DIGITAL ASSIGNMENT -1

NAME: ROHIT KANNA.S
REG NO: 24BBS0212
COURSE CODE: CBS1003
COURSE NAME: DATA STRUCTURES AND ALGORITHM
SUBMISSION DATE: 25/01/25

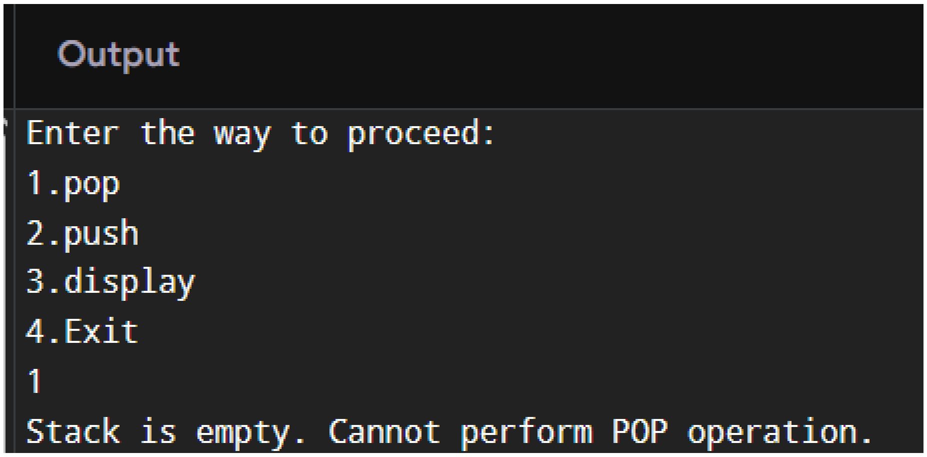
Q1)1. Write a menu driven program to implement the following operations on stack.
a. PUSH() b. POP() c. Display()

ANS:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node *next;
};

struct node *start = NULL;

void push() {
    int num;
    struct node *t = (struct node *)malloc(sizeof(struct node));
    if (t == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }

    printf("Enter the data to be pushed: ");
    scanf("%d", &num);
    t->data = num;
    t->next = start;
    start = t;
    printf("%d pushed onto the stack.\n", num);
}

void pop() {
    if (start == NULL) {
        printf("Stack is empty. Cannot perform POP operation.\n");
    } else {
        struct node *t = start;
        printf("%d popped from the stack.\n", t->data);
        start = start->next;
        free(t);
    }
}
void display() {
    if (start == NULL) {
        printf("Stack is empty.\n");
    } else {
        struct node *temp = start;
        printf("Stack contents: ");
        while (temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}
int main()
{
    int n;
    while(1)
    {
     printf("Enter the way to proceed:\n");
     printf("1.pop\n2.push\n3.display\n4.Exit\n");
     scanf("%d",&n);

     switch(n)
     {
         case 1:
         {
             pop();
             break;
         }
         case 2:
         {
             push();
             break;
         }
         case 3:
         {
             display();
             break;
         }
         case 4:
         {
             printf("Exiting");
             break;
         }
         default:
              printf(“Enter a valid choice\n”);
              break;
     }
}
}
```

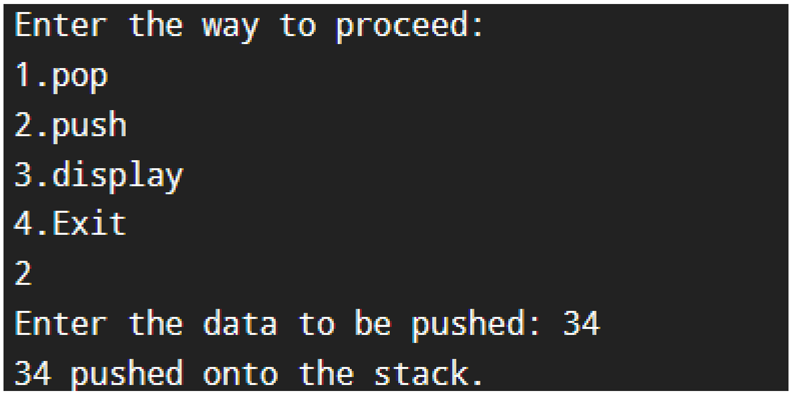
TEST CASE

1)



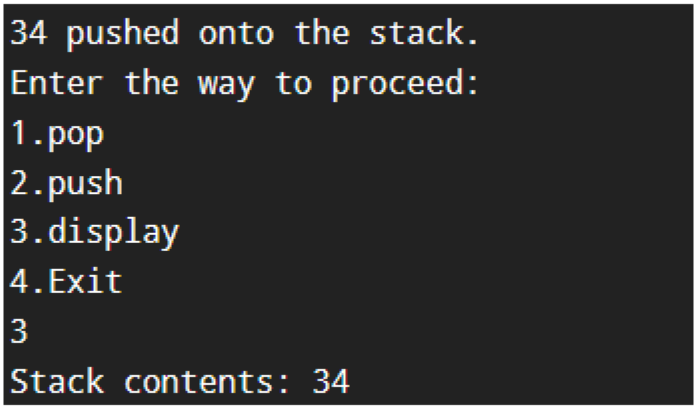
```
Enter the way to proceed:
1.pop
2.push
3.display
4.Exit
1
Stack is empty. Cannot perform POP operation.
```

2)

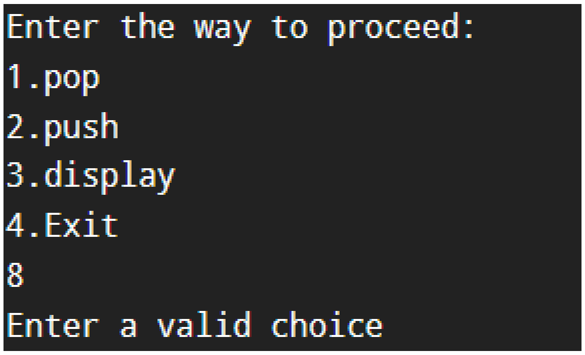


```
Enter the way to proceed:
1.pop
2.push
3.display
4.Exit
2
Enter the data to be pushed: 34
34 pushed onto the stack.
```

3)



```
34 pushed onto the stack.
Enter the way to proceed:
1.pop
2.push
3.display
4.Exit
3
Stack contents: 34
```



