

## **CS3353 C Programming and Data Structures**

### **Syllabus**

**Variables – Data Types – Expressions using operators in C – Managing Input and Output operations – Decision Making and Branching – Looping statements. Arrays – – One dimensional and Two- dimensional arrays.**

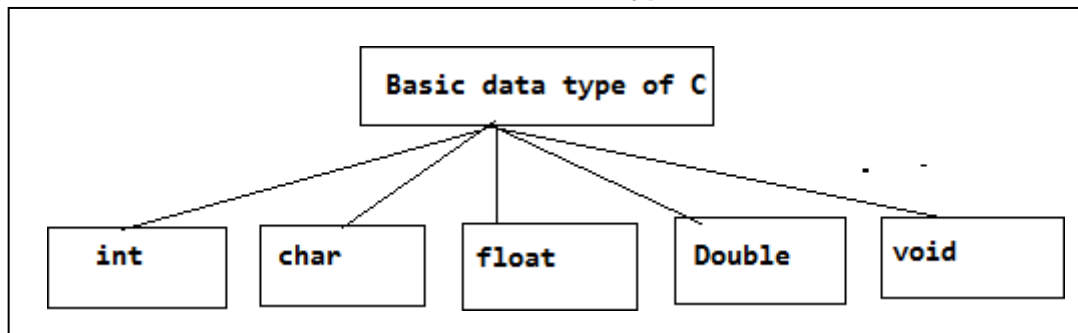
## DATA TYPES IN C

➤ The data type, of a variable determines a set of values that a variable might take and a set of operations that can be applied to those values.

➤ Data types refer to the type and size of data associated with the variable and functions.

➤ Data types can be broadly classified as shown in Figure

**Basic data type of C**



	Data Type	Size in Bytes	Range	Format-Specifier
<b>Int</b>	int	2	-32768 to +32767	%d
	short signed int (or) signed int	2	32768 to +32767	%d
	short unsigned int (or) unsigned int	2	0 to 65535	%u
	long signed int (or) long int	4	-2147483648 to 2147483647	%ld
	long unsigned int	4	0 to 4294967295	%lu
<b>Char</b>	char or signed char	1	-128 to 127	%c
	unsigned char	1	0 to 255	%c
	<b>float</b>  Allows 6 digits after decimal point.	4	$-3.4e^{-38}$ to $+3.4e^{38}$	%f
	<b>double</b>  Allows 15 digits after decimal point.	8	$-1.7e^{-308}$ to $+1.7e^{308}$	%lf
	<b>long double</b>  Allows 15 digits after decimal point.	10	$-1.7e^{-4932}$ to $1.7e^{4932}$	%LF

**/\*Program\*/**

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

```
int main()
```

```
{
```

```
char a;
```

```
unsigned char b;
```

```
int i;
```

```
unsigned int j;
```

```
long int k;
```

```
unsigned long int m;
```

```
float x;
```

```
double y
```

```
long double z;
```

```
printf(—\n char and unsigned char\l);
```

```
scanf(—%c %c\l,&a,&b) //get char and unsigned char value
```

```
printf(—%c %c\l,a,b) //display char and unsigned char value
```

```
printf(—\n int unsigned int\l);
```

```
scanf(—%d %u\l,&i,&j) //get int unsigned int value
```

```
printf(—%d %u\l,i,j) //display int unsigned int value
```

```
printf(—\n long int unsigned long int\l);
```

```
scanf(—%ld %lu\l,&i,&j) //get long int and long unsigned int value
```

```
printf(—%ld %lu\l,i,j) //display int unsigned int value
```

```
printf(—\n float,double and long double\l);
```

```
scanf(—%f %lf %Lf\l,&i,&j) //get float,double and long double value
```

```
printf(—%f %lf %Lf\l,i,j) //display float,double and long double value
```

```
return 0;
```

```
}
```

The specifiers and qualifiers for the data types can be broadly classified into three types

- **Size specifiers**— short and long
- **Sign specifiers**— signed and unsigned
- **Type qualifiers**— const, volatile and restrict.

**Size qualifiers** alter the size of the basic data types. There are two such qualifiers that can be used with the data type int; these are short and long.

**short**, when placed in front of the data type int declaration, tells the C compiler that the particular variable being declared is used to store fairly small integer values. **Long** specifies it is a very big integer value. Long integers require twice the memory of than small ints.

Table: Sizes (bytes) of short int ,int,long int

	16-bit Machine (size in bytes)	16-bit Machine (size in bytes)	16-bit Machine (size in bytes)
<b>short int</b>	2	2	2
<b>Int</b>	2	4	4
<b>long int</b>	4	4	8

Table:Size and range of *long long* type (64-bit machine)

Data type	Size (in bytes)	Range
<b>long long int</b>	8	-9, 223, 372, 036, 854, 775, 808 to +9, 223, 372, 036, 854, 775, 808
<b>unsigned long int    or unsigned long</b>	4	0 to + 4, 294, 967, 295
<b>unsigned long long int    or unsigned long long</b>	8	0 to + 18, 446, 744, 073,709, 551, 615

**Sign specifiers:** for example for int data type out of 2bytes(2\*8=16bits) of its size the highest bit(the sixteenth bit) is used to store the sign of the integer value. The bit is 1 if number is negative and 0 if the number is positive.

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Bit 9	Bit 10	Bit 11	Bit 12	Bit 13	Bit 14	Bit 15	Bit 16
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Sign of number (1 for -ve and 0 for +ve)

**Type qualifiers** : There are two type qualifiers, const and volatile;

Eg: **const float pi = 3.14156;** // specifies that the variable pi can never be changed by the Program.

**Table:Size and range in (16-bit machines)**

Data type	Size (in bits) note:[1byte=8bits]	Range
Char	8	−128 to 127
Int	16	−32768 to 32767
Float	32	$1.17549 \times 10^{-38}$ to $3.40282 \times 10^{38}$
Double	64	$2.22507 \times 10^{-308}$ to $1.79769 \times 10^{308}$
Void	8	Valueless

**Table:Size and range of (32-bit machine)**

Data type	Size (in bits) note:[1byte=8bits]	Range
Char	8	−128 to 127
Int	32	−2147483648 to 2147483647
Float	32	$1.17549 \times 10^{-38}$ to $3.40282 \times 10^{38}$
Double	64	$2.22507 \times 10^{-308}$ to $1.79769 \times 10^{308}$
Void	8	Valueless

**Allowed combinations of basic data types and modifiers in C for a 16-bit computer**

Data Type	Size (bits)	Range
char	8	−128 to 127
unsigned char	8	0 to 255
signed char	8	−128 to 127
int	16	−32768 to 32767
unsigned int	16	0 to 65535
signed int	16	−32768 to 32767
short int	16	−32768 to 32767
unsigned short int	16	0 to 65535
signed short int	16	−32768 to 32767
long int	32	−2147483648 to 2147483647
unsigned long int	32	0 to 4294967295
signed long int	32	−2147483648 to 2147483647
float	32	3.4E−38 to 3.4E+38
double	64	1.7E−308 to 1.7E+308
long double	80	3.4E−4932 to 1.1E+4932

## VARIABLES

Variable is the name of memory location which holds the data. Unlike constant, variables are changeable, value of a variable can be changed during execution of a program. A programmer must choose a meaningful variable name.

Variables are used for holding data values so that they can be utilized for various computations in a program. A variable must be declared and then used for computation work in program. A variable is an identifier used for storing and holding some data (value).

All variables have three important attributes.

1. A **data type**: Like int, double, float. Once defined, the type of a C variable cannot be changed.

2. A **name** of the variable.

3. A **value** that can be changed by assigning a new value to the variable. The kind of values a variable can assume depends on its type.

Eg : for variable int salary, it can only take integer values can only take integer values like 65000 and not 6500.0

### Rules For Constructing Variables

1. A variable name can be a combination of alphabets, numbers and special character underscore ( \_ ).
2. The first character in the variable name must be an alphabet.
3. No commas or blank spaces are available are allowed within a variable name.
4. No special symbol other than an underscore is allowed.
5. Upper and Lower case names are treated as different, as C is case sensitive, so it is suggested to keep the variable names in lower case.

### Declaring and Initializing a variable:==

- Declaration of a variable must be done before it is used for any computation in the program.
- Declaration tells the compiler what the variable name is.
- Declaration tells what type of data the variable will hold.
- Until the variable is not defined/or/declared compiler will not allocate memory space to the variables.
- A variable can also be declared outside main() function.
- A variable can also be declared in other program and declared using extern keyword.

```
int yearly_salary;  
float monthly_salary;  
int a;  
double x;  
int ECE1111;
```

### Initializing a variable:==

Initializing a variable means to provide a value to variable

```
int yearly_salary=5,00,000  
float monthly_salary= 41666.66
```

### Difference between identifier and variable

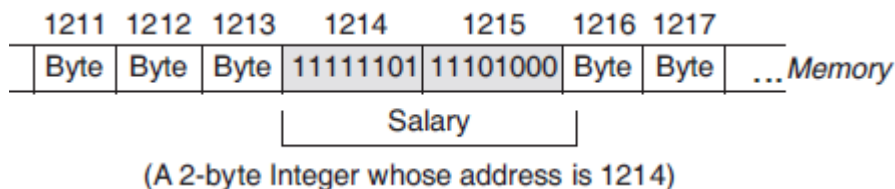
Identifier	Variable
Identifier is the name given to a variable,function etc.	While variable is used to name a memory location which stores data
An identifier can be a variable ,but not all identifiers are variables	All variables names are identifiers
Example : void average() { }	Example: int average

Variables are a way of reserving memory to hold some data and assign names to them so that we don't have to remember the numbers like REG46735 or memory address like FFFF0xFF and instead we can use the memory location by simply referring to the variable.

Every variable is mapped to a unique memory address.

And variable will be having a data type associated

```
int salary = 65000;
```



[[[[[note A computer memory is made up of registers and cells. It accesses data in a collection of bits, typically 8 bits, 16 bit, 32 bit or 64 bit. A computer memory holds information in the form of binary digits 0 and 1 (bits).]]]]]

## Operators in C

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators –

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Special Operators

### Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20

Operator	Description	Example
+	Adds two operands.	$A + B = 30$
-	Subtracts second operand from the first.	$A - B = -10$
*	Multiplies both operands.	$A * B = 200$
/	Divides numerator by de-numerator.	$B / A = 2$
%	Modulus Operator and remainder of after an integer division.	$B \% A = 0$
++	Increment operator increases the integer value by one.	$A++ = 11$
--	Decrement operator decreases the integer value by one.	$A-- = 9$

### Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	$(A == B)$ is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	$(A != B)$ is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	$(A > B)$ is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	$(A < B)$ is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	$(A >= B)$ is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	$(A <= B)$ is true.

## Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then –

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A    B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

## **Bitwise Operators**

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows –

P	q	p & q	p   q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows –

A = 0011 1100

B = 0000 1101

.....  
A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then –

Operator	Description	Example
&	Binary AND <b>It takes 1 if both operands has value 1.</b>	(A & B) = 12, i.e., 0000 1100
	Binary OR Operator copies a bit if it exists in either operand <b>The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1.</b>	(A   B) = 61, i.e., 0011 1101
^	Binary XOR <b>1 if the corresponding bits of two operands are opposite</b>	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement 'flipping' bits- 0 changed to 1 and 1 changed to 0	(~A) = -60, i.e., 1100 0100
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

## Assignment Operators

The following table lists the assignment operators supported by the C language

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	$C = A + B$ will assign the value of $A + B$ to $C$
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	$C /= A$ is equivalent to $C = C / A$
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	$C \% = A$ is equivalent to $C = C \% A$
<<=	Left shift AND assignment operator.	$C <<= 2$ is same as $C = C << 2$
>>=	Right shift AND assignment operator.	$C >>= 2$ is same as $C = C >> 2$
&=	Bitwise AND assignment operator.	$C \&= 2$ is same as $C = C \& 2$
^=	Bitwise exclusive OR and assignment operator.	$C \wedge= 2$ is same as $C = C \wedge 2$
=	Bitwise inclusive OR and assignment operator.	$C  = 2$ is same as $C = C   2$

## special Operators

Operator	Description	Example
sizeof()	Returns the size of a variable.	int a; sizeof(a), where a is integer, will return 2.
&	Returns the address of a variable.	&a; returns the actual address of the variable a .(0xFFA)
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

## Operators Precedence in C

For example,  $x = 7 + (3 * 2)$ ; here, x is assigned 13, not 20 because operator \* has a higher precedence than +, so it first gets multiplied with  $3*2$  and then adds into 7.

Table showing highest precedence to lowest precedence

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

## Expression

Expression is a combination of variables(like a,b,m,n..), constants(3,2,1) and operators(+,/\*).

Eg :  $c+d$

$x/y+b+a*a*a$

$3.14 * r * r$

Algebraic Expression	C Expression
$ab-c$	$a*b-c$
$(m+n)(k+j)$	$(m+n)*(k+j)$
$(ab/c)$	$a*b/c$
$3x^2+2x+1$	$3*x^2+2*x+1$

### Example Program

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

#### Program

```
int main()  
{  
int x=2,y=3,result;  
result=x*5+y*7;  
printf("result =:%d",result);  
return 0;  
}
```

#### Expression evaluation

```
result=x*5 + y*7;  
result=2*5 + 3*7;  
result=2*5 + 3*7;  
result=10 + 3*7;  
result=10 + 21;  
result=31;
```

### Example program -find greatest of 3 numbers

Example of logical(&& logical AND) and relational operators(>)

```
#include<stdio.h>  
int main()  
{  
    int num1,num2,num3;  
  
    printf("\nEnter value of a, b and c:");  
  
    scanf("%d %d %d",&a,&b,&c);  
  
    if((a>b)&&(a>c))  
        printf("\n %d is greatest",a);  
    else if(b>c)  
        printf("\n %d is greatest",b );  
    else  
        printf("\n %d is greatest",c);  
    return 0;  
}
```

### Example program -find odd or even number

Example of Arithmetic(% mod) and relational operators(==)

```
#include<stdio.h>  
int main()  
{  
    int num,result;  
    if(num%2==0)  
        printf("even number \n");  
    else  
        printf("odd number \n");  
    return 0;  
}
```

### **Bitwise XOR**

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a^b);
    return 0;
}
```

**Output = 21**

#### **Explanation**

12 = 00001100 (In Binary)  
25 = 00011001 (In Binary)

**Bitwise XOR Operation of 12 and 25**

00001100  
00011001

00010101 = 21 (In decimal)

### **Bitwise complement 1's compliment**

```
#include <stdio.h>
int main()
{
    printf("complement = %d\n", ~35);
    return 0;
}
```

**OutPut:**

**complement = 220**

#### **Explanation**

35 = 00100011 (In Binary)

**Bitwise complement Operation of 35**  
~ 00100011

11011100 = 220 (In decimal)

### **Bitwise AND and OR operator**

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("OutputAND = %d", a&b);
    printf("OutputOR = %d", a|b);
    return 0;
}
```

**OutputAND = 8**

**OutputOR = 29**

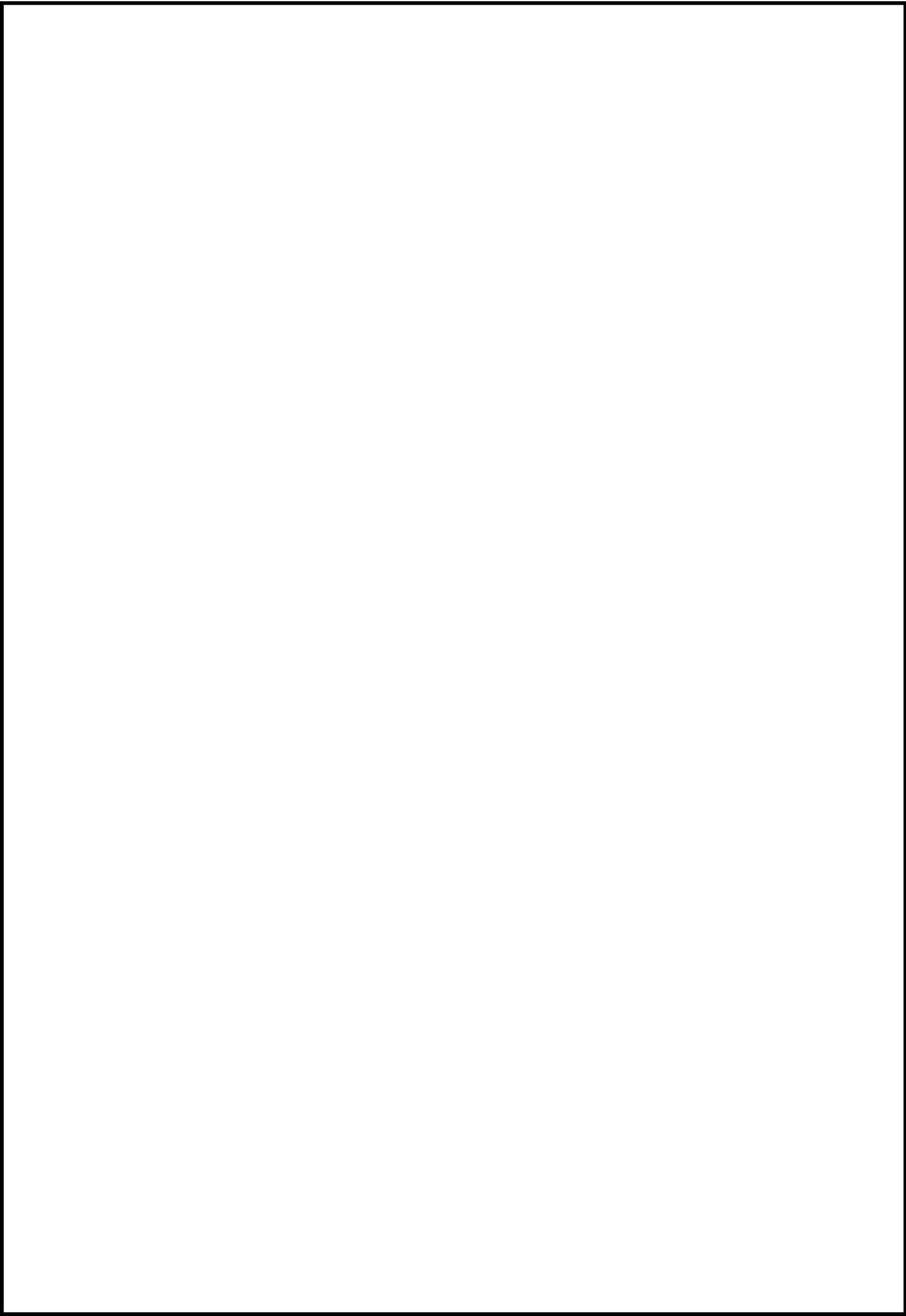
12 = 00001100 (In Binary)  
25 = 00011001 (In Binary)  
**Bitwise AND Operation of 12 and 25**  
00001100  
& 00011001

00001000 = 8 (In decimal)

**Bitwise OR Operation of 12 and 25**

00001100  
| 00011001

00011101 = 29 (In decimal)



## Decision Making and Branching

### Conditional Branching

- if statement
- nested if statement
- if ..else statement
- nested if else statement

### Conditional Branching

- break
- continue
- goto

These include conditional type branching and unconditional type branching.

### if statement

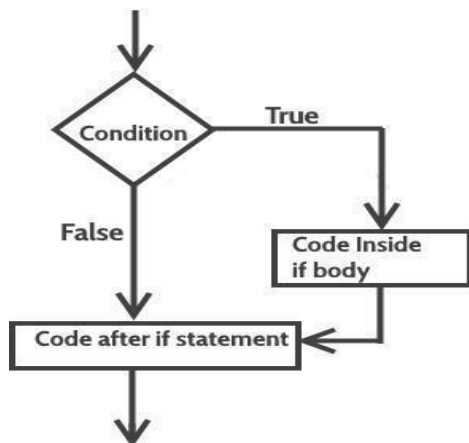
It takes the following form

```
if(test-expression)
```

It allows the computer to evaluate the expression first and then depending on whether the value of the expression is "true" or "false", it transfer the control to a particular statements. This point of program has two paths to flow, one for the true and the other for the false condition.

```
if(test-expression)
{
    statement-block
}
statement-x;
```

The statement-block may be a single statement or group of statements. If the test expression is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to the statement-x. But when is condition true both the statement-block and the statement-x are executed in sequence.



#### Eg-2

```
if (code == 1)
{
    salary = salary + 500;
}
printf("%d",salary);
```

#### Eg: Example program: C Program to check equivalence of two numbers using if statement

```
#include<stdio.h>
void main()
{
    int m,n;
    clrscr();
    printf(" \n enter two numbers:");
    scanf(" %d %d", &m, &n);
    if(m-n == 0)
    {
        printf(" \n two numbers are equal");
    }
    getch();
}
```

o/p

4 4

**two numbers are equal**

## Nested if

The syntax for a nested if statement is as follows

```
if( cond 1)
{
    /* Executes boolean expression when cond 1 is true */
    if(cond 2) {
        /* Executes when the boolean expression 2 is true */
    }
}
```

### Example:

```
#include <stdio.h>
int main ()
{
    int a = 100;
    int b = 200;
    if( a == 100 ) {
        /* if condition is true then check the following */
        if( b == 200 ) {
            /* if condition is true then print the following */
            printf("Value of a is 100 and b is 200\n" );
        }
    }
    printf("Exact value of a is : %d\n", a );
    printf("Exact value of b is : %d\n", b );
    return 0;
}
```

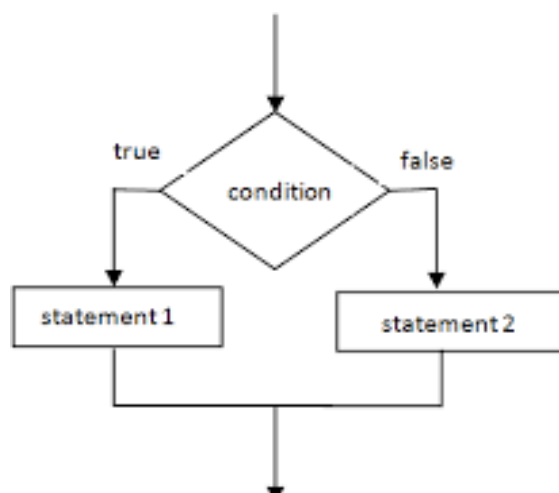
## The if-else statement

The if-else statement is an extension of the simple if statement. The general form is If the test- expression is true, then true-block statements immediately following if statement are executed otherwise the false-block statements are executed.

### Example: C program to find largest of two numbers

```
#include<stdio.h>
int main()
{
    int m,n,large;
    printf(" \n enter two numbers:"); scanf(" %d %d", &m, &n); if(m>n)
    large=
    m; else
    large=n
    ;
    printf(" \n large number is = %d", large); return 0;
}
```

```
if(test-expression)
{true-block statements
}
else
{
false-block statements
}statement-x
```



## Nested if-else statement

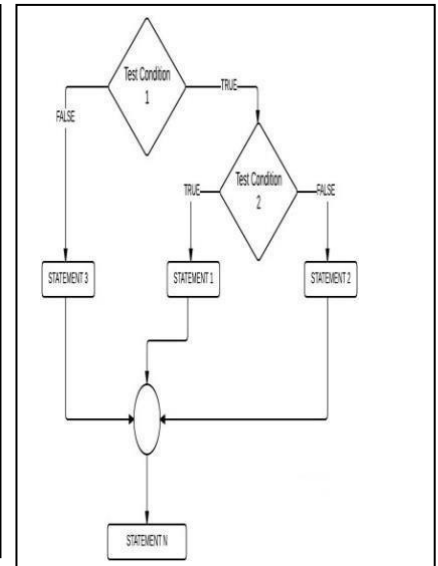
Nested if construct is also known as **if-else-if** construct.

Syntax-1

```
if(test-condition-1)
{
    (stmts)
}
else
{
    if(condition 2)
    {
        Statement-1;
    }
    else
    {
        statement-2;
    }
}
statement-x
```

Syntax-2

```
If(test-condition-1)
{
    if(test-condition-2)
    {
        statement-1;
    }
    else
    {
        statement-2;
    }
}
else
{
    statement-3;
}
statement-x
```



If the test-condition-1 is false, the statement-3 will be executed; other wise it continues the second test. If the condition-2 is true, the statement-2 will be evaluated and then the control is transferred to the statement-x.

Example: Program to relate two integers using =, > or <

```
#include <stdio.h>
int main()
{
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    //checks if two integers are equal.
    if(number1 == number2)
    {
        printf("Result: %d = %d", number1, number2);
    }

    //checks if number1 is greater than number2.
    else if (number1 > number2)
    {
        printf("Result: %d > %d", number1, number2);
    }

    // if both test expression is false
    else
    {
        printf("Result: %d < %d", number1, number2);
    }
    return 0;
}
```

## The switch statement:

When many conditions are to be checked then using nested if...else is very difficult, confusing and cumbersome. So C has another useful built-in decision-making statement known as switch.

This statement can be used as a multiway decision statement. The switch statement tests the value of a given variable or expression against a list of case values and when a match is found, a block of statements associated with that case is executed.

### Eg-1

```
int i = 1;
switch(i)
{
    case 1:
        printf("A");
        break;
    case 2:
        printf("B");
        break;
    case 3:
        printf("C");
        break;
    default:
}
```

```
switch( code)
{
    case 1:
        stmts1;
        break;
    case 2:
        stmts2;
        break;
    case 3:
        stmts3;
        break;
    default:
        stmtsx
}
```

### Example2: C program to find largest of two numbers

```
#include<stdio.h>

void main ()
{
    float basic , da ,
    salary ; int code ;
    char name[25];
    da=0.0;
    printf("Enter employee
    name\n");
    scanf("%s",&name);
    printf("Enter basic salary\n");
    scanf("%f",&basic);
    printf("Enter code of the
    Employee\n"); scanf("%d",&code);
    switch (code)
    {
        case 1:
            da = basic * 0.10;
            break;
        case 2:
            da = basic * 0.15;
            break;
        case 3:
            da = basic * 0.20;
            break;
        default :
            da = 0;
    }
    salary = basic + da;
    printf("Employee name
    is\n");
    printf("%s\n",name);
    printf ("DA is %f and Total salary is =%f\n",da,
    salary); getch();
}
```

For case 1, da=10% of basic salary. For case 2, da=15% of basic salary. For case 3, da=20% of basic salary. For default case >3 da is not

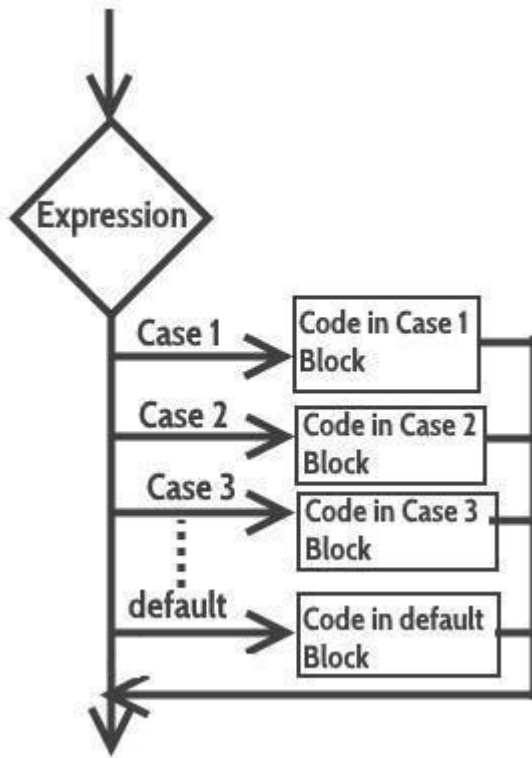
o/p

Enter name of employee:

Kartiyani  
Enter Basic  
salary 5000

Enter code of  
employee 1

Employee name  
is Kartiyani  
DA is 500 and total salary is 5500



### Rules for using `switch` statement

1. The expression (after `switch` keyword) must yield an **integer** value
2. The case **label** values must be unique.
3. The case label must end with a colon(:)

### Difference between `switch` and `if`

- `if` statements can evaluate `float` conditions. `switch` statements cannot evaluate `float` conditions.
- `if` statement can evaluate relational operators. `switch` statement cannot evaluate relational operators i.e they are not allowed in `switch` statement.

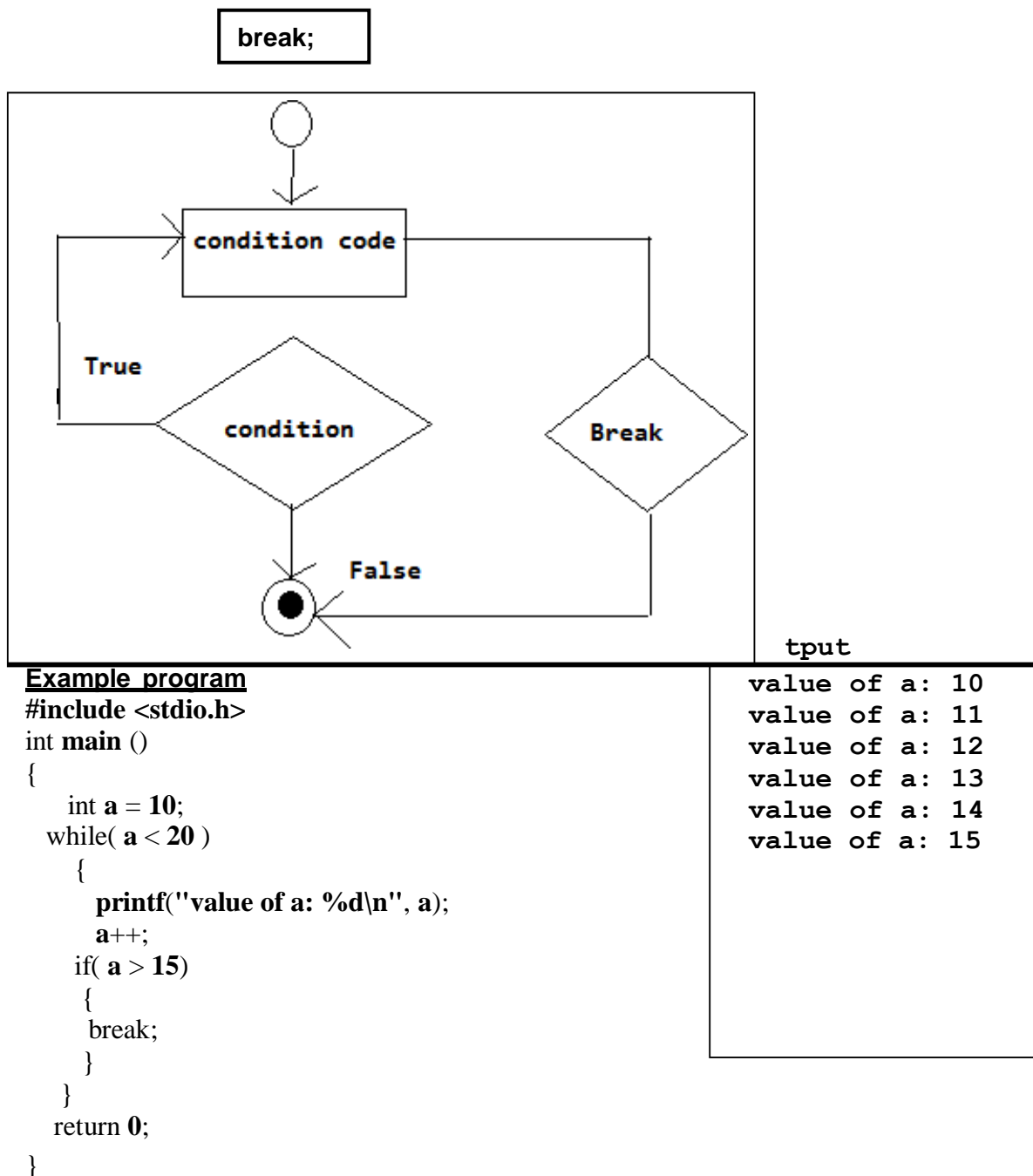
## **BREAK**

The break statement in C programming has the following two usages –

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

**BREAK** is a keyword that allows us to jump out of a loop instantly, without waiting to get back to the conditional test.

The syntax for a break statement in C is as follows –



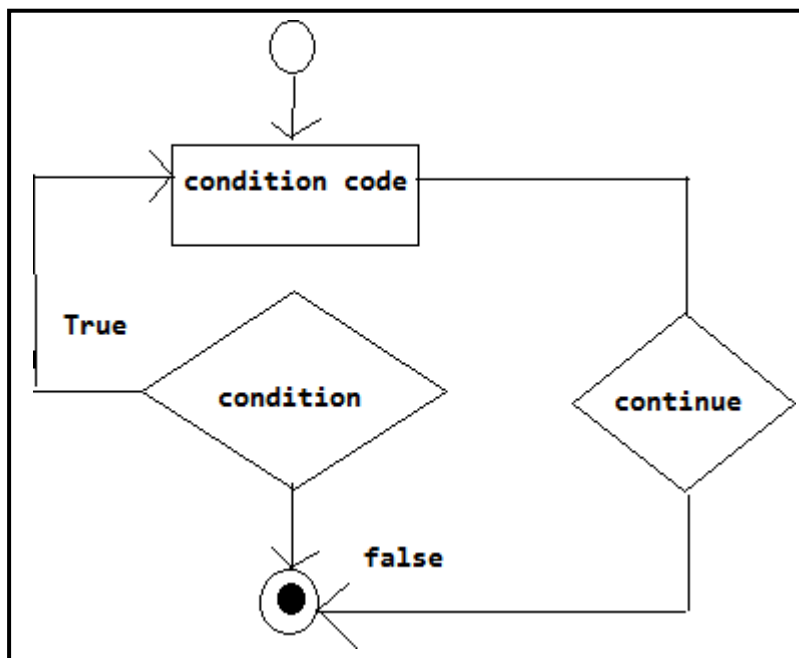
## Continue

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

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

Syntax:

```
continue;
```



### Example program

```
#include <stdio.h>
int main ()
{
    int a = 10;
    do {
        if( a == 15)
        {
            /* skip the iteration */
            a = a + 1;
            continue;
        }
        printf("value of a: %d\n", a);
        a++;
    } while( a < 20 );
    return 0;
}
```

#### Output

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

Break	Continue
The break statement can be used in both switch and loop (for, while, do while) statements.	The continue statement can appear only in loops. You will get an error if this appears in switch statement.
A break causes the switch or loop statements to terminate the moment it is executed. Loop or switch ends abruptly when break is encountered.	A continue doesn't terminate the loop, it causes the loop to go to the next iteration. The continue statement is used to skip statements in the loop that appear after the continue.
The break statement can be used in both switch and loop statements.	The continue statement can appear only in loops. You will get an error if this appears in switch statement.
When a break statement is encountered, it terminates the block and gets the control out of the switch or loop.	When a continue statement is encountered, it gets the control to the next iteration of the loop.

## **GOTO**

### **GOTO STATEMENT**

„C“ supports goto statement to branch unconditionally from one point to another in the program.

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

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

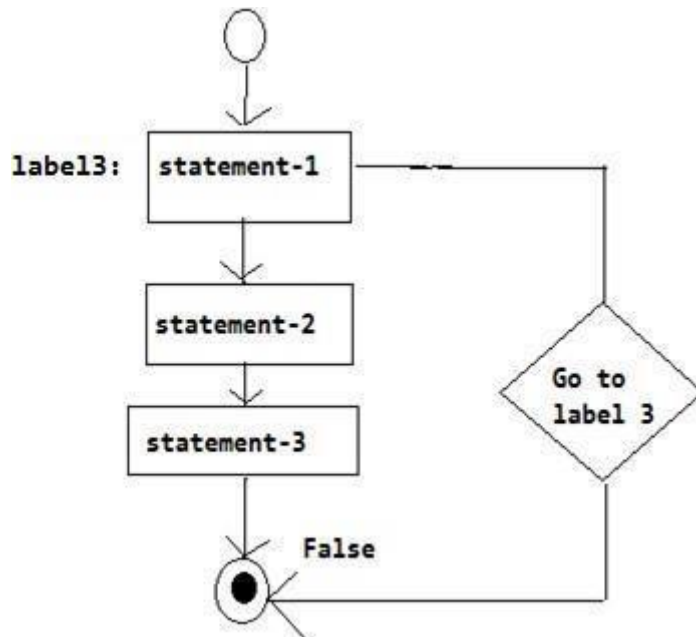
### **Syntax**

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

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

Or

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



#### Example program

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

```
int main ()
{
    int a = 10;
    ABCL:do
    {
        if( a == 15)
        {
            a = a + 1;
            goto ABCL;
        }
        printf("value of a: %d\n", a);
        a++;
    }while( a < 20 )
    return 0;
}
```

#### program to print..n" natural number

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

```
void main( )
```

```
{
    int n,i=1;
    clrscr();
    printf("enter number");
    scanf("%d\t",n);
    printf("natural numbers from 1 to %d", n);
    lb: printf("%d\t",i);
        i++;
    if(i<=n)
        goto lb;
    getch();
}
```

#### Output

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

## LOOPING STATEMENTS

Loop constructs supports repeated execution of statements when a condition match.

If the loop Test Condition is true, then the loop is executed, the sequence of statements to be executed is kept inside the curly braces { } is known as the **Loop body**. After every execution of the loop body, **condition** is verified, and if it is found to be **true** the loop body is executed again. When the condition check returns **false**, the loop body is not executed, and execution breaks out of the loop.

### **Types of Loop**

There are 3 types of Loop in C language, namely:

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

### **while loop**

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

```
while(condition)
{
    statements;
}
```

### **Example-- Example: //Program to print first 10 natural numbers**

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

**Output:** 1 2 3 4 5 6 7 8 9 10

## do while loop

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of do-while loop. do statement executes the body of the loop first and at the end, the condition is checked using while statement. It means that the body of the loop will be executed at least once, even though the starting condition inside while is initialized to be false.

