KARNATAK LAW SOCIETY'S

GOGTE INSTITUTE OF TECHNOLOGY

UDYAMBAG, BELGAUM-590008

(An Autonomous Institution under Visvesvaraya Technological University, Belgaum)

(APPROVED BY AICTE, NEW DELHI)

Department of Information Science Engineering

Course Activity Report

FORMAT LANGUAGE AND AUTOMATION THEORY

Submitted in the partial fulfilment for the academic requirement of

Third Semester B.E.

Submitted by

SL No.	Name	USN
1.	G John Bright Nixon	2GI19IS016

KARNATAK LAW SOCIETY'S

GOGTE INSTITUTE OF TECHNOLOGY

UDYAMBAG, BELGAUM-590008

(An Autonomous Institution under Visvesvaraya Technological University, Belgaum)

(APPROVED BY AICTE, NEW DELHI)

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that G John Bright Nixon of Third Semester **bearing USN: 2GI19IS016** has satisfactorily completed the course in Data Structures with C Lab. It can be considered as a bonafide work carried out for partial fulfilment of the academic requirement of 3rd Semester B.E.(Information Science & Engineering) prescribed by KLS Gogte Institute of Technology, Belagavi during the academic year 2020- 2021. The report has been approved as it satisfies the academic requirements prescribed for the said degree.

Signature of the Faculty Member: **Signature of the HOD:**

Date : 30/12/2020

TERM WORK - 1

PROBLEM DEFINITION

1) Write a C program to merge contents of two files containing USNs of students in a sorted order in to the third file such that the third file contains unique USNs. Program should also display common USNs in both the files.

AIM :

The purpose of this term work is to learn the concept of File handling in C. Basic operations using files and implementation of this concept in solving problems.

THEORY :

In software industry, most of the programs have to store information fetched from programs. One such way is to store fetched information in a file. Different operations that can be performed on a file are

-> Creating a new file.
-> Opening an existing file.
-> Reading from file.
-> Writing to a file.
-> Closing a file.

PROGRAM

```
#include <stdio.h>
#include <string.h>
int main (int argc, char * argv[])
{
    FILE *f1, *f2, *f3;
    f1 = fopen ("file1.txt", "r"); // opening file in read mode
    f2 = fopen ("file2.txt", "r");
    if (f1 == NULL)
        printf ("File1 cannot be opened \n");
    if (f2 == NULL)
        printf ("File2 cannot be opened \n");
    f3 = fopen ("file3.txt", "w");
    int maxchar = 15;
    char usn1 [maxchar], usn2 [maxchar];
    fgets (usn1, maxchar, f1);
    fgets (usn2, maxchar, f2);
    while (!feof (f1) && !feof (f2))
    {
        if ((strcmp (usn1, usn2)) < 0)
        {
            fprintf (f3, "\n %s \n", usn1);
            fprintf (f3, "\n %s \n", usn2);
        }
        else
```

```
{
    fprintf (f3, "\n%.s\n", num2);
    fprint f (f3, "\n%.s\n", num1);
}
    fgets (num1, manchar, f1); // reading ment nums
    fgets (num2, manchar, f2);
}
if (feof (f2))
{
    while (! feof (f1))
    {
        fgets (num1, manchar, f1);
        fgets( fprintf (f3, "\n% \n", num1);
    }
}
else
{
    while (! feof (f2)) // put num's of file 2 in file 3
    {
        fgets (num2, manchar, f2);
        fprintf (f3, "\n%.s\n", num2);
    }
}
fclose (f1);
fclose (f2);
fclose (f3);
```

```
    printf ("Done");
    return 0;
}
```

REFERENCES:

→ Richard F Gilberg, Behrouz A Forouzan, Data structures: A Pseudo code Approach with C, Cengage 2007.

→ Horowitz, Sahni, Anderson-Freed, Fundamentals of Data structures in C, Universe Press 2nd Edition.

CONCLUSION:

In this teamwork, we have learnt about files, basic operations of files and their implementation to solve problems. We have also learned basic problem solving techniques and programming paradigms.

```
Done
Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

TERM WORK - 2

PROBLEM DEFINITION:

2. Consider a calculator that needs to perform checking the correctness of parenthesized arithmetic expression and convert the same to postfix expression for evaluation. Develope and execute a program in C using suitable datastructures to perform the same and print both the expressions. The input expression consists of a single character operands and the binary operators + (plus), - (minus) * (multiple) and / (divide).

AIM :

Aim of the teamwork is to learn the implementation of stacks in solving infix / postfix expressions.

THEORY :

Stack is a linear data structure which follows a particular order in which the operations are performed. The order is LIFO [Last in, First out].

The basic operations performed by stack are :

-> PUSH : Adds an item to the stack. If stack is full, then it is said to overflow.

-> POP : Removes an item from the top of the stack. If the stack is empty, then it is said to underflow.

-> TOP : Returns top element of the stack.

-> isEmpty : Returns true if the stack is empty.

PROGRAM :

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

  // stack type
{
    int top ;
    int capacity ;
   int *a;
};

  // stack operations
int  isEmpty (struct stack *s)
 {
   return s -> top = = -1;
}
char peek (struct stack *s)
  {
      return s -> a [s -> top];
  }
char pop (struct stack *s)
{
   if (! isEmpty (s))
        return s -> a [s -> top--];
      return '$';
}
```

```
void push (struct stack *s, char op)
{
    s->a[++s->top] = op;
    printf ("Added in stack : %c", s->a[s->top]);
}
int isoperand (char ch)
{
   return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}
// A utility function to return
// precedence of a given operator
// Higher returned value means
// Higher precedence
int Prec (char ch)
{
    switch (ch)
    {
     case '+':
     case '-': return 1;
     case '*':
     case '/': return 2;
     case '^': return 3;
    }
```

```
    return -1;
}
// The main function that converts given infix
// expression to postfix expression.
int infixToPostfix(char* exp)
{
    int i, k, l = strlen(exp);
// create a stack of capacity
// equal to expression size
    struct stack* s = (struct stack*) malloc( sizeof(struct
                                                stack));

    s->top = -1;
    s->capacity = strlen(exp);
    s->a = (int*) malloc(s->capacity * sizeof(int));
    if (!s) // checking if stack was created successfully
        return -1;
    for(i=0; k=-1; i<l; ++i)
    {
        printf("\n\nI:%d", i);
// If the scanned character is an operand,
// add it to output
        if (isOperand(exp[i]))
```

```
{
  exp[++k] = exp[i];
  printf ("OUTPUT : %c", exp[k]);
}
// if the scanned character is an '(', push it to
// the stack
else if (exp[i] == '(')
  push (s, exp[i]);
// if the scanned character is an ')', pop and
// output from the stack untill an '(' is encountered
else if [exp[i] == ')')
{
  while (! is Empty (s) && peek (s) ! = '(')
  {
    exp[++k] = pop (s);
    printf ("OUTPUT By POPPING : (%c", exp[k]);
  }
  if (! is Empty (s) && peek (s) ! = '(')
    return -1; // invalid expression
  else
  {
    pop (s;
    printf ("OUTPUT BY POPPING :) OP %c", exp[x]);
  }
```

```
}
else // on operator is encountered
{
    while (!isEmpty(s) && Prec(exp[i]) <= Prec(peek(s)))
    {
        exp[++k] = pop(s);
        printf("\n OUTPUT BY POPPING: OP %c", exp[k]);
    }
    push(s, exp[i]);
  }
}
printf("\n\n outside the for");
// pop all the operators from the stack
while(!isEmpty(s))
{
    exp[++k] = pop(s);
    printf("\n OUTPUT: %c", exp[k]);
}
exp[++k] = '\0';
printf("\n\n%s\n", exp);
}
// Driver Program to test the above functions
```

```
int main()
{
    char * exp;
    exp = (char *) malloc (1000 * sizeof (char));
    printf ("Enter an expression :");
    scanf ("%s", exp);
    infix To postfix (exp);
    return 0;
}
```

REFERENCES:

-> Richard F Gilberg, Behrouz A Fourouzan, Data structures: A Pseudo code Approach with C, Cengage 2007.

