1. Write a menu driven program to implement the
a.PUSH()
b.Pop()c.Display()

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

#define MAX 5

struct Stack {
    int arr[MAX];
    int top;
};

void initStack(struct Stack* stack) {
    stack->top = -1;
}

int isFull(struct Stack* stack) {
    return stack->top == MAX - 1;
}

int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

void PUSH(struct Stack* stack, int value) {
    if (isFull(stack)) {
        printf("Stack Overflow! Cannot push %d\n", value);
    } else {
        stack->arr[++stack->top] = value;
        printf("%d pushed to stack\n", value);
    }
}

void POP(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow! Cannot pop\n");
    } else {
        int poppedValue = stack->arr[stack->top--];
        printf("%d popped from stack\n", poppedValue);
    }
}

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

int main() {
    struct Stack stack;
    initStack(&stack);

    int choice, value;

    while (1) {
        printf("\nStack Menu:\n");
        printf("1. PUSH\n");
        printf("2. POP\n");
        printf("3. DISPLAY\n");
        printf("4. EXIT\n");

        printf("Enter your choice (1/2/3/4): ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                PUSH(&stack, value);
                break;
            case 2:
                POP(&stack);
                break;
            case 3:
                display(&stack);
                break;
            case 4:
                printf("Exiting the program.\n");
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

OUTPUT:

```
Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to push: 20
20 pushed to stack

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to push: 10
10 pushed to stack

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 2
10 popped from stack

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 2
20 popped from stack

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 2
Stack Underflow! Cannot pop

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to push: 30
30 pushed to stack

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to push: 40 50 60 70
40 pushed to stack

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): Invalid choice! Please try again.

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): Invalid choice! Please try again.

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): Invalid choice! Please try again.

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to push: 40 50 60 70
40 pushed to stack

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): Invalid choice! Please try again.

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): Invalid choice! Please try again.

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): Invalid choice! Please try again.

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to push: 40
40 pushed to stack

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to push: 50
50 pushed to stack

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to push: 60
Stack Overflow! Cannot push 60

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to push: 70
Stack Overflow! Cannot push 70

Stack Menu:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4):
```

PSEUDOCODE:

```
START
    DEFINE MAX_SIZE as 5
    DEFINE stack[MAX_SIZE]
    DEFINE top as -1

    FUNCTION initStack:
        SET top to -1

    FUNCTION isFull:
        RETURN (top == MAX_SIZE - 1)

    FUNCTION isEmpty:
        RETURN (top == -1)

    FUNCTION PUSH(value):
        IF isFull() IS TRUE:
            PRINT "Stack Overflow! Cannot push " + value
        ELSE:
            INCREMENT top by 1
            SET stack[top] to value
            PRINT value + " pushed to stack"

    FUNCTION POP:
        IF isEmpty() IS TRUE:
            PRINT "Stack Underflow! Cannot pop"
        ELSE:
            SET poppedValue to stack[top]
            DECREMENT top by 1
            PRINT poppedValue + " popped from stack"

    FUNCTION DISPLAY:
        IF isEmpty() IS TRUE:
            PRINT "Stack is empty"
        ELSE:
            PRINT "Stack elements:"
            FOR i FROM top DOWN TO 0:
                PRINT stack[i]

    FUNCTION MENU:
        REPEAT
            PRINT "Stack Menu:"
            PRINT "1. PUSH"
            PRINT "2. POP"
            PRINT "3. DISPLAY"
            PRINT "4. EXIT"
            PRINT "Enter your choice (1/2/3/4):"

            READ choice

            IF choice IS 1:
                PRINT "Enter value to push:"
                READ value
                CALL PUSH(value)
            ELSE IF choice IS 2:
                CALL POP()
            ELSE IF choice IS 3:
                CALL DISPLAY()
            ELSE IF choice IS 4:
                PRINT "EXITING THE PROGRAM"
                EXIT
            ELSE:
                PRINT "Invalid Choice! Please try again"
        UNTIL choice IS 4

    MAIN:
        CALL initStack()
        CALL MENU()

END
```

2.Write a menu driven program to implement the following operations on Queue:

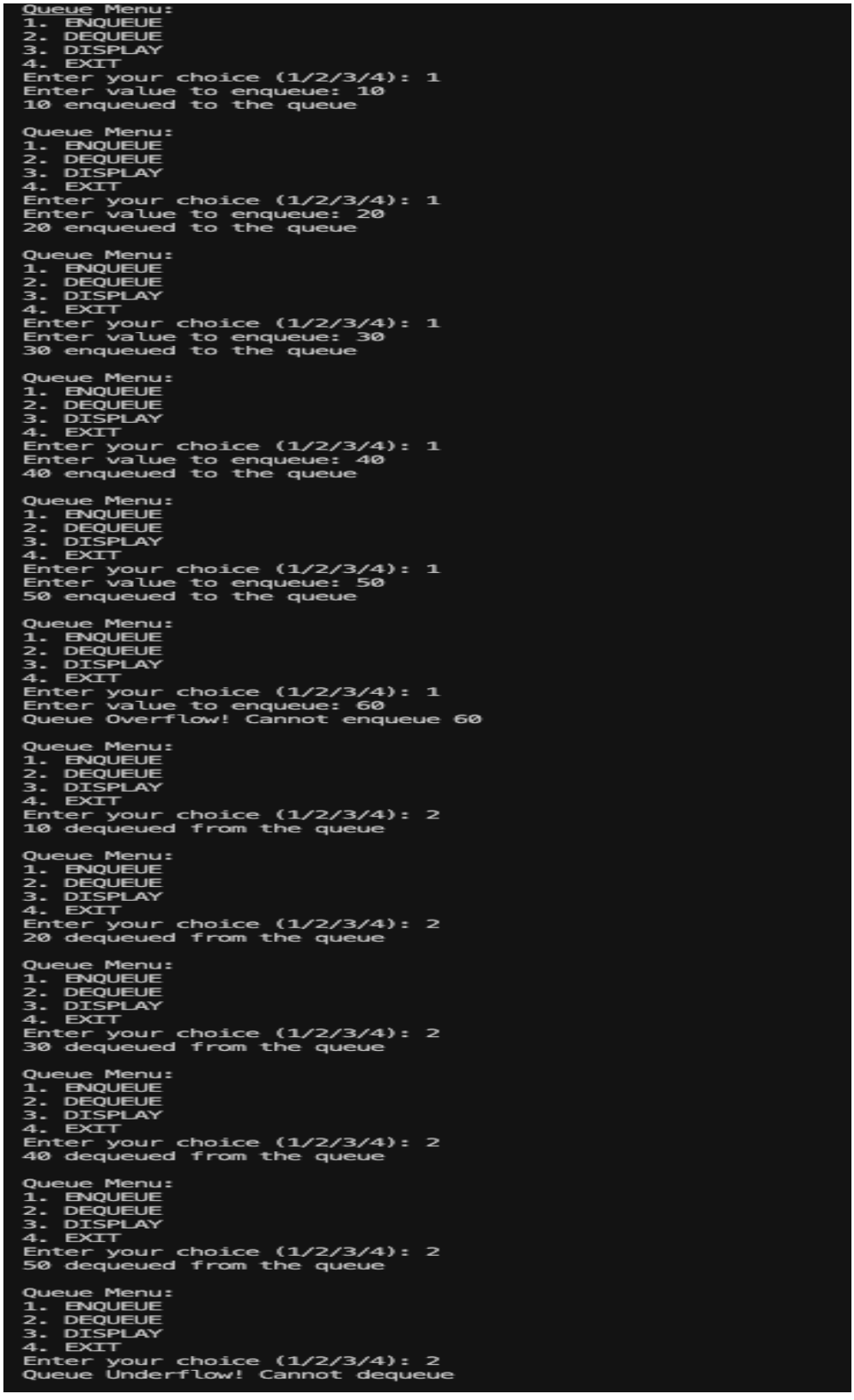
a.Enqueue( )

b.Dequeue( )

c.Display( )

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

#define MAX 5

struct Queue {
    int arr[MAX];
    int front;
    int rear;
};

