A Course Material on

Programming and Data Structures-I

By

M.SOWMIYA

ASSISTANT PROFESSOR

DEPARTMENT OF INFORMATION TECHNOLOGY

EASWARI ENGINEERING COLLEGE

CS6202 PROGRAMMING AND DATA STRUCTURES I L T P C 3 0 0 3

OBJECTIVES:

- To introduce the basics of C programming language
- To introduce the concepts of ADTs
- To introduce the concepts of Hashing and Sorting

UNIT I C PROGRAMMING FUNDAMENTALS- A REVIEW 9

Conditional statements – Control statements – Functions – Arrays – Preprocessor - Pointers - Variation in pointer declarations – Function Pointers – Function with Variable number of arguments

UNIT II C PROGRAMMING ADVANCED FEATURES 9

Structures and Unions - File handling concepts – File read – write – binary and Stdio - File Manipulations

UNIT III LINEAR DATA STRUCTURES – LIST 9

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation —singly linked lists- circularly linked lists- doubly-linked lists – applications of lists –Polynomial Manipulation – All operation (Insertion, Deletion, Merge, Traversal)

UNIT IV LINEAR DATA STRUCTURES – STACKS, QUEUES 9

Stack ADT – Evaluating arithmetic expressions- other applications- Queue ADT – circular queue implementation – Double ended Queues – applications of queues

UNIT V SORTING, SEARCHING AND HASH TECHNIQUES 9

Sorting algorithms: Insertion sort - Selection sort - Shell sort - Bubble sort - Quick sort - Merge sort -Radix sort – Searching: Linear search –Binary Search Hashing: Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing

TOTAL: 45 PERIODS

TEXT BOOKS:

1. Brian W. Kernighan and Dennis M. Ritchie, “The C Programming Language”, 2nd Edition, Pearson
Education, 1988.
2. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C”, 2nd Edition, Pearson Education,1997.

REFERENCES:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, Clifford Stein, “Introduction to Algorithms", Second Edition, Mcgraw Hill, 2002.
2. Reema Thareja, “Data Structures Using C”, Oxford University Press, 2011
3. Aho, Hopcroft and Ullman, “Data Structures and Algorithms”, Pearson Education,1983.
4. Stephen G. Kochan, “Programming in C”, 3rd edition, Pearson Ed.,

UNIT I
C PROGRAMMING FUNDAMENTALS - A REVIEW

TOPICS COVERED

1. Conditional statements
2. Control statements
3. Functions
4. Arrays
5. Pre-processor
6. Pointers Variation in pointer declarations
7. Function Pointers
8. Function with Variable number of arguments

INTRODUCTION TO C

C is a general purpose, structured programming language. Its instructions consist of terms that resemble algebraic expressions, augmented by certain English keywords such as if, else, for, do and while. In this respect it resembles high level structured programming languages such as Pascal and Fortran. C also contains additional features, that allow it to be used at a lower level, thus bridging the gap between machine language and high level language. This flexibility allows C to be used for systems programming as well as for applications programming. Therefore C is called a ***middle level language.***

C is characterized by the ability to write very concise source programs, due in part to the large number of operators included within the language. It has a relatively small instruction set, though actual implementations include extensive library functions which enhance the basic instructions. C encourages users to create their own library fuctions.

An important characteristic of C is that its programs are highly portable. The reason for this is that C relegates most computer dependent features to its library functions. Thus, every version of C is accompanied by its own set of library functions which are relatively standardized. Therefore most C programs can be processed on many different computers with little or no alteration.

History of C:

C was developed in the 1970′s by **Dennis Ritchie** at **Bell Telephone Laboratories**,Inc. (now a part of AT&T). It is an outgrowth of two earlier languages, called BCPL and B, which were also developed at Bell Laboratories.

The **Combined Programming Language(CPL)** was developed at Cambridge University in 1963 with the goal of developing a common programming language which can be used to solve different types of problems on various hardware platforms. However it turned out to be too complex, hard to learn and difficult to implement. Subsequently in 1967, a subset of CPL, **Basic CPL(BCPL)** was developed by **Martin Richards** incorporating only the essential features. However it was not found to be sufficiently powerful. Around the same time another subset of CPL, a language called **B** was developed by **Ken Thompson** at Bell Labs. However it also turned out to be insufficient . Then, in 1972, Dennis Ritchie at Bell Labs developed the C language incorporating the best features of both BCPL and B.

C was largely confined to use within Bell Labs until 1978, when Brian Kernighan and Ritchie published a definitive description of the language . The **Kerninghan and Ritchie description of C** is commonly referred to as ***'K &R C'***.

Following the publication of 'K&R C',computer professionals, impressed with C's many desirable features, began to promote the use of C. By the mid 1980′s the popularity of C had become widespread-many c compilers and interpreters had been written for computers of all sizes and many commercial application programs had been developed. Moreover, many commercial software products that had originally been written in other languages were rewritten in C in order to take advantage of its efficiency and portability.

Early commercial implementations of C differed a little from Kerninghan and Ritchie's original description, resulting in minor incompatibilities between different implementations. As a result, **the American National Standards Institute(ANSI committee X3J11)** developed a standardized definition of C. Virtually all commercial compilers and interpreters adhere to the ANSI standard. Many provide additional features of their own.

C and Systems Programming:

There are several features of C, which make it suitable for systems programming. They are as follows:

- C is a machine independent and highly portable language.
- It is easy to learn; it has only 28 keywords.
- It has a comprehensive set of operators to tackle business as well as scientific applications with ease.
- Users can create their own functions and add to the C library to perform a variety of tasks.
- C language allows the manipulation of bits, bytes and addresses.
- It has a large library of functions.
- C operates on the same data types as the computer, so the codes generated are fast and efficient.

Structure of a C Program:

Every C program consists of one or more modules called ***functions.*** One of the functions must be called ***main***. The program will always begin by executing the *main* function, which may access other functions. The main function is normally,but not necessarily located at the beginning of the program. The group of statements within main( ) are executed sequentially. When the closing brace of main( ) is encountered, program execution stops and control is returned to the operating system.

Any other function defintions must be defined separately, either ahead or after main( ). Each function must contain:

1. A ***function heading***, which consists of the ***function name***, followed by an optional list of arguments, enclosed in parantheses.
2. A ***return type*** written before the function name. It denotes the type of data that the function will return to the program.
3. A list of ***argument declarations***, if arguments are included in the heading.
4. A ***compound statement***, which comprises the remainder of the function.

The arguments(also called parameters) are symbols that represent information being passed between the function and other parts of the program.

Each compound statement is enclosed between a pair of braces{ }. The braces may contain one or more elementary statements (called ***expression statements***) and other compound statements. Thus compound statements may be nested one within another. Each expression statement must end with a semicolon(;).

Comments (remarks) may appear anywhere within a program as long as they are enclosed within the delimiters /* and */. Comments are used for documentation and are useful in identifying the program's principal features or in explaining the underlying logic of various program features.

Components of C Language:

There are five main components of the C Language:-

1. **The character set:** C uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9 and certain special characters as building blocks to form basic program elements(e. g. constants, variables, expressions, statements etc.).
2. **Data Types:** The C language is designed to handle five *primary data types,* namely, *character, integer, float, double and void;* and *secondary data types* like *array, pointer, structure, union and enum.*
3. **Constants:** A constant is a fixed value entity that does not change its value throughout program execution.
4. **Variables:** A variable is an entity whose value can change during program execution. They are used for storing input data or to store values generated as a result of processing.
5. **Keywords:** Keywords are ***reserved*** words which have been assigned specific meanings in the C language. Keywords cannot be used as variable names.

The components of C language will be discussed in greater detail in the following articles. This section gives only a brief introduction to the components of C.

Example 1: The following program reads in the radius of a circle, calculates the area and then prints the result.

```
/* program to calculate the area of a circle*/

#include<stdio.h> /*Library file access*/

#include<conio.h> /*Library file access*/

void main( )          /* Function Heading*/

  {

    float radius, area; /*Variable declarations*/
```

```
 /*Output Statement(prompt)*/

 printf("Enter the radius :");

/*Input Statement*/

scanf("%f", &radius);

/*Assignment Statement*/

area = 3.14159*radius*radius;

/*Output Statement*/

printf("Area of the circle :%f", area);

 getch( );

}
```

Program output:-
Enter the radius: 3
Area of the circle: 28. 27431

The following points must be considered to understand the above program:-

1. The program is typed in lowercase. C is **case sensitive** i. e. uppercase and lowercase characters are not equivalent in C. It is customary to type C instructions in lowercase. Comments and messages(such as those printed using printf()) can be typed in anycase.
2. The first line is a comment that identifies the purpose of the program.
3. The instruction **#include <stdio.h>** contains a reference to a special file called stdio. h . This file contains the definition of certain functions required to read and print data such as printf() and scanf() . It is a header file and hence the extension . h.
4. Similarly **#include <conio.h>** links the file conio. h which is another header file that contains the definitions of functions used for reading and printing data at the console. The function **getch()** is defined in conio. h. # denotes a preprocessor directive. More about this in a later article.
5. The instruction **void main()** is a heading for the function main( ). The keyword **void** denotes the return type of main and indicates that the function does not return any value to the program after the program has finished executing. The empty parantheses ( ) after main indicates that this function does not include any arguments. Program execution always begins from main( ).
6. The remaining five lines of the program are indented and enclosed in a pair of braces { }. These five lines comprise the compound statement within the function main( ).
7. The instruction **float radius, area;** is a ***variable declaration.*** It establishes the symbolic names ‘radius’ and ‘area’ as floating point variables. These variables can accept values of type ‘float ‘ i. e numbers containing a decimal point or an exponent.
8. The next four instructions are ***expression statements.*** The instruction **printf(“Enter the radius :”);** generates a request for information namely,the value for the radius. This statement generates a prompt where the user enters the value .

9. The value of the radius is read into (or stored in) the variable *radius* with the help of the scanf ( ) function. The instruction **scanf("%f", &radius);** is used for reading data. "%f" is a ***conversion character*** which is used to accept a floating point value.
10. The next instruction, **area = 3.14159*radius*radius;** is called an ***assignment statement.*** This instruction calculates the area by using the value of radius entered by the user and assigns the value to the variable *area.*
11. The next printf( ) statement prints the message *Area of the circle* followed by the calculated area.
12. The statement **getch();** is used to pause the screen so that you can read the output. If getch( ) is not used the screen will just flash and go away. This function waits for the user to input some character(as it accepts a character as input), after the program has finished executing. Any key present on the keyboard pressed by the user is accepted by the getch function as input and its ASCII value is returned to main( ).

1.1 CONDITIONAL STATEMENTS

Conditional statements are used to execute a statement or a group of statement based on certain conditions. The ability to control the flow of our program, letting it make decisions on what code to execute, is valuable to the programmer. One of the important functions of the conditional statement is that it allows the program to select an action based upon the user's input.

C Statements and Blocks

C has three types of statement.

- Assignment
  =
- Conditional (branching)
  if (expression)
  else
  switch
- Control (looping)
  while (expression)
  for (expression;expression;expression)
  do {block}

Blocks

These statements are grouped into *blocks*, a block is identified by curly brackets...There are two types of block.

- Statement blocks

```
if ( i == j)
{
 printf("martin \n");
}
```

The *statement block* containing the **printf** is only executed if the **i == j** *expression* evaluates to **TRUE.**

- Function blocks

```
int add( int a, int b)      /* Function definition */
{
  int c;
  c = a + b;
  return c;
```

}

The statements in this block will only be executed if the *add* function is called.

We will look into following conditional statements.

1. if
2. if else
3. else if
4. switch
5. goto

1.1. 1 IF STATEMENT

If statement syntax

```
if (test expression){
      statement/s to be executed if test expression is true;
}
```

If the test expression is true then, statements for the body if, i.e, statements inside parenthesis are executed. But, if the test expression is false, the execution of the statements for the body of if statements are skipped.

Flowchart of if statement

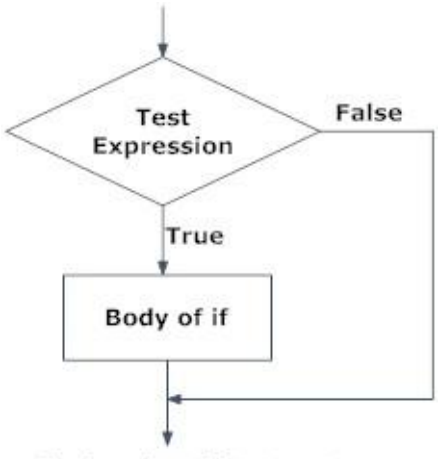


Fig: Operation of if statement

Example of if statement

Write a C program to print the number entered by user only if the number entered is negative.

```
  #include <stdio.h>
int main(){
int num;
printf("Enter a number to check.\n");
scanf("%d",&num);
if(num<0)       /* checking whether number is less than 0 or not. */
printf("Number=%d\n",num);
/*If test condition is true, statement above will be executed, otherwise it will not be executed */
printf("The if statement in C programming is easy.");
return 0;
}
```

Output 1

Enter a number to check.-2
Number=-2
The if statement in C programming is easy.

When user enters -2 then, the test expression (num<0) becomes true. Hence, Number=-2 is displayed in the screen.

Output 2

Enter a number to check.5
The if statement in C programming is easy.

When the user enters 5 then, the test expression (num<0) becomes false. So, the statement for body of if is skipped and only the statement below it is executed.

1.1.2 IF ..ELSE

If...else statement

The if...else statement is used, if the programmer wants to execute some code, if the test expression is true and execute some other code if the test expression is false.

Syntax of if...else

```
if (test expression)
    statements to be executed if test expression is true;
else
    statements to be executed if test expression is false;
```

Flowchart of if...else statement

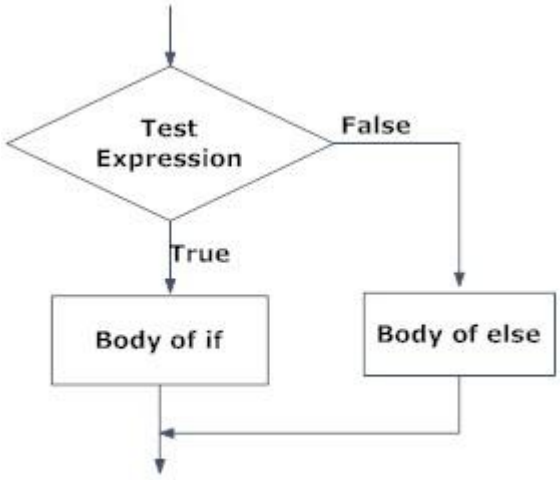


Fig: Operation of if...else statement

Example of if...else statement

Write a C program to check whether a number entered by user is even or odd

```
#include <stdio.h>
int main(){
    int num;
    printf("Enter a number you want to check.\n");
    scanf("%d",&num);
    if((num%2)==0)          //checking whether remainder is 0 or not.
        printf("%d is even.",num);
    else
        printf("%d is odd.",num);
    return 0;
}
```

Output 1

```
Enter a number you want to check.
25
25 is odd.
```

Output 2

```
Enter a number you want to check.
2
```

2 is even.

1.1.3 IF.. ELSEIF..ELSE

Nested if...else statement (if...elseif....else Statement)

The if...else statement can be used in nested form when a serious decision are involved.

Syntax of nested if...else statement.

```
if (test expression)
    statements to be executed if test expression is true;
else
    if(test expression 1)
        statements to be executed if test expressions 1 is true;
      else
        if (test expression 2)
         .
         .
         .
          else
            statements to be executed if all test expressions are false;
```

How nested if...else works?

If the test expression is true, it will execute the code before else part but, if it is false, the control of the program jumps to the else part and check test expression 1 and the process continues. If all the test expression are false then, the last statement is executed.

The ANSI standard specifies that 15 levels of nesting may be continued.

Example of nested if else statement

Write a C program to relate two integers entered by user using = or > or < sign.

```
#include <stdio.h>
int main(){
    int numb1, numb2;
    printf("Enter two integers to check".\n);
    scanf("%d %d",&numb1,&numb2);
    if(numb1==numb2) //checking whether two integers are equal.
        printf("Result: %d=%d",numb1,numb2);
    else
      if(numb1>numb2) //checking whether numb1 is greater than numb2.
        printf("Result: %d>%d",numb1,numb2);
      else
        printf("Result: %d>%d",numb2,numb1);
return 0;
```

```
}
```

Output 1

```
Enter two integers to check.
5
3
Result: 5>3
```

Output 2

```
Enter two integers to check.
-4
-4
Result: -4=-4
```

1.1.4 SWITCH....CASE STATEMENT

Decision making are needed when, the program encounters the situation to choose a particular statement among many statements. If a programmar has to choose one among many alternatives if...else can be used but, this makes programming logic complex. This type of problem can be handled in C programming using switch...case statement.

Syntax of switch...case

```
switch (expression)
{
case constant1:
  codes to be executed if expression equals to constant1;
  break;
case constant2:
  codes to be executed if expression equals to constant3;
  break;
  .
  .
  .
default:
  codes to be executed if expression doesn't match to any cases;
}
```

In switch...case, expression is either an integer or a character. If the value of switch expression matches any of the constant in case, the relevant codes are executed and control moves out of the switch...case statement. If the expression doesn't matches any of the constant in case, then The

default statement is executed.

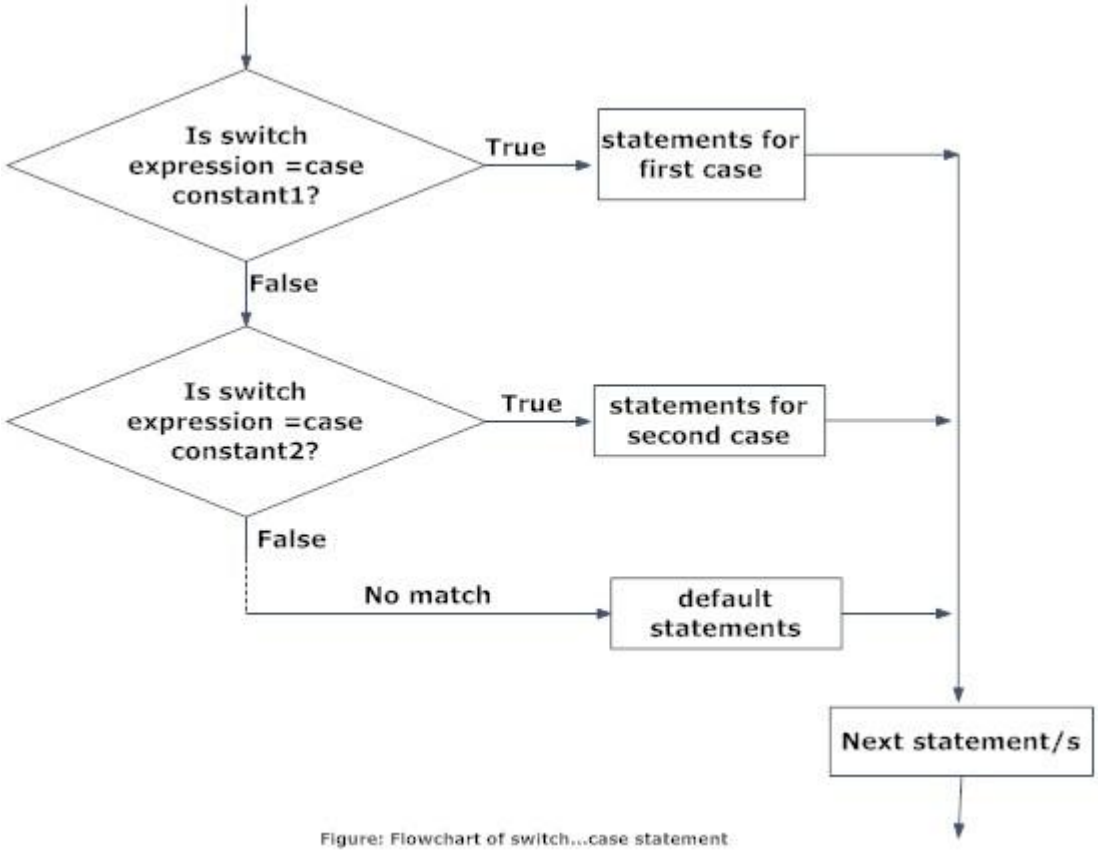


Figure: Flowchart of switch...case statement

Example of switch...case statement

Write a program that asks user an arithmetic operator('+','-','*' or '/') and two operands and perform the corresponding calculation on the operands.

```
/* C program to demonstrate the working of switch...case statement */
/* Program to create a simple calculator for addition, subtraction, multiplication and division */
# include <stdio.h>
int main(){
    char operator;
    float num1,num2;
    printf("Enter operator +, - , * or / :\n");
    operator=getche();
    printf("\nEnter two operands:\n");
    scanf("%f%f",&num1,&num2);
    switch(operator)
    {
    case '+':
            printf("num1+num2=%.2f",num1+num2);
            break;

  case '-':
```

```
            printf("num1-num2=%.2f",num1-num2);
            break;
    case '*':
            printf("num1*num2=%.2f",num1*num2);
            break;
    case '/':
            printf("num2/num1=%.2f",num1/num2);
            break;
    default:
/* if operator is other than +, -, * or /, error message is shown */
            printf(Error! operator is not correct");
            break;
    }
    return 0;
}
```

Output

```
Enter operator +, -, * or / :
/
Enter two operators:
34
3
num2/num1=11.33
```

Notice break statement at the end of each case, which cause switch...case statement to exit. If break statement are not used, all statements below that case statement are also executed.

1.1.5 GOTO

In C programming, goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

Syntax of goto statement

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

In this syntax, label is an identifier. When, the control of program reaches to goto statement, the control of the program will jump to the label: and executes the code/s after it.

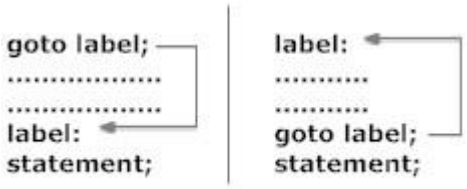


Figure: Working of goto statement

Example of goto statement

```
/* C program to demonstrate the working of goto statement.*/
# include <stdio.h>
int main(){
   float num,average,sum;
   int i,n;
   printf("Maximum no. of inputs: ");
   scanf("%d",&n);
   for(i=1;i<=n;++i){
      printf("Enter n%d: ",i);
      scanf("%f",&num);
      if(num<0.0)
      goto jump;            /* control of the program jumps to label jump */
      sum=sum+num;
}
jump:
  average=sum/(i-1);
  printf("Average: %.2f",average);
  return 0;
}
```

Output

```
Maximum no. of inputs: 4
Enter n1: 1.5
Enter n2: 12.5
Enter n3: 7.2
Enter n4: -1
Average: 7.07
```

Though goto statement is included in ANSI standard of C, use of goto statement should be reduced as much as possible in a program.

Reasons to avoid goto statement

Though, using goto statement give power to jump to any part of program, using goto statement makes the logic of the program complex and tangled. In modern programming, goto statement is considered a harmful construct and a bad programming practice.

The goto statement can be replaced in most of C program with the use of break and continue statements. In fact, any program in C programming can be perfectly written without the use of goto statement. All programmer should try to avoid goto statement as possible as they can.

1.2. CONTROL STATEMENTS

Control statements enable us to specify the flow of program control; ie, the order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.

There are four types of control statements in C:

1. Decision making statements
2. Selection statements
3. Iteration statements
4. Jump statements

1.2.1. Decision Making Statement: the if-else Statement

The if-else statement is used to carry out a logical test and then take one of two possible actions depending on the outcome of the test (ie, whether the outcome is true or false).

Example Discussed above.

1.2.2. Selection Statement: the switch-case Statement

A switch statement is used for **multiple way selections** that will branch into different code segments based on the value of a variable or expression. This expression or variable must be of integer data type

1.2.3. Iteration Statements

Iteration statements are used to execute a particular set of instructions repeatedly until a particular condition is met or for a fixed number of iterations.

THE FOR STATEMENT

The for statement or the for loop repeatedly executes the set of instructions that comprise the body of the for loop until a particular condition is satisfied.

Syntax:

```
for(initialization; termination; increment/decrement)
 {
//statements to be executed
 }
```

The for loop consists of three expressions:

- The **initialization expression**, which initializes the **looping index**. The looping index controls the looping action. The initialization expression is executed only once, when the loop begins.
- The **termination expression**, which represents a condition that must be true for the loop to continue execution.
- The **increment/decrement expression** is executed after every iteration to update the value of the looping index.

The following program uses the for loop to print the series: 0,1,1,2,3,5,8,13 … to n terms.

```
#include<stdio.h>
int main()
{
int i,n, a, b, sum=0;
printf("Enter the number of terms:");
scanf("%d",&n);  a=0; b=1;
 printf("%dn %d",a,b);
for(i=2;i<n;i++)
{
sum=a+b;
printf("n%d",sum);
 a=b;
b=sum;
}
return 0;
}
```

If the first two elements 0 and 1 are excluded, it is seen that each of the next elements in the series is the sum of the previous two elements. For example, the third element is a sum of the first two elements (0+1), the fourth element is a sum of the second and the third elements (1+1=2), the fifth element is a sum of the third and the fourth elements (1+2=3) and so on. Therefore, each time it is the sum of the previous two elements that is printed. The previous value of b becomes the new value of a and the previous value of sum becomes the new value of b. The newly calculated

sum becomes the next element and is subsequently printed. These steps are executed repeatedly until the looping index i does not reach one less than the number of terms entered by the user. The looping index i begins from 2 as the first two terms have already been printed. It could also range from 0 to 1 less than n-2 as:

for(i=0;i<(n-2);i++) or from 1 to n-2 as:
for(i=1; i<(n-2);i++)

The for statement is very versatile and can be written in four different ways:

1. Omitting the Initialization Expression

In this case, the looping index is initialized **before** the for loop. Thus the for loop takes the following form:

i=0;/*initialization*/ for(; condition;increment/decrement expression) { //statements to be executed }

Notice that the semicolon that terminates the initialization expression is present before the condition expression.

