PROGRAM 1

Write A C Program to Perform these Operation on An Array.

1. Traverse

2. Insert

3. Delete

4. Linear Search

5. Binary Search

6. Binary Search Using Recursion

7. Bubble Sort

8. Insertion Sort

9. Selection Sort

10. Find Second Largest Element in Array

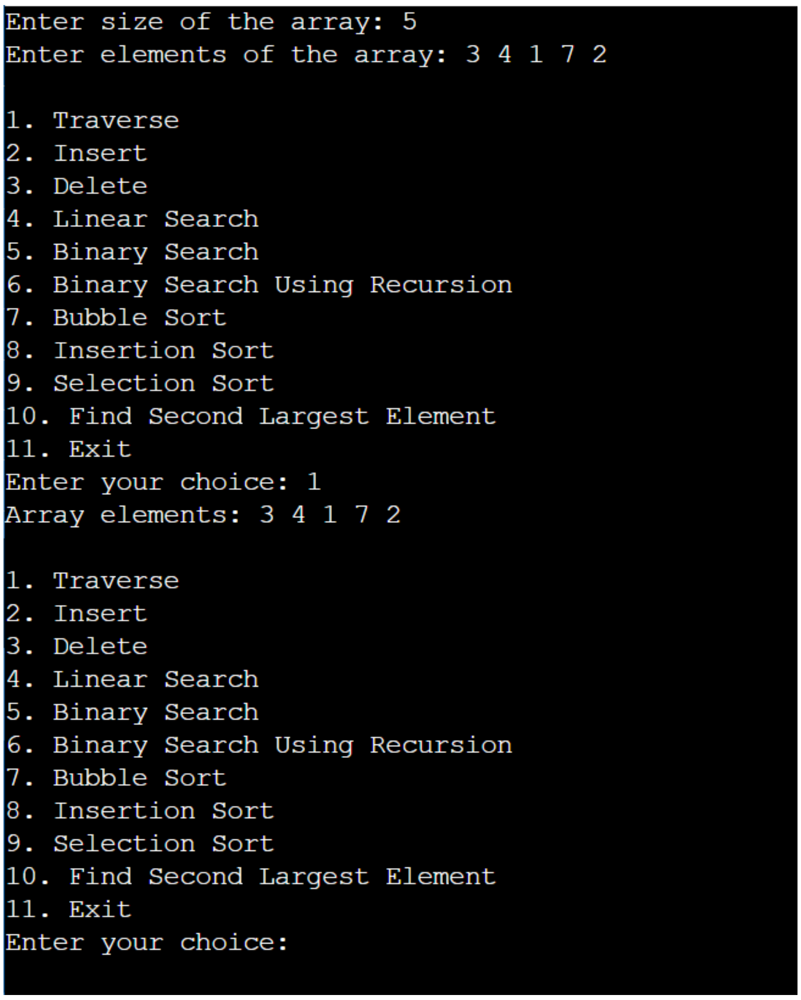
Sol –

```
#include <stdio.h>


// Function to traverse an array
void traverse(int arr[], int size) {
    printf("Array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}


// Function to insert an element at a specified position in the array
void insert(int arr[], int *size, int element, int position) {
    for (int i = *size; i > position; i--) {
        arr[i] = arr[i - 1];
    }
    arr[position] = element;
    (*size)++;
}

// Function to delete an element from a specified position in the array
void delete(int arr[], int *size, int position) {
    for (int i = position; i < *size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    (*size)--;
}

// Function to perform linear search in an array
int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1; // Element not found
}

// Function to perform binary search in a sorted array
int binarySearch(int arr[], int size, int key) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] < key) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1; // Element not found
}

// Function to perform binary search using recursion
int binarySearchRecursive(int arr[], int left, int right, int key) {
    if (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] < key) {
            return binarySearchRecursive(arr, mid + 1, right, key);
        } else {
            return binarySearchRecursive(arr, left, mid - 1, key);
        }
    }
    return -1; // Element not found
}

// Function to perform bubble sort
```

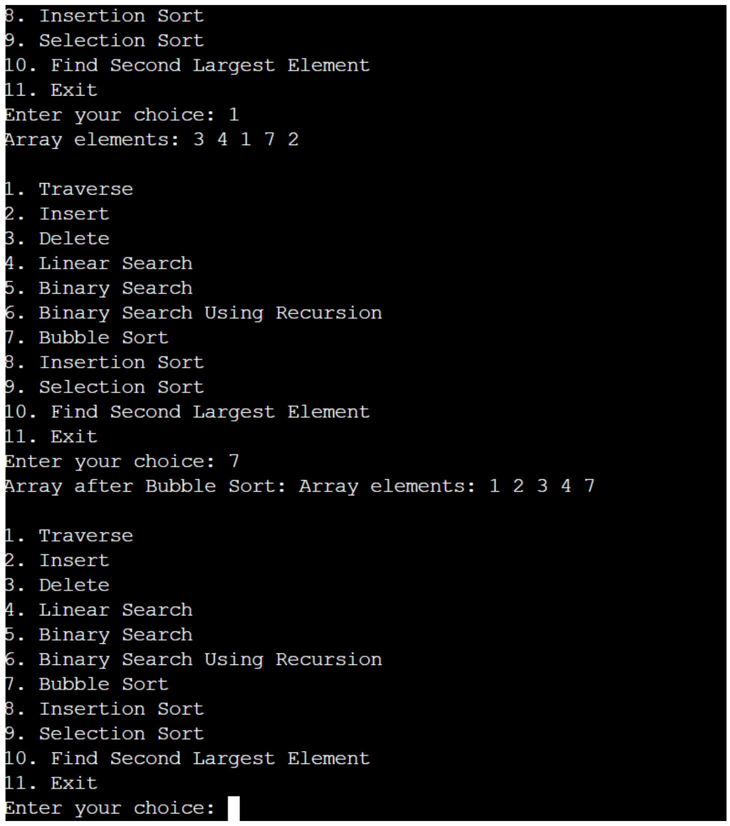
```
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to perform insertion sort
void insertionSort(int arr[], int size) {
    for (int i = 1; i < size; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// Function to perform selection sort
void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        // Swap arr[i] and arr[min_index]
        int temp = arr[i];
        arr[i] = arr[min_index];
        arr[min_index] = temp;
    }
}

// Function to find the second largest element in an array
int findSecondLargest(int arr[], int size) {
    int max = arr[0];
    int second_max = arr[1];
    if (second_max > max) {
        max = arr[1];
        second_max = arr[0];
    }
    for (int i = 2; i < size; i++) {
        if (arr[i] > max) {
            second_max = max;
            max = arr[i];
        } else if (arr[i] > second_max && arr[i] != max) {
            second_max = arr[i];
        }
    }
    return second_max;
}

int main() {
    int arr[100], size, choice, element, position, key, result;

    printf("Enter size of the array: ");
    scanf("%d", &size);
    printf("Enter elements of the array: ");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    while (1) {
        printf("\n1. Traverse\n");
        printf("2. Insert\n");
        printf("3. Delete\n");
        printf("4. Linear Search\n");
        printf("5. Binary Search\n");
        printf("6. Binary Search Using Recursion\n");
        printf("7. Bubble Sort\n");
        printf("8. Insertion Sort\n");
        printf("9. Selection Sort\n");
        printf("10. Find Second Largest Element\n");
        printf("11. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);


    switch (choice) {
      case 1:
        traverse(arr, size);
        break;
      case 2:
        printf("Enter element to insert: ");
        scanf("%d", &element);
        printf("Enter position to insert: ");
        scanf("%d", &position);
        insert(arr, &size, element, position);
        break;
      case 3:
        printf("Enter position to delete: ");
        scanf("%d", &position);
        delete(arr, &size, position);
        break;
      case 4:
        printf("Enter key to search: ");
        scanf("%d", &key);
        result = linearSearch(arr, size, key);
        if (result != -1) {
          printf("Element found at index %d\n", result);
        } else {
          printf("Element not found\n");
        }
        break;
    case 5:
        printf("Enter key to search: ");
        scanf("%d", &key);
        result = binarySearch(arr, size, key);
        if (result != -1) {
            printf("Element found at index %d\n", result);
        } else {
            printf("Element not found\n");
        }
        break;
    case 6:
        printf("Enter key to search: ");
        scanf("%d", &key);
        result = binarySearchRecursive(arr, 0, size - 1, key);
        if (result != -1) {
            printf("Element found at index %d\n", result);
        } else {
            printf("Element not found\n");
        }
        break;
    case 7:
        bubbleSort(arr, size);
        printf("Array after Bubble Sort: ");
        traverse(arr, size);
        break;
    case 8:
        insertionSort(arr, size);
```

```
            printf("Array after Insertion Sort: ");
            traverse(arr, size);
            break;
        case 9:
            selectionSort(arr, size);
            printf("Array after Selection Sort: ");
            traverse(arr, size);
            break;
        case 10:
            result = findSecondLargest(arr, size);
            printf("Second largest element in the array: %d\n", result);
            break;
        case 11:
            printf("Exiting program.\n");
            return 0;
        default:
            printf("Invalid choice\n");
        }
    }

    return 0;
}
```

OUTPUT



```
Enter size of the array: 5
Enter elements of the array: 3 4 1 7 2

1. Traverse
2. Insert
3. Delete
4. Linear Search
5. Binary Search
6. Binary Search Using Recursion
7. Bubble Sort
8. Insertion Sort
9. Selection Sort
10. Find Second Largest Element
11. Exit
Enter your choice: 1
Array elements: 3 4 1 7 2

1. Traverse
2. Insert
3. Delete
4. Linear Search
5. Binary Search
6. Binary Search Using Recursion
7. Bubble Sort
8. Insertion Sort
9. Selection Sort
10. Find Second Largest Element
11. Exit
Enter your choice:
```



```
8. Insertion Sort
9. Selection Sort
10. Find Second Largest Element
11. Exit
Enter your choice: 1
Array elements: 3 4 1 7 2

1. Traverse
2. Insert
3. Delete
4. Linear Search
5. Binary Search
6. Binary Search Using Recursion
7. Bubble Sort
8. Insertion Sort
9. Selection Sort
10. Find Second Largest Element
11. Exit
Enter your choice: 7
Array after Bubble Sort: Array elements: 1 2 3 4 7

1. Traverse
2. Insert
3. Delete
4. Linear Search
5. Binary Search
6. Binary Search Using Recursion
7. Bubble Sort
8. Insertion Sort
9. Selection Sort
10. Find Second Largest Element
11. Exit
Enter your choice:
```

PROGRAM 2

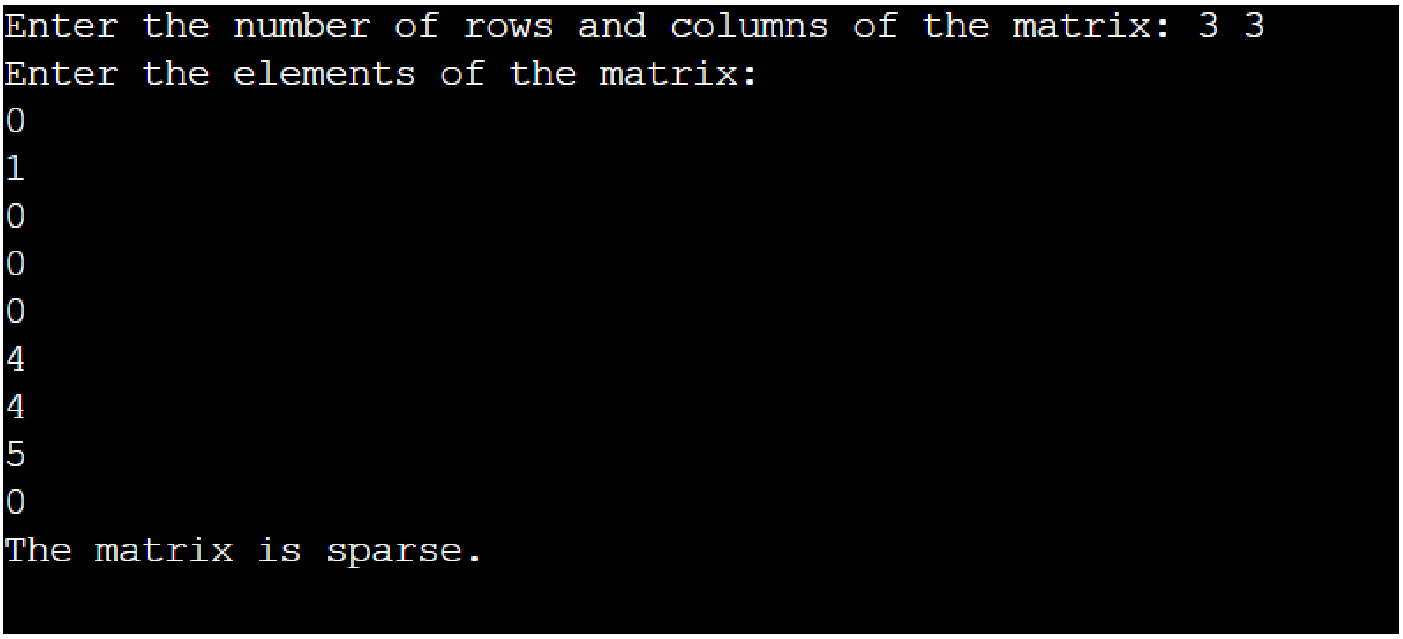
Write A C Program to Check Whether a Matrix Is Sparse or Not.

SOL –

```
#include <stdio.h>


#define MAX_ROWS 100

#define MAX_COLS 100


int main() {

   int matrix[MAX_ROWS][MAX_COLS];

   int rows, cols;

   int i, j, count_zeros = 0;


   // Input the number of rows and columns of the matrix

   printf("Enter the number of rows and columns of the matrix: ");

   scanf("%d %d", &rows, &cols);


   // Input the elements of the matrix

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

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

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

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

         if (matrix[i][j] == 0) {

            count_zeros++; // Counting zeros

         }

      }

   }
```

```
    // Checking if the matrix is sparse
    if (count_zeros > (rows * cols) / 2) {
        printf("The matrix is sparse.\n");
    } else {
        printf("The matrix is not sparse.\n");
    }

    return 0;
}
```

OUTPUT –



```
Enter the number of rows and columns of the matrix: 3 3
Enter the elements of the matrix:
0
1
0
0
0
4
4
5
0
The matrix is sparse.
```

PROGRAM 3

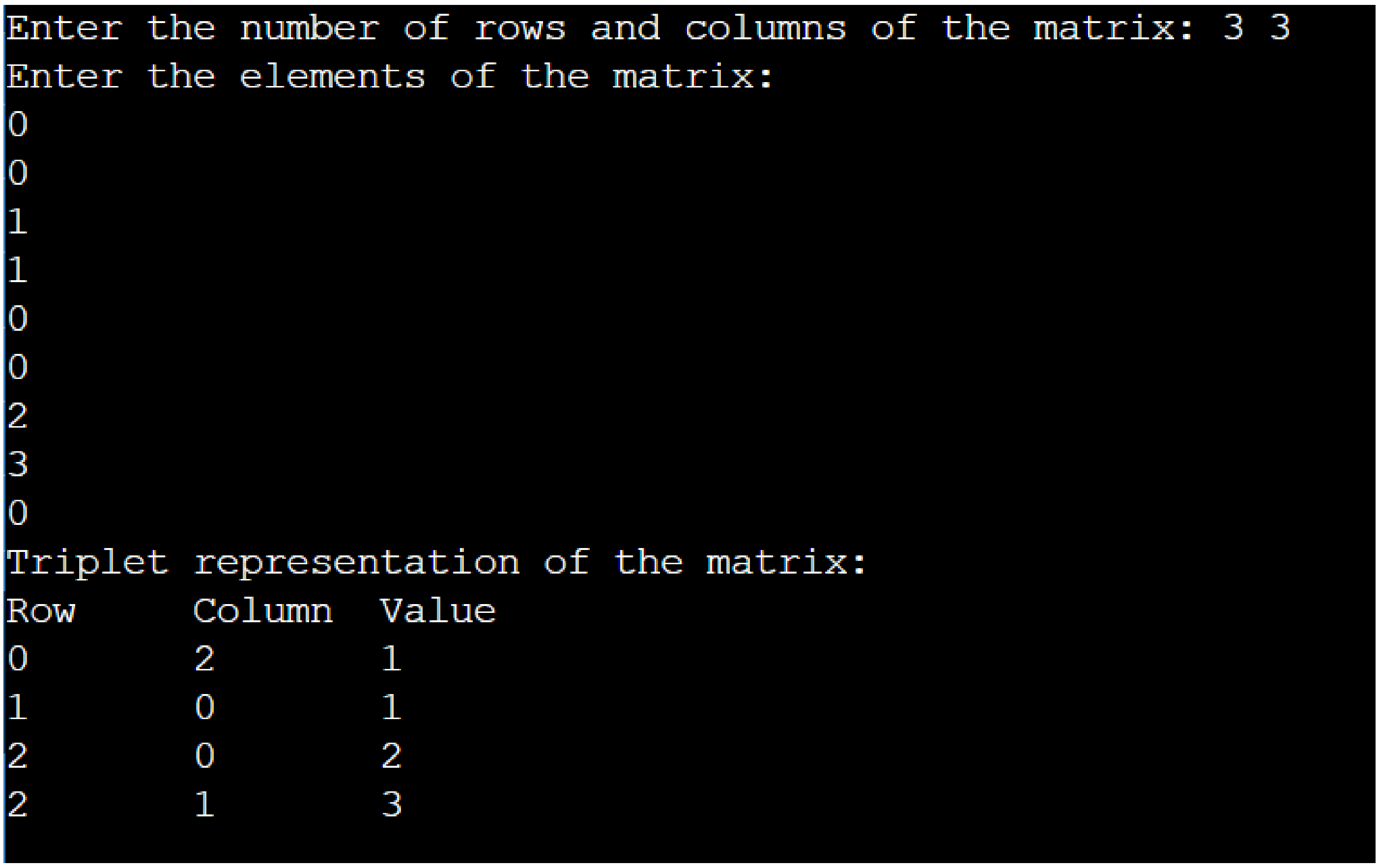
Write A C Program to Covert Sparse Matrix Into 3- Triplet Representation

SOL –

```
#include <stdio.h>


#define MAX_ROWS 100

#define MAX_COLS 100


// Structure to represent a triplet

struct Triplet {

   int row;

   int col;

   int value;

};


int main() {

   int matrix[MAX_ROWS][MAX_COLS];

   int rows, cols;

   int i, j, count_zeros = 0;

   struct Triplet triplets[MAX_ROWS * MAX_COLS]; // Maximum possible number of non-
zero elements


   // Input the number of rows and columns of the matrix

   printf("Enter the number of rows and columns of the matrix: ");

   scanf("%d %d", &rows, &cols);


   // Input the elements of the matrix

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

```
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            scanf("%d", &matrix[i][j]);
            if (matrix[i][j] != 0) {
                // Store non-zero elements as triplets
                triplets[count_zeros].row = i;
                triplets[count_zeros].col = j;
                triplets[count_zeros].value = matrix[i][j];
                count_zeros++; // Increment count of non-zero elements
            }
        }
    }

