VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Data Structures using C Lab

(23CS3PCDST)

Submitted by

SUJAN G E(1BM23CS347)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Data Structures using C Lab (23CS3PCDST)"carried out by **SUJAN G E (1BM23CS347),** who is bonafide student of **B.M.S.College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering inComputer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academicrequirements in respect of Data Structures using C Lab (23CS3PCDST) work prescribed for the said degree.

Namratha M Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor &HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Sujan279/DS

Program 1

Write a program to simulate the working of stack using an array with the following:
a) Push
b) Pop
c) Display
The program should print appropriate messages for stack overflow, stack underflow

OBSERVATION:

D.S. 30/7/24

Stack operation:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
Struct stack
{
  int arr[20];
  int top;
  int size;
};
bool isempty (struct stack * sp)
{
 if (sp->top == -1)
 {
  return true;
 }
 return false;
}

bool is_Full (struct stack * sp)
{
 if (sp->top == sp->size-1)
 {
  return true;
 }
  return false;
 }

void push (struct stack * sp, int val)
{
 if (! is full (sp))
 {
  sp->arr(top) = val;
  sp->top++;
  sp->arr[sp->top] = val;
 }
 else { printf ("stack overflow"); }
int pop (struct stack * sp)
{ if (! is Empty (sp)) }
{
 int data = sp->arr[top];
    sp->top--;
    return data;
 }
 else ("stack underflow");
}
```

```
void init_stack (struct stack * sp, int size)
{
 sp->top = -1;
 sp->size = size;
}
```

```
int peek (struct stack * sp);
{
  return sp -> arr[sp->top];

void traverse (struct stack * sp)
{
 for (int i = sp -> size - 1; i >= 0; i - -)
 {
  Printf ( "%d\n", sp -> arr[i]);
 }
}

void main()
{
   struct stack * sp;
   sp = (struct stack*) malloc (sizeof (struct stack));
    push (sp, 1);
    push (sp, 2);
    pop (sp);
    peek (sp);
    traverse (sp);
}
```

OUTPUT:

Popped value : 2

Top value : 1

Stack elements : 1

OUTPUT:

2 → 3

1 → 1

operation:

pop(v);

pop(v);

pop(v); underflow → Peek(v) → o/p empty stack

push(v, 30);

push(v, 40);

push(v, 50);

push(v, 60); → overflow

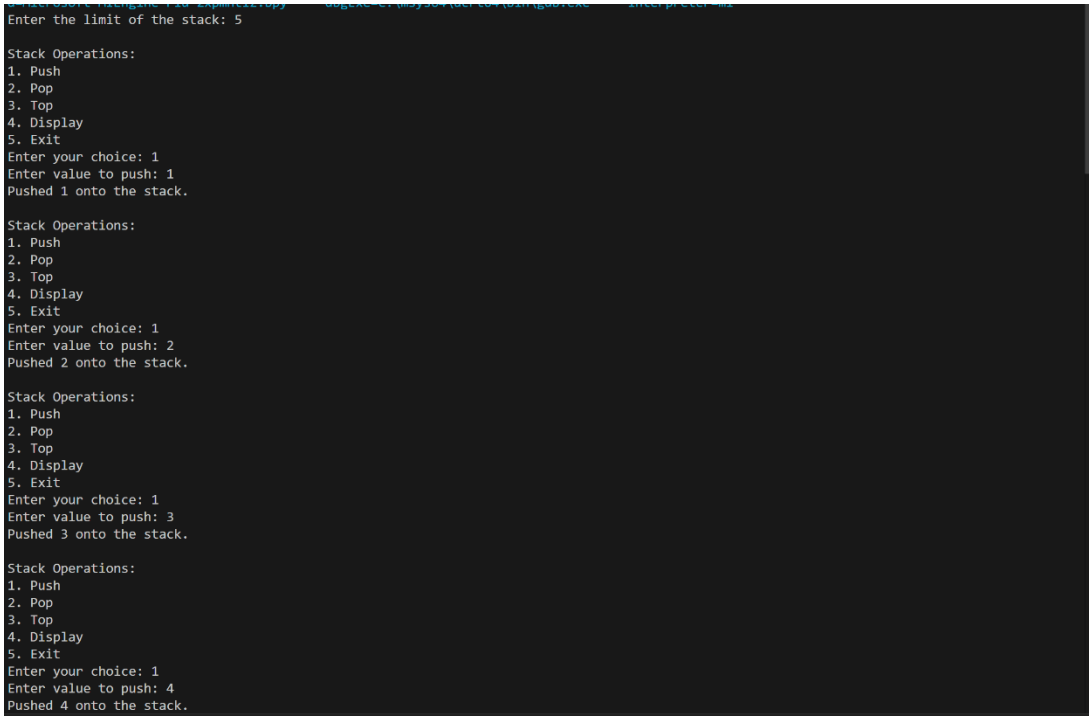
Peek(v) : → 60

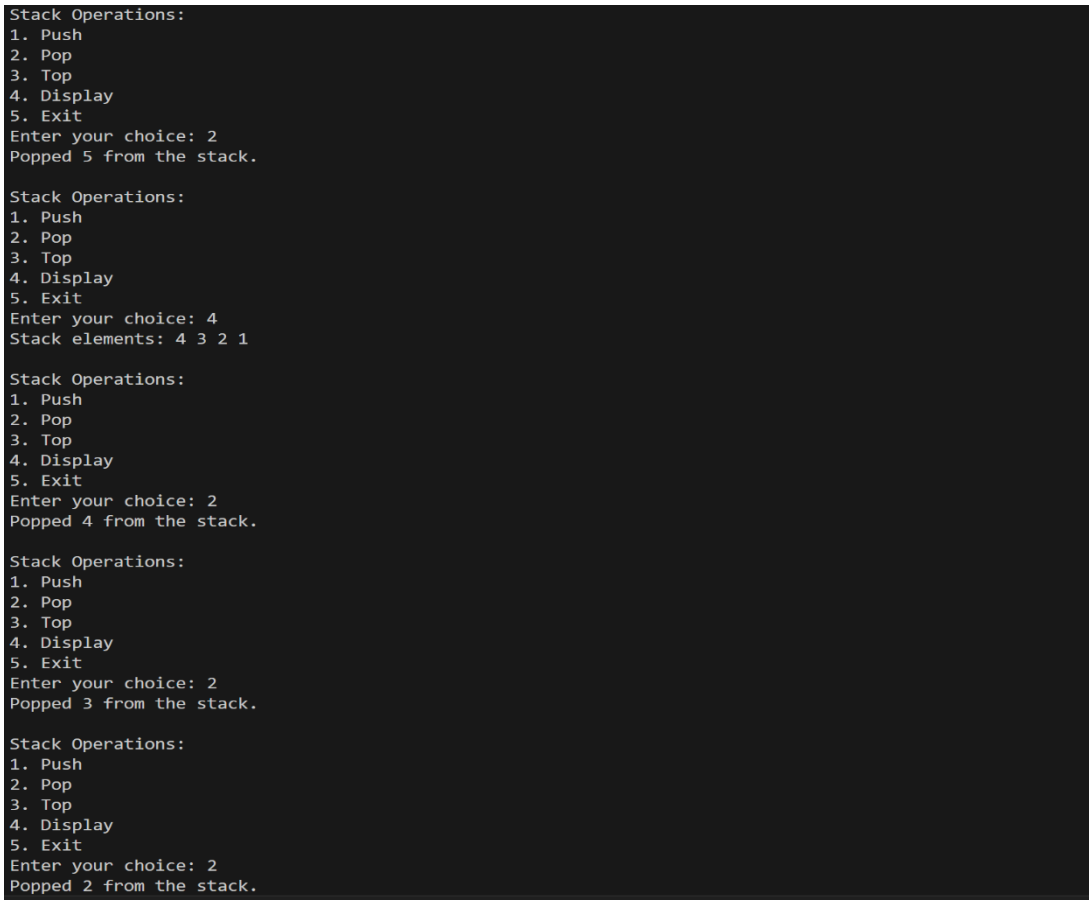
CODE :

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

int stack[MAX];
int top = -1;

void push(int element) {
    if (top == MAX - 1) {
        printf("Stack Overflow! Unable to push %d\n", element);
    } else {
        stack[++top] = element;
        printf("Pushed: %d\n", element);
    }
}

void pop() {
    if (top == -1) {
        printf("Stack Underflow! The stack is empty.\n");
    } else {
        printf("Popped: %d\n", stack[top--]);
    }
}

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

int main() {
    int choice, element;
```

```
    while (1) {
        printf("\n--- Stack Operations ---\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element to push: ");
                scanf("%d", &element);
                push(element);
                break;

            case 2:
                pop();
                break;

            case 3:
                display();
                break;

            case 4:
                printf("Exiting program.\n");
                return 0;

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
}
```

OUTPUT:

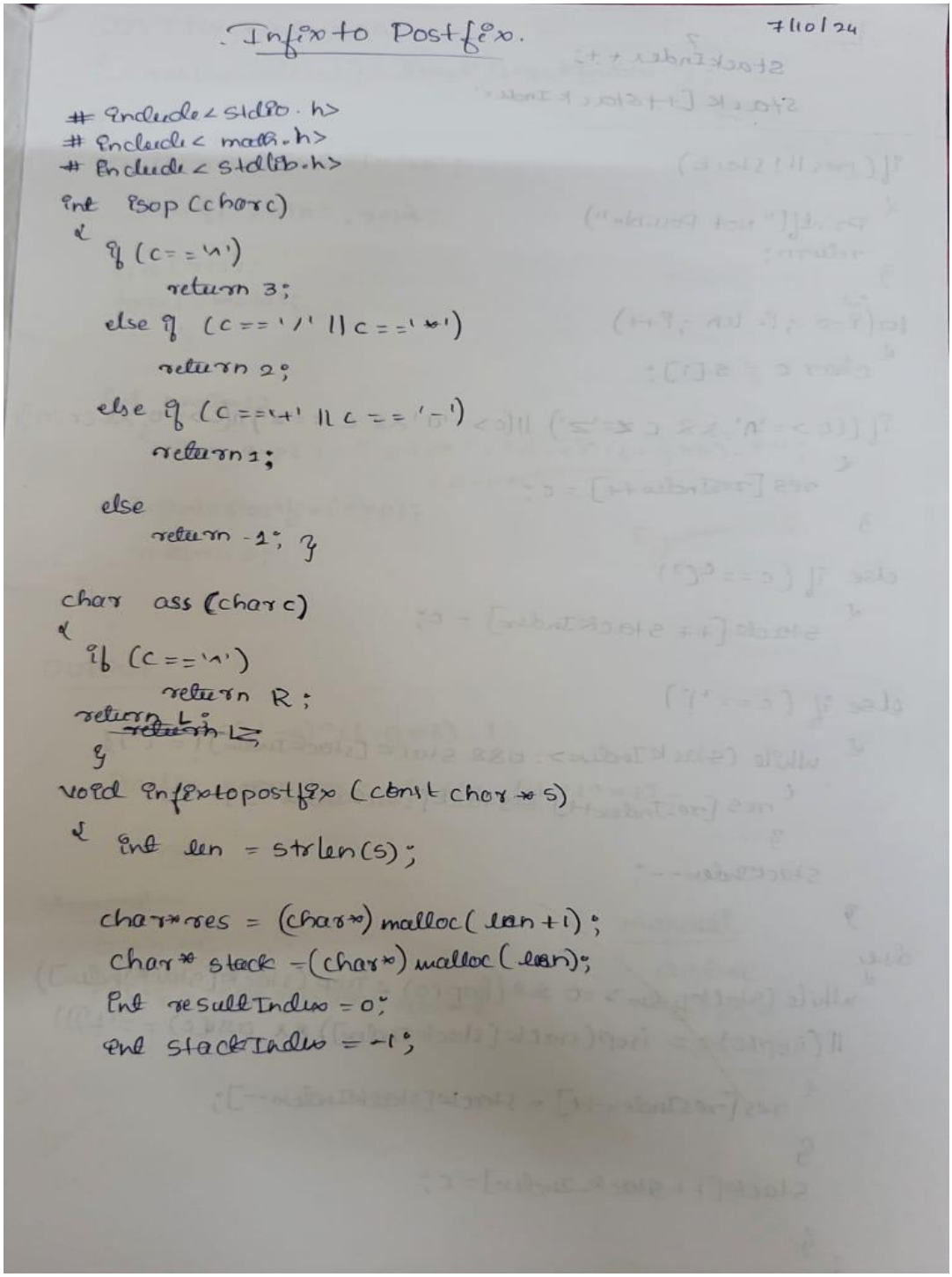

Enter the limit of the stack: 5
Stack Operations:
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 1
Pushed 1 onto the stack.
Stack Operations:
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 2
Pushed 2 onto the stack.
Stack Operations:
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 3
Pushed 3 onto the stack.
Stack Operations:
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 4
Pushed 4 onto the stack.


