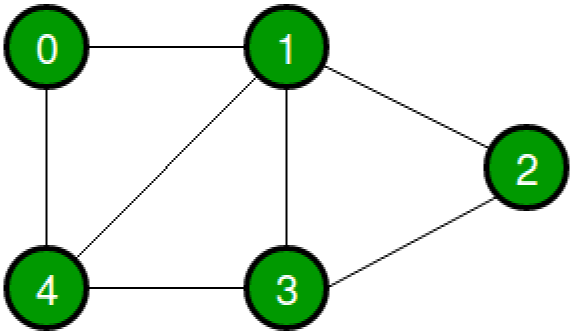
Each representation has its own advantages and disadvantages depending on the application, and choosing the right representation can have a significant impact on the performance of graph algorithms.

Adjacency Matrix:

*Adjacency Matrix is a 2D array of size **V x V** where **V** is the number of vertices in a graph. Let the 2D array be **adj[][]**, a slot **adj[i][j] = 1** indicates that there is an edge from vertex **i** to vertex **j**. The adjacency matrix for an undirected graph is always symmetric.*
*Adjacency Matrix is also used to represent weighted graphs. If **adj[i][j] = w**, then there is an edge from vertex **i** to vertex **j** with weight **w**.*
We follow the below pattern to use the adjacency matrix in code:

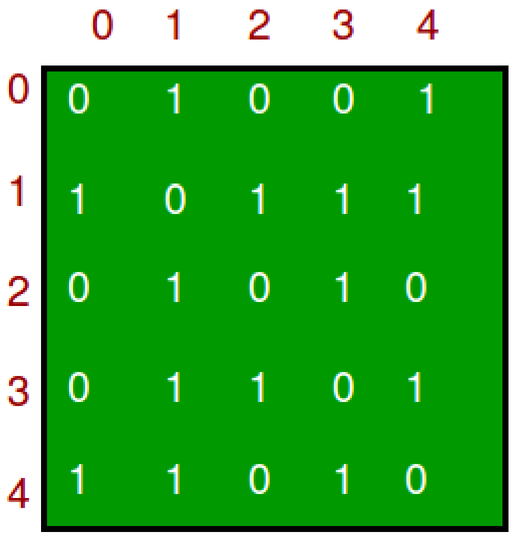
- In the case of an undirected graph, we need to show that there is an edge from vertex **i** to vertex **j** and vice versa. In code, we assign adj[i][j] = 1 and adj[j][i] = 1.
- In the case of a directed graph, if there is an edge from vertex **i** to vertex **j** then we just assign **adj[i][j]=1**.\

See the undirected graph shown below:



Example of undirected graph with 5 vertices

The adjacency matrix for the above example graph is:



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency matrix representation

Advantages of Adjacency Matrix:

- Representation is easier to implement and follow.
- Removing an edge takes O(1) time.
- Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

Disadvantages of Adjacency Matrix:

- Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space.
- Adding a vertex takes O(V^2) time. Computing all neighbors of a vertex takes O(V) time (Not efficient).

Implementation of Adjacency Matrix:

- C
- C++
- Java
- Python3
- C#
- Javascript

```
if __name__ == '__main__':
```

```
#  n is the number of vertices

#  m is the number of edges

n, m = map(int, input().split())

adjMat = [[0 for i in range(n)]for j in range(n)]

for i in range(n):

    u, v = map(int, input().split())

    adjMat[u][v] = 1

    adjMat[v][u] = 1

    # for a directed graph with an edge

    # pointing from u to v,we just assign

    # adjMat[u][v] as 1
```

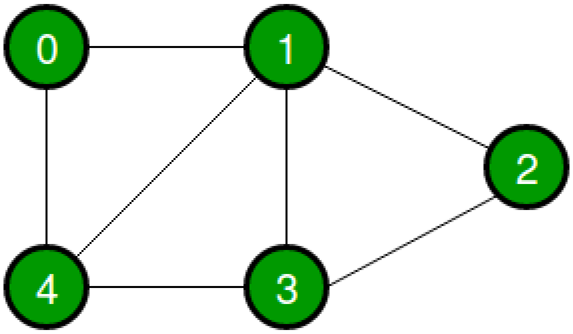
Adjacency List:

An array of linked lists is used. The size of the array is equal to the number of vertices. Let the array be an ***array[]****. An entry* ***array[i]*** *represents the linked list of vertices adjacent to the* ***i****th vertex.*
This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.