    // Displaying the triplet representation
    printf("Triplet representation of the matrix:\n");
    printf("Row\tColumn\tValue\n");
    for (i = 0; i < count_zeros; i++) {
        printf("%d\t%d\t%d\n", triplets[i].row, triplets[i].col, triplets[i].value);
    }

    return 0;
}
```

OUTPUT –



```
Enter the number of rows and columns of the matrix: 3 3
Enter the elements of the matrix:
0
0
1
1
0
0
2
3
0
Triplet representation of the matrix:
Row     Column  Value
0       2       1
1       0       1
2       0       2
2       1       3
```

PROGRAM 4

Write A C Program to Perform Following Operation on Matrix

1. Addition Of Two Matrix.

2. Subtraction Of Two Matrix.

3. Multiplication Of Two Matrix.

4. Transpose Of Two Matrix.

5. Diagonal Sum of Two Matrix

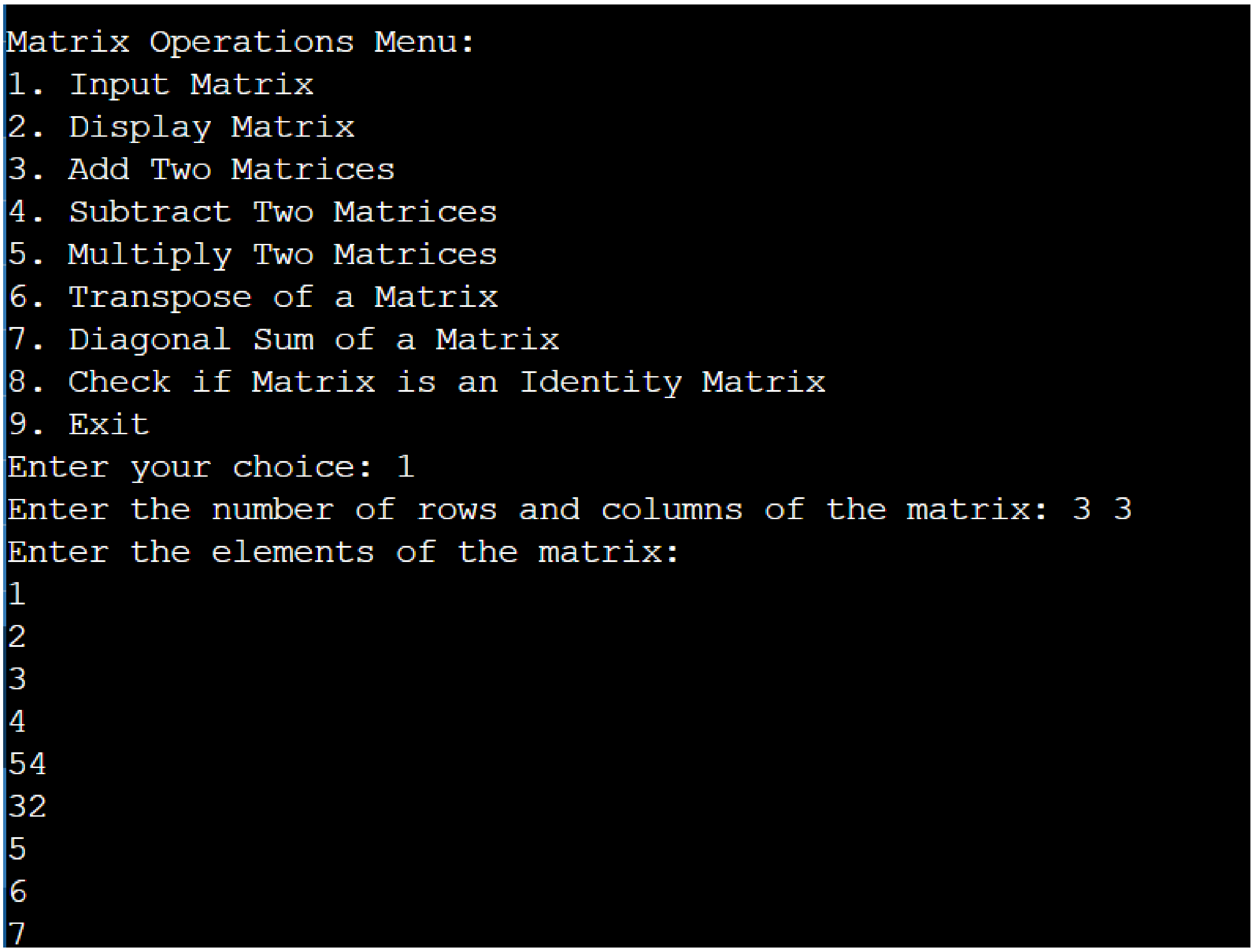
6. Check If Matrix Is Identity Or not.

SOL –

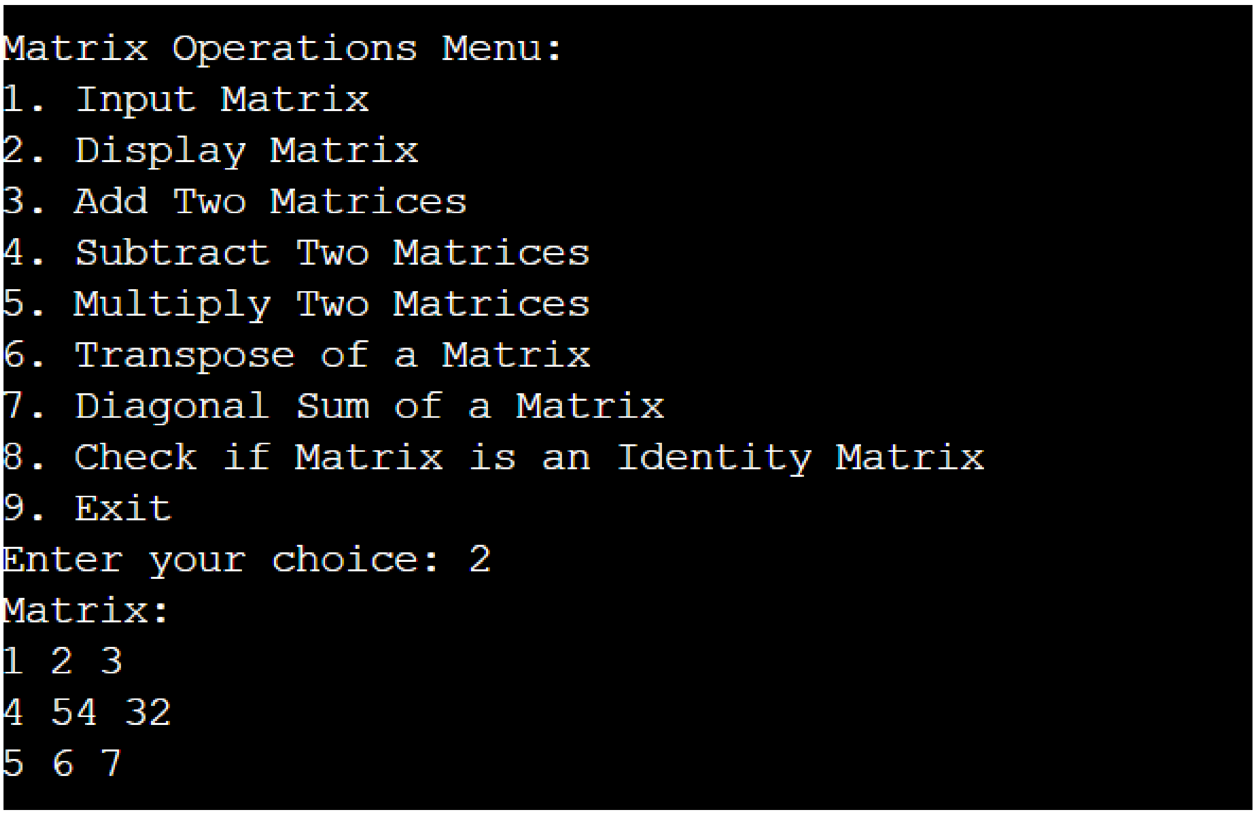
```
#include <stdio.h>


#define MAX_ROWS 10

#define MAX_COLS 10


// Function to input elements of a matrix

void inputMatrix(int mat[MAX_ROWS][MAX_COLS], int rows, int cols) {

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

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

      for (int j = 0; j < cols; j++) {

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

      }

   }

}


// Function to display elements of a matrix

void displayMatrix(int mat[MAX_ROWS][MAX_COLS], int rows, int cols) {

   printf("Matrix:\n");
```

```
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}

// Function to add two matrices
void addMatrix(int mat1[MAX_ROWS][MAX_COLS], int mat2[MAX_ROWS][MAX_COLS], int
result[MAX_ROWS][MAX_COLS], int rows, int cols) {
    printf("Resultant Matrix (Addition):\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = mat1[i][j] + mat2[i][j];
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }
}

// Function to subtract two matrices
void subtractMatrix(int mat1[MAX_ROWS][MAX_COLS], int mat2[MAX_ROWS][MAX_COLS],
int result[MAX_ROWS][MAX_COLS], int rows, int cols) {
    printf("Resultant Matrix (Subtraction):\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = mat1[i][j] - mat2[i][j];
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }
}

// Function to multiply two matrices
void multiplyMatrix(int mat1[MAX_ROWS][MAX_COLS], int mat2[MAX_ROWS][MAX_COLS],
int result[MAX_ROWS][MAX_COLS], int rows1, int cols1, int rows2, int cols2) {
    printf("Resultant Matrix (Multiplication):\n");
    for (int i = 0; i < rows1; i++) {
        for (int j = 0; j < cols2; j++) {
            result[i][j] = 0;
            for (int k = 0; k < cols1; k++) {
                result[i][j] += mat1[i][k] * mat2[k][j];
            }
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }
}

// Function to compute transpose of a matrix
void transposeMatrix(int mat[MAX_ROWS][MAX_COLS], int
transpose[MAX_ROWS][MAX_COLS], int rows, int cols) {
    printf("Transpose of Matrix:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            transpose[j][i] = mat[i][j];
            printf("%d ", transpose[j][i]);
        }
        printf("\n");
    }
}

// Function to compute diagonal sum of a matrix
int diagonalSum(int mat[MAX_ROWS][MAX_COLS], int rows, int cols) {
    int sum = 0;
    int min_dimension = rows < cols ? rows : cols;
    for (int i = 0; i < min_dimension; i++) {
        sum += mat[i][i];
    }
    return sum;
}

// Function to check if a matrix is an identity matrix
int isIdentityMatrix(int mat[MAX_ROWS][MAX_COLS], int rows, int cols) {
    if (rows != cols) {
        return 0;
    }
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (i == j && mat[i][j] != 1) {
                return 0;
            } else if (i != j && mat[i][j] != 0) {
                return 0;
            }
        }
    }
    return 1;
}


int main() {
    int mat1[MAX_ROWS][MAX_COLS], mat2[MAX_ROWS][MAX_COLS];
    int result[MAX_ROWS][MAX_COLS], transpose[MAX_ROWS][MAX_COLS];
    int rows1, cols1, rows2, cols2;
    int choice;

    do {
        printf("\nMatrix Operations Menu:\n");
        printf("1. Input Matrix\n");
        printf("2. Display Matrix\n");
        printf("3. Add Two Matrices\n");
        printf("4. Subtract Two Matrices\n");
        printf("5. Multiply Two Matrices\n");
        printf("6. Transpose of a Matrix\n");
        printf("7. Diagonal Sum of a Matrix\n");
        printf("8. Check if Matrix is an Identity Matrix\n");
        printf("9. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the number of rows and columns of the matrix: ");
                scanf("%d %d", &rows1, &cols1);
                inputMatrix(mat1, rows1, cols1);
                break;
            case 2:
                displayMatrix(mat1, rows1, cols1);
                break;
            case 3:
                printf("Enter the number of rows and columns of the second matrix: ");
                scanf("%d %d", &rows2, &cols2);
                inputMatrix(mat2, rows2, cols2);
                addMatrix(mat1, mat2, result, rows1, cols1);
                break;
            case 4:
                subtractMatrix(mat1, mat2, result, rows1, cols1);
                break;
            case 5:
                multiplyMatrix(mat1, mat2, result, rows1, cols1, rows2, cols2);
                break;
            case 6:
                transposeMatrix(mat1, transpose, rows1, cols1);
                break;
            case 7:
                printf("Diagonal Sum of the Matrix: %d\n", diagonalSum(mat1, rows1, cols1));
                break;
            case 8:
                if (isIdentityMatrix(mat1, rows1, cols1)) {
                    printf("The Matrix is an Identity Matrix.\n");
                } else {
                    printf("The Matrix is not an Identity Matrix.\n");
                }
                break;
            case 9:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
          }
       } while (choice != 9);


       return 0;
}
```

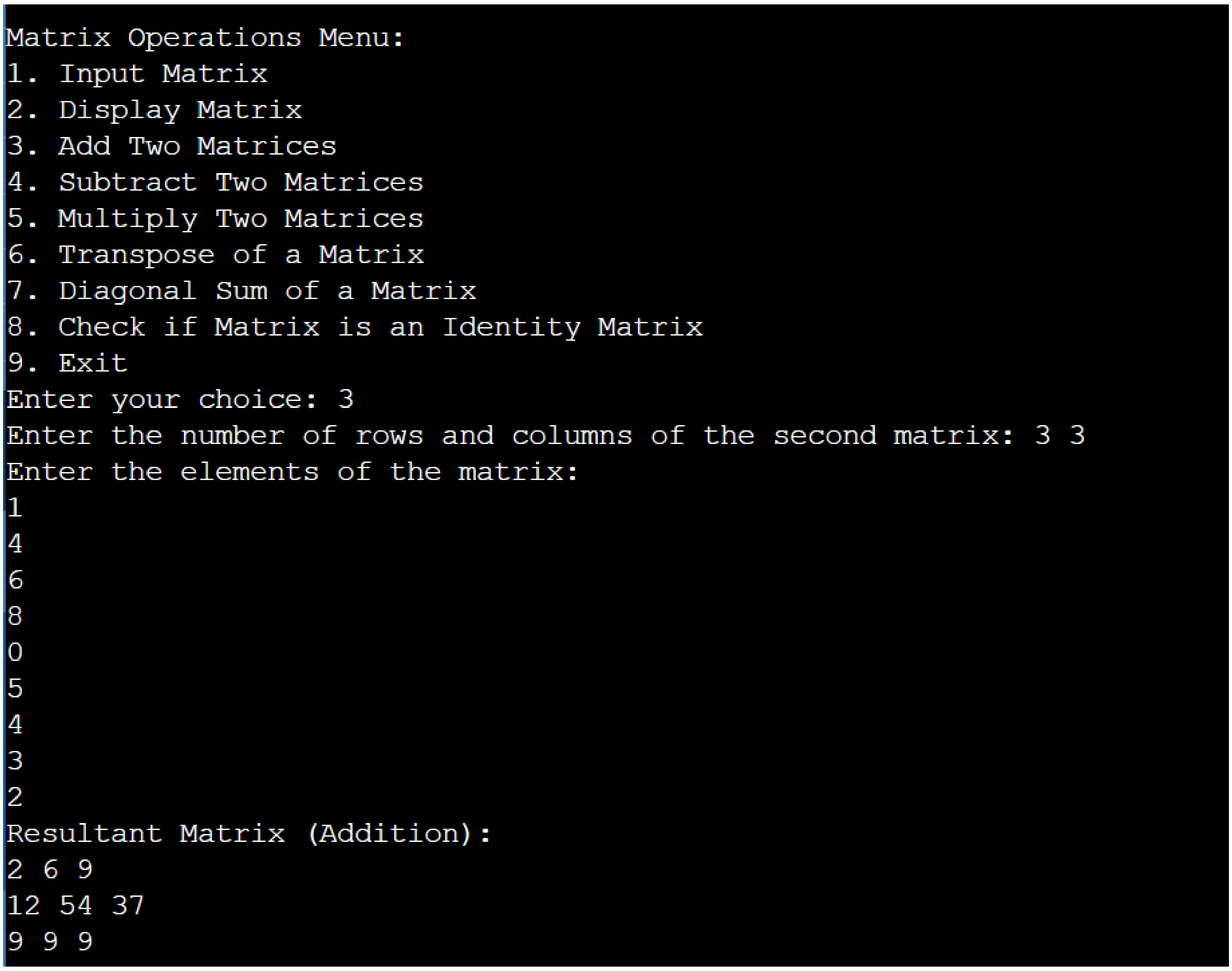
OUTPUT –



```
Matrix Operations Menu:
1. Input Matrix
2. Display Matrix
3. Add Two Matrices
4. Subtract Two Matrices
5. Multiply Two Matrices
6. Transpose of a Matrix
7. Diagonal Sum of a Matrix
8. Check if Matrix is an Identity Matrix
9. Exit
Enter your choice: 1
Enter the number of rows and columns of the matrix: 3 3
Enter the elements of the matrix:
1
2
3
4
54
32
5
6
7
```



```
Matrix Operations Menu:
1. Input Matrix
2. Display Matrix
3. Add Two Matrices
4. Subtract Two Matrices
5. Multiply Two Matrices
6. Transpose of a Matrix
7. Diagonal Sum of a Matrix
8. Check if Matrix is an Identity Matrix
9. Exit
Enter your choice: 2
Matrix:
1 2 3
4 54 32
5 6 7
```



```
Matrix Operations Menu:
1. Input Matrix
2. Display Matrix
3. Add Two Matrices
4. Subtract Two Matrices
5. Multiply Two Matrices
6. Transpose of a Matrix
7. Diagonal Sum of a Matrix
8. Check if Matrix is an Identity Matrix
9. Exit
Enter your choice: 3
Enter the number of rows and columns of the second matrix: 3 3
Enter the elements of the matrix:
1
4
6
8
0
5
4
3
2
Resultant Matrix (Addition):
2 6 9
12 54 37
9 9 9
```

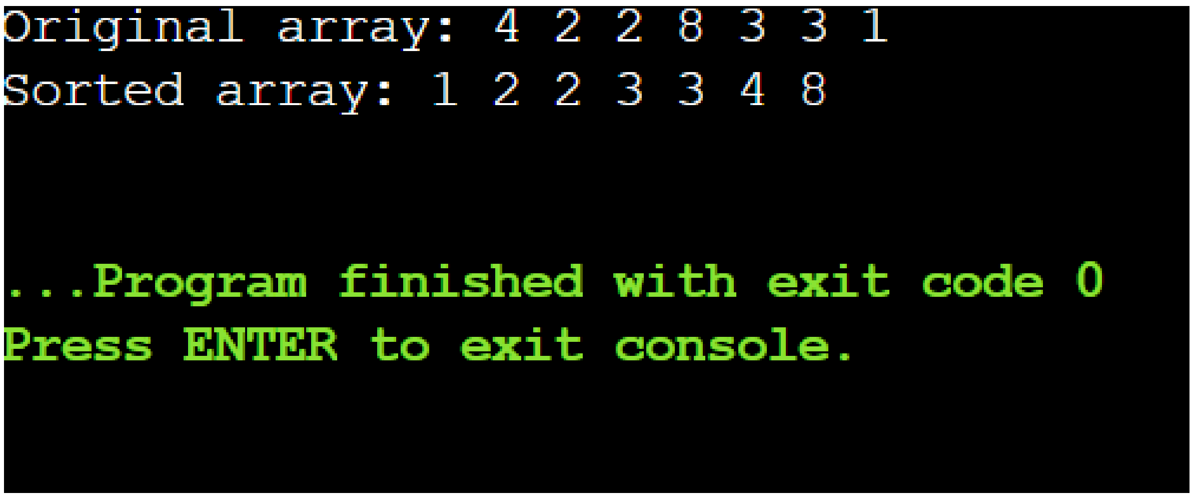
PROGRAM 5

Write A C Program to Perform Count Sort on An Unsorted Array

SOL –

```
#include <stdio.h>

// Function to find the maximum element in the array
int findMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

// Function to perform Count Sort
void countSort(int arr[], int n) {
    int max = findMax(arr, n);

    // Create a count array to store the count of each element
    int count[max + 1];
    for (int i = 0; i <= max; i++) {
        count[i] = 0;
    }

    // Count occurrences of each element
    for (int i = 0; i < n; i++) {
        count[arr[i]]++;
    }

    // Modify the count array to contain the actual position of each element in the sorted
array
    for (int i = 1; i <= max; i++) {
        count[i] += count[i - 1];
    }

    // Create a temporary array to store the sorted output
    int output[n];
    for (int i = n - 1; i >= 0; i--) {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }

    // Copy the sorted elements back to the original array
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

int main() {
    int arr[] = {4, 2, 2, 8, 3, 3, 1};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");


    countSort(arr, n);


    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");


    return 0;
}
```

OUTPUT –



```
Original array: 4 2 2 8 3 3 1
Sorted array: 1 2 2 3 3 4 8


...Program finished with exit code 0
Press ENTER to exit console.
```

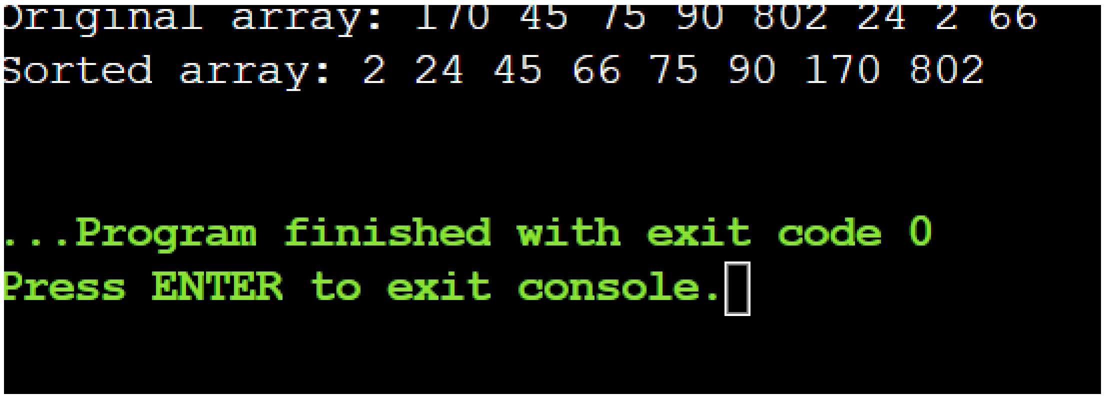
PROGRAM 6

Write A C Program to Perform Radix Sort on An Unsorted Array

SOL –

```
#include <stdio.h>

// Function to find the maximum element in the array
int findMax(int arr[], int n) {
   int max = arr[0];
   for (int i = 1; i < n; i++) {
      if (arr[i] > max) {
         max = arr[i];
      }
   }
   return max;
}

// Function to perform counting sort based on a specific digit
void countingSort(int arr[], int n, int exp) {
   int output[n];
   int count[10] = {0};

   // Store count of occurrences in count[]
   for (int i = 0; i < n; i++) {
      count[(arr[i] / exp) % 10]++;
   }

   // Change count[i] so that count[i] now contains actual position of this digit in output[]
   for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // Build the output array
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[] so that arr[] now contains sorted numbers based on
current digit
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

// Function to perform Radix Sort
void radixSort(int arr[], int n) {
    int max = findMax(arr, n);

    // Perform counting sort for every digit starting from the least significant digit to the
most significant digit
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSort(arr, n, exp);
    }
}

int main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr) / sizeof(arr[0]);


    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");


    radixSort(arr, n);


    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");


    return 0;
}
```

OUTPUT –



```
Original array: 170 45 75 90 802 24 2 66
Sorted array: 2 24 45 66 75 90 170 802