-> Horowitz, Sahni, Anderson-Freed, Fundamentals of Data structures in C, Universe Press 2nd edition.

CONCLUSION:

In this termwork, we learnt stacks, basic operations of stacks and their implementation to solve expressions. We also learnt basic problem solving techniques and programming paradigms.

```
Enter an expression : A/B-C/D


I : 0
OUTPUT : A

I : 1
Added in stack : /

I : 2
OUTPUT : B

I : 3
OUTPUT BY POPPING : OP /
Added in stack : -

I : 4
OUTPUT : C

I : 5
Added in stack : /

I : 6
OUTPUT : D

Outside the for
OUTPUT : /
OUTPUT : -
AB/CD/-

Process returned 0 (0x0)   execution time : 23.231 s
Press any key to continue.
```

```
Enter an expression : A+B*C/D


I : 0
OUTPUT : A

I : 1
Added in stack : +

I : 2
OUTPUT : B

I : 3
Added in stack : *

I : 4
OUTPUT : C

I : 5
OUTPUT BY POPPING : OP *
Added in stack : /

I : 6
OUTPUT : D

Outside the for
OUTPUT : /
OUTPUT : +
ABC*D/+

Process returned 0 (0x0)   execution time : 27.707 s
Press any key to continue.
```

TERM WORK - 3

3. **PROBLEM STATEMENT :**

A calculator needs to evaluate a postfix expression. Develope and execute a program in C using a suitable data structure to evaluate a valid postfix expression. Assume that the postfix expression is read as a single line consisting of a non negative single digit operands and binary arithmatic operators. The arithmatic operators are + (add), - (substract), * (multiply), and / (divide).

Aim

Aim of this teamwork is to learn the implementation of stacks in solving problems.

THEORY

stack is a linear data structure which follows a particular order in which the operation are performed. [LIFO -> Last in, first out].

The basic operations performed on stack are:

-> PUSH : Inserts an item into the stack. If stack is full, it is said to overflow.

-> POP : Removes an item from stack top. If stack is empty, it is said to underflow.

-> Peek : Returns the top element of the stack.

-> isEmpty : Returns true if the stack is empty.

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
struct stack
{
    int capacity;
    int *a;
    int top;
};
int isEmpty(struct stack *s) // checks if stack is empty
{
    return s->top==s->capacity;
}
void push(struct stack *s, int op) // add an item to the
                                   top of the stack
{
    s->a[++s->top]=op;
    printf("\n Added in stack : %d", s->a[s->top]);
}
```

```
int pop ( struct stack * s) // remove an item from the
                               top

{
    if ( ! isempty (s))
    {
        return  s -> a [ s -> top -- ];

    }
}
int peek ( struct stack * s) // prints the top item
{
    if ( ! isEmpty (s))
    {
        return s -> a [s -> top];

    }
}
void postFixToInFix ( char * exp) // converts postfix to infix
{
    int i, op1 = 0, op2 = 0, result = 0, x = 0;
    struct stack * s = ( struct stack *) malloc ( sizeof ( struct
                                                    stack));

    s -> top = -1;
    s -> capacity = strlen (exp);
    s -> a = (int *) malloc (8 -> capacity * sizeof(int));
```

```
for (i=0; i<s->capacity; i++)
{
    // if it is a number, add it to a stack.
    if (exp[i]>='0' && exp[i]<='9')
    {
        x = exp[i]-'0';
        push(s, x);
    } // if it is an operand, then pop two items
      // and perform the particular operation.
    else
    {
        op2 = pop(s);
        printf("\n Popped : %d", op2);
        op1 = pop(s);
        printf("\n Popped : %d", op1);
        switch (exp[i])
        {
        case '+': result = op2+op1;
                  printf("\n Added both : %d", result);
                  break;
        case '-': result = op1-op2;
                  printf("\n subtracted both : %d", result);
                  break;
```

```
case '*': result = op1 * op2;
          printf ("\n Multiplication both : %d", result);
          break;
case '/' : result = op1 / op2;
          printf ("\n Divided both : %d", result);
          break;
case '^':
case '$' : result = pow (op1, op2);
          break;
default : printf ("\n Incorrect operation!");
}
push (s, result);
}
}
result = pop(s);
printf ("\n Answer : %d", result);
}
int main()
{
    char exp[50];
    printf ("\n Enter a POSTFIX expression :");
    scanf ("%s", exp);
    postFixToInfix (exp);
}
```

REFERENCES:

→ Richard F Gilberg, Behrouz A Fourouzan, Data structures : A Pseudo code Approach with C, Engage 2007.

→ Horowitz, Sahni, Anderson-Freed, Fundementals of Data structures in C, Universe Press 2nd Edition.

CONCLUSION:

In this teamwork, we learnt basic operations of stacks and their implementation to solve problems. We have also learned basic problem solving techniques and programming paradigms.