Recommended Problem

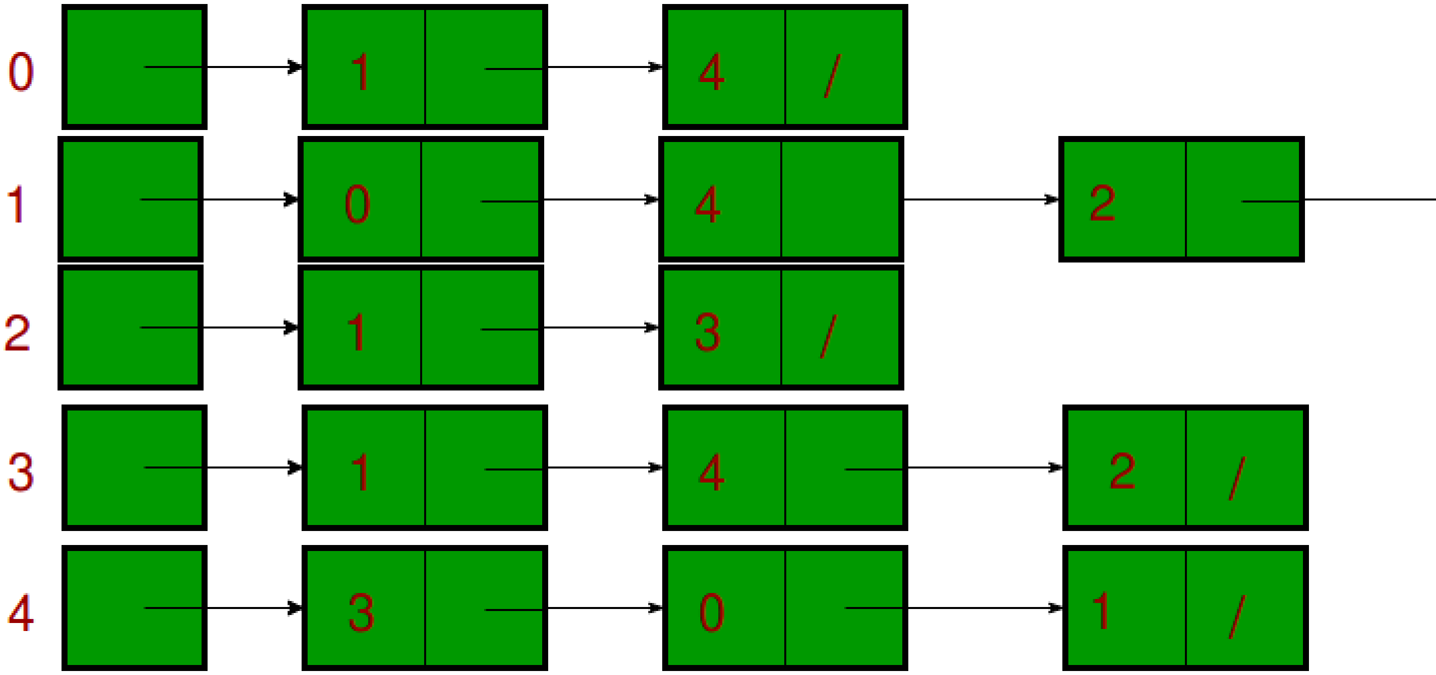
Print adjacency list

Consider the following graph:



Example of undirected graph with 5 vertices

Following is the adjacency list representation of the above graph.



Adjacency List representation of the above graph

Advantages of Adjacency List:

- Saves space. Space taken is O(|V|+|E|). In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space.
- Adding a vertex is easier.
- Computing all neighbors of a vertex takes optimal time.

Disadvantages of Adjacency List:

Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

Implementation of Adjacency List:

Note that in the below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of the linked list. The vector implementation has the advantage of cache friendliness.

Output

```
 Adjacency list of vertex 0
 head -> 1-> 4

 Adjacency list of vertex 1
 head -> 0-> 2-> 3-> 4

 Adjacency list of vertex 2
 head -> 1-> 3

 Adjacency list of vertex 3
 head -> 1-> 2-> 4

 Adjacency list of vertex 4
 head -> 0-> 1-> 3
```

Breadth First Search or BFS for a Graph

*The **Breadth First Search (BFS)** algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.*

Relation between BFS for Graph and Tree traversal:

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.

How does BFS work?

Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.

To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

Recommended Problem

BFS of graph

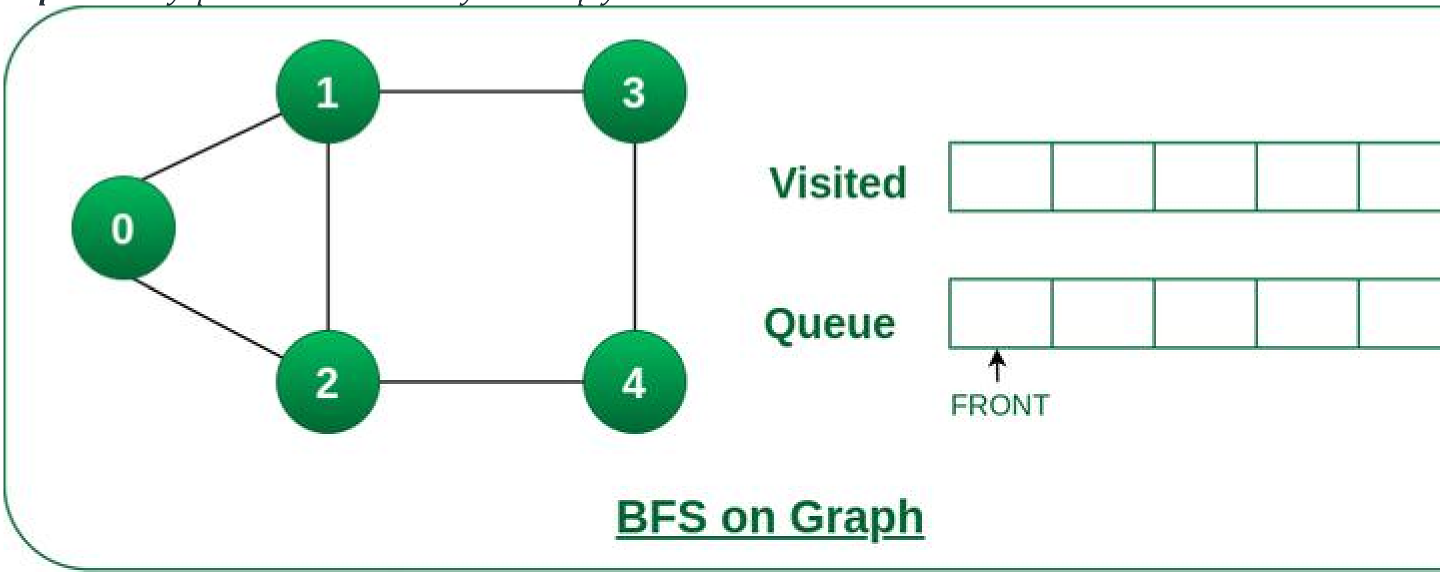
Graph

BFS

Illustration:

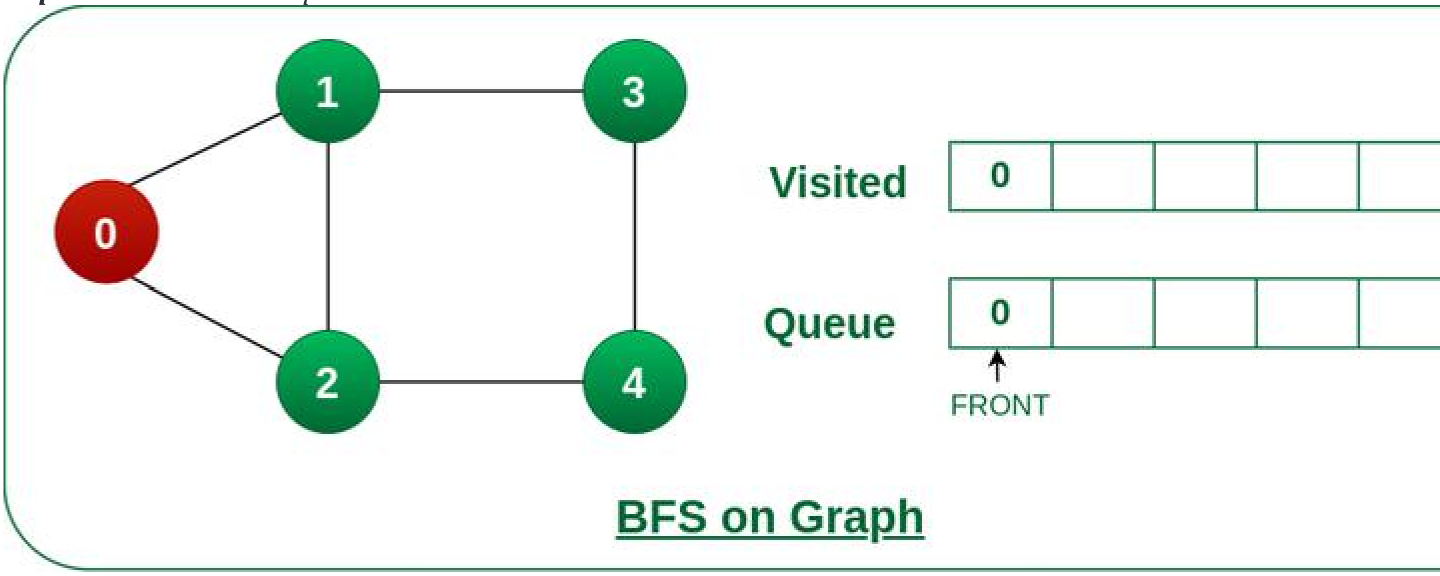
Let us understand the working of the algorithm with the help of the following example.

Step1: *Initially queue and visited arrays are empty.*



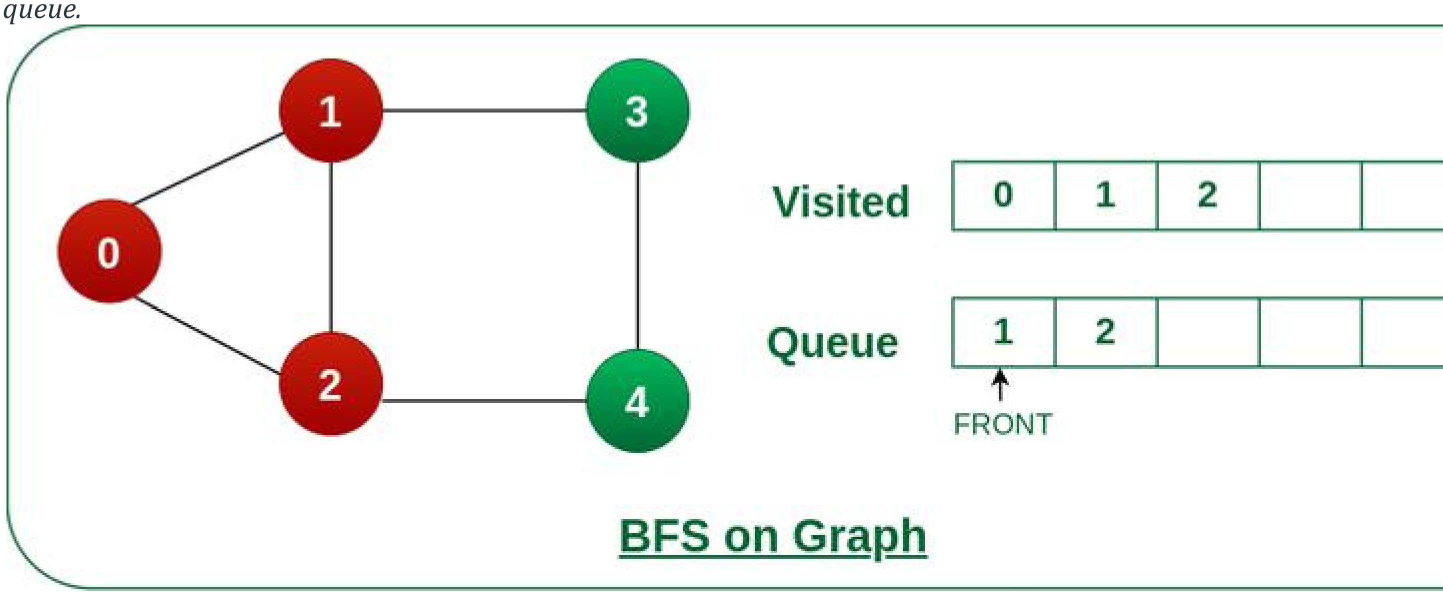
Queue and visited arrays are empty initially.

