Question 1:

Write a program to find the sum of elements of an array.

Answer:

Algorithm:

```
Step 1: Start.
Step 2: Define an array of size N: arr[N].
Step 3: Take the value of N from the user.
Step 4: Input the elements of the array.
Step 5: Initialize a variable sum= 0.
Step 6: Start a loop that iterates over each element of the array:
        For each element arr[i] (where i is the index of the element):
                Perform sum = sum + arr[i]
Step 7: After the end of the loop , display the value of sum.
Step 8: End.
```

Example:

let's suppose, the user enters 5 for the size of the array and the following elements:

arr[5] = {5, 8, 4, 2, 3}

Initially, sum = 0 and loop executes 5 times.

- At first iteration, sum = sum + arr[0]
  - sum = 0 + 5 = 5
- At second iteration, sum = sum + arr[1]
  - sum = 5 + 8 = 13
- At third iteration, sum = sum + arr[2]
  - sum = 13 + 4 = 17
- At fourth iteration, sum = sum + arr[3]
  - sum = 17 + 2 = 19
- At last iteration, sum = sum + arr[4]

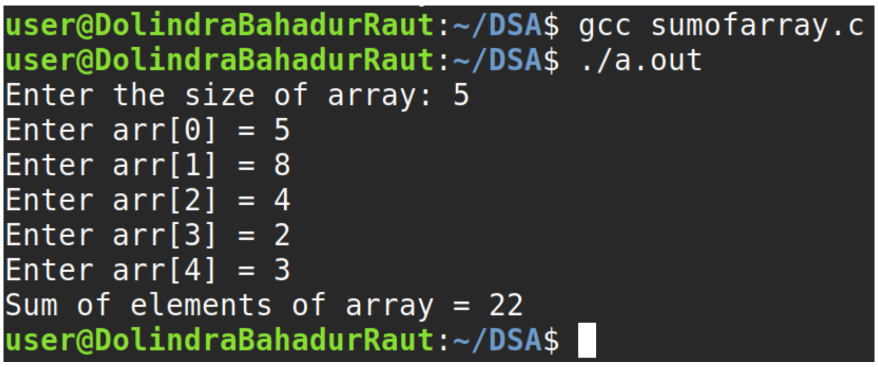
- sum = 19 + 3 = 22

Hence, the sum of the elements of arr[] = {5, 8, 4, 2, 3} is 22.

Program:

```
#include<stdio.h>
int main(){
   int i,N,sum = 0;
   printf("Enter the size of array: ");
   scanf("%d",&N);
   int arr[N];
   for(i=0; i<N; i++){
      printf("Enter arr[%d] = ",i);
      scanf("%d",&arr[i]);
   }
   for(i=0; i<N; i++){
         sum = sum + arr[i];
   }
   printf("Sum of elements of array = %d\n",sum);
   return 0;
}
```

Output:



```
user@DolindraBahadurRaut:~/DSA$ gcc sumofarray.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
Enter the size of array: 5
Enter arr[0] = 5
Enter arr[1] = 8
Enter arr[2] = 4
Enter arr[3] = 2
Enter arr[4] = 3
Sum of elements of array = 22
user@DolindraBahadurRaut:~/DSA$
```

Conclusion:

Hence, we successfully created a program that computes the sum of the elements of an array.

Question 2:

Write a program to find the multiplication of two matrices.

Answer:

Algorithm:

Step 1: Start

Step 2: Declare variables:

- Integer m, n for the number of rows and columns of the first matrix A.
- Integer p, q for the number of rows and columns of the second matrix B.
- Declare two 2D arrays A[m][n] and B[p][q] to store the matrices.
- Declare a 2D array C[m][q] to store the result of matrix multiplication.

Step 4: Input elements for matrix A and matrix B.

Step 3: if (n != p), display “matrix multiplication is not possible”.

Step 4: Otherwise Perform matrix multiplication:

    - For each element C[i][j], compute the sum of products of the i-th row of A and the j-th column of B

Step 5: display the resulting matrix C.

Step 6: End

Example:

let assume the two matrices A and B:

Matrix A:		Matrix B:	
4	2	0	5
1	1	2	6

Then the multiplication of Matrix A and B is,

```
4*0 + 2*2      4*5 + 2*6

1*0 + 1*2      1*5 + 1*6


4        32

2        11
```

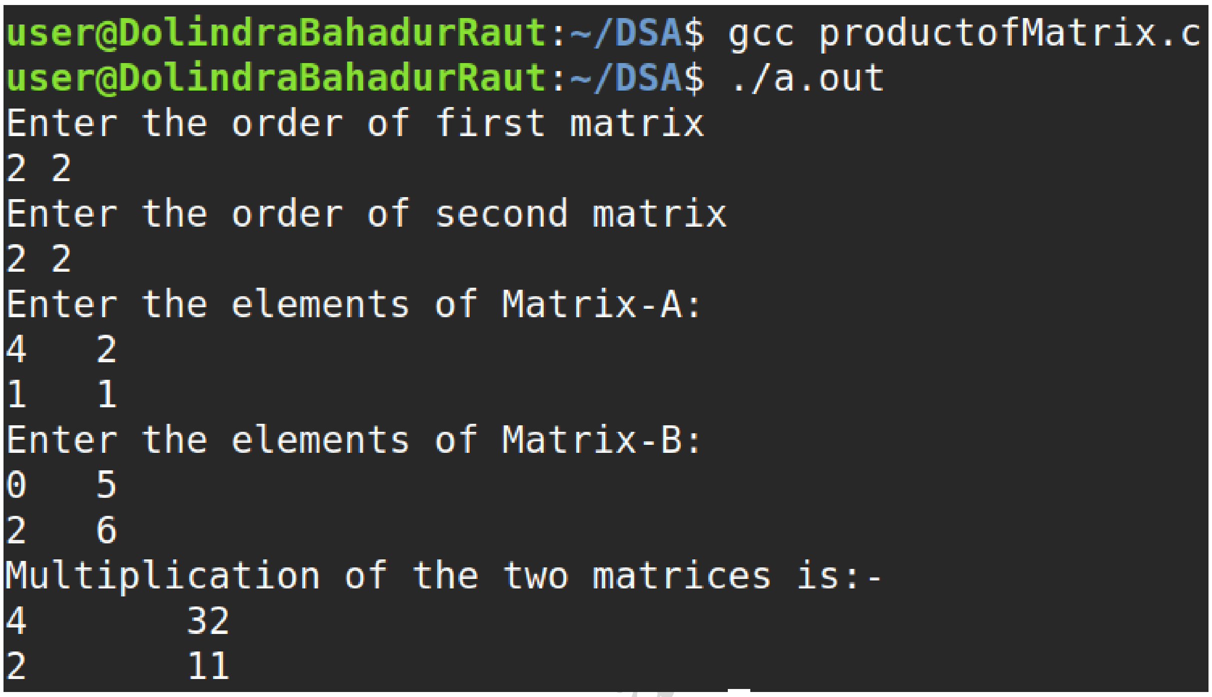
Program:

```
#include<stdio.h>
int main() {
   int m, n, p, q, i, j, k;
   printf("Enter the order of first matrix\n");
   scanf("%d%d", & m, & n);
   printf("Enter the order of second matrix\n");
   scanf("%d%d", & p, & q);
   int A[m][n],B[p][q],C[m][q];
   if (n != p) {
      printf("Matrix multiplication is not possible.\n");
   }
else {
      printf("Enter the elements of Matrix-A:\n");
      for (i = 0; i < m; i++) {
         for (j = 0; j < n; j++) {
            scanf("%d", & A[i][j]);
```

```
        }
    }
    printf("Enter the elements of Matrix-B:\n");
    for (i = 0; i < p; i++) {
        for (j = 0; j < q; j++) {
            scanf("%d", & B[i][j]);
        }
    }
    for (i = 0; i < m; i++) {
        for (j = 0; j < q; j++) {
            C[i][j] = 0;
            for (k = 0; k < p; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    printf("Multiplication of the two matrices is:-\n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < q; j++) {
            printf("%d\t", C[i][j]);
        }
        printf("\n");
    }
}
```

```
    return 0;

}
```

Output:



```
user@DolindraBahadurRaut:~/DSA$ gcc productofMatrix.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
Enter the order of first matrix
2 2
Enter the order of second matrix
2 2
Enter the elements of Matrix-A:
4   2
1   1
Enter the elements of Matrix-B:
0   5
2   6
Multiplication of the two matrices is:-
4       32
2       11
```

Conclusion:

Hence, We successfully implemented a matrix multiplication program that correctly computes the product of two matrices using the concept of 2-D array.

