CS3353 - C PROGRAMMING AND DATA STRUCTURES

UNIT - I C PROGRAMMING FUNDAMENTALS

Data Types - Variables - Operations - Expressions and Statements - Conditional Statements - Functions - Recursive Functions - Arrays - Single and Multi-Dimensional Arrays.

C Programming Basics:

→ C is developed by "Dennis Ritche" at AT&T Bell Laboratories at USA in 1972

→ C is a powerful Structured / Procedure - Oriented Programming Language.

Features of Procedure - Oriented Programming Language:

(i) It specifies the steps the program must take to reach the desired state.

(ii) It follows Top-down approach.

(iii) Functions play an important role.

(iv) Global data

(v) Easy movement of data and the data can be directly accessed. Hence there is no security to data.

Structure of a C Program:

→ C Language is block structured and variables declared within the blocks are allocated automatically upon block entry.

Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Declaration
Main Function Definition Declaration Part Execution Part
Function Definition Function 1 Function 2 ⋮ Function N

Documentation Section:

→ This section is used for specifying Comments.

⇒ Comments → Two ways of representing the Comment

(i) Single Line Comment ⇒ //

(ii) Multiple Line Comment ⇒ /* Line 1
Line 2
*/

Link Section:

→ It is used to instruct the Compiler to connect to the various functions of header files and preprocessors from the system Library.

→ two types of Link section

(i) Header Files

(ii) Preprocessors.

Header Files:

→ A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.

→ Header files are included in C program to use the corresponding built in functions.

Syntax of using Header Files:

(i) #include <file>

→ This form is used for system header files.

(ii) #include "file"

→ This form is used for header files of our own program

⇒ Stdio – Standard Input Output Stream

printf() → used to print the character
scanf() → used to read the character
getc() → Reads the character from file
putc() → Writes a character to file
getchar() → Reads a character from keyboard
putchar() → Writes a character to screen
gets() → Reads line from keyboard
puts() → Writes line to output screen.

⇒ Conio – Console Input Output

clrscr() → Clears the screen
putch() → Writes a character directly to the Console
getch() → Get char entry from the Console.

⇒ Math - Mathematical Functions

abs() - Computes absolute value of an integer value

div() - Computes the quotient and remainder

sqrt() - Computes square root

pow() - Raises a number to the given power

sin(), cos(), tan() - Computes sine, cosine, tangent values

ceil() - Returns the nearest integer not less than the given value

floor() - Returns the nearest integer not greater than the given value.

⇒ String - String Processing

strlen (str1) - Gives the length of str1

strcat (str1, str2) - Concatenates str2 at the end of str1

strcpy (str1, str2) - Copies str2 into str1

strcmp (str1, str2) - Returns 0 if str1 = str2
Returns <0 if str1 < str2
Returns >0 if str1 > str2

strrev() - Reverses the given string.

Preprocessor :

→ Preprocessor is a program that processes our source code before compilation. The processor is a part of the compiler.

→ Preprocessor directive begin with a # (hash) symbol.

→ This # symbol at the beginning of a statement in a C program.

→ Each preprocessor directive must be its own line.

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

Definition Section:

→ describes all the symbolic constants.

→ In C program user may define the value of a constant using macros which we have defined earlier.

eg #define PI 3.14

↳ Whenever we are using PI in our C program, automatically the value 3.14 will be expanded for PI.

Global Declaration Section:

→ Variables that can be accessible throughout the program can be declared.

→ A variable with a global scope is accessible everywhere inside the program.

Main Function Definition:

→ Every C program starts always from the function main(). The computation that we want to solve will be given in the main function.

→ main() has two parts.

① Declaration Part - to declare all the variables used in executable part.

② Executable Part - is the actual program where processing is done.

Sub Program Section:

→ User defined functions will be defined, that are called by main() function.

C Tokens :

→ The tokens are usually referred as individual text and punctuation in a passage of text. C tokens have following types.

① keywords :

→ keywords are having fixed meaning
→ keywords are also called Reserved words.
→ There are 32 keywords in C.

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

② Identifiers / Variables :

→ a variable or an identifier is an entity that may change.
→ A particular type of variable can hold only the same type of constant.

eg int a=5 , float f = 2.45 char ch='y'

③ Constants :

→ It refers a fixed value – could not be changed during the execution.

Intoger Constant → 10, -125, 4500

Floating point Constant → 1.5, -41.870001

Character Constant → 'a', 'z', 'm'

String Constant → "India", "Success"

④ Operators :

Arithmetic Operators → +, −, *, /, %

Relational Operators → <, <=, >, >=, ==, !=

Logical Operators → && (AND), || (OR), ! (NOT)

Assignment Operators → +=, −=, *=, /=, %=

Increment / decrement operators → ++, −−

Pre increment & Post Increment → ++a, a++

Pre decrement & Post decrement → −−a, a−−

Conditional operator or Ternary → ?, :

Bitwise Operator → & (AND), | (OR), ^ (EX-OR), <<, >>

Special operator → sizeof(), *

⑤ Strings :

→ Collection of character is known as string

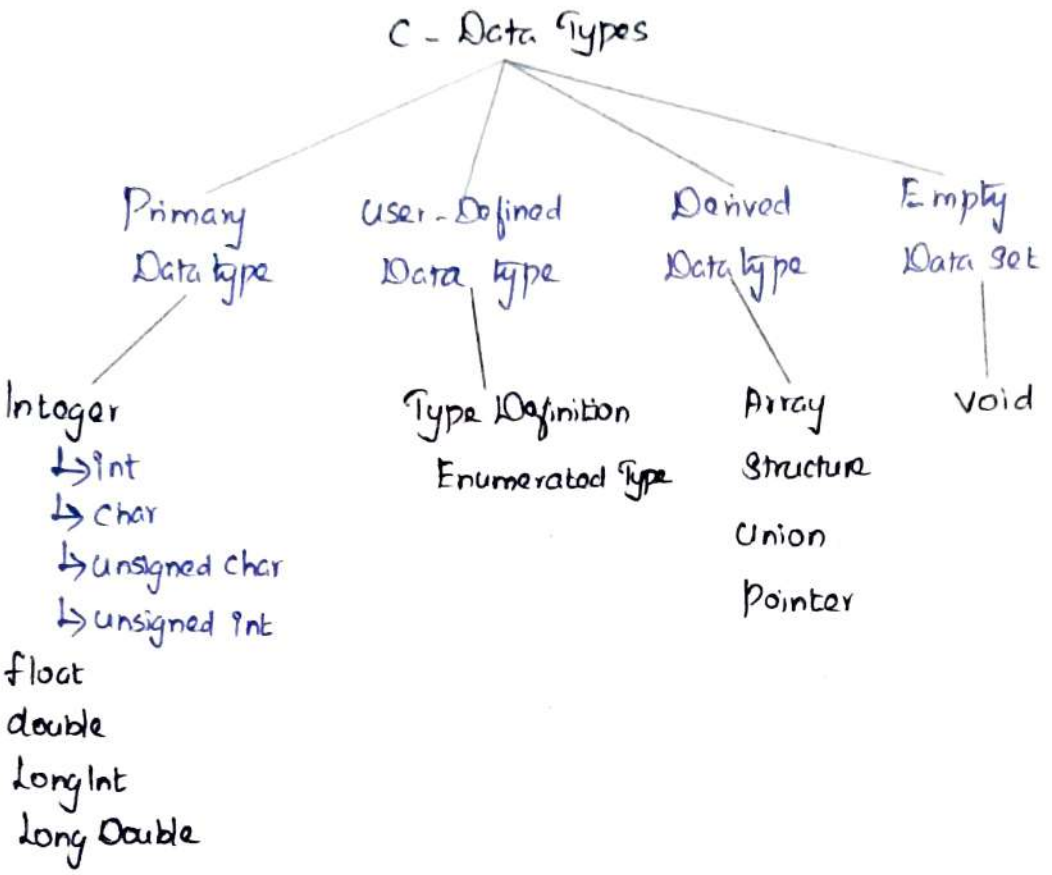
→ Sequence of characters enclosed within a double quote is known as String Constant

eg char name[15]

⑥ Special symbols :

→ Special symbols include : [] # () [] $

Data Types:



→ Data types specify the type of data we enter in our program. In C there are some predefined set of data types which are also called as primitive data types.

→ Primitive data types are fundamental data types or primary data types.

Primitive Datatype or Primary Data type:

① Integer datatype:

→ allows a variable to store numeric values.

→ These data types are used to store the whole number (the number without fraction).

Size and Range of Integer Type on 16-bit Machine

Type	Size (bytes)	Range
int or signed int	2	-32768 to 32767
Unsigned int	2	0 to 65535
Short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2147,483,648 to 2147,483,647
unsigned long int	4	0 to 4,294,967,295

② Float data type:

→ These are the data types used to store the real numbers. (the numbers with fractional part).

Size and range of Integer type on 16-bit Machine

Type	Size (bytes)	Range
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

③ Char data type

→ This data type is used to store the character value.

Size and Range of Integer Type on 16-bit Machine

Type	Size (bytes)	Range
Char or Signed char	1	-128 to 127
Unsigned char	1	0 to 255

. void data type (Empty Data set)

→ Void data types mean no value. This data type is normally associated with a function that return no value.

Syntax: `Data-type variable-name;`

eg: `int a;`
`float x,y,z;`

User defined Data type :

① Type Definition :

→ Type definition allows user to define an identifier that would represent a data type using an existing data type.

→ typedef is the keyword.

Syntax : `typedef datatype new_datatype ;`

eg : typedef float rupee ;

Enumerated Data type :

→ It is used to assign names to integral constants to make a program easy to read and maintain.

→ enum is the keyword.

Syntax : `enum identifier [optional {enumerator-list}] ;`

For example :

⇒ Enumerated list starts with 0 by default and value is incremented by 1 for the sequential identifiers in the list.

Enumerated list Constant	Value
Sun day	0
Mon day	1
Tues day	2
Wednes day	3
Thurs day	4
Fri day	5
Saturday	6

⇒ We can also explicitly assign value to enumerated list. Consider the declaration

enum week_day {Sun = 10, Mon, Tue = 15, Wed, Thurs, Fri = 20, Sat};

Enumerated List Constant	Value
Sunday	10
Monday	11
Tues day	15
Wednes day	16
Thurs day	17
Friday	20
Saturday	21

Derived Data type :

→ Data types which are derived from fundamental data types are called derived data types.

① Array : A finite collection of data of same types or homogeneous data type.

② Pointer : Pointer is a variable that stores or points the address of another variable.

③ Structure : A collection of related variables of the same or different data types.

④ Union : allows storing different data types in the same memory location.

Expressions Using Operators in C :

→ An expression is a collection of operators and operands that represent a specific value.

→ Operator is a symbol which performs tasks like arithmetic operations, logical operations, relational operations and conditional operations etc.

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

→ Operators, functions, constants and variables are combined together to form expressions.

Evaluation of Expressions:

→ Expressions are evaluated using an assignment statement of the form,

```
Variable = Expression;
```

Type	Operator	Meaning	For Example.
Arithmetic	+	Addition or unary plus	c = a+b
	-	Subtraction or unary minus	d = -a, c = a-b
	*	Multiplication	c = a*b
	/	Division	c = a/b
	%	Mod	a%b
Relational	<	Less than	a<4
	>	Greater than	a>4
	<=	Less than equal to	a<=4
	>=	Greater than equal to	a>=4
	==	Equal to	a==4
	!=	Not Equal to	a!=4
Logical	&&	And	0 && 1
	||	Or	0 || 1
Assignment	=	Is assigned to	a=5
Increment	++	Increment by one	++i or i++
Decrement	--	Decrement by one	--x or x--

Managing Input and Output Operations:

→ Each program that uses standard input/output function must contain the statement #include <stdio.h> at the beginning.

→ Maximum input and output functions are defined in the header file "stdio.h"

→ stdio.h (Standard Input output header file) includes function like getc(), getchar(), gets(), printf(), put(), putchar(), puts(), scanf() etc.

Types of Input/output functions:

Formatted I/O Functions	Unformatted I/o Functions
Input Functions	Input Functions
scanf() fscanf()	getc() getch() getchare() getche() gets()
Output Functions	Output Functions
printf() fprintf()	putc() putch() putchar() puts()

Unformatted Input Functions:

① getc()

→ used to accept the character from the file and returns the corresponding integer value.

Syntax:

```
Variable_name = getc (FILE *stream);
```

② getch()

→ The character entered through getch() is not displayed in the screen. The character is invisible on the screen. It is present in conio.h header file.

Syntax : `Variable_name = getch();`

③ getchar()

→ used to accept a character in a C program.

Syntax : `Variable_name = getchar();`

④ getche()

→ accepts only single character, but unlike getch(), getche() displays the entered character in the screen.

Syntax : `Variable_name = getche();`

⑤ gets()

→ accepts any line of string including spaces from the standard input device (keyboard).

Syntax : `gets (variable_name);`

Unformatted output Functions :

① putc()

→ putc() sends a character to the given file instead of the standard output device.

Syntax : `putc (variable_name, file);`

② putch()

→ putch() displays any alphanumeric characters

to the standard output device. It displays only one character at a time. It is present in conio.h header file

Syntax: `variable_name = putch();`

③ putchar()

→ putchar() writes a single character, one at a time to the standard output device.

Syntax: `putchar (variable_name);`

④ puts()

→ puts() is used to write a string to output device.

Syntax: `puts (variable_name);`

Formatted Input Functions:

① scanf()

→ used to read information the standard input device.

Syntax: `scanf("Format string", list of addresses of variables);`

eg
```
int a;
float b;
```

② fscanf()

→ used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

Syntax: `fscanf (FILE *stream, const char * format (argument);`

eg `fscanf(fp, "%d", &age);`

Formatted Output Functions:

① printf()

→ The output data or result of an operation can be displayed from the computer to a standard output device such as monitor.

Syntax: `printf("format string", list of variables);`

② fprintf()

→ Is used to write a set of characters into file. It sends formatted output to a stream.

Syntax: `fprintf(FILE *stream, const char *format[, argument, ...]`

eg: `fprintf(fp, "%s %d", var1, var2);`

Control Statements:

→ Control statements are program statements that cause a jump of control from one part of program to another part of program.

Control Statements:

(A) Conditional Statements

↳ Decision Making and Branching Statements

① Simple if Statement
② if else Statement
③ Nested if..else Statement
④ Else if Statement
⑤ Switch-case Statement
⑥ Conditional operator.

↳ Looping Statements

① While Loop

② Do... While Loop

③ For Loop

(ii) Unconditional Statements

① Goto statement

② Break Statement

③ Continue Statement

Decision Making and Branching Statements:

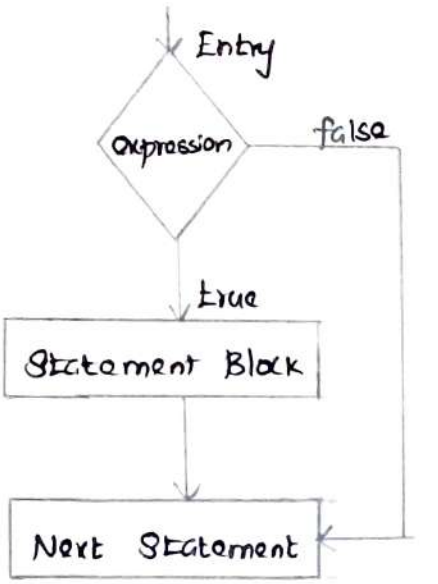
→ Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met.

Simple if Statement:

→ It is the simple selection or branching statement

Syntax:

```
if (condition)
{
  Statement;
}
```



Example:

```
Void main() {
  int a,b;
  printf("Enter the value of a & b");
  scanf("%d%d", &a, &b);
  if(a>b) {
```

```
    printf(" a is greater");
  }

  getch();
}
```

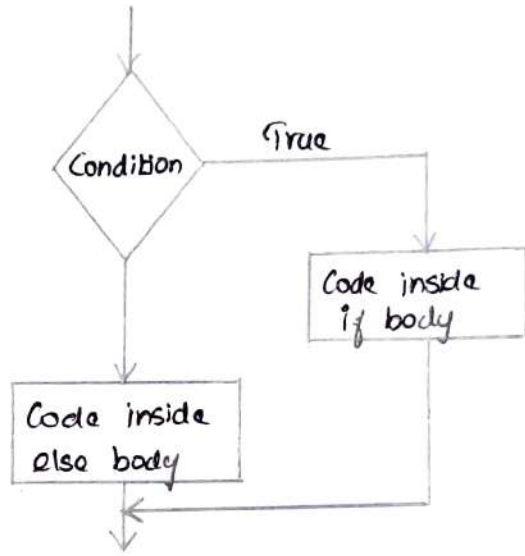
Output:

```
Enter the value of a & b : 5
                           3
a is greater
```

if else Statement :

Syntax :

```
if (condition)
{
  statement ;
}
else
  {
    statement ;
  }
```



Example :

```
void main()
{
 int a,b;
 printf(" Enter the values of a & b");
 scanf ("%d %d", &a, &b);
 if(a>b)
  {
   printf(" a is greater");
  }
 else
  {
   printf (" b is greater");
  }
 getch();
}
```

Output :

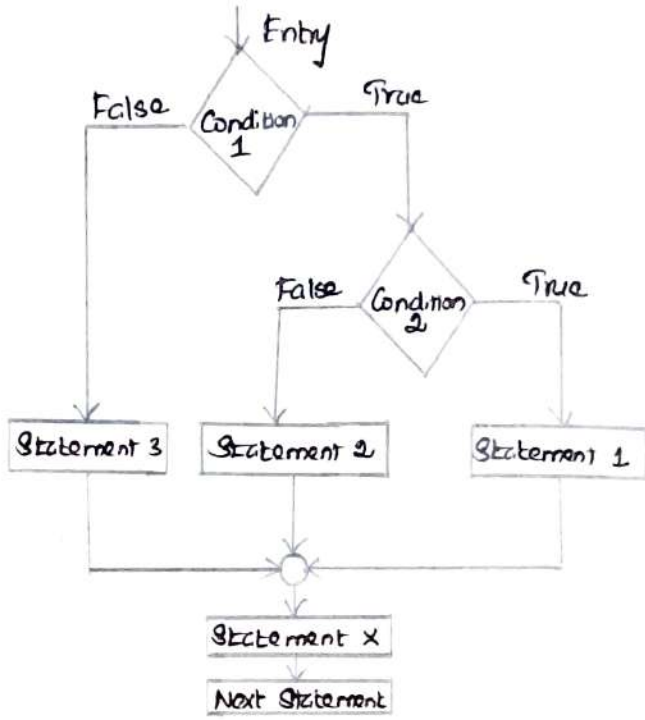
```
Enter the values of a & b : 5
                            7
b is greater
```

Nested if else Statement :

→ An entire if-else construct can be placed within either the body of if statement or the body of an else statement.

Syntax :

```
if (condition 1)
 {
  Statement;
  if (condition 2)
   {
    statement;
   }
 }
 else
   {
    Statement;
   }
```



Example :

```
Void main() {
    int a,b;
    printf (" Enter the values of a &b");
    scanf ("%d%d", &a, &b);
    if(a>b)
     { printf("%d is larger \n", a);
     }
    if( b> a)
      {
       printf (" %d is larger \n" ,b);
      }
     else
      {
       printf (" both are equal\n");
      }
    getch();
    }
```

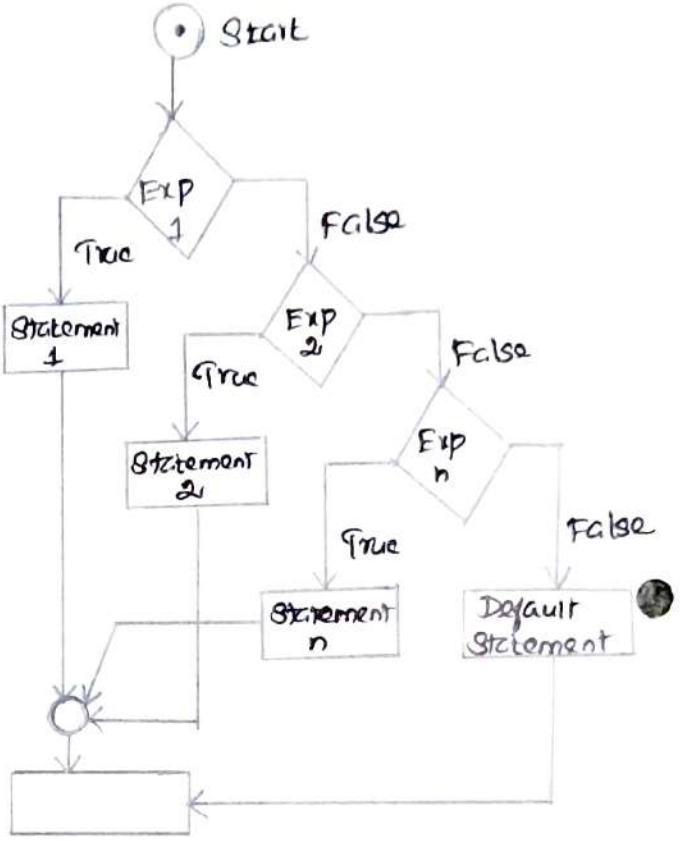
Output :

```
Enter the values of a & b : 5
                            15

15 is larger.
```

Else if Statement

Syntax:

```
if (expression 1)
 { statement ;
 }
 else if (  prossion 2)
  { statement ;
 }
 else if ( expression 3)
 {
  statement ;
 }
 else
    {
      statement ;
    }
```



Example :

```
Void main()
 {
 int a, b, c, d;
 printf (" Enter Four values");
 scanf ("%d%d%d%d", &a, &b, &c, &d);
 if ( (a>b) && (a>c) && (a>d))
  {printf (" %d is greater", a);
  }
  else if ((b>a) && (b>c) && (b>d))
   {printf (" %d is greater ", b);
   }
   else if ( (c>a) && (c>b) && (c<d))
   {printf ("%d is greater", c);
   }
   else {
     printf (" %d is greater", d);
   }}
```

Output:

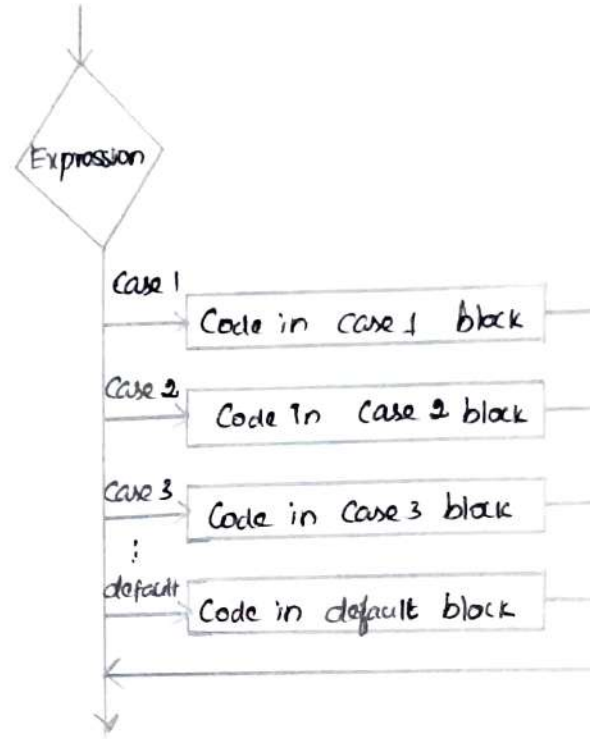
```
Enter four values : 7
                    4
                    11
                    6

11 is greater
```

Switch-case Statement:

→ Switch statement is a multi-way control statement that allows us to make a decision from the number of choices.

Syntax:

```
Switch (condition)
{
  Case 1:
     Statement;
     break
  Case 2:
     Statement;
     break;
  Case n:
     Statement;
     break
  default:
     Statements;
}
```



Example:

```
Void main(){
  int day;
  printf(" Week days");
  printf(" Enter value of day 1-...7:");
  scanf("%d", &day);
  Switch (day)
  {
    case 1:
       printf(" Sunday");
       break;
    case 2:
       printf(" Monday");
       break;
    Case 3:
       printf(" Tuesday");
       break;
    Case 4:
       printf("Wednesday");
       break;
    case 5:
       printf(" Thursday");
       break;
    case 6:
       printf(" Friday");
       break;
    case 7:
       printf("Saturday");
       break;
    default:
       printf(" Invalid Input");
       break;
  }
}
```

Output:

```
Week days
Enter value of day 1...7:5

Thursday
```

Conditional Operator:

The conditional operator ? : is otherwise known as Ternary operator. It is similar to if else statement.

Syntax:

```
expression 1 ? expression 2 : expression 3 ;
```

Example:

```
Void main()
{
int n;
printf("Enter a number:");
scanf("%d",&n);
n % 2 ==0 ? printf("Even number") : printf("odd number");
}
```

Output:

Enter a number : 15

Odd number

Looping Statements:

→ Looping statements are used for repeating some portion of the program either a specified number of times or until a particular condition is being satisfied.

→ three types

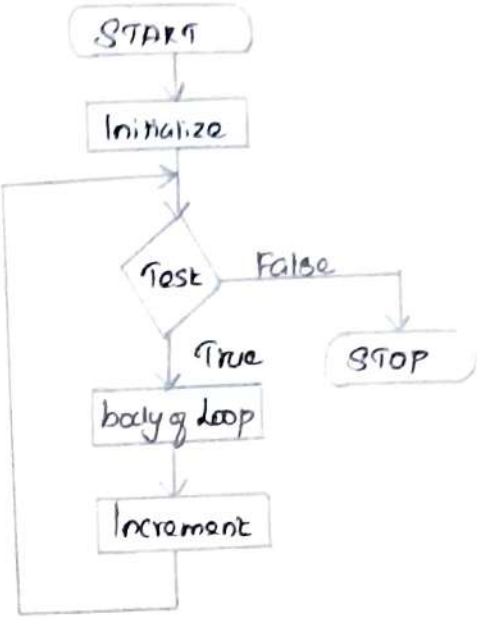
(i) While Loop
(ii) Do While Loop
(iii) For Loop.

While Loop:

→ It is an entry controlled looping statement. Because the condition will be evaluated first and if the condition is true then only the statement block will be executed.

Syntax:

```
While (condition)
{
  Statement;
}
```



Example:

```
Void main() {
    Int n, sum, i;
    printf("\n Enter the value of n:");
    Scanf ("%d", &n);
    sum = 0;
    i = 1;
    while (i <= n)
    {
      sum = sum + i;
      i++;
    }
    printf("\n sum is %d", sum);
    }
```

Output:

Enter the value of n: 5

Sum is 15

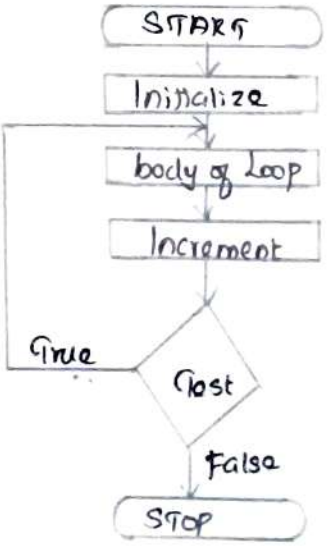
Do While Loop:

→ It is an exit-controlled looping statement.

→ The condition will be verified only at the end of the looping construct.

Syntax:

```
do
{
  Statement;

}While (Condition);
```



Example:

```
Void main(){
    Int n, sum, i;
    printf("Enter Value for n:");
    Scanf("%d", &n);
    sum = 0;
    i = 1;
    do {
        sum = sum + i;
        i++;
    } while (i <= n);
    printf("\n sum is %d", sum);
}
```

Output:

Enter value for n: 5

The sum is 15

For Loop:

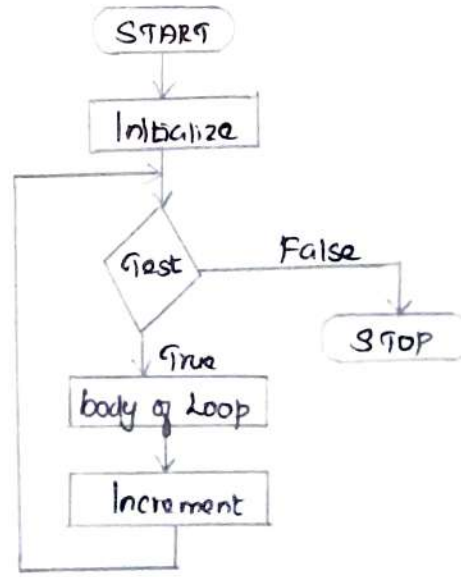
→ is a loop in which the repeated execution of statements occurs. It is an entry controlled looping statement.

Syntax:

```
for(initialization; Condition; increment/decrement)
{
  Statement;
}
```

Example:

```
Void main() {
    Int n, sum, i;
    printf("Enter Value for n:");
    Scanf("%d", &n);

    Sum = 0;
    for (i = 1; i <= n; i++)
    {
```



```
    sum = sum + i;
    }
  printf("\n Sum is %d", sum);
  }
```

Output :

Enter value for n : 5

Sum is 15

Unconditional Control Statement :

① Go to Statement :

→ It provides an unconditional jump from the goto to a labelled statement in the same function.

Syntax :

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

→ The label is an identifier. When goto statement is encountered control of the program jumps to label : and starts executing the code.

Example :

```
Void main()
 {
  int n;
  printf("\n Enter number :");
  scanf("%d", &n);
  if ( n%2 == 0)
     goto even;
  else
     goto odd;
even : printf("\n %d is Even number", n);
     return;
odd : printf("\n %d is odd number", n);
     return; }
```

Output :

Enter number : 15

15 is odd number

② Break Statement:

⇒ When the break statement is encountered inside the loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

⇒ It can be used to terminate a case in switch statement

Syntax:

```
break;
```

Example:

```
Void main() {
 int num,i,n;
 Printf("How many values to Enter:");
 Scanf("%d", &n);
 for(i=1;i<=n;i++)
  {
   printf("\n Enter value %d :",i);
   scanf("%d", &num);
   if(num<0)
    {
     printf("User Entered a Negative value. Loop has broken");
     break;
    }
   printf("\n Value is %d", num);
  }
}
```

Output:

```
How many values to Enter: 5
Enter value 1 : 10
    Value is 10
Enter value 2 : 20
    Value is 20
Enter value 3: -5
User Entered a negative value
                Loop has broken
```

③ Continue Statement:

→ Is used for continuing next iteration of loop after skipping some statement of loop.

Syntax:

```
Continue;
```

Example :

```
Void main(){
  Int i, n;
printf("\n Enter Value of n:");
Scanf("%d", &n);
printf("\n Even numbers Found between 1 and %d:");
for(i=1; i<=n; i++)
  {
    if(i%2 != 0)
      continue;
      printf("%d \t", i);
  }
}
```

Output :

Enter value of n: 5

Even numbers found between 1 and 5

2 4

Functions :

→ Functions are the building blocks of C. It performs the same set of instructions on different sets of data or at different portions of a program.

→ Functions are also known as modules, procedures, sub programs or subroutines.

Two kinds of C Functions :

① Library Functions → printf(), scanf(), floor(), exp() etc

② User defined Functions → main() function.

Components of C Function :

→ Four Components

(i) Function Declaration or Prototype

(ii) Function Call

(iii) Function Definition

(iv) Return Statement.

Function Declaration :

→ Function declaration is optional. It is needed only when we want to define the called function after the calling function.

Syntax :

```
return_type function_name ( type argument1, argument2, .. argument N);
                    (or)
return_type function_name ( type, type, ... type);
```

Example :

```
Void fndisplay(int, double, char);
                (or)
Void fndisplay(int x, double y, char z);
```

Function call :

→ It is used to call the actual function.

Syntax :

```
Return_Variable = function_name (Actual Arguments);
```

Example :

```
c = fnSum(a, b);
```

Function Calling Methods :

→ There are two ways that a C function can be called

(i) Call by value

(ii) Call by Reference

Call by value :

→ When a function is called then the function's actual values of the parameters are passed.

→ The parameters are simple variables.

→ A copy of the variable is passed to the function.

Example :

```
#include <stdio.h>
void swap( int n1, int n2)
{
  int temp =n1;
  n1 =n2;
  n2 = temp;
}
int main() {
    int x = 20;
    int y = 68 ;
    printf(" The numbers before swapping n1 and n2: %d %d \n",
                                                    x, y);
    swap(x,y);
    printf("The numbers after swapping n1 and n2: %d %d \n",
                                                    x,y);
    return 0;
}
```

Output :

```
The numbers before swapping n1 and n2 : 20
                                        68
The numbers after swapping n1 and n2 : 20
                                        68
```

Call by Reference :

→ When a function is called then the function's address of the parameters are passed

→ The parameters are pointer variables.

→ An address of the variables is passed to the function.

Example :

```
#include <stdio.h>
void swap( int *n1 , int *n2)
{
   int temp = *n1;
   *n1 = *n2;
   *n2 = temp;
}
int main() {
   int n1 = 20;
   int n2 = 68;
   printf("The numbers before swapping n1 and n2 : %d %d \n", n1,n2);
   swap (&n1, &n2);
   printf("The numbers after swapping n1 and n2 : %d %d \n",n1,n2);
   return 0;
}
```

Output :

```
The numbers before swapping n1 and n2 : 20
                                        68
The numbers after swapping n1 and n2 : 68
                                       20
```

Function Definition :

→ Function definition includes all the statements to be executed. It contains the block of code to perform a specific task.

Syntax :

```
return_type Function_name ( parameter list)
{
   Local variable declarations;
   statements;
   return( expression);
}
```

Example :

```
int fnSum(int x, int y) {
    int z;
    z = x+y;
    return z; }
```

Return Statement :

→ return statement terminates the execution of a function and returns a value to the calling function.

→ The program control is transferred to the calling function after return statement.

Syntax :

```
return (expression);
    (or)
return;
```

return(expression);

↳ returns both data and control from called function to calling function.

return;

↳ returns only control from the called function to calling function and parameter list is optional.

Example :

```
return (a+b);
return(10);
```

Types of Functions :

→ A function in C can be called either with arguments or without arguments. These function may or may not return values to the calling functions.

→ Five types

① Function with no arguments and no return value

② Functions with arguments and no return value.

③ Functions with Arguments and Return value.

④ Functions with No arguments but with return value

⑤ Functions return multiple values.

Functions With No Arguments and No Return values.

