Course Objectives

1. To learn the features of C
2. To learn the linear and non-linear data structures
3. To explore the applications of linear and non-linear data structures
4. To learn to represent data using graph data structure
5. To learn the basic sorting and searching algorithms

Course Outcomes - Upon completion of the course, students will be able to:

1. Implement linear and non-linear data structure operations using C
2. Suggest appropriate linear / non-linear data structure for any given data set.
3. Apply hashing concepts for a given problem
4. Modify or suggest new data structure for an application
5. Appropriately choose the sorting algorithm for an application

UNIT I - C PROGRAMMING BASICS

Structure of a C program – compilation and linking processes – Constants, Variables – Data Types – Expressions using operators in C – Managing Input and Output operations – Decision Making and Branching – Looping statements. Arrays – Initialization – Declaration – One dimensional and Two-dimensional arrays. Strings- String operations – String Arrays. Simple programs- sorting- searching – matrix operations.

UNIT II - FUNCTIONS, POINTERS, STRUCTURES AND UNIONS

Functions – Pass by value – Pass by reference – Recursion – Pointers – Definition – Initialization – Pointers arithmetic. Structures and unions – definition – Structure within a structure – Union – Programs using structures and Unions – Storage classes, Pre-processor directives.

UNIT III - LINEAR DATA STRUCTURES

Arrays and its representations
Stacks and Queues – Applications
Linked lists – Single, circular and doubly Linked list-Application

UNIT IV - NON-LINEAR DATA STRUCTURES

Trees – Binary Trees – Binary tree representation and traversals , – Applications of trees.
Binary Search Trees , AVL trees.
Graph and its representations – Graph Traversals.

UNIT V - SEARCHING AND SORTING ALGORITHMS

Linear Search – Binary Search.
Sorting: Selection Sort, Bubble Sort, Insertion sort , Merge sort , Quick Sort
Hashing, Types of Hashing. Collision resolution techniques

Suggested Readings:

1. Brian W. Kernighan / Dennis Ritchie ,The C Programming Language ,Second Edition , Pearson 2015
2. Pradip Dey and Manas Ghosh, —Programming in C, Second Edition, Oxford University Press, 2011.
3. Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed, —Fundamentals of Data Structures in C, Second Edition, University Press, 2008.
4. Mark Allen Weiss, —Data Structures and Algorithm Analysis in C, Second Edition, Pearson Education, 1996
5. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, —Data Structures and Algorithms, Pearson Education, 1983.

SHAIK SHABANA ASSISTANT PROFESSOR MCA

MCA I YEAR / I SEMESTER

SUBJECT SYNOPSIS – DATA STRUCTURE USING C

UNIT-I

OVERVIEW OF C

HISTORY OF C

STRUCTURE OF C PROGRAM

CREATEING A STRUCTURE

COMPILATION AND LLINKING PROCESS

CONSTANT, VARIABLE

DATA TYPES

C EXPRESSION

MANAGING INPUT/ OUTPUT

DECISION MAKING AND BRANCHING

LOOPING STATEMENTS

ARRAY

C ARRAY DECLARISION

C ARRAY INTIALIZATION

TYPES OF ARRAY

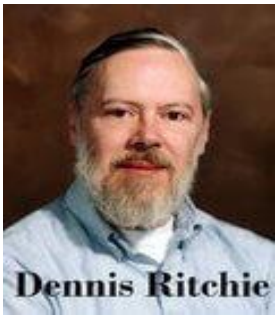
STRING, STRING ARRAY

UNIT - I

OVERVIEW OF C

C is a programming language. It is most popular computer language today because it is structured high level, machine independent language. Dennis Ritchie invented C language. Ken Thompson created a language which was based upon a language known as BCPL and it was called as B. B language was created in 1970, basically for Unix operating system Dennis Ritchie used ALGOL, BCPL and B as the basic reference language from which he created C.C has many qualities which any programmer may desire. It contains the capability of assembly language with the features of high level language which can be used for creating software packages, system software etc. It supports the programmer with a rich set of built-in functions and operators. C is highly portable. C programs written on one computer can run on other computer without making any changes in the program. Structured programming concept is well supported in C, this helps in dividing the programs into function modules or code blocks.

HISTORY OF C LANGUAGE



History of C language is interesting to know. Here we are going to discuss a brief history of the c language.

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.

It was developed to overcome the problems of previous languages such as B, BCPL, etc.

Initially, C language was developed to be used in UNIX operating system. It inherits many features of previous languages such as B and BCPL.

Let's see the programming languages that were developed before C language.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie

K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

STRUCTURE OF A C PROGRAM

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.

Unlike an array, a structure can contain many different data types (int, float, char, etc.).

CREATE A STRUCTURE

You can create a structure by using the struct keyword and declare each of its members inside curly braces:

```
struct MyStructure

{                       / Structure declaration
   int myNum;          // Member (int variable)
   char myLetter;      // Member (char variable)
};                     // End the structure with a semicolon
```

To access the structure, you must create a variable of it.

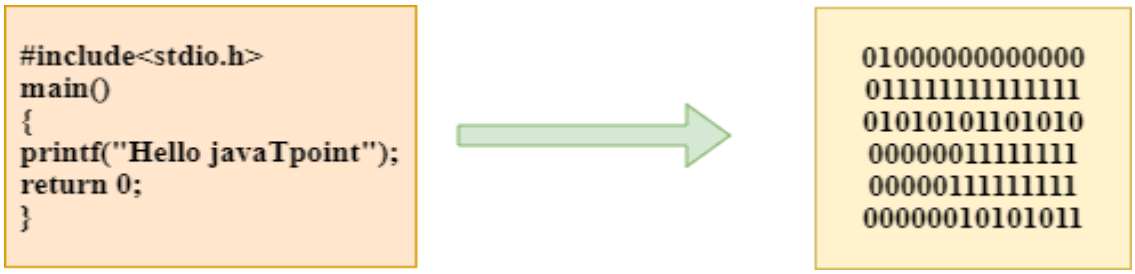
Use the struct keyword inside the main() method, followed by the name of the structure and then the name of the structure variable:

Create a struct variable with the name "s1":

```
struct myStructure {
    int myNum;
    char myLetter;
};
int main() {
  struct myStructure s1;
  return 0;
}
```

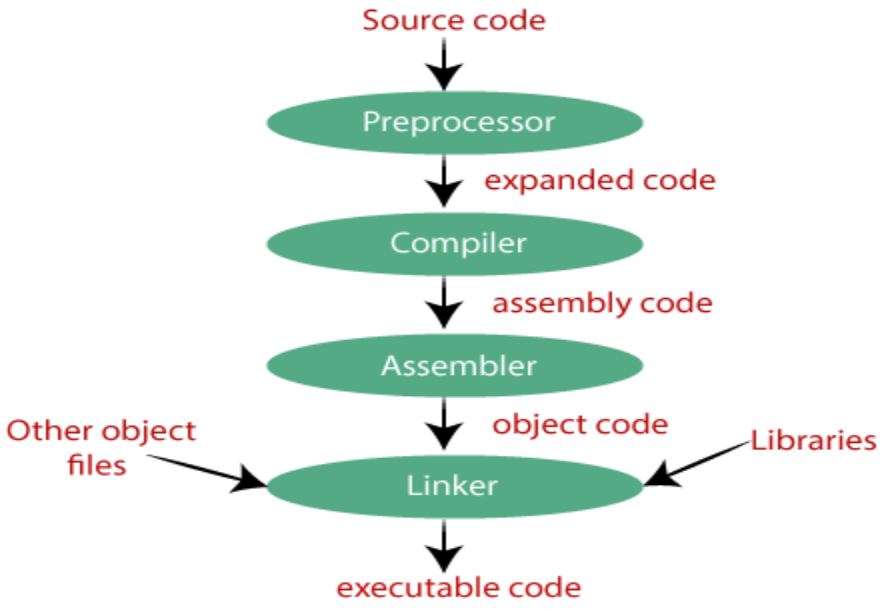
COMPILATION AND LINKING PROCESSES

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.



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.



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.

CONSTANTS

The constants in C are the read-only variables whose values cannot be modified once they are declared in the C program. The type of constant can be an integer constant, a floating pointer constant, a string constant, or a character constant. In C language, the **const** keyword is used to define the constants.

Syntax to Define Constant

const *data_type var_name = value*;

VARIABLES

A **variable in C language** is the name associated with some memory location to store data of different types. There are many types of variables in C depending on the scope, storage class, lifetime, type of data they store, etc. A variable is the basic building block of a C program that can be used in expressions as a substitute in place of the value it stores.

C VARIABLE SYNTAX

The syntax to declare a variable in C specifies the name and the type of the variable.

data_type variable_name = value; // defining single variable
or
data_type variable_name1, variable_name2; // defining multiple variable

DATA TYPES

Each variable in C has an associated data type. It specifies the type of data that the variable can store like integer, character, floating, double, etc. Each data type requires different amounts of memory and has some specific operations which can be performed over it. The data type is a collection of data with values having fixed values, meaning as well as its characteristics.

The data types in C can be classified as follows:

Types	Description
Primitive Data Types	Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters, etc.
User Defined Data Types	The user-defined data types are defined by the user himself.
Derived Types	The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.

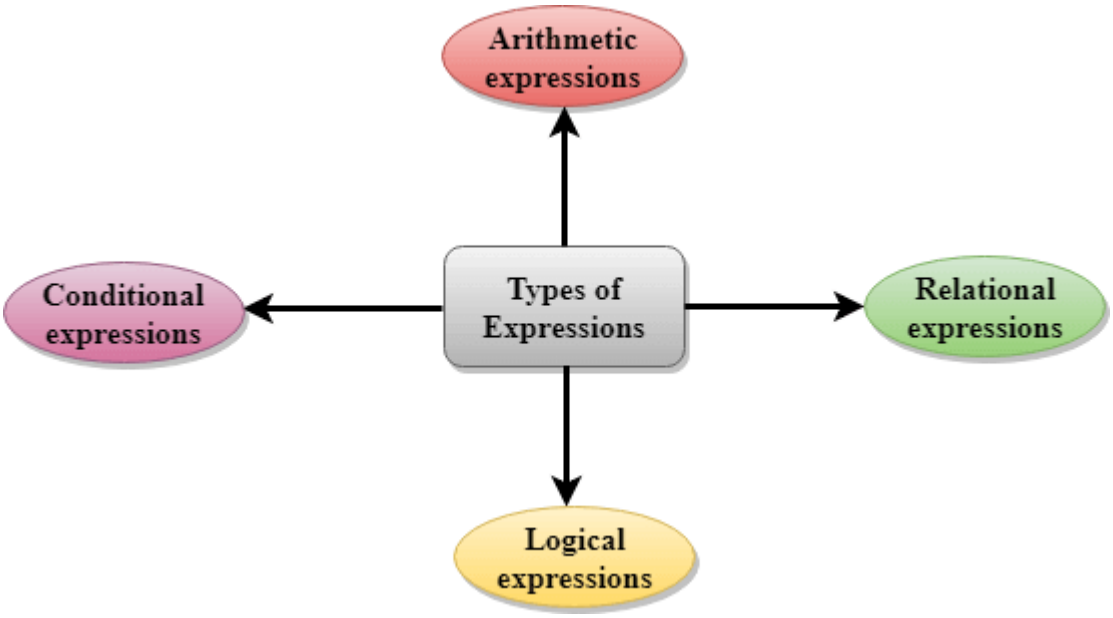
Data Type	Size (bytes)	Range	Format Specifier

Types	Description		
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
Data Type	**Size (bytes)**	**Range**	**Format Specifier**

Types		Description	
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	16	3.4E-4932 to 1.1E+4932	%Lf

C EXPRESSIONS

An expression is a formula in which operands are linked to each other by the use of operators to compute a value. An operand can be a function reference, a variable, an array element or a constant.



ARITHMETIC EXPRESSIONS

An arithmetic expression is an expression that consists of operands and arithmetic operators. An arithmetic expression computes a value of type int, float or double.

When an expression contains only integral operands, then it is known as pure integer expression when it contains only real operands, it is known as pure real expression, and when it contains both integral and real operands, it is known as mixed mode expression.

RELATIONAL EXPRESSIONS

A relational expression is an expression used to compare two operands.

It is a condition which is used to decide whether the action should be taken or not.

In relational expressions, a numeric value cannot be compared with the string value.

The result of the relational expression can be either zero or non-zero value. Here, the zero value is equivalent to a false and non-zero value is equivalent to true.

LOGICAL EXPRESSIONS

A logical expression is an expression that computes either a zero or non-zero value.

It is a complex test condition to take a decision.

CONDITIONAL EXPRESSIONS

A conditional expression is an expression that returns 1 if the condition is true otherwise 0.

A conditional operator is also known as a ternary operator.

MANAGING INPUT/OUTPUT

I/O operations are helpful for a program to interact with users. C stdlib is the standard C library for input-output operations. Two essential streams play their role when dealing with input-output operations in C. These are:

1. Standard Input (stdin)
2. Standard Output (stdout)

Standard input or stdin is used for taking input from devices such as the keyboard as a data stream.

Standard output or stdout is used to give output to a device such as a monitor. For I/O functionality, programmers must include the stdio header file within the program.

DECISION MAKING AND BRANCHING

The **conditional statements** (also known as decision control structures) such as if, if else, switch, etc. are used for decision-making purposes in C programs.
They are also known as Decision-Making Statements and are used to evaluate one or more conditions and make the decision whether to execute a set of statements or not. These decision-making statements in programming languages decide the direction of the flow of program execution.

Need of Conditional Statements

There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. For example, in C if x occurs then execute y else execute z. There can also be multiple conditions like in C if x occurs then execute p, else if condition y occurs execute q, else execute r. This condition of C else-if is one of the many ways of importing multiple conditions.

Following are the decision-making statements available in C

1. **if Statement**
2. **if-else Statement**
3. **Nested if Statement**
4. **if-else-if Ladder**
5. **switch Statement**
6. **Conditional Operator**
7. **Jump Statements:**
    - **break**
    - **continue**
    - **goto**
    - **return**

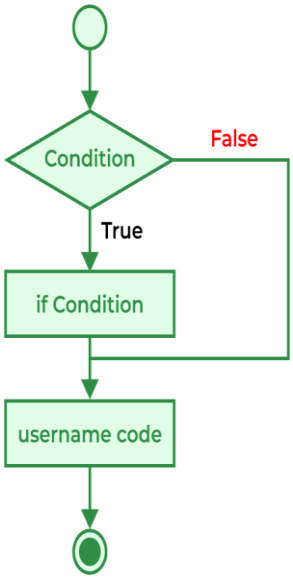
1. if

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

Syntax of if Statement

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

Flowchart of if Statement



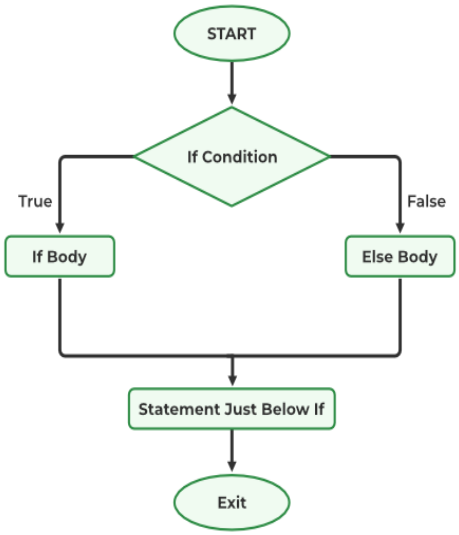
2. if-else

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

Syntax of if else

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

Flowchart of if-else Statement



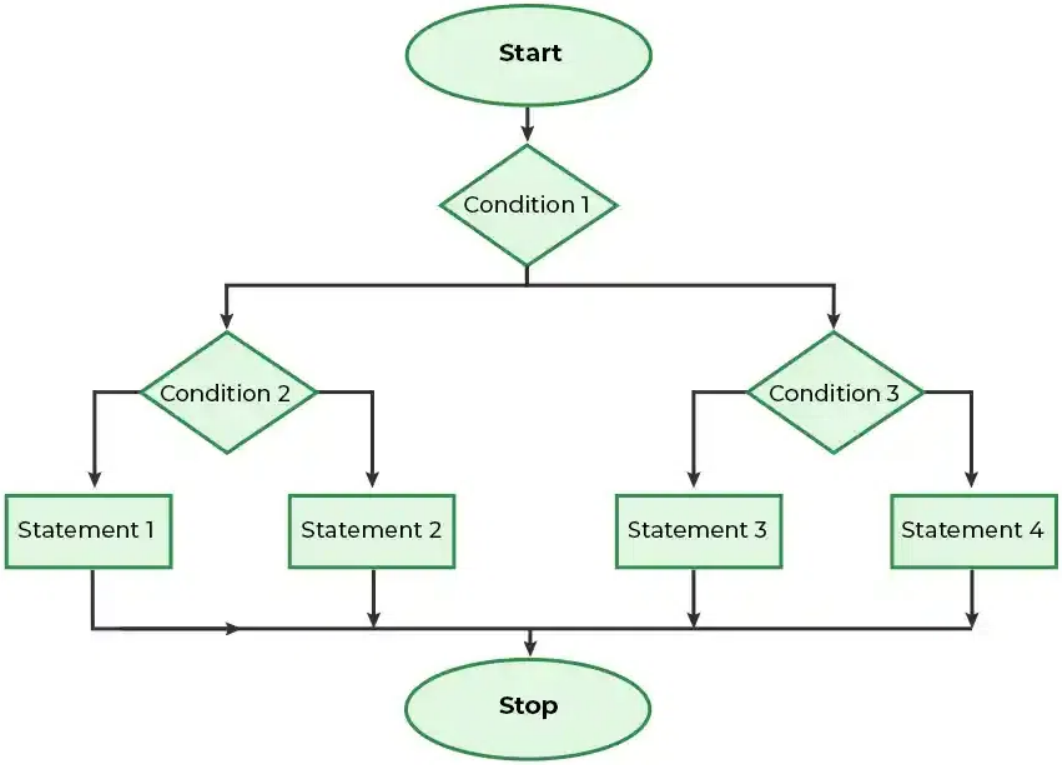
3. Nested if-else

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

Syntax of Nested if-else

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

Flowchart of Nested if-else



4. if-else-if Ladder in C/C++

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

Syntax of if-else-if Ladder

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

else
statement;

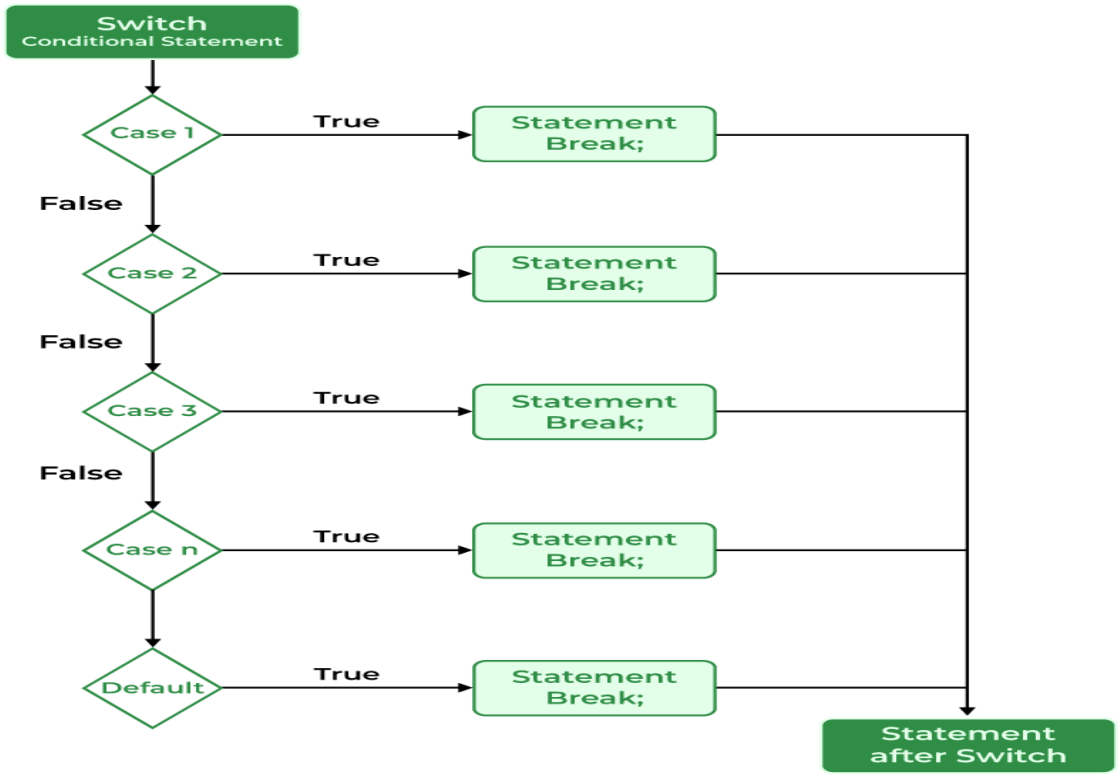
5. switch Statement in C/C++

The switch case statement is an alternative to the if else if ladder that can be used to execute the conditional code based on the value of the variable specified in the switch statement. The switch block consists of cases to be executed based on the value of the switch variable.