Question 3:

Write a program to find the largest element of the array of integer type.

Answer:

Algorithm:

Step 1: Start.

Step 2: Declare the variable N as the size of the array.

Step 3: Read the value of N and Define: int arr[N].

Step 4: Read elements of the array.

Step 5: Initialize, largest = arr[0].

Step 6: Iterate Through the Array: Traverse the array starting from the second element (i = 1):

- For each element arr[i]:
    - If arr[i] > largest, largest = arr[i].

Step 7: Print the largest element.

Step 8: End.

Example:

let's suppose, N = 5 and arr = {8,0,-4,1,2}

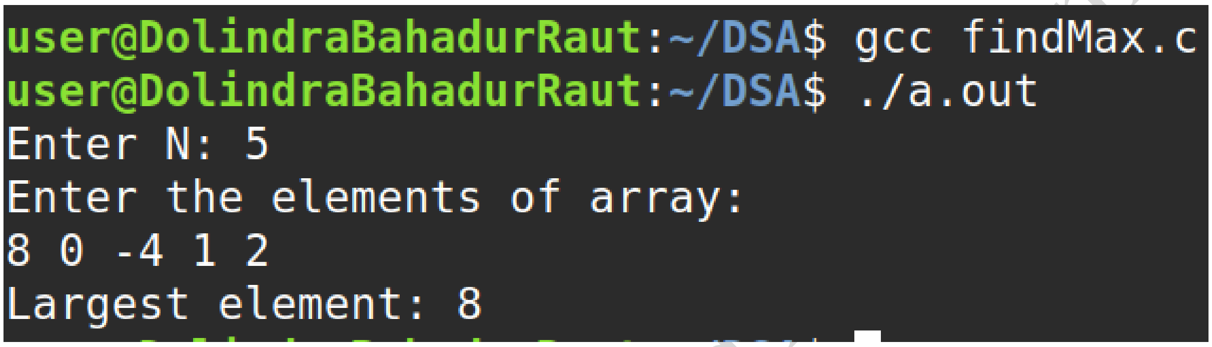
Then, largest element of the array = 8.

Program:

```
#include <stdio.h>

int main() {

   int N,i;

   printf("Enter N: ");

   scanf("%d",&N);

   int arr[N];

   printf("Enter the elements of array:\n");

   for(i=0; i<N; i++){

      scanf("%d",&arr[i]);

   }

   int largest = arr[0];

   for (int i = 1; i < N; i++) {

      if (arr[i] > largest) {

         largest = arr[i]; // Update largest element
```

```
        }
    }
    printf("Largest element: %d\n", largest);
    return 0;
}
```

Output:



```
user@DolindraBahadurRaut:~/DSA$ gcc findMax.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
Enter N: 5
Enter the elements of array:
8 0 -4 1 2
Largest element: 8
```

Conclusion:

In this question, we successfully implemented a program to find the largest element in an array.

Question 4:

Write a program to search an element in a 2-D array/Matrix.

Answer:

Algorithm:

Step 1: Start.

Step 2: Declare and read values for m and n.

Step 3: Declare a 2D array Matrix[m][n] to store the elements.

Step 4: Read the elements of the 2D array from the user:

Step 5: Declare: int target, int index1, int index2.

Step 6: Read the value of the variable: target.

Step 8: Start a loop for rows.

- Inside the row loop, start another loop for columns.
    - Compare Matrix[i][j] with the target.
    - If Matrix[i][j] == target, then:
        - Set index1 = i (row index of the target).
        - Set index2 = j (column index of the target).
    - break the loop

Step 11: print the position of the target element.

Step 12: End.

Example:

let assume, m = 3 and n =4 and 2-D array is given by,
[{2,3,4,0},

{-9,11,8,23},

{18,32,-6,-7}]

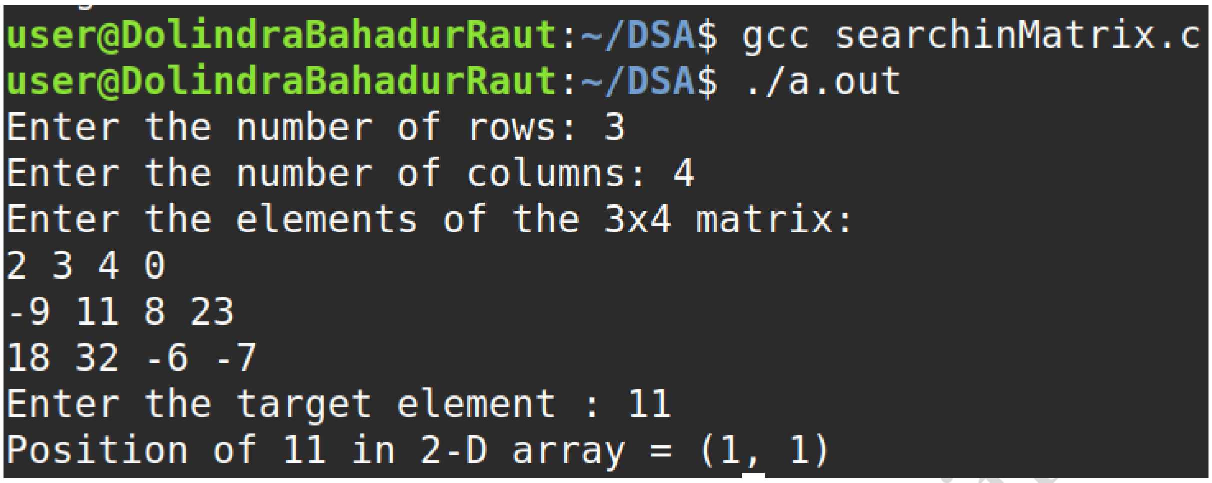
targeted element = 11

Position of 11 in 2-D array = (1,1).

Program:

```
#include <stdio.h>

int main() {

    int m,n,i,j;

    printf("Enter the number of rows: ");

    scanf("%d", &m);

    printf("Enter the number of columns: ");

    scanf("%d", &n);

    int Matrix[m][n];
    printf("Enter the elements of the %dx%d matrix:\n", m, n);

    for (i = 0; i < m; i++) {

        for (j = 0; j < n; j++) {

            scanf("%d", &Matrix[i][j]);

        }

    }

    int target, index1,index2;

    printf("Enter the target element : ");

    scanf("%d", &target);

    for (i = 0; i < m; i++) {

        for (j = 0; j < n; j++) {

            if (Matrix[i][j] == target) {

                index1 = i;

                index2 = j;

                break;  // Exit the loop once the target is found

            }

        }

    }

printf("Position of %d in 2-D array = (%d, %d)\n", target, index1, index2);

    return 0;

}
```

Output:



```
user@DolindraBahadurRaut:~/DSA$ gcc searchinMatrix.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
Enter the number of rows: 3
Enter the number of columns: 4
Enter the elements of the 3x4 matrix:
2 3 4 0
-9 11 8 23
18 32 -6 -7
Enter the target element : 11
Position of 11 in 2-D array = (1, 1)
```

Conclusion:

Hence, we sucessfully provided a program in c language to find the targeted element in a 2-D array with output.

Question 5:

Write a program to implement Stack using array.