```
Enter a POSTFIX expression : 213+*4-

Added in stack : 2
Added in stack : 1
Added in stack : 3
Popped : 3
Popped : 1
Added both : 4
Added in stack : 4
Popped : 4
Popped : 2
Multiplication both : 8
Added in stack : 8
Added in stack : 4
Popped : 4
Popped : 8
Subtracted both : 4
Added in stack : 4
Answer : 4
Process returned 0 (0x0)   execution time : 21.941 s
Press any key to continue.
```

TERM WORK - 4

PROBLEM DEFINITION :

Write a C program to simulate the working of Messaging system in which a message is placed in a Queue by a Message sender, a message is removed from the queue by a Message reciever, which can also display the contents of the queue.

Aim:

The purpose of this teamwork is to learn the concept of Queues in C language. Basic operations of queues using circular linked list in solving problems.

THEORY:

Like stack, queue is a linear data structure that follows a particular order in which the operations are performed [FIFO -> First in, first out].

The four basic queue operations are:

-> Enqueue :-> Inserts the item to the rear end of the Queue.

-> Dequeue :-> Deletes the item from the front end of the Queue

-> Front :-> Gets the front item from queue.

-> Rear :-> Gets the item from rear end of the queue.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#datanode structure
typedef struct SMS
{
    char message[100];
    struct Queue * link;
} Node;
// prototype
void Send (Node **, Node **, char*);
char * recieve (Node **, Node**);
void display (Node*);
// driver code
int main ()
{
    // create an empty queue
    Node * front, *rear;
    front = rear = NULL;
    int choice;
    char * item;
    while (1)
    {
```

```
printf ("\n1 : Send a message \t 2: Receive the message \t
        3: Display \t 4: Exit \n");
printf ("Enter choice : \t");
scanf ("%d", &choice);
switch (choice)
{
case 1: printf ("Enter the message to send:");
        item = (char*) malloc (25);
        if (item == NULL)
        {
          printf ("Malloc failure \n");
          exit (2);
        }
        scanf ("%s", &item);
        send (&front, &rear, item);
case 2: item = receive (&front, &rear);
        if (strcmp (item, "#") == 0)
          printf ("Messaging system has no msgs\n");
        else
          printf("Recieving message is: %s\n", item);
        break;
case 3: display (front);
```

```
            break;
        case 4: exit(0);
    }
}
    return 0;
}
void send(Node **front, Node **rear, char *item)
{
    Node *temp = (Node*)malloc(sizeof(Node));
    if(!temp)
    {
        printf("Malloc failure\n");
        exit(1);
    }
    temp -> message = (char*) malloc(sizeof(item));
    strcpy(temp -> message, item);
    temp -> link = NULL;
    if(*rear == NULL)
        *front = *rear = temp;
    else
    {
        (*rear) -> link = temp;
        *rear = temp;
    }
}
```

```
char * reciever (Node ** front, Node * * rear)
{
    Node * temp;
    char * k;
    if (*front == NULL) // empty queue
        return "#";
    // queue is not empty
    temp = * front;
    k = (char *) malloc (sizeof (temp -> message));
    strcpy (k, temp -> message);
    * front = (* front) -> link;
    if (* front = NULL)
        * rear = NULL;
    free (temp);
    return k;
}
void display (Node * front)
{
    Node * temp = front;
    if (temp == NULL)
    {
        printf ("Messaging system has no messages \n");
        return;
    }
    printf ("Messages are : \n");
```

```
while (temp)
{
    printf ("%s\t\t", temp -> message);
    temp = temp -> link;
    }
}
```

REFERENCES:

-> Richard F Gilberg, Behrouz A Fouruzan, Data structures: A Pseudo code approach with C, vangage 2007.

-> Horowitz, sahni, Anderson - Freed, Fundamentals of Data structures in C, Universe Press 2nd edition.

CONCLUSION:

In this teamwork, we learnt concepts of queues, basic operations of queues and their implementation to solve problem we also learned basic problem solving techniques and programming paradigms.

```
 1: Send a message
 2: Receive a message
 3: Display
 4:Exit
1

Enter the message: hello

Message sent Successfully!
 1: Send a message
 2: Receive a message
 3: Display
 4:Exit
2
Message=hello
Message read Successfully!
 1: Send a message
 2: Receive a message
 3: Display
 4:Exit
4

Process returned 0 (0x0)   execution time : 13.202 s
Press any key to continue.
```

Term work : 05

Consider a supermarket scenario where sales manager wants to search for the customer details using a customer id. customer information like (custid, custname and custphno) are stored as a structure and custid will be used as hash key. Develope and execute a program in C using suitable data structures to implement the following operations :

a. Insertion of a new data entry.

b. search for customer information using custid.

c. display the records (Demonstrate collision and its handling using linear probing methods)

Aim:

To learn the implementation of hashing in solving problems

Theory:

Hashing is an important data structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

Let a hash function H(x) maps the value x at the index x% 10 in an array eg. if the list of values is [11, 12, 13, 14, 15] it will be stored at positions {1, 2, 3, 4, 5} in the array or Hash Table respectively.

Source code:

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

#define SIZE 10

struct customer {
    int custid;
    char custname[30];
    double custphno;
};
struct Record {
    struct customer info;
    int empty;
};
int Hashfn(int key){
    return (key % SIZE);
}
int search(int key, struct Record ht[]){
    int count, temp, pos;
    temp = Hashfn(key);
    pos = temp;
    for(count = 1; count <= size; count++){
        if(ht[pos].empty == 1){
        return -1;
        }
    if(ht[pos].info).custid == key){
```

```
        return pos;
      }
    pos = (temp + count) % SIZE;
    }
    return -1;
}
void InsHT LP(struct customer cust, struct Record ht[]){
      int count, port, temp;
      int key = cust custid;
      temp = Hashfn(key);
      pos = temp;
      for(count = 1; count <= SIZE; count++){
          if(ht[port].empty == 1){
              ht[pos].info = cust;
              ht[pos].empty = -1;
              printf("\n Record Inserted into Hash Table \n");
              return;
          }
      if((ht[pos].info).custid == key){
        printf("\n Duplicate Record cannot be Inserted \n");
      }
      pos = (temp + count) % SIZE;
    }
    printf("\n Hash Table is Full \n");
}
void Display(struct Record ht[]){
int count;
printf("\n Hash Table");
for(count = 0; count < SIZE; count++){
    printf("\n [%.d]: \t", count);
```

```
if (ht[count].empty == -1){
printf ("\n Countmer. ID:%d \t Name : %s\t Phone : %lf ",
 [ht[count].info).custid, ht[count].info).cutname,
 (ht[count].info.custphno);
}
else
  printf ("\n No Hash Entry \n");
}
}
}

int main (){
  int count, key, option;
  struct Record ht[SIZE];
  struct customer cust;
 for ( count = 0; count < SIZE ; count ++){
     ht[count].empty = 1;
  }
  while (1) {
   printf ("\n 1. Insert a Record \n");
   printf ("\n 2. Search a Record \n");
   printf ("\n 3. Display all Records \n");
   printf ("\n 4. Exit \n");
   printf ("Enter your option :");
   scanf ("%d", &option);
   switch (option) {
    case 1 : printf ("\n Enter customer id, name, ph :");
   scanf ("%d %s %lf", &cust.custid, &cust.custname,
   &cust.custphno);
```

```
        INSHT_LP (cust, ht);
        break;
    case 2: printf("\n Enter the key to search: ");
        scanf("%d", &key);
        count = search(key, ht);
        if (count == -1){
            printf("\n Record Not Found \n");
        }
        else{
            printf("\n Record Found at Index: %d \n", count);
        }
        break;
    case 3: Display(ht);
        break;
    case 4: exit(1);
    }
}
return 0;
```

REFERENCES:

Books

1. Richard F. Libberg, Behrouz A Forouzan, Data structures : A Pseudo code Approach with C, Cengage 2007.
2. Horowitz, Sahni, Anderson-Fread, Fundamentals and Data structures in C, Universe Press 2nd Edition.

E-Resources:

1. https://geeksforgeeks.org/

CONCLUSION:

In this term work, we learn about hashing, basic operations of hashing and their implementation to solve problems. we also learnt basic problem solving techniqu and programming paradesigns.

