INDEX

Name : Sagar. S. Talegaonkar

Std : Engg. III-Sem Div : B Roll No :

Sub : Data Structures

~~School~~ / College : GIT

Sr. No.	Date	Title	Remark	Pg. No.

Unit 1

☆ Structures:-

Structure is a collection of one or more variables of same or dissimilar data types, grouped under single name.

☆ Declaring structures:-

A structure can be declared in 3 ways:

i) Tagged structure.
ii) Using typedef keyword.
iii) Structure without tag.

i) Tagged structure – Declaring a tagged structure consists of 2 parts:
a) Structure definition
b) Defining / Declaring structure variables.

Syntax of structure definition:

```
struct tagname
{
    data type 1   variable 1;
    data type 2   variable 2;
        .
        .
        .
    data type n   variable n;
};
```

Ex:

```
struct student
{
    char name[20], usn[20];
    int marks;
};
```

Syntax of declaring structure variables -
struct tag_name var1, var2, ..., varn;
Ex struct student cse;

ii) Declaring structure using typedef - It consists of two parts:
a) Structure definition.
b) Defining / declaring structure variables

Syntax of structure definition:-

```
struct typedef
{
    data type 1  variable 1;
    data type 2  variable 2;
        :
    data type n  variable n;
} tag_name;
```

Syntax of declaring structure variables:-
tag_name var1, var2, var3, ..., varn;

☆ Structure initialization:-

There are 2 types of structure initializations:
i) Compile time initialization.
ii) Run time initialization.

Syntax to initialize structure (Compile time):-
struct tag_name variablename = { value 1, value 2, ..., value n };

Ex:
struct student cse={"NAME", "2GI18CS001", 98};

Syntax to initialize structure (run time).

Ex: Struct student ss;
scanf("%s %c %d", ss.name, ss.sex, ss.marks);

* Write a C program to store following information & print the same using structures:
  Roll no, name, marks & grade.

```
#include <stdio.h>

struct student
{
    int rollno;
    char name[20];
    int marks;
    char grade[3];
} s1;

main()
{
    printf("Enter the student roll number:");
    scanf("%d", &s1.rollno);
    printf("\n\nEnter the student name:");
    scanf("%s", s1.name);
    printf("\n\nEnter the marks scored:");
    scanf("%d", &s1.marks);
    printf("\n\nEnter the grade:");
    scanf("%s", s1.grade);

    printf("\n\n\n Entered student information:\n");
    printf("Student roll no: %d", s1.rollno);
    printf("\nStudent name: %s", s1.name);
    printf("\n Marks scored: %d", s1.marks);
    printf("\n Grade: %s", s1.grade);
}
```

☆ Accessing structures:-

Syntax:
variable_name . structure_member

☆ Array of structures:-

✳ Write a C-program to read & display following details of n students using structure:
roll no, name, usn, marks, grade.

→
```
#include <stdio.h>

struct student
{
    int rollno;
    char name[20];
    char usn[11];
    int marks;
    char grade[3];
};

main()
{
    struct student s[80];
    int n, i;

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

    for(i=0; i<n; i++)
    {
        printf("\n\nEnter student %d information:\n", (i+1));
        scanf("%d %s %s %d %s", &s[i].rollno, s[i].name,
              s[i].usn, &s[i].marks, s[i].grade);
    }
```

```
        printf("\n\n Entered student information:");
        for(i=0;i<n;i++)
        {
            printf("\n Student %d:\n", (i+1));
            printf(" Roll no: %d", s[i].rollno);
            printf("\n Name: %s", s[i].name);
            printf("\n Usn: %s", s[i].usn);
            printf("\n Marks: %d", s[i].marks);
            printf("\n Grade: %s", s[i].grade);
        }
}
```

* Write a C-program to maintain a record of n employee details: employee ID, name, salary.
Then print details of employees whose salary is above 5000.

→
```
#include <stdio.h>

struct employee
{
    char id[15];
    char name[20];
    float sal;
} emp[50];

main()
{
    int i,n;

    printf("Enter number of employees:");
    scanf("%d",&n);
    printf("\n Enter the details of %d employees\n", n);
    for(i=0;i<n;i++)
    {
        scanf("%s %s %f", emp[i].id, emp[i].name, &emp[i].sal);
    }
```

```
printf("\n\n The details of employees whose salary is above");
for (i=0; i<n; i++)
{
    if (emp[i].sal > 5000)
    {
        printf("\n Employee id: %s", emp[i].id);
        printf("\n Employee name: %s", emp[i].name);
        printf("\n Employee salary: %f", emp[i].sal);
    }
}
}
```

☆ Union:-

Union is a collection of one or more variables of same/dissimilar data types.

Syntax (declaration of union):

```
union tag_name
{
    data_type 1   var 1;
    data_type 2   var 2;
        :
    data_type n   var n;
};
```

✳ Diff. betwn. structure & union:-

Structure	Union
i) For a structure variable, the compiler allocates memory seperately for each member. Size of a structure is greater than or equal to the sum of sizes of its members.	For a union variable, the compiler allocates memory by considering the size of largest member. Size of union is equal to the size of the largest member

iii) Each member within a structure is assigned with unique storage area	Memory allocated is shared by individual members of union
iii) Address is different for each structure member.	Address is same for all the union members
iv) Altering the value of a member will not alter value of other member	Altering value of a member will alter value of other member
v) Individual members can be accessed at a time.	Only one member can be accessed at a time

☆ Dynamic Memory Management In C :-

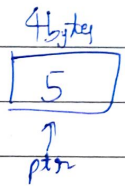
1) malloc() : The malloc fn. is used to allocate exact amount of memory required during execution.

Syntax:

```
ptr = (data type *) malloc (size);   // ptr is pointer variable
```

Ex:

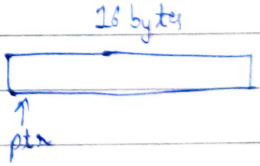
```
int *ptr;
ptr = (int *) malloc (4);
*ptr = 5;
```



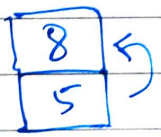
* void pointer (generic pointer) – Malloc fn. returns void pointer upon allocation of memory, hence we use type casting.