Answer:

Push Operation:

1. If (top == MAXSIZE-1) , then print the “Stack Overflow” message.
2. Otherwise, perform: top = top + 1 , and the new value is inserted at top position .
3. The elements can be pushed into the stack till we reach the capacity of the stack.

Pop Operation:

1. If (top == -1), then print the “Stack Underflow” message.
2. Otherwise, decrement the value of top by 1 (top = top – 1) and return the value pointed by top.

Top operation:

1. If the stack is empty (top == -1), we simply print “Stack is empty”.

2. Otherwise, we return the element pointed by the top.

Isempty operation:

1. If (top == -1), then return 1.
2. Otherwise, return 0.

Example:

Push(1) , Stack: 1

Push(8), Stack: 1,8

Push(10), Stack: 1,8,10

Pop(), Stack: 1,8

Push(12), Stack: 1,8,12

Top(), Top element = 12

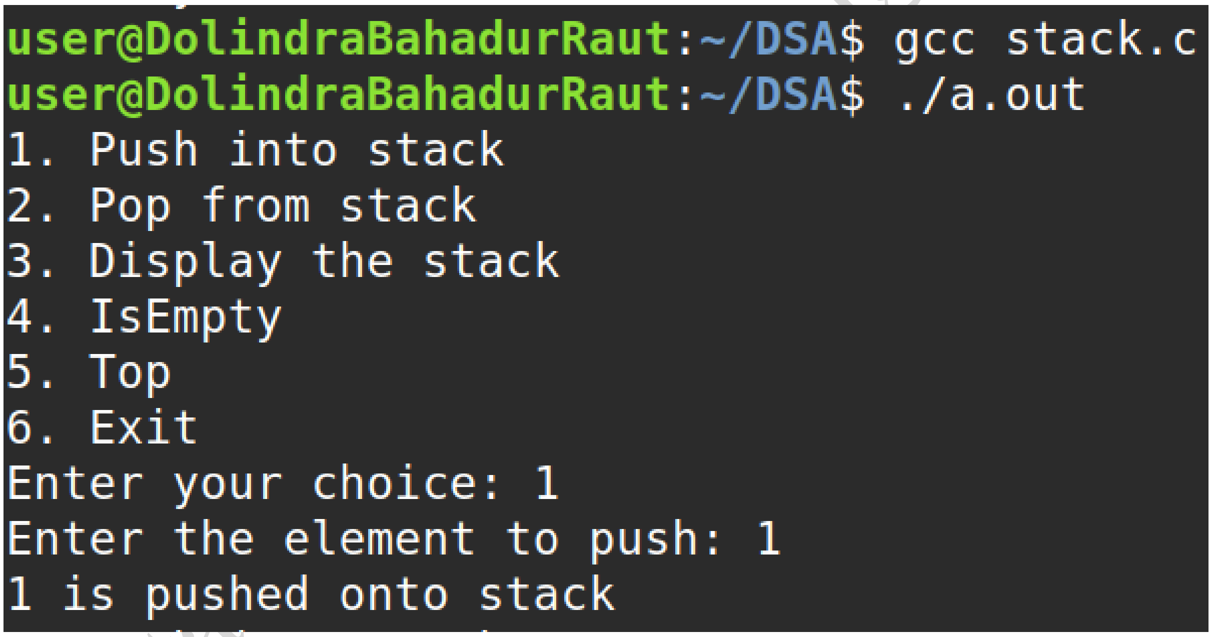
Program:

```
#include<stdio.h>

#define MAX_SIZE 8

int Stack[MAX_SIZE];

int top = -1;

void Push(int x){

   if(top == MAX_SIZE - 1){

     printf("Stack overflow\n");

  }

  else{
        top++;
         Stack[top] = x;
          printf("%d is pushed onto stack\n",x);
    }
}
void Pop(){
    if(top == -1){
        printf("Stack underflow\n");
    }
    else{
        printf("Poped element is %d\n",Stack[top]);
        top = top - 1;
    }
}
int  IsEmpty(){
    if(top == -1){
        return 1;
    }
    else{
        return 0;
    }
      printf("\n");
}
```

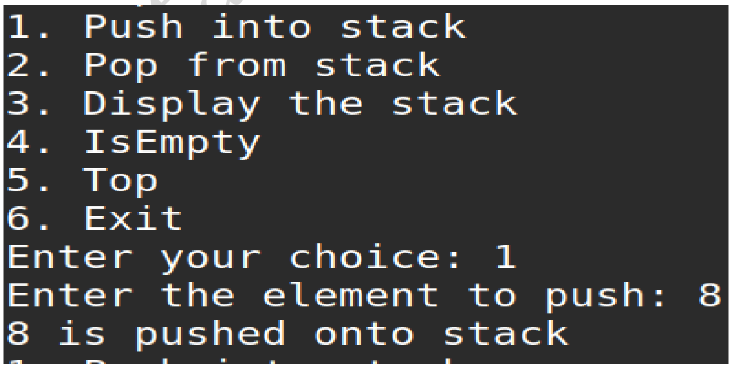
```
int Top(){
   if(IsEmpty() == 1){
      printf("Stack is empty.\n");
   }
   else{
      return Stack[top];
   }
}
void display_Stack(){
   int i;
   printf("Stack: ");
   for(i=0; i<=top; i++){
      printf("%d ",Stack[i]);
   }
   printf("\n");
}
int main(){
   int choice;
   do{
      printf("1. Push into stack\n");
      printf("2. Pop from stack\n");
      printf("3. Display the stack\n");
      printf("4. IsEmpty\n");
      printf("5. Top\n");

      printf("6. Exit\n");

      printf("Enter your choice: ");

      scanf("%d",&choice);

      int a;

      switch(choice){

        case 1:

              printf("Enter the element to push: ");

              scanf("%d",&a);

              Push(a);

              break;

        case 2:

              Pop();

              break;

        case 3:

              display_Stack();

              break;

        case 4:

              if(IsEmpty() == 1){

                printf("Stack is empty.\n");

              }else{

                printf("Stack is not empty.\n");

              }
```

```
                break;

            case 5:

                printf("Top element = %d\n",Top());

                break;

        }

    }while(choice != 6);

    return 0;

}
```

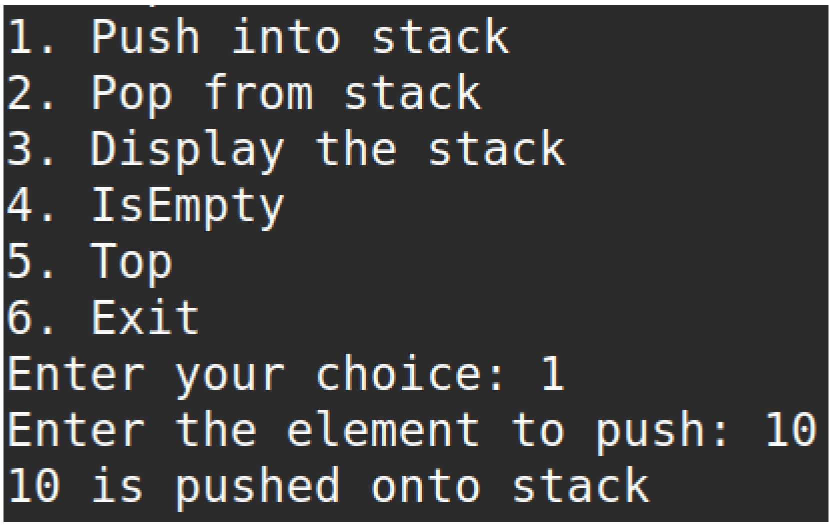
Output:



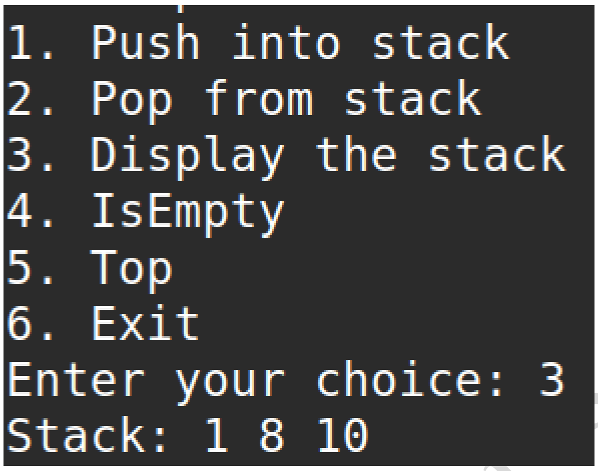
```
user@DolindraBahadurRaut:~/DSA$ gcc stack.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
1. Push into stack
2. Pop from stack
3. Display the stack
4. IsEmpty
5. Top
6. Exit
Enter your choice: 1
Enter the element to push: 1
1 is pushed onto stack
```



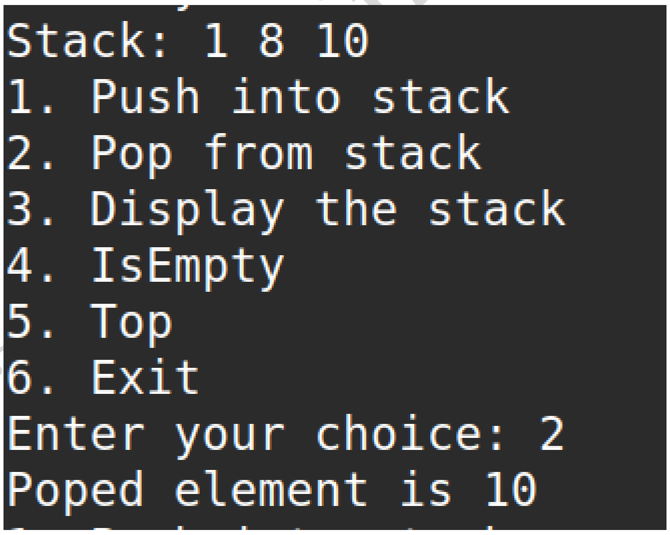
```
1. Push into stack
2. Pop from stack
3. Display the stack
4. IsEmpty
5. Top
6. Exit
Enter your choice: 1
Enter the element to push: 8
8 is pushed onto stack
```



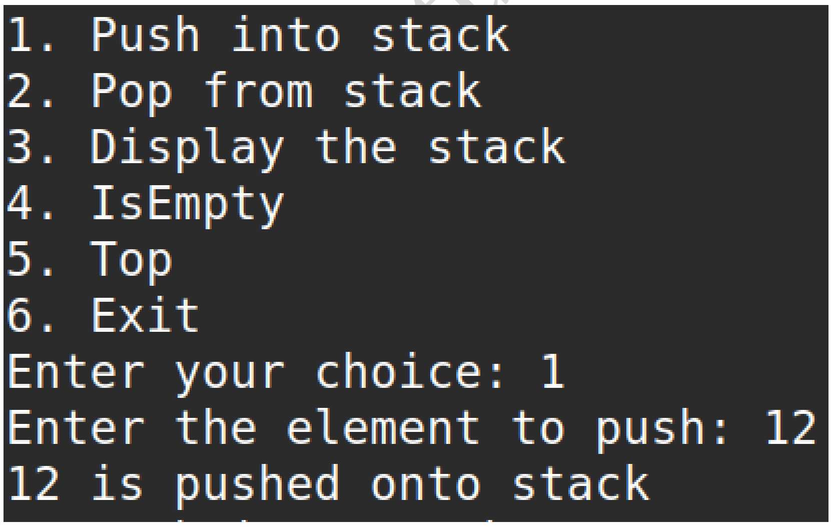
```
1. Push into stack
2. Pop from stack
3. Display the stack
4. IsEmpty
5. Top
6. Exit
Enter your choice: 1
Enter the element to push: 10
10 is pushed onto stack
```



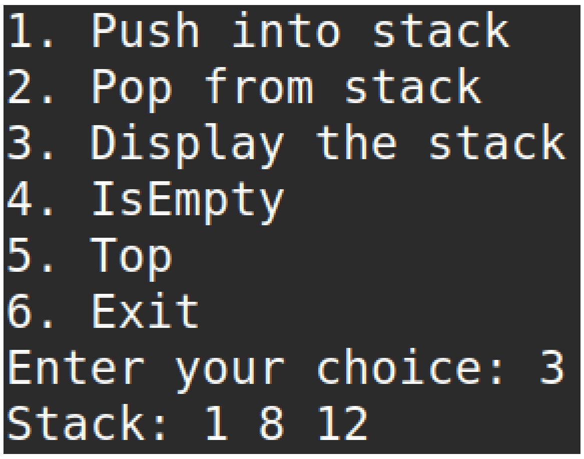
```
1. Push into stack
2. Pop from stack
3. Display the stack
4. IsEmpty
5. Top
6. Exit
Enter your choice: 3
Stack: 1 8 10
```



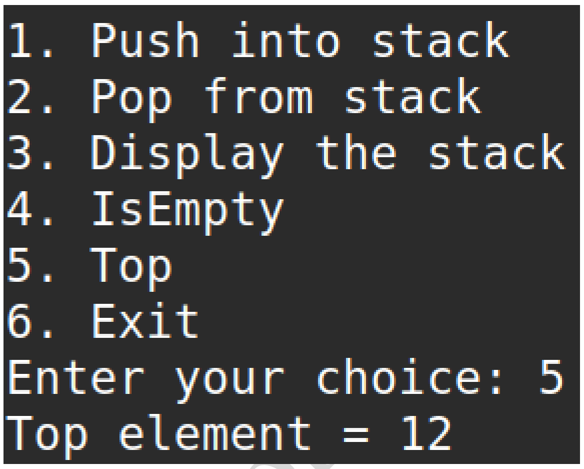
```
Stack: 1 8 10
1. Push into stack
2. Pop from stack
3. Display the stack
4. IsEmpty
5. Top
6. Exit
Enter your choice: 2
Poped element is 10
```



```
1. Push into stack
2. Pop from stack
3. Display the stack
4. IsEmpty
5. Top
6. Exit
Enter your choice: 1
Enter the element to push: 12
12 is pushed onto stack
```



```
1. Push into stack
2. Pop from stack
3. Display the stack
4. IsEmpty
5. Top
6. Exit
Enter your choice: 3
Stack: 1 8 12
```



```
1. Push into stack
2. Pop from stack
3. Display the stack
4. IsEmpty
5. Top
6. Exit
Enter your choice: 5
Top element = 12
```

Conclusion:

Hence, We successfully gave a program for the array based implementation of Stack with Push, Pop, Top and IsEmpty operation.

Question 6:

Write a program to check the balance of parentheses using stack.

Answer:

Algorithm:

Step 1: Start.

Step 2: Initialize a stack: Create an empty stack to store opening parentheses.

Step 3: Traverse through each character:

- If the character is an opening parenthesis ('(', '{', '['), push it onto the stack.
- If the character is a closing parenthesis (')', '}', ']'), check:
    - If the stack is empty, return false (unbalanced).

   - Otherwise, pop the top element from the stack and check if it matches the type of closing parenthesis.
   - If it doesn’t match, return false (unbalanced).

Step 4: Final check:

- After traversing the entire string, if the stack is not empty, return false (unbalanced).
- If the stack is empty, return true (balanced).

