

**SYNERGY INSTITUTE OF ENGINEERING &  
TECHNOLOGY**  
**Department of Computer Science & Engineering**  
**Academic Session 2023-24**  
**LECTURE NOTE**

<b>Name of Faculty</b>	<b>: Mrs. Ipsita Panda</b>
<b>Name of Subject</b>	<b>: Programming in C &amp; Data Structure</b>
<b>Subject Code</b>	<b>: 23ES1003 / 23ES1004</b>
<b>Subject Credit</b>	<b>: 3</b>
<b>Semester</b>	<b>: I / II</b>
<b>Year</b>	<b>: 1st</b>
<b>Course</b>	<b>: B. TECH</b>
<b>Branch</b>	<b>: All</b>
<b>Admission</b>	
<b>Batch</b>	<b>: 2023-24-2026</b>

## **Module-I**

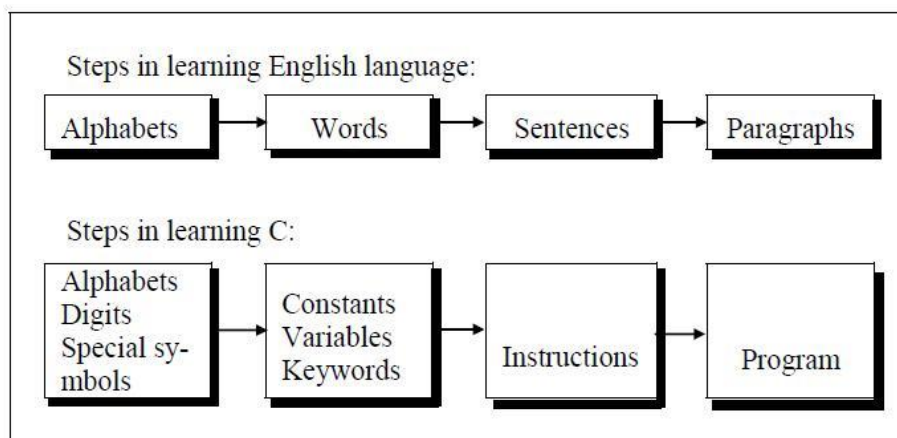
## Introduction to C

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc

ANSI C standard emerged in the early 1980s, this book was split into two titles: The original was still called Programming in C, and the title that covered ANSI C was called Programming in ANSI C. This was done because it took several years for the compiler vendors to release their ANSI C compilers and for them to become ubiquitous. It was initially designed for programming UNIX operating system. Now the software tool as well as the C compiler is written in C.

Major parts of popular operating systems like Windows, UNIX, Linux is still written in C. This is because even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system to work with new devices one needs to write device driver programs. These programs are exclusively written in C. C seems so popular is because it is reliable, simple and easy to use. often heard today is – “C has been already superseded by languages like C++, C# and Java.

There is a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables and keywords are constructed, and finally how are these combined to form an instruction. A group of instructions would be combined later on to form a program. So



a computer program is just a collection of the instructions necessary to solve a specific problem. The basic operations of a computer system form what is known as the computer's instruction set. And the approach or method that is used to solve the problem is known as an algorithm.

So far as programming language concern these are of two types.

- 1) Low level language
- 2) High level language

### **Structure of C Language program**

- 1 ) Comment line
- 2) Preprocessor directive
- 3 ) Global variable declaration
- 4) main function( )

```
{  
    Local variables;  
    Statements;  
  
}  
User defined function  
}  
}
```

### **Comment line**

It indicates the purpose of the program. It is represented as

```
/*.....*/
```

Comment line is used for increasing the readability of the program. It is useful in explaining the program and generally used for documentation. It is enclosed within the decimeters. Comment line can be single or multiple line but should not be nested. It can be anywhere in the program except inside string constant & character constant.

### **Preprocessor Directive:**

#include<stdio.h> tells the compiler to include information about the standard input/output library. It is also used in symbolic constant such as #define PI 3.14(value). The stdio.h (standard input output header file) contains definition & declaration of system defined function such as printf( ), scanf( ), pow( ) etc. Generally printf() function used to display and scanf() function used to read value

### **Global Declaration:**

This is the section where variable are declared globally so that it can be access by all the functions used in the program. And it is generally declared outside the function :

**main()**

It is the user defined function and every function has one main() function from where actually program is started and it is enclosed within the pair of curly braces.

The main( ) function can be anywhere in the program but in general practice it is placed in the first position.

Syntax :

```
main()  
{  
.....
```

```
.....  
}
```

The main( ) function return value when it declared by data type as

```
int main()  
{  
return 0  
}
```

The main function does not return any value when void (means null/empty) as

```
void main(void ) or void main()  
{  
printf ("C language");  
}
```

Output: C language

The program execution start with opening braces and end with closing brace. And in between the two braces declaration part as well as executable part is mentioned. And at the end of each line, the semi-colon is given which indicates statement termination.

**/\*First c program with return statement\*/**

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

```
int main (void)  
{  
printf ("welcome to c Programming language.\n");  
return 0;  
}
```

Output: welcome to c programming language.

## Steps for Compiling and executing the Programs

A compiler is a software program that analyzes a program developed in a particular computer language and then translates it into a form that is suitable for execution on a particular computer system. Figure below shows the steps that are involved in entering, compiling, and executing a computer program developed in the C programming language and the typical Unix commands that would be entered from the command line.

**Step 1:** The program that is to be compiled is first typed into a *file* on the computer system. There are various conventions that are used for naming files, typically be any name provided the last two characters are “.c” or file with extension .c. So, the file name **prog1.c** might be a valid filename for a C program. A text editor is usually used to enter the C program into a file. For example, vi is a popular text editor used on Unix systems. The program that is entered into the file is known as the *source program* because it represents the original form of the program expressed in the C language.

**Step 2:** After the source program has been entered into a file, then proceed to have it compiled. The compilation process is initiated by typing a special command on the system. When this command is entered, the name of the file that contains the source program must also be specified. For example, under Unix, the command to initiate program compilation is called **cc**. If we are using the popular GNU C compiler, the command we use is **gcc**.

Typing the line

gcc prog1.c or cc prog1.c

In the first step of the compilation process, the compiler examines each program statement contained in the source program and checks it to ensure that it conforms to the syntax and semantics of the language. If any mistakes are discovered by the compiler during this phase, they are reported to the user and the compilation process ends right there. The errors then have to be corrected in the source program (with the use of an editor), and the compilation process must be restarted. Typical errors reported during this phase of compilation might be due to an expression that has unbalanced parentheses (**syntactic error**), or due to the use of a variable that is not “defined” (**semantic error**).

**Step 3:** When all the syntactic and semantic errors have been removed from the program, the compiler then proceeds to take each statement of the program and translate it into a “lower” form that is equivalent to assembly language program needed to perform the identical task.

**Step 4:** After the program has been translated the next step in the compilation process is to translate the assembly language statements into actual machine instructions. The assembler takes each assembly language statement and converts it into a binary format known as *object code*, which is then written into another file on the system. This file has the same name as the source file under Unix, with the last letter an “o” (**for object**) instead of a “c”.

**Step 5:** After the program has been translated into object code, it is ready to be *linked*. This process is once again performed automatically whenever the cc or gcc command is issued under Unix. The purpose of the linking phase is to get the program into a final form for execution on the computer.

If the program uses other programs that were previously processed by the compiler, then during this phase the programs are linked together. Programs that are used from the system’s program *library* are also searched and linked together with the object program during this phase.

The process of compiling and linking a program is often called *building*.

The final linked file, which is in an executable *object* code format, is stored in another file on the system, ready to be run or *executed*. Under Unix, this file is called **a.out** by default. Under Windows, the executable file usually has the same name as the source file, with the c extension replaced by an exe extension.

## Character set

A character denotes any alphabet, digit or special symbol used to represent information. Valid alphabets, numbers and special symbols allowed in C are

Alphabets	A, B, ....., Y, Z a, b, ....., y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ' ! @ # % ^ & * ( ) _ - + =   \ { } [ ] : ; " ' < > , . ? /

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

## Identifiers

Identifiers are user defined word used to name of entities like variables, arrays, functions, structures etc. Rules for naming identifiers are:

- 1) name should only consists of alphabets (both upper and lower case), digits and underscore (\_) sign.
- 2) first characters should be alphabet or underscore
- 3) name should not be a keyword
- 4) since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.
- 5) identifiers are generally given in some meaningful name such as value, net\_salary, age, data etc. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters. ANSI standard compiler recognize 31 characters. Some invalid identifiers are 5cb, int, res#, avg no etc.

## Keyword

There are certain words reserved for doing specific task, these words are known as **reserved word** or **keywords**. These words are predefined and always written in lower case or small letter. These keywords can't be used as a variable name as it assigned with fixed meaning. Some examples are **int, short, signed, unsigned, default, volatile, float, long, double, break, continue, typedef, static,**



**do, for, union, return, while, do, extern, register, enum, case, goto, struct, char, auto, const etc.**

### **Data types**

Data types refer to an extensive system used for declaring variables or functions of different types before its use. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. The value of a variable can be changed any time.

C has the following 4 types of data types

**basic built-in data types:** int, float, double, char

**Enumeration data type:** enum

**Derived data type:** pointer, array, structure, union

**Void data type:** void

A variable declared to be of type int can be used to contain integral values only—that is, values that do not contain decimal places. A variable declared to be of type float can be used for storing floating-point numbers (values containing decimal places). The double type is the same as type float, only with roughly twice the precision. The char data type can be used to store a single character, such as the letter *a*, the digit character *6*, or a semicolon similarly. A variable declared char can only store character type value.

There are two types of type qualifier in c

**Size qualifier:** short, long

**Sign qualifier:** signed, unsigned

When the qualifier unsigned is used the number is always positive, and when signed is used number may be positive or negative. If the sign qualifier is not mentioned, then by default sign qualifier is assumed. The range of values for signed data types is less than that of unsigned data type. Because in signed type, the left most bit is used to represent sign, while in unsigned type this bit is also used to represent the value. The size and range of the different data types on a 16 bit machine is given below:

Basic data type	Data type with type qualifier	Size (byte)	Range
char	char or signed char	1	-128 to 127
	Unsigned char	1	0 to 255
int	int or signed int	2	-32768 to 32767
	unsigned int	2	0 to 65535
	short int or signed short int	1	-128 to 127
	unsigned short int	1	0 to 255
	long int or signed long int	4	-2147483648 to 2147483647
	unsigned long int	4	0 to 4294967295
float	float	4	-3.4E-38 to 3.4E+38
double	double	8	1.7E-308 to 1.7E+308
	Long double	10	3.4E-4932 to 1.1E+4932

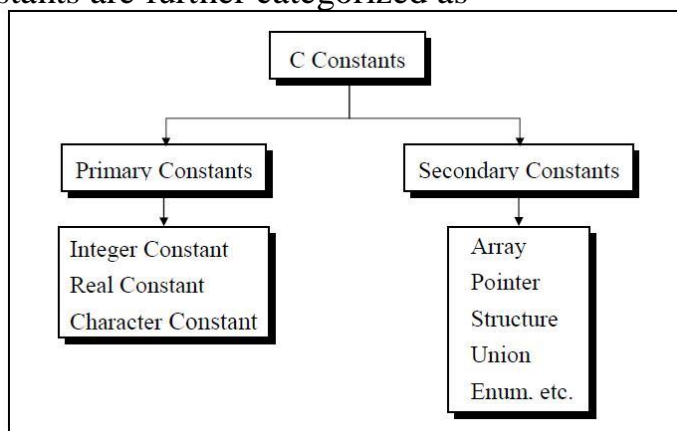
## Constants

Constant is a any value that cannot be changed during program execution. In C, any number, single character, or character string is known as a *constant*. A constant is an entity that doesn't change whereas a variable is an entity that may change. For example, the number 50 represents a constant integer value. The character string "Programming in C is fun.\n" is an example of a constant character string. C constants can be divided into two major categories:

Primary Constants

Secondary Constants

These constants are further categorized as



## **Numeric constant**

## **Character constant**

## **String constant**

**Numeric constant:** Numeric constant consists of digits. It required minimum size of 2 bytes and max 4 bytes. It may be positive or negative but by default sign is always positive. No comma or space is allowed within the numeric constant and it must have at least 1 digit. The allowable range for integer constants is -32768 to 32767. Truly speaking the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is -32768 to 32767. For a 32-bit compiler the range would be even greater. Mean by a 16- bit or a 32-bit compiler, what range of an Integer constant has to do with the type of compiler.