Stack Operations:
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 2
Popped 5 from the stack.
Stack Operations:
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 4
Stack elements: 4 3 2 1
Stack Operations:
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 2
Popped 4 from the stack.
Stack Operations:
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 2
Popped 3 from the stack.
Stack Operations:
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 2
Popped 2 from the stack.

Program 2

WAP to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operators + (plus), - (minus), * (multiply) and / (divide)

OBSERVATION :



Infix to Postfix. 7/10/24

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
int isop(char c)
{
   if(c=='^')
      return 3;
   else if (c=='/'||c=='*')
      return 2;
   else if (c=='+'||c=='-')
      return 1;
   else
      return -1; }
char ass(char c)
{
  if (c=='^')
      return R;
 return L;
 }
void infixtopostfix (const char *s)
{  int len = strlen(s);

   char*res = (char*) malloc(len+1);
   char* stack =(char*) malloc(len);
   int resultIndex = 0;
   int stackIndex = -1;
```

```
stackIndex++;
stack[++stackIndex]

if(res||!stack)
{
  Printf("Not Possible")
  return;
}
for(int i=0 ;i< len ;i++)
{
  char c = s[i];
  if((c>='A' && c<='Z') ||(c>='a' && c<='z') ||(c>='0' && c<=9))
  {
    res[resIndex++] = c;
  }
  else if (c=='(')
  {
    stack[++stackIndex] = c;
  }
  else if (c==')')
  {
    while (stackIndex>=0 && stack[stackIndex]!='(')
    {
      res[resIndex++] = stack[stackIndex--];
    }
    stackIndex--;
  }
  else
  {
    while (stackIndex>=0 &&(isop(c) < isop(stack[stackIndex])
    ||(isop(c)== isop(stack[stackIndex]) && assoc(c)=='L')))
    {
      res[resIndex++] = stack[stackIndex--];
    }
    stack[++stack_Index] = c;
  }
}
```

```
while (stackIndex >= 0)
{   res[resIndex++] = stack[stackIndex--];
}
result[resultIndex] = '\0';
   Printf("%s\n", res);
   free(res);
   free(stack);
}

int main()
{   char exp[] = "a+b*(c^d-e)^(f+g*h)-i";
                  (a+b*c)
   infixtoPostfix(exp);
   return 0;
}
```

OUTPUT:

a+b*(c^d-e)^(f+g*h)-i

Postfix expression : abcd^e-fgh*+^*+i-

(or)

~~Manual:~~

a+b*c

Postfix expression: abc*+

Manual

a+b*c

a+bc*

abc*+

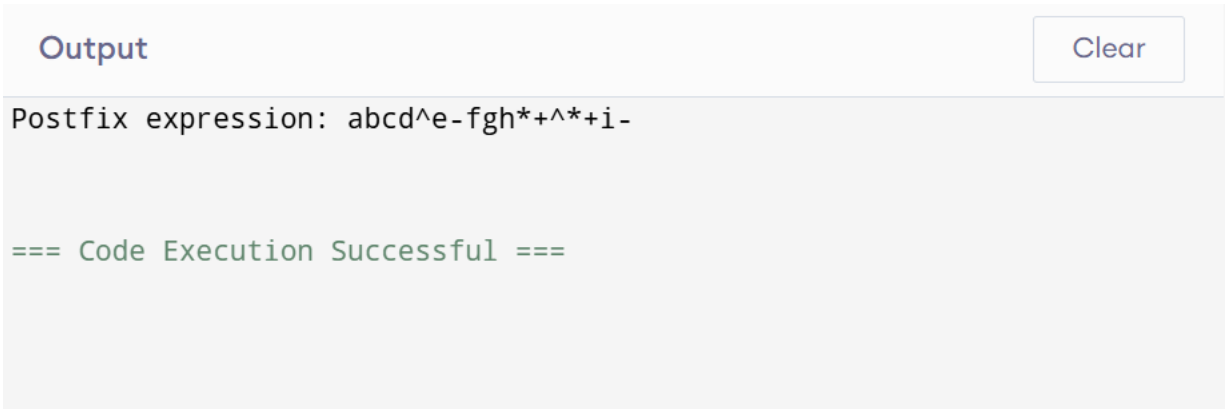
14/10

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int precedence(char c) {
    if (c == '^') return 3;
    else if (c == '*' || c == '/') return 2;
    else if (c == '+' || c == '-') return 1;
    else return -1;
}

char associativity(char c) {
    if (c == '^') return 'R';  // Right-to-left associativity
    return 'L';  // Left-to-right associativity
}

void infixToPostfix(const char *expr) {
    int len = strlen(expr);

        char *result = (char *)malloc(len + 1);
    char *stack = (char *)malloc(len);
    int resultIndex = 0;
    int stackIndex = -1;

    if (!result || !stack) {
        printf("Memory allocation failed\n");
        return;
    }

    for (int i = 0; i < len; i++) {
        char c = expr[i];

            if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
            result[resultIndex++] = c;
        }
            else if (c == '(') {
            stack[++stackIndex] = c;
        }
```

```
        else if (c == ')') {
            while (stackIndex >= 0 && stack[stackIndex] != '(') {
                result[resultIndex++] = stack[stackIndex--];
            }
            stackIndex--;  // Pop the '(' from the stack
        }
                else {
            while (stackIndex >= 0 && precedence(c) <= precedence(stack[stackIndex])) {
                if (precedence(c) == precedence(stack[stackIndex]) && associativity(c) ==
'R') break;
                result[resultIndex++] = stack[stackIndex--];
            }
            stack[++stackIndex] = c;
        }
    }


    while (stackIndex >= 0) {
        result[resultIndex++] = stack[stackIndex--];
    }

    result[resultIndex] = '\0';  // Null-terminate the result
    printf("Postfix expression: %s\n", result);


    free(result);
    free(stack);
}

int main() {
    char expr[] = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(expr);
    return 0;
}
```

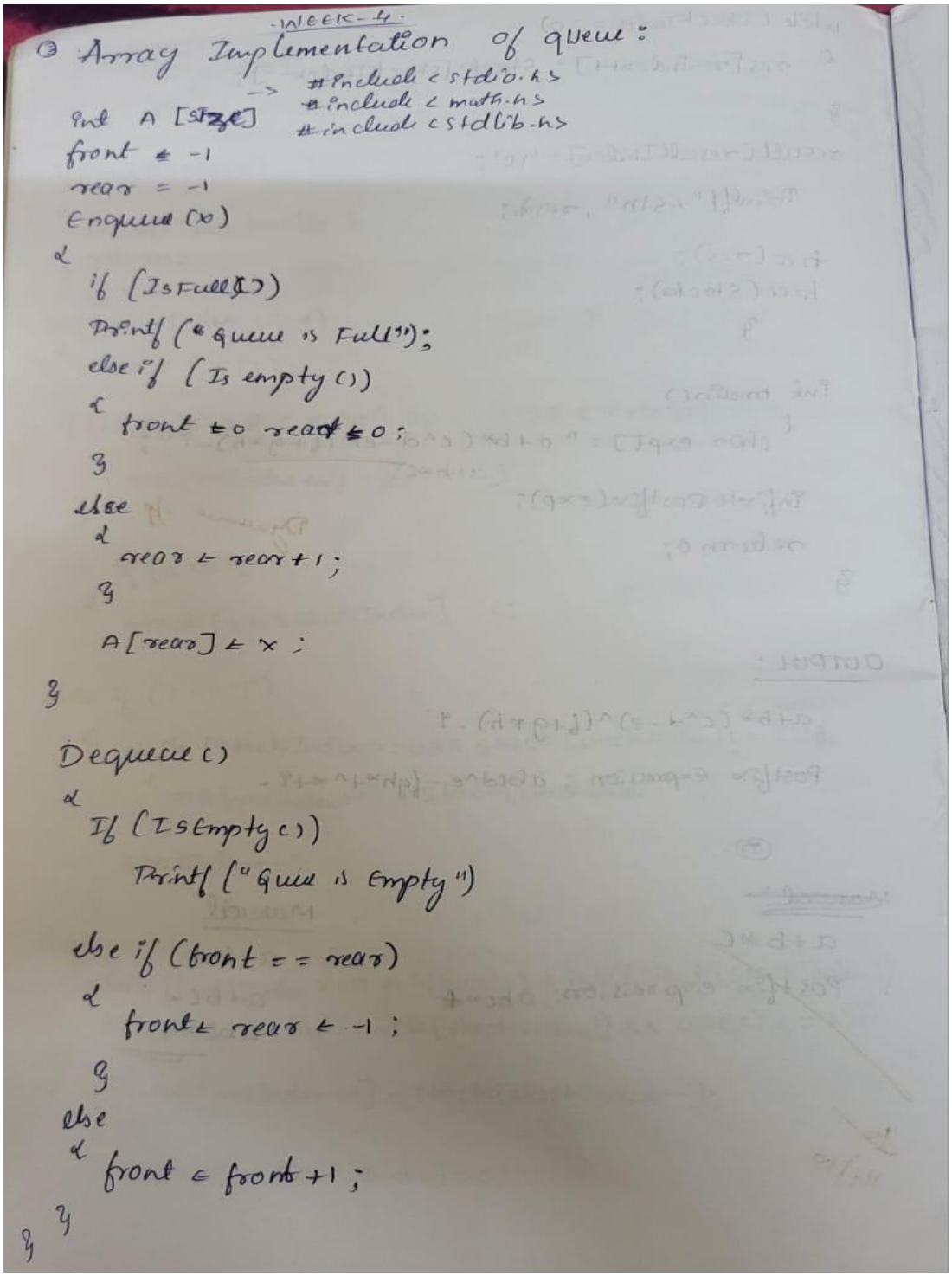
OUTPUT :


Output
Clear
Postfix expression: abcd^e-fgh*+^*+i-
=== Code Execution Successful ===

Program 3

a) WAP to simulate the working of a queue of integers using an array. Provide the following operations: Insert, Delete, Display The program should print appropriate messages for queue empty and queue overflow conditions

OBSERVATION :



```
-WEEK-4-
3 Array Implementation of queue:
                -> #include <stdio.h>
                   #include <math.h>
                   #include <stdlib.h>
int A [SIZE]
front = -1
rear = -1
Enqueue (x)
{
  if (IsFull())
  Printf ("Queue is Full");
  else if (Is empty ())
  {
    front = 0 rear = 0;
  }
  else
  {
    rear = rear + 1;
  }
  A[rear] = x;
}

Dequeue ()
{
  If (Isempty ())
     Printf ("Queue is Empty")

  else if (front == rear)
  {
    front = rear = -1;
  }
  else
  {
    front = front + 1;
  }
}
```

```
IsFull()
{
  if (rear == SIZE-1)
    return True;
  else
    return False;
}

IsEmpty()
{
  if (front==-1 && rear == -1)
    return True;
  else
    return False;
}

void Display()
{
  If (isempty())
  {
    Printf("Queue is empty");
    return;
  }
  Printf("Queue elements");
  for(i=0; i<rear; i++)
  {
    Printf("%d", A[i]);
  }
  Printf("\n");
}
```