```
Enter the way to proceed:
1.pop
2.push
3.display
4.Exit
8
Enter a valid choice
```

Pseudocode:

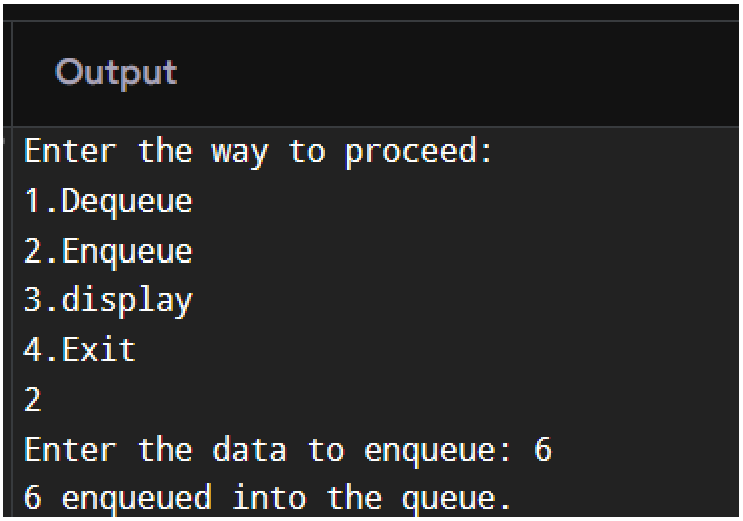
```
START
    Initialize start as NULL
    DO
        Display menu:
        1. POP
        2. PUSH
        3. DISPLAY
        4. EXIT
        Read user choice as n
        SWITCH n:
        CASE 1:
                IF start == NULL THEN
                Print "Stack is empty."
                ELSE
                TEMP = start
                Print TEMP.data "popped from the stack."
                start = start.next
                Free memory of TEMP
                END IF
                BREAK
        CASE 2:
                Allocate memory for new node T
                IF memory allocation failed THEN
                Print "Memory allocation failed."
                EXIT CASE
                END IF
                Read num from user
                T.data = num
                T.next = start
                start = T
                Print num "pushed onto the stack."
                BREAK
        CASE 3:
                IF start == NULL THEN
                Print "Stack is empty."
                ELSE
                TEMP = start
                Print "Stack contents: "
                WHILE TEMP != NULL
                Print TEMP.data
                TEMP = TEMP.next
                END WHILE
                END IF
```

```
        BREAK

    CASE 4:

        Print "Exiting"

        EXIT

    DEFAULT:

        Print "Invalid choice."

    END SWITCH

WHILE TRUE

STOP
```

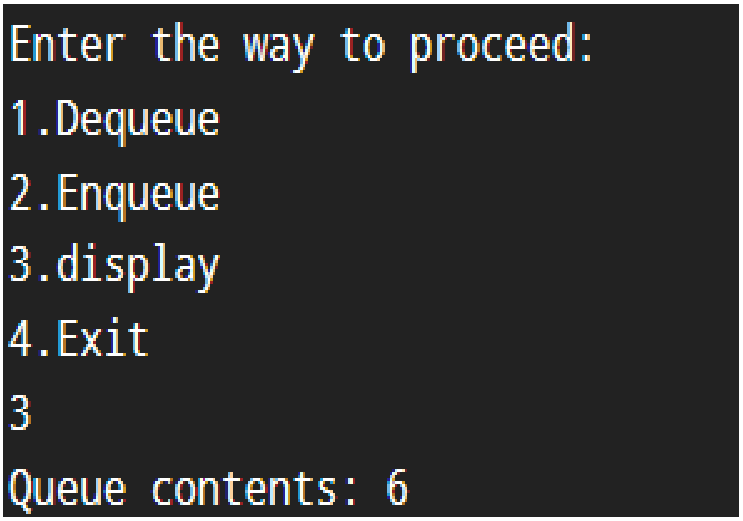
2. Write a menu driven program to implement the following operations on Queue: a. Enqueue() b. Dequeue() c. Disaply()

ANS:

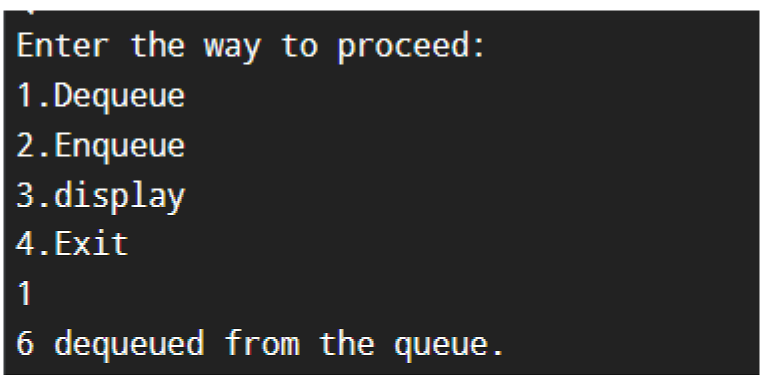
```
#include <stdio.h>
#include <stdlib.h>

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

struct node *front = NULL, *rear = NULL;

void enqueue() {
    int num;
    struct node *t = (struct node *)malloc(sizeof(struct node));
    if (t == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }

    printf("Enter the data to enqueue: ");
    scanf("%d", &num);
    t->data = num;
    t->next = NULL;

    if (rear == NULL) {
        front = rear = t;
    } else {
        rear->next = t;
        rear = t;
    }
    printf("%d enqueued into the queue.\n", num);
}

void dequeue() {
    if (front == NULL) {
        printf("Queue is empty. Cannot perform DEQUEUE operation.\n");
    } else {
        struct node *t = front;
        printf("%d dequeued from the queue.\n", t->data);
        front = front->next;
        if (front == NULL) {
            rear = NULL;
        }
        free(t);
    }
}

void display() {
    if (front == NULL) {
        printf("Queue is empty.\n");
    } else {
        struct node *temp = front;
        printf("Queue contents: ");
        while (temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}
int main()
{
    int n;
    while(1)
    {
     printf("Enter the way to proceed:\n");
     printf("1.Dequeue\n2.Enqueue\n3.display\n4.Exit\n");
     scanf("%d",&n);

     switch(n)
     {
         case 1:
         {
             dequeue();
             break;
         }
         case 2:
         {
             enqueue();
             break;
         }
         case 3:
         {
             display();
             break;
         }
         case 4:
         {
             printf("Exiting");
             return 0;
         }
         default:
         printf("Enter a valid choice");
         break;
     }

}
}
```



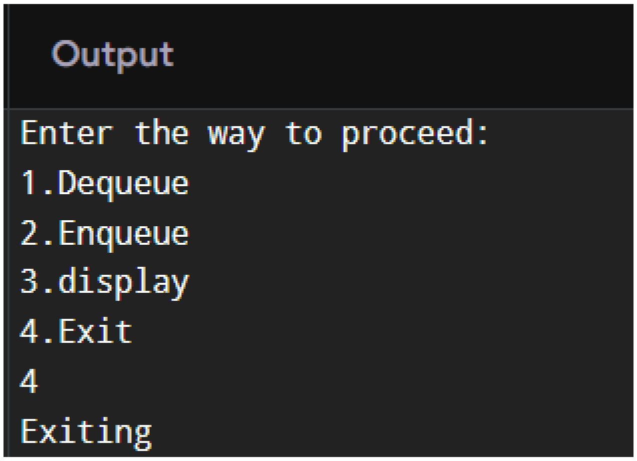
```
Enter the way to proceed:
1.Dequeue
2.Enqueue
3.display
4.Exit
2
Enter the data to enqueue: 6
6 enqueued into the queue.
```