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

int isFull(struct Queue* queue) {
    return (queue->rear == MAX - 1);
}

int isEmpty(struct Queue* queue) {
    return (queue->front == -1 || queue->front > queue->rear);
}

void ENQUEUE(struct Queue* queue, int value) {
    if (isFull(queue)) {
        printf("Queue Overflow! Cannot enqueue %d\n", value);
    } else {
        if (queue->front == -1)
            queue->front = 0;
        queue->rear++;
        queue->arr[queue->rear] = value;
        printf("%d enqueued to the queue\n", value);
    }
}

void DEQUEUE(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue Underflow! Cannot dequeue\n");
    } else {
        int dequeuedValue = queue->arr[queue->front];
        printf("%d dequeued from the queue\n", dequeuedValue);
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
    }
}

void DISPLAY(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
    } else {
        printf("Queue elements: ");
        for (int i = queue->front; i <= queue->rear; i++) {
            printf("%d ", queue->arr[i]);
        }
        printf("\n");
    }
}

void MENU() {
    struct Queue queue;
    initQueue(&queue);
    int choice, value;

    while (1) {
        printf("\nQueue Menu:\n");
        printf("1. ENQUEUE\n");
        printf("2. DEQUEUE\n");
        printf("3. DISPLAY\n");
        printf("4. EXIT\n");

        printf("Enter your choice (1/2/3/4): ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                ENQUEUE(&queue, value);
                break;
            case 2:
                DEQUEUE(&queue);
                break;
            case 3:
                DISPLAY(&queue);
                break;
            case 4:
                printf("Exiting the program.\n");
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
}

int main() {
    MENU();
    return 0;
}
```

OUTPUT:


Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to enqueue: 10
10 enqueued to the queue
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to enqueue: 20
20 enqueued to the queue
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to enqueue: 30
30 enqueued to the queue
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to enqueue: 40
40 enqueued to the queue
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to enqueue: 50
50 enqueued to the queue
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 1
Enter value to enqueue: 60
Queue Overflow! Cannot enqueue 60
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 2
10 dequeued from the queue
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 2
20 dequeued from the queue
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 2
30 dequeued from the queue
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 2
40 dequeued from the queue
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 2
50 dequeued from the queue
Queue Menu:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice (1/2/3/4): 2
Queue Underflow! Cannot dequeue

PSEUDOCODE:

```
START

  DEFINE MAX as 5
  DEFINE queue[MAX]
  DEFINE front as -1
  DEFINE rear as -1

  FUNCTION initQueue:
    SET front to -1
    SET rear to -1

  FUNCTION isFull:
    IF rear == MAX - 1 THEN
      RETURN TRUE
    ELSE
      RETURN FALSE

  FUNCTION isEmpty:
    IF front == -1 OR front > rear THEN
      RETURN TRUE
    ELSE
      RETURN FALSE

  FUNCTION ENQUEUE(value):
    IF isFull() IS TRUE THEN
      PRINT "Queue Overflow! Cannot enqueue " + value
    ELSE
      IF front == -1 THEN
        SET front to 0
      END IF
      INCREMENT rear by 1
      SET queue[rear] to value
      PRINT value + " enqueued to the queue"

  FUNCTION DEQUEUE:
    IF isEmpty() IS TRUE THEN
      PRINT "Queue Underflow! Cannot dequeue"
    ELSE
      SET dequeuedValue to queue[front]
      PRINT dequeuedValue + " dequeued from the queue"
      INCREMENT front by 1
      IF front > rear THEN
        SET front to -1
        SET rear to -1
      END IF

  FUNCTION DISPLAY:
    IF isEmpty() IS TRUE THEN
      PRINT "Queue is empty"
    ELSE
      PRINT "Queue elements: "
      FOR i FROM front TO rear DO
        PRINT queue[i]
      END FOR

  FUNCTION MENU:
    REPEAT
      PRINT "Queue Menu:"
      PRINT "1. ENQUEUE"
      PRINT "2. DEQUEUE"
      PRINT "3. DISPLAY"
      PRINT "4. EXIT"
      PRINT "Enter your choice (1/2/3/4):"