It is categorized a **integer constant** and **real constant**. An integer constants are whole number which have no decimal point. Types of integer constants are:

Decimal constant:        0----- 9(base 10)

Octal constant:            0----- 7(base 8)

Hexa decimal constant: 0----9, A-----F(base 16)

In decimal constant first digit should not be zero unlike octal constant first digit must be zero(as 076, 0127) and in hexadecimal constant first two digit should be 0x/ 0X (such as 0x24, 0x87A). By default type of integer constant is integer but if the value of integer constant is exceeds range then value represented by integer type is taken to be unsigned integer or long integer. It can also be explicitly mention integer and unsigned integer type by suffix l/L and u/U.

**Real constant** is also called floating point constant. To construct real constant we must follow the rule of ,

- real constant must have at least one digit.

- It must have a decimal point.

- It could be either positive or negative.

- Default sign is positive.

- No commas or blanks are allowed within a real constant. Ex.: +325.34

426.0

-32.76

To express small/large real constant exponent(scientific) form is used where number is written in mantissa and exponent form separated by e/E. Exponent can be positive or negative integer but mantissa can be real/integer type, for example  $3.6 \times 10^5 = 3.6e+5$ . By default type of floating point constant is double, it can also be explicitly defined it by suffix of f/F.

## Character constant

Character constant represented as a single character enclosed within a single quote. These can be single digit, single special symbol or white spaces such as '9','c','\$', ' ' etc. Every character constant has a unique integer like value in machine's character code as if machine using ASCII (American standard code for information interchange). Some numeric value associated with each upper and lower case alphabets and decimal integers are as:

A----- Z ASCII value (65-90)

a-----z ASCII value (97-122)

0----- 9 ASCII value (48-59)

; ASCII value (59)

## String constant

Set of characters are called string and when sequence of characters are enclosed within a double quote (it may be combination of all kind of symbols) is a string constant. String constant has zero, one or more than one character and at the end of the string null character(\0) is automatically placed by compiler. Some examples are “,sarathina” , “908”, “3”, ” ”, “A” etc. In C although same characters are enclosed within single and double quotes it represents different meaning such as “A” and ‘A’ are different because first one is string attached with null character at the end but second one is character constant with its corresponding ASCII value is 65.

## Symbolic constant

Symbolic constant is a name that substitute for a sequence of characters and, characters may be numeric, character or string constant. These constant are generally defined at the beginning of the program as

#define name value , here name generally written in upper case for example

```
#define MAX 10
#define CH 'b'
#define NAME “sony”
```

## Variables

Variable is a data name which is used to store some data value or symbolic names for storing program computations and results. The value of the variable can be change during the execution. The rule for naming the variables is same as the naming identifier. Before used in the program it must be declared. Declaration of variables specify its name, data types and range of the value that variables can store depends upon its data types.

Syntax:

```
int a;  
char c;  
float f;
```

Variable initialization

When we assign any initial value to variable during the declaration, is called initialization of variables. When variable is declared but contain undefined value then it is called garbage value. The variable is initialized with the assignment operator such as

Data type variable name=constant;

Example: int a=20;

Or int a;

a=20;

## **Module-II**

## Control Statement

Generally C program statement is executed in a order in which they appear in the program. But sometimes we use decision making condition for execution only a part of program, that is called control statement. Control statement defined how the control is transferred from one part to the other part of the program. There are several control statement like if...else, switch, while, do....while, for loop, break, continue, goto etc.

## Loops in C

Loop:-it is a block of statement that performs set of instructions. In loops Repeating particular portion of the program either a specified number of time or until a particular no of condition is being satisfied.

There are three types of loops in c

**1.While loop**

**2.do while loop**

**3.for loop**

### While loop

Syntax:-

```
while(condition)
```

```
{
```

```
Statement 1;
```

```
Statement 2;
```

```
}
```

Or     while(test condition)

        Statement;

The test condition may be any expression .when we want to do something a fixed no of times but not known about the number of iteration, in a program then while loop is used.

Here first condition is checked if, it is true body of the loop is executed else, If condition is false control will be come out of loop.

Example:-

```
/* wap to print 5 times welcome to C */
```

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

```
void main()
```

```
{
```

```
int p=1;
```

```
While(p<=5)
```

```
{
```

```
printf("Welcome to C\n");
p=p+1;
}
}
```

Output: Welcome to C  
Welcome to C  
Welcome to C  
Welcome to C  
Welcome to C

So as long as condition remains true statements within the body of while loop will get executed repeatedly.

### **do while loop**

This (do while loop) statement is also used for looping. The body of this loop may contain single statement or block of statement. The syntax for writing this statement is:

Syntax:-

```
do
{
Statement;
}while(condition);
```

Example:-

```
#include<stdio.h>
void main()
{
int X=4;
do
{
Printf("%d",X);

X=X+1;

}while(X<=10);

Printf(" ");
```



```
}
```

Output: 4 5 6 7 8 9 10

### **for loop**

In a program, for loop is generally used when number of iteration are known in advance. The body of the loop can be single statement or multiple statements. Its syntax for writing is:

Syntax:-

```
for(exp1;exp2;exp3)
{
Statement;
}
```

Or

```
for(initialized counter; test counter; update counter)
{
Statement;
}
```

Here exp1 is an initialization expression, exp2 is test expression or condition and exp3 is an update expression. Expression 1 is executed only once when loop started and used to initialize the loop variables. Condition expression generally uses relational and logical operators. And updation part executed only when after body of the loop is executed.

Example:-

```
void main()
{
int i;
for(i=1;i<10;i++)
{
Printf(“ %d ”, i);
}
}
```

Output:-1 2 3 4 5 6 7 8 9

### **Nesting of loop**

When a loop written inside the body of another loop then, it is known as nesting of loop. Any type of loop can be nested in any type such as while, do while, for. For example nesting of for loop can be represented as :

```

void main()
{
int i,j;
for(i=0;i<2;i++)
    for(j=0;j<5; j++)
        printf("%d %d", i, j);
}

```

Output: i=0

```

j=0 1 2 3 4
i=1
j=0 1 2 3 4

```

### **Break statement(break)**

Sometimes it becomes necessary to come out of the loop even before loop condition becomes false then break statement is used. Break statement is used inside loop and switch statements. It cause immediate exit from that loop in which it appears and it is generally written with condition. It is written with the keyword as **break**. When break statement is encountered loop is terminated and control is transferred to the statement, immediately after loop or situation where we want to jump out of the loop instantly without waiting to get back to conditional state.

When break is encountered inside any loop, control automatically passes to the first statement after the loop. This break statement is usually associated with **if** statement.

#### **Example :**

```

void main()
{
int j=0;
for(;j<6;j++)
    if(j==4)
        break;
}

```

Output:        0123

### **Continue statement (key word continue)**

Continue statement is used for continuing next iteration of loop after skipping some statement of loop. When it encountered control automatically passes through

the beginning of the loop. It is usually associated with the if statement. It is useful when we want to continue the program without executing any part of the program.

The difference between break and continue is, when the break encountered loop is terminated and it transfer to the next statement and when continue is encounter control come back to the beginning position.

In while and do while loop after continue statement control transfer to the test condition and then loop continue where as in, for loop after continue control transferred to the updating expression and condition is tested.

### **Example:-**

```
void main()
{
int n;
for(n=2; n<=9; n++)
{
if(n==4)
continue;
printf("%d", n);
}
}
Printf("out of loop");
}
```

Output: 2 3 5 6 7 8 9 out of loop

### **if statement**

Statement execute set of command like when condition is true and its syntax is

```
if (condition)
    Statement;
```

The statement is executed only when condition is true. If the if statement body is consists of several statement then better to use pair of curly braces. Here in case condition is false then compiler skip the line within the if block.

```
void main()
{
    int n;
    printf (" enter a number:");
    scanf("%d",&n);
    If (n>10)
        Printf(" number is grater");
}
```

```
}
```

Output:

```
Enter a number:12
Number is greater
```

### **if.....else ... Statement**

it is bidirectional conditional control statement that contains one condition & two possible action. Condition may be true or false, where non-zero value regarded as true & zero value regarded as false. If condition are satisfy true, then a single or block of statement executed otherwise another single or block of statement is executed.

Its syntax is:-

```
if (condition)
{
Statement1;
Statement2;
}
else
{
Statement1;
Statement2;
}
```

Else statement cannot be used without if or no multiple else statement are allowed within one if statement. It means there must be a if statement with in an else statement.

Example:-

```
/* To check a number is eve or odd */
```

```

void main()
{
    int n;
    printf ("enter a number:");
    scanf ("%d", &n);
    if (n%2==0)
        printf ("even number");
    else
        printf ("odd number");
}

```

Output: enter a number:121  
odd number

### **Nesting of if ...else**

When there are another if else statement in if-block or else-block, then it is called nesting of if-else statement.

Syntax is :-

```

    if (condition)
    {
        if (condition)
            Statement1;
        else
            statement2;
    }
    Statement3;

```

### **If....else LADDER**

In this type of nesting there is an if else statement in every else part except the last part. If condition is false control pass to block where condition is again checked with its if statement.

Syntax is :-

```

    if (condition)
        Statement1;
    else if (condition)
        statement2;
    else if (condition)
        statement3;
    else
        statement4;

```

This process continue until there is no if statement in the last block. if one of the condition is satisfy the condition other nested “else if” would not executed.

## **ARRAY**

Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. Array of character is a string. Each data item of an array is called an element. And each element is unique and located in separated memory location. Each of elements of an array share a variable but each element having different index no. known as subscript.

An array can be a single dimensional or multi-dimensional and number of subscripts determines its dimension. And number of subscript is always starts with zero. One dimensional array is known as vector and two dimensional arrays are known as matrix.

**ADVANTAGES:** array variable can store more than one value at a time where other variable can store one value at a time.

Example:

```
int arr[100];  
int mark[100];
```

### **DECLARATION OF AN ARRAY :**

Its syntax is :

```
Data type array name [size];  
int arr[100];  
int mark[100];  
int a[5]={ 10,20,30,100,5}
```

The declaration of an array tells the compiler that, the data type, name of the array, size of the array and for each element it occupies memory space. Like for int data type, it occupies 2 bytes for each element and for float it occupies 4 byte

for each element etc. The size of the array operates the number of elements that can be stored in an array and it may be a int constant or constant int expression.

We can represent individual array as :

```
int ar[5];  
  
ar[0], ar[1], ar[2], ar[3], ar[4];
```

Symbolic constant can also be used to specify the size of the array as:

```
#define SIZE 10;
```

## **INITIALIZATION OF AN ARRAY:**

After declaration element of local array has garbage value. If it is global or static array then it will be automatically initialize with zero. An explicitly it can be initialize that

```
Data type array name [size] = {value1, value2, value3...}
```

Example:

```
in ar[5]={20,60,90, 100,120}
```

Array subscript always start from zero which is known as lower bound and upper value is known as upper bound and the last subscript value is one less than the size of array. Subscript can be an expression i.e. integer value. It can be any integer, integer constant, integer variable, integer expression or return value from functional call that yield integer value.

So if i & j are not variable then the valid subscript are

```
ar [i*7],ar[i*i],ar[i++],ar[3];
```

The array elements are standing in continuous memory locations and the amount of storage required for hold the element depend in its size & type.

### **Total size in byte for 1D array is:**

Total bytes=size of (data type) \* size of array.

Example : if an array declared is:

```
int [20];
```

Total byte= 2 \* 20 =40 byte.