Step2: *Push node 0 into queue and mark it visited.*



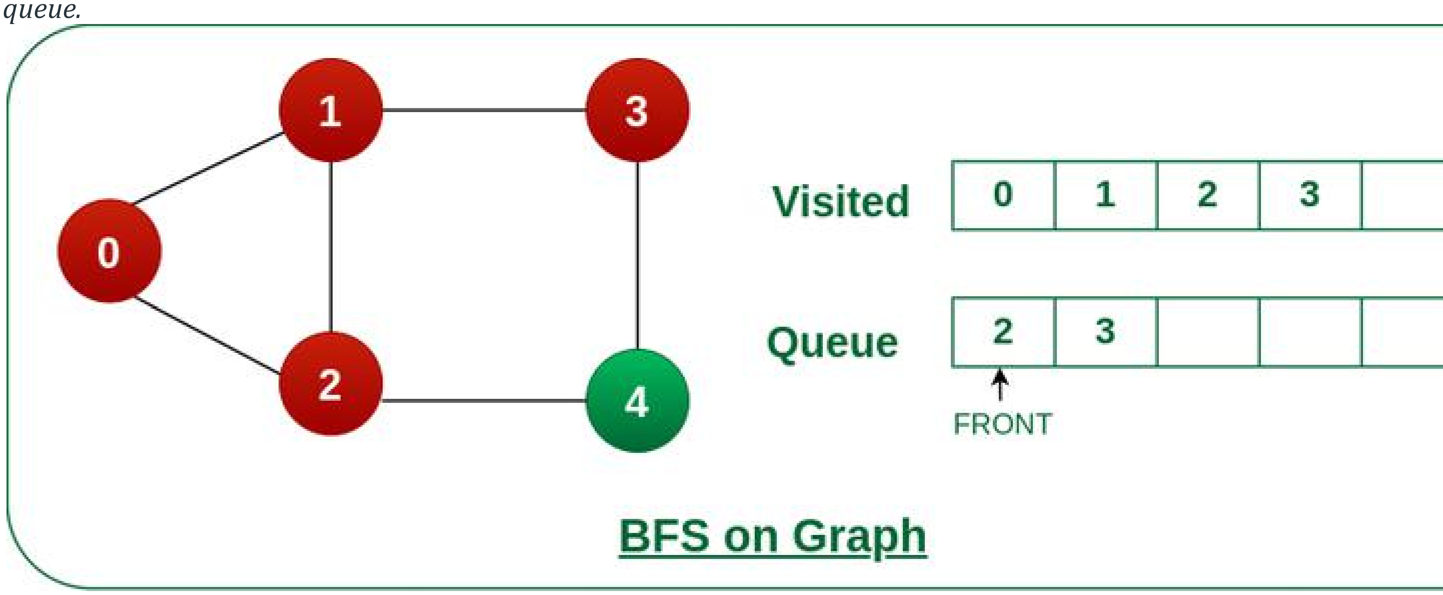
Push node 0 into queue and mark it visited.

Step 3: *Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.*



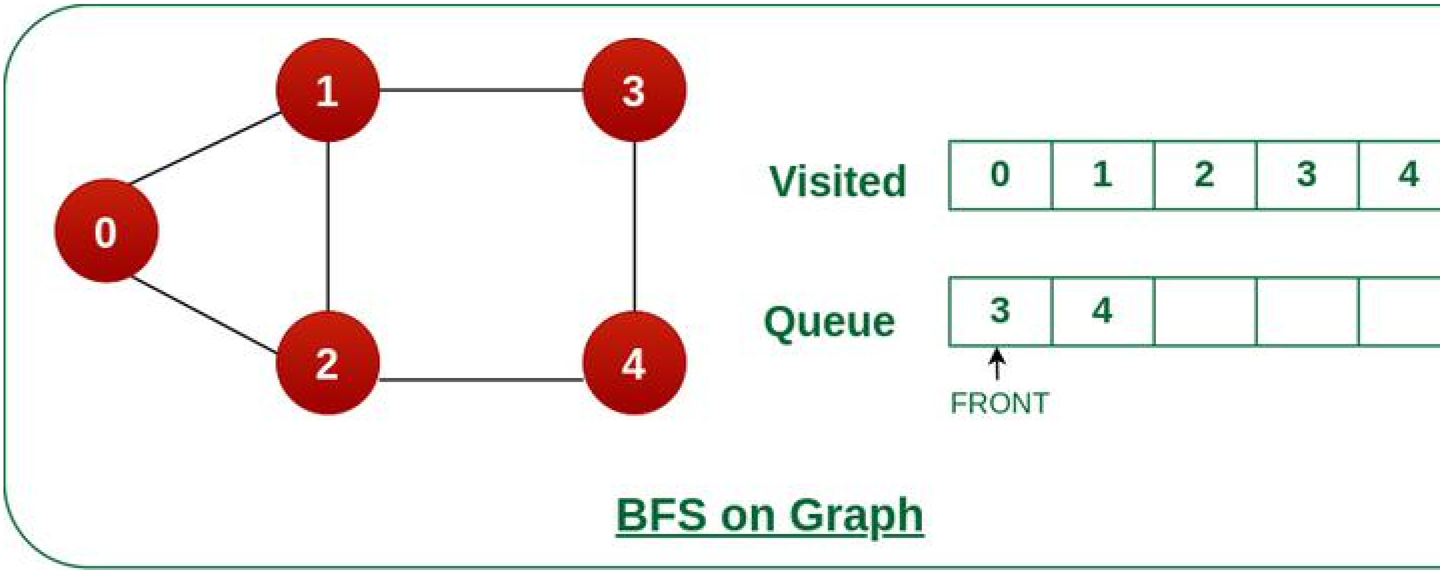
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: *Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*