### General syntax

#### **Example:// Program to print first 10 multiples of 5.**

```
#include<stdio.h>
> void main()
{
    int a,
    i; a =
    5;
    i = 1;
    do
    {
        printf("%d\t", a*i);
        i++;
    }
    while(i <= 12);
}
```

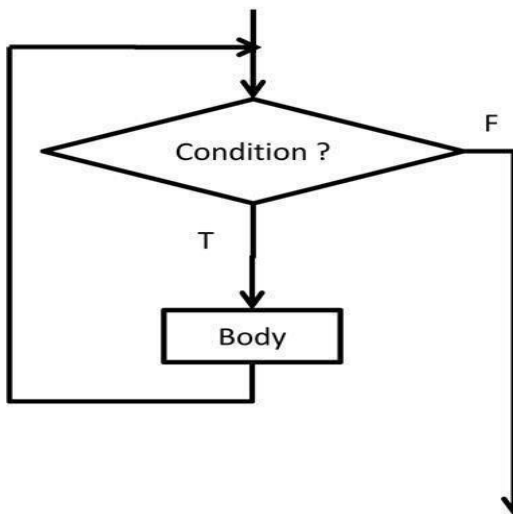
### Output

5 10 15 20 25 30 35 40 45 50 55 60

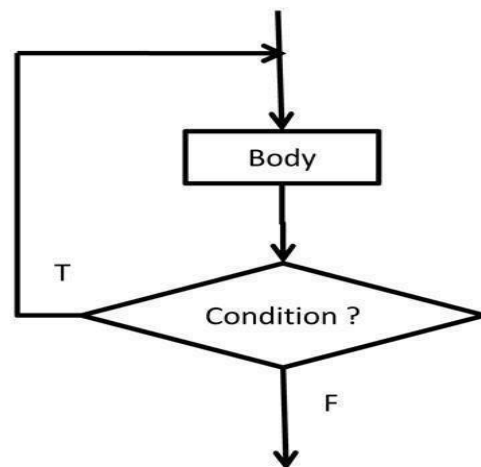
```
do
{
    .....
    .....
}
while(condition)
```

## While versus Do-While Loops

`while( condition )  
body;`



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



## Jumping Out of Loops

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop

### 1) **break statement**

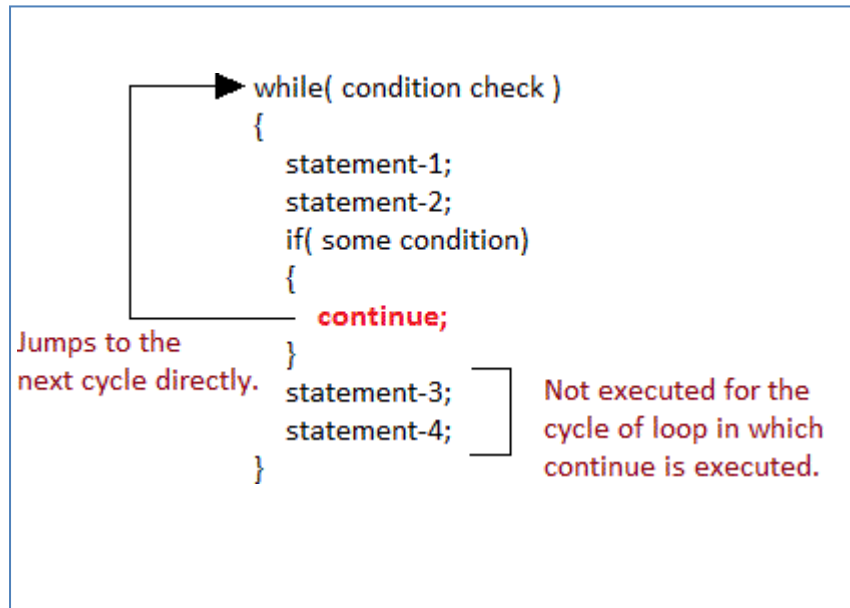
When break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

```
while( condition check )  
{  
    statement-1;  
    statement-2;  
    if( some condition )  
    {  
        break;  
    }  
    statement-3;  
    statement-4;  
}
```

**Jumps out of the loop, no matter how many cycles are left, loop is exited.**

## 2) continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.



## For Loop

for loop is used to execute a set of statements repeatedly until a particular condition is satisfied. We can say it is an **open ended loop**.. General format is,

```
for(initialization; condition; increment/decrement)
{
    statement-block;
}
```

In for loop we have exactly two semicolons, one after initialization and second after the condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. But it can have only one **condition**.

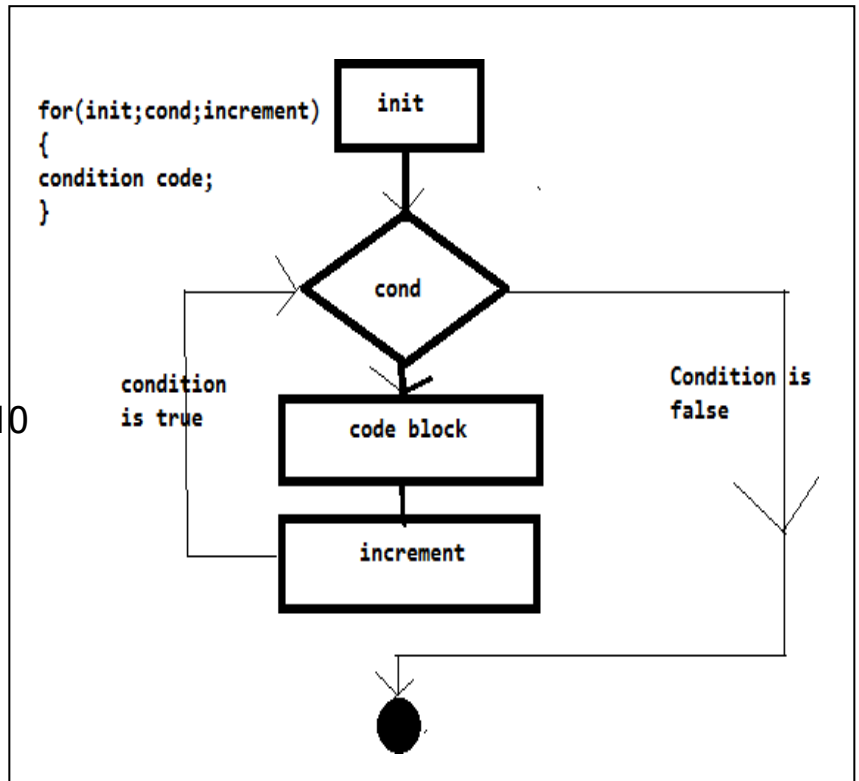
The for loop is executed as follows:

1. It first evaluates the initialization code.
2. Then it checks the condition expression.
3. If it is **true**, it executes the for-loop body.
4. Then it evaluate the increment/decrement condition and again follows from step 2.
5. When the condition expression becomes **false**, it exits the loop.

### Example: Program to print first 10 natural numbers

```
#include<stdio.h>
void main( )
{
    int x;
    for(x = 1; x <= 10; x++)
    {
        printf("%d\t", x);
    }
}
```

Output: 1 2 3 4 5 6 7 8 9 10



### Nested for loop

We can also have nested for loops, i.e one for loop inside another for loop. Basic syntax is,

```
for(initialization; condition; increment/decrement)
{
    for(initialization; condition; increment/decrement)
    {
        statement ;
    }
}
```

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

```
void main( )
```

```
{
```

```
    int i, j;
```

```
    for(i = 1; i < 5; i++) /* first for loop */
```

```
    {        printf("\n");
```

```
                /* second for loop inside the first
```

```
                */ for(j = i; j > 0; j--)
```

```
                {
```

```
                    printf("%d", j);
```

```
                }
```

```
    }
```

```
}
```

Output

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
```

## ARRAYS

### <Arrays – Initialization – Declaration – One dimensional and Two-dimensional arrays.>

1. ONE DIMENSIONAL ARRAY

2. TWO DIMENSIONAL ARRAY

3. STRING ARRAYS (ONE DIMENSIONAL ARRAY and TWO DIMENSIONAL ARRAY)

4. MULTIDIMENSIONAL ARRAYS

An **array** is a collection of similar data items, accessed using a common name. The collection of element can all be integers or be all decimal value or be all characters or be all strings.

- A one-dimensional **array** is like a list
- A two dimensional **array** is like a table
- The **C** language places no limits on the number of dimensions in an **array**

### ONE DIMENSIONAL ARRAY

## Array Declaration:

To declare an array in C, a programmer specifies the type of the elements and the number of elements required. The **arraySize** must be an integer constant greater than zero and **datatype** can be any valid C data type.

Syntax1:

```
datatype arrayName[ arraySize ];
```

### Example-1

```
int number[20];
int marks[44];
float salary[10];
double
value[25];
```

```
int n=25;
double x[n], y[n]; //array
                    declaration
```

```
#include<stdio.h>
#define N 100
int main( )
{
    int marks[N]; //array
    dec
    ....
    return 0;
}
```

```
int n;
scanf("%d",&n); //get size
int x[n]; //array declaration
```

```
int main( )
{
    int N=10,M=20;

    int marks[N*M]; //array dec
    ....

    return 0;
}
```

### Syntax-2

```
<storage class> datatype arrayName[ arraySize ];
```

Example: static int marks[20];

## Array initialization:

Example-1 `int mark[5] = {55, 66, 77, 88, 99};`

```
mark[0] = 55  
mark[1] = 66  
mark[2] = 77  
mark[3] = 88  
mark[4] = 99
```

	mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
array elements value	55	66	77	88	99
array index	0	1	2	3	4

Example-2 `double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};`

it means....

```
balance[0] = 1000.0;  
balance[1] = 2.0;  
balance[2] = 3.4;  
balance[3] = 7.0;  
balance[4] = 50.0;
```

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

### Automatic sizing

`int arr[] = {3,1,5,7,9};`

Here, the C compiler will deduce the size of the array automatically based on the number of

## **OPERATIONS ON ARRAYS**

- Traversing an array
- Inserting an element in an array
- Searching an element in an array
- Deleting an element from an array
- Merging two arrays
- Sorting an array in ascending or descending order

### **Working with one dimensional array**

#### **STORE and DISPLAY VALUES IN AN ARRAY (traversing an array)**

```
#include<stdio.h>
int main( )
{
    int k,array[10];//array declaration
    printf(—Enter the array elements:\n);
    for(k=0;k<5;k++)
    {
        scanf(—%d \n,&array[i]); // storing values in array
    }
    printf(—\n Display the array elements:\n);
    for(k=0;k<5;k++)
    {
        printf(—%d \n\n,array[i]);//displaying values of array
    }
    return 0;
}
```

#### **Output:**

```
Enter the array elements
2
4
3
1
8
Display the array elements
2
4
3
1
8
```

#### **FIND SUM AND AVERAGE OF N NUMBERS**

```
#include<stdio.h>
int main( )
{
    int k,n,sum=0;array[10];//array declaration
    float avg;
    printf(—\n Enter the array size:\n);
    scanf(—%d \n,&n);
    printf(—\n Enter the array elements:\n);
    for(k=0;k<n;k++)
    {
        scanf(—%d \n,&array[i]); // storing values in array
    }
    for(k=0;k<n;k++)
    {
        sum=sum+array[i]; //sum of array elements
    }
    avg=sum/n;
    printf(—\n sum=%d and avg=%f \n,sum,avg);
}
return 0;
```

#### **Output:**

```
Enter the array size: 6
Enter the array elements
9
2
4
3
1
8

sum=27 and avg=4.50000
```

#### **REVERSE OF ARRAY ELEMENTS**

```
#include<stdio.h>
int main( )
{
    int k,n,array[10];//array declaration
    printf(—\n Enter the array size:\n);
    scanf(—%d \n,&n);
    printf(—\n Enter the array elements:\n);
    for(k=0;k<n;k++)
```

#### **Output:**

```
Enter the arrayelements
2
4
3
1
8
Display the arrayelements
8
1
3
4
2
```

```

{
scanf("%d",&array[i]); // storing values in array
}
printf("\n array elements in reverse order:");
for(k=n-1;k>=0;k--)
{
printf("%d \n",array[i]); //displaying values of array
}
return 0;
}

```

**Write a program to print the position of the smallest number of  $n$  numbers using arrays.**

```


#include <stdio.h>
int main()
{
int i, n, arr[20], small, pos;
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements : ");
for(i=0;i<n;i++)
scanf("%d",&arr[i]);
small = arr[0]
for(i=1;i<n;i++)
{
if(arr[i]<small)
{
small = arr[i];
pos = i;
}
}
printf("\n The smallest element is : %d", small);
printf("\n The position of the smallest element in the array is:
%d", pos);
return 0;
}

```

**Output**

Enter the number of elements in the array : 5  
Enter the elements : 7 6 5 14 3  
The smallest element is : 3  
The position of the smallest element in the array is : 4

**Program example-1 Printing binary equivalent of a decimal number using array**

**Logic**  Here the remainders of the integer division of a decimal number by 2 are stored as consecutive array elements. The division procedure is repeated until the number becomes 0.

```

#include <stdio.h>
int main()
{
int bi[20],i,m,num,rem;
printf("\n Enter the decimal Integer");
scanf("%d",&n);
m=n;
for(i=0;i>n;i++)
{
rem=num%2;

```

**Output:**


Enter the decimal Integer: 12  
Binary equivalent of 12 is: 1100

```

bi[i]=rem;
num=num/2;
}
printf("\n Binary equivalent of %d is: \t\l,m);
for(i--;i>=0;i--)
printf(-%d\l,a[i]);
return 0;
}

```

### **Program example- Fibonacci series using an array**

**Logic**  In Fibonacci series each element is the sum of the previous two elements. The program stores the series in an array

```

#include <stdio.h>
int main()
{
int fib[15]; // array declaration
int i;
fib[0] = 0; // first array element value=0
fib[1] = 1; // second array element value=1
for(i = 2; i < 15; i++)
{
fib[i] = fib[i-1] + fib[i-2];
}
printf("\n Display the fibonacci elements:");
for(i = 0; i < 15; i++)
{
printf(-%d\l, fib[i]);
}
return 0;
}

```

#### **Display the fibonacci elements**

```

0
1
1
2
3
5
8
13
21
34
55
89
144
233
377

```

### **Example- Inserting an Element in an Array**

If an element has to be inserted at the end of an existing array, then the task of insertion is quite simple. We just have to add 1 to the **upper bound** and assign the value. Here, we assume that the memory space allocated for the array is still available. For example, if an array is declared to contain 10 elements, but currently it has only 8 elements, then obviously there is space to accommodate two more elements. But if it already has 10 elements, then we will not be able to add another element to it.

### **Program to insert a number at a given location in an array**

```

#include <stdio.h>
int main()
{
int i, n, num, pos, arr[10];
clrscr();
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
for(i=0; i<n; i++)
{
printf("\n arr[%d] = ", i);
scanf("%d", &arr[i]);
}
printf("\n Enter the number to be inserted : ");
scanf("%d", &num);
printf("\n Enter the position at which the number has to be added: ");

```

```

scanf("%d", &pos);
for(i=n-1; i>=pos; i--)
arr[i+1] = arr[i];
arr[pos] = num;
n = n+1;
printf("\n The array after insertion of %d is :num ");
for(i=0; i<n; i++)
printf("\n arr[%d] = %d", i, arr[i]);
getch();
return 0;
}

```

### Output

```

Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
Enter the number to be inserted : 0
Enter the position at which the number has to be added : 3
The array after insertion of 0 is :
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 0
arr[4] = 4
arr[5] = 5

```

### 3. Deleting an Element from an Array

Algorithm to delete an element from the middle of an array

```

Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N - 1
Step 3: SET A[I] = A[I + 1]
Step 4: SET I = I + 1
[END OF LOOP]
Step 5: SET N = N - 1
Step 6: EXIT

```

### Write a program to delete a number from a given location in an array.

```

#include <stdio.h>
int main()
{
int i, n, pos, arr[10];
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
for(i=0; i<n; i++)
{
printf("\n arr[%d] = ", i);
scanf("%d", &arr[i]);
}
printf("\n Enter the position from which the number has to be deleted : ");

```

```
scanf("%d", &pos);
for(i=pos; i<n-1;i++)
arr[i] = arr[i+1];
n--;
printf("\n The array after deletion is : ");
for(i=0;i<n;i++)
printf("\n arr[%d] = %d", i, arr[i]);
getch();
return 0;
}
```

**Output**

Enter the number of elements in the array :  
5

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

Enter the position from which the number  
has to be deleted : 3

The array after deletion is :

arr[0] = 1  
arr[1] = 2  
arr[2] = 3  
arr[3] = 5

## TWO DIMENTIONAL ARRAY

Two dimentional arrays stores data in tabular column format represented as rows and columns

### Array Declaration:

`datatype arrayname[size][size];`

### Array Initialization:

```
int a[2][2]={ {1,4 },{2,3}}
```

```
int b[2][2]={1,4,2,3}
```

```
1  4  
2  3
```

```
float[ ][ ]={12.3, 45.2,19.3,23.4}
```

```
12.3  45.2
```

```
19.3   23.4
```

### Accessing two-dimensional Arrays

#### Program-sample two dimnentional array

```
#include <stdio.h>  
int main()  
{  
    int i,j;  
    int a[3][2] = {{4,7},{1,0},{6,2}};  
    for(i = 0; i < 3; i++)  
    {  
        for(j = 0; j < 2; j++)  
        {  
            printf("%d", a[i][j]);  
        }  
        printf("\n");  
    }  
    return 0;  
}
```

Row-1	4	7
Row -2	1	0
Row-3	6	2

The above array actually „looks“ like this

1	2	3	4	5	6	7	8	9
Row 0			Row 1			Row 2		

## WORKING WITH TWO-DIMENSIONAL ARRAYS

### Transpose of a matrix

Example program:-Transpose of a matrix

Transpose of A is  $AT=(aji)$ , where  $i$  is the row number and  $j$  is the column number.

#### Program

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

```
int main()
```

```
{
    int a[10][10], transpose[10][10], r, c, i, j;
    printf("Enter rows and columns of matrix: ");
    scanf("%d %d", &r, &c);

    // getting elements of the matrix
    printf("\nEnter elements of matrix:\n");
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("Enter element a%d%d: ", i+1, j+1);
            scanf("%d", &a[i][j]);
        }

    // Displaying the matrix a[][] */
    printf("\n Entered Matrix: \n");
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("%d ", a[i][j]);
            if (j == c-1)
                printf("\n\n");
        }

    // Finding the transpose of matrix a
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            transpose[j][i] = a[i][j];
        }

    // Displaying the transpose of matrix a
    printf("\nTranspose of Matrix:\n");
    for(i=0; i<c; ++i)
        for(j=0; j<r; ++j)
        {
            printf("%d ", transpose[i][j]);
            if(j==r-1)
                printf("\n\n");
        }

    return 0;
}
```

$$A = \begin{pmatrix} 5 & 2 & 3 \\ 4 & 7 & 1 \\ 8 & 9 & 9 \end{pmatrix} \quad A^T = \begin{pmatrix} 5 & 4 & 8 \\ 2 & 7 & 1 \\ 3 & 1 & 9 \end{pmatrix}$$

#### Sample output

Enter rows and columns of matrix: 2  
3

Enter element of matrix:  
Enter element a11: 2  
Enter element a12: 3  
Enter element a13: 4  
Enter element a21: 5  
Enter element a22: 6  
Enter element a23: 4

Entered Matrix:  
2 3 4  
5 6 4

Transpose of Matrix:  
2 5  
3 6  
4 4

## Program -2 (Transpose)

```
#include <stdio.h>

void main()
{
    int array[10][10];
    int i, j, m, n;

    printf("Enter the order of the matrix \n");
    scanf("%d %d", &m, &n);
    printf("Enter the coefficients of the matrix\n");
    for (i = 0; i < m; ++i)
    {
        for (j = 0; j < n; ++j)
        {
            scanf("%d", &array[i][j]);
        }
    }
    printf("The given matrix is \n");
    for (i = 0; i < m; ++i)
    {
        for (j = 0; j < n; ++j)
        {
            printf(" %d", array[i][j]);
        }
        printf("\n");
    }
    printf("Transpose of matrix is \n");
    for (j = 0; j < n; ++j)
    {
        for (i = 0; i < m; ++i)
        {
            printf(" %d", array[i][j]);
        }
        printf("\n");
    }
}
```

\$ cc pgm85.c

\$ a.out

Enter the order of the matrix

3 3

Enter the coefficients of the matrix

3 7 9

2 7 5

6 3 4

The given matrix is

3 7 9

2 7 5

6 3 4

Transpose of matrix is

3 2 6

7 7 3

9 5 4

## **Matrix addition and subtraction**

**Addition** If  $A$  and  $B$  above are matrices of the same type

$$A + B = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 0 & 2 \end{pmatrix} + \begin{pmatrix} 2 & 1 & 2 \\ 1 & 0 & 3 \end{pmatrix} = \begin{pmatrix} 3 & 3 & 5 \\ 2 & 0 & 5 \end{pmatrix}$$

**Subtraction** If  $A$  and  $B$  are matrices of the same type, then

$$A - B = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 0 & 2 \end{pmatrix} - \begin{pmatrix} 2 & 1 & 2 \\ 1 & 0 & 3 \end{pmatrix} = \begin{pmatrix} -1 & 1 & 1 \\ 0 & 0 & -1 \end{pmatrix}$$

### **Program to Add Two Matrices**

```
#include <stdio.h>
int main()
{
    int r, c, a[100][100], b[100][100], sum[100][100], i, j;

    printf("Enter number of rows (between 1 and 100): ");
    scanf("%d", &r);
    printf("Enter number of columns (between 1 and 100): ");
    scanf("%d", &c);
    printf("\nEnter elements of 1st matrix:\n");
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("Enter element a%d%d: ", i+1, j+1);
            scanf("%d", &a[i][j]);
        }
    printf("Enter elements of 2nd matrix:\n");
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("Enter element a%d%d: ", i+1, j+1);
            scanf("%d", &b[i][j]);
        }
    // Adding Two matrices
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            sum[i][j] = a[i][j] + b[i][j];
        }
    // Displaying the result
    printf("\nSum of two matrix is: \n\n");
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("%d ", sum[i][j]);
            if(j==c-1)
            {
                printf("\n\n");
            }
        }
    return 0;
}
```

### **Output**

```
Enter number of rows (between
1 and 100): 2
Enter number of columns
(between 1 and 100): 3
```

```
Enter elements of 1st matrix:
Enter element a11: 2
Enter element a12: 3
Enter element a13: 4
Enter element a21: 5
Enter element a22: 2
Enter element a23: 3
Enter elements of 2nd matrix:
Enter element a11: -4
Enter element a12: 5
Enter element a13: 3
Enter element a21: 5
Enter element a22: 6
Enter element a23: 3
```

```
Sum of two matrix is:
```

```
-2    8    7
10    8    6
```

## **Matrix multiplication**

Matrix multiplication for two  $2 \times 2$  matrices.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} (ae+bg) & (af+bh) \\ (ce+dg) & (ef+dh) \end{pmatrix}$$

### ***Finding norm of a matrix***

The norm of a matrix is defined as the square root of the sum of the squares of the elements of a matrix.

```
#include <stdio.h>
#include <math.h>
#define row 10
#define col 10
int main()
{
    float mat[row][col], s;
    int i,j,r,c;
    printf("\n Input number of rows:");
    scanf("%d", &r);
    printf("\n Input number of cols:");
    scanf("%d", &c);
    for(i = 0 ; i< r; i++)
    {
        for(j = 0 ;j<c; j++)
        {
            scanf("%f", &mat[i][j]);
        }
    }
    printf("\n Entered 2D array is as follows:\n");
    for(i = 0; i < r; i++)
    {
        for(j = 0; j < c; j++)
        {
            printf("%f", mat[i][j]);
        }
        printf("\n");
    }
    s = 0.0;
    for(i = 0; i < r; i++)
    {
        for(j = 0; j < c; j++)
        {
            s += mat[i][j] * mat[i][j];
        }
    }
    printf("\n Norm of above matrix is: %f", sqrt(s));
    return 0;
}
```

## **C Program to read a matrix and find sum, product of all elements of two dimensional (matrix)**

### **array**

```
include <stdio.h>
#define MAXROW 10
#define MAXCOL 10
int main()
{
    int matrix[MAXROW][MAXCOL];
    int i,j,r,c;
    int sum,product;

    printf("Enter number of Rows :");
    scanf("%d",&r);
    printf("Enter number of Cols :");
    scanf("%d",&c);

    printf("\nEnter matrix elements :\n");
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
        {
            printf("Enter element [%d,%d] : ",i+1,j+1);
            scanf("%d",&matrix[i][j]);
        }
    }
    sum=0;
    product=1;
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
        {
            sum+=matrix[i][j];
            product*= matrix[i][j];
        }
    }
    printf("\nSUM of all elements : %d \nProduct of all elements :%d",sum,product);
    return 0;
}
```

Enter number of Rows :3  
Enter number of Cols :3

Enter matrix elements :  
Enter element [1,1] : 1  
Enter element [1,2] : 1  
Enter element [1,3] : 1  
Enter element [2,1] : 2  
Enter element [2,2] : 2  
Enter element [2,3] : 2  
Enter element [3,1] : 3  
Enter element [3,2] : 3  
Enter element [3,3] : 3

SUM of all elements : 18  
Product of all elements :216

### **Find the sum of diagonal elements of a matrix**

```
#include < stdio.h >
int main()
{
    int a[10][10],i,j,sum=0,r,c;
    clrscr();
    printf("\n Enter the number of rows and column ");
    scanf("%d%d",&r,&c);
    printf("\nEnter the %dX%d matrix",r,c);
    for(i=0;i < r;i++)
    {
        for(j=0;j < c;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    for(i=0;i < r;i++)
    {
        for(j=0;j < c;j++)
        {
            if(i==j)
            {
                sum+=a[i][j];
            }
        }
    }
}
```

1	2	3
2	4	6
3	5	8

**Sum of diagonal=13**

```

        }//for
        printf("\nThe sum of diagonal elements is %d",sum); return 0;
    }//main

```

### **Sum of rows and columns**

```

#include <stdio.h>
void main ()
{
    int array[10][10];
    int i, j, m, n, sum = 0;
    printf("Enter the order of the matrix\n");
    scanf("%d %d", &m, &n);
    printf("Enter the co-efficients of the matrix\n");
    for (i = 0; i < m; ++i)
    {
        for (j = 0; j < n; ++j)
        {
            scanf("%d", &array[i][j]);
        }
    }
    for (i = 0; i < m; ++i)
    {
        for (j = 0; j < n; ++j)
        {
            sum = sum + array[i][j] ;
        }
        printf("Sum of the %d row is = %d\n", i, sum);
        sum = 0;
    }
    sum = 0;
    for (j = 0; j < n; ++j)
    {
        for (i = 0; i < m; ++i)
        {
            sum = sum + array[i][j];
        }
        printf("Sum of the %d column is = %d\n", j, sum);
        sum = 0;
    }
}

```

### **Output**

```

Enter the order of the matrix
2 2
Enter the co-efficients of the matrix
23 45

80 97
Sum of the 0 row is = 68
Sum of the 1 row is = 177
Sum of the 0 column is = 103
Sum of the 1 column is = 142

```

### C Program to do the Sum of the Main & Opposite Diagonal Elements of a MxN Matrix

```
#include <stdio.h>
void main ()
{
    static int array[10][10];
    int i, j, m, n, a = 0, sum = 0;
    printf("Enter the order of the matrix \n");
    scanf("%d %d", &m, &n);
    if (m == n )
    {
        printf("Enter the co-efficients of the matrix\n");
        for (i = 0; i < m; ++i)
        {
            for (j = 0; j < n; ++j)
            {
                scanf("%d", &array[i][j]);
            }
        }
        printf("The given matrix is \n");
        for (i = 0; i < m; ++i)
        {
            for (j = 0; j < n; ++j)
            {
                printf(" %d", array[i][j]);
            }
            printf("\n");
        }

        for (i = 0; i < m; ++i)
        {
            sum = sum + array[i][i];
            a = a + array[i][m - i - 1];
        }
        printf("\nThe sum of the main diagonal elements is = %d\n", sum);
        printf("The sum of the off diagonal elements is = %d\n", a);
    }
    else
        printf("The given order is not square matrix\n");
}
```

```
Enter the order of the matrix
2 2
Enter the co-efficients of the matrix
40 30
38 90
The given matrix is
40 30
38 90

The sum of the main diagonal elements is
= 130
The sum of the off diagonal elements is
= 68
```

### C Program to Find the Frequency of Odd & Even Numbers in the given Matrix

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

    static int array[10][10];
    int i, j, m, n, even = 0, odd = 0;

    printf("Enter the order of the matrix \n");
    scanf("%d %d", &m, &n);

    printf("Enter the coefficients of matrix \n");
    for (i = 0; i < m; ++i)
    {
        for (j = 0; j < n; ++j)
        {
            scanf("%d", &array[i][j]);
            if ((array[i][j] % 2) == 0)
            {
                ++even;
            }
        }
    }
```

```

        }
        else
            ++odd;
    }

}

printf("The given matrix is \n");
for (i = 0; i < m; ++i)
{
    for (j = 0; j < n; ++j)
    {
        printf(" %d", array[i][j]);
    }
    printf("\n");
}
printf("\n The frequency of occurrence of odd number = %d \n", odd);
printf("The frequency of occurrence of even number = %d\n", even);

}

```

Enter the order of the matrix

3 3

Enter the coefficients of matrix

34 36 39

23 57 98

12 39 49

The given matrix is

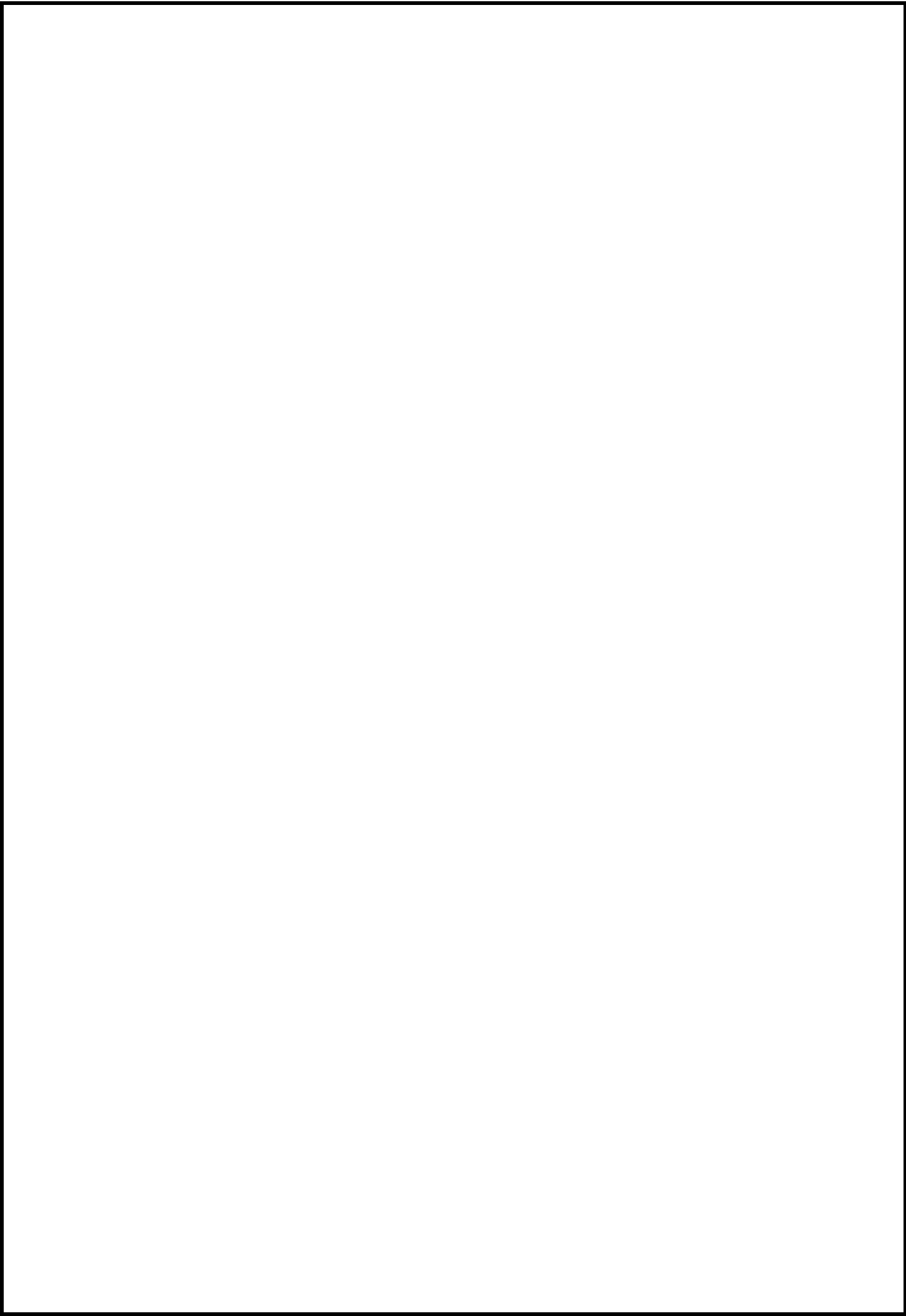
34 36 39

23 57 98

12 39 49

The frequency of occurrence of odd number = 5

The frequency of occurrence of even number = 4



## **SYLLABUS --UNIT II-FUNCTIONS, POINTERS, STRUCTURES AND UNIONS**

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

<b>S. No</b>	<b>Functions</b>
<b>1</b>	<b>Functions</b>
	Introduction
	Types of Function Implementation
	Passing Arguments(Parameters) To Function
	Call by Value
	Call by Reference
	Recursive Functions
	Example Programs using functions
<b>2</b>	<b>Pointers</b>
	Introduction --Definition --Declaration
	Pointer Arithmetic
	Pointers and arrays
	Pointers and Strings
	Pointers and Functions
<b>3</b>	<b>Structures</b>
	Intro
	Declaration— Initialization --Accessing
	Programs using structures
	Structure within Structure[Nested Structure]
	Arrays and Structures
	Structure and functions
	Pointers and structure
<b>4</b>	<b>Union</b>
<b>5</b>	<b>Union using Structure</b>
<b>6</b>	<b>Storage classes</b>
<b>7</b>	<b>Pre-processor directives.</b>

## FUNCTION

A function is self contained program segment that carries out some specific, well defined task.

### Need For Functions:

Many programs require a set of instructions to be executed repeatedly from several places in a program, then the repeated code can be placed within a single function, which can be accessed whenever it is required.

→ Dividing the program into separate well-defined functions facilitates each function to be written and tested separately.

→ Understanding, coding, and testing multiple separate functions is easier than doing the same for one big function main function.

→ A big program is broken into comparatively smaller functions.

### Advantages of Functions

- Improves readability of code.
- Divides a complex problem into simpler ones
- Improves reuseability of code.
- Debugging errors is easier.
- modifying a program is easier.

### Terms

→ A function  $f$  that uses another function  $g$  is known as the calling function, and  $g$  is known as the called function.

→ The inputs that a function takes are known as arguments.

→ When a called function returns some result back to the calling function, it is said to return that result.

→ The calling function may or may not function, which can then pass parameters to the called function.

→ **Function Declaration**

→ **Function Definition**

→ **Function Call**

### Function Definition

When a function is defined, space is allocated for that function in the memory. A function definition comprises of two parts:

- Function header
- Function body

The syntax of a function definition can be given as:

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{
    statements
    return(variable);
}
```

```
void sum(int a,int b)
{
    int c;
    c=a+b;
    return(c);
}
```

```
int sum(int a,int b)
{
    int c;
    c=a+b;
    return(c);
}
```

Note that the number of arguments and the order of arguments in the function header must be the same as that given in the function declaration statement.

## **Function Call**

The function call statement invokes the function. When a function is invoked, the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function. A function call statement has the following

Syntax-1:

```
function_name(variable1, variable2, ...);
```

```
function_name(argument1, argument2, ...);
```

If the return type of the function is not void then the following syntax can be used.

Syntax-2

```
variable_name = function_name(variable1, variable2, ...);
```

```
variable_name = function_name(parameter1, parameter2, ...);
```

The following points are to be noted while calling a function:

→Function name and the number and the type of arguments in the function call must be same as that given in the function declaration and the function header of the function definition.

→Names (and not the types) of variables in function declaration, function call, and header of function definition may vary.

## **Function Declaration**

It is also called as function prototype. A function prototype consists of 4 parts

- (1) Return type
- (2) function name
- (3) parameter list
- (4) terminating (;) semicolon

```
data_type function_name(data_type1 variable1, ...,data_type n variable n);
```

A function prototype tells the compiler that the function may later be used in the program. The function prototype is not needed if function is placed before main function code.

### Example program

```
#include <stdio.h>
void sum(int a,int b); //FUNCTION DECLARATION
int main()
{   int a,b;
    scanf("%d%d",&a,&b);    //get input values for a and b
    sum(a,b); //FUNCTION CALL
}
void sum(int x,int y) // FUNCTION DEFINITION
{
    int c;
    c=x+y;
    printf("c=%d",c);
}
```

o/p  
2  
3  
c=5

**void** data type indicates that the function is returning nothing .(i.e) if the function is not returning anything then its datatype is as specified as void.

### Example program without function prototype(function declaration)

```
#include <stdio.h>
void sum(int x,int y) // FUNCTION DEFINITION
{ int c;
  c=x+y;
  printf("c=%d",c);
}
int main()
{   int a,b;
    scanf("%d%d",&a,&b);    //get input values for a and b
    sum(a,b); //FUNCTION CALL
}
```

o/p  
2  
3  
c=5

### Eg-Write a program to find whether a number is even or odd using functions.