### **ACCESSING OF ARRAY ELEMENT:**

/\*Write a program to input values into an array and display them\*/

```
#include<stdio.h>
int main()
{
int arr[5],i;
for(i=0;i<5;i++)
{
printf("enter a value for arr[%d] \n",i);
scanf("%d",&arr[i]);
}
printf("the array elements are: \n");

for (i=0;i<5;i++)
{
printf("%d\t",arr[i]);
}
return 0;
}
```

### **OUTPUT:**

Enter a value for arr[0] = 12

Enter a value for arr[1] =45



Enter a value for arr[2] =59  
Enter a value for arr[3] =98  
Enter a value for arr[4] =21  
The array elements are 12 45 59 98 21

Example: From the above example value stored in an array are and occupy its memory addresses 2000, 2002, 2004, 2006, 2008 respectively.

a[0]=12, a[1]=45, a[2]=59, a[3]=98, a[4]=21

ar[0]	ar[1]	ar[2]	ar[3]	ar[4]
12	45	59	98	21
2000	2002	2004	2006	2008

## Single dimensional arrays and functions

/\*program to pass array elements to a function\*/

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

```
void main()
```

```
{
```

```
int arr[10],i;
```

```
printf("enter the array elements\n");
```

```
for(i=0;i<10;i++)
```

```
{
```

```
scanf("%d",&arr[i]);
```

```
check(arr[i]);
```

```
}
```

```
}
```

```
void check(int num)
```

```
{
```

```
if(num%2==0)
{
printf("%d is even \n",num);
}
else
{
printf("%d is odd \n",num);
}
}
```

## **Two dimensional arrays**

Two dimensional array is known as matrix. The array declaration in both the array i.e.in single dimensional array single subscript is used and in two dimensional array two subscripts are is used.

Its syntax is

Data-type array name[row][column];

Or we can say 2-d array is a collection of 1-D array placed one below the other.

Total no. of elements in 2-D array is calculated as **row\*column**

Example:-

```
int a[2][3];
```

Total no of elements=row\*column is  $2*3=6$

It means the matrix consist of 2 rows and 3 columns

For example:-

20    2    7

8    3    15

Positions of 2-D array elements in an array are as below

00    01    02

10    11    12

a [0][0]    a [0][0]    a [0][0]    a [0][0]    a [0][0]    a [0][0]

20	2	7	8	3	15
2000	2002	2004	2006	2008	

## **Accessing 2-d array /processing 2-d arrays**

For processing 2-d array, we use two nested for loops. The outer for loop corresponds to the row and the inner for loop corresponds to the column.

For example

```
int a[4][5];
```

**for reading value:-**

```
for(i=0;i<4;i++)
{
    for(j=0;j<5;j++)
    {
        scanf("%d",&a[i][j]);
    }
}
```

For displaying value:-

```
for(i=0;i<4;i++)
{
for(j=0;j<5;j++)
{
printf("%d",a[i][j]);
}
}
```

### **Initialization of 2-d array:**

2-D array can be initialized in a way similar to that of 1-D array. for example:-  
int mat[4][3]={ 11,12,13,14,15,16,17,18,19,20,21,22};

These values are assigned to the elements row wise, so the values of elements after this initialization are

Mat[0][0]=11,      Mat[1][0]=14,      Mat[2][0]=17      Mat[3][0]=20

Mat[0][1]=12,   Mat[1][1]=15, Mat[2][1]=18      Mat[3][1]=21

Mat[0][2]=13,      Mat[1][2]=16,      Mat[2][2]=19      Mat[3][2]=22

While initializing we can group the elements row wise using inner braces. for example:-

```
int mat[4][3]={ { 11,12,13},{ 14,15,16},{ 17,18,19},{ 20,21,22}};
```

And while initializing , it is necessary to mention the 2<sup>nd</sup> dimension where 1<sup>st</sup> dimension is optional.

```
int mat[][3];
```

```
int mat[2][3];
```

```
int mat[][]; }      invalid  
int mat[2][];
```

If we **initialize an array** as

```
int mat[4][3]={ { 11},{ 12,13},{ 14,15,16},{ 17}};
```

Then the compiler will assume its all rest value as 0, which are not defined.

```
Mat[0][0]=11, Mat[1][0]=12, Mat[2][0]=14, Mat[3][0]=17
```

```
Mat[0][1]=0,      Mat[1][1]=13,      Mat[2][1]=15      Mat[3][1]=0
```

```
Mat[0][2]=0,      Mat[1][2]=0,      Mat[2][2]=16, Mat[3][2]=0
```

In memory map whether it is 1-D or 2-D, elements are stored in one contiguous manner.

We can also give the size of the 2-D array by using symbolic

constant Such as

```
#define ROW 2;
```

```
#define COLUMN 3;

int mat[ROW][COLUMN];
```

## String

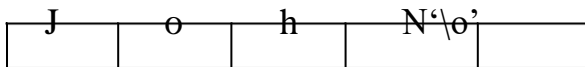
Array of character is called a string. It is always terminated by the NULL character. String is a one dimensional array of character.

We can initialize the string as

```
char name[]={ 'j', 'o', 'h', 'n', '\0' };
```

Here each character occupies 1 byte of memory and last character is always NULL character. Where '\0' and 0 (zero) are not same, where **ASCII** value of '\0' is 0 and ASCII value of 0 is 48. Array elements of character array are also stored in contiguous memory allocation.

From the above we can represent as;



The terminating NULL is important because it is only the way that the function that work with string can know, where string end.

String can also be **initialized** as;

```
char name[]="John";
```

Here the NULL character is not necessary and the compiler will assume it automatically.

## String constant (string literal)

A string constant is a set of character that enclosed within the double quotes and is also called a literal. Whenever a string constant is written anywhere in a program it is stored somewhere in a memory as an array of characters terminated by a NULL character ('\0').

Example – “m”


“Tajmahal”

“My age is %d and height is %f\n”

The string constant itself becomes a pointer to the first character in array.

Example-char arr[20]="Taj mahal";

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
T	a	j		M	A	H	a	l	\0



It is called base address.



### 1. strlen()

#### **C Program To Count Length Of String Using Library Function**

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

void main() {
    char str[100];
    int len;

    printf("\nEnter the String : ");
    scanf("%s",str);

    /*
        strlen() is the pre-defined function
        to find the length of a string
    */
    len = strlen(str);

    printf("\nLength of the given string is %d", len);
}
```

### 2. strcpy()

#### **C Program To Copy one String to another Using Library Function**

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

void main()
{
    char string1[]="Hello";
    char string2[]="World";

    printf("String-1 before strcpy() function : %s",string1);
    strcpy(string1,string2);
    printf("\nString-1 after strcpy() function : %s",string1);
}
```

### 3. strcat()

#### **C Program for concatenation of two String Using Library Function**

```

#include <stdio.h>
#include<string.h>
void main()
{
    //Declaring the value of str1 and str2
    char str1[50] = "Coding";
    char str2[50] = "Ninjas";
    strcat (str1, str2); //performing the function
    printf("%s", str1); //printing the value of str1
}

```

#### 4. strcmp()

#### **C Program for comparing two String Using Library Function**

```

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

void main()
{
    char string1[]="Ninja";
    char string2[]="Ninja";

    int val=strcmp(string1,string2);
    printf("On comparing %s and %s the value returned is %d",string1,string2,val);
}

```



## **Structure**

It is the collection of dissimilar data types or heterogenous data types grouped together. It means the data types may or may not be of same type.

### **Structure declaration-**

struct tagname

```
{  
Data type member1;  
Data type member2;  
Data type member3;  
.....
```

```
Data type member n;  
};
```

### **Structure variable declaration;**

struct student

```
{  
    int age;  
    char name[20];  
    char branch[20];  
}; struct student s;
```

### **Initialization of structure variable-**

Like primary variables structure variables can also be initialized when they are declared. Structure templates can be defined locally or globally. If it is local it can be used within that function. If it is global it can be used by all other functions of the program.

A structure can be initialized as

```
struct student
{
    int age,roll;
    char name[20];
} struct student s1={16,101,"sona"};
    struct student s2={17,102,"rupa"};
```

If initialiser is less than no.of structure variable, automatically rest values are taken as zero

### **Accessing structure elements-**

Dot operator is used to access the structure elements. Its associativity is from left to right.

```
structure variable ;
s1.name[];
s1.roll;
s1.age;
```

Elements of structure are stored in contiguous memory locations. Value of structure variable can be assigned to another structure variable of same type using assignment operator.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int roll, age;
    char branch;
} s1,s2;
```

```
printf("\n enter roll, age, branch=");  
scanf("%d %d %c", &s1.roll, &s1.age, &s1.branch);  
s2.roll=s1.roll;  
printf(" students details=\n");  
printf("%d %d %c", s1.roll, s1.age, s1.branch);  
printf("%d", s2.roll);  
}
```

### **Size of structure-**

Size of structure can be found out using sizeof() operator with structure variable name or tag name with keyword.

```
sizeof(struct student); or  
sizeof(s1);  
sizeof(s2);
```

Size of structure is different in different machines. So size of whole structure may not be equal to sum of size of its members.

### **Array of structures**

When database of any element is used in huge amount, we prefer Array of structures.

Example: suppose we want to maintain data base of 200 students, Array of structures is used.

```
#include<stdio.h>  
#include<string.h>  
struct student  
{
```

```
char name[30];
char branch[25];
int roll;
};

void main()
{
    struct student s[200];
    int i;
    s[i].roll=i+1;
    printf("\nEnter information of students:");
    for(i=0;i<200;i++)
    {
        printf("\nEnter the roll no:%d\n",s[i].roll);
        printf("\nEnter the name:");
        scanf("%s",s[i].name); printf("\nEnter the
        branch:"); scanf("%s",s[i].branch); printf("\n");
    }
    printf("\nDisplaying information of students:\n\n");
    for(i=0;i<200;i++)
    {
        printf("\n\nInformation for roll no%d:\n",i+1);

        printf("\nName:");
        puts(s[i].name);
        printf("\nBranch:");
        puts(s[i].branch);
    }
}
```

## **Nested structure**

When a structure is within another structure, it is called Nested structure. A structure variable can be a member of another structure and it is represented as

```
struct student
```

```
{  
element 1;  
element 2;  
.....
```

```
struct student1
```

```
{  
member 1;  
member 2;  
}variable 1;  
.....
```

```
.....
```

```
element n;  
}variable 2;
```

It is possible to define structure outside & declare its variable inside other structure.

```
struct date
```

```
{  
int date,month;  
};
```

```
struct student
```



```

{
    char nm[20];
    int roll;
    struct date d;
}; struct student s1;
    struct student s2,s3;

```

Nested structure may also be initialized at the time of declaration like in above example.

```

struct student s={"name",200, {date, month}};
                {"ram",201, {12,11}};

```

### **Passing entire structure to function**

```

#include<stdio.h>
#include<string.h>
struct student
{
    char name[30];
    int age,roll;
};
display(struct student);
void main()
{
    struct student s1={"sona",16,101 };
    struct student s2={"rupa",17,102 };
    display(s1);
    display(s2);
}
display(struct student s)
{
    printf("\n name=%s, \n age=%d ,\n roll=%d", s.name, s.age, s.roll);}

```

## UNION

**Union** is derived data type contains collection of different data type or dissimilar elements. All definition declaration of union variable and accessing member is similar to structure, but instead of keyword struct the keyword union is used, the main difference between union and structure is each member of structure occupy the memory location, but in the unions members share memory. Union is used for saving memory and concept is useful when it is not necessary to use all members of union at a time.

Where union offers a memory treated as variable of one type on one occasion where (struct), it read number of different variables stored at different place of memory.

### **Syntax of union:**

```
union student
{
datatype member1;
datatype member2;
};
```

Like structure variable, union variable can be declared with definition or separately such as

```
union union name
{
Datatype member1;
}var1;
```

Example:- union student s;

Union members can also be accessed by the dot operator with union variable and if we have pointer to union then member can be accessed by using (arrow) operator as with structure.

Example:- struct student

```
struct student
{
int i;
char ch[10];
};struct student s;
```

Here datatype/member structure occupy 12 byte of location is memory, where as in the union side it occupy only 10 byte.

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

## **Module-III**

# 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.*

## Syntax of C Pointers

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

```
datatype * ptr;
```

## Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the ( \* ) **dereference operator** before its name.

### Example

```
int *ptr;
```

## Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the ( & ) **addressof operator** to get the memory address of a variable and then store it in the pointer variable.

### Example

```
int var = 10;
int * ptr;
ptr = &var;
```

```
/ C program to illustrate Pointers
#include <stdio.h>

void geeks()
{
    int var = 10;
    int* ptr;
    ptr = &var;
    printf("Value at ptr = %p \n", ptr);
    printf("Value at var = %d \n", var);
    printf("Value at *ptr = %d \n", *ptr);
}

int main()
{
    geeks();
    return 0;
}
```

## C program to add two number using pointer

