<u>24BBS0161-SOHAM SANDEEP NAPHADE</u>

<u>DATA STRUCTURES AND ALGORITHM ASSIGNMENT</u>

1. Write a menu driven program to implement the following operations on stack.

a. PUSH() b. POP() c. Display()

Step 1: Initialize

1. Define the maximum size of the stack as MAX.
2. Initialize the stack as an integer array of size MAX.
3. Set top = -1 to indicate that the stack is initially empty.

Step 2: Operations

1. **Push Operation**
    - Input: value to be added to the stack.
    - Check if the stack is full:
        - If top == MAX - 1, print "Stack is full; cannot push."
    - Else:
        - Increment top by 1.
        - Add value to stack[top].
        - Print the message "value pushed to stack."
2. **Pop Operation**
    - Check if the stack is empty:
        - If top == -1, print "Stack is empty" and return -1.
    - Else:
        - Store the value at stack[top] in a variable, say item.
        - Decrement top by 1.
        - Print "item popped from stack."
        - Return the item.

3. **Display Operation**
    - Check if the stack is empty:
        - If top == -1, print "Stack is empty."
    - Else:
        - Print all elements in the stack from stack[0] to stack[top].

Step 3: Main Program Loop

1. Start an infinite loop to continuously provide a menu for operations
2. Accept the user's choice (choice):
    1. If choice == 1:
        1. Input the value to be pushed.
        2. Call the push() function with the input value.
    2. If choice == 2:
        1. Call the pop() function.
    3. If choice == 3:
        1. Call the display() function.
    4. If choice == 4:
        1. Print "Exiting program" and terminate the program.
    5. For any other value of choice:
        1. Print "Invalid choice."

Step 4: End

3. Exit the program when the user selects the "Exit" option in the menu

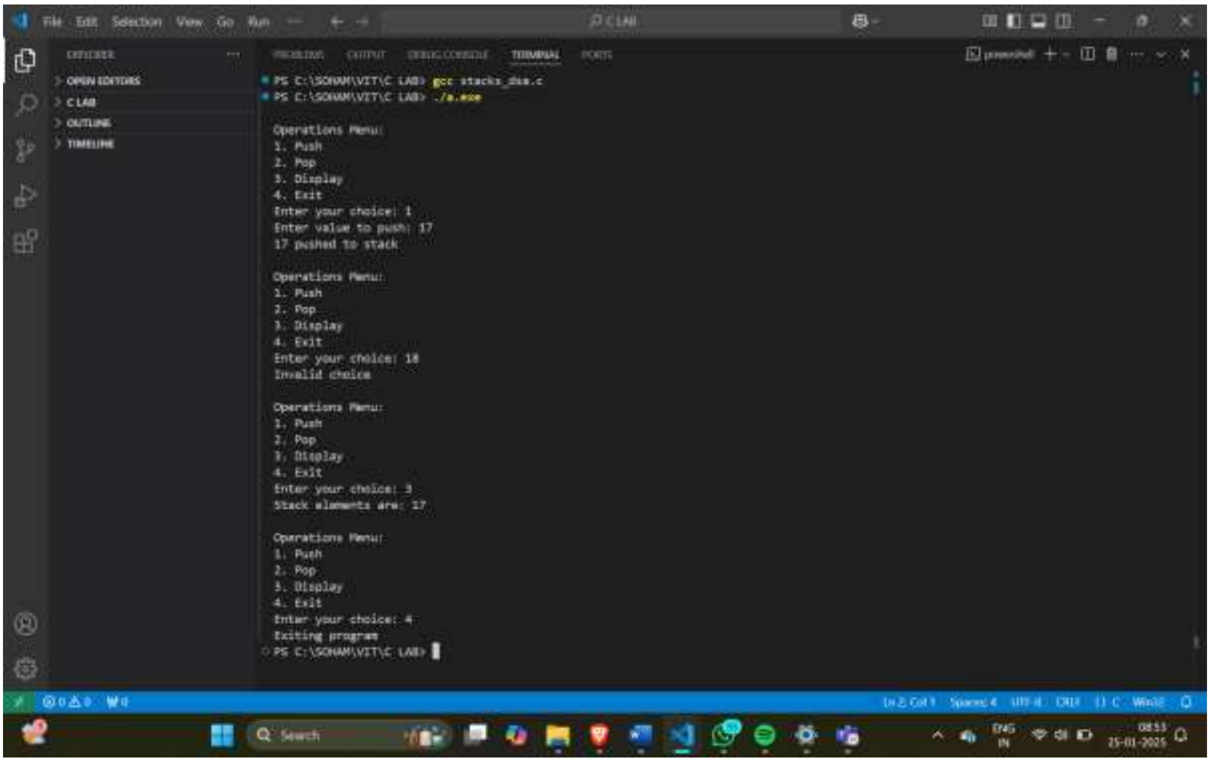
```
#include <stdio.h>

#define MAX 5


int stack[MAX];

int top = -1;


void push(int value);
```

```
int pop();
void display();

void push(int value) {
    if (top == MAX - 1) {
        printf("Stack is full cannot push %d\n", value);
    } else {
        top++;
        stack[top] = value;
        printf("%d pushed to stack\n", value);
    }
}

int pop() {
    if (top == -1) {
        printf("Stack is empty\n");
        return -1;
    } else {
        int item = stack[top];
        top--;
        printf("%d popped from stack\n", item);
        return item;
    }
}

void display() {
    if (top == -1) {
        printf("Stack is empty\n");
```