```
#include <stdio.h>
int evenodd(int); //FUNCTION DECLARATION
int main()
{
    int num, flag;
    printf("\n Enter the number : ");
    scanf("%d", &num);
    flag = evenodd(num); //FUNCTION CALL
    if (flag == 1)
        printf("\n %d is EVEN", num);
    else
        printf("\n %d is ODD", num);
    return 0;
}
int evenodd(int a) // FUNCTION DEFINITION
{
    if(a%2 == 0)
        return 1;
    else
        return 0;
}
```

**Output**  
Enter the number : 78  
78 is EVEN

#### TYPES OF FUNCTION IMPLEMENTATION

Not passing arguments & not returning anything

Not passing arguments & returning some value

passing arguments & not returning anything

passing arguments & returning some value

### **CASE-1 NOT PASSING ARGUMENTS & NOT RETURNING SOME VALUE**

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

```
int sum( ); //FUNCTION DECLARATION
```

```
int main( )
```

```
    int a,b,c;
    scanf("%d%d",&a,&b); // get input values for a and b
    c=x+y;
    printf("c=%d",c);
}
```

### **CASE-2 NOT PASSING ARGUMENTS & RETURNING SOME VALUE**

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

```
int sum( ); //FUNCTION DECLARATION
```

```
int main( )
```

```
{    sum( ); //FUNCTION CALL
    printf("c=%d",c);
}
```

```
int sum( ) // FUNCTION DEFINITION
```

```
{
    int a,b,c;
    scanf("%d%d",&a,&b); // get input values for a and b
    c=x+y;
    return c;
}
```

o/p  
2  
3  
c=5

### **CASE-3 PASSING ARGUMENTS & NOT RETURNING ANYTHING**

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

```
int sum( int a,int b); //FUNCTION DECLARATION
```

```
int main( )
```

```
{    int a,b;
    scanf("%d%d",&a,&b); // get input values for a and b
    sum(int a,int b); //FUNCTION CALL
}
```

```
int sum( int x,int y) // FUNCTION DEFINITION
```

```
{
    int c;
    c=x+y;
    printf("c=%d",c);
}
```

o/p  
2  
3  
c=5

### **CASE-4 NOT PASSING ARGUMENTS & RETURNING SOME VALUE**

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

```
int sum( int a,int b); //FUNCTION DECLARATION
```

```
int main( )
```

```
{    int a,b,c;
    scanf("%d%d",&a,&b); // get input values for a and b
    c=sum(int a,int b ); //FUNCTION CALL
    printf("c=%d",c);
}
```

```
return 0;
```

```
}
```

```
int sum( int x,int y) // FUNCTION DEFINITION
```

```
{    int result
    result=x+y;
    return result; }
```

o/p  
2  
3  
c=5

## PASSING ARGUMENTS(PARAMETERS) TO FUNCTION

In programming function argument is commonly referred as **actual parameter** and function parameter is referred as **formal parameter**.

```
void add(int num1, int num2) // Formal parameters // Function definition
{
    // Function body
}

int main()
{
    add(10, 20); // Actual parameters // Function call

    return 0;
}
```

There are two ways by which parameters can be passed to a function

1. Call by Value
2. Call by Reference

### Call by value

In Call by value, during function call actual parameter value is copied and passed to formal parameter. Changes made to the formal parameters does not affect the actual parameter.

#### Eg Program - C program to swap two numbers using call by value

```
#include<stdio.h>
int main()
{
    int n1, n2;
    printf("Enter two numbers: ");
    scanf("%d%d", &n1, &n2);
    printf("In Main values before swapping: %d %d\n", n1, n2);
    swap(n1, n2);
    printf("In Main values after swapping: %d %d", n1, n2);
    return 0;
}

void swap(int num1, int num2)
{
    int temp;
    printf("In Function values before swapping: %d %d\n", num1, num2);
    temp = num1;
    num1 = num2;
    num2 = temp;
    printf("In Function values after swapping: %d %d\n", num1, num2);
}
```

### **Output -**

Enter two numbers: 10 20

In Main values before swapping: 10 20

In Function values before swapping: 10 20

In Function values after swapping: 20 10

In Main values after swapping: 10 20

NOTE : In the above program swap() function does not alter actual parameter value. Before passing the value of *n1* and *n2* to the swap() function, the C runtime copies the value of actual parameter *n1* and *n2* to a temporary variable and passes copy of actual parameter. Therefore inside the swap() function values has been swapped, however original value of *n1* and *n2* in main() function remains unchanged.

### **Call by reference**

In Call by reference we pass memory location (reference) of actual parameter to formal parameter. It uses pointers to pass reference of an actual parameter to formal parameter. Changes made to the formal parameter immediately reflects to actual parameter.

#### **Eg Program - C program to swap two numbers using call by reference**

```
#include <stdio.h>
int main()
{   int n1, n2;
    printf("Enter two numbers: ");
    scanf("%d%d", &n1, &n2);
    printf("In Main values before swapping: %d %d\n\n", n1, n2);
    swap(&n1, &n2);
    printf("In Main values after swapping: %d %d", n1, n2);
    return 0;
}
void swap(int * num1, int * num2)
{
    int temp;
    printf("In Function values before swapping: %d %d\n", *num1, *num2);
    temp = *num1;
    *num1 = *num2;
    *num2 = temp;
    printf("In Function values after swapping: %d %d\n\n", *num1, *num2);
}
```

### **Output :-**

Enter two numbers: 10 20

In Main values before swapping: 10 20

In Function values before swapping: 10 20

In Function values after swapping: 20 10

In Main values after swapping: 20 10

In above example instead of passing a copy of *n1* and *n2*, to swap() function. Operations performed on formal parameter is reflected to actual parameter (original value). Hence, actual swapping is performed inside swap() as well as main() function.

## **WRITE THESE SWAP PROGRAM(S) FOR YOUR EXAMS**

### **Call by value**

```
#include<stdio.h>
Void swap(int x,int y);
int main( )
{
    int a = 10, b = 20 ;
    swap (a,b) ; // calling by value
    printf ( "\n Before swapping x and y) ;
    printf ( "\na = %d b = %d", a, b ) ;
    return 0;
}
void swap( int x, int y)
{
    int t ;
    t = x ;
    x = y ;
    y = t ;
    printf ( "\n After swapping x and y) ;
    printf( "\nx = %d y = %d", *x,*y);
}
```

<b>Before swapping x and y</b>	
10	20
<b>After swapping x and y</b>	
20	10

### **Call by reference**

```
#include<stdio.h>
Void swap(int *x,int *y);
int main( )
{
    int a = 10, b = 20 ;
    swap ( &a, &b ) ; // calling by reference
    printf ( "\n Before swapping x and y) ;
    printf ( "\na = %d b = %d", a, b ) ;
    return 0;
}
void swap( int *x, int *y)
{
    int t ;
    t = *x ;
    *x = *y ;
    *y = t ;
    printf ( "\n After swapping x and y) ;
    printf( "\nx = %d y = %d", *x,*y);
}
```

<b>Before swapping x and y</b>	
10	20
<b>After swapping x and y</b>	
20	10

<b><u>Call by value</u></b>	<b><u>Call by reference</u></b>
When a function is called the actual values are passed	When a function is called the address of values(arguments) are passed
The parameters passed are normal variables	The parameters passed are pointer variables
In call by value, actual arguments cannot be modified.	Modification to actual arguments is possible within from called function.
Actual arguments remain preserved and no chance of modification accidentally.	Actual arguments will not be preserved.
Calling Parameters: swap(a,b)	Calling Parameters: swap(&a,&b)
Receiving parameters: void swap( int x, int y)	Receiving parameters: void swap( int *x, int *y)

## PASSING ARRAYS TO A FUNCTION

### Passing Array Element to a Function

```
#include <stdio.h>
void display(int a);
int main()
{
    int age[] = { 21, 31, 41 };
    display(age[2]);    //Passing array element age[2] only.
    return 0;
}
void display(int age)
{
    printf("%d", age);
}
```

output

41

### Passing Entire Array(1-Dimentional array) Element to a Function

```
#include <stdio.h>
float average(float age[]);
int main()
{
    float avg, age[] = { 30.5,40.5,60.5,80.5 };
    avg = average(age);
    printf("Average age=%.2f", avg);
    return 0;
}
float average(float age[])
{
    int i;
    float avg, sum = 0.0;
    for (i = 0; i < 4; ++i)
    {
        sum += age[i];
    } avg = (sum / 4);
    return avg;
}
```

Output

53.000000

### Passing Entire Array(2-Dimentional array) Element to a Function

```
#include <stdio.h>
void displaynumbers(int num[2][2]);
int main( )
{
    int num[2][2],i,j;
    Printf("enter numbers");
    for(i=0;i<2;i++)
    for(i=0;i<2;i++)
        scanf("%d",&num[i][j]);
    displaynumbers(num) // calling function
    return 0;
}
void displaynumbers(int num[2][2]) // function declaration
{
    int i,j;
    Printf("display numbers");
    for(i=0;i<2;i++)
    for(i=0;i<2;i++)
        printf("%d",num[i][j]);
}
```

Output

Enter numbers  
10  
20  
30  
40  
Display numbers  
10  
20  
30  
40

## **RECURSION FUNCTION**

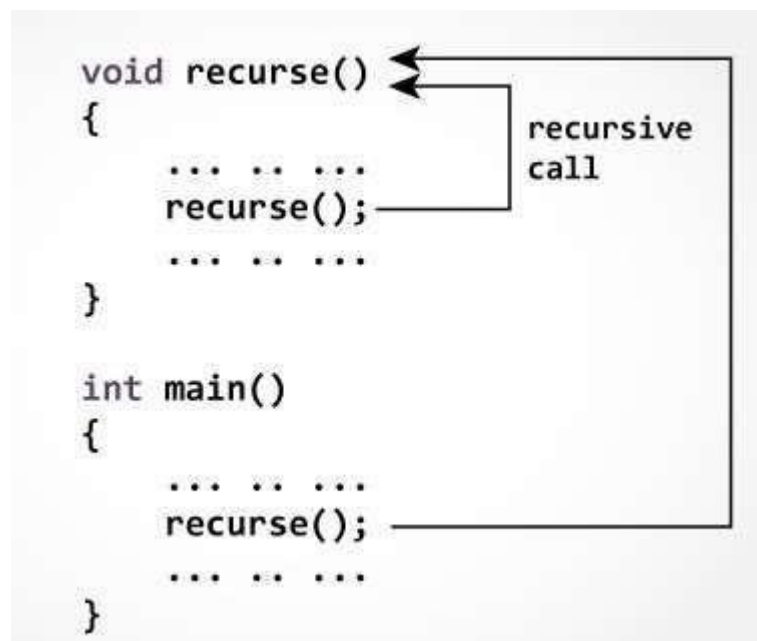
A function that calls itself is known as a recursive function. And, this technique is known as recursion.

### **Advantage of Recursion**

- Function calling related information will be maintained by recursion.
- Stack evaluation will be take place by using recursion.
- In fix prefix, post-fix notation will be evaluated by using recursion.

### **Disadvantage of Recursion**

- It is a very slow process due to stack overlapping.
- Recursive programs can create stack overflow.
- Recursive functions can create as loops.



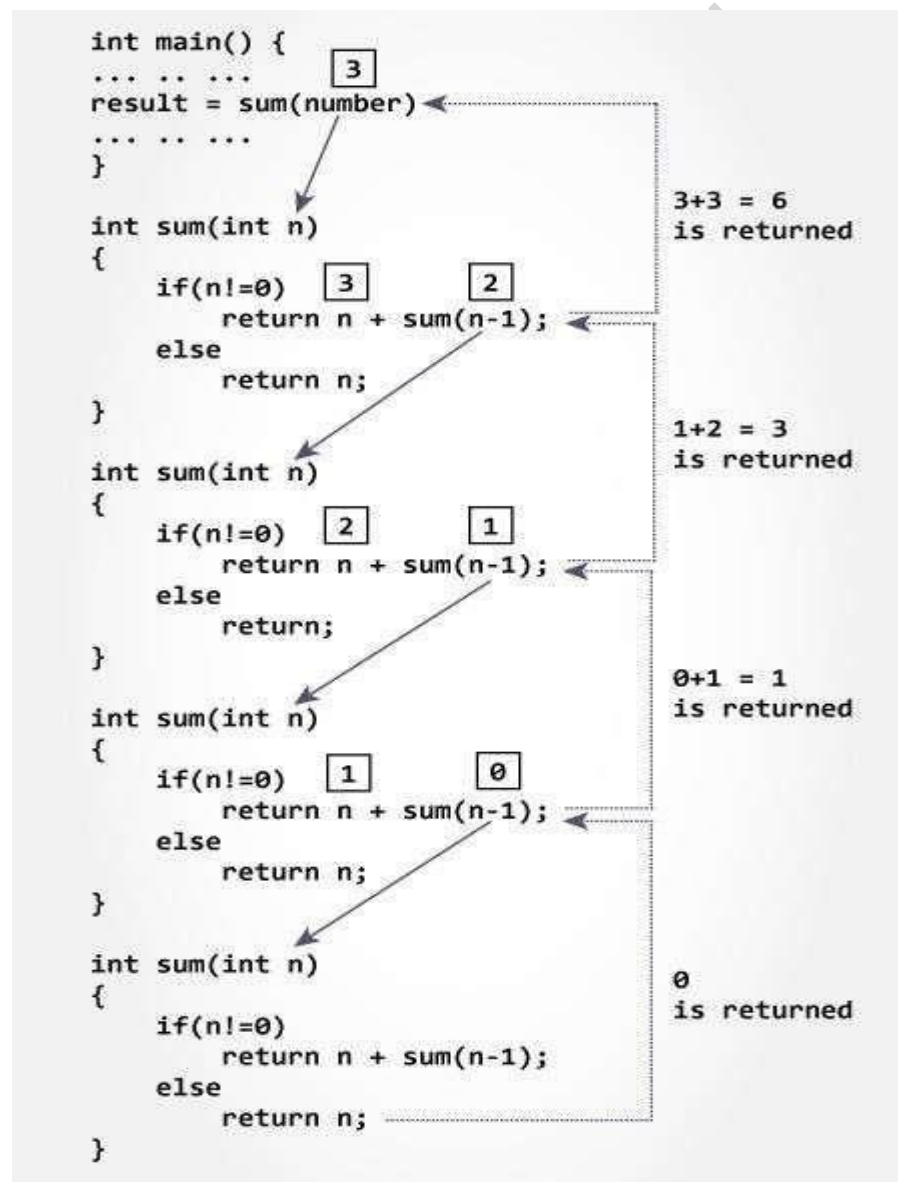
## Example: Sum of Natural Numbers Using Recursion

```
#include <stdio.h>
int sum(int n);
int main()
{
    int number, result;
    printf("Enter a positive integer: ");
    scanf("%d", &number);
    result = sum(number);
    printf("sum=%d", result);
}

int sum(int num)
{
    if (num!=0)
        return num + sum(num-1); // sum() function calls itself
    else
        return num;
}
```

### Output

```
Enter a positive integer:3
6
```



## FACTORIAL OF A NUMBER USING RECURSIVE FUNCTION

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

```
int factorial( int ) ;
```

```
void main()
```

```
{
```

```
    int fact, n ;
```

```
    printf("Enter any positive integer: ") ;
```

```
    scanf("%d", &n) ;
```

```
    fact = factorial( n ) ;
```

```
    printf("Factorial of %d is %d", n, fact) ;
```

```
}
```

```
int factorial( int n )
```

```
{
```

```
    int temp ;
```

```
    if( n == 0 )
```

```
        return 1 ;
```

```
    else
```

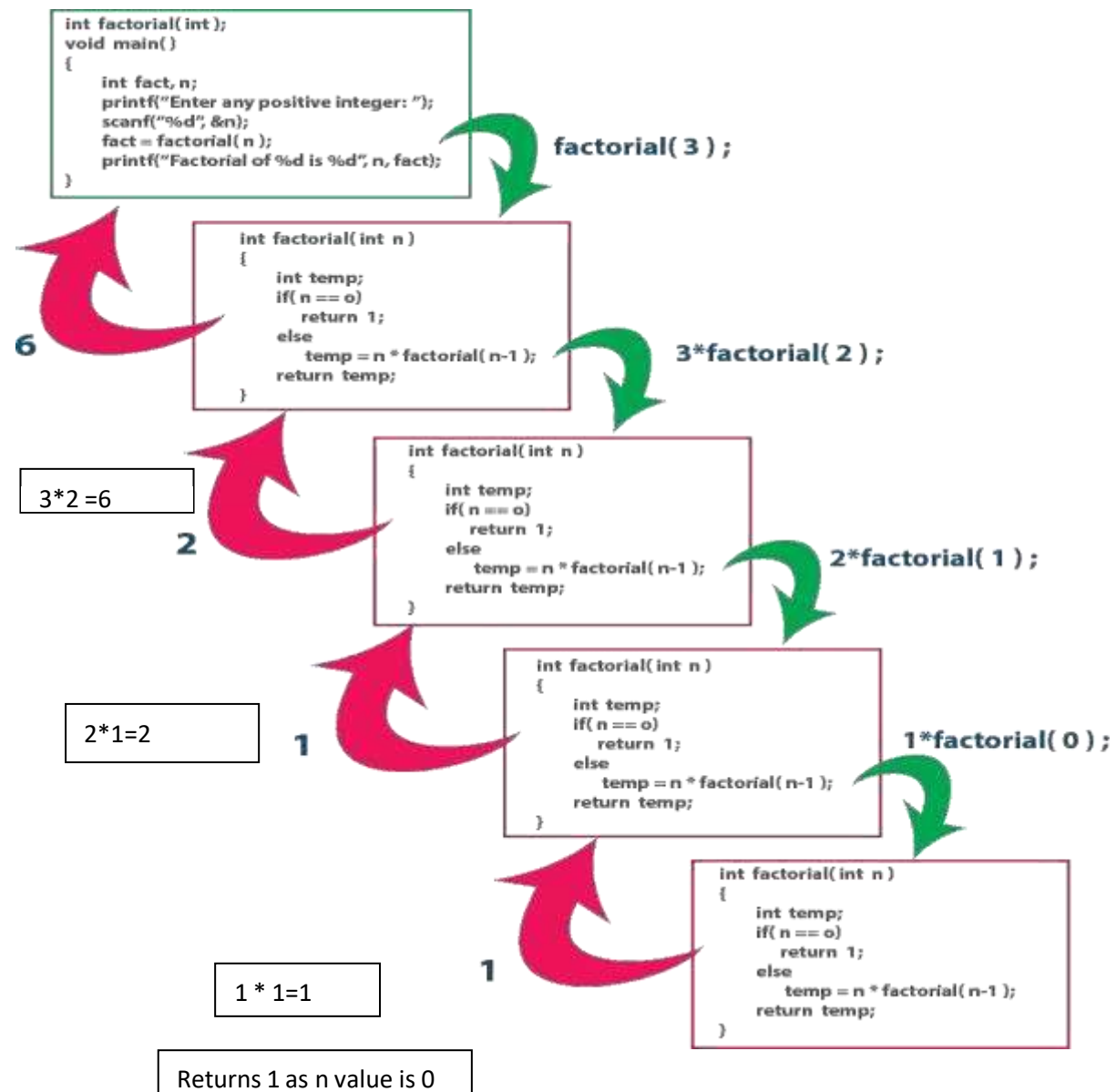
```
        temp = n * factorial( n-1 ) ; // recursive function call
```

```
    return temp ;
```

```
}
```

Enter any positive integer: 3

Factorial of 3 is 6



## TOWER OF HANOI OF A NUMBER USING RECURSIVE FUNCTION

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

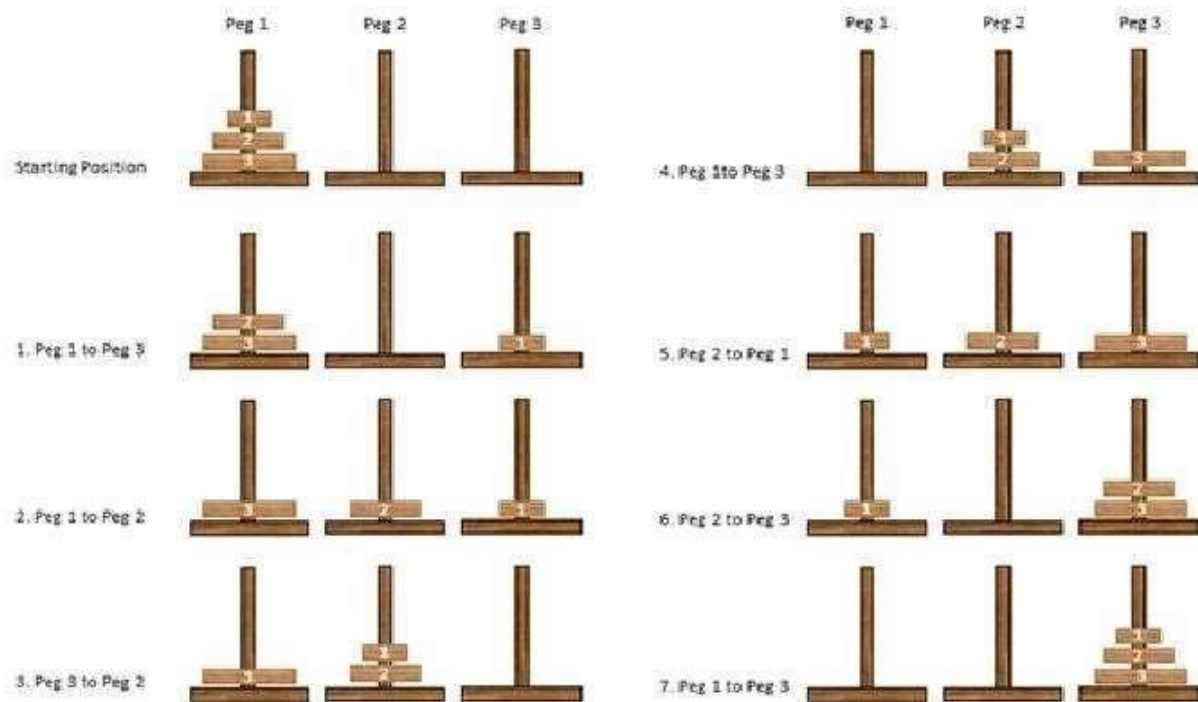
```
void hanoi(int n, char from, char to, char temp)
```

```
{  
    if (n == 1)  
    {  
        printf("\n Move Disk 1 from Peg %c to %c", from, to);  
        return;  
    }  
    hanoi(n-1, from, temp, to);  
    printf("\n Move disk %d from rod %c to rod %c", n, fr, tr);  
    hanoi(n-1, temp, to, from);  
}
```

```
int main()
```

```
{  
    Printf(" Towers of Honoi");  
    int n;  
    printf("\n Enter number of Disks");  
    scanf(" %d ",&n); // n implies the number of discs  
    hanoifun(n, 'A', 'C', 'B'); // A, B and C are the name of Peg  
    return 0;  
}
```

To Transfer from disks from peg A to C taking help of B



### Output

- Move Disk 1 from Peg 1 to Peg 3.
- Move Disk 2 from Peg 1 to Peg 2.
- Move Disk 1 from Peg 3 to Peg 2.
- Move Disk 3 from Peg 1 to Peg 3.
- Move Disk 1 from Peg 2 to Peg 1.
- Move Disk 2 from Peg 2 to Peg 3.
- Move Disk 1 from Peg 1 to Peg 3.

## Example: GCD of Two Numbers using Recursion

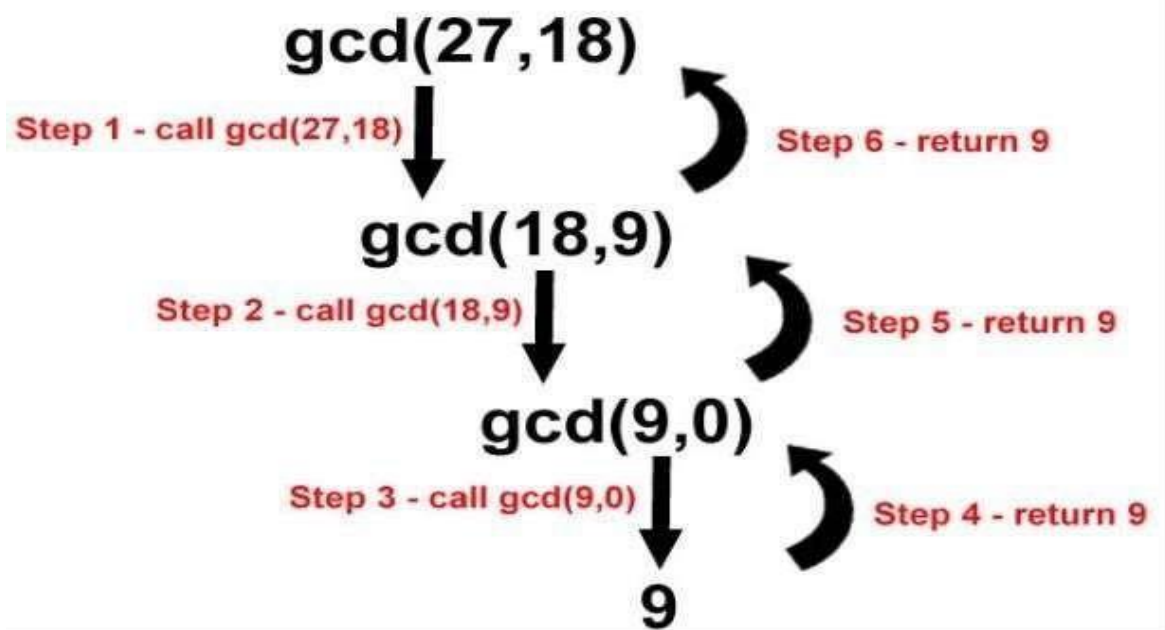
```
#include <stdio.h>
int hcf(int n1, int n2);
int main()
{
    int n1, n2;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);

    printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1,n2));
    return 0;
}

int hcf(int n1, int n2)
{
    if (n2 != 0)
        return hcf(n2, n1%n2);
    else
        return n1;
}
```

### Output

Enter two positive integers: 27 18  
G.C.D of 27 and 8 is 9.



### **Example: Fibonacci Series using Recursion**

```
#include <stdio.h>
int fibonacci(int i)
{
    if(i == 0)
    {
        return 0;
    }
    if(i == 1)
    {
        return 1;
    }
    return fibonacci(i-1) + fibonacci(i-2);
}

int main()
{
    int i,f;
    for (i = 0; i < 10; i++)
    {
        f= fibonacci(i);
        printf("%d \n ",f);
    }
    return 0;
}
```

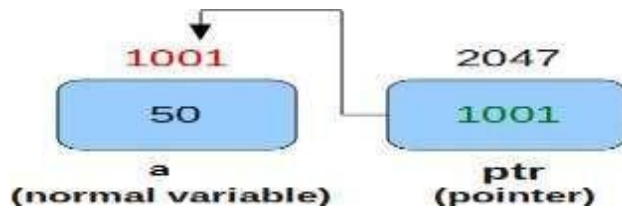
#### **Output**

0  
1  
1  
2  
3  
5  
8  
13  
21  
**34**

## Pointers in C Programming

A pointer is variable which stores address of another variable.

Pointers can only store addresses of other variable.



```
int x=10, y=20;  
printf("%u %u", &x, &y);
```

Note :Here %u is a format specifier. It stands for unsigned, so it will only display positive values.

**You will get output of the above program like below.**

605893 605897

### &-address of operator.

& is the “address of” operator. It is used to tell the C compiler to refer to the address of variables. Address of any variable can’t be negative. This is the reason %u format specifier is used to print the address of variables on the screen.

### value at address (\*) Operator

This is the second operator used for pointers. It is used to access the value present at some address. And it is used to declare a pointer.

#### Declaration and initialization of pointers

```
int x=10;  
int *ptr; // Declaration of Pointer variable  
ptr=&x; // Storing address of x variable in y pointer variable
```

### Example program-1

```
#include<stdio.h>
void main()
{
int a=6,b=12;
int *x,*y;
x=&a;
y=&b;
printf("%d t %d n",a,b);
printf("%u t %u n",&a,&b);
printf("%u t %u n",x,y);
printf("%d t %d n",*x,*y);
printf("%d t %d",(&a),(&b));
printf("%d t %d",*(&a),*(&b));
}
```

6	12
65524	65522
65524	65522
6	12
65524	65522
6	12

```
#include <stdio.h>
int main ()
{
    int var = 20;                /* actual variable declaration */
    int *ip;                    /* pointer variable declaration */
    ip = &var;                  /* store address of var in pointer variable*/
    printf("Address of var variable: %x\n", &var );
                                /*address stored in pointer variable*/
    printf("Address stored in ip variable: %x\n", ip );
                                /*access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );
    return 0;
}
```

A **pointer** is a variable whose value is the memory address of another variable

syntax

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable.

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

## NULLPointers

A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero.

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned.

```
#include <stdio.h>
int main ()
{
    int *ptr = NULL;
    printf("The value of ptr is : %x\n", ptr);
    return 0;
}
```

### output

The value of ptr is 0

## Incrementing a Pointer(32-bit )

```
#include <stdio.h>
const int MAX = 3;
int main () {int
var[] = {10, 100,
200};
    int i, *ptr;

                                /* let us have array address in pointer */
    ptr = var;
    for ( i = 0; i < MAX; i++) {
        printf("Address of var[%d] = %x\n", i, ptr );
        printf("Value of var[%d] = %d\n", i, *ptr );
                                /* move to the next location */
        ptr++;
    }

    return 0;
}
```

### output

Address of var[0] = bf882b30  
Value of var[0] = 10  
Address of var[1] = bf882b34  
Value of var[1] = 100  
Address of var[2] = bf882b38

## Decrementing a Pointer(32-bit machine)

decreases its value by the number of bytes of its data type.

```
#include <stdio.h>
const int MAX = 3;
int main () {
    int var[] = {10, 100, 200};
    int i, *ptr;

    /* let us have array address in pointer */
    ptr = &var[MAX-1];
    for ( i = MAX; i > 0; i--) {
        printf("Address of var[%d] = %x\n", i-1, ptr );
        printf("Value of var[%d] = %d\n", i-1, *ptr );
        /* move to the previous location */
        ptr--;
    }

    return 0;
}
```

### output

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

## Program for pointer arithmetic(32-bit machine)

```
#include <stdio.h>
int main()
{
    int m = 5, n = 10, val = 0;
    int *p1;
    int *p2;
    int *p3;

    p1 = &m;    //printing the address of m
    p2 = &n;    //printing the address of n

    printf("p1 = %d\n", p1);
    printf("p2 = %d\n", p2);

    printf(" *p1 = %d\n", *p1);
    printf(" *p2 = %d\n", *p2);

    val = *p1+*p2;
    printf("*p1+*p2 = %d\n", val); //point 1

    p3 = p1-p2;
    printf("p1 - p2 = %d\n", p3); //point 2
```

```
    p1++;  
    printf("p1++ = %d\n", p1); //point 3  
  
    p2--;  
    printf("p2-- = %d\n", p2); //point 4  
  
    return 0;  
}
```

#### OUTPUT

```
p1 = 2680016  
p2 = 2680012
```

```
*p1=5;  
*p2=10;
```

```
*p1+*p2 = 15
```

```
p1-p2 = 1  
p1++ = 2680020  
p2-- = 2680008
```

## STRUCTURES

Structure is a user-defined data type that can store related information (of different data types) together. The major difference between a structure and an array is that an array can store only information of same data type. A structure is therefore a collection of variables under a single name. The variables within a structure are of different data types.

- A structure is a collection of variables under a single name. These variables can be of different types. Therefore, a structure is a convenient way of grouping together several pieces of related information.
- Complex hierarchies can be created by nesting structures.

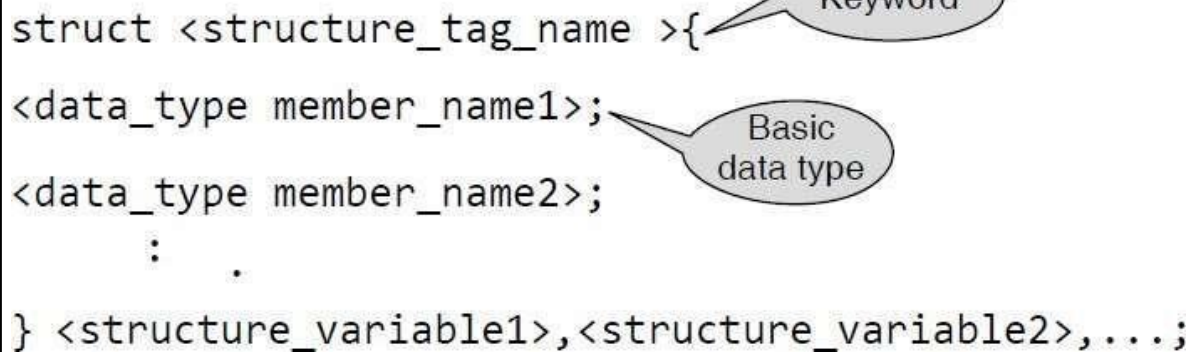
- Declaration/Initializing/Accessing
- Nesting of Structures
- Arrays of Structures
- Structures and Pointers
- Structures and Functions

### DECLARING STRUCTURES AND STRUCTURE VARIABLES

A structure is declared by using the **keyword struct** followed by an **optional structure tag** followed by the **body of the structure**. The **variables or members** of the structure are declared within the body.

The general format of declaring a simple structure is given as follows.

```
struct <structure_tag_name >{  
    <data_type member_name1>;  
    <data_type member_name2>;  
    :  
    .  
} <structure_variable1>, <structure_variable2>, ...;
```

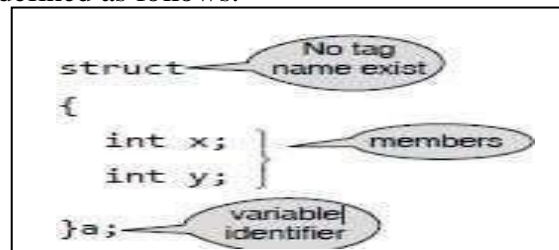


There are three different ways to declare and/or define a structure. These are

- Variable structure
- Tagged structure
- Type-defined structure

1. A variable structure may be defined as follows.

```
struct  
{  
    member_list  
}variable_identifier;
```

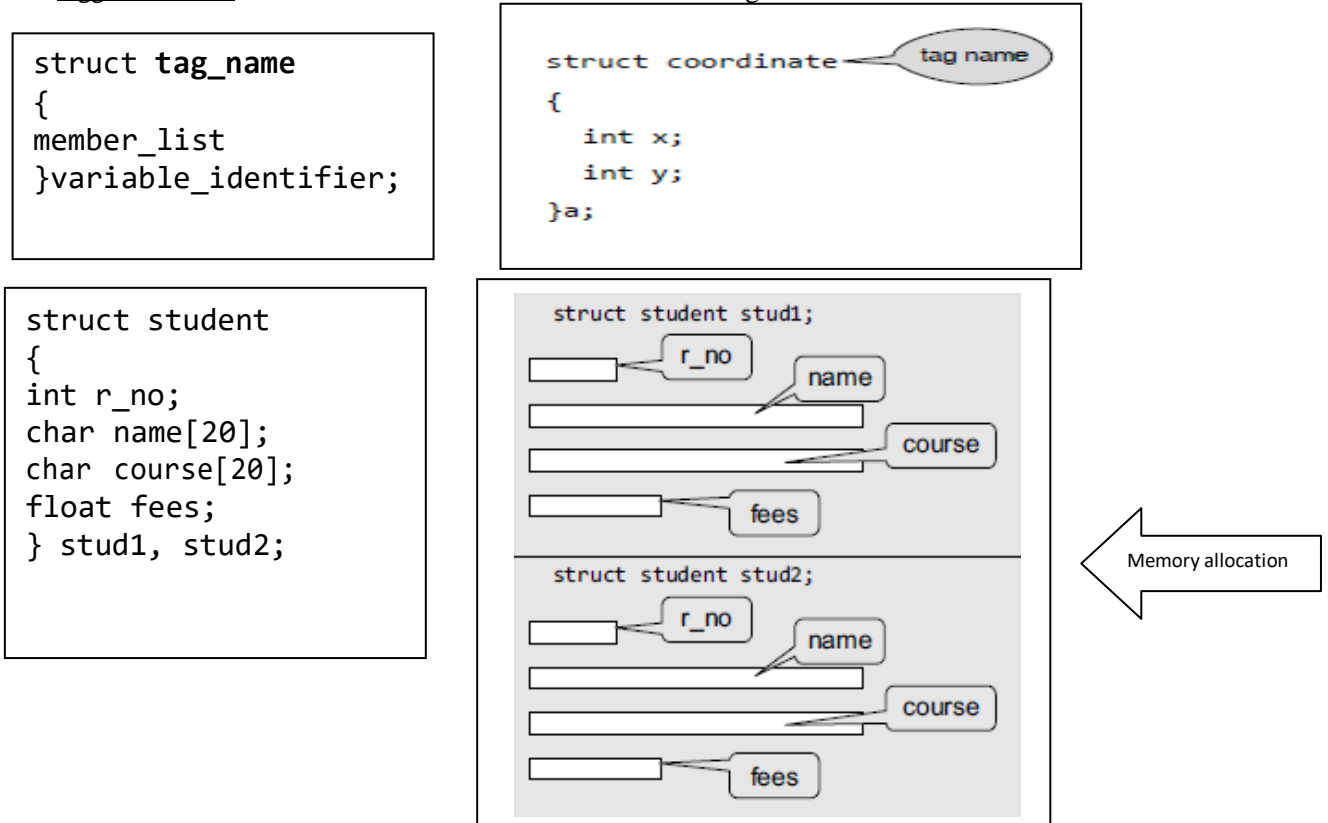


```
struct  
{  
    int r_no;  
    char name[20];  
    char course[20];  
    float fees;  
} student1;
```

where. ....a, student1-are called as structure variables

...

2. A tagged structure has been described earlier. It has the following format:



3. Type-defined structure declaration is as follows

- typedef keyword enables the programmer to create a new data type name by using an existing data type.

**typedef existing\_data\_type new\_data\_type;**

```
typedef struct newdatatype
{
    member_list;
} newdatatype variable_identifier;
```

```
typedef struct student
{
    int r_no;
    char name[20];
    char course[20];
    float fees;
} student stud1;
```

When we precede a struct name with the **typedef** keyword, then the **struct** becomes a new type. **student** becomes a new data type. To declare a variable of structure student, you may write **student stud1**;

Note that we have not written **struct student stud1**.

## INITIALIZATION OF STRUCTURES

A structure can be initialized in the same way as other data types are initialized. Initializing a structure means assigning some constants to the members of the structure.

```
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    data_type member_name3;
    .....
}struct_var = {constant1, constant2, constant3,...};
```

### Example-1

```
struct student
{
    int r_no;
    char name[20];
    char course[20];
    float fees;
}stud1 = {01, "Rahul", "BCA", 45000};
```

```
struct student stud1
= {01, "Rahul", "BCA", 45000};
```

01	Rahul	BCA	45000
r_no	name	course	fees

## ACCESSING THE MEMBERS OF A STRUCTURE

The members are accessed by relating them to the structure variable with a dot operator. The general form of the statement for accessing a member of a structure is as follows.

```
< structure_variable >.< member_name > ;
```

```
stud1.r_no
stud1.name
stud1.course
```

```
Stud2.r_no
Stud2.name
Stud2.course
```

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. A structure member variable is generally accessed using a '.' (dot) operator.