```
#include <stdio.h>
int main()
{
    int first, second, *p, *q, sum;
    printf("Enter two integers to add\n");
    scanf("%d%d", &first, &second);
    p = &first;
    q = &second;
    sum = *p + *q;
    printf("Sum of the numbers = %d\n", sum);
    return 0;
}
```

## C Example to swap two numbers using pointers

```
#include <stdio.h>

void swap(int *x,int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}

int main()
{
    int num1,num2;

    printf("Enter value of num1: ");
    scanf("%d",&num1);
    printf("Enter value of num2: ");
    scanf("%d",&num2);

    printf("Before Swapping: num1 is: %d, num2 is: %d\n",num1,num2);

    swap(&num1,&num2);

    printf("After Swapping: num1 is: %d, num2 is: %d\n",num1,num2);

    return 0;
}
```

## Dynamic memory Allocation

- The process of allocating memory at the time of execution or at the runtime, is called dynamic memory location.
- Two types of problem may occur in static memory allocation.
- If number of values to be stored is less than the size of memory, there would be wastage of memory.
- If we would want to store more values by increase in size during the execution on assigned size then it fails.
- Allocation and release of memory space can be done with the help of some library function called dynamic memory allocation function. These library function are called as **dynamic memory allocation function**. These library function prototype are found in the header file, "alloc.h" where it has defined.
- Function take memory from memory area is called heap and release when not required.
- Pointer has important role in the dynamic memory allocation to allocate memory.

### malloc():

This function use to allocate memory during run time, its declaration is

```
void*malloc(size);
```

**malloc ()**

returns the pointer to the 1st byte and allocate memory chunk that was allocated and it returns null on unsuccessful and from the above declaration a pointer of type(**datatype**) and size in byte.

And **datatype** pointer used to typecast the pointer returned by malloc and this typecasting is necessary since, malloc() by default returns a pointer to void.

Example `int *p=(int*)malloc(10);`

So, from the above pointer p, allocated IO contiguous memory space address of 1st byte and is stored in the variable.

We can also use, the size of operator to specify the the size, such as

`*p=(int*)malloc(5*size of int)` Here, 5 is the no. of data.

Moreover , it returns null, if no sufficient memory available , we should always check the malloc return such as,

```
if(p==null)
```

```
printf("not sufficient memory");
```

### Example:

```
// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>
#include <stdlib.h>
int main() {
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
```

```

    scanf("%d", &n);
ptr = (int*) malloc(n * sizeof(int));
if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
}
printf("Enter elements: ");
for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
}
printf("Sum = %d", sum);
free(ptr);
return 0;
}

```

### **calloc()**

Similar to malloc only difference is that calloc function use to allocate multiple block of memory .

two arguments are there

1st argument specify number of blocks

2nd argument specify size of each block.

Example:-

```

int *p= (int*) calloc(5, 2);
int*p=(int *)calloc(5, size of (int));

```

Another difference between malloc and calloc is by default memory allocated by malloc contains garbage value, where as memory allocated by calloc is initialised by zero(but this initialisation) is not reliable.

### **realloc()**

The function realloc use to change the size of the memory block and it alter the size of the memory block without loosing the old data, it is called reallocation of memory.

It takes two argument such as;

```

int *ptr=(int *)malloc(size);
int*p=(int *)realloc(ptr, new size);

```

The new size allocated may be larger or smaller.

If new size is larger than the old size, then old data is not lost and newly allocated bytes are uninitialized. If old address is not sufficient then starting address contained in pointer may be changed and this reallocation function moves content of old block into the new block and data on the old block is not lost.



**Example:**

```

#include<stdio.h>
#include<alloc.h>
void main()
int i,*p;
p=(int*)malloc(5*size of (int));
if(p==null)
{
printf("space not available");
exit();
printf("enter 5 integer");
for(i=0;i<5;i++)
{
scanf("%d",(p+i));
int*ptr=(int*)realloc(9*size of (int) );
if(ptr==null)
{
printf("not available");
exit();
}
printf("enter 4 more integer");
for(i=5;i<9;i++)
scanf("%d",(p+i));
for(i=0;i<9;i++)
printf("%d",*(p+i));
}

```

**free()**

Function free() is used to release space allocated dynamically, the memory released by free() is made available to heap again. It can be used for further purpose.

Syntax for free declaration .

```
void(*ptr)
```

Or

**free(p)**

When program is terminated, memory released automatically by the operating system. Even we don't free the memory, it doesn't give error, thus lead to memory leak.

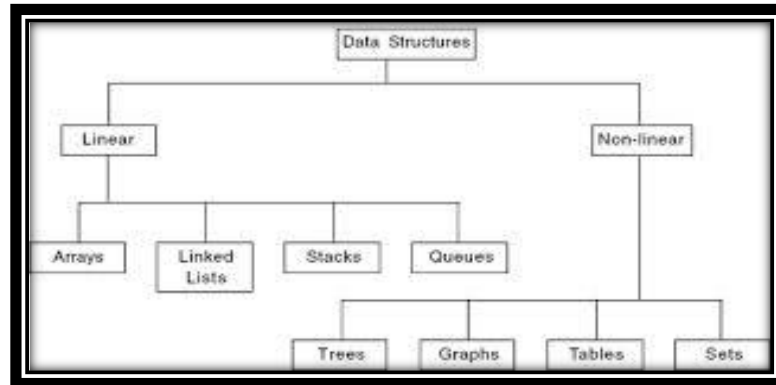
We can't free the memory, those didn't allocated.

<b>Malloc()</b>	<b>Calloc()</b>
The name malloc stands for memory allocation	The name calloc stands for contiguous allocation
It is used to dynamically allocate a single large block of memory with the specified size	It is used to dynamically allocate the specified number of blocks of memory of specified type
Syntax : ptr = (data_type*) malloc(byte-size)	Syntax : ptr =(float*) calloc (n, element-size);
malloc() takes one argument that is number of bytes	Calloc() takes two arguments those are : number of blocks and size of each block

## **Module-IV**

## Introduction to data structures

- In computer science, a **data structure** is a particular way of organizing **data** in a computer so that it can be used efficiently.
- Different types of Data Structure.



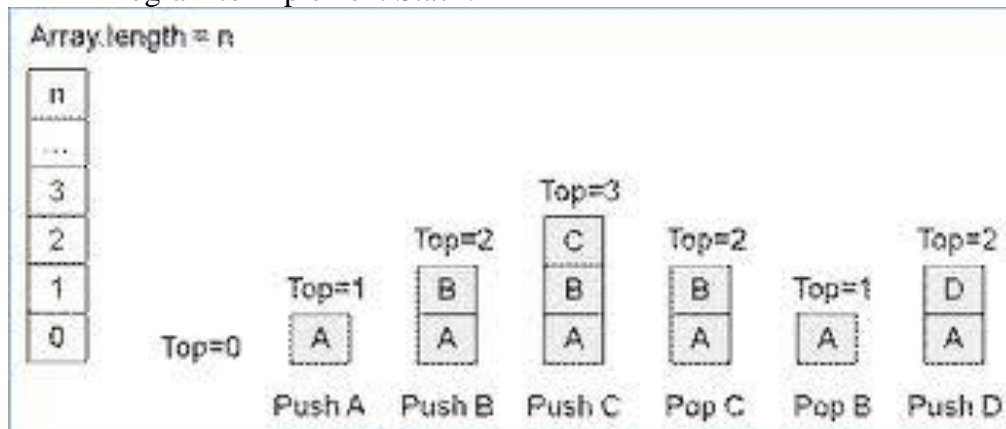
- **Linear data structure:** A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached. Ex: Arrays, Linked Lists
- **Non-Linear data structure:** Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure. Ex: Trees, Graphs.
- An **abstract data type (ADT)** is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics.
- **Introduction to Algorithm and efficiency**
  - Algorithm is a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.
  - Algorithmic **efficiency** is the properties of an **algorithm** which relate to the amount of resources used by the **algorithm**. An **algorithm** must be analyzed to determine its resource usage.
  - The **time complexity** of an algorithm quantifies the amount of **time** taken by an algorithm to run as a function of the length of the string representing the input.
  - *Space Complexity* of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

## Stacks

- What is STACK?
  - The relation between the push and pop operations is such that the **stack** is a Last-In-First-Out (LIFO) data structure. In a LIFO data structure, the last element added to the structure must be the first one to be removed.

### Array representation of stack.

- Operation performed on Stack
  - PUSH Operation with algorithm
  - POP Operation with algorithm
- Program to implement Stack.



### **Example :**

1. **Practical daily life :** a pile of heavy books kept in a vertical box, dishes kept one on top of another
2. **In computer world :** In processing of subroutine calls and returns ; there is an explicit use of stack of return addresses.  
Also in evaluation of arithmetic expressions , stack is used.

### Algorithms

#### **Push (item,array , n, top)**

```
{  
If ( n >= top)  
Then print "Stack is full" ;  
Else  
top = top + 1;  
array[top] = item ;  
}
```

#### **Pop (item, array, top)**