```
1. Insert a Record
2. Search a Record
3. Display All Records
4. Exit

Enter Your Option:2

Enter the Key to Search:2

Record Found at Index pos:2

1. Insert a Record
2. Search a Record
3. Display All Records
4. Exit

Enter Your Option:1

Enter Customer id, name, ph:3 Ximi 986564346

Record Inserted into Hash Table

1. Insert a Record
2. Search a Record
3. Display All Records
4. Exit

Enter Your Option:3

Hash Table
[0]:    Customer - ID: 0        Name: minho     Phone: 765645474.000000
[1]:    Customer - ID: 1        Name: hazel     Phone: 767825626.000000
[2]:    Customer - ID: 2        Name: Olivia    Phone: 98764356.000000
[3]:    Customer - ID: 3        Name: Ximi      Phone: 986564346.000000
[4]:    No Hash Entry
[5]:    No Hash Entry
[6]:    No Hash Entry
[7]:    No Hash Entry
[8]:    No Hash Entry
[9]:    No Hash Entry
1. Insert a Record
2. Search a Record
3. Display All Records
4. Exit

Enter Your Option:4

Process returned 1 (0x1)   execution time : 122.016 s
Press any key to continue.
```

TERM WORK : 06

Consider a warehouse where the items have to be rearranged in an ascending order. Development and execute a program in C using suitable data structures to implement warehouse such that items can be traced easily.

AIM:

To learn the implementation of linked list in solving problems.

THEORY

A linked list is a sequence of data structures which are connected together via links. Linked list is a sequence of links which contains items. Each link contains a connection to another link. It is the second most used data structure after array.

Basic operations of linked list.

Insertion - Adds an elements at the beginning of the list.

Deletion - Deletes an element at the beginning of the list.

Display - Displays the complete list.

Delete - Deletes an element using the given key.

SOURCE CODE:

```
#include <stdio.h>
struct node
{
  int data;
  struct node * next;
};
void Display (struct node * head)
{
  struct node * temp;
  temp = head;
  while (temp != NULL)
  {
  printf ("%d", temp -> data);
  temp = temp -> next;
  }
}
struct node * Add (struct node * head, int value)
{
  struct node * newnode, * prev * curr;
  newnode = (struct node * )malloc (sizeof (struct node));
  newnode -> data = value;
  newnode -> next => NULL;
  if (newnode == NULL)
  {
     printf ("Error : Could not allocate memory ; ");
  }
  else
     {
       if (head == NULL)
```

```
head = new node;
else
{
   if(newnode -> data   < head -> data)
  {
    newnode -> next = head;
    head= newnode;
  }
 else
   {
     curr  = head -> next;
     prev = head;
    while (curr ! = NULL && newnode -> data -> curr -> data)
    {
      prev = prev -> next;
      curr = curr -> next;
    }
    prev -> next = newnode;
    newnode -> next = curr;
   }
  }
 }
  return  head;
}
void main()
{
```

```
int choice, value;
struct node * head = NULL;

for(;;)
{
printf("\n Enter 1. Add 2. Display 3. Exit ");
printf("\n Enter choice :");
scanf("%d", &choice);
switch (choice)
}
case 1: printf("\n Enter value");
        scanf("%d", &value);
        head = Add (head, value);
        break;
case 2: if (head == NULL)
        {
        printf("List is empty");
        }
        Display (head);
        break;
case 3: exit(1);
        break;

  }

}
}
```

REFERENCE :

Books :

1. Richard F Wilberg, Behrouz A Forouzan, Data Structures - A Pseudo code Approach with C, gngage 2007.
2. Horowitz, Sahni, Anderson - Freed Fundamentals of Data structures inc, Universe Press 2nd Edition.

E-Resources.

1- http://geeksforgeeks.org/

CONCLUSION:

In this term work, we learnt about linked list basic operations of linked list and their implementations to solve problems. We also learnt basic problem solving techniques and programming paradigms.

```
Enter 1.Add 2.Display 3.Exit
Enter choice:2
List is empty
Enter 1.Add 2.Display 3.Exit
Enter choice:1
Enter value:7

Enter 1.Add 2.Display 3.Exit
Enter choice:1
Enter value:4

Enter 1.Add 2.Display 3.Exit
Enter choice:2
4  7
Enter 1.Add 2.Display 3.Exit
Enter choice:1
Enter value:5

Enter 1.Add 2.Display 3.Exit
Enter choice:1
Enter value:12

Enter 1.Add 2.Display 3.Exit
Enter choice:1
Enter value:7

Enter 1.Add 2.Display 3.Exit
Enter choice:2
4  5  7  7  12
Enter 1.Add 2.Display 3.Exit
Enter choice:1
Enter value:0

Enter 1.Add 2.Display 3.Exit
Enter choice:1
Enter value:3

Enter 1.Add 2.Display 3.Exit
Enter choice:2
0  3  4  5  7  7  12
Enter 1.Add 2.Display 3.Exit
Enter choice:3

Process returned 1 (0x1)   execution time : 51.828 s
Press any key to continue.
```

TERM WORK - 07:

PROBLEM DEFINITION:

Consider a polynomial addition for two polynomials. Develop and execute a program in C using suitable data structures to implement the same.

THEORY:

A polynomial may be represented using array (or) structure. A structure may be defined such that it contains two parts :-

1) one is the coefficient and
2) second is the corresponding exponent.

The basic idea of polynomial addition is to add coefficient parts of the polynomials having same exponent.

Terms are arranged in descending order of the exponents of terms in 'x'.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node
{
        int coef;
        int exp;
        struct node * next;
} NODE;
typedef struct
{
       int count;
       NODE *head;
} POLY;
NODE * getnode (int c , int e)
{
     NODE * newnode = (NODE *) malloc (sizeof (NODE));
     newnode -> coef = c;
     newnode -> exp = e;
     newnode -> next = NULL;
     return newnode;
}
void addterm (POLY *pp, int c , int e) // insert rear
{
     NODE * newterm, *temp;
     newterm = getnode (c, e);
     if (pp -> head == NULL) // list empty case
     {
        pp -> head = newterm;
        pp -> count ++;
        return;
      }
```

```
    temp = pp -> head;
    // loop till the last node is reached
    while ( temp -> next != NULL)
        temp = temp -> next;
    // add the node as last node
    temp -> next = newterm;
    pp -> count++;
}
void display ( POLY p)
{
    NODE * temp = p.head;
    while (temp != NULL)
    {
        printf ("%d x ^ %d -> ", temp -> coef, temp -> exp);
        temp = temp -> next;
    }
    printf ("NULL");
    printf ("\n Total terms : %d \n", p.count);
}
void polyadd ( POLY *p1, POLY *p2, POLY *p3)
{
    int c1, c2;
    NODE *temp, *t1, *t2;
    c1 = p1 -> count;
    c2 = p2 -> count;
    t1 = p1 -> head;
    t2 = p2 -> head;
    while (c1 != 0 && c2 != 0) // loop till one of the polynomial is
                               processed completely
```

```
{
temp = (NODE *) malloc (sizeof (NODE));
if (t1 -> exp == t2 -> exp)
{
    temp = (NODE *) malloc
    // terms from both polynomials are processed
    addterm (p3, t1 -> coef + t2 -> coef, t1 -> exp);
    c1--;
    c2--;
    t1 = t1 -> next;
    t2 = t2 -> next;
}
else if (t1 -> exp > t2 -> exp)
{
    // term from both polynomial 1 is processed
    addterm (p3, t1 -> coef, t1 -> exp);
    c1--;
    t1 = t1 -> next;
}
else // t1 -> exp < t2 -> exp
{
    // term from both polynomial 2 is processed
    addterm (p3, t2 -> coef, t2 -> exp);
    c2--;
    t2 = t2 -> next;
}
}
while (c2 != 0) // add remaining terms from second polynomial
                ... if any
{   addterm (p3, t2 -> coef, t2 -> exp);
    c2--;
    t2 = t2 -> next;
```

```
}
while (c1!=0) // add remaining terms from second polynomial
                                        ... if any
{
    addterm (p3, t1->coef, t1->exp);
    c1--;
    t1=t1->ment;
}
int main (int argc, char *argv[]){
    POLY p1, p2, p3;
    // initialize polynomial 1
    p1.head = NULL;
    p1.count=0;
    // construct polynomial 1 by adding each node
    addterm (&p1,3,14);
    addterm (&p1,2,8);
    addterm (&p1,1,0);
    printf ("Polynomial 1 : \n");
    display (p1);
    // initialize polynomial 2
    p2.head = NULL;
    p2.count=0;
    // construct polynomial 1 by adding each node
    addterm (&p2,100,45);
    addterm (&p2,8,14);
    addterm (&p2,4,6);
    addterm (&p2,1,5);
```

```
printf ("Polynomial 2: \n");
display(p2);
// Initialize polynomial 3
p3.head = NULL;
p3.count = 0;
// p3 = p1 + p2
polyadd (&p1, &p2, &p3);
printf ("Resultant polynomial : \n");
display(p3);
return 0;
}
```