Ex 2:

```
int *ptr;
ptr = (int *) malloc (4 * sizeof(int));
ptr[0] = 5;
```

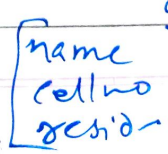



```
int a[5];
a[0]=10;
int *p;
p = &ptr;
```



ele
top

address of perso



* Write a C program to find sum of even & odd numbers in n elements using malloc fn

→
```
#include <stdio.h>
#include <stdlib.h>      // holds malloc fn
void main()
{
    int *ptr, n, sum_even = 0, sum_odd = 0, i;

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

    ptr = (int *) malloc(n * sizeof(int));

    printf("\n Enter array elements: \n");
    for(i = 0; i < n; i++)
    {
        scanf("%d", &ptr[i]);
    }

    for(i = 0; i < n; i++)
    {
        if( ptr[i] % 2 == 0)
            sum_even += *ptr[i];
        else
            sum_odd += *ptr[i];
    }

    printf("\n Sum of even elements = %d", sum_even);
    printf("\n Sum of odd elements = %d", sum_odd);
}
```

16 2 19

ii) calloc(): This fn is used to allocate memory dynamically & used for allocating multiple blocks of memory. This fn is mostly used for arrays.

Syntax:

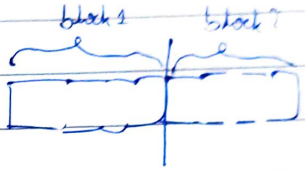
ptr = (data type *) calloc (n, size);

n → No. of blocks to be allocated

size → size of each block

Ex:

ptr = (int *) calloc (2, 4);



* C-program to read & display marks of 3 subjects using calloc fn.

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

void main()
{
    int *ptr, i;
    ptr = (int *) calloc (3, 4);
    printf ("\n Enter the marks of 3 subjects: ");
    for (i=0; i<3; i++)
        scanf ("%d", &ptr[i]);

    printf("\n The marks of 3 subjects:\n");
    for (i=0; i<3; i++)
        printf ("%d ", *ptr[i]);
}
```

iii) realloc(): Used to extend size of already allocated memory or to add memory at the end of the block. Before using this fn, the memory must be allocated using malloc or calloc.

Syntax:
ptr = (data_type *) realloc (ptr, size);

iv) free(). Used to deallocate the allocated blocks of memory (memory allocated malloc, calloc or realloc).

Syntax:
free (pointer);

* Diff Between malloc & calloc:-

	Malloc	Calloc
i)	Doesn't initialize allocated memory.	Initializes allocated memory to zero.
ii)	malloc takes only one argument, i.e, size.	calloc takes two arguments, i.e, no. of blocks & size of each block
iii)	Has better time efficiency than calloc.	Time efficiency is lesser than malloc.
iv)	allocates memory even if memory is not available contiguously.	allocates required blocks of memory contiguously.

21 8 19

Unit - II

Files:

File is a collection of data which is usually stored on secondary storage device

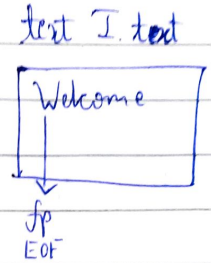
Types of files (few examples): text file, binary files, executable file

A text file is one in which data is stored as sequence of characters that can be processed sequentially.

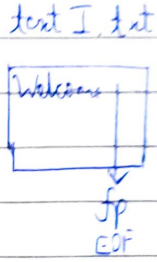
Modes of file: There are 3 types of file modes
read (r), write (w), append (a).

i) read mode (r)

ii) write mode (w)



iii) append mode (a)



Steps to use files:

i) Declare a file pointer.
ii) Open the file.
iii) Read/write data from/to the file
iv) Close file.

i) Syntax:
FILE *fileptrname;
Ex: FILE *fp; // file pointer can be used only with files

ii) Syntax:
FILE *fp;
fp = fopen("file_name", "mode"); // file name along with file extensi

To open a file, file must exist otherwise we get error. (r read mode)

ii) Syntax
fclose(fp);

☆ I/O operations in files:-

list of fn's for I/O operations:

i) fscanf()
ii) fprintf()
iii) fgets()
iv) fputs()
v) fgetc()
vi) fputc()
vii) getc()

Syntax:
ch = getc(fp);

i) fscanf():

Syntax:
fscanf(fp, "format string", list); // returns no. of values read & returns 0 if it reaches EOF

ii) fprintf():

Syntax:
fprintf(fp, "format string", list);

✱ Write a C program to create a file called text1.txt & write following information into text1.txt :
name, rollno & usn. Then read from text1.txt & display info on screen.

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

void main()
{
    FILE *fp;
    int rollno;
    char name[20], usn[11];

    printf("Enter the name, usn, rollno: ");
    scanf("%s %s %d", name, usn, &rollno);
```

```
fp = fopen("text1.txt", "w");
fprintf(fp, "%s %s %d", name, usn, rollno);

fclose(fp);

fp = fopen("text1.txt", "r");
fscanf(fp, "%s %s %d", name, usn, &rollno);
fclose(fp);
printf("\n\n Name %s\n Usn %s\n Rollno %d", name, usn, rollno);
fclose(fp);
}
```

iii) fgets(): Used to read string (or sequence of characters) from a file pointed by file pointer

Syntax:

```
fgets(str, n, fp);
```

↳ string variable, n - No. of characters to be read

Ex:

```
fgets(name, 5, fp);
```

text1.txt

Welcome

iv) fputs: Used to write a string or sequence of characters to a file pointed by file pointer

Syntax:

```
fputs(str, fp);   // str can be variable or string constant also
```

v) fgetc: Used to read single character from file pointed by file pointer

Syntax:

```
fgetc(ch, fp);   // ch is a variable of data type character (char)
```

after reading a character, fp is automatically incremented to next character in file.

vi) fputc: Used to write single character into a file pointed by file pointer.

Syntax:

```
fputc(ch, fp);
```

* Write a C program to read contents of a file called abc.txt & count number of characters, number of lines & number of tabs.