```
{  
if ( top < 0)  
Then print "stack is empty".  
Else  
item = array[top];  
top = top - 1;  
}
```

## Infix to post fix conversion

### **RULES FOR EVALUATION OF ANY EXPRESSION:**

An expression can be interpreted in many different ways if parentheses are not mentioned in the expression.

- For example the below given expression can be interpreted in many different ways:
- Hence we specify some basic rules for evaluation of any expression :

**A priority table is specified for the various types of operators being used:**

PRIORITY LEVEL	OPERATORS
6	** ; unary - ; unary +
5	* ; /
4	+ ; -
3	< ; > ; <= ; >= ; !> ; !< ; !=
2	Logical and operation
1	Logical or operation

### Infix to postfix conversion algorithm

There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

In this algorithm, all operands are printed (or sent to output) when they are read. There are more complicated rules to handle operators and parentheses.

Example:

1.  $A * B + C$  becomes  $A B * C +$

The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '\*', so the '\*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6			A B * C +

The rule used in lines 1, 3 and 5 is to print an operand when it is read. The rule for line 2 is to push an operator onto the stack if it is empty. The rule for line 4 is if the operator on the top of the stack has higher precedence than the one being read, pop and print the one on top and then push the new operator on. The rule for line 6 is that when the end of the expression has been reached, pop the operators on the stack one at a time and print them.

2.  $A + B * C$  becomes  $A B C * +$

Here the order of the operators must be reversed. The stack is suitable for this, since operators will be popped off in the reverse order from that in which they were pushed.

	current symbol	operator stack	postfix string
1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

In line 4, the '\*' sign is pushed onto the stack because it has higher precedence than the '+' sign which is already there. Then when they are both popped off in lines 6 and 7, their order will be reversed.

3.  $A * (B + C)$  becomes  $A B C + *$

A subexpression in parentheses must be done before the rest of the expression.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A B
4	B	* (	A B
5	+	* ( +	A B
6	C	* ( +	A B C
7	)	*	A B C +
8			A B C + *

Since expressions in parentheses must be done first, everything on the stack is saved and the left parenthesis is pushed to provide a marker. When the next operator is read, the stack is treated as though it were empty and the new operator (here the '+' sign) is pushed on. Then when the right parenthesis is read, the stack is popped until the corresponding left parenthesis is found. Since postfix expressions have no parentheses, the parentheses are not printed.

4.  $A - B + C$  becomes  $A B - C +$

When operators have the same precedence, we must consider association. Left to right association means that the operator on the stack must be done first, while right to left association means the reverse.

	current symbol	operator stack	postfix string
1	A		A
2	-	-	A
3	B	-	A B
4	+	+	A B -
5	C	+	A B - C
6			A B - C +

In line 4, the '-' will be popped and printed before the '+' is pushed onto the stack. Both operators have the same precedence level, so left to right association tells us to do the first one found before the second.

5.  $A * B ^ C + D$  becomes  $A B C ^ * D +$

Here both the exponentiation and the multiplication must be done before the addition.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	^	* ^	A B
5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D
8			A B C ^ * D +

When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack. Since it has lower precedence, the '^' is popped and printed. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '\*'. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

6.  $A * (B + C * D) + E$  becomes  $A B C D * + * E +$

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* ( +	A B
6	C	* ( +	A B C
7	*	* ( + *	A B C
8	D	* ( + *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

**Example:  $4 + 7 - 5 / 6$**

Scanned element	Stack	Postfix notation
4	-	4
+	+	4
7	+	4 7
-	-	4 7 +
5	-	4 7 + 5
/	- /	4 7 + 5
6	- /	4 7 + 5 6

### Infix to post fix conversion more examples

- Discussion on example of Infix to post fix conversion
  - Infix to post fix conversion more examples?
    1.  $300+23*(43-21)/(84+7)$
    2.  $(4+8)*(6-5)/((3-2)*(2+2))$
    3.  $(a+b)*(c+d)$
    4.  $a\%(c-d)+b*e$
    5.  $a-(b+c)*d/e$

### Postfix expression evaluation

#### **EVALUATING AN EXPRESSION IN POSTFIX NOTATION:**

Evaluating an expression in postfix notation is trivially easy if you use a stack. The postfix expression to be evaluated is scanned from left to right. Variables or constants are pushed onto the stack. When an operator is encountered, the indicated action is performed using the top two elements of the stack, and the result replaces the operands on the stack.

#### **Steps to be noted while evaluating a postfix expression using a stack:**

- Traverse from left to right of the expression.
- If an operand is encountered, push it onto the stack.
- If you see a binary operator, pop two elements from the stack, evaluate those operands with that operator, and push the result back in the stack.
- If you see a unary operator, pop one elements from the stack, evaluate those operands with that operator, and push the result back in the stack.
- When the evaluation of the entire expression is over, the only thing left on the stack should be the final result. If there are zero or more than 1 operands left on the stack, either your program is inconsistent, or the expression was invalid.

#### **ALGORITHM TO EVALUATE A POSTFIX EXPRESSION:**

EVALPOST (POSTEXP:STRING)

Where POSTEXP contains postfix expression

STEP 1: initialize the Stack

STEP 2: while (POSTEXP!=NULL)

STEP 3: CH=get the character from POSTEXP

STEP 4: if (CH==Operand ) then

Else if (CH==operator) Then

Pop the two operand from the stack and perform arithmetic operation with the operator and Push the resultant value into the stack

STEP 5: [End of STEP 2 While Structure]

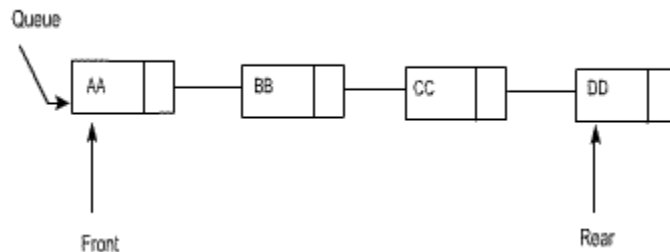
STEP 6:Pop the data from the Stack and return the popped data.



Step No.	Value of i	Operation	Stack
1	2	Push 2 in stack	2
2	3	Push 3 in stack	3 2
3	4	Push 4 in stack	4 3 2
4	+	Pop 2 elements from stack and perform addition operation. And push result back to stack. i.e. $4+3 = 7$	7 2
5	*	Pop 2 elements from stack and perform multiplication operation. And push result back to stack. i.e. $7 * 2 = 14$	14
6	6	Push 6 in stack	6 14
7	-	Pop 2 elements from stack and perform subtraction operation. And push result back to stack. i.e. $14 - 6 = 8$	8
8		Pop result from stack and display	

## Queues

- What is queue?
  - **Definition of Queue.** A **queue** is a data collection in which the items are kept in the order in which they were inserted, and the primary operations are enqueue (insert an item at the end) and dequeue (remove the item at the front).
  - Queues are also called “first-in first-out ” (FIFO) list. Since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter in a queue is the order in which they leave. The real life example: the people waiting in a line at Railway ticket Counter form a queue, where the first person in a line is the first person to be waited on. An important example of a queue in computer science occurs in timesharing system, in which programs with the same priority form a queue while waiting to be executed.



- Array representation of Queue.
- Different types of Queues
  - Linear queue, Dequeue, Circular queue, Priority Queue.
- Operation performed on Queue.
  - Traverse insertion, deletion operation and algorithm.

### **Example:**

1. **PRACTICAL EXAMPLE:** A line at a ticket counter for buying tickets operates on above rules
2. **IN COMPUTER WORLD:** In a batch processing system, jobs are queued up for processing.

### **Conditions in Queue**

- $FRONT < 0$  ( Queue is Empty )
- $REAR = \text{Size of Queue}$  ( Queue is Full )
- $FRONT < REAR$  ( Queue contains at least one element )
- No of elements in queue is :  $( REAR - FRONT ) + 1$

### **Restriction in Queue**

We cannot insert element directly at middle index (position) in Queue and vice versa for deletion. Insertion operation possible at REAR end only and deletion operation at FRONT end, to insert we increment REAR and to delete we increment FRONT.

### Algorithm for ENQUEUE (insert element in Queue)

*Input* : An element say ITEM that has to be inserted.

*Output* : ITEM is at the REAR of the Queue.

*Data structure*: Que is an array representation of queue structure with two pointer FRONT and REAR.

Steps:

1. If ( REAR = size ) then //Queue is full
2.     print "Queue is full"
3.     Exit
4. Else
5.     If ( FRONT = 0 ) and ( REAR = 0 ) then //Queue is empty
6.         FRONT = 1
7.     End if
8.     REAR = REAR + 1 // increment REAR
9.     Que[ REAR ] = ITEM
10. End if
11. Stop

### Algorithm for DEQUEUE (delete element from Queue)

*Input* : A que with elements. FRONT and REAR are two pointer of queue .

*Output* : The deleted element is stored in ITEM.

*Data structure* : Que is an array representation of queue structure..

Steps:

1. If ( FRONT = 0 ) then
2.     print "Queue is empty"
3.     Exit
4. Else
5.     ITEM = Que [ FRONT ]
6.     If ( FRONT = REAR )
7.         REAR = 0
8.         FRONT = 0
9.     Else
10.         FRONT = FRONT + 1
11.     End if
12. End if
13. Stop

### //Program of Queue using array

```
#include<stdio.h>
#define MAX 5
int queue_arr[MAX];
int rear = -1;
int front = -1;
void insert();
void del();
```

```

void display();
void main()
{
    int choice;
    while(1)    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice)    {
            case 1 :
                insert();
                break;
            case 2 :
                del();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(1);
            default:
                printf("Wrong choice\n");
        }
    }
}

```

```

void insert()
{
    int added_item;
    if (rear==MAX-1)
        printf("Queue Overflow\n");
    else
    {
        if (front==-1) /*If queue is initially empty */
            front=0;

        printf("Input the element for adding in queue : ");
        scanf("%d", &added_item);

        rear=rear+1;
        queue_arr[rear] = added_item ;
    }
}

```

```

void del()
{
    if (front == -1 || front > rear)
    {
        printf("Queue Underflow\n");
        return ;
    }
}

```

```

    else
    {
        printf("Element deleted from queue is : %d\n", queue_arr[front]);
        front=front+1;
    }
}

void display()
{
    int i;
    if (front == -1)
        printf("Queue is empty\n");
    else
    {
        printf("Queue is :\n");
        for(i=front;i<= rear;i++)
            printf("%d ",queue_arr[i]);
        printf("\n");
    } /*End of display() */
}

```

## Circular Queue

### Primary operations defined for a circular queue are:

1. **Insert** - It is used for addition of elements to the circular queue.

2. **Delete** - It is used for deletion of elements from the queue.

We will see that in a circular queue, unlike static linear array implementation of the queue ; the memory is utilized more efficient in case of circular queue's.

The shortcoming of static linear that once rear points to n which is the max size of our array we cannot insert any more elements even if there is space in the queue is removed efficiently using a circular queue.

As in case of linear queue, we'll see that condition for zero elements still remains the same i.e..  
 $\text{rear} = \text{front}$

### ALGORITHM FOR ADDITION AND DELETION OF ELEMENTS

Data structures required for circular queue:

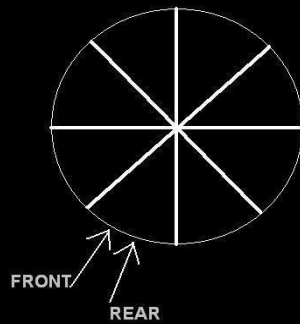
1. **front** counter which points to one position anticlockwise to the 1st element
2. **rear** counter which points to the last element in the queue
3. an **array** to represent the queue

```

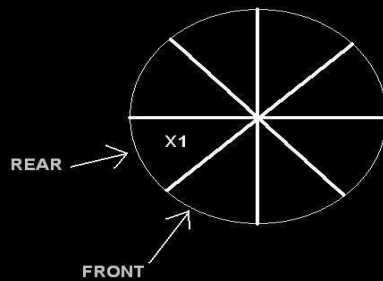
add ( item,queue,rear,front)
{
    rear=(rear+1)mod n;
    if (front == rear )
        then print " queue is full "
    else { queue [rear]=item;
    }
}

```

A VISUAL IDEA OF ADDITION OF ITEMS TO A CIRCULAR QUEUE .



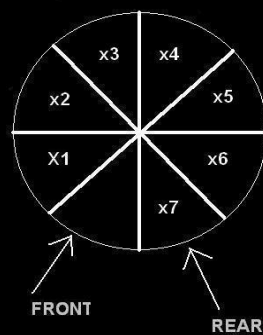
A VISUAL IDEA OF ADDITION OF ITEMS TO A CIRCULAR QUEUE .



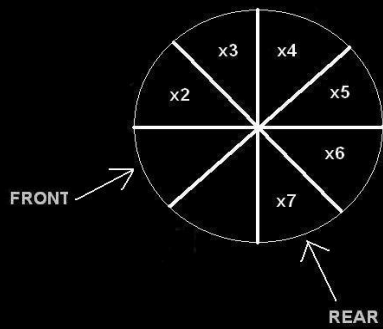
## delete operation :

```
delete_circular (item, queue, rear, front)
{
    if (front == rear)
        print ("queue is empty");
    else { front= front+1;
item= queue[front];
    }
}
```

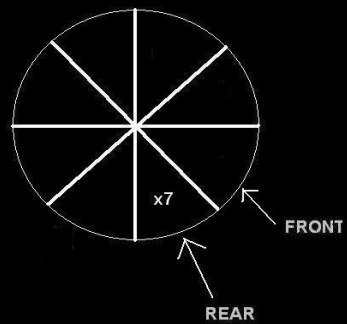
A VISUAL IDEA OF DELETION OF ITEMS TO A CIRCULAR QUEUE .



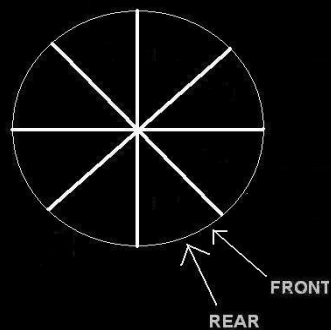
A VISUAL IDEA OF DELETION OF ITEMS TO A CIRCULAR QUEUE .



A VISUAL IDEA OF DELETION OF ITEMS TO A CIRCULAR QUEUE .



A VISUAL IDEA OF DELETION OF ITEMS TO A CIRCULAR QUEUE .



### //Program of circular queue using array

```
#include<stdio.h>
#define MAX 5
int cqueue_arr[MAX];
int front = -1;
int rear = -1;
```

```
void insert();
void del();
void display();
```

```

void main()
{
    int choice;
    while(1)
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1 :
                insert();
                break;
            case 2 :
                del();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(1);
            default:
                printf("Wrong choice\n");
        }
    }
}

```

```

void insert()
{
    int added_item;
    if(front==(rear+1)%max)
    {
        printf("Queue Overflow \n");
        return;
    }
    else
    {
        rear=(rear+1)%max;
        printf("Input the element for insertion in queue : ");
        scanf("%d", &added_item);
        cqueue_arr[rear] = added_item ;

        if(f==-1)
f=0;
    }
}

void del()
{
    if (front ==rear== -1)
    {
        printf("Queue Underflow\n");
    }
}

```