...Program finished with exit code 0
Press ENTER to exit console.
```

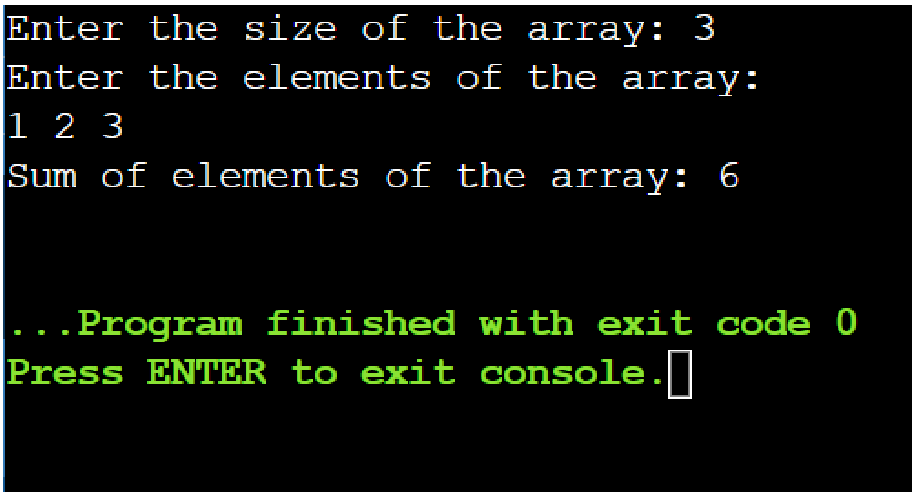
PROGRAM 7

Write A C To Dynamically Allocate an Array and Calculate Sum of Its Elements.

SOL –

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

int main() {
    int *arr; // Pointer to store the base address of the dynamically allocated array
    int n;    // Size of the array
    int sum = 0; // Variable to store the sum of elements

    // Input the size of the array
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    // Dynamically allocate memory for the array
    arr = (int *)malloc(n * sizeof(int));

    // Check if memory allocation was successful
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1; // Return 1 to indicate failure
    }

    // Input elements of the array
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }


    // Calculate the sum of elements
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }


    // Output the sum of elements
    printf("Sum of elements of the array: %d\n", sum);


    // Free dynamically allocated memory
    free(arr);


    return 0;
}
```

OUTPUT –



```
Enter the size of the array: 3
Enter the elements of the array:
1 2 3
Sum of elements of the array: 6


...Program finished with exit code 0
Press ENTER to exit console.
```

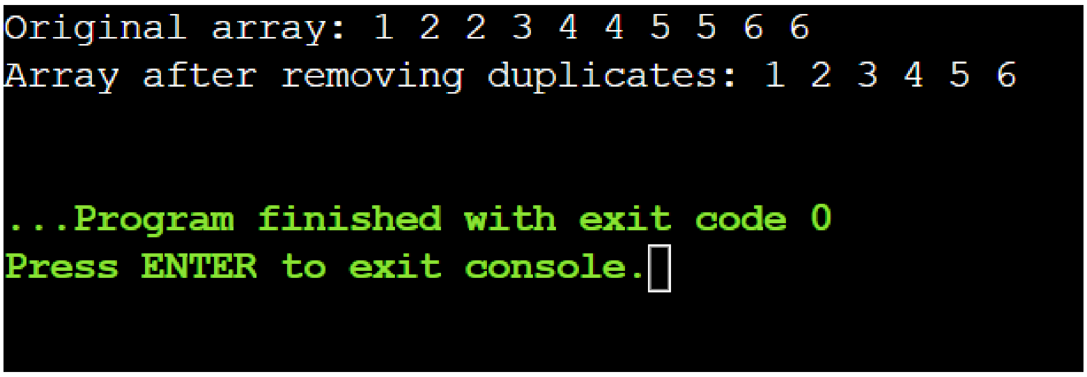
PROGRAM 8

Write A C Program to Remove Duplicate Elements from An Array

SOL –

```
#include <stdio.h>


// Function to remove duplicate elements from an array
int removeDuplicates(int arr[], int n) {
    if (n == 0 || n == 1) {
        return n;
    }

    // Create a temporary array to store unique elements
    int temp[n];
    int j = 0;

    // Traverse the original array
    for (int i = 0; i < n - 1; i++) {
        // If current element is not equal to next element, then store it
        if (arr[i] != arr[i + 1]) {
            temp[j++] = arr[i];
        }
    }

    // Store the last element of the original array
    temp[j++] = arr[n - 1];

    // Copy the contents of temp[] to arr[]
    for (int i = 0; i < j; i++) {
        arr[i] = temp[i];
    }   return j; // Return the new size of the array without duplicates
}
int main() {
    int arr[] = {1, 2, 2, 3, 4, 4, 5, 5, 6, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
   printf("Original array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]); }
    printf("\n");
    // Remove duplicates
    n = removeDuplicates(arr, n);
    printf("Array after removing duplicates: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

OUTPUT –



```
Original array: 1 2 2 3 4 4 5 5 6 6
Array after removing duplicates: 1 2 3 4 5 6


...Program finished with exit code 0
Press ENTER to exit console.
```

PROGRAM 9

Write a program to following operations on singly linked list

1. Create a Linked list

2. Addition/Insertion of a node at the end of the list.

3. Addition of a node at the beginning of linked list.

4. Addition of a node at the specific position.

5. Traversing a linked list.

6. Counting number of nodes/Length of linked list.

7. Deleting a node from beginning

8. Delete a node from last

9. Delete a node from specific position

10. Reverse A linked List.

11. Sorting.

12. Search

13. Insert a node in a sorted linked list such that linked list remains sorted.

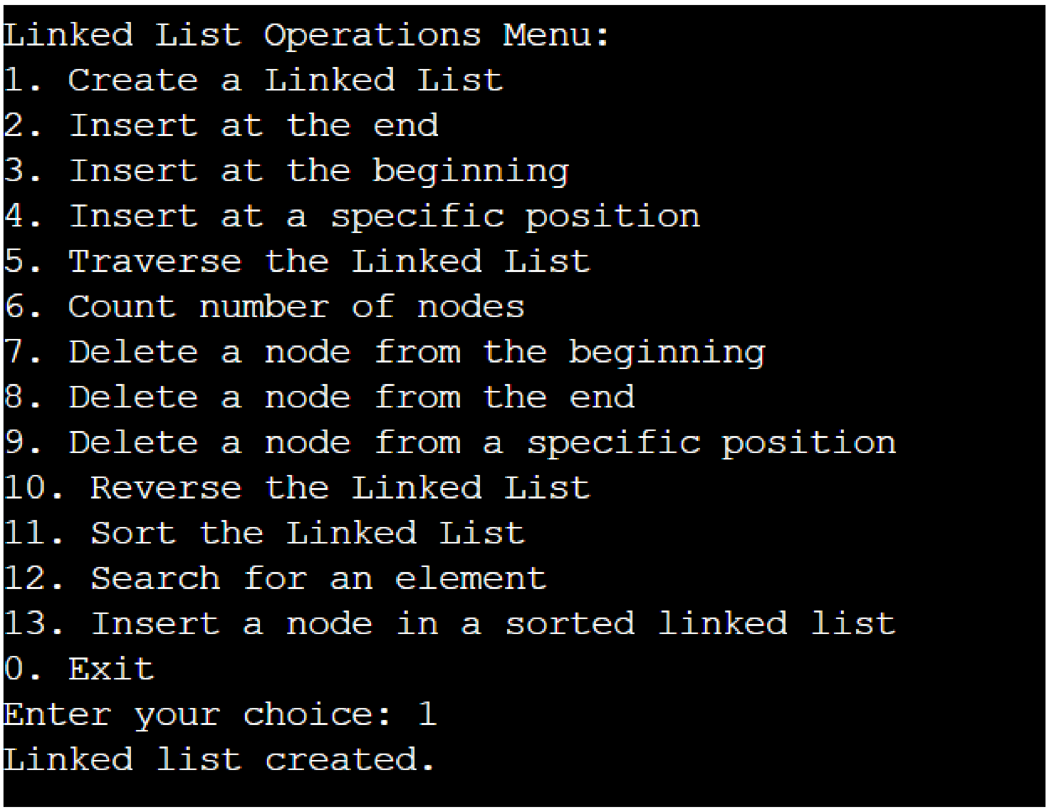
Sol –

```
#include <stdio.h>

#include <stdlib.h>


// Structure for a node

struct Node {

   int data;

   struct Node* next;

};


// Function to create a new node

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
       printf("Memory allocation failed.\n");
       exit(1);
   }
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
}

// Function to add a node at the end of the linked list
void insertAtEnd(struct Node* head, int data) {
   struct Node* newNode = createNode(data);
   while (head->next != NULL) {
       head = head->next;
   }
   head->next = newNode;
}

// Function to add a node at the beginning of the linked list
struct Node* insertAtBeg(struct Node* head, int data) {
   struct Node* newNode = createNode(data);
   newNode->next = head;
   return newNode;
}

// Function to add a node at a specific position in the linked list
void insertAtPosition(struct Node* head, int data, int position) {
   if (position < 0) {
       printf("Invalid position.\n");
       return;
   }
   if (position == 0) {
       printf("Cannot insert at position 0. Use prepend instead.\n");
       return;
   }
   struct Node* newNode = createNode(data);
   struct Node* current = head;
   for (int i = 0; i < position - 1 && current != NULL; i++) {
       current = current->next;
   }
   if (current == NULL) {
       printf("Invalid position.\n");
       return;
   }
   newNode->next = current->next;
   current->next = newNode;
}

// Function to traverse and print the linked list
void traverse(struct Node* head) {
   printf("Linked list: ");
   while (head != NULL) {
       printf("%d ", head->data);
       head = head->next;
   }
   printf("\n");
}

// Function to count the number of nodes in the linked list
int countNodes(struct Node* head) {
   int count = 0;
   while (head != NULL) {
       count++;
       head = head->next;
   }
   return count;
}

// Function to delete the first node from the linked list
struct Node* deleteFirstNode(struct Node* head) {
   if (head == NULL) {
       printf("Linked list is empty.\n");
       return NULL;
   }
   struct Node* temp = head;
   head = head->next;
   free(temp);
   return head;
}

// Function to delete the last node from the linked list
struct Node* deleteLastNode(struct Node* head) {
   if (head == NULL) {
       printf("Linked list is empty.\n");
       return NULL;
   }
   if (head->next == NULL) {
       free(head);
       return NULL;
   }
   struct Node* current = head;
   struct Node* prev = NULL;
   while (current->next != NULL) {
       prev = current;
       current = current->next;
   }
   free(current);
   prev->next = NULL;
   return head;
}

// Function to delete a node at a specific position from the linked list
struct Node* deleteAtPosition(struct Node* head, int position) {
   if (head == NULL) {
       printf("Linked list is empty.\n");
       return NULL;
   }
   if (position < 0) {
       printf("Invalid position.\n");
       return head;
   }
   if (position == 0) {
       printf("Cannot delete at position 0. Use deleteFirstNode instead.\n");
       return head;
   }
   struct Node* current = head;
   struct Node* prev = NULL;
   for (int i = 0; i < position && current != NULL; i++) {
       prev = current;
       current = current->next;
   }
   if (current == NULL) {
       printf("Invalid position.\n");
       return head;
   }
   prev->next = current->next;
   free(current);
   return head;
}

// Function to reverse the linked list
struct Node* reverse(struct Node* head) {
   struct Node* prev = NULL;
   struct Node* current = head;
   struct Node* next = NULL;
   while (current != NULL) {
       next = current->next;
       current->next = prev;
       prev = current;
       current = next;
   }
   return prev;
}


// Function to sort the linked list
struct Node* sortLinkedList(struct Node* head) {
   struct Node* current = head;
   struct Node* index = NULL;
   int temp;
   if (head == NULL) {
       return NULL;
   }
   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;
   }
   return head;
}
```

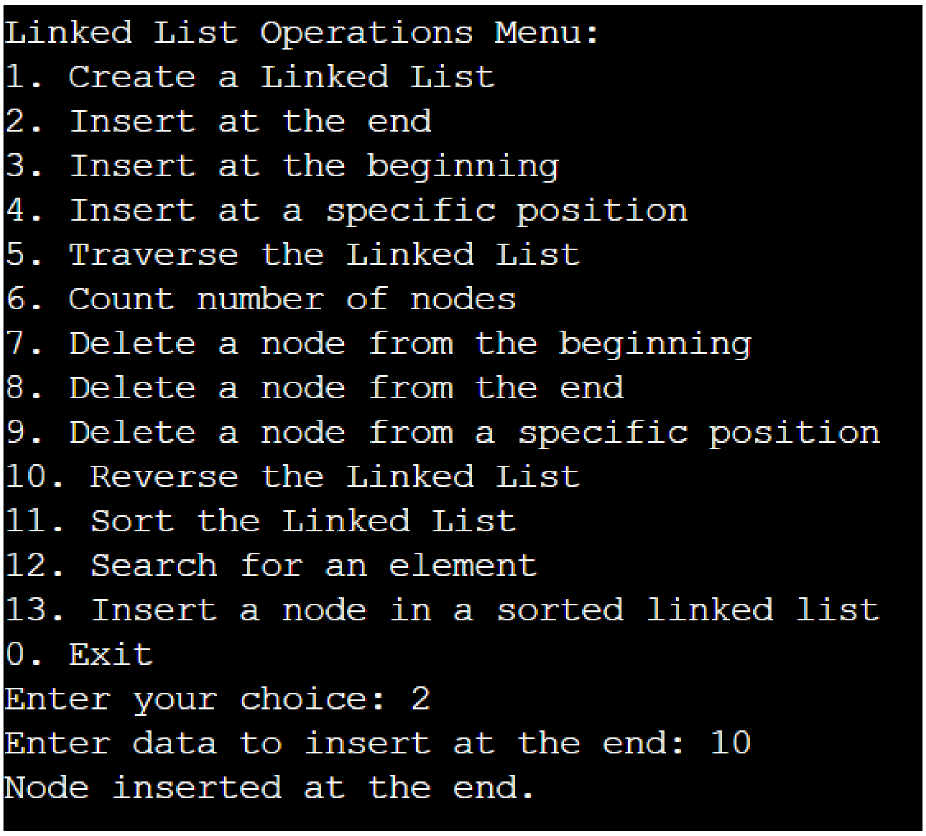
```
// Function to search for a value in the linked list
int search(struct Node* head, int key) {
    int position = 0;
    while (head != NULL) {
        if (head->data == key) {
            return position;
        }
        position++;
        head = head->next;
    }
    return -1;
}

// Function to insert a node in a sorted linked list
struct Node* insertInSorted(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    struct Node* current;
    if (head == NULL || head->data >= newNode->data) {
        newNode->next = head;
        return newNode;
    } else {
        current = head;
        while (current->next != NULL && current->next->data < newNode->data) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
        return head;
    }
}

// Function to display menu
void displayMenu() {
    printf("\nLinked List Operations Menu:\n");
    printf("1. Create a Linked List\n");
    printf("2. Insert at the end\n");
    printf("3. Insert at the beginning\n");
    printf("4. Insert at a specific position\n");
    printf("5. Traverse the Linked List\n");
    printf("6. Count number of nodes\n");
    printf("7. Delete a node from the beginning\n");
    printf("8. Delete a node from the end\n");
    printf("9. Delete a node from a specific position\n");
    printf("10. Reverse the Linked List\n");
    printf("11. Sort the Linked List\n");
    printf("12. Search for an element\n");
    printf("13. Insert a node in a sorted linked list\n");
    printf("0. Exit\n");
}

int main() {
    struct Node* head = NULL;
    int choice, data, position, key;

    while (1) {
        displayMenu();
    printf("Enter your choice: ");
    scanf("%d", &choice);


    switch (choice) {
      case 1:
        printf("Linked list created.\n");
        break;
      case 2:
        printf("Enter data to insert at the end: ");
        scanf("%d", &data);
        if (head == NULL)
          head = createNode(data);
        else
          insertAtEnd(head, data);
        printf("Node inserted at the end.\n");
        break;
      case 3:
        printf("Enter data to insert at the beginning: ");
        scanf("%d", &data);
        head = insertAtBeg(head, data);
        printf("Node inserted at the beginning.\n");
        break;
      case 4:
        printf("Enter data to insert: ");
        scanf("%d", &data);
        printf("Enter position to insert: ");
        scanf("%d", &position);
        insertAtPosition(head, data, position);
```

```
            printf("Node inserted at position %d.\n", position);
            break;
        case 5:
            traverse(head);
            break;
        case 6:
            printf("Length of the linked list: %d\n", countNodes(head));
            break;
        case 7:
            head = deleteFirstNode(head);
            printf("First node deleted.\n");
            break;
        case 8:
            head = deleteLastNode(head);
            printf("Last node deleted.\n");
            break;
        case 9:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            head = deleteAtPosition(head, position);
            printf("Node deleted from position %d.\n", position);
            break;
        case 10:
            head = reverse(head);
            printf("Linked list reversed.\n");
            break;
        case 11:
            head = sortLinkedList(head);
            printf("Linked list sorted.\n");
            break;
        case 12:
            printf("Enter element to search: ");
            scanf("%d", &key);
            position = search(head, key);
            if (position != -1)
                printf("Element found at position %d.\n", position);
            else
                printf("Element not found.\n");
            break;
        case 13:
            printf("Enter data to insert in sorted linked list: ");
            scanf("%d", &data);
            head = insertInSorted(head, data);
            printf("Node inserted in sorted linked list.\n");
            break;
        case 0:
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice.\n");
      }
  }

  return 0;
}
```

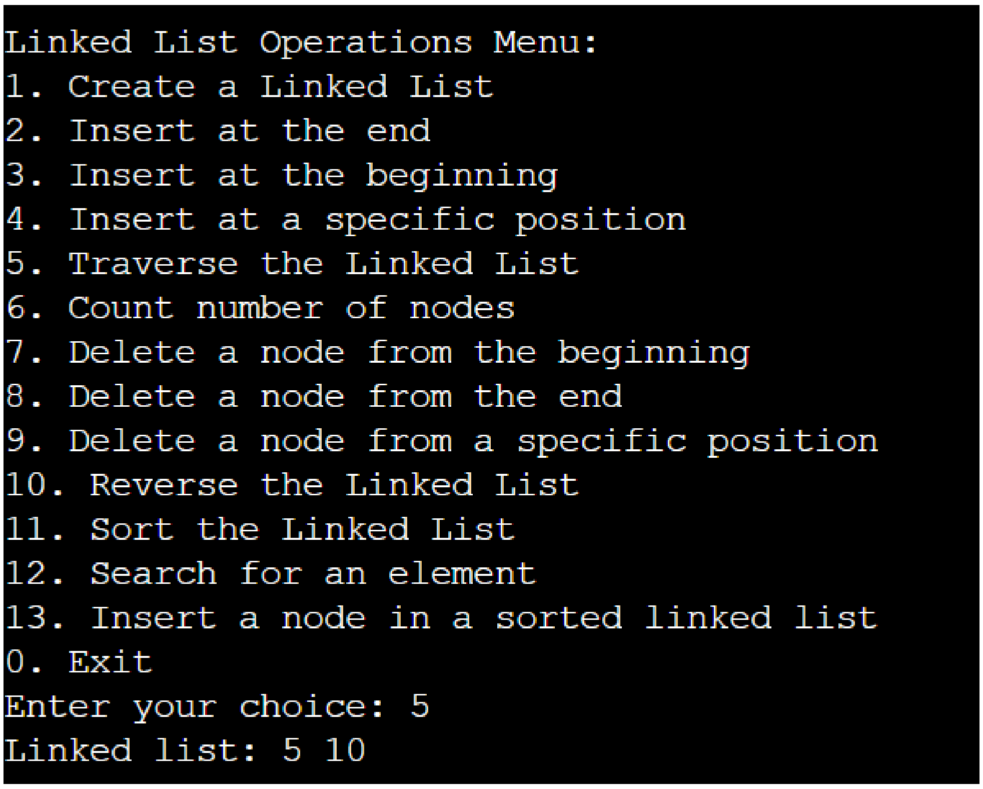
OUTPUT –



```
Linked List Operations Menu:
1. Create a Linked List
2. Insert at the end
3. Insert at the beginning
4. Insert at a specific position
5. Traverse the Linked List
6. Count number of nodes
7. Delete a node from the beginning
8. Delete a node from the end
9. Delete a node from a specific position
10. Reverse the Linked List
11. Sort the Linked List
12. Search for an element
13. Insert a node in a sorted linked list
0. Exit
Enter your choice: 1
Linked list created.
```



```
Linked List Operations Menu:
1. Create a Linked List
2. Insert at the end
3. Insert at the beginning
4. Insert at a specific position
5. Traverse the Linked List
6. Count number of nodes
7. Delete a node from the beginning
8. Delete a node from the end
9. Delete a node from a specific position
10. Reverse the Linked List
11. Sort the Linked List
12. Search for an element
13. Insert a node in a sorted linked list
0. Exit
Enter your choice: 2
Enter data to insert at the end: 10
Node inserted at the end.
```