```
Enter the way to proceed:
1.Dequeue
2.Enqueue
3.display
4.Exit
3
Queue contents: 6
```



```
Enter the way to proceed:
1.Dequeue
2.Enqueue
3.display
4.Exit
1
6 dequeued from the queue.
```



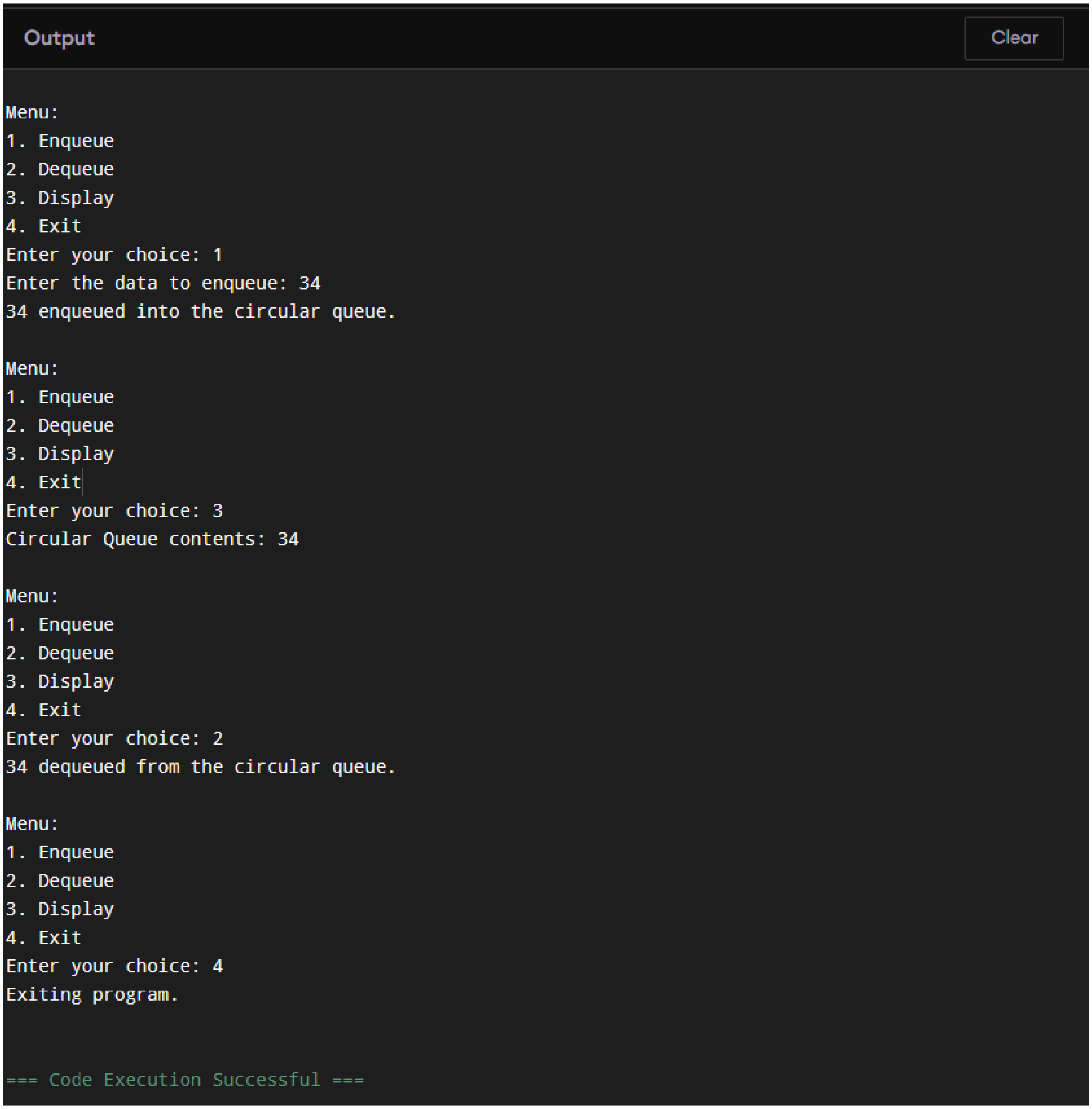
```
Enter the way to proceed:
1.Dequeue
2.Enqueue
3.display
4.Exit
4
Exiting
```

Pseudocode:

```
START

Set FRONT = NULL
Set REAR = NULL

FUNCTION ENQUEUE()
    Allocate memory for NEW_NODE
    IF memory allocation fails THEN
        Print "Memory allocation failed"
        RETURN
    ENDIF
    Print "Enter data to enqueue:"
    Input DATA
    Set NEW_NODE.data = DATA
    Set NEW_NODE.next = NULL

    IF REAR == NULL THEN
        Set FRONT = NEW_NODE
        Set REAR = NEW_NODE
    ELSE
        Set REAR.next = NEW_NODE
        Set REAR = NEW_NODE
    ENDIF
    Print DATA + " enqueued into the queue"
END FUNCTION

FUNCTION DEQUEUE()
    IF FRONT == NULL THEN
        Print "Queue is empty. Cannot perform DEQUEUE operation"
        RETURN
    ENDIF
    Set TEMP = FRONT
    Print TEMP.data + " dequeued from the queue"
    Set FRONT = FRONT.next
    IF FRONT == NULL THEN
        Set REAR = NULL
    ENDIF
    Free TEMP
END FUNCTION

FUNCTION DISPLAY()
    IF FRONT == NULL THEN
        Print "Queue is empty"
        RETURN
    ENDIF
    Set TEMP = FRONT
    Print "Queue contents:"
    WHILE TEMP != NULL DO
        Print TEMP.data
        Set TEMP = TEMP.next
    ENDWHILE
END FUNCTION

MAIN PROGRAM
WHILE TRUE DO
    Print "Enter the way to proceed:"
    Print "1. Dequeue"
    Print "2. Enqueue"
    Print "3. Display"
    Print "4. Exit"
    Input CHOICE

    SWITCH CHOICE
        CASE 1:
            CALL DEQUEUE()
            BREAK
        CASE 2:
            CALL ENQUEUE()
            BREAK
        CASE 3:
            CALL DISPLAY()
            BREAK
        CASE 4:
            Print "Exiting"
            EXIT
        DEFAULT:
            Print "Enter a valid choice"
    END SWITCH
ENDWHILE