```

        return ;
    }
    printf("Element deleted from queue is : %d\n",cqueue_arr[front]);
    if(front == rear)          /* queue has only one element */
    {
        front = -1;
        rear=-1;
    }
    else
        front=(front+1)%max;
}

```

```

void display()
{
    int front_pos = front,rear_pos = rear;
    if(front == -1)
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements :\n");
    if( front_pos <= rear_pos )
        while(front_pos <= rear_pos)
        {
            printf("%d ",cqueue_arr[front_pos]);
            front_pos++;
        }
    else
    {
        while(front_pos <= MAX-1)
        {
            printf("%d ",cqueue_arr[front_pos]);
            front_pos++;
        }
        front_pos = 0;
        while(front_pos <= rear_pos) {
            printf("%d ",cqueue_arr[front_pos]);
            front_pos++;
        }
    }

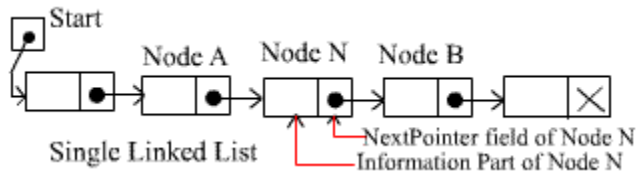
    printf("\n");
}

```

## [Linked lists: Single linked lists](#)

### [What is a Linked Lists?](#)

A linked list is simply a chain of structures which contain a pointer to the next element. It is dynamic in nature. Items may be added to it or deleted from it at will.



**each node of the list has two elements:**

- the item being stored in the list and
- a pointer to the next item in the list

**Some common examples of a linked list:**

- Hash tables use linked lists for collision resolution
- Any "File Requester" dialog uses a linked list
- Binary Trees
- Stacks and Queues can be implemented with a doubly linked list
- Relational Databases (e.g. Microsoft Access)

- Operations on linked list :
  - Traversing
  - Searching in a linked list
    - Sorted
    - Unsorted

### **Types of Link List**

1. Linearly-linked List
  - Singly-linked list
  - Doubly-linked list
2. Circularly-linked list
  - Singly-circularly-linked list
  - Doubly-circularly-linked list
3. Sentinel nod

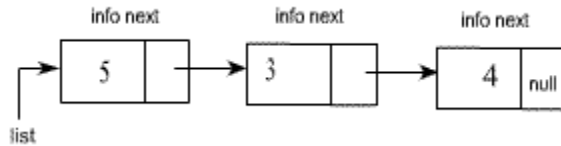
### **Operations on Linked lists**

- Insertion
  - Insert at the beginning
  - Insert at the end
  - Insert after a given node
- Deletion
  - Delete the first node
  - \Delete the last node
  - Delete a particular node

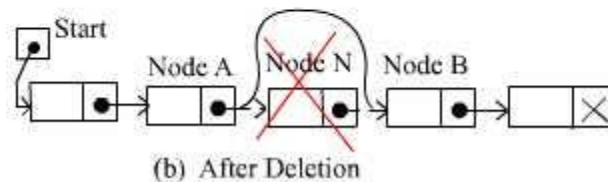
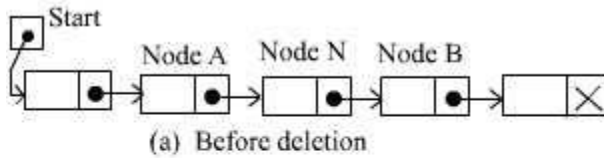
### **Algorithm for inserting a node to the List**

- allocate space for a new node,
- copy the item into it,
- make the new node's next pointer point to the current head of the list and
- Make the head of the list point to the newly allocated node.

This strategy is fast and efficient, but each item is added to the head of the list. Below is given C code for inserting a node after a given nod



### Algorithm for deleting a node from the List



Step-1: Take the value in the 'nodevalue' field of the TARGET node in any intermediate variable. Here node N.

Step-2: Make the previous node of TARGET to point to where TARGET is currently pointing

Step-3: The nextpointer field of Node N now points to Node B, Where Node N previously pointed.

Step-4: Return the value in that intermediate variable

### //Program of single linked list

```

#include <stdio.h>
#include <malloc.h>
struct node
{
    int info;
    struct node *link;
} *start;
main()
{
    int choice,n,m,position,i;
    start=NULL;
    while(1)
    {
        printf("1.Create List\n");
        printf("2.Add at begining\n");
        printf("3.Add after \n");
        printf("4.Delete\n");
        printf("5.Display\n");
        printf("6.Count\n");
        printf("7.Reverse\n");
        printf("8.Search\n");
        printf("9.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
  
```

```

        printf("How many nodes you want : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
            printf("Enter the element : ");
            scanf("%d",&m);
            create_list(m);
        }
        break;
    case 2:
        printf("Enter the element : ");
        scanf("%d",&m);
        addatbeg(m);
        break;
    case 3:
        printf("Enter the element : ");
        scanf("%d",&m);
        printf("Enter the position after which this element is inserted : ");
        scanf("%d",&position);
        addafter(m,position);
        break;
    case 4:
        if(start==NULL)
        {
            printf("List is empty\n");
            continue;
        }
        printf("Enter the element for deletion : ");
        scanf("%d",&m);
        del(m);
        break;
    case 5:
        display();
        break;
    case 6:
        count();
        break;
    case 7:
        rev();
        break;
    case 8:
        printf("Enter the element to be searched : ");
        scanf("%d",&m);
        search(m);
        break;
    case 9:
        exit();
    default:
        printf("Wrong choice\n");
    }/*End of switch */
}/*End of while */
}/*End of main()*/

```

```

create_list(int data)
{
    struct node *q,*temp;
    temp= malloc(sizeof(struct node));

```

```

temp->info=data;
temp->link=NULL;
if(start==NULL) /*If list is empty */
    start=temp;
else {
/*Element inserted at the end */
    q=start;
    while(q->link!=NULL)
        q=q->link;
    q->link=temp;
}
}/*End of create_list()*/
addatbeg(int data)
{
    struct node *temp;
    temp=malloc(sizeof(struct node));
    temp->info=data;
    temp->link=start;
    start=temp;
}/*End of addatbeg()*/

addafter(int data,int pos)
{
    struct node *temp,*q;
    int i;
    q=start;
    for(i=0;
i<pos-1;
i++)
    {
        q=q->link;
        if(q==NULL)
        {
            printf("There are less than %d elements",pos);
            return;
        }
    }
/*End of for*/
temp=malloc(sizeof(struct node ));
temp->link=q->link;
temp->info=data;
q->link=temp;
}/*End of addafter()*/
del(int data)
{
    struct node *temp,*q;
    if(start->info == data)
    {
        temp=start;
        start=start->link;
/*First element deleted*/
        free(temp);
        return;
    }
    q=start;
    while(q->link->link != NULL)

```

```

    {
        if(q->link->info==data)
/*Element deleted in between*/
        {
            temp=q->link;
            q->link=temp->link;
            free(temp);
            return;
        }
        q=q->link;
    }/*End of while */
    if(q->link->info==data) /*Last element deleted*/
    {
        temp=q->link;
        free(temp);
        q->link=NULL;
        return;
    }
    printf("Element %d not found\n",data);
}/*End of del()*/
display()
{
    struct node *q;
    if(start == NULL)
    {
        printf("List is empty\n");
        return;
    }
    q=start;
    printf("List is :\n");
    while(q!=NULL)
    {
        printf("%d ", q->info);
        q=q->link;
    }
    printf("\n");
}/*End of display() */
count()
{
    struct node *q=start;
    int cnt=0;
    while(q!=NULL)
    {
        q=q->link;
        cnt++;
    }
    printf("Number of elements are %d\n",cnt);
}/*End of count() */
rev()
{
    struct node *p1,*p2,*p3;
    if(start->link==NULL) /*only one element*/
        return;
    p1=start;
    p2=p1->link;

```

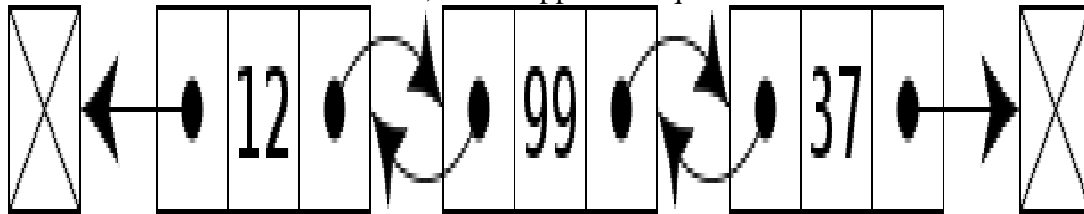
```