REFERENCES:

* Richard F Gilberg, Behrouz A Fourouzan, Data structures: A Pseudo code Approach with C. Cengage 2007.
* Horowitz, sahni Anderson-Freed, Fundamentals of Data Structures in C, Universe Press 2nd Edition.

CONCLUSION:

In this team work, we learnt about concept of polynomial addition for two polynomials and their implementation to solve problems. We also learned basic problem solving techniques and programming paradigms.

```
3x^14 -> 2x^8 -> 1x^0 -> NULL
Total terms: 3
3x^14 -> 2x^6 -> 1x^0 -> NULL
Total terms: 3
11x^14 -> 2x^8 -> 2x^6 -> 2x^0 -> NULL
Total terms: 4
Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

TERM WORK - 08

PROBLEM DEFINITION:

Develop and execute a program in C to perform following operations on binary search tree:

a) To count number of non-terminal nodes.

b) To count number of terminal nodes.

c) To count nodes with degree 2.

d) To count total number of nodes.

THEORY:

A Binary search tree (BST) is a tree in which all the nodes follow the mentioned properties:

① The value of the key of the left sub-tree is less than the value of its parent (root) node's key.

② The value of the key of the right sub tree is greater than (or) equal to the value of its ~~produc~~ parent (root) nodes key.

Thus, BST divides all its sub-trees into two segments: the left sub-tree and the right sub-tree.

```
#include <stdio.h>
#include <stdlib.h>
/* A binary tree node has data, pointer to left child and
   a pointer to right child */
struct node
{
        int data;
        struct node* left;
        struct node right;
};
/* Function to get the count of leaf nodes in a binary tree */
    int getLeafCount (struct node* node)
{
 if(node == NULL)
        return 0;
 if(node->left == NULL && node->right == NULL)
        return 1;
 else
        return getLeafCount(node->left) + getLeafCount(node->righ
}
// Non-leaf nodes
    int nonleafnodes(struct node** root)
    {
        if(root == NULL || (root->left == NULL && root->right
                                                == NULL))
        return 0;
```

```
    return 1 + nonleafnodes(root->left) + nonleafnodes(root->right);
}

// Function to count the nodes with degree 2
int counttwo(struct node* node)
{
if (node == NULL)
{
return 0;}
if (node->left != NULL && node->right != NULL)
 {
 return 1;
}
return counttwo(node->left) + counttwo(node->right);
}

// Function to count the no of nodes in bt

int countnodes(struct node * root)

{
 if (root == NULL) return 0;

else {
return 1 + countnodes(root->left) + countnodes(root->right);

}

}
/* Helper funtion that allocates a new node with the given
  data and NULL left and right pointers. */

struct node* newNode(int data)

{
struct node * node = (struct node*)
```

```
    malloc (sizeof (struct node));
    node -> data = data;
    node -> ~~data~~ left = NULL;
    node -> right = NULL;
    return (node);
    }
  /* Driver program to test above functions */
  int main()
  {
  /* create a tree */
  struct node * root = newNode (1);
  root -> left = newNode (2);
  root -> right = newNode (3);
  root -> left -> left = newNode (4);
  root -> left -> right = newNode (5);
  /* get leaf count of the above created tree */
  printf ("Leaf count of the tree is %d \n", getLeafCount (root));
  printf ("non Leaf count of the tree is %d \n", nonLeafnode (root));
  printf ("nodes with degree 2 is %d \n", counttwo (root));
  printf ("Total count of nodes is %d \n", countnodes (root);

  getchar ();
  return 0;
```

}

REFERENCES :-

* Richard F Gilberg · Behrouz A Fourouzan. Data structures: A Pseudo code Approach with C, Cengage 2007.

* Horowitz, Sahni Anderson-Fred. Fundamentals of Data structures in C, Universe Press 2nd Edition.

CONCLUSION:

In this termwork, we learnt about binary search tree its operations of counting the number of non-terminal nodes, terminal nodes, nodes with degree 2. and total number of nodes. We also learned basic problem solving techniques and programming paradigms.

```
Leaf count of the tree is 3
non Leaf count of the tree is 2
nodes with degree 2 is 1
Total count of nodes is 5