→

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

void main()
{
    FILE *fp;
    int number_of_char=0, number_of_lines=0, number_of_tabs=0;
    char ch;
    fp=fopen("abc.txt","r");
    if(fp==NULL)
    {
        printf("\nFile does not exist");
        exit 0;
    }

    while((ch=getc(fp))!=EOF)
    {
        number_of_char++;
        if(ch=='\n')
            number_of_lines++;
        if(ch=='\t')
            number_of_tabs++;
    }
    fclose(fp);

    printf("\nNumber of characters=%d", number_of_char);
    printf("\nNumber of lines=%d", number_of_lines);
    printf("\nNumber of tabs=%d", number_of_tabs);
}
```

15
18 8 19

☆ Pointers To Structures:-

1) Write a C-program to store details of student, i.e., name, usn, rollno & marks, using pointer to structure.

→
```
#include <stdio.h>

struct student
{
    int rollno, marks;
    char name[30], usn[11];
} s;

void main()
{
    struct student s, *ps;
    ps = &s;
    printf("Enter student name, usn, rollno, marks:\n");
    scanf("%s %s %d %d", ps->name, ps->usn, &ps->rollno,
                                            &ps->marks);

    printf("\n\n Student details are:\n");
    printf("%s %s %d %d", ps->name, ps->usn, ps->rollno,
                                            ps->marks);
}
```

☆ Pointers to array of structures:-

1) Write a C program to store details of n students & print the same using pointer (to array of structure concept).

→
```
#include <stdio.h>
struct student
{
    int rollno, marks;
    char name[30], usn[20];
} s[50];
```

```
void main()
{
    struct student *ptr;
    int i, n;
    ptr = &s[0];

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

    printf("Enter name, usn, rollno, marks of %d students:\n", n);
    for(i=0; i<n; i++)
    {
        scanf("%s %s %d %d", (ptr+i)->name, (ptr+i)->usn,
                &(ptr+i)->rollno, &(ptr+i)->marks);
    }

    printf("\n\n Details of %d students are:", n);
    for(i=0; i<n; i++)
        printf("\n %s %s %d %d", (ptr+i)->name, (ptr+i)->usn,
                (ptr+i)->rollno, (ptr+i)->marks);
}
```

☆ Allocating memory dynamically to structure:-

* Write a program to store details of a student such as name, usn & marks using structure. Allocate memory dynamically to structure.

→
```
#include <stdio.h>

typedef struct
{
    char name[20], usn[11];
    int marks;
} student;
```

```
void main()
{
    int n;
    student *ptr;
    ptr = (student *) malloc (sizeof (student));
    printf ("Enter student's name, usn & marks: \n");
    scanf ("%s %s %d", ptr->name, ptr->usn, &ptr->marks);
}
```

* Write a C-program to store details of n students using structure & allocate memory dynamically to structure.

→
```
#include <stdio.h>

typedef struct
{
    char name[20], usn[11];
    int marks;
} student;

void main()
{
    student *ptr;
    int n, i;

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

    ptr = (student *) malloc (n * sizeof (student));

    printf("Enter the details of %d students \n", n);
    for(i=0; i<n; i++)
        scanf("%s %s %d", (ptr+i)->name, (ptr+i)->usn, &(ptr+i)->marks);
}
```

* Program to read n student details and display highest marks

```
#include <stdio.h>

typedef struct
{
    char name[20], usn[11];
    int marks;
} student;

void main()
{
    student *ptr;
    int n, hrmarks, i, index=0;

    printf("Enter n:");
    scanf("%d", &n); ptr = (student *) malloc (n * sizeof(student));
    printf("Enter details:");
    for(i=0; i<n; i++)
    {
        scanf("%s %s %d", (ptr+i)->name, (ptr+i)->usn, &(ptr+i)->marks);
    }

    hrmarks = ptr->marks;