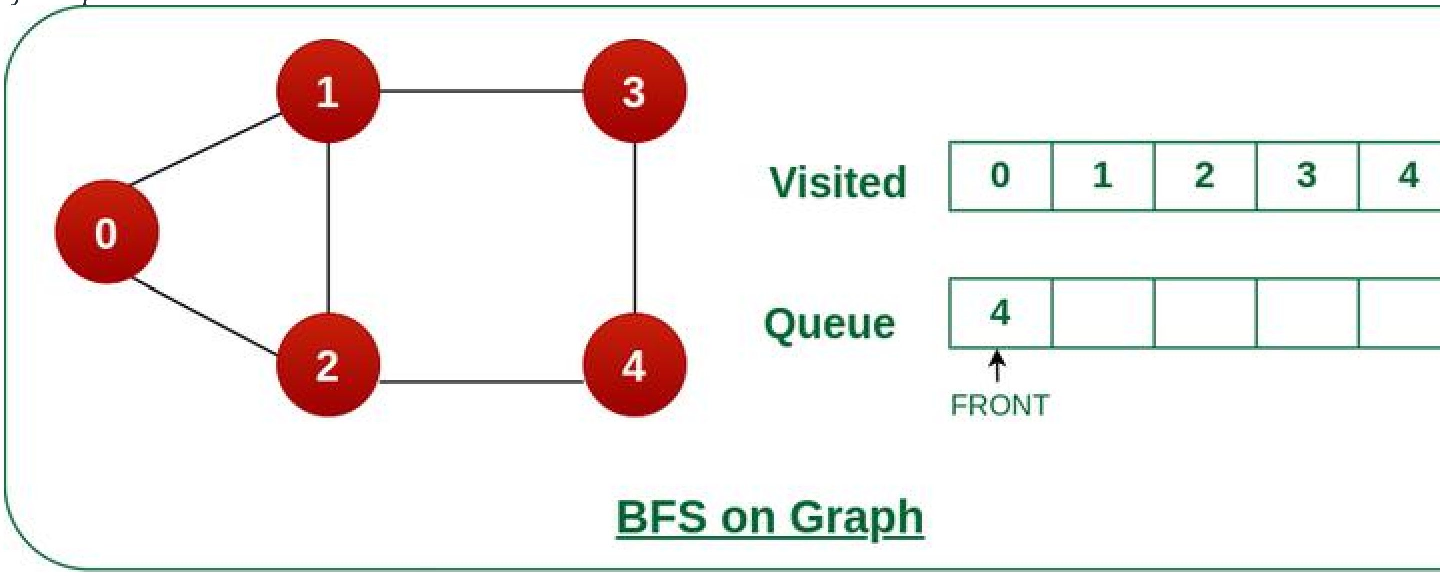
Remove node 1 from the front of queue and visited the unvisited neighbours and push

Step 5: *Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*



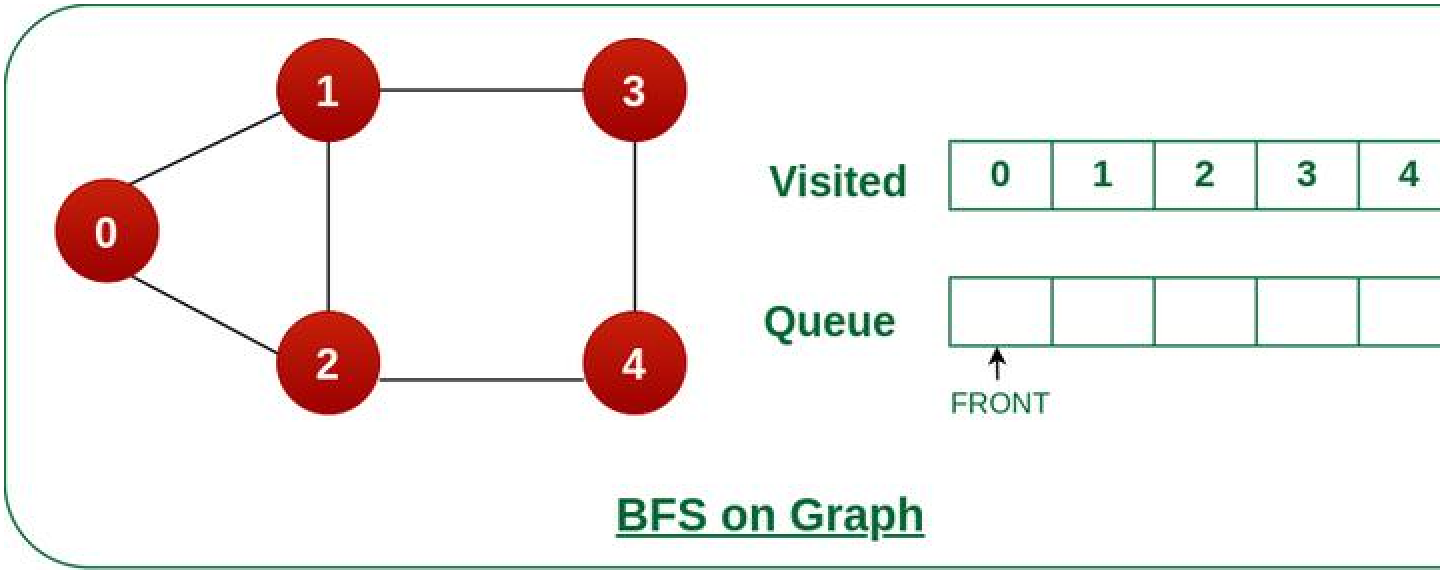
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

Step 6: *Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

Steps 7: *Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.*
As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Now, Queue becomes empty, So, terminate these process of iteration.