2. Omitting the Condition

In this case the condition is specified **inside** the body of the for loop, generally using an if statement. The while or do-while statements may also be used to specify the condition. Thus the for loop takes the following form:

for(initialization; ; increment/decrement expression) { condition //statements to be executed }

Again, it can be seen that the semicolon that terminates the condition is present in the for statement. The following program explains how the condition can be omitted:

```
#include<stdio.h>
 int main()
 {
int i,n, a, b, sum=0;
 printf("Enter the number of terms:");
scanf("%d",&n);
a=0; b=1;
printf("%dn %d",a,b);
for(i=2; ;i++)
{
 if(i==(n-1)) //condition specified using if statement
break;
sum=a+b;
printf("n%d",sum);
 a=b;
```

```
 b=sum;
}
 return 0;
}
```

3. Omitting the increment /decrement Expression:

In this case the increment/decrement expression is written **inside the body** of the for loop.

Example:

```
#include<stdio.h>
int main()
{
int i,n, a, b,
sum=0;
 printf("Enter the number of terms:");
scanf("%d",&n); a=0; b=1;
printf("%dn %d",a,b);
 for(i=2;i<n;)
{
sum=a+b;
 printf("n%d",sum);
 a=b;
 b=sum;
i++;  //increment expression
}
return 0;
 }
```

4. Omitting all Three Expressions:

It is also possible to omit all three expressions, but they should be present in the ways discussed above. If all three expressions are omitted entirely — ie, they are not mentioned in the ways discussed above — then the for loop becomes an **infinite or never-ending loop**. In this case, the for loop takes the following form:

```
for(;;)  { //statements }
```

2. The while statement

The while statement executes a block of statements repeatedly while a particular condition is true.

```
while (condition)  { //statement(s) to be executed }
```

The statements are executed repeatedly until the condition is true.

Example: Program to calculate the sum of the digits of a number (eg, 456; 4+5+6 = 15)

```
#include<stdio.h>
 int main()
{
int n, a,sum=0;
 printf("n Enter a number:");
scanf("%d", &n);
 while(n>0)
{
a=n%10; //extract the digits of the number
sum=sum+a; //sum the digits
 n=n/10; //calculate the quotient of a number when divided by 10.
}
printf("n Sum of the digits=t %d",sum);
return 0;
}
```

The above program uses the while loop to calculate the sum of the digits of a number. For example, if the number is 456, the while loop will calculate the sum in four iterations as follows. It would be helpful to remember that % gives the remainder and / the quotient.

Iteration 1: n>0 Condition is true(n=456)

a=n%10=6;

sum=sum+a=6;

n=n/10= 45; New value of n is 45.

Iteration 2: n>0 Condition is true(n=45)

a=n%10=5;

sum=sum+a=6+5=11;

n=n/10= 4; New value of n is 4.

Iteration 3: n>0 Condition is true(n=4)

a=n%10=4;

sum=sum+a=11+4=15;

n=n/10= 0; ew value of n is 0.

Iteration 4: n>0 Condition is false(n=0).

After the fourth iteration control exits the while loop and prints the sum to be 15.

Example 2: Program to check whether the entered number is a palindrome or not.

A palindrome is a number which remains the same when its digits are read or written from right to left or vice versa, eg 343 is a palindrome, but 342 is not. The following program works on the logic that if the reverse of the number is same as the original number then the entered number is a palindrome, otherwise it is not.

```
#include<stdio.h>
int main()
{
int a, n,m, reverse=0;
printf("n Enter a number:");
scanf("%d", &n);
m=n;
while(n>0)
{
a=n%10;
reverse=reverse*10 +a;
n=n/10;
}
if (m== reverse)
{
printf(" n The number is a palindrome.");
}
else
{
printf("n The number is not a palindrome.");
}
return 0;
}
```

The above program uses almost the same logic as the program concerning the sum of digits. As was seen in that program, n becomes 0 in the last iteration. However, we need to compare the original value of n to the reverse of the number to determine whether it is a palindrome or not. Therefore, the value of n has been stored in m, **before** entering the while loop. The value of m is later compared with reverse to decide whether the entered number is a palindrome or not.

The while loop works in the following way:
Let n=343;
Iteration 1: a= n%10=3;
reverse=reverse*10+a=0*10+3=3;
n=n/10=34;
Iteration 2: a= n%10=4;
reverse=reverse*10+a=3*10+4=34;

n=n/10=3;
Iteration 3: a= n%10=3;
reverse=reverse*10+a=34*10+3=343;
n=n/10=0;
Iteration 4: n>0 condition false(n=0).
Control exits from the while loop.

3. The do-while loop

The do-while statement evaluates the condition at the end of the loop after executing the block of statements at least once. If the condition is true the loop continues, else it terminates after the first iteration.

Syntax:

```
do
{
 //statements to be executed;
 } while(condition);
```

Note the semicolon which ends the do-while statement. The difference between while and do-while is that the while loop is an **entry-controlled loop** — it tests the condition at the beginning of the loop and will not execute even once if the condition is false, whereas the do-while loop is an **exit-controlled loop** — it tests the condition at the end of the loop after completing the first iteration.

For many applications it is more natural to test for the continuation of a loop at the beginning rather than at the end of the loop. For this reason, the do-while statement is used less frequently than the while statement.

Most programs that work with while can also be implemented using do-while. The following program calculates the sum of digits in the same manner, except that it uses the do-while loop:

```
#include<stdio.h>
 int main()
 {
  int n, a,sum=0;
  printf("n Enter a number:");
  scanf("%d", &n);
  do
  {
   a=n%10;
   sum=sum+a;
   n=n/10;
  }while(n>0);
```

```
  printf("n Sum of the digits=t %d",sum);
  return 0;
 }
```

However, the do-while statement should be used only in situations where the loop must execute at least once whether or not the condition is true.

A practical use of the do-while loop is in an interactive menu-driven program where the menu is presented at least once, and then depending upon the choice of the user, the menu is displayed again or the session is terminated. Consider the same example that we saw in switch-case. Without using an iteration statement like do-while, the user can choose any option from the menu only once. Moreover, if a wrong choice is entered by mistake the user doesn't have the option of entering his choice again. Both these faults can be corrected by using the do-while loop.

1.3. FUNCTIONS

A function is a module or block of program code which deals with a particular task. Making functions is a way of isolating one block of code from other independent blocks of code.

Functions serve two purposes.

- They allow a programmer to say: `this piece of code does a specific job which stands by itself and should not be mixed up with anyting else',
- Second they make a block of code reusable since a function can be reused in many different contexts without repeating parts of the program text.

A function can take a number of parameters, do required processing and then return a value. There may be a function which does not return any value.

You already have seen couple of built-in functions like printf(); Similar way you can define your own functions in C language.

Consider the following chunk of code

```
  int total = 10;
  printf("Hello World");
  total = total + l;
```

To turn it into a function you simply wrap the code in a pair of curly brackets to convert it into a single compound statement and write the name that you want to give it in front of the brackets:

```
Demo()
{
  int total = 10;
  printf("Hello World");
  total = total + l;
```

```
}
```

curved brackets after the function's name are required. You can pass one or more paramenters to a function as follows:

```
Demo( int par1, int par2)
{
   int total = 10;
   printf("Hello World");
   total = total + l;
}
```

By default function does not return anything. But you can make a function to return any value as follows:

```
int Demo( int par1, int par2)
{
   int total = 10;
   printf("Hello World");
   total = total + l;

   return total;
}
```

A *return* keyword is used to return a value and datatype of the returned value is specified before the name of function. In this case function returns *total* which is *int* type. If a function does not return a value then *void* keyword can be used as return value.

Once you have defined your function you can use it within a program:

```
main()
{
  Demo();
}
```

Functions and Variables:

Each function behaves the same way as C language standard function *main()*. So a function will have its own local variables defined. In the above example *total* variable is local to the function Demo.

A global variable can be accessed in any function in similar way it is accessed in *main()* function.

Declaration and Definition

When a function is defined at any place in the program then it is called function definition. At the time of definition of a function actual logic is implemented with-in the function.

A function declaration does not have any body and they just have their interfaces.

A function declaration is usually declared at the top of a C source file, or in a separate header file.

A function declaration is sometime called function prototype or function signature. For the above *Demo()* function which returns an integer, and takes two parameters a function declaration will be as follows:

```
int Demo( int par1, int par2);
```

Passing Parameters to a Function

There are two ways to pass parameters to a function:

- **Pass by Value:** mechanism is used when you don't want to change the value of passed paramters. When parameters are passed by value then functions in C create copies of the passed in variables and do required processing on these copied variables.
- **Pass by Reference** mechanism is used when you want a function to do the changes in passed parameters and reflect those changes back to the calling function. In this case only addresses of the variables are passed to a function so that function can work directly over the addresses.

Here are two programs to understand the difference: First example is for *Pass by value:*

```
#include <stdio.h>

/* function declaration goes here.*/
void swap( int p1, int p2 );

int main()
{
   int a = 10;
   int b = 20;

   printf("Before: Value of a = %d and value of b = %d\n", a, b );
   swap( a, b );
   printf("After: Value of a = %d and value of b = %d\n", a, b );
}

void swap( int p1, int p2 )
```

```
{
   int t;

   t = p2;
   p2 = p1;
   p1 = t;
  printf("Value of a (p1) = %d and value of b(p2) = %d\n", p1, p2 );
}
```

Here is the result produced by the above example. Here the values of a and b remain unchanged before calling *swap* function and after calling *swap* function.

```
Before: Value of a = 10 and value of b = 20
Value of a (p1) = 20 and value of b(p2) = 10
After: Value of a = 10 and value of b = 20
```

Following is the example which demonstrate the concept of pass by reference

```
#include <stdio.h>

/* function declaration goes here.*/
void swap( int *p1, int *p2 );

int main()
{
  int a = 10;
  int b = 20;

  printf("Before: Value of a = %d and value of b = %d\n", a, b );
  swap( &a, &b );
  printf("After: Value of a = %d and value of b = %d\n", a, b );
}

void swap( int *p1, int *p2 )
{
   int t;

   t = *p2;
   *p2 = *p1;
   *p1 = t;
  printf("Value of a (p1) = %d and value of b(p2) = %d\n", *p1, *p2 );
}
```

Here is the result produced by the above example. Here the values of a and b are changes after calling *swap* function.

```
Before: Value of a = 10 and value of b = 20
Value of a (p1) = 20 and value of b(p2) = 10
```

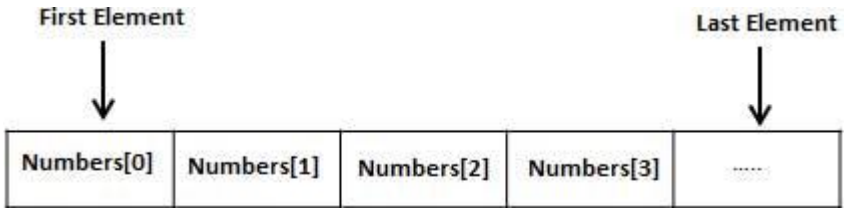
After: Value of a = 20 and value of b = 10

1.4. ARRAYS

C programming language provides a data structure called **the array**, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

type arrayName [arraySize];

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement:

double balance[10];

Now *balance* is avariable array which is sufficient to hold upto 10 double numbers.

Initializing Arrays

You can initialize array in C either one by one or using a single statement as follows:

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

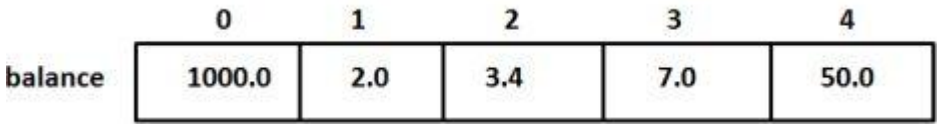
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th ie. last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>
 int main ()
{
   int n[ 10 ]; /* n is an array of 10 integers */
   int i,j;
   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ )
   {
      n[ i ] = i + 100; /* set element at location i to i + 100 */
   }
   /* output each array element's value */
   for (j = 0; j < 10; j++ )
```

```
   {
      printf("Element[%d] = %d\n", j, n[j] );
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

C Arrays in Detail

Arrays are important to C and should need lots of more details. There are following few important concepts related to array which should be clear to a C programmer:

Concept	Description
Multi-dimensional arrays	C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.
Return array from a function	C allows a function to return an array.
Pointer to an array	You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

Multidimensional Array

Arrays of Arrays (``Multidimensional'' Arrays)

When we said that ``Arrays are not limited to type int; you can have arrays of... any other type," we meant that more literally than you might have guessed. If you have an ``array of int," it means that you have an array each of whose elements is of type int. But you can have an array each of whose elements is of type *x*, where *x* is any type you choose. In particular, you can have an array each of whose elements is another array! We can use these arrays of arrays for the same sorts of tasks as we'd use multidimensional arrays in other computer languages (or matrices in mathematics). Naturally, we are not limited to arrays of arrays, either; we could have an array of arrays of arrays, which would act like a 3-dimensional array, etc.

The declaration of an array of arrays looks like this:

```
int a2[5][7];
```

You have to read complicated declarations like these ``inside out." What this one says is that a2 is an array of 5 somethings, and that each of the somethings is an array of 7 ints. More briefly, ``a2 is an array of 5 arrays of 7 ints," or, ``a2 is an array of array of int." In the declaration of a2, the brackets closest to the identifier a2 tell you what a2 first and foremost is. That's how you know it's an array of 5 arrays of size 7, not the other way around. You can think of a2 as having 5 ``rows" and 7 ``columns," although this interpretation is not mandatory. (You could also treat the ``first" or inner subscript as ``x" and the second as ``y." Unless you're doing something fancy, all you have to worry about is that the subscripts when you access the array match those that you used when you declared it, as in the examples below.)

To illustrate the use of multidimensional arrays, we might fill in the elements of the above array a2 using this piece of code:

```
int i, j;
for(i = 0; i < 5; i = i + 1)
        {
        for(j = 0; j < 7; j = j + 1)
                a2[i][j] = 10 * i + j;
        }
```

This pair of nested loops sets a[1][2] to 12, a[4][1] to 41, etc. Since the first dimension of a2 is 5, the first subscripting index variable, i, runs from 0 to 4. Similarly, the second subscript varies from 0 to 6. We could print a2 out (in a two-dimensional way, suggesting its structure) with a similar pair of nested loops:

```
for(i = 0; i < 5; i = i + 1)
        {
        for(j = 0; j < 7; j = j + 1)
                printf("%d\t", a2[i][j]);
        printf("\n");
        }
```

(The character \t in the printf string is the tab character.)

Just to see more clearly what's going on, we could make the ``row" and ``column" subscripts explicit by printing them, too:

```
for(j = 0; j < 7; j = j + 1)
```

```
            printf("\t%d:", j);
        printf("\n");

        for(i = 0; i < 5; i = i + 1)
            {
            printf("%d:", i);
            for(j = 0; j < 7; j = j + 1)
                printf("\t%d", a2[i][j]);
            printf("\n");
            }
```

This last fragment would print

```
     0:   1:   2:   3:   4:   5:   6:
0:   0    1    2    3    4    5    6
1:   10   11   12   13   14   15   16
2:   20   21   22   23   24   25   26
3:   30   31   32   33   34   35   36
4:   40   41   42   43   44   45   46
```

Finally, there's no reason we have to loop over the ``rows" first and the ``columns" second; depending on what we wanted to do, we could interchange the two loops, like this:

```
        for(j = 0; j < 7; j = j + 1)
            {
            for(i = 0; i < 5; i = i + 1)
                printf("%d\t", a2[i][j]);
            printf("\n");
            }
```

Notice that i is still the first subscript and it still runs from 0 to 4, and j is still the second subscript and it still runs from 0 to 6.

1.5. C Preprocessor

The **C Preprocessor** is not part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP.

All preprocessor commands begin with a pound symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column. Following section lists down all important preprocessor directives:

Directive	Description
#define	Substitutes a preprocessor macro
#include	Inserts a particular header from another file

#undef	Undefines a preprocessor macro
#ifdef	Returns true if this macro is defined
#ifndef	Returns true if this macro is not defined
#if	Tests if a compile time condition is true
#else	The alternative for #if
#elif	#else an #if in one statement
#endif	Ends preprocessor conditional
#error	Prints error message on stderr
#pragma	Issues special commands to the compiler, using a standardized method

2. Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

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

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

This tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
  #define MESSAGE "You wish!"
#endif
```

This tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
  /* Your debugging statements here */
```

```
#endif
```

This tells the CPP to do the process the statements enclosed if DEBUG is defined. This is useful if you pass the *-DDEBUG* flag to gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

3. Predefined Macros

ANSI C defines a number of macros. Although each one is available for your use in programming, the predefined macros should not be directly modified.

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format
__TIME__	The current time as a character literal in "HH:MM:SS" format
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.
__STDC__	Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example:

```
#include <stdio.h>

main()
{
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );

}
```

When the above code in a file **test.c** is compiled and executed, it produces the following result:

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

The C preprocessor offers following operators to help you in creating macros:

Macro Continuation (\)

A macro usually must be contained on a single line. The macro continuation operator is used to continue a macro that is too long for a single line. For example:

```
#define  message_for(a, b)  \
   printf(#a " and " #b ": We love you!\n")
```

Stringize (#)

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro that has a specified argument or parameter list. For example:

```
#include <stdio.h>

#define  message_for(a, b)  \
   printf(#a " and " #b ": We love you!\n")

int main(void)
{
  message_for(Carole, Debra);
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Carole and Debra: We love you!
```

Token Pasting (##)

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example:

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void)
{
  int token34 = 40;

  tokenpaster(34);
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
token34 = 40
```

How it happened, because this example results in the following actual output from the preprocessor:

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**.

The defined() Operator

The preprocessor **defined** operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows:

```
#include <stdio.h>

#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif

int main(void)
{
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Here is the message: You wish!
```

Parameterized Macros

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows:

```
int square(int x)
{
   return x * x;
}
```

We can rewrite above code using a macro as follows:

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the **#define** directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between and macro name and open parenthesis. For example:

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
  printf("Max between 20 and 10 is %d\n", MAX(10, 20));
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Max between 20 and 10 is 20
```

1.6. POINTERS

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

```
#include <stdio.h>
int main ()
{
  int  var1;
  char var2[10];
  printf("Address of var1 variable: %x\n", &var1  );
  printf("Address of var2 variable: %x\n", &var2  );
  return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6

So you understood what is memory address and how to access it, so base of the concept is over. Now let us see what is a pointer.

What Are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

type *var-name;

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

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

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. **(a)** we define a pointer variable **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

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

```
   printf("Address stored in ip variable: %x\n", ip );
   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );
   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers in C

It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

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

When the above code is compiled and executed, it produces the following result:

```
The value of ptr is 0
```

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows:

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

C Pointers in Detail:

Pointers have many but easy concepts and they are very important to C programming. There are following few important pointer concepts which should be clear to a C programmer:

Concept	Description
C - Pointer arithmetic	There are four arithmetic operators that can be used on pointers: ++, --, +, -
C - Array of pointers	You can define arrays to hold a number of pointers.
C - Pointer to pointer	C allows you to have pointer on a pointer and so on.
Passing pointers to functions in C	Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.
Return pointer from functions in C	C allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.

1.8 FUNCTION POINTERS

C programming language allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

```
#include <stdio.h>
#include <time.h>
 void getSeconds(unsigned long *par);
int main ()
{
   unsigned long sec;
   getSeconds( &sec );
   /* print the actual value */
   printf("Number of seconds: %ld\n", sec );
   return 0;
}
void getSeconds(unsigned long *par)
{
   /* get the current number of seconds */
   *par = time( NULL );
   return;
}
```

When the above code is compiled and executed, it produces the following result:Number of

seconds : 1294450468

The function, which can accept a pointer, can also accept an array as shown in the following example:

```
#include <stdio.h>

 /* function declaration */
double getAverage(int *arr, int size);
 int main ()
{
   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
    /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;

   /* output the returned value  */
   printf("Average value is: %f\n", avg );
       return 0;
}

double getAverage(int *arr, int size)
{
  int    i, sum = 0;
  double avg;
   for (i = 0; i < size; ++i)
  {
    sum += arr[i];
  }
   avg = (double)sum / size;
   return avg;
}
```

When the above code is compiled together and executed, it produces the following result:

Average value is: 214.40000

1.9 FUCTION WITH VARIABLE NUMBER OF ARGUMENTS

Sometimes, you may come across a situation, when you want to have a function, which can take variable number of arguments, i.e., parameters, instead of predefined number of parameters. The C programming language provides a solution for this situation and you are allowed to define a function which can accept variable number of parameters based on your requirement. The following example shows the definition of such a function.

```
int func(int, ... )
{
   .
```

```
   . .
}

int main()
{
  func(1, 2, 3);
  func(1, 2, 3, 4);
}
```

It should be noted that function **func()** has last argument as ellipses i.e. three dotes (**...**) and the one just before the ellipses is always an **int** which will represent total number variable arguments passed. To use such functionality you need to make use of **stdarg.h** header file which provides functions and macros to implement the functionality of variable arguments and follow the following steps:

- Define a function with last parameter as ellipses and the one just before the ellipses is always an **int** which will represent number of arguments.
- Create a **va_list** type variable in the function definition. This type is defined in stdarg.h header file.
- Use **int** parameter and **va_start** macro to initialize the **va_list** variable to an argument list. The macro va_start is defined in stdarg.h header file.
- Use **va_arg** macro and **va_list** variable to access each item in argument list.
- Use a macro **va_end** to clean up the memory assigned to **va_list** variable.

Now let us follow the above steps and write down a simple function which can take variable number of parameters and returns their average:

```
#include <stdio.h>
#include <stdarg.h>
double average(int num,...)
{
   va_list valist;
   double sum = 0.0;
   int i;
   /* initialize valist for num number of arguments */
   va_start(valist, num);
   /* access all the arguments assigned to valist */
   for (i = 0; i < num; i++)
   {
     sum += va_arg(valist, int);
   }
   /* clean memory reserved for valist */
   va_end(valist);

   return sum/num;
}
```

```
int main()
{
   printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
   printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

When the above code is compiled and executed, it produces the following result. It should be noted that the function **average()** has been called twice and each time first argument represents the total number of variable arguments being passed. Only ellipses will be used to pass variable number of arguments.

UNIT II
C PROGRAMMING ADVANCED FEATURES

TOPICS DISCUSSED

1. Structures and Unions
2. File handling concepts
3. File read
4. File write
5. Binary File
6. Stdio
7. File Manipulations

2.1. STRUCTURES

C arrays allow you to define type of variables that can hold several data items of the same kind but **structure** is another user defined data type available in C programming, which allows you to combine data items of different kinds.

Structures are used to represent a record, Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member for your program. The format of the struct statement is this:

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure:

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

struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};

int main( )
{
  struct Books Book1;        /* Declare Book1 of type Book */
  struct Books Book2;        /* Declare Book2 of type Book */

  /* book 1 specification */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "Nuha Ali");
  strcpy( Book1.subject, "C Programming Tutorial");
  Book1.book_id = 6495407;

  /* book 2 specification */
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Zara Ali");
  strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;
```

```
   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example:

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

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* function declaration */
void printBook( struct Books book );
int main( )
{
```

```
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printBook( Book1 );
   /* Print Book2 info */
   printBook( Book2 );
   return 0;
}
void printBook( struct Books book )
{
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

struct_pointer = &Book1;

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

struct_pointer->title;

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

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

struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};

/* function declaration */
void printBook( struct Books *book );
int main( )
{
  struct Books Book1;        /* Declare Book1 of type Book */
  struct Books Book2;        /* Declare Book2 of type Book */

  /* book 1 specification */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "Nuha Ali");
  strcpy( Book1.subject, "C Programming Tutorial");
  Book1.book_id = 6495407;

  /* book 2 specification */
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Zara Ali");
  strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;

  /* print Book1 info by passing address of Book1 */
  printBook( &Book1 );

  /* print Book2 info by passing address of Book2 */
```

```
   printBook( &Book2 );

   return 0;
}
void printBook( struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

- Packing several objects into a machine word. e.g. 1 bit flags can be compacted.
- Reading external file formats -- non-standard file formats could be read in. E.g. 9 bit integers.

C allows us do this in a structure definition by putting :bit length after the variable. For example:

```
struct packed_struct {
  unsigned int f1:1;
  unsigned int f2:1;
  unsigned int f3:1;
  unsigned int f4:1;
  unsigned int type:4;
  unsigned int my_int:9;
} pack;
```

Here, the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4 bit type and a 9 bit my_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this

is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word.

2.2. UNION

A **union** is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

Defining a Union

To define a union, you must use the **union** statement in very similar was as you did while defining structure. The union statement defines a new data type, with more than one member for your program. The format of the union statement is as follows:

```
union [union tag]
{
  member definition;
  member definition;
  ...
  member definition;
} [one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data which has the three members i, f, and str:

```
union Data
{
  int i;
  float f;
  char  str[20];
} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. This means that a single variable ie. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

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

```
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;
   printf( "Memory size occupied by data : %d\n", sizeof(data));
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Memory size occupied by data : 20

Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use **union** keyword to define variables of union type. Following is the example to explain usage of union:

```
#include <stdio.h>
#include <string.h>
 union Data
{
   int i;
   float f;
   char  str[20];
};
 int main( )
{
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i :          1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that values of **i** and **f** members of union got corrupted because final value assigned to the variable has occupied the memory location and this is the reason that the value if **str** member is getting printed very well. Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having union:

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

union Data
{
   int i;
   float f;
   char  str[20];
};
 int main( )
{
   union Data data;
   data.i = 10;
   printf( "data.i : %d\n", data.i);
    data.f = 220.5;
   printf( "data.f : %f\n", data.f);
    strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

2.3. FILE HANDLING CONCEPTS

In this section, we will discuss about files which are very important for storing information permanently. We store information in files for many purposes, like data processing by our programs.

What is a File?

Abstractly, a file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file. It is conceivable (and it sometimes happens) that a graphics file will be read and displayed by a program designed to process textual data. The result is that no meaningful output occurs (probably) and this is to be expected. A file is simply a machine decipherable storage media where programs and data are stored for machine usage.

Essentially there are two kinds of files that programmers deal with text files and binary files. These two classes of files will be discussed in the following sections.

ASCII Text files

A text file can be a stream of characters that a computer can process sequentially. It is not only processed sequentially but only in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time.

Similarly, since text files only process characters, they can only read or write data one character at a time. (In C Programming Language, Functions are provided that deal with lines of text, but these still essentially process data one character at a time.) A text stream in C is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending on whether data is being written to, or read from, the file. Other character conversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signalled the intention to process a text file.

Binary files

A binary file is no different to a text file. It is a collection of bytes. In C Programming Language a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

1. No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
2. C Programming Language places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be either processed sequentially or, depending on the needs of the application, they can be processed using random access techniques. In C Programming Language, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files.
They a generally processed using read and write operations simultaneously.

For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common to many binary files, but are rarely found in applications that process text files.

Creating a file and output some data

In order to create files we have to learn about File I/O i.e. how to write data into a file and how to read data from a file. We will start this section with an example of writing data to a file. We begin as before with the include statement for stdio.h, then define some variables for use in the example including a rather strange looking new type.