Syntax of switch

switch (*expression*) {
case*value1:*
statements;
case *value2:*
statements;
....
....
....
default:
statements;
}

Flowchart of switch



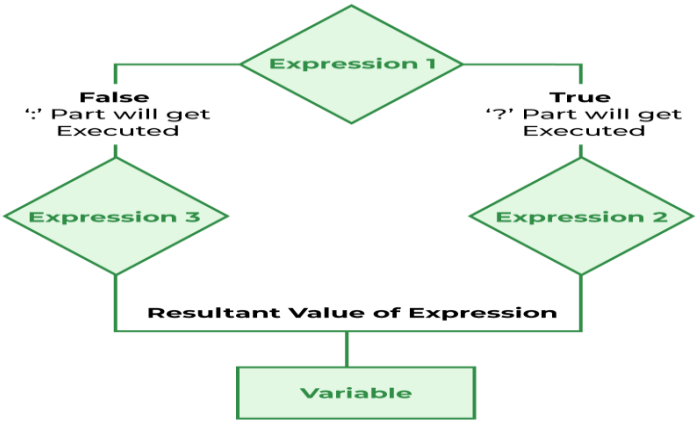
6. Conditional Operator in C

The conditional operator is used to add conditional code in our program. It is similar to the if-else statement. It is also known as the ternary operator as it works on three operands.

Syntax of Conditional Operator

(*condition*) **?***[true_statements]***:***[flase_statements]*;

Flowchart of Conditional Operator

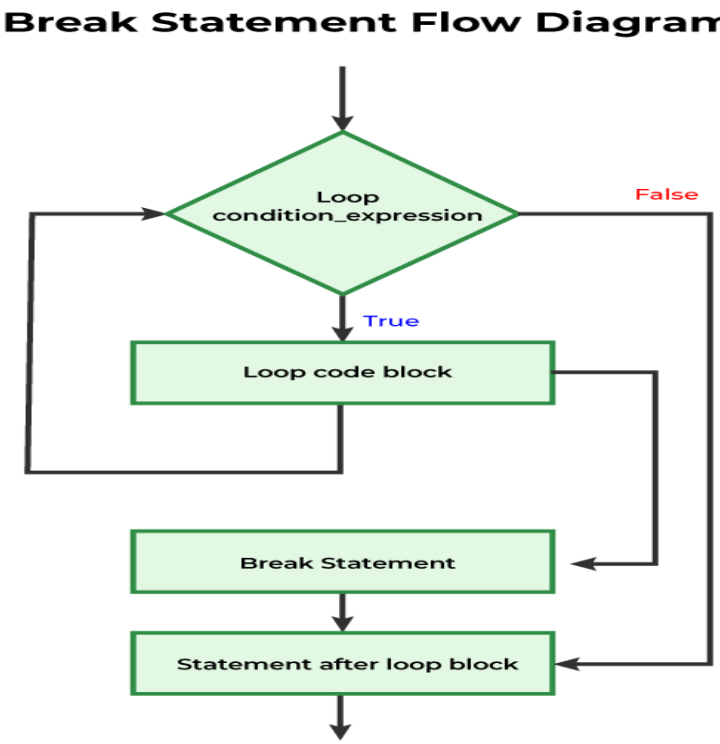


7. Jump Statements in C

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

A) break

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



B) continue

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

C) goto

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

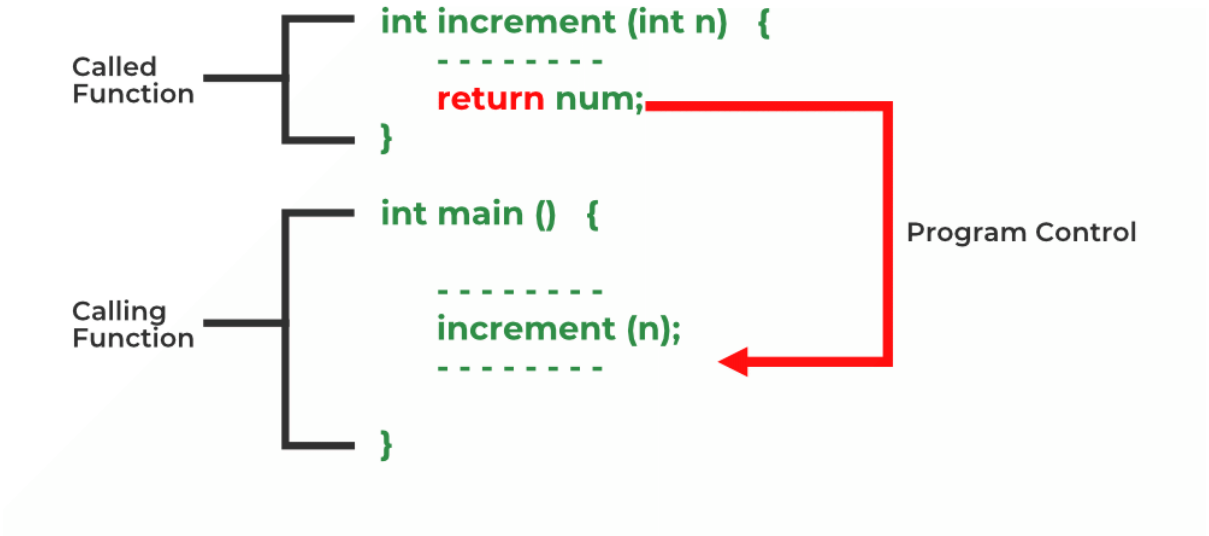
Syntax1	**Syntax2**
goto*label;*	*label:*
.	.
.	.
.	.

label: | **goto***label;*

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

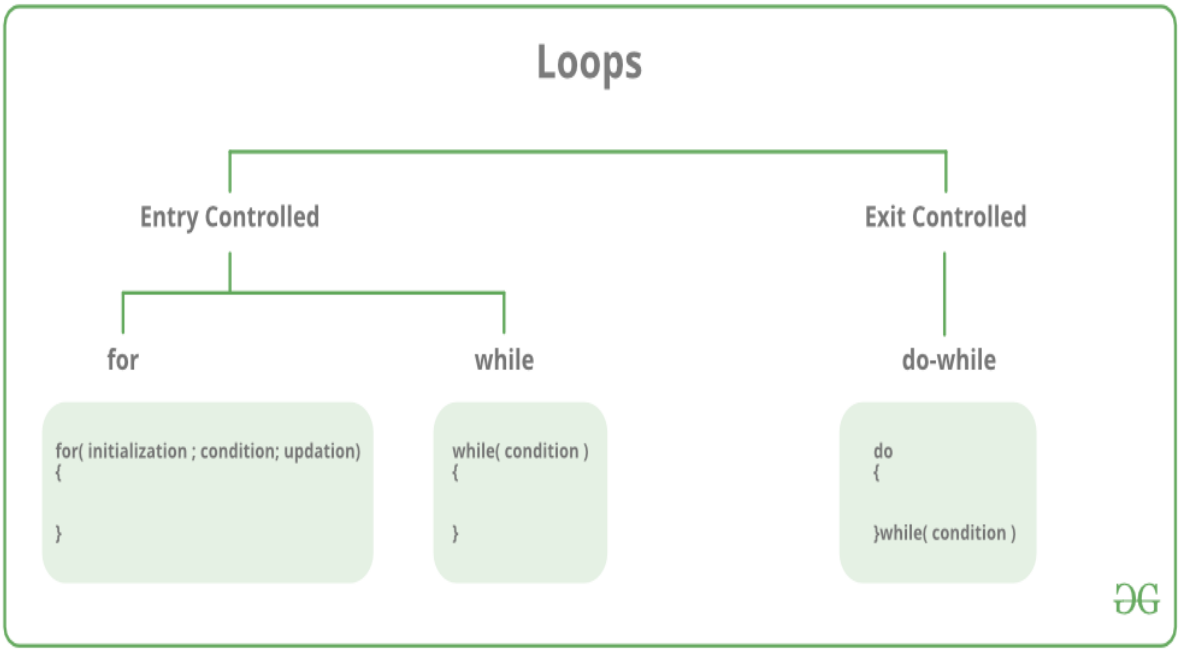
D) return

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



LOOPING STATEMENTS

Loops in programming are used to repeat a block of code until the specified condition is met. A loop statement allows programmers to execute a statement or group of statements multiple times without repetition of code.



Loop Type	Description
for loop	first Initializes, then condition check, then executes the body and at last, the update is done.
while loop	first Initializes, then condition checks, and then executes the body, and updating can be inside the body.
do-while loop	do-while first executes the body and then the condition check is done.

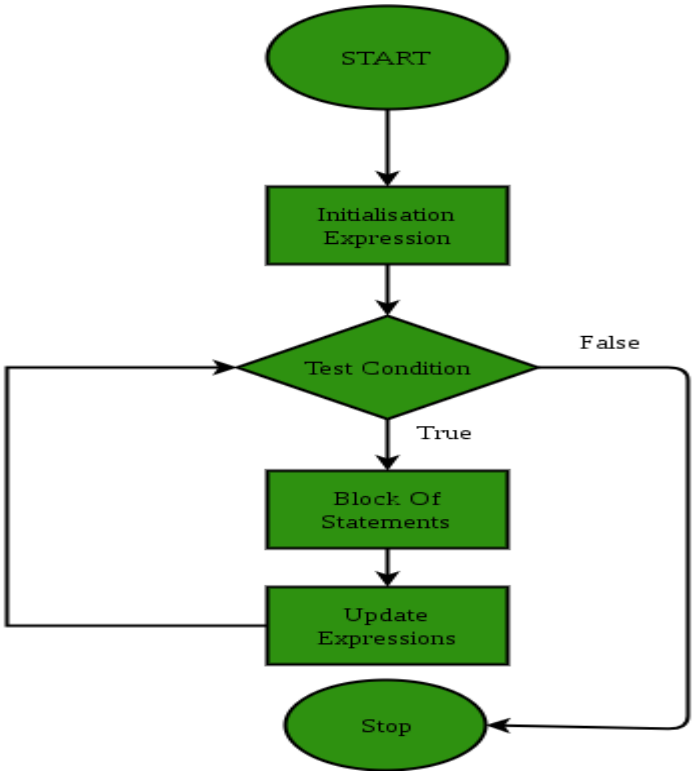
for Loop

for loop in C programming is a repetition control structure that allows programmers to write a loop that will be executed a specific number of times. for loop enables programmers to perform n number of steps together in a single line.

Syntax:

```
for (initialize expression; test expression; update expression)
{
   //
   // body of for loop
   //
}
```

Flow Diagram for loop:



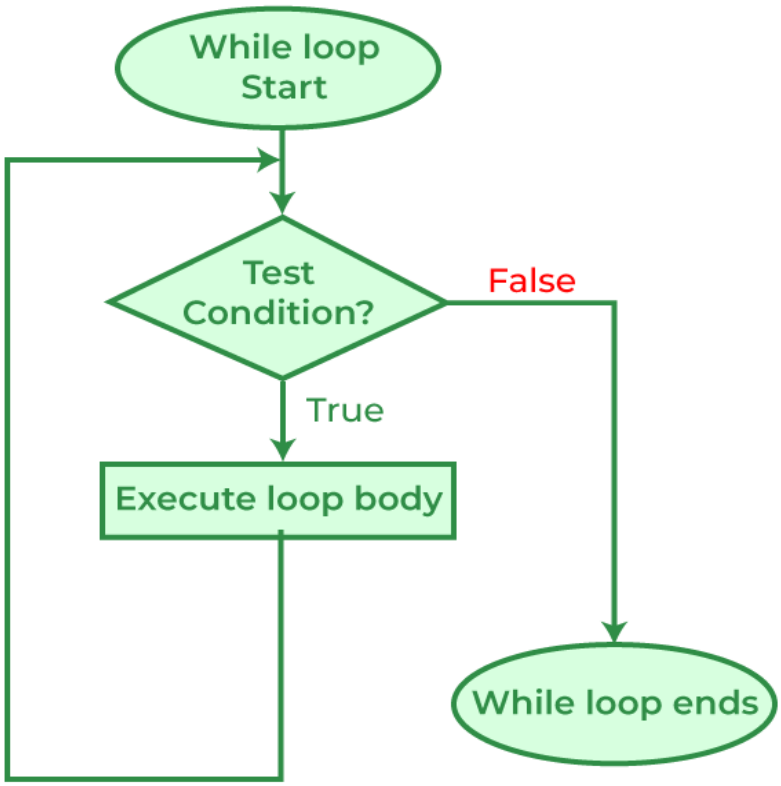
While Loop

While loop does not depend upon the number of iterations. In for loop the number of iterations was previously known to us but in the While loop, the execution is terminated on the basis of the test condition. If the test condition will become false then it will break from the while loop else body will be executed.

Syntax:

```
initialization_expression;


while (test_expression)
{
   // body of the while loop


   update_expression;
}
```

Flow Diagram for while loop:

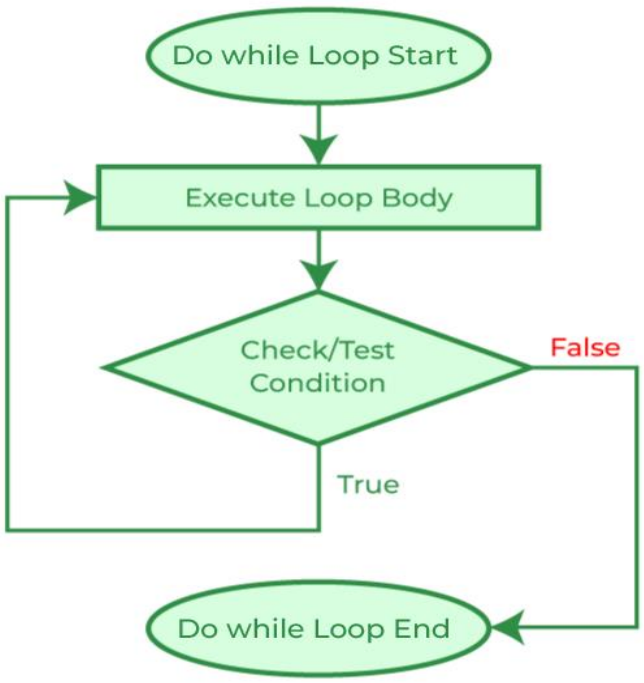


do-while Loop

The do-while loop is similar to a while loop but the only difference lies in the do-while loop test condition which is tested at the end of the body. In the do-while loop, the loop body will **execute at least once** irrespective of the test condition.

Syntax:

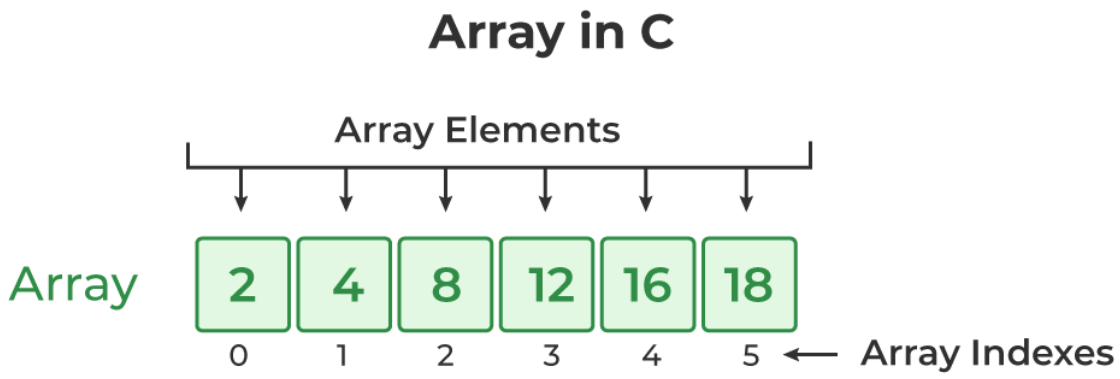
```
initialization_expression;

do

{

   // body of do-while loop



   update_expression;


} while (test_expression);
```



Array in C is one of the most used data structures in C programming. It is a simple and fast way of storing multiple values under a single name. In this article, we will study the different aspects of array in C language such as array declaration, definition, initialization, types of arrays, array syntax, advantages and disadvantages, and many more.

What is Array in C

An array in C is a fixed-size collection of similar data items stored in contiguous memory locations. It can be used to store the collection of primitive data types such as int, char, float, etc., and also derived and user-defined data types such as pointers, structures, etc.