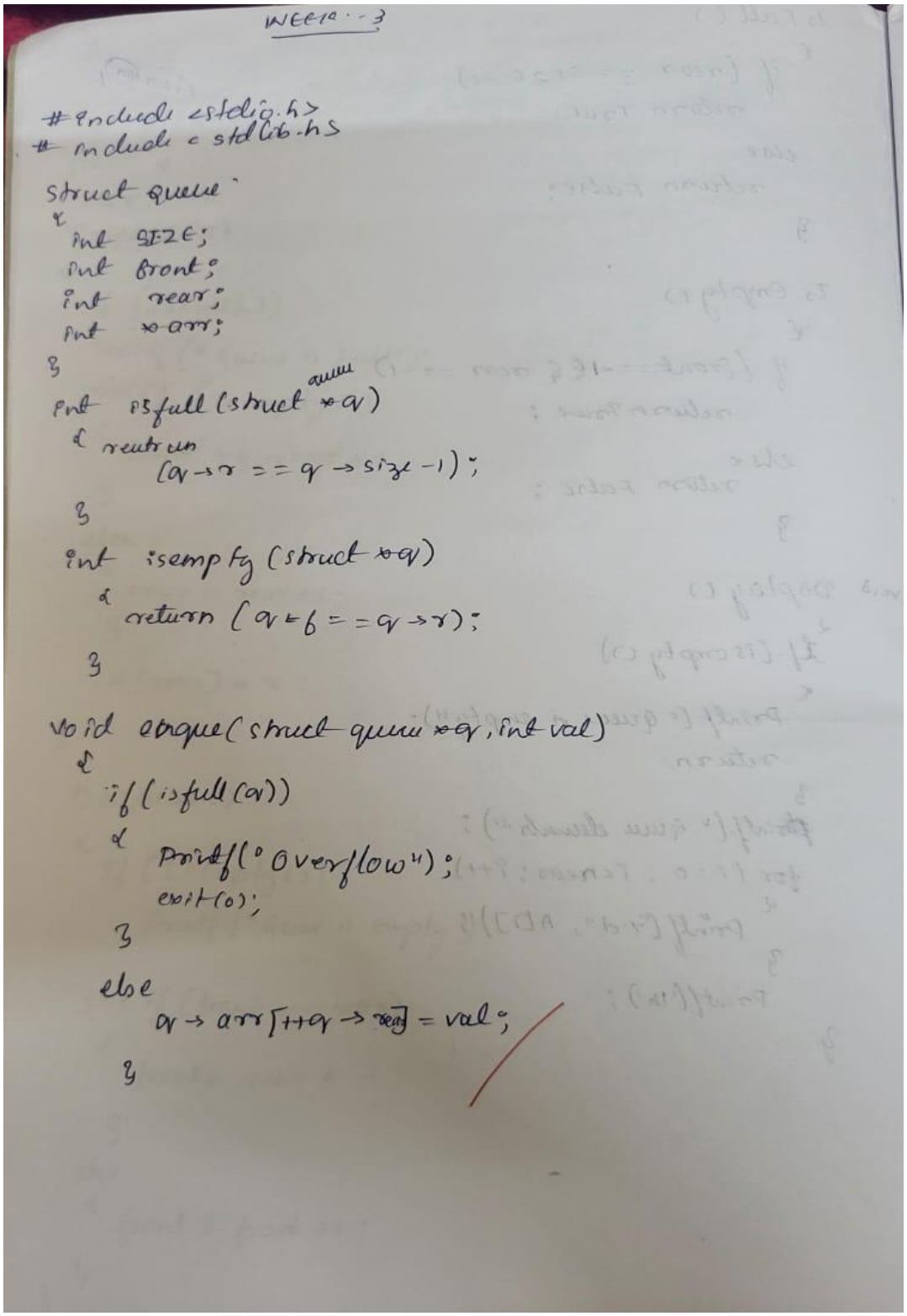
```
#include <stdio.h>
#define SIZE 5

int queue[SIZE];
int front = -1, rear = -1;

void insert(int value) {
   if (rear == SIZE - 1) {
      printf("Queue Overflow\n");
      return;
   }
   if (front == -1) front = 0;
   queue[++rear] = value;
}

void delete() {
   if (front == -1 || front > rear) {
      printf("Queue Underflow\n");
      return;
   }
   front++;
}

void display() {
   if (front == -1 || front > rear) {
      printf("Queue is Empty\n");
      return;
   }
   for (int i = front; i <= rear; i++) {
      printf("%d ", queue[i]);
   }
   printf("\n");
}

int main() {
   insert(10);
   insert(20);
   insert(30);
   display();
   delete();
```

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

OUTPUT:

Program 3

b) WAP to simulate the working of a circular queue of integers using an array. Provide the following operations: Insert, Delete & Display The program should print appropriate messages for queue empty and queue overflow conditions



```
WEEK - 3

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

struct queue
{
  int SIZE;
  int front;
  int rear;
  int *arr;
};

int isfull (struct queue *q)
{
  return
      (q->r == q->size - 1);
}

int isempty (struct *q)
{
  return (q->f == q->r);
}

void enque(struct queue *q, int val)
{
  if (isfull(q))
  {
    Printf("Overflow");
    exit(0);
  }
  else
    q->arr[++q->rear] = val;
}
```

```
int dequeue (struct queue * qv)
{
  if (isempty(qv))
  {
    Printf(" underflow");
    exit(0);
  }
  else {
  {
    int a = qv->arr[qv->front];
  }
  return a;
  }
void display (struct queue *qv)
{
  if (!isempty(qv)
  {
    for (i = qv->front; i<=qv->rear; i++)
    {
      Printf(" element %d is %d", i, qv->arr[i]);
    }
  }
else
{
  Printf(" underflow");
  exit(0);
}
}
```

14/10

```
int main()
{
 struct queue q; int ch;
 q.size =10;
 q. front =-1
 q. rear = -1;
 q.arr = (int*) malloc (q.size*sizeof(int));
 Printf(" Enter choice");
Printf("1. Enter at rear 2. Delete at front 3. Display");
Scanf("%d",&ch);

switch(ch)
 {
    case 1 : Printf("Enter element")
             scanf("%d", &val);
             enqueue (&q, val);
             break;

    case 2 : dequeue (&q);
             Printf("Poped %d", dequeue (&q);
              break;
    case 3 : display(&q);
              break;

   default : Printf("Invalid input");
 }
}
```

output:

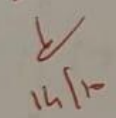
MENU:
1. Enqueue
2. Dequeue
3. Display
4. Exit

Enter choice : 1
Enter element : 10

Enter choice : 1
Enter element : 20

Enter choice : 1
Enter element : 30

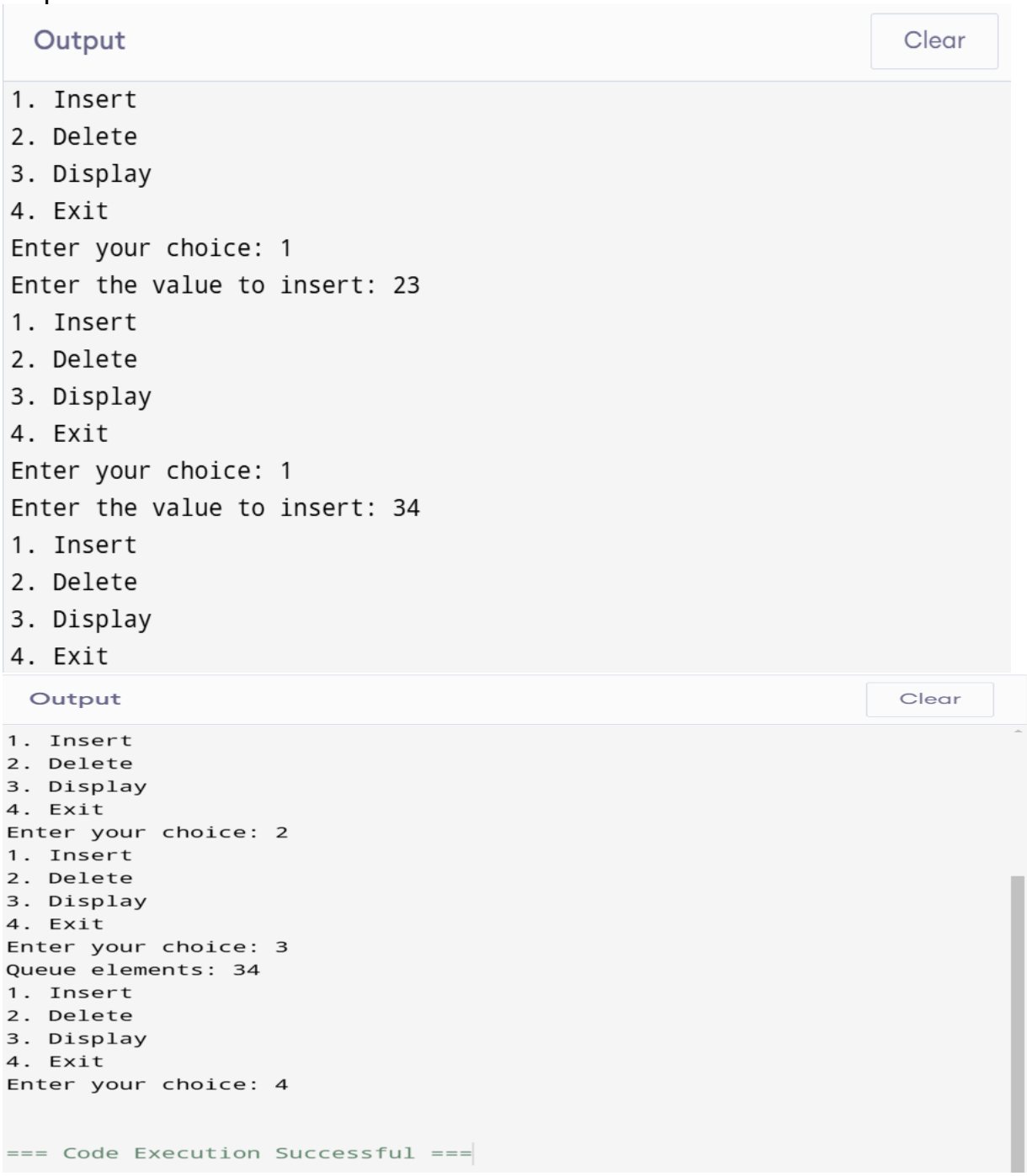
Enter choice : 1
Enter element : 40



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

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

void insert(int value) {
   if ((front == 0 && rear == MAX - 1) || (rear == (front - 1) % (MAX - 1))) {
      printf("Queue Overflow\n");
      return;
   }
   if (front == -1) {
      front = rear = 0;
   } else if (rear == MAX - 1 && front != 0) {
      rear = 0;
   } else {
      rear++;
   }
   queue[rear] = value;
}

void delete() {
   if (front == -1) {
      printf("Queue Underflow\n");
      return;
   }
   if (front == rear) {
      front = rear = -1;
   } else if (front == MAX - 1) {
      front = 0;
   } else {
      front++;
   }
}

void display() {
   if (front == -1) {
      printf("Queue is Empty\n");
      return;
   }
```

```
    if (rear >= front) {
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
    } else {
        for (int i = front; i < MAX; i++) {
            printf("%d ", queue[i]);
        }
        for (int i = 0; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
    }
    printf("\n");
}

int main() {
    int choice, value;
    do {
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the value to insert: ");
                scanf("%d", &value);
                insert(value);
                break;
            case 2:
                delete();
                break;
            case 3:
                display();
                break;
            case 4:
                break;
            default:
                printf("Invalid choice\n");
        }
    } while (choice != 4);
    return 0;
```

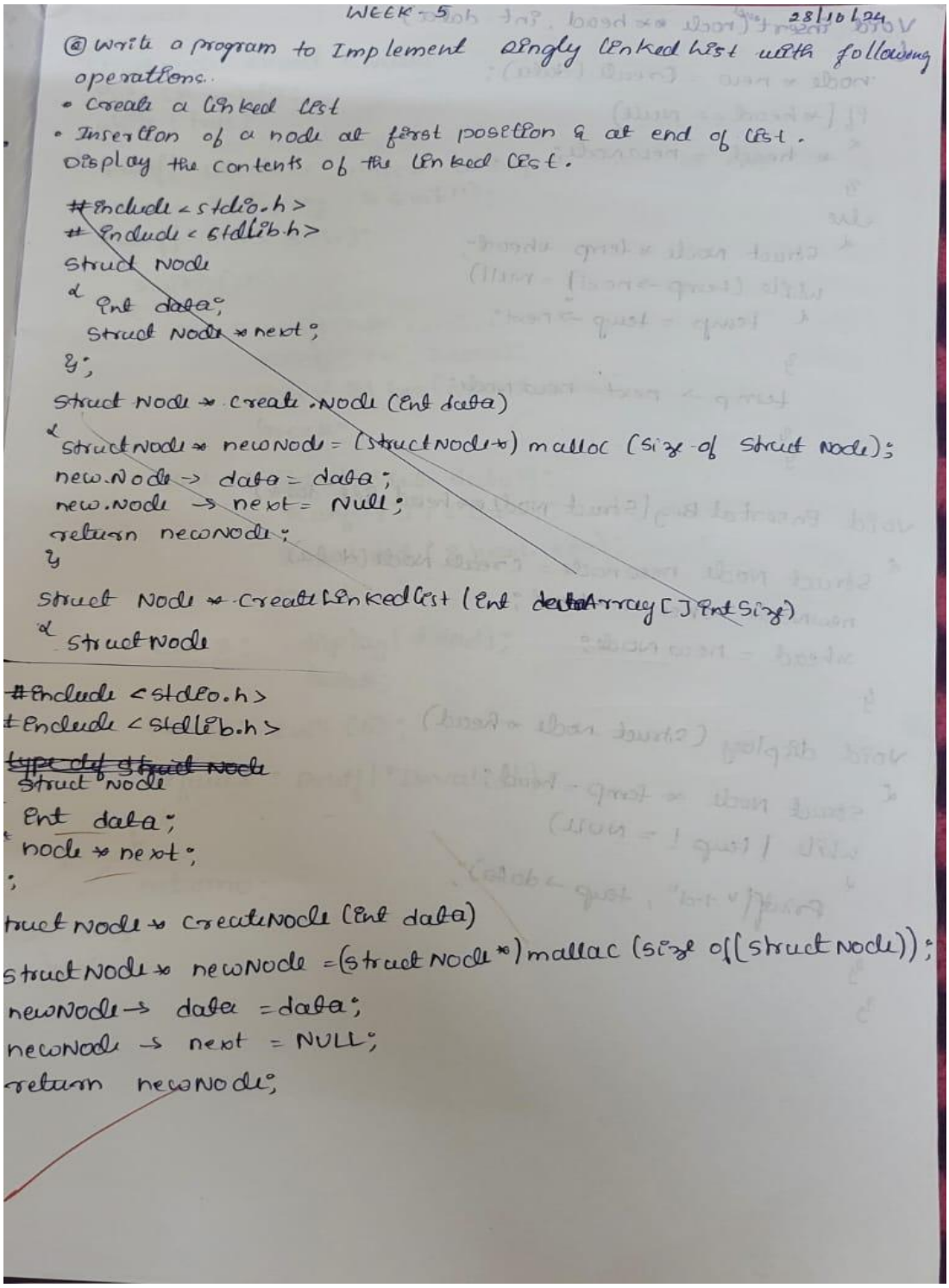
output :


Output
Clear
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the value to insert: 23
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the value to insert: 34
1. Insert
2. Delete
3. Display
4. Exit
Output
Clear
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements: 34
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4
=== Code Execution Successful ===

Program 4

WAP to Implement Singly Linked List with following operations a) Create a linked list. b) Deletion of first element, specified element and last element in the list. c) Display the contents of the linked list

OBSERVATION :



WEEK-5 28/10/24

@ Write a program to Implement Singly Linked List with following operations.

- Create a Linked List
- Insertion of a node at first position & at end of List.

Display the contents of the Linked List.

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
  int data;
  struct Node * next;
};
struct Node * createNode (int data)
{
 struct Node * newNode = (struct Node *) malloc (size of struct Node);
 newNode -> data = data;
 newNode -> next = Null;
 return newNode;
}
struct Node * CreateLinkedList (int dataArray[] int Size)
{
 struct Node
```