Process returned 0 (0x0)   execution time : 1960.964 s
Press any key to continue.
```

TERMWORK - 09

PROBLEM DEFINITION:

Write a Program in C using suitable data structures to create a binary tree for an expression. The tree traversals in some proper method should result in conversion of original expression into prefix, infix and postfix forms.

Display the original expression along with 3 different forms also.

THEORY:

In a Preorder traversal the sequence is node -> left -> right. Since the node is visited before the left sub-tree it is known as preorder traversal.

In an inorder traversal the sequence is left -> node -> right. Since we will visit the nodes "in order" from left to ~~eighth~~ right it is known an In order traversal.

In a Postorder traversal the sequence is left -> right -> node. Since the left subtree is visited before the right sub tree it is known as Postorder traversals.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
struct node // Binary tree node
{
  struct node* left; // left child
  char data; // operator or operand
  struct node* right; // right child
};
char expression[256]; // same arithmetic expression
char operatorstack[128]; // stack: holds arithmetic operators
int operatorStackTop = -1;
struct node* outputstack[128]; // Nodes / tree of operands or sub
                                  expressions address
int outputstackTop = -1;
struct node* createNode(char value)
{ // allocate space for new node, ennode holds address if
                                  malloc successful
struct node * newNode = (struct node*) malloc(sizeof(struct
                                  node));
newNode -> data = value; // value is either operand or operator
newNode -> left = NULL;
newNode -> right = NULL; // Initialise newnode's left and
                                  right node with NULL
return newNode;
}
```

```
void pushInOutputStack (struct node *newNode) // Push address of
                                                  result tree with
{ // operator as root and subexpressions / operands as children
  onto output stack outputStack[++outputStackTop] = newNode;
}
struct node* popFromOutputStack () // pop address of result tree
    from output stack
{
return outputStack [outputStackTop--];
}
void pushIntoOperatorStack (char operation) // push operator on
                                               operator stack
{
  operatorStack [++operatorStackTop] = operation;
}
char popFromOperatorStack () // pop operator from operator stack.
{
  return operatorStack [operatorStackTop--];
}
void buildExpression () // build expression tree with operator
                           from operator stack
{ // and results from output stack
char operation = popFromOperatorStack (); // op = pop operator from
                                             the operator stack
struct node* subExp2 = popFromOutputStack(); // subExp2 = pop from
                                                output stack
```

```
struct node * ~~nnew~~ subExp1 = popFromOutputStack(); // subExp1 =
                                                    pop from output stack
struct node * newNode = createNode (operation); // build tree with
                                                    op as root
newNode -> left = subExp1; // and subExp1 and subExp2 as left
                                                    and right
newnode -> right = subExp2; // children respectively
pushIntooutputStack (newNode); // push result on result stack
}
void printErrorMessage()
{
printf ("\n Invalid Expression, Expression should have single
                                                    character");
printf (" or digit operand and + - * / as operators \n");
}
int precedence (char operation) // return precedence of operators
{
switch (operation)
{
case 'C': return 0;
case '+':
case '-': return 1; // lower
case '/':
case '*': return 2; // higher precedence
}
}
```

```
struct node* infixToBinaryTree ( char *expression) // return
                                          address of binary tree
{ // representation of expression
int i = 0;
while ( expression [i] != '\0') // while there are tokens to be
                                          read
{
  if ( expression [i] == ' ' ); // over look white space
  else if ( isdigit ( expression [i]) || isalpha ( expression [i]))
  // if the token is a number / alphabet. then push it to the
                                          output stack
  pushIntoOutputStack ( createNode ( expression [i]));
else if ( expression [i] == '(') // if the token is a left bracket
                                          "(", then:
pushIntoOperatorStack ( expression [i]); // push it onto the operator
                                          stack
else if ( expression [i] == ')') // if the token is a right bracket
                                          ")", then:
{
int j = operatorStackTop ; // while the operator at the top of the
                                          operator
while ( operatorStack [j] != '(' && j >=0) // stack is not a
                                          left bracket:
{
  buildExpression ();
  j--;
} // if (j<0) { printErrorMessage (); exit (1); }
char temp = popFromOperatorStack (); // pop the left bracket
    from the stack
```

```
}
else if ( expression [i] == '+' || expression [i] == '-' ||
expression [i] == '*' || expression [i] == '/')
{ // if the token is an operator, then:
while ( precedence ( operatorStack [operatorStackTop])
>= precedence (expression [i])) // while there is an operator at
// the top of the operator stack with >= precedence:
buildExpression ();
push1ntoOperatorStack ( expression [i]); // push the read operator
                                          onto the operator stack
}
else
{
printErrorMessage (); exit (1);
}
i++;
}
// if there are no more tokens to read:
while (operatorStackTop != -1) // while there are still operator tokens
buildExpression (); // on the operator stack: build expression
return ( popFromOutputStack ()); // pops from output stack, return
                                    result address
}
void preorder (struct node * root) // The tree traversals in some
                                    proper method should
{ // result in conversion of original expression
if ( root != NULL) // into prefix, infix and postfix forms
```

```
{ // preorder gives prefix
printf ("%c", root -> data); // Print node
preorder (root -> left); // process left sub tree
preorder (root -> right); // process right sub tree
}
}
void inorder (struct node *root) // Inorder traversal of AST gives
{ // infix notation of corresponding infix expression
if (root != NULL)
{
 inorder (root -> left); // Process left sub tree
 printf ("%c", root -> data); // print node
 inorder (root -> right); // Process right sub tree
 }
 }
 void postorder (struct node *root) // postorder gives postfix
{
if (root != NULL)
{
postorder (root -> left); // Process left sub tree
postorder (root -> right); // Process right sub tree
printf ("%c", root -> data); // Print node
}
}
int main ()
{
```

```
printf("\n Expression should have single character or
                digit as operand");
printf("and + - * / operators \n Enter valid arithemetic
                expression :");
scanf("%[^\n]", expression); // Why format specifier as %[^\n]?
struct node *ast = infixToBinaryTree(expression);
printf("\n original expression = %s\n", expression); // Display
                original expression
printf("\n Prefix = "); // along with the three different
                forms also
preorder(ast);
printf("\n Infix = ");
inorder(ast);
printf("\n Postfix = ");
postorder(ast);
return(0);
}
```

REFERENCES:

* Richard F Gilberg, Behrouz A. Fouruzan, Data structure : A Pseudo code Approach with C, Cengage 2007.
* Horowitz, Sahni Anderson - Fred, Fundamentals of Data structures in C, Universe Press 2nd Edition.

CONCLUSION:

In this termwork we learnt about binary tree traversals in some proper method resulting in conversion of original expression into prefix, infix and postfix terms.

We also learned basic problem solving techniques and programming paradigms.

```
Expression should have single character or digit as operand and + - * / operators
Enter valid arithmetic expression : (2+3*(4/2)-1)

Original expression = (2+3*(4/2)-1)

Prefix = - + 2 * 3 / 4 2 1
Infix = 2 + 3 * 4 / 2 - 1
Postfix = 2 3 4 2 / * + 1 -
Process returned 0 (0x0)   execution time : 129.288 s
Press any key to continue.
```

TERM WORK - 10

PROBLEM DEFINITION:

Develop and execute a program with C using suitable data structures to perform searching a data item in an ordered list of items in both directions and implement the following operations:

a. Create a doubly linked by adding each node at the start.

b. Insert a new node at the end of list.

c. Display the content of a list.

Consider an integer number as a data item.

THEORY:

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.