END PROGRAM
```

3. Write a menu driven program to implement the following operations on circular Queue: a. Enqueue() b. Dequeue() c. Display()

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

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

void enqueue() {
    int num;
    printf("Enter the data to enqueue: ");
    scanf("%d", &num);

    if ((rear + 1) % MAX == front) {
        printf("Queue is full. Cannot enqueue %d.\n", num);
    } else {
        if (front == -1) {
            front = 0;
        }
        rear = (rear + 1) % MAX;
        circularQueue[rear] = num;
        printf("%d enqueued into the circular queue.\n", num);
    }
}

void dequeue() {
    if (front == -1) {
        printf("Queue is empty. Cannot perform DEQUEUE operation.\n");
    } else {
        int num = circularQueue[front];
        printf("%d dequeued from the circular queue.\n", num);

        if (front == rear) {
            // Queue becomes empty
            front = rear = -1;
        } else {
            front = (front + 1) % MAX;
        }
    }
}

void display() {
    if (front == -1) {
        printf("Queue is empty.\n");
    } else {
        printf("Circular Queue contents: ");
        int i = front;
        while (1) {
            printf("%d ", circularQueue[i]);
            if (i == rear) {
                break;
            }
            i = (i + 1) % MAX;
        }
        printf("\n");
    }
}

int main() {
    int choice;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                enqueue();
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}
```



```
Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the data to enqueue: 34
34 enqueued into the circular queue.

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Circular Queue contents: 34

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
34 dequeued from the circular queue.

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 4
Exiting program.


=== Code Execution Successful ===
```

PSEUDOCODE:

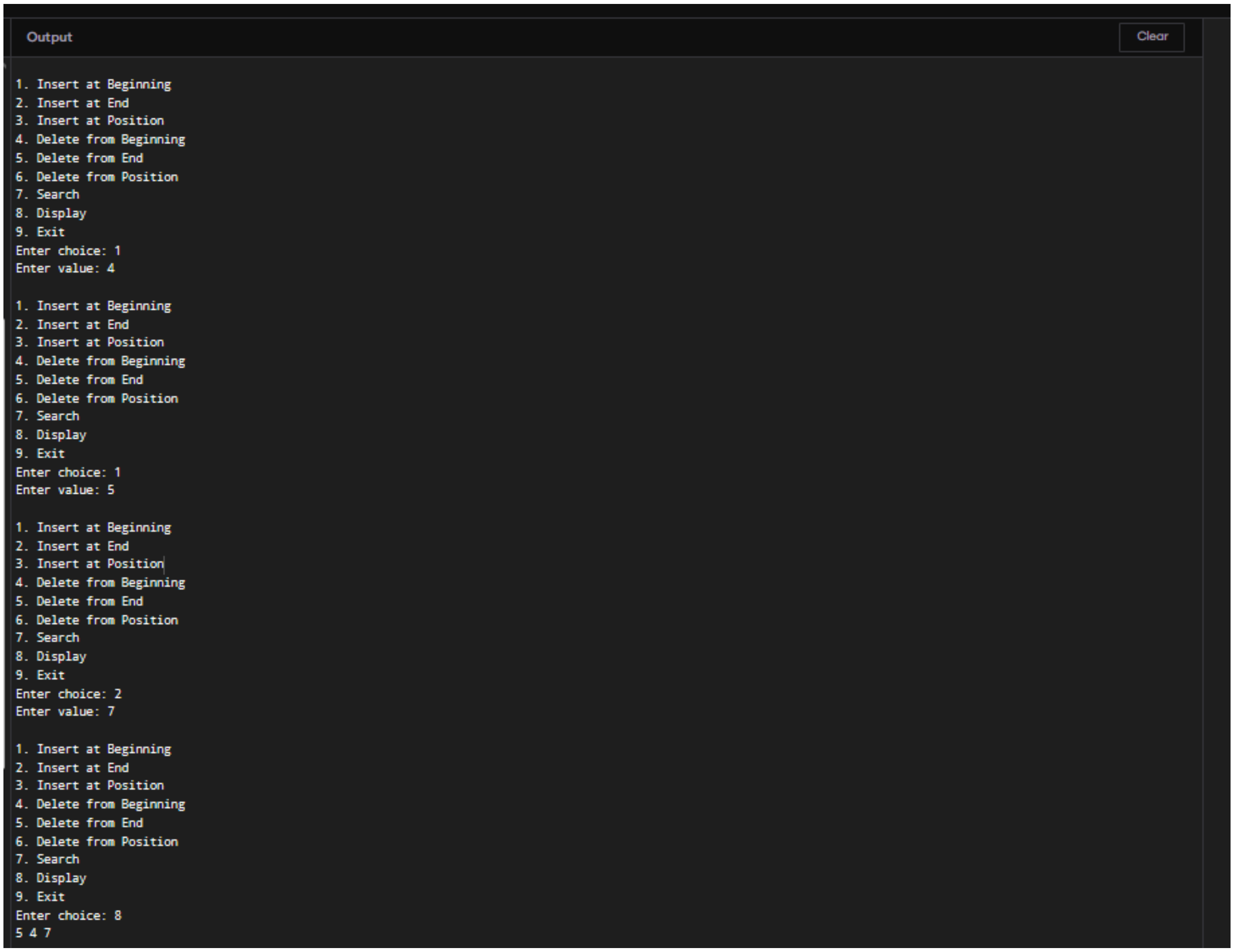
```
START

DEFINE MAX = 5
INITIALIZE circularQueue[MAX], FRONT = -1, REAR = -1

FUNCTION ENQUEUE()
    PRINT "Enter the data to enqueue:"
    INPUT DATA
    IF (REAR + 1) MOD MAX == FRONT THEN
        PRINT "Queue is full. Cannot enqueue DATA."
    ELSE
        IF FRONT == -1 THEN
            SET FRONT = 0
        ENDIF
        SET REAR = (REAR + 1) MOD MAX
        SET circularQueue[REAR] = DATA
        PRINT "DATA enqueued into the circular queue."
    ENDIF
END FUNCTION

FUNCTION DEQUEUE()
    IF FRONT == -1 THEN
        PRINT "Queue is empty. Cannot perform DEQUEUE operation."
    ELSE
        SET DATA = circularQueue[FRONT]
        PRINT "DATA dequeued from the circular queue."
        IF FRONT == REAR THEN
            SET FRONT = -1
            SET REAR = -1
        ELSE
            SET FRONT = (FRONT + 1) MOD MAX
        ENDIF
    ENDIF
END FUNCTION

FUNCTION DISPLAY()
    IF FRONT == -1 THEN
        PRINT "Queue is empty."
    ELSE
        PRINT "Circular Queue contents:"
        SET INDEX = FRONT
        WHILE TRUE DO
            PRINT circularQueue[INDEX]
            IF INDEX == REAR THEN
                BREAK
            ENDIF
            SET INDEX = (INDEX + 1) MOD MAX
        ENDWHILE
    ENDIF
END FUNCTION

MAIN PROGRAM
WHILE TRUE DO
    PRINT "Menu:"
    PRINT "1. Enqueue"
    PRINT "2. Dequeue"
    PRINT "3. Display"
    PRINT "4. Exit"
    PRINT "Enter your choice:"
    INPUT CHOICE

    SWITCH CHOICE
        CASE 1:
            CALL ENQUEUE()
        CASE 2:
            CALL DEQUEUE()
        CASE 3:
            CALL DISPLAY()
        CASE 4:
            PRINT "Exiting program."
            EXIT
        DEFAULT:
            PRINT "Invalid choice. Please try again."
    END SWITCH
ENDWHILE

END PROGRAM
```