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
 struct Node
 int data;
 node * next;
;
truct Node * createNode (int data)
struct Node * newNode = (struct Node *) malloc (size of([struct Node]));
newNode -> data = data;
newNode -> next = NULL;
return newNode;
```

```
void insert(node **head, int data)
{
  node *new = Create(data);
  if (*head == NULL)
  {
    *head = newNode;
  }
  else
  {
    Struct Node *temp = *head;
    while (temp->next) = NULL)
    {
      temp = temp->next;
    }
    temp -> next = new Node;
  }
}

void InsertatBeg(Struct Node **head, int data)
{
  Struct Node newnode = Create Node(data)
  newNode->next = *head;
  *head = new Node;
}

Void display (Struct Node *head)
{
  Struct Node *temp = head;
  while (temp != NULL)
  {
    Printf("%d", temp->data);
  }
}
```

```
int main()
{
struct Node *head = Null;
int ch, data;
while (true)
{ Printf("\n Enter choice 1.insert at beg 2.Insert at end
        3. display , 4 exit");
Scanf("%d", &ch);
  switch(ch)
  {
    case 1: printf("Enter data");
            scanf("%d", &data);
            Insert at beg(&head, data);
            break;

    case 2: Printf("Enter data");
            scanf("%d", &data);
            Insert end (&head, data);
            break;

    case 3: display(head);
            break;

    case 4: exit(0);

    default: Printf("Invalid");
  }

  return 0;
}
```

Proceed

OUTPUT:

```
Choose an option:
1. Insert at end
2. Insert at begg
3. Display
4. Exit

Enter your choice: 1
Enter data : 10
Enter your choice : 2
Enter data : 20
Enter your choice : 2
Enter data : 33
Enter your choice : 3
current list: 33 10 20
Enter your choice: 4
Exit
```

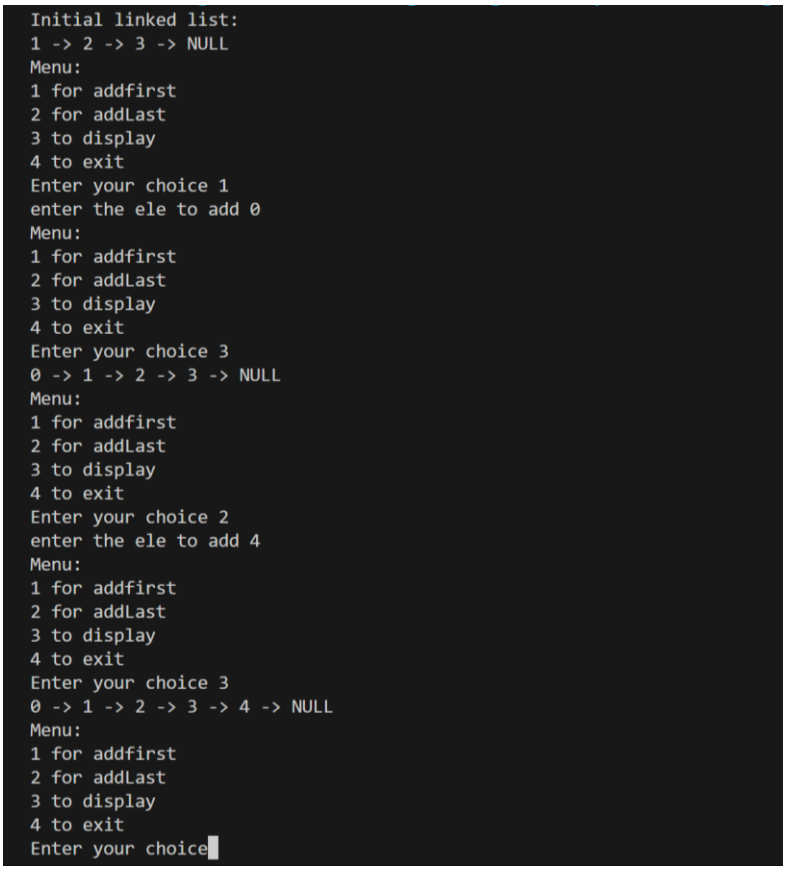
O/P scan

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

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

struct Node* head = NULL;

void createList(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   if (head == NULL) {
      head = newNode;
   } else {
      struct Node* temp = head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
   }
}

void insertFirst(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = head;
   head = newNode;
}

void insertAtPosition(int data, int position) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   if (position == 1) {
      newNode->next = head;
      head = newNode;
      return;
   }
```

```
    struct Node* temp = head;
    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range\n");
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}

void insertLast(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void displayList() {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, data, position;

    while (1) {
```

```
    printf("1. Create List\n");
    printf("2. Insert at First Position\n");
    printf("3. Insert at Any Position\n");
    printf("4. Insert at Last Position\n");
    printf("5. Display List\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter data to create list: ");
            scanf("%d", &data);
            createList(data);
            break;
        case 2:
            printf("Enter data to insert at first position: ");
            scanf("%d", &data);
            insertFirst(data);
            break;
        case 3:
            printf("Enter data and position to insert: ");
            scanf("%d %d", &data, &position);
            insertAtPosition(data, position);
            break;
        case 4:
            printf("Enter data to insert at last position: ");
            scanf("%d", &data);
            insertLast(data);
            break;
        case 5:
            displayList();
            break;
        case 6:
            exit(0);
        default:
            printf("Invalid choice\n");
    }
}
```

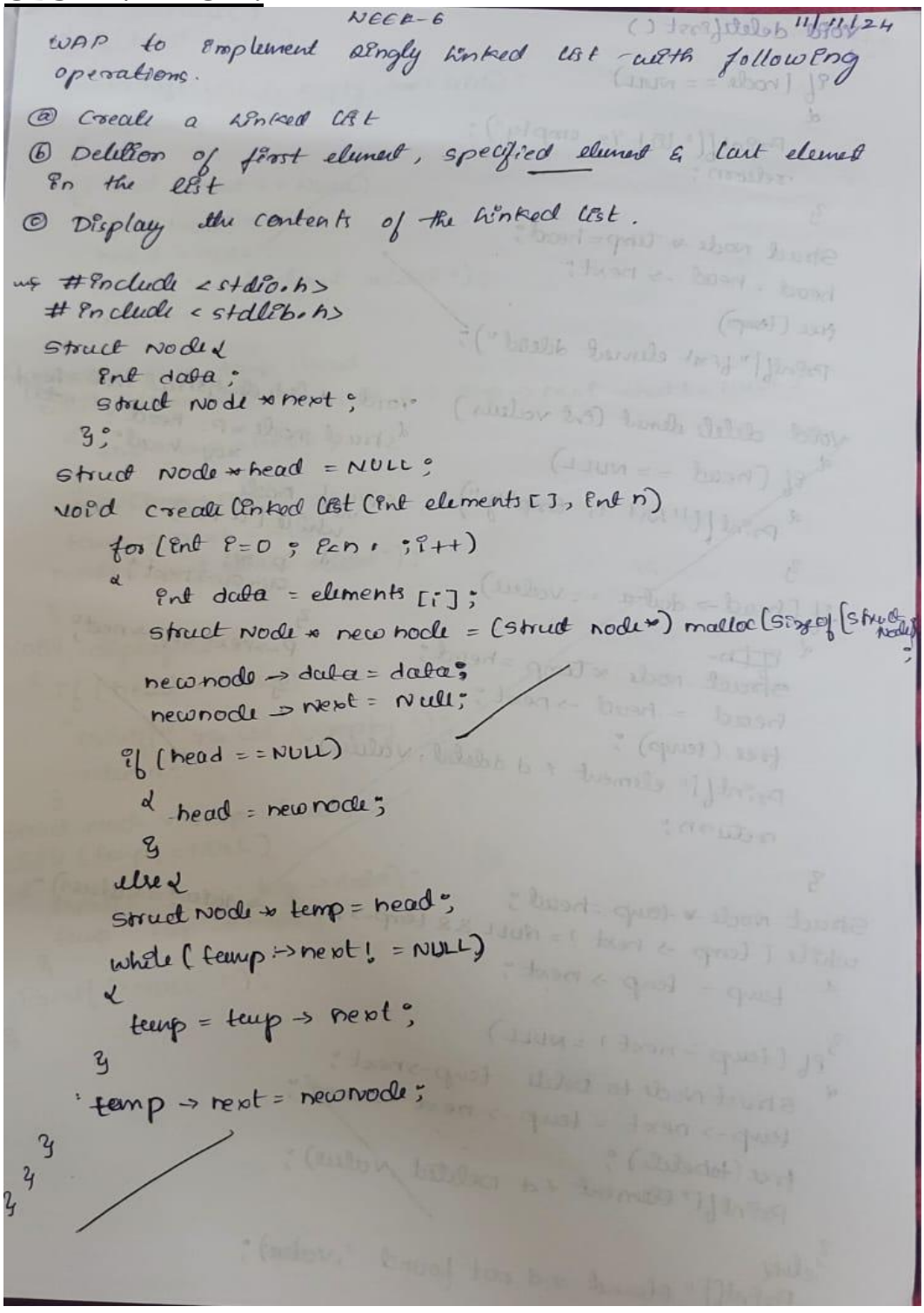
OUTPUT :


Initial linked list:
1 -> 2 -> 3 -> NULL
Menu:
1 for addfirst
2 for addLast
3 to display
4 to exit
Enter your choice 1
enter the ele to add 0
Menu:
1 for addfirst
2 for addLast
3 to display
4 to exit
Enter your choice 3
0 -> 1 -> 2 -> 3 -> NULL
Menu:
1 for addfirst
2 for addLast
3 to display
4 to exit
Enter your choice 2
enter the ele to add 4
Menu:
1 for addfirst
2 for addLast
3 to display
4 to exit
Enter your choice 3
0 -> 1 -> 2 -> 3 -> 4 -> NULL
Menu:
1 for addfirst
2 for addLast
3 to display
4 to exit
Enter your choice

Program 5

WAP to Implement Singly Linked List with following operations a) Create a linked list. b) Deletion of first element, specified element and last element in the list. c) Display the contents of the linked list.

OBSERVATION :



NEEA-6 11/11/24

WAP to Implement Singly linked list with following operations.