Example:

Input: [{()}]

Output: Balanced

Input: {[(])}

Output: Unbalanced

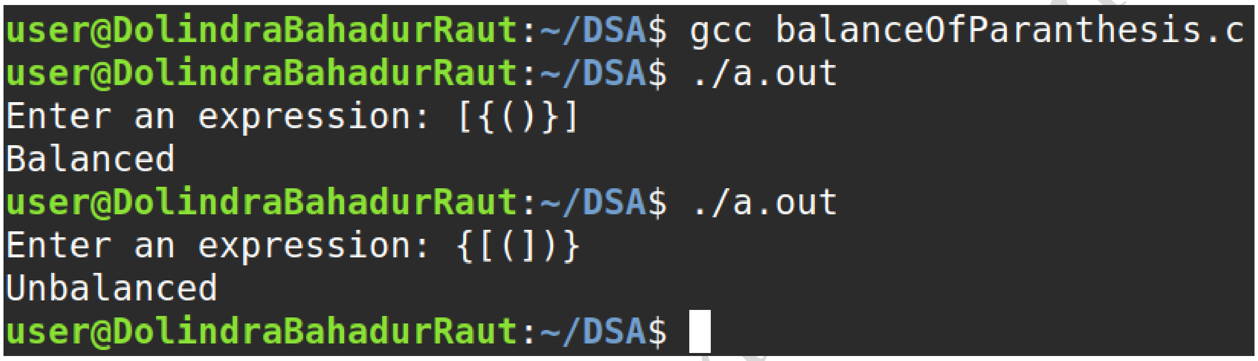
Program:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

    char expression[100];  // Array to store the expression

    int top = -1;  // Stack pointer

    char stack[100];  // Stack to store opening parentheses

    printf("Enter an expression: ");  // Take input from the user

    fgets(expression, sizeof(expression), stdin);  // Using fgets to take input, allows spaces

    // Remove newline character if it's present at the end of input

    expression[strcspn(expression, "\n")] = '\0';
```

```
for (int i = 0; expression[i] != '\0'; i++) {
    char ch = expression[i];
    // If the character is an opening bracket, push it onto the stack
    if (ch == '(' || ch == '{' || ch == '[') {
        stack[++top] = ch;
    }
     else if (ch == ')' || ch == '}' || ch == ']') {   // If the character is a closing bracket
        // If stack is empty or the top of the stack doesn't match the closing bracket
        if (top == -1) {
            printf("Unbalanced\n");
            return 0;  // Exit early as the expression is unbalanced
        }
        char top_element = stack[top--];
        // Check for matching opening bracket
        if ((ch == ')' && top_element != '(') ||
            (ch == '}' && top_element != '{') ||
            (ch == ']' && top_element != '[')) {
            printf("Unbalanced\n");
            return 0;  // Exit early as the expression is unbalanced
        }
    }
}
 if (top == -1) {  // If the stack is empty after processing all characters, it's balanced
```

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

    } else {

        printf("Unbalanced\n");
    }
    return 0;
}
```

Output:



```
user@DolindraBahadurRaut:~/DSA$ gcc balanceOfParanthesis.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
Enter an expression: [{()}]
Balanced
user@DolindraBahadurRaut:~/DSA$ ./a.out
Enter an expression: {[(])}
Unbalanced
user@DolindraBahadurRaut:~/DSA$
```

Conclusion:

Hence, we have successfully provided a program that checks whether an expression has balanced parentheses using stack.

Question 7:

Write a program to convert an Infix expression to Postfix using stack.

Answer:

Algorithm:

Step 1: Start.

Step 2: Initialize an empty stack for operators and an empty string (array of character) for the postfix expression.

Step 3: Scan the infix expression from left to right, one character at a time.

Step 4: If the scanned character is an operand append it directly to the postfix expression.

Step 5: if the scanned character is operator, do the following:

- If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.
- Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto the stack.

Step 6: If the scanned character is a '(', push it to the stack.

Step 7: If the scanned character is a ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.

Step 8: Repeat steps 4-7 until the infix expression is scanned.

Step 9: Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.

Step 10: print the postfix expression.

Step 11: End.

Example:

let's assume, the infix expression= (A - B) + C * D / E

- Push '(' to stack, append operands (A, B) to postfix.
- Pop - after), discard (, then push + to stack.
- Append C to postfix, push * to stack (higher precedence than +).
- Append D to postfix, then push / (equal precedence to *).
- Append E to postfix, pop / and * from stack to postfix.
- Postfix Expression: A B - C D * E / +

Program:

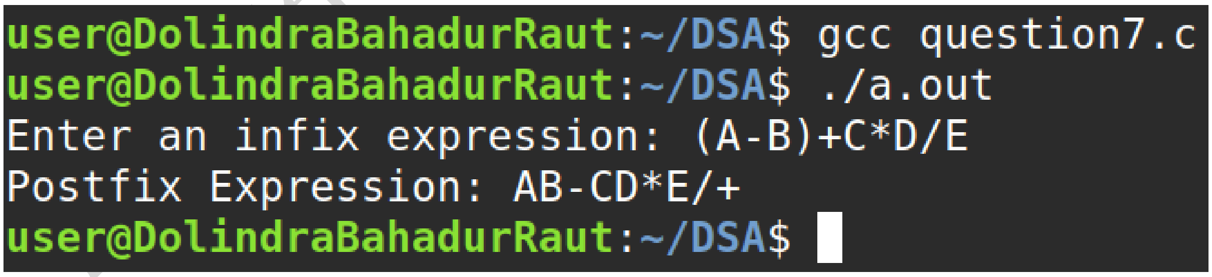
```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>
```

```
#include <string.h>

int prec(char c);

int isOperand(char c);

void infixToPostfix(char* exp);

int main() {

   char exp[100];

   printf("Enter an infix expression: ");

   fgets(exp, sizeof(exp), stdin);  // Read the expression including spaces if needed

   exp[strcspn(exp, "\n")] = 0;      // Remove the newline character at the end if present

   infixToPostfix(exp);      // Perform the conversion from infix to postfix

   return 0;
}
// Function to return precedence of operators
int prec(char c) {

   if (c == '^')

      return 3;

   else if (c == '*' || c == '/')

      return 2;

   else if (c == '+' || c == '-')

      return 1;

   else

      return -1;

}

// Function to check if a character is an operand (alphanumeric).
int isOperand(char c) {
   return isalnum(c);
```

```
}

void infixToPostfix(char* exp) {  // Function to perform infix to postfix conversion

   int len = strlen(exp);

   char result[len + 1];

   char stack[len];

   int j = 0;  // Index for the result

   int top = -1;  // Index for the stack

   for (int i = 0; i < len; i++) {

      char c = exp[i];

      if (isOperand(c)) {  // If the scanned character is an operand, add it to the output string.

         result[j++] = c;

      }

      else if (c == '(') {   // If the scanned character is '(', push it to the stack.

         stack[++top] = c;

      }

       // If the scanned character is ')', pop and add to the output string from the stack until an
'(' is encountered.
      else if (c == ')') {

         while (top != -1 && stack[top] != '(') {

            result[j++] = stack[top--];

         }

         top--;  // Pop the '(' from the stack

      }
```

```
        else if (c == '+' || c == '-' || c == '*' || c == '/' || c == '^') {    // If an operator is scanned

            // Handle precedence and associativity
              while (top != -1 && (prec(c) < prec(stack[top]) || (prec(c) == prec(stack[top]) && c
!= '^'))) {
                result[j++] = stack[top--];

            }

            stack[++top] = c;

        }

    }

// Pop all the remaining elements from the stack
    while (top != -1) {
        result[j++] = stack[top--];

    }

    result[j] = '\0';  // Null-terminate the postfix expression

    printf("Postfix Expression: %s\n", result);

}
```

Output:



```
user@DolindraBahadurRaut:~/DSA$ gcc question7.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
Enter an infix expression: (A-B)+C*D/E
Postfix Expression: AB-CD*E/+
user@DolindraBahadurRaut:~/DSA$
```

Conclusion:

Hence, we successfully demonstrated the conversion of an infix expression to a postfix expression using a stack.

Question 8:

Define Queue. Write a program to demonstrate Array implementation of Queue (Linear) with the following operations:

1. **Enqueue.**
2. **Dequeue.**
3. **Front.**
4. **IsEmpty.**

Answer:

Queue is defined as a list or collection with the restriction that insertion can be performed at one end (rear) and deletion can be performed at other end (front). It follows the FIFO(first-in first out) principle.