```
Linked List Operations Menu:
1. Create a Linked List
2. Insert at the end
3. Insert at the beginning
4. Insert at a specific position
5. Traverse the Linked List
6. Count number of nodes
7. Delete a node from the beginning
8. Delete a node from the end
9. Delete a node from a specific position
10. Reverse the Linked List
11. Sort the Linked List
12. Search for an element
13. Insert a node in a sorted linked list
0. Exit
Enter your choice: 5
Linked list: 5 10
```

PROGRAM 10

Write a program to following operations on Doubly linked list

1. Create a Linked list

2. Addition/Insertion of a node at the end of the list.

3. Addition of a node at the beginning of linked list.

4. Addition of a node at the specific position.

5. Traversing a linked list.

6. Counting number of nodes/Length of linked list.

7. Deleting a node from beginning

8. Delete a node from last

9. Delete a node from specific position

10. Reverse A linked List.

11. Sorting.

12. Search

SOL –

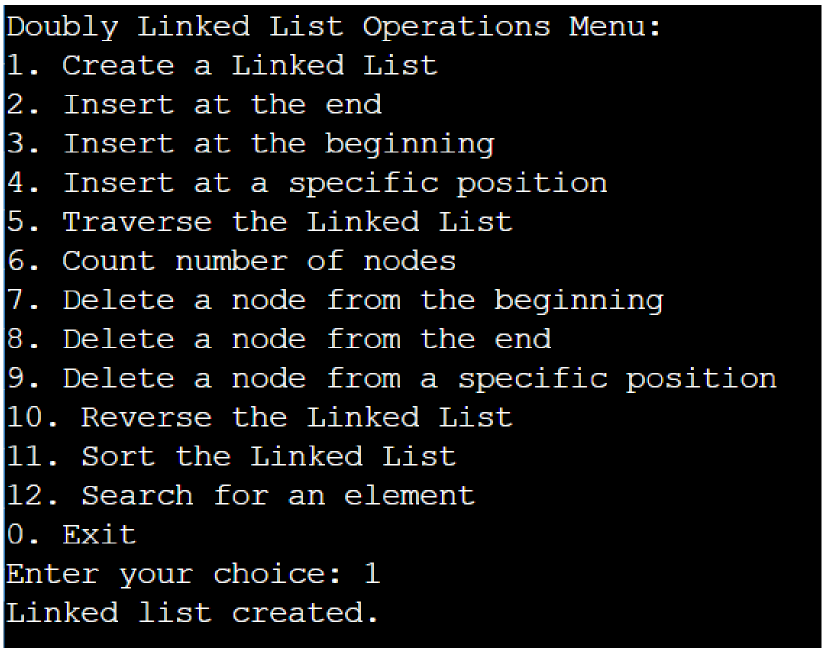
```
#include <stdio.h>

#include <stdlib.h>


// Structure for a node in doubly linked list

struct Node {

   int data;

   struct Node* prev;

   struct Node* next;

};


// Function to create a new node

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
       printf("Memory allocation failed.\n");
       exit(1);
   }
   newNode->data = data;
   newNode->prev = NULL;
   newNode->next = NULL;
   return newNode;
}

// Function to add a node at the end of the doubly linked list
void InsertAtEnd(struct Node* head, int data) {
   struct Node* newNode = createNode(data);
   while (head->next != NULL) {
       head = head->next;
   }
   head->next = newNode;
   newNode->prev = head;
}

// Function to add a node at the beginning of the doubly linked list
struct Node* InsertAtBeg(struct Node* head, int data) {
   struct Node* newNode = createNode(data);
   newNode->next = head;
   if (head != NULL) {
       head->prev = newNode;
   }
   return newNode;
}

// Function to add a node at a specific position in the doubly linked list
struct Node* insertAtPosition(struct Node* head, int data, int position) {
   if (position < 0) {
       printf("Invalid position.\n");
       return head;
   }
   if (position == 0) {
       printf("Cannot insert at position 0. Use prepend instead.\n");
       return head;
   }
   struct Node* newNode = createNode(data);
   struct Node* current = head;
   for (int i = 0; i < position - 1 && current != NULL; i++) {
       current = current->next;
   }
   if (current == NULL) {
       printf("Invalid position.\n");
       return head;
   }
   newNode->next = current->next;
   if (current->next != NULL) {
       current->next->prev = newNode;
   }
   current->next = newNode;
   newNode->prev = current;
   return head;
}

// Function to traverse and print the doubly linked list
void traverse(struct Node* head) {
   printf("Doubly linked list: ");
   while (head != NULL) {
       printf("%d ", head->data);
       head = head->next;
   }
   printf("\n");
}

// Function to count the number of nodes in the doubly linked list
int countNodes(struct Node* head) {
   int count = 0;
   while (head != NULL) {
       count++;
       head = head->next;
   }
   return count;
}

// Function to delete the first node from the doubly linked list
struct Node* deleteFirstNode(struct Node* head) {
   if (head == NULL) {
       printf("Linked list is empty.\n");
       return NULL;
   }
   struct Node* temp = head;
   head = head->next;
   if (head != NULL) {
       head->prev = NULL;
   }
   free(temp);
   return head;
}

// Function to delete the last node from the doubly linked list
struct Node* deleteLastNode(struct Node* head) {
   if (head == NULL) {
       printf("Linked list is empty.\n");
       return NULL;
   }
   if (head->next == NULL) {
       free(head);
       return NULL;
   }
   struct Node* current = head;
   while (current->next != NULL) {
       current = current->next;
   }
   current->prev->next = NULL;
   free(current);
   return head;
}
```

```
// Function to delete a node at a specific position from the doubly linked list
struct Node* deleteAtPosition(struct Node* head, int position) {
    if (head == NULL) {
        printf("Linked list is empty.\n");
        return NULL;
    }
    if (position < 0) {
        printf("Invalid position.\n");
        return head;
    }
    if (position == 0) {
        printf("Cannot delete at position 0. Use deleteFirstNode instead.\n");
        return head;
    }
    struct Node* current = head;
    for (int i = 0; i < position && current != NULL; i++) {
        current = current->next;
    }
    if (current == NULL) {
        printf("Invalid position.\n");
        return head;
    }
    if (current->prev != NULL) {
        current->prev->next = current->next;
    }
    if (current->next != NULL) {
        current->next->prev = current->prev;
    }
    free(current);
    return head;
}

// Function to reverse the doubly linked list
struct Node* reverse(struct Node* head) {
    struct Node* temp = NULL;
    struct Node* current = head;
    while (current != NULL) {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }
    if (temp != NULL) {
        head = temp->prev;
    }
    return head;
}

// Function to sort the doubly linked list
struct Node* sortLinkedList(struct Node* head) {
    struct Node* current = head;
    struct Node* index = NULL;
    int temp;
    if (head == NULL) {
        return NULL;
    }
    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;
    }
    return head;
}

// Function to search for a value in the doubly linked list
int search(struct Node* head, int key) {
    int position = 0;
    while (head != NULL) {
        if (head->data == key) {
            return position;
        }
        position++;
        head = head->next;
    }
    return -1;
}
```

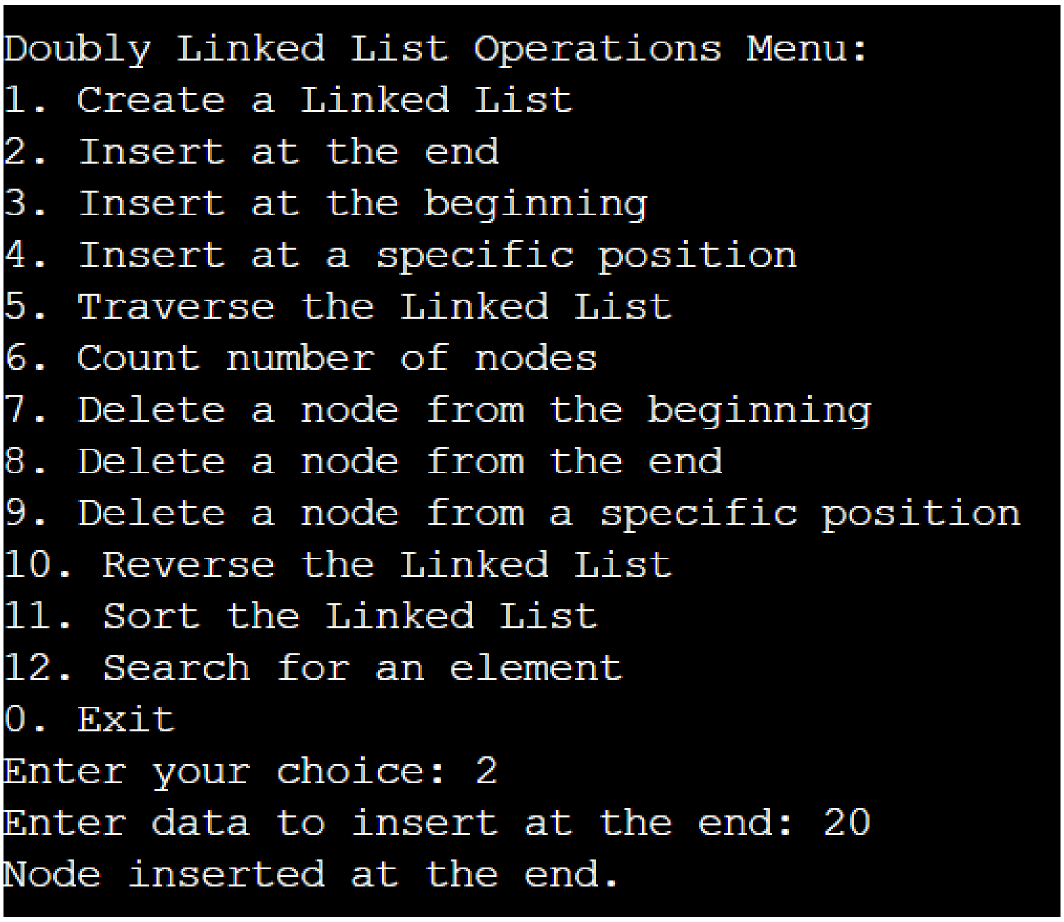
```
// Function to display menu
void displayMenu() {
    printf("\nDoubly Linked List Operations Menu:\n");
    printf("1. Create a Linked List\n");
    printf("2. Insert at the end\n");
    printf("3. Insert at the beginning\n");
    printf("4. Insert at a specific position\n");
    printf("5. Traverse the Linked List\n");
    printf("6. Count number of nodes\n");
    printf("7. Delete a node from the beginning\n");
    printf("8. Delete a node from the end\n");
    printf("9. Delete a node from a specific position\n");
    printf("10. Reverse the Linked List\n");
    printf("11. Sort the Linked List\n");
    printf("12. Search for an element\n");
    printf("0. Exit\n");
}

int main() {
    struct Node* head = NULL;
    int choice, data, position, key;

    while (1) {
        displayMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);
    switch (choice) {
        case 1:
            printf("Linked list created.\n");
            break;
        case 2:
            printf("Enter data to insert at the end: ");
            scanf("%d", &data);
            if (head == NULL)
                head = createNode(data);
            else
                InsertAtEnd(head, data);
            printf("Node inserted at the end.\n");
            break;
        case 3:
            printf("Enter data to insert at the beginning: ");
            scanf("%d", &data);
            head = InsertAtBeg(head, data);
            printf("Node inserted at the beginning.\n");
            break;
        case 4:
            printf("Enter data to insert: ");
            scanf("%d", &data);
            printf("Enter position to insert: ");
            scanf("%d", &position);
            head = insertAtPosition(head, data, position);
            printf("Node inserted at position %d.\n", position);
            break;
        case 5:
```

```
        traverse(head);
        break;
    case 6:
        printf("Length of the linked list: %d\n", countNodes(head));
        break;
    case 7:
        head = deleteFirstNode(head);
        printf("First node deleted.\n");
        break;
    case 8:
        head = deleteLastNode(head);
        printf("Last node deleted.\n");
        break;
    case 9:
        printf("Enter position to delete: ");
        scanf("%d", &position);
        head = deleteAtPosition(head, position);
        printf("Node deleted from position %d.\n", position);
        break;
    case 10:
        head = reverse(head);
        printf("Linked list reversed.\n");
        break;
    case 11:
        head = sortLinkedList(head);
        printf("Linked list sorted.\n");
        break;
    case 12:
            printf("Enter element to search: ");
            scanf("%d", &key);
            position = search(head, key);
            if (position != -1)
               printf("Element found at position %d.\n", position);
            else
               printf("Element not found.\n");
            break;
         case 0:
            printf("Exiting...\n");
            exit(0);
         default:
            printf("Invalid choice.\n");
      }
   }
   return 0;
}
```

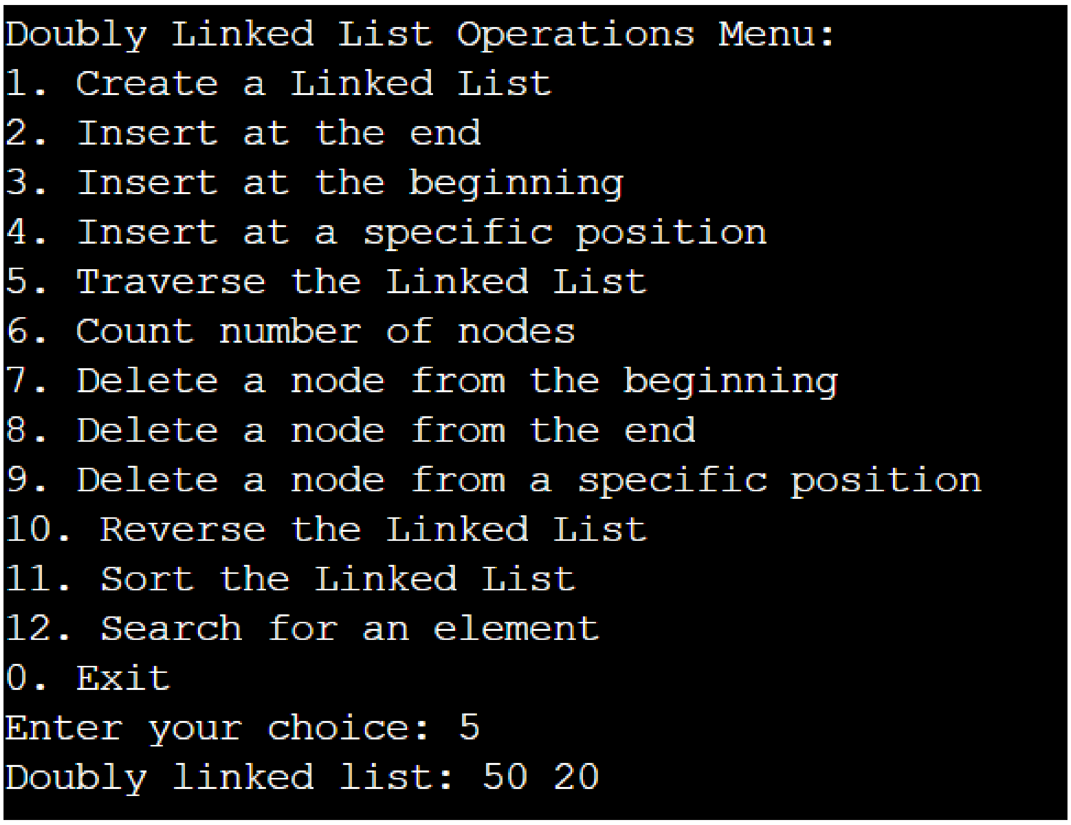
OUTPUT



```
Doubly Linked List Operations Menu:
1. Create a Linked List
2. Insert at the end
3. Insert at the beginning
4. Insert at a specific position
5. Traverse the Linked List
6. Count number of nodes
7. Delete a node from the beginning
8. Delete a node from the end
9. Delete a node from a specific position
10. Reverse the Linked List
11. Sort the Linked List
12. Search for an element
0. Exit
Enter your choice: 1
Linked list created.
```



```
Doubly Linked List Operations Menu:
1. Create a Linked List
2. Insert at the end
3. Insert at the beginning
4. Insert at a specific position
5. Traverse the Linked List
6. Count number of nodes
7. Delete a node from the beginning
8. Delete a node from the end
9. Delete a node from a specific position
10. Reverse the Linked List
11. Sort the Linked List
12. Search for an element
0. Exit
Enter your choice: 2
Enter data to insert at the end: 20
Node inserted at the end.
```



```
Doubly Linked List Operations Menu:
1. Create a Linked List
2. Insert at the end
3. Insert at the beginning
4. Insert at a specific position
5. Traverse the Linked List
6. Count number of nodes
7. Delete a node from the beginning
8. Delete a node from the end
9. Delete a node from a specific position
10. Reverse the Linked List
11. Sort the Linked List
12. Search for an element
0. Exit
Enter your choice: 5
Doubly linked list: 50 20
```

PROGRAM 11

Write a program to following operations on Singly Circular linked list

1. Create a Linked list

2. Addition/Insertion of a node at the end of the list.

3. Addition of a node at the beginning of linked list.

4. Addition of a node at the specific position.

5. Traversing a linked list.

6. Counting number of nodes/Length of linked list.

7. Deleting a node from beginning

8. Delete a node from last

9. Delete a node from specific position

10. Reverse A linked List.

11. Sorting.

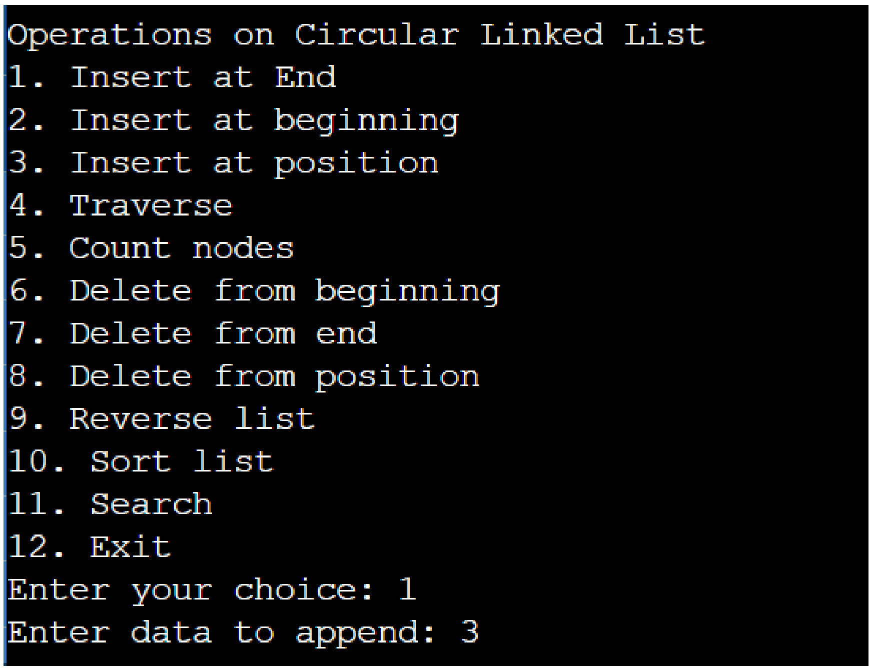
12. Search.

SOL –

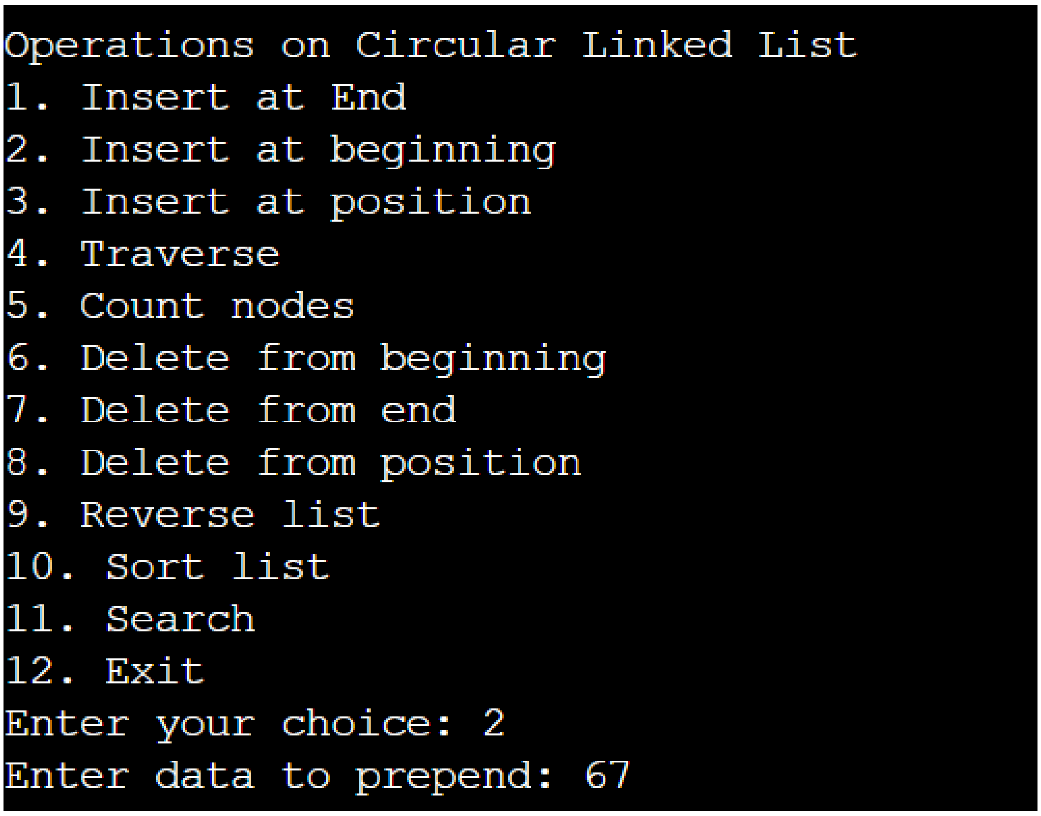
```
#include <stdio.h>
#include <stdlib.h>