```
/* Program to create a file and write some data the file */
#include <stdio.h>
#include <stdio.h>
main( )
{
    FILE *fp;
    char stuff[25];
    int index;
    fp = fopen("TENLINES.TXT","w"); /* open for writing */
    strcpy(stuff,"This is an example line.");
    for (index = 1; index <= 10; index++)
            fprintf(fp,"%s Line number %d\n", stuff, index);
    fclose(fp); /* close the file before ending program */
}
```

The type FILE is used for a file variable and is defined in the stdio.h file. It is used to define a file pointer for use in file operations. Before we can write to a file, we must open it. What this really means is that we must tell the system that we want to write to a file and what the file name is. We do this with the fopen() function illustrated in the first line of the program. The file pointer, fp in our case, points to the file and two arguments are required in the parentheses, the file name first, followed by the file type.

The file name is any valid DOS file name, and can be expressed in upper or lower case letters, or even mixed if you so desire. It is enclosed in double quotes. For this example we have chosen the name TENLINES.TXT. This file should not exist on your disk at this time. If you have a file with this name, you should change its name or move it because when we execute this program, its contents will be erased. If you donâ€™t have a file by this name, that is good because we will

create one and put some data into it. You are permitted to include a directory with the file name.The directory must, of course, be a valid directory otherwise an error will occur. Also, because of the way C handles literal strings, the directory separation character â€˜\â€™ must be written twice. For example, if the file is to be stored in the \PROJECTS sub directory then the file name should be entered as "\\PROJECTS\\TENLINES.TXT". The second parameter is the file attribute and can be any of three letters, r, w, or a, and must be lower case.

Reading (r)

When an r is used, the file is opened for reading, a w is used to indicate a file to be used for writing, and an a indicates that you desire to append additional data to the data already in an existing file. Most C compilers have other file attributes available; check your Reference Manual for details. Using the r indicates that the file is assumed to be a text file. Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program.

Here is a small program that reads a file and display its contents on screen.

```
/* Program to display the contents of a file on screen */
#include <stdio.h>
void main()
{
  FILE *fopen(), *fp;
  int c;
  fp = fopen("prog.c","r");
  c = getc(fp) ;
  while (c!= EOF)
  {
                    putchar(c);
                    c = getc(fp);
  }
  fclose(fp);
}
```

Writing (w)

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does, resulting in the deletion of any data already there. Using the w indicates that the file is assumed to be a text file.

Here is the program to create a file and write some data into the file.

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

```
 /*Create a file and add text*/
 fprintf(fp,"%s","This is just an example :)"); /*writes data to the file*/
 fclose(fp); /*done!*/
 return 0;
}
```

Appending (a)

When a file is opened for appending, it will be created if it does not already exist and it will be initially empty. If it does exist, the data input point will be positioned at the end of the present data so that any new data will be added to any data that already exists in the file. Using the a indicates that the file is assumed to be a text file.

Here is a program that will add text to a file which already exists and there is some text in the file.

```
#include <stdio.h>
int main()
{
   FILE *fp
   file = fopen("file.txt","a");
   fprintf(fp,"%s","This is just an example :)"); /*append some text*/
   fclose(fp);
   return 0;
}
```

Outputting to the file

The job of actually outputting to the file is nearly identical to the outputting we have already done to the standard output device. The only real differences are the new function names and the addition of the file pointer as one of the function arguments. In the example program, fprintf replaces our familiar printf function name, and the file pointer defined earlier is the first argument within the parentheses. The remainder of the statement looks like, and in fact is identical to, the printf statement.

Closing a file

To close a file you simply use the function fclose with the file pointer in the parentheses. Actually, in this simple program, it is not necessary to close the file because the system will close all open files before returning to DOS, but it is good programming practice for you to close all files in spite of the fact that they will be closed automatically, because that would act as a reminder to you of what files are open at the end of each program.

You can open a file for writing, close it, and reopen it for reading, then close it, and open it again for appending, etc. Each time you open it, you could use the same file pointer, or you could use a different one. The file pointer is simply a tool that you use to point to a file and you decide what file it will point to. Compile and run this program. When you run it, you will not get any output to the monitor because it doesnâ€™t generate any. After running it, look at your directory for a file

named TENLINES.TXT and type it; that is where your output will be. Compare the output with that specified in the program; they should agree! Do not erase the file named TENLINES.TXT yet; we will use it in
some of the other examples in this section.

Opening Files

You can use the **fopen( )** function to create a new file or to open an existing file, this call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. Following is the prototype of this function call:

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

Here, **filename** is string literal, which you will use to name your file and access **mode** can have one of the following values:

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for reading and writing both.
w+	Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
a+	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

"rb", "wb", "ab", "ab+", "a+b", "wb+", "w+b", "ab+", "a+b"

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is:

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

The **fclose( )** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.

Writing a File

Following is the simplest function to write individual characters to a stream:

```
int fputc( int c, FILE *fp );
```

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream:

```
int fputs( const char *s, FILE *fp );
```

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example:

```
#include <stdio.h>
main()
{
   FILE *fp;
   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in next section.

Reading a File

Following is the simplest function to read a single character from a file:

```
int fgetc( FILE * fp );
```

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error it returns **EOF**. The following functions allow you to read a string from a stream:

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that

point including new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file but it stops reading after the first space character encounters.

```
#include <stdio.h>
main()
{
   FILE *fp;
   char buff[255];
   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );
   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );
   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

When the above code is compiled and executed, it reads the file created in previous section and produces the following result:

```
1 : This
2: is testing for fprintf...

3: This is testing for fputs...
```

Let's see a little more detail about what happened here. First **fscanf()** method read just thi**s** because after that it encountered a space, second call is for **fgets()** which read the remaining line till it encountered end of line. Finally last call **fgets()** read second line completely.

Binary I/O Functions

There are following two functions, which can be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
        size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
        size_t number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

Now for our first program that reads from a file. This program begins with the familiar include, some data definitions, and the file opening statement which should require no explanation except for the fact that an r is used here because we want to read it.

```
#include <stdio.h>
  main( )
  {
   FILE *fp;
   char c;
   funny = fopen("TENLINES.TXT", "r");
   if (fp == NULL)
                        printf("File doesn't exist\n");
   else {
    do {
     c = getc(fp); /* get one character from the file
     */
       putchar(c); /* display it on the monitor
     */
     } while (c != EOF); /* repeat until EOF (end of file)
    */
    }
   fclose(fp);
  }
```

In this program we check to see that the file exists, and if it does, we execute the main body of the program. If it doesn't, we print a message and quit. If the file does not exist, the system will set the pointer equal to NULL which we can test. The main body of the program is one do while loop in which a single character is read from the file and output to the monitor until an EOF (end of file) is detected from the input file. The file is then closed and the program is terminated. At this point, we have the potential for one of the most common and most perplexing problems of programming in C. The variable returned from the getc function is a character, so we can use a char variable for this purpose. There is a problem that could develop here if we happened to use an unsigned char however, because C usually returns a minus one for an EOF – which an unsigned char type variable is not
capable of containing. An unsigned char type variable can only have the values of zero to 255, so it will return a 255 for a minus one in C. This is a very frustrating problem to try to find. The program can never find the EOF and will therefore never terminate the loop. This is easy to prevent: always have a char or int type variable for use in returning an EOF. There is another problem with this program but we will worry about it when we get to the next program and solve it with the one following that.

After you compile and run this program and are satisfied with the results, it would be a good exercise to change the name of TENLINES.TXT and run the program again to see that the NULL test actually works as stated. Be sure to change the name back because we are still not finished with TENLINES.TXT.

Opening Files

You can use the **fopen( )** function to create a new file or to open an existing file, this call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. Following is the prototype of this function call:

FILE *fopen(const char * filename, const char * mode);

Here, **filename** is string literal, which you will use to name your file and access **mode** can have one of the following values:

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for reading and writing both.
w+	Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
a+	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

"rb", "wb", "ab", "ab+", "a+b", "wb+", "w+b", "ab+", "a+b"

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is:

int fclose(FILE *fp);

The **fclose( )** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.

Writing a File

Following is the simplest function to write individual characters to a stream:

int fputc(int c, FILE *fp);

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream:

```
int fputs( const char *s, FILE *fp );
```

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example:

```
#include <stdio.h>

main()
{
  FILE *fp;

  fp = fopen("/tmp/test.txt", "w+");
  fprintf(fp, "This is testing for fprintf...\n");
  fputs("This is testing for fputs...\n", fp);
  fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in next section.

Reading a File

Following is the simplest function to read a single character from a file:

```
int fgetc( FILE * fp );
```

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error it returns **EOF**. The following functions allow you to read a string from a stream:

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file but it stops reading after the first space character encounters.

```
#include <stdio.h>

main()
```

```
{
  FILE *fp;
  char buff[255];
  fp = fopen("/tmp/test.txt", "r");
  fscanf(fp, "%s", buff);
  printf("1 : %s\n", buff );
  fgets(buff, 255, (FILE*)fp);
  printf("2: %s\n", buff );
  fgets(buff, 255, (FILE*)fp);
  printf("3: %s\n", buff );
  fclose(fp);
}
```

When the above code is compiled and executed, it reads the file created in previous section and produces the following result:

```
1 : This
2: is testing for fprintf...
3: This is testing for fputs...
```

Let's see a little more detail about what happened here. First **fscanf()** method read just this because after that it encountered a space, second call is for **fgets()** which read the remaining line till it encountered end of line. Finally last call **fgets()** read second line completely.

Binary I/O Functions

There are following two functions, which can be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
        size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
        size_t number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

1. **Redirection:**

   One way to get input into a program or to display output from a program is to use ***standard input*** and ***standard output***, respectively. All that means is that to read in data, we use scanf() (or a few other functions) and to write out data, we use printf().

   When we need to take input from a file (instead of having the user type data at the keyboard) we can use input redirection:

% **a.out < inputfile**

This allows us to use the same scanf() calls we use to read from the keyboard. With input redirection, the operating system causes input to come from the file (e.g., inputfile above) instead of the keyboard.

Similarly, there is ***output redirection*** :

% **a.out > outputfile**

that allows us to use printf() as before, but that causes the output of the program to go to a file (e.g., outputfile above) instead of the screen.

Of course, the 2 types of redirection can be used at the same time.

% **a.out < inputfile > outputfile**

2. **C File I/O:**

While redirection is very useful, it is really part of the operating system (not C). In fact, C has a general mechanism for reading and writing files, which is more flexible than redirection alone.

stdio.h

There are types and functions in the library **stdio.h** that are used for file I/O. Make sure you always include that header when you use files.

Type

For files you want to read or write, you need a file pointer, e.g.:

FILE *fp;

What is this type "FILE *"? Realistically, you don't need to know. Just think of it as some abstract data structure, whose details are hidden from you. In other words, the only way you can use a FILE * is via the functions that C gives you.

Note: In reality, FILE is some kind of structure that holds information about the file. We must use a FILE * because certain functions will need to change that information, i.e., we need to pass the information around ***by reference*** .

Functions

Reading from or writing to a file in C requires 3 basic steps:

1. Open the file.
2. Do all the reading or writing.
3. Close the file.

Following are described the functions needed to accomplish each step.

3. **Opening a file:**

In order to open a file, use the function fopen(). Use it as:

fp = fopen(***filename*** , ***mode***);

where:

- ***filename*** is a string that holds the name of the file on disk (including a ***path*** like /cs/course if necessary).
- ***mode*** is a string representing how you want to open the file. Most often you'll open a file for reading ("r") or writing ("w").

Note that fopen() returns a FILE * that can then be used to access the file. When the file cannot be opened (e.g., we don't have permission or it doesn't exist when opening for reading), fopen() will return NULL.

Here are examples of opening files:

```
FILE *ifp, *ofp;
char *mode = "r";
char outputFilename[] = "out.list";

ifp = fopen("in.list", mode);

if (ifp == NULL) {
  fprintf(stderr, "Can't open input file in.list!\n");
  exit(1);
}

ofp = fopen(outputFilename, "w");

if (ofp == NULL) {
  fprintf(stderr, "Can't open output file %s!\n",
          outputFilename);
  exit(1);
```

```
}
```

Note that the input file that we are opening for reading ("r") must already exist. In contrast, the output file we are opening for writing ("w") does not have to exist. If it doesn't, it will be created. If this output file does already exist, its previous contents will be thrown away (and will be lost).

Note: There are other modes you can use when opening a file, such as append ("a") to append something to the end of a file without losing its contents...or modes that allow you to both read and write. You can look up these other modes in a good C reference on stdio.h.

Reading from or writing to a file:

Once a file has been successfully opened, you can read from it using fscanf() or write to it using fprintf(). These functions work just like scanf() and printf(), except they require an extra first parameter, a FILE * for the file to be read/written.

Note: There are other functions in **stdio.h** that can be used to read or write files. Look them up in a good C reference.

Continuing our example from above, suppose the input file consists of lines with a ***username*** and an ***integer test score***, e.g.:

```
in.list
------
foo 70
bar 98
...
```

and that each username is no more than 8 characters long.

We might use the files we opened above by copying each username and score from the input file to the output file. In the process, we'll increase each score by 10 points for the output file:

```
char username[9];  /* One extra for nul char. */
int score;

...

while (fscanf(ifp, "%s %d", username, &score) != EOF) {
  fprintf(ofp, "%s %d\n", username, score+10);
}

...
```

The function fscanf(), like scanf(), normally returns the number of values it was able to read in. However, when it hits the end of the file, it returns the special value EOF. So, testing the return value against EOF is one way to stop the loop.

The bad thing about testing against EOF is that if the file is not in the right format (e.g., a letter is found when a number is expected):

```
in.list
------
foo 70
bar 98
biz A+
...
```

then fscanf() will not be able to read that line (since there is no integer to read) and it won't advance to the next line in the file. For this error, fscanf() will not return EOF (it's not at the end of the file)....

Errors like that will at least mess up how the rest of the file is read. In some cases, they will cause an ***infinite loop***.

One solution is to test against the number of values we expect to be read by fscanf() each time. Since our format is "%s %d", we expect it to read in 2 values, so our condition could be:

```
while (fscanf(ifp, "%s %d", username, &score) == 2) {
```

Now, if we get 2 values, the loop continues. If we don't get 2 values, either because we are at the end of the file or some other problem occurred (e.g., it sees a letter when it is trying to read in a number with %d), then the loop will end.

Another way to test for end of file is with the library function feof(). It just takes a file pointer and returns a true/false value based on whether we are at the end of the file.

To use it in the above example, you would do:

```
while (!feof(ifp)) {
  if (fscanf(ifp, "%s %d", username, &score) != 2)
    break;
  fprintf(ofp, "%s %d", username, score+10);
}
```

Note that, like testing != EOF, it might cause an infinite loop if the format of the input file was not as expected. However, we can add code to make sure it reads in 2 values

Note: When you use fscanf(...) != EOF or feof(...), they will not detect the end of the file until they try to read past it. In other words, they won't report end-of-file on the last valid read, only on the one after it.

Closing a file:

When done with a file, it must be closed using the function fclose().

To finish our example, we'd want to close our input and output files:

```
fclose(ifp);
fclose(ofp);
```

Closing a file is very important, especially with output files. The reason is that output is often ***buffered***. This means that when you tell C to write something out, e.g.,

```
fprintf(ofp, "Whatever!\n");
```

it doesn't necessary get written to disk right away, but may end up in a ***buffer*** in memory. This output buffer would hold the text temporarily:

Sample output buffer:

a	b	c	W	h	a	t	e	v	e	r
!	\n									

...

(The buffer is really just 1-dimensional despite this drawing.)

When the buffer fills up (or when the file is ***closed***), the data is finally written to disk.

So, if you forget to close an output file then whatever is still in the buffer may not be written out.

Note: There are other kinds of buffering than the one we describe here.

Special file pointers:

There are 3 special FILE *'s that are always defined for a program. They are stdin (***standard input***), stdout (***standard output***) and stderr (***standard error***).

Standard Input

Standard input is where things come from when you use scanf(). In other words,

scanf("%d", &val);

is equivalent to the following fscanf():

fscanf(stdin, "%d", &val);

Standard Output

Similarly, ***standard output*** is exactly where things go when you use printf(). In other words,

printf("Value = %d\n", val):

is equivalent to the following fprintf():

fprintf(stdout, "Value = %d\n", val):

Remember that standard input is normally associated with the keyboard and standard output with the screen, unless ***redirection*** is used.

Standard Error

Standard error is where you should display error messages. We've already done that above:

fprintf(stderr, "Can't open input file in.list!\n");

Standard error is normally associated with the same place as standard output; however, redirecting standard output does not redirect standard error.

For example,

% **a.out > outfile**

only redirects stuff going to standard output to the file **outfile**... anything written to standard error goes to the screen.

Using the Special File Pointers

We've already seen that ***stderr*** is useful for printing error messages, but you may be asking, "When would I ever use the special file pointers ***stdin*** and ***stdout***?" Well, suppose you create a function that writes a bunch of data to an opened file that is specified as a parameter:

```
void WriteData(FILE *fp)
{
  fprintf(fp, "data1\n");
  fprintf(fp, "data2\n");
  ...
}
```

Certainly, you can use it to write the data to an output file (like the one above):

```
WriteData(ofp);
```

But, you can also write the data to standard output:

```
WriteData(stdout);
```

Without the special file pointer ***stdout***, you'd have to write a second version of WriteData() that wrote stuff to standard output.

UNIT III

LINEAR DATASTRUCTURES - LIST

1. Abstract Data Types (ADTs)
2. List ADT array-based implementation
3. List linked list implementation
4. Singly linked lists
5. Circularly linked lists
6. Doubly-linked lists
7. Applications of lists
8. Polynomial Manipulation –All operation (Insertion, Deletion, Merge, Traversal)

3.1. ABSTRACT DATATYPE

An abstract data type (ADT) is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.[1]

For example, an abstract stack could be defined by three operations: push, that inserts some data item onto the structure, pop, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and peek, that allows data on top of the structure to be examined without removal. When analyzing the efficiency of algorithms that use stacks, one may also specify that all operations take the same time no matter how many items have been pushed into the stack, and that the stack uses a constant amount of storage for each element.

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages. However, an ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language. ADTs are often implemented as modules: the module's interface declares procedures that correspond to the ADT operations, sometimes with comments that describe the constraints. This information hiding strategy allows the implementation of the module to be changed without disturbing the client programs.

3.2.LIST ADT ARRAY-BASED IMPLEMENTATION

List Abstract Data Type

A list is a sequence of zero or more elements of a given type

$a_1, a_2,..., a_n \quad (n \geq 0)$

n : length of the list

a_1 : first element of the list

a_n : last element of the list

$n = 0$: empty list

elements can be linearly ordered according to their position in the list

We say a_i precedes a_{i+1}, a_{i+1} follows a_i, and a_i is at position i

Let us assume the following:

L : list of objects of type element type

x : an object of this type

p : of type position

END(L) : a function that returns the position following the last position in the list L

Define the following operations:

1.Insert (*x, p, L*)

- Insert *x* at position *p* in list *L*
- If *p* = *END*(*L*), insert *x* at the end
- If *L* does not have position *p*, result is undefined

2. Locate (*x, L*)

- returns position of *x* on *L*
- returns END(L) if *x* does not appear

3.Retrieve (*p, L*)

- returns element at position *p* on *L*
- undefined if *p* does not exist or *p* = *END*(*L*)

4.Delete (*p, L*)

- delete element at position *p* in *L*
- undefined if *p* = *END*(*L*) or does not exist

5.Next (*p, L*)

- returns the position immediately following position *p*

6.Prev (*p, L*)

returns the position previous to *p*

7.Makenull (*L*)

- causes *L* to become an empty list and returns position *END*(*L*)

8. First (*L*)

- returns the first position on *L*

9.Printlist (*L*)

- print the elements of *L* in order of occurrence

3.3 ARRAY IMPLEMENTATION OF LISTS

LIST:

A list is a sequential data structure, ie. a collection of items accessible one after another beginning at the head and ending at the tail.

- It is a widely used data structure for applications which do not need random access
- Addition and removals can be made at any position in the list
- lists are normally in the form of $a_1,a_2,a_3.....a_n$. The size of this list is n.The first element of the list is a_1,and the last element is a_n.The position of element a_i in a list is i.
- List of size 0 is called as null list.

Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

Implementation of List:

A list can be implemented in two ways

1. Array list
2. Linked list

1.Storing a list in a static data structure(Array List)

- This implementation stores the list in an array.
- The position of each element is given by an index from 0 to n-1, where n is the number of elements.
- The element with the index can be accessed in constant time (ie) the time to access does not depend on the size of the list.
- The time taken to add an element at the end of the list does not depend on the size of the list. But the time taken to add an element at any other point in the list depends on the size of the list because the subsequent elements must be shifted to next index value.So the additions near the start of the list take longer time than the additions near the middle or end.
- Similarly when an element is removed,subsequent elements must be shifted to the previous index value. So removals near the start of the list take longer time than removals near the middle or end of the list.

Problems with Array implementation of lists:

- Insertion and deletion are expensive. For example, inserting at position 0 (a new first element) requires first pushing the entire array down one spot to make room, whereas

deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations is $O(n)$.

- Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high over-estimate, which wastes considerable space. This could be a serious limitation, if there are many lists of unknown size.
- Simple arrays are generally not used to implement lists. Because the running time for insertion and deletion is so slow and the list size must be known in advance

2. Storing a list in a dynamic data structure(Linked List)

- The Linked List is stored as a sequence of linked nodes which are not necessarily adjacent in memory.
- Each node in a linked list contains data and a reference to the next node
- The list can grow and shrink in size during execution of a program.
- The list can be made just as long as required.It does not waste memory space because successive elements are connected by pointers.
- The position of each element is given by an index from 0 to n-1, where n is the number of elements.
- The time taken to access an element with an index depends on the index because each element of the list must be traversed until the required index is found.
- The time taken to add an element at any point in the list does not depend on the size of the list,as no shifts are required
- Additions and deletion near the end of the list take longer than additions near the middle or start of the list. because the list must be traversed until the required index is found

Array versus Linked Lists

1.Arrays are suitable for

- Randomly accessing any element.
- Searching the list for a particular value
- Inserting or deleting an element at the end.

2.Linked lists are suitable for

-Inserting/Deleting an element.
-Applications where sequential access is required.
-In situations where the number of elements cannot be predicted beforehand.

3.4 LINKED LIST IMPLEMENTATION OF LISTS(POINTERS)

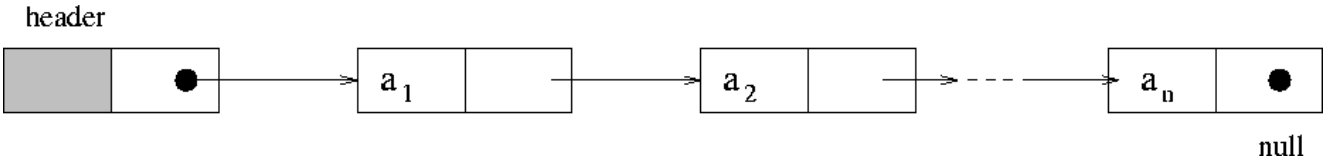
- The Linked List is stored as a sequence of linked nodes which are not necessarily adjacent in memory.
- Each node in a linked list contains data and a reference to the next node
- The list can grow and shrink in size during execution of a program.
- In the array implementation,
  1. we are constrained to use contiguous space in the memory
  2. Insertion, deletion entail shifting the elements
- Pointers overcome the above limitations at the cost of extra space for pointers.