```
stud1.r_no = 01;
stud1.name = "Rahul";
stud1.course = "BCA";
stud1.fees = 45000;
```

To input values for data members of the structure variable stud1, we may write

```
scanf("%d", &stud1.r_no);
scanf("%s", stud1.name);
```

Similarly, to print the values of structure variable stud1, we may write

```
printf("%s", stud1.course);
printf("%f", stud1.fees);
```

Example-2 both initializing and accessing member data

```
struct {  
float p, q,  
int r;  
} k = {k .p = 3.0, k.q = 7.9, k.r = 5};
```

**Write a program using structures to read and display the information about a student**

```
#include <stdio.h>  
  
struct student  
{  
int roll_no;  
char name[80];  
float fees;  
char DOB[80];  
};  
  
int main()  
{  
struct student stud1;  
  
printf("\n Enter the roll number : ");  
scanf("%d", &stud1.roll_no);  
  
printf("\n Enter the name : ");  
scanf("%s", stud1.name);  
  
printf("\n Enter the fees : ");  
scanf("%f", &stud1.fees);  
  
printf("\n Enter the DOB : ");  
scanf("%s", stud1.DOB);  
  
printf("\n *****STUDENT'S DETAILS *****");  
printf("\n ROLL No. = %d", stud1.roll_no);  
printf("\n NAME = %s", stud1.name);  
printf("\n FEES = %f", stud1.fees);  
printf("\n DOB = %s", stud1.DOB);  
getch();  
return 0;  
}
```

### **Output**

```
Enter the roll number : 01  
Enter the name : Rahul  
Enter the fees : 45000  
Enter the DOB : 25-09-1991  
*****STUDENT'S DETAILS *****  
ROLL No. = 01  
NAME = Rahul  
FEES = 45000.00  
DOB = 25-09-1991
```

## C program to read and print employee's record using structure

```
#include <stdio.h>

struct employee
{
    char   name[30];
    int    empId;
    float  salary;
};

int main()
{
    struct employee emp;           // declare structure variable
    printf("\nEnter details :\n");
    printf("\n *****");
    printf("Name ?");
    scanf("%s",emp.name);
    printf("ID ?");
    scanf("%d",&emp.empId);
    printf("Salary ?");
    scanf("%f",&emp.salary);

    /*print employee details*/

    printf("\n Employee detail is:");
    printf("\n *****");
    printf("Name: %s",emp.name);
    printf("Id: %d",emp.empId);
    printf("Salary: %f\n",emp.salary);
    return 0;
}
```

### Output

Enter details :

\*\*\*\*\*

Name ?:Mike

ID ?:1120

Salary ?:76543

Employee detail is:

\*\*\*\*\*

Name: Mike

ID: 1120

Salary: 76543.000000

### **Write a program to copy and compare structures for employee details**

// A structure can be assigned to another structure of the same type. Here is an example of assigning one structure to another.

```
#include <stdio.h>
struct employee
{
char grade;
int basic;
float allowance;
};

int main()
{
    struct employee ramesh={„B“, 6500, 812.5};    /* creating & initializing
    member of employee */

    struct employee vivek;                        /* creating another member of
    employee */

    vivek = ramesh;                               /* copy respective members of
    ramesh to vivek */

    printf(“\n vivek’s grade is %c, vivek.grade);
    printf(“\n vivek’s basic is Rs %d, vivek. Basic);
    printf(“\n vivek’s allowance is Rs %f”, vivek.allowance);
    return 0;
}
```

#### **Output:**

```
vivek’s grade is B
vivek’s basic is Rs 6500
vivek’s allowance is Rs
812.500000
```

**Using Typedef :- A program that prints the weight of various sizes of fruits.**

```
#include <stdio.h>
typedef struct fruits
{
    float big;
    float medium;
    float small;
}fruits weight;

int main()
{
    weight apples={200.75,145.5,100.25};
    weight pears={150.50,125,50};
    weight mangoes={1000, 567.25, 360.25};

    printf("\n\n apples: big size %f kg, medium size %f kg, small size %f kg", apples.big,
    apples.medium, apples.small);

    printf("\n\n pears: big size %f kg, medium size %f kg,small size %f kg", pears.big,
    pears.medium, pears.small);

    printf("\n\n mangoes: big size %f kg, medium size %f kg,small size %f kg", mangoes.big,
    mangoes.medium, mangoes.small);
    return 0;
}
```

**Output:**

```
apples: big 200.75kg, medium 145.50kg, small 100.25kg
pears: big 150.50kg, medium 125.00kg, small 50.00kg
mangoes: big 1000kg, medium 567.25kg, small 360.25kg
```

## Nesting of Structures

A structure can be placed within another structure. In other words, structures can contain other structures as members. A structure within a structure means nesting of structures.

In such cases, the dot operator in conjunction with the structure variables are used to access the members of the innermost as well as the outermost structures.

```
typedef struct name
{
char first_name[20];
char mid_name[20];
char last_name[20];
}NAME;
```

```
typedef struct dob
{
int dd;
int mm;
int yy;
}DATE;
```

```
typedef struct student
{
int r_no;
NAME name;
char course[20];
DATE DOB;
float fees;
} STU;
```

```
assign values to the structure fields, we will write
student stud1;
stud1.r_no = 01;
stud1.name.first_name = "Janak";
stud1.name.mid_name = "Raj";
stud1.name.last_name = "Thareja";
stud1.course = "BCA";
stud1.DOB.dd = 15;
stud1.DOB.mm = 09;
stud1.DOB.yy = 1990;
stud1.fees = 45000;
```

**Write a program to read and display the information of a student using a nested structure.**

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

```
struct DOB
```

```
{  
int day;  
int month;  
int year;  
};
```

```
struct student
```

```
{  
int roll_no;  
char name[100];  
float fees;  
struct DOB date;  
};
```

```
int main()
```

```
{  
struct student stud1;
```

```
printf("\n Enter the roll number : ");  
scanf("%d", &stud1.roll_no);
```

```
printf("\n Enter the name : ");  
scanf("%s", stud1.name);
```

```
printf("\n Enter the fees : ");  
scanf("%f", &stud1.fees);
```

```
printf("\n Enter the DOB : ");  
scanf("%d %d %d", &stud1.date.day, &stud1.date.month, &stud1.date.year);
```

```
printf("\n *****STUDENT'S DETAILS *****");  
printf("\n ROLL No. = %d", stud1.roll_no);  
printf("\n NAME = %s", stud1.name);  
printf("\n FEES = %f", stud1.fees);  
printf("\n DOB = %d - %d - %d", stud1.date.day, stud1.date.month, stud1.date.year);  
getch();  
return 0;  
}
```

### **Output**

Enter the roll number : 01

Enter the name : Rahul

Enter the fees : 45000

Enter the DOB : 25 09 1991

\*\*\*\*\*STUDENT'S DETAILS \*\*\*\*\*

ROLL No. = 01

NAME = Rahul

FEES = 45000.00

DOB = 25 - 09 - 1991

**Write A program to demonstrate nesting of structures and accessing structure members.**

```
#include <stdio.h>

struct outer                                /* declaration of outer structure */
{
    int out1;                               /* member of outer structure */
    float out2;                             /* member of outer structure */

    struct inner                            /* declaration of inner structure */
    {
        int in1;                           /* member of inner structure */
        float in2;                         /* member of inner structure */
    } invar;                               /* structure_variable of inner structure */
};

int main()
{
    struct outer outvar;                    /* declaring structure_variable of outer */
    outvar.out1= 2;                         /* assigning values to members */
    outvar.out2= 10.57;
    outvar.invar.in1= 2* outvar.out1;       /* assigning values to members */
    outvar.invar.in2= outvar.out2 + 3.65;

    printf(" out1=%d, out2=%f, in1=%d, in2=%f",outvar.out1, outvar.out2,outvar.invar.in1,
    outvar.invar.in2);
    return 0;
}
```

**Output:**

out1=2, out2= 10.57, in1=4, in2= 14.22

**Write a program to read, display, add, and subtract two complex numbers.**

```
#include <stdio.h>
#include <conio.h>
int main()
{
typedef struct complex
{
int real;
int imag;
}COMPLEX;
COMPLEX c1, c2, sum_c, sub_c;

int option;

do
{
printf("\n ***** MAIN MENU *****");
printf("\n 1. Read the complex numbers");
printf("\n 2. Display the complex numbers");
printf("\n 3. Add the complex numbers");
printf("\n 4. Subtract the complex numbers");
printf("\n 5. EXIT");
printf("\n Enter your option : ");

scanf("%d", &option);

switch(option)
{
case 1:
printf("\n Enter the real and imaginary parts of the first complex number : ");
scanf("%d %d", &c1.real, &c1.imag);
printf("\n Enter the real and imaginary parts of the second complex number : ");
scanf("%d %d", &c2.real, &c2.imag);
break;
case 2:
printf("\n The first complex number is : %d+%di", c1.real, c1.imag);
printf("\n The second complex number is : %d+%di", c2.real, c2.imag);
break;
case 3:
sum_c.real = c1.real + c2.real;
sum_c.imag = c1.imag + c2.imag;
printf("\n The sum of two complex numbers is : %d+%di", sum_c.real, sum_c.imag);
break;
case 4:
sub_c.real = c1.real - c2.real;
sub_c.imag = c1.imag - c2.imag;
printf("\n The difference between two complex numbers
is : %d+%di", sub_c.real, sub_c.imag);
break;
}
}while(option != 5);
return 0;
}
```

**Output**

\*\*\*\*\* MAIN MENU \*\*\*\*\*

1. Read the complex numbers
2. Display the complex numbers
3. Add the complex numbers
4. Subtract the complex numbers
5. EXIT

**Enter your option : 1**

Enter the real and imaginary parts of the first complex number : 4 5

Enter the real and imaginary parts of the second complex number : 2 3

**Enter your option : 2**

The first complex numbers is : 4+5i

The second complex numbers is : 2+3i

**Enter your option : 3**

The sum of two complex numbers is : 6+8i

**Enter your option : 4**

The difference between two complex numbers is : 2+2i

**Enter your option : 5**

## ARRAYS OF STRUCTURES

*Arrays of structures* means that the structure variable would be an array of objects, each of which contains the member elements declared within the structure construct.

### Why would need an array of structures

1. In a class, we do not have just one student. But there may be at least 60 students. So, the same definition of the structure can be used for all the 30 students. This would be possible when we make an array of structures.
2. Another example where an array of structures is desirable is in case of an organization. An organization has a number of employees. So, defining a separate structure for every employee is not a viable solution. So, here we can have a common structure definition for all the employees.

#### Syntax

```
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    data_type member_name3;
    .....
};
struct struct_name struct_var[index];
```

```
struct student
```

```
{
    int r_no;
    char name[20];
    char course[20];
    float fees;
};
```

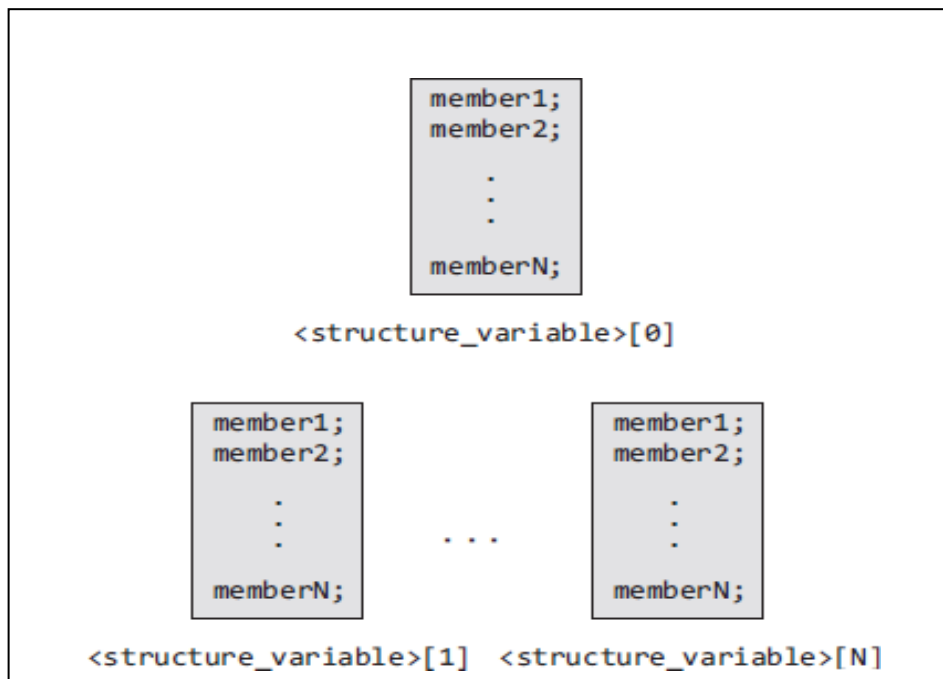
A student array can be declared by writing,  
struct student stud[30];

Now, to assign values to the i<sup>th</sup> student of the class, we can write as

```
stud[i].r_no = 09;
stud[i].name = "RASHI";
stud[i].course = "MCA";
stud[i].fees = 60000;
```

In order to initialize the array of structure variables

```
struct student stud[3][4] = {{01, "Aman", "BCA", 45000},{02,
"Aryan", "BCA", 60000}, {03,"John", "BCA", 45000}};
```



**Write a program to print the tickets of the boarders of a boat using array of structures with initialization in the program.**

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

```
struct boat    // declaration of structure //
```

```
{
char name[20];
int seatnum;
float fare;
};
```

```
int main()
```

```
{
```

```
int i;
```

```
struct boat ticket[4][3]={{"Vikram",1,15.50},{ "Krishna",2,15.50},
                           {"Ramu",3,25.50},{ "Gouri",4, 25.50 } };
```

```
printf("\n passenger   Ticket num.   Fare");
```

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

```
printf("\n %s %d %f", ticket[i].name,ticket[i].seatnum,ticket[i].fare);
```

```
return 0;
```

```
}
```

**Output:**

Passenger	Ticket num.	Fare
Vikram	1	15.500000
Krishna	2	15.500000
Ramu	3	25.500000
Gouri	4	25.500000

## C program to generate salary slip of employees using structures

```
#include<stdio.h>
struct emp
{
    int empno ;
    char name[10] ;
    int bpay, allow, ded, npay ;
} e[10] ;

void main()
{
    int i, n ;

    printf("Enter the number of employees : ") ;
    scanf("%d", &n) ;

    for(i = 0 ; i < n ; i++)
    {
        printf("\nEnter the employee number : ") ;
        scanf("%d", &e[i].empno) ;

        printf("\nEnter the name : ") ;
        scanf("%s", e[i].name) ;

        printf("\nEnter the basic pay, allowances & deductions : ") ;
        scanf("%d %d %d", &e[i].bpay, &e[i].allow, &e[i].ded) ;

        e[i].npay = e[i].bpay + e[i].allow - e[i].ded ;
    }
    printf("\nEmp. No.\t Name \t Salary \n") ;
    printf("\n *****") ;
    for(i = 0 ; i < n ; i++)
    {
        printf("%d \t %s \t %f \t ", e[i].empno, e[i].name, e[i].npay) ;
    }
    return 0;
}
```

Enter the employee number 2

Enter the employee number: 1001

Enter the employee number: Rina

Enter the basic pay, allowances & deductions: 75000 10000 2000

Enter the employee number: 2001

Enter the employee number: Bina

Enter the basic pay, allowances & deductions: 80000 10000 3000

Emp.No.	Name	NetSalary
1001	Rina	83000
2001	Bina	87000

**Write a program to read and display the information of all the students in a class. Then edit the details of the ith student and redisplay the entire information.**

```
#include <stdio.h>
#include <string.h>
struct student
{
int roll_no;
char name[80];
int fees;
char DOB[80];
};
int main()
{
struct student stud[50];
int n, i, num, new_rollno;
int new_fees;
char new_DOB[80], new_name[80];
clrscr(); // clear screen
printf("\n Enter the number of students : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n Enter the roll number : ");
scanf("%d", &stud[i].roll_no);

printf("\n Enter the name : ");
gets(stud[i].name);

printf("\n Enter the fees : ");
scanf("%d", &stud[i].fees);

printf("\n Enter the DOB : ");
gets(stud[i].DOB);
}

for(i=0;i<n;i++)
{
printf("\n *****DETAILS OF STUDENT %d*****", i+1);
printf("\n ROLL No. = %d", stud[i].roll_no);
printf("\n NAME = %s", stud[i].name);
printf("\n FEES = %d", stud[i].fees);
printf("\n DOB = %s", stud[i].DOB);
}

printf("\n Enter the student number whose record has to be edited : ");
scanf("%d", &num);

printf("\n Enter the new roll number : ");
scanf("%d", &new_rollno);

printf("\n Enter the new name : ");
gets(new_name);

printf("\n Enter the new fees : ");
scanf("%d", &new_fees);
```

```

printf("\n Enter the new DOB : ");
gets(new_DOB);

stud[num].roll_no = new_rollno;
strcpy(stud[num].name, new_name);
stud[num].fees = new_fees;
strcpy (stud[num].DOB, new_DOB);

for(i=0;i<n;i++)
{
printf("\n *****DETAILS OF STUDENT %d*****", i+1);
printf("\n ROLL No. = %d", stud[i].roll_no);
printf("\n NAME = %s", stud[i].name);
printf("\n FEES = %d", stud[i].fees);
printf("\n DOB = %s", stud[i].DOB);
}
getch();
return 0;
}

```

### Output

```

Enter the number of students : 2
Enter the roll number : 1
Enter the name : kirti
Enter the fees : 5678
Enter the DOB : 9- 9- 99

Enter the roll number : 2
Enter the name : kangana
Enter the fees : 5678
Enter the DOB : 27- 8- 99
*****DETAILS OF STUDENT 1*****
ROLL No. = 1
NAME = kirti
FEES = 5678
DOB = 9- 9- 99
*****DETAILS OF STUDENT 2*****
ROLL No. = 2
NAME = kangana
FEES = 5678
DOB = 27- 8 -99

Enter the student number whose record has to be edited : 2
Enter the new roll number : 2
Enter the new name : kangana khullar
Enter the new fees : 7000
Enter the new DOB : 27- 8 -92

*****DETAILS OF STUDENT 1*****
ROLL No. = 1
NAME = kirti
FEES = 5678
DOB = 9 -9 -99
*****DETAILS OF STUDENT 2*****
ROLL No. = 2
NAME = kangana khullar
FEES = 7000
DOB = 27- 8 -92

```

## **STRUCTURES AND FUNCTIONS**

Passing structures to functions

Passing the entire structure

Passing the address of structure

### **Eg: Passing Individual Members**

```
#include <stdio.h>
typedef struct
{
    int x;
    int y;
}POINT;
```

**void display(int, int); // function declaration**

```
int main()
{
    POINT p1 = {2, 3};
    display(p1.x, p1.y); // function call
    return 0;
}
```

#### **Output**

The coordinates of the point are: 2 3

```
void display(int a, int b) // function definition
{
    printf(" The coordinates of the point are: %d %d", a, b);
}
```

### **Eg: Passing the Entire Structure**

```
#include <stdio.h>
typedef struct
{
    int x;
    int y;
}POINT;
```

**void display(POINT);**

```
int main()
{
    POINT p1 = {2, 3};
    display(p1);
    return 0;
}

void display(POINT p)
{
    printf("The coordinates of the point are: %d %d", p.x, p.y);
}
```

- **Passing Structure by Value**
- **Passing Structure by Reference**

### **Passing Structure by Value**

In this approach, the structure object is passed as function argument to the definition of function, here object is representing the members of structure with their values.

#### **Program**

```
#include<stdio.h>

struct Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;
};

void Display(struct Employee);

void main()
{
    struct Employee Emp = { 1,"Kumar",29,45000};

    Display(Emp);
}

void Display(struct Employee E)
{
    printf("\n\nEmployee Id : %d",E.Id);
    printf("\nEmployee Name : %s",E.Name);
    printf("\nEmployee Age : %d",E.Age);
    printf("\nEmployee Salary : %ld",E.Salary);
}
```

#### **Output :**

```
Employee Id : 1
Employee Name : Kumar
Employee Age : 29
Employee Salary : 45000
```

## **Passing Structure by Reference**

In this approach, the reference/address structure object is passed as function argument to the definition of function.

### **Program**

```
#include<stdio.h>

struct Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;
};

void Display(struct Employee*);
void main()
{
    struct Employee Emp = { 1,"Kumar",29,45000};

    Display(&Emp);
}

void Display(struct Employee *E)
{
    printf("\n\nEmployee Id : %d",E->Id);
    printf("\nEmployee Name : %s",E->Name);
    printf("\nEmployee Age : %d",E->Age);
    printf("\nEmployee Salary : %ld",E->Salary);
}
```

### **Output :**

```
Employee Id : 1
Employee Name : Kumar
Employee Age : 29
Employee Salary : 45000
```

## Passing Structures through Pointers

```
struct struct_name
{
data_type member_name1;
data_type member_name2;
data_type member_name3;
.....
}*ptr;
```

or,  
struct struct\_name \*ptr;

we can declare a pointer variable by writing

```
struct student *ptr_stud, stud;
```

The next thing to do is to assign the address of stud to the pointer using the address operator (&).

```
ptr_stud = &stud;
```

To access the members of a structure, we can write

```
(*ptr_stud).roll_no;
```

(->) This operator is known as „pointing-to“ operator

```
ptr_stud -> roll_no = 01;
```

**Write a program to initialize the members of a structure by using a pointer to the structure.**

```
#include <stdio.h>

struct student
{
    int r_no;
    char name[20];
    char course[20];
    int fees;
};

int main()
{
    struct student stud1, *ptr_stud1;

    ptr_stud1 = &stud1;

    printf("\n Enter the details of the student :");
    printf("\n *****");

    printf("\n Enter the Roll Number =");
    scanf("%d", &ptr_stud1 -> r_no);

    printf("\n Enter the Name = ");
    gets(ptr_stud1 -> name);

    printf("\n Enter the Course = ");
    gets(ptr_stud1 -> course);

    printf("\n Enter the Fees = ");
    scanf("%d", &ptr_stud1 -> fees);

    printf("\n DETAILS OF THE STUDENT");
    printf("\n *****");

    printf("\n ROLL NUMBER = %d", ptr_stud1 -> r_no);
    printf("\n NAME = %s", ptr_stud1 -> name);
    printf("\n COURSE = %s", ptr_stud1 -> course);
    printf("\n FEES = %d", ptr_stud1 -> fees);
    return 0;
}
```

**Output**

Enter the details of the student:

\*\*\*\*\*

Enter the Roll Number = 02

Enter the Name = Aditya

Enter the Course = MCA

Enter the Fees = 60000

DETAILS OF THE STUDENT

\*\*\*\*\*

ROLL NUMBER = 02

NAME = Aditya

COURSE = MCA

FEES = 60000

## Storage Classes in C++ Programming

Storage class of a variable defines the **lifetime and visibility** of a variable. Lifetime means the duration till which the variable remains active and visibility defines in which module of the program the variable is accessible. They are:

1. Automatic
2. External
3. Static
4. Register

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage

### 1. Automatic Storage Class

This variable is **visible** only within the function it is declared and its **lifetime** is same as the lifetime of the function as well. This is the default storage class we have been using so far. It applies to local variables only and the variable is visible only inside the function in which it is declared and it dies as soon as the function execution is over. If not initialized, variables of class auto contains garbage value.

- The value is lost after the execution of function.

**Syntax:** `auto datatype var_name1 [= value];`

**Example:**

```
int var; // by default, storage class is auto
auto int var; // auto int j=10;
```

**Example Program:**

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

```
void display();
void main()
{
    auto int a=10;          //OR int a=10;
    printf("\n A1 : %d",a);
    display();
    printf("\n A3 : %d",a);
}
void display()
{
    int a=20;               //OR auto int a=20;
    printf("\n A2 : %d",a);
}
```

**Output :**

```
A1 : 10
A2 : 20
A3 : 10
```

## 2. External Storage Class

External storage class reference to a **global variable** declared outside the given program. *extern* keyword is used to declare external variables. They are **visible** throughout the program and its **lifetime** is same as the lifetime of the program where it is declared. This is visible to all the functions present in the program.

**Syntax :** `extern datatype var_name1;`  
**Example:** `extern float a;`

The *extern* keyword is optional, there is no need to write it.

- The scope of external variable is the entire program.
- If not initialized external variable is assigned a zero value.
- The value is not lost after the execution of function.

### Example Program:

```
#include<stdio.h>

void display();
extern int a=10;                                //global variable
void main()
{
    printf("\nA : %d",a);
    increment();
    display();
    printf("\nA : %d",a);

}
void increment()
{
    a = 20;
}
void display()
{
    printf("\nA : %d",a);
}
```

Output :

```
A : 10
A : 20
A : 20
```

### 3. Static Storage Class

Static storage class ensures a variable has the **visibility** mode of a local variable but **lifetime** of an external variable. It can be used only within the function where it is declared but destroyed only after the program execution has finished. The default initial value of static variable is 0. The value of a static variable persists between function calls

**Syntax:** static datatype var\_name1 [= value];

**Example:** static int x = 101;  
static float sum;

During multiple calling static variables retains their previous value.

- We must declare variable as static.
- Static variables can't be accessed outside the function.
- If not initialized static variables have zero as initial value.

### Example of static storage class

```
#include<stdio.h>
void display();
void main()
{
    display();
    display();
    display();
}
void display()
{
    static int a=1;
    printf("\nA : %d",a);
    a++;
}
```

#### Output:

A : 1  
A : 2  
A : 3

In the above example, we does not use static keyword then the output will be :

Output : A : 1  
A : 1  
A : 1

### 4. Register Storage Class

Variables of class 'register' are stored in CPU registers instead of memory which allows faster access. It has its lifetime and visibility same as automatic variable. The scope of the variable is local to the function in which it is defined and it dies as soon as the function execution is over. It contains some garbage value if not initialized. The purpose of creating register variable is to increase access speed and makes program run faster. As register space is very limited, so only those variables which requires fast access should be made register It is declared as:

**Syntax:** register datatype var\_name1 [= value];

**Example:** register int count  
register char a;

### **C program to create automatic, global(extern) variables.**

```
#include<stdio.h>
void main()
{
    register int a=10;
    printf("\nA : %d",a);
}
```

Output :

A : 10

## PREPROCESSOR DIRECTIVES

The C Preprocessor is not part of the compiler but it extends the power of C programming language. . The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control. The preprocessor's functionality comes before compilation of source code and it instruct the compiler to do required pre-processing before actual compilation. Working procedure of C program is shown in Fig. 2.8. In general, preprocessor directives

- begin with a # symbol
- do not end with semicolon
- are processed before compilation of source code

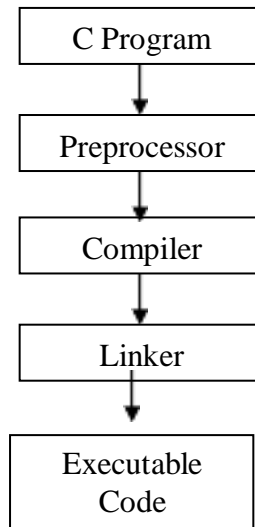


Fig. 2.8 Working Procedure of C Program

There are four types of Preprocessor Directives supported by C language. They are:

- File Inclusion directive
- Macro Substitution directive
- Conditional directive
- Miscellaneous directive

List of all possible directives belong to each of the above is listed in Fig 2.9.

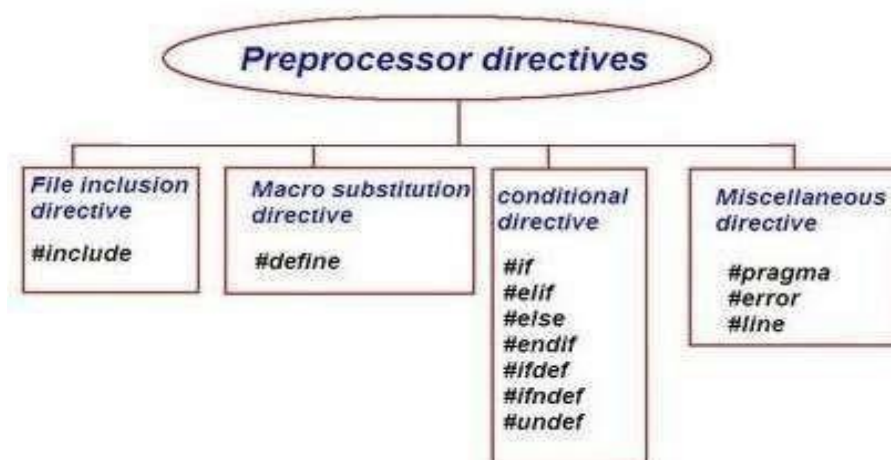


Fig Preprocessor Directives

The details of above listed preprocessor directives are narrated in Table.

Table Preprocessor directives and their description

Directive	Description
#include	It includes header file inside a C Program.
#define	It is substitution macro. It substitutes a constant with an expression.
#if	It includes a block of code depending upon the result of conditional expression.
#else	It is a complement of #if
#elif	#else and #if in one statement. It is similar to if else ladder.
#endif	It flags the end of conditional directives like #if, #elif etc.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true If constant is defined earlier using #define.
#ifndef	Returns true If constant is not defined earlier using #define.
#pragma	Issues special commands to the compiler.
#error	Prints error message on stderr.

### **File Inclusion directive**

#### **#include**

It is used to include header file inside C Program. It checks for header file in current directory, if path is not mentioned. To include user defined header file double quote is used (") instead of using triangular bracket (< >).

#### **Example:**

```
#include <stdio.h>           // Standard Header File
#include "big.h"             // User Defined Header File
```

Preprocessor replaces #include <stdio.h> with the content of stdio.h header file. #include "Sample.h" instructs the preprocessor to get Sample.h from the current directory and add the content of Sample.h file.

### **Macro Substitution directive**

#### **#define**

It is a simple substitution macro. It substitutes all occurrences of the constant and replace them with an expression. There are two types of macro supported by C. They are:

1. Simple macro
2. macro with arguments

#### **Simple macro**

#### **Syntax:**

```
#define identifier value
```

Where

- |            |  |
|------------|--|
| #define    | - is a preprocessor directive used for text substitution.  |
| identifier | - is an identifier used in program which will be replaced by value. (In general the identifiers are represented in capital letters in order to differentiate them from variable) |
| value      | - It is the value to be substituted for identifier.  |

**Example:**

```
#define PI 3.14
#define NULL 0
```

**Example:**

//Program to find the area of a circle using simple macro

```
#include <stdio.h>

#define PI 3.14

int main()
{
    int radius;
    float area;
    printf("Enter the radius of circle \n");
    scanf("%d", &radius);
    area= PI * radius * radius;
    printf("Area of Circle=%f", radius);
}
```

**Output**

Enter the radius of circle

10

Area of Circle = 314.000000

**macro with arguments**

#define Preprocessing directive can be used to write macro definitions with parameters. Whenever a macro identifier is encountered, the arguments are substituted by the actual arguments from the C program.

Data type definition is not necessary for macro arguments. Any numeric values like int, float etc can be passed as a macro argument . Specifically, argument macro is not case sensitive.

**Example:**

```
#define area(r) (3.14*r*r)
```

**Example:**

//Program to find the area of a circle using macro with arguments

```
#include <stdio.h>
#define area(r) (3.14*r*r)

int main()
{
    int radius;
    float a;
    printf("Enter the radius of circle \n");
    scanf("%d", &radius);
    a= area(radius);
    printf("Area of Circle=%f", a);
}
```

## Output

Enter the radius of circle

10

Area of Circle = 314.000000

## Predefined Macros in C Language

C Programming language defines a number of macros. Table 2.8 is the list of some commonly used macros in C

Table 2.8 Predefined macros in C

Macro	Description
NULL	Value of a null pointer constant.
EXIT_SUCCESS	Value for the exit function to return in case of successful completion of program.
EXIT_FAILURE	Value for the exit function to return in case of program termination due to failure.
RAND_MAX	Maximum value returned by the rand function.
__FILE__	Contains the current filename as a string.
__LINE__	Contains the current line number as an integer constant.
__DATE__	Contains current date in "MMM DD YYYY" format.
__TIME__	Contains current time in "HH:MM:SS" format.

## Example:

### // Program to print the values of Predefined macros

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("NULL : %d\n", NULL );
    printf("EXIT_SUCCESS : %d\n", EXIT_SUCCESS );
    printf("EXIT_FAILURE : %d\n", EXIT_FAILURE );
    printf("RAND_MAX : %d\n", RAND_MAX );
    printf("File Name : %s\n", __FILE_);
    printf("DATE : %s\n", __DATE__);
    printf("Line : %d\n", __LINE__);
    return 0;
}
```

## Output

NULL : 0

EXIT\_SUCCESS : 0

EXIT\_FAILURE : 1

RAND\_MAX : 32767

File Name : BuiltinMacro.c

DATE : Aug 16 2017

Line : 12

## **Conditional directive**

### **#if, #elif, #else and #endif**

The Conditional directives permit to include a block of code based on the result of conditional expression.

#### **Syntax:**

```
#if <expression>
    statements;
#elif <expression>
    statements;
#else
    statements;
#endif
```

Where

Expression represents a condition which produces a boolean value as a result.

Conditional directive is similar to if else condition but it is executed before compilation. Condition\_Expression must be only constant expression.

#### **Example:**

//Program to illustrate the conditional directives

```
#include <stdio.h>
#define A 10
int main()
{
    #if (A>5)
        printf("A=%d", X);
    #elif (A<5)
        printf("A=%d", 4);
    #else
        printf("A=%d", 0);
    #endif
    return 0;
}
```

**Output**  
**X=10**

### **#undef**

The #undef directive undefines a constant or preprocessor macro defined previously using #define.

#### **Syntax:**

```
#undef <Constant>
```

**Example:**

```
#include<stdio.h>
#define P 100
#ifdef P
    #undef P
    #define P 30
#else
    #define P 100
#endif
int main()
{
    printf("%d",P);
    return 0;
}
```

**Output**  
30

### **#ifdef #ifndef, #ifndef**

**#ifdef**

**#ifdef directive is used to check whether the identifier is currently defined. Identifiers can be defined by a #define directive or on the command line.**

**#ifndef**

**#ifndef directive is used to check whether the identifier is not currently defined.**

**Example:**

```
#ifdef PI
    printf( "Defined \n" );
#endif
#ifndef PI
    printf( "First define PI\n" );
#endif
```

**Output:**

First define PI

### **Miscellaneous directive**

The pragma directive is used to access compiler-specific preprocessor extensions. Each pragma directive has different implementation rule and use . There are many type of pragma directive and varies from one compiler to another compiler .If compiler does not recognize particular pragma then it ignores the pragma statement without showing any error or warning message.

**Example:**

```
#pragma sample
int main()
{
    printf("Pragma verification ");
    return 0;
}
```

**Output**

Pragma verification

Since #pragma sample is unknown for Turbo c compiler, it ignores sample directive without showing error or warning message and execute the whole program

assuming `#pragma` sample statement is not present. The following are the list of possible `#pragma` directives supported by C.

1. `#pragma startup`
2. `#pragma exit`
3. `pragma warn`
4. `#pragma option`
5. `#pragma inline`
6. `#pragma argsused`
7. `#pragma hdrfile`
8. `#pragma hdrstop`
9. `#pragma savereg`

## `#error`

The `#error` directive causes the preprocessor to emit an error message. `#error` directive is used to prevent compilation if a known condition that would cause the program not to function properly.

### Syntax:

```
#error "message"
```

### Example:

```
int main()
{
    #ifndef PI
        #error "Include PI"
    #endif
    return 0;
}
```

### Output

compiler error --> Error directive : Include PI

### `#line`

It tells the compiler that next line of source code is at the line number which has been specified by constant in `#line` directive

### Syntax:

```
#line <line number> [File Name]
```

Where

File Name is optional

### Example:

```
int main()
{
    #line 700
    printf(Line Number %d",__LINE__);
    printf(Line Number %d",__LINE__);
    printf(Line Number %d",__LINE__);
    return 0;
}
```

### Output

```
700
701
702
```

## UNIT III LINEAR DATA STRUCTURES

Arrays and its representations – Stacks and Queues – Linked lists – Linked list-based implementation of Stacks and Queues – Evaluation of Expressions – Linked list based polynomial addition.

### INTRODUCTION

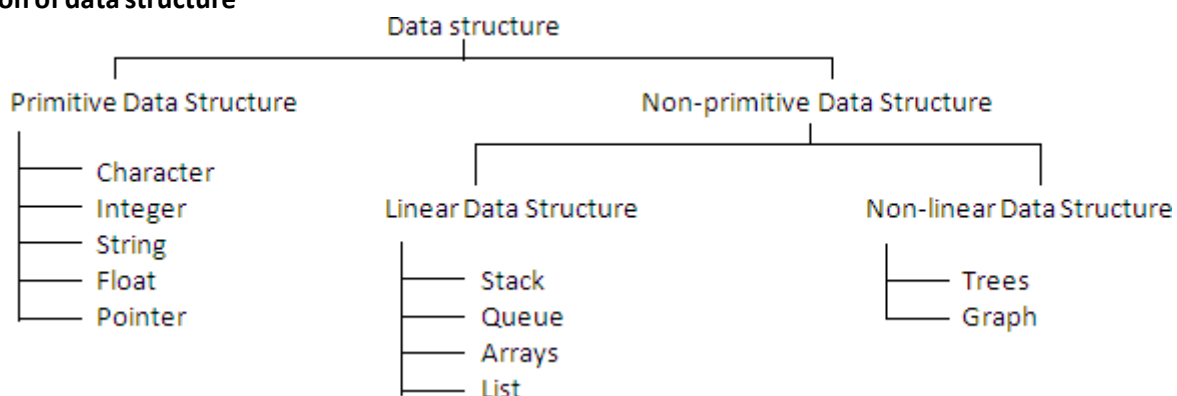
#### Definition

- Data structure is a particular way of organizing, storing and retrieving data, so that it can be used efficiently. It is the structural representation of logical relationships between elements of data.