    for(i=1; i<n; i++)
        if((ptr+i)->marks > hrmarks)
        { hrmarks = (ptr+i)->marks;
          index = i; }
    printf("\nHighest marks = %d", hrmarks);
    printf("\nDetails of student scoring highest marks:");
    printf("\nName: %s", (ptr+index)->name);
    printf("\nUsn: %s", (ptr+index)->usn);
    printf("\nMarks: %d", (ptr+index)->marks);
}
```

* Program to read & print student records Name, Usn, Semester, Marks Of 3 subjects to be read from user Code. Average & print the result with appropriate headings Take in a number of students. Use appropriate data structure.

27 & 28 / 2019

Unit - III

* Data Structure is a method of storing data in memory so that it can be used efficiently.

☆ Stack:-

It is a data structure where elements can be inserted from one end & deleted from the same end. It works according to Last In First Out (LIFO) principle.

☆ Stack ADT :-

Objects: A finite ordered list with zero or more elements.

~~Functions:~~

for all Stack ∈ Stack, item ∈ element, maxsize ∈ positive integer

Functions: i) Stack CreateS(maxsize) ::= Create an empty stack whose maximum size is maxsize.

ii) Boolean isFull (Stack, maxsize) ::= if (Number of elements in Stack == maxsize)
return TRUE
else
return FALSE

iii) Stack push (Stack, item) ::= if (isFull (Stack)) stack full
else insert item into top of stack & return

iv) Boolean isEmpty (Stack) ::= if (Stack == CreateS (maxsize))
return TRUE
else
return FALSE

v) Element pop (Stack) ::= if (isEmpty (Stack)) return
else remove & return element on the top of stack.

☆ Applications Of Stack:-

i) Conversion of expressions.
ii) Evaluation of expressions

i) Conversion of expressions:-

Precedence & Associativity Of Operators:-

Operator	Priority	Associativity
^, $	6	R → L
*	4	L → R
/	4	L → R
%	4	L → R
+	2	L → R
-	2	L → R

Precedence — The order in which different operators are evaluated

* Converting exp. from infix to postfix:-

Q. Convert following exp. from infix to postfix:-

a) A+B*C
→ A+ BC*
ABC*+

b) A*B+C*D
→ AB* + C*D
AB* + CD*
AB*CD*+

c) A+B+C+D
→ AB+ + (+D
~~AB++CD+~~ AB+C+ +D
~~AB+CD++~~ AB+C+D+

d) $(A+B)*(C+D)$

→ $AB+*(C+D)$

$AB+*CD+$

$AB+CD+*$

e) $(A+(B-C)*D)$

→ $(A+BC-*D)$

$(A+BC-D*)$

$ABC-D*+$

f) $((A+(B-C)*D)\hat{}E+F)$

→ $((A+BC-*D)\hat{}E+F)$

$((A+BC-D*)\hat{}E+F)$

$((ABC-+D*)\hat{}E+F)$

$(ABC-+D*E\hat{}+F)$

$ABC-+D*E\hat{}F+$

g) $X\hat{}Y\hat{}Z-M+N+P/Q$

→ $X\hat{}YZ\hat{}-M+N+P/Q$

$XYZ\hat{}\hat{}-M+N+P/Q$

$XYZ\hat{}\hat{}-M+N+PQ/$

$XYZ\hat{}\hat{}M-+N+PQ/$

$XYZ\hat{}\hat{}M-N++PQ/$

$XYZ\hat{}\hat{}M-N+PQ/+$

h) $(a+b)*d+e/(f+a+d)+c$

i) $4+3-5*6/2$

j) $a+b-(c*(a+b)/d)$

k) $(A+B-(C*(A+B)/D))$

h) $(a+b)*d+e/(f+a+d)+c$

→ $ab+*d+e/(f+a+d)+c$

$ab+*d+e/(fa++d)+c$

$ab+*d+e/fa+d++c$

ab+dX+e/fo+d+c
ab+dX+efo+d+/+c
ab+dXefo+d+/+c
ab+dXefo+d+/+c

i) 9+3-5X6/2
→ = 9+3-56X/2
= 9+3-56X2/
= 93+-56X2/
= 93+56X2/-

j) a+b-(c X(a+b)/d)
→ = a+b-(c X ab+/d)
= a+b-(cab+X/d)
= a+b-cab+Xd/
= ab+-cab+Xd/
= ab+cab+Xd/-

k) (A+B-(C X(A+B)/D))
→ = (A+B-(C X AB+/D))
= (A+B-(CAB+X/D))
= (A+B-CAB+XD/)
= AB+-CAB+XD/
= AB+CAB+XD/-

P.T.O.

☆ Convert following infix exp to postfix exp using tabular method:

i) $x+y$

→

Input string	Stock	Output string
x	empty	x
$+$	$+$	x
y	$+$	xy
		$xy+$

ii) $x+(y*z)$

→

Input string	Stock	Output string
x	empty	x
$+$	$+$	x
$($	$+,($	x
y	$+,($	xy
$*$	$+,(,*$	xy
z	$+,(,*$	xyz
$)$	$+$	$xyz*$
		$xyz*+$

iii) $x*y+z$

→

Input string	Stock	Output string
x	empty	x
$*$	$*$	x
y	$*$	xy
$+$	$+$	$xy*$
z	$+$	$xy*z$
		$xy*z+$

→

iv) (A+B) * (C+D)

→

I/P string	Stack empty	O/P string
(	(	
A	(	A
+	(, +	A
B	(, +	AB
)		AB+
*	*	AB+
(	*, (	AB+
C	*, (	AB+C
+	*, (, +	AB+C
D	*, (, +	AB+CD
)	*	AB+CD+

AB+CD+*

v) A*B+C*D

→

I/P String	Stack empty	O/P string
A	empty	A
*	*	A
B	*	AB
+	+	AB*
C	+	AB*C
*	+, *	AB*C
D	+, *	AB*CD

AB*CD*+

★ Points to be noted while converting expression from infix to postfix:-

1) Define priorities of operators on stack & operator on I/P string:
   i) If operators are L→R associative, then stack precedence is higher than I/P string.
   ii) If operators are R→L associative, then stack precedence is lower than I/P string.

2) Scan characters in an infix expression one at a time:
i) If character is opening parenthesis, then push it on the stack.
ii) If character is operand, just output it to postfix string.
iii) If character is operator, & stack is empty, then push the character/operator on stack.
iv) If the precedence of operator on the top of stack is greater than or equal to input character, then pop the operator from the top of stack & write it to the postfix string & push input character into the stack.
v) If precedence of operator on the top of stack is less than that of I/P character, then just push the character into the stack at top position.
vi) If I/P character is closing parenthesis, pop the operators in the stack till corresponding opening " is reached.
vii) Pop all elements of stack into postfix string once you reach end of expression.

* Write a program to convert infix exp. to postfix.

→
```
int sp(char c)      // stack precedence (sp)
{
   switch(c)
   {
      case '#' : return -1;    // to indicate stack is empty.
      case '(' : return 1;
      case '+' :
      case '-' : return 3;
      case '*' :
      case '/' : return 5;
      case '^' : return 7;
   }
}
int ip(char c)
{
      switch(c)
```

```
{
        case '(' : return 8;
        case '+' :
        case '-' : return 2;
        case '*' :
        case '/' : return 4;
        case '^' : return 5;
    }
}

void convert(char infix[], char postfix[])
{
    int i=0, j=0;
    char symb;
    int top=0;
    char stk[30];
    stk[top] = '#';

    while(infix[i] != '\0')
    {
        symb = infix[i];
        i++;
        if((symb >= '0' && symb <= 9) || (symb >= 'a' && symb <= 'z'))
        {
            postfix[j] = symb;
            j++;
        }
        else
        {
            if(symb == ')')
            {
                while(stk[top] != '(')
                {
                    postfix[j++] = stk[top--];
                }
```

```
            top--;
        }
        else
        {
            while(sp(stk[top]) >= sp(symb))
                postfix[j++] = stk[top--];
            stk[++top] = symb;
        }
    }
}

while(stk[top] != '#')
    postfix[j++] = stk[top--];