- data structure that will be used for the nodes. For list where each node holds a float:example, the following struct could be used to create a

```
struct Node

{
        int node ;
        struct Node *next;
};
```

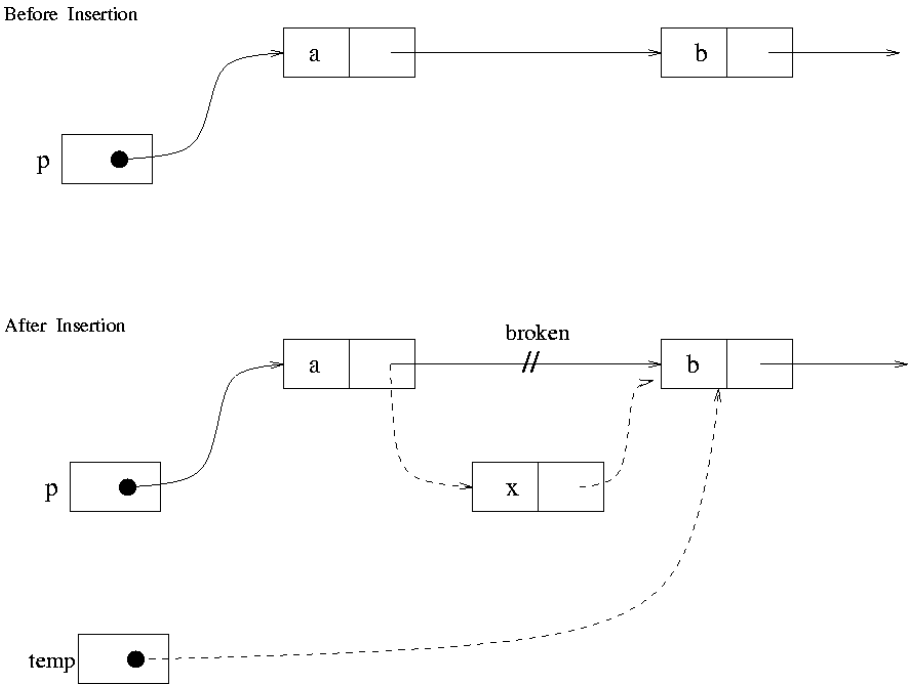
3.5 SINGLY LINKED LIST

Figure 2.1: A singly linked list

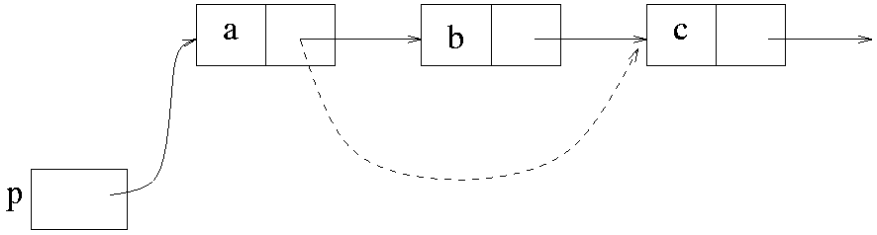


- Let us follow a convention that position *i* is a pointer to the cell holding the pointer to the cell containing a_i, (for $i = 1, 2, ..., n$). Thus,
  - Position 1 is a pointer to the header
  - End (L) is a pointer to the last cell of list L
- If position of a_i is simply a pointer to the cell holding a_i, then
  - Position 1 will be the address in the header
  - end (L) will be a null pointer
- Insert (x, p, L) : See Figure
- Delete (x, L) : See Figure

Insertion in a singly linked list



Deletion in a singly linked list



Singly Linked List

```
#include <stdio.h>
#include <malloc.h>
struct node
{
int data;
struct node *next;
};

struct node *create_node(int);
void insert_node_first();
void insert_node_last();
void insert_node_pos();
void delete_pos();
void search();
```

```
void display();
struct node *p;

void main()
{
int ch;
char ans='Y';

while(ans=='Y'||ans=='y')
{
printf("\n---------------------------------\n");
printf("\nOperations on singly linked list\n");
printf("\n---------------------------------\n");
printf("\n1.Insert node at first");
printf("\n2.Insert node at last");
printf("\n3.Insert node at position");
printf("\n4.Delete Node from any Position");
printf("\n5.Search Element in the linked list");
printf("\n6.Display List from Beginning to end");
printf("\n7.Exit\n");
printf("\nEnter your choice");
scanf("%d",&ch);

switch(ch)
{
case 1:
printf("\n...Inserting node at first...\n");
insert_node_first();
break;
case 2:
printf("\n...Inserting node at last...\n");
insert_node_last();
break;
case 3:
printf("\n...Inserting node at position...\n");
insert_node_pos();
break;
case 4:
printf("\n...Deleting Node from any Position...\n");
delete_pos();
break;
case 5:
printf("\n...Searching Element in the List...\n");
search();
break;
case 6:
printf("\n...Displaying List From Beginning to End...\n");
```

```
display();
break;
case 7:
exit(0);
printf("\n...Exiting...\n");

break;
default:
printf("\n...Invalid Choice...\n");
break;
}
printf("\nYOU WANT TO CONTINUE (Y/N)");
scanf(" %c",&ans);
}
}

struct node *create_node(int ele)
{
struct node *temp;
temp=(struct node*)malloc(sizeof(struct node));

temp->data =ele;
temp->next = NULL;
return temp;

}

void insert_node_first()
{
int  ele;
struct node *temp;
printf("\nEnter the value for the node:");
scanf("%d",&ele);
temp=create_node(ele);
if(p==NULL)
{
p=temp;
}
else
{
temp->next=p;
p=temp;
}
printf("\n----INSERTED----");
}

void insert_node_last()
```

```
{
int  ele;
struct node *temp,*r;
printf("\nEnter the value for the Node:");
scanf("%d",&ele);
temp=create_node(ele);
if(p==NULL)
{
p=temp;
}
else
{
r=p;
while(r->next!=NULL)
{
r=r->next;
}
 r->next=temp;
}
printf("\n----INSERTED----");
}

void insert_node_pos()
{
int pos,ele,i;
struct node *temp,*r,*t;
printf("\nEnter the value for the Node:");
scanf("%d",&ele);
temp=create_node(ele);
printf("\nEnter the position ");
scanf("%d",&pos);
if(pos==1)
{
if(p==NULL)
{
p=temp;
}
}
else
{
r=p;

for(i=1;i<pos;i++)
{
 t=r;
r=r->next;
}
```

```
t->next=temp;
temp->next=r;
}
printf("\nInserted");
}

void delete_pos()
{
int pos,i;
struct node *r,*t;
if(p==NULL)
{
printf(":No node to delete\n");
}
else
{
printf("\nEnter the position of value to be deleted:");
scanf("%d",&pos);
if(pos==1)
{
p=p->next;
printf("\nElement deleted");
}
else
{
 r=p;
for(i=1;i<pos;i++)
{
t=r;
r=r->next;
}
t->next=r->next;
free(r);
}
}
}
void search()
{
struct node *r;
int ele;
if(p==NULL)
{
 printf(":No nodes in the list\n");
}
else
{
printf("\nEnter the value to search");
```

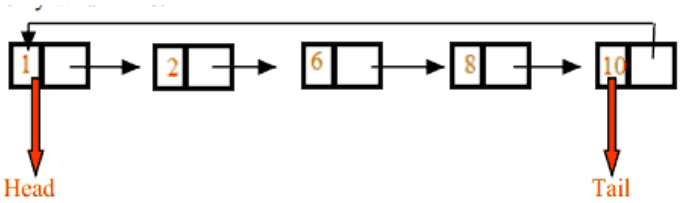
```
scanf("%d",&ele);
r=p;
while(r!=NULL)
{
 if(r->data==ele)
{
printf("found");
return;
}
else
{
r=r->next;
}
printf("not found");
}
}}

void display()
{
 struct node *r;
if(p==NULL)
{
printf(":No nodes in the list to display\n");
}
else
{
r=p;
while(r!=NULL)
{
printf("%d\t",r->data);
r=r->next;
}
}
}
```

3.6. CIRCULAR LINKED LIST

In circular linked list the pointer of the last node points to the first node. Circular linked list can be implemented as Singly linked list and Doubly linked list with or without headers.

Advantages of Circular Linked List

- It allows to traverse the list starting at any point.
- It allows quick access to the first and last records.



Basic operations of a singly-linked list are:

1. Insert – Inserts a new element at the end of the list.
2. Delete – Deletes any node from the list.
3. Find – Finds any node in the list.
4. Print – Prints the list.

C PROGRAM FOR CIRCULAR LINKED LISTS

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

 struct node
{
     int data;
     struct node *next;
 };

 struct node *head = NULL, *tail = NULL;

 struct node * createNode(int data) {
     struct node *newnode;
     newnode = (struct node *)malloc(sizeof (struct node));
     newnode->data = data;
     newnode->next = NULL;
 }

  /*
   * create dummy head and tail to make
   * insertion and deletion operation simple.
   * While processing data in our circular
   * linked list, skip these dummies.
   */

 void createDummies() {
     head = createNode(0);
     tail = createNode(0);
     head->next = tail;
     tail->next = head;
 }
```

```
/* insert data next to dummy head */
void circularListInsertion(int data) {
    struct node *newnode, *temp;
    newnode = createNode(data);
    temp = head->next;
    head->next = newnode;
    newnode->next = temp;
}

/* Delete the node that has the given data */
void DeleteInCircularList(int data) {
    struct node *temp0, *temp;
    if (head->next == tail && tail->next == head) {
        printf("Circular Queue/list is empty\n");
    }
    temp0 = head;
    temp = head->next;
    while (temp != tail) {
        if (temp->data == data) {
            temp0->next = temp->next;
            free(temp);
            break;
        }
        temp0 = temp;
        temp = temp->next;
    }
    return;
}

/*
 * travese the circular linked list for
 * the given no of times.
 */
void display(int count) {
    int n = 0;
    struct node *tmp = head;

    if (head->next == tail && tail->next == head) {
        printf("Circular linked list is empty\n");
        return;
    }

    while (1) {
        /* skip the data in dummy head and tail */
        if (tmp == head || tmp == tail) {
            if (tmp == tail) {
```

```
                        n++;
                        printf("\n");
                        if (n == count)
                            break;
                    } else {
                        tmp = tmp->next;
                        continue;
                    }
                } else {
                    printf("%-3d", tmp->data);
                }
                tmp = tmp->next;
            }
            return;
        }


        int main()
{
            int data, ch, count;
            createDummies();
            while (1) {
                printf("1. Insertion\t2. Deletion\n");
                printf("3. Display\t4. Exit\n");
                printf("Enter ur choice:");
                scanf("%d", &ch);
                switch (ch) {
                    case 1:
                        printf("Enter the data to insert:");
                        scanf("%d", &data);
                        circularListInsertion(data);
                        break;

                    case 2:
                        printf("Enter the data to delete:");
                        scanf("%d", &data);
                        DeleteInCircularList(data);
                        break;

                    case 3:
                        printf("No of time u wanna traverse:");
                        scanf("%d", &count);
                        display(count);
                        break;

                    case 4:
                        exit(0);
```

```
                default:
                    printf("Pls. enter correct option\n");
                    break;
            }
        }
        return 0;
    }
```

3.7. DOUBLY LINKED LIST

Doubly linked list

In a **doubly-circularly-linked list**, each node has two links, similar to a *doubly-linked list*, except that the previous link of the first node points to the last node and the next link of the last node points to the first node. As in doubly-linked lists, insertions and removals can be done at any point with access to any nearby node.

Sentinel nodes

Linked lists sometimes have a special *dummy* or *sentinel node* at the beginning and/or at the end of the list, which is not used to store data.

Basic Operations on Linked Lists

1. Insertion
    a. At first
    b. At last
    c. At a given location (At middle)
2. Deletion
    a. First Node
    b. Last Node
    c. Node in given location or having given data item

Initial Condition

HEAD = NULL;
/* Address of the first node in the list is stored in HEAD. Initially there is no node in the list. So, HEAD is initialized to NULL (No address) */

What are the Applications of linked list?

- To implement of Stack, Queue, Tree, Graph etc.,
- Used by the Memory Manager
- To maintain Free-Storage List

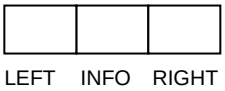
Doubly Linked Lists (or) Two – Way Lists

There are some problems in using the Single linked list. They are

1. A singly linked list allows traversal of the list in only one direction. (Forward only)
2. Deleting a node from a list requires keeping track of the previous node, that is, the node whose link points to the node to be deleted.

These major drawbacks can be avoided by using the double linked list. The doubly linked list is a ***linear collection*** of data elements, called ***nodes***, where each node is divided into three parts. They are:

1. *A pointer field LEFT which contains the address of the preceding node in the list*
2. *An information field INFO which contains the data of the Node*
3. *A pointer field RIGHT which contains the address of the next node in the list*



Example:


```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    struct node *previous;
    int data;
    struct node *next;
}*head, *r;

void insert_begning(int value)
{
   struct node *tem,*t;
    tem=(struct node *)malloc(sizeof(struct node));
    tem->data=value;
    if(head==NULL)
    {
       head=tem;
       head->previous=NULL;
       head->next=NULL;
       r=head;
    }
    else
    {
       t=tem;
       t->previous=NULL;
       t->next=head;
```

```
        head->previous=t;
        head=t;
    }
}

void insert_end(int value)
{
    struct node *tem,*t;
    tem=(struct node *)malloc(sizeof(struct node));
          tem->data=value;
    if(head==NULL)
    {
        head=tem;
        head->previous=NULL;
        head->next=NULL;
        r=head;
    }
    else
    {
       r=head;
       while(r!=NULL)
       {
          t=r;
          r=r->next;
       }
    r=tem;
    t->next=r;
    r->previous=t;
    r->next=NULL;
    }
}

int insert_after(int value, int loc)
{
    struct node *t,*tem,*t1;
    tem=(struct node *)malloc(sizeof(struct node));
    tem->data=value;
          if(head==NULL)
    {
           head=tem;
           head->previous=NULL;
           head->next=NULL;
    }
    else
    {
           t=head;
           while(t!=NULL && t->data!=loc)
```

```
            {
                    t=t->next;
            }
            if(t==NULL)
            {
                    printf("\n%d is not present in list ",loc);
            }
            else
            {
            t1=t->next;
            t->next=tem;
            tem->previous=t;
           tem->next=t1;
            t1->previous=tem;
            }
    }
    r=head;
    while(r->next!=NULL)
    {
            r=r->next;
    }
    return;
}
int delete_from_end()
{
     struct node *t;
     t=r;
     if(t->previous==NULL)
     {
            free(t);
            head=NULL;
            r=NULL;
            return 0;
     }
     printf("\nData deleted from list is %d \n",r->data);
     r=t->previous;
     r->next=NULL;
     free(t);
     return 0;
}


int delete_from_middle(int value)
{
   struct node *temp,*var,*t, *temp1;
   temp=head;
   while(temp!=NULL)
```

```
    {
        if(temp->data == value)
        {
            if(temp->previous==NULL)
            {
                free(temp);
                head=NULL;
                return 0;
            }
            else
            {
                var->next=temp1;
                temp1->previous=var;
                free(temp);
                return 0;
            }
        }
        else
        {
             var=temp;
             temp=temp->next;
             temp1=temp->next;
        }
    }
    printf("data deleted from list is %d",value);
    return 0;
}

void display()
{
     struct node *t;
     t=head;
     if(t==NULL)
      {
           printf("List is Empty");
      }
     while(t!=NULL)
     {
            printf("-> %d ",t->data);
            t=t->next;
     }
}

void main()
{
    int value, ch, loc;
    clrscr();
```

```
head=NULL;
printf("Doubly Linked List");
printf("\n1.insert at begning\n2. insert at end\n3.insert at middle");
printf("\n4.delete from end\n5.delete middle\n6.display list\n7.exit");
while(1)
{
        printf("\n\nenter the choice  ");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1:
                {
                 printf("enter the value to insert in node ");
                 scanf("%d",&value);
                 insert_begning(value);
                 display();
                 break;
                 }
                 case 2:
                 {
                 printf("enter the value to insert in node at end ");
                 scanf("%d",&value);
                 insert_end(value);
                 display();
                 break;
                 }
                 case 3:
                 {
                 printf("after which data you want to insert data ");
                 scanf("%d",&loc);
                 printf("enter the data you want to insert in list ");
                 scanf("%d",&value);
                 insert_after(value,loc);
                 display();
                 break;
                 }
                 case 4:
                 {
                 delete_from_end();
                 display();
                 break;
                 }
                 case 5:
                 {
                 printf("enter the position to delete");
                 scanf("%d",value);
                 delete_from_middle(value);
```

```
                display();
                break;
                }
                case 6 :
                {
                display();
                break;
                }
                case 7 :
                {
                    exit(0);
                    break;
                }
            }
    }

    getch();
}
```

Array	Linked list
Static memory	Dynamic memory
Insertion and deletion required to modify the existing element location	Insertion and deletion are made easy.
Elements stored as contiguous memory as on block.	Element stored as Non-contiguous memory as pointers
Accessing element is fast	Accessing element is slow

3.8 APPLICATIONS OF LINKED LIST

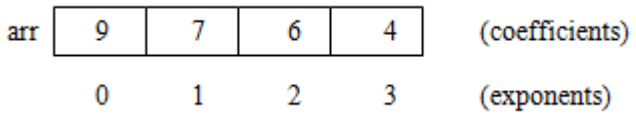
1. Polynomial ADT
2. Radix Sort
3. Multilist

3.9 POLYNOMIAL MANIPULATION

- Representation
- Addition
- Multiplication

Representation of a Polynomial: A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is $P(x) = 4x^3+6x^2+7x+9$

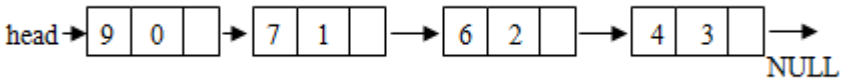
A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. The array representation for the above polynomial expression is given below:



A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```
struct polynomial
{
int coefficient;
int exponent;
struct polynomial *next;
};
```

Thus the above polynomial may be represented using linked list as shown below:



Addition of two Polynomials:

For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result. The complete program to add two polynomials is given in subsequent section.

Multiplication of two Polynomials:

Multiplication of two polynomials however requires manipulation of each node such that the exponents are added up and the coefficients are multiplied. After each term of first polynomial is operated upon with each term of the second polynomial, then the result has to be added up by comparing the exponents and adding the coefficients for similar exponents and including terms as such with dissimilar exponents in the result. The 'C' program for polynomial manipulation is given below:

Program for Polynomial representation, addition and multiplication

/*Polynomial- Representation, Addition, Multiplication*/

Representation using structure

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

typedef struct  node
 { int power;
   int coeff;
```

```
  struct node *next;
 }node;

     node *  insert(node *head,int power1,float coeff1);
     node *  create();
     node *  padd(node *p1,node *p2);
     void print(node *head);

node *insert(node *head,int power1,float coeff1)
     {

     node *p,*q;

    p=(node*) malloc(sizeof(node));
      p->power=power1;
      p->coeff=coeff1;
      p->next=NULL;
      if(head==NULL)
    {
        head=p;
        head->next=head;
        return(head);
    }
      if(power1>head->power)
     {
        p->next=head->next;
        head->next=p;
        head=p;
        return(head);
     }
      if(power1==head->power)
     {
        head->coeff=head->coeff+coeff1;
        return(head);
     }

    q=head;
    while(q->next!=head && power1>=q->next->power)
         q=q->next;
    if(p->power==q->power)
         q->coeff=q->coeff+coeff1;
    else
      {
         p->next=q->next;
         q->next=p;
      }
   return(head);
```

```
 }


node *create()
  {
     int n,i,power1;
     float coeff1;
     node *head=NULL;
     printf("\nEnter No. of Terms:");
     scanf("%d",&n);
     printf("\nenter a term as a tuple of (power,coefficient) : ");
     for(i=1;i<=n;i++)
    {   scanf("%d%f",&power1,&coeff1);
        head=insert(head,power1,coeff1);
    }
     return(head);
 }

node * padd(node *p1,node *p2)
{   node *p;
    node *head=NULL;
    int power;float coeff;
    p=p1->next;
    do
    {
       head=insert(head,p->power,p->coeff);
       p=p->next;
    } while(p!=p1->next);
    p=p2->next;
    do
    {
       head=insert(head,p->power,p->coeff);
       p=p->next;
    } while(p!=p2->next);
    return(head);
 }

void print( node *head)
{  node *p;
   p=head->next;
   printf("\n");
   do
    {
   printf("%6.2fx^%d    ",p->coeff,p->power);
   p=p->next;
    }while(p!=head->next);
}
```

UNIT IV

LINEAR DATA STRUCTURES – STACKS, QUEUES

1. StackADT
2. Infix to Postfix conversion
3. Postfix Evaluation
4. Queue ADT
5. Infix to Prefix Conversion
6. Prefix Evaluation
7. Circular queue implementation
8. Double ended Queues

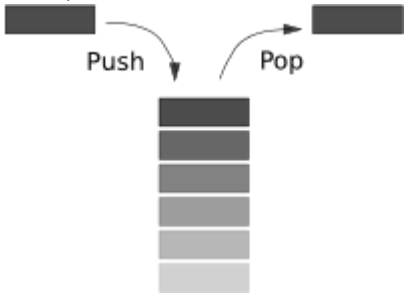
4.1. STACK ADT

STACK :

"A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top".* stacks are sometimes referred to as Last In First Out (LIFO) lists
Stacks have some useful terminology associated with them:

- **Push** To add an element to the stack
- **Pop** To remove an element from the stock
- **Peek** To look at elements in the stack without removing them
- **LIFO** Refers to the last in, first out behavior of the stack
- **FILO** Equivalent to LIFO

stack (data structure)



Simple representation of a stack

Given a stack S=(a[1],a[2],.....a[n]) then we say that a1 is the bottom most element and element a[i]) is on top of element a[i-1], 1<i<=n.

Implementation of stack :

1. array (static memory).
2. linked list (dynamic memory)

The operations of stack is

1. PUSH operations
2. POP operations
3. PEEK operations

The Stack ADT

A stack S is an abstract data type (ADT) supporting the following three methods:
push(n) : Inserts the item ***n*** at the top of stack

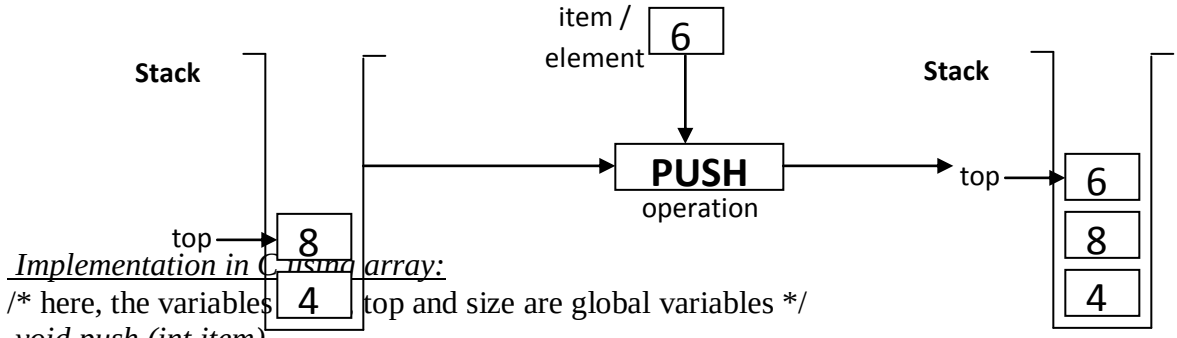
pop() : Removes the top element from the stack and returns that top element. An error occurs if the stack is empty.
peek(): Returns the top element and an error occurs if the stack is empty.

1. Adding an element into a stack. (called PUSH operations)
Adding element into the TOP of the stack is called PUSH operation.

Check conditions :
TOP = N , then **STACK FULL**
where N is maximum size of the stack.

Adding into stack (PUSH algorithm)

procedure add(item : items);
{add item to the global stack stack ; top is the current top of stack
and n is its maximum size}
begin
if top = n **then** stackfull;
top := top+1;
stack(top) := item;
end: {of add}



Implementation in C using array:
/* here, the variables top and size are global variables */
void push (int item)
{
if (top == size-1)
printf("Stack is Overflow");
else
{
top = top + 1;
stack[top] = item;
}
}

2. Deleting an element from a stack. (called POP operations)

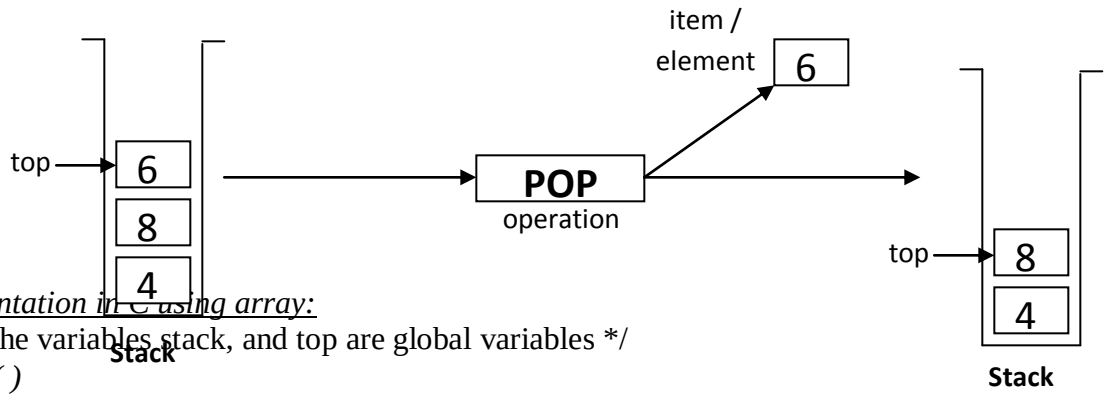
Deleting or Removing element from the TOP of the stack is called POP operations.

Check Condition:

TOP = 0**, then** STACK EMPTY

Deletion in stack (POP Operation)

```
procedure delete(var item : items);
{remove top element from the stack stack and put it in the item}
begin
   if top = 0 then stackempty;
   item := stack(top);
   top := top-1;
end; {of delete}
```

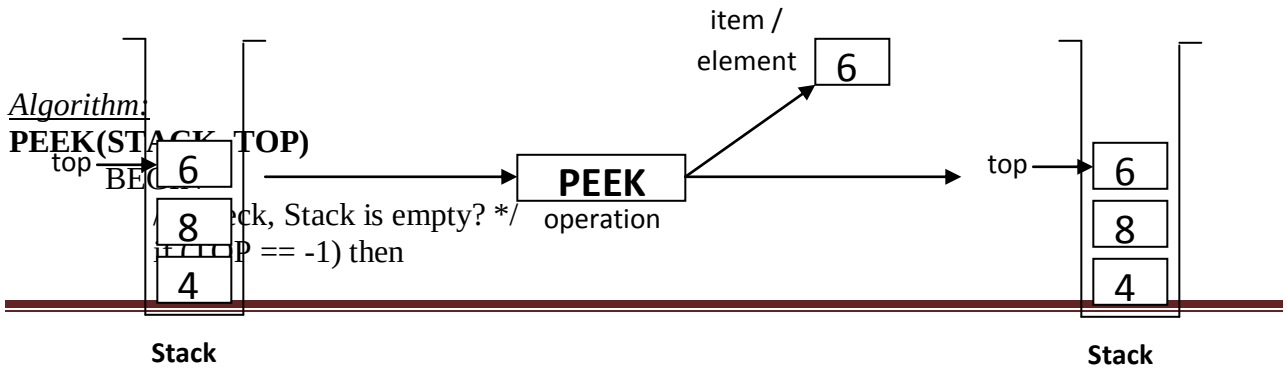


Implementation in C using array:

```
/* here, the variables stack, and top are global variables */
 int pop ( )
{
        if (top == -1)
        {
                printf("Stack is Underflow");
                return (0);
        }
        else
        {
                return (stack[top--]);
        }
}
```

3. Peek Operation:

- ✓ Returns the item at the top of the stack but does not delete it.
- ✓ This can also result in ***underflow*** if the stack is empty.



Algorithm:

PEEK(STACK, TOP)

BE[illegible]

/*[illegible]ck, Stack is empty? */

[illegible]P == -1) then

```
            print "Underflow" and return 0.
        else
            item = STACK[TOP] / * stores the top element into a local variable */
            return item          / * returns the top element to the user */
    END
```

Implementation in C using array:

```
/* here, the variables stack, and top are global variables */
 int pop ( )
{
    if (top == -1)
    {
        printf("Stack is Underflow");
        return (0);
    }
    else
    {
        return (stack[top]);
    }
}
```

Applications of Stack

1. It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
2. Infix to Postfix Transformation
3. It is useful during the execution of recursive programs
4. A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
5. A stack (memory stack) can be used in function calls including recursion.
6. Reversing Data
7. Reverse a List
8. Convert Decimal to Binary
9. Parsing – It is a logic that breaks into independent pieces for further processing
10. Backtracking

Note :

1. Infix notation A+(B*C)
equivalent Postfix notation ABC*+
2. Infix notation (A+B)*C
Equivalent Postfix notation AB+C*

4.2. EXPRESSION EVALUATION AND SYNTAX PARSING

Calculators employing reverse Polish notation (also known as **postfix notation**)use a stack structure to hold values.

Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based

machines. Note that natural languages are context sensitive languages and stacks alone are not enough to interpret their meaning.

Infix, Prefix and Postfix Notation

We are accustomed to write arithmetic expressions with the operation between the two operands: **a+b** or **c/d**. If we write **a+b*c**, however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?).

There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c		
40 - 3 * 5 + 1		

Examples of use: (application of stack)
Arithmetic Expressions: Polish Notation

- An arithmetic expression will have operands and operators.
- Operator precedence listed below:

  Highest : ($)
  Next Highest : (*) and (/)
  Lowest : (+) and (-)

- For most common arithmetic operations, the operator symbol is placed in between its two operands. This is called ***infix notation***.

  *Example: A + B , E * F*
- Parentheses can be used to group the operations.

  *Example: (A + B) * C*
- Accordingly, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

- Polish notation refers to the notation in which the operator symbol is placed before its two operands. This is called ***prefix notation***.

  *Example: +AB, *EF*
- The fundamental property of polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression.
- Accordingly, one never needs parentheses when writing expressions in Polish notation.

- ***Reverse Polish Notation*** refers to the analogous notation in which the operator symbol is placed after its two operands. This is called ***postfix notation***.

*Example: AB+, EF**

- Here also the parentheses are not needed to determine the order of the operations.

- *The computer usually evaluates an arithmetic expression written in infix notation in two steps,*
  1. *It converts the expression to **postfix notation**.*
  2. *It evaluates the postfix expression.*
- *In each step, the stack is the main tool that is used to accomplish the given task.*

(1)Question : (Postfix evaluation)

How to evaluate a mathematical expression using a stack The algorithm for Evaluating a postfix expression ?

• Initialise an empty stack

• While token remain in the input stream

– Read next token
– If token is a number, push it into the stack
– Else, if token is an operator, pop top two tokens off the stack, apply the operator, and push the answer back into the stack

• Pop the answer off the stack.

Algorithm postfixexpression

Initialize a stack, opndstk to be empty.