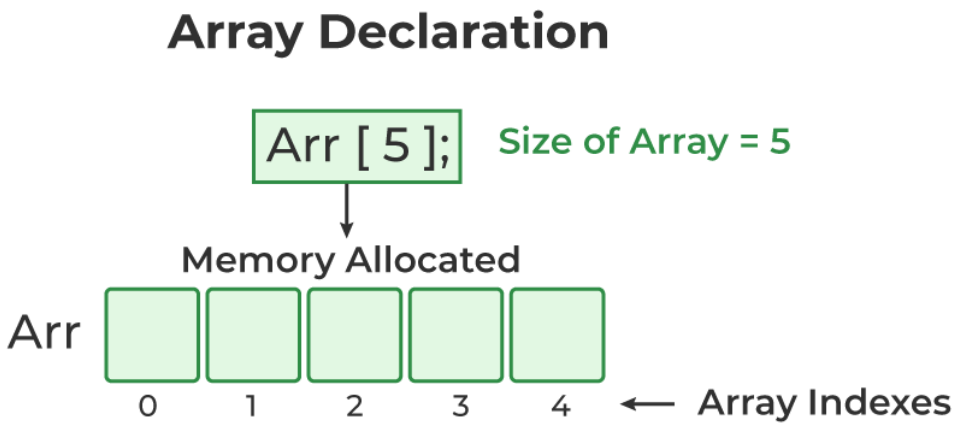


C Array Declaration

In C, we have to declare the array like any other variable before using it. We can declare an array by specifying its name, the type of its elements, and the size of its dimensions. When we declare an array in C, the compiler allocates the memory block of the specified size to the array name.

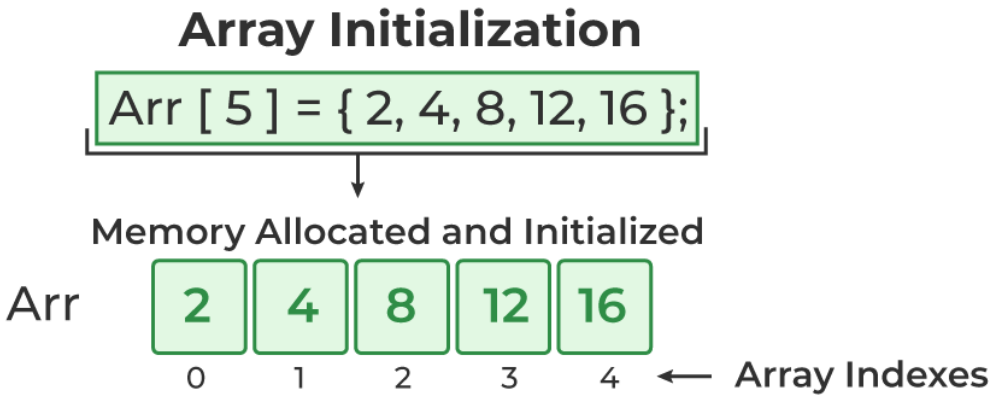
Syntax of Array Declaration

data_type array_name [size];
or
data_type array_name [*size1*] [*size2*]...[*sizeN*];



C Array Initialization

Initialization in C is the process to assign some initial value to the variable. When the array is declared or allocated memory, the elements of the array contain some garbage value. So, we need to initialize the array to some meaningful value. There are multiple ways in which we can initialize an array in C.



1. Array Initialization with Declaration

In this method, we initialize the array along with its declaration. We use an initializer list to initialize multiple elements of the array. An initializer list is the list of values enclosed within braces **{ }** separated b a comma.
data_type array_name [*size*] = {value1, value2, ... valueN};

Types of Array in C

There are two types of arrays based on the number of dimensions it has. They are as follows:

1. One Dimensional Arrays (1D Array)
2. Multidimensional Arrays

1. One Dimensional Array in C

The One-dimensional arrays, also known as 1-D arrays in C are those arrays that have only one dimension.

Syntax of 1D Array in C

array_name [size];

2. Multidimensional Array in C

Multi-dimensional Arrays in C are those arrays that have more than one dimension. Some of the popular multidimensional arrays are 2D arrays and 3D arrays. We can declare arrays with more dimensions than 3d arrays but they are avoided as they get very complex and occupy a large amount of space.

A. Two-Dimensional Array in C

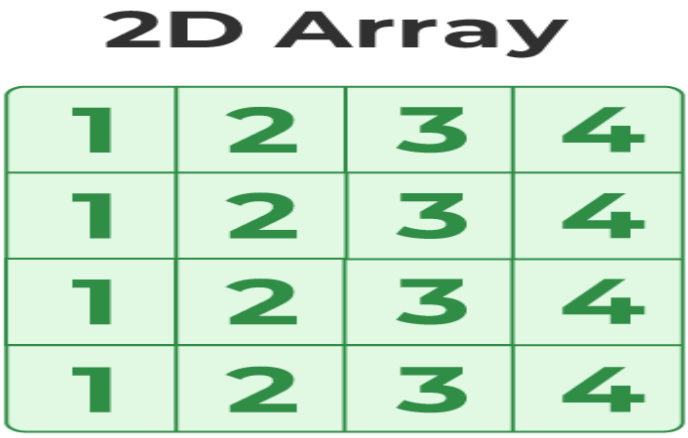
A Two-Dimensional array or 2D array in C is an array that has exactly two dimensions. They can be visualized in the form of rows and columns organized in a two-dimensional plane.

Syntax of 2D Array in C

array_name[size1] [size2];

Here,

- **size1:** Size of the first dimension.
- **size2:** Size of the second dimension.



STRINGS

A String in C programming is a sequence of characters terminated with a null character '\0'. The C String is stored as an array of characters. The difference between a character array and a C string is the string is terminated with a unique character '\0'.

C String Declaration Syntax

Declaring a string in C is as simple as declaring a one-dimensional array. Below is the basic syntax for declaring a string.

char *string_name*[*size*];

In the above syntax **str_name** is any name given to the string variable and size is used to define the length of the string, i.e the number of characters strings will store.
There is an extra terminating character which is the *Null character ('\0') used to indicate the termination of a string that differs strings from normal character arrays***.**

C String Initialization

A string in C can be initialized in different ways. We will explain this with the help of an example. Below are the examples to declare a string with the name str and initialize it with "abc".

Ways to Initialize a String in C

We can initialize a C string in 4 different ways which are as follows:

1. *Assigning a string literal without size*

String literals can be assigned without size. Here, the name of the string str acts as a pointer because it is an array.

```
char str[] = "abc";
```

2. *Assigning a string literal with a predefined size*

String literals can be assigned with a predefined size. But we should always account for one extra space which will be assigned to the null character. If we want to store a string of size n then we should always declare a string with a size equal to or greater than n+1.

```
char str[50] = "abc";
```

3. Assigning character by character with size

We can also assign a string character by character. But we should remember to set the end character as '\0' which is a null character.

```
char str[14] = { 'a','b','c','\0'};
```

4. Assigning character by character without size

We can assign character by character without size with the NULL character at the end. The size of the string is determined by the compiler automatically.

```
char str[] = { 'a','b','c','\0'};
```

Note: *When a Sequence of characters enclosed in the double quotation marks is encountered by the compiler, a null character '\0' is appended at the end of the string by default.*

Below is the memory representation of the string "abc".

Note: *After declaration, if we want to assign some other text to the string, we have to assign it one by one or use built-in strcpy() function because the direct assignment of string literal to character arrray is only possible in declration.*

Let us now look at a sample program to get a clear understanding of declaring, and initializing a string in C, and also how to print a string with its size.

C String Example C

```
// C program to illustrate strings
#include <stdio.h>
#include <string.h>
int main()
{
    // declare and initialize string
    charstr[] = "abc";

    // print string
    printf("%s\n", str);

    intlength = 0;
    length = strlen(str);

    // displaying the length of string
    printf("Length of string str is %d", length);

    return0;
}
```

Output

abc

Length of string str is 3

We can see in the above program that strings can be printed using normal printf statements just like we print any other variable. Unlike arrays, we do not need to print a string, character by character.

Note: *The C language does not provide an inbuilt data type for strings but it has an access specifier "**%s**" which can be used to print and read strings directly.*

Read a String Input From the User
The following example demonstrates how to take string input using scanf() function in C

```
// C program to read string from user
#include<stdio.h>
int main()
{
    // declaring string
    char str[50];

    // reading string
    scanf("%s",str);

    // print string
    printf("%s",str);

    return 0;
}
```

Input : abc
Output : abc

You can see in the above program that the string can also be read using a single scanf statement. Also, you might be thinking that why we have not used the '&' sign with the string name 'str' in scanf statement! To understand this you will have to recall your knowledge of scanf. We know that the '&' sign is used to provide the address of the variable to the scanf() function to store the value read in memory. As str[] is a character array so using str without braces '[' and ']' will give the base address of this string. That's why we have not used '&' in this case as we are already providing the base address of the string to scanf.

Now consider one more example,

```
// C Program to take input string which is separated by
// whitespaces
#include <stdio.h>

// driver code
int main()
{

    char str[20];
    // taking input string
    scanf("%s", str);

    // printing the read string
    printf("%s", str);

    return 0;
}
```

Input : abc
Output: abc

Here, the string is read-only till the whitespace is encountered. To read the string containing whitespace characters we can use the methods described below:

How to Read a String Separated by Whitespaces in C?
We can use multiple methods to read a string separated by spaces in C. The two of the common ones are:

1. We can use the fgets() function to read a line of string and gets() to read characters from the standard input (stdin) and store them as a C string until a newline character or the End-of-file (EOF) is reached.
2. We can also scanset characters inside the scanf() function

1. Example of String Input using gets()

```
// C program to illustrate
// fgets()
#include <stdio.h>
#define MAX 50
int main()
{
    char str[MAX];

    // MAX Size if 50 defined
    fgets(str, MAX, stdin);

    printf("String is: \n");

    // Displaying Strings using Puts
    puts(str);

    return 0;
}
```

Input: abc
Output
String is: **: abc**

2. Example of String Input using scanset

```
// C Program to take string separated by whitespace using
// scanset characters
#include <stdio.h>

// driver code
int main()
{

    char str[20];

    // using scanset in scanf
    scanf("%[^\n]s", str);

    // printing read string
    printf("%s", str);

    return 0;
}
```

Input: abc
Output: abc

C String Length
The length of the string is the number of characters present in the string except for the NULL character. We can easily find the length of the string using the loop to count the characters from the start till the NULL character is found.

Passing Strings to Function
As strings are character arrays, we can pass strings to functions in the same way we **pass an array to a function**. Below is a sample program to do this:

```
// C program to illustrate how to
// pass string to functions
#include <stdio.h>

void printStr(char str[]) { printf("String is : %s", str); }

int main()
{
    // declare and initialize string
    char str[] = ": abc";

    // print string by passing string
    // to a different function
    printStr(str);

    return 0;
}
```

Output:
String is : **: abc**

Note: We can't read a string value with spaces, we can use either gets() or fgets() in the C programming language.

Strings and Pointers in C

In Arrays, the variable name points to the address of the first element. Similar to arrays, In C, we can create a character pointer to a string that points to the starting address of the string which is the first character of the string. The string can be accessed with the help of pointers as shown in the below example.

```
// C program to print string using Pointers

#include <stdio.h>


int main()

{

    char str[20] = ": abc";
   // Pointer variable which stores

    // the starting address of

    // the character array str

    char* ptr = str;


    // While loop will run till

    // the character value is not

    // equal to null character

    while(*ptr != '\0') {

        printf("%c", *ptr);


        // moving pointer to the next character.

        ptr++;

    }

    return 0;

}
```

Output: abc

STANDARD C LIBRARY – STRING.H FUNCTIONS

The C language comes bundled with **<string.h>** which contains some useful string-handling functions. Some of them are as follows:

Function Name	Description
strlen(string_name)	Returns the length of string name.
strcpy(s1, s2)	Copies the contents of string s2 to string s1.
strcmp(str1, str2)	Compares the first string with the second string. If strings are the same it returns 0.
strcat(s1, s2)	Concat s1 string with s2 string and the result is stored in the first string.
strlwr()	Converts string to lowercase.
strupr()	Converts string to uppercase.
strstr(s1, s2)	Find the first occurrence of s2 in s1.

STRING ARRAYS

In C programming String is a 1-D array of characters and is defined as an array of characters. But an array of strings in C is a two-dimensional array of character types. Each String is terminated with a null character (\0). It is an application of a 2d array.

Syntax:
char variable_name[r] = {list of string};

Here,

- **var_name** is the name of the variable in C.
- **r** is the maximum number of string values that can be stored in a string array.
- **c** is the maximum number of character values that can be stored in each string array.

UNIT II

Functions

– Pass by value

– Pass by reference

– Recursion

Pointers

– Definition

–Initialization

– Pointers arithmetic.

Structures and unions

– definition

– Structure within structure

– Union

Storage classes

Pre-processor directives

FUNCTIONS

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Create a Function

To create (often referred to as *declare*) your own function, specify the name of the function, followed by parentheses () and curly brackets {}:

Syntax

```
void myFunction() {
  // code to be executed
}
```

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed when they are called.

To call a function, write the function's name followed by two parentheses () and a semicolon ;

In the following example, myFunction() is used to print a text (the action), when it is called:

Example

Inside main, call myFunction():

```
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

PASS BY VALUE

Pass by Value, means that a copy of the data is made and stored by way of the name of the parameter. Any changes to the parameter have NO affect on data in the calling function.

Pass by value is termed as the values which are sent as arguments in C programming language.

Algorithm

An algorithm is given below to explain the working of pass by value in C language.

START

Step 1: Declare a function that to be called.

Step 2: Declare variables.

Step 3: Enter two variables a,b at runtime.

Step 4: calling function jump to step 6.

Step 5: Print the result values a,b.

Step 6: Called function swap.

PASS BY REFERENCE

Passing by by reference refers to a method of passing the address of an argument in the calling function to a corresponding parameter in the called function. In C, the corresponding parameter in the called function must be declared as a pointer type.

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.
To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

```
/* function definition to swap the values */
void swap(int *x, int *y) {

   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */

   return;
}
```

RECURSION

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {
  recursion(); /* function calls itself */
}

int main() {
  recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Number Factorial

The following example calculates the factorial of a given number using a recursive function −

```
#include <stdio.h>

unsigned long long int factorial(unsigned int i) {

  if(i <= 1) {
    return 1;
  }
  return i * factorial(i - 1);
}

int  main() {
  int i = 12;
  printf("Factorial of %d is %d\n", i, factorial(i));
  return 0;
}
```

POINTERS

A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.
As the pointers store the memory addresses, their size is independent of the type of data they are pointing to. This size of pointers only depends on the system architecture.

Syntax
The syntax of pointers is similar to the variable declaration in C, but we use the **(*) dereferencing operator** in the pointer declaration.
datatype ***ptr***;

where

- **ptr** is the name of the pointer.
- **datatype** is the type of data it is pointing to.

Types of Pointer

Pointers can be classified into many different types based on the parameter on which we are defining their types. If we consider the type of variable stored in the memory location pointed by the pointer, then the pointers can be classified into the following types:

1. Integer Pointers

As the name suggests, these are the pointers that point to the integer values.
Syntax
int *ptr;

These pointers are pronounced as **Pointer to Integer.**
Similarly, a pointer can point to any primitive data type. It can point also point to derived data types such as arrays and user-defined data types such as structures.

2. Array Pointer

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as Pointer to Arrays. We can create a pointer to an array using the given syntax.
Syntax
char *ptr = &*array_name*;

Pointer to Arrays exhibits some interesting properties which we discussed later in this article.

3. Structure Pointer

The pointer pointing to the structure type is called Structure Pointer or Pointer to Structure. It can be declared in the same way as we declare the other primitive data types.
Syntax
struct *struct_name* *ptr;

In C, structure pointers are used in data structures such as linked lists, trees, etc.

4. Function Pointers

Function pointers point to the functions. They are different from the rest of the pointers in the sense that instead of pointing to the data, they point to the code. Let's consider a function prototype – **int func (int, char)**, the function pointer for this function will be
Syntax
int (*ptr)(int, char);

Note: The syntax of the function pointers changes according to the function prototype.

5. Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or pointers-to-pointer. Instead of pointing to a data value, they point to another pointer.
Syntax
datatype ** pointer_name;

6. NULL Pointer

The Null Pointers are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.
Syntax
data_type **pointer_name* = NULL;
or
pointer_name = NULL

It is said to be good practice to assign NULL to the pointers currently not in use.

7. Void Pointer

The Void pointers in C are the pointers of type void. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

Syntax

void * *pointer_name*;

One of the main properties of void pointers is that they cannot be dereferenced.

8. Wild Pointers

The Wild Pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash.

Example

```
int *ptr;
char *str;
```

9. Constant Pointers

In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

Syntax

data_type * const *pointer_name*;

10. Pointer to Constant

The pointers pointing to a constant value that cannot be modified are called pointers to a constant. Here we can only access the data pointed by the pointer, but cannot modify it. Although, we can change the address stored in the pointer to constant.

Syntax

const *data_type* * *pointer_name*;

Size of Pointers in C

The size of the pointers in C is equal for every pointer type. The size of the pointer does not depend on the type it is pointing to. It only depends on the operating system and CPU architecture. The size of pointers in C is

- **8 bytes** for a **64-bit System**
- **4 bytes** for a **32-bit System**

Pointer Arithmetic

The Pointer Arithmetic refers to the legal or valid arithmetic operations that can be performed on a pointer. It is slightly different from the ones that we generally use for mathematical calculations as only a limited set of operations can be performed on pointers. These operations include:

- Increment in a Pointer
- Decrement in a Pointer
- Addition of integer to a pointer
- Subtraction of integer to a pointer
- Subtracting two pointers of the same type
- Comparison of pointers of the same type.
- Assignment of pointers of the same type.