4. Write a menu driven program to implement the following operations on singly linked list: a. Insertion() i. Beginning ii. End iii. At a given position b. Deletion() i. Beginning ii. End iii. At a given position c. Search(): search for the given element on the list

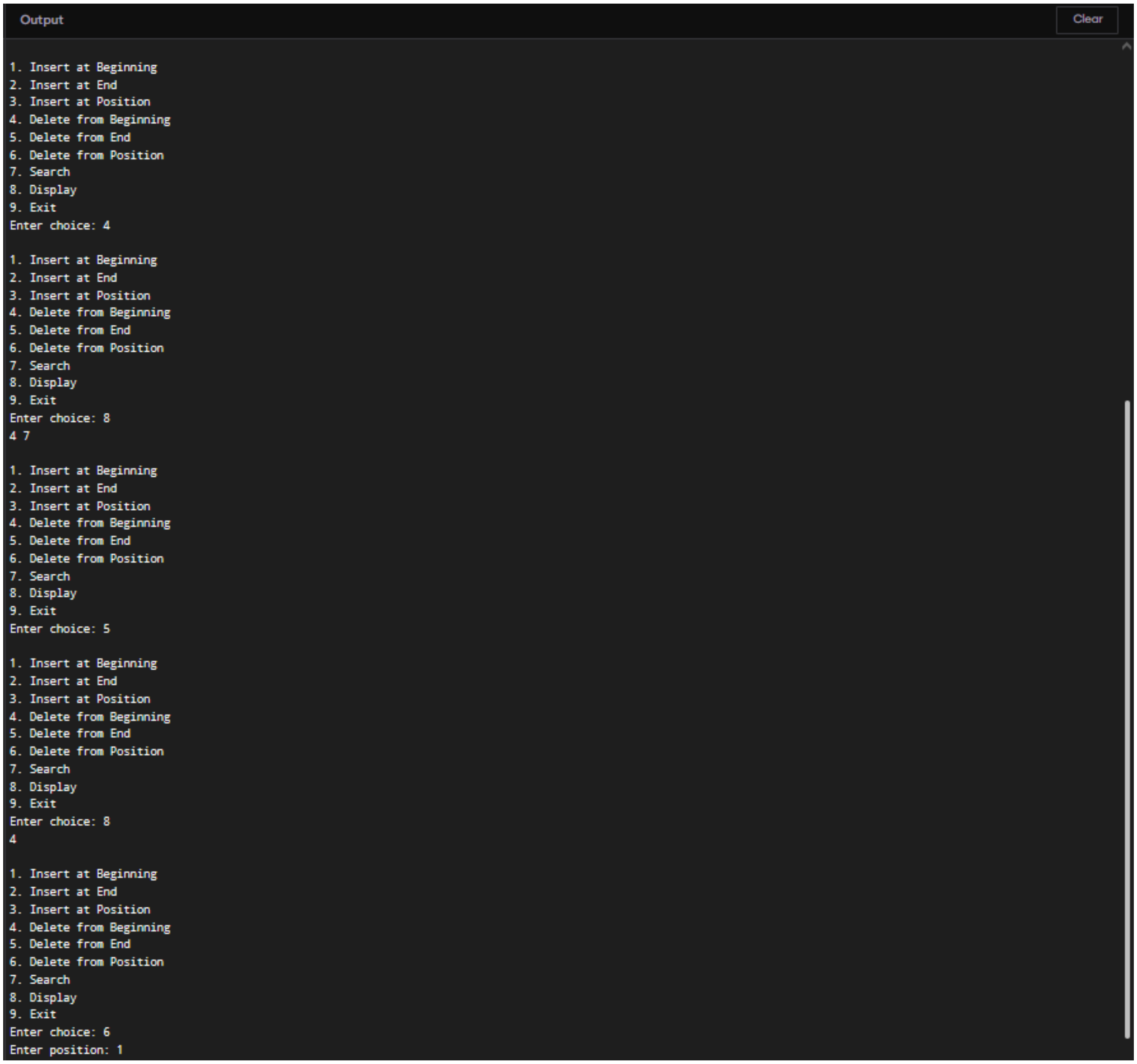
Ans:

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

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

struct node *head = NULL;

void insertAtBeginning() {
    int value;
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    if (newNode == NULL) return;

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

    newNode->data = value;
    newNode->next = head;
    head = newNode;
}

void insertAtEnd() {
    int value;
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    if (newNode == NULL) return;

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

    newNode->data = value;
    newNode->next = NULL;

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

void insertAtPosition() {
    int value, position;
    printf("Enter position: ");
    scanf("%d", &position);

    if (position < 1) return;

    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    if (newNode == NULL) return;

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

    newNode->data = value;

    if (position == 1) {
        newNode->next = head;
        head = newNode;
    } else {
        struct node *temp = head;
        for (int i = 1; i < position - 1 && temp != NULL; i++) temp = temp->next;
        if (temp == NULL) return;

        newNode->next = temp->next;
        temp->next = newNode;
    }
}

void deleteFromBeginning() {
    if (head == NULL) return;

    struct node *temp = head;
    head = head->next;
    free(temp);
}

void deleteFromEnd() {
    if (head == NULL) return;

    if (head->next == NULL) {
        free(head);
        head = NULL;
        return;
    }
    struct node *temp = head;
    while (temp->next->next != NULL) temp = temp->next;
    free(temp->next);
    temp->next = NULL;
}

void deleteFromPosition() {
    int position;
    printf("Enter position: ");
    scanf("%d", &position);

    if (position < 1 || head == NULL) return;

    if (position == 1) {
        struct node *temp = head;
        head = head->next;
        free(temp);
        return;
    }

    struct node *temp = head, *prev = NULL;
    for (int i = 1; i < position && temp != NULL; i++) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) return;

    prev->next = temp->next;
    free(temp);
}

void search() {
    int value;
    printf("Enter value: ");
    scanf("%d", &value);

    struct node *temp = head;
    int position = 1;
    while (temp != NULL) {
        if (temp->data == value) {
            printf("%d found at position %d.\n", value, position);
            return;
        }
        temp = temp->next;
        position++;
    }
    printf("%d not found.\n", value);
```

```
}

void display() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

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

int main() {
    int choice;
    while (1) {
        printf("\n1. Insert at Beginning\n2. Insert at End\n3. Insert at Position\n4. Delete from Beginning\n5. Delete from End\n6. Delete from Position\n7. Search\n8. Display\n9. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: insertAtBeginning(); break;
            case 2: insertAtEnd(); break;
            case 3: insertAtPosition(); break;
            case 4: deleteFromBeginning(); break;
            case 5: deleteFromEnd(); break;
            case 6: deleteFromPosition(); break;
            case 7: search(); break;
            case 8: display(); break;
            case 9: return 0;
        }
    }
}
```



```
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter choice: 1
Enter value: 4

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter choice: 1
Enter value: 5

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter choice: 2
Enter value: 7

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter choice: 8
5 4 7
```



```
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter choice: 4

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter choice: 8
4 7

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter choice: 5

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter choice: 8
4

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter choice: 6
Enter position: 1
```

PSEUDOCODE:

```
1. Initialize head to NULL (indicating an empty linked list).