#### Where data structures are used?

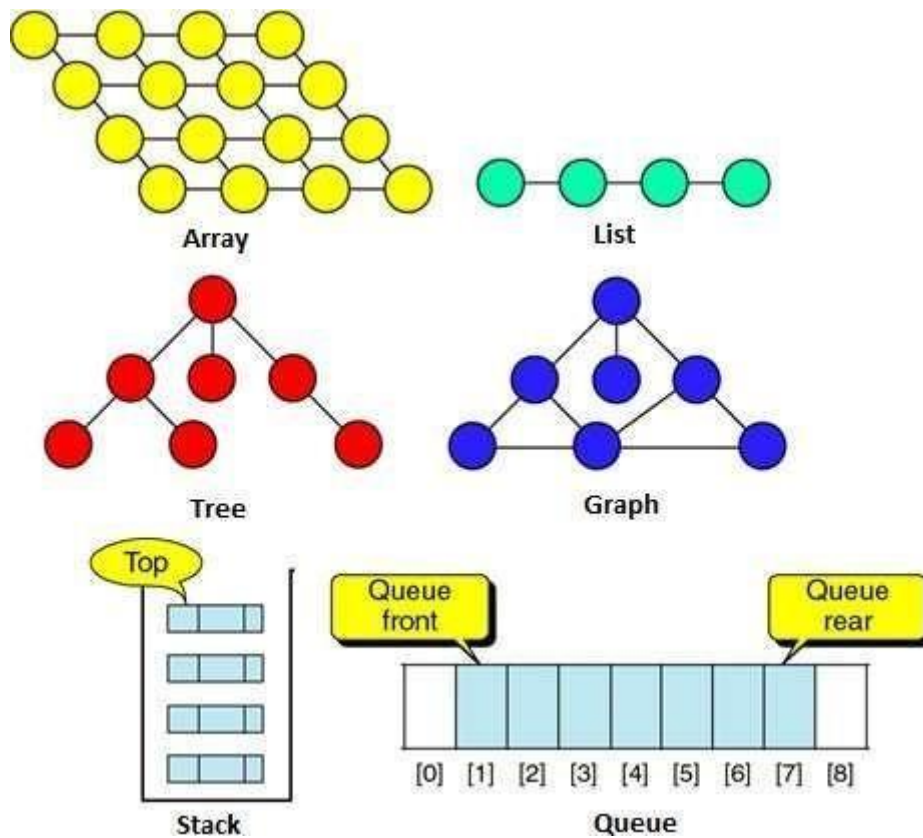
- Data structures are used in almost every program or software system. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.
- Applications** in which data structures are applied extensively
  - Compiler design (Hash tables),
  - Operating system,
  - Database management system (B+Trees),
  - Statistical analysis package,
  - Numerical analysis (Graphs),
  - Graphics,
  - Artificial intelligence,
  - Simulation

#### Classification of data structure



- Primitive Data Structure** - Primitive data structures are predefined types of data, which are supported by the programming language. These are the basic data structures and are directly operated upon by the machine instructions, which is in a primitive level.
- Non-Primitive Data Structure** - Non-primitive data structures are not defined by the programming language, but are instead created by the programmer. It is a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.
- Linear data structure**- only two elements are adjacent to each other. (Each node/element has a single successor)
  - Restricted list (Addition and deletion of data are restricted to the ends of the list)
    - ✓ Stack (addition and deletion at **top** end)
    - ✓ Queue (addition at **rear** end and deletion from **front** end)
  - General list (Data can be inserted or deleted anywhere in the list: at the beginning, in the middle or at the end)
- Non-linear data structure**- One element can be connected to more than two adjacent elements.(Each node/element can have more than one successor)
  - Tree (Each node could have multiple successors but just one predecessor)
  - Graph (Each node may have multiple successors as well as multiple predecessors)

**Note** - Array and Linked list are the two basic structures for implementing all other ADTs.



### MODULARITY

- **Module-** A module is a self-contained component of a larger software system. Each module is a logical unit and does a specific job. Its size kept small by calling other modules.
- **Modularity** is the degree to which a system's components may be separated and recombined. Modularity refers to breaking down software into different parts called modules.
- **Advantages** of modularity
  - It is easier to debug small routines than large routines.
  - Modules are easy to modify and to maintain.
  - Modules can be tested independently.
  - Modularity provides reusability.
  - It is easier for several people to work on a modular program simultaneously.

### ABSTRACT DATA TYPE

#### What is Abstract Data Type (ADT)?

- ADT is a mathematical specification of the data, a list of operations that can be carried out on that data. It **includes** the specification of what it does, but **excludes** the specification of how it does. Operations on **set ADT**: Union, Intersection, Size and Complement.
- The **primary objective** is to separate the implementation of the abstract data types from their function. The program must know what the operations do, but it is actually better off not knowing how it is done. Three most common used abstract data types are Lists, Stacks, and Queues.
- ADT is an **extension of modular design**. The **basic idea** is that the implementation of these operations is **written once in the program**, and any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function. If for some reason implementation details need to change, it should be easy to do so by merely changing the routines that perform the ADT operations. This change, in a perfect world, would be completely transparent to the rest of the program.
- Examples of ADT: Stack, Queue, List, Trees, Heap, Graph, etc.
- **Benefits of using ADTs or Why ADTs**

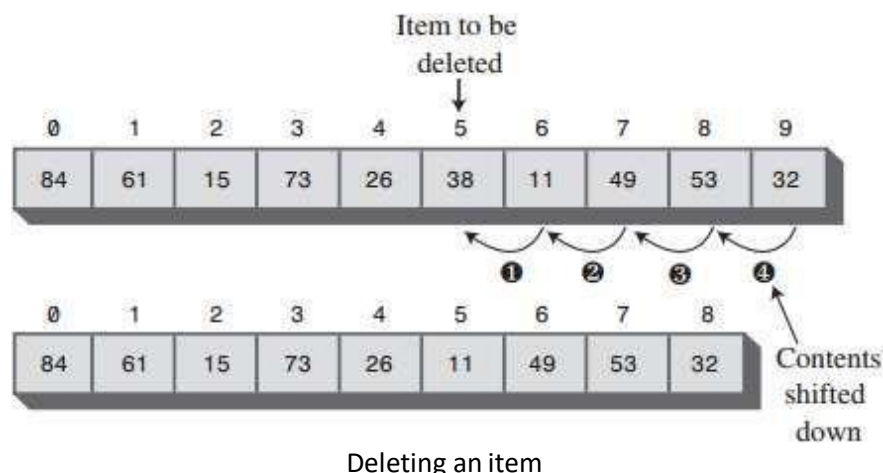
- Code is easier to understand. Provides modularity and reusability.
- Implementations of ADTs can be changed without requiring changes to the program that uses the ADTs.

### LIST ADT

- List is a linear collection of ordered elements. The general form of the list of size  $N$  is:  $A_0, A_1, \dots, A_{N-1}$ 
  - Where  $A_1$  - First element  
 $A_N$  - Last element  
 $N$  - Size of the list
  - If the element at position 'i' is  $A_i$  then its successor is  $A_{i+1}$  and its predecessor is  $A_{i-1}$ .
- Various operations performed on a List ADT
  - Insert ( $X, 5$ ) - Insert the element  $X$  after the position 5.
  - Delete ( $X$ ) - The element  $X$  is deleted.
  - Find ( $X$ ) - Returns the position of  $X$
  - Next ( $i$ ) - Returns the position of its successor element  $i+1$ .
  - Previous ( $i$ ) - Returns the position of its Predecessor element  $i-1$ .
  - PrintList - Displays the List contents.
  - MakeEmpty - Makes the List empty.
- Implementation of List ADT
  - Array implementation
  - Linked List implementation
  - Cursor implementation

### ARRAY IMPLEMENTATION OF LIST ADT

- An array is a collection of homogeneous data elements described by a single name. Each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis. In array implementation, elements of list are stored in contiguous cells of an array. FindKth operation takes constant time. **PrintList, Find** operations take linear time.
- Advantages - **Searching** an array for an **individual** element can be **very efficient** - Fast, random access of elements.
- Limitations - Array implementation has some limitations such as
  1. Maximum size must be known in advance, even if it is dynamically allocated.
  2. The size of array can't be changed after its declaration (static data structure). i.e., the size is fixed.
  3. Data are stored in continuous memory blocks.
  4. The running time for Insertion and deletion of elements is so slow. Inserting and deletion requires shifting other data in the array. For example, inserting at position 0 requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations is  $O(n)$ . On average, half the list needs to be moved for either operation, so linear time is still required.
  5. Memory is wasted, as the memory remains allocated to the array throughout the program execution even few nodes are stored.



**Type Declarations****#define Max 10**

int A[Max],N;

**Routine to insert an Element in the specified position**

void insert(int x, int p, int A[], int N)

```
{
int i;
If(p==N)
printf("Array Overflow");
else
{
for(i=N-1;i>=p-1;i--)
A[i+1]=A[i];
A[p-1]=x;
N=N+1;
}
}
```

**Routine to delete an Element in the specified**

int deletion(int p, int A[],int N)

```
{
int Temp;
If(p==N)
Temp=A[p-1];
else
{
Temp=A[p-1];
For(i=p-1;i<=N-1;i++)
A[i]=A[i+1];
}
N=N-1;
return Temp;
}
```

**Find Routine**

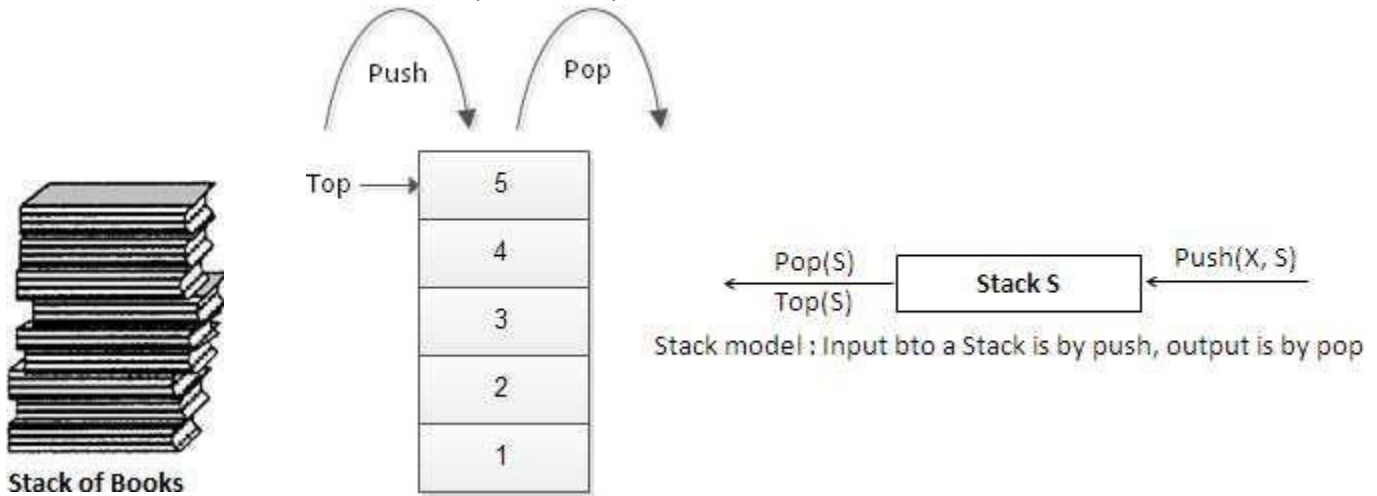
void Find (int X)

```
{
int i,f=0;
for(i=0;i<N;i++)
if(a[i]==x)
{
f=1;
break;
}
if (f==1)
printf("Element found");
else
printf("Element not found");
}
```

## STACK

### ➤ Definition

Stack is a linear list in which elements are added and removed from **only one end**, called the **top**. It is a "last in, first out" (**LIFO**) data structure. At the logical level, a stack is an **ordered** group of **homogeneous** items or elements. Because items are added and removed only from the top of the stack, the **last element** to be added is the first to be removed. Stacks are also referred to as "piles" and "push-down lists".

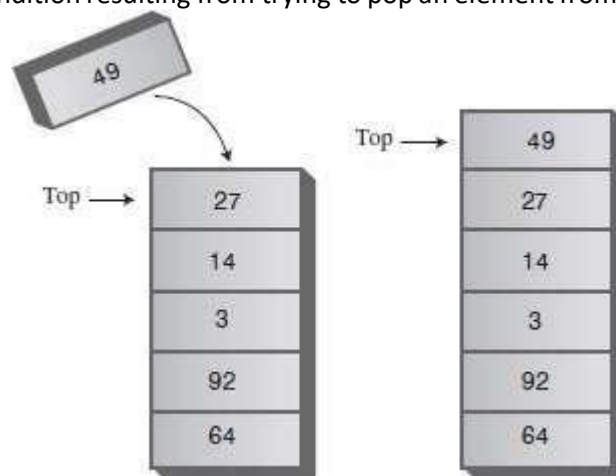


### ➤ Operations on stacks

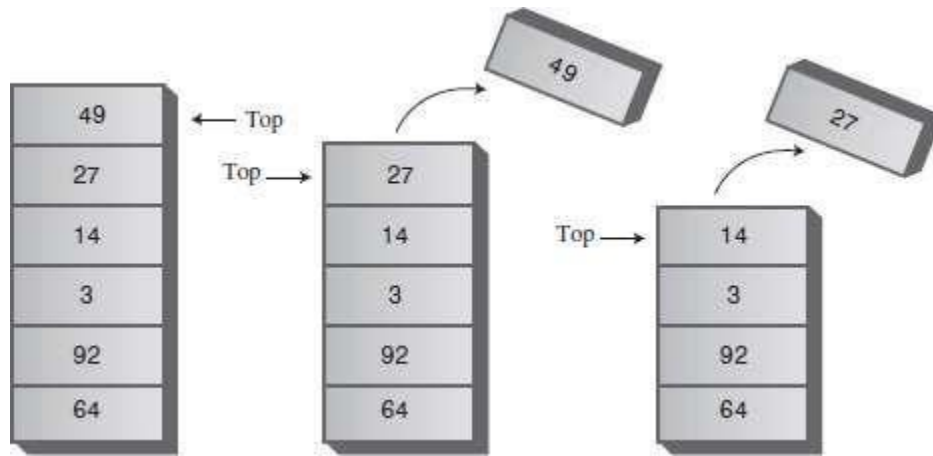
- **Push** - Inserts new item to the top of the stack. After the push, the new item becomes the top.
- **Pop** - Deletes top item from the stack. The next older item in the stack becomes the top.
- **Top** - Returns a copy of the top item on the stack, but does not delete it.
- **MakeEmpty** - Sets stack to an empty state.
- Boolean **IsEmpty** - Determines whether the stack is empty. IsEmpty should compare top with -1.
- Boolean **IsFull** - Determines whether the stack is full. IsFull should compare top with **MAX\_ITEMS - 1**.

### ➤ Conditions

- **Stack overflow** - The condition resulting from trying to push an element onto a full stack.
- **Stack underflow** - The condition resulting from trying to pop an element from an empty stack.



New item pushed on Stack



Two items popped from Stack

### APPLICATIONS OF STACKS

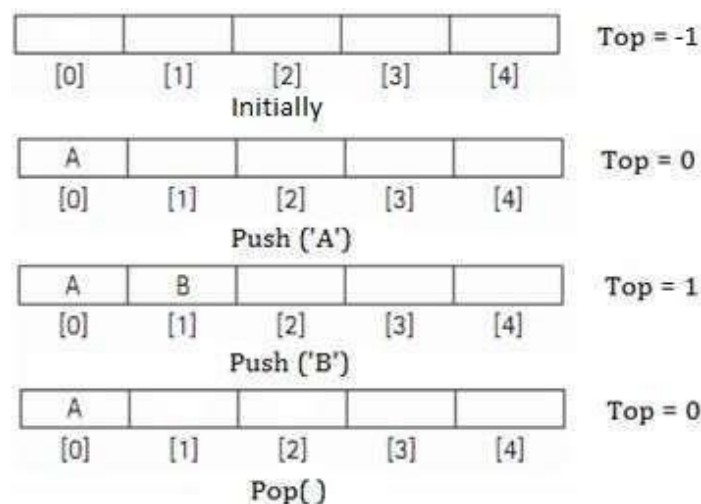
- **Recursion** - Example, Factorial, Tower of Hanoi.
- **Balancing Symbols**, i.e., finding the unmatched/missing parenthesis. For example,  $((A+B)/C$  and  $(A+B)/C)$ . Compilers often use stacks **to perform syntax analysis of language statements**.
- **Conversion** of infix expression to postfix expression and decimal number to binary number.
- **Evaluation** of postfix expression.
- **Backtracking**- For example, 8-Queens problem.
- **Function calls** - When a call is made to a new function, all the variables local to the calling routine need to be saved by the system, since otherwise the new function will overwrite the calling routine's variables. Similarly the current location in the routine must be saved so that the new function knows where to go after it is done. For example, the main program calls operation A, which in turn calls operation B, which in turn calls operation C. When C finishes, control returns to B; when B finishes, control returns to A; and so on. The call-and-return sequence is essentially a LIFO sequence, so a stack is the perfect structure for tracking it.

#### ➤ Implementations of stack

1. Array implementation of stack
2. Linked list implementation of stack

#### ➤ Array implementation of stack

Stack can be represented using one dimensional array and it is probably the **more popular** solution. Here the stack is of fixed size. That is maximum limit for storing elements is specified. Once the maximum limit is reached, it is not possible to store the elements into it. So array implementation is not flexible and not an efficient method when resource optimization is concerned.



**Push and Pop operation****Array implementation of Stack**

```
#include<stdio.h>
#include<conio.h>
#define MAX 5

void push();
void pop();
void display();
int stack[MAX], top=-1, item;

void push()
{
    if(top == MAX-1)
        printf("Stack is full");
    else
    {
        printf("Enter item: ");
        scanf("%d",&item);
        top++;
        stack[top] = item;
        printf("Item pushed = %d", item);
    }
}

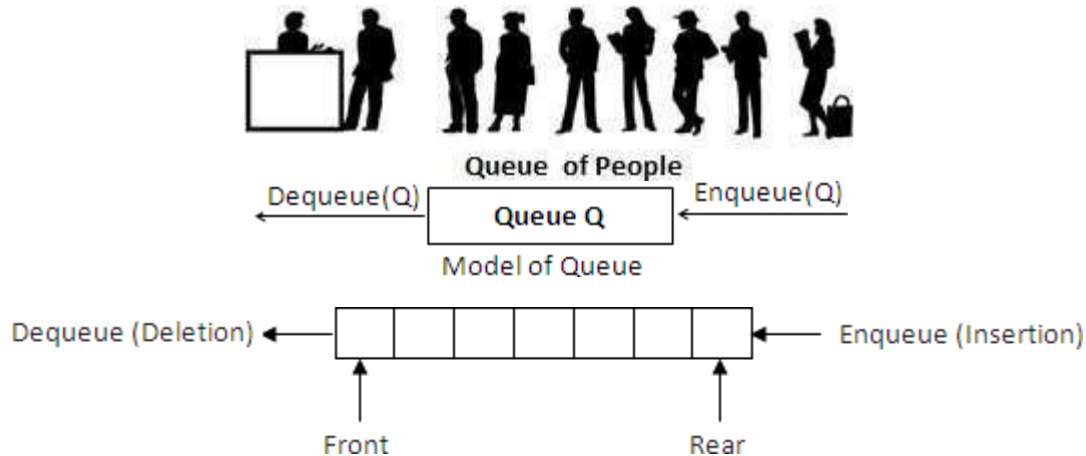
void pop()
{
    if(top == -1)
        printf("Stack is empty");
    else
    {
        item = stack[top];
        top--;
        printf("Item popped = %d", item);
    }
}

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

## QUEUE (LINEAR QUEUE)

### ➤ Definition

A queue is an **ordered** group of **homogeneous** items or elements, in which new elements are added at one end (the “rear”) and elements are removed from the other end (the “front”). It is a "First in, first out" (**FIFO**) linear data structure. Example, a line of students waiting to pay for their textbooks at a university bookstore.



### ➤ Types of Queues

There are three major variations in a simple queue. They are

- Linear queue
- Circular queue
- **Double ended queue (Deque)**
  - Input restricted deque
  - Output restricted deque
- Priority queue

### ➤ Operations on Queue

- **Enqueue** - Inserts an item at the rear end of the queue.
- **Dequeue** - Deletes an item at the front end of the queue and returns.

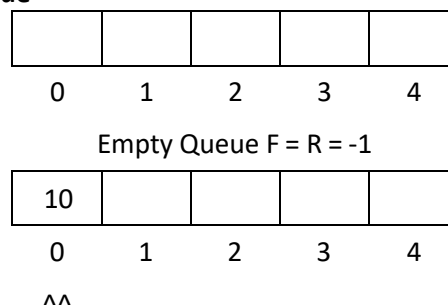
### ➤ Conditions

- **Queue overflow** - The condition resulting from trying to enqueue an element onto a full Queue.
- **Queue underflow** - The condition resulting from trying to dequeue an element from an empty Queue.

### ➤ Implementation of Queue

1. Array implementation
2. Linked list implementation
  - Array and linked list implementations give fast **O(1)** running times for every operation

### ➤ Array implementation of Linear Queue



FR

After Enqueue (10)

10	3			
0	1	2	3	4
^	^			
F	R			

After Enqueue (3)

10	3	41		
0	1	2	3	4
^		^		
F		R		

After Enqueue (41)

	3	41		
0	1	2	3	4
	^	^		
	F	R		

After Dequeue ()

	3	41	76	
0	1	2	3	4
	^		^	
	F		R	

After Enqueue (76)

	3	41	76	66
0	1	2	3	4
	^			^
	F			R

After Enqueue (66)

		41	76	66
0	1	2	3	4
		^		^
		F		R

After Dequeue ()

- There is one potential **problem** with array implementation. From the above queue, now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be inserted. Because in a queue, elements are

always inserted at the rear end and hence rear points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it is empty.

- The simple solution is that whenever front or rear gets to the end of the array, it is wrapped around to the beginning. This is known as a **circular array** implementation.

#### Array implementation of Linear Queue

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

#define MAX 3

void enqueue();
void dequeue();
void display();

int queue[MAX], rear=-1, front=-1, item;

void enqueue()
{
    if(rear == MAX-1)
        printf("Queue is full");
    else
    {
        printf("Enter item : ");
        scanf("%d", &item);
        if (rear == -1 && front == -1)
            rear = front = 0;
        else
            rear = rear + 1;
        queue[rear] = item;
        printf("Item enqueued : %d", item);
    }
}

void dequeue()
{
    if(front == -1)
        printf("Queue is empty");
    else
    {
        item = queue[front];
        if (front == rear)
            front = rear = -1;
        else
            front = front + 1;
        printf("Item dequeued : %d", item);
    }
}

void display()
{
    int i;
    if(front == -1)
```

```

printf("Queue is empty");
else
    for(i=front; i<=rear; i++)
        printf("%d ", queue[i]);
}

```

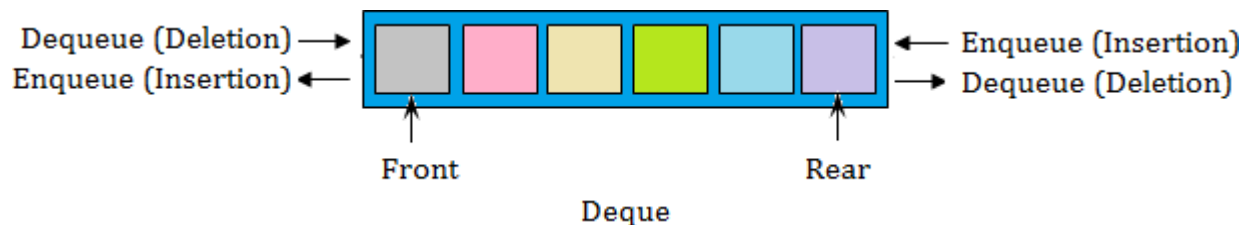
### Circular Queue

- In circular queues the elements  $Q[0], Q[1], Q[2], \dots, Q[n-1]$  is represented in a circular fashion with  $Q[1]$  following  $Q[n]$ . A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.
- Initially **Front = Rear = -1**. That is, front and rear are at the same position.
- At any time the **position** of the element to be **inserted** will be calculated by the relation: **Rear = (Rear + 1) % SIZE**
- After deleting an element from circular queue the position of the front end is calculated by the relation: **Front = (Front + 1) % SIZE**.
- After locating the position of the new element to be inserted, rear, compare it with front. If **Rear = Front**, the queue is full and **cannot be inserted anymore**.
- No of elements in a queue = **(Rear – Front + 1) % N**

### Deque - Double Ended Queue

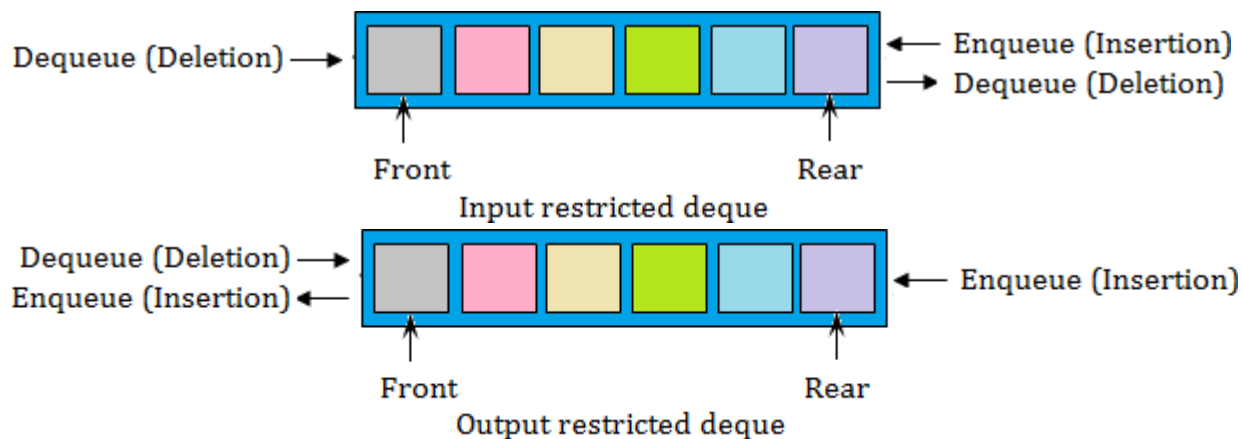
#### ➤ Definition

A deque is a homogeneous list in which inserted and deleted operations are performed at either ends of the queue. That is, we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called double ended queue. The most common ways of **representing** deque are: **doubly linked list, circular list**.



#### ➤ Types of dequeues

1. Input restricted deque
2. Output restricted deque



- ✓ An **input restricted deque** is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists.
- ✓ An **output-restricted deque** is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.

### Priority Queue

#### ➤ Definition

- Priority Queue is a queue where each element is assigned a priority. The priority may **implicit** (decided by its value) or **explicit** (assigned). In priority queue, the elements are deleted and processed by following rules.
  - An element of higher priority is processed before any element of lower priority.
  - Two elements with the same priority are processed according to the order in which they were inserted to the queue.
- Example for priority queue:
  - In a telephone answering system, calls are answered in the order in which they are received;
  - Hospital emergency rooms see patients in priority queue order; the patient with the most severe injuries sees the doctor first.



Queue of people with priority

- A node in the priority queue will contain Data, Priority and Next field. Data field will store the actual information; Priority field will store its corresponding priority of the data and Next will store the address of the next node.
- When an element is inserted into the priority queue, it will check the priority of the element with the element(s) present in the linked list to find the suitable position to insert. The node will be inserted in such a way that the data in the priority field(s) is in ascending order. We do not use rear pointer when it is implemented using linked list, because the new nodes are not always inserted at the rear end.

#### ➤ Types of priority queues

1. **Ascending priority queue** - It is a queue in which items can be inserted arbitrarily (in any order) and from which only the **smallest** item can be deleted first.
2. **Descending priority queue** - It is a queue in which items can be inserted arbitrarily (in any order) and from which only the **largest** item can be deleted first.

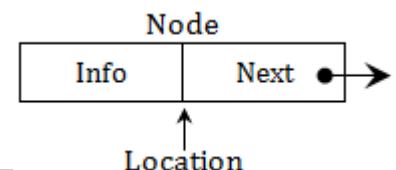
### Applications of queue

- When jobs are submitted to a printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a line printer are placed on a queue.
- Virtually every real-life line is (supposed to be) a queue. For instance, lines at ticket counters are queues, because service is first-come first-served.
- Another example concerns computer networks. There are many network setups of personal computers in which the disk is attached to one machine, known as the file server. Users on other machines are given access to files on a first-come first-served basis, so the data structure is a queue.
- Calls to large companies are generally placed on a queue when all operators are busy.
- There are several algorithms that use queues to solve problems easily. For example, BFS, Binary tree traversal etc.
- Round robin techniques for processor scheduling is implemented using queue.

### LINKED LIST

#### Definition

Linked list is **adynamic** data structure which is an ordered collection of homogeneous data elements called nodes, in which each element contains two parts: **data** or **Info** and one or more **links**. The data holds the application data to be processed. The link contains (the pointer) the address of the next element in the list.



**Why Linked List?**

- Even though **searching** an array for an **individual** element can be **very efficient**, array has some limitations. So arrays are generally not used to implement Lists.

**Advantages of Linked List**

1. Linked list are dynamic data structures - The size is not fixed. They can grow or shrink during the execution of a program.
2. Efficient memory utilization - memory is not pre-allocated. Memory is allocated, whenever it is required and it is de-allocated whenever it is not needed. Data are stored in non-continuous memory blocks.
3. Insertion and deletion of elements are easier and efficient. Provides flexibility. No need to shift elements of a linked list to make room for a new element or to delete an element.

**Disadvantages of Linked List**

1. More memory - Needs space for pointer (link field).
2. Accessing arbitrary element is time consuming. Only sequential search is supported not binary search.

**Operations on Linked List**

The primitive operations performed on the linked list are as follows

1. **Creation**- This operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.
2. **Insertion**- This operation is used to insert a new node at any specified location in the linked list. A new node may be inserted,
  - ✓ At the beginning of the linked list,
  - ✓ At the end of the linked list,
  - ✓ At any specified position in between in a linked list.
3. **Deletion**- This operation is used to delete an item (or node) from the linked list. A node may be deleted from the,
  - ✓ Beginning of a linked list,
  - ✓ End of a linked list,
  - ✓ Specified location of the linked list.
4. **Traversing** - It is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit the nodes only from left to right (forward traversing). But in doubly linked list forward and backward traversing is possible.
5. **Searching**- It is the process finding a specified node in a linked list.
6. **Concatenation**- It is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1) the node in A. After concatenation A will contain (name) nodes.

**Types of linked list**

1. Singlylinked list or Linear list or One-waylist
  2. Doubly linked list or Two-way list
  3. Circular linked list
  4. Doubly circular linked list
-

## SINGLY LINKED LIST

### ➤ Definition

In singly linked list, each element (except the first one) has a unique predecessor, and each element (except the last one) has a unique successor. Each node contains two parts: **data** or **Info** and **link**. The data holds the application data to be processed. The link contains the address of the next node in the list. That is, each node has a single pointer to the next node. The last node contains a NULL pointer indicating the end of the list.



### ➤ SentinelNode

- It is also called as **Header node** or **Dummy node**.

### ▪ Advantages

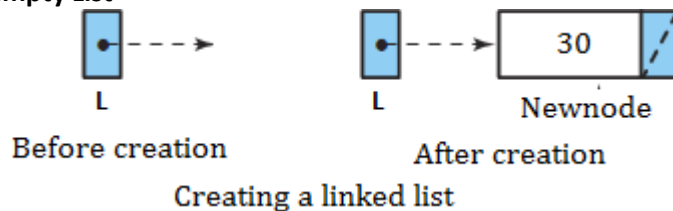
- Sentinel node is used to solve the following problems
  - ✓ First, there is **no really obvious way to insert at the front** of the list from the definitions given.
  - ✓ Second, **deleting from the front of the list** is a special case, because it changes the start of the list; careless coding will lose the list.
  - ✓ A third problem concerns deletion in general. Although the pointer moves above are simple, the deletion algorithm requires us to **keep track of the cell before the one that we want to delete**.

### ▪ Disadvantages

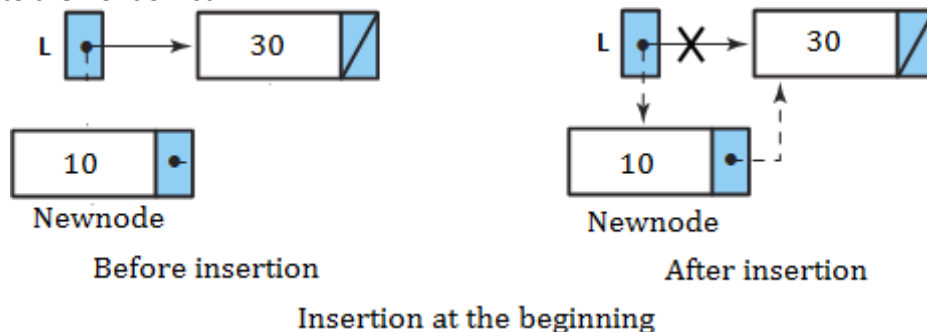
- It consumes extra space.

### ➤ Insertion

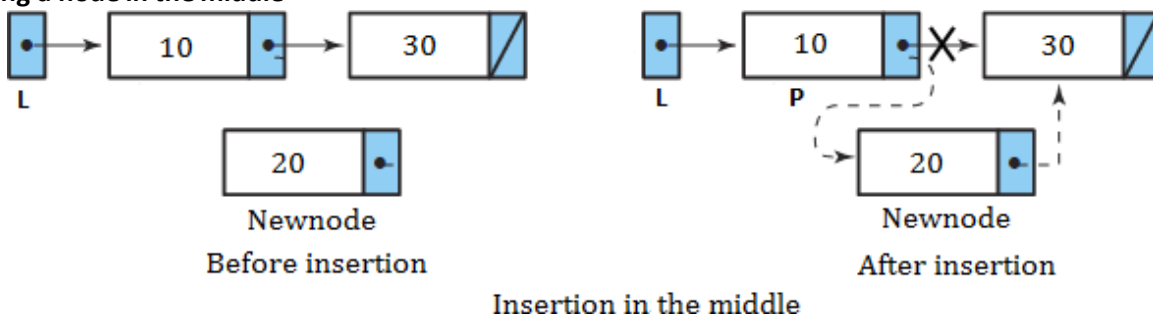
#### a. Creating a newnode from empty List



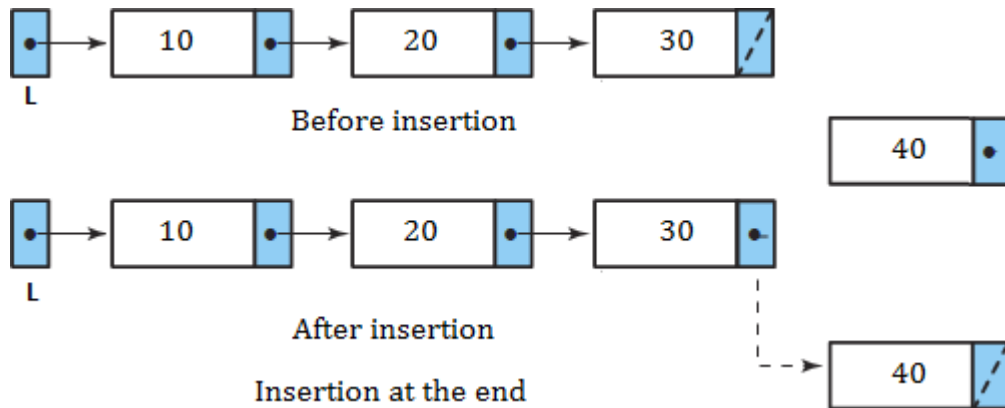
#### b. Inserting a node to the front of list



#### c. Inserting a node in the middle

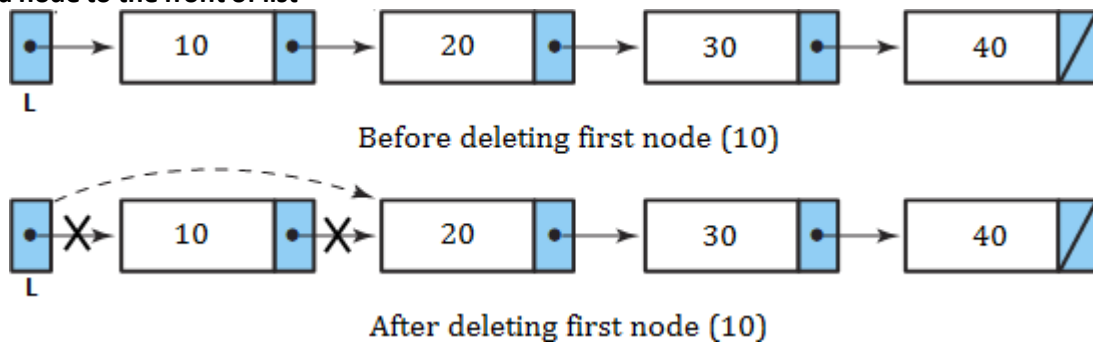


## d. Inserting a node to the end of list

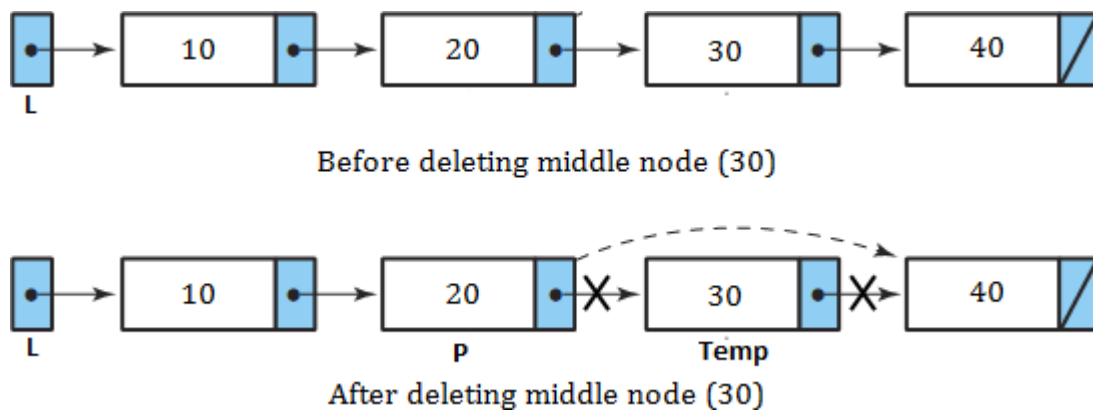


## ➤ Deletion

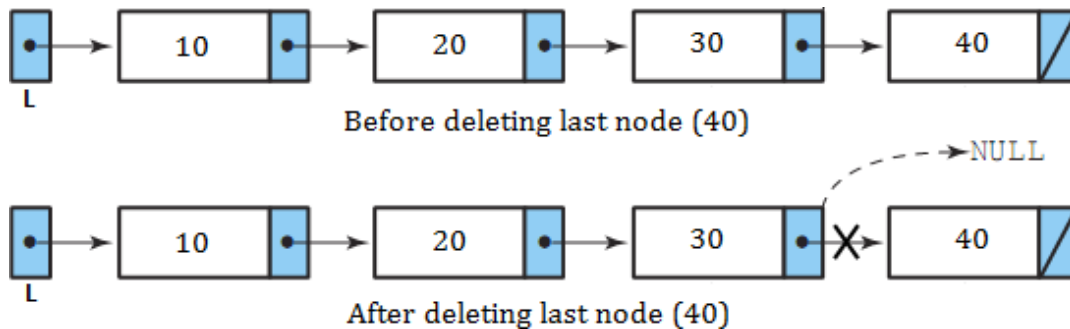
## a. Inserting a node to the front of list