postfix[j] = '\0';
}

main()
{
    char infix[30], postfix[30];
    printf("Enter an infix expression.\n");
    scanf("%s", infix);
    convert(infix, postfix);
    printf("%s", postfix);
}
```

☆ Evaluation Of Expressions:-

I) Evaluation of postfix string:-

Steps:

i) Scan the symbol from left to right.
ii) If the scanned symbol is an operand, then push it onto the stack.
iii) " " " " operator, pop two elements from stack

Store

iii) First popped element into operand 2 & second popped element in operand 1

iv) Perform indicated operation.

v) Push result onto stack.

vi) Repeat above procedure till end of input is encountered.

* Evaluate following postfix exp.'s, show evaluation steps using tabulation method & also write a final value of expression:-

1) 632-5*+1$7+

→

Postfix exp.	Scanned Symbol	OP2	OP1	Result OP1 op OP2	Stack Contents
632-5*+1$7+	6				6
32-5*+1$7+	3				6,3
2-5*+1$7+	2				6,3,2
-5*+1$7+	-	2	3	3-2=1	6,1
5*+1$7+	5				6,1,5
*+1$7+	*	5	1	1*5=5	6,5
+1$7+	+	5	6	6+5=11	11
1$7+	1				11,1
$7+	$	1	11	11$1=11	11
7+	7				11,7
+	+	7	11	11+7=18	18

∴ Result = 18

2) ABC+*CBA-+*

A=1, B=2, C=3

3) 12+3-21+3$-

4) 63/835+2**+

2) ABC+*CBA-+*

A = 1, B = 2, C = 3

→ Postfix exp as:

123+*321-+*

Postfix exp.	Scanned symbol	OP2	OP1	Result = OP1 op OP2	Stack contents
123+*321-+*	1				1
23+*321-+*	2				1,2
3+*321-+*	3				1,2,3
+*321-+*	+	3	2	2+3=5	1,5
321-+	*	5	1	1*5=5	5
321-+*	3				5,3
21-+*	2				5,3,2
1-+*	1				5,3,2,1
-+*	-	1	2	2-1=1	5,3,1
+*	+	1	3	3+1=4	5,4
*	*	4	5	5*4=20	20

∴ Result = 20

3) 12+3-21+3$-

→

Postfix exp.	Scanned symbol	OP2	OP1	Result = OP1 op OP2	Stack contents
12+3-21+3$-	1				1
2+3-21+3$-	2				1,2
+3-21+3$-	+	2	1	1+2=3	3
3-21+3$-	3				3,3
-21+3$-	-	3	3	3-3=0	0
21+3$-	2				0,2
1+3$-	1				0,2,1
+3$-	+	1	2	2+1=3	0,3
3$-	3				0,3,3
$-	$	3	3	3$3=27	0,27
-	-	27	0	0-27=-27	-27

Result = -27

4) 63/835+2**+

Postfix exp	Sym scanned	OP1	OP2	Result = OP1 op OP2	Stack contents
63/835+2**+	6				6
3/835+2**+	3				6,3
/835+2**+	/	3	6	6/3=2	2
835+2**+	8				2,8
35+2**+	3				2,8,3
5+2**+	5				2,8,3,5
+2**+	+	5	3	3+5=8	2,8,8
2**+	2				2,8,8,2
**+	*	2	8	8*2=16	2,8,16
*+	*	16	8	8*16=128	2,128
+	+	128	2	2+128=130	130

Result = 130

5) 53*5/4+25*2/-

6) 684*3+61-9+3-4+

Postfix exp	Sym scanned	OP1	OP2	Result = OP1 op OP2	Stack
53*5/4+25*2/-	5				5
3*5/4+25*2/-	3				5,3
*5/4+25*2/-	*	5	3	5*3=15	15
5/4+25*2/-	5				15,5
/4+25*2/-	/	15	5	15/5=3	3
4+25*2/-	4				3,4
+25*2/-	+	3	4	3+4=7	7
25*2/-	2				7,2
5*2/-	5				7,2,5
*2/-	*	2	5	2*5=10	7,10
2/-	2				7,10,2
/-	/	10	2	10/2=5	7,5
-	-	7	5	7-5=2	2

Result = 2

6)

Postfix expression	Scanned symb	OP2	OP1	Result = OP1 op OP2	Stack
684*3+61-9+3-4+	6				6
84*3+61-9+3-4+	8				6,8
4*3+61-9+3-4+	4				6,8,4
*3+61-9+3-4+	*	4	8	8*4=32	6,32
3+61-9+3-4+	3				6,32,3
+61-9+3-4+	+	3	32	32+3=35	6,35
61-9+3-4+	6				6,35,6
1-9+3-4+	1				6,35,6,1
-9+3-4+	-	1	6	6-1=5	6,35,5
9+3-4+	9				6,35,5,9
+3-4+	+	9	5	5+9=14	6,35,14
3-4+	3				6,35,14,3
-4+	-	3	14	14-3=11	6,35,11
4+	4				6,35,11,4
+	+				

* C-Program to evaluate given postfix expression:-

→
```
#include <stdio.h>
void main()
{
    char postfix[30];
    double res;
    printf("Enter a postfix exp.: ");
    scanf("%s", postfix);

    res = Postfix(postfix);

    printf("%.1f", res);
}
```

→

```
double PostFun(char exp[])
{
    double stk[];
    char symb;
    int t=-1;
    double op1,op2;
    int i=0;

    do
    {
        symb=exp[i];
        i++;
        if(symb>='0' && symb<='9')
            stk[++t]=symb-'0';
        else
        {
            op2=stk[t--];
            op1=stk[t--];
            switch(symb)
            {
                case '+': stk[++t]=op1+op2;
                          break;
                case '-': stk[++t]=op1-op2;
                          break;
                case '*': stk[++t]=op1*op2;
                          break;
                case '/': stk[++t]=op1/op2;
                          break;
                case '^': stk[++t]=pow(op1,op2);
                          break;
            }
        }
    }while(exp[i]!='\0');
    return stk[t];
}
```

☆ Queue:-

It is a data structure where elements are inserted from one end & deleted from the other end. The end at which new elements are inserted is called rear end & the end from which elements are deleted is called front end.

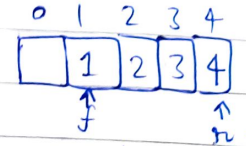
☆ Implementing Queue Using Array:-

```
#include <stdio.h>
#define maxsize 5

int f, r, q[maxsize];

void main()
{
    int choice;
    f=0;
    r=-1;

    for(;;)
    {
        printf("1) Insert \n2) Delete \n3) Display");
        printf("Enter your choice:");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
                insert();
                break;
            case 2:
                delete();
                break;
```

```
            case 3:
                display();
                break;
            }
        }
    }

void insert()
{
    int item;

    if( r == maxsize-1)
    {
        printf("Queue full!!!");
        exit(0);
    }
    printf("Enter the element:");
    scanf("%d", &item);
    q[++r] = item;
}

void delete()
{
    if (f > r)
        printf("Queue is empty");
    else
    {
        printf("Deleted element is: %d", q[f++]);
    }
    if (f > r)
    {
        f = 0;
        r = -1;
    }
}
```