Syntax.

```
Function declaration:
    Void function-name(void);
Function call:
    function_name();
Function definition:
    Void function_name()
     {
      Statements;
     }
```

Example:

```
#include<stdio.h>
 // Function declaration
Void fnSum();
// Function definition
 Void main()
 {
  fnsum();
 }
 Void fnsum()
  {int x,y,z;
   printf("\n Enter value for x and y :");
   scanf("%d%d",&x,&y);
   z = x+y;
   printf("\n sum of %d and %d is : %d",x,y,z);
  }
```

Output:

```
Enter value for x and y : 10
                          20

Sum of 10 and 20 is : 30
```

Functions with Arguments But no Return values:

Syntax:

```
Function declaration:
    Void function-name (argument);
Function call:
    function-name (argument);
Function definition:
    Void function-name (Formal Parameters)
    {
      Statements;
    }
```

Example:

```
#include<stdio.h>
// Function declaration
 void fnsum(int, int);
// Function definition
 Void main()
 {
  int a,b;
  printf("\n Enter Value for a and b:");
  scanf("%d%d", &a, &b);
  fnsum(a,b);
 }
 Void fnsum(int x, int y)
 {
  int z;
  z = x+y;
  printf("\n Sum of %d and %d is : %d", x,y,z);
 }
```

Output:

```
Enter value for a and b: 10
                         20

Sum of 10 and 20 is : 30
```

Functions with Arguments and Return values:

Syntax

```
Function declaration:
    return-type function-name (argument);
Function call:
    Var = function-name (argument);
Function definition:
    return-type function-name (Formal Parameters)
    {
        Statements;
    }
```

Example:

```
#include <stdio.h>
#include <conio.h>
// Function declaration
 int fnsum (int, int);
// Function definition
 void main()
 {
 int a,b,c;
 printf("\n Enter value for a and b");
 scanf("%d %d", &a, &b);
 c = fnsum (a, b);
 printf("\n Sum of %d and %d is : %d", a,b,c);
 getch();
 }
 int fnsum (int x, int y)
 {
   int z;
   z = x+y;
   return z;
 }
```

Output:

```
Enter value for a and b : 10
                          20
Sum of 10 and 20 is : 30
```

Functions with NO Arguments but with Return value :

Syntax :

```
Function declaration :
    return_type function_name();
Function call :
    var = function_name();
Function definition :
    return_type function_name()
    {
      Statements;
    }
```

Example :

```
#include <stdio.h>
#include <conio.h>
// Global variable declaration
 int a,b,c;
// Function declaration
  int fnsum();
// Function Definition
   void main() {
      printf("\n Enter value for A and B:");
      scanf("%d %d", &a, &b);
      c = fnsume);
      printf("\n Sum of %d and %d is : %d", a,b,c);
      getch();
   }
  int fnsume)
   {
     c=a+b;
     return c;
   }
```

Output :

```
Enter value for A and B : 10
                          20

Sum of 10 and 20 is : 30
```

Functions Return Multiple values:

Syntax:

```
Function declaration:
    return_type function_name();
Function call:
    var = function_name();
Function definition:
    return_type function_name()
    {
      Statements;
      if(condition 1)
          return value 1;
      else
          return value 2;
    }
```

Example:

```
#include<stdio.h>
#include<conio.h>
// Function Declaration
 int fnLarge(int,int);

// Function Definition
  void main(){
    int a,b,large;
    printf(" Enter the value of a and b:");
    scanf("%d%d",&a,&b);
    c = fnLarge(a,b);
    printf("\n The Largest number is:%d",large);
    getch();
  }
```

```
int fnlarge (int x, int y)
{
    if (x>y)
        return x;
    else
        return y;
}
```

Output:

```
Enter the value of a and b: 5
                            10
The Largest number is : 10
```

Recursive Function:

⇒ The process of calling a function by itself is called recursion and the function which calls itself is called recursive functio

→ Recursion is used to solve various mathematical problems by dividing the given problem into smaller sub problems. This method of solving a problem is called divide and conquer.

Syntax:

```
returntype functionName ( [argument list])
{
    Statements;
    ...
    functionName ([actual argument]);
    ...
}
```

Example:

```
#include <stdio.h>
#include <conio.h>
// Recursive Function
int fact(int n)
{
    int f=1,i;
```

```
    for(i=1 ; i<=n; i++)
        f = fnFact(n-1)*n;
    return(f);
    }
// Function definition
void main()
{
 int n, factorial;
 printf("\n Enter a number:");
 scanf("%d ", &n);
 factorial = fnFact(n);
 printf("\n Factorial of %d is : %d", n, Factorial);
 getche();
}
```

Output:

Enter a number : 5

Factorial of 5 is : 120

Arrays:

→ An array can be defined as a group of related data items that share a common name.

→ An array can also be defined as an ordered sequence of finite data items of the same data type that share a common name.

Syntax:

```
datatype  arrayname [size] ;
```

Example:

```
int a[10];
double b[10];
```

Types of Array:

(i) One Dimensional Array

(ii) Two Dimensional Array

(iii) Multi Dimensional Array

One Dimensional Array / single Dimensional Array:

→ To initialize an array during the declaration only.

→ The list of values enclosed in braces { }, separated by commas, provides the initial values for successive elements of the array.

→ If there are fewer initializers than elements in the array the remaining elements are automatically initialized to 0.

For example

```
int a[10] = {0, 1, 2, 3, 4, 5, 6}
```

↳ can initialize a[7], a[8], a[9] to 0

→ Suppose A is an array of 5 elements, we can reference each element as,

A[0] → First element
A[1] → Second Element
A[2] → Third Element
A[3] → Fourth Element
A[4] → Fifth element.

A[4]	500
A[3]	400
A[2]	300
A[1]	200
A[0]	100

Syntax:

```
datatype arrayname[size] = {value1, value2, value3 ... };
```

Example:

```
int a[5] = {20, 60, 90, 100, 120};
```

Example :

```
void main()
{
 int a[10], i, size, b[5] = {-5, 10, -15};
 printf(" Enter the size of an array : \t");
 scanf(" %d", &size);
 printf(" \n Enter %d elements of an array :", size);
 for(i=0 ; i< size ; i++)
  {
   scanf ("%d", &a[i]);
   printf(" \n The %d elements of array A are :\n", size);
  }
 for(i=0 ; i<5 ; i++)
  {
   printf(" \t %d ", a[i]);
   printf(" \n The 3 elements initialized to array B are : \n");
   for(i=0 ; i<5 ; i++)
    printf("%d \t", b[i]);
  }
}
```

Output :

Enter the size of array: 4

Enter 4 elements of an array : 1
2
3
4

The 4 elements of Array A are :

1 2 3 4

The 3 elements initialized to array B are :

-5 10 -15 0 0

Two Dimensional Array:

→ An array of arrays is known as Two dimensional array. Two dimensional array is a type of array which has finite number of rows and finite number of columns.

→ The two dimensional array is also known as matrix. A matrix can be represented as a table of rows and columns.

Syntax:

```
datatype  arrayname [row size] [column size] ;
```

Example:

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

```
int a[2][3] = { {1,2} ,{3,4,5}}; (Or)
int a[2][3] = {1,2,3,4,5};
```

Example:

```
void main()
{
  int a[10][10], i,j, row, col;
  printf("Row and Column of an array:\t");
  scanf("%d %d", &row, &col);
  printf("Enter %d elements of an array:\t", (row*col));
  for(i=0; i<row; i++)
      for(j=0; j<col; j++)
          scanf("%d", &a[i][j]);
  printf("In The %d elements of array A are:\n", (row*col));
```

```
    for(i=0 ;i<row; i++)
    {
      for j=0 ; j<col ; j++)
            printf("\t%d", a[i][j]);
      printf("\n");
    }
}
```

Output :

```
Row and Column of an array : 2  2
Enter 4 elements of an array : 1  2  3  4
The 4 elements of array A are :
        1   2
        3   4
```

Multi Dimensional Array :

→ A multi dimensional array is an array containing one or more arrays.

Syntax :

```
datatype arrayname [size1][size2][size 3] .... [size n];
```

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

Example :

```
Void main()
{
  int i, regno, mark[5], total=0 ;
  char name[20];
```

```
printf(" Enter the Register Number :");
scanf("%d", &regno);
printf(" Enter the Name :");
scanf("%s", &name);
for(i=1; i<=5 ; i++)
   {
    printf("\n Enter the Mark %d:", i);
    scanf("%d", &mark[i]);
    total = mark[i];
   }
  printf("\n\t Student Mark List");
  printf("\n\t ***********");
  printf("\n Student Number :%d", regno);
  printf("\n  Student Name : %s ", name);
  for(i=1;i<=5;i++)
       printf("\n Mark %d : %d ", i, mark[i]);
  printf("\n Total : %d", total);
}
```

Output :

```
Enter the Register Number : 1001
Enter the Name : Jwin
Enter the Mark 1 : 96
Enter the Mark 2 : 95
Enter the Mark 3 : 99
Enter the Mark 4 : 100
Enter the Mark 5 : 89
```

Student Mark List

* * * * * * * * * * * *

Student Number :	1001
Student Name :	Jwin
Mark 1 :	96
Mark 2 :	95
Mark 3 :	99
Mark 4 :	100
Mark 5 :	89
Total :	479

Advantages of using Arrays:

(i) Reduces memory access time, because all the elements are stored sequentially

(ii) Reduces the number of variables in a program.

(iii) Easy to use for the programmers.

Disadvantages of using arrays:

(i) Wastage of memory space is possible.

(ii) Storing heterogeneous elements are not possible.

(iii) Array bound checking is not available in C. So manually we have to do that.

UNIT - II

C PROGRAMMING - ADVANCED FEATURES

Structures - Union - Enumerated Data types - Pointers: Pointers to Variables, Arrays and Functions - File handling - Preprocessor Directives.

Structures:

→ A structure is a collection of variables of different data types under a single name.

→ The variables are called members of the structure.

→ The structure is declared with the keyword struct followed by an identifier (tag name) followed by a brace enclosed list of member declarations. The members of the structure are public in nature.

Syntax:

```
struct tagname
{
  member 1;
  member 2;
    :
  member n;
};
```

Example

```
struct student
{
  int mark;
  char name[5];
  float average;
};
```

(or)

```
struct student
{
  int mark;
  char name[5];
  float average;
} s1, s2, s3;
```

Example :

```
#include<stdio.h>
#include<conio.h>
Struct emp
{
 int num;
 Char name[10];
 float sal;
 };
Void main()
 {
  Struct emp e;
  printf("\n Enter Employee Number :\t");
  scanf("%d", &e.num);
  printf("\n Enter Employee Name : \t");
  scanf("%d", & e.name);
  printf("\n Enter Employee Salary : \t");
  scanf("%d", & e.sal);
  printf("\n \t Number : %d\n \t Name : %s \n \t Salary : %f ",
                                  e.num, e.name, e.sal);
  getch();
 }
```

Output :

```
Enter Employee Number : 1001
Enter Employee Name : Jwin
Enter Employee Salary : 45700.00
Number : 1001
Name : Jwin
Salary : 45700.000000
```

Structure Initialization:

Syntax :

```
Struct structure_name structure_variable = {value1, value2, .... ValueN};
```

Example :

```
Struct emp e = {10, "Jwin", 45000.00};
```

↳ It will assign to to e.num, "Jwin" to e.name and 45000.00 to e.sal

Array of structures:

→ While declaring the structure variable, we can declare array of structures as,

① struct emp
```
{
int num;
char name[10];
float sal;
}
struct emp e[10];
```
⇒ 10 employee records will be created

② struct emp
```
{
int num;
char name[10];
float sal;
} e[10];
```
⇒ 10 employee records will be created.

Example:
```
#include <stdio.h>
#include <conio.h>
struct student
{
int num;
char name[10];
int mark[5];
int tot;
} s[10];
void main()
{
int i,j,n;
```

```
printf("\n How many Students Records You want to Create:");
scanf("%d", &n);
for(i=0;i<n;i++)
 {
  printf("\n Enter Student Number, Student Name, Five Marks:\n");
  scanf("%d%s%d", &s[i].num, &s[i].name);
  s[i].tot=0;
  for(j=0;j<5;j++)
     {
      scanf("%d", &s[i].mark[j]);
      s[i].tot += s[i].mark[j];
     }
 }
 for(i=0;i<n;i++)
    printf("\n\t Student Number: %d\n\t Student Name: %s
          \n\t Total Marks : %d", s[i].num, s[i].name,
                                   s[i].tot);
   getch();
}
```

Output :

How many Students Records You want to Create : 2

Enter Student Number, Student Name, Five marks :

1001 Rishan 90 100 89 94 99

Enter Student Number, Student Name, Five marks :

1002 Rigan 99 95 100 86 99

Student Number : 1001
Student Name : Rishan
Total Marks : 492
Student Number : 1002
Student Name : Rigan
Total Marks : 459

Nested structures / structure within a structure:

→ Structure within another structure is known as Nested structure.

Example:

```
#Include <stdio.h>
#include <string.h>
struct date1
  {
   int date, month, year;
  };
struct student
  {
   int num;
   char name[20];
   struct date1 dob;
   } stud;
int main()
  {
  struct student stud = {100," Jwin", 13.08.2016};
  printf("\n Student Number is : %d ", stud.num);
  printf("\n Name is : %s ", stud.name);
  printf("\n Date of birth is : %d %d %d ", stud.dob.date,
                                  stud.dob.month, stud.dob.year);
  return 0;
}
```

Output:

Student Number is : 100
Name is : Jwin
Date of birth is : 13.08.2016

Advantages of Structures:

(i) Structures can store heterogeneous values
(ii) Structure is user defined datatype, with default access specifier public.
(iii) Structure can be used for dynamic memory allocation.

(ii) A struct allocates memory for each member, such that all members can be used at once.

(iii) All the members can be initialized.

Disadvantages of structures:

(i) Struct defaults member to public

(ii) Structure provides single way to view each memory location.

(iii) Anonymous structure is not possible.

Union:

→ A union is a collection of variables of different types. Union and structure in C are same in concepts except allocating memory for their members.

→ Structures allocates storage space for all its members separately.

→ Whereas union allocates one common storage space for all its members. Hence we can access only one member of union at a time. We can't access all member values at the same time in union.

→ The size of a union is equal to the size of its largest data or element.

Syntax:

```
union tagname
{
    datatype  var_name1;
    datatype  var_name2;
    datatype  var_name3;
};
```

Example:

```
#include <stdio.h>
#include <conio.h>
union emp
 {
  int num;
  char name[5];
  float sal;
 };
void main()
 {
   union emp e;
   printf("\n "Enter Employee Number : \t");
   scanf("%d", & e.num);
   printf("\n\t Number :%d ", e.num);
   printf("\n\t Size of Number in Union is :%d", sizeof(e.num));
   printf("\n Enter Employee Name : \t");
   scanf("%s ", e.name);
   printf("\n\t Name : %s", e.name);
   printf("\n\t Size of Name in Union is : %s", sizeof(e.name));
   printf("\n Enter Employee Salary : \t");
   scanf("%f ", &e.sal);
   printf("\n\t Size of Salary in Union is :%d", sizeof(e.sal));
   printf("\n\t Size of Employee union is :%d ", sizeof(e));
   getch();
 }
```

Output:

```
Enter Employee Number : 1001
Number : 1001
Size of Number in union is : 2
Enter Employee Name : Jwin
Name : Jwin
Size of Name in union is : 5
Enter Employee Salary : 45000.00
Salary : 45000.000000
Size of Salary in Union is : 4
Size of Employee union is : 5
```

Advantages of Union:

(i) Anonymous union can be declared.

(ii) Union provides multiple way to view same memory location

(iii) It saves memory location.

Disadvantages of Union:

(i) Since all members of the union share the same memory location, it is not possible to access more than one member at a time.

(ii) Only the first member of an union can be initialized.

Enumerated Data types:

→ Enumerated data types are a special way of creating your own type in C.

→ Enumerated types are used to make a program clearer to the reader/maintainer of the program.

→ Enumeration is a user defined data type in C language. It is used to assign names to the integral constants which makes a program easy to read and maintain.

→ The keyword "enum" is used to declare an enumeration.

Syntax:

```
enum enum_name {Const 1, Const 2, ....};
```

Example:

```
enum week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
enum week day;
```

Example :

```
#include <stdio.h>
enum week { Mon = 10, Tue, Wed, Thur, Fri = 10, Sat = 16, Sun};
enum day { Mond, Tues, Wedn, Thurs, Frid = 18, Satu = 11, Sund};
int main()
{
  printf(" The value of enum week :
        %d\t %d\t %d\t %d\t %d\t %d\t %d\t \n",
        Mon, Tue, Wed, Thur, Fri, Sat, Sun);

  printf(" The default value of enum day :
        %d\t %d\t %d\t %d\t %d\t %d\t %d\t",
        Mond, Tues, Wedn, Thurs, Frid, Satu, Sund);

  return 0;
}
```

Output :

The value of enum week : 10 11 12 13 10 16 17

The default value of enum day : 0 1 2 3 18 11 12

Pointers :

→ Pointer is a data object that refers to a memory location, which is an address.

→ It is a variable that stores / points the address of another variable;

→ A pointer in C is used to allocate memory dynamically at run time.

→ The data type of pointer may be : int, float, char, double, short etc.

Pointer Variable Declaration:

→ before using the pointer variable, it should be declared as given below

Syntax:

```
datatype *ptr_var_name;
```

Example:

```
int *nptr;   → Pointer to an Integer
char *cptr;  → Pointer to a Character
double *dptr; → Pointer to a Double
```

Pointer Variable Initialization:

→ It is the process of assigning address of a variable to a pointer variable.

→ Pointer variable can only contain address of a variable of the same data type.

→ Address-of operator (&) is used to determine the address of a variable. The & immediately preceding a variable name returns the address of the variable associated with it.

Syntax:

```
ptr_var_name = &var_name;
```

Example:

① `int num, *pnum;` → Pointer Declaration
`pnum = &num;` → Pointer Initialization

② `int num;`
`int *pnum = &num;` → Pointer Declaration with Initialization

Pointer Variable Indirection:

→ The Indirection or dereference operator * gives the "contents of an object pointed to by a pointer".

→ The unary operator * is used to access the value stored in the address pointed by pointer.

Syntax:

```
*ptr_var_name;
```

Example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
 int num=10, *pnum;

 pnum = &num;
 printf("\n It Value of Integer Variable Number is :%d", num);
 printf("\n It Address of Integer Variable Number is :%u", &num);
 printf("\n It Value of Pointer variable PNum is :%u", pnum);
 printf("\n It Content of Pointer Variable PNum is :%d", *pnum);
 printf("\n It Address of Pointer variable PNum is :%d", &pnum);

 *pnum = 100; // value has been changed to both num and pnum
 printf("\n After Modification:");
 printf("\n It Value of Integer Variable Number is :%d", num);
 printf("\n It Content of Pointer Variable PNum is :%d", *pnum);

 getch();
}
```

Output :

```
Value of Integer Variable Number is : 10
Address of Integer variable Number is : 1638212
Value of Pointer variable PNum is : 1638212
Content of Pointer variable PNum is : 10
Address of Pointer variable PNum is : 1638208
After Modification :
    Value of Integer Variable Number is : 100
    Content of Pointer variable PNum is : 100
```

Advantages of Pointer :

(i) Pointers are used to allocate memory to structure variables during run time.

(ii) Using pointer it is efficient to pass arrays and strings to Functions

(iii) It makes possible to return more than one value from function.

(iv) Pointers can be used to access a variable that is defined outside the function.

(v) Addresses of objects can be extracted using pointers.

(vi) Provides an alternate way to access array elements.

Disadvantages of Pointer :

(i) Uninitialized pointers might cause segmentation fault.

(ii) Dynamically allocated block needs to be freed explicitly

(iii) Pointer issues are hard to debug.

Pointer to an array:

→ Pointers to an array points the address of memory block of an array variable.

Syntax:

```
datatype * variable_name [size];
```

Example:

```
#include <stdio.h>
int main()
{
 int *arr[3];
 int *a;
 printf("Value of array Pointer variable : %d ", arr);
 printf(" Value of pointer variable : %d ", &a);
 return 0;
}
```

Output:

```
Value of array pointer variable : 1481173888
Value of pointer variable : 1481173880
```

Function Pointer:

→ A pointer to a function is known as Function Pointer in C

Syntax:

```
returntype (*function_pointer) (function argument list);
```

Example:

⇒ void (*ptrfn1)(int);

⇒ double (*ptrfn2) (double, char)

Example:

```
#include <stdio.h>
int sum(int a, int b)
 { return a+b;
 }
void main()
 {
  int a,b;
  int (*funPtrSum)(int, int) = &sum;
  printf("\n\t Enter the value for a and b: \n");
  scanf("%d%d", &a, &b);
  printf("\n\t Sum of %d and %d using Function Pointer is:
                %d", a,b, funPtrSum(a,b));
  printf("\n\t Sum of %d and %d using Normal Function
                is %d", a,b, sum(a,b));
 }
```

Output:

```
Enter the value for a and b: 15
45

Sum of 15 and 45 Using Function Pointer is: 60
Sum of 15 and 45 Using Normal Function is: 60
```

File Handling:

→ C files I/O functions handle data on a secondary storage device such as a hard disk.

→ C can handle files as two types.

(i) Stream - Oriented Data (Text) files

(ii) System Oriented Data (Binary) Files

Stream Oriented Data files:

→ The data is stored in the same manner as it appears on the screen. The I/O operations like buffering, data conversions etc, take place automatically.

System Oriented Data Files:

→ System oriented data files are more closely associated with the OS and data stored in memory without converting into text format.

Operations on files:

⇒ Creation of a new file
⇒ Opening an existing file
⇒ Reading data from a file.
⇒ Writing data in a file
⇒ Closing a file.

Input / output Functions:

→ To handle files in C, file input/output functions available in the stdio library are,

Function	Uses / Purpose
fopen	Opens a file
fclose	Closes a file
getc	Reads a character from a file
putc	Writes a character to a file
getw	Read integer
putw	Write an integer
fprintf	Prints formatted output to a file
fscanf	Reads formatted input from a file
fgets	Read a string of characters from a file.
fputs	Write a string of characters to a file
feof	Detects end-of-file marker in a file.

Basic Format of fopen():

```
FILE *fopen(const char *filePath, const char *mode);
```

⇒ File path → The first argument is a pointer to a string containing the name of file to be opened.

⇒ mode → The second argument is an access mode.

→ fopen() function returns NULL in case of failure and returns FILE stream pointer on success.

Example:

```
#include <stdio.h>
int main()
{
  FILE *fp;
  fp = fopen("fileName.txt", "w");
  return 0;
}
```

Basic Format of fclose()

```
int fclose(FILE *stream);
```

→ fclose() function is a C library function that releases the memory stream opened by the fopen() function.

Example:

```
#include <stdio.h>
int main() {
  FILE *fp;
  fp = fopen("fileName.txt", "w");
  fprintf(fp, "%s", "Sample Text");
```

```
    fclose(fp);
    return 0;
}
```

Basic Format of getc()

```
int getc (FILE *stream);
```

→ getc() function is a C library function, which reads a character from a file that has been opened in read mode by the fopen() function.

- Example:

```
#include <stdio.h>
int main()
{
 FILE *fp = fopen ("fileName.txt", "r");
 int ch = getc(fp);
 while (ch != EOF)
   {
     putchar(ch);
     ch = getc(fp);
   }
  if (feof(fp))
      printf ("\n Reached the end of file :");
  else
      printf ("\n Something gone wrong");
  fclose(fp);
  getchar();
  return 0;
}
```

Basic Format of putc()

```
int putc (int c, FILE *Stream);
```

→ The putc() function is a C library function, which is used to write a character to a file.

Example:

```
int main (void)
{
  FILE *fileName;
  char ch;
  fileName = fopen ("anything.txt", "wt");
  for (ch = 'D'; ch <= 'S'; ch++)
  {
    putc(ch, fileName);
  }
  fclose(fileName);
  return 0;
}
```

Basic Format of getw()

```
int getw (FILE *Stream);
```

→ getw() function is used to read an integer from a file that has been opened in read mode.

Basic Format of putw()

```
int putw (int c, FILE *Stream);
```

→ putw() function is used to write an integer to a file.

Example:

```
int main(void)
{
    FILE *fileName;
    int i=2, j=3, k=4, n;
    fileName = fopen("anything.c", "w");
    putw(i, fileName);
    putw(j, fileName);
    putw(k, fileName);
    fclose(fileName);

    fileName = fopen("test.c", "r");
    while (getw(fileName) != EOF)
    {
      n = getw(fileName);
      printf("Value is %d\t", n);
    }
  fclose(fp);
  return 0;
}
```

Basic Format of fprintf()

```
int fprintf(FILE *stream, const char *format...);
```

- fprintf() function passes arguments according to the specified format to the file indicated by the stream.

Example:

```
int main(void)
{
```

```
FILE *fileName;
fileName = fopen ("anything.txt", "r");
fprintf (fileName, "%s %s %d", "Welcome", "to", 2018);
fclose (fileName);
return (0);
}
```

Basic Format of fscanf()

```
int fscanf (FILE *stream, const char *format, ...);
```

→ fscanf function reads formatted input from a file.

Example:

```
int main()
{
char str1[10], str2[10];
int yr;
FILE *fileName;
fileName = fopen ("anything.txt", "w+");
fputs ("Welcome to", fileName);
rewind (fileName);
fscanf (fileName, "%s %s %d", str1, str2, &yr);
printf (" First word %s\t", str1);
printf (" Second word %s\t", str2);
printf (" Year-name %d\t", yr);
fclose (fileName);
return (0);
}
```

Basic format of fgets()

```
char *fgets (char *str, int n, FILE *stream);
```

→ fgets() function is implemented in file-related programs for reading strings from any particular file.

Example:

```
Void main(void)
{
 FILE *fileName;
 char ch[100];
 fileName = fopen ("anything.txt", "r");
 printf("%s", fgets(ch, 50, fileName));
 fclose (fileName);
}
```

Basic Format of fputs()

```
int fputs (const char *str, FILE *stream);
```

→ fputs() function is implemented in file-related programs for writing strings to any particular File.

Example:

```
int main()
{
 FILE *fp;
 fp= fopen ("fileName.txt", "w");
 fputs ("This is a sample text file", fp);
 fputs ("This file contains some sample text data", fp);
 fclose(fp);
 return 0;
}
```

Basic Format of feof()

```
int feof (FILE *stream);
```

→ feof() function returns true in case the end of file is reached, otherwise it returns false.

Example:

```
int main()
{
  FILE * filee = NULL;
  char buf[50];
  filee = fopen ("infor.txt","r");
  if (filee)
  {
   while ( !feof(filee))
   {
    fgets (buf, sizeof(buf), filee);
    puts(buf);
   }
  fclose (filee);
  }
 return 0;
}
```

Preprocessor Directives:

→ Preprocessors are programs that processes our source code before compilation.

→ Preprocessor programs provides preprocessor directives which tell the compiler to preprocess the source code before compiling.

→ The preprocessor directive begins with a # (hash) symbol.

Types of Preprocessor Directives :

① Macros
② File Inclusion
③ Conditional Compilation
④ Other directives.

Macro:

Syntax : # define.

→ This macro defines constant value and can be any of the basic data types.

File Inclusion:

Syntax : #include <file_name>

→ The source code of the file "file name" is included in the main program at the specified place.

Conditional Compilation

Syntax : #ifdef, #endif, #if, #else, #ifndef.

→ Set of commands are included or excluded in source program before compilation with respect to condition.

Other Directives

Syntax : #undef, #pragma

⇒ #undef is used to undefine a defined macro variable.
⇒ #pragma is used to call a function before and after main function in c program.

Other Pragma Directives:

→ Pragma is used to call a function before and after main function in C program.

① #Pragma startup <function_name_1> → This directive executes function named "function_name_1" before program startup (before the control passes to main())

② #Pragma exit <function_name_2> → This directive executes function named "function_name_2" just before the control returns from main()

Preprocessor Examples:

① #define SIZE 10

↳ This directive tells the compiler to replace the instance of SIZE with 10.

② #include <stdio.h>

↳ This directive tells to get stdio.h from system libraries and add the text to the correct source file.

③ #include "myfile.h"

↳ This line tells the compiler to get myfile.h from local director and add the contents with current source file.

④ #undef F_SIZE
#define F_SIZE 80

↳ It tells to undefine existing F_SIZE and define it as 80

⑤ #ifndef MSG
#define MSG "welcome"
#endif

↳ It tells to define MSG only if MSG isn't already defined.

UNIT - III LINEAR DATA STRUCTURES

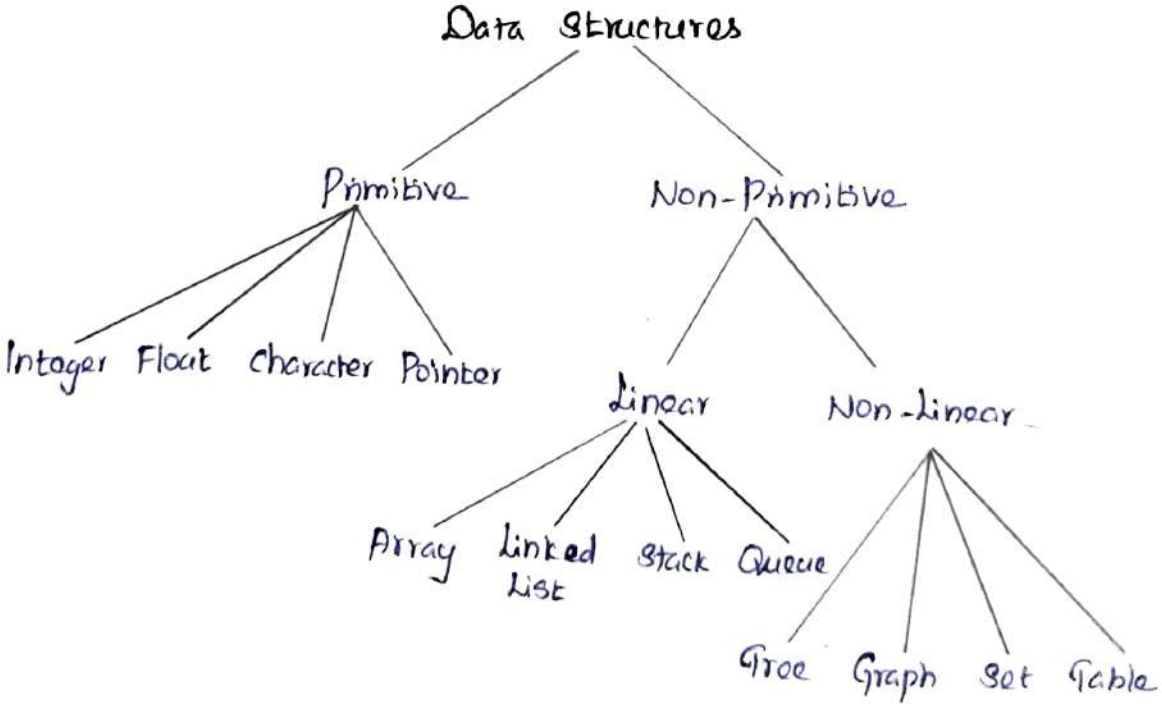
Abstract Data Types (ADTs) – List ADT – Array-Based Implementation – Linked List – Doubly Linked List – Circular Linked List – Stack ADT – Implementation of Stack – Applications – Queue ADT – Priority Queues – Queue Implementation – Applications.

Data Structures:

→ A data structure is a specalized format for organizing, processing, retrieving and storing data.

→ A data structure is defined as a way of organizing and storing data items by considering its relationship to each other, in a computer so that it can be used efficiently.

Types of Data Structures:



⇒ In Linear Data Structures values are arranged in linear or sequential Fashion. eg Array, Linked List, Stack Queue.

⇒ In Non-linear data structures, the data values are not arranged in sequential order. In non-linear data structure each node may point to several other nodes.

Eg tree, graph, set and table.

Characteristics of data structure:

(i) It depicts the logical representation of data in computer memory.

(ii) It represents the logical relationship between the various data elements.

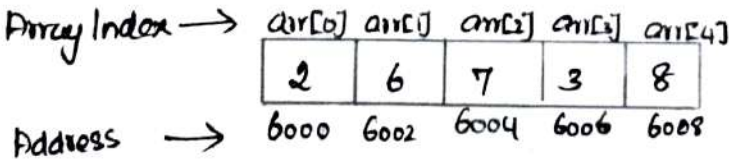
(iii) It helps in efficient manipulation of stored data elements.

(iv) It allows the programs to process the data in an efficient manner.

Arrays and its Representations:

→ an array is a collection of similar type data elements stored at consequetive locations in the memory.

→ The group of array elements is referred with a common name called array name.



arr[5] = [2,6,7,3,8]

Abstract Data Type (ADT) Array:

→ ADT array is an object.

→ Two types of array.

(i) One dimensional Array

(ii) Two dimensional Array.

One Dimonsional Array:

→ a group of same type data elements such as integers, float or characters. `array []`

→ a group of integers A[1,2,3,4,5 }

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
1	2	3	4	5
2000	2002	2004	2006	2008

One Dimensional Array - Insertion:

Initial Array

a[0]	a[1]	a[2]	a[3]	a[4]
-1	3	5	22	77

Insert 4 at end

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
-1	3	5	22	77	4

Insert 2 at index location 2

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
-1	3	2	5	22	77	4

One Dimensional Array - Deletion:

Initial Array

a[0]	a[1]	a[2]	a[3]	a[4]
-1	3	5	22	77

Delete at end

a[0]	a[1]	a[2]	a[3]
-1	3	5	22

Delete at Index Location 2

a[0]	a[1]	a[2]
-1	3	22

Two or Multidimensional Array:

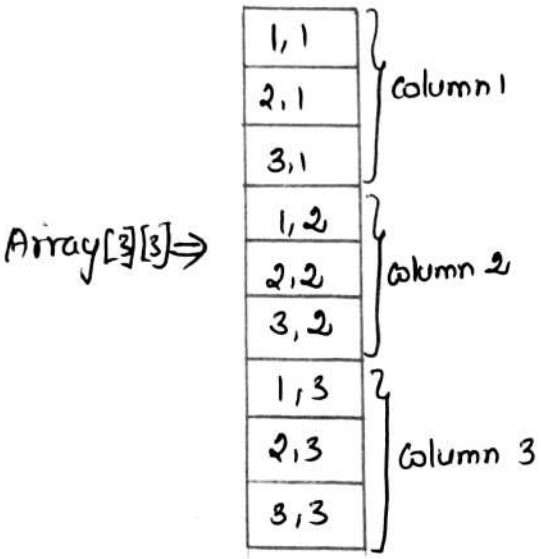
→ It is a group of data elements where each element is itself an array. `array [] []`

→ a group of strings or integers {aaa, bbb, ccc, ddd, eee}

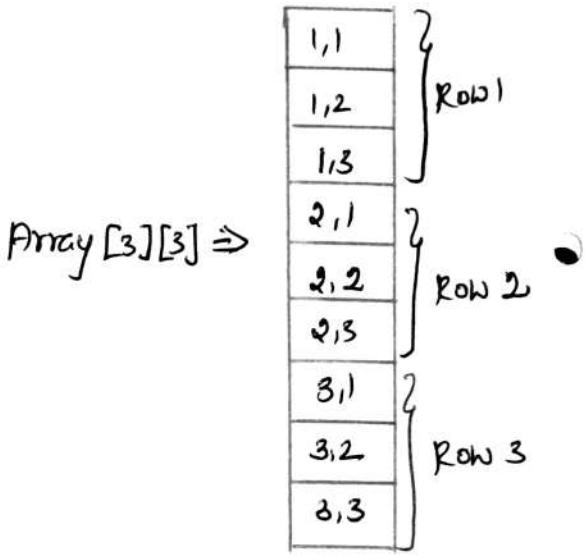
Multidimensional Array:

eg - matrix 3x3

Representation of array array[3][3] in column order



Representation of array array[3][3] in row order



List ADT:

→ An ordered list or list ADT is either empty or it can be written as $(a_1, a_2, a_3, \ldots a_n)$, where a_i is an atom from some set S.

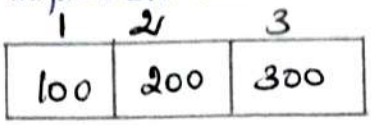
Example:

(i) Days of the week: (Monday, Tuesday, Wednesday, Thursday Friday, Saturday, Sunday)

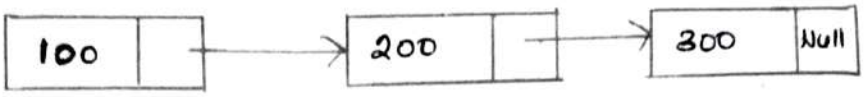
(ii) Months of the year: (January, February, March, April, May, June, July, August, September, October, November, December)

Two ways of implementing a List.

① Array based Implementation.



② Linked List Implementation:



Various Operations of List:

1) Find the length of the list n
2) Read the list from left-to-right (or) right-to-left.
3) Retrieve the ith element.
4) Store a new value into the ith position.
5) Insert a new element at position causing elements numbered $i, i+1, \ldots, n$ to become numbered $i+1, i+2, \ldots, n+1$
6) Delete the element at position causing elements numbered $i+1, \ldots n$ to become numbered $i, i+1, \ldots n-1$.

Array-Based Implementation of List:

→ In array-based implementation all the elements of list are stored in cells of an array.

→ Array representation is known as a sequential mapping, as we are storing a_i and a_{i+1} into consecutive locations 1 and $i+1$ of the array.

→ Hence it is possible to retrieve or modify the values of random elements in the list in a constant amount of time.

Array Implementation Operations:

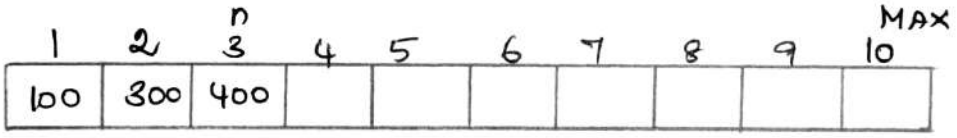
1) Creation of a list
2) Inserting an element into the list
3) Displaying the elements
4) Deleting an existing node
5) Finding the size of the list
6) Search an element in the list
7) Reversing the elements of the list
8) Checking whether the list is Empty.

⇒ The data members of the class are,

1) An array a[MAX]
2) The actual size of the list namely, n.

1) Creation of a List:

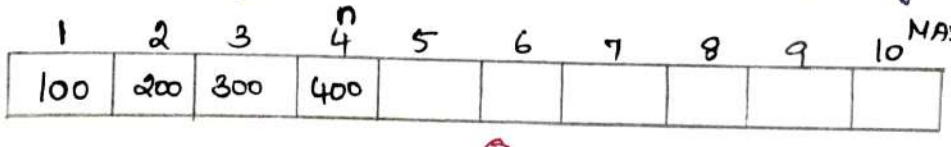
→ To create a list with an array-based implementation



2) Inserting an element into the List.

→ In order to insert a new element, we have to specify the position of insertion and value that to be inserted

→ When we want to insert 200 at the position all the elements from the position 2 to n have to be shifted right one position to make a space for insertion. Also the size of list n will be incremented by 1.



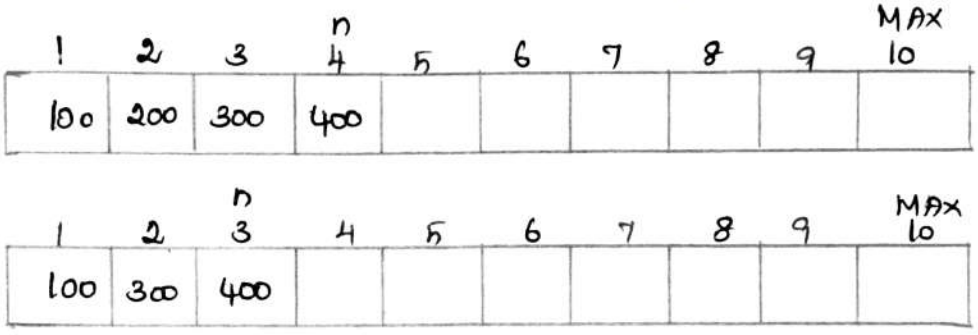
3) Displaying the elements:

→ All the elements of the list will be displayed starting from the 1st position to nth position.

4) Deleting an existing node:

→ In order to delete an existing node at the position p all the elements from p+1 to n have to be shifted left by one position.

→ To delete the node having the data 200.



→ Also the size of the list will be decremented by 1.

5) Finding the size of the list:

→ The data member n is used to tell the size of list.

6) Search an element in the list:

→ For searching a node we have to start the searching from the first node to the desired node having that key. It returns the position of first occurrence of an item in a list.

7) Reversing the elements of the List.

In array based implementation - a[n] to a[1] will display all the elements in reverse order.

8) Checking whether the List is Empty:

→ When the list is empty True will be displayed, Otherwise false will be displayed.

Linked List :

→ A linked list is a linear data structure and is made up of number of nodes.

→ Linked List is a collection of data elements stored in such a manner that each element points at the next element in the list.

→ The elements of a linked list are also referred to as nodes.

→ Each node has two parts – INFO and NEXT.

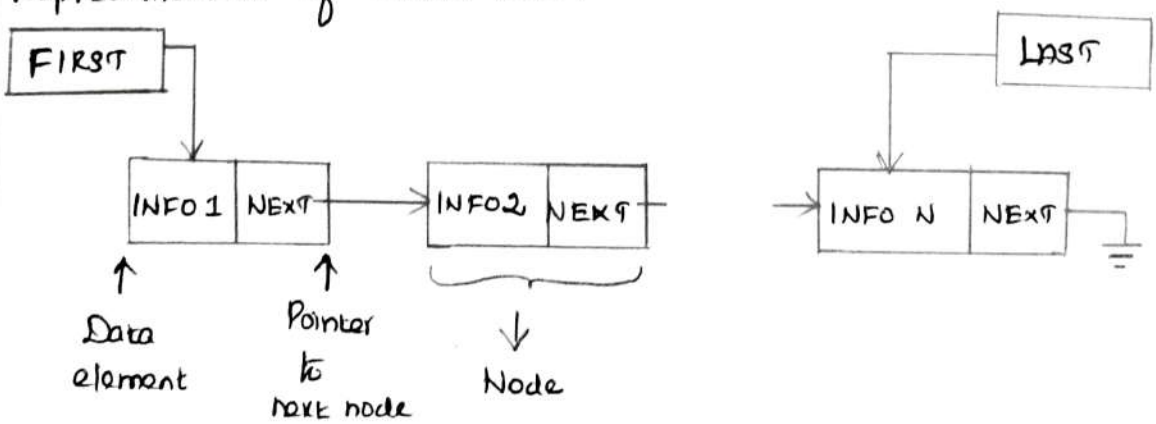
→ The INFO part contains the data element while the NEXT part contains the address of next node.

→ The NEXT part of the last node of the list contains a NULL value indicating the end of list.

→ The beginning of the list is indicated with the help of a special pointer called FIRST.

→ Similarly the end of list is indicated by a pointer called LAST.

Representation of Linked List :



Advantages:

(i) Linked list facilitate dynamic memory management by allowing elements to be added or deleted at any time during program execution.

(ii) The use of linked list ensures efficient utilization of memory space as only that much amount of memory space is reserved as is required for storing the list of elements.

(iii) It is easy to insert or delete elements in a linked list unlike arrays which require shuffling of other elements with each insert and deletion operation.

Disadvantages:

(i) A linked list element requires more memory space in comparison to an array element because it has also store the address of next element in list.

(ii) Accessing an element is a little more difficult in linked list than arrays because unlike arrays there is no index identifier associated with each list element.

Linked List Implementation:

(i) Declaring the List

(ii) Defining the Linked List Operations.

Linked List node Declaration:

```
struct node {
    int INFO;
    struct node *NEXT;
};
typedef struct node NODE;
```

Linked List Operations:

(i) Insertion
(ii) Deletion
(iii) Search
(iv) Print

Insertion:

(i) Inserting the new element at the beginning of list.
(ii) Inserting the new element at the end of list.
(iii) Inserting the new element somewhere at the middle of list.

Deletion:

(i) Deleting an element from the beginning of list.
(ii) Deleting an element from the end of list.
(iii) Deleting an element somewhere from the middle of list.

Search:

→ To find an element in linked list

(i) Traverse the list sequentially starting from first node
(ii) Return the location of searched node as soon as a match is found.
(iii) Return a search failure notification if the entire list is traversed without any match.

Print:

→ It displays all the linked list elements on the screen.

Types of Linked List:

(i) Singly Linked List
(ii) Doubly Linked List
(iii) Circular Linked List

Singly Linked List:

→ Each node has info part and only one link part.

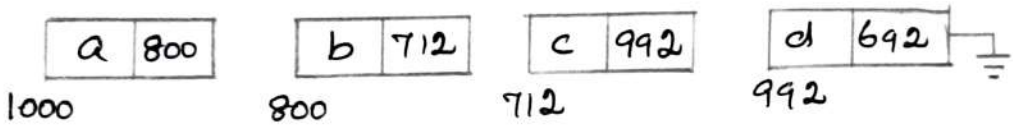
→ Each link part contains the address of next node in list.

→ Link part of the last node contains NULL value which signifies the end of node.

→ It allows for traversing the list in only one direction.

The general form of Singly Linked List:

Actual Representation:



Operations of Singly Linked List:

(i) Creating a List

(ii) Inserting an element in a List

(iii) Deleting an element in a List

(iv) Viewing a List

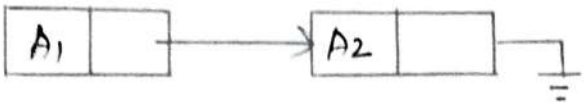
(v) Modifying an element in List.

Creating a List:

→ The node structure contains

(i) A info part called number and name.

(ii) A link part called next.




```
Struct node
{
  ElementType  Element;
  Position Next;
};
```

Is Empty:

→ This function is used to check whether the list is empty or not.

```
int IsEmpty (List L)
{
  return L->Next == NULL;
}
```

⇒ This routine will return 1 value if the list is empty.

Is Last:

→ This function is used to check whether the current position in the last position in the Linked List.

```
int IsLast (Position P, List L)
{
  return P->Next == NULL;
}
```

⇒ This routine will return 1 if P is the last position in the list L.

Find:

→ This function is used to find the element x in the list.

```
Position Find (ElementType x, List L)
{
  Position P;
  P = L->Next;
  While (P != NULL && P->Element != x)
  P = P->Next;
  return P;
}
```

Find Previous:

→ This function is used to find the previous position of the given element x.

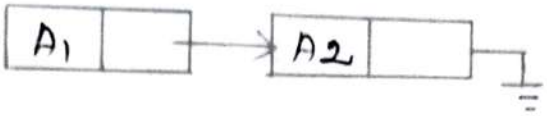
```
Position FindPrevious (ElementType x, List L)
{
  Position P;
  P=L;
  While (P->NEXT != NULL && P->NEXT->Element != x)
     P = P->NEXT;
  return P;
}
```

Insertion:

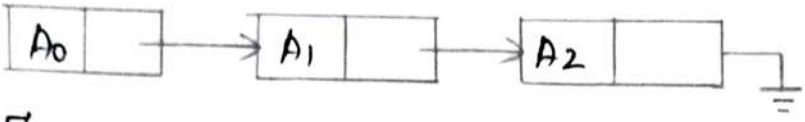
(i) Insert at the beginning

(ii) Insert at the end

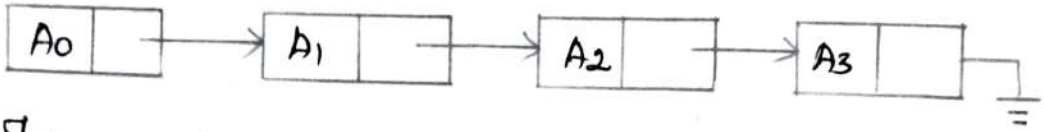
(iii) Insert after a particular node



Insert at the beginning:

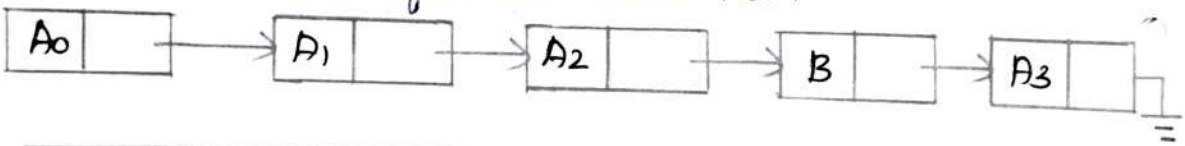


Insert at the end:



Insert after a particular Node:

→ To insert after the Node A2.




```
Void Insert (ElementType x, List L, Position P)
{
  Position Tmpcell;
  Tmpcell = malloc (sizeof (struct Node));
  If (Tmpcell == NULL)
     Fatal Error (" Out of space");
  Tmpcell → Element = x;
  Tmpcell → Next = P → Next;
  P → Next = Tmpcell;
}
```

⇒ This routine will return the position of previous elements of X. This function is used to insert a new element in the list.

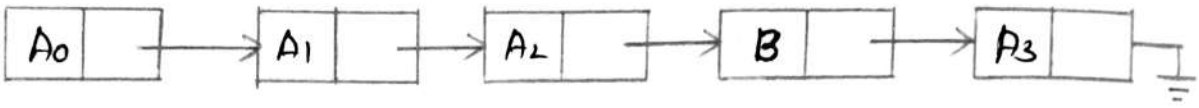
Deletion:

(i) Delete at beginning

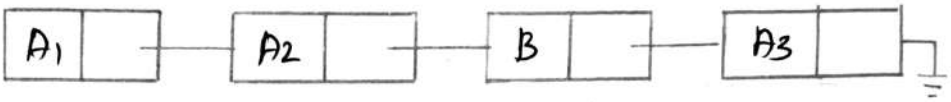
(ii) Delete at end

(iii) Delete a particular node.

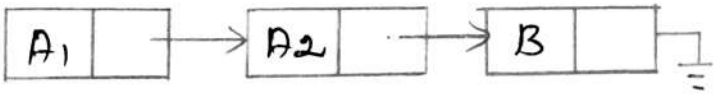
Delete at beginning:



After deleting first node, then the List will be,

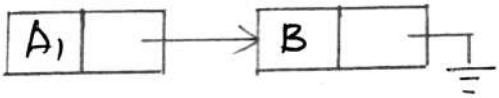


Delete at the end:



Delete a particular node:

→ To delete a node A2 then the List will be,




```
Void Delete (ElementType x, List L)
{
  Position P, TmpCell;
  P = FindPrevious (x, L);
  if ( !IsLast (P, L))
  {
    TmpCell = P->Next;
    P->Next = TmpCell->Next;
    free (TmpCell);
  }
}
```

Delete List:

→ This function is used to delete the complete List L.

```
Void DeleteList (List L)
{

  Position P, Tmp;
  P = L → NEXT;
  L → NEXT = NULL;
  While ( P != NULL)
   {
     Tmp = P → NEXT;
     Free (P);
     P = Tmp;
   }
}
```

Viewing a List:

→ Traverse all the elements of the list starting from the first node.

Modifying an element in a List:

→ A particular node can be searched and the existing content may be replaced by a new content.

Doubly Linked List:

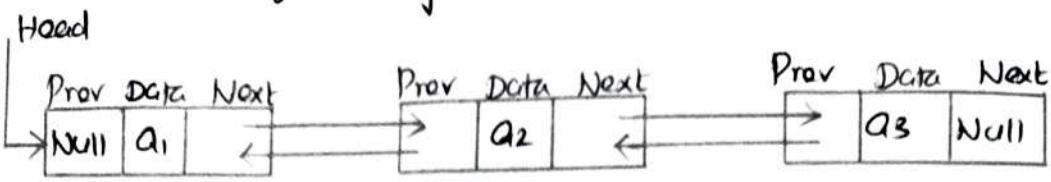
→ Doubly Linked List has

(i) A node

(ii) A pointer to previous node

(iii) A pointer to next node.

General Form of Doubly Linked List:



Operations of Doubly Linked List:

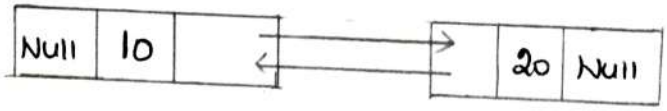
(i) Insertion
(ii) Deletion
(iii) Find
(iv) Find Previous
(v) Find Next
(vi) Print List.

Node Declaration:

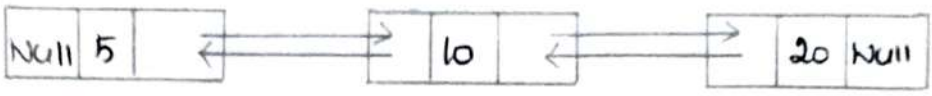
```
struct Node
{
 int element;
 struct node *NEXT;
 struct node *PREVIOUS;
};
```

Insertion:

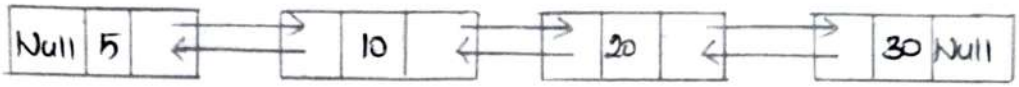
(i) Inserting at first node in the list
(ii) Inserting at last node in the list
(iii) Inserting at an intermediate node in the list.



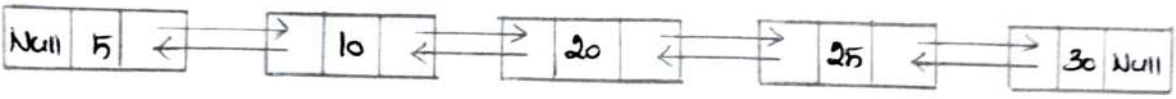
Insert at first node in the list:



Insert at Last node in the List:



Insert at an Intermediate node in the List:




```
Void insert (int x , List L , Position P)
{
  Struct node *newnode;
  newnode = malloc ( sizeof( StructNode));
  If (newnode != NULL)
  {
    newnode -> Element = x;
    newnode -> NEXT = P -> NEXT;
    newnode -> PREVIOUS = P -> NEXT -> PREVIOUS;
    P -> NEXT = newnode;
    P -> NEXT -> PREVIOUS = newnode;
  }
}
```

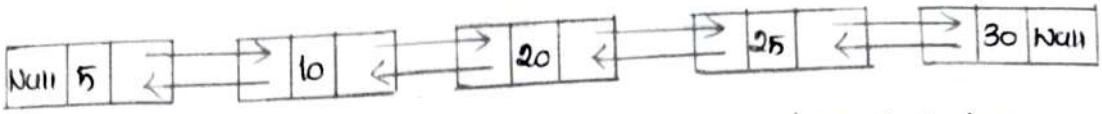
Deletion:

(i) Deletion at first node in the List

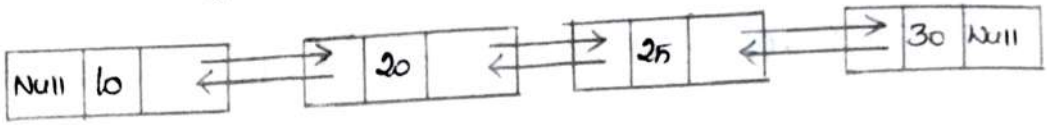
(ii) Deletion at Last node in the List

(iii) Deletion at intermediate node in the List

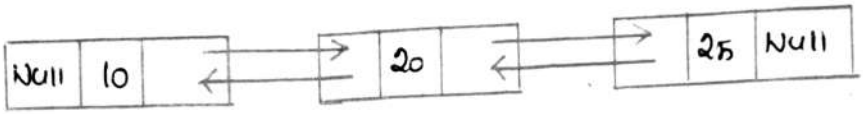
Deletion at First node in the List:



After deleting the first node then the List will be,

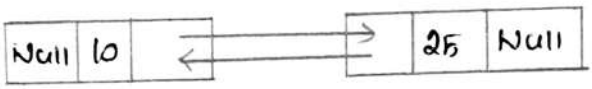


Deletion at Last node in the List:



Deletion at Intermediate node in the List:

To delete a node 20 then the List will be,




```
Void Delete (int x, List L)
{
  Position P;
  P = Find (x, L);
  if (IsLast (P, L)) // deleting the Last node
  {
     Temp = P;
     P -> PREVIOUS -> NEXT = NULL;
     Free (Temp);
  }
  else
   {
    Temp = P;
    P -> PREVIOUS -> NEXT = P -> NEXT;
    P -> NEXT -> PREVIOUS = P -> PREVIOUS;
    free (Temp);
  }
}
```

Advantages of doubly Linked List :

(i) Deletion operation is easier

(ii) Finding predecessor and successor of a node is easier

Disadvantage of doubly Linked List:

(i) More memory space is required, since it has two pointers.

Circular Linked List :

→ Circular linked list is a linked list in which the last node of the list points to the first node of the list.

Types of circular Linked List:

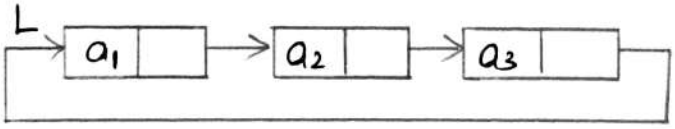
— 2 types

(i) Singly circular Linked List

(ii) Doubly circular Linked List.

Singly Circular Linked List :

→ Singly circular linked list is a linked list in which the last node points to the first node.



Operations :

(i) Insertion

(ii) Deletion

(iii) View

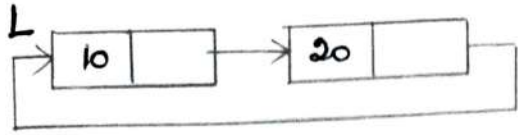
(iv) Find.

Insertion :

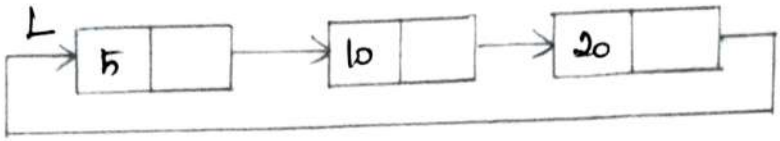
(i) Insertion at beginning

(ii) Insertion at end

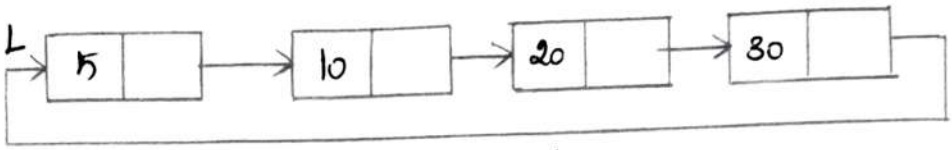
(iii) Insertion at intermediate node.



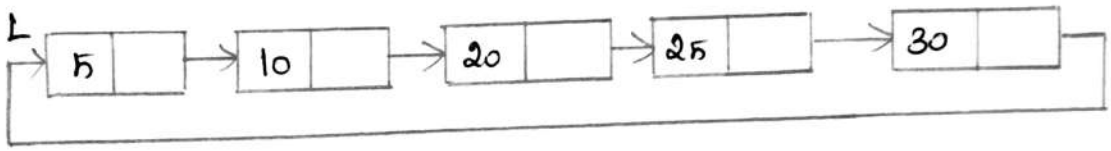
Insertion at beginning :



Insertion at end :



Insertion at intermediate node :




```
Void insert (int x, List L)
{
  Struct node * newnode;
  newnode = malloc (sizeof (Struct node));
  if(newnode != NULL)
  {
     P=L;
     While (P -> Next != L)
        P = P -> Next;
     newnode -> Element = x;
     P -> Next = newnode;
     newnode -> Next = L;
  }
}
```

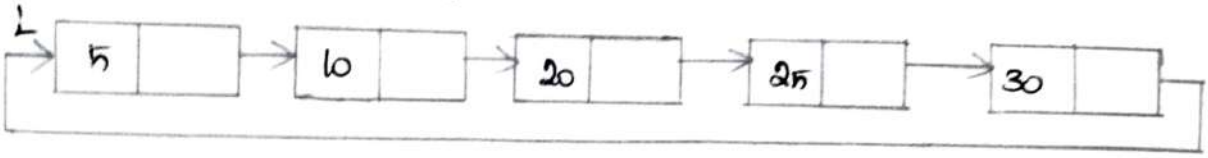
Deletion:

(i) Deletion at beginning

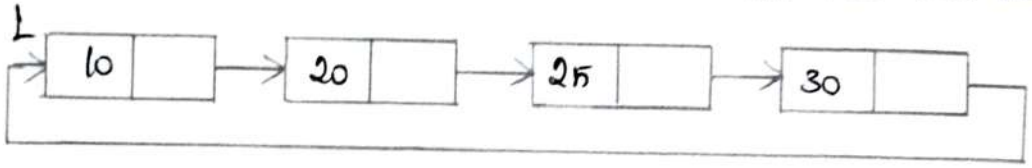
(ii) Deletion at end

(iii) Deletion at intermediate node.

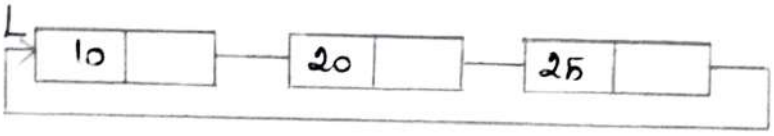
Deletion at beginning:



To delete the first node then the list will be,

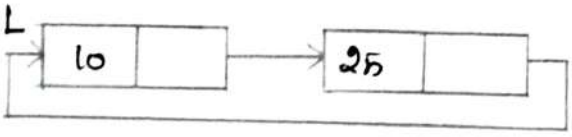


Deletion at end:



Deletion at Intermediate node:

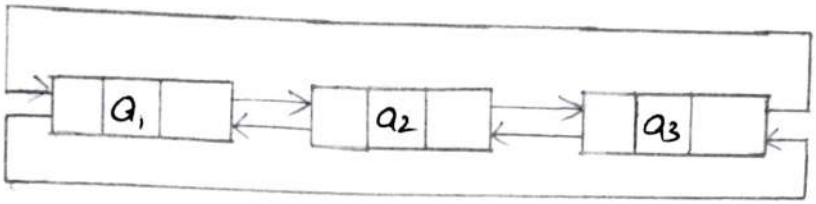
→ To delete node 20 then the list will be,




```
Void deleteLast (List L)
{
  Position Temp, P, L;
  P = L;
  While (P->Next->Next != L) // Find Previous
     P = P->Next;
  Temp = P->Next;
  P->Next = L;
  free(Temp);
}
```

Doubly Circular Linked List:

→ A doubly circular linked list is a doubly linked list in which the forward link of the last node points to the first node and the backward link of the first node points to the last node of the list.

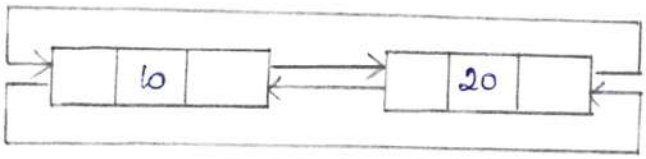


Operations:

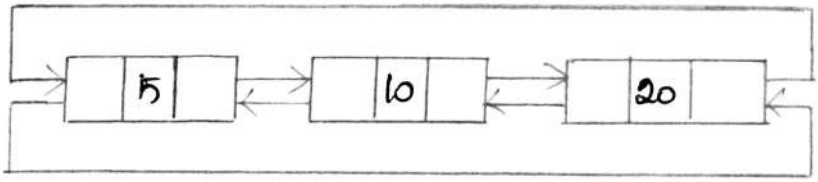
(1) Insertion
(2) Deletion
(3) View
(4) Find.

Insertion:

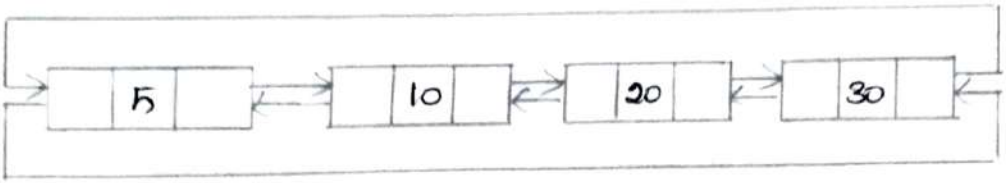
(1) Insertion at the beginning
(2) Insertion at the end
(3) Insertion at intermediate node.



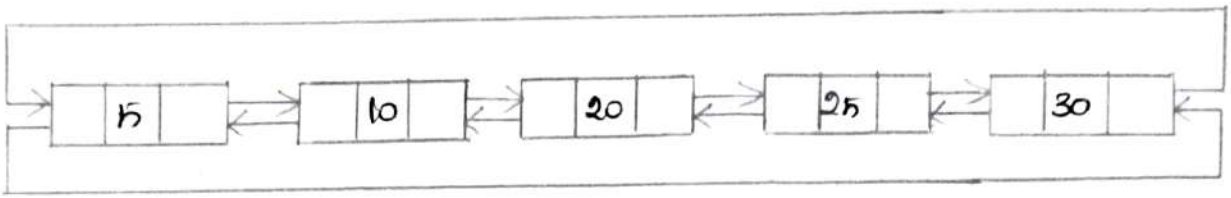
Insertion at the beginning:



Insertion at end:



Insertion at intermediate node:

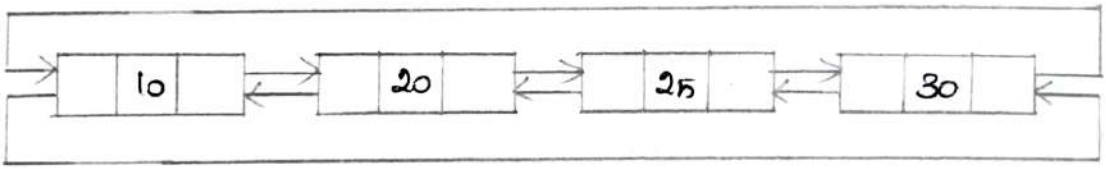


Deletion:

(i) Deletion at beginning
(ii) Deletion at end
(iii) Deletion at intermediate node.

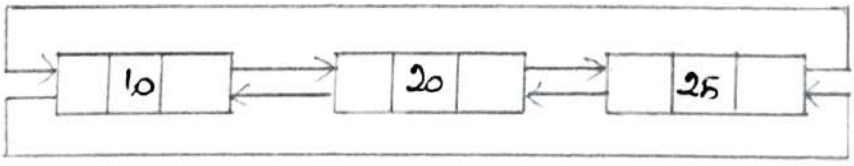
Deletion at beginning:

→ To delete node 5 then the list will be,



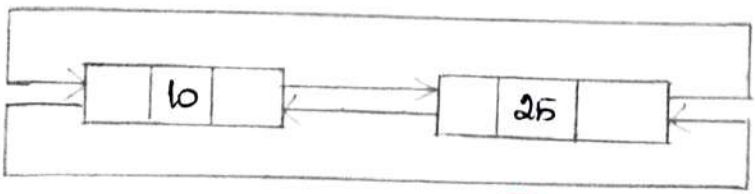
Deletion at end:

→ To delete node 30, then the list will be,



Deletion at intermediate node:

→ To delete node 20, then the list will be,



Advantages of Circular Linked List:

(i) It allows traversing the list starting at any position

(ii) It allows quick access to the first and last nodes

(iii) It can traverse the list in either direction.

Application of Linked List:

→ Three Applications

(i) Polynomial Manipulation

(ii) Sorting method in linear time - Radix Sort

(iii) MultiList - Keep track of Course Registration at a University.

Polynomial Manipulation:

→ Polynomial manipulations such as addition, subtraction multiplication, division can be performed using Linked List.

Declaration of Linked List: - Polynomial ADT:

```
struct poly
{
  int coeff;
  int power;
  struct poly *Next;
  *list1, *list2, *list3;
}
```

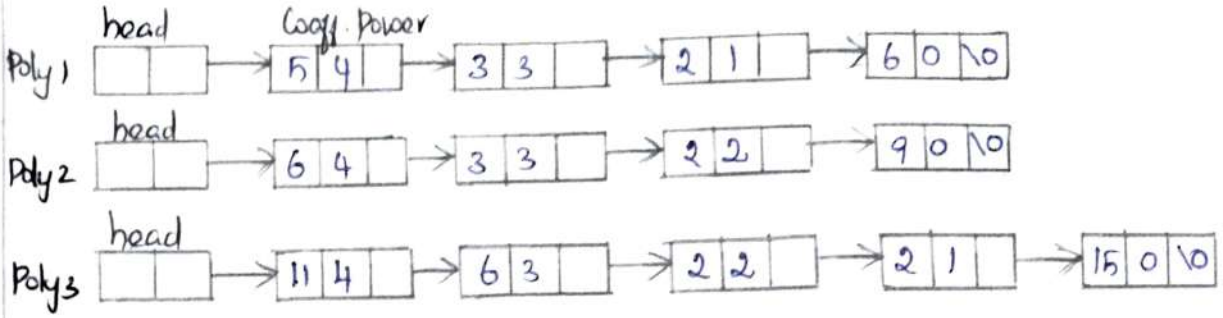
Example:

$Poly1 = 5x^4 + 3x^3 + 2x + 6$

$Poly2 = 6x^4 + 3x^3 + 2x^2 + 9$

$Poly3 = 11x^4 + 6x^3 + 2x^2 + 2x + 15$

Linked List Representation:



Creation of Polynomial:

```
poly create (poly *head, poly *newnode)
{
  poly *p;
  if (head == NULL)
  {
     head = newnode;
     return head;
  }
  else
  {
     p = head;
     while (p -> Next != NULL)
     p = p -> Next;
     p -> Next = newnode;
  }
  return head;
}
```

Addition of two Polynomial:

```
Void add()
{
  poly *poly1, *poly2, *newnode;
```

```
poly1 = list1;
poly2 = list2;
while (poly1 != NULL && poly2 != NULL)
{
  newnode = malloc (sizeof (struct poly));
  if (poly1 -> power == poly2 -> power)
  {
    newnode -> coeff = poly1 -> coeff + poly2 -> coeff;
    newnode -> power = poly1 -> power;
    newnode -> Next = NULL;
    list3 = Create (list3, newnode);
    poly1 = poly1 -> Next;
    poly2 = poly2 -> Next;
  }
  else
  {
    if (poly1 -> power > poly2 -> power)
    {
      newnode -> Coeff = poly1 -> Coeff;
      newnode -> power = poly1 -> power;
      newnode -> Next = NULL;
      list3 = Create (list3, newnode);
      poly1 = poly1 -> Next;
    }
    else
    {
      newnode -> Coeff = poly2 -> Coeff;
      newnode -> power = poly2 -> power;
      newnode -> Next = NULL;
```