2. Function insertAtBeginning:
   - Allocate memory for newNode.
     - If memory allocation fails, print "Memory allocation failed." and return.
   - Prompt user to enter the value to insert.
   - Set newNode's data to the entered value.
   - Set newNode's next pointer to point to the current head.
   - Update head to point to newNode, making it the new first node of the list.
   - Print a message confirming the value inserted at the beginning.

3. Function insertAtEnd:
   - Allocate memory for newNode.
     - If memory allocation fails, print "Memory allocation failed." and return.
   - Prompt user to enter the value to insert.
   - Set newNode's data to the entered value.
   - Set newNode's next pointer to NULL (as it will be the last node).
   - If head is NULL (list is empty), set head to newNode, making it the first node.
   - Otherwise, traverse the list to find the last node (the node with next == NULL).
     - Traverse until temp->next is NULL, then set the last node's next pointer to newNode.
   - Print a message confirming the value inserted at the end.

4. Function insertAtPosition:
   - Prompt user to enter the position at which to insert the new node.
   - If the position is less than 1, print "Invalid position" and return (positions start from 1).
   - Allocate memory for newNode.
     - If memory allocation fails, print "Memory allocation failed." and return.
   - Prompt user to enter the value to insert.
   - Set newNode's data to the entered value.
   - If position is 1, insert the node at the beginning:
     - Set newNode's next pointer to the current head.
     - Set head to newNode, making it the new first node.
   - Otherwise, traverse the list until reaching the (position - 1)-th node.
     - If the position is out of range (i.e., reaching the end of the list before the desired position), print
"Position out of range" and return.
     - Set newNode's next pointer to the (position-th node).
     - Set the (position - 1)-th node's next pointer to newNode, inserting it at the desired position.
   - Print a message confirming the value inserted at the specified position.

5. Function deleteFromBeginning:
   - If head is NULL (list is empty), print "List is empty" and return.
   - Otherwise:
     - Save head in a temporary variable (temp).
     - Set head to the next node in the list (head = head->next).
     - Free the memory occupied by temp.
     - Print a message confirming the value deleted from the beginning.
```

```
6. Function deleteFromEnd:
  - If head is NULL (list is empty), print "List is empty" and return.
  - If head's next pointer is NULL (only one element in the list):
    - Free the memory occupied by head.
    - Set head to NULL (indicating an empty list).
    - Print a message confirming the value deleted from the end.
  - Otherwise:
    - Traverse the list to find the second-to-last node (the node before the last node).
    - Once found, free the memory of the last node and set the second-to-last node's next pointer to NULL.
    - Print a message confirming the value deleted from the end.

7. Function deleteFromPosition:
  - Prompt user to enter the position at which to delete a node.
  - If the position is less than 1 or the list is empty (head is NULL), print "Invalid position or list is empty" and return.
  - If position is 1, call deleteFromBeginning to remove the first node.
  - Otherwise, traverse the list until reaching the (position - 1)-th node:
    - If the position is out of range (i.e., reaching the end of the list before the desired position), print "Position out of range" and return.
    - Save the (position-th node) in a temporary variable (temp).
    - Set the (position - 1)-th node's next pointer to point to temp's next node, effectively removing temp from the list.
    - Free the memory occupied by temp.
    - Print a message confirming the value deleted from the specified position.

8. Function search:
  - Prompt user to enter the value to search for in the list.
  - Initialize a variable to track the current position (start from 1).
  - Traverse the list node by node:
    - If a node's data matches the entered value, print a message with the value and its position, then return.
    - Otherwise, move to the next node and increment the position.
  - If the value is not found by the end of the list, print "Value not found in the list."

9. Function display:
  - If head is NULL (list is empty), print "List is empty" and return.
  - Otherwise, traverse the list and print the data of each node, separated by spaces.
  - Print a newline after displaying all values.

10. Main loop:
   - Display a menu with options:
     - 1. Insert at Beginning
     - 2. Insert at End
     - 3. Insert at Position
     - 4. Delete from Beginning
     - 5. Delete from End
     - 6. Delete from Position
```

  - 7. Search
  - 8. Display
  - 9. Exit
- Prompt user to enter their choice.
- Depending on the choice, call the corresponding function:
  - Case 1: Call insertAtBeginning.
  - Case 2: Call insertAtEnd.
  - Case 3: Call insertAtPosition.
  - Case 4: Call deleteFromBeginning.
  - Case 5: Call deleteFromEnd.
  - Case 6: Call deleteFromPosition.
  - Case 7: Call search.
  - Case 8: Call display.
  - Case 9: Exit the program.
- Repeat the process until the user chooses to exit.

5. Write a menu driven program to implement the following operations on Doubly linked list: a. Insertion() i. Beginning ii. End iii. At a given position b. Deletion() i. Beginning ii. End iii. At a given position c. Search(): search for the given element on the list

ANS:

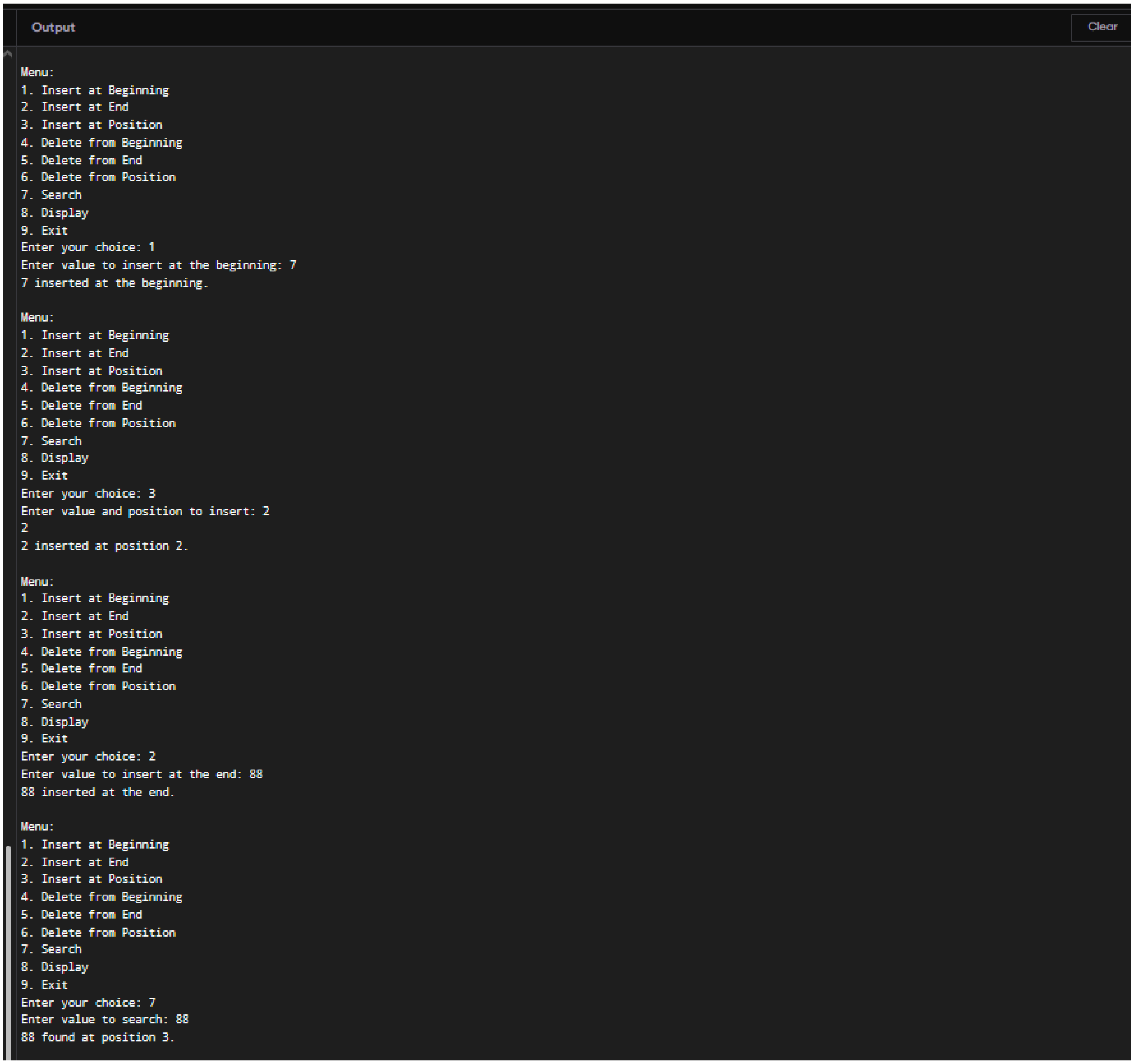
```
#include <stdio.h>
#include <stdlib.h>