```
void display()
{
    int i;
    if (f > r)
        printf("Queue is empty");
    else
    {
        printf("Contents of queue are \n");
        for (i = f; i <= r; i++)
            printf("%d ", q[i]);
    }
}
```

* Disadvantage of linear queue:-



In situation as shown above, where it doesn't allow us to insert the elements into the queue even if the queue is empty.

* ADT of Queue:-

ADT Queue is
Objects: A finite ordered list with zero or more elements.
Functions:
for all queue ∈ Queue, item ∈ element, maxQueueSize ∈ positive integer.

Queue CreateQ(maxQueueSize) ::= Create an empty where maximum size is maxQueueSize.

Boolean isFull(queue, maxQueueSize) ::= if (number of elements in queue == maxQueueSize) return TRUE
else return FALSE

Queue AddQ(queue, item) ::= if (isFull(queue)) QueueFull
else insert item at rear of queue & return queue

Boolean isEmpty(queue) ::= if (queue == CreateQ(maxQueueSize)) return TRUE
else return FALSE

element Deleted (queue) :: = if (isEmpty(queue))
return
else remove & return item at front of
queue

☆ Queue Applications:-

1) Messaging System:-

Write a C program to simulate working of Messaging System in which a message is placed in a queue by message sender & a message is removed from the queue by a message receiver, which can also display contents of the queue.

→
```
#include <stdio.h>
#define SIZE 10
#include <string.h>
struct queue
{
    int id;
    char msg[30];
}q[SIZE];
int f=0, r=-1;
void main()
{
    int choice;

    do
    {
        printf("Messaging System operations");
        printf("1) Place a message");
        printf("2) Remove a message");
        printf("3) Display");
        printf("4) Exit");
```

```
        printf("Enter your choice ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                insert();
                break;
            case 2:
                delete();
                break;
            case 3:
                display();
                break;
            case 4:
                break;
        }
    } while(choice!=4);
}

void insert()
{
    int x;
    char txt[30];
    if(r == SIZE-1)
    {
        printf("Queue is full");
        return;
    }

    printf("Enter the id & message:\n");
    scanf("%d",&x);
    gets(txt);
    r++;
```

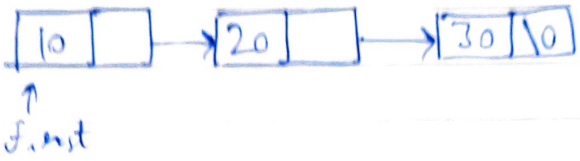
```
    q[r].id = x;
    strcpy(q[r].msg, text);
    printf("%d msg is inserted\n", x);
}

void delete()
{
    if (f > r)
    {
        printf("Queue is empty\n");
        f = 0;
        r = -1;
        return;
    }

    printf("%d msg is deleted\n", q[f++].id);
}

void display()
{
    int i;

    if (f > r)
    {
        printf("Queue is empty");
        return;
    }
    printf("Contents of queue are:\n");
    for(i=f; i<=r; i++)
        printf("%d\t%s\n", q[i].id, q[i].msg);

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

Linked List:-

It is a collection of zero or more nodes where each node is connected to one/more nodes. Each node has 2 fields, i.e., info field & link field.

Ex (Pictorial rep. of link list)

10, 20, 30



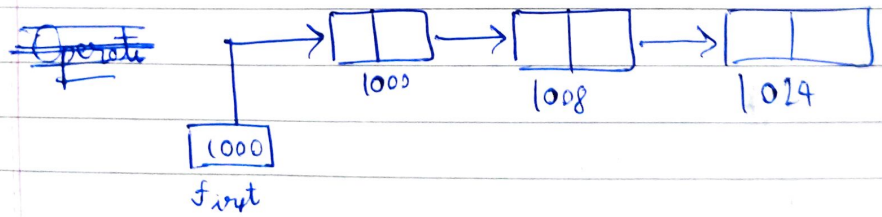
Advantages of linked lists over arrays:-

i) Size of array is fixed whereas linked lists are dynamic
ii) Insertion & deletion operations involving arrays are tedious jobs as its very easy to do so in linked lists.

Types of linked lists:-

1) Singly linked lists.
2) Doubly linked lists.
3) Circular singly linked lists.
4) Circular doubly linked lists.

1) Singly linked list - Each node of the list has only one address / link field.




```
struct node
{
    int data;
```

```
    struct node *link;
};

struct node *first;

first = (struct node *) malloc (sizeof (struct node));
```

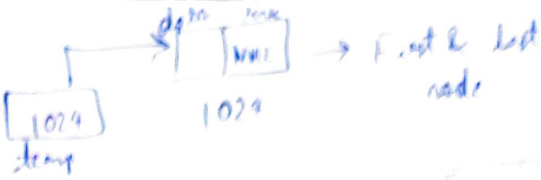
Note: Self referential structure : Contains a pointer of type structure within that structure.

* Operations on singly linked list :-

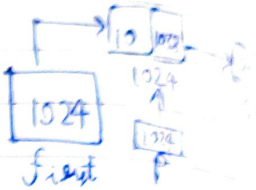
i) Inserting a node into a singly linked list.
ii) Deleting a node from the singly linked list
iii) Search a singly linked list
iv) Display contents of singly linked list

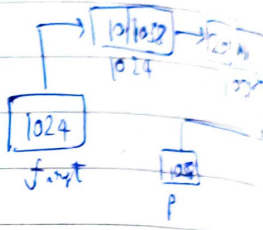
i) Inserting a node into a singly linked list -
a) Inserting a node at the end of the list.
b) " " " " " front " " "
c) " " " at a given position in the list.

a)
```
struct node
{
    int data;
    struct node *link;
};

struct node *first = NULL;

void addatend()
{
    struct node *temp;
    temp = (struct node *) malloc (sizeof (struct node));
```

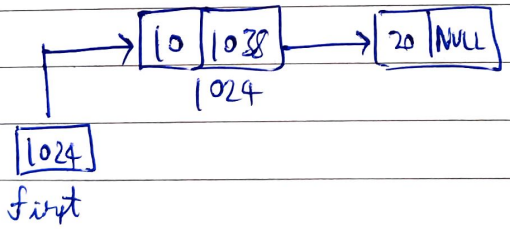



```
printf("Enter the element");
scanf("%d", &temp->data);
temp -> link = NULL;

if (first == NULL)
        first = temp;
else
{
    struct node *p;
    p = first;
    while (p -> link != NULL)
    {
        p = p -> link;
    }
    p -> link = temp;
}
}
```

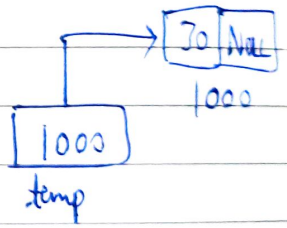




b) Inserting node at the front of the list.