```
            list3 = create(list3, nownode);
            poly2 = poly2 -> next;
        }
    }
}
}
```

Radix sort:

→ Radix sort is the generalized form of bucket sort.

→ It distributes the list of elements across different buckets from 0 to 9.

Procedure:

⇒ In first pass all elements are sorted accordingly to the least significant bit.

⇒ In second pass the numbers are arranged accordingly to the second least significant.

⇒ This process is repeated until it reaches the most significant bit of all numbers.

⇒ The number of passes depends upon number of digits.

Example: 64, 8, 216, 512, 27, 729, 0, 1, 343, 125

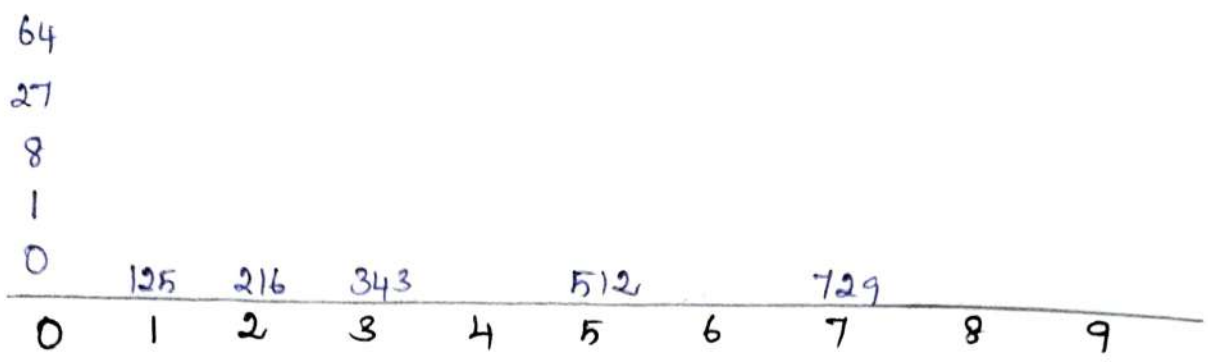
Pass 1:

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

Pass 2:

8		729							
1	216	27		64					
0	512	125		343					
0	1	2	3	4	5	6	7	8	9

Pass 3 :



The Sorted list is :

0, 1, 8, 27, 64, 125, 216, 343, 512, 729

Multilist :

→ A University with 40,000 students and 2500 courses needs to be able to generate two type of repoits.

① Report lists the registration for each class.

② Report list by student the classes that each student is registered.

→ In this method use a two dimensional array, it would have two million entries. The average student registers for about three courses have 120000 entries or roughly 0.1 percent have meaningful data.

→ This can be implemented by circular linked list.

→ A multilist is a structure in which a number of lists are combined to form a single aggregate structure.

Stack ADT:

→ Stack is a linear data structure in which items are added or removed only at one end called top of stack.

→ A Stack is based on Last In First Out (LIFO) principle that means the data item that is inserted last into the stack is the first one to be removed from stack.

→ Stack is said to be in overflow state when it is completely full and is said to be in underflow state if it is completely empty.

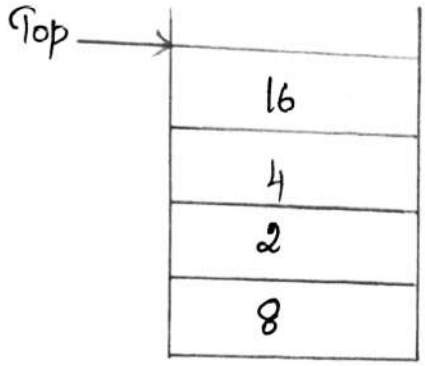
Example:

(i) Stack of coins

(ii) Stack of pancakes

(iii) Stack of bangles.

Logical Representation of Stack:



Stack - Operations:

① Push – Add or insert an element

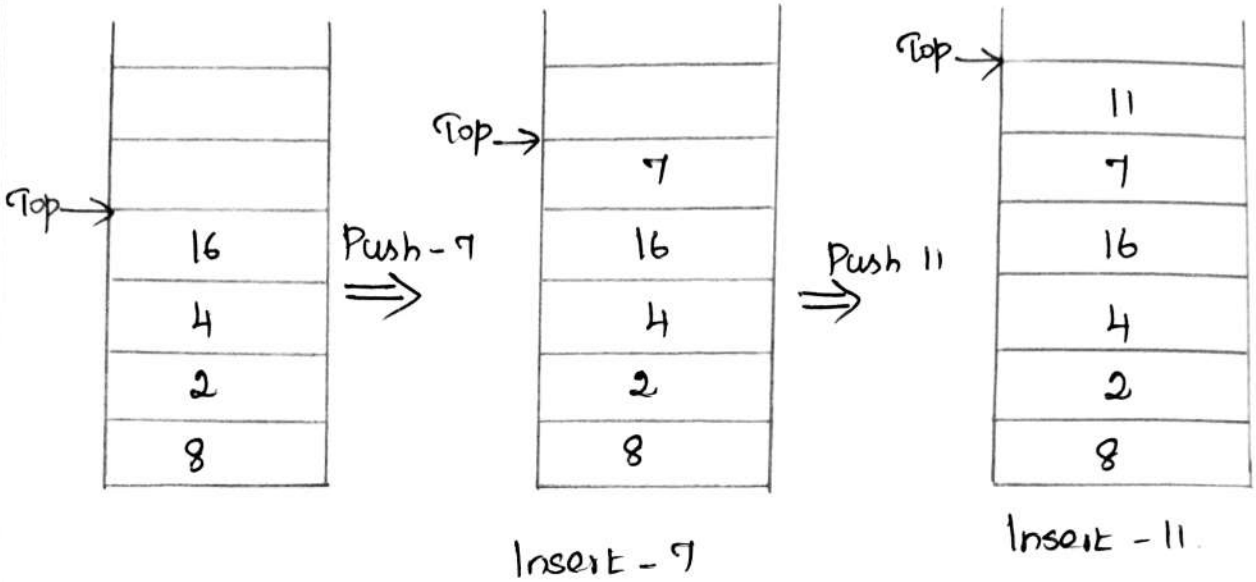
② Pop – Delete or remove an element.

Push:

(i) Receiving the element to be inserted.

(ii) Incrementing the stack pointer, top

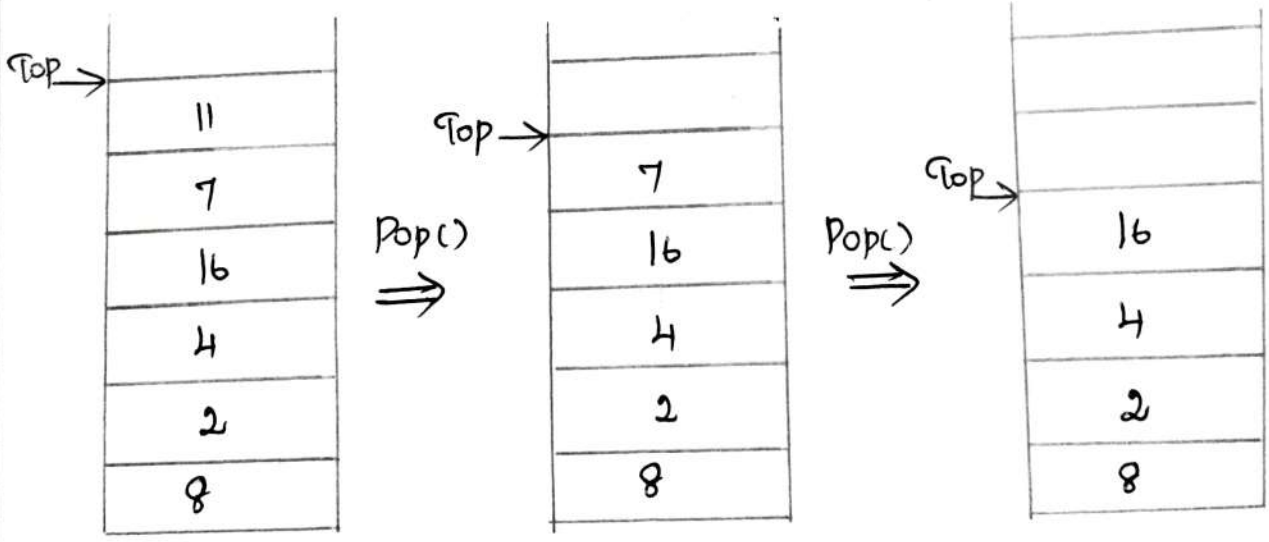
(iii) Storing the received element at new location of top.



Pop:

(i) Retrieving or removing the element at the top of stack.

(ii) Decrementing the stack pointer, top.



Implementation of Stacks:

(i) Array Implementation of Stack

(ii) Linked List Implementation of stack.

Array Implementation of Stack:

→ Operations performed in stack are

(i) Creating a stack

(ii) Inserting an element on to the stack - Push.

(iii) Deleting an element from the stack - Pop.

(iv) Checking which element is at the top of the stack.

(v) Checking whether a stack is empty or not.

Creating a Stack:

→ The stack is capable of storing size number of elements. The first element will be at position stack[0]. The last element is at position stack[size-1].

```
#define size 10
Struct stack
{
  int s[size];
  int top;
} s;
```

Stack Empty Operation:

→ When a stack is created it is empty and top is initialized to 1.

→ Before inserting or deleting elements into the stack, it is necessary to find out whether a stack is empty or not.

```
int isempty()
{
 if (s[top] == -1)
      return 1;
 else
    return 0;
}
```

⇒ If the stack is empty it will return the value **1**.

Stack full operation:

→ To insert an element into the stack, it is necessary to check whether the stack is full or not.

→ If it is full then the element cannot be inserted.

```
int isfull()
{
 if (s[top] >= size-1)
      return 1;
 else
    return 0;
}
```

⇒ If the stack is full it will the return the value 1

Push:

→ Push is a function which inserts new element at the top of the stack.

→ To push an element 'x' onto a stack, top pointer is incremented and then set stack[top] = x.

```
Void push (int X, Stack S)
{
  if (Isfull (S))
     Error ("Full Stack Overflow");
  else
   {
     top = top + 1;
     S[top] = X;
   }
}
```

Pop :

→ To pop an element, the stack[top] value is returned and top pointer is decremented.

```
void pop (stack S)
 {
  if (Isempty(S))
     Error ("Empty Stack Underflow");
   else
     {
      X = S[top];
      top = top - 1;
      return X;
     }
 }
```

Making Empty Stack :

```
Void MakeEmpty (Stack S)
 {
   S[top] = -1;
 }
```

Return top Element of the stack:

```
int TopElement (Stack S)
{
 if ( ! IsEmpty (S))
     return S[top];
 else
    Error ("Empty Stack");
        return 0;
}
```

Top and Pop:

→ This function is used to display the top element in the stack and then pop the element from the stack.

```
int Topand Pop (stack)
{
  if (Is Empty (S))
      Error ("Empty stack");
  else
      return S[top--];
}
```

Linked List Implementation of Stack:

→ The linked list implementation of a stack uses singly Linked List.

→ In which push operation is performed by inserting an element at the front of the list.

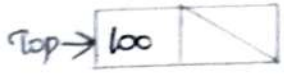
→ Pop operation is performed by deleting an element in the front of the list.

→ This involves dynamically allocating memory space at run time while performing stack operations.

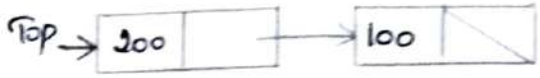
⇒ Initial Stack

Empty Stack

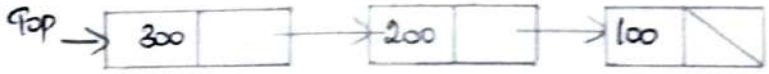
⇒ Push 100



⇒ Push 200

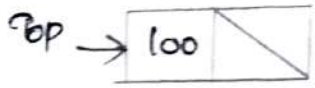


⇒ Push 300



⇒ Pop()

⇒ Pop()



⇒ Push 500

Node Declaration:

```
Struct Node
{
  ElementType Element;
  Pointer Next;
};
```

Is Empty:

→ This routine is used to check whether the stack is empty or not.

```
int IsEmpty (Stack S)
{
  return S → Next == NULL;
}
```

Create a Stack :

→ This routine is used to create a stack.

```
Create_Stack()
{
  Stack S;
  S = malloc (sizeof (struct Node));
  if (S == NULL)
        Fatal Error ("Out of Space");
  Make Empty (S);
  return S;
}
```

Make Empty :

→ This routine pop all the elements from the stack and makes the stack as empty.

```
Void MakeEmpty (Stack S)
{ if (S == NULL)
        Error ("Create a Stack First");
  else
        while ( ! IsEmpty (S))
          pop(S);
}
```

Top :

→ This routine will return top element in the stack.

```
int Top (Stack S)
{ if ( ! IsEmpty (S))
        return S → Next → Element;
  Error ("Empty Stack");
  return 0;
}
```

Push:

→This routine is used to push new element into the stack.

```
Void push (ElementType x, Stack S)
{
  Pointer Tmpcell;
  Tmpcell = malloc (sizeof (Struct Node));
  if (Tmpcell == NULL)
    FatalError ("Out of space");
  else
  {
    Tmpcell → Element = X;
    Tmpcell → Next = S → Next;
    S → Next = Tmpcell;
  }
}
```

Pop:

→This routine is used to pop the element from the stack.

```
Void pop (Stack S)
{
  Pointer Firstcell;
  if (IsEmpty (S))
      Error ("Empty stack");
  else
  {
    Firstcell = S → Next;
    S → Next = S → Next → Next;
    free (Firstcell);
  }
}
```

Applications of stack:

(i) Evaluating Arithmetic Expressions
(ii) Balancing the Symbols
(iii) Towers of Hanoi
(iv) Function Calls
(v) 8 Queen Problem.

① Evaluating Arithmetic Expressions:

(i) Infix Notation
(ii) Postfix Notation
(iii) Prefix Notation.

Infix Notation:

→ The arithmetic operator appears between the two operands.

A/B+C
x+y * z
(x+y)*(p+q)

Postfix Notation:

→ The arithmetic operator appears directly after the two operands. This is also called reverse polish notation.

x y+
ABC+*D/

Prefix Notation:

→ The arithmetic operator is placed before the two operands. This is called polish notation

+x y
/*A+BCD

Infix to Postfix Conversion:

Algorithm:

(i) Read one character at a time.

(ii) If the character is an operand place it on to output

(iii) If the character is operator push it on to the stack.

(iv) If the stack operator has a high or equal priority than input operator then pop that operator from the stack and place it on to the output

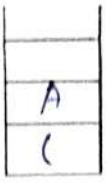
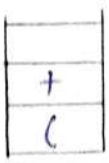
(v) If the character is a left paranthesis push it on to the stack.

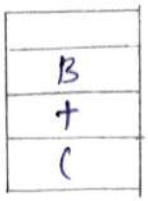
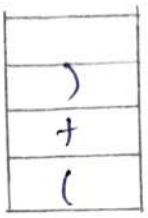
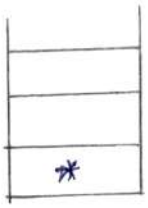
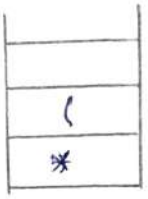
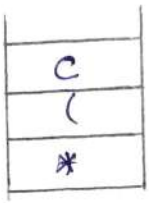
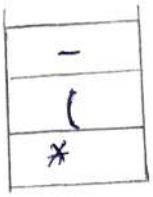
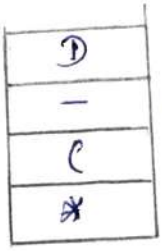
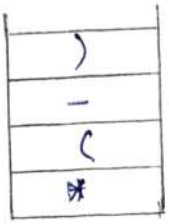
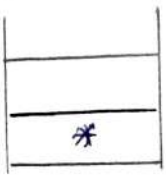
(vi) If the character is a right paranthesis pop all the operators from the stack, till it encounters left paranthesis, discard both the paranthesis in the output.

Example 1:

Convert it into postfix Expression: (A+B) * (C-D)

⇒ (A+B) * (C-D)

Input	Stack	Output
(	Push (	—
A	Push A  pop()	A
+	Push + 	A

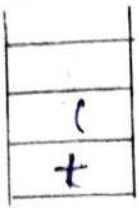
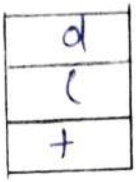
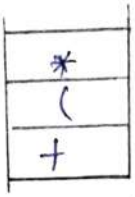
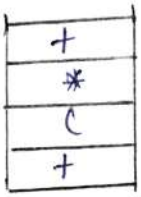
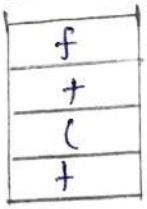
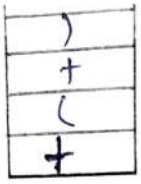
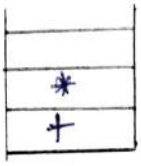
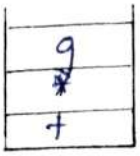
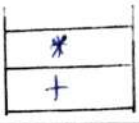
Input	Stack	Output.
B	Push B  pop()	AB
)	Push)  pop()	AB +
*	Push * 	AB +
(	Push (	AB +
C	Push C  pop()	AB + C
−	Push − 	AB + C
D	Push D  pop()	AB + CD
)	Push)  pop()	AB + CD −
	 pop()	AB + CD − *

The result of (A+B) * (C-D) is AB+CD-*

Example 2:

a+(b*c)+(d*e+f)*g

Input	Stack		Output
a	Push a	pop()	a
+	Push +		a
(	push (		a
b	push b	pop()	ab
*	push *		ab
c	push c	pop()	abc
)	push)	pop()	abc *
+	push +	pop()	abc *+

Input	Stack		Output
(	push (		
			abc*+
d	push d	 pop()	
			abc*+d
*	push *		
			abc*+d
e	Push e	 pop()	
			abc*+de
+	push +	 pop()	
			abc*+de*
f	push f	 pop()	
			abc*+de*f
)	push)	 pop()	
			abc*+de*f+
*	push *		
			abc*+de*f+
g	push g	 pop()	
			abc*+de*f+g
		 pop() pop()	abc*+de*f+g*+

The result of a+(b*c)+(d*e+f)*g is abc*+de*f+g*+

Evaluation of Postfix Expression:

(i) Read one character at a time.

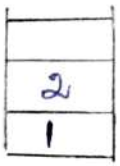
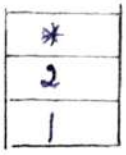
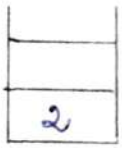
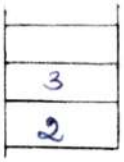
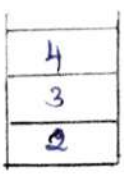
(ii) If the character is an operand push its associated value on to the stack.

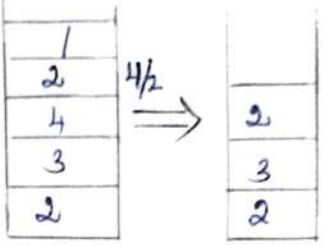
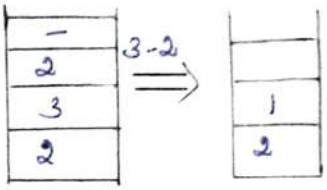
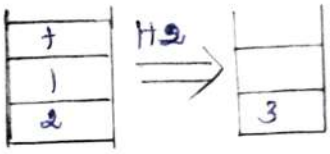
(iii) If the character is an operator pop two values from the stack, apply the operator to them and push the result on to the stack.

Example 1:

Consider the postfix Expression: A B * C D E / - +

then the values of A, B, C, D, and E are 1, 2, 3, 4, 2

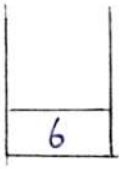
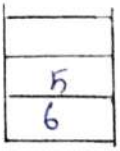
Read Character	Stack
A	Push A 
B	Push B 
*	Push *  1*2 ⟹ 
C	Push C 
D	Push D 

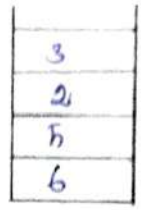
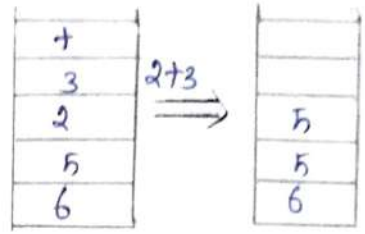
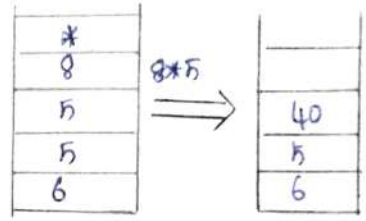
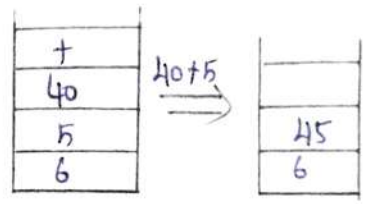
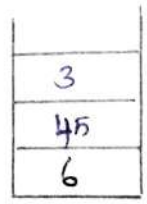
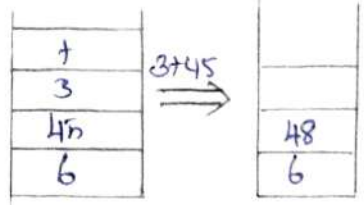
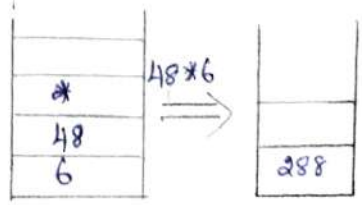
Read Character	Stack
E	Push E 
/	Push / 
−	Push − 
+	Push + 

The result of AB*CDE/-+ is 3

Example : 2

65 23 + 8 * + 3 + *

Read Character	Stack
6	Push 6 
5	Push 5 
2	Push 2 

Read Character	Stack
3	Push 3 
+	Push + 
8	Push 8 
*	Push * 
+	Push + 
3	Push 3 
+	Push + 
*	Push * 

The result of 6523+8*+3+* is 288

② Balancing Symbols:

→ The common mistake which can be made frequently by the programmer is unbalance of paranthesis. For correct representation of mathematical expression the following precautions must be taken.

(i) There must be equal number of left and right parenthesis.

(ii) Each left paranthesis must be balanced by right paranthesis.

→ In C right braces, brackets and paranthesis must correspond to their left counter parts.

⇒ [()] is legal
⇒ [(]) is wrong

Algorithm:

(i) Read characters until end of file.

(ii) If the character is an open paranthesis then push it on to the stack.

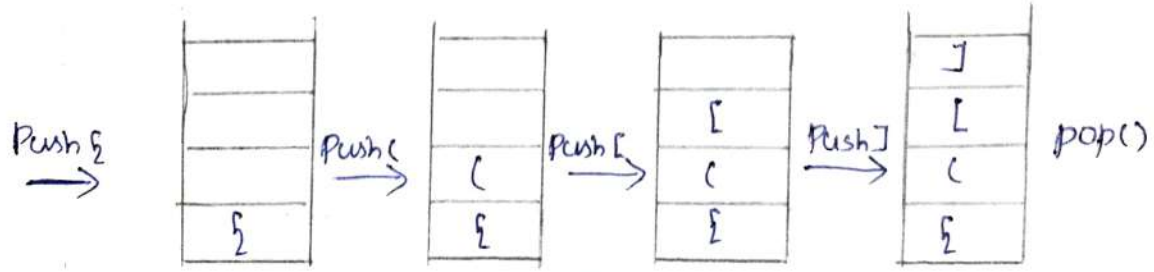
(iii) If it is a close paranthesis and if the stack is empty report an error otherwise pop the stack.

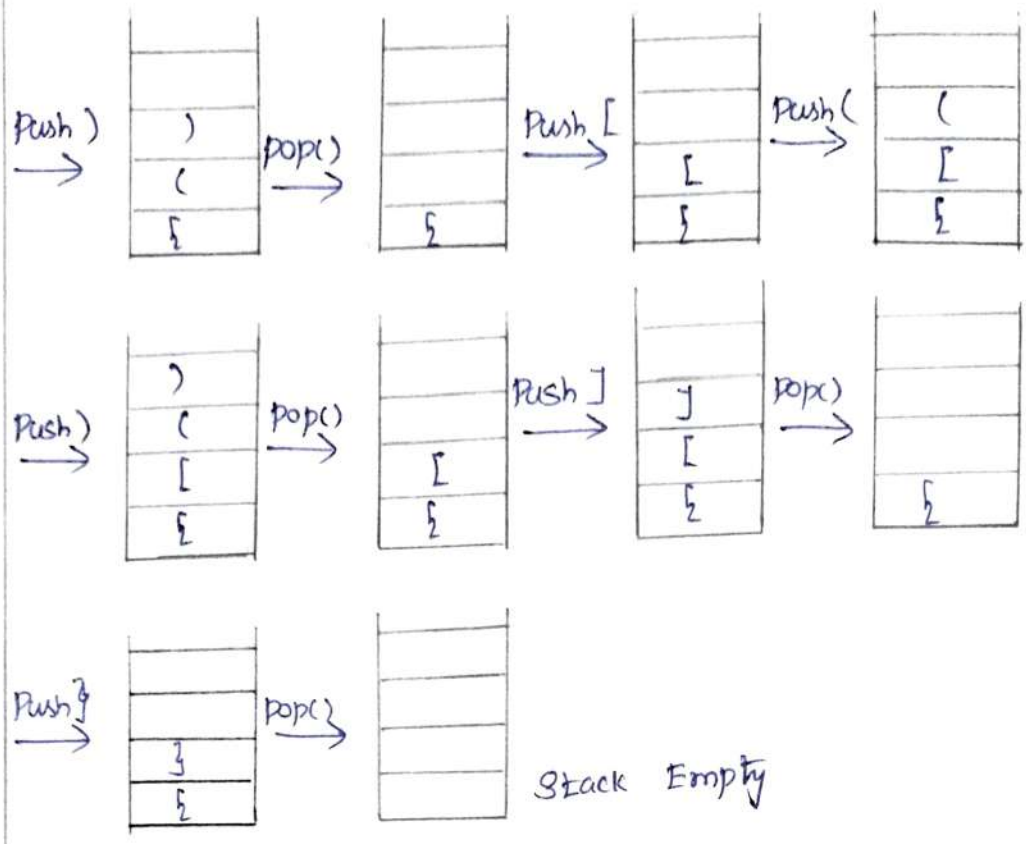
(iv) If the paranthesis is popped is not the corresponding opening paranthesis then report an error.

(v) At the end of file, if the stack is not empty report an error.

Example:

{ (P+Q − [M+N]) * J − [(x+y)] }

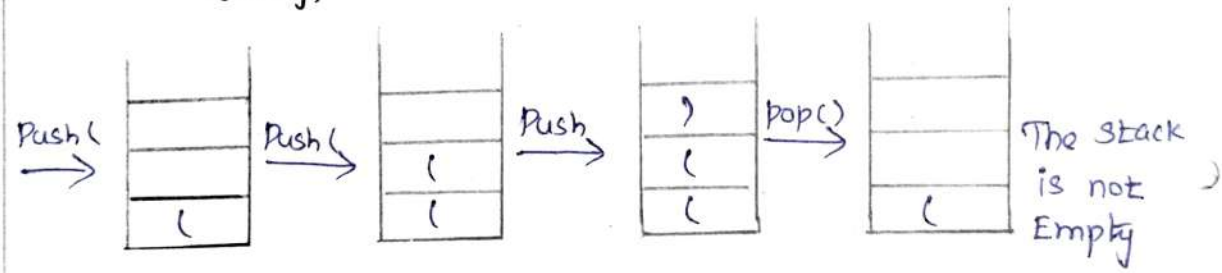




∴ The above expression is balanced.

Example : 2

((x+y)



∴ Unbalanced Expression.

③ Towers of Hanoi :

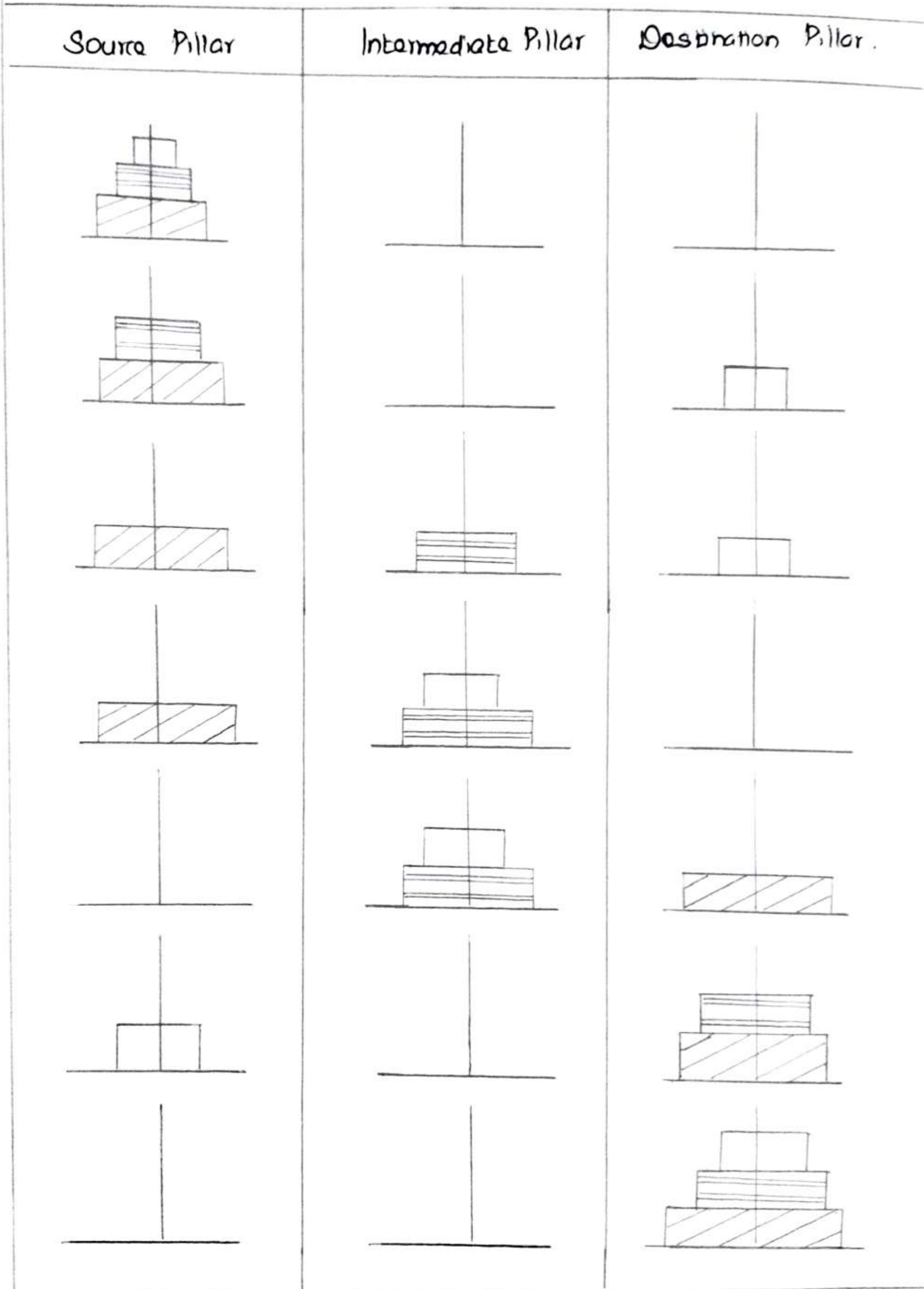
→ The problem is to move a collection of N disks of decreasing size from one pillar to another pillar.

Rules :

① Only one disk could be moved at a time.

② No larger disk could ever reside on a pillar on top of a smaller disk.

(3) The third pillar can be used as an intermediate to store one or more disks while they were being moved from source to destination.

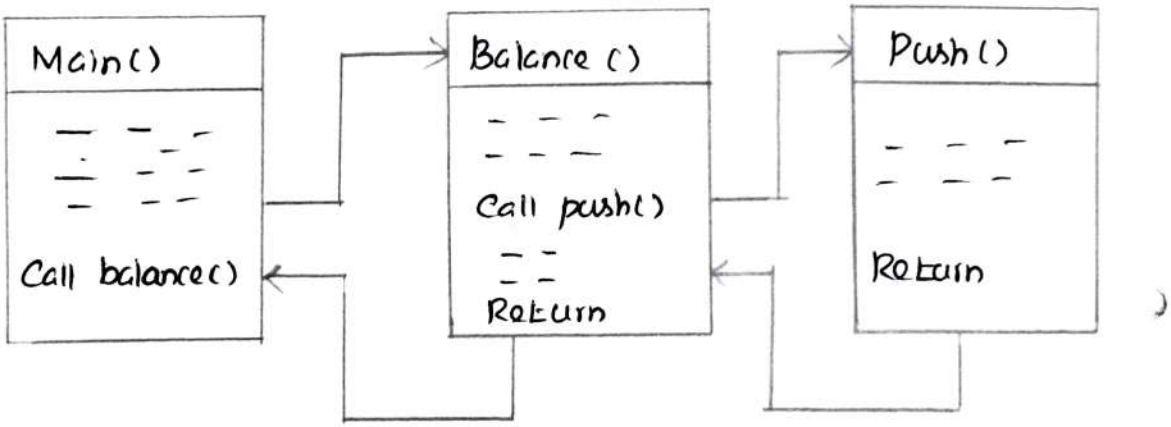


(4) Function calls:

→ A great use of stack is during the function call and return process. When we call a function from one another function that function call statement may not be the first statement. After calling the function we also have to come back from the function area to the place where we have left our control. So we want to resume our task not restart.

→ For that reason we store the address of the program counter into the stack, then go to the function body to execute it.

→ After completion of the execution it pops out the address from stack and assign it into the program counter to resume the task again.



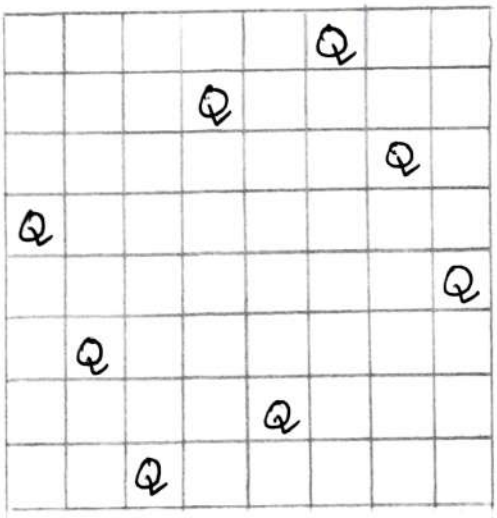

```
int fact (int n) {
    int s;
    if (n==1)
        return (1);
    else
        s = n * fact (n-1);
        return (s);
}
```

(5) 8 - Queens Problem:

→ The problem of placing eight chess queens on an 8x8 chessboard, so that no two queens threaten each other

→ A solution requires that no two queens share the same row, column or diagonal.

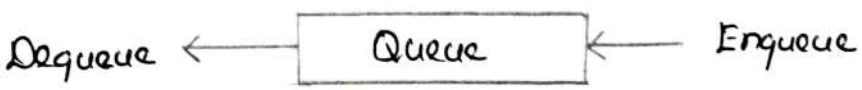
→ The 8-queen puzzle is an example of the more general n queens problem of placing n non attacking queens on an nxn chessboard for which solutions exists for all natural numbers n with the exception of $n=2$ and $n=3$



Queue ADT:

→ Queue is a linear data structure in which items are inserted at one end called "Rear" and deleted from the other end called "Front"

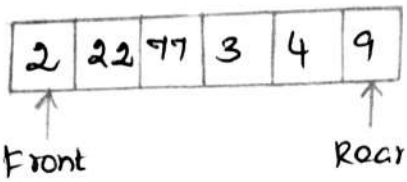
→ Queues are based on the First-In-First-Out (FIFO) principle that means the data items that is inserted first in the queue is also the first one to be removed from queue.



Logical Representation of Queues:

→ Each successive insert operation adds an element at the rear end of the queue while each delete operation removes an element from the front end of the queue.

→ The location of the front and rear ends are marked by two distinct pointers called front and rear.



Queue Operations:	Queue Implementation
(i) Enqueue (ii) Dequeue (iii) Overflow (iv) Underflow	(i) Array Implementation of Queue (ii) Linked List Implementation of Queue.

(1) Array Implementation of Queue:

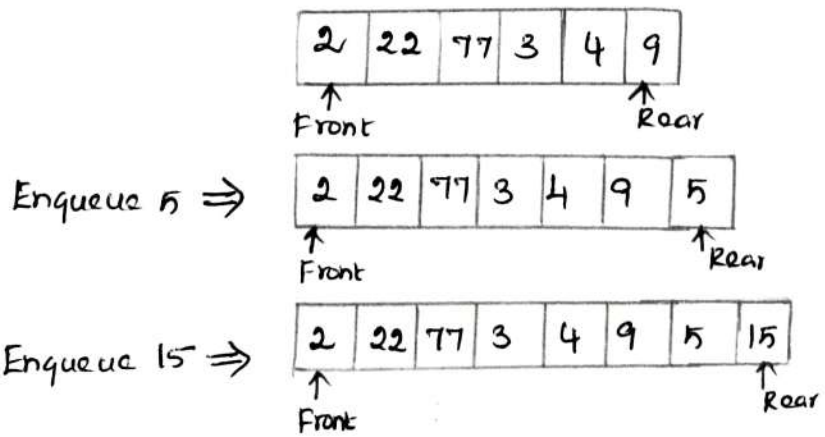
Enqueue:

→ Adding a new element

→ Receiving the element to be inserted.

→ Incrementing the queue pointer, rear

→ Storing the received element at new location of rear.

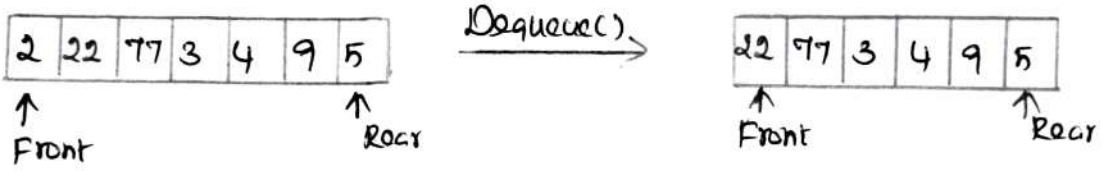


```
void enqueue (int item)
{
 if (front = = -1)
     front = front + 1;
 else
   {
    rear = rear + 1;
    Queue[rear] = item;
   }
}
```

Dequeue :

(i) Retrieving or removing the element from the end of queue.

(ii) Incrementing the queue pointer front, to make it point to the next element in the queue.