Implementation of BFS for Graph using Adjacency List:

- C
- C++
- Java
- Python3
- C#
- Javascript

```
# Python3 Program to print BFS traversal

# from a given source vertex. BFS(int s)

# traverses vertices reachable from s.



from collections import defaultdict
```

```
# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # Default dictionary to store graph
        self.graph = defaultdict(list)

    # Function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)
```

```
# Create a queue for BFS
queue = []

# Mark the source node as
# visited and enqueue it
queue.append(s)
visited[s] = True

while queue:

    # Dequeue a vertex from
    # queue and print it
    s = queue.pop(0)
    print(s, end=" ")

    # Get all adjacent vertices of the
    # dequeued vertex s.
    # If an adjacent has not been visited,
    # then mark it visited and enqueue it
    for i in self.graph[s]:
```

```
                if visited[i] == False:

                    queue.append(i)

                    visited[i] = True




# Driver code

if __name__ == '__main__':



    # Create a graph given in

    # the above diagram

    g = Graph()

    g.addEdge(0, 1)

    g.addEdge(0, 2)

    g.addEdge(1, 2)

    g.addEdge(2, 0)

    g.addEdge(2, 3)

    g.addEdge(3, 3)



    print("Following is Breadth First Traversal"

          " (starting from vertex 2)")
```

```
  g.BFS(2)



# This code is contributed by Neelam Yadav
```

Output

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

Time Complexity: O(V+E), where V is the number of nodes and E is the number of edges.

DFS (Depth First Search) algorithm

In this article, we will discuss the DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

The applications of using the DFS algorithm are given as follows -

- DFS algorithm can be used to implement the topological sorting.
- It can be used to find the paths between two vertices.
- It can also be used to detect cycles in the graph.
- DFS algorithm is also used for one solution puzzles.

- DFS is used to determine if a graph is bipartite or not.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

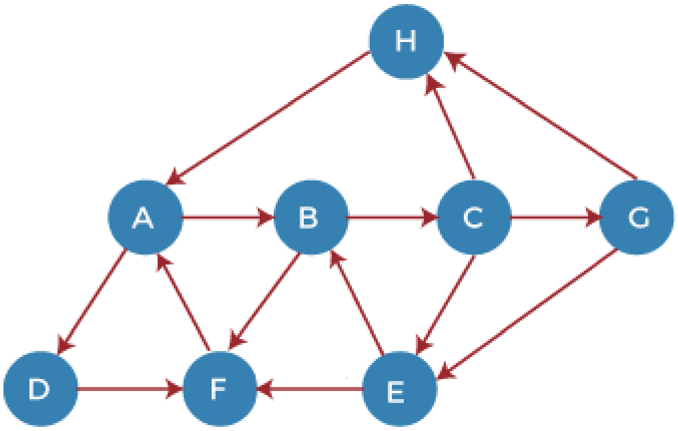
Step 6: EXIT

Pseudocode

```
DFS(G,v)   ( v is the vertex where the search starts )
    Stack S := {};   ( start with an empty stack )
    for each vertex u, set visited[u] := false;
    push S, v;
    while (S is not empty) do
      u := pop S;
      if (not visited[u]) then
        visited[u] := true;
        for each unvisited neighbour w of uu
          push S, w;
      end if
    end while
  END DFS()
```

Example of DFS algorithm

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

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

Now, let's start examining the graph starting from Node H.

Step 1 - First, push H onto the stack.

1. STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H]STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A
2. STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1. Print: D
2. STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

1. Print: G
2. STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Complexity of Depth-first search algorithm

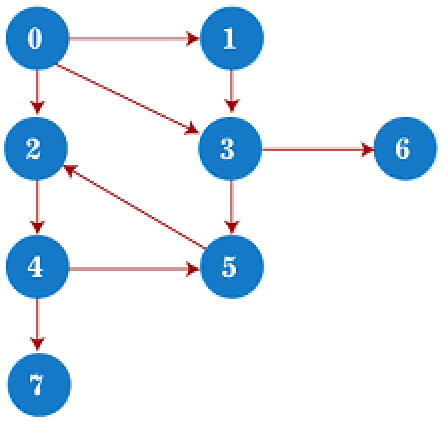
The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the DFS algorithm is O(V).

Implementation of DFS algorithm

Now, let's see the implementation of DFS algorithm in Java.

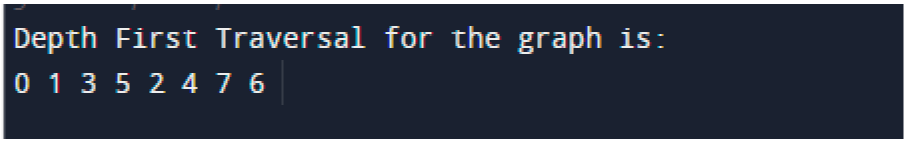
In this example, the graph that we are using to demonstrate the code is given as follows -



```
/*A sample java program to implement the DFS algorithm*/

import java.util.*;

class DFSTraversal {
  private LinkedList<Integer> adj[]; /*adjacency list representation*/
  private boolean visited[];

  /* Creation of the graph */
  DFSTraversal(int V) /*'V' is the number of vertices in the graph*/
  {
    adj = new LinkedList[V];
    visited = new boolean[V];

    for (int i = 0; i < V; i++)
      adj[i] = new LinkedList<Integer>();
  }

  /* Adding an edge to the graph */
  void insertEdge(int src, int dest) {
    adj[src].add(dest);
  }

  void DFS(int vertex) {
    visited[vertex] = true; /*Mark the current node as visited*/
    System.out.print(vertex + " ");

    Iterator<Integer> it = adj[vertex].listIterator();
    while (it.hasNext()) {
      int n = it.next();
```

```
   if (!visited[n])
    DFS(n);
  }
 }

 public static void main(String args[]) {
  DFSTraversal graph = new DFSTraversal(8);

    graph.insertEdge(0, 1);
    graph.insertEdge(0, 2);
    graph.insertEdge(0, 3);
    graph.insertEdge(1, 3);
    graph.insertEdge(2, 4);
    graph.insertEdge(3, 5);
    graph.insertEdge(3, 6);
    graph.insertEdge(4, 7);
    graph.insertEdge(4, 5);
    graph.insertEdge(5, 2);

    System.out.println("Depth First Traversal for the graph is:");
    graph.DFS(0);
 }
}
```

Output



```
Depth First Traversal for the graph is:
0 1 3 5 2 4 7 6
```

Conclusion

In this article, we have discussed the depth-first search technique, its example, complexity, and implementation in the java programming language. Along with that, we have also seen the applications of the depth-first search algorithm.