existing




```
struct node
{
    int data;
    struct node *link;
};

struct node *first = NULL;
```

```
void insertAtEnd()
{
    struct node *temp;
    temp = (struct node *) malloc(sizeof(struct node));

    printf("Enter the data to be inserted : ");
    scanf("%d", &temp->data);

    temp->link = NULL;
    temp->link = first;
    first = temp;
}
```

* Counting the number of nodes in a singly linked list -

```
void length()
{
    struct node *temp;
    int count = 0;

    temp = first;          // moving first to temp

    while (temp != NULL)
    {
        count++;
        temp = temp->link;
    }
    printf("Length of the singly linked list = %d\n", count);
}
```

c) Inserting a node at given position. (after given position, rest of the nodes)

```
void insertatloc()
{
    struct node *temp, *p;
    int len, loc, i = 1;

    printf("Enter the location at which a node is to be inserted");
    scanf("%d", &loc);

    len = length();    // assuming length fn. returns an integer value

    if (loc > len)
        printf("Invalid location");
    else
    {
        p = first;
        while (i < loc)
        {
            p = p -> link;
            i++;
        }
        temp = (struct node *) malloc (sizeof (struct node));
        printf("Enter the element to be stored at inserted node");
        scanf("%d", &temp -> data);
        temp -> link = NULL;

        temp -> link = p -> link;
        p -> link = temp;
    }
}
```

24 9 19

iv) Displaying elements of the singly linked list

→

```
void display()
{
    struct node *temp;

    if(first==NULL)
        printf("List is empty");
    else
    {
        temp=first;
        while(temp [redacted] !=NULL)
        {
            printf("%d", temp->data);
            temp = temp->link;
        }
    }
}
```

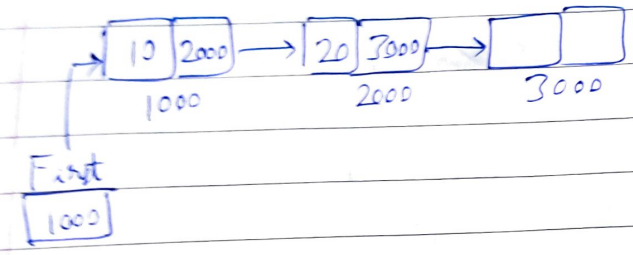
iii) Searching the list for an element.

→

```
void search()
{
    Struct node *temp;
    int ele, flag = 0;

    temp = first;
    if(temp==NULL)
        printf("List empty");
    else
    {   printf("Enter element to search:"); scanf("%d", &ele);
        while(temp!=NULL)
        {
            if(temp->data == ele)
            {
                flag=1;
                break;
```

```
        temp = temp -> link;
    }
    if (flag == 1)
        printf("Element found");
    else
        printf("Element not found");
}
```

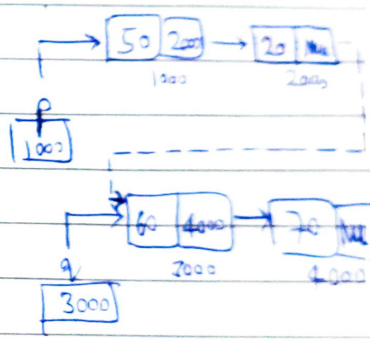
ii) Deleting element from singly linked list :-




```
void delete()
{
    struct node *temp, *p, *q;
    int loc, len, i = 1;

    printf("Enter location: ");
    scanf("%d", &loc);
    len = length();
    if (loc > len)
        printf("Invalid location");
    else if (loc == 1)
    {
        temp = first;
        first = temp -> link;
        temp -> link = NULL;
        free (temp);
    }
```

```
else
{
    p = first;
    while (i < loc - 1)
    {
        p = p -> link;
        i++;
    }
    q = p -> link;
    p -> link = q -> link;
    q -> link = NULL;
    free(q);
}
}
```

☆ Merging of two linked lists:-



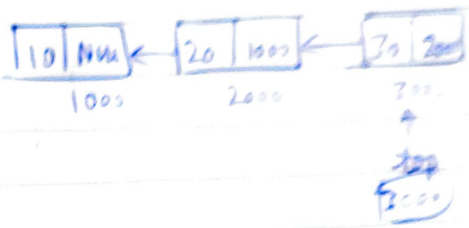

```
struct node *
[redacted] merge()
{
    struct node *temp;

    temp = [redacted];

    while (temp [redacted] != NULL)
        temp = temp -> link;

    temp -> link = q;
    q = NULL;
    free(q);

    return(p);
}
```

☆ Reversing the data list

```
void reverse()
{
    int i, j, len, k, temp;
    struct node *p, *q;
    len = length();
    i = 1;
    j = len;

    p = q = first;

    while (i < j)
    {
        k = 1;
        while (k < j)
        {
            q = q -> link;
            k++;
        }
        temp = p -> data;
        p -> data = q -> data;
        q -> data = temp;
        i++;
        j--;
        p = p -> link;
        q = first;
    }
}
```

☆ Implementation Of Stack Using Linked List

```
struct node
{
    int data;
    struct node *link;
};

struct node *top = NULL;

void insert()          // no need to check overflow as ll is dynamic
{
    struct node *temp;

    temp = (struct node *)malloc(sizeof(struct node));
    printf("Enter element to be inserted:");
    scanf("%d", &temp->data);
    temp->link = top;
    top = temp;
}

void delete()
{
    if (top == NULL)
        printf("Stack is empty");
    else
    {
        printf("%d deleted", top->data);
        top = top->link;
    }
}
```



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

☆ Implementation of queue using singly linked list :-

```
struct node
{
    int data;
    struct node *link;
};

struct node *first = NULL;
struct node *rear = NULL;

void insert()
{
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    printf("Enter element to be inserted:");
    scanf("%d", &temp->data);
```

```
        temp->link = NULL;
        if (rear == NULL)
        {
            rear = temp;
            front = temp;
        }
        else
        {
            rear->link = temp;
            rear = temp;
        }
}

void delete()
{
        if (front == NULL)
            printf("Queue is empty");
        else
        {
            printf("Deleted element is %d", front->data);
            front = front->link;
        }
}

void display()
{
        struct node *temp;
        if (front == NULL)
            printf("Queue empty");
        else
        {
            temp = front;
```

```
while(temp!=NULL)
{
    printf("%d ", temp->data);
    temp = temp->link;
}
}
}
```