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

struct Node *head = NULL;

void insertAtBeginning(int value) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = head;
    if (head != NULL) {
    head->prev = newNode;
    }
    head = newNode;
    printf("%d inserted at the beginning.\n", value);
}

void insertAtEnd(int value) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL) {
    newNode->prev = NULL;
    head = newNode;
    } else {
    struct Node *temp = head;
    while (temp->next != NULL) {
    temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
    }
    printf("%d inserted at the end.\n", value);
}

void insertAtPosition(int value, int position) {
    if (position == 1) {
    insertAtBeginning(value);
    return;
    }

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

    newNode->data = value;

    struct Node *temp = head;

    for (int i = 1; i < position - 1 && temp != NULL; i++) {

    temp = temp->next;

    }

    if (temp == NULL) {

    printf("Invalid position.\n");

    free(newNode);

    return;

    }

    newNode->next = temp->next;

    newNode->prev = temp;

    if (temp->next != NULL) {

    temp->next->prev = newNode;

    }

    temp->next = newNode;

    printf("%d inserted at position %d.\n", value, position);

}


void deleteFromBeginning() {

    if (head == NULL) {

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

    return;

    }
    struct Node *temp = head;
    head = head->next;
    if (head != NULL) {
    head->prev = NULL;
    }
    printf("%d deleted from the beginning.\n", temp->data);
    free(temp);
}

void deleteFromEnd() {
    if (head == NULL) {
    printf("List is empty.\n");
    return;
    }
    struct Node *temp = head;
    if (head->next == NULL) {
    head = NULL;
    } else {
    while (temp->next != NULL) {
    temp = temp->next;
    }
    temp->prev->next = NULL;
    }
    printf("%d deleted from the end.\n", temp->data);
    free(temp);
}
```

```
void deleteFromPosition(int position) {
        if (head == NULL) {
        printf("List is empty.\n");
        return;
        }
        if (position == 1) {
        deleteFromBeginning();
        return;
        }
        struct Node *temp = head;
        for (int i = 1; i < position && temp != NULL; i++) {
        temp = temp->next;
        }
        if (temp == NULL) {
        printf("Invalid position.\n");
        return;
        }
        if (temp->next != NULL) {
        temp->next->prev = temp->prev;
        }
        if (temp->prev != NULL) {
        temp->prev->next = temp->next;
        }
        printf("%d deleted from position %d.\n", temp->data, position);
        free(temp);
```

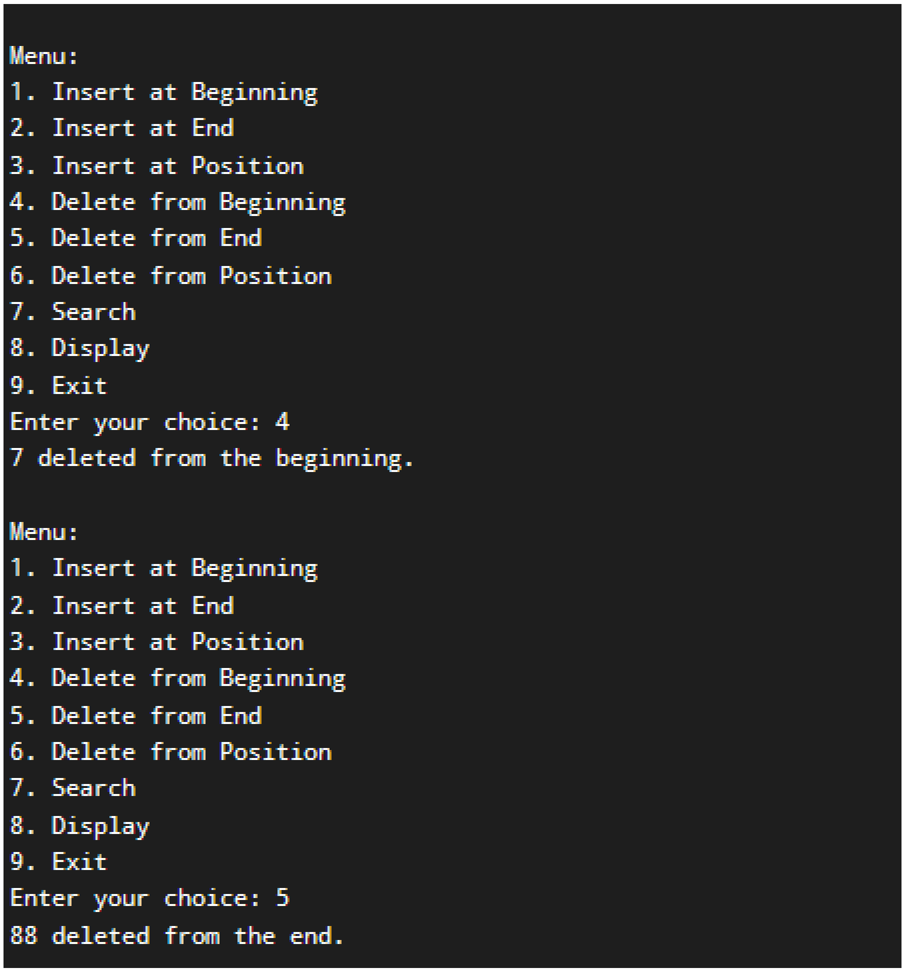
```
}

void search(int value) {
    struct Node *temp = head;
    int position = 1;
    while (temp != NULL) {
    if (temp->data == value) {
    printf("%d found at position %d.\n", value, position);
    return;
    }
    temp = temp->next;
    position++;
    }
    printf("%d not found in the list.\n", value);
}

void display() {
    if (head == NULL) {
    printf("List is empty.\n");
    return;
    }
    struct Node *temp = head;
    printf("List contents: ");
    while (temp != NULL) {
    printf("%d ", temp->data);
    temp = temp->next;
    }
    printf("\n");
}