```
// C program to illustrate Pointer Arithmetic

#include <stdio.h>

int main()
{

    // Declare an array
    int v[3] = { 10, 100, 200 };

    // Declare pointer variable
    int* ptr;

    // Assign the address of v[0] to ptr
    ptr = v;

    for (int i = 0; i < 3; i++) {
```

```
        // print value at address which is stored in ptr
        printf("Value of *ptr = %d\n", *ptr);

        // print value of ptr
        printf("Value of ptr = %p\n\n", ptr);

        // Increment pointer ptr by 1
        ptr++;
    }
    return 0;
}
```

Output

```
Value of *ptr = 10
Value of ptr = 0x7ffe8ba7ec50

Value of *ptr = 100
Value of ptr = 0x7ffe8ba7ec54

Value of *ptr = 200
Value of ptr = 0x7ffe8ba7ec58
```

ADVANTAGES OF POINTERS

Following are the major advantages of pointers in C:

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

DISADVANTAGES OF POINTERS

Pointers are vulnerable to errors and have following disadvantages:

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are a little bit complex to understand.
- Pointers are majorly responsible for memory leaks in C.
- Pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a segmentation fault.

STRUCTURES AND UNIONS

In C programming, a **Structure** is a userdefined datatype. It is basically used to combine different datatypes into a single datatype. A structure in a C program can contain multiple members and structure variables. In order to define structures, the 'struct' keyword is used. To access the members of a structure, we use the dot (.) operator.

Syntax

The syntax of structures in C language is,

```
struct structure_name {
   member definition;
} structure_variables;
```

Where,

- **structure_name** is the name given to the structure.
- **member definition** is the set of member variables.
- **structure_variable** is the object of structure.

Example

```
struct Data {
  int a;
  long int b;
} data, data1;
```

In C programming, a **union** is also a user-defined datatype. All the members of a union share the same memory location. Therefore, if we need to use the same memory location for two or more members, then union is the best data type for that. The largest union member defines the size of the union.

In C programming, unions are similar to structures, as the union variables are also created in the same manner as the structure variables. To define a union in a C program, the keyword "union" is used.

Syntax

The syntax of unions in C language is,

```
union union_name {
  member definition;
} union_variables;
```

Where,

- **union_name** is any name given to the union.
- **member definition** is the set of member variables.
- **union_variable** is the object of union.

Example

```
union Data {
  int i;
  float f;
} data, data1;
```

Difference between Structure and Union

Key	Structure	Union
Definition	A Structure is a container defined in C to store data variables of different type and also supports for the userdefined variables storage.	A Union is also a similar kind of container in C which can hold different types of variables along with the userdefined variables.
Internal implementation	Structures in C are internally implemented. There is separate memory location allotted for each input member.	While in case Union memory is allocated only to one member having largest size among all other input variables and the same location is being get shared among all of these.
Syntax	Syntax of declare a Structure in C is as follows − struct struct_name{ type element1; type element2; . . } variable1, variable2, ...;	The syntax of declare a Union in C is as follows union u_name{ type element1; type element2; . . } variable1, variable2, ...;
Size	Structures do not have shared location for their members, so the size of a Structure is equal or greater than the sum of size of all the data members.	Unions do not have separate locations for each of their members, so their size or equal to the size of largest member among all data members.
Value storage	In case of a Structure, there is specific memory location for each input data member and hence it can store multiple values of the different members.	In a Union, there is only one shared memory allocation for all input data members so it stores a single value at a time for all members
Initialization	In a Structure, multiple members can be can be initializing at same time.	In a Union, only the first member can get initialize at a time.

Structure within structure (or) Nested structures

A structure inside another structure is called nested structure.

Consider the following example,

```
struct emp{
   int eno;
   char ename[30];
```

```
  float sal;
  float da;
  float hra;
  float ea;
}e;
```

All the items comes under allowances can be grouped together and declared under a sub – structure as shown below.

```
stuct emp{
  int eno;
  char ename[30];
  float sal;
  struct allowance{
    float da;
    float hra;
    float ea;
  }a;
}e;
```

The inner most member in a nested structure can be accessed by changing all the concerned structure variables (from outer most to inner most) with the member using dot operator.

Following program is to demonstrate nested structure (structure within the structure)

```
#include<stdio.h>
//Declaring outer and inter structures//
struct Person//Main Structure//{
  char Name[500];
  int Age;
  char Gender;
  char temp;//To clear buffer//
  struct Address//Nested Structure//{
    char Apartment[500];
```

```
    char Street[500];
    char City[100];
    char State[100];
    int Zipcode;
  }a[20];//Nested Structure Variable//
  }p[20];//Main Structure Variable//
void main(){
  //Declaring variable for For loop//
  int i;
  //Reading User I/p//
  for (i=1;i<3;i++){//Declaring function to accept 2 people's data//
    printf("Enter the Name of person %d : ",i);
    gets(p[i].Name);
    printf("Enter the Age of person %d : ",i);
    scanf("%d",&p[i].Age);
    scanf("%c",&p[i].temp);//Clearing Buffer//
    printf("Enter the Gender of person %d : ",i);
    scanf("%c",&p[i].Gender);
    scanf("%c",&p[i].temp);//Clearing Buffer//
    printf("Enter the City of person %d : ",i);
    gets(p[i].a[i].City);
    printf("Enter the State of person %d : ",i);
    gets(p[i].a[i].State);
    printf("Enter the Zip Code of person %d : ",i);
    scanf("%d",&p[i].a[i].Zipcode);
    scanf("%c",&p[i].temp);//Clearing Buffer//
  }
  //Printing O/p//
```

```
  for (i=1;i<3;i++){
    printf("The Name of person %d is : %s
",i,p[i].Name);
    printf("The Age of person %d is : %d
",i,p[i].Age);
    printf("The Gender of person %d is : %c
",i,p[i].Gender);
    printf("The City of person %d is : %s
",i,p[i].a[i].City);
    printf("The State of person %d is : %s
",i,p[i].a[i].State);
    printf("The Zip code of person %d is : %d
",i,p[i].a[i].Zipcode);
  }
}
```

Output

Enter the Name of person 1 : Enter the Age of person 1 : Enter the Gender of person 1 : Enter the City of person 1 : Enter the State of person 1 : Enter the Zip Code of person 1 : Enter the Name of person 2 : Enter the Age of person 2 : Enter the Gender of person 2 : Enter the City of person 2 : Enter the State of person 2 : Enter the Zip Code of person 2 : The Name of person 1 is :

The Age of person 1 is : 0

The Gender of person 1 is :

The City of person 1 is :

The State of person 1 is :

The Zip code of person 1 is : 0

The Name of person 2 is :

The Age of person 2 is : 0

The Gender of person 2 is :

The City of person 2 is :

The State of person 2 is :

The Zip code of person 2 is : 0

UNION

The Union is a user-defined data type in C language that can contain elements of the different data types just like structure. But unlike structures, all the members in the C union are stored in the same memory location. Due to this, only one member can store data at the given instance.

Syntax:

Union unionname

{

Char x:

Float y:

}obj:

STORAGE CLASSES

These features basically include the scope, visibility, and lifetime which help us to trace the existence of a particular variable during the runtime of a program.

Storage specifier	Storage	Initial Value	Scope	Life
Auto	Stack	Garbage	Within block	End of Block
Extern	Data segment	Zero	Global Multiple Files	Till end of program
Static	Data segment	Zero	Within block	Till end of the program
Register	CPU Register	Garbage	Within block	End of block

1. auto

This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared.

`Syntax` **storage_class** var_data_type *var_name*;

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

2 extern

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block.

Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead, we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

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

3. <u>static</u>

This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared.

Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