Therefore, in a doubly linked list, a node consists of three parts:

node data, pointer to the next node in sequence (next pointer), pointer to the previous node (previous pointer). The previous part of the node and the next part of the last node will always contain null indicating end in each direction.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node{
    int data;
    struct node *prev;
    struct node *next;
} NODE;
typedef struct {
    int count;
    NODE *head;
    NODE *rear;
} LIST;
LIST * createList()
{
        LIST *lptr = (LIST*) malloc (sizeof (LIST));
        lptr -> count = 0;
        lptr -> head = lptr -> rear = NULL;
        return lptr;
}
NODE * getNode (int e)
{
    NODE *t = (NODE*) malloc (sizeof (NODE));
    t -> data = e;
    t -> next = t -> prev = NULL;
    return t;
}
```

```
void insertFront (LIST *lp, int ele)
{
    NODE * NewNode = getNode (ele);
    if (lp -> count == 0) // list is empty
    {
        lp -> head = lp -> rear = newNode;
    }
    else
    { // non empty case
        newNode -> next = lp -> head;
        (lp -> head) -> prev = newNode;
        lp -> head = newNode;
    }
    lp -> count++;
    return;
}
void insertRear (LIST *lp, int a)
{
    NODE * newnode;
    newnode = getNode (a);
    if (lp -> head == NULL)
    {
        insertFront (lp, a);
        return;
    }
    lp -> rear -> next = newnode;
    newnode -> prev = lp -> rear;
    lp -> rear = newnode;
    lp -> count++;
    lp -> rear = newnode;
    return;
}
```

```
void Displaylist ( LIST lp)
{
        NODE * temp;
        if ( lp . count ==0)
        {
             printf ("\n List is empty");
             return ;
         }
        temp = lp . head ;
        printf ( " NULL");
        while ( temp ! = NULL)
             {
        printf (" <- %d ->", temp -> data );
       temp = temp -> next ;
    }
    printf ( "NULL");
    printf ("\n Node Count : %d \n", lp . count);
 }
// dir =1 for left to right and 2 for right to left
int searchList ( LIST lp, int ele, int dir)
{
     NODE *np;
     int cnt =1;
     if ( dir == 1) {
          np = lp . head ;
          while ( np -> next  != NULL) && ( np -> data  != ele))
                  {
               np = np -> next;
               cnt ++;
             }
```

```
        if (np -> next == NULL)
            return 0;
        return cnt;
    }
    else if ( dir == 2)
    {
        np = lp.rear;
        while (np -> prev != NULL) && (np->data != ele)){
        np = np -> prev;
        cnt++;
    }
    if ( np->prev == NULL)
        return 0;
    return cnt;
    }
    else
    {
        printf ("Wrong Direction Parameter!!");
        return 0;
    }
}
int main ( int argc, char *argv[])
{
    LIST *pl = createList();
    int ch, inp, dir;
    while(1)
    {
        printf ("\n1. Insert Front \n2. Insert Rear \n3. Display
                List \n4. Search \n5. Exit");
        printf (" Enter your choice: ");
        scanf ("%d", &ch);
```

```
switch (ch)
{
Case 1:
    printf (" Enter the element to be inserted : ");
    scanf ("%d", & inp);
    insertFront ( pl, inp);
    break;

Case 2:
    printf (" Enter the element to be inserted : ");
    scanf ("%d", & inp);
    insertRear ( pl, inp);
    break;

Case 3:
    displayList (*pl);
    break;

Case 4:
    printf (" Enter the element to be searched : ");
    scanf ("%d", & inp );
    printf (" Enter the Direction : (1: From Begining 2: From End)");
    scanf ("%d", & dir);
    int temp = searchList (*pl, inp , dir);
    if( temp)
        printf (" Search Element found at position %d", temp);
    else
        printf (" Search Element not found !");
    break;

Case 5:
    printf (" Exiting ...");
```

```
            exit (0);
            break;
        default:
            printf ("Invalid Input!");
            break;
        }
    }
    return 0;
}
```

REFERENCES:

* Richard F Gilberg, Behrouz A Forouzan, Data Structures: A Pseudo code Approach with C Cengage 2007.

* Horowitz, Sahni, Anderson-Freed, Fundamentals of Data Structures in C, Universe Press 2nd Edition.

CONCLUSION:

In this termwork, we learnt about a doubly linked list, operations like searching a data item, inserting a new node at the end and at the same start (or) begining of a list and displaying the contents.

We also learned basic problem solving techniques and programming paradigms.

```
1.Insert Front
2.Insert Rear
3.Display List
4.Search
5.ExitEnter your choice: 1
Enter the element to be inserted: 23

1.Insert Front
2.Insert Rear
3.Display List
4.Search
5.ExitEnter your choice: 1
Enter the element to be inserted: 20

1.Insert Front
2.Insert Rear
3.Display List
4.Search
5.ExitEnter your choice: 2
Enter the element to be inserted: 22

1.Insert Front
2.Insert Rear
3.Display List
4.Search
5.ExitEnter your choice: 2
Enter the element to be inserted: 31

1.Insert Front
2.Insert Rear
3.Display List
4.Search
5.ExitEnter your choice: 3
NULL<- 20 -> <- 23 -> <- 22 -> <- 31 -> NULL
Node Count: 4

1.Insert Front
2.Insert Rear
3.Display List
4.Search
5.ExitEnter your choice: 4
Enter the element to be searched: 23
Enter the Direction:(1: From Begining 2: From End)1
Search Element found at position 2
1.Insert Front
2.Insert Rear
3.Display List
4.Search
5.ExitEnter your choice: 5
```

Department of Information Science & Engineering

Semester III **Academic Year 2020-21**

Data Structures with C Lab **Open Ended Experiments**

1. Develop a program in C to reverse the given number using stack
2. Develop a C program to Interchange the two adjacent nodes in a given circular linked list

Sample Input/output

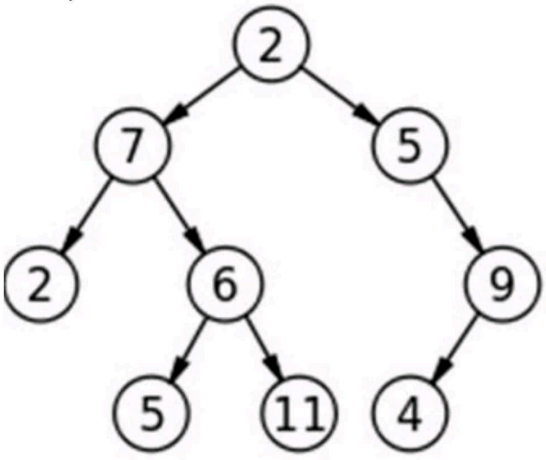
Initial List:

1->2->3->4->5->1

Enter the data of the node which you want to exchange with its next node: 3

1->2->4->3->5->1

3. C program to convert a Binary Tree into a Singly Linked List by Traversing Level by Level
   Example:



The above binary tree is converted to 2 → 7 → 5 → 2 → 6 → 9 → 5 → 11 → 4 → NULL

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#define MAX 20

int top = -1;
int item;

int stack[MAX];

void push(int item);

int pop(void);

int isempty(void);

int isFull(void);

int main()
{
    int stack[MAX];
    int result[MAX];
    int i, n, count=0, temp, temp2, ans=0;
    double a[MAX];
    a[0]=1;
    for(i=1; i<MAX; i++)
    {
        a[i] = a[i-1] * 10;
    }
    printf("Input a number : ");
    scanf("%d", &n);
    temp=n;
    temp2 = n;
    while(temp!=0){
        temp/=10;
        count++;
    }
```

```
// printf ("%d\n", count);
for (i=0; i<count; i++)
{
    push (temp2%10);
    temp2 = temp2/10;
}
for(i=0; i<count; i++)
{
    result[i] = pop() * a[i];
}
for (i=0; i<count; i++)
{
    printf ("%d\n", result[i]);
}
for (i=0; i<count; i++)
{
    ans = ans + result[i];
}
printf ("Reversed number is %d", ans);
return 0;
}
void push (int item)
{
    if (isFull())
    {
        printf ("\nStack is Full!!!\n");
        return;
    }
    // printf ("%d", item);
    top = top+1;
    stack[top] = item;
}
```

```
int pop()
{
  if(isEmpty)
  {
    printf("\n Stack is EMPTY!!!\n");
    return 0;
  }
  item = stack[top];
  top = top-1;
  return item;
}
  int isEmpty()
  {
      if (top==-1)
          return 1;
      else
          return 0;
  }
  int isFull()
  {
      if (top==MAX-1)
          return 1;
      else
          return 0;
  }
```

```
Input a number: 538247
Reversed number is 742835
Process returned 0 (0x0)   execution time : 12.881 s
Press any key to continue.
```