```
{scan the input string reading one element at a time into symb }
While ( not end of input string )
        {
        Symb := next input character;

        If symb is an operand Then

                push (opndstk,symb)
        Else
                [symbol is an operator]
                {
                Opnd1:=pop(opndstk);
                Opnd2:=pop(opndnstk);
                Value := result of applying symb to opnd1 & opnd2
                Push(opndstk,value);
                }
Result := pop (opndstk);
```

Example:
6 2 3 + - 3 8 2 / + * 2 $ 3 +

Symbol	*Operand 1 (A)*	*Operand 2 (B)*	*Value (A ⊗ B)*	*STACK*
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3				1, 3
8				1, 3, 8
2				1, 3, 8, 2
/	8	2	/	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2				7, 2
$	7	2	49	49
3				49, 3
+	49	3	52	***52***

The Final value in the STACK is 52. This is the answer for the given expression.

(2) run time stack for function calls (write factorial number calculation procedure)

push local data and return address onto stack
return by popping off local data and then popping off address and returning to it
return value can be pushed onto stack before returning, popped off by caller

(3) expression parsing

e.g. matching brackets: [... (... (...) [...(...) ...] ...) ...]
push left ones, pop off and compare with right ones

4) INFIX TO POSTFIX CONVERSION

Infix expressions are often translated into postfix form in which the operators appear after their operands.

Steps:

1. Initialize an empty stack.
2. Scan the Infix Expression from left to right.
3. If the scannned character is an operand, add it to the Postfix Expression.
4. If the scanned character is an operator and if the stack is empty, then push the character to stack.
5. If the scanned character is an operator and the stack is not empty, Then
   (a) Compare the precedence of the character with the operator on the top of the stack.
   (b) While operator at top of stack has higher precedence over the scanned character & stack is not empty.
      (i) POP the stack.
      (ii) Add the Popped character to Postfix String.
   (c) Push the scanned character to stack.
6. Repeat the steps 3-5 till all the characters

7. While stack is not empty,
   (a) Add operator in top of stack
   (b) Pop the stack.
8. Return the Postfix string.

Algorithm Infix to Postfix conversion (without parenthesis)

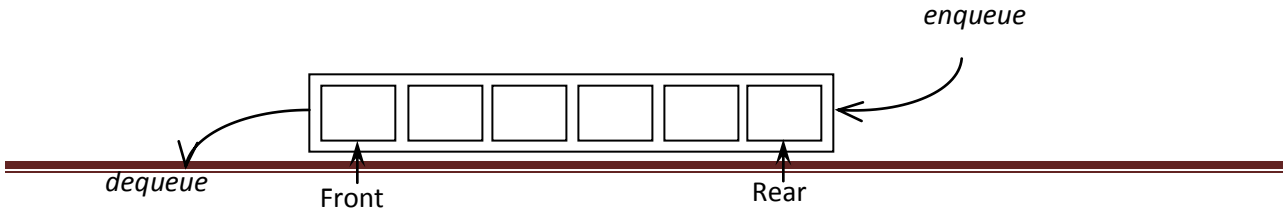
```
1. Opstk = the empty stack;
2. while ( not end of input )
   {
   symb = next input character;
3.  if ( symb is an operand )
            add symb to the Postfix String
4. else
     {
5.   While( ! empty (opstk) && prec ( stacktop ( opstk), symb) )
        {
          topsymb = pop ( opstk )
          add topsymb to the Postfix String;
          }  / * end of while */
      Push(opstk, symb);
       } /* end  else */
6.      }  /*  end while * /

7. While( ! empty ( opstk ) )
            {
            topsymb = pop (opstk)
            add topsymb to the Postfix String
            }  /*  end of while */
8. Return the Postfix String.
```

4..3 OTHER APPLICATIONS OF STACK

Some of the applications of stack are :
(i) Evaluating arithmetic expression
(ii) Balancing the symbols
(iii) Towers of Hannoi
(iv) Function Calls.
(v) 8 Queen Problem.

4.4 QUEUE ADT

"A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT*". queue*s are sometimes referred to as First In First Out (FIFO) lists.



Example

1. The people waiting in line at a bank cash counter form a queue.
2. In computer, the jobs waiting in line to use the processor for execution. This queue is called ***Job Queue***.

Operations Of Queue

There are two basic queue operations. They are,

Enqueue – Inserts an item / element at the rear end of the queue. An error occurs if the queue is full.

Dequeue – Removes an item / element from the front end of the queue, and returns it to the user. An error occurs if the queue is empty.

1. Addition into a queue

```
procedure addq (item : items);
{add item to the queue q}
begin
   if rear=n then queuefull
   else begin
       rear :=rear+1;
       q[rear]:=item;
   end;
end;{of addq}
```

2. Deletion in a queue

```
procedure deleteq (var item : items);
{delete from the front of q and put into item}
begin
   if front = rear then queueempty
   else begin
       front := front+1
       item := q[front];
   end;
end
```

Uses of Queues (Application of queue)

Queues remember things in first-in-first-out (FIFO) order. Good for fair (first come first served) ordering of actions.

Examples of use: (Application of stack)

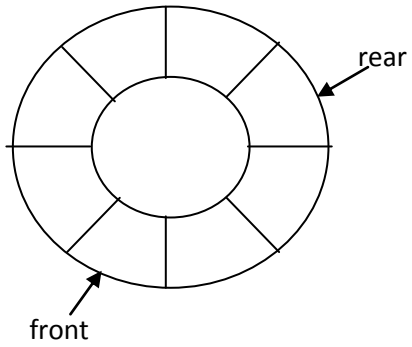
1• scheduling
processing of GUI events
printing request

2• simulation
orders the events

models real life queues (e.g. supermarkets checkout, phone calls on hold)

4.6. CIRCULAR QUEUE IMPLEMENTATION:

Location of queue are viewed in a circular form. The first location is viewed after the last one. Overflow occurs when all the locations are filled.



Algorithm Circular Queue Insert

```
Void  CQInsert ( int queue[ ], front, rear, item)
        {
        if ( front = = 0 )
                front = front +1;
        if ( ( ( rear = maxsize ) && ( front = = 1 ) )  || ( ( rear ! = 0 ) && ( front = rear +1)))
          {
          printf( "  queue overflow ");

          if( rear = = maxsize )
                rear = 1;
          else
                rear = rear + 1;
                q [ rear ] = item;
          }
       }
```

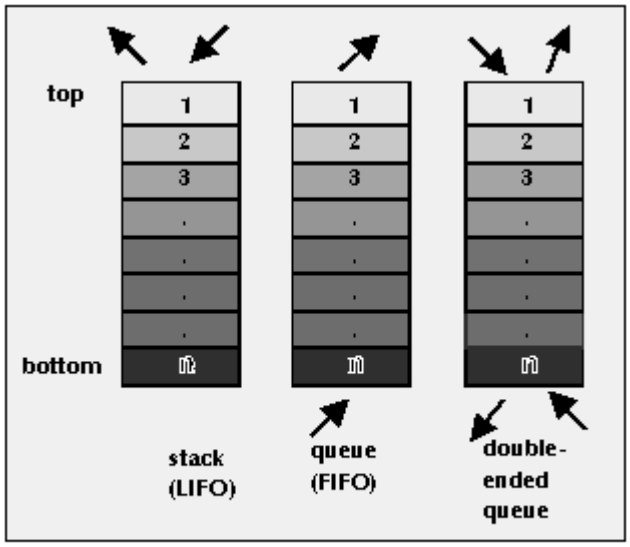
Algorithm Circular Queue Delete

```
int CQDelete ( queue [ ], front, rear )
        {
        if ( front = = 0 )
                printf ( " queue underflow ");
        else
           {
            item = queue [ front ];
            if(front = = rear )
                {
                front = 0; rear = 0;
                }
        else if ( front = = maxsize )
                {
```

```
            front = 1;
            }
    else
            front = front + 1;
      }
    return item;
   }
```

4.7. **DOUBLE ENDED QUEUE IMPLEMENTATION**

A **deque** (short for *double-ended queue*) is an abstract data structure for which elements can be added to or removed from the front or back(both end). This differs from a normal queue, where elements can only be added to one end and removed from the other. Both queues and stacks can be considered specializations of deques, and can be implemented using dequeue.



double-ended queue

Two types of Dqueue are

1. Input Restricted Dqueue
2. Ouput Restricted Dqueue.

1. Input Restricted Dqueue

Where the input (insertion) is restricted to the rear end and the deletions has the options either end

2. Ouput Restricted Dqueue.

Where the output (deletion) is restricted to the front end and the insertions has the option either end.

Example: *Timesharing system using the prototype of priority queue – programs of high priority are processed first and programs with the same priority form a standard queue.*

Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

Two types of queue are

1. Ascending Priority Queue
2. Descending Priority Queue

1. Ascending Priority Queue

Collection of items into which item can be inserted arbitrarily & from which only the Smallest item can be removed.

2. Descending Priority Queue

Collection of items into which item can be inserted arbitrarily & from which only the Largest item can be removed.

4.7. APPLICATIONS OF QUEUE

- Batch processing in an operating system
- To implement Priority Queues.
- Priority Queues can be used to sort the elements using Heap Sort.
- Simulation and Mathematics user Queueing theory.
- Computer networks where the server takes the jobs of the client as per thequeue strategy.

UNIT V

SOTING AND SEARCHING

1. Bubble sort
2. Quick sort
3. Insertion sort
4. Selection sort
5. Shell sort
6. Merge sort
7. Radix sort
8. Linear search
9. Binary Search
10. Hashing

Sorting

Definition:

Sorting: process by which a collection of items is placed into order

- Operation of arranging data
- Order typically based on a data field called a key

May be done to facilitate some other operation

- Searching for elements

Sorting is one of the most important operations performed by computers. In the days of magnetic tape storage before modern data-bases, it was almost certainly the *most* common operation performed by computers as most "database" updating was done by sorting transactions and merging them with a master file. It's still important for presentation of data extracted from databases:

In general sorting consists of comparisons and swapping. All Sorting algorithms are categorized into Stable and unstable Sorting.

- Stable Sorting: If the sorting taken place inside the given input array.
- Unstable Sorting: This kind of sorting needs a temporary array. This means sorting takes place outside of the array.

Next two categories of sorting are Internal sorting and External sorting. If the sorting takes place in internal main memory then it is internal sorting. If sorting happens in external memory storage such as magnetic tapes, Hard disk then it is external sorting.

The various sorting algorithms are available here some of them are discussed. They are as follows,

i) Exchange Sorts.
   a) Bubble sort
   b) Quick Sort
ii) Selection sorts
   a) Straight selection sort
iii) Insertion sorts

a) Simple insertion sort
b) Shell sort

iv) Merge and Radix Sorts
a) Merge sort
b) Radix sort

Exchange Sorts

The second class of sorting algorithm that we consider comprises algorithms that ***sort by exchanging*** pairs of items until the sequence is sorted. In general, an algorithm may exchange adjacent elements as well as widely separated ones.

Two types of exchange sorting procedures are Bubble sort and Merge sort.

5.1 BUBBLE SORT:

- *Bubble sort* is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them.
- It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.
- Bubble sort can be used to sort a small number of items (where its inefficiency is not a high penalty). Bubble sort may also be efficiently used on a list that is already sorted except for a very small number of elements.
- For example, if only one element is not in order, bubble sort will take only *2n* time. If two elements are not in order, bubble sort will take only at most *3n* time.

Bubble Sort Algorithm:

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

void main( )
{
    int a[100];
    int i, j, temp, n ;
    printf("how many numbers you want to sort : \n");
    scanf("%d",&n);
    printf("Enter %d number values you want to sort\n", n);
    for(j=0; j<n; j++)
    scanf("%d",&a[j]);

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

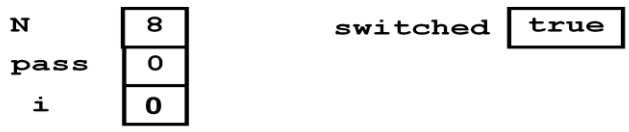
```
            if(a[i]>a[i+1])
            {
                temp=a[i];
                a[i]=a[i+1];
                a[i+1]=temp;
            }
        }
    }

    printf ( "\n\nArray after sorting:\n") ;

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

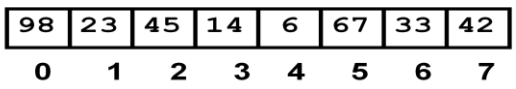
Algorithm Explanation

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping.
  - After the first traversal only the largest value is correctly placed
  - All other values are still out of order
  - So we need to repeat this process.
  - If we have N elements… And if each time we bubble an element, we place it in its correct location…
  - Then we repeat the "bubble up" process N – 1 times.
  - This guarantees we'll correctly place all N elements.
- We can use a Boolean variable (Here it is switched) to determine if any swapping occurred during the "bubble up."
- If no swapping occurred, then we know that the collection is already sorted.
- This Boolean "flag" needs to be reset after each "bubble up."

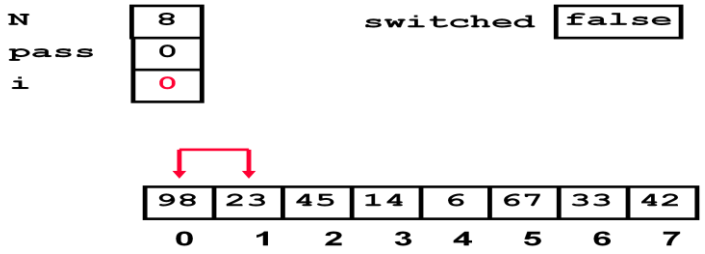
Execution of Algorithm:

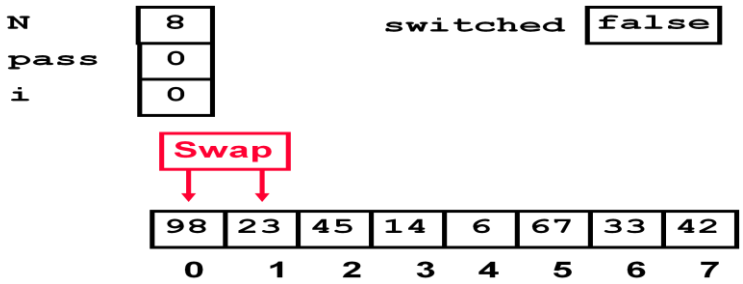
The given input values are

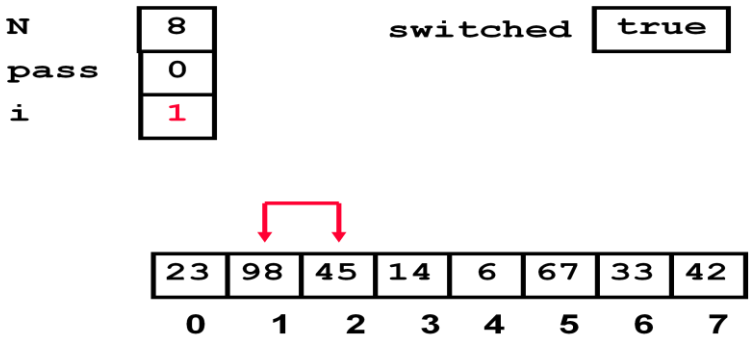




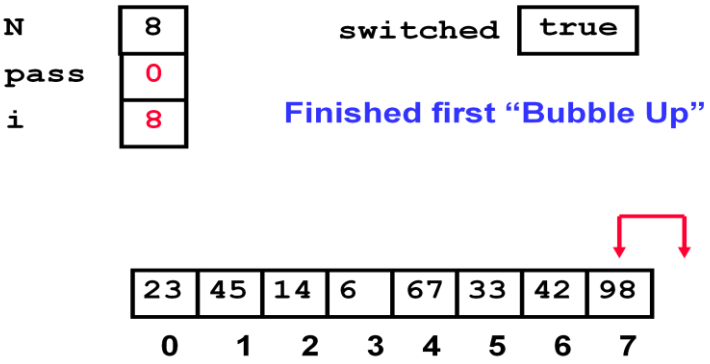
Pass 0







Like wise the iterations will continue for the Pass 0 with adjacent values comparisons and swapping. Finally after I pass that pass 0 the list will look as follows..



Now the last element will freeze out. That is it will not get considered for the next passes. Such that for each passes one element will get frozen out. The pass will get over if the elements haven't swapped for a complete iteration. It has been identified by means of switched Boolean variable. It will be false if swapping not happened.

Efficiency Analysis of Bubble sort:

- One traversal = move the maximum element at the end
- Traversal : n – i + 1 operations
- Running time:

$(n – 1) + (n – 2) + … + 1 = (n + 1) n / 2 = O(n^2)$ by applying the summation formula.

5.2QUICK SORT

- A sorting technique that sequences a list by continuously dividing the list into two parts and moving the lower items to one side and the higher items to the other. It starts by picking one item in the entire list to serve as a pivot point.
- The pivot could be the first item or a randomly chosen one. All items that compare lower than the pivot are moved to the left of the pivot; all equal or higher items are moved to the right.
- It then picks a pivot for the left side and moves those items to left and right of the pivot and continues the pivot picking and dividing until there is only one item left in the group. It then proceeds to the right side and performs the same operation again. Also known as **Partition Exchange Sort.**

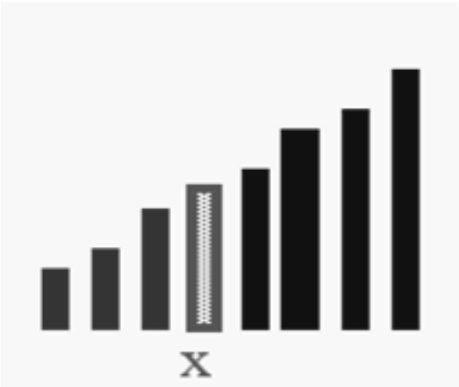
Quick sort algorithm applies a special designing technique called Divide and Conquer,

The *divide-and-conquer* strategy solves a problem by:
1. Breaking it into *sub problems* that are themselves smaller instances of the same type of problem
2. Recursively solving these sub problems
3. Appropriately combining their answers

Divide and Conquer in Quick sort
Divide: If the sequence S has 2 or more elements, select an element x from S to be your **pivot**. Any arbitrary element, like the last, will do. Remove all the elements of S and divide them into 3 sequences:
L, holds S's elements less than x
E, holds S's elements equal to x
G, holds S's elements greater than x
2) **Recurse**: Recursively sort L and G
3) **Conquer**: Finally, to put elements back into S in order, first inserts the elements of L, then those of E, and those of G.
Step 1: Select: pick an element

Step 2: Divide: rearrange elements so that x goes to its final position E

Step 3: Recurse and conquer: recursively sort



Quick Sort Algorithm:

```
#include<stdio.h>
#include<conio.h>
void qsort(int arr[20], int left, int right);
int main()
{
 int arr[30];
 int i,n;
 printf("Enter total no. of the elements : ");
```

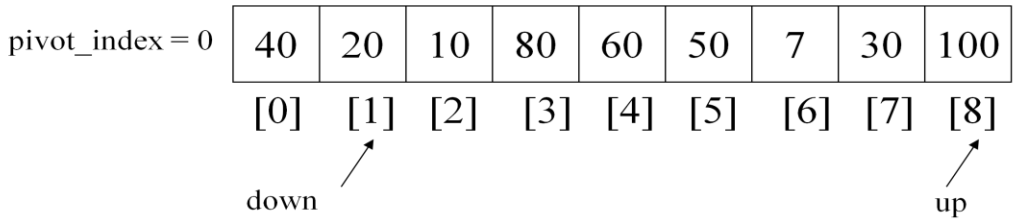
```
 scanf("%d",&n);
 printf("Enter total %d elements : \n",size);
 for(i=0; i<n; i++)
    scanf("%d",&arr[i]);
 qsort(arr,0,n-1);
 printf("Quick sorted elements are as  : \n");
 for(i=0; i<n; i++)
    printf("%d\t",arr[i]);
 getch();
 return 0;
}
void qsort(int arr[20], int left, int right)
{
 int i,j,pivot,tmp;
 if(left<right)
 {
   pivot=left;
   i=left+1;
   j=right;
   while(i<j)
   {
     while(arr[i]<=arr[pivot] && i<right)
        i++;
     while(arr[j]>arr[pivot])
        j--;
     if(i<j)
     {
        tmp=arr[i];
        arr[i]=arr[j];
        arr[j]=tmp;
     }
   }
   tmp=arr[pivot];
   arr[pivot]=arr[j];
   arr[j]=tmp;
   qsort(arr,left,j-1);
   qsort(arr,j+1,right);
 }
}
```

The partition method uses two while statements one is for incrementing the variable to arrange the sequence in left side with least values. Another while statement for decrementing the variable to arrange the sequence in right side with greatest values when compared with the pivot element.

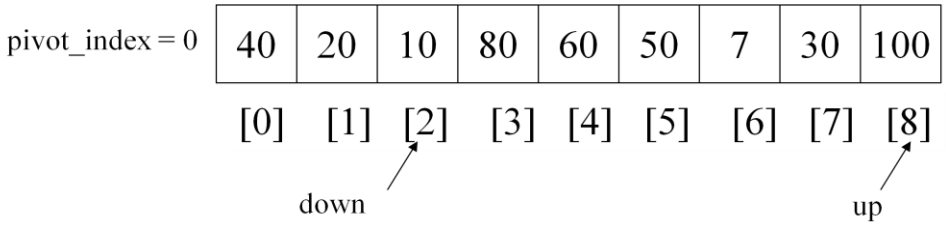
Execution for Partition method:

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array

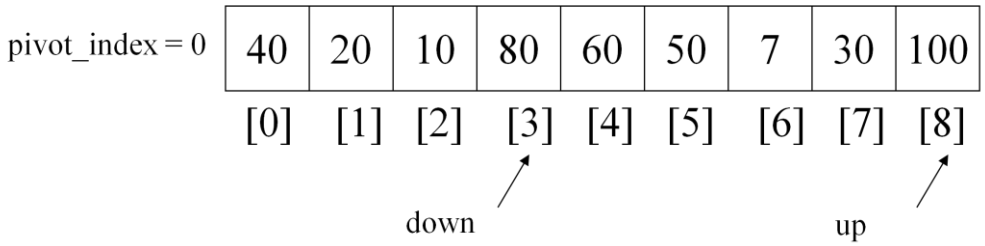
40	20	10	80	60	50	7	30	100



Compare 20 and pivot element 40 the 20 is least value so down alone get incremented.