Algorithm:

1. Initialize Queue:
    - Define a macro MAX_Size for maximum size of queue.
    - Define an array Queue[MAX_SIZE].
    - Set front = -1 and rear = -1 to initialize the queue as empty.
2. Enqueue Operation:
    - Input a value to be inserted into the queue.
    - If rear == MAX_SIZE - 1, print "Queue is full!" (overflow condition).
    - If the queue is empty (front == -1), set front = 0.
    - Increment rear by 1 and insert the value at queue[rear].
3. Dequeue Operation:
    - Output: Remove and return the front element of the queue.
    - If front == -1, print "Queue is empty!" (underflow condition).
    - Store the element at queue[front] and increment front by 1.
    - If front > rear, reset front = rear = -1 to indicate the queue is now empty.
4. Front Operation:

- Output: Return the element at the front of the queue.
- If front == -1, print "Queue is empty!" (queue has no elements).
- Otherwise, return Queue[front] (the front element of the queue).

5. IsEmpty Operation:

- Output: Check if the queue is empty.
- If front == -1, return true (queue is empty).
- Otherwise, return false (queue is not empty).

Example:

1.Enqueue (50), Queue: 50 2. Enqueue (55), Queue: 50, 55

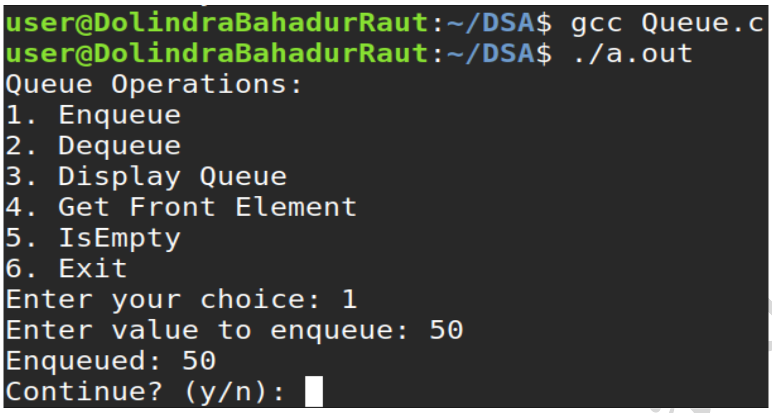
3. Dequeue (), Queue: 55 4. Enqueue (70), Queue: 55, 70

5. IsEmpty ( ), Queue is not empty. 6. Front ( ), Front element = 55

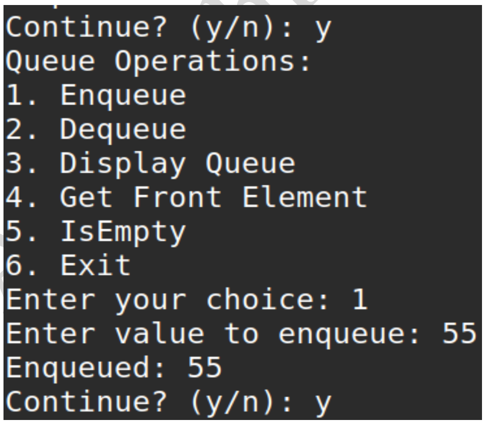
Program:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 5  // Define the maximum size of the queue

int Queue[MAX];  // Array to store the queue elements

int front = -1, rear = -1;

int isEmpty() {

   return front == -1 || front > rear;

}

void enqueue(int value) {

   if (rear == MAX - 1) {  // Check if the queue is full

      printf("Queue is full!\n");

   } else {
      if (front == -1) {
          front = 0;
      }
      rear++;
      Queue[rear] = value;
      printf("Enqueued: %d\n", value);
   }
}
int dequeue() {
   if (isEmpty()) {
      printf("Queue is empty!\n");
      return -1;  // Return -1 if the queue is empty
   } else {
      int value = Queue[front];
      if (front == rear) {
          front = rear = -1;
      } else {
          front++;
      return value;
   }
}
}
int getFront() {
   if (isEmpty()) {
      printf("Queue is empty!\n");
      return -1;
   }
   return Queue[front];
}
void displayQueue() {
   if (isEmpty()) {
      printf("Queue is empty!\n");
   } else {
      printf("Queue : ");
      for (int i = front; i <= rear; i++) {
          printf("%d ", Queue[i]);
      }
      printf("\n");
   }
}
int main() {
   int value, choice;
   char continueChoice;
   do {
      printf("Queue Operations:\n");
      printf("1. Enqueue\n");
      printf("2. Dequeue\n");
      printf("3. Display Queue\n");
      printf("4. Get Front Element\n");
      printf("5. IsEmpty\n");
      printf("6. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch(choice) {

          case 1:
              printf("Enter value to enqueue: ");
              scanf("%d", &value);
              enqueue(value);

              break;

          case 2:
              printf("Dequeued: %d\n", dequeue());
              break;

          case 3:
              displayQueue();
              break;

          case 4:
              printf("Front element: %d\n", getFront());
              break;

          case 5:

              if (isEmpty()) {

                  printf("Queue is empty!\n");

              } else {

                  printf("Queue is not empty.\n");

              }
              break;

          case 6:
              printf("Exiting the program.\n");
              exit(0);  // Exit the program

          default:
              printf("Invalid choice!\n");
      }
      printf("Continue? (y/n): ");
      getchar();
      scanf("%c", &continueChoice);

  } while (continueChoice == 'y' || continueChoice == 'Y');

  return 0;
}
```

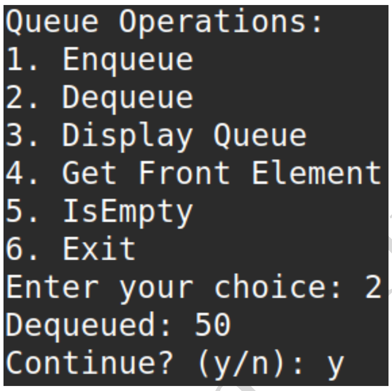
Output:



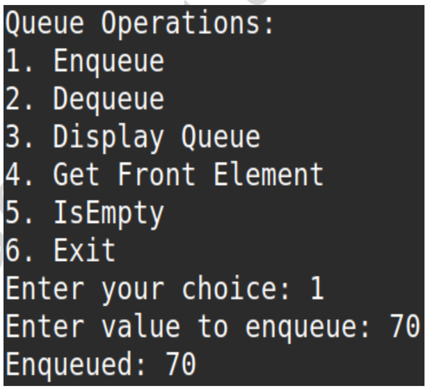
```
user@DolindraBahadurRaut:~/DSA$ gcc Queue.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Get Front Element
5. IsEmpty
6. Exit
Enter your choice: 1
Enter value to enqueue: 50
Enqueued: 50
Continue? (y/n):
```



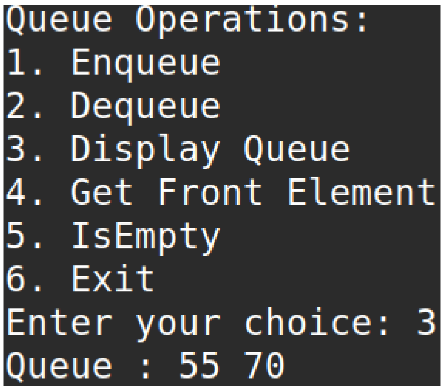
```
Continue? (y/n): y
Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Get Front Element
5. IsEmpty
6. Exit
Enter your choice: 1
Enter value to enqueue: 55
Enqueued: 55
Continue? (y/n): y
```



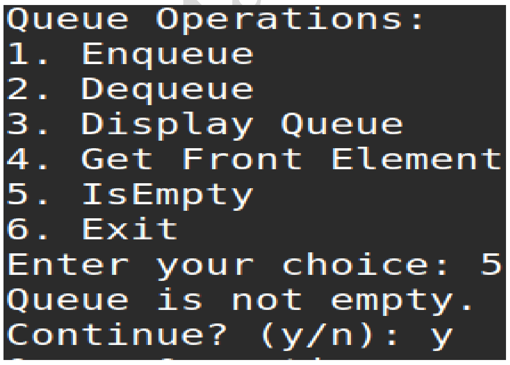
```
Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Get Front Element
5. IsEmpty
6. Exit
Enter your choice: 2
Dequeued: 50
Continue? (y/n): y
```



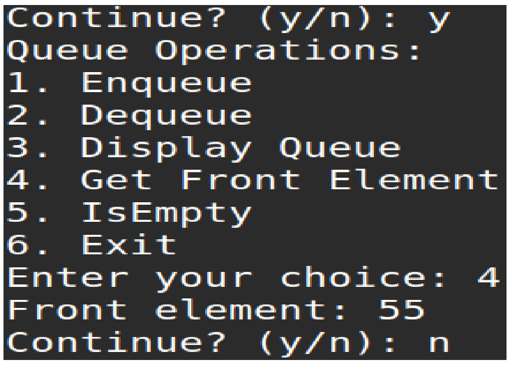
```
Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Get Front Element
5. IsEmpty
6. Exit
Enter your choice: 1
Enter value to enqueue: 70
Enqueued: 70
```