(a) Create a Linked List

(b) Deletion of first element, specified element & last element in the list

(c) Display the contents of the linked list.

```
#include <stdio.h>
#include <stdlib.h>
struct node{
    int data;
    struct node *next;
};
struct node *head = NULL;
void create(int elements[], int n)
    for(int i=0 ; i<n ; i++)
    {
        int data = elements[i];
        struct node * newnode = (struct node*) malloc(sizeof(struct node));
        newnode -> data = data;
        newnode -> next = NULL;
      if(head == NULL)
        { head = newnode;
        }
      else{
      struct node * temp = head;
      while(temp -> next != NULL)
      {
        temp = temp -> next;
    }
    temp -> next = newnode;
}
}
}
```

```
void deletefirst()
{
if (node == NULL)
{
    Printf(" list is empty");
    return;
}
Struct node * temp = head;
head = head -> next;
free (temp)
Printf(" first element deleted ");
```

```
void delete element (int values)
{
if (head == NULL)
{ Printf(" list is Empty");
}
if (head -> data == value)
{
    struct node * temp = head;
    head = head -> next;
    free (temp);
    Printf(" element %d deleted, values);
    return;
}
Struct node * temp = head;
while ( temp -> next != NULL && temp -> next -> data != values)
{ temp = temp -> next;
}
if (temp -> next != NULL)
{ Struct Node to Delete = temp -> next;
  temp -> next = temp -> next -> next;
  free (toDelete);
  Printf(" Element %d Deleted ", value);
}
else
  Printf(" Element %d not found ", values);
}
```

```
void delete element (struct node * head, int
{ struct node * p = head;
  * qv = head;
  struct node
  int i;
  while (i < pos)
  { p = qv;
    qv = qv -> next;
  }
  p -> next = qv -> next;
}
```

```
void deletlast()
{ if(head ==NULL)
  { Printf("List is empty, can't delete");
    return ;
  }
  if( head ->next = = NULL)
  { free(head);
    head = NULL;
    Printf(" Last elemnt deleted ");
    return;
  }
struct node * temp =head
while (temp ->next ! =NULL && temp -> next ->next ! = NULL)
  { temp = temp ->next;
  }
  free (temp-> next)
  temp -> next = NULL ;
  Printf(" Last element deleted");
}
void displaylist()
{ if (head ==NULL)
  { Printf(" The List is empty ");
    return;
  }
struct node * temp =head;
while (temp ! =NULL)
{ Printf("%d -> ", temp ->data);
  temp = temp -> next;
}
Printf ("NULL ");
}
```

```
int main()
{
  int ch, value, n;
  int elements[10];

  while (1)
  {
    Printf(" 1. Create Linked List");
    Printf(" 2. Delete the first element");
    Printf(" 3. Delete particular position element");
    Printf(" 4. Delete last element");
    Printf(" 5. Display the Linked List");
    Printf(" 6. Exit");
    Printf(" Enter your choice");
    Scanf(" %d", &ch);

switch(ch)
{ case 1: Printf(" Enter the no. of elements you want to insert");
          Scanf(" %d", &n);
          Printf(" Enter the elements");
          for(int i=0; i<n; i++)
          { Scanf(" %d", &elem[i]);
          }
          CreateLinkedList(elem, n);
          Printf(" Linked List Created")
          break;
  Case 2: deleteFirst();
          break;

  Case 3: Printf(" Enter position to delete")
          Scanf(" %d", &d);
          deleteElement(head, d);
          break;
```

```
case 4 : deleteCert();
         break;

case 5 : displayList();
         break;

case 6 :
         Printf("Exit");
         exit(0)

default :
         Printf("Invalid")
         return 0;
```

Proceed

Executed

OUTPUT:

MENU:
1. Create a linked list
2. Delete the first element
3. Delete an element by Position

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

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

struct Node* head = NULL;

void createList(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   if (head == NULL) {
      head = newNode;
   } else {
      struct Node* temp = head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
   }
}

void deleteFirst() {
   if (head != NULL) {
      struct Node* temp = head;
      head = head->next;
      free(temp);
   }
}

void deleteLast() {
   if (head != NULL) {
      if (head->next == NULL) {
         free(head);
         head = NULL;
      } else {
         struct Node* temp = head;
```

```
            while (temp->next != NULL && temp->next->next != NULL) {
                temp = temp->next;
            }
            free(temp->next);
            temp->next = NULL;
        }
    }
}

void deleteSpecified(int value) {
    if (head != NULL) {
        if (head->data == value) {
            struct Node* temp = head;
            head = head->next;
            free(temp);
        } else {
            struct Node* temp = head;
            while (temp->next != NULL && temp->next->data != value) {
                temp = temp->next;
            }
            if (temp->next != NULL) {
                struct Node* toDelete = temp->next;
                temp->next = temp->next->next;
                free(toDelete);
            }
        }
    }
}

void displayList() {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, data, value;
```

```
while (1) {
    printf("1. Create List\n");
    printf("2. Delete First Element\n");
    printf("3. Delete Last Element\n");
    printf("4. Delete Specified Element\n");
    printf("5. Display List\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter data to create list: ");
            scanf("%d", &data);
            createList(data);
            break;
        case 2:
            deleteFirst();
            printf("First element deleted.\n");
            break;
        case 3:
            deleteLast();
            printf("Last element deleted.\n");
            break;
        case 4:
            printf("Enter value to delete: ");
            scanf("%d", &value);
            deleteSpecified(value);
            printf("Specified element deleted.\n");
            break;
        case 5:
            displayList();
            break;
        case 6:
            exit(0);
        default:
            printf("Invalid choice\n");
    }
}
```

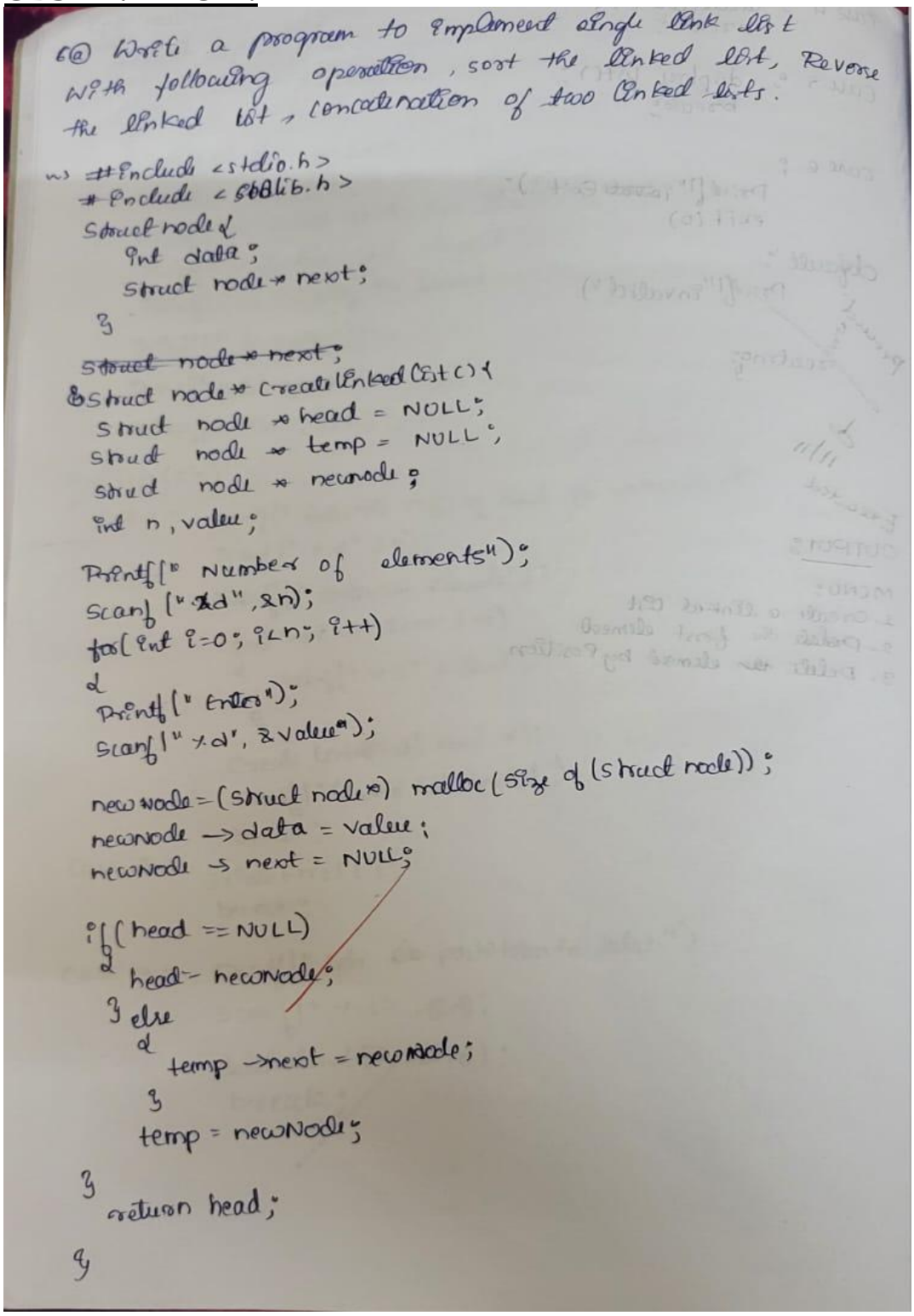
OUTPUT :

```
Initial linked list:
1 -> 2 -> 3 -> 4 -> 5 -> NULL
Menu
1. Delete first node
2. Delete last node
3. Delete specified node
4. Display list
5. Exit
Enter your choice: 1
After deleting the first node:
2 -> 3 -> 4 -> 5 -> NULL
Menu
1. Delete first node
2. Delete last node
3. Delete specified node
4. Display list
5. Exit
Enter your choice: 3
Enter the value to be deleted: 3
After deleting the specified node:
2 -> 4 -> 5 -> NULL
Menu
1. Delete first node
2. Delete last node
3. Delete specified node
4. Display list
5. Exit
Enter your choice: 2
After deleting the last node:
2 -> 4 -> NULL
Menu
1. Delete first node
2. Delete last node
3. Delete specified node
4. Display list
5. Exit
Enter your choice: 4
2 -> 4 -> NULL
```

Program 6

a) WAP to Implement Single Link List with following operations: Sort the linked list, Reverse the linked list, Concatenation of two linked lists

OBSERVATION :



6@ Write a program to implement single link list with following operation, sort the linked list, Reverse the linked list, concatenation of two linked lists.

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node * next;
};
struct node * next;
struct node * CreateLinkedList() {
    struct node * head = NULL;
    struct node * temp = NULL;
    struct node * newnode;
    int n, value;
    Printf("Number of elements");
    Scanf("%d", &n);
    for(int i=0; i<n; i++)
    {
        Printf("Enter");
        Scanf("%d", &value);
        newNode = (struct node*) malloc(sizeof(struct node));
        newNode -> data = value;
        newNode -> next = NULL;
        if(head == NULL)
        {
            head = newNode;
        } else
        {
            temp -> next = newNode;
        }
        temp = newNode;
    }
    return head;
}
```

```
void sortLinkedList(struct Node** head)
{
    if(*head == NULL) return;
    struct Node* current = *head;
    struct Node* Index = NULL;
    int temp;

    while(current != NULL)
    {
        Index = current->next;
        while(Index != NULL){
            if(current->data > Index->data)
            {
                temp = current->data;
                current->data = Index->data
                Index->data = temp;

            }
            Index = Index->next;

        }
        current = current->next;
    }
}

void reverseLinkedList(struct Node **head)
{
    struct node* Prev = NULL;
    *current = *head;
    * next = NULL;
    while(current != NULL)
    {
        next = current->next;
        current->next = Prev;
        Prev = current;
        current = next;

    }
    *head = Prev;
}
struct node* concatenateList(struct node* head1, struct head* head)
if(head1 == NULL) return head2;
if(head 2== NULL) return head1;
```

```
struct node * temp = head1;
while (temp -> next != NULL)
  { temp = temp -> next
  }
temp -> next = head2;
return head1;
}
void PrintList (struct node * head)
{ struct node * temp = head;
while (temp != NULL)
{
Printf("%d", temp -> data);
temp = temp -> next;
}
Printf("\n");
}
int main()
{ struct node * list1 = CreateLinkedList();
struct node * list2 = CreateLinkedList();

Printf ("List 1:\n");
PrintList (list1);

Printf ("List 2:\n");
PrintList (list2);

SortLinkedList (&list1);
Printf ("stored list 1:\n");
Printlist(list1);

reverse LinkedList (&list1);
Printf ("Reversed list 1:\n");
PrintList (list1);

struct node * concatenated list = concatenatedList(list1, list2);
Printf("concatenated list:\n");
PrintList (concatenated list);
return 0;
}
```