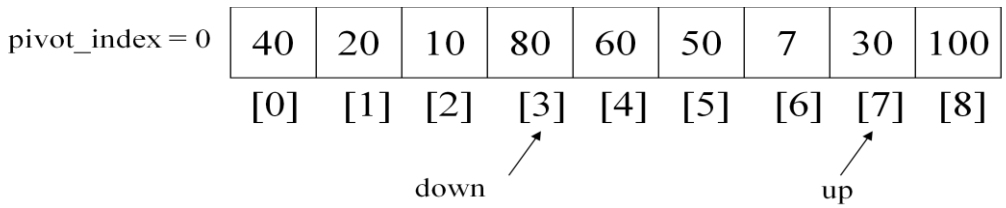


Again compare pivot element and element pointing down, 10 is least value again so down value get incremented.

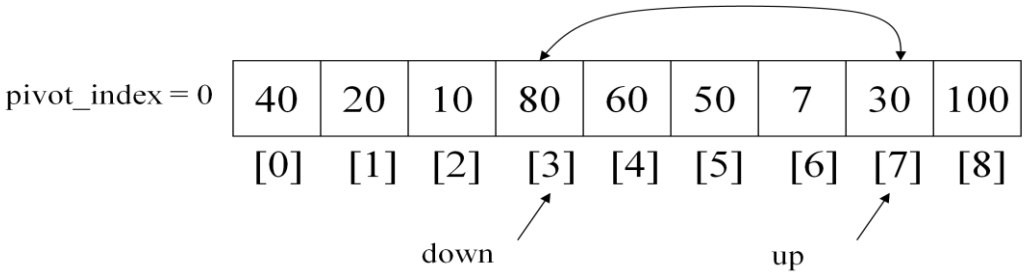


But Here the situation is 80 value pointed by down index is high when compared with the pivot element. So 1st while statement get terminated.

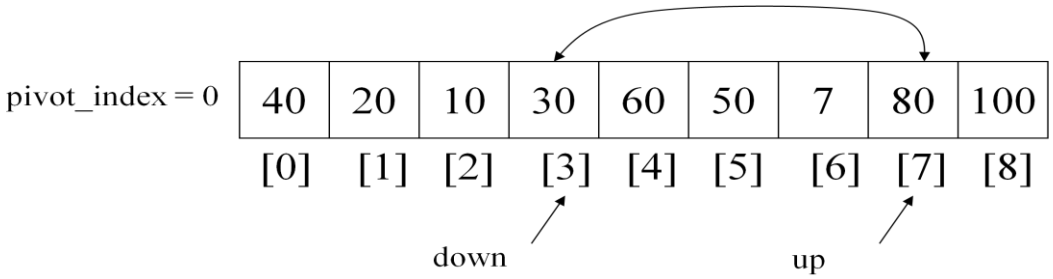
Now, up value will start to get decremented. The array will be as follows



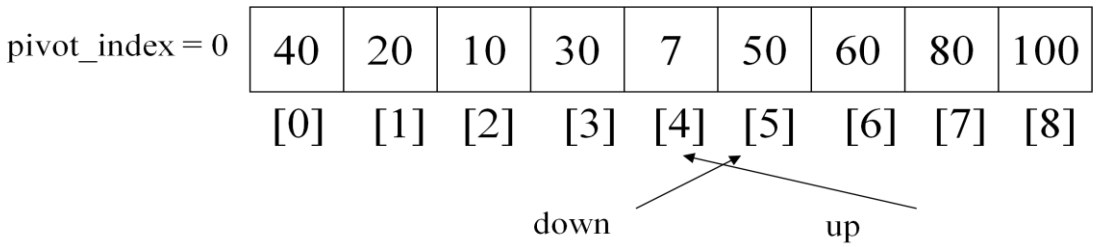
Up value will get decrement if the value pointing up is greater than pivot element, but now the situation is up pointing element is least when compared with the pivot element. So now second while also get terminated.



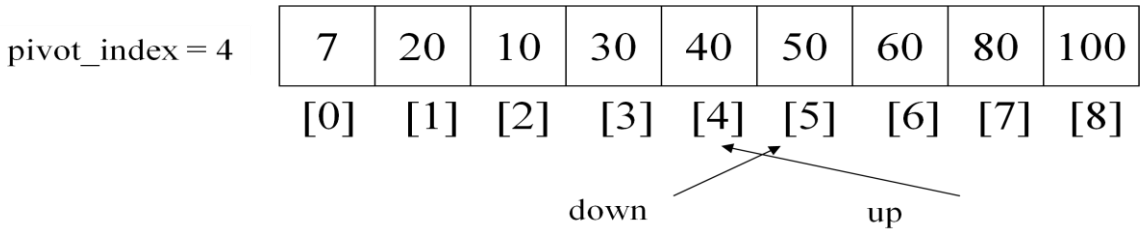
Now after the end of two while statements if down is less than up swap these two values to arrange left sub array with least values in the left side of the pivot element and greatest values at the right side of the pivot element.



Continue the two whiles again and find two values to get swapped until down < up



So now stop the iteration and swap the pivot element towards the up position like below.



Current pivot index has been updated with the value 4 and the pivot element also swapped to the pivot index.

These Sub arrays are now again introduced into the partition with the above steps. The two sub arrays are from 0 to 3 and 4 to 8. So the first sub array again partitioned into two, depending upon pivot element position. The quick sort procedure with the partition method is as follows.

```
void quicksort ( int a[ ], int left, int right )
{
        int mid;
        if ( right > left )
        {
                mid = quicksort (a, left, right) ;
                quicksort ( a, left, mid - 1 ) ;
                quicksort ( a, mid+ 1, right ) ;
        }
}
```

Here the variable i is the partition index. And accordingly the quick sort performs two partitions each time depending upon the pivot index.

Complexity Analysis:

- Recursion:
  - Partition splits array in two sub-arrays of size n/2
  - Quick sort each sub-array
- Depth of recursion tree $O(\log_2 n)$
- Number of accesses in partition O(n)

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

The Master theorem can be applied to calculate the efficiency. The theorem is as follows,

$$T(n) = \begin{cases} \Theta(n) & a < b \\ \Theta(n \log_b n) & a = b \\ \Theta(n^{\log_b a}) & a > b \end{cases}$$

For the Quick sort the standard format is T(n) = aT(n/b)+F(n)

Where a and b are Number of recursions and number of partitions respectively.

For quick sort a=2 and b=2, so a and b are equal so the efficiency is **θ (n log n)**

Selection sorts

- **Selection sort** is a sorting algorithm, specifically an in-place comparison sort. It has O (n^2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort.
- Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

Three types of selection sorts are discussed here,

i) Straight Selection sort
ii) Binary tree Sort
iii) Heap sort

3.3 SELECTION SORT

Definition:

Selection sort or push-down sort implements selection phase in which either largest or smallest element find from the list and it is swapped to the last or first position respectively.

Algorithm:

The algorithm works as follows:

1. Find the largest value in the list
2. Swap it with the value in the last position
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

Implementation in C:

```
#include <stdio.h>
#include<conio.h>
void main()
{
  int array[100], n, c, d, position, swap;
   printf("Enter number of elements\n");
  scanf("%d", &n);
   printf("Enter %d integers\n", n);
   for ( c = 0 ; c < n ; c++ )
     scanf("%d", &array[c]);
  for ( c = 0 ; c < ( n - 1 ) ; c++ )
  {
    position = c;
```

```
    for ( d = c + 1 ; d < n ; d++ )
    {
      if ( array[position] > array[d] )
        position = d;
    }
    if ( position != c )
    {
      swap = array[c];
      array[c] = array[position];
      array[position] = swap;
    }
  }
  printf("Sorted list in ascending order:\n");
   for ( c = 0 ; c < n ; c++ )
    printf("%d\n", array[c]);
  getch();
}
```

Execution:

For example, consider the following array, shown with array elements in sequence separated by commas:

63, 75, 90, 12, 27

The leftmost element is at index zero, and the rightmost element is at the highest array index, in our case, 4 (the effective size of our array is 5).

The largest element in this effective array (index 0-4) is at index 2. We have shown the largest element and the one at the highest index in **bold**. We then swap the element at index 2 with that at index 4. The result is:

63, **75, 27, 12, 90**

We reduce the effective size of the array to 4, making the highest index in the effective array now 3. The largest element in this effective array (index 0-3) is at index 1, so we swap elements at index 1 and 3 (in **bold**):

63, 12, 27, 75, 90

The next two steps give us:

27, 12, 63, 75, 90
12, 27, 63, 75, 90

Now the list is sorted.

Complexity Analysis:

- Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array.
- Selecting the lowest element requires scanning all n elements (this takes $n - 1$ comparisons) and then swapping it into the first position.
- Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on, for $(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2 \in \Theta(n^2)$ comparisons.

Insertion Sorts

- A simple sorting technique that scans the sorted list, starting at the beginning, for the correct insertion point for each of the items from the unsorted list.
- Similar to the way people manually sort items, an insertion sort is not efficient for large lists, but is relatively fast for adding a small number of new items periodically.

Three insertion sort related sorting techniques are discussed here they are

i) Simple Insertion Sort
ii) Shell sort
iii) Address Calculation sort.

3.4. INSERTION SORT:

Definition:

An insertion sort is one that sorts a set of records by inserting records into an existing sorted file.

- Based on the technique used by card players to arrange a hand of cards
  - Player keeps the cards that have been picked up so far in sorted order
  - When the player picks up a new card, he makes room for the new card and then inserts it in its proper place

Algorithm:

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

```
void main()
{
 int n, array[1000], i, j, t;

 printf("Enter number of elements\n");
 scanf("%d", &n);

 printf("Enter %d integers\n", n);

 for (i = 0; i< n; i++) {
  scanf("%d", &array[i]);
 }

 for (j = 1 ; j < n; j++)
 {
  k=j ;

  while ( k> 0 && array[k] < array[k-1])
  {
   t          = array[k];
   array[k]   = array[k-1];
   array[k-1] = t;

   k--;
  }
 }

 printf("Sorted list in ascending order:\n");

 for (i= 0; i< n; i++) {
  printf("%d\n", array[i]);
 }
getch();
}
```

Execution with example:

Initially x[0] may be thought of as a sorted file, after each repetition elements from x[0] to x[k] are ordered. By moving all elements greater than y to the right direction we can insert y in the correct position.

Sort: 34 8 64 51 32 21

- 34 8 64 51 32 21
  - The algorithm sees that 8 is smaller than 34 so it swaps.

- 8 34 64 51 32 21
  - 51 is smaller than 64, so they swap.
- 8 34 51 64 32 21
  - The algorithm sees 32 as another smaller number and moves it to its appropriate location between 8 and 34.

- 8 32 34 51 64 21
  - The algorithm sees 21 as another smaller number and moves into between 8 and 32.
- Final sorted numbers:
- 8 21 32 34 51 64

Efficiency Analysis:

The algorithm needs n pass through for n elements with n comparisons each time so the efficiency of the algorithm is O (n^2).

3.5 SHELL SORT

- Shell sort, also known as the **diminishing increment sort**, is one of the oldest sorting algorithms, named after its inventor Donald. L. Shell (1959).

Founded by Donald Shell and named the sorting algorithm after him in 1959.

- 1st algorithm to break the quadratic time barrier but few years later, a sub quadratic time bound was proven
- Shell sort works by comparing elements that are distant rather than adjacent elements in an array or list where adjacent elements are compared.
- Shell sort uses a sequence h_1, h_2, …, h_t called the ***increment sequence***. Any increment sequence is fine as long as $h_1 = 1$ and some other choices are better than others.

We can choose the increment sequence with the following range,

$$h_t = \left\lfloor \frac{N}{2} \right\rfloor$$

$$h_k = \left\lfloor \frac{h_{k+1}}{2} \right\rfloor$$

$$h_1 = 1$$

```
#include<stdio.h>
#include<conio.h>
int main()
{
 int arr[30];
 int i,j,k,tmp,num;
 printf("Enter total no. of elements : ");
 scanf("%d", &num);
 for(k=0; k<num; k++)
 {
  printf("\nEnter %d number : ",k+1);
  scanf("%d",&arr[k]);
 }
 for(i=num/2; i>0; i=i/2)
 {
  for(j=i; j<num; j++)
  {
```

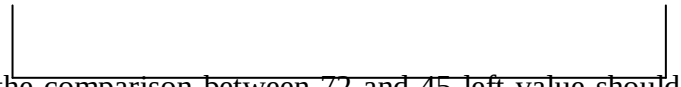
```
    tmp=arr[j];
       for(k=j; k>=i; k=k-i)
    {
     if(tmp<arr[k-i])

          {
          arr[k]=arr[k-i];
                                                          }
     else
     {
         break;
          }
          arr[k-i]=tmp;
      }
     }
    }
 printf("\t**** Shell Sorting ****\n");
 for(k=0; k<num; k++)
    printf("%d\t",arr[k]);
 getch();
 return 0;
}
```

Execution of Algorithm:

72	**8**	**45**	**65**	**82**	**45**	**46**	**42**	**13**

For the Span value 5 the comparisons and swapping will be as follows

72	**8**	**45**	**65**	**82**	**45**	**46**	**42**	**13**

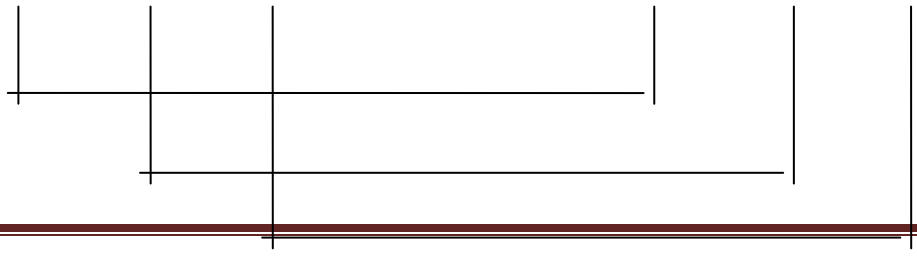


Now the comparison between 72 and 45 left value should be least value so 45 swapped to first location. Now the altered list was,

45	**8**	**45**	**65**	**82**	**72**	**46**	**42**	**13**

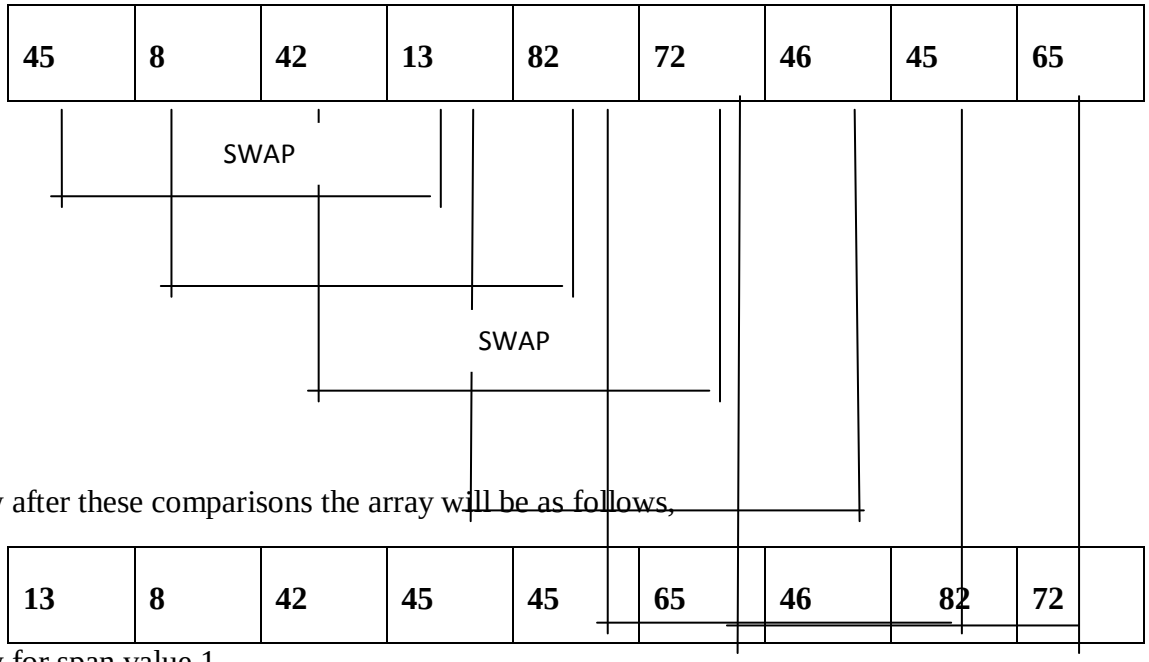
Then span 5 repeatedly compare following elements,

45	**8**	**45**	**65**	**82**	**72**	**46**	**42**	**13**



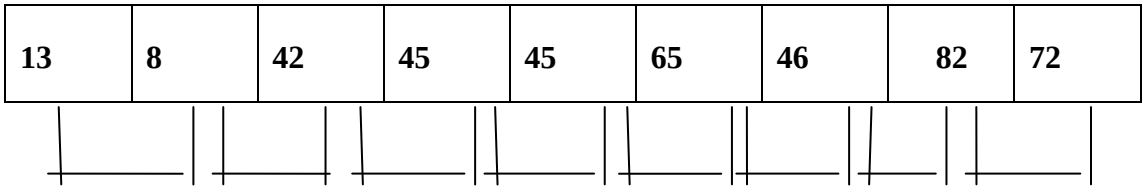
On these comparisons with span value 5 the elements 45 and 42 , 65 and 13 are get interchanged.

For SPAN=3



Now after these comparisons the array will be as follows,

Now for span value 1



Then the altered array will be as follows,

8	13	42	45	45	46	65	72	82

The resultant array is sorted now.

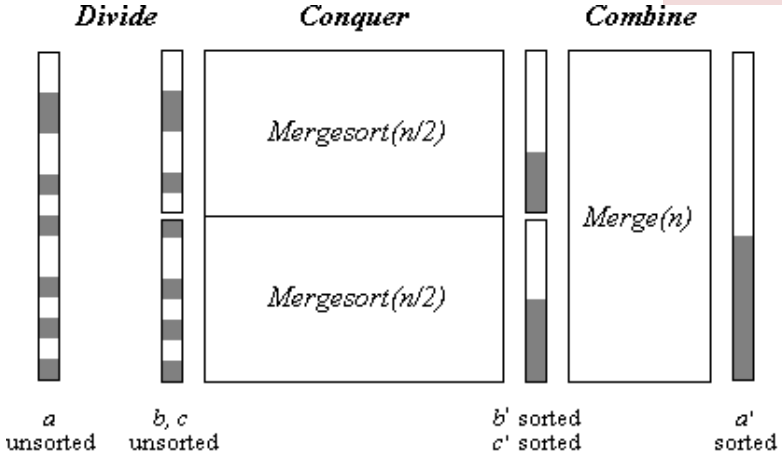
Complexity Analysis:

- In particular, when $h = \Theta(N^{7/15})$ and $g = \Theta(h^{1/5})$,
- the average time of sorting is $O(N^{23/15})$.

5.6 MERGE SORT

Definition;

The Merge sort algorithm is based on divide and conquers strategy. First, the sequence to be sorted is decomposed into two halves (***Divide***). Each half is sorted independently (***Conquer***). Then the two sorted halves are merged to a sorted sequence (***Combine***).



Algorithm:

The following procedure *merge sort* sorts a sequence *a* from index *min* to index max.

```
#include<stdio.h>

#include<conio.h>

void merge(int [],int ,int ,int );

void part(int [],int ,int );

int main()

{

 int arr[30];

 int i,size;

 printf("\n\t------- Merge sorting method -------\n\n");

 printf("Enter total no. of elements : ");

 scanf("%d",&size);

 for(i=0; i<size; i++)

 {
```

```
  printf("Enter %d element : ",i+1);
  scanf("%d",&arr[i]);
 }
 part(arr,0,size-1);
 printf("\n\t------- Merge sorted elements -------\n\n");
 for(i=0; i<size; i++)
 printf("%d ",arr[i]);
 getch();
 return 0;
}
void part(int arr[],int min,int max)
{
 int mid;
 if(min<max)
 {
  mid=(min+max)/2;
  part(arr,min,mid);
  part(arr,mid+1,max);
  merge(arr,min,mid,max);
 }
}
```

First, index *m* in the middle between *lo* and *hi* is determined. Then the first part of the sequence (from *lo* to *m*) and the second part (from *m*+1 to *hi*) are sorted by recursive calls of *merge sort*. Then the two sorted halves are merged by procedure *merge*. Recursion ends when *lo* = *hi*, i.e. when a subsequence consists of only one element.

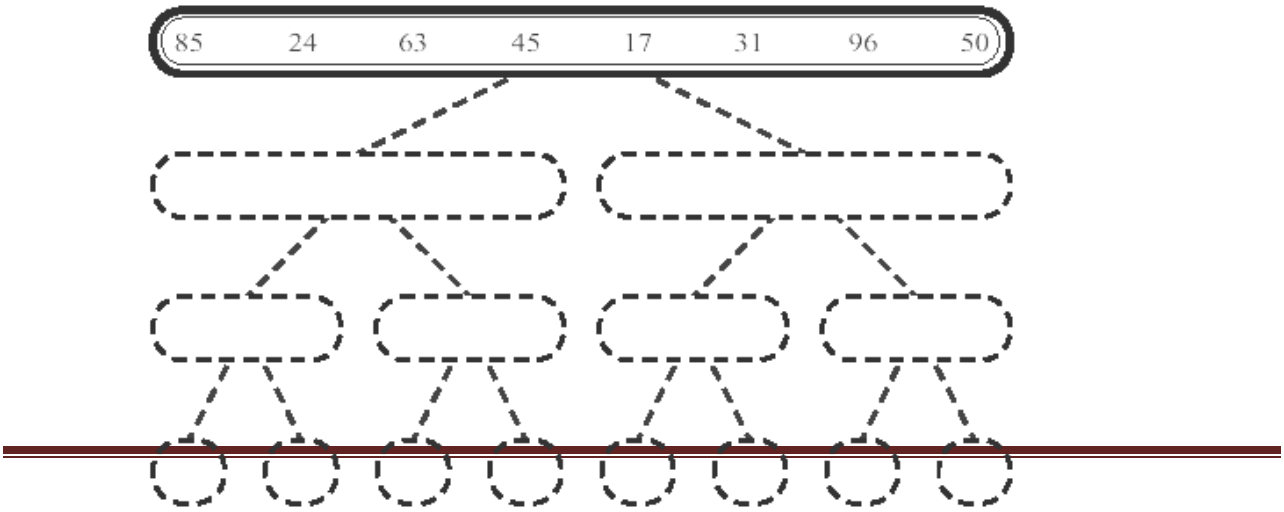
The main work of the Merge sort algorithm is performed by function *merge*. There are different possibilities to implement this function.
The Merge method defined as follows,

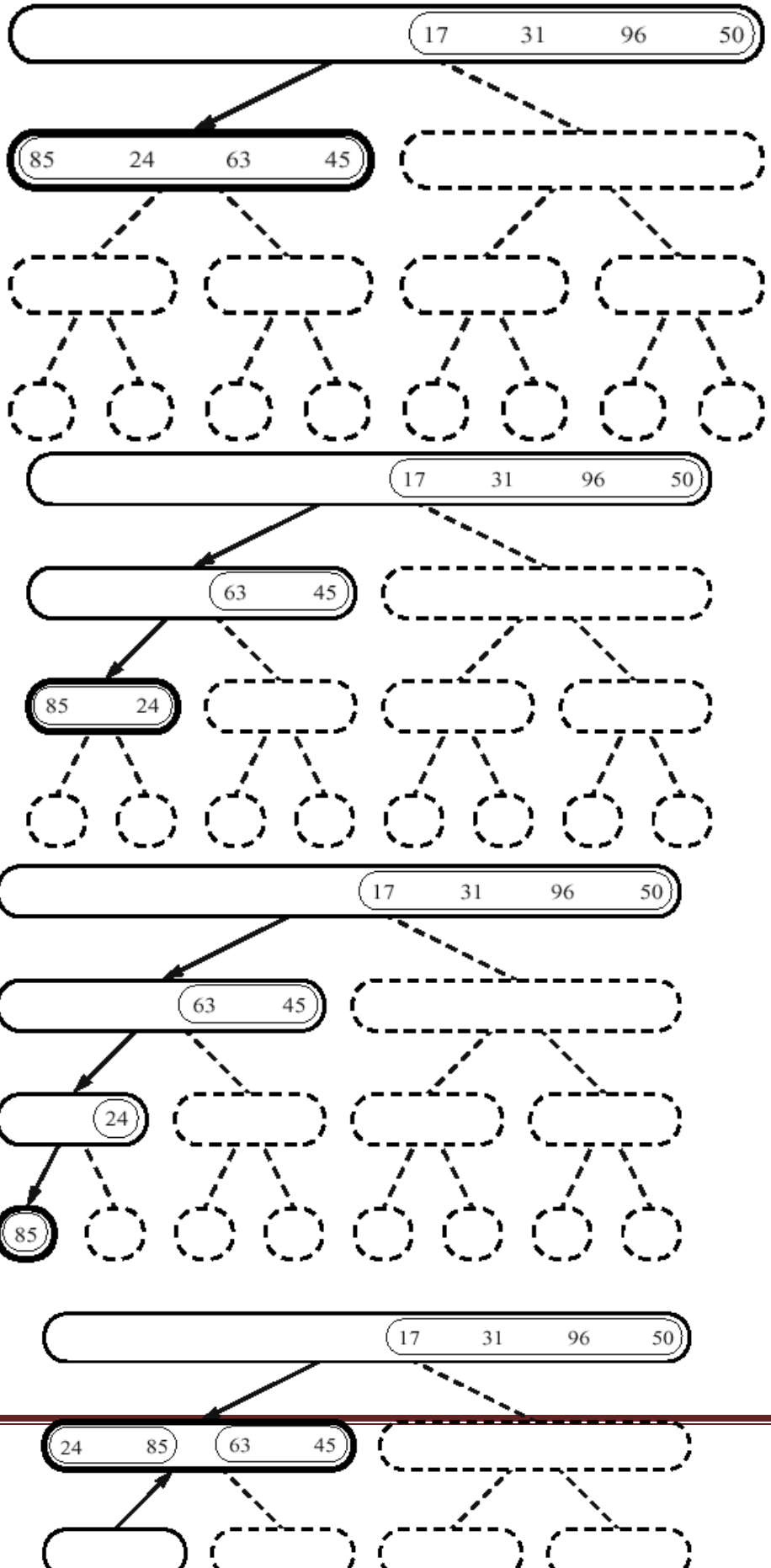
```
void merge(int arr[],int min,int mid,int max)

{

  int tmp[30];

  int i,j,k,m;

  j=min;

  m=mid+1;

  for(i=min; j<=mid && m<=max ; i++)

  {

    if(arr[j]<=arr[m])

    {

       tmp[i]=arr[j];

       j++;

    }

    else

    {

       tmp[i]=arr[m];

       m++;

    }

  }

  if(j>mid)

  {

    for(k=m; k<=max; k++)
```

```
    {
      tmp[i]=arr[k];
      i++;
    }
  }
  else
  {
    for(k=j; k<=mid; k++)
    {
      tmp[i]=arr[k];
      i++;
    }
  }
  for(k=min; k<=max; k++)
    arr[k]=tmp[k];
}
```

This variant of function *merge* requires in each case exactly the same number of steps – no matter if the input sequence is random, sorted or sorted in opposite direction.