## b. Deleting the middle node



## c. Deleting the last node



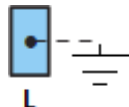
### Singly linked list implementation

#### Type Declarations

```
struct node
{
    int data;
    struct node *next;
}*head=NULL;
typedef struct Node *position;
```

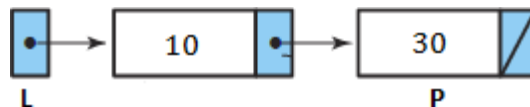
#### Routine to check whether the List is empty

```
/* Returns 1 if List is empty */
int IsEmpty (position head)
{
    if (head->next == NULL)
        return(1);
}
```



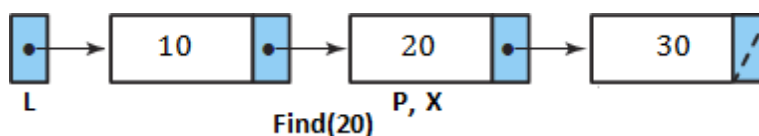
#### Routine to check whether the current position is last

```
/* Returns 1 if P is the last position in L */
int IsLast (position p)
{
    if (p->Next == NULL)
        return(1);
}
```



#### Find Routine

```
/* Returns the position of X in L; NULL if not found */
Position Find (int X)
{
    position p;
    P = head->next;
    while( (p!= NULL) && (p->data != X) )
        p = p->next;
    return P;
}
```



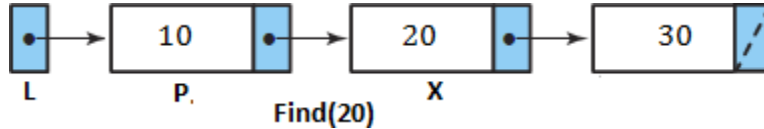
#### FindPrevious Routine

```
/* Returns the previous position of X in L */
```

```

position FindPrevious (int X)
{
    position P;
    P = head;
    while( (P->Next!= NULL) && (P->Next->data != X) )
        P = P->Next;
    return P;
}

```



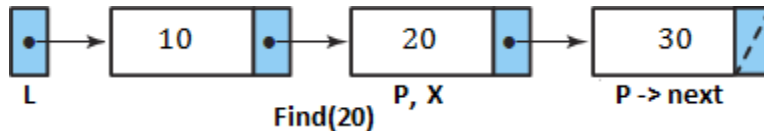
### FindNext Routine

/\* Returns the position of X in L; NULL if not found \*/  
 Position Find (int X)

```

{
    position p;
    P = head->next;
    while( (p!= NULL) && (p->data != X) )
        p = p->next;
    return P->next;
}

```



### void traversal ()

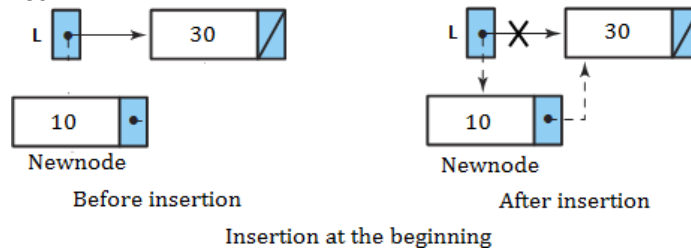
```

{
    position p;
    P = head->next;
    while( p!= NULL)
    {
        printf(p->data);
        p = p->next;
    }
}

```

### Insertion

#### Inserting a node to the front of list



#### Insert at Beginning

```

void Insert_beg (int X)
{

```

```

    position NewNode;
    NewNode = malloc (sizeof(struct Node));

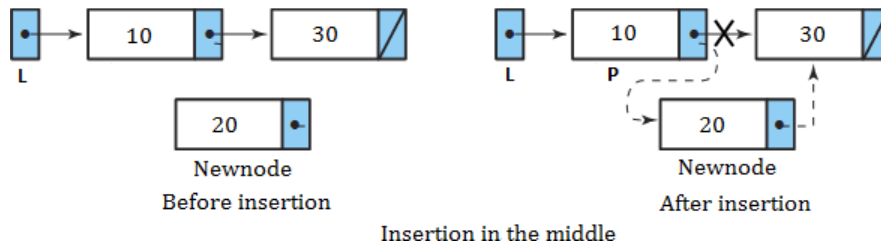
```

```

if(NewNode != NULL)
{
    NewNode->data = X;
    NewNode->next = L->Next;
    head->next = NewNode;
}
}

```

### b. Inserting a node in the middle



### Insertion at Middle

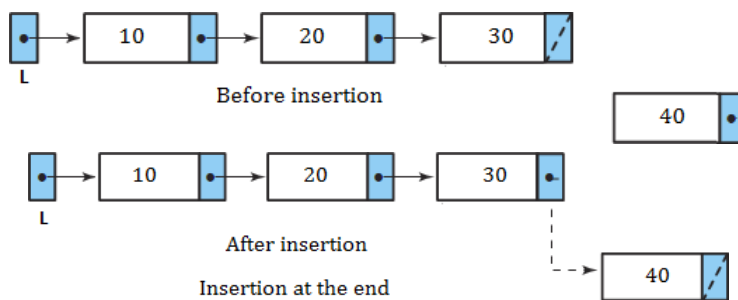
/\* Insert element X after position P \*/

```

void Insert_mid (int X, position P)
{
    position NewNode;
    NewNode = malloc (sizeof(struct Node));
    if(NewNode != NULL)
    {
        NewNode->data = X;
        NewNode->next = P->next;
        P->next = NewNode;
    }
}

```

### c. Inserting a node to the end of list



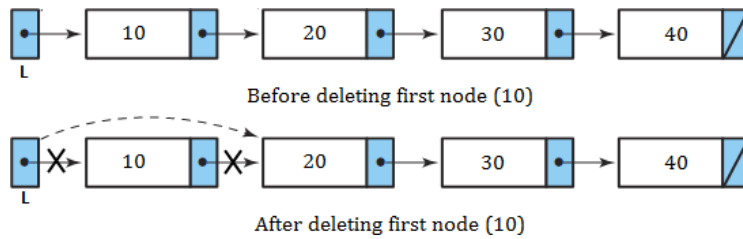
### Insert at Last

```

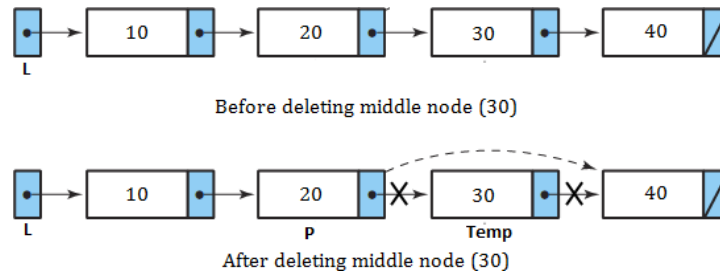
void Insert_last (int X)
{
    position NewNode,P;
    NewNode = malloc (sizeof(struct Node));
    if(NewNode != NULL)
    {
        while(P->next!=NULL)
        P = P->next;
        NewNode->data = X;
        NewNode->next = NULL;
        P->next = NewNode;
    }
}

```

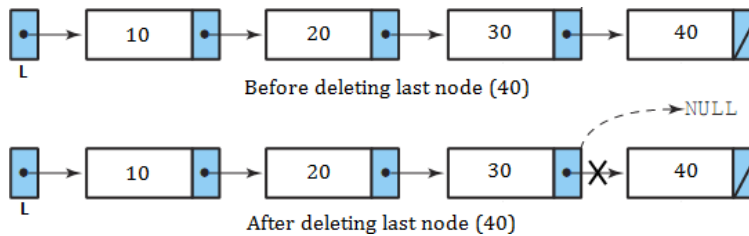
}

➤ **Deletion****Deleting a node to the front of list****Delete at Beginning**

```
void Delete_beg ()
{
    position TempCell;
    if(head->next!=NULL)
    {
        TempCell = head->next;
        head->next = TempCell ->next;
        free(TempCell);
    }
}
```

**Deleting the middle node****Delete at Middle Routine**

```
void Delete (int X)
{
    position P, TempCell;
    P = FindPrevious(X);
    TempCell = P->next;
    P->Next = TempCell ->Next;
    free(TempCell);
}
```

**Deleting the last node****Delete at Last**

```
void Delete_last ()
{
    position TempCell,P;
    while(P->next->next!=NULL)
```

```

    P = P->next;
    TempCell = P->next;
    P->next = NULL;
    free(TempCell);
}

```

### CIRCULAR LINKED LIST

#### ➤ Why circular linked list? Or advantages over singly linked list

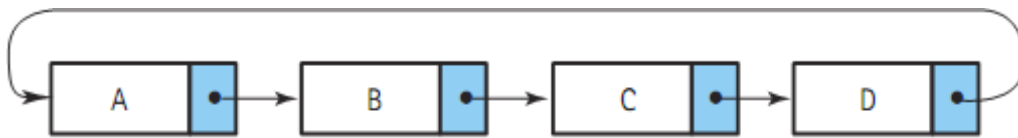
- With a singly linked list structure, given a pointer to a node anywhere in the list, we can access all the nodes that follow but none of the nodes that precede it. We must always have a pointer to the beginning of the list to be able to access all the nodes in the list. In a circular linked list, every node is accessible from a given node.
- In deletion of singly linked list, to find the predecessor requires that a search be carried out by chaining through the nodes from the first node of the list. But this requirement does not exist for a circular list, since the search for the predecessor of node X can be initiated from X itself.
- Concatenation and splitting becomes more efficient.

#### ➤ Disadvantages

- The circular linked list requires extra care to detect the end of the list. It may be possible to get into an infinite loop. So it needs a header node to indicate the start or end of the list.

#### ➤ Definition

- A circular linked list is one, which has no beginning and no end. Circular linked list is a list in which every node has a successor; the "last" element is succeeded by the "first" element. We can start at any node in the list and traverse the entire list.

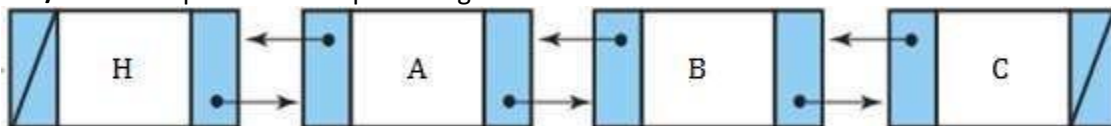
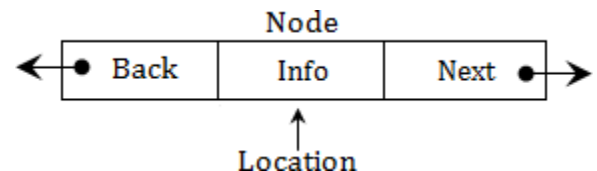


### DOUBLY LINKED LIST

#### ➤ Definition

- Doubly linked list is a linked list in which each node is linked to both its successor and its predecessor. In a doubly linked list, the nodes are linked in both directions. Each **node** of a doubly linked list contains three parts:

- **Info:** the data stored in the node
- **Next/FLink:** the pointer to the following node.
- **Back/BLink:** the pointer to the preceding node



#### ➤ Why doubly linked list?

- In singly linked list, it is difficult to perform traversing the list in reverse.
- To delete a node, we need find its predecessor of that node.

#### ➤ Advantages

- Traversing in reverse is possible.
- Deletion operation is easier, since it has pointers to its predecessor and successor.

- Finding the predecessor and successor of a node is easier.

#### ➤ Disadvantages

- A doubly linked list needs **more operations** while inserting or deleting and it needs **more space** (to store the extra pointer). There are more pointers to keep track of in a doubly linked list. For example, to insert a new node after a given node, in a singly linked list, we need to change two pointers. The same operation on a doubly linked list requires four pointer changes.

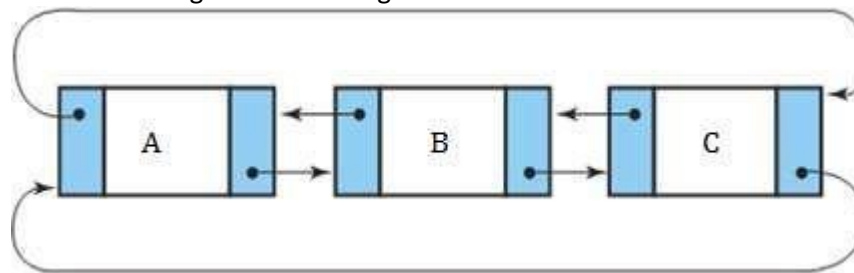
### DOUBLY CIRCULAR LINKED LIST

#### ➤ Why doubly circular linked list?

The aim of considering doubly circular linked list is to simplify the insertion and deletion operations performed on doubly linked list.

#### ➤ Definition

A circular linked list is one, which has no beginning and no end. A doubly circular linked list is a doubly linked list with circular structure in which the last node points to the first node and the first node points to the last node and there are two links between the nodes of the linked list. In doubly circular linked list, the left link of the leftmost node contains the address of the rightmost node and the right link of the rightmost node contains the address of the leftmost node.



A Doubly Linked List

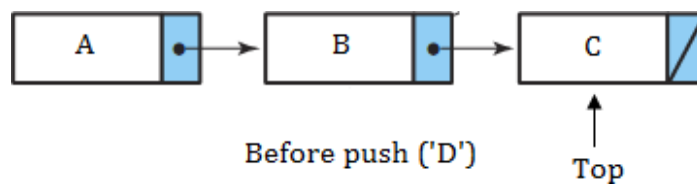
### APPLICATIONS OF LINKED LIST

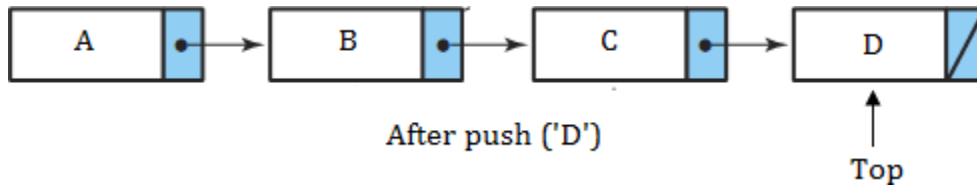
All kinds of dynamic allocation related problems can be solved using linked lists. Some of the applications are given below:

1. Polynomial ADT
2. Radix sort or Card sort
3. Multi-list
4. Stacks and Queues

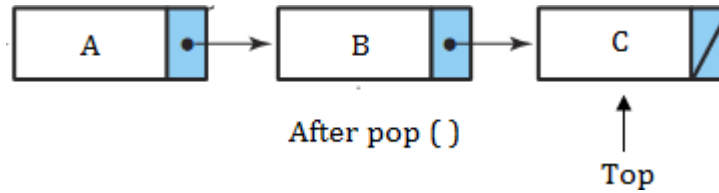
#### Linked list implementation of stack

The limitations of array implementation can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers. In linked list implementation, the stack does not need to be of fixed size. Insertions and deletions are done more efficiently. Memory space also not wasted, because memory space is allocated only when it is necessary (when an element is pushed) and is de-allocated when the element is deleted.





**Push operation**



**Pop operation**

### Linked list implementation of Stack

```

void push(int x);
void pop();
void display();

struct node
{
    int data;
    struct node *next;
} *top = NULL;

typedef struct node * position;

void push(int x)
{
    position p;
    p=(struct node *)malloc(sizeof(struct node));
    if (p == NULL)
        printf("Memory allocation error \n");
    else
    {
        if (top == NULL)
        {
            top=(struct node *)malloc(sizeof(struct node));
            p->data=x;
            p->next = NULL;
            top->next = p;
        }
        else
        {
            p->data=x;
            p->next = top->next;
            top->next=p;
        }
    }
}

```

```

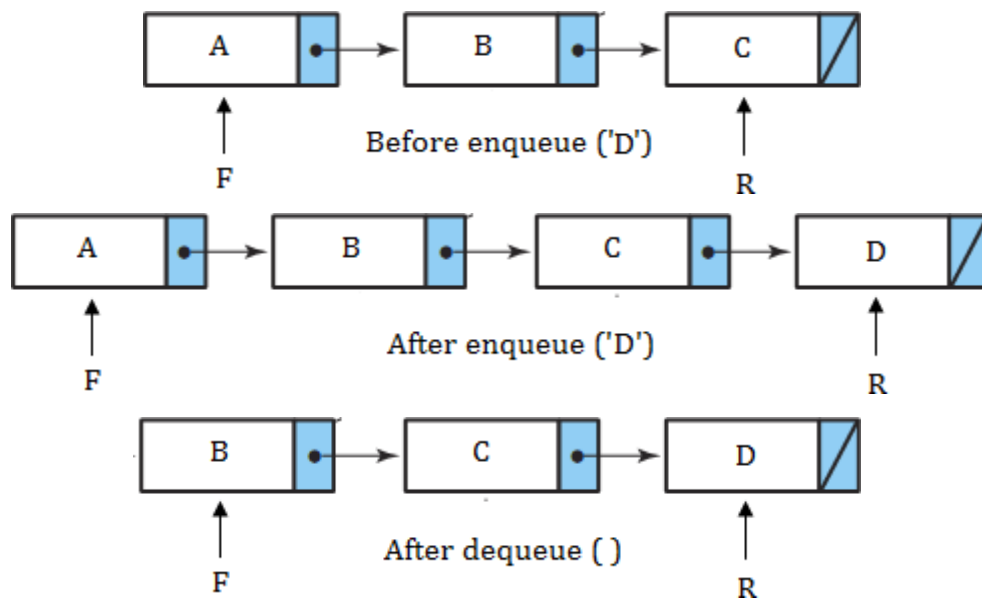
void pop()
{
    position p;
    p = top->next;

    if (top == NULL)
        printf("Stack is empty");
    else
    {
        top->next= top->next->next;
        printf("\n Popped value : %d\n", p->data);
        free(p);
    }
}

void display()
{
    struct node *p;
    if (top == NULL)
        printf("Stack is empty");
    else
    {
        p = top;
        while(p != NULL)
        {
            printf("\n%d", p->data);
            p = p->next;
        }
    }
}

```

### Linked list implementation of Linear Queue



## Linked list implementation of Linear Queue

```

struct node
{
    int data;
    struct node *next;
} *front = NULL, *rear=NULL;

typedef struct node * position;

void enqueue(int x);
void dequeue();
void display();
int item;

struct node
{
    int data;
    struct node *next;
} *top = NULL;

typedef struct node * position;

void dequeue()
{
    position p;
    p = front->next;

    if (front == NULL)
        printf("Queue is empty\n");
    else
    {
        printf("\n Dequeued value : %d\n", p->data);
        front->next=front->next->next;
        free(p);
    }
}

void display()
{
    position p;
    p = front->next;

    if (front == NULL)
        printf("Queue is empty\n");
    else
    {
        printf("Queue elements are : \n");
        while (p != NULL)
        {

```

```

        printf("%d ",p->data);
        p = p->next;
    }
}
}

```

### APPLICATIONS OF STACKS

- **Recursion** - Example, Factorial, Tower of Hanoi.
- **Balancing Symbols**, i.e., finding the unmatched/missing parenthesis. For example, ((A+B)/C and (A+B)/C). Compilers often use stacks **to perform syntax analysis of language statements**.
- **Conversion** of infix expression to postfix expression and decimal number to binary number.
- **Evaluation** of postfix expression.
- **Backtracking**- For example, 8-Queens problem.
- **Function calls** - When a call is made to a new function, all the variables local to the calling routine need to be saved by the system, since otherwise the new function will overwrite the calling routine's variables. Similarly the current location in the routine must be saved so that the new function knows where to go after it is done. For example, the main program calls operation A, which in turn calls operation B, which in turn calls operation C. When C finishes, control returns to B; when B finishes, control returns to A; and so on. The call-and-return sequence is essentially a LIFO sequence, so a stack is the perfect structure for tracking it.

#### Conversion of infix expression into postfix expression

1. Scan the infix expression from left to right. Repeat Steps 3 to 6 for each element of expression until the stack is empty.
2. If an operand is encountered, add it to the postfix expression.
3. If an opening parenthesis is encountered, push it onto the stack and do not remove it until closing parenthesis is encountered.
4. If an operator 'op' is encountered, then
  - a. Repeatedly pop from stack and add each operator (on the top of stack), which has the same precedence as, or higher precedence than 'op'.
  - b. Add 'op' to stack.
5. If a closing parenthesis is encountered, then
  - a. Repeatedly pop from stack and add to postfix expression (on the top of stack) until an opening parenthesis is encountered.
  - b. Remove the opening parenthesis from the stack. [Do not add the opening parenthesis to postfix expression.]

Operator precedence	
(	Highest
^	-
*, /	-
+, -	Least

Infix	Stack	Postfix
A+B*C-D/E		
+B*C-D/E		A
B*C-D/E	+	A
*C-D/E	+	AB
C-D/E	+	AB
-D/E	+	ABC
D/E	-	ABC++
/E	-	ABC++D
E	/	ABC++D
	/	ABC++DE
		ABC++DE/-

➤ **Evaluation of postfix expression**

1. Scan the postfix expression from left to right and repeat steps 2 & 3 for each element of postfix expression.
2. If an operand is encountered, push it onto the stack.
3. If an operator 'op' is encountered,
  - a. Pop two elements from the stack, where A is the top element and B is the next top element.
  - b. Evaluate B 'op' A.
  - c. Push the result onto stack.
4. The evaluated value is equal to the value at the top of the stack.

Postfix	Stack
246+*	
46+*	2
6+*	4 2
+*	6 4 2
*	10 2
	20

### Evaluation of postfix expression

#### ➤ Balancing parenthesis

- One common programming problem is unmatched parenthesis in an algebraic expression. When parentheses are unmatched, two types of errors can occur:
  - Opening parenthesis can be missing. For example,  $[A+B]/C$ .
  - Closing parenthesis can be missing. For example,  $\{(A+B)/C$ .
- The steps involved in checking the validity of an arithmetic expression
  1. Scan the arithmetic expression from left to right.
  2. If an opening parenthesis is encountered, push it onto the stack.
  3. If a closing parenthesis is encountered, the stack is examined.
    - a. If the stack is **empty**, the closing parenthesis does not have an opening parenthesis. So the expression is invalid.
    - b. If the stack is **not empty**, pop from the stack and check whether the popped item corresponds to the closing parenthesis. If a match occurs, continue. Otherwise, the expression is invalid.
  4. When the end of the expression is reached, the stack must be empty; otherwise one or more opening parenthesis does not have corresponding closing parenthesis. So the expression is invalid.

Exp	Stack
{(A+B)*C	
(A+B)*C	{
A+B)*C	{
+B)*C	{
)*C	{
*C	{
C	{
	{

Finally, the stack is non-empty.  
So the expression is invalid.

**Balancing parenthesis**

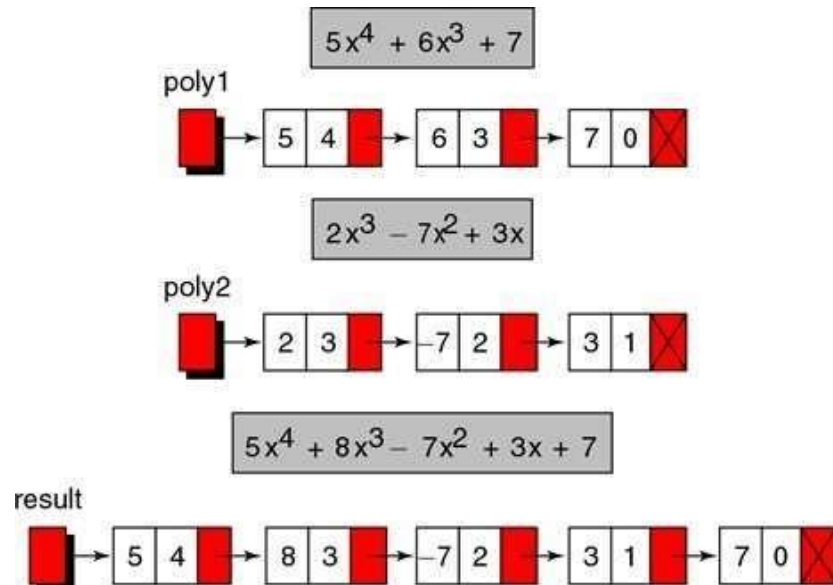
### ➤ Polynomial ADT

Polynomials are expressions containing terms with non-zero coefficients and exponents. Linked list is generally used to represent and manipulate single variable polynomials. Different operations, such as addition, subtraction, division and multiplication of polynomials can be performed using linked list. In this representation, each term/element is referred as a node. Each node contains three fields namely,

1. Coefficient - Holds value of the coefficient of a term.
2. Exponent - Holds exponent value of a term.
3. Link - Holds the address of the next term.

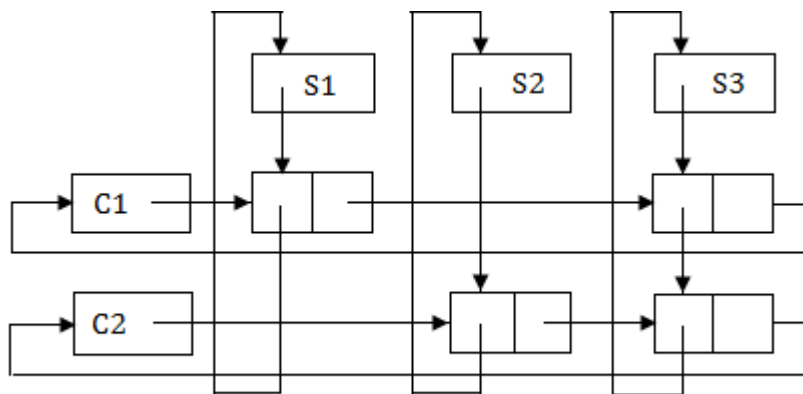
Coefficient	Exponent	Link
-------------	----------	------

For example,



➤ **Multi-list**

Multi-list is the **most complicated applications** of linked list. It is useful to maintain student registration in a university, employee involvement in different projects etc. The student registration contains two reports. The first report lists the registration for each class (C) and the second report lists, by student, the classes that each student (S) is registered for. In this implementation, we have combined two lists into one. All lists use a header and are circular. Circular list **saves space** but does so at the **expense of time**.



## Multi-list implementation for student registration problem

## Polynomial Addition

### Type Declaration:

```
struct node
```

```
{
    int coeff;
    int pow;
    struct node *next;
}*poly1=NULL,*poly2=NULL,*poly=NULL;
```

```
void polyadd(struct node *poly1, struct node *poly2, struct node *poly)
```

```

while((poly1->next != NULL)&& (poly2->next != NULL))
{
    if(poly1->pow > poly2->pow)

```

```

{
    poly->pow=poly1->pow;
    poly->coeff=poly1->coeff;
    poly1=poly1->next;
}
else if(poly1->pow<poly2->pow)
{
    poly->pow=poly2->pow;
    poly->coeff=poly2->coeff;
    poly2=poly2->next;
}
else
{
    poly->pow=poly1->pow;
    poly->coeff=poly1->coeff+poly2->coeff;
    poly1=poly1->next;
    poly2=poly2->next;
}
poly->next=(struct node *)malloc(sizeof(struct node));
poly=poly->next;
poly->next=NULL;
}
while(poly1->next !=NULL)
{

    poly->pow=poly1->pow;
    poly->coeff=poly1->coeff;
    poly1=poly1->next;
}
while(poly2->next!=NULL)
{

    poly->pow=poly2->pow;
    poly->coeff=poly2->coeff;
    poly2=poly2->next;
}
}

```

#### PART A

##### 1. Why do we need ADT for implementing data structures? Or Benefits of ADT.

- Code is easier to understand.
- Provides modularity and reusability.
- Implementations of ADTs can be changed without requiring changes to the program that uses the ADTs.

##### 2. What is an ADT? What are all concerned and not concerned in an ADT? Give example.

- ADT is a mathematical specification of the data, a list of operations that can be carried out on that data. It **includes** the specification of what it does, but **excludes** the specification of how it does. Operations on **set ADT**: Union, Intersection, Size and Complement.
- Examples of ADT: Stack, Queue, List, Trees, Heap, Graph, etc.

##### 3. Define a linear and non-linear data structure.

- **Linear data structure**- only two elements are adjacent to each other. (Each node/element has a single successor)
  - Restricted list (Addition and deletion of data are restricted to the ends of the list)
    - ✓ Stack (addition and deletion at **top** end)

✓ Queue (addition at **rear** end and deletion from **front** end)

○ General list (Data can be inserted or deleted anywhere in the list: at the beginning, in the middle or at the end)

- **Non-linear data structure-** One element can be connected to more than two adjacent elements.(Each node/element can have more than one successor)
  - Tree (Each node could have multiple successors but just one predecessor)
  - Graph (Each node may have multiple successors as well as multiple predecessors)

#### 4. Define List ADT with example.

- List is a linear collection of ordered elements. The general form of the list of size N is:  $A_0, A_1, \dots, A_{N-1}$ 
  - Where  $A_1$  - First element  
 $A_N$  - Last element  
 $N$  - Size of the list
  - If the element at position 'i' is  $A_i$  then its successor is  $A_{i+1}$  and its predecessor is  $A_{i-1}$ .

#### 5. Which operations are supported by the list ADT?

- Various operations performed on a List ADT
  - Insert (X,5) - Insert the element X after the position 5.
  - Delete (X) - The element X is deleted.
  - Find (X) - Returns the position of X
  - Next (i) - Returns the position of its successor element i+1.
  - Previous (i) - Returns the position of its Predecessor element i-1.
  - PrintList - Displays the List contents.
  - MakeEmpty - Makes the List empty.

#### 6. What is the advantage linked list over arrays?. Mention their relative advantages and disadvantages?

- Advantages of Array
  - Searching an array for an individual element can be very efficient,
- Advantages of Linked List
  - Linked list are dynamic data structures - The size is not fixed. They can grow or shrink during the execution of a program.
  - Efficient memory utilization - memory is not pre-allocated. Memory is allocated, whenever it is required and it is de-allocated whenever it is not needed. Data are stored in non-continuous memory blocks.
  - Insertion and deletion of elements are easier and efficient. Provides flexibility. No need to shift elements of a linked list to make room for a new element or to delete an element.
- Disadvantages of Linked List
  - More memory - Needs space for pointer (link field).
  - Accessing arbitrary element is time consuming. Only sequential search is supported not binary search.

#### 7. What are the advantages of doubly linked list over singly linked list?

- In singly linked list, it is difficult to perform traversing the list in reverse. To delete a node, we need find its predecessor of that node.

#### 8. List certain applications of lists.

- All kinds of dynamic allocation related problems can be solved using linked lists. Some of the applications are given below:
  - Polynomial ADT
  - Radix sort or Card sort
  - Multi-list
  - Stacks and Queues

#### 9. What are the two important features present in a pointer implementation of linked list?

- The two **important features** present in a **pointer implementation** of linked list are as follows:

1. The data are stored in a collection of structures. Each structure contains data and a pointer to the next structure.
2. A new structure contains data from the system's global memory by a call to malloc and released by a call to free.

#### 10. Write operations that can be done on stack?

- **Push** - Inserts new item to the top of the stack. After the push, the new item becomes the top.
- **Pop** - Deletes top item from the stack. The next older item in the stack becomes the top.
- **Top** - Returns a copy of the top item on the stack, but does not delete it.
- **MakeEmpty** - Sets stack to an empty state.
- Boolean **IsEmpty** - Determines whether the stack is empty. IsEmpty should compare top with -1.
- Boolean **IsFull** - Determines whether the stack is full. IsFull should compare top with **MAX\_ITEMS - 1**.

#### 11. What are the conditions that followed in the array implementation of stack?

- **Stack overflow** - The condition resulting from trying to push an element onto a full stack.
- **Stack underflow** - The condition resulting from trying to pop an element from an empty stack.

#### 12. Mention any four applications of stack.

- Recursion - Example, Factorial, Tower of Hanoi.
- Balancing Symbols,
- Conversion of infix expression to postfix expression and decimal number to binary number.
- Evaluation of postfix expression.
- Backtracking- For example, 8-Queens problem.

#### 13. What is circular queue?

- In circular queues the elements  $Q[0], Q[1], Q[2] \dots Q[n-1]$  is represented in a circular fashion with  $Q[1]$  following  $Q[n]$ . A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.

#### 14. Why is circular queue better than standard linear queue?

- There is one potential **problem** in linear queue with array implementation. Sometimes if we attempt to add more elements, even though queue cells are free, the elements cannot be inserted. Because in a queue, elements are always inserted at the rear end and if the rear points to last location of the queue array. That is queue is full (overflow condition) though it is empty.

#### 15. What is deque?

- A deque is a homogeneous list in which inserted and deleted operations are performed at either ends of the queue. That is, we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called double ended queue. The most common ways of **representing** deque are: **doubly linked list, circular list**.

#### 16. What is the difference between a stack and a Queue?

Stack	Queue
✓ Stack is a restricted ordered list where in all insertions and deletions are done at one end called top.	✓ Queue is a restricted ordered list where in all insertions are done at rear end and deletions are done at front end.
✓ Stack is called as LIFO structure.	✓ Queue is called as FIFO structure.
✓ To insert an element into the stack top is incremented by 1.	✓ To insert an element into the queue rear end is incremented by 1.
✓ To delete an element from the stack top is decremented by 1.	✓ To delete an element from the queue front end is incremented by 1.
✓ Stack full condition: $\text{Top} = \text{MAX} - 1$	✓ Queue full condition : $\text{Rear} = \text{MAX} - 1$
✓ Stack empty condition: $\text{Top} = -1$	✓ Queue empty condition : $\text{Front} > \text{Rear}$

**17. Mention any two applications of queue.**

- Calls to large companies are generally placed on a queue when all operators are busy.
- There are several algorithms that use queues to solve problems easily. For example, BFS, Binary tree traversal etc.
- Round robin techniques for processor scheduling is implemented using queue

**PART B**

1. Explain the array and linked list implementation of stack.
2. Explain the array and linked list implementation of queue.
3. Explain how stack is applied for evaluating an arithmetic expression.
4. Explain the applications of stack, queue and linked list.
5. Explain polynomial addition using Linked list.

## UNIT IV

### NON-LINEAR DATA STRUCTURES

#### Syllabus:

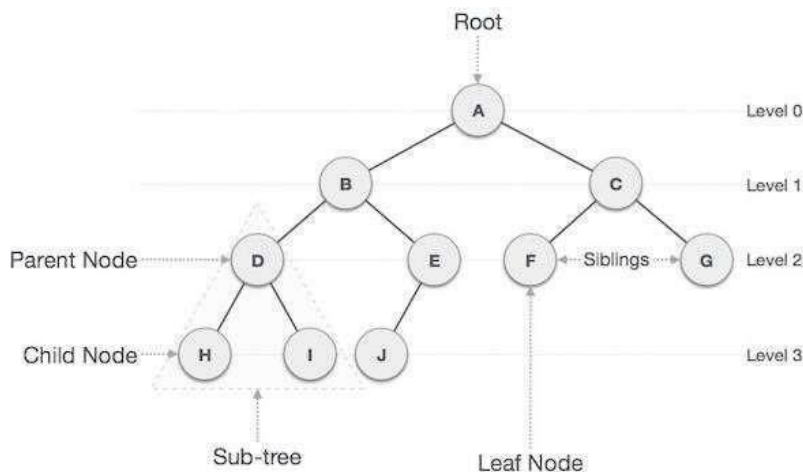
Trees – Binary Trees – Binary tree representation and traversals – Binary Search Trees – Applications of trees. Set representations – Union-Find operations. Graph and its representations – Graph Traversals.

#### TREES STRUCTURE

##### Definition:

Tree is a non-linear data structure. It organized the data in hierarchical manner. A tree is a finite set of one or more nodes such that there is a specially designated node called the root node and root node can have zero or more sub trees T<sub>1</sub>, T<sub>2</sub>, T<sub>3</sub>, ..... , T<sub>n</sub>. Each of whose roots are connected by a directed edge from root R.

Tree is collection of nodes in which the first node is called root and root has many number of sub tree T<sub>1</sub>, T<sub>2</sub>, T<sub>3</sub>... T<sub>n</sub>.



#### Terms:

**1. Root**

A node which does not have a parent is called as root node.

**2. Node**

Each data element in the tree is called as node.

**3. Leaf node**

A node which does not have any children is called leaf node.

**4. Siblings**

A child of same parent is called sibling.

**5. Path**

A path from node  $n_1$  to  $n_k$  is defined as sequence of nodes  $n_1, n_2, n_3, \dots, n_k$ . Such that  $n_i$  is a parent of  $n_{i+1}$ . Example: A → B → E → J

**6. Length for a path**

Number of edges in the path.

Example: Consider path from A to J is 3

**7. Degree**

Number of sub trees of the node is called degree.

**8. Level**

Root is at level 1 then  $i$ 's children are at level  $2+i$

Example: level

**9. Depth**

For any node  $n$ , the depth  $n$  is length of unique path from root to  $n$ .

### 10. Height

For any node  $n$ , the height of node  $n$  is the length of longest path from  $n$  to left.

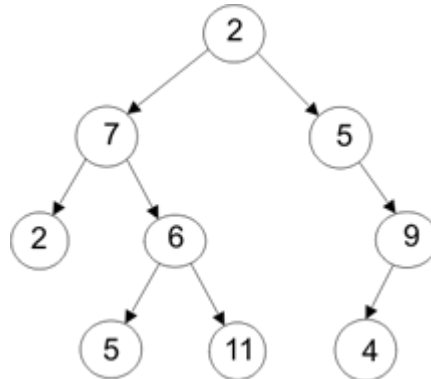
### 11. Forest

Collection of tree node is known as forest.

## BINARY TREE ADT

### Definition: -

Binary Tree is a special type of tree in which no node can have most two children. Typically, child nodes of a binary tree on the left is called left child and node on right is called right child. Maximum number of nodes



at level  $i$  of a binary tree is  $2^{i-1}$ .