```
#include<stdio.h>
static char c;
```

- **static int** i;
- **static float** f;
- **static char** s[100];
- **void** main ()
- {
- printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printed.
- }

4. register

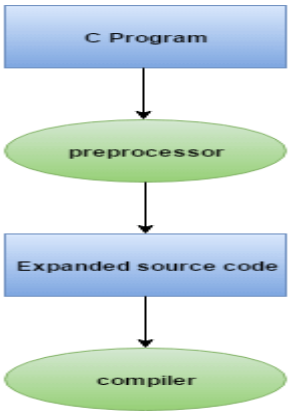
This storage class declares register variables that have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.

If a free registration is not available, these are then stored in the memory only. Usually, a few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

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

C PREPROCESSOR DIRECTIVES

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.



All preprocessor directives starts with hash # symbol.

Let's see a list of preprocessor directives.

- #include
- #define
- #undef
- #ifdef
- #ifndef
- #if
- #else
- #elif
- #endif
- #error
- #pragma

UNIT – III

LINEAR DATA STRUCTURES

Arrays and its representations

Stacks and Queues – Applications

Linked lists

- Single
- Circular
- Doubly Linked list
- Application

UNIT III - LINEAR DATA STRUCTURES

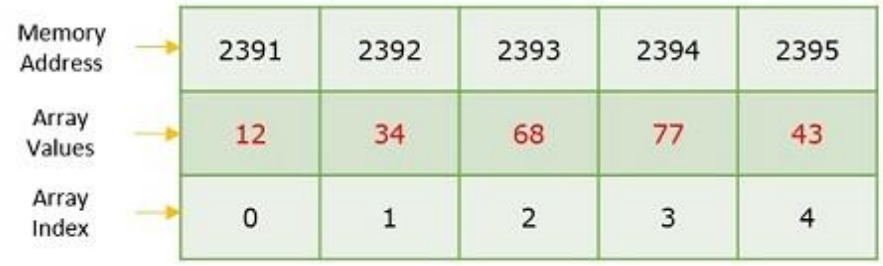
ARRAYS AND ITS REPRESENTATIONS

Array is a type of linear data structure that is defined as a collection of elements with same or different data types. They exist in both single dimension and multiple dimensions. These data structures come into picture when there is a necessity to store multiple elements of similar nature together at one place.

The difference between an array index and a memory address is that the array index acts like a key value to label the elements in the array. However, a memory address is the starting address of free memory available.

Following are the important terms to understand the concept of Array.

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.



Syntax

Creating an array in **C** and **C++** programming languages −

data_type array_name[array_size] = {elements separated using commas}
or,
data_type array_name[array_size];

Creating an array in **Java** programming language −

data_type[] array_name = {elements separated by commas}

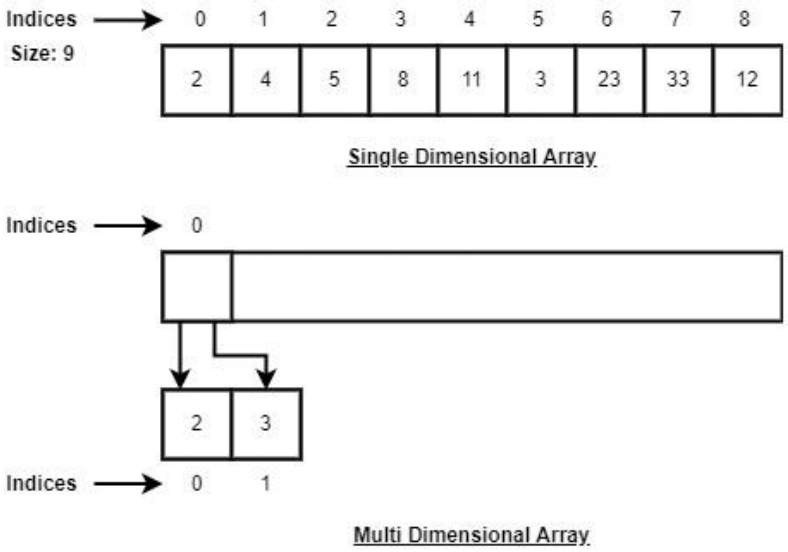
or,

data_type array_name = new data_type[array_size];

Array Representation

Arrays are represented as a collection of buckets where each bucket stores one element. These buckets are indexed from '0' to 'n-1', where n is the size of that particular array. For example, an array with size 10 will have buckets indexed from 0 to 9.

This indexing will be similar for the multidimensional arrays as well. If it is a 2-dimensional array, it will have sub-buckets in each bucket. Then it will be indexed as array_name[m][n], where m and n are the sizes of each level in the array.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 9 which means it can store 9 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 23.

Basic Operations in the Arrays

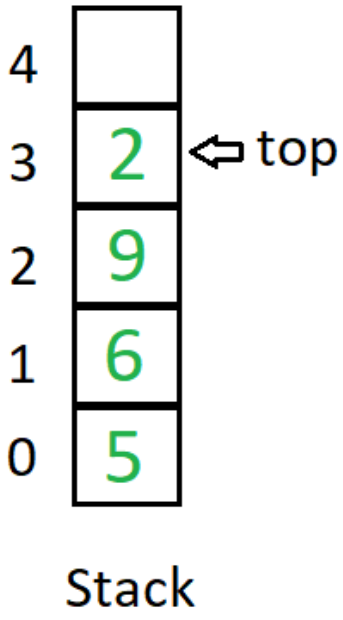
The basic operations in the Arrays are insertion, deletion, searching, display, traverse, and update. These operations are usually performed to either modify the data in the array or to report the status of the array.

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.
- **Display** − Displays the contents of the array.

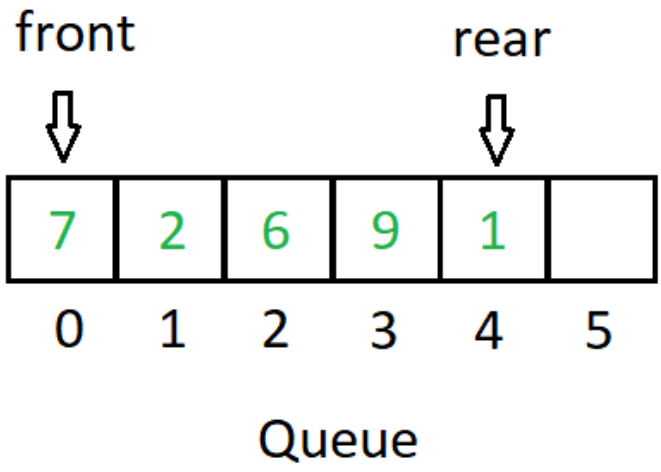
STACKS AND QUEUES – APPLICATIONS

Stack: A stack is a linear data structure in which elements can be inserted and deleted only from one side of the list, called the **top**. A stack follows the **LIFO** (Last In First Out) principle, i.e., the element inserted at the last is the first element to come out. The insertion of an element into the stack is called **push** operation, and the deletion of an element from the stack is called **pop** operation. In stack, we always keep track of the last element present in the list with a pointer called **top**.
The diagrammatic representation of the stack is given below:



Queue is a linear data structure in which elements can be inserted only from one side of the list called **rear**, and the elements can be deleted only from the other side called the **front**. The queue data structure follows the **FIFO** (First In First Out) principle, i.e. the element inserted at first in the list, is the first element to be removed from the list. The insertion of an element in a queue is called an **enqueue** operation and the deletion of an element is called a **dequeue** operation. In queue, we always maintain two pointers, one pointing to the element which was inserted at the first and still present in the list with the **front** pointer and the second pointer pointing to the element inserted at the last with the **rear** pointer.

The diagrammatic representation of the queue is given below:



Difference between Stack and Queue Data Structures are as follows:

Stacks	Queues
A stack is a data structure that stores a collection of elements, with operations to push (add) and pop (remove) elements from the top of the stack.	A queue is a data structure that stores a collection of elements, with operations to enqueue (add) elements at the back of the queue, and dequeue (remove) elements from the front of the queue.
Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list.	Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.
Stacks are often used for tasks that require backtracking, such as parsing expressions or implementing undo functionality.	Queues are often used for tasks that involve processing elements in a specific order, such as handling requests or scheduling tasks.

Stacks	Queues
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
Insert operation is called push operation.	Insert operation is called enqueue operation.
Stacks are implemented using an array or linked list data structure.	Queues are implemented using an array or linked list data structure.
Delete operation is called pop operation.	Delete operation is called dequeue operation.
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.
Stack is used in solving problems works on recursion.	Queue is used in solving problems having sequential processing.
Stacks are often used for recursive algorithms or for maintaining a history of function calls.	Queues are often used in multithreaded applications, where tasks are added to a queue and executed by a pool of worker threads.
Stack does not have any types.	Queue is of three types – 1. Circular Queue 2. Priority queue 3. double-ended queue.

Stacks	Queues
Can be considered as a vertical collection visual.	Can be considered as a horizontal collection visual.
Examples of stack-based languages include PostScript and Forth.	Examples of queue-based algorithms include Breadth-First Search (BFS) and printing a binary tree level-by-level.

Applications of stack:

- Some CPUs have their entire assembly language based on the concept of performing operations on registers that are stored in a **stack.**
- **Stack structure** is used in the C run-time system.

Applications of queue:

- **Queue** data structure is implemented in the hardware microinstructions inside a CPU.
- **Queue** structure is used in most operating systems.

LINKED LISTS

A linked list is a collection of "nodes" connected together via links. These nodes consist of the data to be stored and a pointer to the address of the next node within the linked list. In the case of arrays, the size is limited to the definition, but in linked lists, there is no defined size. Any amount of data can be stored in it and can be deleted from it.

There are three types of linked lists −

- **Singly Linked List** − The nodes only point to the address of the next node in the list.
- **Doubly Linked List** − The nodes point to the addresses of both previous and next nodes.
- **Circular Linked List** − The last node in the list will point to the first node in the list. It can either be singly linked or doubly linked.

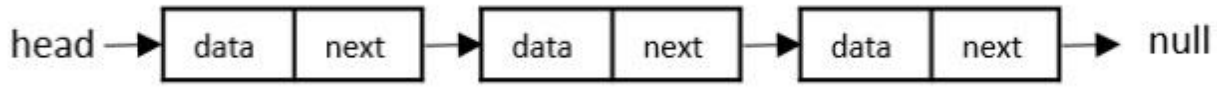
Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

TYPES OF LINKED LIST

SINGLY LINKED LISTS

Singly linked lists contain two “buckets” in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.



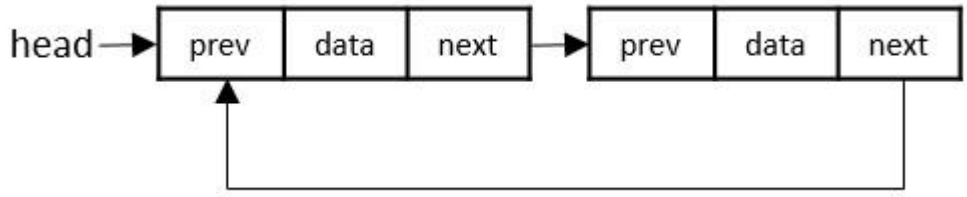
DOUBLY LINKED LISTS

Doubly Linked Lists contain three “buckets” in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.

CIRCULAR LINKED LISTS

Circular linked lists can exist in both singly linked list and doubly linked list.

Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.



BASIC OPERATIONS IN THE LINKED LISTS

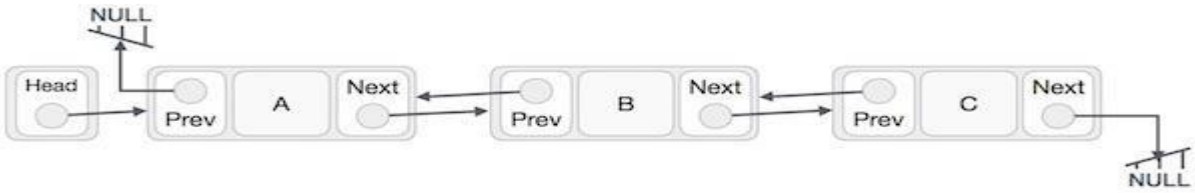
The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key. These operations are performed on Singly Linked Lists as given below −

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **Prev** − Each link of a linked list contains a link to the previous link called Prev.
- **Linked List** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations

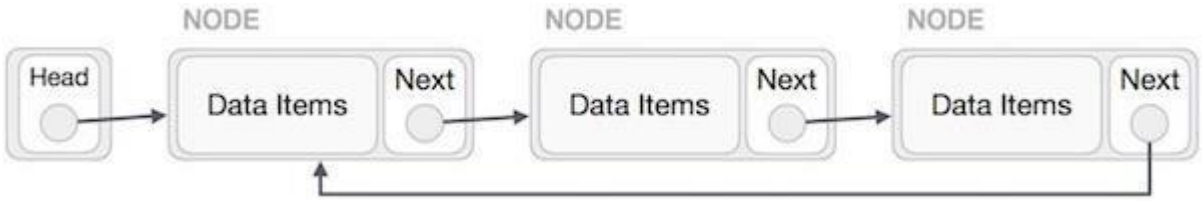
Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.
- **Display backward** − Displays the complete list in a backward manner.

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

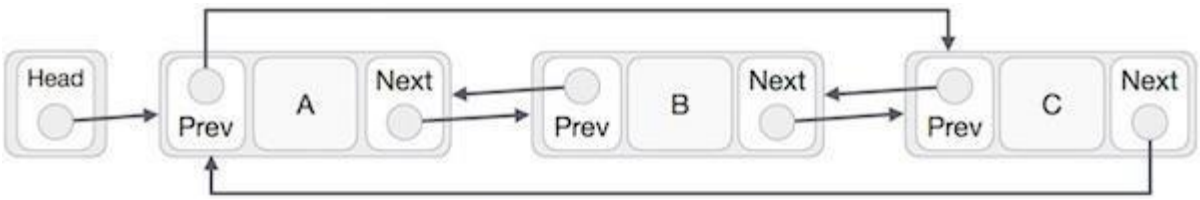
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- **insert** − Inserts an element at the start of the list.
- **delete** − Deletes an element from the start of the list.
- **display** − Displays the list.