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


    while (1) {
    printf("\nMenu:\n");
    printf("1. Insert at Beginning\n");
    printf("2. Insert at End\n");
    printf("3. Insert at Position\n");
    printf("4. Delete from Beginning\n");
    printf("5. Delete from End\n");
    printf("6. Delete from Position\n");
    printf("7. Search\n");
    printf("8. Display\n");
    printf("9. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);


    switch (choice) {
    case 1:
    printf("Enter value to insert at the beginning: ");
    scanf("%d", &value);
    insertAtBeginning(value);
```

```
break;
case 2:
printf("Enter value to insert at the end: ");
scanf("%d", &value);
insertAtEnd(value);
break;
case 3:
printf("Enter value and position to insert: ");
scanf("%d %d", &value, &position);
insertAtPosition(value, position);
break;
case 4:
deleteFromBeginning();
break;
case 5:
deleteFromEnd();
break;
case 6:
printf("Enter position to delete: ");
scanf("%d", &position);
deleteFromPosition(position);
break;
case 7:
printf("Enter value to search: ");
scanf("%d", &value);
search(value);
```

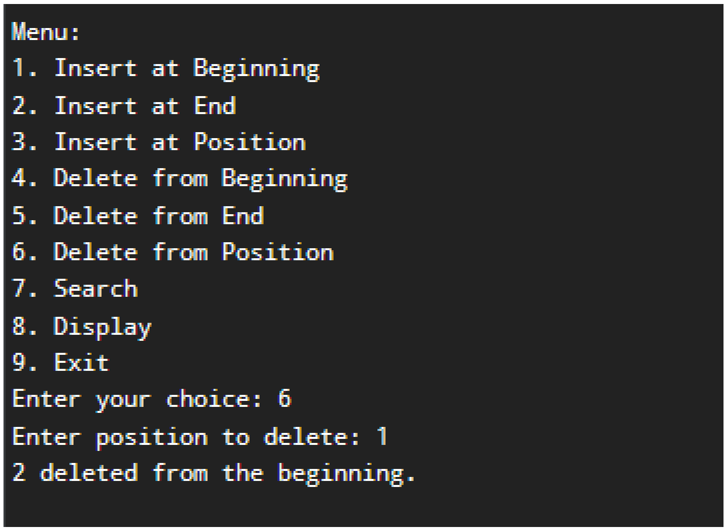
```
        break;
        case 8:
        display();
        break;
        case 9:
          printf("Exiting...\n");
        return 0;
        default:
        printf("Invalid choice. Try again.\n");
        }
        }

        return 0;
}
```



```
Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 1
Enter value to insert at the beginning: 7
7 inserted at the beginning.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 3
Enter value and position to insert: 2
2
2 inserted at position 2.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 2
Enter value to insert at the end: 88
88 inserted at the end.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 7
Enter value to search: 88
88 found at position 3.
```



```

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 4
7 deleted from the beginning.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 5
88 deleted from the end.
```



```
Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 6
Enter position to delete: 1
2 deleted from the beginning.
```

PSEUDOCODE:

```
Node:
    data
    prev
    next

Function insertAtBeginning(value):
    Create newNode
    newNode.data = value
    newNode.prev = NULL
    newNode.next = head
    If head is not NULL:
        head.prev = newNode
    head = newNode
    Print "value inserted at the beginning"

Function insertAtEnd(value):
    Create newNode
    newNode.data = value
    newNode.next = NULL
    If head is NULL:
        newNode.prev = NULL
        head = newNode
    Else:
        Set temp = head
        Traverse to last node (temp.next == NULL)
        temp.next = newNode
        newNode.prev = temp
    Print "value inserted at the end"

Function insertAtPosition(value, position):
    If position is 1:
        Call insertAtBeginning(value)
        Return
    Create newNode
    newNode.data = value
    Set temp = head
    Traverse to node at position - 1
    If temp is NULL:
        Print "Invalid position"
        Return
    newNode.next = temp.next
    newNode.prev = temp
    If temp.next is not NULL:
        temp.next.prev = newNode
    temp.next = newNode
    Print "value inserted at position"
```

```
Function deleteFromBeginning():
    If head is NULL:
        Print "List is empty"
        Return
    Set temp = head
    head = head.next
    If head is not NULL:
        head.prev = NULL
    Free temp
    Print "value deleted from the beginning"

Function deleteFromEnd():
    If head is NULL:
        Print "List is empty"
        Return
    Set temp = head
    If head.next is NULL:
        head = NULL
    Else:
        Traverse to last node (temp.next == NULL)
        temp.prev.next = NULL
    Free temp
    Print "value deleted from the end"

Function deleteFromPosition(position):
    If head is NULL:
        Print "List is empty"
        Return
    If position is 1:
        Call deleteFromBeginning()
        Return
    Set temp = head
    Traverse to node at position
    If temp is NULL:
        Print "Invalid position"
        Return
    If temp.next is not NULL:
        temp.next.prev = temp.prev
    If temp.prev is not NULL:
        temp.prev.next = temp.next
    Free temp
    Print "value deleted from position"

Function search(value):
    Set temp = head
    Set position = 1
    While temp is not NULL:
        If temp.data == value:
            Print "value found at position"
            Return
        temp = temp.next
        position = position + 1
    Print "value not found"

Function display():
    If head is NULL:
        Print "List is empty"
        Return
    Set temp = head
    Print "List contents: "
    While temp is not NULL:
        Print temp.data
        temp = temp.next
    Print a new line

Function main():
    While True:
        Print menu options
        Get user input for choice
        If choice is 1:
            Get value
            Call insertAtBeginning(value)
        Else if choice is 2:
            Get value
            Call insertAtEnd(value)
        Else if choice is 3:
            Get value and position
            Call insertAtPosition(value, position)
        Else if choice is 4:
            Call deleteFromBeginning()
        Else if choice is 5:
            Call deleteFromEnd()
        Else if choice is 6:
            Get position
            Call deleteFromPosition(position)
        Else if choice is 7:
            Get value
            Call search(value)
        Else if choice is 8:
            Call display()
        Else if choice is 9:
            Print "Exiting program"
            Exit the loop
        Else:
            Print "Invalid choice"
```