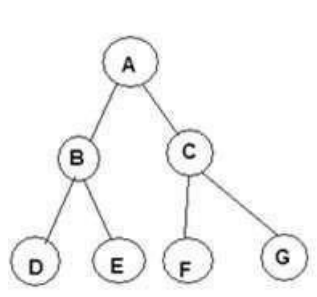
A simple binary tree of size 9 and height 3, with a root node whose value is 2. The above tree is neither a sorted nor a balanced binary tree

### Representation of tree.

1. Sequential representation or array representation.
2. Linked representation.

#### 1. Sequential representation or array representation.

The elements are represented using arrays. For any element in position  $i$ , the left child is in position  $2i$ , the right child is in position  $(2i + 1)$ , and the parent is in position  $(i/2)$ .



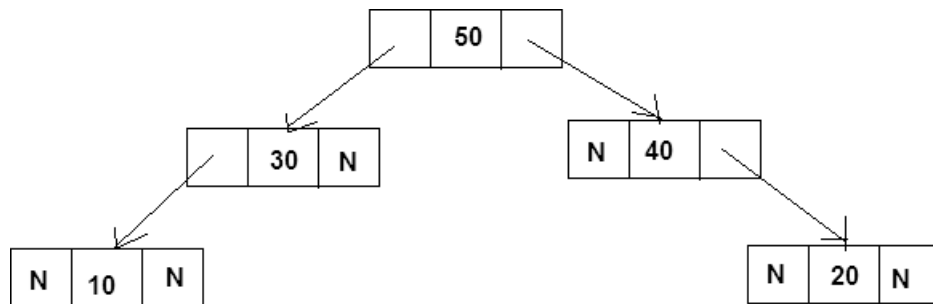
A	B	C	D	E	F	G
0	1	2	3	4	5	6

Array representation of Binary Tree

#### 2. Linked representation

The elements are represented using pointers. Each node in linked representation has three fields, namely,

- \* Pointer to the left subtree
- \* Data field
- \* Pointer to the right subtree



**Linked representation of Binary Tree**

Routine of creating tree using linked list.

```
struct tree
{
int data;
struct tree *leftchild;
struct tree *rightchild;
};
```

### **Recursive Traversals of Tree**

Traversing means visiting each node at once. Tree traversal is a method for visiting all the nodes in the tree exactly once.

There are three types of tree traversal techniques, namely

1. Inorder Traversal or symmetric order
2. Preorder Traversal or depth-first order
3. Postorder Traversal

### **Inorder Traversal**

The inorder traversal of a binary tree is performed as

- \* Traverse the left subtree in inorder
- \* Visit the root
- \* Traverse the right subtree in inorder.

### **Recursive Routine for Inorder Traversal**

```
void Inorder (Tree T)
{
if (T!=NULL)
{
Inorder (T->left);
printf("%d",T->data);
Inorder (T->right);
}
}
```

### **Preorder Traversal**

The preorder traversal of a binary tree is performed as follows,

- \* Visit the root
- \* Traverse the left subtree in preorder
- \* Traverse the right subtree in preorder.

### Recursive Routine for Preorder Traversal

```
void Preorder (Tree T)
{
if (T != NULL)
{
printf("%d",T->data);
Preorder (T->left);
Preorder (T->right);
}
```

### Postorder Traversal

The postorder traversal of a binary tree is performed by the following steps.

- \* Traverse the left subtree in postorder.
- \* Traverse the right subtree in postorder.
- \* Visit the root.

### Recursive Routine for Postorder Traversal

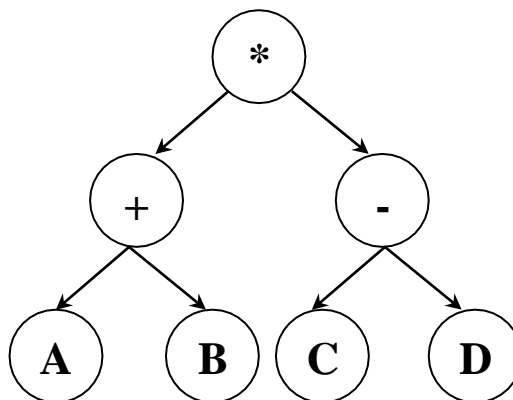
```
void Postorder (Tree T)
{
if (T != NULL)
{
Postorder (T->Left);
Postorder (T->Right);
printf("%d",T->data);
}
```

### EXPRESSION TREE

An expression tree is tree in which left nodes have the operands and interior node have the operators. Like binary tree, expression tree can also be traversed by inorder, preorder and postorder traversal.

Example:

$(A+B)*(C-D)$



### Constructing an Expression Tree

Let us consider postfix expression given as an input for constructing an expression tree by performing the following steps:

1. Read one symbol at a time from the postfix expression and then scan the expression from left to right manner.
2. Check whether the symbol is an operand or operator.
  - a. If the symbol is an operand, create a one - node tree and push a pointer on to the stack.
  - b. If the symbol is an operator pop two pointers from the stack namely  $T_1$  and  $T_2$  and form a new tree with root as the operator and  $T_2$  as a left child and  $T_1$  as a right child. A pointer to this new tree is then pushed onto the stack.
3. Repeated the above steps until reach the end of the expression.
4. The final pointer in the stack is complete expression tree.

Example:

$ABC*+DE*F+G*+$

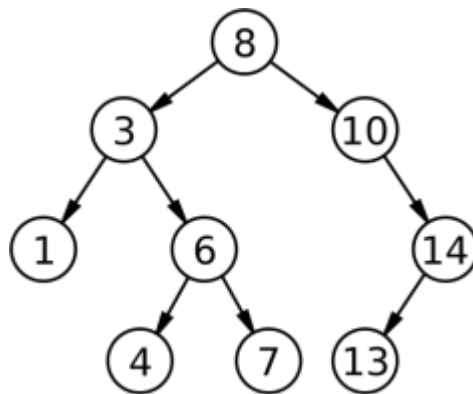
## BINARY SEARCH TREE OR SEARCH TREE ADT

### Definition: -

When we place constraints on how data elements can be stored in the tree, the items must be stored in such a way that the key values in left subtree of the root less than the key value of the root, and then the key values of all the node in the right subtree of the root are greater than the key values of the root. When this relationship holds in the entire node in the tree then the tree is called as a binary search tree.

The property that makes a binary tree into a binary search tree. That is every node  $X$  in the tree, the values of all the keys in its left subtree are smaller than the key value in  $X$ , and the values of all the keys in its right subtree are larger than the key value in  $X$ .

Example:

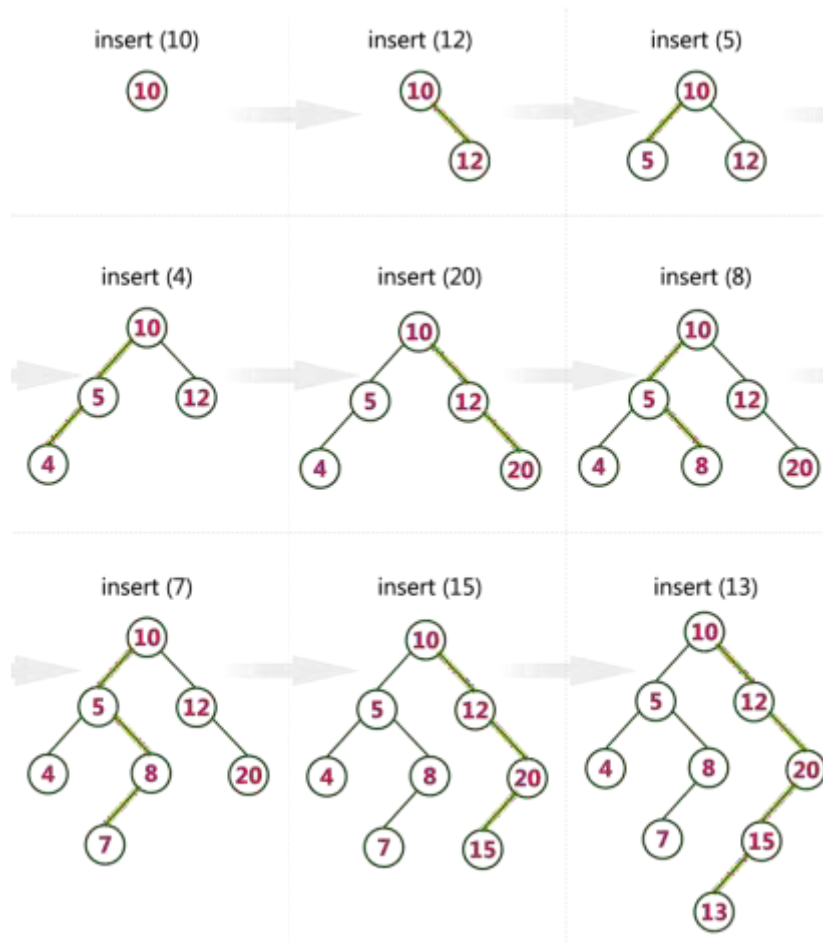


Operations of BST.

1. Insertion
2. Deletion
3. Find
4. Find min
5. Find max
6. Retrieve

**Notes:** when you're constructing the binary tree the given elements are read from first.

Example:



```
struct treenode
{
    int data;
    struct treenode *left;
    struct treenode *right;
};
```

**Routine for perform find.**

```
struct treenode * find(struct treenode *T,int x)
{
    if(T==NULL)
        return NULL;
    else if(x<T->data)
        return find(T->left,x);
    else if(x>T->data)
        return find(T->right,x);
    else
        return T;
}
```

**Routine for perform insertion.**

```
struct treenode* insert(struct treenode *t,int x)
{
    if(t==NULL)
```

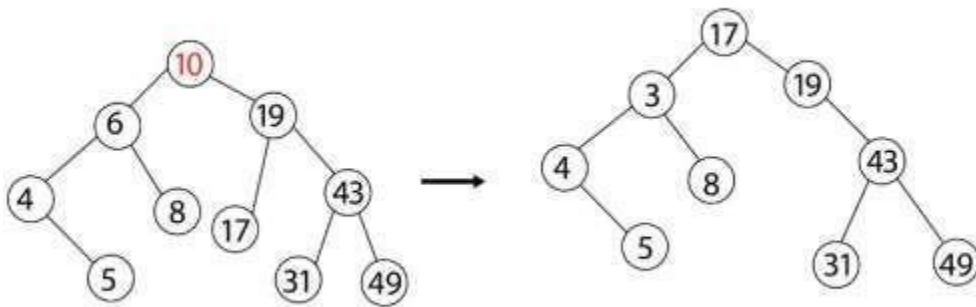
```

{
    t=(struct treenode *)malloc(sizeof(struct treenode));
    t->data=x;
    t->left=t->right=NULL;
    return t;
}
else if(x<t->data)
    t->left=insert(t->left,x);
else if(x>t->data)
    t->right=insert(t->right,x);
return t;
}

```

Deletion — I/P = 10

Node deleted to have 2 child.

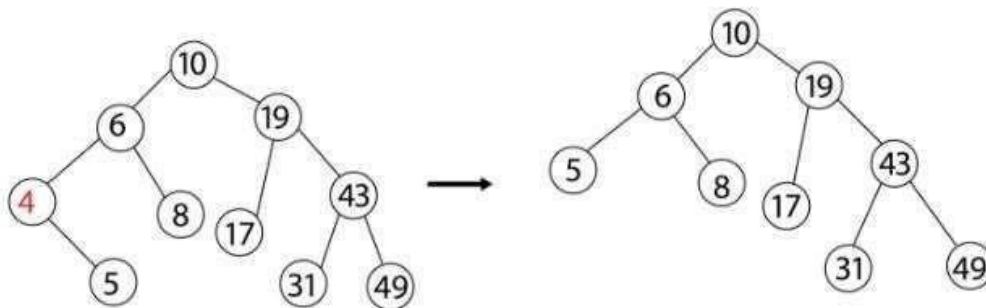


Before Deletion

After Deletion

Deletion — I/P = 4

Node deleted to have 1 child.



Before Deletion

After Deletion

### Routine for perform deletion

```

struct treenode * findmin(struct treenode *t)
{
    if(t==NULL)
        return NULL; /* There is no element in the tree */
    else if(t->left==NULL) /* Go to the left sub tree to find the min element */
        return t;
    else
        return findmin(t->left);
}

```

```

}

struct treenode* deletion(struct treenode *t,int x)
{
    struct treenode *temp;
    if(t==NULL)
        printf("Element not found\n");
    else if(x < t->data )
        t->left = deletion (t->left,x);
    else if(x > t->data )
        t->right = deletion (t->right,x);
    else if(T->right && T->left)
    {
        /* Here we will replace with minimum element in the right sub tree */
        temp = findmin(t->right);
        t->data = temp->data ;
        /* As we replaced it with some other node, we have to delete that node */
        t->right = deletion (t->right,t->data);
    }
    else
    {
        /* If there is only one or zero children then we can directly remove it from the tree and connect its
        parent to its child */
        temp = T;
        if(t->left==NULL)
            t = t->right;
        else if(T->right == NULL)
            t = t->left;
        free(temp); /* temp is longer required */
    }
    return T;
}

void inorder(struct treenode *t)
{
    if(t!=NULL)
    {
        inorder(t->left);
        printf("%d \t",t->data);
        inorder(t->right);
    }
}

```

### Applications of Tree

1. Manipulate hierarchical data.
2. Make information easy to search
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

## DIFFERENCE BETWEEN BINARY AND BINARY SEARCH TREES:

BINARY TREE	BINARY SEARCH TREE
It is a tree with only two children	It is also a tree with only two children.
It has no restrictions regarding its children	In this the left child is lesser than the parent and the right child is greater than the parent

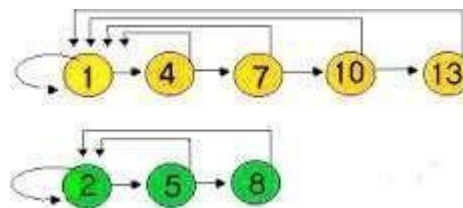
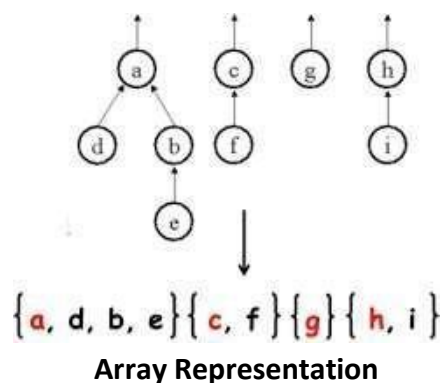
## THE DISJOINT SET ADT

A disjoint set data structure keeps note of a set of non-overlapping subsets. It is a data structure that helps us solve the dynamic equivalence problem.

### Various Representations Of Disjoint Set Adt

Array Representation

Linked List Representation



**Linked List Representation**

### Array Representation

This representation assigns one position for each element. Each position stores the element and an index to the representative. To make the Find-Set operation fast we store the name of each equivalence class in the array. Thus the find takes constant time,  $O(1)$ . Assume element a belongs to set i and element b belongs to set j. When we perform Union(a,b) all j's have to be changed to i's. Each union operation unfortunately takes  $\Theta(n)$  time. So for n-1 unions the time taken is  $\Theta(n^2)$ .

### Tree Representation

A tree data structure can be used to represent a disjoint set ADT. Each set is represented by a tree. The elements in the tree have the same root and hence the root is used to name the set.

### Operations

1. Find
2. Union

The trees do not have to be binary since we only need a parent pointer.

Make-set (DISJ\_SET S )

```
int i;  
for( i = N; i > 0; i-- )  
    p[i] = 0;
```

Initially, after the Make-set operation, each set contains one element. The Make-set operation takes  $O(1)$  time. set\_type

Find-set( element\_type x, DISJ\_SET S )

```
if( p[x] <= 0 )  
    return x;  
else
```

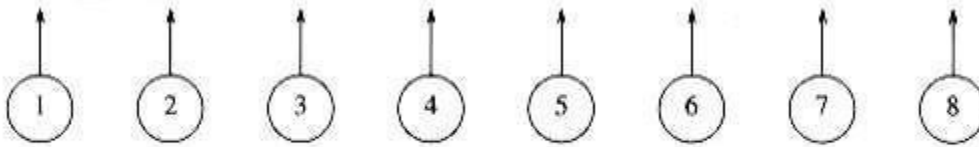
```
return( find( p[x], S ) );
```

The Find-Set operation takes a time proportional to the depth of the tree. This is inefficient for an unbalanced tree

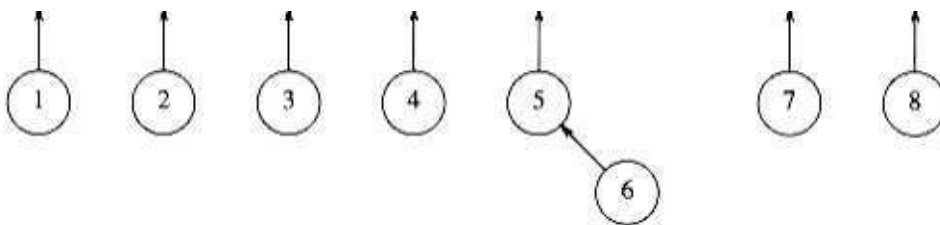
```
void Union( DISJ_SET S, set_type root1, set_type root2 )
```

```
    p[root2] = root1;
```

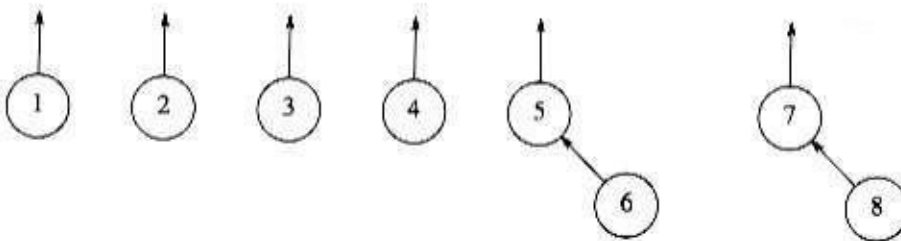
The union operation takes a constant time of  $O(1)$



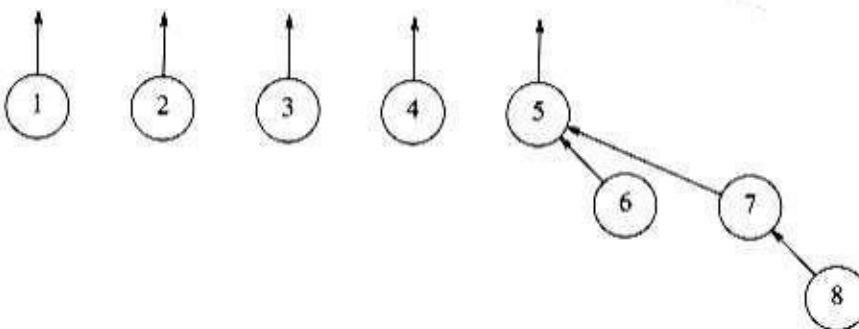
**Figure 1.4** Tree representation of disjoint set ADT after Make-set operation

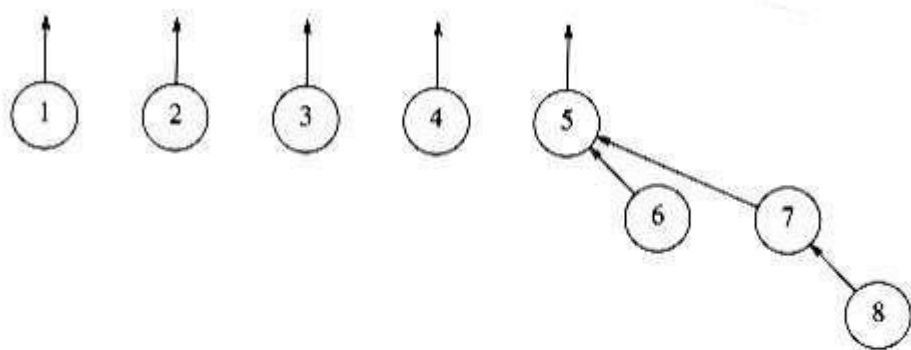


**Figure 1.5** Tree representation of disjoint set ADT after union (5,6)

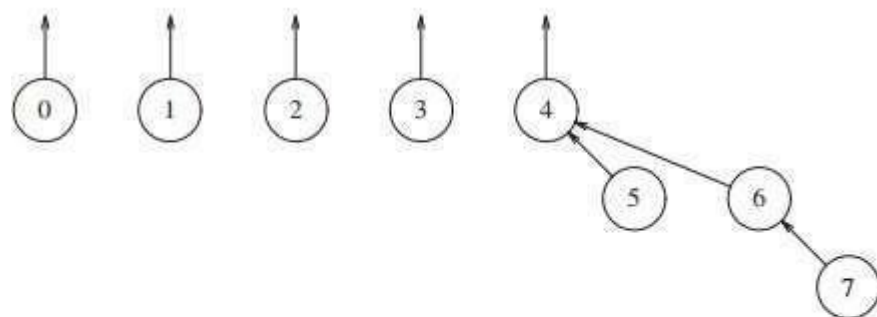


**Figure 1.6** Tree representation of disjoint set ADT after union (7,8)





**Figure 1.7** Tree representation of disjoint set ADT after union (5,7)



**Figure 8.4** After union(4,6)

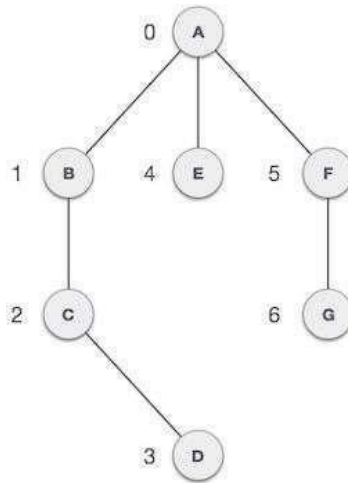
-1	-1	-1	-1	-1	4	4	6
0	1	2	3	4	5	6	7

**Figure 8.5** Implicit representation of previous tree

## Graphs

### Introduction to graphs

A graph is a data structure that is used to represent a relational data e.g. a set of terminal in network or roadmap of all cities in a country. Such complex relationship can be represented using graph data structure. A graph is a structure made of two components, a set of vertex  $V$  and the set of edges  $E$ . Therefore, a graph is  $G = (V, E)$  where  $G$  is graph. The graph may be directed or undirected. Vertices are referred to as nodes and the arc between the nodes are referred to as Edges.



### Terms associated with graphs

#### 1. Directed graph

A directed graph  $G$  is also called digraph which is the same as multigraph except that each edge  $e$  in  $G$  is assigned a direction or in other words each edge in  $G$  is identified with an order pair  $(U, V)$  of node in  $G$  rather than an unordered pair

#### 2. Undirected graph

An undirected graph  $g$  is a graph in which each edge  $e$  is not assigned a direction.

Undirected Graph

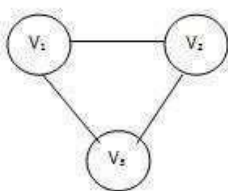


Figure 1: An Undirected Graph

Directed Graph

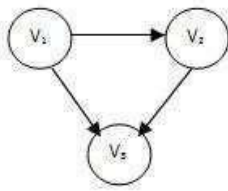
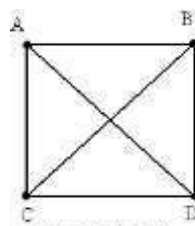
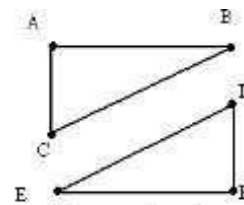


Figure 2: A Directed Graph



Connected graph



Disconnected graph

#### 3. Connected graph

A graph is called connected if there is a path from any vertex to any other vertex.

**Strongly Connected Graph** If there is a path from every vertex to every other vertex in a directed graph then it is said to be strongly connected graph. Otherwise, it is said to be weakly connected graph.

#### 4. Multiple edges

5. Distinct edges  $e$  and  $e'$  are called multiple edges if they connected the same and point.

Ex:  $e=(U,V)$  then  $e'=(U,V)$

## 6. Loop

An edge  $e$  is called loop if it has identical end points.

$E=(U,U)$

## 7. Path

A path is a sequence of distinct vertices, each adjacent to the next. The length of such a path is number of edges on that path.

## 8. Cycle

A path from a node to itself is called cycle. Thus, a cycle is a path in which the initial and final vertices are same.

**Acyclic Graph** A directed graph which has no cycles is referred to as acyclic graph. It is abbreviated as DAG [Directed Acyclic Graph]

**Note:** a graph need not be a tree but a tree must be graph

## 9. Degree, incidence and adjacent

A vertex  $V$  is incident to an edge  $e$  if  $V$  is one of the two vertices in the ordered pair of vertices that constitute  $e$ .

The degree of a vertex is the number of edges incident to it.

The **indegree** of vertex  $V$  is the number of edges that have  $V$  as head and the **outdegree** of vertex  $V$  is number of edges that have  $V$  as the tail.

A vertex  $V$  is **adjacent** to vertex  $U$  if there is an edge from  $U$  to  $V$ . If  $V$  is adjacent to  $U$ ,  $V$  is called a **successor** of  $U$ , and  $U$  a **predecessor** of  $V$ .

## 10. Weighted graph

A weighted graph is a graph in which edges are assigned weights. Weights of an edge are called as cost.

## 11. complete graph

If there is an edge from each vertex to all other vertices in graph is called as complete graph.

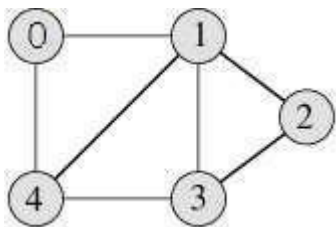
## Graph and its representations

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form  $(u, v)$  called as edge. The pair is ordered because  $(u, v)$  is not same as  $(v, u)$  in case of directed graph (di-graph). The pair of form  $(u, v)$  indicates that there is an edge from vertex  $u$  to vertex  $v$ . The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender and locale. See [this](#) for more applications of graph.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

### Adjacency Matrix:

Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of vertices in a graph. Let the 2D array be  $adj[][]$ , a

slot  $\text{adj}[i][j] = 1$  indicates that there is an edge from vertex  $i$  to vertex  $j$ . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If  $\text{adj}[i][j] = w$ , then there is an edge from vertex  $i$  to vertex  $j$  with weight  $w$ .

The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

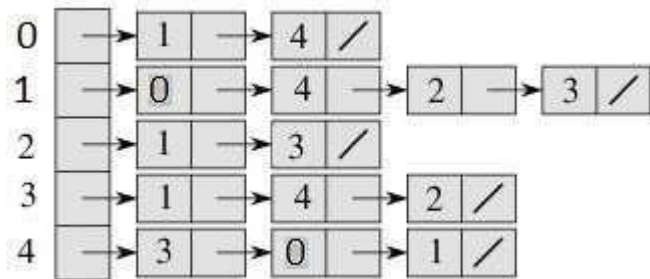
*Adjacency Matrix Representation of the above graph*

**Pros:** Representation is easier to implement and follow. Removing an edge takes  $O(1)$  time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done  $O(1)$ .

**Cons:** Consumes more space  $O(V^2)$ . Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is  $O(V^2)$  time.

#### Adjacency List:

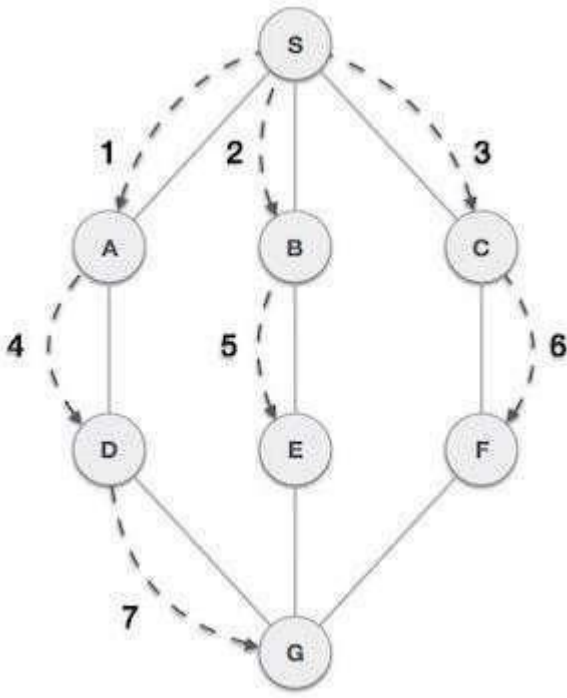
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be  $\text{array}[]$ . An entry  $\text{array}[i]$  represents the linked list of vertices adjacent to the  $i$ th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



*Adjacency List Representation of the above Graph*

## Data Structure - Breadth First Traversal

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



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

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

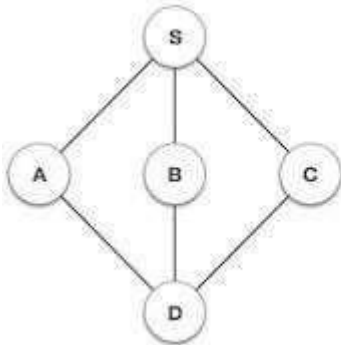
```
void BFS(int v1)
{
    front=rear=-1;
    for(int i = 0; i < n; i++)
        visited[i] = 0;
    visited[v1] = 1;
    Q[++rear]=v1;
    while(front!=rear())
    {
        v1 = Q[++front];
        printf("%d ", v1);
        // Get all adjacent vertices of the dequeued vertex v1
        // If a adjacent has not been visited, then mark it visited
        // and enqueue it
        for(int v2=0;v2<n;v2++)
        {
            if(g[v1][v2]==1&&visited[v2]==0)
            {
                visited[v2] = 1;
                Q[++rear]=v2;
            }
        }
    }
}
```

## Step

## Traversal

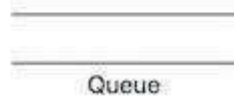
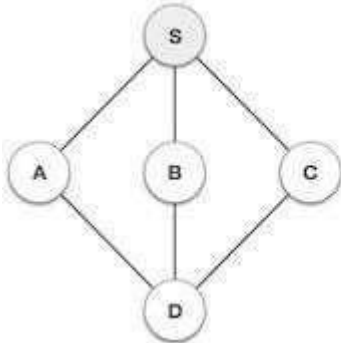
## Description

1.



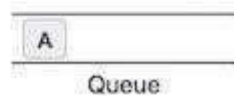
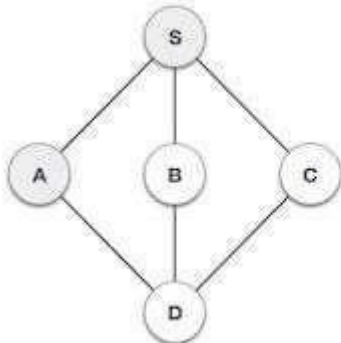
Initialize the queue.

2.



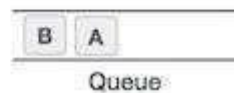
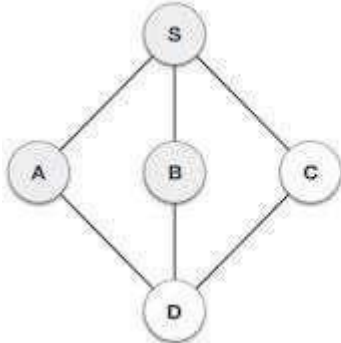
We start from visiting **S** (starting node), and mark it as visited.

3.



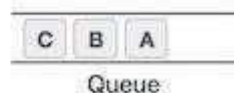
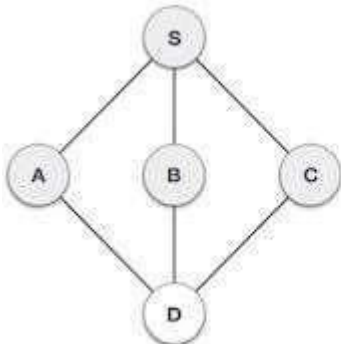
We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

4.

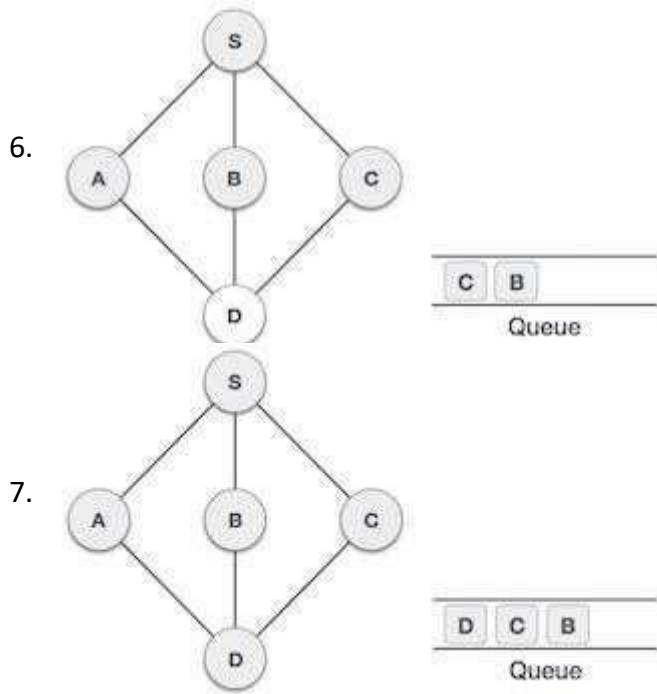


Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

5.



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



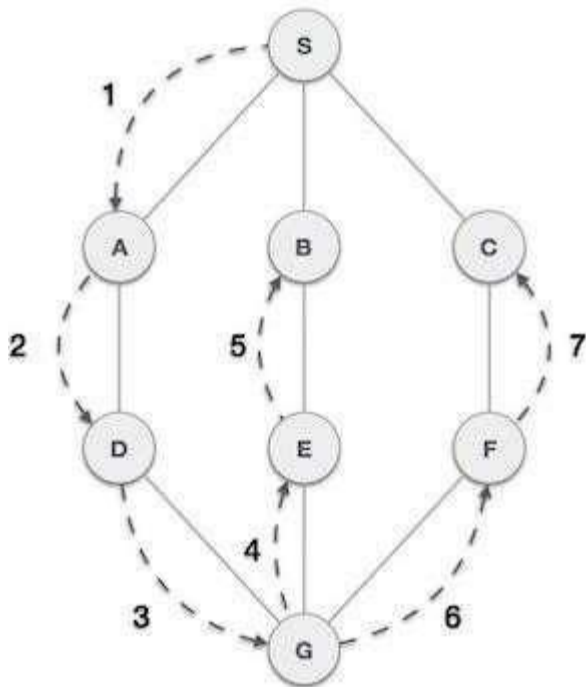
Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

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

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

## Data Structure - Depth First Traversal

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

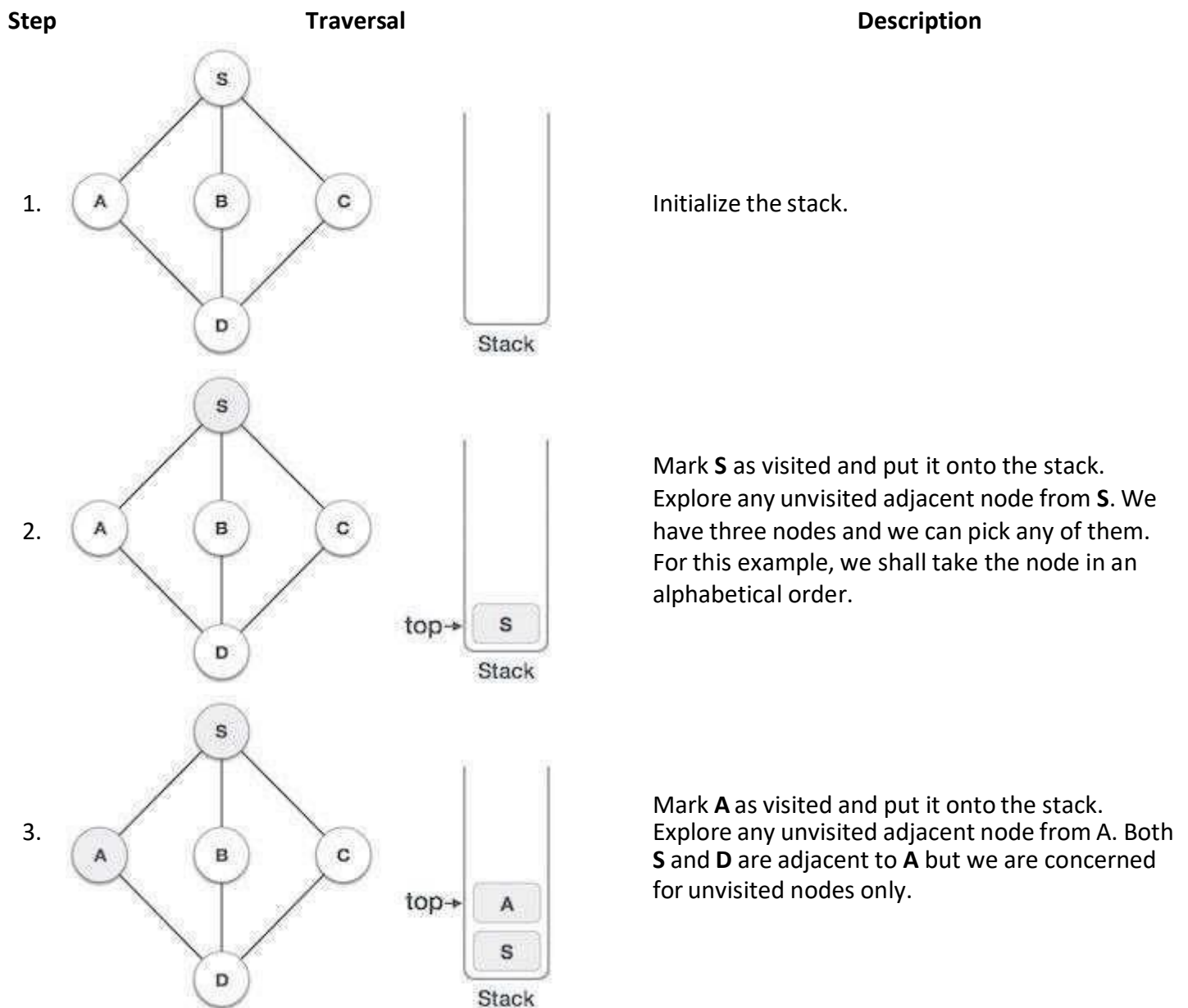


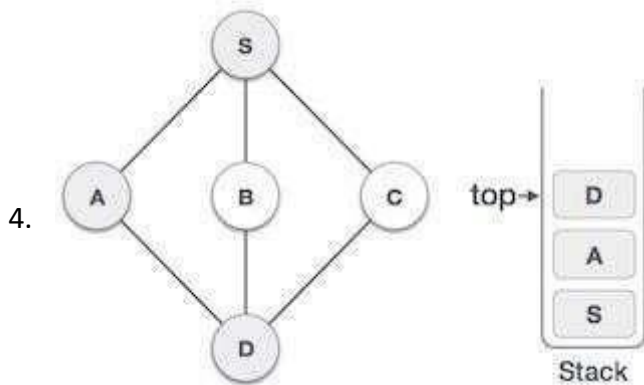
As in the example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

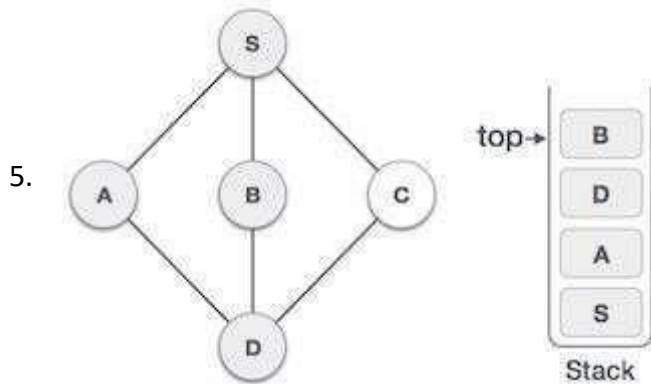
```
void dfs( Vertex v )
{
    v.visited = true;
    for each Vertex w adjacent to v if( !w.visited )
        dfs( w );
}
```

Figure Template for depth-first search (pseudocode)

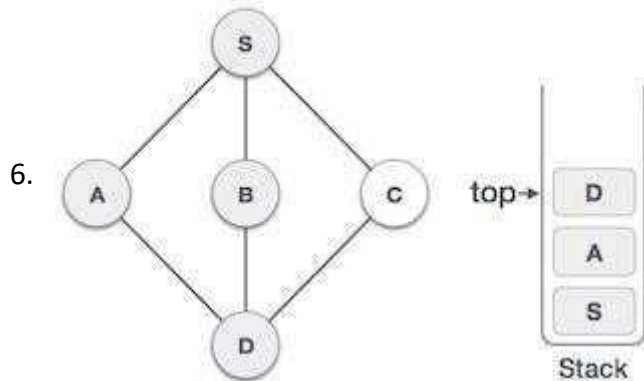




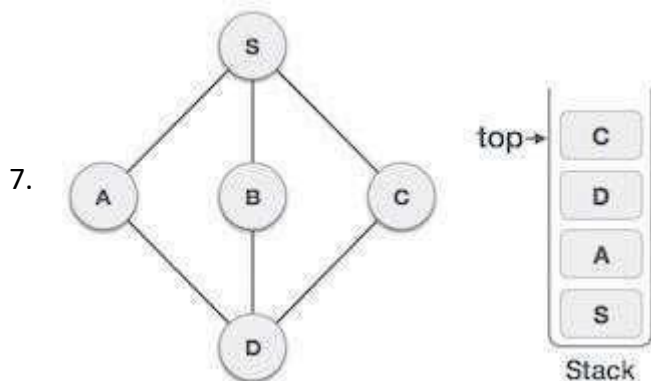
Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.