      READ choice

      IF choice == 1 THEN
        PRINT "Enter value to enqueue:"
        READ value
        CALL ENQUEUE(value)
      ELSE IF choice == 2 THEN
        CALL DEQUEUE()
      ELSE IF choice == 3 THEN
        CALL DISPLAY()
      ELSE IF choice == 4 THEN
        PRINT "Exiting the program."
        EXIT
      ELSE
        PRINT "Invalid choice! Please try again."
    UNTIL choice == 4

  MAIN:
    CALL initQueue()
    CALL MENU()

END
```

3.Write a menu driven program to implement the following operations on circular Queue:

a. Enqueue()
b. Dequeue()
c. Display()

OUTPUT:

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

#define MAX 5

struct CircularQueue {
    int front, rear;
    int queue[MAX];
};

int isFull(struct CircularQueue *q) {
    return ((q->rear + 1) % MAX == q->front);
}

int isEmpty(struct CircularQueue *q) {
    return (q->front == -1);
}

void enqueue(struct CircularQueue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full, cannot enqueue!\n");
    } else {
        if (q->front == -1) {
            q->front = 0;
        }
        q->rear = (q->rear + 1) % MAX;
        q->queue[q->rear] = value;
        printf("Enqueued: %d\n", value);
    }
}

void dequeue(struct CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty, cannot dequeue!\n");
    } else {
        int dequeuedValue = q->queue[q->front];
        if (q->front == q->rear) {
            q->front = q->rear = -1;
        } else {
            q->front = (q->front + 1) % MAX;
        }
        printf("Dequeued: %d\n", dequeuedValue);
    }
}

void display(struct CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
    } else {
        printf("Queue elements: ");
        int i = q->front;
        while (1) {
            printf("%d ", q->queue[i]);
            if (i == q->rear) {
                break;
            }
            i = (i + 1) % MAX;
        }
        printf("\n");
    }
}

int main() {
    struct CircularQueue q;
    q.front = q.rear = -1;
    int choice, value;

    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:
                printf("Enter the value to enqueue: ");
                scanf("%d", &value);
                enqueue(&q, value);
                break;
            case 2:
                dequeue(&q);
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

```
Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
Queue is empty, cannot dequeue!

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice:
1
Enter the value to enqueue: 10
Enqueued: 10

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 20
Enqueued: 20

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 30
Enqueued: 30

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 40
Enqueued: 40

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 50
Enqueued: 50

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 60
Queue is full, cannot enqueue!
```

PSEUDOCODE:

```
BEGIN
    MAX = 5
    Declare queue[MAX]
    Declare front = -1
    Declare rear = -1

    FUNCTION isFull()
        IF (rear + 1) % MAX == front
            RETURN TRUE
        ELSE
            RETURN FALSE
        ENDIF
    END FUNCTION

    FUNCTION isEmpty()
        IF front == -1
            RETURN TRUE
        ELSE
            RETURN FALSE
        ENDIF
    END FUNCTION

    FUNCTION enqueue(value)
        IF isFull() == TRUE
            PRINT "Queue is full, cannot enqueue!"
        ELSE
            IF front == -1
                front = 0
            ENDIF
            rear = (rear + 1) % MAX
            queue[rear] = value
            PRINT "Enqueued: ", value
        END IF
    END FUNCTION

    FUNCTION dequeue()
        IF isEmpty() == TRUE
            PRINT "Queue is empty, cannot dequeue!"
        ELSE
            value = queue[front]
            PRINT "Dequeued: ", value
            IF front == rear
                front = -1
                rear = -1
            ELSE
                front = (front + 1) % MAX
            END IF
        END IF
    END FUNCTION


    FUNCTION display()
        IF isEmpty() == TRUE
            PRINT "Queue is empty!"
        ELSE
            PRINT "Queue elements: "
            i = front
            WHILE TRUE
                PRINT queue[i]
                IF i == rear
                    BREAK
                END IF
                i = (i + 1) % MAX
            END WHILE
        END IF
    END FUNCTION

    WHILE TRUE
        PRINT "Menu:"
        PRINT "1. Enqueue"
        PRINT "2. Dequeue"
        PRINT "3. Display"
        PRINT "4. Exit"
        PRINT "Enter your choice: "
        READ choice

        SWITCH choice
            CASE 1:
                PRINT "Enter value to enqueue: "
                READ value
                CALL enqueue(value)
                BREAK