Spanning tree

In this article, we will discuss the spanning tree and the minimum spanning tree. But before moving directly towards the spanning tree, let's first see a brief description of the graph and its types.

Graph

A graph can be defined as a group of vertices and edges to connect these vertices. The types of graphs are given as follows -

- **Undirected graph:** An undirected graph is a graph in which all the edges do not point to any particular direction, i.e., they are not unidirectional; they are bidirectional. It can also be defined as a graph with a set of V vertices and a set of E edges, each edge connecting two different vertices.
- **Connected graph:** A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.
- **Directed graph:** Directed graphs are also known as digraphs. A graph is a directed graph (or digraph) if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

Now, let's move towards the topic spanning tree.

What is a spanning tree?

A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

A complete undirected graph can have $\mathbf{n^{n-2}}$ number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be $\mathbf{5^{5-2} = 125.}$

Applications of the spanning tree

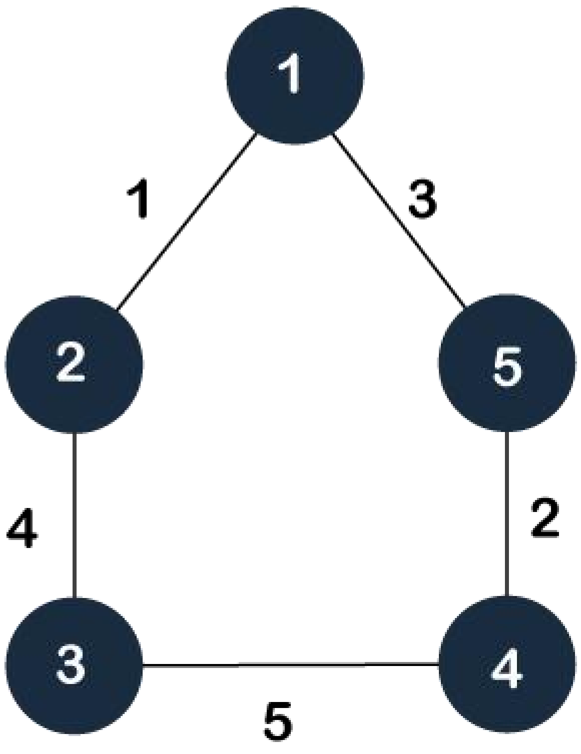
Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

- Cluster Analysis
- Civil network planning
- Computer network routing protocol

Now, let's understand the spanning tree with the help of an example.

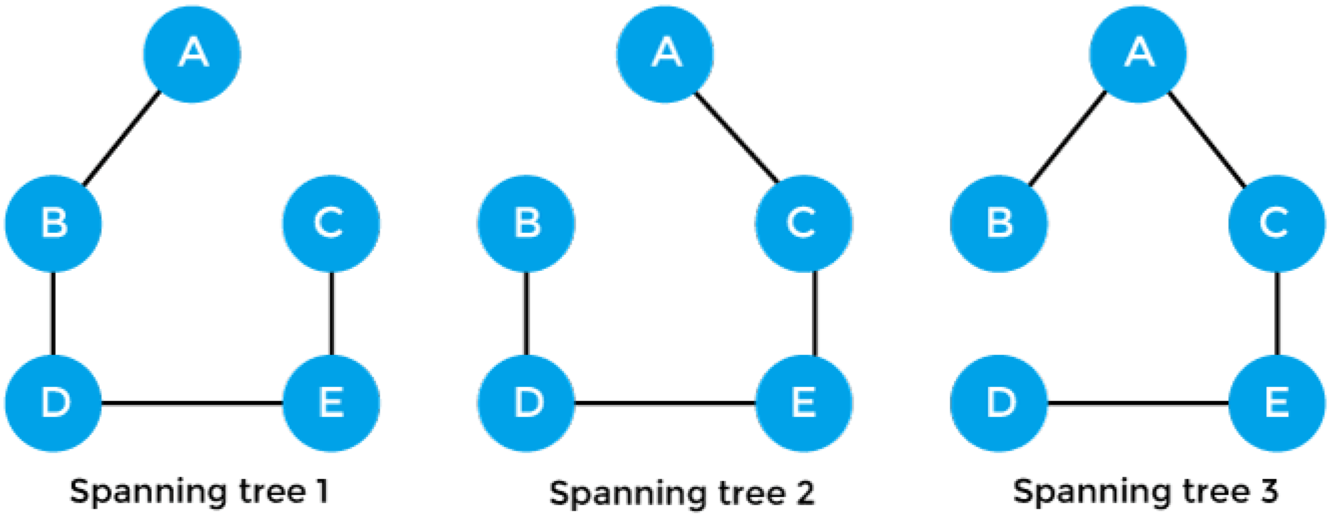
Example of Spanning tree

Suppose the graph be -



As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices. The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.