* Circular queue :-

```
#include<stdio.h>
#define MAXSIZE 5

int front = 0, rear = -1, count = 0, q[MAXSIZE];

void insert()
{
    int ele;

    if(count == MAXSIZE)
        printf("Queue is full");
    else
    {
        printf("Enter an element to insert:");
        scanf("%d", &ele);
        rear = (rear+1) % MAXSIZE;
        q[rear] = ele;
        count++;
    }
}
```

1 10 19

```
void delete()
{
    if (count == 0)
        printf("Queue is empty");
    else
    {
        printf("Deleted element is: %d", q[front]);
        front = (front + 1) % MAXSIZE;
        count--;
    }
}

void display()
{
    int i;

    if (count == 0)
        printf("Queue is empty");
    else
    {
        printf("Contents of queue are:");
        for (i = front; i <= rear; i = (i+1) % MAXSIZE)
            printf("%d ", q[i]);
    }
}

void main()
{
    int choice;
    do
    {
        printf("Circular Queue operations: \n 1) Insert \n 2) Delete");
        printf("\n 3) Display \n 4) Exit \n Enter your choice:");
        scanf("%d", &choice);
```

```
switch (choice)
{
    case 1:
        insert();
        break;
    case 2:
        delete();
        break;
    case 3:
        display();
        break;
    case 4:
        break;
    default:
        printf("Invalid choice");
}
}while (choice!=4);
}
```

☆ Implementation Of Circular Singly Linked List:-

If the link field of last node contains the address of the first node in the list, then its called circular singly linked list.

```
#include <stdio.h>

struct node
{
    int item;
    struct node *link;
};

struct node *first = NULL, *tail = NULL;
```

1 10 19

i) void insert() // adding node at rear end

```
{
    struct node *temp;

    temp = (struct node *) malloc (sizeof (struct node));

    printf ("Enter element to be inserted ");
    scanf ("%d", &temp->item);
    temp->link = NULL;

    if (first == NULL)
    {
        first = temp;
        tail = temp;  tail->link = temp;
    }
    else
    {
        tail->link = temp;
        tail = temp;
        tail->link = first;
    }
}
```

ii) adding node at front end :-

```
void insertatfront()
{
    struct node *temp;

    temp = (struct node *) malloc (sizeof (struct node));
    printf ("Enter element to be inserted:");
    scanf ("%d", &temp->item);
```

```
    if(first == NULL)
    {
        first = temp;
        tail = temp;    tail -> link = temp;
    }
    else
    {
        temp -> link = first;
        tail -> link = temp;
        first = temp;
    }
}
```

iii) Deleting the node at the front :-

```
void deletenodeatfront()
{
    struct node *p;

    if (first == NULL && tail == NULL)
    {
        printf("List is empty");
    }
    else if(first == tail)
    {
        p = first;
        first = NULL;
        tail = NULL;    free(p);
    }
    else
    {
        p = first;
        first = first -> link;    tail -> link = first;    p -> link = NULL;
        free(p);
    }
}
```

iv) Deleting the node at the tail

```
void deletenodeattail()
{
        struct node *p;

        if ( first == NULL && tail == NULL)
                printf("List is empty");
        else if (first == tail)
        {
                p = first;
                first = NULL;
                tail = NULL;
                free(p);
        }
        else
        {
                p = first;
                while (p -> link != tail)
                {
                        p = p -> link;
                }
                tail -> link = NULL;
                tail = p;
                p = p -> link;
                tail -> link = first;
                free(p);
        }
}
```

☆ Linked List ADT:-

ADT Linked list is
Objects: A finite ordered list with zero or more elements.
Functions:
for all linked lists ∈ list, item ∈ element

i) insert() ::= inserting a node at the front or at the end or at a given position in the list.

ii) delete() ::= if the list is empty, then return false
else delete & return appropriate element.

iii) display() ::= if the list is empty then return false
else display the contents of the list.

iv) count() ::= if the list is empty, then return false
else return number of elements in the list.

☆ Doubly linked list:-

* Advantages of doubly linked list over singly linked list:-

i) Using DLL, its possible to traverse backwards.
ii) Insertion & deletion of node to the left of designated node is easier.
iii) Finding the predecessor of given node is less time consuming compared to singly linked list.

Defn. – A DLL is a linear collection of nodes where each node contains 3 parts:
a) Info/data field.
b) Left link field.
c) Right link field.

7 10 19

* Operations on DLL :-

i) Inserting node at rear end of the list.
ii) Inserting node at front end of the list.
iii) Displaying contents of the list.
iv) Inserting node at given position.
v) Deleting the node at the rear end of the list.
vi) Deleting the node at the front end of the list.

```
struct node
{
    int data;
    struct node *right;
    struct node *left;
};

struct node *first;

first = NULL;
```

i)
```
void insertatrear()
{
    struct node *temp, *t;

    temp = (struct node *) malloc (sizeof (struct node));

    printf("Enter an element: ");
    scanf("%d", &temp->data);
    if (first == NULL)
    {
        temp->left = NULL;
        temp->right = NULL;
        first = temp;
    }
```

```
t -> left = temp;
temp -> right = NULL;
temp -> left = t;
```

```
struct node *temp;

temp = (struct node *) malloc (sizeof (struct node));

if (first == NULL)
{
    first = temp;
    first -> left = NULL;
    first -> right = NULL;
}
else
{
    first -> left = temp;
    temp -> right = first;
    temp -> left = NULL;
    first = temp;
}
```

```
void display()
{
    struct node *t;
    if (first == NULL)
        printf("List is empty");
    else
    {
        printf("Contents of list are:");
        for (t = first; t != NULL; t = t->right)
            printf("%d", t->data);
    }
}
```

* Fn. to count no. of nodes:-

```
int count()
{
    int c = 0;
    struct node *t;
    if (first == NULL)
        return 0;
    else
    {
        t = first;
        while (t != NULL)
        {
            c++;
            t = t->right;
        }
    }
    return c;
}
```

iv) void insertatposition()
{
```
struct node *temp, *t
int length, i, loc;
temp = (struct node *) malloc (sizeof (struct node));
length = Count();
printf("Enter the element to be inserted ");
scanf ("%d", &temp->data);
printf("Enter the position : ");
scanf ("%d", &loc);

if (loc > length || loc <= 0)
        printf("Invalid location!!!");
else
{
```