p3=p2->link;
p1->link=NULL;
p2->link=p1;
while(p3!=NULL)
{
    p1=p2;
    p2=p3;
    p3=p3->link;
    p2->link=p1;
}
start=p2;
}/*End of rev()*/
search(int data)
{
    struct node *ptr = start;
    int pos = 1;
    while(ptr!=NULL)
    {
        if(ptr->info==data)
        {
            printf("Item %d found at position %d\n",data,pos);
            return;
        }
        ptr = ptr->link;
        pos++;
    }
    if(ptr == NULL)
        printf("Item %d not found in list\n",data);
}/*End of search()*/

```

## Doubly linked list

In computer science, a doubly-linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called *links*, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.



A doubly-linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.

The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly-linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Doubly linked lists are like singly linked lists, in which for each node there are two pointers -- one to the next node, and one to the previous node. This makes life nice in many ways:

- You can traverse lists forward and backward.
- You can insert anywhere in a list easily. This includes inserting before a node, after a node, at the front of the list, and at the end of the list and
- You can delete nodes very easily.

Doubly linked lists may be either linear or circular and may or may not contain a header node

### Operations on double linked list :

- Traversing
- Insertion
  - Insert at the beginning
  - Insert at the end
  - Insert after a given node
- Deletion
  - Delete the first node
  - Delete the last node
  - Delete a particular node

## CIRCULAR LIST

Circular lists are like singly linked lists, except that the last node contains a pointer back to the first node rather than the null pointer. From any point in such a list, it is possible to reach any other point in the list. If we begin at a given node and travel the entire list, we ultimately end up at the starting point.



Note that a circular list does not have a natural "first or "last" node. We must therefore, establish a first and last node by convention - let external pointer point to the last node, and the following node be the first node.

- Single circular linked list
- Double circular linked list

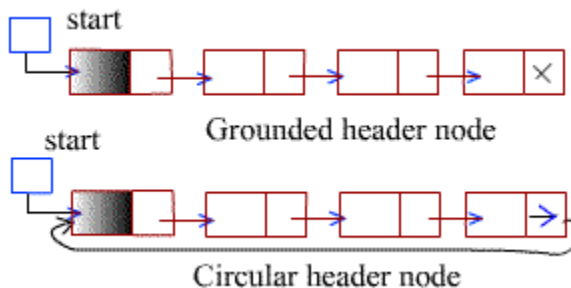
## Header Nodes

A header linked list is a linked list which always contains a special node called the header node at the beginning of the list. It is an extra node kept at the front of a list. Such a node does not represent an item in the list. The information portion might be unused. There are two types of header list

1. Grounded header list: is a header list where the last node contain the null pointer.
2. Circular header list: is a header list where the last node points back to the header node.

More often, the information portion of such a node could be used to keep global information about the entire list such as:

- number of nodes (not including the header) in the list  
count in the header node must be adjusted after adding or deleting the item from the list
- pointer to the last node in the list  
it simplifies the representation of a queue
- pointer to the current node in the list  
eliminates the need of a external pointer during traversal



## Module-V

### Trees: Tree Terminology

A tree is a finite set of nodes together with a finite set of directed edges that define parent-child relationships. Each directed edge connects a parent to its child.

Nodes={ A,B,C,D,E,f,G,H} Edges={ (A,B),(A,E),(B,F),(B,G),(B,H),(E,C),(E,D)}

This structure is mainly used to represent data containing a hierarchical relationship between elements, e.g. record, family tree and table of contents.

A tree satisfies the following properties:

1. It has one designated node, called the root that has no parent.
2. Every node, except the root, has exactly one parent.
3. A node may have zero or more children.
4. There is a unique directed path from the root to each node.

**Ancestor** of a node v: Any node, including v itself, on the path from the root to the node.

**Proper ancestor** of a node v: Any node, excluding v, on the path from the root to the node.

**Descendant** of a node v: Any node, including v itself, on any path from the node to a leaf node (i.e., a node with no children).

**Proper descendant** of a node  $v$ : Any node, excluding  $v$ , on any path from the node to a leaf node.

**Subtree** of a node  $v$ : A tree rooted at a child of  $v$ .

**Leaf**: A node with degree 0.

**Internal** or interior node: a node with degree greater than 0.

**Siblings**: Nodes that have the same parent.

**Size**: The number of nodes in a tree.

**Level** (or depth) of a node  $v$ : The length of the path from the root to  $v$ .

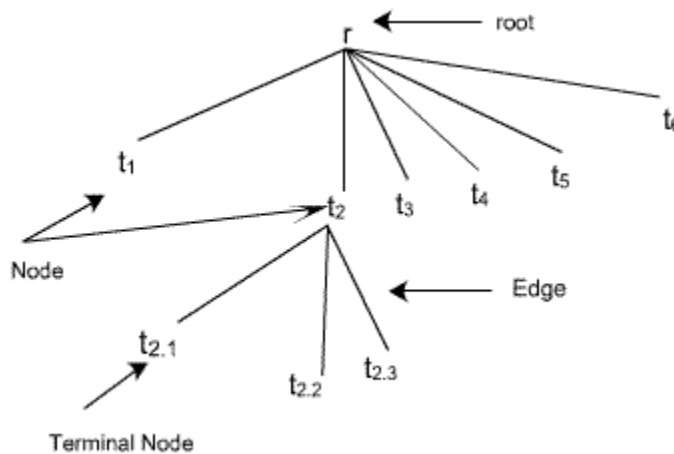
**Height** of a node  $v$ : The length of the longest path from  $v$  to a leaf node.

–The height of a tree is the height of its root node.

–By definition the height of an empty tree is -1.

A tree consists of a distinguished node  $r$ , called the **root** and zero or more (sub) tree  $t_1, t_2, \dots, t_n$ , each of whose roots are connected by a directed edge to  $r$ .

In the tree of figure, the root is  $A$ , Node  $t_2$  has  $r$  as a parent and  $t_{2.1}, t_{2.2}$  and  $t_{2.3}$  as children. Each node may have arbitrary number of children, possibly zero. Nodes with no children are known as leaves.



An **internal node** (also known as an **inner node**, **inode** for short, or **branch node**) is any node of a tree that has child nodes. Similarly, an **external node** (also known as an **outer node**, **leaf node**, or **terminal node**) is any node that does not have child nodes.

The **height** of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The **depth** of a node is the length of the path to its root (i.e., its *root path*).

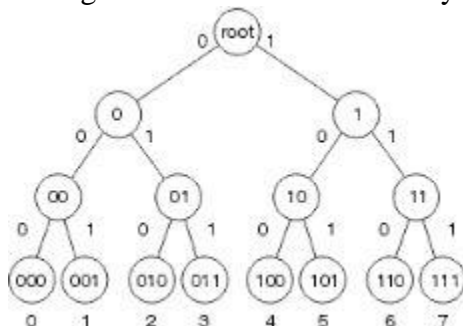
## Binary Tree :

A **tree** is a finite set of nodes having a distinct node called root.

**Binary Tree** is a tree which is either empty or has at most two subtrees, each of the subtrees also being a binary tree. It means each node in a binary tree can have 0, 1 or 2 subtrees. A left or right subtree can be empty.

A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers point to smaller "**subtrees**" on either side. A null pointer represents a binary tree with no elements -- the empty tree. The formal recursive definition is: a binary tree is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree.

The figure shown below is a binary tree.

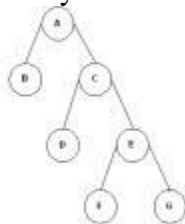


It has a distinct node called root i.e. 2. And every node has either 0, 1 or 2 children. So it is a binary tree as every node has a maximum of 2 children.

If A is the root of a binary tree & B the root of its left or right subtree, then A is the parent or father of B and B is the left or right child of A. Those nodes having no children are leaf nodes. Any node say A is the ancestor of node B and B is the descendant of A if A is either the father of B or the father of some ancestor of B. Two nodes having same father are called brothers or siblings.

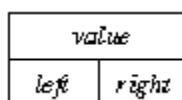
Going from leaves to root is called climbing the tree & going from root to leaves is called descending the tree.

A binary tree in which every non leaf node has non empty left & right subtrees is called a strictly binary tree. The tree shown below is a strictly binary tree.



The structure of each node of a binary tree contains one data field and two pointers, each for the right & left child. Each child being a node has also the same structure.

The structure of a node is shown below.



The structure defining a node of binary tree in C is as follows.

Struct node

```
{
struct node *lc ; /* points to the left child */
int data; /* data field */
```

```

struct node *rc; /* points to the right child */
}

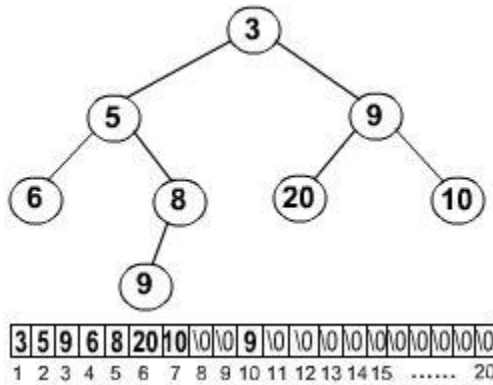
```

There are two ways for representation of binary tree.

- Linked List representation of a Binary tree
- Array representation of a Binary tree

### Array Representation of Binary Tree:

- A single array can be used to represent a binary tree.
- For these nodes are numbered / indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered. Then each node having an index  $i$  is put into the array as its  $i$ th element.
- In the figure shown below the nodes of binary tree are numbered according to the given scheme.



### Linked Representation of Binary Tree :

Binary trees can be represented by links where each node contains the address of the left child and the right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.

The structure defining a node of binary tree in C is as follows.

Struct node

```

{
struct node *lc ; /* points to the left child */
int data; /* data field */
struct node *rc; /* points to the right child */
}

```

[AVL Tree](#)

[INTRODUCTION](#)

As we know that searching in a binary search tree is efficient if the height of the left sub-tree and right sub-tree is same for a node. But frequent insertion and deletion in the tree affects the efficiency and makes a binary search tree inefficient. The efficiency of searching will be ideal if the difference in height of left and right sub-tree with respect of a node is at most one. Such a binary search tree is called balanced binary tree (sometimes called AVL Tree).

### REPRESENTATION OF AVL TREE

In order to represent a node of an AVL Tree, we need four fields :- One for data, two for storing address of left and right child and one is required to hold the balance factor. The balance factor is calculated by subtracting the right sub-tree from the height of left sub - tree. The structure of AVL Tree can be represented by : -

Struct AVL

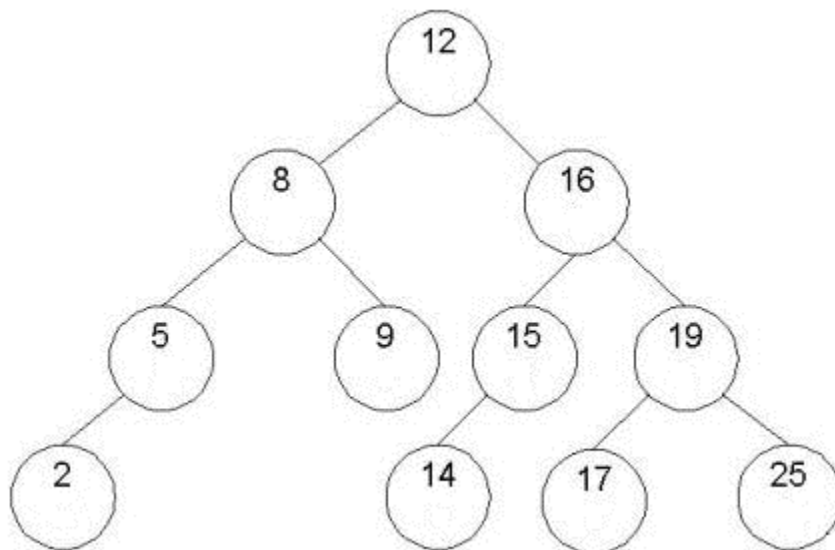
```
{  
    struct AVL *left;  
    int data;  
    struct AVL *right;  
    int balfact;  
};
```

### DETERMINATION OF BALANCE FACTOR

The value of balance factor may be -1, 0 or 1.

Any value other than these represent that the tree is not an AVL Tree

1. If the value of balance factor is -1, it shows that the height of right sub-tree is one more than the height of the left sub-tree with respect to the given node.
2. If the value of balance factor is 0, it shows that the height of right sub-tree is equal to the height of the left Sub-tree with respect to the given node.
3. If the value of balance factor is 1, it shows that the height of right sub-tree is one less than the height of the left sub-tree with respect to the given node .



- Types of rotation-
  - LL rotation,RR rotation,LR rotation,RL rotation
- Operations on AVL tree.
  - Insertion,deletion

## INSERTION OF A NODE IN AVL TREE

Insertion can be done by finding an appropriate place for the node to be inserted. But this can disturb the balance of the tree if the difference of height of sub-trees with respect to a node exceeds the value one. If the insertion is done as a child of non-leaf node then it will not affect the balance, as the height doesn't increase. But if the insertion is done as a child of leaf node, then it can bring the real disturbance in the balance of the tree.

This depends on whether the node is inserted to the left sub-tree or the right sub-tree, which in turn changes the balance factor. If the node to be inserted is inserted as a node of a sub-tree of smaller height then there will be no effect. If the height of both the left and right sub-tree is same then insertion to any of them doesn't affect the balance of AVL Tree. But if it is inserted as a node of sub-tree of larger height, then the balance will be disturbed.

To rebalance the tree, the nodes need to be properly adjusted. So, after insertion of a new node the tree is traversed starting from the new node to the node where the balance has been disturbed. The nodes are adjusted in such a way that the balance is regained.

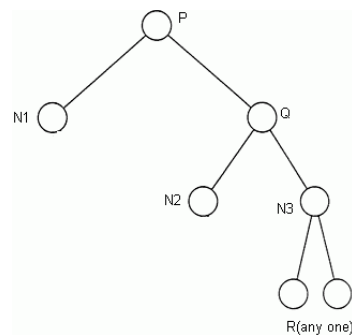
## DELETION

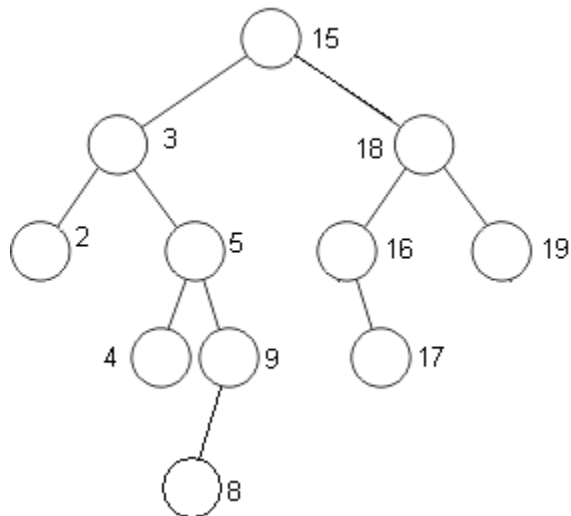
- A node in AVL Tree is deleted as it is deleted in the binary search tree. The only difference is that we have to do rebalancing which is done similar to that of insertion of a node in AVL Tree. The algorithm for deletion and rebalancing is given below:

## **REBALANCING OF AVL TREE**

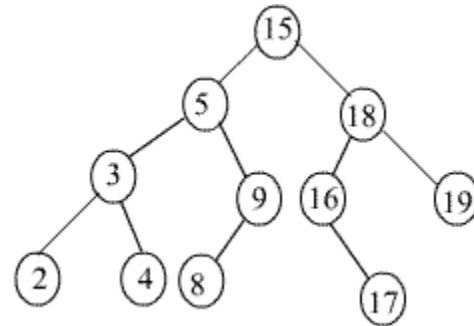
When we insert a node to the taller sub-tree, four cases arise and we have different rebalancing methods to bring it back to a balanced tree form.

1. Left Rotation
2. Right Rotation
3. Right and Left Rotation
4. Left and Right Rotation





Before Rotation



After Rotation

### EXPLANATION OF EXAMPLE

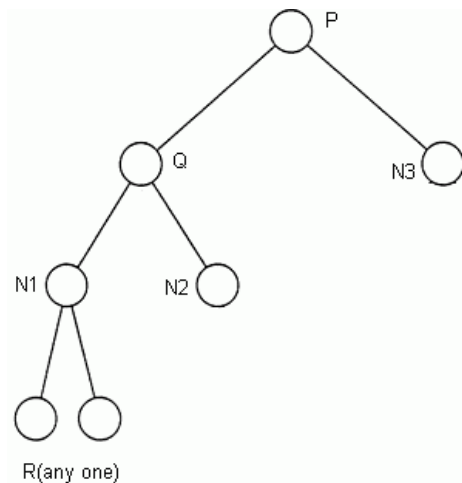
In the given AVL tree when we insert a node 8, it becomes the left child of node 9 and the balance doesn't exist, as the balance factor of node 3 becomes -2. So, we try to rebalance it. In order to do so, we do left rotation at node 3. Now node 5 becomes the left child of the root. Node 9 and node 3 becomes the right and left child of node 5 respectively. Node 2 and node 4 becomes the left and right child of node 3 respectively. Lastly, node 8 becomes the left child of node 9. Hence, the balance is once again attained and we get AVL Tree after the left rotation.

### RIGHT ROTATION

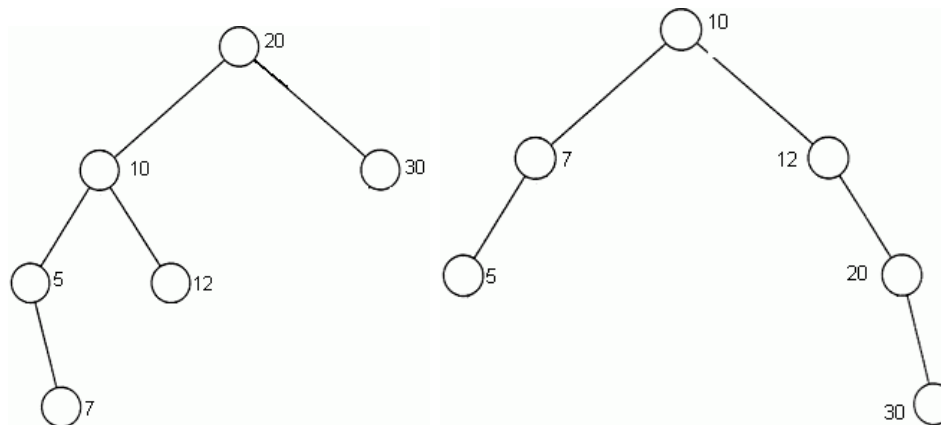
In general if we want to insert a node R (either as left or right child) to N1 as shown in figure. Here, as we see the balance factor of node P becomes 2. So to rebalance it, we have a technique called right rotation.

### EXAMPLE

#### GENERAL DIAGRAM







Before Rotation

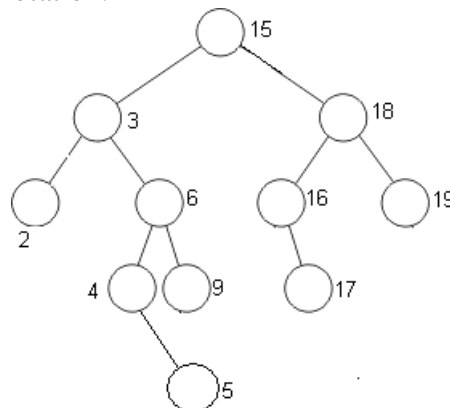
After Rotation

### EXPLANATION OF EXAMPLE

In the given AVL tree when we insert a node 7, it becomes the right child of node 5 and the balance doesn't exist, as the balance factor of node 20 becomes 2. So, we try to rebalance it. In order to do so, we do right rotation at node 20. Now node 10 becomes the root. Node 12 and node 7 becomes the right and left child of root respectively. Node 20 becomes the right child of node 12. Node 30 becomes the right child of node 20. Lastly, node 5 becomes the left child of node 7. Hence, the balance is once again attained and we get AVL Tree after the right rotation

### RIGHT AND LEFT ROTATION

In general if we want to insert a node R (either as a left or right child) to N2 as shown in figure. Here, as we see the balance factor of node P becomes -2. So to rebalance it, we have to do two rotations. Firstly, right rotation and then left rotation.



## Tree traversals

Traversal of a binary tree means to visit each node in the tree exactly once. The tree traversal is used in all the applications of it.



In a linear list nodes are visited from first to last, but a tree being a non linear one we need definite rules. There are a no. of ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are:

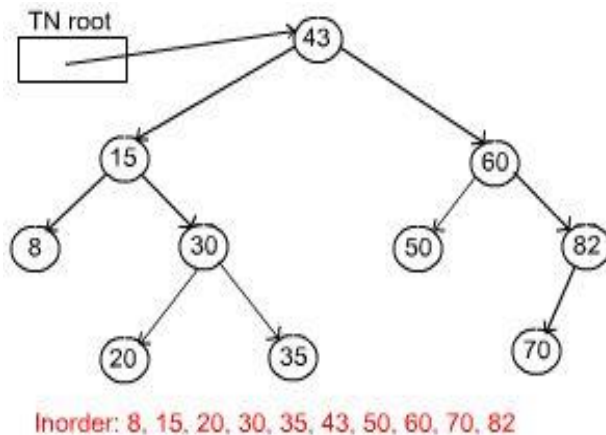
- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

In all of them we do not require to do anything to traverse an empty tree. All the traversal methods are based on recursive functions since a binary tree is itself recursive as every child of a node in a binary tree is itself a binary tree.

To traverse a non empty tree in inorder the following steps are followed recursively.

- Visit the Root
- Traverse the left subtree
- Traverse the right subtree

The inorder traversal of the tree shown below is as follows.



## Inorder Traversal :

Algorithm

The algorithm for inorder traversal is as follows.

Struct node