Some of the possible spanning trees that will be created from the above graph are given as follows -



Spanning tree 1 Spanning tree 2 Spanning tree 3

Properties of spanning-tree

Some of the properties of the spanning tree are given as follows -

- There can be more than one spanning tree of a connected graph G.
- A spanning tree does not have any cycles or loop.
- A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
- A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
- There can be a maximum $\mathbf{n^{n-2}}$ number of spanning trees that can be created from a complete graph.
- A spanning tree has **n-1** edges, where 'n' is the number of nodes.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

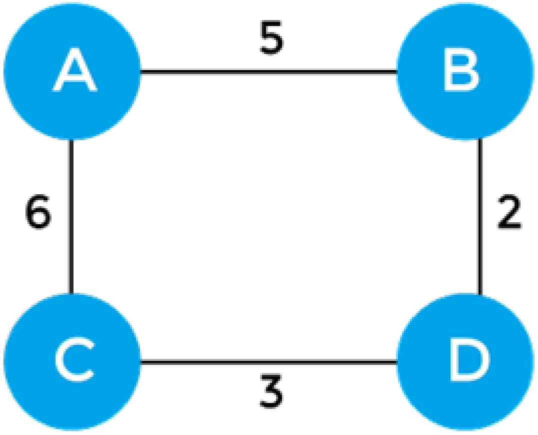
So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

Minimum Spanning tree

A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

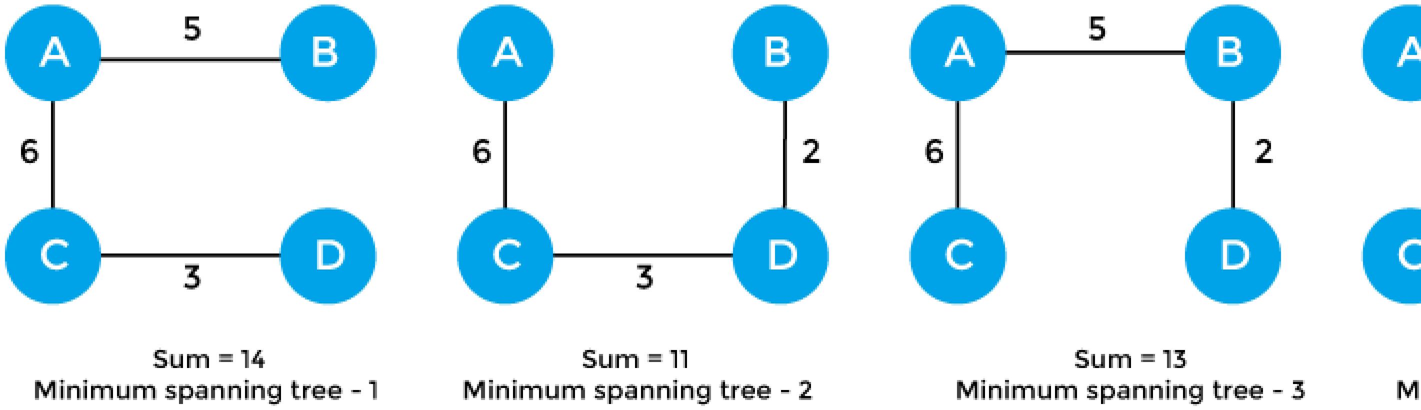
Example of minimum spanning tree

Let's understand the minimum spanning tree with the help of an example.

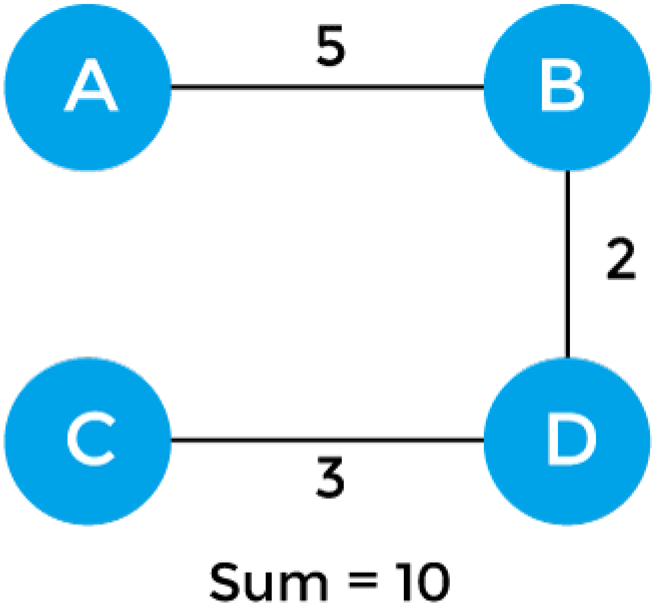


Weighted graph

The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -


A
B
C
D
5
6
3
Sum = 14
Minimum spanning tree - 1
A
B
C
D
6
2
3
Sum = 11
Minimum spanning tree - 2
A
B
C
D
5
6
2
Sum = 13
Minimum spanning tree - 3
A
C
M

So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



Applications of minimum spanning tree

The applications of the minimum spanning tree are given as follows -

- Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.
- It can be used to find paths in the map.

Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

- Prim's Algorithm
- Kruskal's Algorithm

Let's see a brief description of both of the algorithms listed above.

Prim's algorithm - It is a greedy algorithm that starts with an empty spanning tree. It is used to find the minimum spanning tree from the graph. This algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

To learn more about the prim's algorithm, you can click the below link - https://www.javatpoint.com/prim-algorithm

Kruskal's algorithm - This algorithm is also used to find the minimum spanning tree for a connected weighted graph. Kruskal's algorithm also follows greedy approach, which finds an optimum solution at every stage instead of focusing on a global optimum.

To learn more about the prim's algorithm, you can click the below link - https://www.javatpoint.com/kruskal-algorithm

So, that's all about the article. Hope the article will be helpful and informative to you. Here, we have discussed spanning tree and minimum spanning tree along with their properties, examples, and applications.