- **Implementation of a Queue:** A circular linked list can be used to implement a queue, where the front and rear of the queue are the first and last nodes of the list, respectively. In this implementation, when an element is enqueued, it is added to the rear of the list and when an element is dequeued, it is removed from the front of the list.
- **Music or Media Player:** Circular linked lists can be used to create a playlist for a music or media player. The playlist can be represented as a circular linked list, where each node contains information about a song or media item, and the next node points to the next song in the playlist. This allows for continuous playback of the playlist.
- **Hash table implementation:** Circular linked lists can be used to implement a hash table, where each index in the table is a circular linked list. This is a form of collision resolution, where when two keys hash to the same index, the nodes are added to the circular linked list at that index.
- **Memory allocation**: In computer memory management, circular linked lists can be used to keep track of allocated and free blocks of memory. Each node in the list represents a block of memory and its status, with the next node pointing to the next block of memory. When a block of memory is freed, it can be added back to the circular linked list.

Real-Life Application of Circular Linked Lists:

- **Music and Media Players:** Circular linked lists can be used to implement playlists in music and media players. Each node in the list can represent a song or media item, and the "next" pointer can point to the next song in the playlist. When the end of the playlist is reached, the pointer can be set to the beginning of the list, creating a circular structure that allows for continuous playback.
- **Task Scheduling**: Circular linked lists can be used to implement task scheduling algorithms, where each node in the list represents a task and its priority. The "next" pointer can point to the next task in the queue, with the end of the queue pointing back to the beginning to create a circular structure. This allows for a continuous loop of task scheduling, where tasks are added and removed from the queue based on their priority.
- **Cache Management**: Circular linked lists can be used in cache management algorithms to manage the replacement of cache entries. Each node in the list can represent a cache entry, with the "next" pointer pointing to the next entry in the list. When the end of the list is reached, the pointer can be set to the beginning of the list, creating a circular structure that allows for the replacement of older entries with newer ones.
- **File System Management**: Circular linked lists can be used in file system management to track the allocation of disk space. Each node in the list can represent a block of disk space, with the "next" pointer pointing to the next available block. When the end of the list is reached, the pointer can be set to the beginning of the list, creating a circular structure that allows for the allocation and deallocation of disk space.

Advantages of Circular Linked Lists:

- **Efficient operations:** Since the last node of the list points back to the first node, circular linked lists can be traversed quickly and efficiently. This makes them useful for applications that require frequent traversal, such as queue and hash table implementations.
- **Space efficiency**: Circular linked lists can be more space-efficient than other types of linked lists because they do not require a separate pointer to keep track of the end of the list. This means that circular linked lists can be more compact and take up less memory than other types of linked lists.
- **Flexibility**: The circular structure of the list allows for greater flexibility in certain applications. For example, a circular linked list can be used to represent a ring or circular buffer, where new elements can be added and old elements can be removed without having to shift the entire list.
- **Dynamic size**: Circular linked lists can be dynamically sized, which means that nodes can be added or removed from the list as needed. This makes them useful for applications where the size of the list may change frequently, such as memory allocation and music or media players.
- **Ease of implementation**: Implementing circular linked lists is often simpler than implementing other types of linked lists. This is because circular linked lists have a simple, circular structure that is easy to understand and implement.
-

Disadvantages of Circular Linked Lists:

- **Complexity**: Circular linked lists can be more complex than other types of linked lists, especially when it comes to algorithms for insertion and deletion operations. For example, determining the correct position for a new node can be more difficult in a circular linked list than in a linear linked list.
- **Memory leaks**: If the pointers in a circular linked list are not managed properly, memory leaks can occur. This happens when a node is removed from the list but its memory is not freed, leading to a buildup of unused memory over time.

- **Traversal can be more difficult:** While traversal of a circular linked list can be efficient, it can also be more difficult than linear traversal, especially if the circular list has a complex structure. Traversing a circular linked list requires careful management of the pointers to ensure that each node is visited exactly once.
- **Lack of a natural end:** The circular structure of the list can make it difficult to determine when the end of the list has been reached. This can be a problem in certain applications, such as when processing a list of data in a linear fashion.

UNIT-IV

NON-LINEAR DATA STRUCTURES

UNIT IV

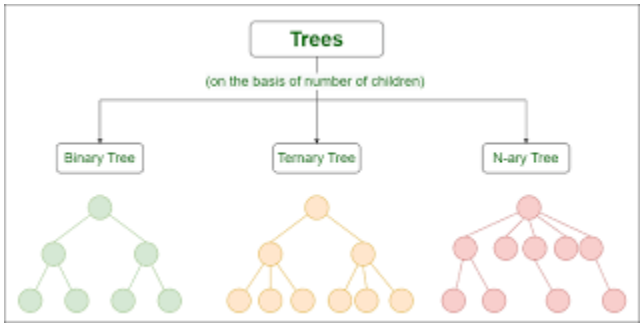
Overview

A Non Linear Data Structure is one in which its elements are not connected in a linear fashion, as suggested by its name itself. In such a data structure elements might be connected in a hierarchical manner like a tree or graph, or it may be non hierarchical like in a LinkedList.

Tree Data Structure is a hierarchical data structure in which a collection of elements known as nodes are connected to each other via edges such that there exists exactly one path between any two nodes.
The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.

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:



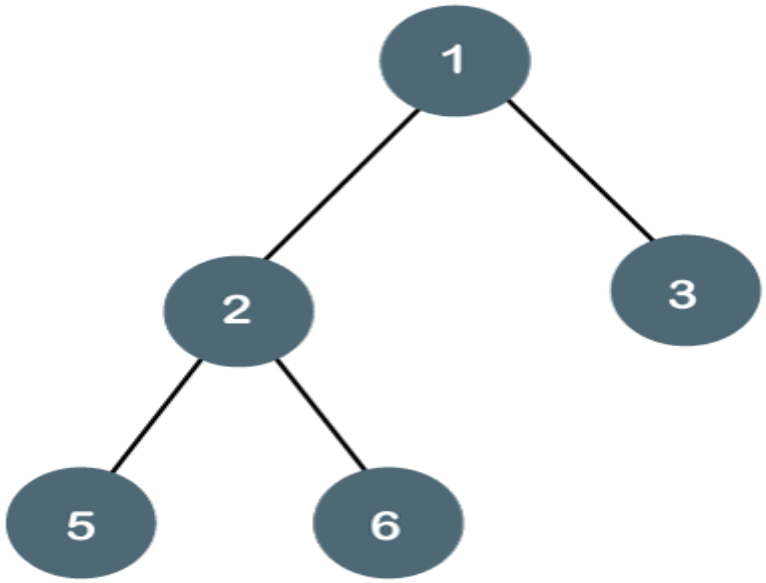
Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{K, L, M, N, O, P}** are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
- **Descendant:** Any successor node on the path from the leaf node to that node. **{E,I}** are the descendants of the node **{B}.**
- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.

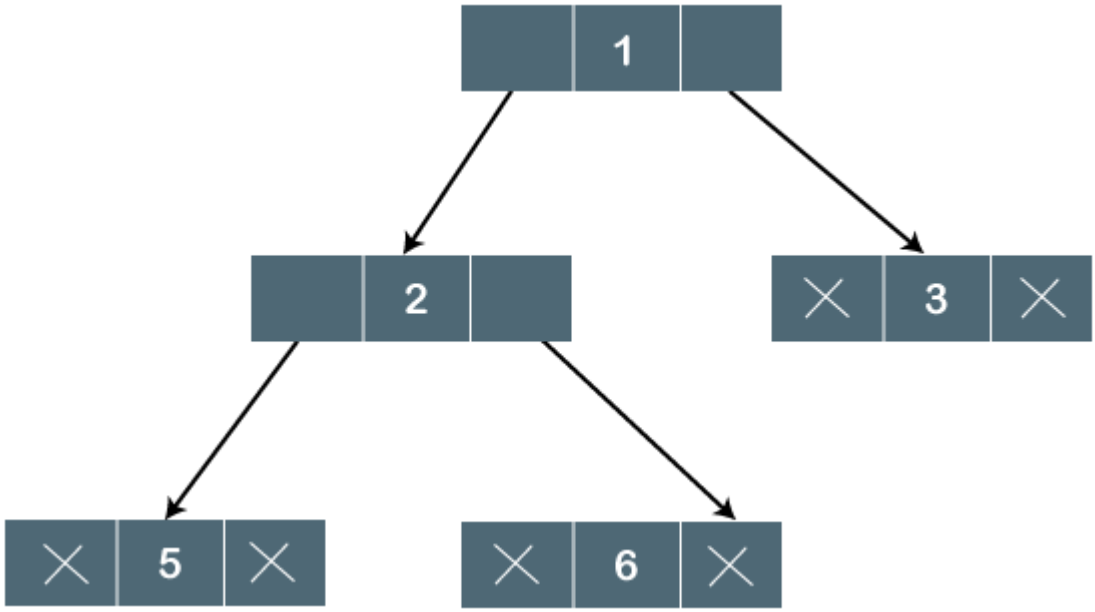
BINARY TREE

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.



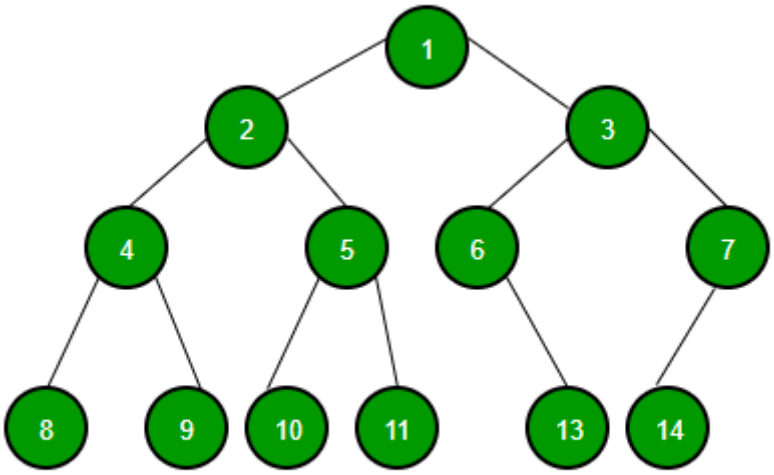
The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



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.

Binary Tree Representation And Traversals

Binary Tree is defined as a tree data structure where each node has at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



Binary Tree Representation

A Binary tree is represented by a pointer to the topmost node (commonly known as the “root”) of the tree. If the tree is empty, then the value of the root is NULL. Each node of a Binary Tree contains the following parts:

1. Data
2. Pointer to left child
3. Pointer to right child

Basic Operation On Binary Tree:

- Inserting an element.
- Removing an element.
- Searching for an element.
- Traversing the tree.

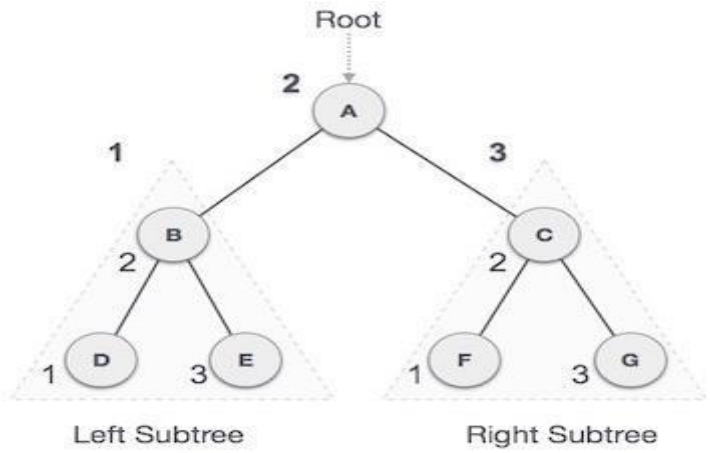
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

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

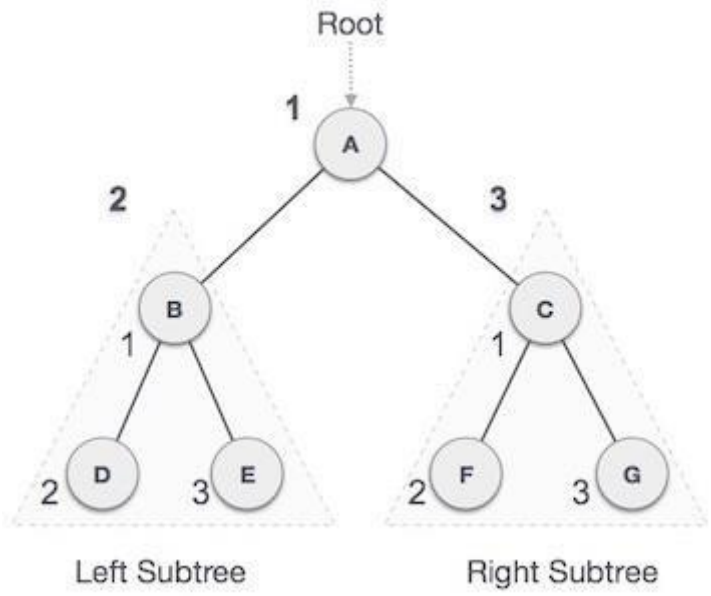


We start from **A**, and following in-order traversal, we move to its left subtree **B**.**B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be −

D → B → E → A → F → C → G

PRE-ORDER TRAVERSAL

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

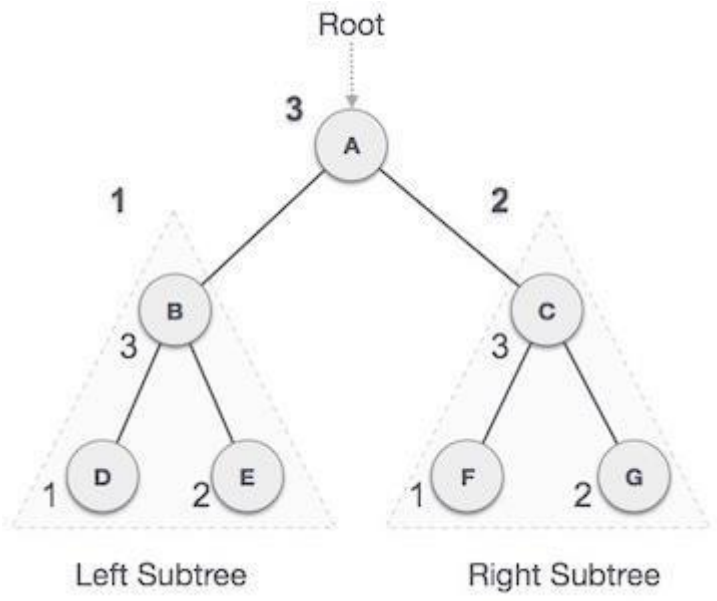


We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

POST-ORDER TRAVERSAL

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



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

D → E → B → F → G → C → A

<u>Applications of Tree:</u>

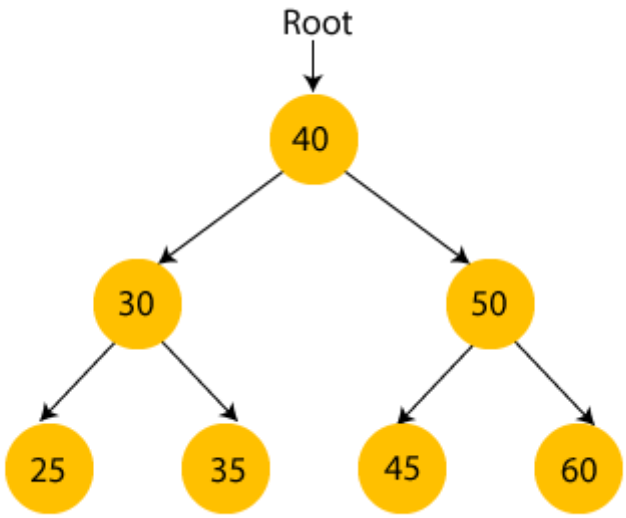
- **File Systems**: The file system of a computer is often represented as a tree. Each folder or directory is a node in the tree, and files are the leaves.
- **XML Parsing:** Trees are used to parse and process XML documents. An XML document can be thought of as a tree, with elements as nodes and attributes as properties of the nodes.
- **Database Indexing:** Many databases use trees to index their data. The B-tree and its variations are commonly used for this purpose.

- **Compiler Design:** The syntax of programming languages is often defined using a tree structure called a parse tree. This is used by compilers to understand the structure of the code and generate machine code from it.
- **Artificial Intelligence:** Decision trees are often used in artificial intelligence to make decisions based on a series of criteria.

BINARY SEARCH TREES