```
Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Get Front Element
5. IsEmpty
6. Exit
Enter your choice: 3
Queue : 55 70
```



```
Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Get Front Element
5. IsEmpty
6. Exit
Enter your choice: 5
Queue is not empty.
Continue? (y/n): y
```



```
Continue? (y/n): y
Queue Operations:
1. Enqueue
2. Dequeue
3. Display Queue
4. Get Front Element
5. IsEmpty
6. Exit
Enter your choice: 4
Front element: 55
Continue? (y/n): n
```

Conclusion:

Hence, we successfully implemented the Enqueue, Dequeue, Front and IsEmpty operations on a linear queue using array.

Question 9:

Write a program in c to implement a circular queue with Enqueue and Dequeue operation.

Answer:

Algorithm:

Step 1: Start.

Step 2: Declare a circular queue with the size of MAX, and two variables: front and rear.

- FRONT tracks the first element of the queue.
- REAR tracks the last elements of the queue.

Step 3: initially, set the value of front and rear to -1.

Step 4: Enqueue Operation:

- check if the queue is full ((rear + 1) % MAX = front).
- for the first element, set value front = rear = 0.
- circularly increase the rear index i.e. rear = (rear + 1) % MAX.
- add the new element in the position pointed to by rear.

Step 4: Dequeue Operation:

- check if the queue is empty.
- return the value pointed by front.
- circularly increase the front index i.e. front = (front + 1) % MAX.
- for the last element, set front = rear = -1.

Step 5: print the elements of the circular queue.

Step 6: End.

Example:

Let's assume the size of the circular queue is 5. Initially, Queue is empty. Now, insert 11in the queue at first. Then, insert 12. After inserting 12, insert 13. Then, insert 14, and at last insert 15.

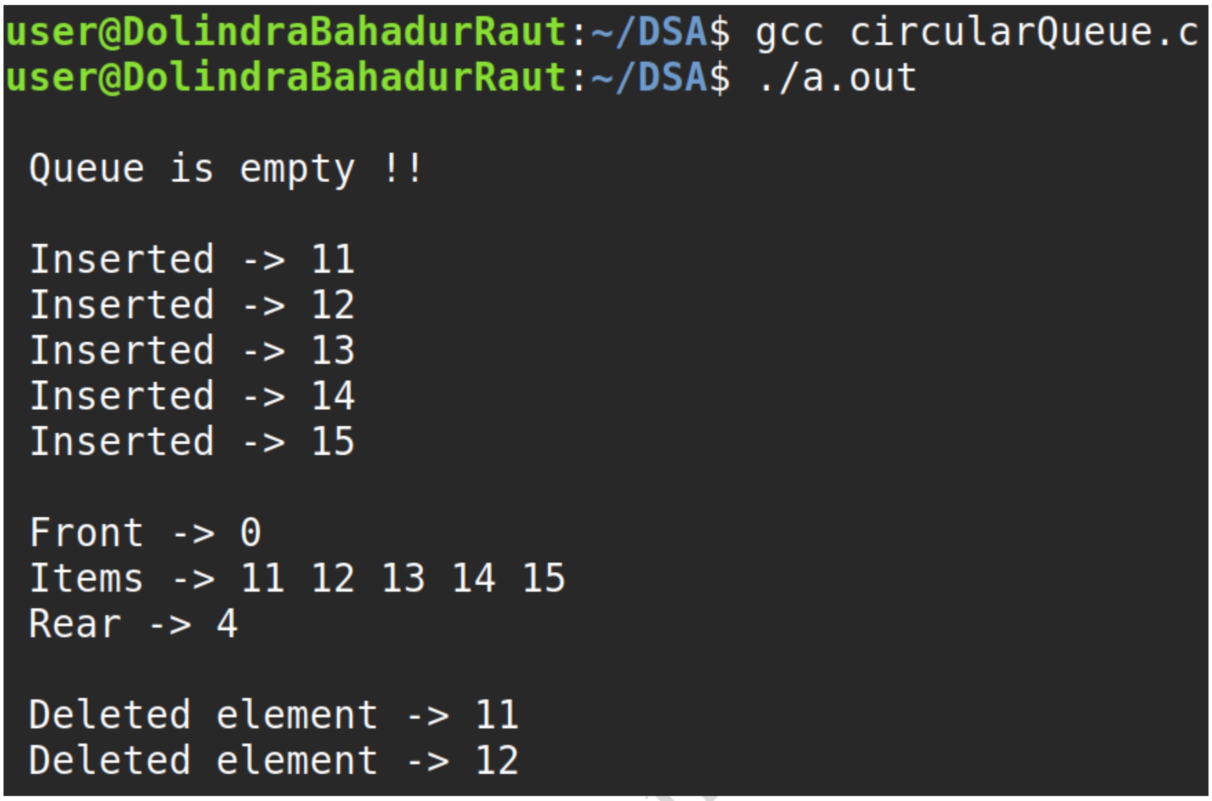
Queue elements: 11, 12, 13, 14, 15 where front = 0 and rear = 4.

Hence, the queue is full i.e rear = 4. Now, delete 11. Then delete 12. Now the elements in Queue will be: 13, 14, 15. Now insert 16 in the queue. so rear = (4+1)%5 = 0.So,16 is inserted at the index of 0. Therefore, Queue elements: 13, 14, 15, 16.

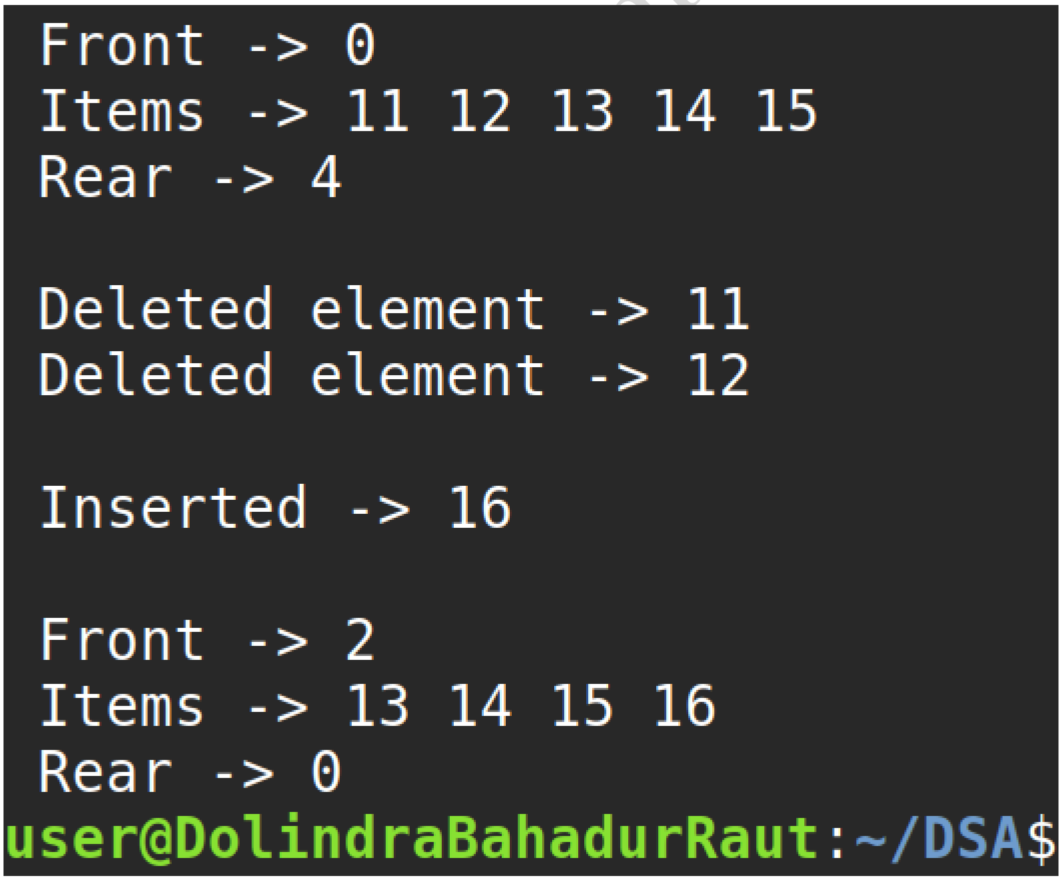
Program:

```
#include <stdio.h>

#define MAX 5

int Queue[MAX];

int front = -1, rear = -1;

int isFull() {

 if ((front == (rear + 1) % MAX) || (front == 0 && rear == MAX - 1)) return 1;

 return 0;

}

int isEmpty() {

 if (front == -1) return 1;

 return 0;

}

void enQueue(int element) {

 if (isFull())

   printf("\n Queue is full!! \n");

 else {
   if (front == -1) front = 0;
   rear = (rear + 1) % MAX;
   Queue[rear] = element;
   printf("\n Inserted -> %d", element);
 }
}
int deQueue() {
 int element;
 if (isEmpty()) {
   printf(" Queue is empty !! \n");
   return (-1);
 } else {
   element = Queue[front];
   if (front == rear) {
     front = -1;
     rear = -1;
   }
   else {
     front = (front + 1) % MAX;
   }
   printf(" Deleted element -> %d \n", element);
   return (element);  }
}
void display() {
 int i;
 if (isEmpty())
   printf(" Empty Queue\n");
 else {
   printf("\n Front -> %d ", front);
   printf("\n Items -> ");
   for (i = front; i != rear; i = (i + 1) % MAX) {
     printf("%d ", Queue[i]);
   }
   printf("%d ", Queue[i]);
   printf("\n Rear -> %d \n", rear);
 }
}
int main() {
 deQueue();  // fails because front = -1
 enQueue(11);  enQueue(12);  enQueue(13);
 enQueue(14);
 enQueue(15);
 printf("\n");
 display();
 printf("\n");
 deQueue();
 deQueue();
 enQueue(16);
 printf("\n");
 display();
 return 0;
}
```

Output:



```
user@DolindraBahadurRaut:~/DSA$ gcc circularQueue.c
user@DolindraBahadurRaut:~/DSA$ ./a.out

 Queue is empty !!

 Inserted -> 11
 Inserted -> 12
 Inserted -> 13
 Inserted -> 14
 Inserted -> 15

 Front -> 0
 Items -> 11 12 13 14 15
 Rear -> 4

 Deleted element -> 11
 Deleted element -> 12
```



```
 Front -> 0
 Items -> 11 12 13 14 15
 Rear -> 4

 Deleted element -> 11
 Deleted element -> 12

 Inserted -> 16

 Front -> 2
 Items -> 13 14 15 16
 Rear -> 0
user@DolindraBahadurRaut:~/DSA$
```

Conclusion:

Hence, we successfully created a C program to implement a circular queue with enqueue and dequeue operation.

Question 10:

Write a c program to evaluate the postfix expression using stack.

Answer:

Algorithm:

```
Step 1: Start.
Step 2: Create a stack to store operands (numbers).
Step 3: Scan the given expression from left to right and do the following :
    ● If the element is a number, push it into the stack.
    ● If the element is an operator, pop operands for the operator from the stack. Evaluate
      the operator and push the result back to the stack.
Step 4: When the expression is ended, the number in the stack is the final answer. so print the
        result stored on stack.
Step 5: End.
```

Example:

Postfix expression = 8 3 - 5 * 4 +

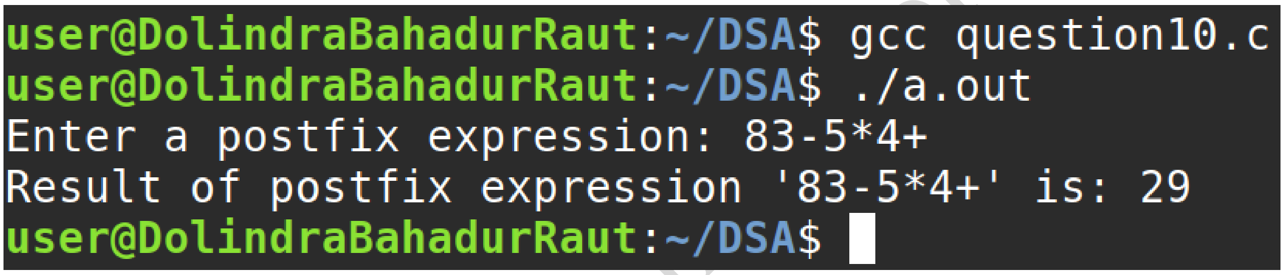
- Push 8 and 3 onto the stack. Stack = [8, 3]
- Apply - : Pop 3 and 8, calculate 8 - 3 = 5, push 5. Stack = [5]
- Push 5 onto the stack. Stack = [5, 5]
- Apply * : Pop 5 and 5, calculate 5 * 5 = 25, push 25. Stack = [25]
- Push 4, apply +: Pop 4 and 25, calculate 25 + 4 = 29, final result is 29.

Program:

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>
```

```
#define MAX 50
int stack[MAX];
int top = -1;
void push(int value) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
        exit(1);
    }
    stack[++top] = value;
}
int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        exit(1);
    }
    return stack[top--];
}
int evaluatePostfix(char* expr) {
    int i = 0;
    int operand1, operand2, result;
    while (expr[i] != '\0') {
        if (isdigit(expr[i])) {
            push(expr[i] - '0');
        }
        else {
            operand2 = pop();
            operand1 = pop();
            switch (expr[i]) {
                case '+':
                    result = operand1 + operand2;
                    break;
                case '-':
                    result = operand1 - operand2;
                    break;
                case '*':
                    result = operand1 * operand2;
                    break;
                case '/':
                    result = operand1 / operand2;
                    break;
                default:
                    printf("Invalid operator: %c\n", expr[i]);
                    exit(1);
            }
            push(result);
        }
        i++;
    }
    return pop();
}
int main() {
    char expr[MAX];
    printf("Enter a postfix expression: ");

    scanf("%s", expr);

    int result = evaluatePostfix(expr);

    printf("Result of postfix expression '%s' is: %d\n", expr, result);

    return 0;

}
```

Output:



```
user@DolindraBahadurRaut:~/DSA$ gcc question10.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
Enter a postfix expression: 83-5*4+
Result of postfix expression '83-5*4+' is: 29
user@DolindraBahadurRaut:~/DSA$ 
```

Conclusion:

Hence, we successfully implemented a C program to evaluate postfix expression using stack.