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\n");
```

```
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

struct Node* append(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        struct Node* current = head;
        while (current->next != head) {
            current = current->next;
        }
        current->next = newNode;
        newNode->next = head;
    }
    return head;
}

struct Node* prepend(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        struct Node* current = head;
        while (current->next != head) {
            current = current->next;
        }
        newNode->next = head;
        current->next = newNode;
        head = newNode;
    }
    return head;
}

struct Node* insertAtPosition(struct Node* head, int data, int position) {
    if (position < 0) {
        printf("Invalid position\n");
        return head;
    }
    if (position == 0) {
        return prepend(head, data);
    }
    struct Node* newNode = createNode(data);
    struct Node* current = head;
    for (int i = 0; i < position - 1; i++) {
        if (current->next == head) {
            printf("Position out of bounds\n");
            return head;
        }
        current = current->next;
    }
    newNode->next = current->next;
    current->next = newNode;
    return head;
}

void traverse(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* current = head;
    do {
        printf("%d ", current->data);
        current = current->next;
    } while (current != head);
    printf("\n");
}

int countNodes(struct Node* head) {
    if (head == NULL) {
        return 0;
    }
    int count = 0;
    struct Node* current = head;
    do {
        count++;
        current = current->next;
    } while (current != head);
    return count;
}

struct Node* deleteFromBeginning(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }
    struct Node* temp = head;
    struct Node* current = head;
    while (current->next != head) {
        current = current->next;
    }
    current->next = head->next;
    free(temp);
    return current->next;
}

struct Node* deleteFromEnd(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }
    struct Node* current = head;
    struct Node* prev = NULL;
    while (current->next != head) {
        prev = current;
        current = current->next;
    }
    prev->next = head;
    free(current);
    return head;
}

struct Node* deleteFromPosition(struct Node* head, int position) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }
    if (position < 0) {
        printf("Invalid position\n");
        return head;
    }
    int count = countNodes(head);
    if (position >= count) {
        printf("Position out of bounds\n");
        return head;
    }
    if (position == 0) {
        return deleteFromBeginning(head);
    }
    struct Node* current = head;
    struct Node* prev = NULL;
    for (int i = 0; i < position; i++) {
        prev = current;
        current = current->next;
    }
    prev->next = current->next;
    free(current);
    return head;
}

struct Node* reverseList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }
    struct Node* current = head;
    struct Node *prev = NULL, *next = NULL;
    do {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    } while (current != head);
    head->next = prev;
    head = prev;
    return head;
}

void sortList(struct Node* head) {
    // You can implement any sorting algorithm here
    // For simplicity, let's assume the list is already sorted
    printf("List is already sorted\n");
}

void search(struct Node* head, int key) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* current = head;
    int position = 0;
    do {
        if (current->data == key) {
            printf("%d found at position %d\n", key, position);
            return;
        }
        position++;
        current = current->next;
    } while (current != head);
    printf("%d not found in the list\n", key);
}

void displayMenu() {
    printf("\nOperations on Circular Linked List\n");
    printf("1. Insert at End\n");
    printf("2. Insert at beginning\n");
    printf("3. Insert at position\n");
    printf("4. Traverse\n");
    printf("5. Count nodes\n");
    printf("6. Delete from beginning\n");
    printf("7. Delete from end\n");
    printf("8. Delete from position\n");
    printf("9. Reverse list\n");
    printf("10. Sort list\n");
    printf("11. Search\n");
    printf("12. Exit\n");
}

int main() {
    struct Node* head = NULL;
    int choice, data, position, key;
    while (1) {
        displayMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to append: ");
                scanf("%d", &data);
                head = append(head, data);
                break;
            case 2:
                printf("Enter data to prepend: ");
                scanf("%d", &data);
                head = prepend(head, data);
                break;
            case 3:
```

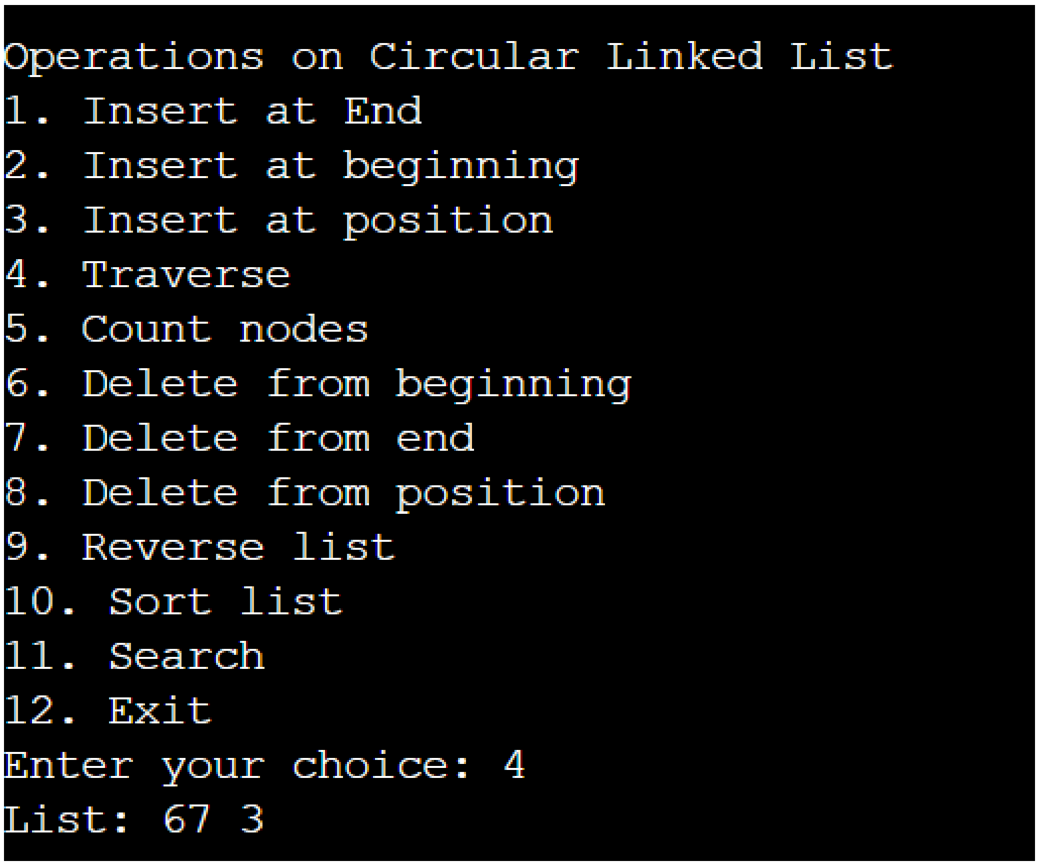
```
        printf("Enter data to insert: ");
        scanf("%d", &data);
        printf("Enter position: ");
        scanf("%d", &position);
        head = insertAtPosition(head, data, position);
        break;
    case 4:
        printf("List: ");
        traverse(head);
        break;
    case 5:
        printf("Number of nodes: %d\n", countNodes(head));
        break;
    case 6:
        head = deleteFromBeginning(head);
        break;
    case 7:
        head = deleteFromEnd(head);
        break;
    case 8:
        printf("Enter position to delete: ");
        scanf("%d", &position);
        head = deleteFromPosition(head, position);
        break;
    case 9:
        head = reverseList(head);
        printf("List reversed\n");
        break;
        case 10:
          sortList(head);
          break;
        case 11:
          printf("Enter key to search: ");
          scanf("%d", &key);
          search(head, key);
          break;
        case 12:
          printf("Exiting...\n");
          exit(0);
        default:
          printf("Invalid choice\n");
      }
    }
  return 0;
}
```



```
Operations on Circular Linked List
1. Insert at End
2. Insert at beginning
3. Insert at position
4. Traverse
5. Count nodes
6. Delete from beginning
7. Delete from end
8. Delete from position
9. Reverse list
10. Sort list
11. Search
12. Exit
Enter your choice: 1
Enter data to append: 3
```



```
Operations on Circular Linked List
1. Insert at End
2. Insert at beginning
3. Insert at position
4. Traverse
5. Count nodes
6. Delete from beginning
7. Delete from end
8. Delete from position
9. Reverse list
10. Sort list
11. Search
12. Exit
Enter your choice: 2
Enter data to prepend: 67
```



```
Operations on Circular Linked List
1. Insert at End
2. Insert at beginning
3. Insert at position
4. Traverse
5. Count nodes
6. Delete from beginning
7. Delete from end
8. Delete from position
9. Reverse list
10. Sort list
11. Search
12. Exit
Enter your choice: 4
List: 67 3
```

PROGRAM 12

Write a program to following operations on Doubly Circular linked list

1. Create a Linked list

2. Addition/Insertion of a node at the end of the list.

3. Addition of a node at the beginning of linked list.

4. Addition of a node at the specific position.

5. Traversing a linked list.

6. Counting number of nodes/Length of linked list.

7. Deleting a node from brginning

8. Delete a node from last

9. Delete a node from specific position

10. Reverse A linked List.

11. Sorting.

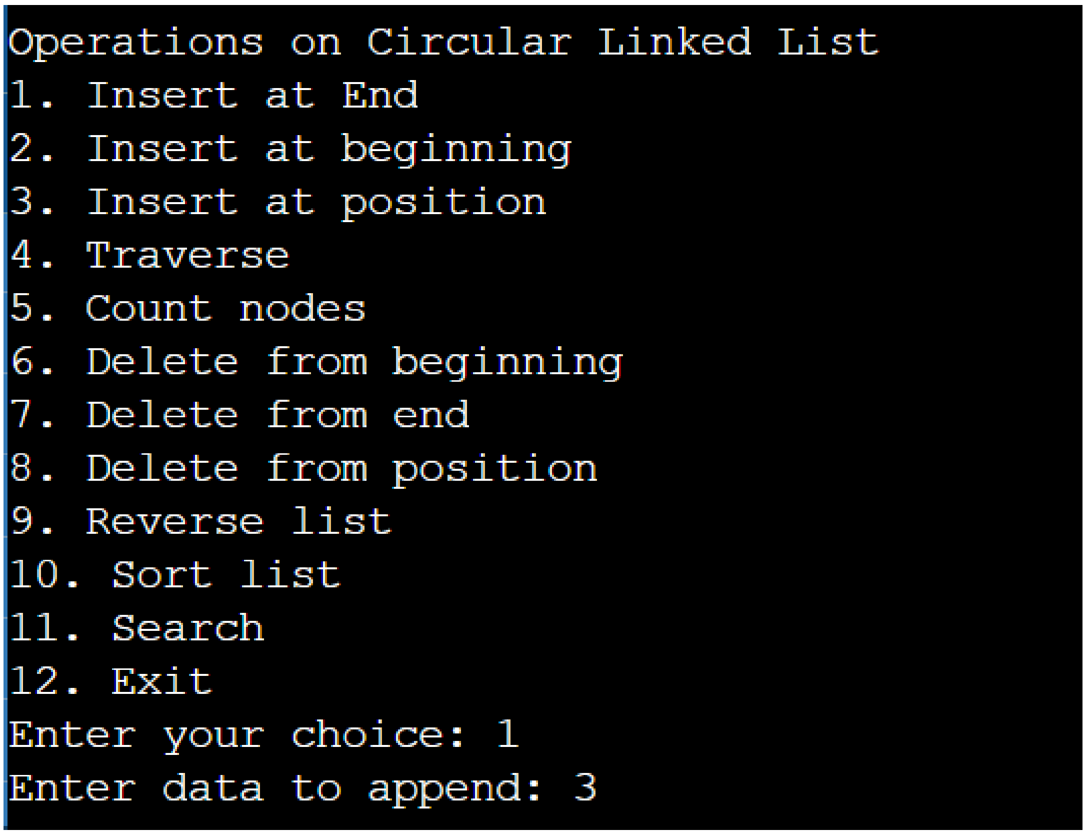
12. Search

```
#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 == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}

struct Node* append(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
        head->next = head;
        head->prev = head;
    } else {
        struct Node* tail = head->prev;
        tail->next = newNode;
        newNode->prev = tail;
        newNode->next = head;
        head->prev = newNode;
    }
    return head;
}

struct Node* prepend(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
        head->next = head;
        head->prev = head;
    } else {
        struct Node* tail = head->prev;
        newNode->next = head;
        newNode->prev = tail;
        head->prev = newNode;
        tail->next = newNode;
        head = newNode;
    }
    return head;
}

struct Node* insertAtPosition(struct Node* head, int data, int position) {
    if (position < 0) {
        printf("Invalid position\n");
        return head;
    }
    if (position == 0) {
        return prepend(head, data);
    }
    struct Node* newNode = createNode(data);
    struct Node* current = head;
    for (int i = 0; i < position - 1; i++) {
        if (current->next == head) {
            printf("Position out of bounds\n");
            return head;
        }
        current = current->next;
    }
    newNode->next = current->next;
    newNode->prev = current;
    current->next->prev = newNode;
    current->next = newNode;
    return head;
}

void traverse(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* current = head;
    do {
        printf("%d ", current->data);
        current = current->next;
    } while (current != head);
    printf("\n");
}

int countNodes(struct Node* head) {
    if (head == NULL) {
        return 0;
    }
    int count = 0;
    struct Node* current = head;
    do {
        count++;
        current = current->next;
    } while (current != head);
    return count;
}

struct Node* deleteFromBeginning(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }
    struct Node* temp = head;
    if (head->next == head) {
        free(head);
        return NULL;
    }
    struct Node* tail = head->prev;
    head = head->next;
    head->prev = tail;
    tail->next = head;
    free(temp);
    return head;
}

struct Node* deleteFromEnd(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }
    struct Node* tail = head->prev;
    if (head->next == head) {
        free(head);
        return NULL;
    }
    struct Node* prevTail = tail->prev;
    prevTail->next = head;
    head->prev = prevTail;
    free(tail);
    return head;
}

struct Node* deleteFromPosition(struct Node* head, int position) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }
    if (position < 0) {
        printf("Invalid position\n");
        return head;
    }
    int count = countNodes(head);
    if (position >= count) {
        printf("Position out of bounds\n");
        return head;
    }
    if (position == 0) {
        return deleteFromBeginning(head);
    }
    if (position == count - 1) {
        return deleteFromEnd(head);
    }
    struct Node* current = head;
    for (int i = 0; i < position; i++) {
        current = current->next;
    }
    current->prev->next = current->next;
    current->next->prev = current->prev;
    free(current);
    return head;
}

struct Node* reverseList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }
    struct Node* current = head;
    do {
        struct Node* temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    } while (current != head);
    head = head->prev;
    return head;
}

void sortList(struct Node* head) {
    // You can implement any sorting algorithm here
    // For simplicity, let's assume the list is already sorted
    printf("List is already sorted\n");
}

void search(struct Node* head, int key) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* current = head;
    int position = 0;
    do {
        if (current->data == key) {
            printf("%d found at position %d\n", key, position);
            return;
        }
        position++;
        current = current->next;
    } while (current != head);
    printf("%d not found in the list\n", key);
}
```

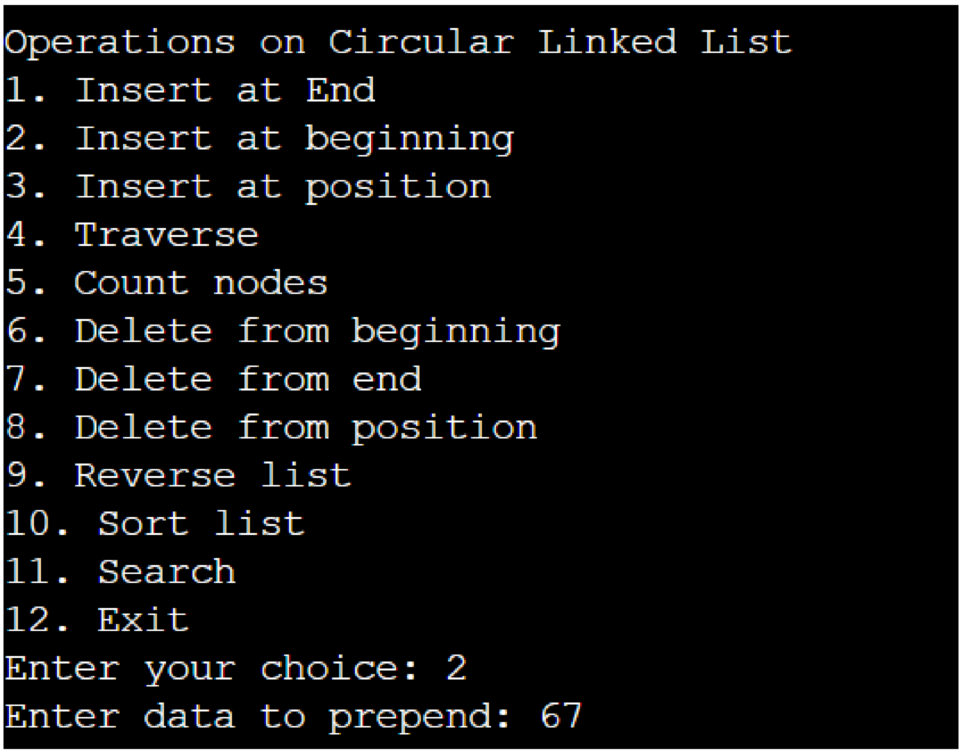
```
void displayMenu() {
    printf("\nOperations on Doubly Circular Linked List\n");
    printf("1. Append\n");
    printf("2. Prepend\n");
    printf("3. Insert at position\n");
    printf("4. Traverse\n");
    printf("5. Count nodes\n");
    printf("6. Delete from beginning\n");
    printf("7. Delete from end\n");
    printf("8. Delete from position\n");
    printf("9. Reverse list\n");
    printf("10. Sort list\n");
    printf("11. Search\n");
    printf("12. Exit\n");
}

int main() {
    struct Node* head = NULL;
    int choice, data, position, key;
    while (1) {
        displayMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to append: ");
                scanf("%d", &data);
                head = append(head, data);
                break;
            case 2:
                printf("Enter data to prepend: ");
                scanf("%d", &data);
                head = prepend(head, data);
                break;
            case 3:
                printf("Enter datato insert: ");
                scanf("%d", &data);
                printf("Enter position: ");
                scanf("%d", &position);
                head = insertAtPosition(head, data, position);
                break;
            case 4:
                printf("List: ");
                traverse(head);
                break;
            case 5:
                printf("Number of nodes: %d\n", countNodes(head));
                break;
            case 6:
                head = deleteFromBeginning(head);
                break;
            case 7:
                head = deleteFromEnd(head);
                break;
            case 8:
                printf("Enter position to delete: ");
```

```
            scanf("%d", &position);
            head = deleteFromPosition(head, position);
            break;
         case 9:
            head = reverseList(head);
            printf("List reversed\n");
            break;
         case 10:
            sortList(head);
            break;
         case 11:
            printf("Enter key to search: ");
            scanf("%d", &key);
            search(head, key);
            break;
         case 12:
            printf("Exiting...\n");
            exit(0);
         default:
            printf("Invalid choice\n");
      }
   }
   return 0;
}
```

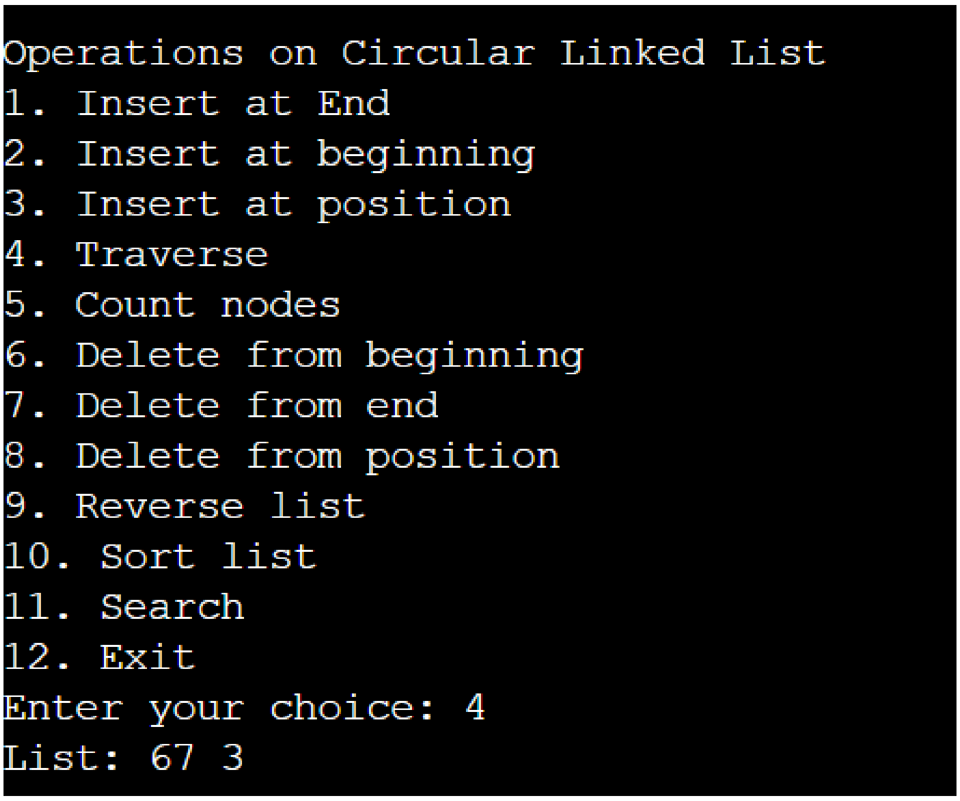
OUTPUT –



```
Operations on Circular Linked List
1. Insert at End
2. Insert at beginning
3. Insert at position
4. Traverse
5. Count nodes
6. Delete from beginning
7. Delete from end
8. Delete from position
9. Reverse list
10. Sort list
11. Search
12. Exit
Enter your choice: 1
Enter data to append: 3
```



```
Operations on Circular Linked List
1. Insert at End
2. Insert at beginning
3. Insert at position
4. Traverse
5. Count nodes
6. Delete from beginning
7. Delete from end
8. Delete from position
9. Reverse list
10. Sort list
11. Search
12. Exit
Enter your choice: 2
Enter data to prepend: 67
```