```
{
struct node * lc;
int data;
struct node * rc;
};
void inorder(struct node * root);
{
if(root != NULL)
{
inorder(roo-> lc);
printf("%d\t",root->data);
inorder(root->rc);
}
}
```

## Searching

### Binary search

In computer science, a **binary search** or **half-interval search algorithm** finds the position of a specified input value (the search "key") within an array sorted by key value.<sup>[1][2]</sup> For binary search, the array should be arranged in ascending or descending order. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

### Program

```
#include <stdio.h>
void main()
{
    int c, first, last, middle, n, search, array[100];
    printf("Enter number of elements\n");
    scanf("%d",&n);
    printf("Enter %d integers\n", n);
    for ( c = 0 ; c < n ; c++ )
        scanf("%d",&array[c]);
    printf("Enter value to find\n");
    scanf("%d",&search);
    first = 0;
    last = n - 1;
    middle = (first+last)/2;
    while( first <= last )
    {
        if ( array[middle] < search )
            first = middle + 1;
        else if ( array[middle] == search )
        {
            printf("%d found at location %d.\n", search, middle+1);
            break;
        }
        else
            last = middle - 1;
        middle = (first + last)/2;
    }
    if ( first > last )
        printf("Not found! %d is not present in the list.\n", search);
}
```

### Linear search

In computer science, **linear search** or **sequential search** is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Linear search is the simplest search algorithm; it is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list; and so is its expected cost, if all list elements are equally likely to be searched for. Therefore, if the list has more than a few elements, other methods (such as binary search or hashing) will be faster, but they also impose additional requirements.

### **Program**

```
#include <stdio.h>
int main()
{
    int array[100], search, c, n;
    printf("Enter the number of elements in array\n");
    scanf("%d",&n);
    printf("Enter %d integer(s)\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    printf("Enter the number to search\n");
    scanf("%d", &search);
    for (c = 0; c < n; c++)
    {
        if (array[c] == search)    /* if required element found */
        {
            printf("%d is present at location %d.\n", search, c+1);
            break;
        }
    }
    if (c == n)
        printf("%d is not present in array.\n", search);

    return 0;
}
```

## **Sorting**

### **Bubble sort**

**Bubble sort**, sometimes referred to as **sinking sort**, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, it is too slow for practical use, even compared to insertion sort.

#### **Step-by-step example**

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

### First Pass:

( **5** 1 4 2 8 )      ( **1** 5 4 2 8 ), Here, algorithm compares the first two elements, and swaps since  $5 > 1$ .

( 1 **5** 4 2 8 )  $\rightarrow$  ( 1 **4** 5 2 8 ), Swap since  $5 > 4$

( 1 4 **5** 2 8 )  $\rightarrow$  ( 1 4 **2** 5 8 ), Swap since  $5 > 2$

( 1 4 2 **5** 8 )  $\rightarrow$  ( 1 4 2 **5** 8 ), Now, since these elements are already in order ( $8 > 5$ ), algorithm does not swap them.

### Second Pass:

( **1** 4 2 5 8 )  $\rightarrow$  ( **1** 4 2 5 8 )

( 1 **4** 2 5 8 )  $\rightarrow$  ( 1 **2** 4 5 8 ), Swap since  $4 > 2$

( 1 2 **4** 5 8 )  $\rightarrow$  ( 1 2 **4** 5 8 )

( 1 2 4 **5** 8 )  $\rightarrow$  ( 1 2 4 **5** 8 )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

### Third Pass:

( **1** 2 4 5 8 )  $\rightarrow$  ( **1** 2 4 5 8 )

( 1 **2** 4 5 8 )  $\rightarrow$  ( 1 **2** 4 5 8 )

( 1 2 **4** 5 8 )  $\rightarrow$  ( 1 2 **4** 5 8 )

( 1 2 4 **5** 8 )  $\rightarrow$  ( 1 2 4 **5** 8 )

The algorithm can be expressed as (0-based array):

```
procedure bubbleSort( A : list of sortable items )
  n = length(A)
  repeat
    swapped = false
    for i = 1 to n-1 inclusive do
      /* if this pair is out of order */
      if A[i-1] > A[i] then
        /* swap them and remember something changed */
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  until not swapped
end procedure
```

## QUICK SORT

**Quicksort**, or **partition-exchange sort**, is a sorting algorithm developed by Tony Hoare that, on average, makes  $O(n \log n)$  comparisons to sort  $n$  items. In the worst case, it makes  $O(n^2)$  comparisons, though this behavior is rare. Quicksort is often faster in practice than other  $O(n \log n)$  algorithms. Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort is a comparison sort and, in efficient implementations, is not a stable sort. Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only  $O(\log n)$  additional space used by the stack during the recursion.

```
#include <stdio.h>
#define MAX 500
```

```

void quickSort(int A[], int maxsize);
void sort(int A[], int left, int right);
int A[MAX];
void main()
{
    int i, max;
    clrscr();
    printf("\nHow many elements you want to sort: ");
    scanf("%d",&max);
    printf("\nEnter the values one by one: ");
        for (i = 0; i < max; i++)
            scanf("%d",&A[i]);
    printf("\nArray before sorting:");
        for (i= 0;i< max;i++)
            printf("%d",A[i]);
    printf ("\n");
    quickSort(A, max);
    printf("\nArray after sorting:");
    for (i = 0; i < max; i++)
        printf("%d ", A[i]);
    getch();
}
void quickSort(int A[], int max)
{
    sort(A,0,max - 1);
}
void sort(int A[], int left, int right)
{
    int pivot, l, r;
    l = left;
    r = right;
    pivot = A[left];
    while (left < right)
    {
        while ((A[right] >= pivot) && (left < right))
            right--;
        if (left != right)
        {
            A[left] = A[right];
            left++;
        }
        while ((A[left] <= pivot) && (left < right))
            left++;
        if(left != right)
        {
            A[right] = A[left];
            right--;
        }
    }
    A[left] = pivot;
}

```

```
pivot = left;  
left = l;  
right = r;  
if (left < pivot)  
    sort(A, left, pivot - 1);  
if (right > pivot)  
    sort(A, pivot + 1, right);  
}
```