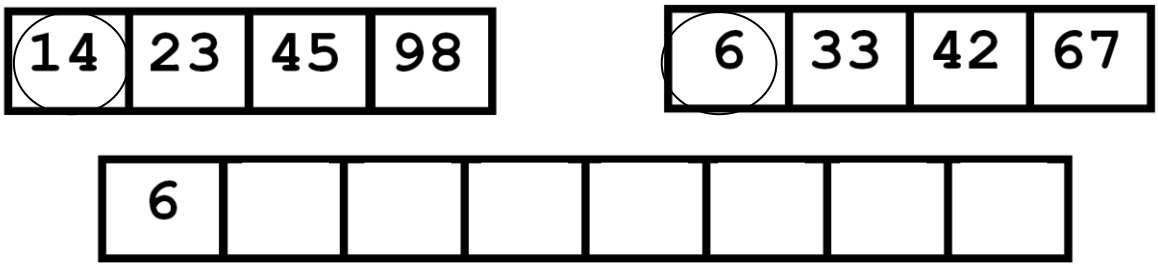
Execution:




17 31 96 50
85 24 63 45
17 31 96 50
63 45
85 24
17 31 96 50
63 45
24
85
17 31 96 50
24 85 63 45

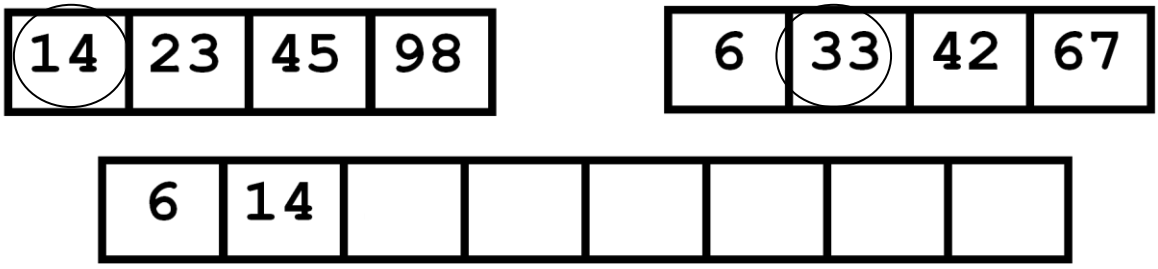
Just like the above steps the second recursion will sort the sequence at the right hand side. Finally the both partitioned sorted sub arrays are get merged in the sorted sequence.

The final merging steps are as follows,

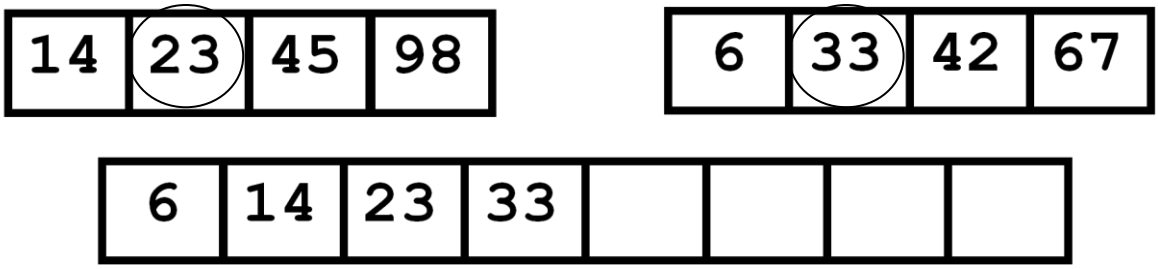


14 and 6 are compared and 6 placed in the first position.

Next comparison will be



Now comparing 14 and 33 14 is least value so place it in the second place. Next comparison,



Likewise the comparisons done till reaches the end of the sub arrays and the list will be sorted at last, the sorted list will be as follows,

6	14	23	33	42	45	67	98

Complexity Analysis:

The straightforward version of function *merge* requires at most 2*n* steps (*n* steps for copying the sequence to the intermediate array *b*, and at most *n* steps for copying it back to array *a*). The time complexity of *merge sort* is therefore

$T(n) \leq 2n + 2\,T(n/2)$ and

$T(1) = 0$

The solution of this recursion yields

$T(n) \leq 2n \log(n) \in O(n \log(n))$

Thus, the merge sort algorithm is optimal, since the lower bound for the sorting problem of Ω (n log (n)) is attained.

In the more efficient variant, function *merge* requires at most 1.5*n* steps (*n*/2 steps for copying the first half of the sequence to the intermediate array *b*, *n*/2 steps for copying it back to array *a*, and at most *n*/2 steps for processing the second half). These yields a running time of *merge sort* of at most 1.5*n* log (*n*) steps.

5.7 RADIX SORT:

Definition:

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value

- sort by the least significant digit first (counting sort)
    - Numbers with the same digit go to same bin
- reorder all the numbers: the numbers in bin 0 precede the numbers in bin 1, which precede the numbers in bin 2, and so on
- sort by the next least significant digit
- continue this process until the numbers have been sorted on all k digits

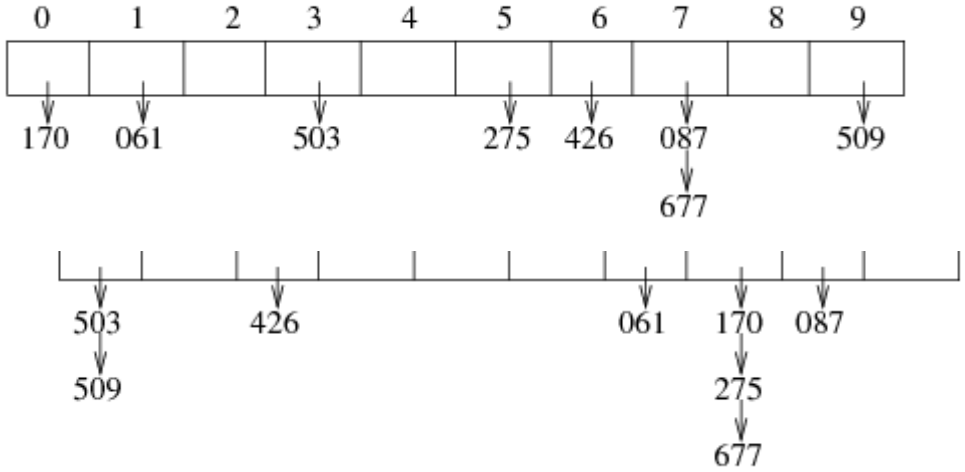
Algorithm:

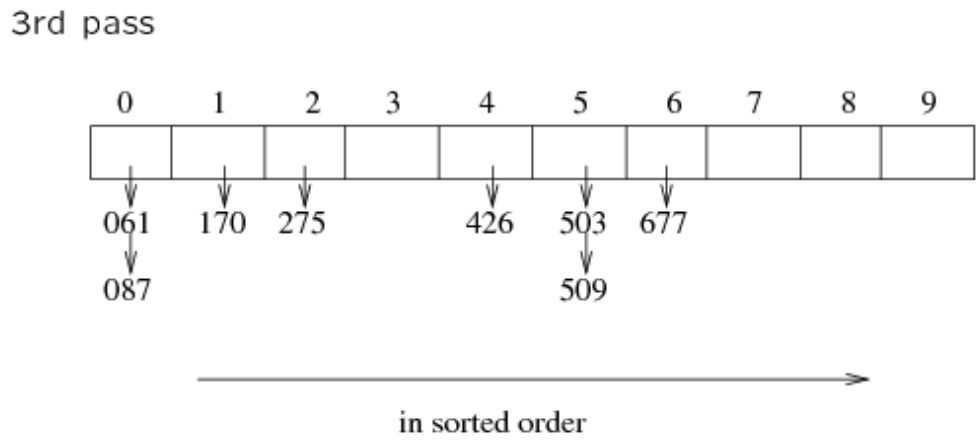
- Extra information: every integer can be represented by at most k digits
    - $d_1d_2\ldots d_k$ where d_i are digits in base r
    - d_1: most significant digit
    - d_k: least significant digit

```
Algorithm RadixSort(A, n, d)
1.   for 0 ≤ p ≤ 9
2.       do Q[p] := empty queue;
3.   D := 1;
4.   for 1 ≤ k ≤ d
5.       do
6.           D := 10 * D;
7.           for 0 ≤ i < n
8.               do t := (A[i] mod D) div (D/10);
9.                   enqueue(A[i], Q[t]);
10.          j := 0;
11.          for 0 ≤ p ≤ 9
12.              do while Q[p] is not empty
13.                     do A[j] := dequeue(Q[p]);
14.                        j := j + 1;
```

Example: 275, 087, 426, 061, 509, 170, 677, 503

1st pass





Complexity Analysis:

- k passes over the numbers (i.e. k counting sorts, with range being 0..r)
- each pass takes O(N+r)
- total: O(Nk+rk)
- r and k are constants: O(N)

5.8.SEARCHING:

Search Algorithm:

A search algorithm is an algorithm that accepts an argument and tries to find a record whose key is 'a'. The algorithm may return the entire record or more commonly return the pointer to that record.

Basic Terminologies:

File: A table or file is group of elements.

Record: Each field in a table or file is known as record.

Key: Associated with each record, which is used to differentiate among different records.

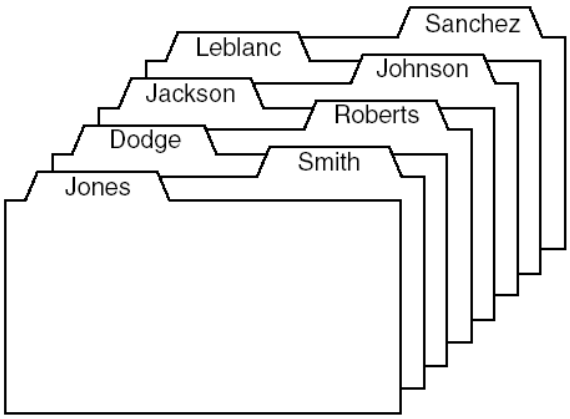
Three different keys,

Internal Key: The key is contained within the record at a specific offset from the start of the record, such a key is Internal or Embedded key.

External Key: There is a separate table of keys that includes pointers to the records. Such keys are External keys.

Primary Key: For every file or table there is at least one set of a key that is unique.

Secondary Key: For a table or file if searching done on the record which is not unique such a key is secondary key.



Records and Keys

Internal search: Searching key data stored in main memory

External search: Searching key data stored in auxiliary memory

Retrieval: Output of a successful search

5.8.1SEQUENTIAL SEARCHING:

Each searching function has two input parameters:

1. First is the *list* to be searched;
2. Second is the *target* key for which we are searching.

Each searching function will also have an output parameter and a returned value:

- The returned value has type Error code and indicates whether or not the search is successful in appending an entry with the target key.
- If the search is successful, then the returned value is success, and the output parameter called position will locate the target within the list.
- If the search is unsuccessful, then the value not present is returned and the output parameter may have an undefined value or a value that will differ from one method to another.

Definition:

Linear search or **sequential search** is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found. Linear search is the simplest search algorithm; it is a special case of brute-force search.

Algorithm:

```c
#include<stdio.h>

main()
{
  int array[100], search, c, number;

  printf("Enter the number of elements in array\n");
  scanf("%d",&number);

  printf("Enter %d numbers\n", number);

  for ( c = 0 ; c < number ; c++ )
    scanf("%d",&array[c]);

  printf("Enter the number to search\n");
  scanf("%d",&search);

  for ( c = 0 ; c < number ; c++ )
  {
    if ( array[c] == search )     /* if required element found */
    {
    printf("%d is present at location %d.\n", search, c+1);
    break;
  }
  }
  if ( c == number )
    printf("%d is not present in array.\n", search);

  return 0;
}
```

This search is applicable to a table organized either as an array or as a linked list. The algorithm examines each key in turn upon finding one that matches the search argument, its index is returned. If no match found -1 is returned.

For efficient searching algorithm we can add a node called sentinel node an extra key inserted at the end of the array.

```c
k[n] = key;
 for (i = 0; key != k[i]; i++) ;
 if (i < n)  return(i);
 else return(-1);
```

Let p (i) be the probability that record i is retrieved.

$p(0)+ p(1)+ ... + p(n-1) = 1.$

Average number of comparisons:

$p(0) + 2p(1) + 3p(2) + ... + np(n-1)$

This number is minimized if

$p(0) \geqq p(1) \geqq p(2) \geqq ... \geqq p(n-1).$

Reordering a List for Maximum Search Efficiency

There are two search methods that accomplish the maximum searching efficiency are

i) Move to Front
ii) Transposition Method.

Move to Front:

In this method whenever a search is successful the retrieved record is removed from its current location in the list and is placed at the head of the list.

e.g. 9 5 6 8 7 2

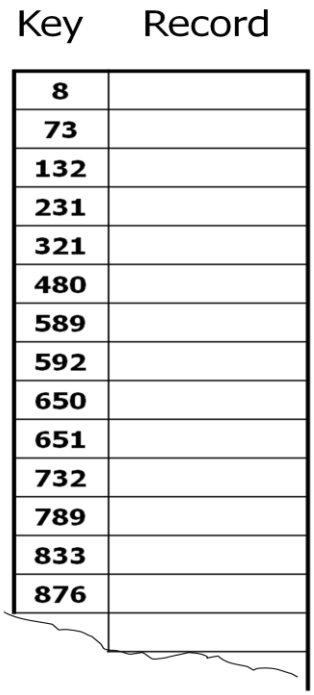
(1) search 6: 6 9 5 8 7 2

(2) search 8: 8 6 9 5 7 2

The retrieved record is moved to the head of the list.

Searching in an ordered table

If the table is in ascending or descending order of the record keys, then there are several techniques that can be used to improve the efficiency of searching.



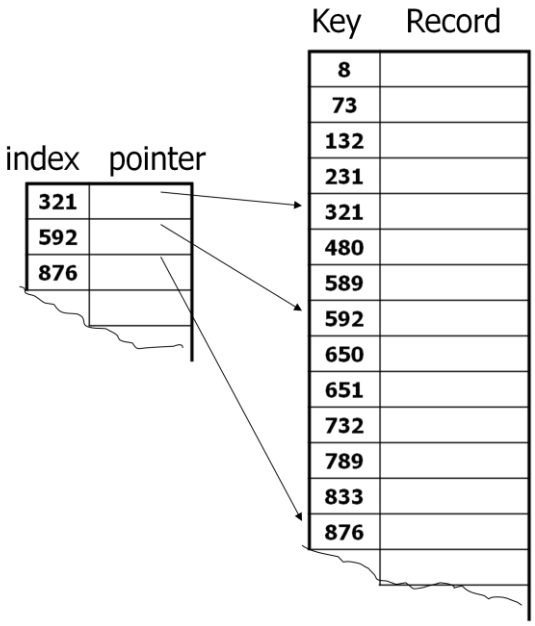
An obvious advantage in searching a sorted file is in the case that the argument keys are uniformly distributed over in the case if records are not sorted the comparisons needed are n. for a sorted sequence the comparisons needed are n/2.

The Indexed sequential Search:

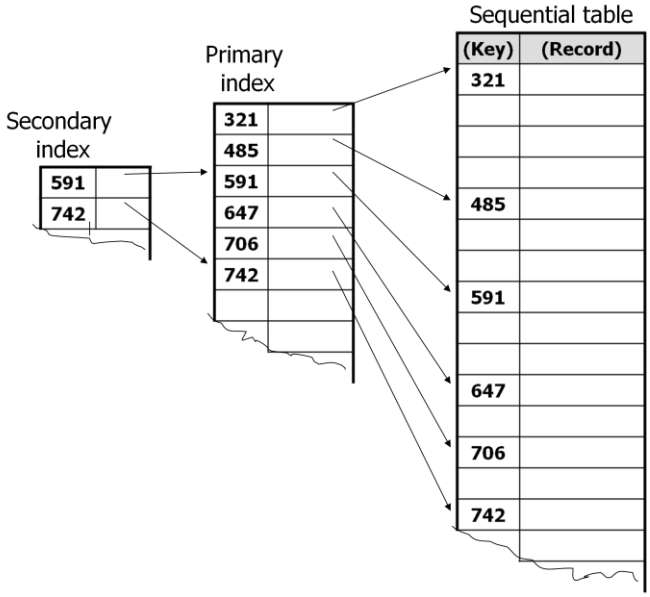
It is another method to improve the efficiency of searching in a sorted list. An auxiliary table called an index is set aside in addition to the sorted file itself.

But it needs extra space; this method is also known as indexed sequential method. Let r, k and key be three input values of this algorithm. Other values are kindex be an array of keys in the index, and let pindex be the array of pointers with in the index.

Indxsize size of the index table. N number of records in the main table.



- Deletion is done by flagging. Through that if the flag is set, the record can be ignored.
- Insertion is more difficult because it is necessary to shift all the records to make room for the new record.
- If the table is so large a single index cannot be sufficient to achieve efficiency. So we can maintain secondary index.
- The secondary index acts as an index to the primary index. Which will points entry to the sequential table.



5.8.2 BINARY SEARCH:

Definition:

A dichotomizing search in which the set of items to be searched is divided at each step into two equal, or nearly equal, parts, Also known as binary chop.

The most efficient method used for searching a sequential table is binary search. It doesn't need any primary or secondary index.

Algorithm Explanation:

A **binary search** or **half-interval search** algorithm finds the position of a specified value (the input "key") within a sorted array. At each stage, the algorithm compares the input key value with the key value of the middle element of the array.

- If the keys match, then a matching element has been found so its index, or position, is returned.
- Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or,
- if the input key is greater, on the sub-array to the right. If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

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

   int a[10],i,n,m,c=0,l,u,mid;

   printf("Enter the size of an array: ");
   scanf("%d",&n);

   printf("Enter the elements in ascending order: ");
   for(i=0;i<n;i++){
       scanf("%d",&a[i]);
   }

   printf("Enter the number to be search: ");
   scanf("%d",&m);

   l=0,u=n-1;
   while(l<=u){
       mid=(l+u)/2;
       if(m==a[mid]){
          c=1;
          break;
       }
       else if(m<a[mid])
      {
```

```
            u=mid-1;
        }
        else
            l=mid+1;
    }
    if(c==0)
        printf("The number is not found.");
    else
        printf("The number is found.");