We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.



We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.



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

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

---

**UNIT V**  
**SEARCHING and SORTING ALGORITHMS****Syllabus**

Linear Search – Binary Search. Bubble Sort, Insertion sort – Merge sort – Quick sort – Hash tables – Overflow handling.

**SEARCHING**

- Linear Search
- Binary Search

**Linear Search**

A linear search scans one item at a time, without jumping to any item.

```
#include<stdio.h>
int main()
{
    int a[20],n,x,i,flag=0;
    printf("How many elements?");
    scanf("%d",&n);
    printf("\nEnter elements of the array\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("\nEnter element to search:");
    scanf("%d",&x);
    for(i=0;i<n;i++)
    {
        if(a[i]==x)
        {
            flag=1;
            break;
        }
    }
    if(flag)
        printf("\nElement is found at position %d ",i+1);
    else
        printf("\nElement not found");
    return 0;
}
```

The worst case complexity is  $O(n)$ , sometimes known as  $O(n)$  search

Time taken to search elements keep increasing as the number of elements are increased.

**Binary Search**

A binary search however, cut down your search to half as soon as you find middle of a sorted list.

The middle element is looked to check if it is greater than or less than the value to be searched. Accordingly, search is done to either half of the given list

```
#include<stdio.h>

int main()
{
    int n,i,a[100],f=0,l,h;
    printf("Enter the no. of Elements:");
    scanf("%d",&n);
    printf("\nEnter Elements of Array in Ascending order\n");
    for(i=0;i<n;++i)
    {
        scanf("%d",&a[i]);
    }
    printf("\nEnter element to search:");
    scanf("%d",&e);
    l=0;
    h=n-1;
    while(l<=h)
    {
        m=(l+h)/2;
        if(e==a[m])
        {
            f=1;
            break;
        }
        else
        {
            if(e>a[m])
                l=m+1;
            else
                h=m-1;
        }
    }
    if(f==1)
        cout<<"\nElement found at position "<<m+1;
    else
        cout<<"\nElement is not found ... !!!";
    return 0;
}
```

### Differences

- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search - $O(n)$  , Binary search has time complexity  $O(\log n)$ .
- Linear search performs equality comparisons and Binary search performs ordering comparisons

## SORTING

Sorting is linear ordering of list of items. Different types of sorting are

1. Bubble Sort
2. Insertion sort
3. Merge sort
4. Quick sort

### BUBBLE SORT ALGORITHM

**Bubble Sort** is a simple algorithm which is used to sort a given set of  $n$  elements provided in form of an array with  $n$  number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on. This process will be repeated for  $n-1$  times. ( $n$ - total elements)

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

#### **Bubble Sort:**

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
}
```

```

}
}

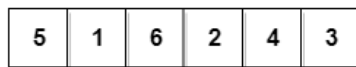
int main()
{
    int arr[100], i, n, step, temp;
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    for(i = 0; i < n; i++)
    {
        printf("Enter elements\n");
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}

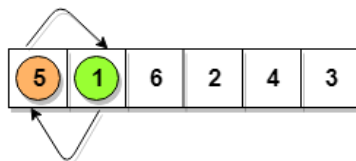
```

**Example:** Let's consider an array with values {5, 1, 6, 2, 4, 3}

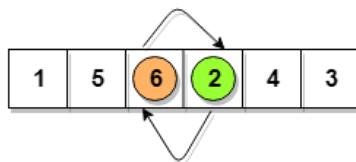
5>1  
so interchange



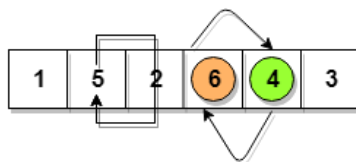
5<6  
No swapping



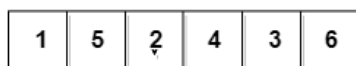
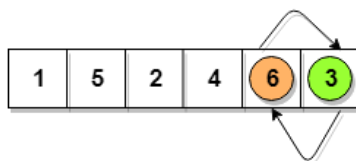
6>2  
so interchange



6>4  
so interchange



6>3  
so interchange



This is first insertion

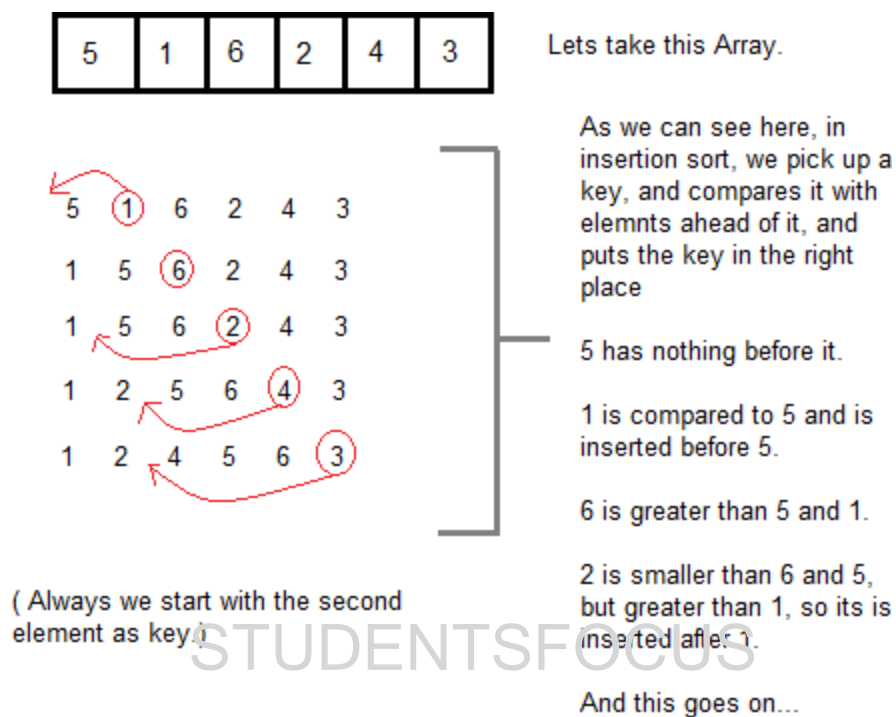
similarly, after all the  
iterations, the array  
gets sorted

As shown above, after the first iteration, 6 is placed at the last index, which is the correct position for it. Similarly after the second iteration, 5 will be at the second last index, and so on.

## INSERTION SORTING

1. It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.
2. It has one of the simplest implementation
3. It is efficient for smaller data sets, but very inefficient for larger lists.
4. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
5. It is better than Selection Sort and Bubble Sort algorithms.
6. Its space complexity is less, like Bubble Sorting, insertion sort also requires a single additional memory space.
7. It is **Stable**, as it does not change the relative order of elements with equal keys

### *How Insertion Sorting Works*



#### INSERTION SORT ROUTINE

```
void insert(int a[],int n)
{
    int i,j,temp;
    for(i=0;i<n;i++)
    {
        temp=a[i];
        for(j=0;j>0&& a[j-1]>temp;j--)
            a[j]=a[j-1];
        a[j]=temp;
    }
}
```

**Complexity Analysis of Insertion Sorting****Worst Case Time Complexity :**  $O(n^2)$ **Best Case Time Complexity :**  $O(n)$ **Average Time Complexity :**  $O(n^2)$ **Space Complexity :**  $O(1)$ **QUICK SORT ALGORITHM**

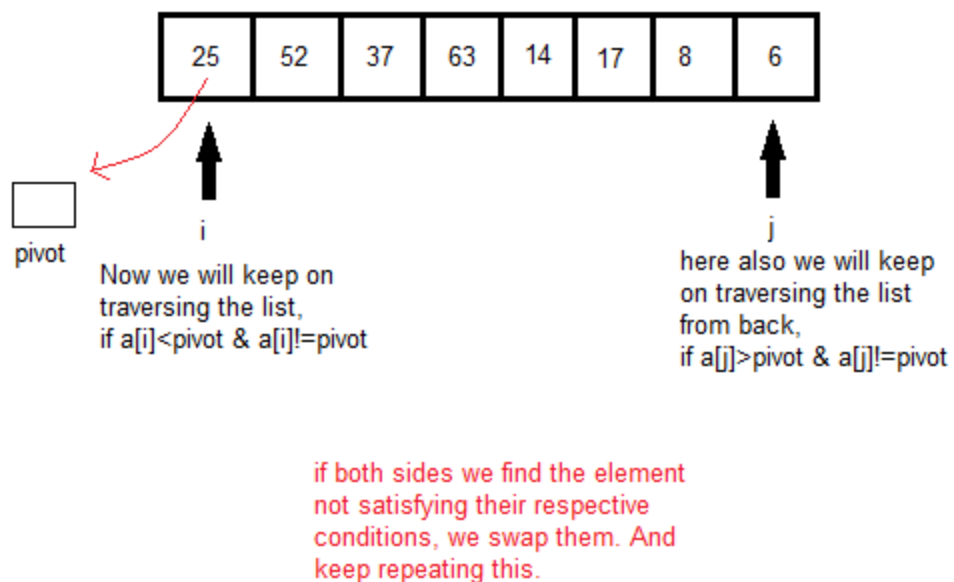
Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of **Divide and Conquer**(also called *partition-exchange sort*). This algorithm divides the list into three main parts :

1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 **25** 63 37 52

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

**DIVIDE AND CONQUER - QUICK SORT****How Quick Sorting Works**

```
void qsort(int arr[], int left, int right)
{
    int i,j,pivot,tmp;
    if(left<right)
```

```

{
    pivot=left;
    i=left+1;
    j=right;
    while(i<j)
    {
        while(arr[i]<=arr[pivot] && i<right)
            i++;
        while(arr[j]>arr[pivot])
            j--;
        if(i<j)
        {
            tmp=arr[i];
            arr[i]=arr[j];
            arr[j]=tmp;
        }
    }
    tmp=arr[pivot];
    arr[pivot]=arr[j];
    arr[j]=tmp;
    qsort(arr,left,j-1);
    qsort(arr,j+1,right);
}
}

```

### Complexity Analysis of Quick Sort

**Worst Case Time Complexity :**  $O(n^2)$

**Best Case Time Complexity :**  $O(n \log n)$

**Average Time Complexity :**  $O(n \log n)$

**Space Complexity :**  $O(n \log n)$

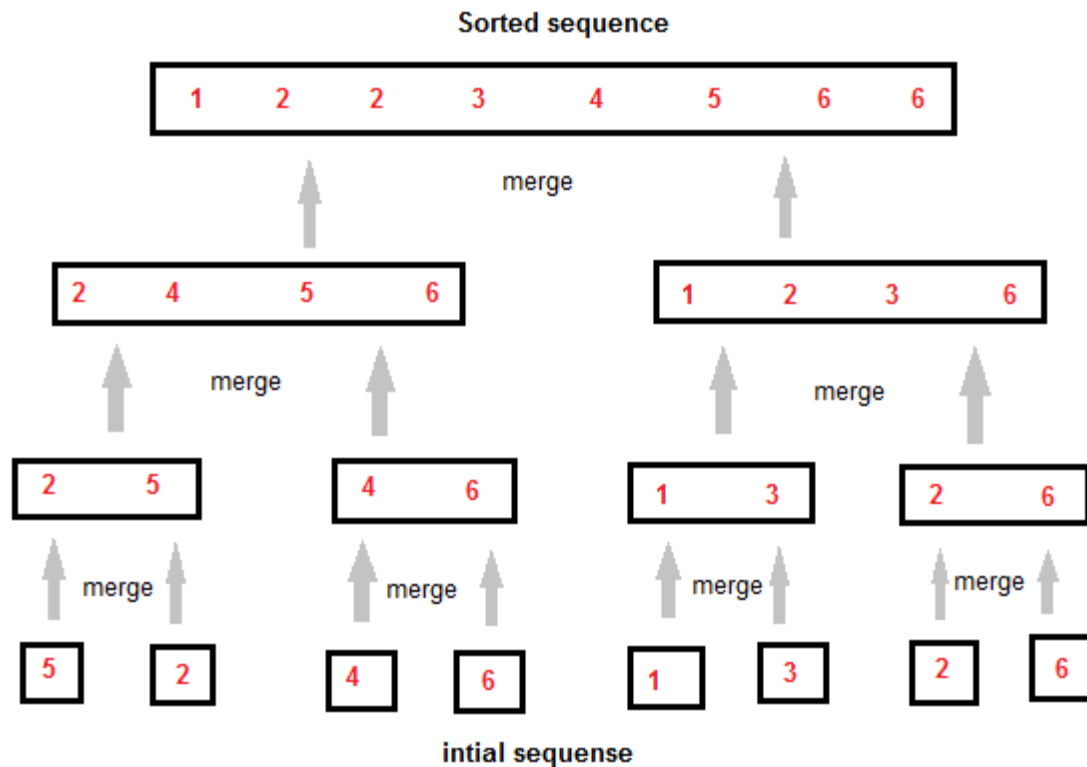
- ➡ Space required by quick sort is very less, only  $O(n \log n)$  additional space is required.
- ➡ Quick sort is not a stable sorting technique, so it might change the occurrence of two similar elements in the list while sorting.

### MERGE SORT ALGORITHM

Merge Sort follows the rule of **Divide and Conquer**. But it doesn't divide the list into two halves. In merge sort the unsorted list is divided into  $N$  sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at last one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of  **$O(n \log n)$** . It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

### How Merge Sort Works



Like we can see in the above example, merge sort first breaks the unsorted list into sorted sublists, and then keep merging these sublists, to finally get the complete sorted list.

#### Sorting using Merge Sort Algorithm

```
#include<stdio.h>
#include<conio.h>
void merge(int [],int ,int ,int );
void part(int [],int ,int );
int main()
{
    int arr[30];
    int i,size;
    printf("\n\t----- Merge sorting method ----- \n\n");
    printf("Enter total no. of elements : ");
    scanf("%d",&size);
    for(i=0; i<size; i++)
    {
        printf("Enter %d element : ",i+1);
        scanf("%d",&arr[i]);
    }
    part(arr,0,size-1);
    printf("\n\t----- Merge sorted elements ----- \n\n");
    for(i=0; i<size; i++)
        printf("%d ",arr[i]);
    getch();
}
```

```
return 0;
}
```

```
void part(int arr[],int min,int max)
{
    int mid;
    if(min<max)
    {
        mid=(min+max)/2;
        part(arr,min,mid);
        part(arr,mid+1,max);
        merge(arr,min,mid,max);
    }
}
```

```
void merge(int arr[],int min,int mid,int max)
{
    int tmp[30];
    int i,j,k,m;
    i=min;
    j=mid+1;k=0;
    while(i<=mid && j<=max )
    {
        if(arr[i]<=arr[j])
            tmp[k++]=arr[i++];
        else
            tmp[k++]=arr[j++];
    }
    while(i<=mid)
        tmp[k++]=arr[i++];
    while(j<=max)
        tmp[k++]=arr[j++];
    for(k=min; k<=max; k++)
        arr[k]=tmp[k];
}
```

### Complexity Analysis of Merge Sort

Worst Case Time Complexity :  $O(n \log n)$

Best Case Time Complexity :  $O(n \log n)$

Average Time Complexity :  $O(n \log n)$

Space Complexity :  $O(n)$

Time complexity of Merge Sort is  $O(n \log n)$  in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves. It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists. It is the best Sorting technique for sorting Linked Lists.

## HASHING

### Hashing

Hashing is a process which uses a function to get the key and using the key it quickly identifies the record, without much strain. The values returned by a hash function are called hash values. Hash table is data structure in which key values are placed in array location

### Hash Table

The hash table data structure is an array of some fixed size, containing the keys. A key is a value associated with each record.

There are two types of hashing :

1. **Static hashing**: In static hashing, the hash function maps search-key values to a fixed set of *locations*.
2. **Dynamic hashing**: In dynamic hashing a hash table can grow to handle more items. The associated hash function must change as the table grows.

A **hash function**, **h**, is a function which transforms a key from a set, **K**, into an index in a table of size **n**:

$$h: K \rightarrow \{0, 1, \dots, n-2, n-1\}$$

- A key can be a number, a string, a record etc. The size of the set of keys, **|K|**, to be relatively very large. It is possible for different keys to hash to the same array location. This situation is called *collision* and the colliding keys are called *synonyms*.
- 

A good hash function should:

- Minimize collisions.
- Be *easy* and *quick* to compute.
- Distribute key values *evenly* in the hash table.
- Use *all the information* provided in the key.

### Various types of Hash Functions:

Type 1: Truncation Method

Type 2: Folding Method

Type 3: Midsquare Method

Type 4: Division Method (Modulo Division)

### 1 Truncation Method

The Truncation Method truncates a part of the given keys, depending upon the size of the hash table.

1. Choose the hash table size.
2. Then the respective right most or left most digits are truncated and used as hash code | value.

Ex: 123,42,56 Table size = 9

$H(123)=1$

$H(42)=4$

$H(56)=5$

0	
1	123
2	
3	
4	42
5	56
6	
7	
8	
9	

## 2 Mid square Method :

It is a Hash Function method.

1. Square the given keys.
2. Take the respective middle digits from each squared value and use that as the hash value | address | index | code, for the respective keys.

$$H(123)=1 \text{ [ } 123^2 = 15\textbf{1}29 \text{ ]}$$

$$H(42)=7 \text{ [ } 42^2 = 17\textbf{6}4 \text{ ]}$$

$$H(56)=3 \text{ [ } 56^2 = 31\textbf{3}6 \text{ ]}$$

0	
1	123
2	
3	56
4	
5	
6	
7	42
8	
9	

## 3 Folding Method:

Partition the key K into number of parts, like K1,K2,.....Kn, then add the parts together and ignore the carry and use it as the hash value.

$$H(123)= [ 1+2+3=6 ]$$

$$H(43)= [ 4+3=7 ]$$

$$H(56)= [ 5+6=11 ]$$

0	
1	56
2	
3	
4	
5	
6	123
7	43

## 4 Division Method :

Choose a number m, larger than the number of keys. The number m is usually chosen to be a prime number.

The formula for the division method :

$$\text{Hash(key)} = \text{key \% tablesize}$$

Tablesize : 10 20,21,24,26,32,34

$$H(20)= 20 \% 10 = 0$$

$$H(21)= 21 \% 10 = 1$$

$$H(24)=24 \% 10 = 4$$

$$H(26) = 26 \% 10 = 6$$

$$H(32) = 32 \% 10 = 2$$

$$H(34) = 34 \% 10 = 4 \quad 34 \text{ IS COLLISION}$$

0	20
1	21
2	32
3	
4	24
5	
6	26
7	42
8	
9	

### Applications

- **Database systems:** Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.
- **Symbol tables:** The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.
- **Data dictionaries:** Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.
- **Network processing algorithms:** Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.
- **Browser Cashes:** Hash tables are used to implement browser caches.

### HASH COLLISION:

A hash collision or hash clash is a situation that occurs when two distinct inputs into a hash function produce identical outputs.

### COLLISION RESOLUTION TECHNIQUES

The techniques are:

#### Closed Addressing

1. Separate Chaining.

#### Open Addressing

2. Linear Probing.
3. Quadratic Probing.
4. Double Hashing.

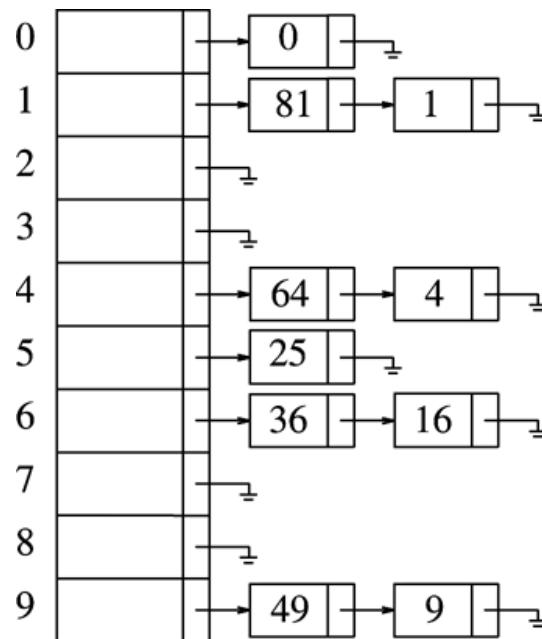
#### Dynamic Hashing

5. Rehashing.
6. Extendable Hashing.
7. Linear Hashing.

## 1 Separate Chaining

It is to keep a list of all elements that hash to the same value. An alternative to open addressing as a method of collision resolution is **separate chaining** hashing. This uses an array as the primary hash table, except that the array is an array of **lists** of entries, each list initially being empty.

If in a set of elements, if an element hashes to the same value as that of an already inserted element then we have a collision and we need to resolve it. In separate chaining, each slot of the bucket array is a pointer to a linked list that contains key-value pairs that are hashed to the same location. It is otherwise called as direct chaining or simply chaining. An array of linked list implementation is used here.



### SEPARATE CHAINING PROCEDURE :

#### TO FIND AN ELEMENT

- To perform a Find, we use the hash function to determine which list to traverse.
- We then traverse it in normal manner, returning the position where the item is found.
- Finding an element in separate chaining is very much similar to the find operation performed in the case of lists.

If the ElementType is a string then a comparison and assignment must be done with *strcmp* and *strcpy* respectively.

#### TO INSERT AN ELEMENT

- To perform an Insert, we traverse down the appropriate list to check whether the element is already in place .
- If it is new then it is either inserted at the front or at the end.
- If the item to be inserted is already present, then we do not perform any operation; otherwise we place it at the front of the list. It is similar to the insertion Operation that takes place in the case of linked lists.
- The disadvantage is that it computes the hash function twice.

#### TO DELETE AN ELEMENT:

- To delete we find the cell P prior to the one containing the element to be deleted.

- Make the cell P to point to the next cell of the element to be deleted.
- Then free the memory space of the element to be deleted.

## PROGRAM

### HEADER FILE FOR SEPARATE CHAINING

```

Typedef int elementtype;
Typedef struct listnode *position;
Typedef struct hashtable *hashtable;
Typedef position list;
Struct hashtable
{
int tablesize;
list *thelists;
};

```

### IMPLEMENTATION OF SEPARATE CHAINING

The Lists will be an array of list.

```

Struct listnode
{
Elementtype element;
Position next;
};
Hashtable initialize table(int tablesize)
{
Hashtable H;
int i;
/*Allocate Table */
H=malloc(sizeof(struct hashtable));
If(H==NULL)
Fatalerror("Out of Space");
H-->tablesize=nextprime(tablesize);
/*Allocate array of list */
H-->thelist=malloc(sizeof(list)*H-->tablesize);
If(H-->thelist==NULL)
Fatalerror("Out of Space");
/*Allocate list header */
For(i=0;i<H-->tablesize;i++)
{
H-->thelists[i]=malloc(sizeof(struct listnode));
If(H-->thelists[i]==NULL)
Fatalerror("Out of Space");
Else
H-->thelists[i]-->next=NULL;
}
return H;
}
Hash(char *key, int tablesize)
{
int hashvalue=0;

```

```

while(*key!='\0')
hashvalue=hashvalue+ *key++;
return(hashvalue % tablesize);
}
Position find(Elementtype key, hashtable H)
{
Position P;
List L;
L=H-->thelists(Hash(key,H-->tablesize));
P=L-->next;
While(P!=NULL && P-->element !=key)
P=P-->next;
Return P;
}
Void insert(elementtype key, hashtable H)
{
Position pos,newcell;
List L;
Pos=find(key,H);
If(pos==NULL)
{
Newcell=malloc(sizeof(struct listnode));
If(newcell==NULL)
Fatalerror("Out of Space");
else
{
L=H-->thelists(hash(key,H-->tablesize));
Newcell-->next=L-->next;
Newcell-->element=key;
L-->next=Newcell;
}
}
}

```

#### ADVANTAGES:

- Separate chaining is used when memory space is a concern.
- It can be very easily implemented.

#### DISADVANTAGES:

- It requires pointers which causes the algorithm to slow down a bit.
- Unevenly distributed keys-long lists-search time increases.

#### Open Addressing

In an open addressing hashing system, if a collision occurs, alternative cells are tried until an empty cell is found.

A bigger table is needed for open addressing hashing, than for separate chaining.

Types of Open Addressing :

1. Linear Probing.
2. Quadratic Probing.
3. Double Hashing.

**Linear Probing**

It is a kind of Open Addressing. In Linear probing,  $F$  is a linear function of  $i$ , typically  $F(i)=i$ . In linear probing, the position in which a key can be stored is found by sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found.

If the end of the table is reached and no empty cell have been found, then the search is continued from the beginning of the table. It has a tendency to create cluster in the table.

In linear probing we get primary clustering problem. Primary Clustering Problem

If the Hashtable becomes half full and if a collision occurs, it is difficult to find an empty location in the hash table and hence an insertion or the deletion process takes a longer time.

Hash function

**$hi(\text{key}) = (\text{Hash}(\text{key}) + F(i)) \% \text{Tablesize}$**   **$F(i)=i$**   **$\text{Hash}(\text{key}) = \text{key} \% \text{tablesize}$**

Ex: 89,18,49,58,69

-> Hashtable

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	

1.  $\text{Hash}(89) = 89 \% 10 = 9$

2.  $\text{Hash}(18) = 18 \% 10 = 8$

3.  $\text{Hash}(49) = 49 \% 10 = 9$  -> collision occurs for first time  $i=1$

$h1(49) = (9+1) \% 10 = 10 \% 10 = 0$

4.  $\text{Hash}(58) = 58 \% 10 = 8$  -> collision occurs for first time  $i=1$

$h1(58) = (8+1) \% 10 = 9$  -> collision occurs for first time  $i=2$

$h2(58) = (8+2) \% 10 = 0$  -> collision occurs for first time  $i=3$

$h3(58) = (8+3) \% 10 = 1$

5.  $\text{Hash}(69) = 69 \% 10 = 9$  -> collision occurs for first time  $i=1$

$h1(69) = (9+1) \% 10 = 0$  -> collision occurs for first time  $i=2$

$h2(69) = (9+2) \% 10 = 1$  -> collision occurs for first time  $i=3$

$h3(69) = (9+3) \% 10 = 2$

**Advantage:**

It does not require pointers.

**Disadvantage:**

It forms clusters, which degrades the performance of the hash table for storing and retrieving data.

**Quadratic Probing**

It is a kind of open addressing technique. It is a collision resolution method that eliminates the primary clustering problem of linear probing.

-> Hash function

**$hi(\text{key}) = (\text{Hash}(\text{key}) + F(i)) \% \text{Tablesize}$**

**$F(i)=i^2$**   **$\text{Hash}(\text{key}) = \text{key} \% \text{tablesize}$**

-> Hashtable Ex: 89,18,49,58,69

0	49
1	58

2	69
3	
4	
5	
6	
7	18
8	89
9	

1.Hash(89)=89 % 10=9

2.Hash(18)=18 % 10= 8

3.Hash(49)=49 % 10=9 -> collision occurs for first time i=1

$h_1(49) = (9+12) \% 10 = 10 \% 10 = 0$

4.Hash(58)= 58 % 10 =8 -> collision occurs for first time i=1

$h_1(58) = (8+12) \% 10 = 9$  -> collision occurs for first time i=2

$h_2(58) = (8+22) \% 10 = 2$

5.Hash(69)=69%10=9 -> collision occurs for first time i=1

$h_1(69) = (9+12) \% 10 = 9$  -> collision occurs for first time i=2

$h_2(69) = (9+22) \% 10 = 3$

Secondary Clustering:

Elements that Hash to the same position will probe the same alternative cells. This is known as secondary clustering.

### Double Hashing

Double hashing is a technique which belongs to open addressing. Open addressing is a collision resolution technique which uses the concept of linked lists. Open addressing handles collision by trying out all the alternative cells until an empty cell is found. Collision is said to have occurred if there exists this situation.i.e., If an element inserted hashes to the same value as that of an already inserted element, then there is collision

PROCEDURE:

- Compute the positions where the data elements are to be inserted by applying the first hash function to it.
- Insert the elements if the positions are vacant.
- If there is collision then apply the second hash function.
- Add the two values and insert the element into the appropriate position.
- Number of probes for the data element is 1 if it is inserted after calculating first hash function.
- Number of probes is 2 if it is inserted after calculating second hash function.

DEFINITION:

It is a collision resolution technique which uses two hash functions to handle collision.

The interval (i.e., the distance it probes) is decided by the second hash function which is independent.

REQUIREMENTS:

The table size should be chosen in such a way that it is prime so that all the cells can be inserted. The second hash function should be chosen in such a way that the function does not evaluate to zero.i.e., 1 can be added to the hash function(non-zero). To overcome secondary clustering, double hashing is used. The collision function is,

**$h_i(\text{key}) = (\text{Hash}(\text{key}) + F(i)) \% \text{Tablesize}$**

**$F(i) = i * \text{hash2}(X)$**

Where  **$\text{hash2}(X) = R - (X \% R)$**  R is a prime number. It should be smaller than the tablesize

**Example 1 :**

89, 18, 49, 58, 69, 60

0	69
1	
2	60
3	58
4	
5	
6	49
7	18
8	89
9	

49

$h_0(49)=9$

$h_1(49) = (9 + (1 * 7)) \% 10 = (9 + 7) \% 10 = 16 \% 10 = 6$

58

$7 - (58 \% 7) = 7 - 2 = 5$

$i=1$

$h_1(58) = (8 + (1 * 5)) \% 10 = 13 \% 10 = 3$

69

$7 - (69 \% 7) = 7 - 6 = 1$

$h_1(69) = (9 + 1) \% 10 = 10 \% 10 = 0$

60

$7 - (60 \% 7) = 7 - 4 = 3$

$h_1(60) = (0 + 1 * 3) \% 10 = 3$

$h_2(60) = (0 + 2 * 3) \% 10 = 6$

$h_3(60) = (0 + 9) \% 10 = 9$

$h_4(60) = (0 + 12) \% 10 = 2$

#### APPLICATIONS:

- It is used in caches.
- It is used in finding duplicate records.
- Used in finding similar records.
- Finding similar substrings.

#### ADVANTAGES:

- No index storage is required.
- It provides rapid updates.

#### DISADVANTAGES:

- Problem arises when the keys are too small.
- There is no sequential retrieval of keys.
- Insertion is more random.

#### Re-Hashing:

It is a technique in which the table is re-sized i.e., the size of the table is doubled by creating a new table. Rehashing is a technique that is used to improve the efficiency of the closed hashing techniques. This can be done by reducing the running time. If the table gets too full, the running time for the operations will start

taking too long and *inserts* might fail for closed hashing with quadratic resolution. This can happen if there are too many deletions intermixed with insertions. A solution, then, is to build another table that is about twice as big (with associated new hash function) and scan down the entire original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new table. Re-Hashing is required when, The table is completely filled in the case of double-Hashing

The table is half-filled in the case of quadratic and linear probing

The insertion fails due to overflow.

#### IMPLEMENTATION:

Rehashing can be implemented in

1. Linear probing

2. Double hashing

3. Quadratic probing

- Rehashing can be implemented in several ways with quadratic probing.
- One alternative is to rehash as soon as the table is half full.
- The other extreme is to rehash only when an insertion fails.
- A third, middle of the road, strategy is to rehash when the table reaches a certain load factor. Since performance does degrade as the load factor increases, the third strategy, implemented with a good cutoff, could be best.

#### Problems:

1. According to linear probing,

13, 15, 6, 24, 23

0	6
1	15
2	23
3	24
4	
5	
6	13

#### Rehashing

Size=7 double size=14 next prime =17

1.  $16 \% 17 = 6$

2.  $15 \% 17 = 15$

3.  $23 \% 17 = 6$

$h_1(X) = 6 + 1 \% 17 = 7$

4.  $24 \% 17 = 7$

$h_1(24) = 7 + 1 \% 17 = 8$

5.  $13 \% 17 = 13$

0	
1	
2	
3	
4	

5	
6	16
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Advantage:

This technique provides the programmer the flexibility to enlarge the table size if required.

Disadvantages:

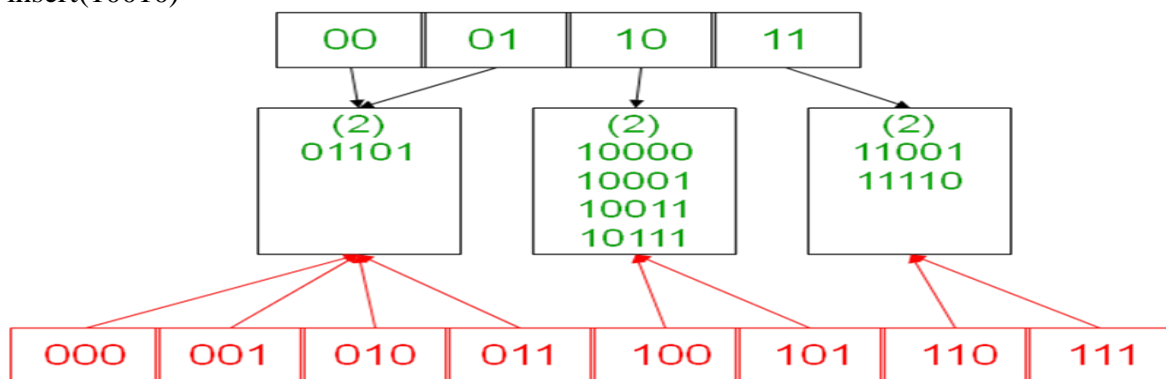
Transfer time is more.

## Extensible Hashing

Hashing technique for huge data sets

- optimizes to reduce disk accesses
- each hash bucket fits on one disk block
- better than B-Trees if order is not important
- Table contains
  - buckets, each fitting in one disk block, with the data
  - a directory that fits in one disk block used to hash to the correct bucket
- Directory - entries labeled by  $k$  bits & pointer to bucket with all keys starting with its bits
- Each block contains keys & data matching on the first  $j \leq k$  bits

- insert(10010)



directory for  $k = 3$

The diagram illustrates the Huffman tree construction process. It shows two stages of the tree structure.

**Top Stage (Initial Tree):**

- Root: 000
- Internal nodes: 001, 010, 011, 100, 101, 110, 111
- Leaf nodes (under 001): (2) 00001, 00011, 00100, 00110
- Leaf nodes (under 010): (2) 01001, 01011, 01100
- Leaf nodes (under 011): (3) 10001, 10011
- Leaf nodes (under 101): (3) 10101, 10110, 10111
- Leaf nodes (under 110): (2) 11001, 11100, 11110

**Bottom Stage (Tree after merging 00001 and 00011):**

- Root: 000
- Internal nodes: 001, 010, 011, 100, 101, 110, 111
- Leaf nodes (under 001): (2) 00001, 00011, 00100, 00110
- Leaf nodes (under 010): (2) 01001, 01011, 01100
- Leaf nodes (under 011): (3) 10001, 10011
- Leaf nodes (under 101): (3) 10101, 10110, 10111
- Leaf nodes (under 110): (2) 11001, 11100, 11110

A large black arrow points from the top stage to the bottom stage, indicating the transformation. In the bottom stage, the leaf nodes 11001 and 11011 are highlighted in red, indicating they are the two smallest nodes to be merged.

- Extendable hashing provides performance that does not degrade as the file grows.
- Minimal space overhead - no buckets need be reserved for future use. Bucket address table only contains one pointer for each hash value of current prefix length.

- Extra level of indirection in the bucket address table
- Added complexity