            CASE 2:
                CALL dequeue()
                BREAK

            CASE 3:
                CALL display()
                BREAK

            CASE 4:
                PRINT "Exiting the program..."
                RETURN

            DEFAULT:
                PRINT "Invalid choice! Please try again."
        END SWITCH

    END WHILE
END
```

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

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

struct Node {
    int data;
    struct Node* next;
};
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
void insertAtBeginning(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = *head;
    *head = newNode;
    printf("Inserted %d at the beginning.\n", value);
}
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    printf("Inserted %d at the end.\n", value);
}
void insertAtPosition(struct Node** head, int value, int position) {
    struct Node* newNode = createNode(value);
    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
        printf("Inserted %d at position %d.\n", value, position);
        return;
    }

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

    if (temp == NULL) {
        printf("Position out of bounds.\n");
    } else {
        newNode->next = temp->next;
        temp->next = newNode;
        printf("Inserted %d at position %d.\n", value, position);
    }
}
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }
```

```c
        if (temp->data == value) {
            printf("Element %d found at position %d.\n", value, position);
            return;
        }
        temp = temp->next;
        position++;
    }
    printf("Element %d not found in the list.\n", value);
}
void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    printf("List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    struct Node* head = NULL;
    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 Given Position\n");
        printf("4. Delete from Beginning\n");
        printf("5. Delete from End\n");
        printf("6. Delete from Given Position\n");
        printf("7. Search for an Element\n");
        printf("8. Display the List\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(&head, value);
                break;

            case 2:
                printf("Enter value to insert at the end: ");
                scanf("%d", &value);
                insertAtEnd(&head, value);
                break;

            case 3:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                printf("Enter position to insert at: ");
                scanf("%d", &position);
```

```c
    if (*head == NULL) {
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
    printf("Deleted from the beginning.\n");
}
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        printf("Deleted from the end.\n");
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL && temp->next->next != NULL) {
        temp = temp->next;
    }

    free(temp->next);
    temp->next = NULL;
    printf("Deleted from the end.\n");
}
void deleteFromPosition(struct Node** head, int position) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    if (position == 1) {
        struct Node* temp = *head;
        *head = (*head)->next;
        free(temp);
        printf("Deleted from position %d.\n", position);
        return;
    }

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

    if (temp == NULL || temp->next == NULL) {
        printf("Position out of bounds.\n");
    } else {
        struct Node* nodeToDelete = temp->next;
        temp->next = temp->next->next;
        free(nodeToDelete);
        printf("Deleted from position %d.\n", position);
    }
}
void search(struct Node* head, int value) {
    struct Node* temp = head;
    int position = 1;
    while (temp != NULL) {
```

```
                insertAtPosition(&head, value, position);
                break;

            case 4:
                deleteFromBeginning(&head);
                break;

            case 5:
                deleteFromEnd(&head);
                break;

            case 6:
                printf("Enter position to delete from: ");
                scanf("%d", &position);
                deleteFromPosition(&head, position);
                break;

            case 7:
                printf("Enter element to search for: ");
                scanf("%d", &value);
                search(head, value);
                break;

            case 8:
                display(head);
                break;

            case 9:
                printf("Exiting...\n");
                return 0;

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

    return 0;
}
```

OUTPUT:

```
Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search for an Element
8. Display the List
9. Exit
Enter your choice: 1
Enter value to insert at the beginning: 10
Inserted 10 at the beginning.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search for an Element
8. Display the List
9. Exit
Enter your choice: 2
Enter value to insert at the end: 20
Inserted 20 at the end.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search for an Element
8. Display the List
9. Exit
Enter your choice: 3
Enter value to insert: 15
Enter position to insert at: 2
Inserted 15 at position 2.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search for an Element
8. Display the List
9. Exit
Enter your choice: 3
Enter value to insert: 25
Enter position to insert at: 5
Position out of bounds.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search for an Element
8. Display the List
9. Exit
Enter your choice: 4
Deleted from the beginning.
```

```
Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search for an Element
8. Display the List
9. Exit
Enter your choice: 5
Deleted from the end.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search for an Element
8. Display the List
9. Exit
Enter your choice: 6
Enter position to delete from: 1
Deleted from position 1.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search for an Element
8. Display the List
9. Exit
Enter your choice: 4
List is empty. Cannot delete.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search for an Element
8. Display the List
9. Exit
Enter your choice: 7
Enter element to search for: 15
Element 15 not found in the list.
```