```
int dequeue()
{
  int item;
  item = Queue [front];
  front = front + 1;
  return item;
}
```

Overflow :

→ To insert an element into the queue it is necessary to check whether the queue is full or not. If the queue is full the element cannot be inserted.

```
int overflow()
{
 if (rear >= size -1)
     return 1;
 else
     return 0;
}
```

⇒ If Queue is overflow it will return the value 1

Underflow:

→ If the queue is empty it is called as underflow. If the queue is empty deletion operation cannot be performed.

```
int underflow()
{
  if((front == -1)|| (front > rear))
      return 1;
  else
      return 0;
}
```

Linked List Implementation of Queue:

.—Queues can be represented by singly Linked List. It is based on dynamic memory management techniques.

Advantages:

(i) Insertion and deletion can be handled easily and efficiently.

(ii) No wastage of memory space in Linked list representation.

(iii) Multiple queues can be represented easily.

Node Declaration:

```
struct node
{
  int element;
  struct node *Next;
}*front = NULL, *rear = NULL;
```

Create an Empty Queue:

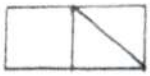
```
struct CreateQueue()
{
  Queue Q;
  Q = malloc (size of (struct Node));
  if (Q == NULL)
     Error("Out of Space");
     MakeEmpty (Q);
     return(Q);
}
```

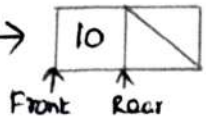
Make Empty:

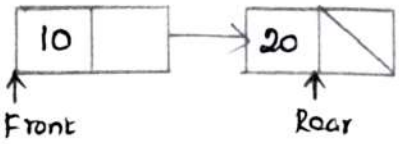
```
Void MakeEmpty (Queue Q)
{
  if (Q == NULL)
      Error ("Create Queue First");
  else
   {
     while (! IsEmpty (Q))
         Dequeue (Q);
   }
}
```

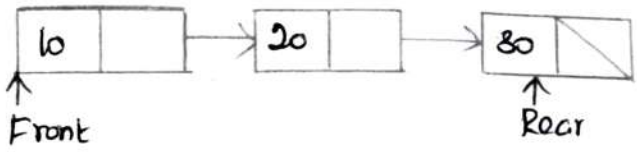
Enqueue:

→ to insert new element at rear end of the queue.

Empty Queue → 

Enqueue (10) → 

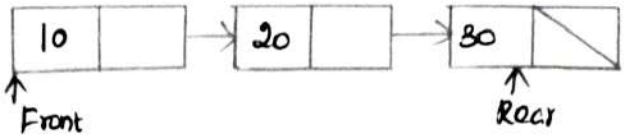
Enqueue (20) → 

Enqueue (30) → 

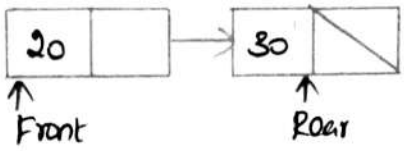

```
Void enqueue (int x)
{
  Struct Node *Newnode;
  Newnode = malloc(sizeof(Struct Node));
  If (rear == NULL)
  {
   Newnode -> Element = X;
   Newnode -> Next = NULL;
   front = Newnode;
   rear = Newnode;
  }
  else
  {
     Newnode -> Element = x;
     Newnode -> Next = NULL;
     rear -> Next = Newnode;
     rear = Newnode;
  }
}
```

Dequeue:

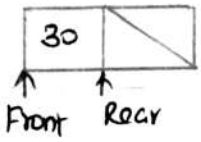
→ to delete an element from the front end of the queue.



Dequeue() →



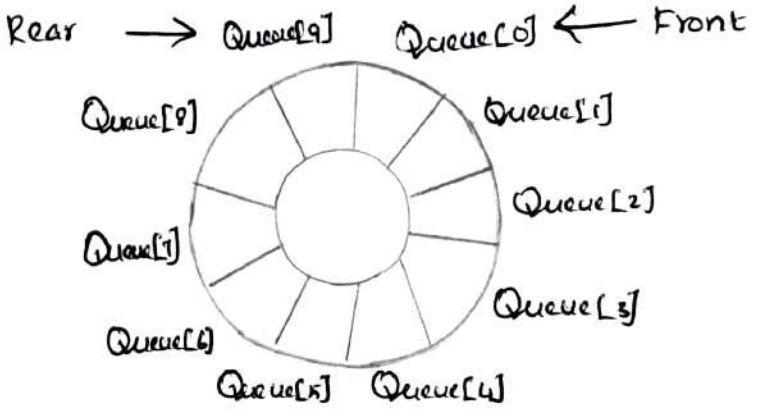
Dequeue() →

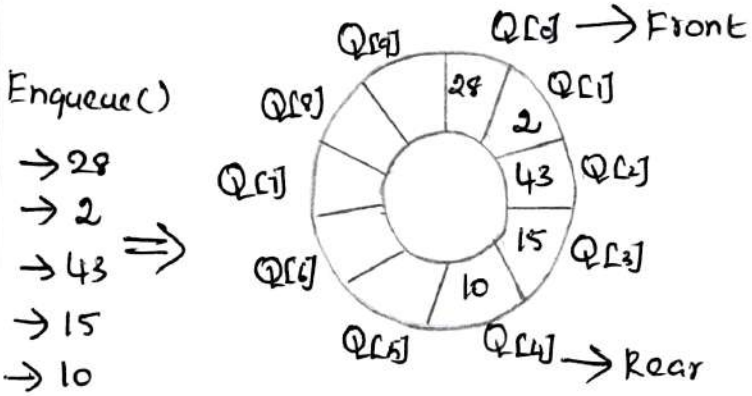


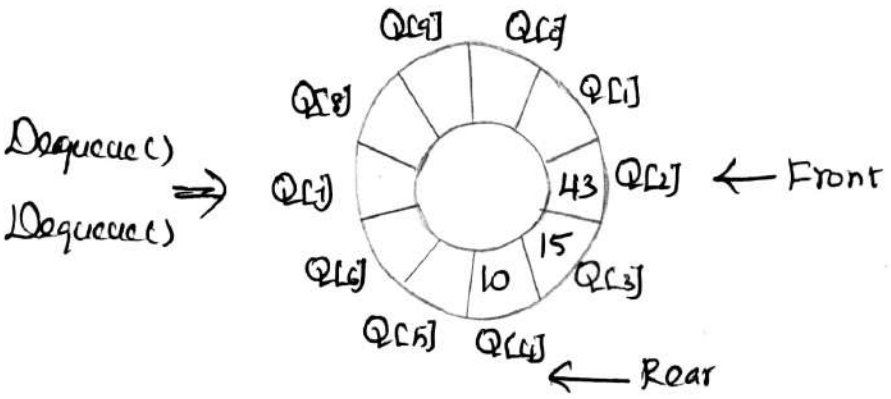

```
void Dequeue()
{
  struct Node *temp;
  if(front == NULL)
      Error("Queue is underflow");
  else
  {
    temp = front;
    if(front == rear)
    {
       return temp->element;
       front = NULL;
       rear = NULL;
    }
    else
    {
       temp = front;
       return temp->element;
       front = front->Next;
    }
  free(temp);
}
```

Circular Queue :

→ Circular queue is a queue whose start and end locations are logically connected with each other. That means the start location comes after the end location. If we continue to add elements in a circular queue till its end location then after the end location has been filled the next element will be added at the beginning of queue.







Enqueue:

→ To perform insertion of an element into the queue, the position of the element is calculated by the relation.

```
rear = (rear + 1) % maxsize ;
Queue [rear] = x;
```

Dequeue:

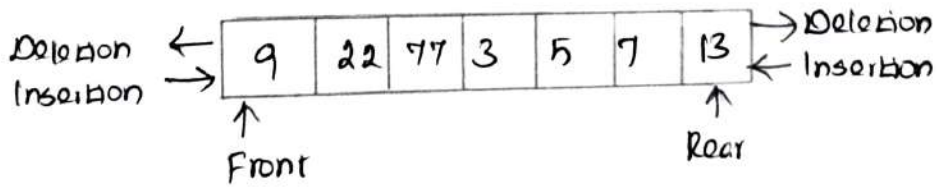
→ To perform deletion the position of the front pointer is calculated by the relation,

```
x = Queue [Front];
front = (front + 1) % maxsize ;
```

Double ended Queues: Or Deque:

→ A double ended queue is a special type of queue that allows insertion and deletion of elements at both ends i) front and rear.

→ A double ended Queue can be referred as a linear list of elements in which insertion and deletion of elements takes place at its two ends but not in the middle.



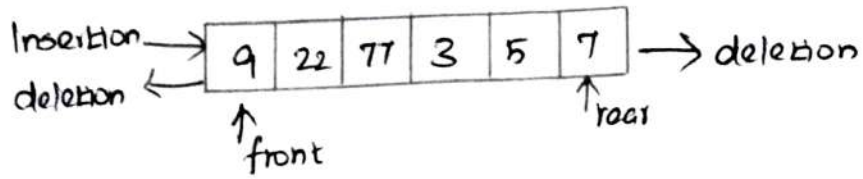
Types of deque:

(i) Input Restricted deque

(ii) Output Restricted deque

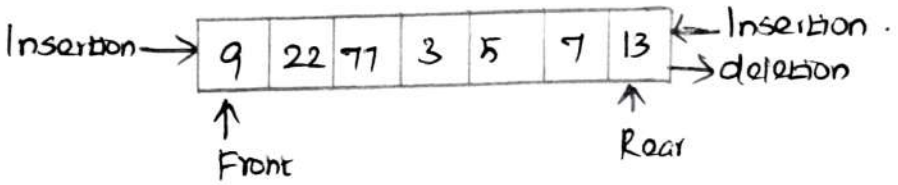
Input Restricted deque:

→ It allows deletion from both ends but restricts the insertion at only one end.



Output Restricted Deque:

→ It allows insertion at both the ends but restricts the deletion at only one end.



Priority Queues:

→ Priority Queue is a type of queues in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities.

Rules:

(i) An element with highest priority is deleted before all other elements of lower priority.

(ii) If two elements have the same priority then they are deleted as per the order in which they are added in to the queue (FIFO)

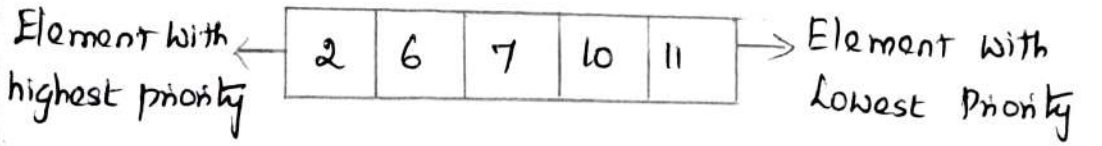
```
Struct queue
{
 int element;
 int priority;
 struct queue *NEXT;
};
```

Types of Priority Queue:

→ two types:

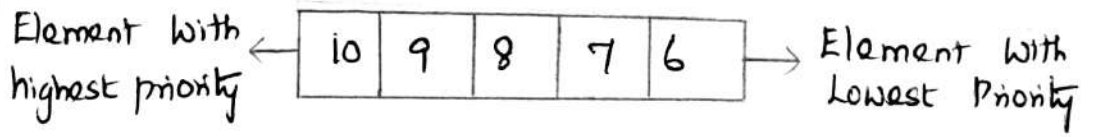
① Ascending Order Priority Queue:

→ In ascending order priority queue, a lower priority number is given as a higher priority in priority.



② Decending Order Priority Queue:

→ In decending order priority queue, a higher priority number is given as a higher priority in a priority.



Implementation of Priority Queue:

(i) Linked List Implementation of Priority Queue

(ii) Array Implementation of Priority Queue.

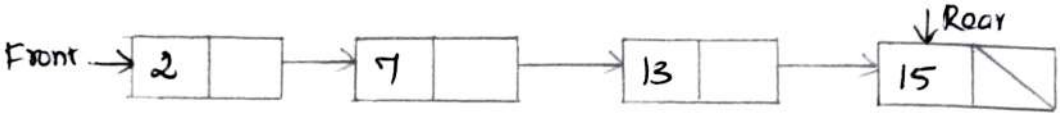
Linked List Implementation of Priority Queue:

→ The linked list of priority queue is created in such a way that the highest priority element is always added at the head of the queue.

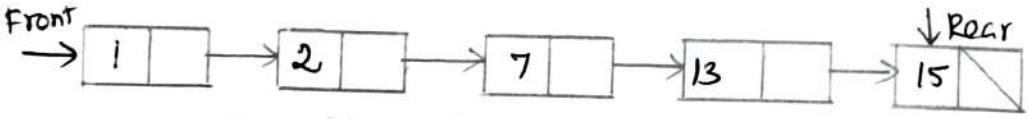
→ The elements are arranged in a decending order based on their priority.

⇒ Consider the linked list consists of elements

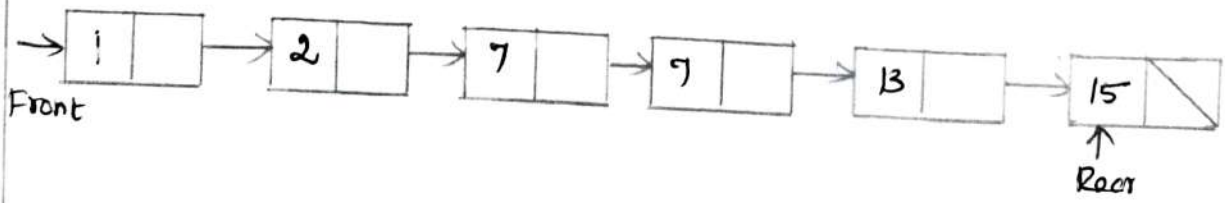
2, 7, 13, 15



⇒ Suppose we want to add the node that contains the value 1. Since the value 1 has more priority than the other nodes, so we insert the node at the beginning of list.



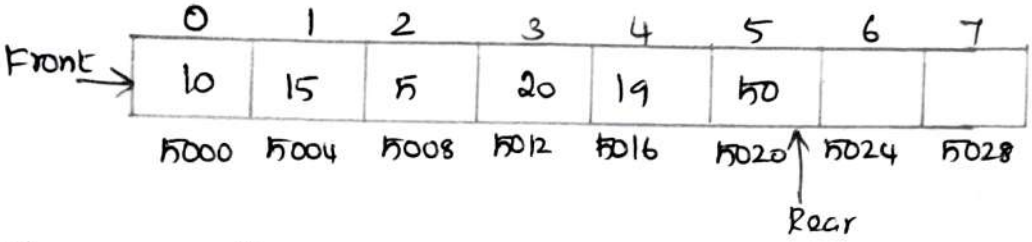
⇒ Now to add 7 element to the linked list. First element 7 is compared with one, since 7 has lower priority than one 1, the element 7 is compared with 2, since the element 7 has lower priority than 2. Now the element 7 is compared with element 7, both the elements has same priority. Then element 7 is added after the element 7.



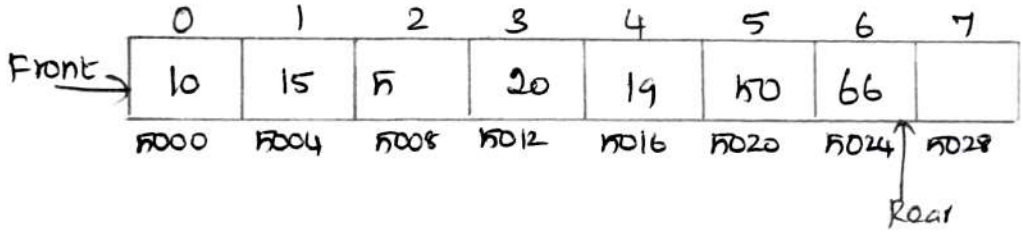
Array Implementation of Priority Queue:

→ Priority Queue implementation using an array is one of the basic methods to implement a queue.

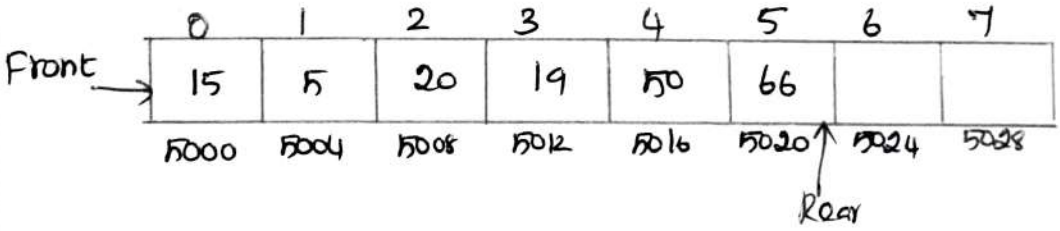
→ In this element is inserted and deleted based on its priority. The priority of the element is used to determine the order in which the elements will be processed.



Insert at the rear:



To delete a Front value:



Applications of Priority Queue:

(1) Executing Jobs in Operating System based on priority.

(2) Interrupt handling.

Applications of Queue:

① Computer Networks:

→ Many personal computers are connected to a single file server through several network setups. Users on other machines are accessing the files from the file server based on First Come First Served Basis. The convenient data structure for this purpose is the queue.

② Queuing Theory:

→ A branch of mathematics known as queuing theory deals with computing probabilistically how long users expect to wait on a line, how long the line gets and other such questions.

→ The answer depends on how frequently users arrive to the line and how long it takes to process a user once the user is served.

③ Simulation:

→ Problems are difficult to solve analytically are often solved by a simulation.

(i) Virtually every real life line is a queue.

(ii) When jobs are submitted to the printer they are arranged in order of survival.

(iii) In graph theory, several algorithms use queue data structure to give efficient running time.

④ Priority Queue :

→ Priority queue is a queue in which inserting an item or removing an item can be performed from any position based on some priority.

Queue Applications can be summarized as follows

(i) To implement a priority queues.

(ii) Priority queues can be used to sort the element using heap sort.

(iii) Simulation

(iv) Batch Processing in an operating system.

(v) Computer networks where the server takes the job of the client as per the queue strategy.

(vi) Mathematics uses queuing theory.

UNIT IV - NON-LINEAR DATA STRUCTURES

Trees - Binary Trees - Tree Traversals - Expression Trees - Binary Search Tree - Hashing - Hash Functions - Separate Chaining - Open Addressing - Linear Probing - Quadratic Probing - Double hashing - Rehashing.

Trees:

→ A tree is a data structure that represents hierarchical relationship between the individual data items.

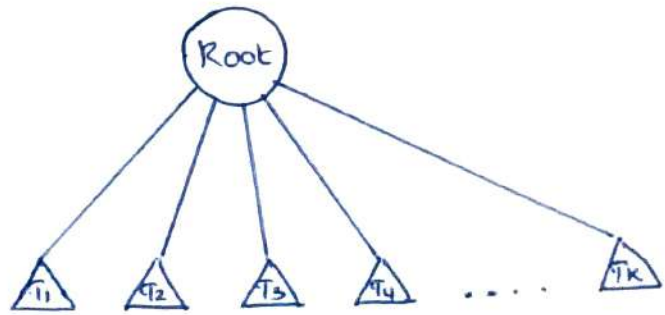
→ A tree is a collection of N nodes, one of which is a root and N-1 edges. There are N-1 edges follows from the fact that each edge connects some node to its parent and every node except the root has one parent.

Definition of tree:

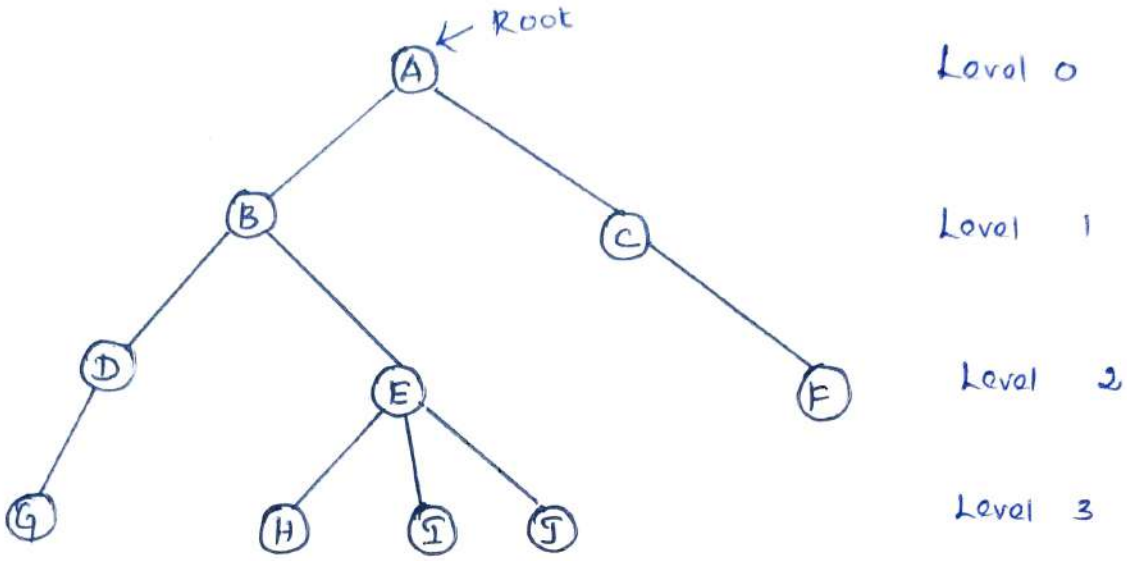
→ A tree is defined as a finite set of elements or nodes such that,

(i) One of the nodes present at the top of the tree is marked as root node.

(ii) The remaining elements are partitioned across multiple sub trees present below the root node.



Example :



Node :

→ It is a data element of a tree. Apart from storing a value it also specifies links to other nodes.

⇒ A, B, C, D, E

Root :

→ Node at the top of the tree is called root.

⇒ A.

Edge :

→ If a tree has N nodes then it will contain N-1 edges to connect the nodes.

⇒ Above example there are 10 nodes and hence 9 (10-1) edges will be used to connect all nodes.

Parent :

→ A node that has one or more child nodes present below is referred as parent node.

⇒ B is a parent of D and E

⇒ E is a parent of H, I and J.

Child:

→ All nodes in a tree except the root node are child nodes of their immediate predecessor node.

⇒ H, I, J are child nodes of E

Leaf:

→ It is a terminal node that does not have any child nodes.

⇒ G, H, I, J and F are leaf nodes.

Internal Node:

→ All nodes except root and leaf nodes are referred as internal nodes.

⇒ B, C, D, and E are internal nodes.

Sibling:

→ All the child nodes of a parent node are referred as siblings.

⇒ D and E are siblings.

Degree:

→ The degree of a node is the number of subtrees coming out of node.

⇒ Degree of A is 2

⇒ Degree of E is 3

Depth or Height:

→ It is the maximum level of a node in the tree.

⇒ Depth of tree is 3

Level:

→ All the tree nodes are present at different levels. Root node is at level 0, its child nodes are at level 1 and so on.

⇒ A is at level 0

⇒ B and C are at level 1

⇒ G, H, I, J are at level 3.

Path:

→ It is the sequence of nodes from source node till destination node.

⇒ A - B - E - J.

Ancestor and descendant nodes:

→ If there is a path from n_1 to n_2 then n_1 is an ancestor of n_2 and n_2 is a descendant of n_1.

⇒ A in an ancestor of B, C, D, E, J and so on.

⇒ E is an descendant of A.

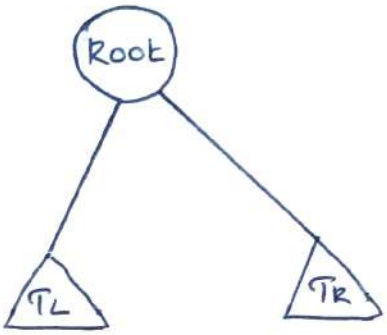
Node declaration for Trees:

```
struct Tree_Node
{
  Element_Type Element;
  Tree_Node  *FirstChild;
  Tree_Node  *NextSibling;
};
```

Binary Trees:

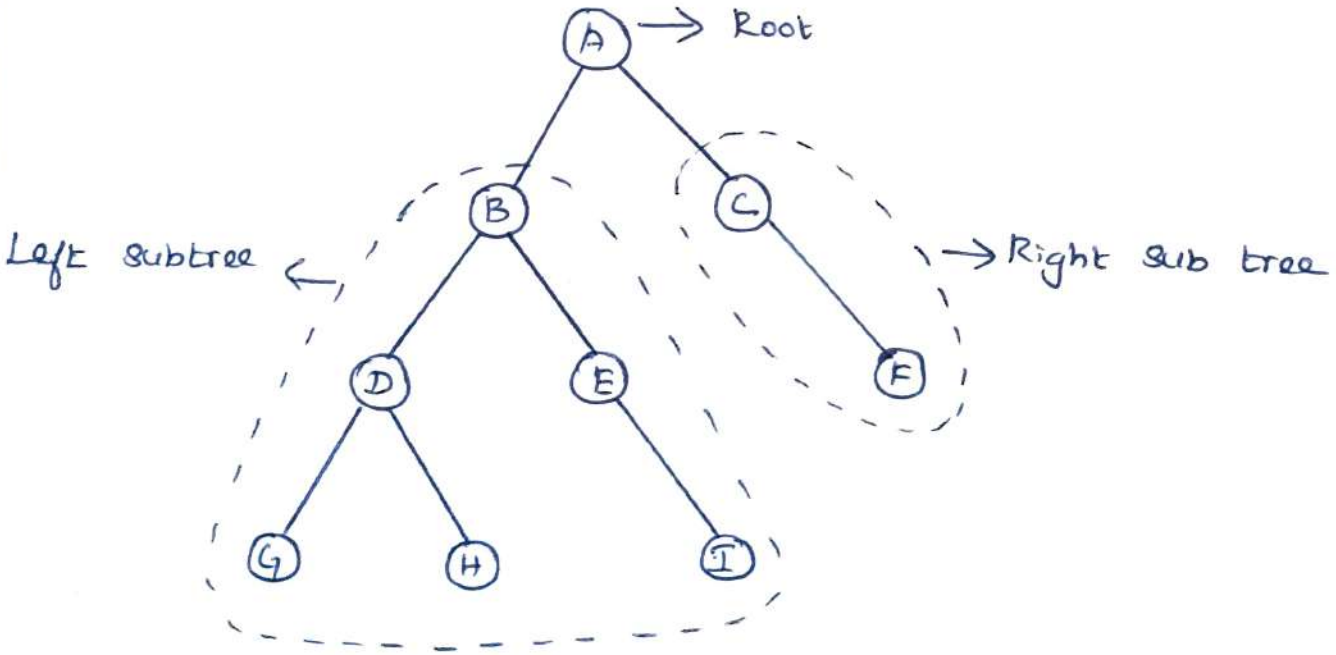
→ A binary tree is a tree that has nodes either empty or not more than two children.

→ A binary tree is either empty or consists of a node called the root together with two binary trees called the left subtree and right subtree.



→ In binary tree all nodes have a maximum degree of 2. The maximum number of nodes that can be present at level n is 2^n.

Example

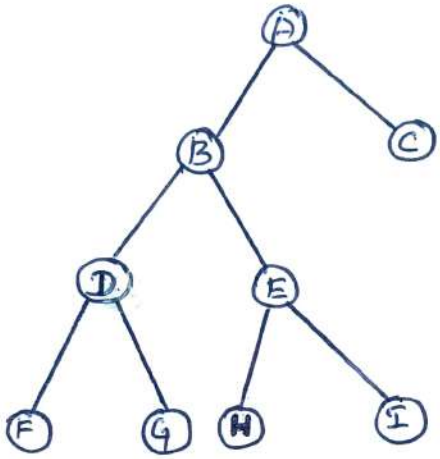


→ The nodes of a binary tree can have zero, one or two child nodes but not more than that.

Types of Binary Tree:

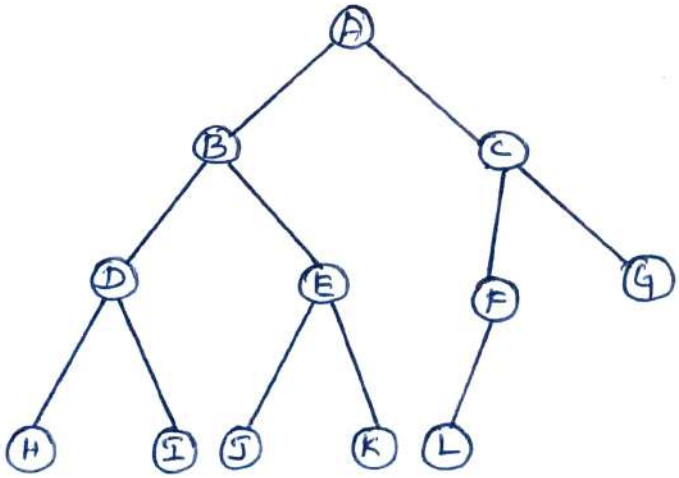
① Strictly Binary Tree:

→ A binary tree is called strictly binary if all its nodes barring the leaf nodes contain two child nodes.



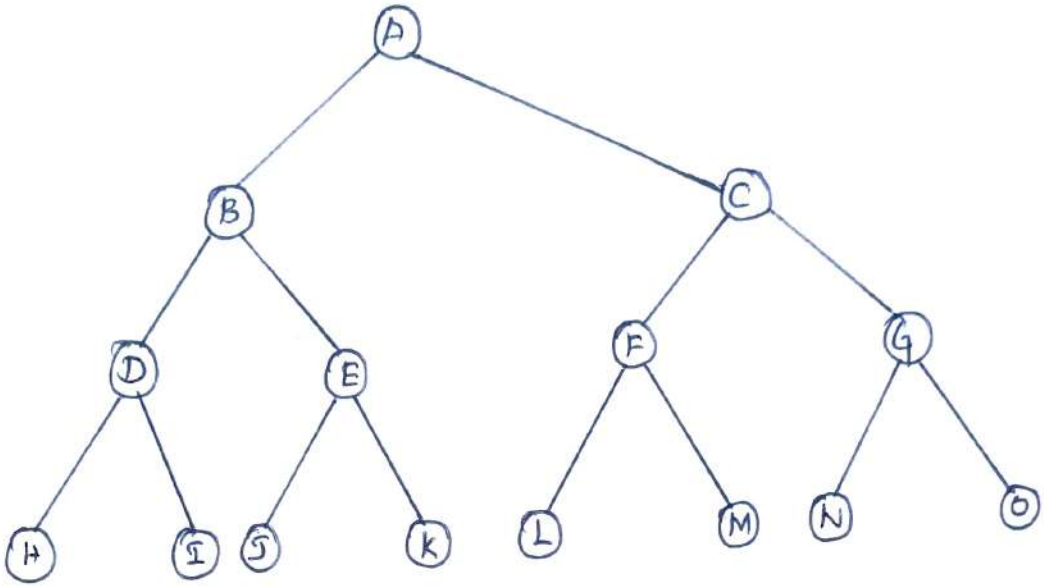
② Complete Binary Tree:

→ A binary tree of depth d is called complete binary tree if all its levels from 0 to d-1 contain maximum possible number of nodes and all the leaf nodes present at level d are placed towards the left side.



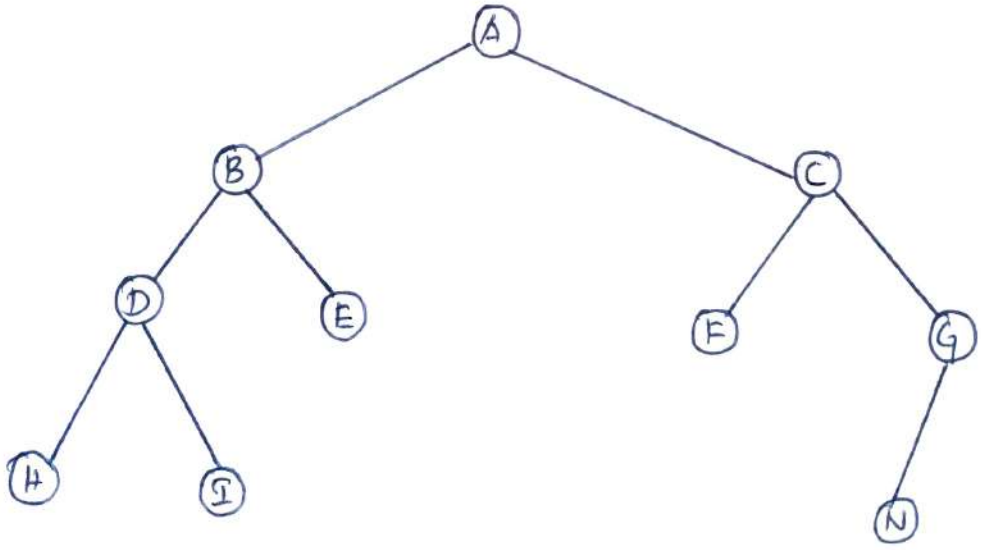
③ Perfect Binary Tree:

→ A binary tree is called perfect binary tree if all its leaf nodes are at the lowest level and all the non leaf nodes contain two child nodes.



④ Balanced Binary Tree:

→ A binary tree is called balanced binary tree if the depth of the subtrees all of its nodes do not differ by more than 1



Binary Tree Representation

(i) Array Representation

(ii) Linked Representation.

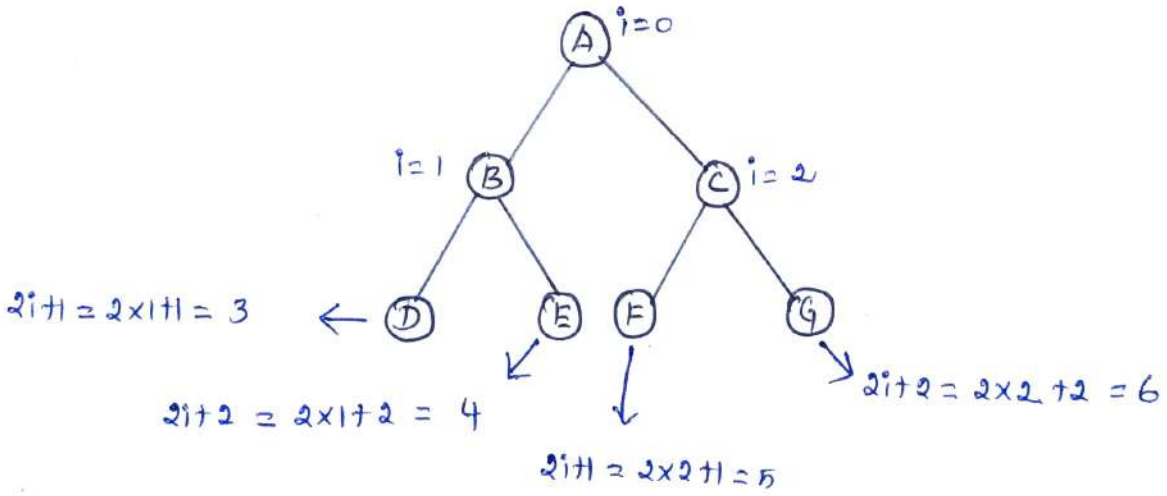
Array Representation:

→ In array representation of binary trees one dimensional array is used for storing the node elements.

Rules:

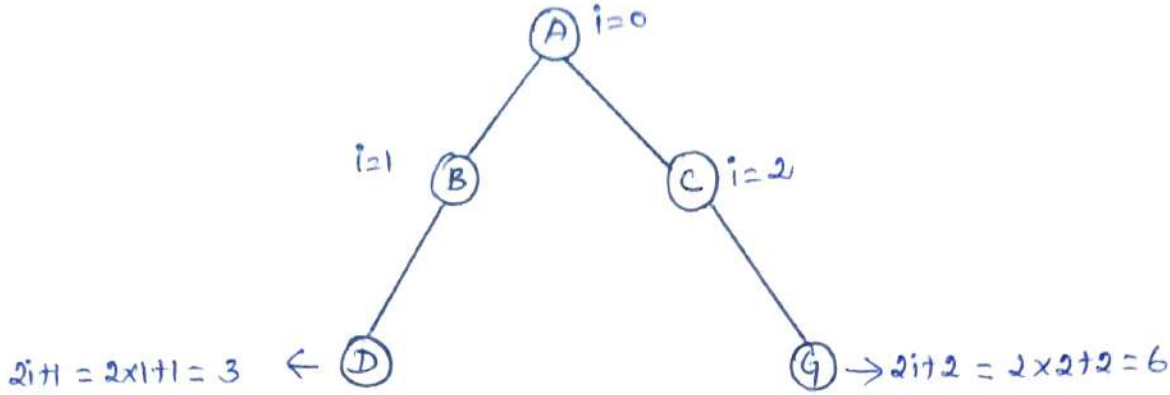
(i) The root node is stored at the first position in the array while its left and right nodes are stored at the successive positions.

(ii) If a node is stored at index location i then its left child node will be stored at location $2i+1$ while the right child will be stored at location $2i+2$



Array A ⇒

A	B	C	D	E	F	G
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]



Array A ⇒

A	B	C	D	-	-	G
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]

Linked Representation:

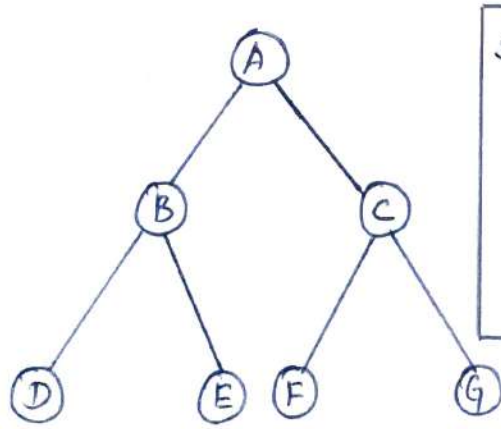
→ To avoid the disadvantages associated with array representation, linked representation is used for implementing binary trees.

→ It uses a linked list for storing the node elements. Each tree node in represented with the help of linked list node comprising the following fields.

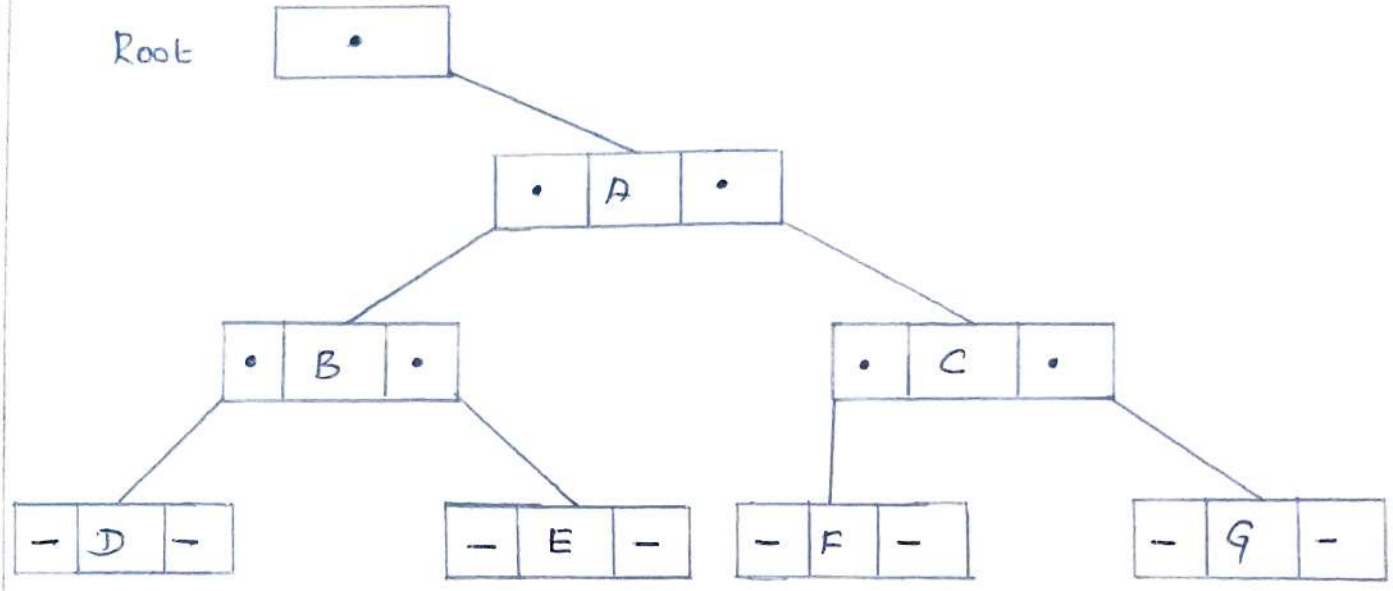
(1) INFO stores the value of tree node.

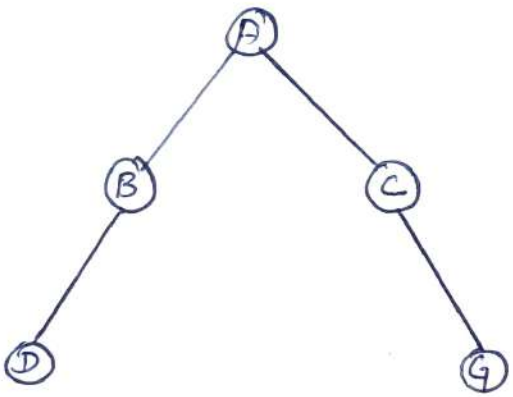
(2) LEFT stores a pointer to the left child.

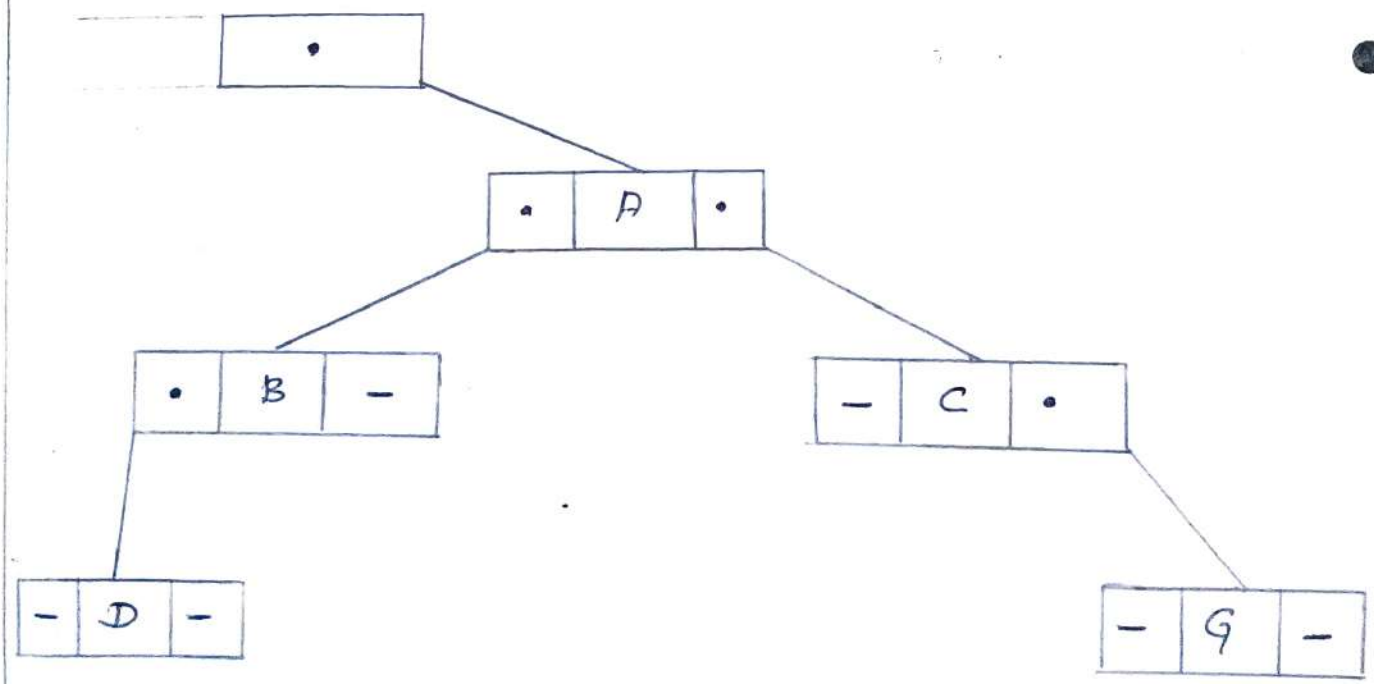
(3) RIGHT stores a pointer to the right child.




```
STRUCT TreeNode
{
   ElementType Element;
   Tree *Left;
   Tree *Right;
};
```


Root
A
B
C
D
E
F
G


A
B
C
D
G


A
B
C
D
G

Tree Traversals:

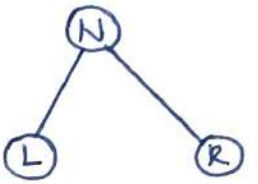
→ Tree Traversal is a procedure that systematically visits every vertex of an ordered rooted tree.

→ Three types

(i) Pre Order Traversal

(ii) In order Traversal

(iii) Post order Traversal



Pre order Traversal:

(i) Process the root node

(ii) Traverse the left sub tree of N(L)

(iii) Traverse the right sub tree of N (R)

Pre order ⇒ N-L-R

```
Void preorder (Tree T)
{
 if (T != NULL)
   {
    printf ("%d", T -> data);

    preorder (T -> left);
    preorder (T -> right);
   }
}
```

In order Tree Traversal:

(i) Traverse the left subtree of N(L)

(ii) Process the root node

(i) Traverse the right subtree of N(R)

Inorder ⇒ L - N - R

```
void inorder (Tree T)
{
  if (T != NULL)
    {
      inorder (T->left);
      printf ("%d", T->data);
      inorder (T->right);
    }
}
```

Post order Tree Traversal:

(i) Traverse the left subtree of N(L)

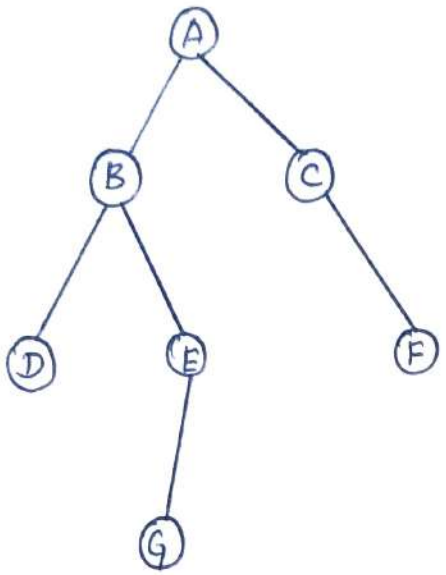
(ii) Traverse the right subtree of N(R)

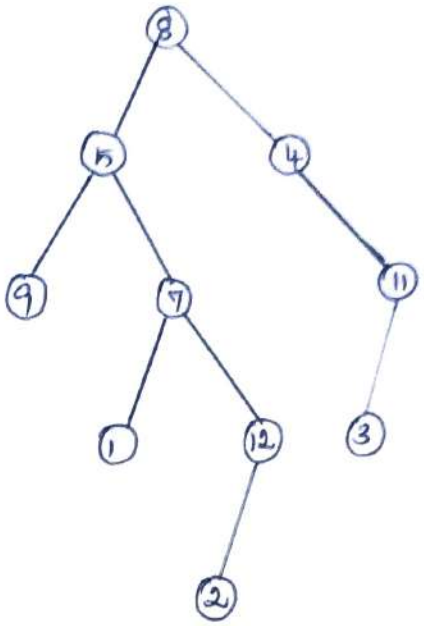
(i) Process the root node (N)

Post order ⇒ L - R - N

```
Void postorder (Tree T)
{
  if (T != NULL)
    {
      postorder (T->left);
      postorder (T->right);
      printf ("%d", T->data);
    }
}
```

Example





Preorder ⇒ A, B, D, E, G, C, F
Inorder ⇒ D, B, G, E, A, C, F
Postorder ⇒ D, G, E, B, F, C, A

Preorder ⇒ 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
Inorder ⇒ 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
Postorder ⇒ 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

Expression Trees:

→ Expression tree is nothing but a binary tree containing mathematical expression.

→ The internal nodes of the tree are used to store operators while the leaf or terminal nodes are used to store operands.

→ Expression tree is a binary tree in which the leaf nodes are operands and the interior nodes are operators.

→ Expression tree can be constructed from an postfix expression.

Steps:

① Read one symbol at a time from postfix expression

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 stack.

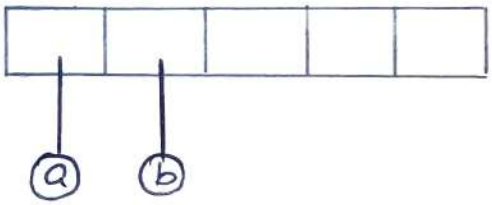
b) If the symbol is an operator pop two pointers from the stack namely T_1 and T_2 and form a new node tree with root as the operator and T_2 as a left child and T_1 as a right child. A pointer to this new tree is then pushed on to the stack.

Example:

ab + cde + * *

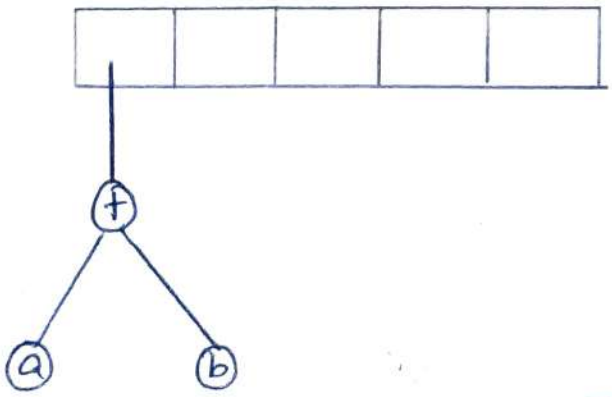
Step 1:

→ The first two symbols are operands, so create one node tree and push pointers to them on to the stack.



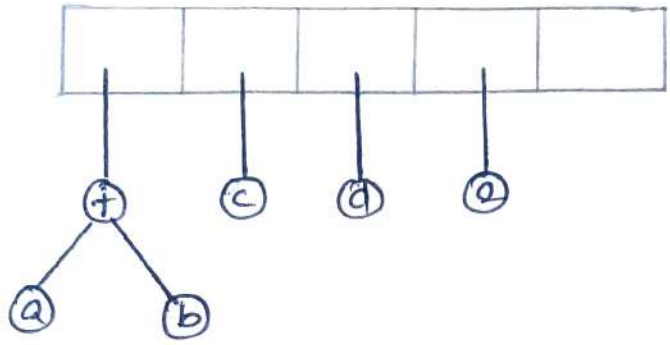
Step 2:

→ Next '+' is read, so two pointers to trees are popped a new node tree is formed and a pointer to it is pushed on to the stack.



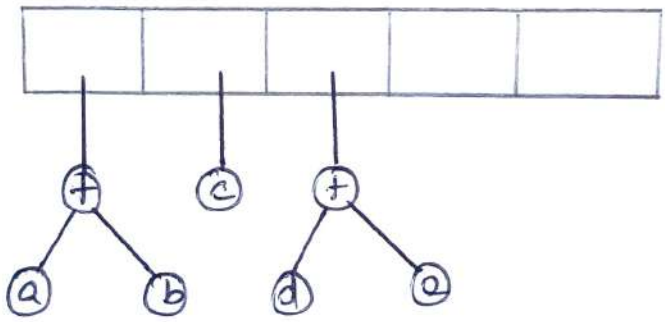
Step 3:

→ Now c, d, e are read and for each one node tree is created and a pointer to the corresponding tree is pushed onto the stack.



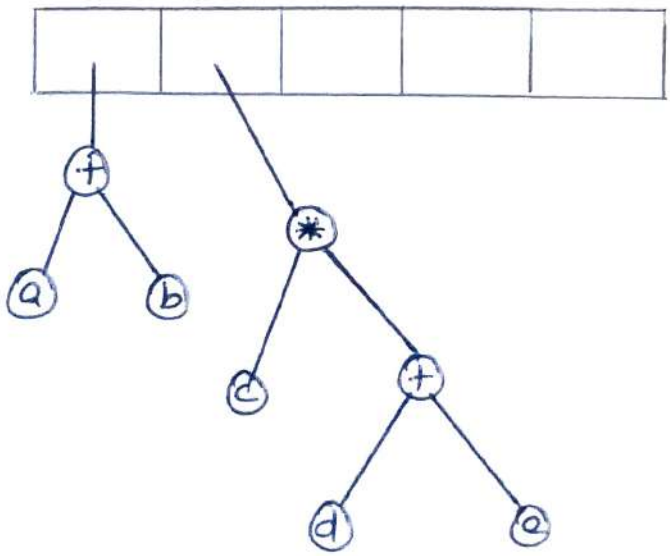
- Step 4:

→ Now '+' read so two trees are merged.



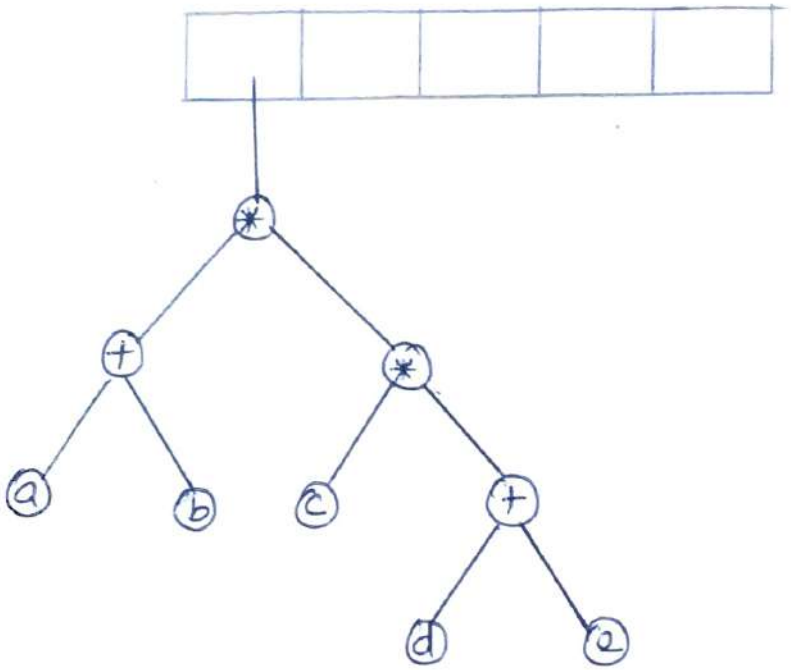
- Step 5:

→ Now '*' is read, so pop two tree pointers and form a new tree with a '*' as a root.



Step 6:

→ Now last symbol '*' is read, two trees are merged and a pointer to the final tree is left on the stack.



Binary Search Tree:

→ A binary tree is referred to as binary search tree if for any node n in the tree,

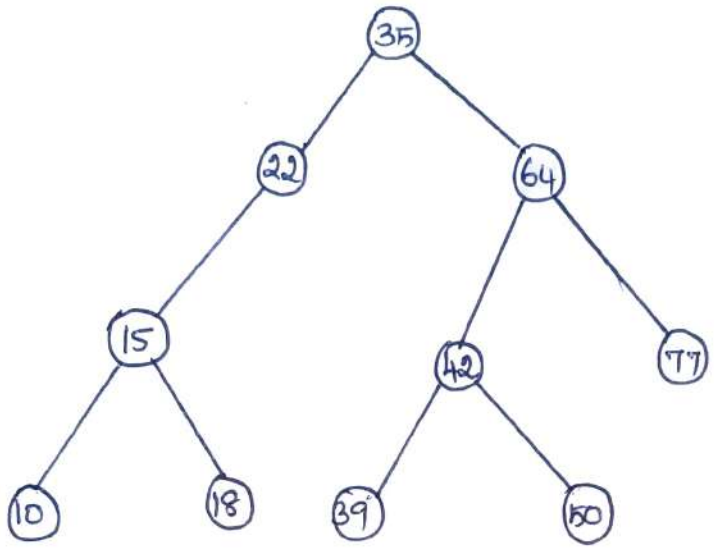
(i) The node elements in the left subtree of n are lesser in value than n

(ii) The node elements in the right subtree of n are greater than or equal to n.

→ Binary search tree arranges its node elements in a sorted manner. As the name suggests the most important application of a binary search tree is searching.

→ The average running time of searching an element in a binary search tree is O(log n) which is better than other data structures like array and linked list.

Example:



- Node Declaration:

```
Struct TreeNode
{
  ElementType Element;
  SearchTree Left;
  SearchTree Right;
};
```

- Operations of binary search Tree:

(i) Make Empty

(ii) Insertion

(iii) Deletion

(iv) Find

(v) Find Min

(vi) Find Max

Make Empty:

→ Make empty operation is used for initialization.

```
SearchTree MakeEmpty (SearchTree T)
{
  if (T != NULL)
  {
    MakeEmpty (T->Left);
    MakeEmpty (T->Right);
    free(T);
  }
 return NULL;
}
```

Find:

→ This operation will find key x in the binary search tree. If key is found then pointer to that node is returned. If the key is not found it will return NULL

① If T is NULL then just return NULL

② If the key stored at T is equal to x then return T. Otherwise make a recursive call on a subtree of T, either left or right depending on the value stored in T.

```
Position Find ( ElementType X, SearchTree T)
{
  if ( T == NULL)
      return NULL;
  if ( X < T -> Element)
      return Find (X, T->Right);
      else
         return T;
  }
```

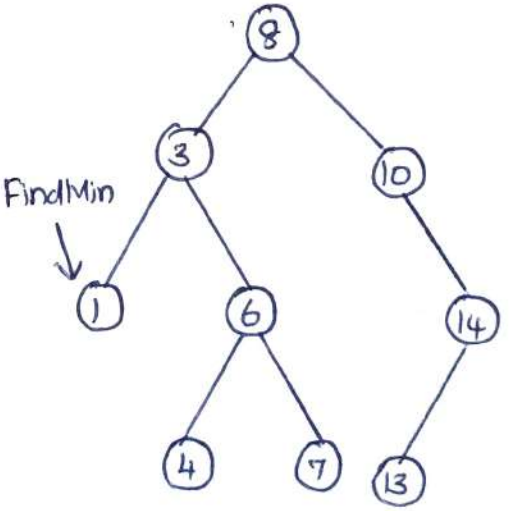
Find Min:

→ This routine return the position of smallest element in the tree.

(or) To perform a findmin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.

```
Position FindMin (SearchTree T)
{
  if ( T == NULL)
      return NULL;
  else
   if ( T -> Left == NULL)
      return T;
  else
     return FindMin ( T -> Left);
}
```

Example:



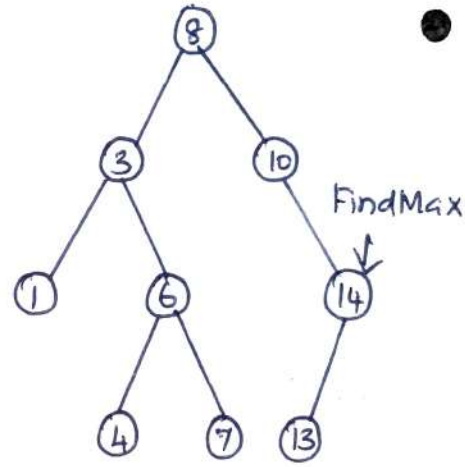
FindMax:

→ This routine return the position of largest element in the tree.

(i) To perform a findMax, start at the root node and go right as long as there is a right child. The stopping point is the largest element.

```
Position FindMax (SearchTree T)
{
  if (T == NULL)
      return NULL;
  else
    if (T->Right == NULL)
       return T;
    else
       return FindMax (T->Right);
}
```

Example:



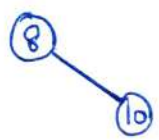
Insertion:

→ This involves adding an element in to the binary tree. The location of new element is determined in such a manner that insertion does not disturb the sort order of the tree.

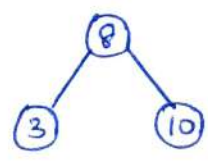
Example: 8, 10, 3, 2, 18, 6, 5, 13, 11

Insert 8

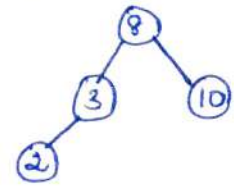
Insert 10



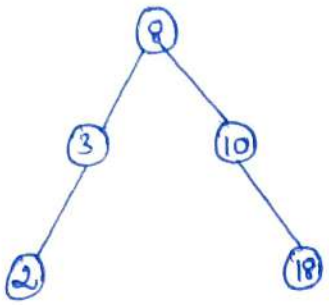
Insert 3



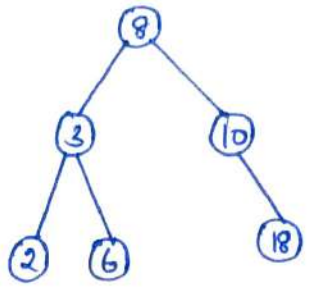
Insert 2



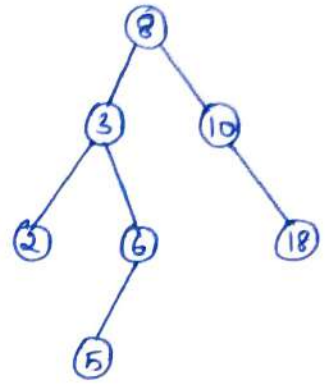
Insert 18



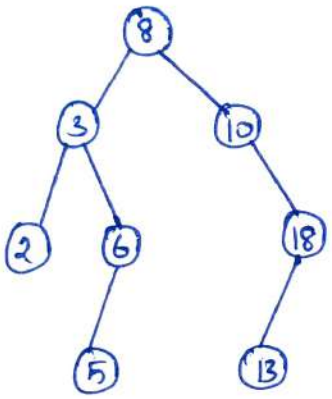
Insert 6



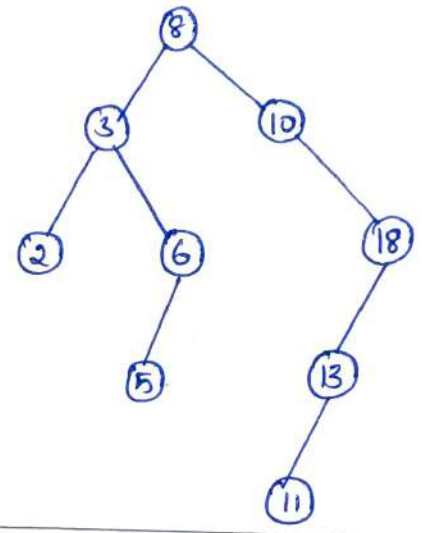
Insert 5



Insert 13



Insert 11



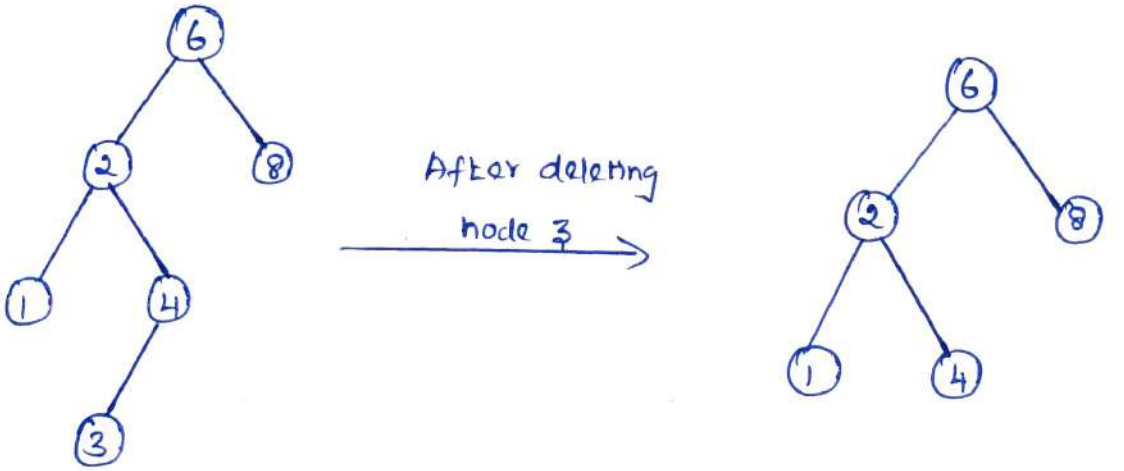

```
SearchTree Insert (ElementType x, SearchTree T)
{
 if (T == NULL)
   {
    T = malloc (sizeof (struct TreeNode));
    if (T == NULL)
        fatalError ("Out of space!!");
    else
      {
        T -> Element = x;
        T -> Left = NULL;
        T -> Right = NULL;
      }
```

```
    else
    {
        if (X > T -> Element)
            T -> Right = Insert (X, T -> Right);
        if ( X < T -> Element)
            T -> Left = Insert (X, T -> Left);
        return T;
    }
}
```

Deletion :

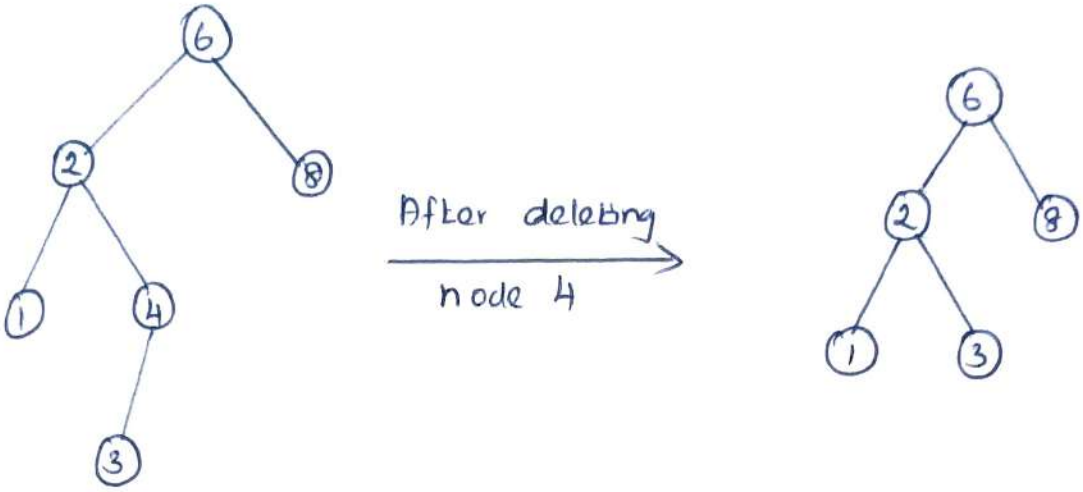
Case 1 : Node with no children.

→ If the node is a leaf, it can be deleted immediately.



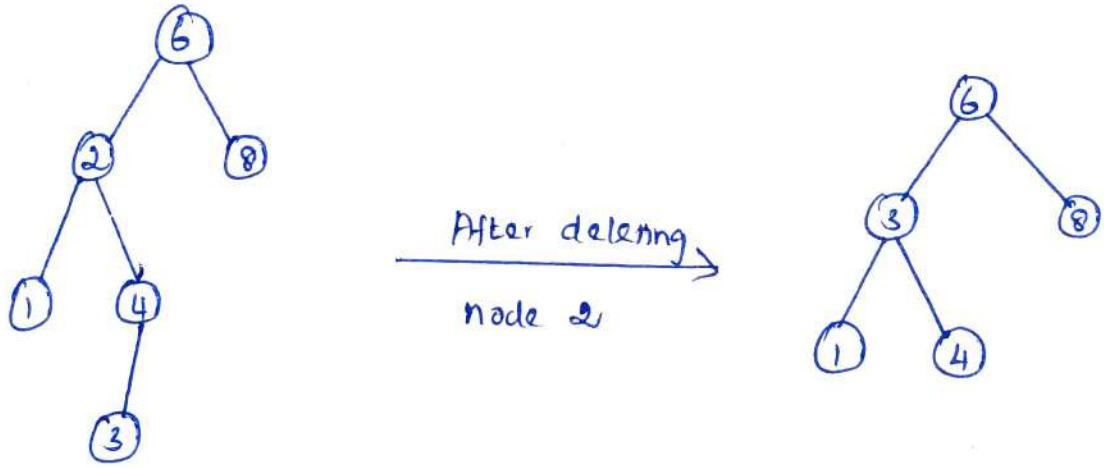
Case 2 : Node with one child.

→ If the nodes has one child, the node can be deleted after its parent adjusts a pointer to bypass the node.



Case 3 : Node with two children.

→ Find node with smaller value in the right subtree or find node with bigger value in the left subtree. Replace the node in the place of deleted node.