```
Operations on Circular Linked List
1. Insert at End
2. Insert at beginning
3. Insert at position
4. Traverse
5. Count nodes
6. Delete from beginning
7. Delete from end
8. Delete from position
9. Reverse list
10. Sort list
11. Search
12. Exit
Enter your choice: 4
List: 67 3
```

PROGRAM 13

WAP to perform addition and multiplication of two polynomials by creating linked list for every polynomial. (Polynomial Arithmetic)

SOL –

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

struct Node {
   int coefficient;
   int exponent;
   struct Node* next;
};

struct Node* createNode(int coefficient, int exponent) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
      printf("Memory allocation failed\n");
      exit(1);
   }
   newNode->coefficient = coefficient;
   newNode->exponent = exponent;
   newNode->next = NULL;
   return newNode;
}

void insertTerm(struct Node** poly, int coefficient, int exponent) {
   struct Node* newNode = createNode(coefficient, exponent);
```

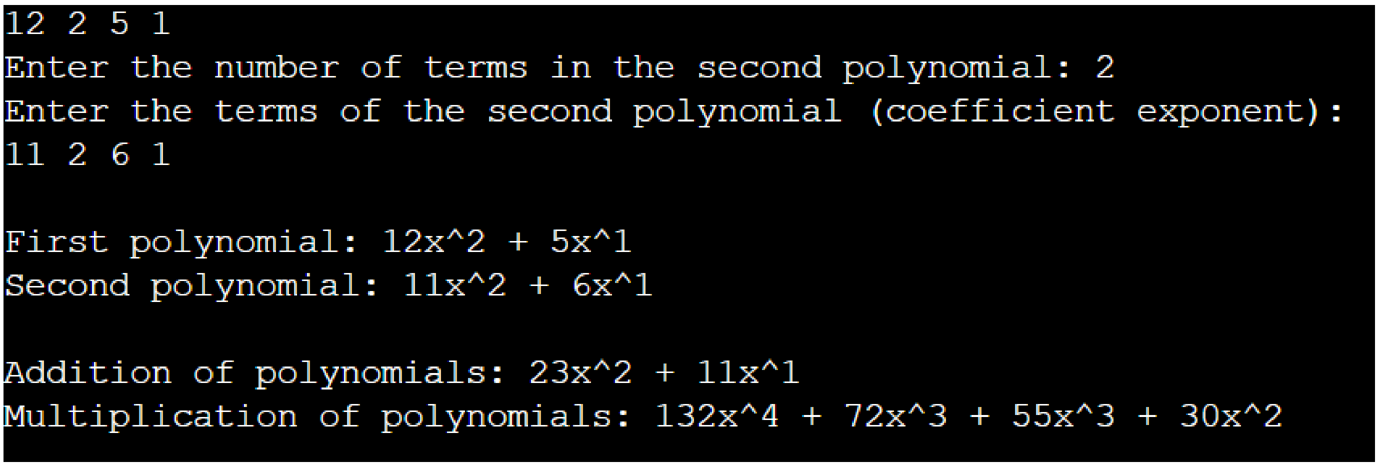
```
    if (*poly == NULL) {
        *poly = newNode;
    } else {
        struct Node* current = *poly;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
}

void displayPolynomial(struct Node* poly) {
    if (poly == NULL) {
        printf("Polynomial is empty\n");
        return;
    }
    struct Node* current = poly;
    while (current != NULL) {
        if (current->coefficient != 0) {
            printf("%dx^%d ", current->coefficient, current->exponent);
            if (current->next != NULL && current->next->coefficient >= 0) {
                printf("+ ");
            }
        }
        current = current->next;
    }
    printf("\n");
}
```

```
struct Node* addPolynomials(struct Node* poly1, struct Node* poly2) {
    struct Node* result = NULL;
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exponent > poly2->exponent) {
            insertTerm(&result, poly1->coefficient, poly1->exponent);
            poly1 = poly1->next;
        } else if (poly1->exponent < poly2->exponent) {
            insertTerm(&result, poly2->coefficient, poly2->exponent);
            poly2 = poly2->next;
        } else {
            insertTerm(&result, poly1->coefficient + poly2->coefficient, poly1->exponent);
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
    }
    while (poly1 != NULL) {
        insertTerm(&result, poly1->coefficient, poly1->exponent);
        poly1 = poly1->next;
    }
    while (poly2 != NULL) {
        insertTerm(&result, poly2->coefficient, poly2->exponent);
        poly2 = poly2->next;
    }
    return result;
}

struct Node* multiplyPolynomials(struct Node* poly1, struct Node* poly2) {
    struct Node* result = NULL;

    struct Node* temp = poly2;

    while (poly1 != NULL) {

        while (temp != NULL) {

            int coefficient = poly1->coefficient * temp->coefficient;

            int exponent = poly1->exponent + temp->exponent;

            insertTerm(&result, coefficient, exponent);

            temp = temp->next;

        }

        temp = poly2;

        poly1 = poly1->next;

    }

    return result;

}


void freePolynomial(struct Node* poly) {

    struct Node* temp;

    while (poly != NULL) {

        temp = poly;

        poly = poly->next;

        free(temp);

    }

}


int main() {

    struct Node* poly1 = NULL;

    struct Node* poly2 = NULL;

    struct Node* result = NULL;
```

```
int numTerms1, numTerms2, coefficient, exponent;


printf("Enter the number of terms in the first polynomial: ");
scanf("%d", &numTerms1);
printf("Enter the terms of the first polynomial (coefficient exponent): \n");
for (int i = 0; i < numTerms1; i++) {
    scanf("%d %d", &coefficient, &exponent);
    insertTerm(&poly1, coefficient, exponent);
}


printf("Enter the number of terms in the second polynomial: ");
scanf("%d", &numTerms2);
printf("Enter the terms of the second polynomial (coefficient exponent): \n");
for (int i = 0; i < numTerms2; i++) {
    scanf("%d %d", &coefficient, &exponent);
    insertTerm(&poly2, coefficient, exponent);
}


printf("\nFirst polynomial: ");
displayPolynomial(poly1);
printf("Second polynomial: ");
displayPolynomial(poly2);


result = addPolynomials(poly1, poly2);
printf("\nAddition of polynomials: ");
displayPolynomial(result);
freePolynomial(result);
```

```
    result = multiplyPolynomials(poly1, poly2);
    printf("Multiplication of polynomials: ");
    displayPolynomial(result);
    freePolynomial(result);


    freePolynomial(poly1);
    freePolynomial(poly2);


    return 0;
}
```

OUTPUT –



```
12 2 5 1
Enter the number of terms in the second polynomial: 2
Enter the terms of the second polynomial (coefficient exponent):
11 2 6 1

First polynomial: 12x^2 + 5x^1
Second polynomial: 11x^2 + 6x^1

Addition of polynomials: 23x^2 + 11x^1
Multiplication of polynomials: 132x^4 + 72x^3 + 55x^3 + 30x^2
```

PROGRAM 14

Write a program to perform following operations on Stack using Arrays.

1. Push.

2. Pop.

3. Peek.

4. Traverse.

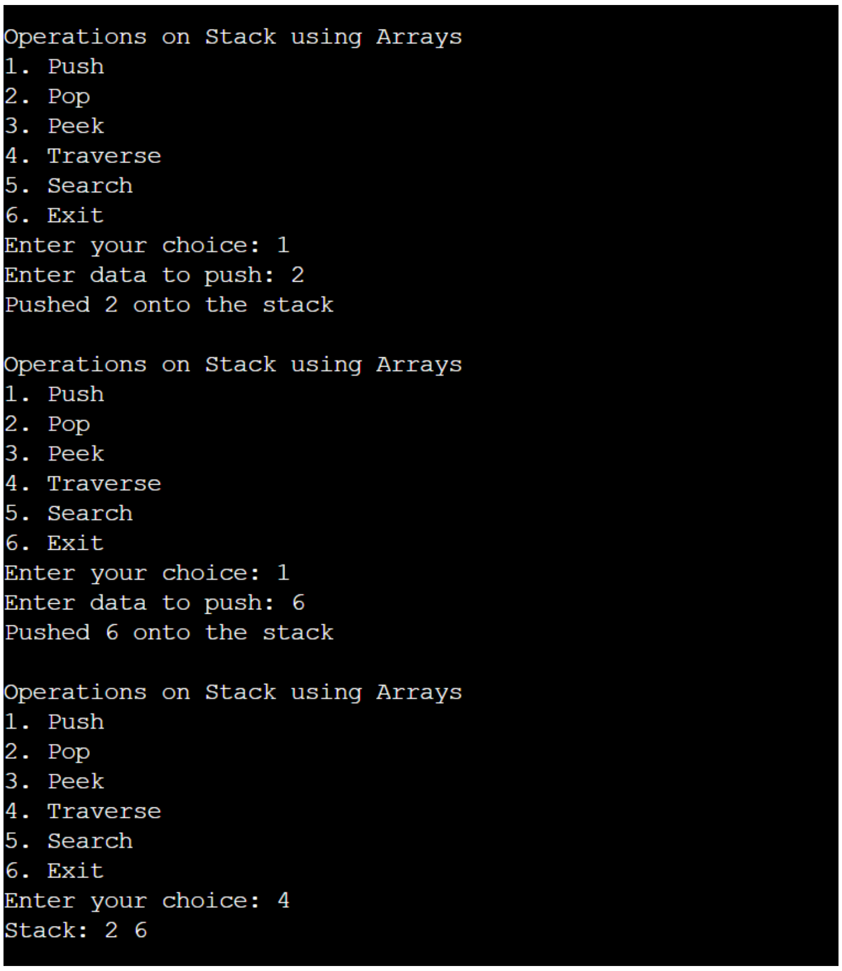
5. Search.

SOL –

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

#define MAX_SIZE 100

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

int isFull() {
    return top == MAX_SIZE - 1;
}

int isEmpty() {
    return top == -1;
}

void push(int data) {
    if (isFull()) {
        printf("Stack overflow\n");
        return;
    }
    stack[++top] = data;
    printf("Pushed %d onto the stack\n", data);
}

int pop() {
    if (isEmpty()) {
        printf("Stack underflow\n");
        return -1;
    }
    int data = stack[top--];
    printf("Popped %d from the stack\n", data);
    return data;
}

int peek() {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return -1;
    }
    return stack[top];
}

void traverse() {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return;
    }
    printf("Stack: ");
    for (int i = 0; i <= top; i++) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}

int search(int key) {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return -1;
    }
    for (int i = 0; i <= top; i++) {
        if (stack[i] == key) {
            return i;
        }
    }
    return -1;
}

void displayMenu() {
    printf("\nOperations on Stack using Arrays\n");
    printf("1. Push\n");
    printf("2. Pop\n");
    printf("3. Peek\n");
    printf("4. Traverse\n");
    printf("5. Search\n");
    printf("6. Exit\n");
```

```
}


int main() {
    int choice, data, key;
    while (1) {
        displayMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to push: ");
                scanf("%d", &data);
                push(data);
                break;
            case 2:
                pop();
                break;
            case 3:
                data = peek();
                if (data != -1) {
                    printf("Top element of stack: %d\n", data);
                }
                break;
            case 4:
                traverse();
                break;
            case 5:
                printf("Enter element to search: ");
            scanf("%d", &key);
            if (search(key) != -1) {
               printf("%d found in the stack\n", key);
            } else {
               printf("%d not found in the stack\n", key);  }
            break;
         case 6:
            printf("Exiting...\n");
            exit(0);
         default:
            printf("Invalid choice\n");
      } }
   return 0;}
```

OUTPUT -



```
Operations on Stack using Arrays
1. Push
2. Pop
3. Peek
4. Traverse
5. Search
6. Exit
Enter your choice: 1
Enter data to push: 2
Pushed 2 onto the stack

Operations on Stack using Arrays
1. Push
2. Pop
3. Peek
4. Traverse
5. Search
6. Exit
Enter your choice: 1
Enter data to push: 6
Pushed 6 onto the stack

Operations on Stack using Arrays
1. Push
2. Pop
3. Peek
4. Traverse
5. Search
6. Exit
Enter your choice: 4
Stack: 2 6
```

PROGRAM 15

Write a program to implement stack using linked list and perform following operations on it.

1. Push.

2. Pop.

3. Peek.

4. Traverse.

5. Search.

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

// Define Node structure
struct Node {
    int data;
    struct Node* next;
};

// Define global top pointer
struct Node* top = NULL;

// Function to push an element onto the stack
void push(int data) {
    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    // Assign data and link it to the current top node
    newNode->data = data;
    newNode->next = top;
    // Update top to point to the new node
    top = newNode;
    printf("Pushed %d onto the stack\n", data);
}

// Function to pop an element from the stack
int pop() {
    if (top == NULL) {
        printf("Stack underflow\n");
        return -1;
    }
    // Get the data from the top node
    int data = top->data;
    // Move top to the next node
    struct Node* temp = top;
    top = top->next;
    // Free the memory of the popped node
    free(temp);
    printf("Popped %d from the stack\n", data);
    return data;
}

// Function to peek at the top element of the stack
```

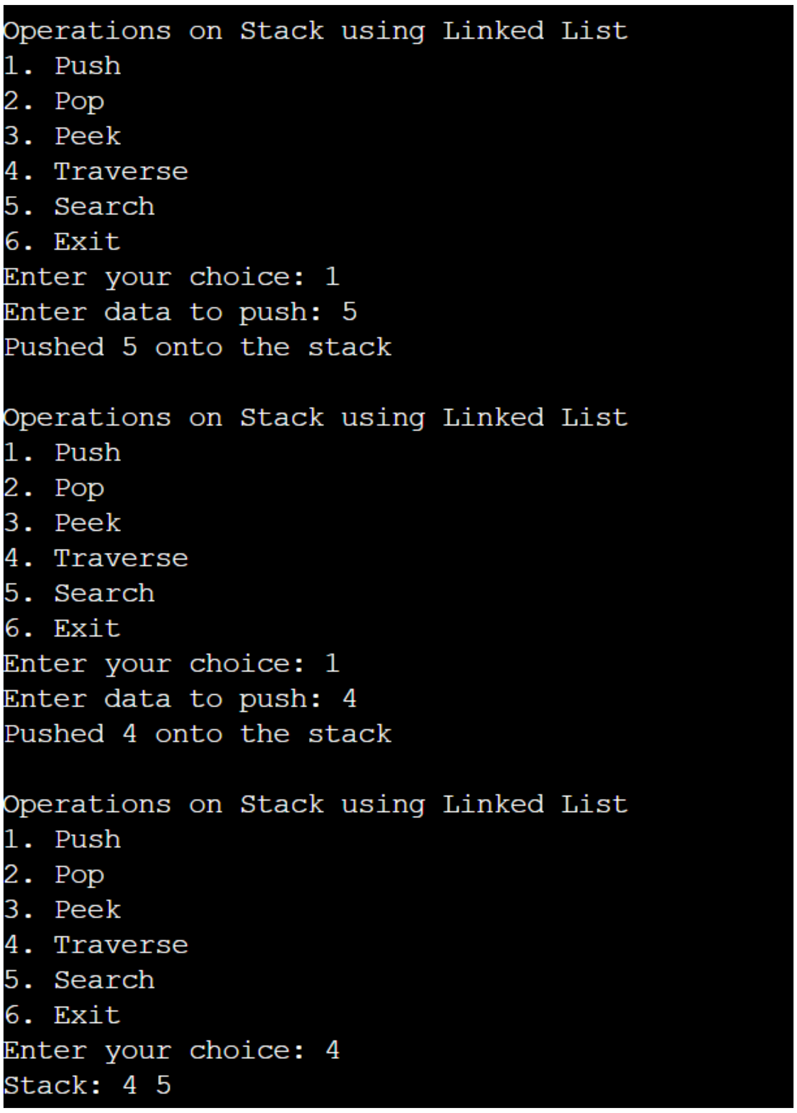
```
int peek() {
   if (top == NULL) {
      printf("Stack is empty\n");
      return -1;
   }
   return top->data;
}

// Function to traverse and display the stack
void traverse() {
   if (top == NULL) {
      printf("Stack is empty\n");
      return;
   }
   printf("Stack: ");
   struct Node* current = top;
   while (current != NULL) {
      printf("%d ", current->data);
      current = current->next;
   }
   printf("\n");
}

// Function to search for an element in the stack
int search(int key) {
   if (top == NULL) {
      printf("Stack is empty\n");
      return -1;
```

```
    }
    struct Node* current = top;
    int position = 0;
    while (current != NULL) {
        if (current->data == key) {
            return position;
        }
        position++;
        current = current->next;
    }
    return -1;
}

// Function to display the menu of operations
void displayMenu() {
    printf("\nOperations on Stack using Linked List\n");
    printf("1. Push\n");
    printf("2. Pop\n");
    printf("3. Peek\n");
    printf("4. Traverse\n");
    printf("5. Search\n");
    printf("6. Exit\n");
}

int main() {
    int choice, data, key;
    while (1) {
        displayMenu();
```

```
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
    case 1:
        printf("Enter data to push: ");
        scanf("%d", &data);
        push(data);
        break;
    case 2:
        pop();
        break;
    case 3:
        data = peek();
        if (data != -1) {
            printf("Top element of stack: %d\n", data);
        }
        break;
    case 4:
        traverse();
        break;
    case 5:
        printf("Enter element to search: ");
        scanf("%d", &key);
        if (search(key) != -1) {
            printf("%d found in the stack\n", key);
        } else {
            printf("%d not found in the stack\n", key);
        }
```

```
            break;

          case 6:

            printf("Exiting...\n");

            exit(0);

          default:

            printf("Invalid choice\n");

        }

    }

    return 0;

}
```

OUTPUT –



```
Operations on Stack using Linked List
1. Push
2. Pop
3. Peek
4. Traverse
5. Search
6. Exit
Enter your choice: 1
Enter data to push: 5
Pushed 5 onto the stack

Operations on Stack using Linked List
1. Push
2. Pop
3. Peek
4. Traverse
5. Search
6. Exit
Enter your choice: 1
Enter data to push: 4
Pushed 4 onto the stack

Operations on Stack using Linked List
1. Push
2. Pop
3. Peek
4. Traverse
5. Search
6. Exit
Enter your choice: 4
Stack: 4 5
```

PROGRAM 16

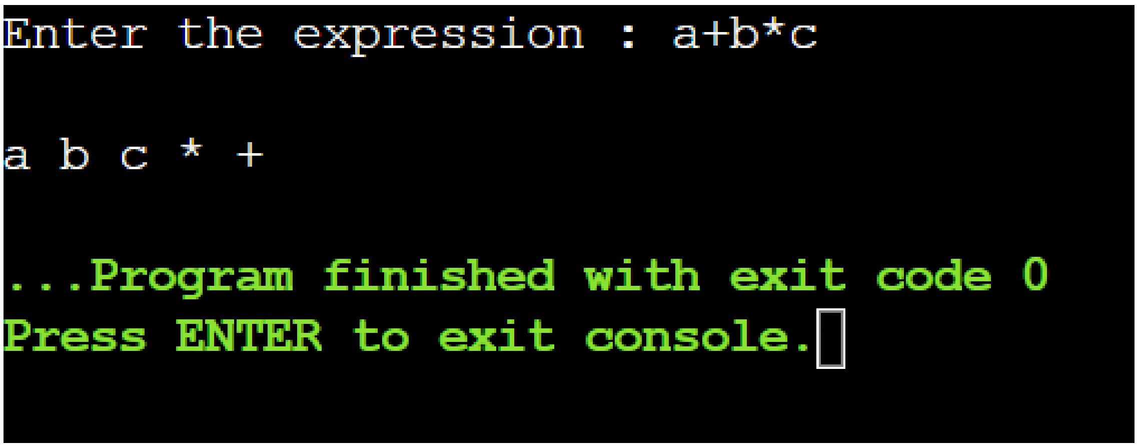
Write a program to convert an expression from Infix to Postfix using Stack.