OUTPUT:

```
Number of elements : 4
Enter value : 3
Enter value : 2
Enter value : 4
Enter value : 8
Number of elements : 4
Enter value : 4
Enter value : 6
Enter value : 7
Enter value : 2

List 1 : 3 2 4 8
List 2 : 4 6 7 2

Sorted List 1 : 2 3 4 8
Reversed List 1 : 8 4 3 2
Concatenated list : 8 4 3 2 4 6 7 2
```

6
2/12

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

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

struct Node* head = NULL;
struct Node* head2 = NULL;

void createList(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

void sortList(struct Node* head) {
    struct Node *i, *j;
    int temp;
    for (i = head; i != NULL; i = i->next) {
```

```
        for (j = i->next; j != NULL; j = j->next) {
            if (i->data > j->data) {
                temp = i->data;
                i->data = j->data;
                j->data = temp;
            }
        }
    }
}

void reverseList(struct Node** head) {
    struct Node *prev = NULL, *current = *head, *next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head = prev;
}

void concatenateLists(struct Node* head1, struct Node* head2) {
    if (head1 == NULL) {
        head1 = head2;
        return;
    }
    struct Node* temp = head1;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = head2;
}

int main() {
    int choice, data;

    while (1) {
        printf("1. Create List 1\n");
        printf("2. Create List 2\n");
        printf("3. Display List 1\n");
```

```
printf("4. Display List 2\n");
printf("5. Sort List 1\n");
printf("6. Reverse List 1\n");
printf("7. Concatenate Lists\n");
printf("8. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter data to create List 1: ");
        scanf("%d", &data);
        createList(&head, data);
        break;
    case 2:
        printf("Enter data to create List 2: ");
        scanf("%d", &data);
        createList(&head2, data);
        break;
    case 3:
        printf("List 1: ");
        displayList(head);
        break;
    case 4:
        printf("List 2: ");
        displayList(head2);
        break;
    case 5:
        sortList(head);
        printf("List 1 sorted.\n");
        break;
    case 6:
        reverseList(&head);
        printf("List 1 reversed.\n");
        break;
    case 7:
        concatenateLists(head, head2);
        printf("Lists concatenated.\n");
        break;
    case 8:
```

```
            exit(0);
        default:
            printf("Invalid choice\n");
    }
  }

  return 0;
}
```

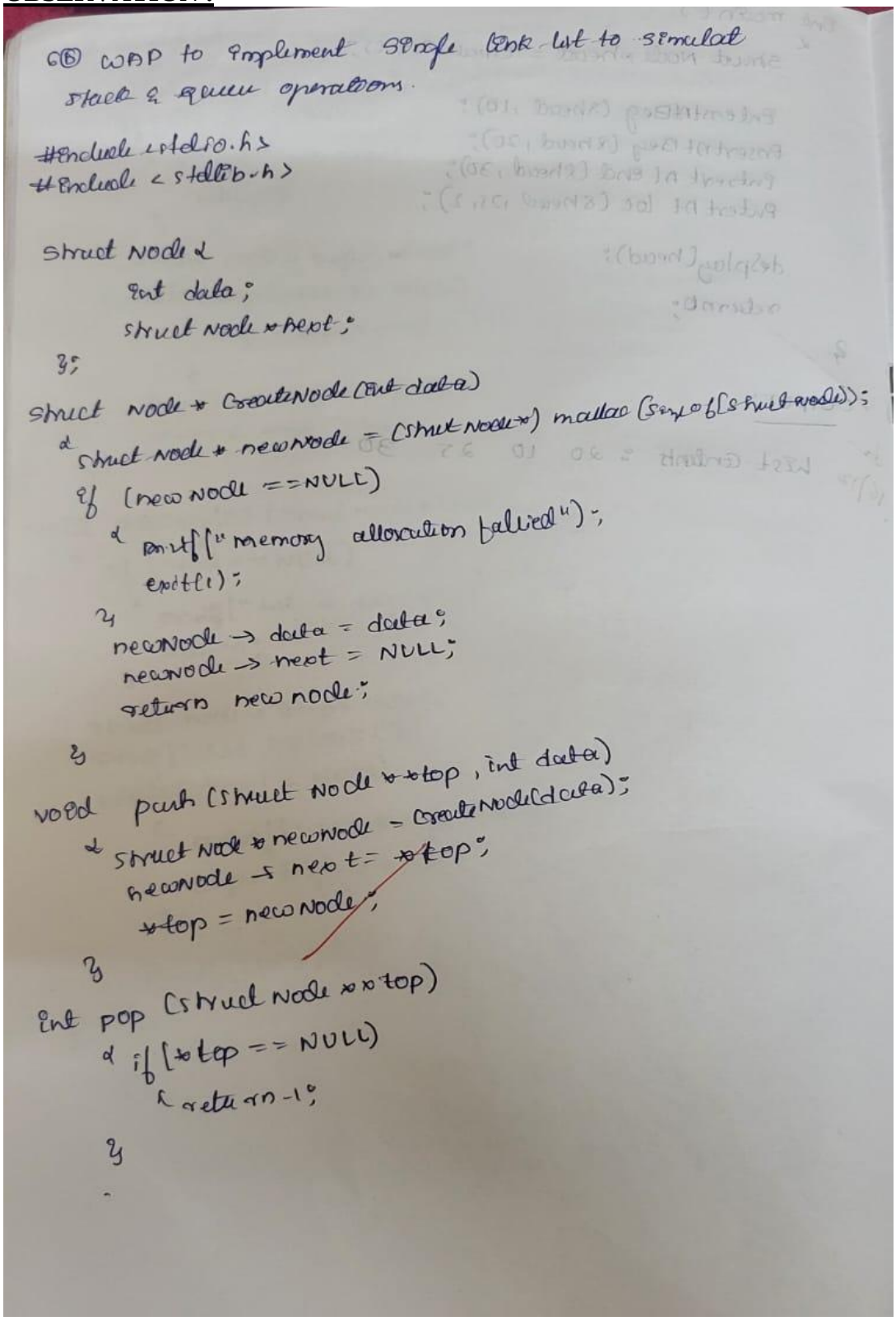
OUTPUT:

```
List 1: 5 -> 1 -> 9 -> 3 -> NULL
List 2: 8 -> 2 -> 4 -> NULL

List 1 after sorting: 1 -> 3 -> 5 -> 9 -> NULL

List 2 after reversing: 4 -> 2 -> 8 -> NULL

List 1 after concatenation with List 2: 1 -> 3 -> 5 -> 9 -> 4 -> 2 -> 8 -> NULL
```

Program 6

b) WAP to Implement Single Link List to simulate Stack & Queue Operations

OBSERVATION :



```
6(b) WAP to Implement Single link list to simulat
  stack & queue operations.

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

struct Node {
      int data;
      struct Node *next;
};
struct Node * CreateNode (int data)
{
  struct Node * newNode = (struct Node*) malloc (sizeof(struct node));
  if (newNode ==NULL)
    { printf("memory allocation failed");
      exit(1);
    }
    newNode -> data = data;
    newNode -> next = NULL;
    return newNode;
}
void push (struct Node **top, int data)
{ struct Node *newNode = CreateNode(data);
  newNode -> next = *top;
  *top = newNode;
}
int pop (struct Node **top)
{ if (*top == NULL)
    { return -1;
    }
```

```
Node *temp = *front;
int dequedData = temp->data;
*front = (*front)->next;
if (*front == NULL)
    *rear = NULL;
free(temp);
return dequedData;
}

int front(Node *front)
{
return front = NULL;
}

int main()
{
  Node *sp = NULL;
  Node *qf = NULL;
  Node *qr = NULL;
Printf("stack");
push(&ST, 10);
push(&ST, 20);
```

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

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

struct Node* stackTop = NULL;
struct Node* queueFront = NULL;
struct Node* queueRear = NULL;

// Stack Operations
void push(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = stackTop;
    stackTop = newNode;
}

int pop() {
    if (stackTop == NULL) {
        printf("Stack is empty.\n");
        return -1;
    }
    struct Node* temp = stackTop;
    int data = temp->data;
    stackTop = stackTop->next;
    free(temp);
    return data;
}

void displayStack() {
    struct Node* temp = stackTop;
    if (temp == NULL) {
        printf("Stack is empty.\n");
    } else {
        printf("Stack: ");
        while (temp != NULL) {
            printf("%d ", temp->data);
```

```
            temp = temp->next;
        }
        printf("\n");
    }
}

// Queue Operations
void enqueue(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (queueRear == NULL) {
        queueFront = queueRear = newNode;
        return;
    }
    queueRear->next = newNode;
    queueRear = newNode;
}

int dequeue() {
    if (queueFront == NULL) {
        printf("Queue is empty.\n");
        return -1;
    }
    struct Node* temp = queueFront;
    int data = temp->data;
    queueFront = queueFront->next;
    if (queueFront == NULL) {
        queueRear = NULL;
    }
    free(temp);
    return data;
}

void displayQueue() {
    struct Node* temp = queueFront;
    if (temp == NULL) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue: ");
```

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

int main() {
    int choice, data;

    while (1) {
        printf("1. Push to Stack\n");
        printf("2. Pop from Stack\n");
        printf("3. Display Stack\n");
        printf("4. Enqueue to Queue\n");
        printf("5. Dequeue from Queue\n");
        printf("6. Display Queue\n");
        printf("7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to push to stack: ");
                scanf("%d", &data);
                push(data);
                break;
            case 2:
                data = pop();
                if (data != -1) {
                    printf("Popped from stack: %d\n", data);
                }
                break;
            case 3:
                displayStack();
                break;
            case 4:
                printf("Enter data to enqueue to queue: ");
                scanf("%d", &data);
```

```
            enqueue(data);
            break;
        case 5:
            data = dequeue();
            if (data != -1) {
                printf("Dequeued from queue: %d\n", data);
            }
            break;
        case 6:
            displayQueue();
            break;
        case 7:
            exit(0);
        default:
            printf("Invalid choice\n");
    }
  }

  return 0;
}
```

OUTPUT :

```
Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 1
Enter data to push: 1
 1 is pushed to stack

Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 1
Enter data to push: 2
 2 is pushed to stack

Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 1
Enter data to push: 3
 3 is pushed to stack
```

```
Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 3
Enter data to enqueue: 4

Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 3
Enter data to enqueue: 5

Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 3
Enter data to enqueue: 6
```

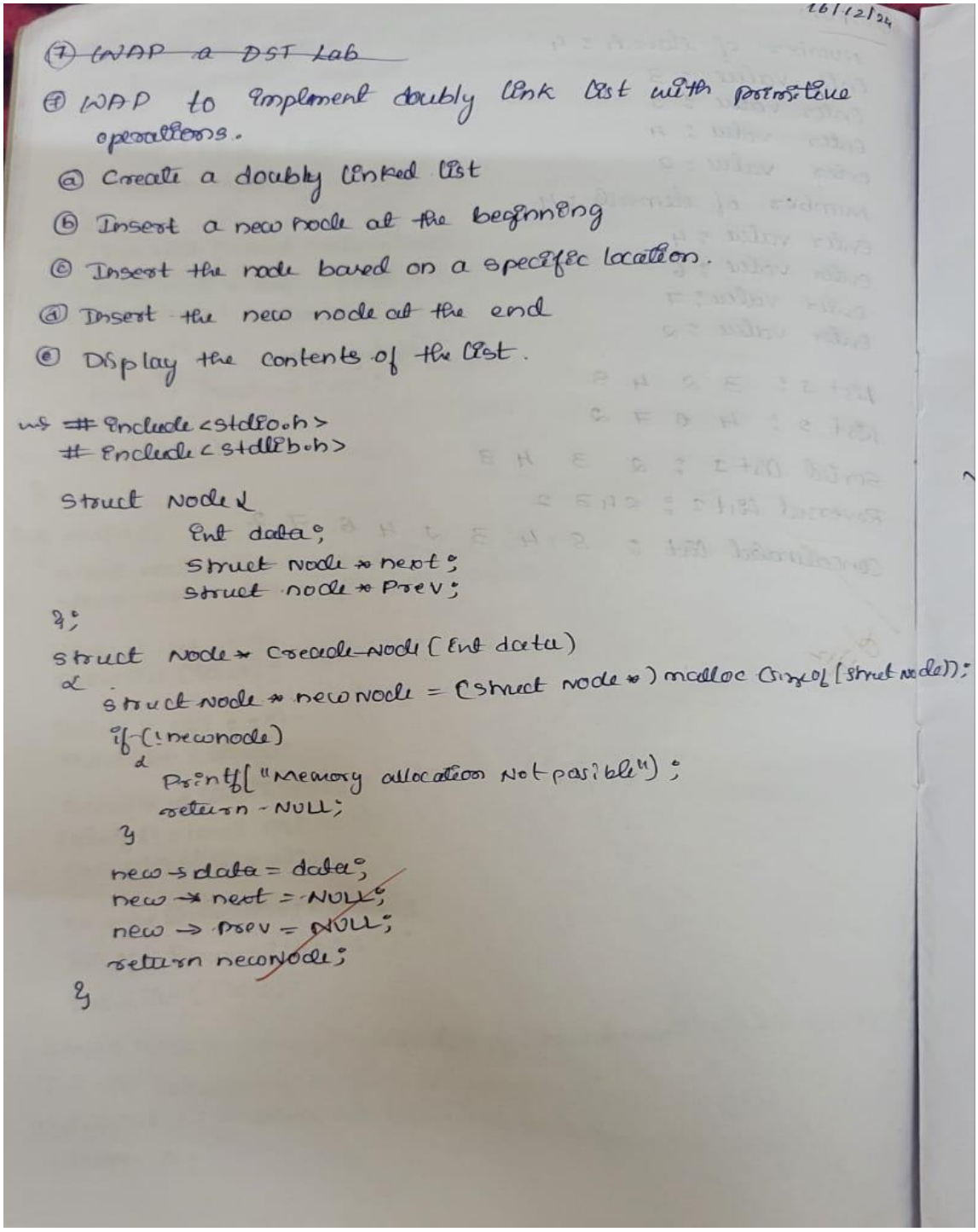
```
Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 5
Stack: 3 -> 2 -> 1 -> NULL

Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 6
Queue: 4 -> 5 -> 6 -> NULL

Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 2
Popped element is 3
```

```
Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 5
Stack: 2 -> 1 -> NULL

Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 4
Dequeued element is 4

Choose operation:
1. Stack Push
2. Stack Pop
3. Queue Enqueue
4. Queue Dequeue
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 6
Queue: 5 -> 6 -> NULL
```

Program 7

WAP to Implement doubly link list with primitive operations a) Create a doubly linked list. b) Insert a new node to the left of the node. c) Delete the node based on a specific value d) Display the contents of the list

OBSERVATION :



7) WAP a DST Lab

WAP to Implement doubly link list with primitive operations.

a) Create a doubly linked list

b) Insert a new node at the beginning

c) Insert the node based on a specific location.

d) Insert the new node at the end

e) Display the contents of the list.

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

struct Node{
        int data;
        struct node *next;
        struct node *Prev;
};
struct Node * CreateNode(int data)
{
    struct node *newNode = (struct node *) malloc (sizeof(struct node));
    if(!newnode)
    {
        Printf("Memory allocation Not pasible");
        return NULL;
    }
    new->data = data;
    new -> next = NULL;
    new -> Prev = NULL;
    return newNode;
}
```

```
Struct Node * CreateList()
{
    return NULL;
}

void InsertAtBeg(Struct Node **head, int data)
{
    Struct Node *newNode = CreateNode(data)
    if (*head == NULL)
    {
        *head = newnode;
        return;
    }
    newnode->next = *head;
    (*head)->prev = newNode;
    *head = newNode;
}

Void InsertAfter(Struct Node **head, int data, int pos)
{ Struct Node * newnode = CreatNode(data);
    if(*head == NULL || pos == 0)
    {  InsertAtBeg(head, data);
       return;
    }
)); Struct node *temp = *head;
    int count = 0;
    while (temp != NULL && count < pos-1)
    {  temp = temp->next;
       count++;
    }
    if (temp == NULL)
    {
      Printf("Position is beyond the current list");
      InsertAtend(head, data);
      return;
    }
  newNode->next = temp->next;
  if (temp->next != NULL)
     { temp->next->prev = newNode;
     }
  temp->next = newNode;
  newnode->prev = temp;
}
```

```
void insertAtEnd (struct Node **head, int data)
{
    struct Node *newNode = CreateNode(data);
    if (*head == NULL)
    {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL)
        temp = temp->next;
    temp->next = newNode;
    newnod->prev = temp;
}
void displayList (struct Node *head)
{
    if (head == NULL)
    {
        printf("List is empty");
        return;
    }
    struct Node *temp = head;
    printf("List contents:");
    while (temp != NULL)
    {
        printf("%d", temp->data);
        temp = temp->next;
    }
    Printf("\n");
}
```

WAP to implement doubly linked list with primitive operations.

a) Create a doubly linked list.
b) insert a new node at the begining.
c) insert the node biased on a operation location
d) insert a new node at the end
e) display the content of the list.

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

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

struct node*' createlist(){
   return NULL;
};

void insertatbeg(struct node** head, int data){
  struct Node * newnode=(struct node*)malloc(sizeof(st
  newnode->data=data
  newnode->prev=NULL;
  newnode->next=*head;

if(*head==NULL)
{(*head)->prev=newnode;}
 *head=newnode;

}
```

```
int main()
{
    struct Node *head = CreateList();

    InsertAtBeg(&head, 10);
    InsertAtBeg(&head, 20);
    InsertAtEnd(&head, 30);
    InsertAtLoc(&head, 25, 2);

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

OUTPUT:

List Contents : 20 10 25 30

16/12

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

struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
};

struct Node* head = NULL;

void createList(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->prev = NULL;
   newNode->next = NULL;

   if (head == NULL) {
      head = newNode;
   } else {
      struct Node* temp = head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
      newNode->prev = temp;
   }
}

void insertLeft(int newData, int existingData) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = newData;
   struct Node* temp = head;

   while (temp != NULL && temp->data != existingData) {
      temp = temp->next;
   }

   if (temp != NULL) {
      newNode->next = temp;
```

```
        newNode->prev = temp->prev;
        if (temp->prev != NULL) {
            temp->prev->next = newNode;
        } else {
            head = newNode;
        }
        temp->prev = newNode;
    } else {
        printf("Node with data %d not found.\n", existingData);
    }
}

void deleteNode(int value) {
    struct Node* temp = head;

    while (temp != NULL && temp->data != value) {
        temp = temp->next;
    }

    if (temp != NULL) {
        if (temp->prev != NULL) {
            temp->prev->next = temp->next;
        } else {
            head = temp->next;
        }
        if (temp->next != NULL) {
            temp->next->prev = temp->prev;
        }
        free(temp);
        printf("Node with value %d deleted.\n", value);
    } else {
        printf("Node with value %d not found.\n", value);
    }
}

void displayList() {
    struct Node* temp = head;

    if (temp == NULL) {
        printf("List is empty.\n");
```

```
    } else {
        printf("Doubly Linked List: ");
        while (temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    int choice, data, existingData;

    while (1) {
        printf("1. Create List\n");
        printf("2. Insert Node to the Left\n");
        printf("3. Delete Node\n");
        printf("4. Display List\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to create list: ");
                scanf("%d", &data);
                createList(data);
                break;
            case 2:
                printf("Enter new node data: ");
                scanf("%d", &data);
                printf("Enter the existing node data to insert left of: ");
                scanf("%d", &existingData);
                insertLeft(data, existingData);
                break;
            case 3:
                printf("Enter the node value to delete: ");
                scanf("%d", &data);
                deleteNode(data);
                break;
```

```
            case 4:
                displayList();
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

OUTPUT :

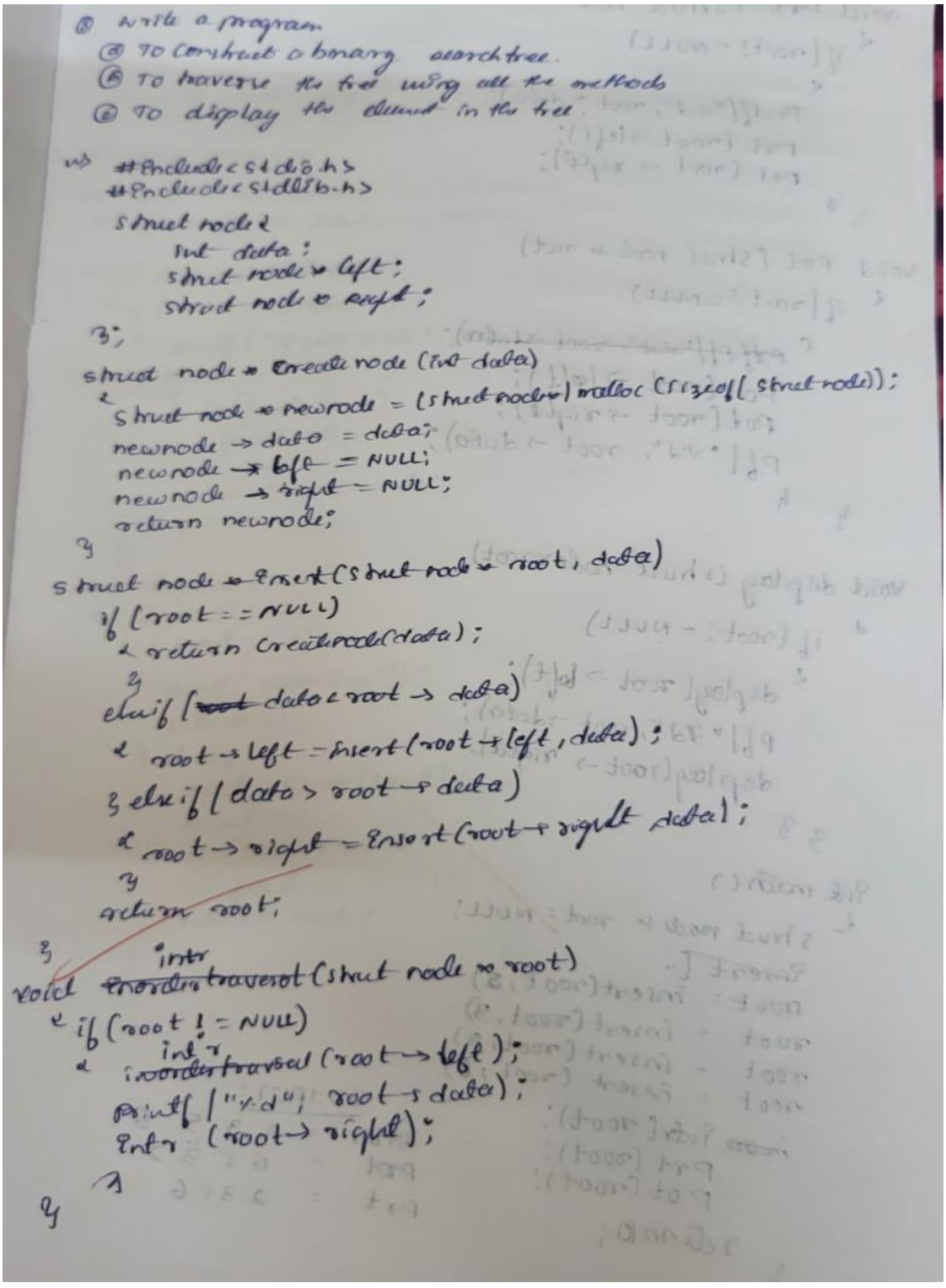
```
Enter the data:1
enter 1 to continue:1
Enter the data:2
enter 1 to continue:1
Enter the data:3
enter 1 to continue:2

Menu:
1. Insert at Beginning
2. Insert at Specific Position
3. Insert at End
4. Display List
5. Exit
Enter your choice: 1
Enter value to insert at the beginning: 4

Menu:
1. Insert at Beginning
2. Insert at Specific Position
3. Insert at End
4. Display List
5. Exit
Enter your choice: 4
4<->1<->2<->3<->NULL

Menu:
1. Insert at Beginning
2. Insert at Specific Position
3. Insert at End
4. Display List
5. Exit
Enter your choice: 2
Enter value to insert: 5
Enter the position:2

Menu:
1. Insert at Beginning
2. Insert at Specific Position
3. Insert at End
4. Display List
5. Exit
Enter your choice: 4
4<->5<->1<->2<->3<->NULL

Menu:
1. Insert at Beginning
2. Insert at Specific Position
3. Insert at End
4. Display List
5. Exit
Enter your choice: 3
```

```
Menu:
1. Insert at Beginning
2. Insert at Specific Position
3. Insert at End
4. Display List
5. Exit
Enter your choice: 3
Enter value to insert at the end: 6

Menu:
1. Insert at Beginning
2. Insert at Specific Position
3. Insert at End
4. Display List
5. Exit
Enter your choice: 4
4<->5<->1<->2<->3<->6<->NULL
```