```
SearchTree Delete (ElementType X , SearchTree T)
{
  Position TmpCell ;
  If (T == NULL)
     Error ("Element not Found");

  else
     if (X < T -> Element)
        T -> Left = Delete (X, T -> Left);
```

```
else
 if ( X > T -> Element )
    T -> Right = Delete ( X, T -> Right );
else
 if ( T -> Left && T -> Right )
 {
   TmpCell = FindMin ( T -> Right );
   T -> Element = TmpCell -> Element;
   T -> Right = Delete ( T -> Element, T -> Right );
 }
 else
  {
    TmpCell = T;
     if ( T -> Left == NULL )
          T = T -> Right;
     else
        if ( T -> Right == NULL )
           T = T -> Left;
       free ( TmpCell );
  }
 return T;
}
```

Hashing:

→ Hashing finds the location of an element in a data structure without making any comparisons.

→ Hashing uses a mathematical function to determine the location of an element. This mathematical function is called hash function.

→ The hash function accepts a value known as key as input and generates an output known as hash key.

→ The hash function generates hash keys and stores element corresponding to each hash key in the hash table.

→ A hash table is a data structure which is implemented by a hash function and used for searching elements in quick time.

→ In a hash table the hash key acts as the addresses of elements.

Key (x)	hash function value h(x)
100	0
101	1
102	2
103	3
104	4
105	5
106	6
107	7

Hash Table:

→ Hash table is a data structure used for storing and retrieving data quickly.

→ Every entry in the hash table is made using hash function.

Hash Function:

→ Hash function is a function used to place data in hash table.

→ Hash function is used to retrieve data from hash table.

Hash Function h(key) = key mod hash table size

Example:

Consider a hash table size of 5 and the elements are 50, 26, 38, 24. Place the elements in the hash table.

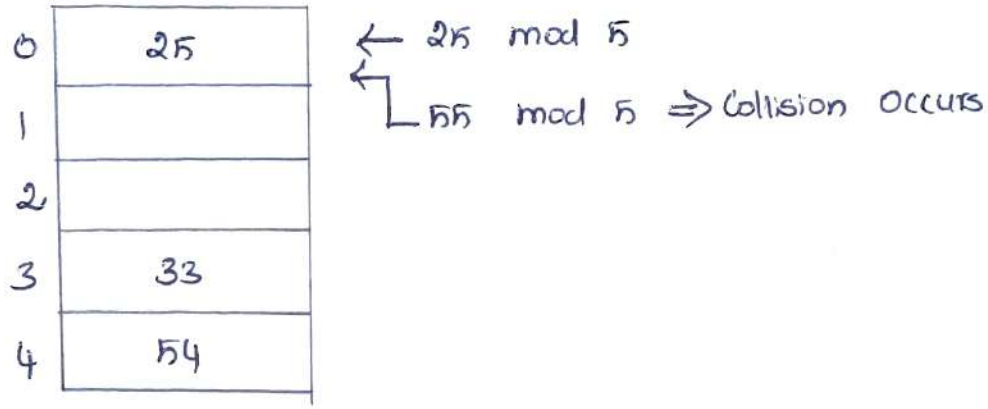
50 mod 5 →	0	50
26 mod 5 →	1	26
	2	
38 mod 5 →	3	38
24 mod 5 →	4	24

Bucket:

→ The hash function is used to map several dictionary entries in the hash table. Each position of the hash table is called the bucket.

Collision:

→ Collision is the situation in which hash function returns the same address for more than one record.



Probe:

→ Each calculation of an address and test for success is known as probe.

Synonym:

→ The set of keys that has to the same location are called synonym.

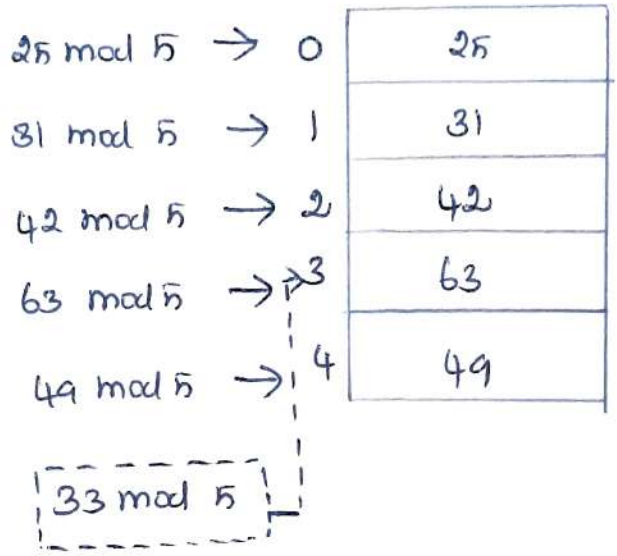
→ For example, in above given hash table computation 25 and 55 are synonym.

Perfect hash Function:

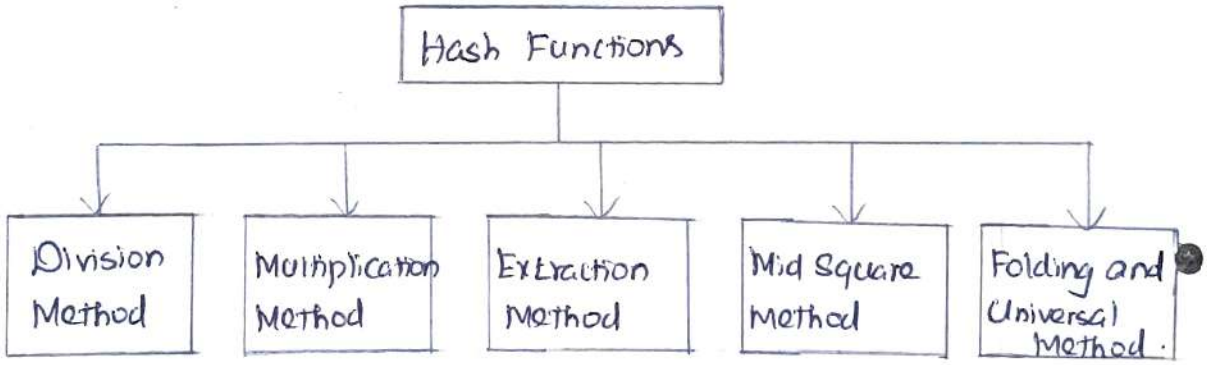
→ The perfect hash function is a function that maps distinct key elements in to the hash table with no collisions.

Overflow:

→ When hash table becomes full and new record needs to be inserted then it is called overflow



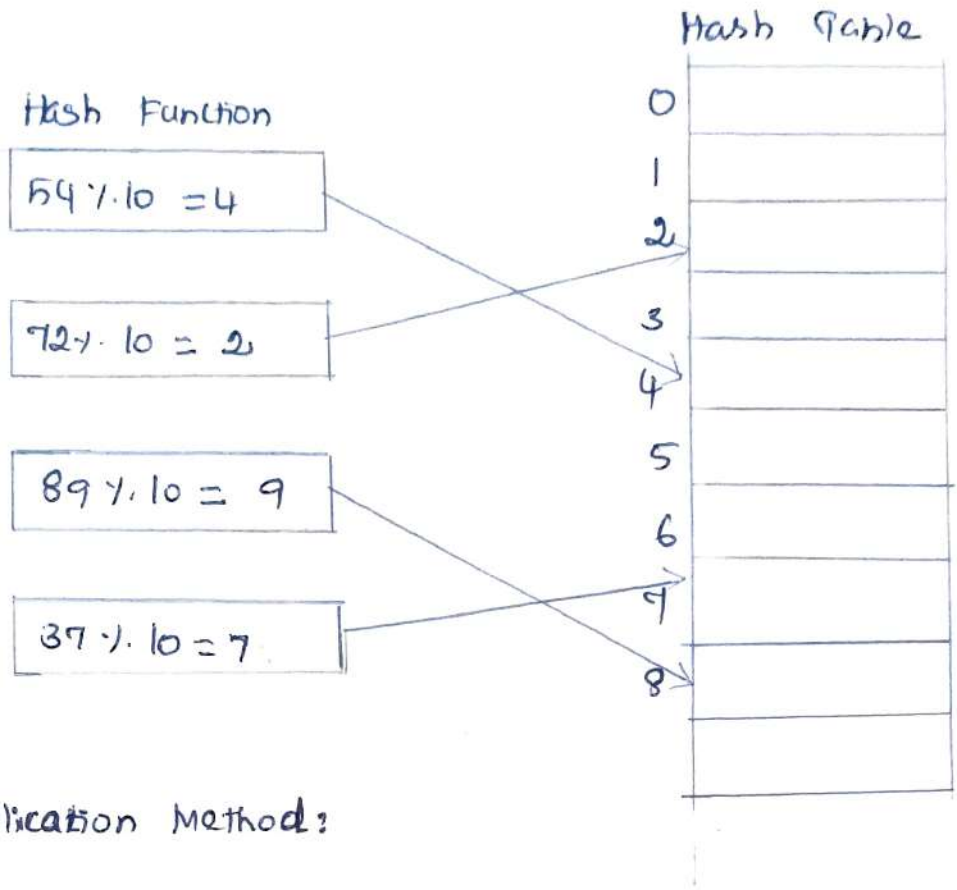
Hash Functions:



Division Method:

→ The hash function depends upon the remainder of division.

→ If the record 54, 72, 89, 37 is to be placed in the hash table and if the table size is 10 then,



Multiplication Method:

Steps:

(i) Multiply the key 'k' by a constant A where A is in the range $0 < A < 1$. Then extract the fractional part of kA.

(ii) Multiply this fractional part by m and take the floor.

→ The above steps can be formulated as

$$h(k) = \lfloor m \{kA\} \rfloor$$

$\{kA\}$ → fractional part.

→ Donald knuth suggested to use $A = 0.61803398987$

Example:

key $k = 107$, $m = 50$, $A = 0.61803398987$

$$h(k) = \lfloor m * (k * 0.61803398987) \rfloor$$

$$h(k) = [50 * (107 * 0.618033988987)]$$

$$= [50 * (66.12)]$$

$$= [50 * (0.12)]$$

↳ Fractional Part

$$h(k) = 50 * 0.12$$

$$h(k) = 6$$

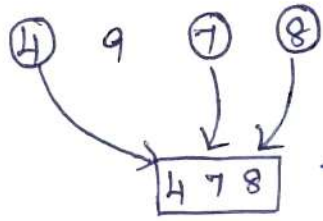
∴ 107 will be placed at index 6 in hash table.

Extraction Method:

→ In this method some digits are extracted from the key to form the address location in hash table.

Example:

Suppose first, third and fourth digit from left is selected for hash key.



→ at 478 location in the hash table of size 1000 the key can be stored.

Mid Square Method:

Steps:

(i) Square the key.

(ii) Extract middle part of result. This will indicate the location of the key element in hash table.

→ Note that if the key element is a string then it has to be preprocessed to produce a number.

Example:

key = 3111

$(3111)^2$

⇓

9 6 (7 8 3) 2 1

For the hash table of size 1000

H(3111) = 783

Folding Method:

→ Two techniques

(i) Fold shift (ii) Fold Boundary.

Fold shift:

→ In this method the key is divided into separate parts whose size matches, with the size of required address.

→ Then left and right parts are shifted and added with the middle part.

key k = 345678123

(345) 678 (123)

⇓

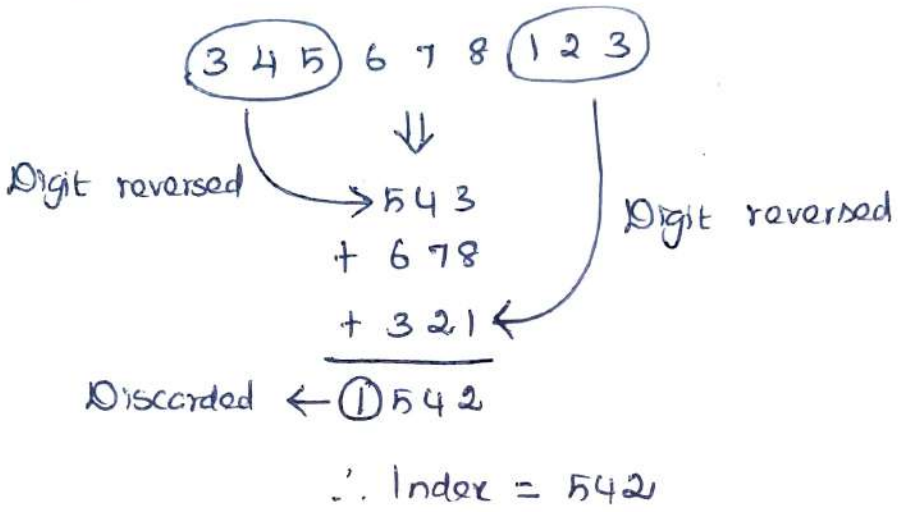
345
+ 678
+ 123

Discarded ← (1)146 ∴ Index = 146

Fold Boundary:

→ In this method the key is divided into separate parts. The left most and right most parts are folded on fixed boundary and added with the middle part

key k = 3 4 5 6 7 8 1 2 3



∴ Index = 542

Properties of Good Hash Function:

Rules:

(i) The hash function should be simple to compute.

(ii) Number of collisions should be less while placing the record in the hash table. Ideally no collision should occur. Such a fraction is called perfect hash function.

(iii) Hash function should produce such keys which will get distributed uniformly over an array.

(iv) The hash function should depend on every bit of the key. Thus the hash function that simply extracts the portion of a key is not suitable.

Overflow Handling:

→ If collision occurs then it should be handled by applying some techniques. Such a technique is called collision handling Technique.

→ Overflow can be handled by two types,

(i) Separate chaining (or) External Hashing

(ii) Open Addressing (or) Closed Hashing.

Separate chaining (or) External Hashing:

→ Chaining is a collision resolution technique.

→ Chaining is a concept in which introduces an additional field with data (i) chain.

→ A separate chain table is maintained for colliding data.

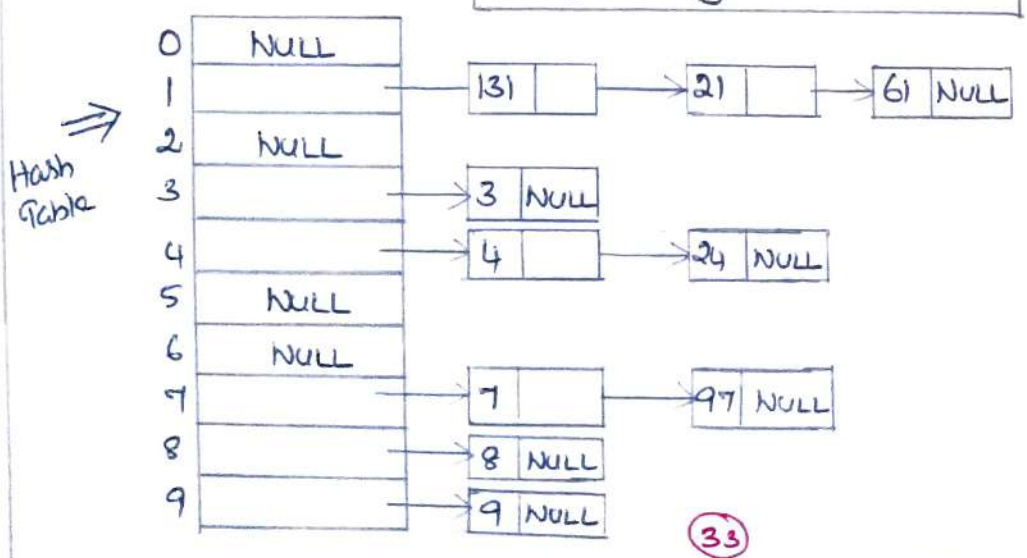
→ When collision occurs then a linked list is maintained at the home bucket.

Example:

Consider the hash table of size 10 with the keys are

131, 3, 4, 21, 61, 24, 7, 97, 8, 9

Hash Function ⇒ H(key) = key %. Table Size.



Open Addressing or Closed Hashing :

→ This is the easiest method of handling collision.

→ When collision occurs when two records demand for the same bucket in the hash table, then collision can be solved by placing the second record linearly down whenever the empty bucket is found

Example :

Consider the hash table of size 10 with the keys are, 131, 4, 8, 7, 21, 5, 31, 61, 9, 29.

Hash Table

0	NULL
1	131
2	NULL
3	NULL
4	4
5	NULL
6	NULL
7	7
8	8
9	NULL

⇒ Initially we put the following keys in the hash table as, 131, 4, 8, 7

→ Now the next key to be inserted is 21

$$H(key) = 21 \% 10 = 1$$

⇒ But the index 1 location is already occupied by 131 i.e. collision occurs.

⇒ To resolve this collision we will linearly move down and at the next empty location we will move the element.

⇒ 21 will be placed at index 2

Hash Table

0	29
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	9

⇒ To insert key 29 at index 9, But the key 9 is already placed at index 9. The overflow occurs.

⇒ To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0 index.

Collision Resolution Strategies:

→ In open Addressing there are three Common Collision Resolution Strategies. They are.

(i) Linear Probing.

(ii) Quadratic Probing

(iii) Double Hashing.

Linear Probing:

→ Probing is the process of getting next available hash table array cell.

→ One major problem with linear probing is primary clustering.

→ primary clustering is a process in which a block of data is formed in the hash table when collision is resolved.

Example:

Consider the following keys, with table size 10

89, 18, 49, 58, 69

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

→ Initially 89 is inserted at slot 9, 18 is inserted at slot 8. The first collision occurs when 49 is inserted it is put in the next available slot namely slot 0, which is open.

→ The key 58 collides with 18, 89 and then 49 before an empty cell is found three away.

→ The collision 69 is handled in a similar way.

Quadratic Probing :

→ Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value.

$$Hi(key) = (Hash(key) + i^2) \% m$$

Where m is the table size or any prime number.

Example :

→ To insert the following elements in the hash table with the table size 10.

37, 90, 55, 22, 11, 17, 49, 87.

Hash Table

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	
9	

37 % 10 = 7

90 % 10 = 0

55 % 10 = 5

22 % 10 = 2

11 % 10 = 1

17 % 10 = 7 ⇒ Bucket 7 already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

$$Hi(key) = (Hash(key) + i^2) \% m$$

Consider i=0 then,

$(17+0^2) \% 10 = 7$

When $i=1 \Rightarrow (17+1^2) \% 10 = 8$

→ The bucket 8 is empty, hence we will place the element at index 8.

→ $49 \% 10 = 9$

→ 87

$(87+0) \% 10 = 7$

$(87+1^2) \% 10 = 8$ → already occupied

$(87+2^2) \% 10 = 1$ → already occupied

$(87+3^2) \% 10 = 6$

Hash Table

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	17
9	49

Double Hashing:

→ To overcome secondary clustering problem double hashing technique is used.

→ It uses two hash functions to arrive at a hash value.

$$H_1(key) = (Hash(key) + i^2) \% m$$

where m is the table size.

$$F(i) = i * H_2(key)$$

$$H_2(key) = R - (key \bmod R)$$

where R is a prime number smaller than table size.

Example:

Consider the hash table of size 10 with the keys are,

89, 18, 49, 58, 69

→ 89 → H(89) = 89 mod 10

= 9 — No Collision

→ 18 → H(18) = 18 mod 10

= 8 — No Collision

→ 49 → H(49) = 49 mod 10

= 9 — Collision occur

$F(i) = i * H_2(key)$

$H_2(49) = R - (key \bmod R)$

$= 7 - (49 \bmod 7)$

$= 7 - 0$

$= 7$

$h(49) = (Hash(49) + F(i)) \bmod 10$

$= (9 + 1 * H_2(49)) \bmod 10 \quad [i = 1]$

$= (9 + 1 * 7) \bmod 10$

$= 16 \bmod 10$

$= 6$ — No Collision.

$58 \rightarrow H(58) = 58 \bmod 10$

$= 8$ — Collision occur

$H_2(58) = R - (key \bmod R)$

$= 7 - (58 \bmod 7)$

$= 7 - 2$

$= 5$

$h(58) = (H(58) + F(i)) \bmod 10$

$= (8 + (i * 5)) \bmod 10 \quad [i = 1]$

$= (8 + 5) \bmod 10$

$= 13 \bmod 10$

$= 3$ — No Collision

$69 \rightarrow H(69) = 69 \bmod 10$

$= 9$ — Collision occur

$H_2(69) = R - (key \bmod R)$

$= 7 - (69 \bmod 7)$

$= 7 - 6$

$= 1$

$h(69) = (H(69) + F(i)) \bmod 10$

$= (9 + (1 * 1)) \bmod 10 \quad [i = 1]$

$= (9 + 1) \bmod 10$

$= 10 \bmod 10 = 0$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Rehashing :

→ Rehashing is a technique in which the table is resized. i) The size of table is doubled by creating a new table.

→ It is preferable if the total size of table a prime number.

→ There are some situations in which the rehashing is required.

(i) When the table is completely full

(ii) With quadratic probing when the table is filled half.

(iii) When insertions fail due to overflow.

Example :

Consider the hash table of size 7 with the keys are, 13, 15, 24, 6, 23.

h(key) = key mod 7

	Hash table
0	6
1	15
2	23
3	24
4	
5	
6	13

13 mod 7 = 6

15 mod 7 = 1

24 mod 7 = 3

6 mod 7 = 0

23 mod 7 = 2

⇒ The old table size is 7, then we should double the size for new table, that becomes 17, because this is the first prime which is twice as large as the old table size. The new hash function is then.

h(key) = key mod 17

0	
1	
2	
3	
4	
5	
6	6
7	24
8	23
9	
10	
11	
12	
13	13
14	
15	15
16	

13 mod 17 = 13

15 mod 17 = 15

24 mod 17 = 7

6 mod 17 = 6

23 mod 17 = 6

Extendible Hashing:

→ Extendible hashing is a technique which is useful in handling large amount of data.

→ It is one form of dynamic hashing because data are frequently inserted and due to which the hash table size gets changed quite often.

→ The data to be placed in the hash table is by extracting certain number of bits.

→ The extendible hashing scheme contains main memory (directory) and one or more buckets stored on Disk.

→ The hash table size is always 2^d where d is called global depth. Each table entry points to one bucket

→ Each bucket is associated with local depth d.

Example:

Consider the key values,

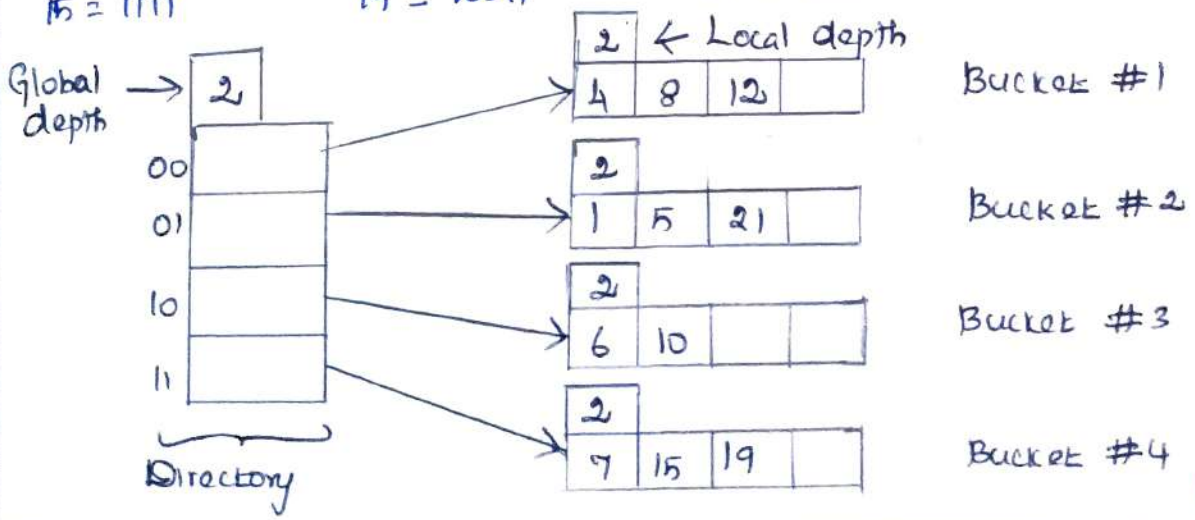
4, 1, 6, 7, 8, 12, 10, 5, 15, 19, 21

The binary representation are;

4 = 0100 1 = 0001 6 = 0110 7 = 0111

8 = 1000 12 = 1100 10 = 1010 5 = 0101

15 = 1111 19 = 10011 21 = 10101



⇒ Also insert 14

14 – 1110

14 will be inserted into Bucket #3 (The bucket # 3 is not overflow)

⇒ Insert 20

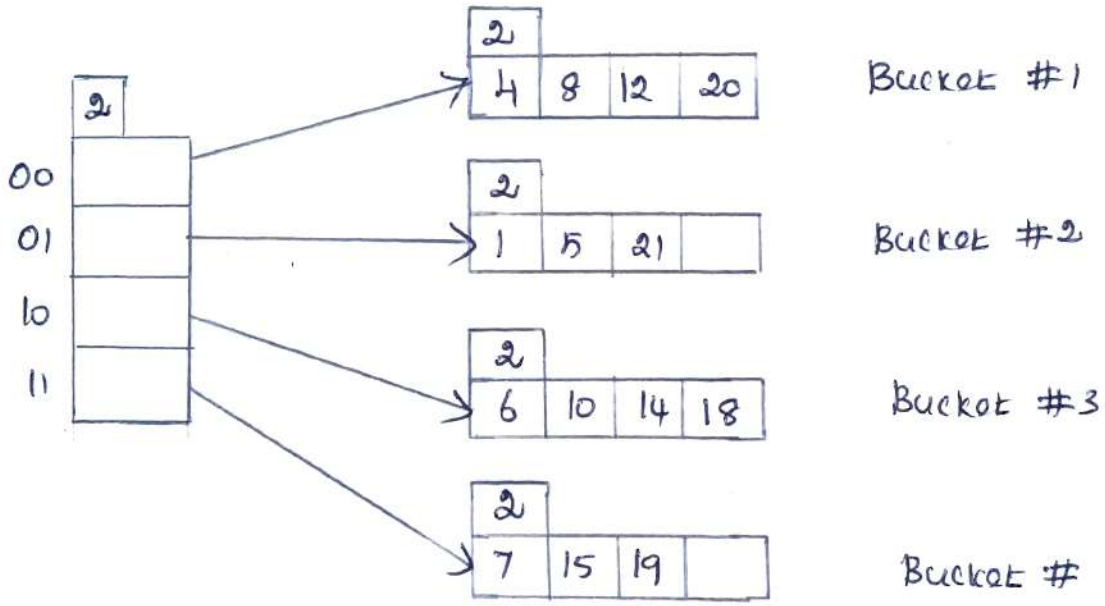
20 – 10100

20 will be inserted into Bucket #1

⇒ Insert 18

18 – 10010

18 will be inserted into Bucket #3 .

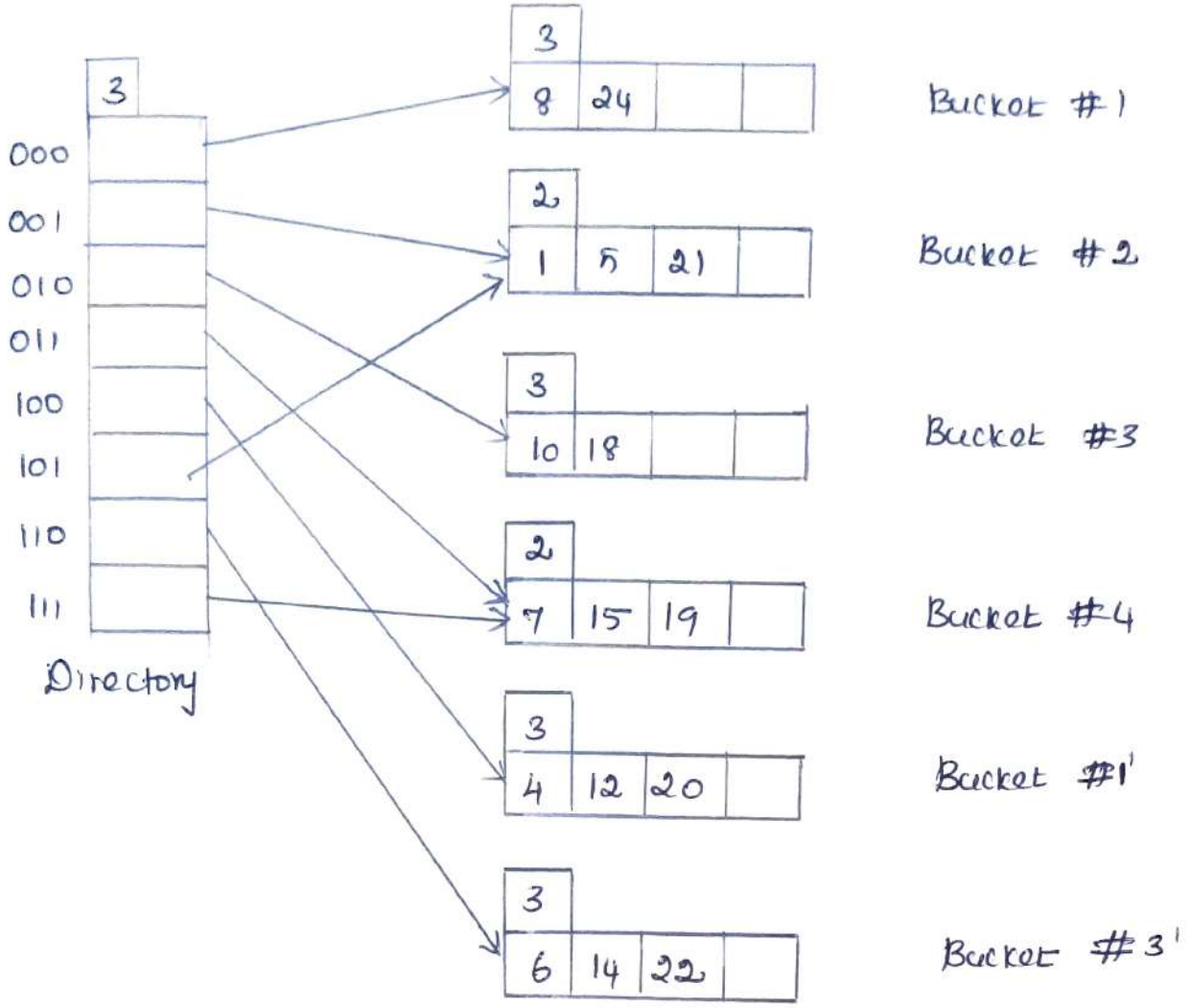


⇒ Insert 22

22 – 10110

22 will be inserted into Bucket #3 .

But Bucket #3 is full we need to create split image of Bucket #3 as Bucket #3'



Advantages :

(i) It requires minimum space overhead.

(ii) The performance of extendible hashing does not get degraded with the growing file size.

Disadvantages :

(i) It contains extra level of indirection for bucket address.

(ii) This method is complex to implement.

UNIT - V SORTING AND SEARCHING TECHNIQUES

Insertion sort - Quick sort - Heap sort - Merge sort - Linear search - Binary search.

Sorting:

→ Sorting is a technique for arranging data in the particular order.

→ The sorting order can be ascending or descending. The ascending order means arranging the data in increasing order whereas decending order means arranging the data in decreasing order.

→ Sorting has two common meanings.

1) Ordering: Arranging items in a sequence ordered by some criterion.

2) Categorizing: Grouping items with similar properties.

Types of Sorting:

→ Two types of sorting.

(i) Internal sorting

(ii) External sorting.

Internal Sorting:

→ This is a type of sorting technique in which data resides on main memory of computer.

External Sorting:

→ This is a sorting technique in which there is a huge amount of data and it resides on secondary devices (hard disk, Magnetic tapes) while sorting.

Techniques of Sorting :

→ Various techniques of sorting are,

(i) Insertion sort (ii) Quick sort

(iii) Heap sort (iv) Merge sort

Insertion Sort :

→ Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time.

→ This method sorts a list of elements by inserting each successive element in the previously sorted sub list. Such insertion of element requires the other elements to be shuffled as required.

Example :

Sort the elements using Insertion sort.

34, 8, 64, 51, 32, 21

Steps	34	8	64	51	32	21	Positions Moved
Step 1	8	34	64	51	32	21	1
Step 2	8	34	64	51	32	21	0
Step 3	8	34	51	64	32	21	1
Step 4	8	32	34	51	64	21	3
Step 5	8	21	32	34	51	64	4

The sorted elements are,

8, 21, 32, 34, 51, 64

Efficiency of Insertion Sort:

→ Assume that an array containing n elements is sorted using insertion sort technique.

→ The minimum number of elements that must be scanned = n-1

⇒ For each of the elements the maximum number of shifts possible = n-1

∴ Efficiency of Insertion Sort = $O(n^2)$

Advantages:

(i) Easy to implement.

(ii) It performs well in case of smaller lists.

Disadvantages:

(i) The efficiency of $O(n^2)$ is not well suited for large sized lists.

(ii) It requires large number of elements to be shifted.

Program:

```
#include <stdio.h>
void main()
{
  int a[10], i, j, n, temp;
  printf("Enter the number of elements...\n");
  scanf("%d", &n);
  printf("Enter %d elements\n", n);
```

```
for (i=0; i<n; i++)
scanf("%d", &a[i]);
printf("Entered elements \n");
for (i=0; i<n; i++)
printf("%d \t", a[i]);
for (i=1; i<n; i++)
{
  temp = a[i];
  for(j=i; j>0 && a[j-1] > temp; j--)
  a[j] = a[j-1];
  a[j] = temp;
}
printf("\n Sorted array is .... \n");
for (i=0; i<n; i++)
printf("%d \t", a[i]);
}
```

Output:

```
Enter the number of elements .... 5
Enter 5 elements
11  7  2  1  4
Entered elements
11  7  2  1  4
Sorted array is ....
1  2  4  7  11
```

Quick Sort :

→ Quick sort is one of the fastest sorting methods that is based on divide and conquer strategy

→ It divides the given list into a number of sub lists. and then works on each of sub lists to obtain the sorted output.

→ It first chooses one of the list elements as a key value and then tries to place the key value at its final position in the list.

→ Once the key value is positioned correctly the two sublists to the left and right of the key value are processed in similar until the entire list becomes sorted.

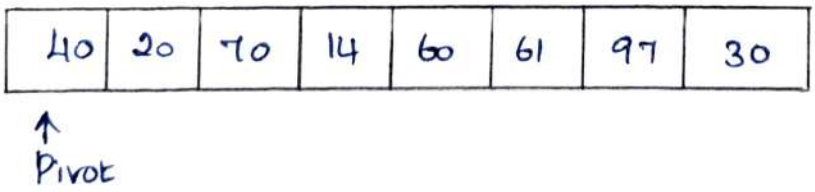
→ The key value is also called as pivot element.

Example :

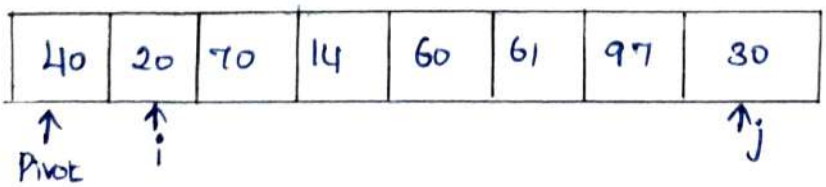
Sort the elements using Quick sort.

40, 20, 70, 14, 60, 61, 97, 30

Step 1 :

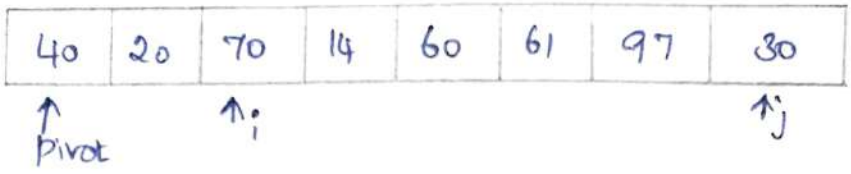


Step 2 :



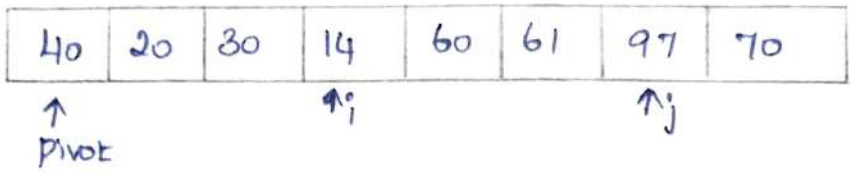
20<40
so move right

Step 3 :



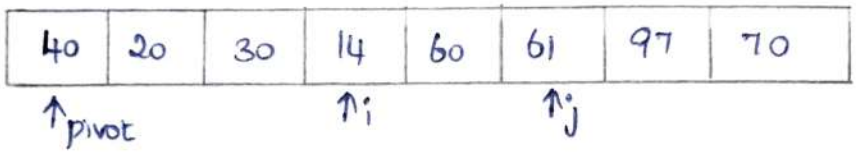
40 < 70
exchange 70 and 30.

Step 4 :



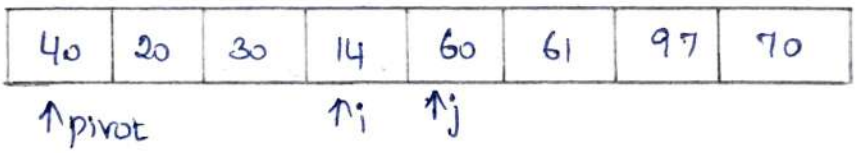
14 < 40
97 > 40
so move Left

Step 5 :



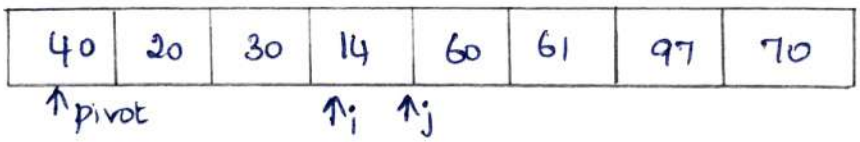
61 > 40
so move left.

Step 6 :



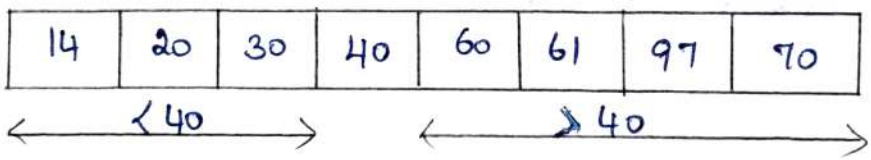
60 > 40
so move left

Step 7 :

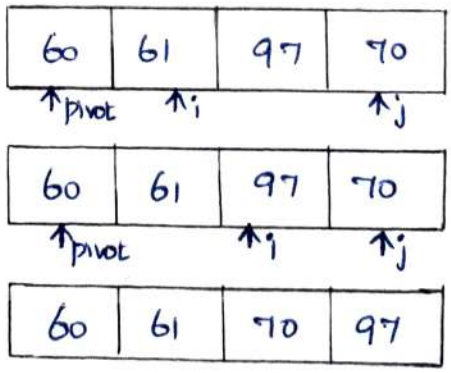


Exchange 40 and 14

Step 8 :



Step 9 :



The Sorted elements are 14, 20, 30, 40, 60, 61, 70, 97

Efficiency of Quick Sort:

Best case:

→ Assume that the number of elements in the list is a power of 2. That means $n = 2^m$ or $m = \log_2 n$

Efficiency of quick sort in best case = $O(n \log n)$

Worst case:

Efficiency of quick sort in worst case = $O(n^2)$

Advantages:

(1) It is one of the fastest sorting algorithms

(2) Its implementation does not require any additional memory.

Disadvantages:

(1) The worst case efficiency of $O(n^2)$ is not well suited for large sized lists.

(2) Its algorithm is considered as a little more complex in comparison to other sorting techniques.

Program:

```
#include <stdio.h>
void quicksort(int a[], int, int);
int n;
void main()
{
  int a[20], i;
  printf("\n Enter the number of elements");
  scanf("%d", &n);
  printf("\n Enter the elements of array");
```

```
for(i=0 ; i<n; i++)
scanf(" %d", &a[i]);
quicksort (a, 0, n-1);
printf("\n The Sorted array is :\n");
for(i=0 ; i<n ; i++)
printf (" %d \t", a[i]);
}
void quicksort (int a[20], int lower, int upper)
{
   int pivot, i, j, temp;
   if ( lower < upper)
   {
      pivot = lower;
      i = lower;
      j = upper;
      while (i <j)
      {
        while (a[i] < = a [pivot] && i < upper)
        i++ ;
        while (a[j] > a [pivot])
        j-- ;
        if (i<j)
        {
          temp = a[i] ;
          a[i] = a[j];
          a[j] = temp;
        }
   }
```

```
        temp = a[pivot];
        a[pivot] = a[j];
        a[j] = temp;
        quicksort(a, lower, j-1);
        quicksort(a, j+1, upper);
    }
}
```

Output:

```
Enter the number of elements: 5
Enter the elements of array: 21
                             12
                             40
                             17
                             35
The sorted array is:
12  17  21  35  40
```

Heap Sort:

What is heap?

→ A heap is a complete binary tree and the binary tree is a tree in which the node can have utmost two children.

Heap Sort:

→ Heap sort is the in-place sorting algorithm.

→ The concept of heap sort is to eliminate the elements one by one from the heap part of the list and then insert them into the sorted part of list.

Working of heapsort:

→ Two phases

(i) The first step includes the creation of a heap by adjusting the elements of the array.

(ii) After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array and then store the heap structure with the remaining elements.

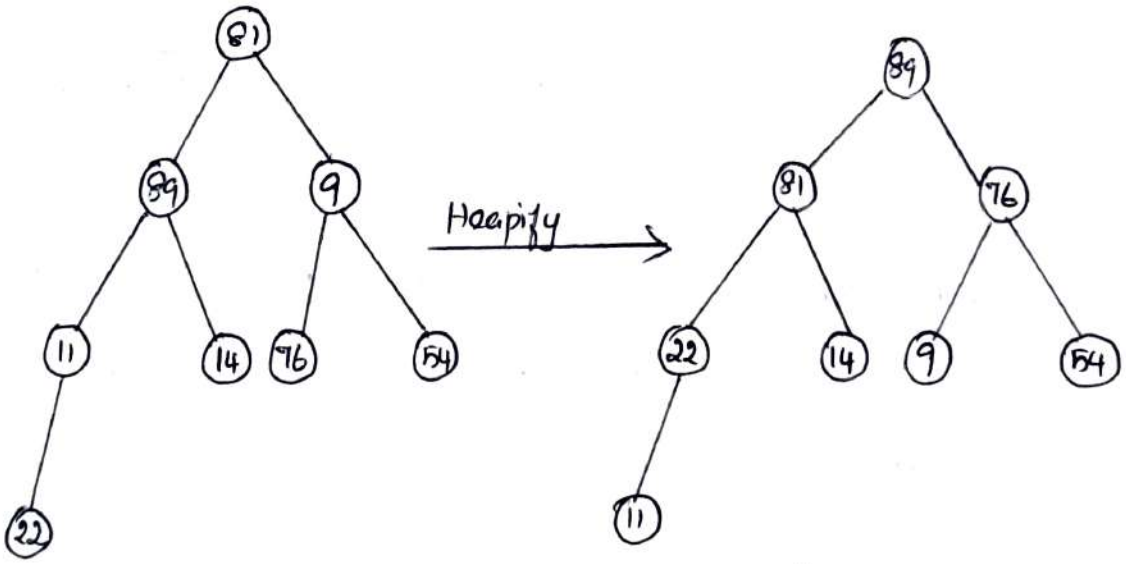
Example:

Sort the elements using heap sort.

81, 89, 9, 11, 14, 76, 54, 22

Step 1:

→ To construct a heap from the given array and convert it into max heap.



Heap

Maxheap

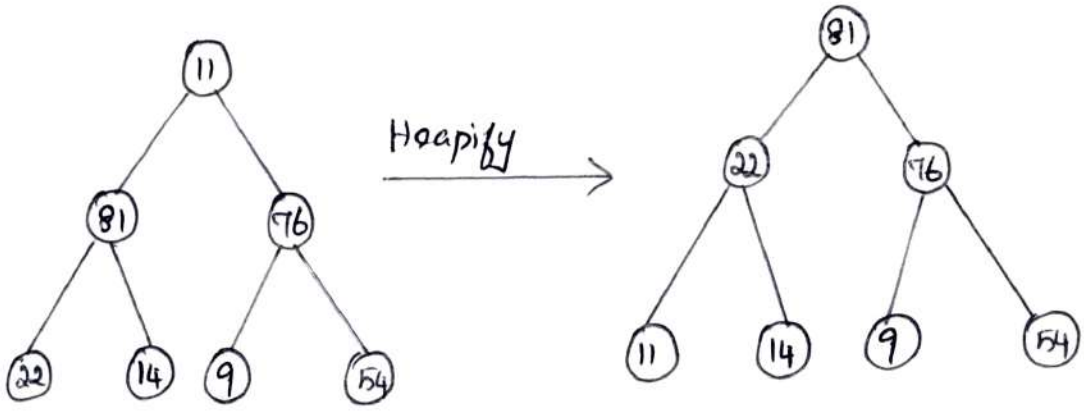
→ After converting max heap the array elements are,

89	81	76	22	14	9	54	11

Step 2:

→ Next, to delete the root element 89 from maxheap.

→ To delete root 89, then swap the last node 11. After deleting the root element convert it into max heap.



After deleting node 89 Max heap

→ After converting max heap the elements are,

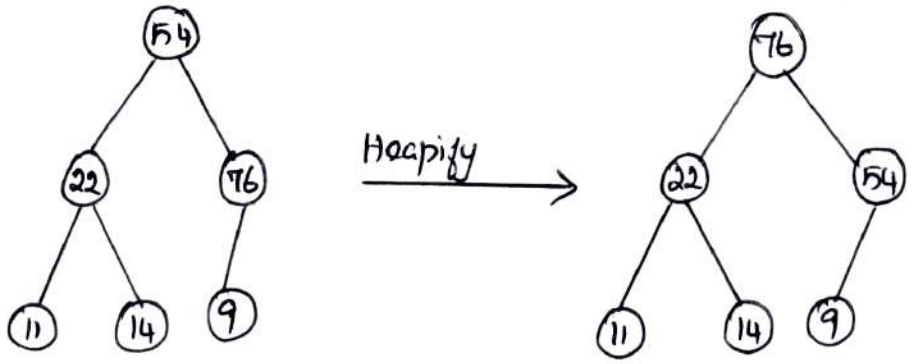
81	22	76	11	14	9	54	89

Step 3:

→ Next, to delete the root element 81 from maxheap.

→ To delete root 81, then swap the last node is 54.

After deleting the root element, convert it into maxheap.



After deleting node 81 Max Heap.

→ After converting max heap the elements are,

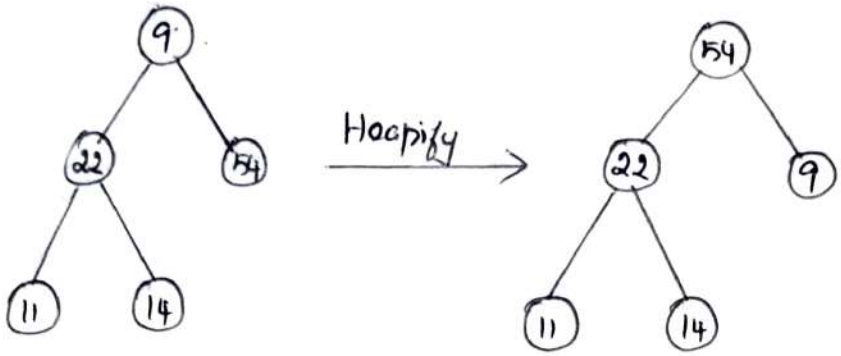
76	22	54	11	14	9	81	89

Step 4:

→ Next, to delete the root element 76 from maxheap.

→ To delete root 76, then swap the last node i) 9

After deleting the root element convert it into max heap.



After deleting node 76 Max Heap.

→ After converting the max heap the elements are,

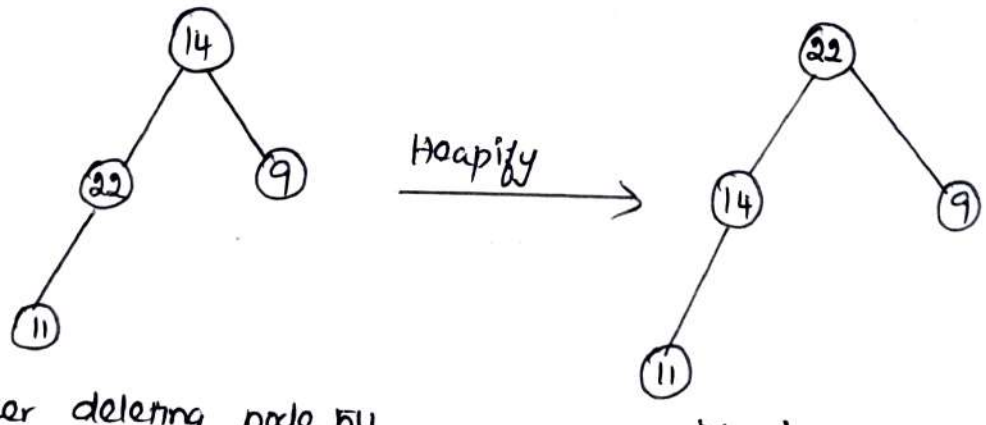
54	22	9	11	14	76	81	89

Step 5:

→ Next to delete the root element 54 from maxheap

→ To delete root 54, then swap the last node i) 14

After deleting the root element convert it into maxheap.



After deleting node 54 Maxheap.

→ After converting the maxheap the elements are,

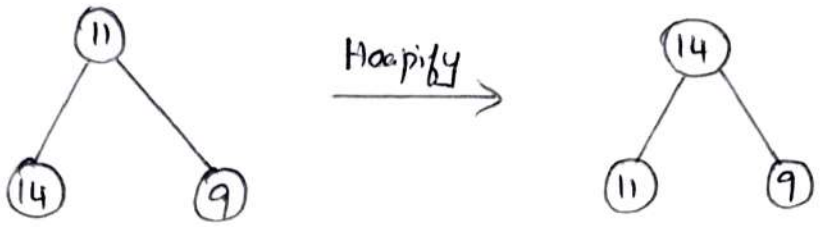
22	14	9	11	54	76	81	89

Step 6 :

→ Next, to delete the root element 22 from max heap.

→ To delete root 22 then swap the last node (i) 11

After deleting root element convert it into max heap.



After deleting node 22 Max heap.

→ After converting the max heap, the elements are,

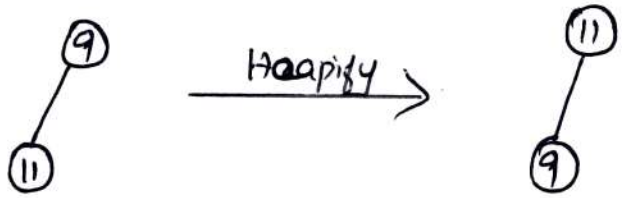
14	11	9	22	54	76	81	89

Step 7 :

→ Next to delete the root element 14 from max heap.

→ To delete root 14, then swap the last node 9.

→ After deleting the root element convert it into max heap.



After deleting node 14 Max heap.

→ After converting the max heap, the elements are

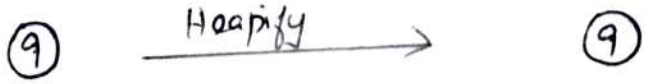
11	9	14	22	54	76	81	89

Step 8 :

→ Next, to delete the root element 11 from max heap.

→ To delete root 11 then swap the last node (ie, 9

After deleting the root element convert it into max heap



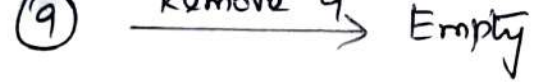
After deleting node 11 — Max Heap

→ After converting the max heap the elements are,

9	11	14	22	54	76	81	89

Step 9 :

→ Now heap has only one element, After deleting it heap will be empty.



⇒ After completion of sorting, the array elements are

9	11	14	22	54	76	81	89

Efficiency of Heap Sort :

1) Time Complexity :

Best case - $O(n \log n)$

Average case - $O(n \log n)$

Worst case - $O(n \log n)$

2) Space Complexity :

Space complexity of Heap sort is $O(1)$.

Program:

```
#include <stdio.h>
void heapify (int a[], int n, int i)
{
  int largest = i;
  int left = 2*i+1;
  int right = 2*i+2;
  if (left < n && a[left] > a[largest])
     largest = left;
  if (right < n && a[right] > a[largest])
     largest = right;
  if (largest != i)
    {
       int temp = a[i];
       a[i] = a[largest];
       a[largest] = temp;
       heapify (a, n, largest);
    }
}
Void heapSort (int a[], int n)
{
  for (int i = n/2 - 1; i >= 0; i--)
     heapify (a, n, i);
```

```
for (int i = n-1; i>=0; i--)
{
    int temp = a[0];
    a[0] = a[i];
    a[i] = temp;
    heapify(a,i,0);
  }
}
Void printArr (int arr[], int n)
{
  for (int i=0; i<n; ++i)
  {
    printf("%d", arr[i]);
    printf(" ");
  }
}
int main()
{
 int a[] = {48,10,23,43,28,26,1};
 int n = sizeof(a) / sizeof(a[0]);
 printf("Before Sorting array elements are :\n");
 printArr(a,n);
 heapSort(a,n);
 printf("\n After Sorting Array elements are :\n");
 printArr(a,n);
 return 0;
}
```

Output :

Before Sorting array elements are :
48 10 23 43 28 26 1
After Sorting array elements are :
1 10 23 26 28 43 48

Merge Sort:

→ Merge sort is another sorting technique that is based on divide and conquer approach.

→ It divides a list into several sub lists of equal sizes and sorts them individually.

→ It then merges the various sub lists in pairs to eventually form the original list, while ensuring that the sort order is not disturbed.

→ Merge sort is an input array with n elements consists of three steps.

(i) Divide → Partition array into two sublists S_1 and S_2 with n/2 elements each.

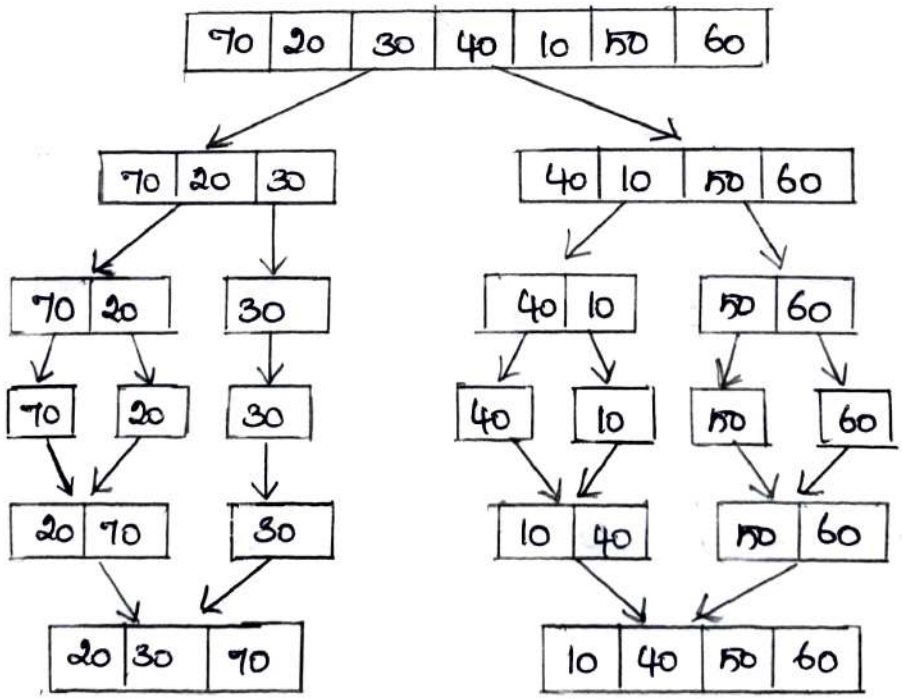
(ii) Conquer → Then sort sublist S_1 and sub list S_2

(iii) Combine → Merge S_1 and S_2 into a sorted group.

Example:

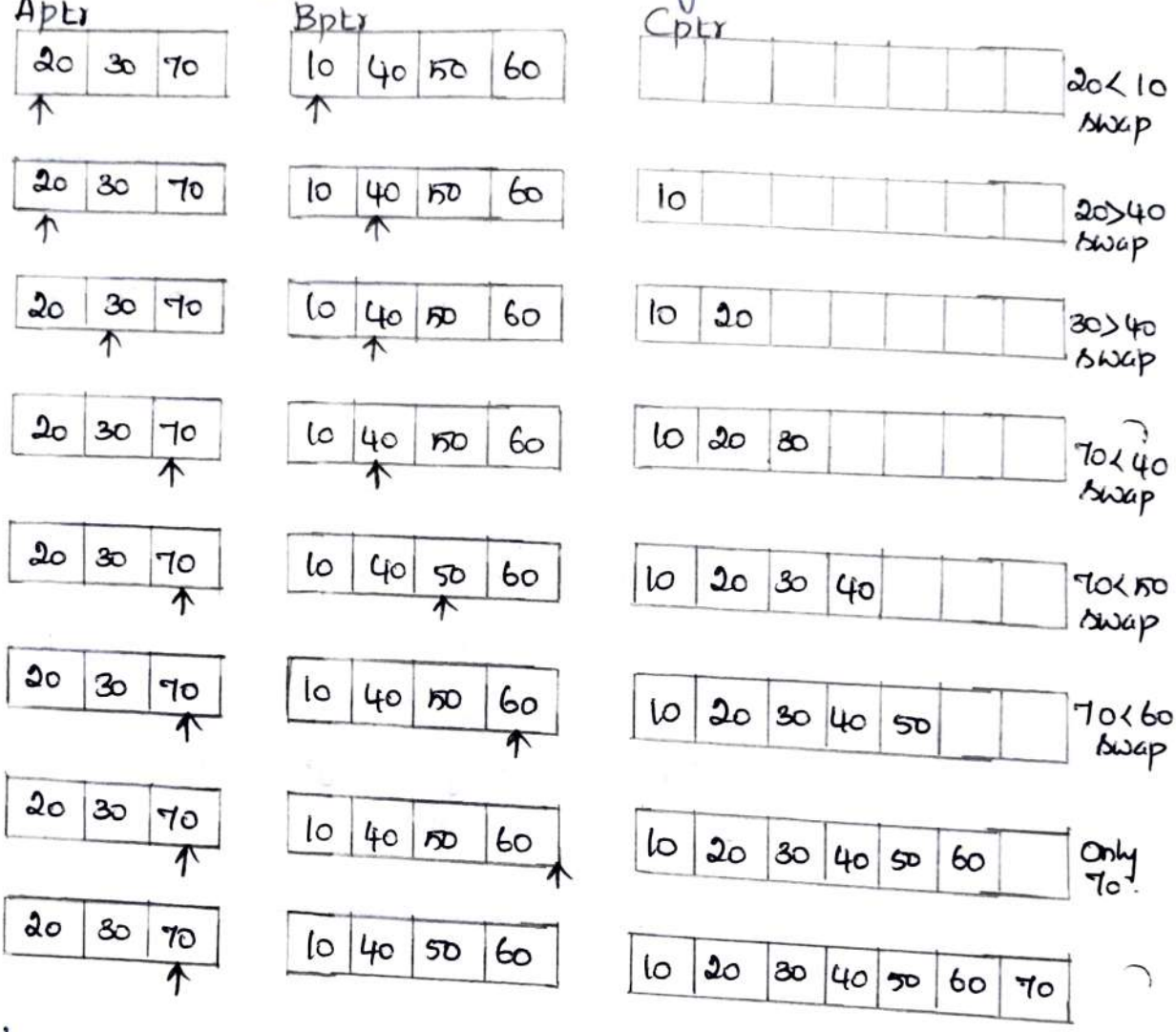
Sort the elements using Merge sort

70, 20, 30, 40, 10, 50, 60.



→ Let us Consider first 3 elements for input array A and next four elements are input array B.

→ aptr, bptr, cptr which are initially set to the beginning of their respective arrays.



The sorted elements are

10, 20, 30, 40, 50, 60, 70.

Efficiency of Merge sort

⇒ Efficiency of Merge sort = $O(n \log_2 n)$

Advantages:

(i) It is a fast and stable sorting method.

(ii) It always ensures an efficiency of $O(n \log n)$

Disadvantages:

(i) It requires additional memory space to perform sorting.

Program:

```
#include <stdio.h>
void mergesort (int a[], int i, int j);
void merge (int a[], int i1, int j1, int i2, int j2);
void main()
 {
    int a[30], i, n;
    printf (" Enter the number of elements :");
    scanf (" %d", &n);
    printf (" Enter array elements");
    for (i=0; i<n; i++)
    scanf (" %d", & a[i]);
    mergesort (a, 0, n-1);
    printf (" In Sorted array is :");
    for (i=0; i<n; i++)
    printf (" %d", a[i]);
 }
 void mergesort (int a[], int i, int j)
  {
    int mid;
    if (i<j)
    {
```

```
        mid = (i+j)/2;
        mergesort (a, i, mid);
        mergesort (a, mid+1, j);
        merge (a, i, mid, mid+1, j);
    }
}
void merge (int a[], int i1, int j1, int i2, int j2)
{
    int temp[50];
    int i, j, k;
    i = i1;
    j = j2;
    k = 0
    while (i <= j1 && j <= j2)
    {
        if (a[i] < a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }
    while (i <= j1)
        temp[k++] = a[i++];
    while (j <= j2)
        temp[k++] = a[j++];
    for (i = i1, j = 0; i <= j2; i++, j++)
        a[i] = temp[j];
}
```

Output:

```
Enter the number of elements : 5
Enter array elements: 11
                      7
                      9
                      3
                      1
Sorted array is :
1 3 7 9 11
```

Searching:

→ Searching is the process of determining whether an element is present in a given list of elements or not.

→ If the element is found to be present in the list then the search is considered as successful otherwise it is considered as an unsuccessful search.

→ The search operation returns the location or address of the element found.

Characteristics of Searching:

(i) It should be efficient.

(ii) Less number of computations must be involved in it.

(iii) The space occupied by searching algorithms must be less.

Types:

→ Two types

(i) Sequential search or Linear search

(ii) Binary search.

Linear search:

→ Sequentially searches for an element in the list.

→ It starts with the first element in the list and moves towards the end in step by step.

→ In each iteration it compares the element to be searched with the list element and if there is a match the location of the list element is returned.

→ The search begins at one end of the list and searches for the required element one by one until the element is found or till the end of the list is searched.

Example :

Consider the array of elements to find 33.

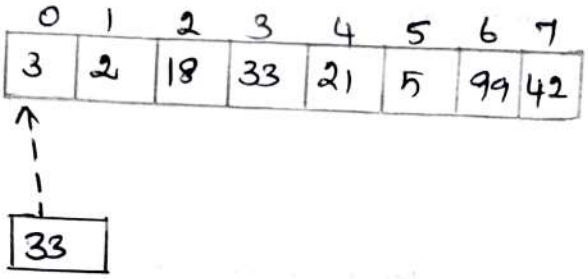
3, 2, 18, 33, 21, 5, 99, 42.

Array A[8]

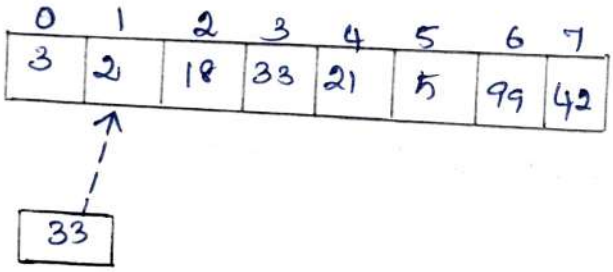
0	1	2	3	4	5	6	7
3	2	18	33	21	5	99	42

key → 33

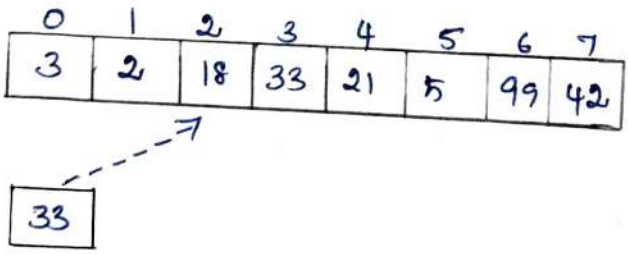
Step 1 :



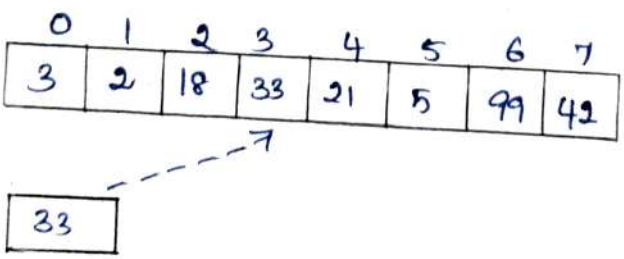
Step 2 :



Step 3 :



Step 4 :



Element 33 is found at location 3

Efficiency of Linear Search:

⇒ Efficiency of Linear search $= (1+2+\cdots+n)/n$

$= n(n+1)/2n$

$= O(n)$

Advantages:

(i) It is a simple searching technique that is easy to implement

(ii) It does not require the list to be sorted in a particular order

Disadvantages:

(i) It is quite inefficient for large sized lists.

(ii) It does not leverage the presence of any pre existing sort order in a list.

Program:

```
#include <stdio.h>
void main()
{
 int array[10],search,i,n,yes=0;
printf(" Enter the number of elements in array \n");
scanf("%d", &n);
printf(" Enter %d integers \n" ,n);
for(i=0; i<n ; i++)
 scanf(" %d", &array [i]);
printf(" Enter a number to search \n");
scanf (" %d ", & search);
for (i=0; i<n; i++)
{
  if (array [i] == search)
  {
```

```
        yes = 1;
        printf(" %d is present at location %d \n", search, i+1);
        break;
    }
  }
  if (yes == 0)
    printf(" %d isn't present in the array \n", search);
}
```

Output:

```
Enter the number of elements in array 5
Enter 5 integers 11
                 13
                 10
                 7
                 15
Enter a number to search: 7
7 is present at location 3
```

Binary search:

→ Binary search requires the element of a list to be already arranged in a sorted manner before search can be performed in it.

→ It begins by comparing the element that is present at the middle of the list.

→ If there is a match then the search ends immediately and the location of the middle element is returned.

→ If there is a mismatch then it focusses the

Search either in the left or right sub list depending on whether the target element is lesser than or greater than middle element.

Rule:

(i) First of all all the middle element in the array is identified.

(ii) Now the search element k is compared with middle element.

(iii) k is greater than middle element the search is focussed on right sub list.

(iv) k is lesser than middle element the search is focussed on left sub list.

(v) The search element k is found in right or left list then the search is successful else the search is unsuccessful.

Example:

Consider the array of elements to find 33

2, 3, 5, 18, 21, 33, 42, 99

Array A[8]

0	1	2	3	4	5	6	7
2	3	5	18	21	33	42	99

key → 33

Step 1:

⇒ Find the Middle Element.

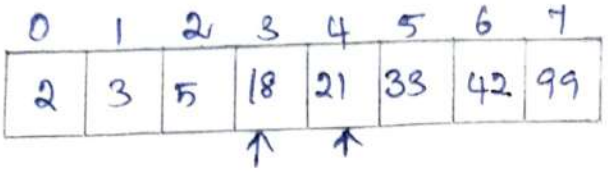
0	1	2	3	4	5	6	7
2	3	5	18	21	33	42	99

↑ Middle element

33

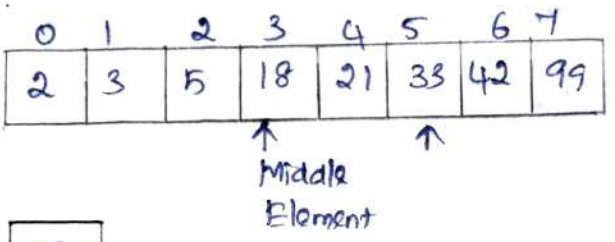
Step 2:

18 < 33 → Search right list



33

Step 3:



33

Element 33 is found at location 5

Efficiency of Linear Search:

Worst case Efficiency = $O(\log_2 n)$

Best case Efficiency = $O(1)$

Advantages:

(1) It requires lesser number of iterations

(2) It is faster than linear search.

Disadvantages:

(1) It requires the list to be sorted before search can be performed.

Program:

```
#include <stdio.h>
void main()
{
int a[10], n, i, s, low, high, mid;
printf(" Enter the number of elements \n");
scanf("%d", &n);
printf(" Enter the elements in ascending order \n");
for(i=0; i<n ; i++)
scanf("%d", &a[i]);
printf(" Entered elements in array \n");
for(i=0 ; i<n ; i++)
printf("\t %d", a[i]);
printf("\n Enter the element to be searched :");
scanf("%d", &s);
low=0;
high = n-1;
while (low <= high)
{
  mid = (low + high)/2 ;
  if (a[mid] == s)
  break ;
  else if (a[mid] < s)
  low = mid +1;
  else
  high = mid -1;
}
```

```
    if (low > high)
      printf (" Element isn't in the array");
    else
      printf (" The element %d is found at the location %d",
                                          s, mid+1);
}
```

Output :

```
Enter the number of elements : 5
Enter the elements in ascending order : 7
                                        10
                                        11
                                        13
                                        15
Entered elements in array :
   7   10   11   13   15
Enter the element to be searched : 13
The element 13 is found at the location 3
```