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

typedef struct list
{
        int data;
        struct list *next;
}node;
void display (node *temp)
{
        node *temp1 = temp;
        printf ("\n Now the list is :\n%d ->", temp->data);
        temp = temp->next;
        while (temp != temp1)
        {
            printf ("%d ->", temp->data);
            temp = temp->next;
        }
        printf ("%d \n", temp1->data);
}
int main (){
        node *head = NULL, *temp, *temp1, *temp2, *temp3;
        int choice, key;
        do
        {
            temp = (node *) malloc (sizeof (node));
            if (temp != NULL)
            {
                printf ("\n Enter the element in the list :");
                scanf ("%d", &temp->data);
```

```
if(head == NULL)
{
        head = temp;
}
else
{
    temp1 = head;
    while (temp1 -> next != head)
    {
            temp1 = temp1 -> next;
    }
    temp1 -> next = temp;
}
temp -> next = head;
}
else
{
  printf ("\n Memory not available... node allocation is not
           possible");
}
printf ("\n If you wish to add more data on the list enter 1:");
scanf ("%d", &choice);
}while (choice == 1);
display (head);
printf ("\n Enter the data of the node which you want to exchange
        with its next :");
scanf ("%d", &key);
temp = head; // Getting the last node in list
```

```
while (temp -> ment != head)
{
      temp = temp -> ment ;
}
temp3 = temp ;
temp = head ; // searching for key data in list
while (temp -> data != key)
{
   temp1 = temp ;
   temp = temp -> ment ;
}
if (temp == head)
{
        if (head -> ment -> ment == head)
        {
                temp = head ;
                head = head -> ment ;
                head -> ment = temp ;
        }
        else {
                temp2 = head -> ment -> ment ;
                temp = head ;
                head = head -> ment ;
                head -> ment = temp ;
                temp -> ment = temp2 ;
                temp3 -> ment = head ;
        }
}
else
{
        if (temp -> ment != head) {
```

```
        temp3 = temp -> next -> next;
        temp1 -> next = temp -> next;
        temp1 -> next -> next = temp;
        temp -> next = temp3;
    }
    else if (temp -> next == head) {
        temp3 = head -> next;
        temp2 = head;

        temp1 -> next = temp2;
        temp2 -> next = temp;
        temp -> next = temp3;
        head = temp;

    }
}
display(head);
return 0;
}
```

```
Enter the element in the list : 1

If you wish to add more data on the list enter 1 : 1

Enter the element in the list : 2

If you wish to add more data on the list enter 1 : 1

Enter the element in the list : 3

If you wish to add more data on the list enter 1 : 4

Now the list is :
1->2->3->1

Enter the data of the node which you want to exchange with it's next : 2

Now the list is :
1->3->2->1

--------------------------------
Process exited after 13.78 seconds with return value 0
Press any key to continue . . .
```

```
// Recursive C program for level
// order traversal of Binary Tree
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data,
pointer to left child
and a pointer to right child */
struct node
{
        int data;
        struct node* left, *right;
};

typedef struct lnode {
    int data;
    struct lnodes *next;
} NODE;

typedef struct list
{
    NODE *head;
    int count;
} LIST;
NODE * getnode (int element)
{
        NODE * newnode;
        newnode = (NODE *) malloc (sizeof (NODE));
        newnode -> data = element;
        newnode -> next = NULL;
        return newnode;
}
```

```
void insertlist (LIST *lp, int ele) // inserting items to list
                                    from level order traversal
{
    NODE * newnode , * temp , *prev;
    newnode = getnode (ele);
if ( lp -> head == NULL) // empty list
   {
     lp -> head = newnode;
           lp -> count ++ ;
           return ;
    } else {
           temp = lp -> head ;
           while ( temp) {
                  prev = temp ;
                  temp = temp -> next ;
                 }
                 prev -> next = newnode;
                 lp -> count ++;
             }
 }
void display (LIST *lp)
{
    NODE * temp ;
    temp = lp -> head ;
    if ( temp = = NULL)
    {
          printf ("\n Stack is empty !! ");
          return ;
    }
    printf ("NULL -> ");
    while (temp)
    {
          printf ("%d -> " , temp -> data);
          temp = temp -> next ;
      }
```

```
    printf ("NULL");
}

/* Function prototype */
void printGivenLevel (struct node* root, int level, List *lp);
int height (struct node* node);
struct node* newNode (int data);

/* Function to print level order traversal a tree*/
void printLevelOrder (struct node* root, LIST *lp)
{
        int h = height (root);
        int i, j;
        for (i=1; i<=h; i++)
            printGivenLevel (root, i, lp);
    }
/* Print nodes at a given level */
void printGivenLevel (struct node* root, int level, LIST *lp)
{
        if (root == NULL)
            return;
        if (level == 1)
        {
            insertList (lp, root->data);
        }
        else if (level > 1)
        {
            printGivenLevel (root->left, level-1, lp);
            printGivenLevel (root->right, level-1, lp);
        }
}
```

```
/* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node. */
int height (struct node* node)
{
    if (node == NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height (node -> left);
        int rheight = height (node -> right);

        /* use the larger one */
        if (lheight > rheight)
            return (lheight + 1);
        else return (rheight + 1);
        }
}
/* Helper function that allocates a new node with the given data
    and NULL left and right pointers. */
struct node * newNode (int data)
{
        struct node* node = (struct node*)
        malloc (sizeof (struct node));
        node -> data = data;
        node -> left = NULL;
        node -> right = NULL;

        return (node);
}
```

```
/* Driver program to test above functions */
int main()
{
    struct node * root = newNode(2);
    LIST lp;
    lp.count = 0;
    lp.head = NULL;
    root -> left       = newNode(7);
    root -> right      = newNode(5);
    root -> left -> left  = newNode(2);
    root -> left -> right = newNode(6);

    printf("Level order traversal of binary tree stored in
    singly linked list is :: \n");
    printLevelorder (root, &lp);
    return 0;
}
```

```
Level Order traversal of binary tree stored in singly linked list is ::
NULL->2->7->5->2->6->NULL
--------------------------------
Process exited after 0.028 seconds with return value 0
Press any key to continue . . .
```