SOL –

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

char stack[100];
int top = -1;

void push(char x)
{
    stack[++top] = x;
}

char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}

int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
```

```
        return 1;
    if(x == '*' || x == '/')
        return 2;
    return 0;
}

int main()
{
    char exp[100];
    char *e, x;
    printf("Enter the expression : ");
    scanf("%s",exp);
    printf("\n");
    e = exp;

    while(*e != '\0')
    {
        if(isalnum(*e))
            printf("%c ",*e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
            while((x = pop()) != '(')
                printf("%c ", x);
        }
        else
        {
            while(priority(stack[top]) >= priority(*e))
                printf("%c ",pop());
            push(*e);
        }
        e++;
    }


    while(top != -1)
    {
        printf("%c ",pop());
    }return 0;
}
```

OUTPUT –



```
Enter the expression : a+b*c

a b c * +

...Program finished with exit code 0
Press ENTER to exit console.
```

PROGRAM 17

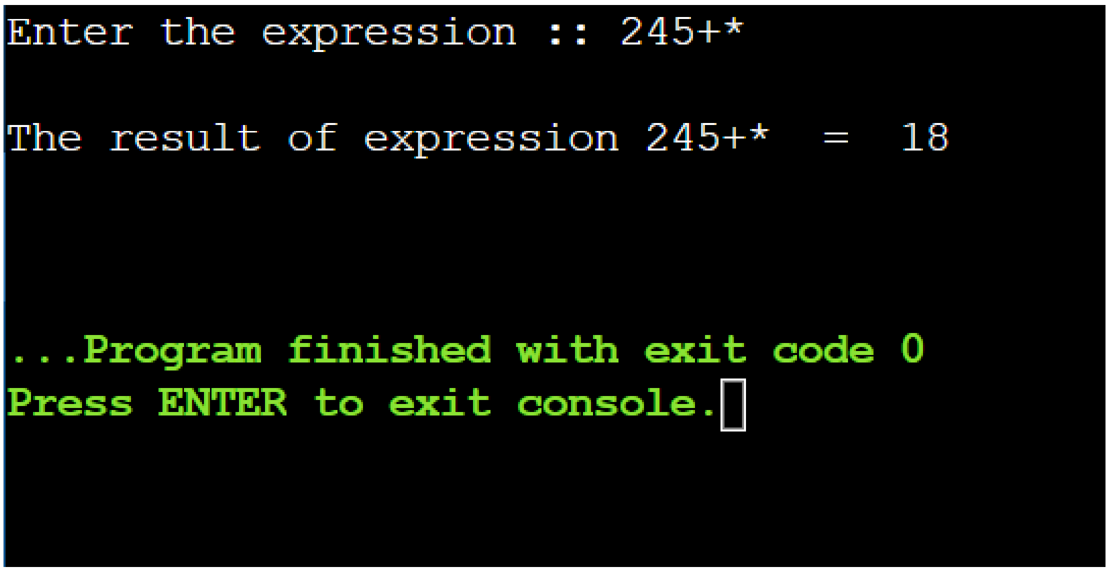
Write a program to evaluate a Postfix expression using Stack.

SOL –

```
#include<stdio.h>
int stack[20];
int top = -1;

void push(int x)
{
   stack[++top] = x;
}

int pop()
{
   return stack[top--];
}

int main()
{
   char exp[20];
   char *e;
   int n1,n2,n3,num;
   printf("Enter the expression :: ");
   scanf("%s",exp);
   e = exp;
   while(*e != '\0')
```

```
{
   if(isdigit(*e))
   {
      num = *e - 48;
      push(num);
   }
   else
   {
      n1 = pop();
      n2 = pop();
      switch(*e)
      {
      case '+':
      {
         n3 = n1 + n2;
         break;
      }
      case '-':
      {
         n3 = n2 - n1;
         break;
      }
      case '*':
      {
         n3 = n1 * n2;
         break;
      }
      case '/':
            {
               n3 = n2 / n1;
               break;
            }
            }
            push(n3);
         }
         e++;
      }
      printf("\nThe result of expression %s  =  %d\n\n",exp,pop());
      return 0;
}
```

OUTPUT –



```
Enter the expression :: 245+*

The result of expression 245+*   =   18



...Program finished with exit code 0
Press ENTER to exit console.
```

PROGRAM 18

Write a program to implement Linear queue using array and perform the following operation on it.

1. Enqueue.

2. Dequeue.

3. Traverse.

4. Peek

5. Search

6. Max element in Queue

SOL –

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX_SIZE 100


// Global variables

int queue[MAX_SIZE];

int front = -1;

int rear = -1;


// Function to check if the queue is full

bool isFull() {

   return rear == MAX_SIZE - 1;

}
```

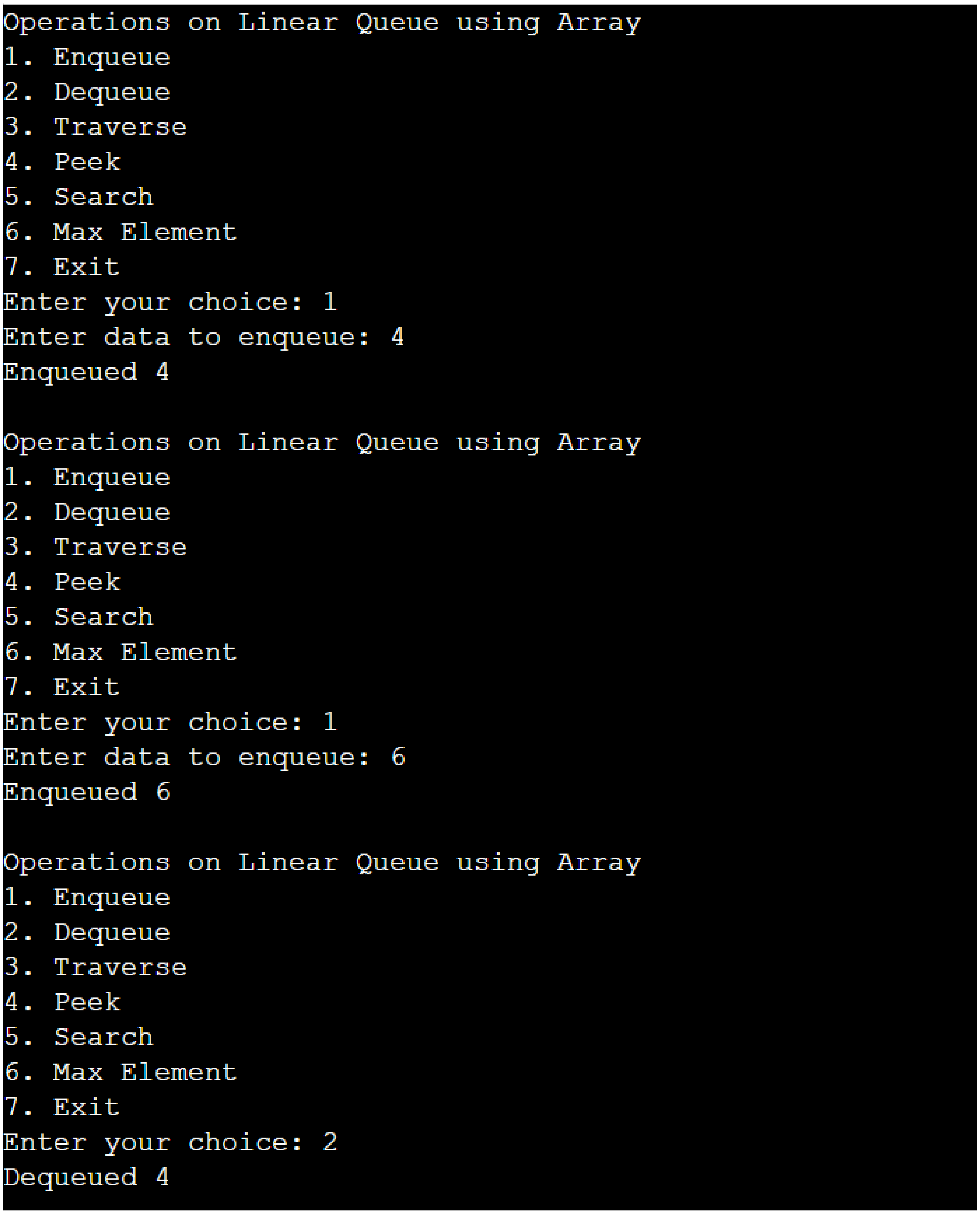
```
// Function to check if the queue is empty
bool isEmpty() {
    return front == -1 || front > rear;
}

// Function to enqueue an element
void enqueue(int data) {
    if (isFull()) {
        printf("Queue overflow\n");
        return;
    }
    if (front == -1) {
        front = 0;
    }
    queue[++rear] = data;
    printf("Enqueued %d\n", data);
}

// Function to dequeue an element
int dequeue() {
    if (isEmpty()) {
        printf("Queue underflow\n");
        exit(1);
    }
    int data = queue[front++];
    printf("Dequeued %d\n", data);
    if (front > rear) { // Reset front and rear if queue becomes empty
        front = -1;
        rear = -1;
    }
    return data;
}

// Function to traverse and display the queue
void traverse() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

// Function to peek at the front element of the queue
int peek() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        exit(1);
    }
    return queue[front];
}

// Function to search for an element in the queue
```

```
bool search(int key) {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return false;
    }
    for (int i = front; i <= rear; i++) {
        if (queue[i] == key) {
            return true;
        }
    }
    return false;
}

// Function to find the maximum element in the queue
int maxElement() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        exit(1);
    }
    int max = queue[front];
    for (int i = front + 1; i <= rear; i++) {
        if (queue[i] > max) {
            max = queue[i];
        }
    }
    return max;
}
```

```
// Function to display the menu of operations
void displayMenu() {
    printf("\nOperations on Linear Queue using Array\n");
    printf("1. Enqueue\n");
    printf("2. Dequeue\n");
    printf("3. Traverse\n");
    printf("4. Peek\n");
    printf("5. Search\n");
    printf("6. Max Element\n");
    printf("7. Exit\n");
}

int main() {
    int choice, data, key;
    while (1) {
        displayMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to enqueue: ");
                scanf("%d", &data);
                enqueue(data);
                break;
            case 2:
                dequeue();
                break;
            case 3:
```

```
            traverse();
            break;
        case 4:
            printf("Front element of queue: %d\n", peek());
            break;
        case 5:
            printf("Enter element to search: ");
            scanf("%d", &key);
            if (search(key)) {
                printf("%d found in the queue\n", key);
            } else {
                printf("%d not found in the queue\n", key);
            }
            break;
        case 6:
            printf("Maximum element in the queue: %d\n", maxElement());
            break;
        case 7:
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}
```

OUTPUT –



```
Operations on Linear Queue using Array
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Search
6. Max Element
7. Exit
Enter your choice: 1
Enter data to enqueue: 4
Enqueued 4

Operations on Linear Queue using Array
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Search
6. Max Element
7. Exit
Enter your choice: 1
Enter data to enqueue: 6
Enqueued 6

Operations on Linear Queue using Array
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Search
6. Max Element
7. Exit
Enter your choice: 2
Dequeued 4
```

PROGRAM 19

Write a program to implement Linear queue using linked list and perform the following operation on it.

1. Enqueue.

2. Dequeue.

3. Traverse.

4. Peek

5. Search

6. Max element in Queue

SOL –

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


// Node structure

struct Node {

   int data;

   struct Node* next;

};


// Global variables

struct Node* front = NULL;

struct Node* rear = NULL;


// Function to create a new node

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to check if the queue is empty
bool isEmpty() {
    return front == NULL;
}

// Function to enqueue an element
void enqueue(int data) {
    struct Node* newNode = createNode(data);
    if (isEmpty()) {
        front = newNode;
    } else {
        rear->next = newNode;
    }
    rear = newNode;
    printf("Enqueued %d\n", data);
}

// Function to dequeue an element
```

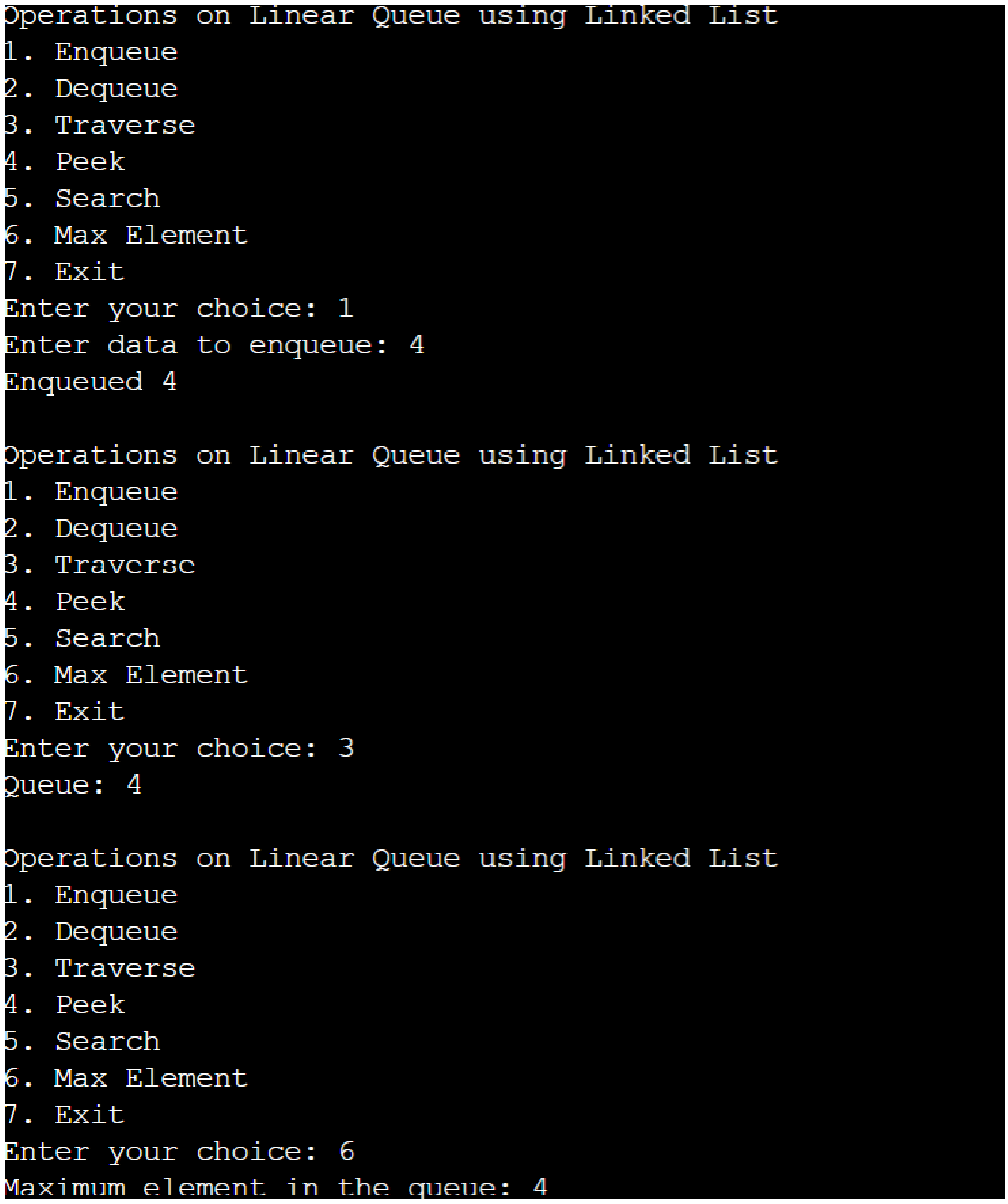
```
int dequeue() {
    if (isEmpty()) {
        printf("Queue underflow\n");
        exit(1);
    }
    struct Node* temp = front;
    int data = temp->data;
    front = front->next;
    free(temp);
    if (front == NULL) {
        rear = NULL;
    }
    printf("Dequeued %d\n", data);
    return data;
}

// Function to traverse and display the queue
void traverse() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    struct Node* current = front;
    printf("Queue: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// Function to peek at the front element of the queue
int peek() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        exit(1);
    }
    return front->data;
}

// Function to search for an element in the queue
bool search(int key) {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return false;
    }
    struct Node* current = front;
    while (current != NULL) {
        if (current->data == key) {
            return true;
        }
        current = current->next;
    }
    return false;
}
```

```
// Function to find the maximum element in the queue
int maxElement() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        exit(1);
    }
    int max = front->data;
    struct Node* current = front->next;
    while (current != NULL) {
        if (current->data > max) {
            max = current->data;
        }
        current = current->next;
    }
    return max;
}

// Function to display the menu of operations
void displayMenu() {
    printf("\nOperations on Linear Queue using Linked List\n");
    printf("1. Enqueue\n");
    printf("2. Dequeue\n");
    printf("3. Traverse\n");
    printf("4. Peek\n");
    printf("5. Search\n");
    printf("6. Max Element\n");
    printf("7. Exit\n");
}
```

```
int main() {
    int choice, data, key;
    while (1) {
        displayMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to enqueue: ");
                scanf("%d", &data);
                enqueue(data);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                traverse();
                break;
            case 4:
                printf("Front element of queue: %d\n", peek());
                break;
            case 5:
                printf("Enter element to search: ");
                scanf("%d", &key);
                if (search(key)) {
                    printf("%d found in the queue\n", key);
                } else {
```

```
                printf("%d not found in the queue\n", key);
            }
            break;
        case 6:
            printf("Maximum element in the queue: %d\n", maxElement());
            break;
        case 7:
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}
```

OUTPUT –



```
Operations on Linear Queue using Linked List
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Search
6. Max Element
7. Exit
Enter your choice: 1
Enter data to enqueue: 4
Enqueued 4

Operations on Linear Queue using Linked List
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Search
6. Max Element
7. Exit
Enter your choice: 3
Queue: 4

Operations on Linear Queue using Linked List
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Search
6. Max Element
7. Exit
Enter your choice: 6
Maximum element in the queue: 4
```

PROGRAM 20

Write a program to implement Circular queue using array and perform the following operation on it.

1. Enqueue.

2. Dequeue.

3. Traverse.

4. Peek

5. Search

6. Max element in Queue

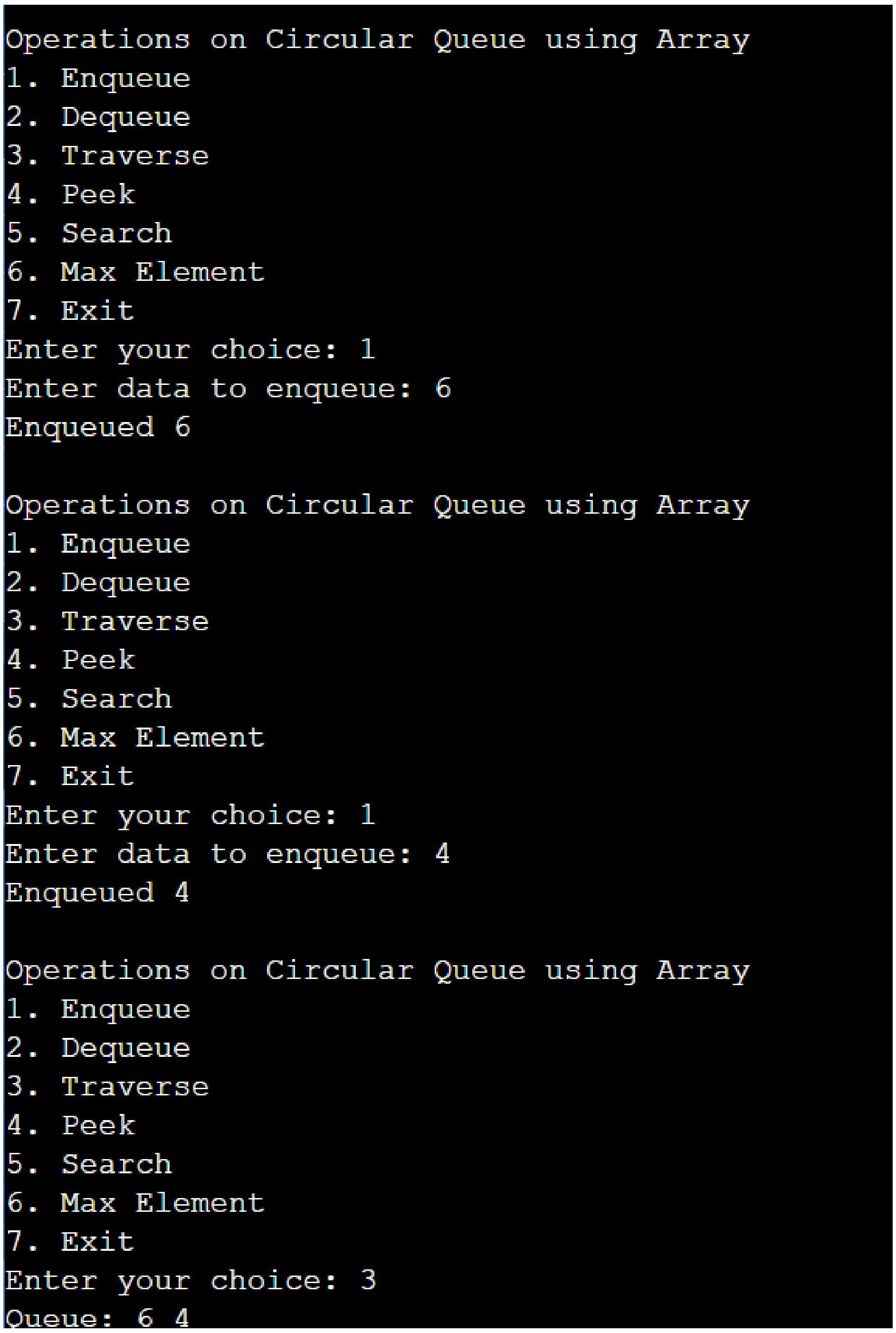
SOL –

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX_SIZE 100


// Global variables

int queue[MAX_SIZE];

int front = -1;

int rear = -1;


// Function to check if the queue is full

bool isFull() {

    return (front == 0 && rear == MAX_SIZE - 1) || (rear == front - 1);

}
```

```
// Function to check if the queue is empty
bool isEmpty() {
    return front == -1;
}

// Function to enqueue an element
void enqueue(int data) {
    if (isFull()) {
        printf("Queue overflow\n");
        return;
    }
    if (front == -1) {
        front = 0;
        rear = 0;
    } else if (rear == MAX_SIZE - 1) {
        rear = 0;
    } else {
        rear++;
    }
    queue[rear] = data;
    printf("Enqueued %d\n", data);
}

// Function to dequeue an element
int dequeue() {
    if (isEmpty()) {
        printf("Queue underflow\n");
        exit(1);
    }
    int data = queue[front];
    if (front == rear) { // Reset front and rear if only one element in queue
        front = -1;
        rear = -1;
    } else if (front == MAX_SIZE - 1) {
        front = 0;
    } else {
        front++;
    }
    printf("Dequeued %d\n", data);
    return data;
}

// Function to traverse and display the queue
void traverse() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue: ");
    if (front <= rear) {
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
    } else {
        for (int i = front; i < MAX_SIZE; i++) {
            printf("%d ", queue[i]);
        }
        for (int i = 0; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
    }
    printf("\n");
}

// Function to peek at the front element of the queue
int peek() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        exit(1);
    }
    return queue[front];
}

// Function to search for an element in the queue
bool search(int key) {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return false;
    }
    if (front <= rear) {
        for (int i = front; i <= rear; i++) {
            if (queue[i] == key) {
                return true;
            }
        }
    } else {
        for (int i = front; i < MAX_SIZE; i++) {
            if (queue[i] == key) {
                return true;
            }
        }
        for (int i = 0; i <= rear; i++) {
            if (queue[i] == key) {
                return true;
            }
        }
    }
    return false;
}

// Function to find the maximum element in the queue
int maxElement() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        exit(1);
    }
    int max = queue[front];
    if (front <= rear) {
        for (int i = front + 1; i <= rear; i++) {
            if (queue[i] > max) {
                max = queue[i];
            }
        }
    } else {
        for (int i = front + 1; i < MAX_SIZE; i++) {
            if (queue[i] > max) {
                max = queue[i];
            }
        }
        for (int i = 0; i <= rear; i++) {
            if (queue[i] > max) {
                max = queue[i];
            }
        }
    }
    return max;
}

// Function to display the menu of operations
void displayMenu() {
    printf("\nOperations on Circular Queue using Array\n");
    printf("1. Enqueue\n");
    printf("2. Dequeue\n");
    printf("3. Traverse\n");
    printf("4. Peek\n");
    printf("5. Search\n");
    printf("6. Max Element\n");
    printf("7. Exit\n");
}
```

```
int main() {
    int choice, data, key;
    while (1) {
        displayMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to enqueue: ");
                scanf("%d", &data);
                enqueue(data);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                traverse();
                break;
            case 4:
                printf("Front element of queue: %d\n", peek());
                break;
            case 5:
                printf("Enter element to search: ");
                scanf("%d", &key);
                if (search(key)) {
                    printf("%d found in the queue\n", key);
                } else {
                    printf("%d not found in the queue\n", key);
            }
            break;
         case 6:
            printf("Maximum element in the queue: %d\n", maxElement());
            break;
         case 7:
            printf("Exiting...\n");
            exit(0);
         default:
            printf("Invalid choice\n");
      }
   }
   return 0;
}
```

OUTPUT –



```
Operations on Circular Queue using Array
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Search
6. Max Element
7. Exit
Enter your choice: 1
Enter data to enqueue: 6
Enqueued 6

Operations on Circular Queue using Array
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Search
6. Max Element
7. Exit
Enter your choice: 1
Enter data to enqueue: 4
Enqueued 4

Operations on Circular Queue using Array
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Search
6. Max Element
7. Exit
Enter your choice: 3
Queue: 6 4
```

PROGRAM 21

Write a program to implement Circular queue using Linked List and perform the following operation on it.