    getch();
}
```

Example:

For example, consider the following sequence of integers sorted in ascending order and say we are looking for the number 55:

0	5	13	19	22	41	55	68	72	81	98
0	1	2	3	4	5	6	7	8	9	10

The search space contains indices 0 through 10 respective low and high indices As described above, we now choose the mid value 0 + 10 /2 =5

The value is in the location 5 is 41 and it is smaller than the target value.

From this we conclude not only that the element at index 5 is not the target value, but also that no element at indices between 0 and 5 can be the target value, because all elements at these indices are smaller than 41, which is smaller than the target value. This brings the search space down to indices 6 through 10:

55	68	72	81	98
6	7	8	9	10

Proceeding in a similar fashion, we chop off the second half of the search space and are left with: (6+10/2 = 8 mid value in 8 is 72 which is greater than 55 so the indices are 6 and 7.

55	68
6	7

Depending on how we choose the median of an even number of elements we will either find 55 in the next step or chop off 68 to get a search space of only one element. Either way, we conclude that the index where the target value is located is 7.

Efficiency Analysis:

- That is, which of the following will first be < 1? < $n/2$, $n/4$, $n/8$, $n/16$, ..., $n/2k$, ...
- We solve the equation: $n/2k < 1$, and get $k > \log 2n$ So if we set $k = [\log 2n]$, then we know that after that many iterations of the while loop, we will have found our item, or concluded that it was not in the list.

5.9 HASHING

Hash Table:

- A hash table is an array-like data structure that associates its input (the key) with the associated output (the record, or value).
- A **hash table** or **hash map** is a data structure that uses a hash function to efficiently map certain identifiers or keys (e.g., person names) to associated values (e.g., their telephone numbers). The hash function is used to transform the key into the index (the *hash*) of an array element (the *slot* or *bucket*) where the corresponding value is to be sought.

Hash Function:

- At the heart of the hash table algorithm is a simple array of items; this is often simply called the *hash table*. Hash table algorithms calculate an index from the data item's key and use this index to place the data into the array.
- The implementation of this calculation is the hash function,

  f: *index* = *f*(*key*,*arrayLength*)
- The hash function calculates an *index* within the array from the data *key*. *arrayLength* is the size of the array**.**

Types of Hashing:

→ **Static Hashing:**
The hash function maps search key value to a fixed set of locations.

→ **Dynamic Hashing:**
The hash table can grow to handle more items. The associated hash function must change as the table grows.

Methods of Hashing Function:

i) Mid Square Method
ii) Module Division (or) Division Remainder

iii) Folding Method
   a) Fold shifting Method
   b) Fold boundary method
iv) Pseudo Random Number Generator Method
v) Digit (or) Character Extraction method.
vi) Radix Transformation.

Applications of Hash Tables:
→ Database Systems
→Symbol Tables
→Data Dictionaries
→Browse caches

COLLISION

- When a memory location filled if another value of the same memory location comes then there occurs collision.
- When an element is inserted it hashes to the same value as an already inserted element, then it produces collision and need to be resolved.

Collision resolving methods:
1. Separate chaining (or) External Hashing
2. Open addressing (or) Closed Hashing (linear probing,quadratic probing, double hashing)

When two keys map to the same location in the hash table is referred as **collision.**

5.10 SEPARATE CHAINING (OPEN HASHING) (OR) EXTERNAL HASHING:

Separate chaining is a collision resolution technique, in which we can keep the list of all elements that hash to same value. This is called as separate chaining because each hash table element is a separate chain (linked list).
Each link list contains the entire element whose keys hash to the same index. All keys that map to the same hash value are kept in a list (or "bucket")..
The example is given as follows.
The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.

Advantages:
- Unlimited number of elements
- Unlimited number of collisions

Disasdvantages:
- Overhead of multiple linked lists
- The elements are not evenly distributed. Some hash index may have more elements and some may not have anything.
- It requires pointers which require more memory space. This leads to slow the algorithm down a bit because of the time required to allocate the new cells, and also essentially requires the implementation of second data structure.

5.11OPEN ADDRESSING (CLOSED HASHING):

Open addressing hashing is an alternating technique for resolving collisions with

linked list. In this system if a collision occurs, alternative cells are tried until an empty cell is found.

The cell h0(x), h1(x), h2(x)……. Are tried in succession, where hi(x) = (Hash (X) + F(i)) mod Table_Size with F(0) = 0. The function F is the collision resolution strategy. This technique is generally used where storage space is large. Arrays are used here as hash tables.

Definition: The technique of finding the availability of another suitable empty location in the hash table when the calculated hash address is already occupied is known as open Addressing. There are three common collisions resolving strategies

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

 Linear probing, in which the interval between probes is fixed (usually 1)
 Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
 Double hashing, in which the interval between probes is computed by another hash function

A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance dramatically degrades when the load factor grows beyond 0.7 or so.

5.12.1 Linear probing (using neighboring slots)

Probing is the process of a placing in next available empty position in the hash table. The Linear probing method searches for next suitable position in a linear manner(next by next). Since this method searches for the empty position in a linear way, this method is referred as linear probing. In linear probing for the ith probe, the position to be tried is, (h(k) + i) mod Table_Size, where 'f' is a linear function of i, F(i)=i. This amounts to trying cells sequentially in search of an empty cell.

Example for Linear Probing:

 Insert the keys 89, 18, 49, 58, 69 into a hash table using in the same hash function as before and the collision resolving strategies. F(i)=i.

size Empty table After 89 After 18 After 49 After 58 After 69
0 49 49 49
1 58 58
2 69
3
4
6 18 18 18 18
7 89 89 89 89 89

Solution: In this e.g initially 89 is inserted at index '9'. Then 18 is inserted at index 8. The first collision occurs when 49 is inserted. It is put in the next available index namely '0' which is open. The key 58 collides with 18, 89, 49 afterwards it found an empty cell at the index

Similarly collision 69 is handled. If the table is big enough, a free cell can be always be found, but the time to do so can get quite large. Even if the table is relatively empty, blocks of occupied cells start forming. This is known as primary clustering means that any key hashes into the cluster will require several attempts to resolve the collision and then it will add to the cluster.

 H(k) = key value mod Table size

 The figure shows linear probing example. h(j)=h(k), so the next hash function,h1 is used. A second collision occurs, so h2 is used. Linear probing suffers from primary clustering.

Algorithm for linear probing:

1. Apply hash function on the key value and get the address of the location.
2. If the location is free, then
   i) Store the key value at this location, else
   ii) Check the remaining locations of the table one after the other till an empty location is reached. Wrap around on the table can be used. When we reach the end of the table, start looking again from the beginning.
   iii) Store the key in this empty location.

3.End

Advantages of linear probing:

1. It does not require pointers.
2. It is very simpler to implement.

Disadvantages of linear probing:

1. It forms clusters, which degrades the performance of the hash table for sorting and retrieving data.
2. If any collision occur when the hash table becomes half full, it is difficult to find an empty location in the hash table and hence the insertion process takes a longer time.

5.12.2 QUADRATIC PROBING

Better behaviour is usually obtained with **quadratic probing**, where the secondary hash function depends on the re-hash index: **address = h(key) + c i2**

• Probe sequence: 0th probe = h(k) mod TableSize
1th probe = (h(k) + 1) mod TableSize
2th probe = (h(k) + 4) mod TableSize
3th probe = (h(k) + 9) mod TableSize
ith probe = (h(k) + i2) mod TableSize

on the **tth** re-hash. Quadratic probing does not suffer from primary clustering: keys hashing to the same area are not bad (A more complex function of **i** may also be used.) Since keys which are mapped to the same value by the primary hash function follow the same sequence of addresses, quadratic probing shows **secondary clustering**.
However, secondary clustering is not nearly as severe as the clustering shown by linear probes.
However, the collision elements are stored in slots to which other key values map directly, thus the potential for multiple collisions increases as the table becomes full.

5.11.3 DOUBLE HASHING:

Double hashing uses a secondary hash function **d**(**k**) and handles collisions by placing an item in the first available cell of the series (**i** + **jd**(**k**)) mod **N** for **j** = 0, 1, … , **N** – 1.
The secondary hash function **d**(**k**) cannot have zero values. The table size **N** must be a

prime to allow probing of all the cells **f(i) = i * g(k)** where g is a second hash function

• Probe sequence:

0th probe = h(k) mod TableSize
1th probe = (h(k) + g(k)) mod TableSize
2th probe = (h(k) + 2*g(k)) mod TableSize
3rd probe = (h(k) + 3*g(k)) mod TableSize
ith probe = (h(k) + i*g(k)) mod TableSize

Consider a hash table storing integer keys that handles collision with double hashing.

CLUSTERING

Linear probing is subject to a **clustering** phenomenon. Re-hashes from one location occupy a block of slots in the table which "grows" towards slots to which other keys hash. This exacerbates the collision problem and the number of re-hashed can become large.

5.12RE-HASHING

Collision:

Collision occurs when a hash value of a record being inserted hashes to an address that already contain a different record(i.e) when two key values hash to the same position.

Collision Resolution:

The process of finding another position for the collide record is said to be collision resolution stategy.

Two categories of Hashing:

→open Hashing
  a) Separate Chaining. -Linked list
→Closed Hashing
  a)Open addressing.
    i) Linear Probing. - $H_i(X)$ = (HASH(X) +i) % tablesize
    ii) Quadratic Probing - $H_i(X)$ = (HASH(X) $+i^2$) % tablesize
  iii)Double hashing. - $H_i(X)$ = (HASH(X) +i* $HASH_2(X)$ % tablesize
    $HASH_2(X)$=R-(X % R) R is a prime number that is smaller than the table size

If the table gets too full,then the rehashing method builds new table that is a about twice as big and scan down the entier original hashtable ,computing the new hash value for each element and inserting it in the new table .

Rehashing is very expensive operation, the running time is O(N),since there are N element to rehashing and the table size roughly 2N.

Rehashing can be implemented in several ways with quadratic probing such as:

→Rehashing,as soon as the table is half full;

→Rehashing only when an insertion fails.

→Rehashing when the table reaches a certain load factor.

Eg:13,15,24,6 ,23are to be inserted in hash table of size 7.

0	6
1	15
2	23
3	24
4	
5	
6	13

The table will be 70% full.

A new table is created as the table is so full. The new hash function is then h(X) =X mod 17

6
23
24

0	13
1	
2	15
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	

ADVANTAGE:

→Programmer does not worry about the table size.

→Simple to implement.

→Can be used in other data structures as well.

5.13 EXTENDIBLE HASHING:

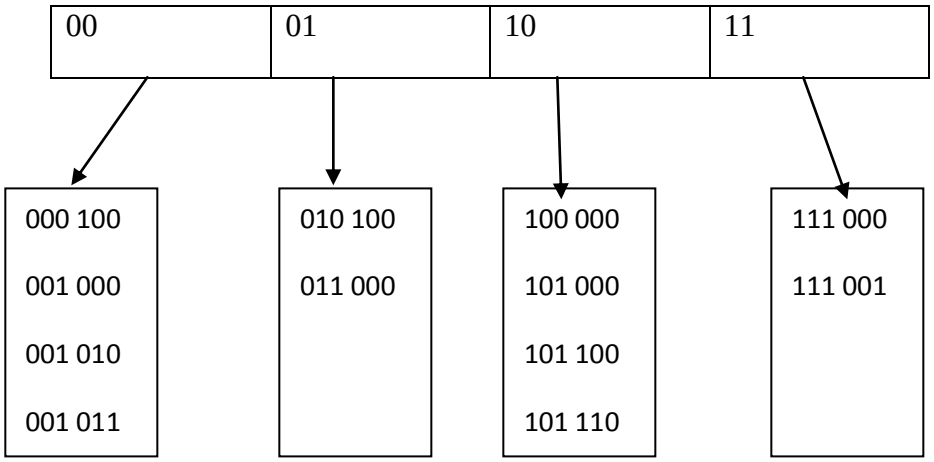
When open address hashing or separate chaining hashing in used ,collisions could causes several blocks to be examined during a find even for a well distributed hashtable .Furtermore,when the table gets too full, an extremely expensive rehashing steps must be performed,which require O(N) disk accesses.

These problem can be overcome by using extendible hashing.

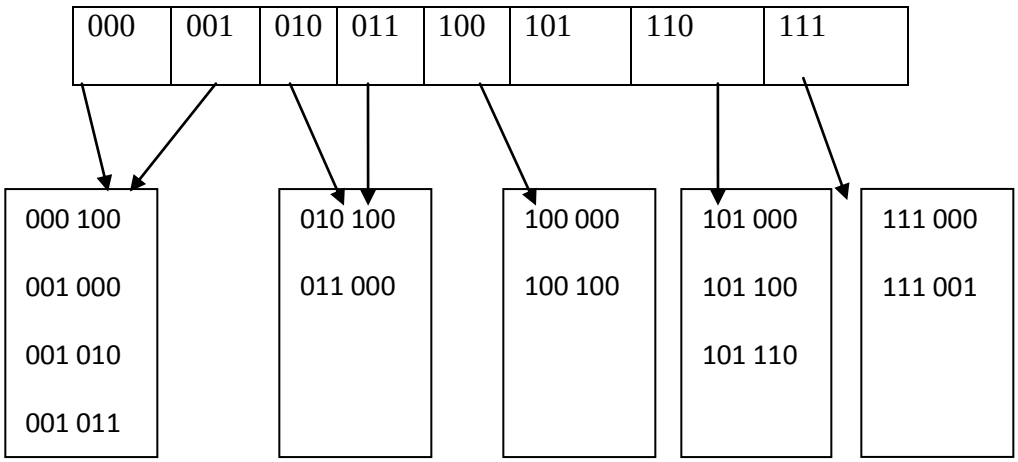
Extendible hashing, allows a find to be performed in two disk accesses come.Insertion also requires few disk accesses.

Let us suppose,consider our data consist of several six bit intergers.The root of the tree contains four pointers determied by the leading two bits of the data.Ecah leaf has upto M=4 element.In each leaf the first two bits are indentified, this is indicated by the number in parenthesis .

D will be represent the number of bits used by the root,also know as the directory.The number of entries in the directory 2^D.d_l is the number of leading bits that all the elements of some leaf L have in common .$d_L \leq$ D.The extendible hashing scheme for this data is given below.



Suppose that we want to insert the key 100100.This would go into the leaf,But as the third leaf is already full there is no room.We thus split this leaf into two leaves,which are now determined by the first three bit. Now the directory size is increased to 3.



Similarly,if the key 000000 is inserted ,then the first leaf split generating two leaves with d_L=3.The 000 and 001 pointers are updated

ADVANTAGE:

→Provides quick access time for insert and find operation are large data bases.

DISADVANTAGES:

→This algorithm does not work if there are more then M duplicates.

→If the elements in a leaf occurs in more then D+1 leading bits ,several directory split is possible the expected size of directory is O ($N^{l+l/M}$/M).

CS6202-PROGRAMMING AND DATASTRUCTURES I

IMPORTANT 2 MARKS

UNIT I- 2

1. Define global declaration?

The variables that are used in more than one function throughout the program are called global variables and declared in outside of all the function that is before the main() function.

2. Define data types?

Data type is the type of data, that are going to access within the program. „C" supports different data types

Primary	**Userdefined**	**Derived**	**Empty**
char		arrays	
int	typedef	pointers	void
float		structures	
double		union	

Example: int a,b; here int is data type

3.Define variable with example?

A variable is an identifier and it may take different values at different times of during the execution .

A variable name can be any combinations of 1 to 8 alphabets, digits or underscore.
Example:

int a,b;
here a,b are variables

4. What is decision making statement?

Decision making statement is used to break the normal flow of the program and execute part of the statement based on some condition.

5.What are the various decision making statements available in C ?

- If statement
- if…else statement
- nested if statement
- if…else ladder statement
- switch statement

6. **Distinguish between Call by value Call by reference.**

Call by value	Call by reference.
In call by value, the value of actual agreements is passed to the formal arguments and the operation is done on formal arguments.	In call by reference, the address of actual argurment values is passed to formal argument values.
Formal arguments values are photocopies of actual arguments values.	Formal arguments values are pointers to the actual argument values.
Changes made in formal arguments valued do not affect the actual arguments values.	Since Address is passed, the changes made in the both arguments values are permanent

7. What is an array?

An array is a collection of data of same data type. the elements of the array are stored in continuous memory location and array elements are processing using its index.

Example: int a[10];
Here „a‟ is an array name.

8. What is two dimensional array?

Two dimensional is an array of one dimensional array. The elements in the array are referenced with help of its row and column index.

Example:

```
Int a[2][2];
```

9. Define is function?

Function are group of statements that can be perform a task. Function reduce the amount of coding and the function can be called from another program.

Example:

```
main()

{                       fun()
-------                 {
fun();                  -------
--------                }
}
```

10. What are the various looping statements available in 'C'?

- While statement
- Do….while statement

- For statement

11. What are the uses of Pointers?

Pointers are used to return more than one value to the function

- Pointers are more efficient in handling the data in arrays
- Pointers reduce the length and complexity of the program
- They increase the execution speed
- The pointers save data storage space in memory

12. What is a Pointer? How a variable is declared to the pointer? (MAY 2009)

- Pointer is a variable which holds the address of another variable.
- **Pointer Declaration:** datatype *variable-name;
- **Example:** int *x, c=5; x=&a;

13. What is the difference between if and while statement?

If	While
It is a conditional statement	It is a loop control statement
If the condition is true, it executes some statements.	Executes the statements within the while block if the condition is true.
If the condition is false then it stops the execution the statements	If the condition is false the control is transferred to the next statement of the loop.

.

14. Define pre-processor in C.

Preprocessor are used to link the library files in to source program, that are placed before the main() function and it have three preprocessor directives that are

- Macro inclusion
- Conditional inclusion
- File inclusion

15. Define recursive function?

A function is a set of instructions used to perform a specified task which repeatedly occurs in the main program. If a function calls itself again and again , hten that function is called recursive function.

16. What is the difference between while and do….while statement?

The while is an entry controlled statement. The statement inside the while may not be executed at all when the condition becomes false at the first attempt itself.

The do …while is an exit controlled statement. The statements in the block are executed at least once.

17. Define Operator with example?

An operator is a symbol that specifies an operation to be performed on operands . some operators require two operands called binary operators, while other acts upon only one operand

called unary operator.
Example: a+b here a,b are operands and + is operator

18. Define conditional operator or ternary operator?

Conditional operator itself checks the condition and execute the statement depending on the condition. (a>b)?a:b if a is greater than b means the a value will be return otherwise b value will be return.

Example: big=a>b?a:b;

19. Compare of switch() case and nested if statement

- The switch() can test only constant values.
- No two case statements have identical constants in the same switch.
- Character constants are automatically converted to integers
- In switch() case statement, nested if can used
- The if can evaluate relational or logical expressions.
- Same conditions may be repeated for number of times
- Character constants are automatically converted to integers
- In nested if statements, switch() case can be used

20. What are steps involved in looping statements?

- Initialization of a condition variable.
- Test the control statement.
- Executing the body of the loop depending on the condition.
- Updating the condition variable.

21. Define break statement?

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

Example:

```
while(condition)
{
……….
If(condition)

break;
……..
}
```

22. Define null pointer?

A pointer is said to be a null pointer when its right value is 0, a null pointer can never point to a valid data. For checking a pointer, if it is assigned to 0, then it is a null pointer and is not valid

Example:

```
int  *a;
int  *b;
a=b=0
;
```

PART-B

1.What are functions? Explain the types of functions in detail with an example program for each type.
2. Explain in detail about conditional statements with example program?
3. What are pointers? When and why they are used? Explain in detail with sample programs.
4. Describe in detail about the Preprocessors in C.
5. Brief call by value and call by reference in detail.

6. Discuss about function prototypes in detail.
7.Explain in detail about Control statements with example program?
8. Define Array ?Explain types of arrays with example program?

UNIT-2

1. Compare arrays and structures.

Arrays	Structures
An array is a collection of data items of same data type.Arrays can only be declared.	A structure is a collection of data items of different data types. Structures can be declared and defined.
There is no keyword for arrays.	The keyword for structures is struct
An array name represents the address of the starting element.	A structrure name is known as tag. It is a Shorthand notation of the declaration.
An array cannot have bit fields.	A structure may contain bit fields.

2. Compare structures and unions.

Structure	Union
Every member has its own memory.	All members use the same memory.
The keyword used is struct.	The keyword used is union.
All members occupy separate memory location, hence different interpretations of the same memory location are not possible.	Different interpretations for the same memory location are possible. Conservation of memory is possible
Consumes more space compared to union.	

3. Define Structure in C.

A structure contains one or more data items of different data type in which the individual elements can differ in type.

A simple structure may contain the integer elements, float elements and character elements etc. and the individual elements are called members.

Example:

```
struct result
{
   int marks;
   float avg;
   char grade;
}std;
```

4. Rules for declaring a structure?

- A structure must end with a semicolon.
- Usually a structure appears at the top of the source program.
- Each structure element must be terminated.

- The structure must be accessed with structure variable with dot (.) operator.

5. Define structure pointers

Pointer is a variable, it contain address of another variable and the structure pointers are declared by placing * in front of a structure variable‟ s name.

Example:
```
struct result
{
   int marks;

   float avg;
   char grade;

};
struct result *sam;
```

6. Define union?

A union, is a collection of variables of different types, just like structure. Union is a derived data type and the difference between union and structure is in terms of storage.

In structure each member has its own storage location, whereas all the members of union use the same memory location.

Example:
```
union result
{
   int marks;
   float avg;

   char grade;
}std;
```

7. Define file?

A file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interrupted, for example, as characters, words, lines, paragraph and pages from a textual document.

Example:
```
FILE *infile;
FILE *outfile;
```

8. Define binary files?

Binary files can be processed sequentially or, depending on the needs of the application, they can process using random access techniques.

In C, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data.

9. Define opening a file?

A file requires to be opened first with the file pointer positioned on the first character. No input-output functions on a stream can be performed unless it is opened.

When a stream is opened, it is connected to named DOS device or file .C provides a various functions to open a file as a stream.

Syntax: FILE *fopen(char * filename, char *mode);

10. Define fseek()?

fseek() will position the file pointer to a particular byte within the file. The file pointer is a pointer is a parameter maintained by the operating system and determines where the next read will comes from , or to where the next write will go.

11. Functions of bit fileds?

- Bit fields do not have address.
- It is not an array.
- It cannot be accessed using pointer.
- It cannot be store values beyond their limlits. If larger values are assigned, the output is undefined.

12. What are the ways to detecting End of File?

In Text file:

- Special character EOF denotes the end of file.
- As soon as character is read, end of the file can be detected
- EOF is defined in stdio.h
- Equivalent value of EOF is -1

In binary file:

- feof function is used to detect the end of file
- it can be used in text file.

- feof returns TRUE if end of the file is reached

Syntax: **int feof(FILE *fp);**

13. What are key functions required to process a file?

- fopen
- fclose
- fseek
- fread
- fwrite

14. List out the file handling functions

fopen()-create a new file or open a existing file
fclose-close a file

getc()-reads a character from a file
putc()-writes a character to file
fscanf()-reads a set of data from a file

PART-B

1 Explain the concept of structure with an example program.
2 Discuss with an example structure within structure.
3 Explain (i)Structure and pointers
(ii)Passing Structure to functions
4 Explian array of structure ?Write a program to maintain N students mark statement?
5 Explain in detail about Union and pointers with an example program?.
6 Explain in details about Stream input and output functions?
7 Explain File handling concepts in c?
8 Write a program to open a file and write some text and close it?

1. What is meant by an abstract data type?

An ADT is an object with a generic description independent of implementation details. This description includes a specification of an components from which the object is made and also behavioral details of objects.

2. Advantages and Disadvantages of arrays?
Advantages:

- Data accessing is faster
- Array‟ s are simple-in terms of understanding point and in terms of programming.

Disadvantages:

- Array size is fixed
- Array elements stored continuously
- Insertion and deletion of elements in an array is difficult.

3. What is an array?

Array may be defined abstractly as a finite ordered set of homogenous elements.
Finite means there is a specific number of elements in the array.

4. What is a linked list?

Linked list is a kind of series of data structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a record containing its successor.

5. What is singly linked list?

A singly linked list is a linked list, there exists only one link field in each and every node and all nodes are linked together in some sequential manner and this type of linked list is called singly linked list.

6. What is a doubly linked list?

In a simple linked list, there will be one pointer named as „NEXT POINTER‟ to point the next element, where as in a doubly linked list, there will be two pointers one to point the next element and the other to point the previous element location.

7. Define double circularly linked list?

In a doubly linked list, if the last node or pointer of the list, point to the first element of the list, then it is a circularly linked list.

8. What is the need for the header?

Header of the linked list is the first element in the list and it stores the number of elements in the list. It points to the first data element of the list.

9. Define Polynomial ADT

- A *polynomial* object is a homogeneous ordered list of pairs <*exponent*, *coefficient*>, where each coefficient is unique.
- Operations include returning the degree, extracting the coefficient for a given exponent, addition, multiplication, evaluation for a given input. $10x^4+5x^2+1$

10. How to search an element in list.

Searching can be initiated from first node and it is compared with given element one after the other until the specified key is found or until the end of the list is encountered.

11. Define Dqueue?

Dqueue is also data structure where elements can be inserted from both ends and deleted from both ends. To implement a dqueue operations using singly linked list operations performed insert_front, delete_front, insert_rear, delete_rear and display functions.

12. How to implement stack using singly linked list

Stack is an Last In First Out (LIFO) data structure. Here , elements are inserted from one end called push operation and the same elements are deleted from the same end called pop operation

So, using singly linked list stack operations are performed in the front or other way ew can perform rear end also.

13. What are the types of Linear linked list?

- Singly linked lists
- Circular singly linked lists
- Doubly linked lists
- Circular doubly linked lists

14. What are advantages of Linked lists?

- Linked lists are dynamic data structures

- The size is not fixed
- Data can store non-continuous memory blocks
- Insertion and deletion of nodes are easier and efficient
- Complex applications

PART-B

1.Write a C program to implement Singly Linked list?
2. Explain Circular Linked list?How do you perform insertion and deletion operation in such a list?
3.Explain in detail about Doubly Linked list in detail?
4. Write a C program to implement polynomial Representation?
7.Explain Binary and Linear Search with C program?

1. Write down the algorithm for solving Towers of Hanoi problem?

- Push parameters and return address on stack.
- If the stopping value has been reached then pop the stack to return to previous level else move all except the final disc from starting to intermediate needle.
- Move final discs from start to destination needle.
- Move remaining discs from intermediate to destination needle.
- Return to previous level by popping stack.

2. What is a Stack ?

A stack is a non-primitive linear data structure and is an ordered collection of homogeneous data elements.The other name of stack is Last-in -First-out list.

One of the most useful concepts and frequently used data structure of variable size for problem solving is the stack.

3. What are the two operations of Stack?

- PUSH
- POP

4. What is a Queue ?

A Queue is an ordered collection of items from which items may be deleted at one end called the front of the queue and into which tems may be inserted at the other end called rear of the queue.Queue is called as First –in-First-Out(FIFO).

5. What is a Priority Queue?

Priority queue is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations. Ascending and Descending priority queue are the two types of Priority queue.

6. What are the different ways to implement list?

- Simple array implementation of list
- Linked list implementation of list
- cursor implementation of list

7. What are the postfix and prefix forms of the expression?

A+B*(C-D)/(P-R)

- Postfix form: ABCD-*PR-/+
- Prefix form: +A/*B-CD-PR

8. Explain the usage of stack in recursive algorithm implementation?

In recursive algorithms, stack data structures is used to store the return address when a recursive call is encountered and also to store the values of all the parameters essential to the current state of the procedure.

9. Write down the operations that can be done with queue data structure?

Queue is a first - in -first out list. The operations that can be done with queue are insert and remove.

10. What is a circular queue?

The queue, which wraps around upon reaching the end of the array is called as circular queue.

11. Give few examples for data structures?

- Stacks
- Queue
- Linked list
- Trees
- Graphs

12. List out Applications of queue

- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
- Computer systems must often provide a "holding area" for messages between two processes, two programs, or even two systems. This holding area is usually called a

  "buffer" and is often implemented as a queue.

13. How do you test for an empty queue?

To test for an empty queue, we have to check whether READ=HEAD where REAR is a pointer pointing to the last node in a queue and HEAD is a pointer that pointer to the dummy header.

In the case of array implementation of queue, the condition to be checked for an empty queue is READ<FRONT.

14. What are applications of stack?

- Conversion of expression
- Evaluation of expression
- Parentheses matching
- Recursion

15. Define recursion?

It is a technique and it can be defined as any function that calls itself is called recursion. There are some applications which are suitable for only recursion such as, tower of Hanoi, binary tree traversals etc, can be implementing very easily and efficiently.

16. What are types of Queues?

- Simple queue (ordinary queue)
- Circular queue
- Double ended queue
- Priority queue

PART B

1.Write a C program to implement Stack operations?
2. Write a C program to implement Queue operations?
3.Explain in detail about circular Queue?
4.Define DQUEUE ?Explain the operations in DQUEUE?
5.Write a C program to convert an infix expression to postfix expression?

UNIT-5

1. What is meant by Sorting and searching?

Sorting and searching are fundamentals operations in computer science. Sorting refers to the operation of arranging data in some given order

Searching refers to the operation of searching the particular record from the existing information

2. What are the types of sorting available in C?

- Insertion sort
- Merge Sort
- Quick Sort
- Radix Sort
- Heap Sort
- Selection sort
- Bubble sort

3. Define Bubble sort.

Bubble sort is the one of the easiest sorting method. In this method each data item is compared with its neighbor and if it is an descending sorting , then the bigger number is moved to the top of all

The smaller numbers are slowly moved to the bottom position, hence it is also called as the exchange sort.

4. Mention the various types of searching techniques in C

- Linear search
- Binary search

5. What is linear search?

In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned. The search in Linear Search starts at the beginning of an array and move to the end, testing for a match at each item.

6. What is binary search?

Binary search is simpler and faster than linear search. Binary search the array to be searched is divided into two parts, one of which is ignored as it will not contain the required element

One essential condition for the binary search is that the array which is to be searched, should be arranged in order.

7. Define merge sort?

Merge sort is based on divide and conquer method. It takes the list to be stored and divide it in half to create two unsorted lists.

The two unsorted lists are then sorted and merge to get a sorted list.

8. Define insertion sort?

Successive element in the array to be sorted and inserted into its proper place with respect to the other already sorted element. We start with second element and put it in its correct place, so that the first and second elements of the array are in order.

9. Define selection sort?

It basically determines the minimum or maximum of the lists and swaps it with the element at the index where its supposed to be.

The process is repeated such that the n^{th} minimum or maximum element is swapped with the element at the $n\text{-}1^{th}$ index of the list.

10. What is the basic idea of shell sort?

Shell sort works by comparing elements that are distant rather than adjacent elements in an array or list where adjacent elements are compared.

Shell sort uses an increment sequence. The increment size is reduced after each pass until increment size is 1.

11. What is the purpose of quick sort and advantage?

- The purpose of the quick sort is to move a data item in the correct direction, just enough for to reach its final place in the array.
- Quick sort reduces unnecessary swaps and moves an item to a greater distance, in one

move.

12. Define quick sort?

The quicksort algorithm is fastest when the median of the array is chosen as the pivot value. That is because the resulting partitions are of very similar size.

Each partition splits itself in two and thus the base case is reached very quickly and it follow the divide and conquer strategy.

13. Advantage of quick sort?

Quick sort reduces unnecessary swaps and moves an item to a greater distance, in one move.

14. Define radix sort?

Radix sort the elements by processing its individual digits. Radix sort processing the digits either by least significant digit(LSD) method or by most significant digit(MSD) method.

Radix sort is a clever and intuitive little sorting algorithm, radix sort puts the elements in order by comparing the digits of the numbers.

15. List out the different types of hashing functions?

The different types of hashing functions are,

- The division method
- The mind square method
- The folding method
- Multiplicative hashing
- Digit analysis

16. Define hashing?

- Search from that position for an empty location
- Use a second hash function.
- Use that array location as the header of a linked list of values that hash to this location

17. Define hash table?

All the large collection of data are stored in a hash table. The size of the hash table is usually fixed and it is bigger than the number of elements we are going to store.

The load factor defines the ration of the number of data to be stored to the size of the hash table

18. What are the types of hashing?

- Static hashing-In static hashing the process is carried out without the usage of an index structure.
- Dynamic hashing- It allows dynamic allocation of buckets, i.e. according to the demand of database the buckets can be allocated making this approach more efficient.

19. Define Rehashing?

Rehashing is technique also called as double hashing used in hash tables to resolve hash collisions, cases when two different values to be searched for produce the same hash key.

It is a popular collision-resolution technique in open-addressed hash tables.

PART B

1.Explain Hashing in detail?
2.Explain Quick Sort with its C program?
3.Explain about collision handling techniques in detail.
4.Write a C program for linear search and binary search.