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.

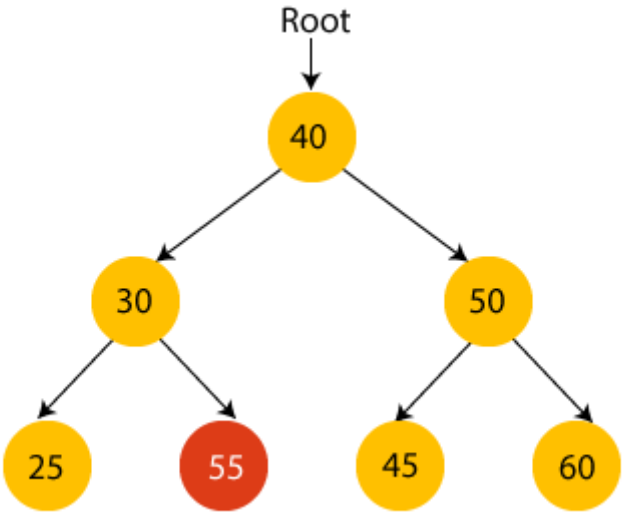
Let's understand the concept of Binary search tree with an example.



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**

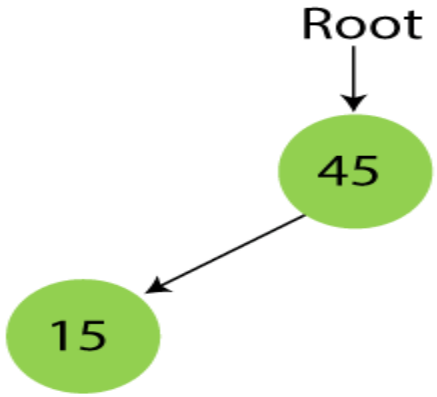
- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left sub tree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right sub tree.

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.

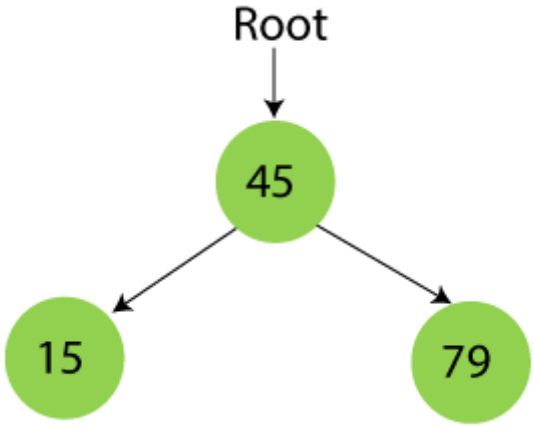
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



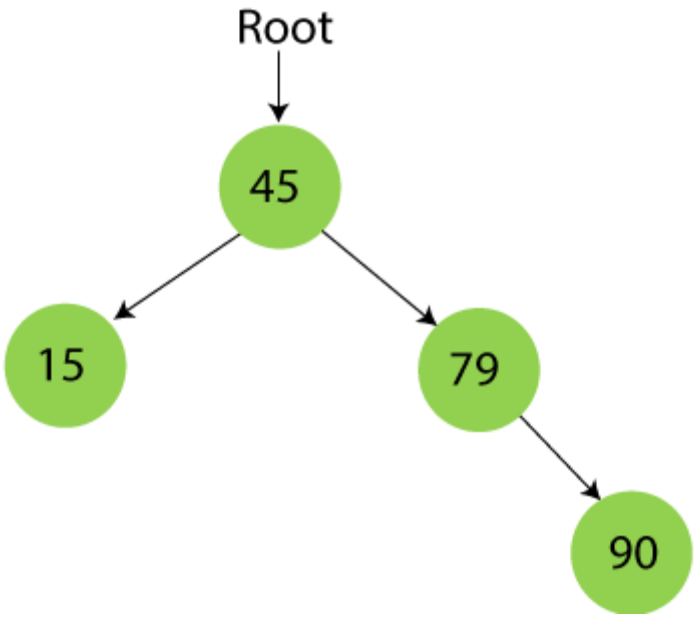
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



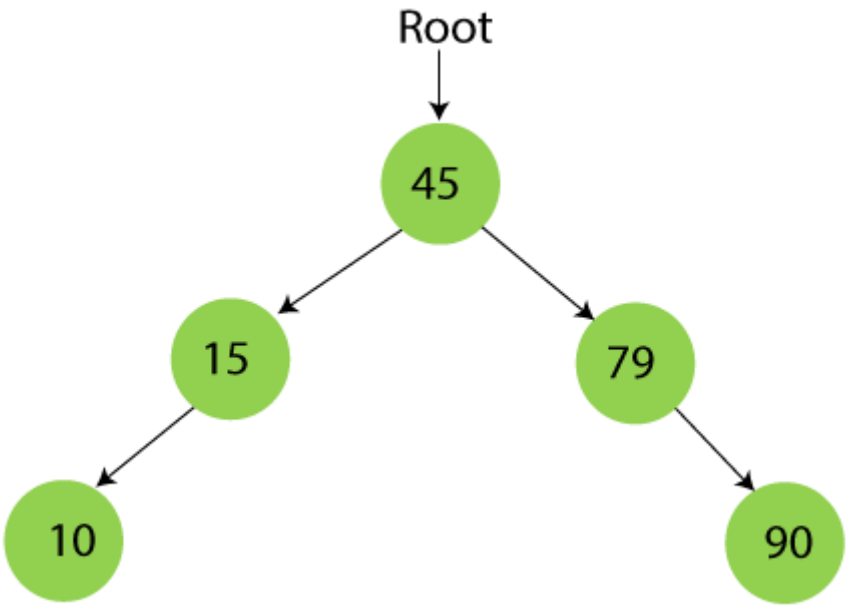
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



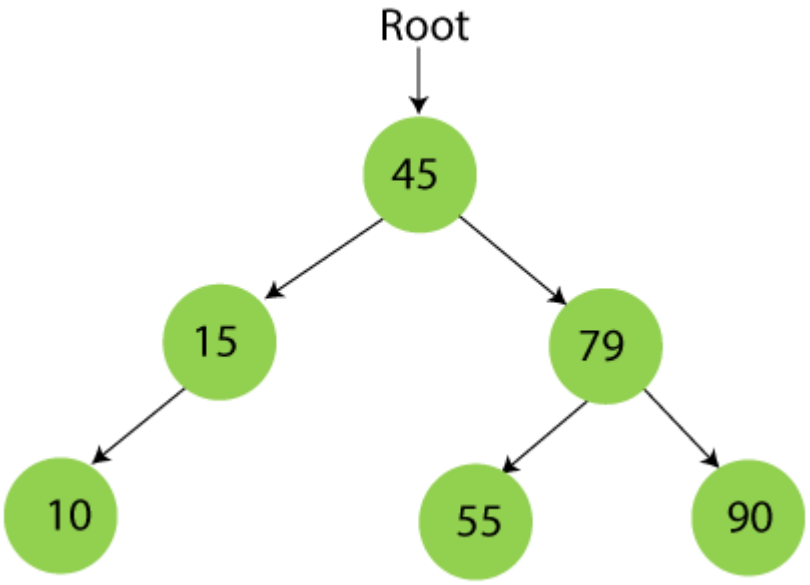
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



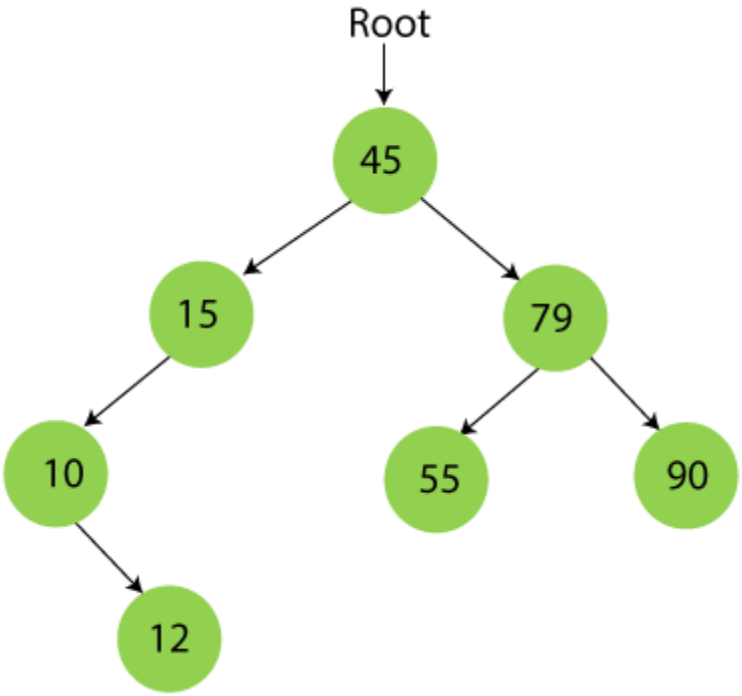
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



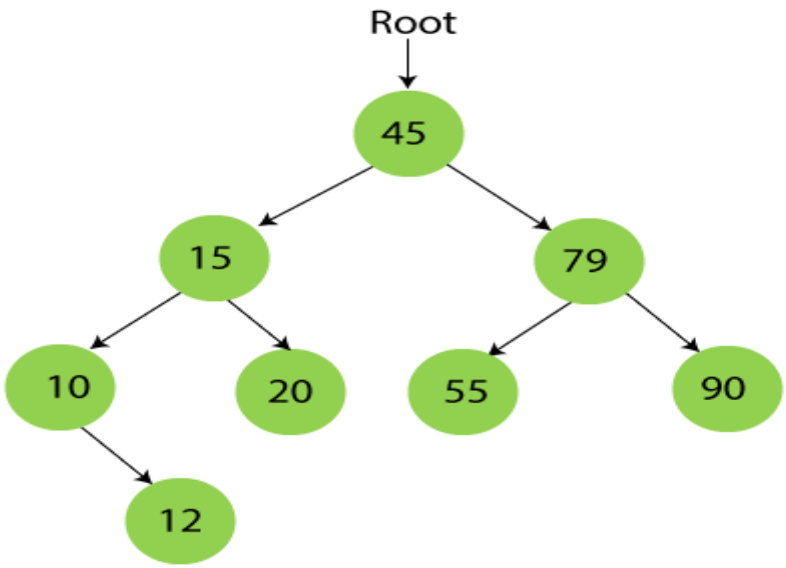
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



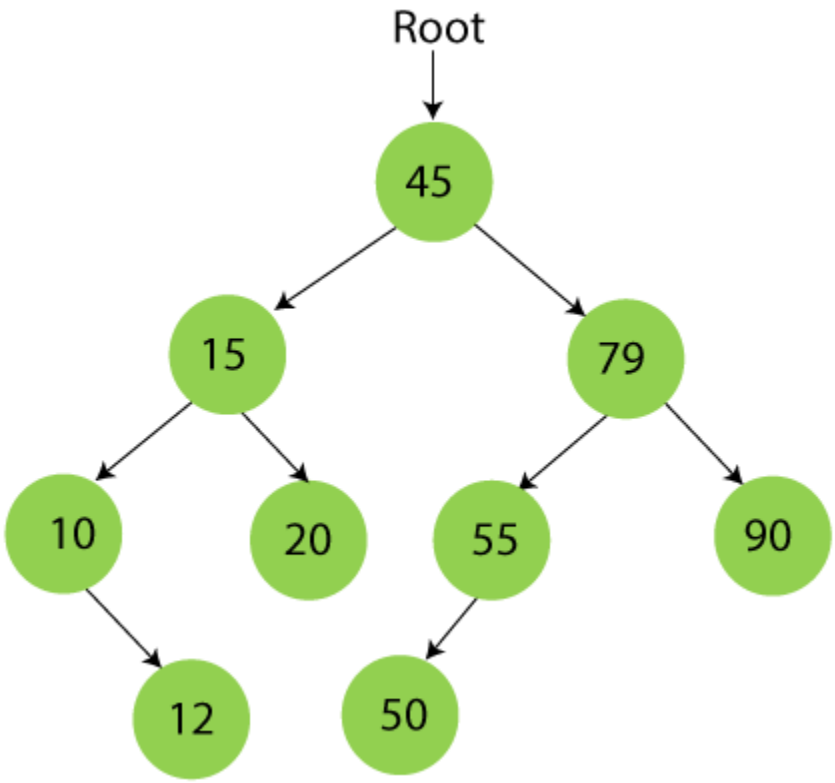
Step 8 - Insert 20.

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.

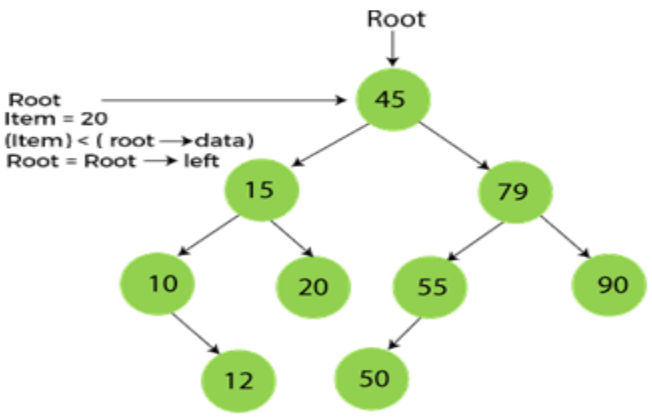
Let's understand how a search is performed on a binary search tree.

SEARCHING IN BINARY SEARCH TREE

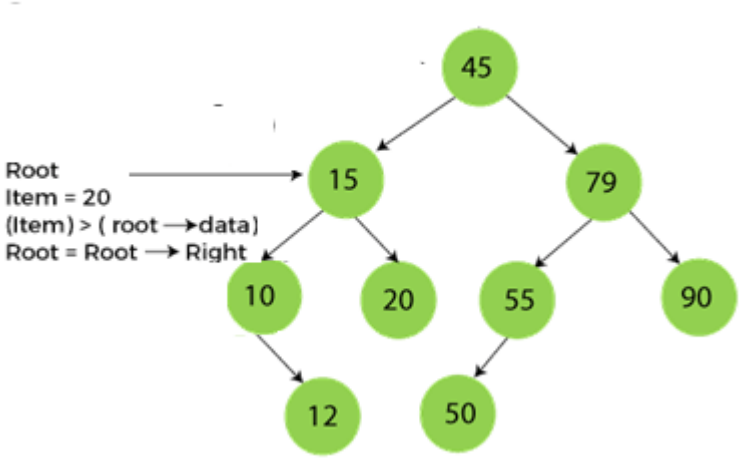
Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

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.
7. 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.

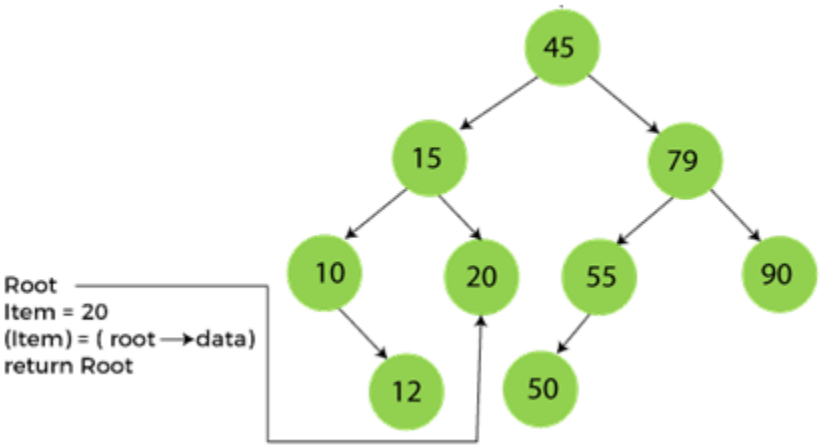
Step1:



8. **Step2:**



9. **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

1. Search (root, item)
2. Step 1 - if (item = root → data) or (root = NULL)
3. return root
4. else if (item < **root** → data)
5. return Search(root → left, item)
6. else

7. return Search(root → right, item)
8. END if
9. Step 2 - END

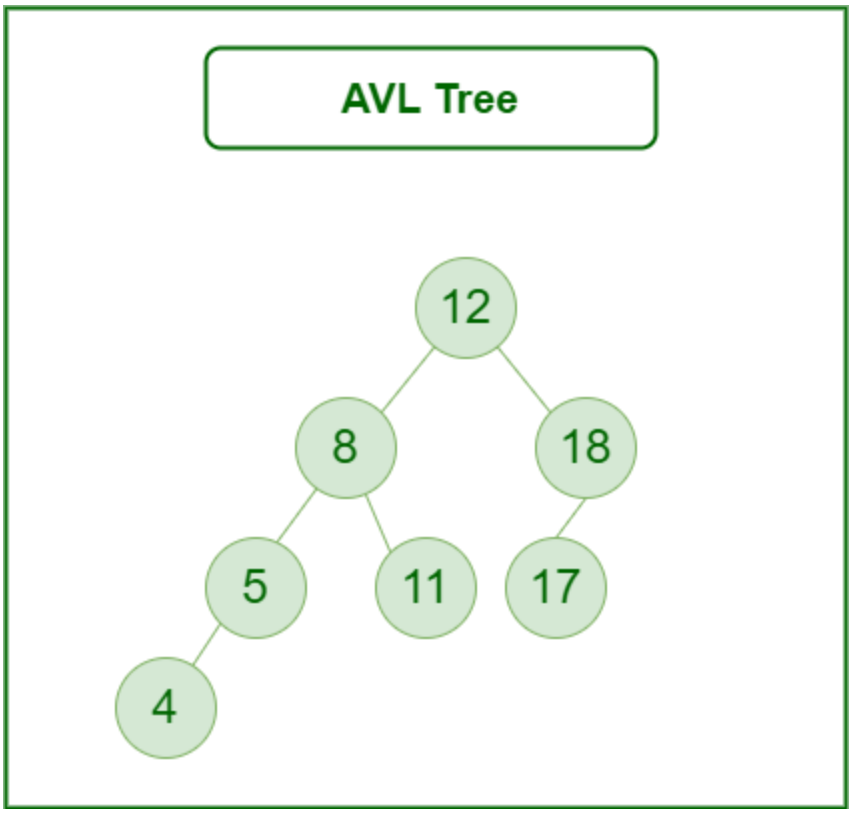
AVL TREES.

*An **AVL tree** defined as a self-balancing **Binary Search Tree** (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.*

The difference between the heights of the left subtree and the right subtree for any node is known as the **balance factor** of the node.

The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper "An algorithm for the organization of information".

Example of AVL Trees:



The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

Operations on an AVL Tree:

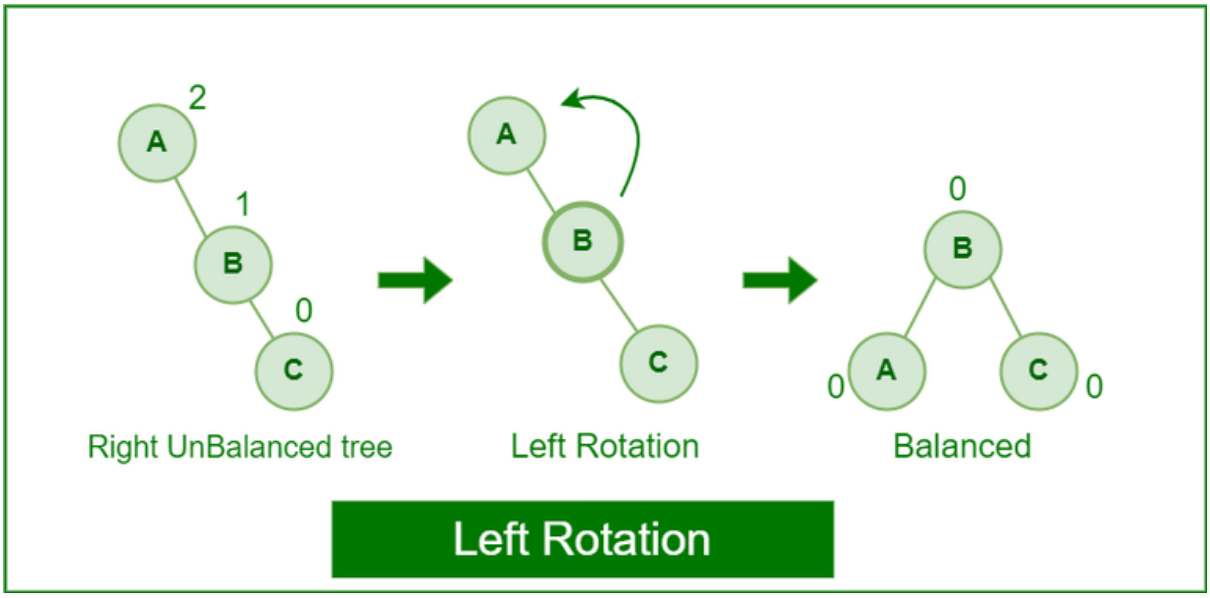
- Insertion
- Deletion
- Searching [It is similar to performing a search in BST]

Rotating the subtrees in an AVL Tree:

An AVL tree may rotate in one of the following four ways to keep itself balanced:

Left Rotation:

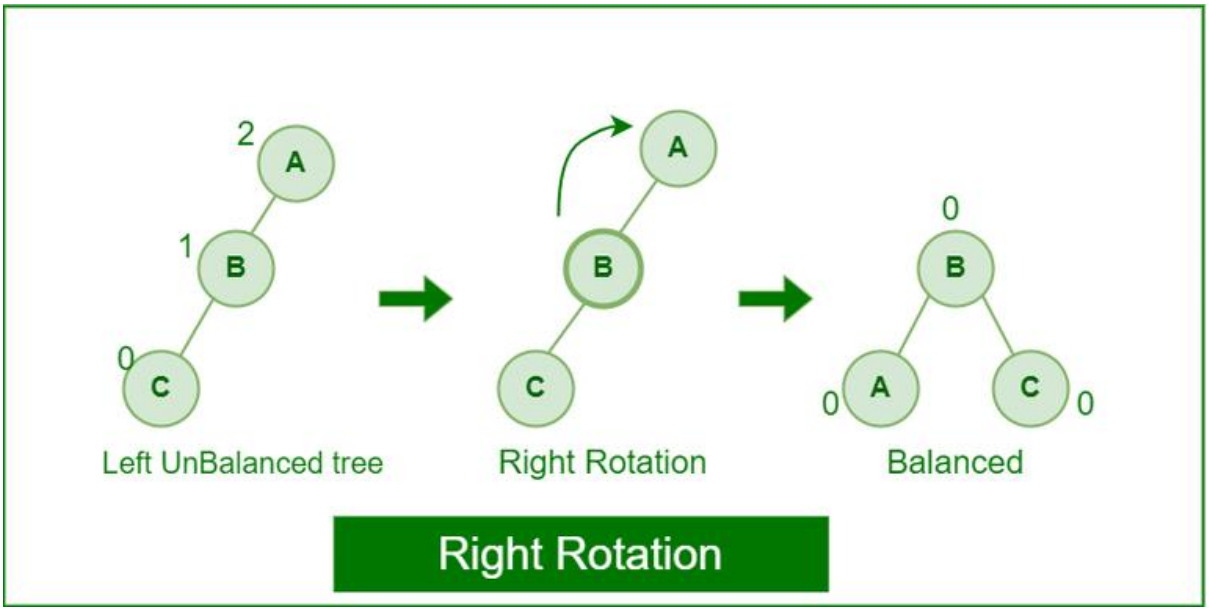
When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.



Left-Rotation in AVL tree

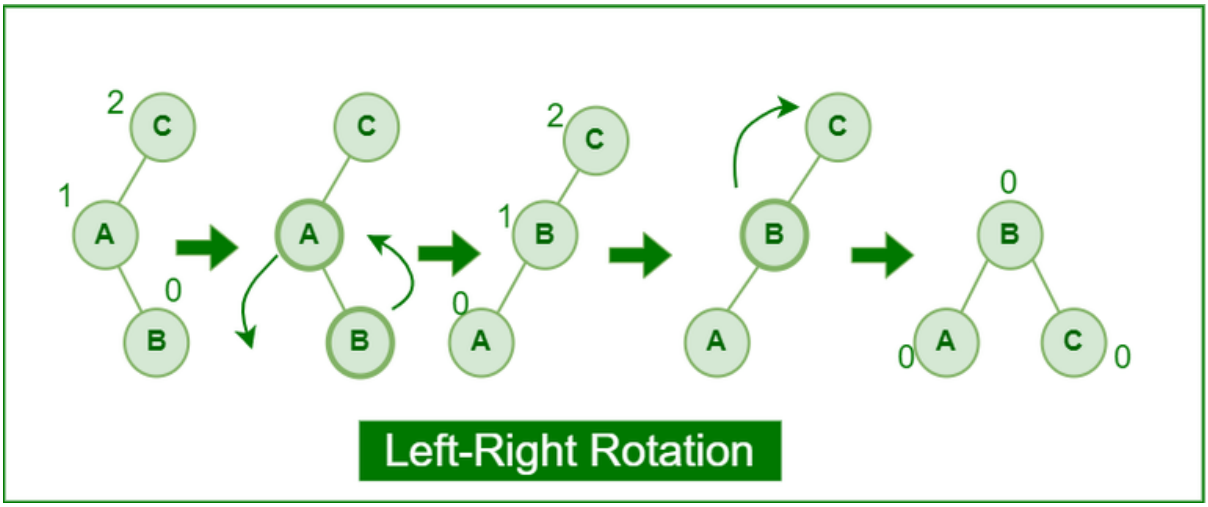
Right Rotation:

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



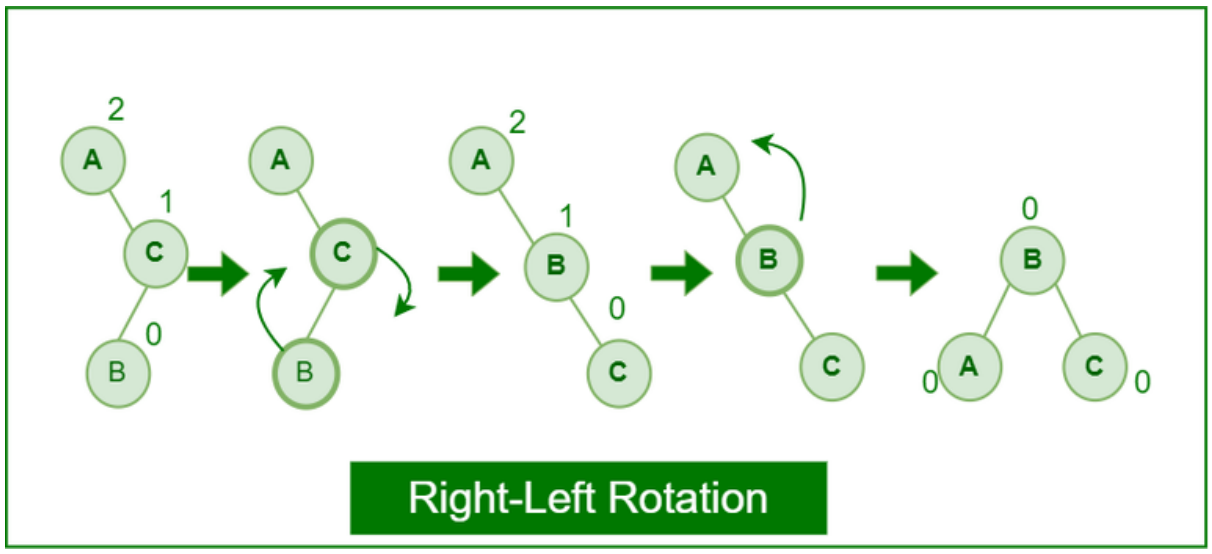
Left-Right Rotation:

A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



Right-Left Rotation:

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



Applications of AVL Tree:

1. It is used to index huge records in a database and also to efficiently search in that.
2. For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.
3. Database applications, where insertions and deletions are less common but frequent data lookups are necessary
4. Software that needs optimized search.
5. It is applied in corporate areas and storyline games.

Advantages of AVL Tree:

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. It provides faster lookups than Red-Black Trees
4. Better searching time complexity compared to other trees like binary tree.
5. Height cannot exceed log(N), where, N is the total number of nodes in the tree.

Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Less used compared to Red-Black trees.
4. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
5. Take more processing for balancing.

GRAPH AND ITS REPRESENTATIONS

Graph Representations

In graph theory, a graph representation is a technique to store graph into the memory of computer.

To represent a graph, we just need the set of vertices, and for each vertex the neighbors of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.

There are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use.

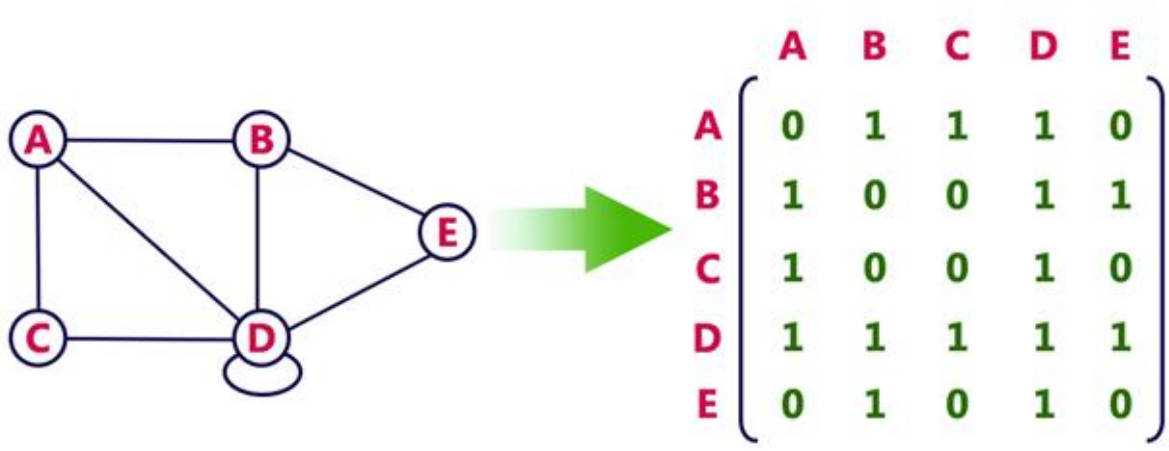
1. Adjacency Matrix

- Adjacency matrix is a sequential representation.
- It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
- In this representation, we have to construct a nXn matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, $a^{i,j} = 1$, otherwise $a^{i,j} = 0$.
- If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

Example

Consider the following **undirected graph representation**:

Undirected graph representation



Graph Representations

In graph theory, a graph representation is a technique to store graph into the memory of computer.

To represent a graph, we just need the set of vertices, and for each vertex the neighbors of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.

There are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use.

1. Adjacency Matrix

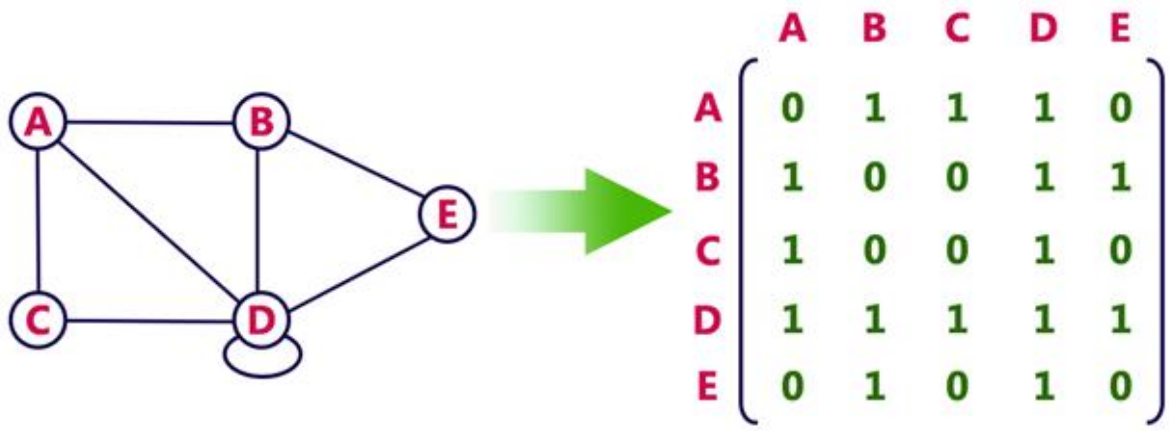
- Adjacency matrix is a sequential representation.
- It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
- In this representation, we have to construct a nXn matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, $a^{i,j} = 1$, otherwise $a^{i,j} = 0$.

- If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

Example

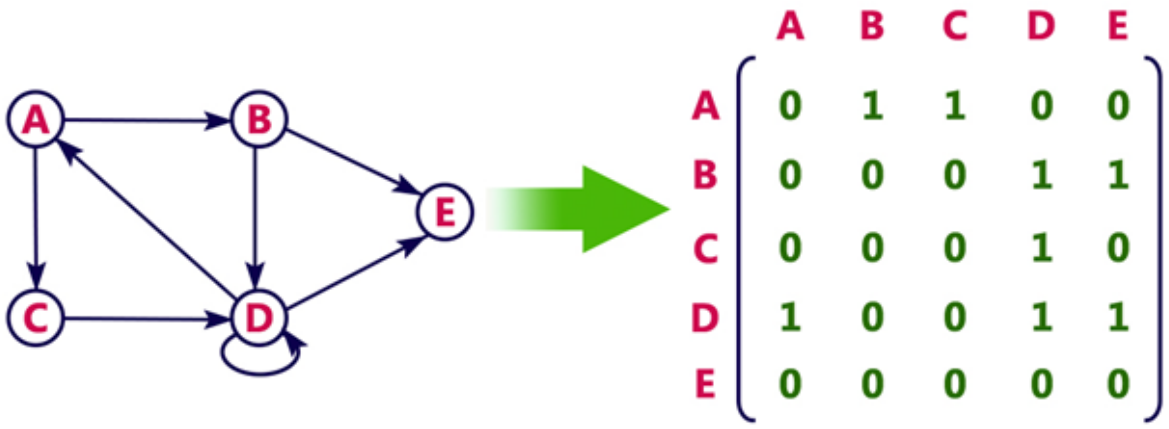
Consider the following **undirected graph representation**:

Undirected graph representation



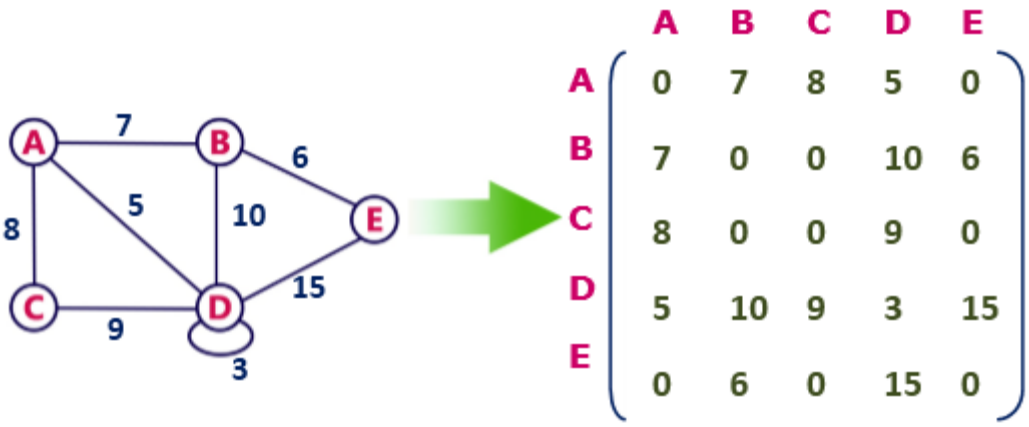
Directed graph represenation

See the directed graph representation:



In the above examples, 1 represents an edge from row vertex to column vertex, and 0 represents no edge from row vertex to column vertex.

Undirected weighted graph represenation



Pros: Representation is easier to implement and follow.

Cons: It takes a lot of space and time to visit all the neighbors of a vertex, we have to traverse all the vertices in the graph, which takes quite some time.

2. Incidence Matrix

In **Incidence matrix representation**, graph can be represented using a matrix of size:

Total number of vertices by total number of edges.

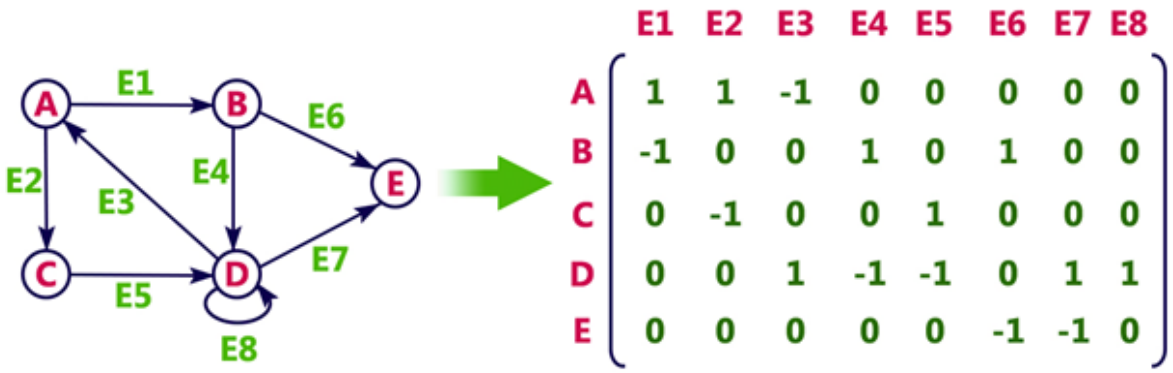
It means if a graph has 4 vertices and 6 edges, then it can be represented using a matrix of 4X6 class. In this matrix, columns represent edges and rows represent vertices.

This matrix is filled with either **0 or 1** or -1. Where,

- 0 is used to represent row edge which is not connected to column vertex.
- 1 is used to represent row edge which is connected as outgoing edge to column vertex.
- -1 is used to represent row edge which is connected as incoming edge to column vertex.

Example

Consider the following directed graph representation.

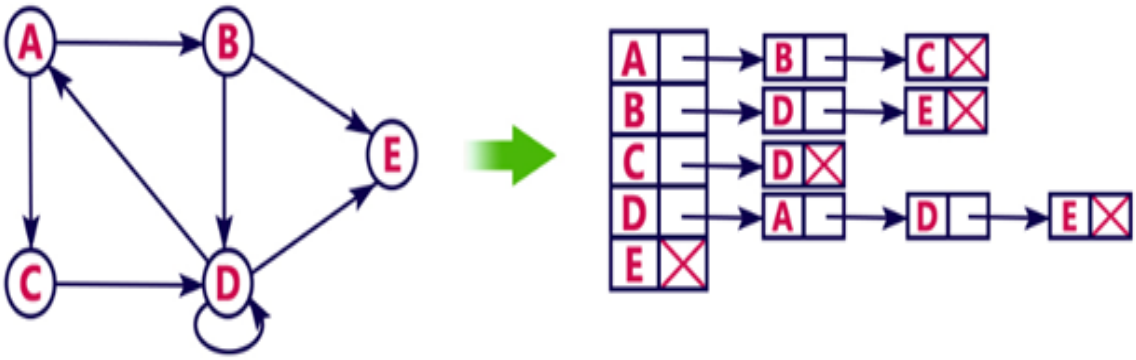


3. Adjacency List

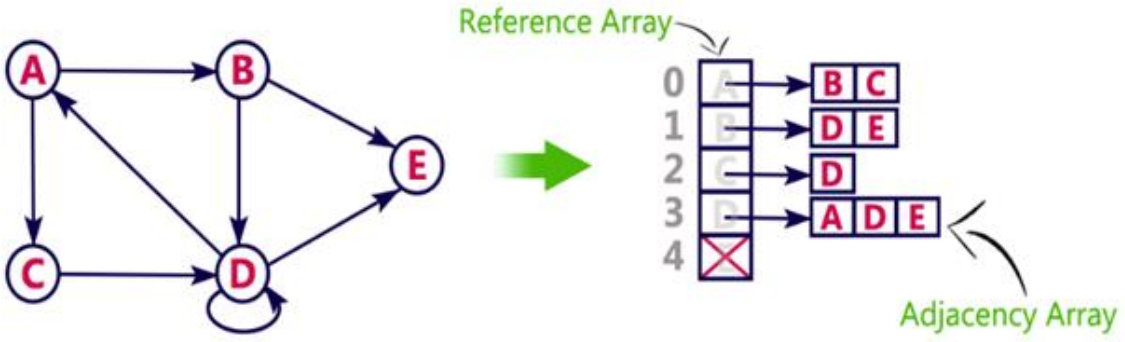
- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a **singly linked list** of neighbors of v.

Example

Let's see the following directed graph representation implemented using linked list:



We can also implement this representation using array as follows:



Pros:

- Adjacency list saves lot of space.
- We can easily insert or delete as we use linked list.
- Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.

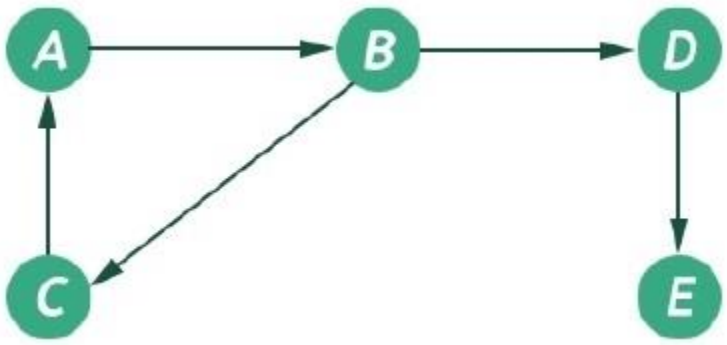
Cons:

- The adjacency list allows testing whether two vertices are adjacent to each other but it is slower to support this operation.

GRAPH TRAVERSALS.

In this section we will see what is a graph data structure, and the traversal algorithms of it.

The **graph** is one non-linear data structure. That is consists of some nodes and their connected edges. The edges may be director or undirected. This graph can be represented as G(V, E). The following graph can be represented as G({A, B, C, D, E}, {(A, B), (B, D), (D, E), (B, C), (C, A)})



- The graph has two types of traversal algorithms. These are called the Breadth First Search and Depth First Search.
- **Breadth First Search (BFS)**
- The **Breadth First Search** (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.
- **Algorithm**
- bfs(vertices, start)
- Input: The list of vertices, and the start vertex.
- Output: Traverse all of the nodes, if the graph is connected.
- Begin
- define an empty queue que
- at first mark all nodes status as unvisited
- add the start vertex into the que
- while que is not empty, do
- delete item from que and set to u

- display the vertex u
- for all vertices 1 adjacent with u, do
- if vertices[i] is unvisited, then
- mark vertices[i] as temporarily visited
- add v into the queue
- mark
- done
- mark u as completely visited
- done
- End

Depth First Search (DFS)

The **Depth First Search** (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

Algorithm

dfs(vertices, start)
Input: The list of all vertices, and the start node.
Output: Traverse all nodes in the graph.
Begin
  initially make the state to unvisited for all nodes
  push start into the stack
  while stack is not empty, do
    pop element from stack and set to u
    display the node u
    if u is not visited, then
      mark u as visited