Program 8

Write a program a) Toconstructabinary Searchtree. b) To traverse the tree using all the methods i.e., inorder, preorder and post order c) To display the elements in the tree

OBSERVATION :



```
8 Write a program
  a) To Construct a binary search tree.
  b) To traverse the tree using all the methods
  c) To display the element in the tree

#include<stdio.h>
#include<stdlib.h>
struct node{
    int data;
    struct node* left;
    struct node* right;
};
struct node* Create node (int data)
{
 struct node* newnode = (struct node*) malloc (sizeof( struct node));
 newnode -> data = data;
 newnode -> left = NULL;
 newnode -> right = NULL;
 return newnode;
}
struct node* Insert(struct node* root, data)
 if (root == NULL)
  return (Createnode(data);
  }
 elsif (data < root -> data)
 { root->left = Insert(root->left, data);
 } else if ( data > root -> data)
 { root -> right = Insert(root->right data);
 }
 return root;
}
void inter traverse(struct node* root)
 if (root != NULL)
 { inorder traverse (root -> left);
   printf ("%d", root->data);
   Intr (root -> right);
 }
}
```

```
void Prt (struct node *root)
{
  if(root!=NULL)
  {
    Printf("%d", root->data);
    Prt(root->left);
    Prt(root->right);
  }
}

void Pot (struct node * root)
{
  if(root!=NULL)
  {
    ~~Pf("%d", root->data);~~
    Pot(root->left);
    Pot(root->right);
    Pf("%d", root->data);
  }
}

void display (struct node*root)
{
  if(root!=NULL)
  {
    display(root->left);
    Pf("%d", root->data);
    display(root->right);
  }
}

int main()
{
  struct node * root = NULL;
  ~~insert~~(
  root = insert(root, 2)
  root = insert(root, 3)
  root = insert(root, 5)
  root = insert(root, 6)
  ~~inor~~ intr(root);
    Prt(root);
    Pot(root);
  return 0;
}
```

OUTPUT:

Intr = 2 3 5 6
Pot = 6 5 3 2
Prt = 2 3 5 6

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

// Structure for a node in the binary search tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function to insert a node in the binary search tree
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else {
        root->right = insert(root->right, data);
    }
    return root;
}

// In-order traversal (Left, Root, Right)
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

```
// Pre-order traversal (Root, Left, Right)
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Post-order traversal (Left, Right, Root)
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

// Function to display the elements in the tree using in-order traversal
void display(struct Node* root) {
    printf("In-order traversal: ");
    inorder(root);
    printf("\n");
}

int main() {
    struct Node* root = NULL;
    int choice, data;

    while (1) {
        printf("1. Insert Node\n");
        printf("2. In-order Traversal\n");
        printf("3. Pre-order Traversal\n");
        printf("4. Post-order Traversal\n");
        printf("5. Display In-order Traversal\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
```

```
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                root = insert(root, data);
                break;
            case 2:
                printf("In-order Traversal: ");
                inorder(root);
                printf("\n");
                break;
            case 3:
                printf("Pre-order Traversal: ");
                preorder(root);
                printf("\n");
                break;
            case 4:
                printf("Post-order Traversal: ");
                postorder(root);
                printf("\n");
                break;
            case 5:
                display(root);
                break;
            case 6:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

OUTPUT:

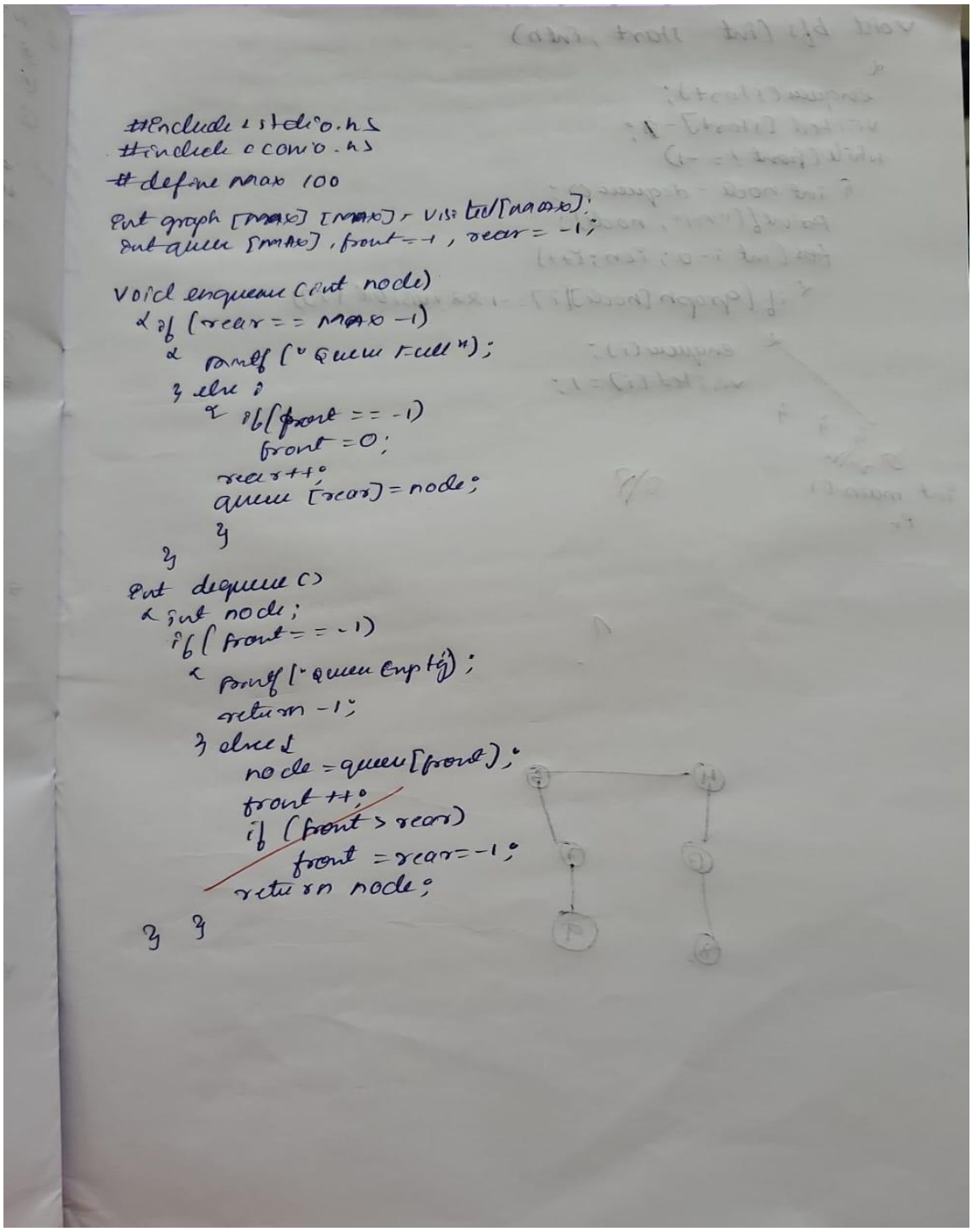
```
Binary Search Tree Operations:
1. Insert a node
2. Display In-order Traversal
3. Display Pre-order Traversal
4. Display Post-order Traversal
5. Exit

Enter your choice: 1
Enter the value to insert: 20

Enter your choice: 1
Enter the value to insert: 30

Enter your choice: 1
Enter the value to insert: 35

Enter your choice: 1
Enter the value to insert: 40

Enter your choice: 1
Enter the value to insert: 25

Enter your choice: 1
Enter the value to insert: 15

Enter your choice: 2
In-order Traversal: 15 20 25 30 35 40

Enter your choice: 3
Pre-order Traversal: 20 15 30 25 35 40

Enter your choice: 4
Post-order Traversal: 15 25 40 35 30 20
```

Program 9

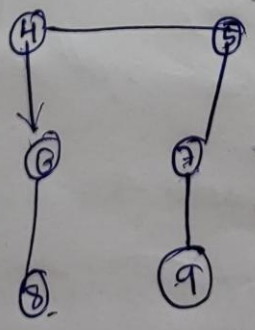
a) Write a program to traverse a graph using BFS method.

b) Write a program to check whether given graph is connected or not using DFS method.

OBSERVATION :



```
void bfs(int start, int n)
{
    enqueue(start);
    visited[start] = 1;
    while(front != -1)
    {
        int node = dequeue();
        printf("%d", node);
        for(int i=0; i<n; i++)
        {
            if(graph[node][i]==1 && !visited[i])
            {
                enqueue(i);
                visited[i] = 1;
            }
        }
    }
}
int main()
{
```



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

#define MAX_VERTICES 100

struct Queue {
    int items[MAX_VERTICES];
    int front, rear;
};

struct Graph {
    int adj[MAX_VERTICES][MAX_VERTICES];
    int vertices;
};

void initQueue(struct Queue* q) {
    q->front = -1;
    q->rear = -1;
}

int isQueueEmpty(struct Queue* q) {
    return q->front == -1;
}

void enqueue(struct Queue* q, int value) {
    if (q->rear == MAX_VERTICES - 1) {
        printf("Queue Overflow\n");
        return;
    }
    if (q->front == -1)
        q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
}

int dequeue(struct Queue* q) {
    if (isQueueEmpty(q)) {
        printf("Queue Underflow\n");
        return -1;
    }
```

```
    int item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }
    return item;
}

void initGraph(struct Graph* g, int vertices) {
    g->vertices = vertices;
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            g->adj[i][j] = 0;
        }
    }
}

void addEdge(struct Graph* g, int u, int v) {
    g->adj[u][v] = 1;
    g->adj[v][u] = 1;
}

void bfs(struct Graph* g, int startVertex) {
    int visited[MAX_VERTICES] = {0};
    struct Queue q;
    initQueue(&q);

    visited[startVertex] = 1;
    enqueue(&q, startVertex);

    printf("BFS Traversal starting from vertex %d: ", startVertex);

    while (!isQueueEmpty(&q)) {
        int currentVertex = dequeue(&q);
        printf("%d ", currentVertex);

        for (int i = 0; i < g->vertices; i++) {
            if (g->adj[currentVertex][i] == 1 && !visited[i]) {
                enqueue(&q, i);
                visited[i] = 1;
```

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

int main() {
    struct Graph g;
    int vertices, edges, u, v, startVertex;

    printf("Enter number of vertices: ");
    scanf("%d", &vertices);
    initGraph(&g, vertices);

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

    for (int i = 0; i < edges; i++) {
        printf("Enter edge (u v): ");
        scanf("%d %d", &u, &v);
        addEdge(&g, u, v);
    }

    printf("Enter the starting vertex for BFS: ");
    scanf("%d", &startVertex);

    bfs(&g, startVertex);

    return
```

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

#define MAX_VERTICES 100

struct Graph {
    int adj[MAX_VERTICES][MAX_VERTICES];
    int vertices;
};

void initGraph(struct Graph* g, int vertices) {
    g->vertices = vertices;
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            g->adj[i][j] = 0;
        }
    }
}

void addEdge(struct Graph* g, int u, int v) {
    g->adj[u][v] = 1;
    g->adj[v][u] = 1;
}

void dfs(struct Graph* g, int vertex, int visited[]) {
    visited[vertex] = 1;
    for (int i = 0; i < g->vertices; i++) {
        if (g->adj[vertex][i] == 1 && !visited[i]) {
            dfs(g, i, visited);
        }
    }
}

int isConnected(struct Graph* g) {
    int visited[MAX_VERTICES] = {0};
    dfs(g, 0, visited);
    for (int i = 0; i < g->vertices; i++) {
        if (!visited[i]) {
            return 0;
        }
```

```
    }
    return 1;
}

int main() {
    struct Graph g;
    int vertices, edges, u, v;

    printf("Enter number of vertices: ");
    scanf("%d", &vertices);
    initGraph(&g, vertices);

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

    for (int i = 0; i < edges; i++) {
        printf("Enter edge (u v): ");
        scanf("%d %d", &u, &v);
        addEdge(&g, u, v);
    }

    if (isConnected(&g)) {
        printf("The graph is connected.\n");
    } else {
        printf("The graph is not connected.\n");
    }

    return 0;
}
```

OUTPUT :

```
Enter the number of vertices: 6
Enter the adjacency matrix:
0 1 1 0 1 1
1 0 0 1 0 1
0 1 1 1 0 0
1 1 1 0 1 0
0 1 0 1 0 1
1 1 1 1 0 0
Enter the starting vertex: 1
BFS Traversal: 1 0 3 5 2 4
```

```
Enter the number of vertices: 4
Enter the adjacency matrix:
0 1 1 1
1 1 1 0
1 0 1 0
0 0 0 1
Enter the starting vertex: 0
DFS Traversal: 0 1 2 3
```