1. Enqueue.

2. Dequeue.

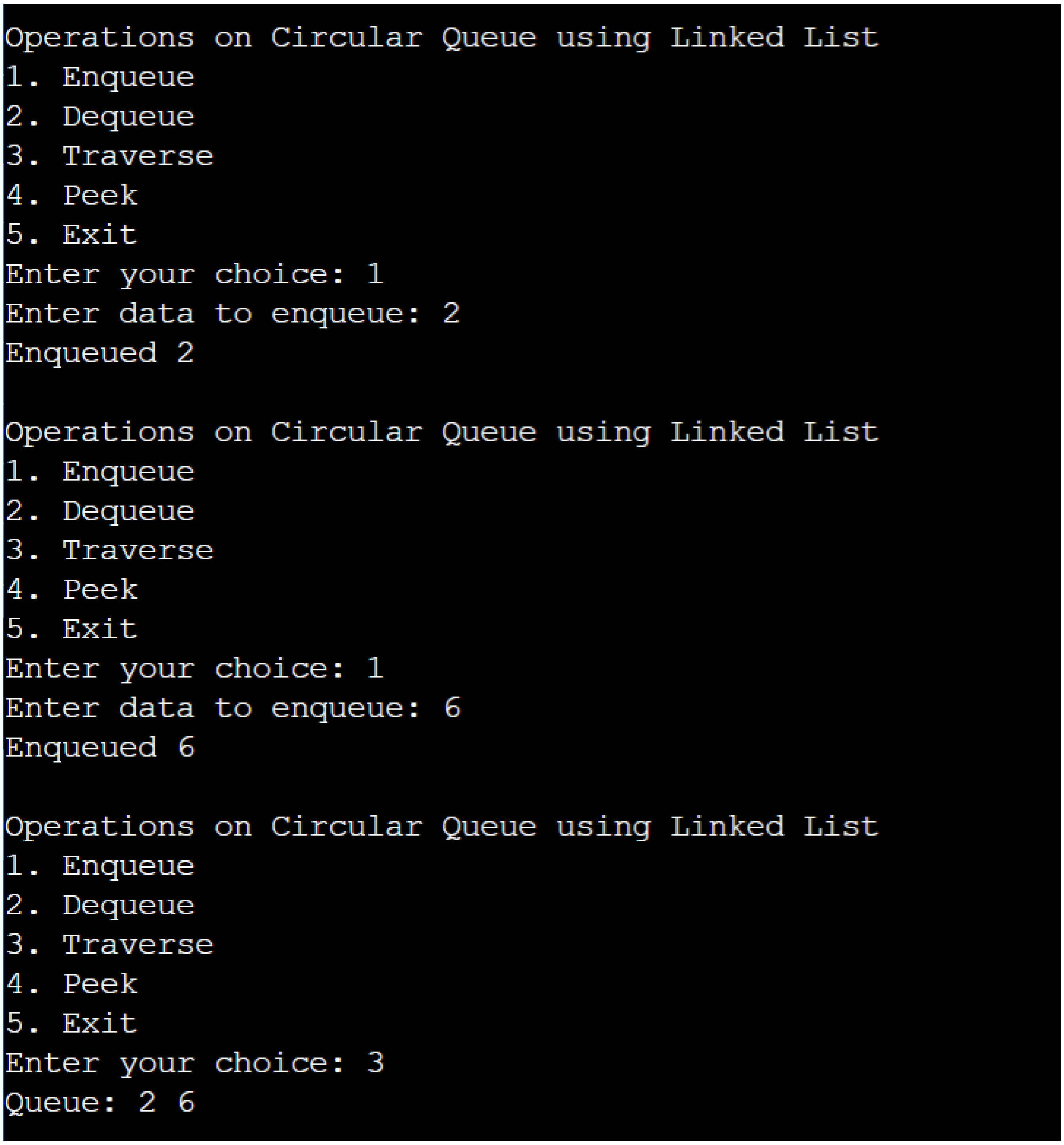
3. Traverse.

4. Peek

SOL –

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


// Node structure

struct Node {

    int data;

    struct Node* next;

};


// Global variables

struct Node* front = NULL;

struct Node* rear = NULL;


// Function to create a new node

struct Node* createNode(int data) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    if (newNode == NULL) {

        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to check if the queue is empty
bool isEmpty() {
    return front == NULL;
}

// Function to enqueue an element
void enqueue(int data) {
    struct Node* newNode = createNode(data);
    if (isEmpty()) {
        front = newNode;
    } else {
        rear->next = newNode;
    }
    rear = newNode;
    rear->next = front; // Make rear node point to front for circularity
    printf("Enqueued %d\n", data);
}

// Function to dequeue an element
int dequeue() {
    if (isEmpty()) {
        printf("Queue underflow\n");
        exit(1);
    }
    int data = front->data;
    struct Node* temp = front;
    if (front == rear) { // Reset front and rear if only one element in queue
        front = NULL;
        rear = NULL;
    } else {
        front = front->next;
        rear->next = front; // Make rear node point to front for circularity
    }
    free(temp);
    printf("Dequeued %d\n", data);
    return data;
}

// Function to traverse and display the queue
void traverse() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    struct Node* current = front;
    printf("Queue: ");
    do {
        printf("%d ", current->data);
        current = current->next;
    } while (current != front);
    printf("\n");
}

// Function to peek at the front element of the queue
int peek() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        exit(1);
    }
    return front->data;
}

// Function to display the menu of operations
void displayMenu() {
    printf("\nOperations on Circular Queue using Linked List\n");
    printf("1. Enqueue\n");
    printf("2. Dequeue\n");
    printf("3. Traverse\n");
    printf("4. Peek\n");
    printf("5. Exit\n");
}

int main() {
    int choice, data;
    while (1) {
        displayMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to enqueue: ");
                scanf("%d", &data);
                enqueue(data);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                traverse();
                break;
            case 4:
                printf("Front element of queue: %d\n", peek());
                break;
            case 5:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
}
```

OUTPUT –



```
Operations on Circular Queue using Linked List
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Exit
Enter your choice: 1
Enter data to enqueue: 2
Enqueued 2

Operations on Circular Queue using Linked List
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Exit
Enter your choice: 1
Enter data to enqueue: 6
Enqueued 6

Operations on Circular Queue using Linked List
1. Enqueue
2. Dequeue
3. Traverse
4. Peek
5. Exit
Enter your choice: 3
Queue: 2 6
```

PROGRAM 22

Write a program to implement concept of MutiStack and Multi-Queue.

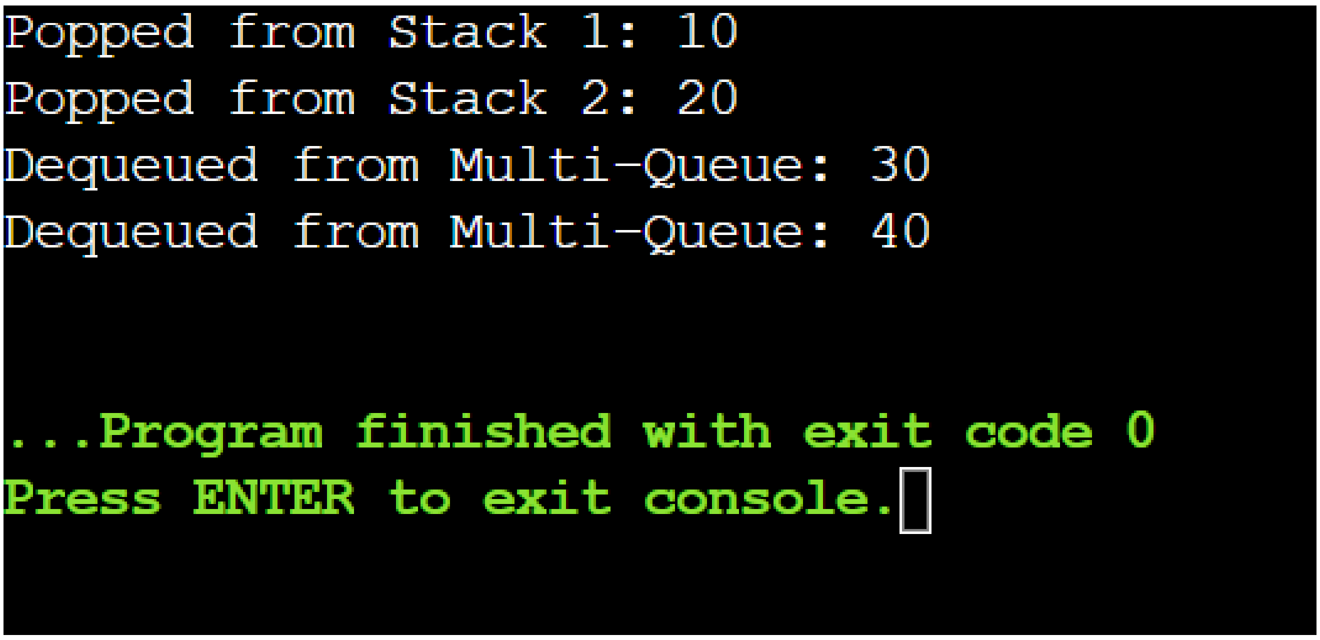
SOL –

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX_SIZE 100


// Global variables for Multi-Stack

int multiStack[MAX_SIZE];

int top1 = -1; // Top index of Stack 1

int top2 = MAX_SIZE; // Top index of Stack 2


// Global variables for Multi-Queue

int front = -1; // Front index of Queue

int rear = -1; // Rear index of Queue


// Function to check if the Multi-Stack is full

bool isStackFull() {

   return (top1 + 1) == top2;

}


// Function to check if the Multi-Stack is empty

bool isStackEmpty() {

   return (top1 == -1 && top2 == MAX_SIZE);

}
```

```
// Function to push an element into Stack 1
void pushStack1(int data) {
    if (isStackFull()) {
        printf("Stack 1 overflow\n");
        exit(1);
    }
    multiStack[++top1] = data;
}

// Function to pop an element from Stack 1
int popStack1() {
    if (top1 == -1) {
        printf("Stack 1 underflow\n");
        exit(1);
    }
    return multiStack[top1--];
}

// Function to push an element into Stack 2
void pushStack2(int data) {
    if (isStackFull()) {
        printf("Stack 2 overflow\n");
        exit(1);
    }
    multiStack[--top2] = data;
}
```

```
// Function to pop an element from Stack 2
int popStack2() {
    if (top2 == MAX_SIZE) {
        printf("Stack 2 underflow\n");
        exit(1);
    }
    return multiStack[top2++];
}

// Function to check if the Multi-Queue is full
bool isQueueFull() {
    return (rear + 1) % MAX_SIZE == front;
}

// Function to check if the Multi-Queue is empty
bool isQueueEmpty() {
    return front == -1;
}

// Function to enqueue an element into the Multi-Queue
void enqueue(int data) {
    if (isQueueFull()) {
        printf("Queue overflow\n");
        exit(1);
    }
    if (front == -1) {
        front = 0;
    }
    rear = (rear + 1) % MAX_SIZE;
    multiStack[rear] = data;
}

// Function to dequeue an element from the Multi-Queue
int dequeue() {
    if (isQueueEmpty()) {
        printf("Queue underflow\n");
        exit(1);
    }
    int data = multiStack[front];
    if (front == rear) { // Reset front and rear if only one element in queue
        front = -1;
        rear = -1;
    } else {
        front = (front + 1) % MAX_SIZE;
    }
    return data;
}

int main() {
    // Example usage of Multi-Stack
    pushStack1(10); // Pushing into Stack 1
    pushStack2(20); // Pushing into Stack 2
    printf("Popped from Stack 1: %d\n", popStack1()); // Popping from Stack 1
    printf("Popped from Stack 2: %d\n", popStack2()); // Popping from Stack 2

    // Example usage of Multi-Queue
    enqueue(30); // Enqueuing into Multi-Queue

    enqueue(40); // Enqueuing into Multi-Queue

    printf("Dequeued from Multi-Queue: %d\n", dequeue()); // Dequeuing from Multi-Queue

    printf("Dequeued from Multi-Queue: %d\n", dequeue()); // Dequeuing from Multi-Queue


    return 0;
}
```

OUTPUT –



```
Popped from Stack 1: 10
Popped from Stack 2: 20
Dequeued from Multi-Queue: 30
Dequeued from Multi-Queue: 40


...Program finished with exit code 0
Press ENTER to exit console.
```

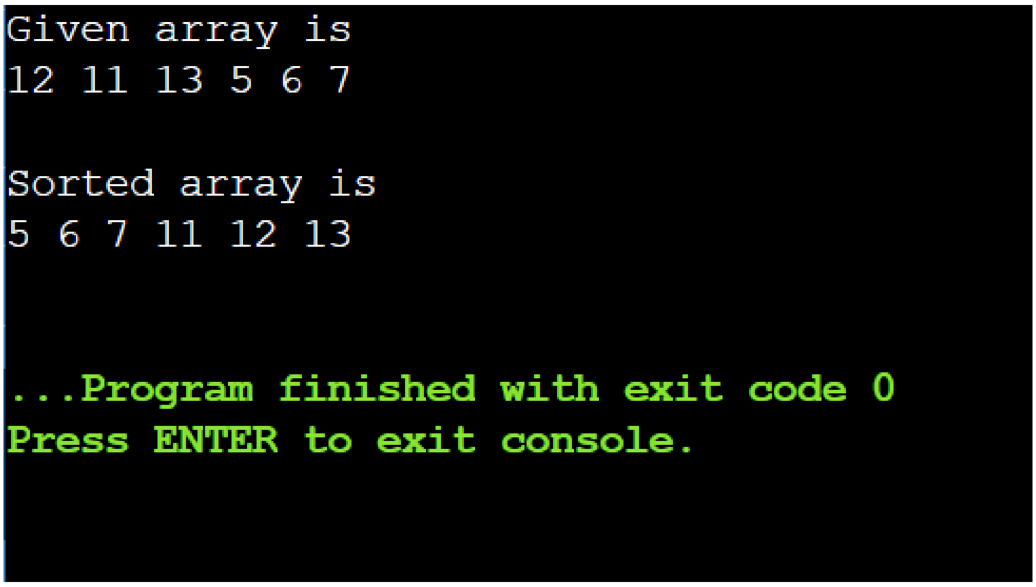
PROGRAM 23

Write a program to implement Merge Sort.

SOL –

```
#include <stdio.h>


// Function to merge two subarrays of arr[]
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int L[n1], R[n2];

    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temporary arrays back into arr[left..right]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Function to perform Merge Sort on arr[left..right]
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        // Find the middle point
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

OUTPUT –



```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13


...Program finished with exit code 0
Press ENTER to exit console.
```

PROGRAM 24

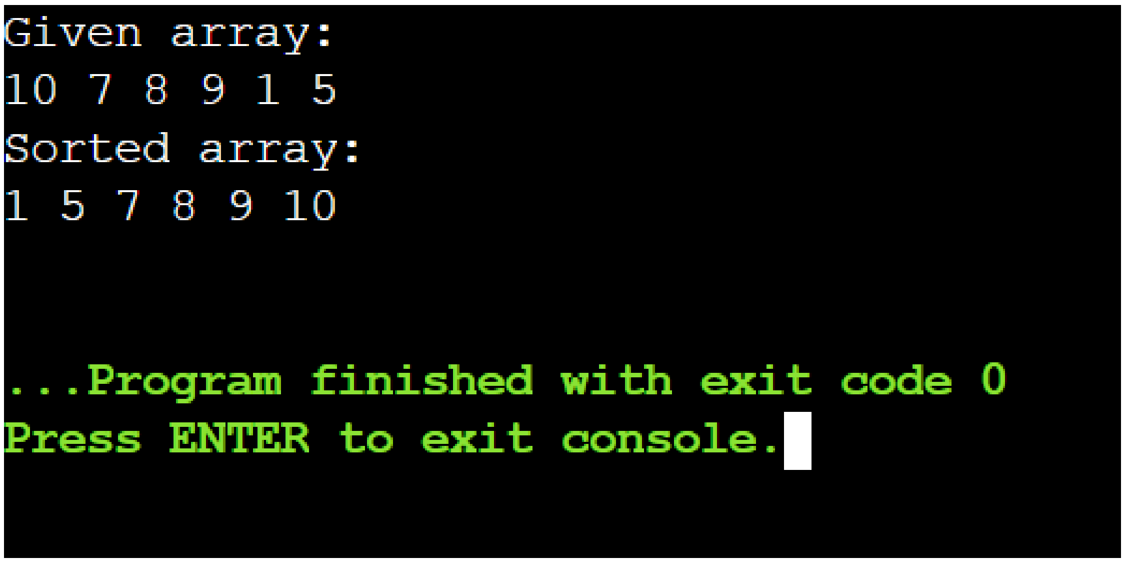
Write a program to implement Quick Sort.

SOL –

```
#include <stdio.h>


// Function to swap two elements
void swap(int arr[], int a, int b) {
   int temp = arr[a];
   arr[a] = arr[b];
   arr[b] = temp;
}


// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
   int pivot = arr[high]; // Pivot element (last element)
   int i = low - 1; // Index of smaller element


   for (int j = low; j <= high - 1; j++) {
      // If current element is smaller than the pivot
      if (arr[j] < pivot) {
         i++; // Increment index of smaller element
         swap(arr, i, j);
      }
   }
   swap(arr, i + 1, high);
   return (i + 1);
}
```

```
// Function to implement Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array
        int pi = partition(arr, low, high);

        // Sort the subarrays recursively
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Given array: \n");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);
    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}
```

OUTPUT –



```
Given array:
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10


...Program finished with exit code 0
Press ENTER to exit console.
```

PROGRAM 25

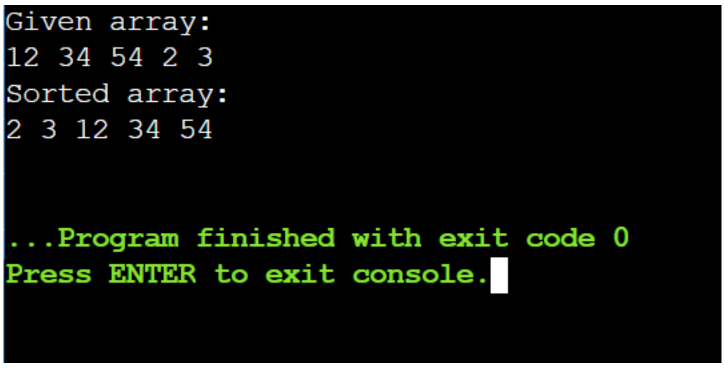
Write a program to implement Shell Sort.

SOL –

```
#include <stdio.h>


// Function to perform Shell Sort
void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}


// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
int main() {
   int arr[] = {12, 34, 54, 2, 3};
   int n = sizeof(arr) / sizeof(arr[0]);


   printf("Given array: \n");
   printArray(arr, n);


   shellSort(arr, n);


   printf("Sorted array: \n");
   printArray(arr, n);


   return 0;
}
```

OUTPUT –



```
Given array:
12 34 54 2 3
Sorted array:
2 3 12 34 54


...Program finished with exit code 0
Press ENTER to exit console.
```