PSEUDOCODE:

```
START
Structure Node:
    Integer data
    Pointer to next node (next)

Function CreateNode(data):
    Allocate memory for a new node
    Set the node's data to the given value
    Set next pointer to NULL
    Return the new node

Function InsertAtBeginning(head, data):
    newNode = CreateNode(data)
    Set newNode's next pointer to head
    Set head to newNode
    Print "Inserted data at the beginning"

Function InsertAtBeginning(head, data):
    newNode = CreateNode(data)
    Set newNode's next pointer to head
    Set head to newNode
    Print "Inserted data at the beginning"

Function InsertAtEnd(head, data):
    newNode = CreateNode(data)

    If head is NULL:
        Set head to newNode
    Else:
        Set temp = head
        While temp's next is not NULL:
            Set temp = temp's next
        Set temp's next to newNode
    Print "Inserted data at the end"

Function InsertAtPosition(head, data, position):
    If position is 1:
        Call InsertAtBeginning(head, data)
        Print "Inserted data at position 1"
        Return

    newNode = CreateNode(data)
    Set temp = head
    currentPosition = 1

    While currentPosition < position - 1 and temp is not NULL:
        Set temp = temp's next
        Increment currentPosition

    If temp is NULL:
```

```
        Print "Position out of bounds"
        Return

    Set newNode's next to temp's next
    Set temp's next to newNode
    Print "Inserted data at position"

Function DeleteFromBeginning(head):
    If head is NULL:
        Print "List is empty. Cannot delete"
        Return

    Set temp = head
    Set head to head's next
    Free temp
    Print "Deleted from the beginning"

Function DeleteFromEnd(head):
    If head is NULL:
        Print "List is empty. Cannot delete"
        Return

    If head's next is NULL:
        Free head
        Set head to NULL
        Print "Deleted from the end"
        Return

    Set temp = head
    While temp's next's next is not NULL:
        Set temp = temp's next
    Set temp's next to NULL
    Free temp's next
    Print "Deleted from the end"

Function DeleteFromEnd(head):
    If head is NULL:
        Print "List is empty. Cannot delete"
        Return

    If head's next is NULL:
        Free head
        Set head to NULL
        Print "Deleted from the end"
        Return

    Set temp = head
    While temp's next's next is not NULL:
        Set temp = temp's next
    Set temp's next to NULL
```

```
    Free temp's next
    Print "Deleted from the end"

Function DeleteFromPosition(head, position):
    If head is NULL:
        Print "List is empty. Cannot delete"
        Return

    If position is 1:
        Call DeleteFromBeginning(head)
        Return

    Set temp = head
    currentPosition = 1

    While currentPosition < position - 1 and temp is not NULL
        Set temp = temp's next
        Increment currentPosition

    If temp is NULL or temp's next is NULL:
        Print "Position out of bounds"
        Return

    Set nodeToDelete = temp's next
    Set temp's next to temp's next's next
    Free nodeToDelete
    Print "Deleted from position"

Function Search(head, data):
    Set temp = head
    Set position = 1

    While temp is not NULL:
        If temp's data is equal to data:
            Print "Element found at position"
            Return
        Set temp = temp's next
        Increment position

    Print "Element not found in the list"

Function Display(head):
    If head is NULL:
        Print "List is empty"
        Return

    Set temp = head
    Print "List: "
    While temp is not NULL:
        Print temp's data
```

```
        Set temp = temp's next
    Print "End of List"

Function Main():
    Initialize head as NULL

    While True:
        Print Menu:
            1. Insert at Beginning
            2. Insert at End
            3. Insert at Given Position
            4. Delete from Beginning
            5. Delete from End
            6. Delete from Given Position
            7. Search for an Element
            8. Display the List
            9. Exit

        Get user's choice

        If choice is 1:
            Prompt user to enter data
            Call InsertAtBeginning(head, data)

        If choice is 2:
            Prompt user to enter data
            Call InsertAtEnd(head, data)

        If choice is 3:
            Prompt user to enter data and position
            Call InsertAtPosition(head, data, position)

        If choice is 4:
            Call DeleteFromBeginning(head)

        If choice is 5:
            Call DeleteFromEnd(head)

        If choice is 6:
            Prompt user to enter position
            Call DeleteFromPosition(head, position)

        If choice is 7:
            Prompt user to enter data
            Call Search(head, data)

        If choice is 8:
            Call Display(head)

        If choice is 9:
```