```
        for all nodes i connected to u, do
          if ith vertex is unvisited, then
            push ith vertex into the stack
            mark ith vertex as visited
        done
    done
End
```

UNIT V

SEARCHING AND SORTING ALGORITHMS

Linear Search

Binary Search

Sorting

Selection Sort

Bubble Sort

Insertion Sort

Merge Sort

Quick Sort

Hashing

Collision

UNIT V - SEARCHING AND SORTING ALGORITHMS

Linear Search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Pseudocode

```
procedure linear_search (list, value)

   for each item in the list
      if match item == value
         return the item's location
```

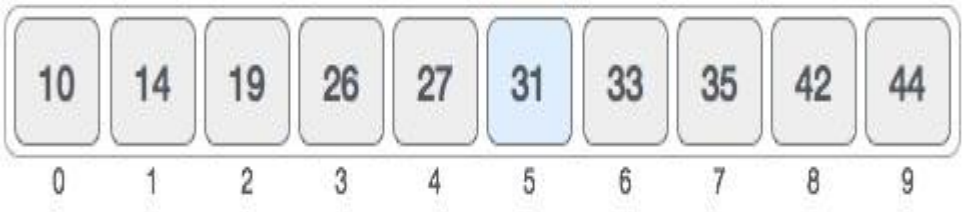
```
    end if
  end for

end procedure
```

BINARY SEARCH

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

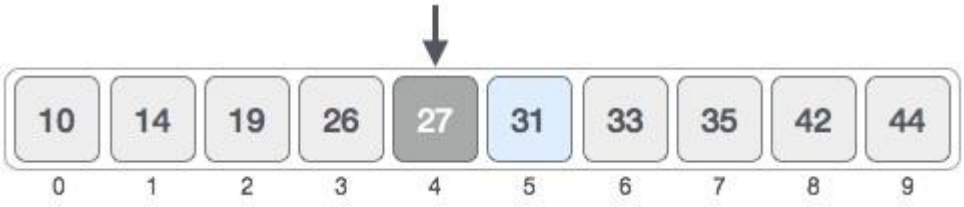
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

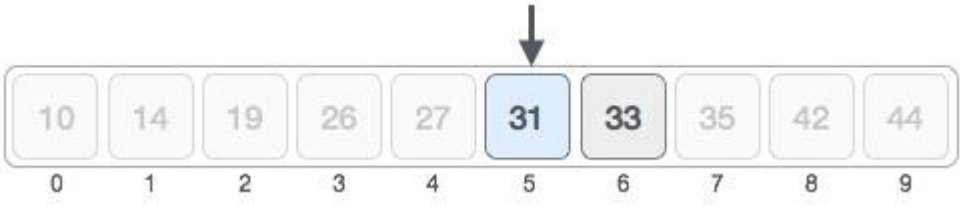
We change our low to mid + 1 and find the new mid value again.

low = mid + 1
mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

SORTING

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

- **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

SELECTION SORT

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

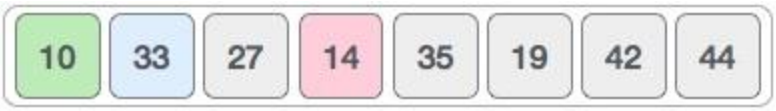
Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

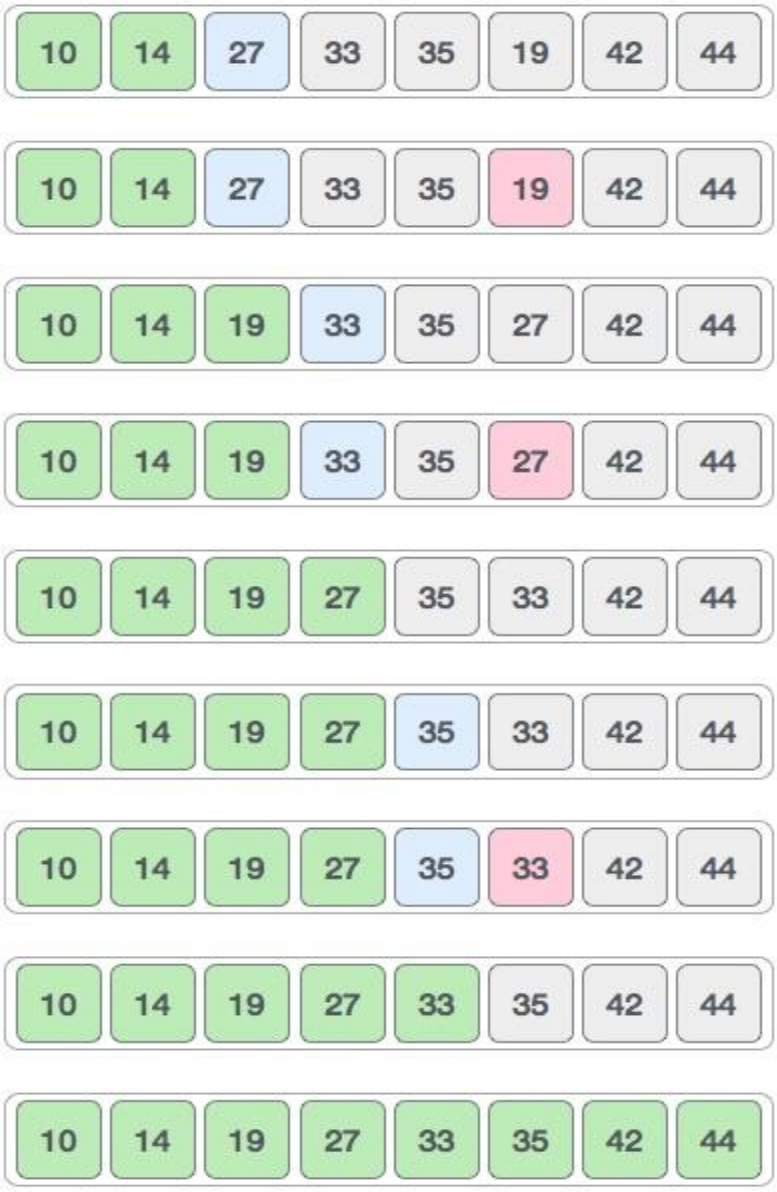
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Pseudocode

```
procedure selection sort
  list  : array of items
  n     : size of list

  for i = 1 to n - 1
  /* set current element as minimum*/
    min = i

    /* check the element to be minimum */

    for j = i+1 to n
      if list[j] < list[min] then
        min = j;
      end if
    end for

    /* swap the minimum element with the current element*/
    if indexMin != i  then
      swap list[min] and list[i]
    end if
  end for

end procedure
```

BUBBLE SORT

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

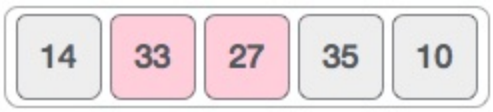
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

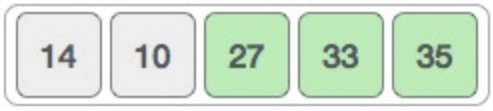
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

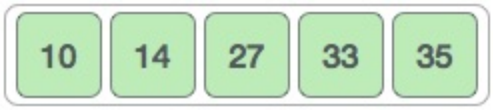
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

INSERTION SORT

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

MERGE SORT

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

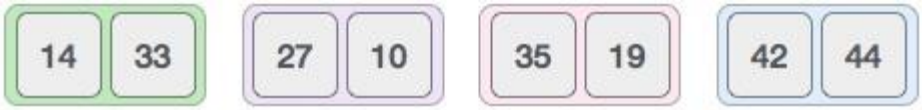
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.



We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

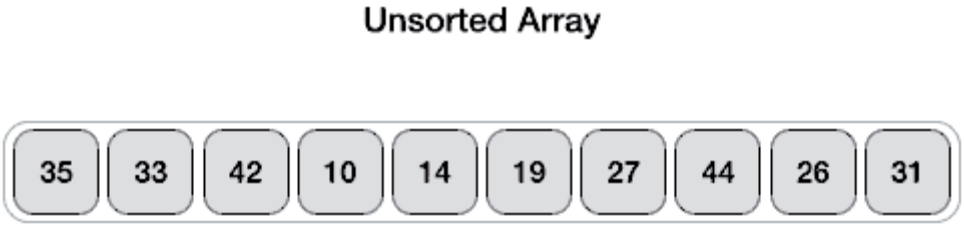
QUICK SORT

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively.

Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

Step 1 − Make the right-most index value pivot

Step 2 − partition the array using pivot value

Step 3 − quicksort left partition recursively

Step 4 − quicksort right partition recursively

HASHING

Hashing is a popular technique in computer science that involves mapping large data sets to fixed-length values. It is a process of converting a data set of variable size into a data set of a fixed size. The ability to perform efficient lookup operations makes hashing an essential concept in data structures.

A hashing algorithm is used to convert an input (such as a string or integer) into a fixed-size output (referred to as a hash code or hash value). The data is then stored and retrieved using this hash value as an index in an array or hash table. The hash function must be deterministic, which guarantees that it will always yield the same result for a given input.

Hashing is commonly used to create a unique identifier for a piece of data, which can be used to quickly look up that data in a large dataset. For example, a web browser may use hashing to store website passwords securely. When a user enters their password, the browser converts it into a hash value and compares it to the stored hash value to authenticate the user.

In the context of hashing, a hash key (also known as a hash value or hash code) is a fixed-size numerical or alphanumeric representation generated by a hashing algorithm. It is derived from the input data, such as a text string or a file, through a process known as hashing.

Hashing involves applying a specific mathematical function to the input data, which produces a unique hash key that is typically of fixed length, regardless of the size of the input. The resulting hash key is essentially a digital fingerprint of the original data.

The hash key serves several purposes. It is commonly used for data integrity checks, as even a small change in the input data will produce a significantly different hash key. Hash keys are also used for efficient data retrieval and storage in hash tables or data structures, as they allow quick look-up and comparison operations.

Types of Hashing.

Hashing based questions are asked in GATE and UGC NET exam are also explained in this tutorial. After reading this tutorial students will be able to attempt the hashing problems asked in previous year GATE Exam.

Two types of hashing in data structure are there

1.Open Hashing (Closed addressing)

Open Hashing is used to avoid collision and this method use an array of linked list to resolve the collision. This method is also known as closed addressing based hashing.

In open hashing it is important to note that each cell of array points to a list that contains collisions. Hashing has produced same index for all item in the linked list.

2.Closed hashing (Open addressing)

In this case there are three methods to resolve the collision which are discussed

1. Linear probing.
2. Quadratic probing.
3. Double hashing.

CHAINING IN HASHING

Let’s understand the chaining method with the help of one example.

Example: Use division method and closed addressing technique to store these values.

Given Keys=3, 2, 9, 6, 11, 13, 7, 12 and Bucket size is 10.

Solution:

$$h(k) = (2k + 3)\%m$$

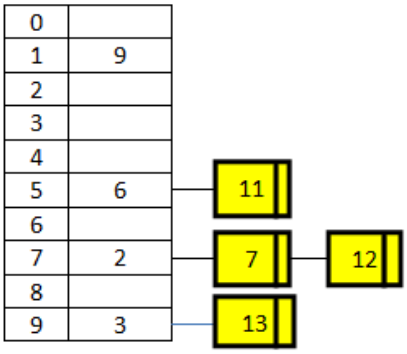
$$h(3) = (2 \times 3 + 3)\%10 = 9\%10 = 9$$

$$h(2) = (2 \times 2 + 3)\%10 = 7\%10 = 7$$

$$h(9) = (2 \times 9 + 3)\%10 = 21\%10 = 1$$

$$h(6) = (2 \times 6 + 3)\%10 = 15\%10 = 5$$

$$h(11) = (2 \times 11 + 3)\%10 = 25\%10 = 5$$



key 11 will be stored in 5th position, but we already have key 6 in 5th position. This is the case of collision. In closed addressing technique, we'll use chaining method, i.e. a linked list will be created and the key 11 will be stored there as shown in the hash table above

Languages like English or Spanish are not understood by computers, users must communicate with computers using a set of languages called programming languages. A computer can be programmed using a variety of language families. Computers are instruments created to address complicated issues, but only when a programming language and programmer are used. Computer software is what powers the various browsers, games, emails, operating systems, and applications. Any problem can be solved creatively through programming.

Computational language is used in **programming** to provide the computer with a set of instructions for a task. With the right instruction groups and specialized software, any complexity issue can be resolved. There are three fundamental ideas in computer programming design. They are repetition, selection, and sequence. The series is the first essential idea that tells you to execute the instructions in a specific order; choosing the right command comes next. The repetition of the same action, often known as iteration, is the fourth idea. In the creative process of programming, the programmer chooses the appropriate commands to address any issue.

Programming Languages

We require a variety of programming languages because no human language can be understood by computers. Every language has advantages and disadvantages, and some are more appropriate for a given task than others. Many diverse specialists, including software developers, computer system engineers, web designers, app developers, etc., require a programming language to do a

variety of jobs; there are many programming languages. More than 50 programming languages are used to perform different tasks and the most commonly used languages are HTML, JAVA, and C-language.

Computation Thinking

Using four fundamental patterns, computational thinking is a method for solving any problem. If we effectively comprehend and use the four fundamental patterns, computational thinking for programming becomes simple. The first step in effectively understanding an issue is to break it down into smaller components. We can more effectively use other computational thinking components when we divide the problems into smaller, more manageable pieces. The second component in this process is pattern recognition; the problems are reviewed to see if there is any sequence. If there are any patterns, they have been categorized appropriately. If no patterns are found, further simplification of that issue is not necessary. An abstraction or generalization of the issue serves as the third component. When you stand back from the specifics of a problem, you can develop a more general answer that can be useful in a number of different ways. The Algorithm, the fourth and final component, is where problems are incrementally addressed. Making a plan for your solution is crucial. A method for figuring out step-by-step directions on how to tackle any problem is to use an algorithm.

The purpose of **collision resolution** during insertion is to locate an open location in the hash table when the record's home position is already taken. Any collision resolution technique may be thought of as creating a series of hash table slots that may or may not contain the record. The key will be in its home position in the first position in the sequence. The collision resolution policy shifts to the following location in the sequence if the home position is already occupied. Another slot needs to be sought if this is also taken, and so on. The probe sequence is a collection of slots that is produced by a probe function that we will refer to as p. This is how insertion operates.

Collision in Hashing

In this, the hash function is used to find the index of the array. The hash value is used to create an index for the key in the hash table. The hash function may return the same hash value for two or

more keys. When two or more keys have the same hash value, a collision happens. To handle this collision, we use collision resolution techniques.

Collision Resolution Techniques

There are two types of collision resolution techniques.

- Separate chaining (open hashing)
- Open addressing (closed hashing)

Separate chaining: This method involves making a linked list out of the slot where the collision happened, then adding the new key to the list. Separate chaining is the term used to describe how this connected list of slots resembles a chain. It is more frequently utilized when we are unsure of the number of keys to add or remove.

Time complexity

- Its worst-case complexity for searching is o(n).
- Its worst-case complexity for deletion is o(n).

Advantages of separate chaining

- It is easy to implement.
- The hash table never fills full, so we can add more elements to the chain.
- It is less sensitive to the function of the hashing.

Disadvantages of separate chaining

- In this, the cache performance of chaining is not good.
- Memory wastage is too much in this method.
- It requires more space for element links.

Open addressing: To prevent collisions in the hashing table open, addressing is employed as a collision-resolution technique. No key is kept anywhere else besides the hash table. As a result, the hash table's size is never equal to or less than the number of keys. Additionally known as closed hashing.