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

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

// Structure for Doubly Linked List Node
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

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

// Insert at the beginning of the list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        newNode->next = *head;
        (*head)->prev = newNode;
        *head = newNode;
    }
}

// Insert at the end of the list
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}

// Insert at a given position
void insertAtPosition(struct Node** head, int data, int position) {
    if (position < 1) {
        printf("Invalid position!\n");
        return;
```

```
    }

    struct Node* newNode = createNode(data);
    if (position == 1) {
        insertAtBeginning(head, data);
        return;
    }

    struct Node* temp = *head;
    int i = 1;
    while (temp != NULL && i < position - 1) {
        temp = temp->next;
        i++;
    }

    if (temp == NULL) {
        printf("Position out of bounds!\n");
    } else {
        newNode->next = temp->next;
        newNode->prev = temp;

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

        temp->next = newNode;
    }


// Delete from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
    if (*head != NULL) {
        (*head)->prev = NULL;
    }
    free(temp);


// Delete from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
```

```
    free(temp);
}

// Delete from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

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

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

    free(temp);
}

// Delete at a given position
void deleteAtPosition(struct Node** head, int position) {
    if (position < 1) {
        printf("Invalid position!\n");
        return;
    }

    struct Node* temp = *head;
    int i = 1;

    while (temp != NULL && i < position) {
        temp = temp->next;
        i++;
    }

    if (temp == NULL) {
        printf("Position out of bounds!\n");
        return;
    }

    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    } else {
        *head = temp->next;
```

```c
    }

    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }

    free(temp);
}

// Search for an element
void search(struct Node* head, int key) {
    struct Node* temp = head;
    int position = 1;

    while (temp != NULL) {
        if (temp->data == key) {
            printf("Element %d found at position %d\n", key, position);
            return;
        }
        temp = temp->next;
        position++;
    }

    printf("Element %d not found in the list.\n", key);
}

// Display the list
void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }

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

// Main function to drive the menu
int main() {
    struct Node* head = NULL;
    int choice, data, position;

    while (1) {
        printf("\nMenu:\n");
```

```c
    printf("1. Insert at beginning\n");
    printf("2. Insert at end\n");
    printf("3. Insert at a given position\n");
    printf("4. Delete from beginning\n");
    printf("5. Delete from end\n");
    printf("6. Delete from a given position\n");
    printf("7. Search for an element\n");
    printf("8. Display the list\n");
    printf("9. Exit\n");

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

    switch (choice) {
        case 1:
            printf("Enter data: ");
            scanf("%d", &data);
            insertAtBeginning(&head, data);
            break;
        case
            int printf(const char *__restrict__, ...)
            printf("Enter data: ");
            scanf("%d", &data);
            insertAtEnd(&head, data);
            break;
        case 3:
            printf("Enter data: ");
            scanf("%d", &data);
            printf("Enter position: ");
            scanf("%d", &position);
            insertAtPosition(&head, data, position);
            break;
        case 4:
            deleteFromBeginning(&head);
            break;
        case 5:
            deleteFromEnd(&head);
            break;
        case 6:
            printf("Enter position: ");
            scanf("%d", &position);
            deleteAtPosition(&head, position);
            break;
        case 7:
            printf("Enter element to search: ");
            scanf("%d", &data);
            search(head, data);
            break;
```

```
            case 8:
                display(head);
                break;
            case 9:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

OUTPUT:

```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert at a given position
4. Delete from beginning
5. Delete from end
6. Delete from a given position
7. Search for an element
8. Display the list
9. Exit
Enter your choice: 1
Enter data: 10

Menu:
1. Insert at beginning
2. Insert at end
3. Insert at a given position
4. Delete from beginning
5. Delete from end
6. Delete from a given position
7. Search for an element
8. Display the list
9. Exit
Enter your choice: 2
Enter data: 20

Menu:
1. Insert at beginning
2. Insert at end
3. Insert at a given position
4. Delete from beginning
5. Delete from end
6. Delete from a given position
7. Search for an element
8. Display the list
9. Exit
Enter your choice: 8
10 <-> 20 <-> NULL
```

PSEUDOCODE:

```
Define Node:
    Integer data
    Node* next
    Node* prev


Function createNode(data):
    Create a new Node
    Set newNode.data = data
    Set newNode.next = NULL
    Set newNode.prev = NULL
    Return newNode


Function insertAtBeginning(head, data):
    Create a newNode by calling createNode(data)
    If head is NULL:
        Set head to newNode
    Else:
        Set newNode.next to head
        Set head.prev to newNode
        Set head to newNode


Function insertAtEnd(head, data):
    Create a newNode by calling createNode(data)
    If head is NULL:
        Set head to newNode
    Else:
        Set temp = head
        While temp.next is not NULL:
            Set temp to temp.next
        Set temp.next to newNode
        Set newNode.prev to temp


Function insertAtPosition(head, data, position):
    If position < 1:
        Print "Invalid position!"
        Return
    Create a newNode by calling createNode(data)
    If position == 1:
        Call insertAtBeginning(head, data)
        Return
    Set temp = head
    Set i = 1
    While temp is not NULL and i < position - 1:
        Set temp to temp.next
        Increment i
```

```
    If temp is NULL:
        Print "Position out of bounds!"
        Return
    Set newNode.next to temp.next
    Set newNode.prev to temp
    If temp.next is not NULL:
        Set temp.next.prev to newNode
    Set temp.next to newNode

g
Function deleteFromBeginning(head):
    If head is NULL:
        Print "List is empty!"
        Return
    Set temp = head
    Set head to head.next
    If head is not NULL:
        Set head.prev to NULL
    Free temp

Function deleteFromEnd(head):
    If head is NULL:
        Print "List is empty!"
        Return
    Set temp = head
    While temp.next is not NULL:
        Set temp to temp.next
    If temp.prev is not NULL:
        Set temp.prev.next to NULL
    Else:
        Set head to NULL
    Free temp


Function deleteAtPosition(head, position):
    If position < 1:
        Print "Invalid position!"
        Return
    Set temp = head
    Set i = 1
    While temp is not NULL and i < position:
        Set temp to temp.next
        Increment i
    If temp is NULL:
        Print "Position out of bounds!"
        Return
    If temp.prev is not NULL:
        Set temp.prev.next to temp.next
    Else:
        Set head to temp.next
```

```
    If temp.next is not NULL:
        Set temp.next.prev to temp.prev
    Free temp


Function search(head, key):
    Set temp = head
    Set position = 1
    While temp is not NULL:
        If temp.data == key:
            Print "Element key found at position position"
            Return
        Set temp to temp.next
        Increment position
    Print "Element key not found in the list"

Function display(head):
    If head is NULL:
        Print "List is empty!"
        Return
    Set temp = head
    While temp is not NULL:
        Print temp.data " <-> "
        Set temp to temp.next
    Print "NULL"

4
Function main():
    Set head = NULL
    Set choice, data, position
    While True:
        Print Menu
        Get user choice
        Switch choice:
            Case 1:
                Print "Enter data"
                Get data
                Call insertAtBeginning(head, data)
            Case 2:
                Print "Enter data"
                Get data
                Call insertAtEnd(head, data)
            Case 3:
                Print "Enter data"
                Get data
                Print "Enter position"
                Get position
                Call insertAtPosition(head, data, position)
            Case 4:
                Call deleteFromBeginning(head)
```

```
            Call deleteFromBeginning(head)
        Case 5:
            Call deleteFromEnd(head)
        Case 6:
            Print "Enter position"
            Get position
            Call deleteAtPosition(head, position)
        Case 7:
            Print "Enter element to search"
            Get data
            Call search(head, data)
        Case 8:
            Call display(head)
        Case 9:
            Exit
        Default:
            Print "Invalid choice! Please try again."
```