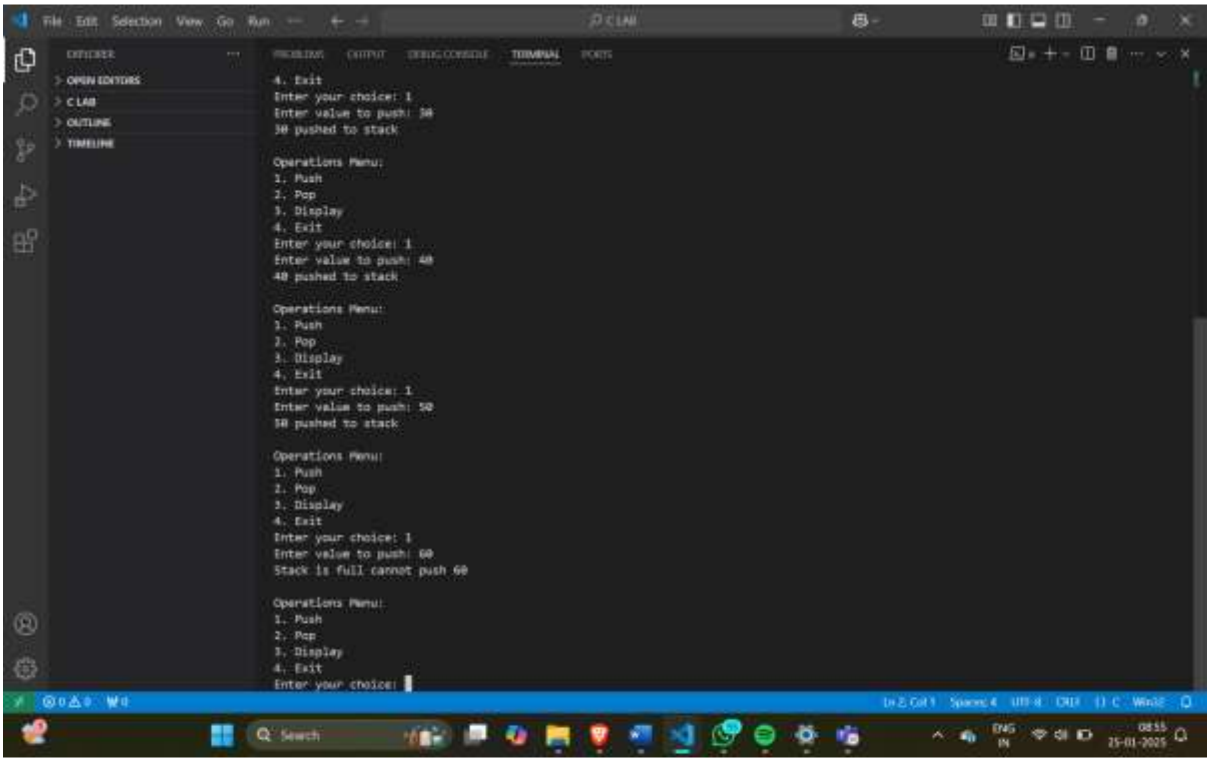
```
    } else {
        printf("Stack elements are: ");
        for (int i = 0; i <= top; i++) {
            printf("%d ", stack[i]);
        }
        printf("\n");
    }
}

int main() {
    int choice, value;

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

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(value);
                break;
            case 2:
```

```
            pop();
            break;
          case 3:
            display();
            break;
          case 4:
            printf("Exiting program\n");
            return 0;
          default:
            printf("Invalid choice\n");
        }
    }

    return 0;
}
```

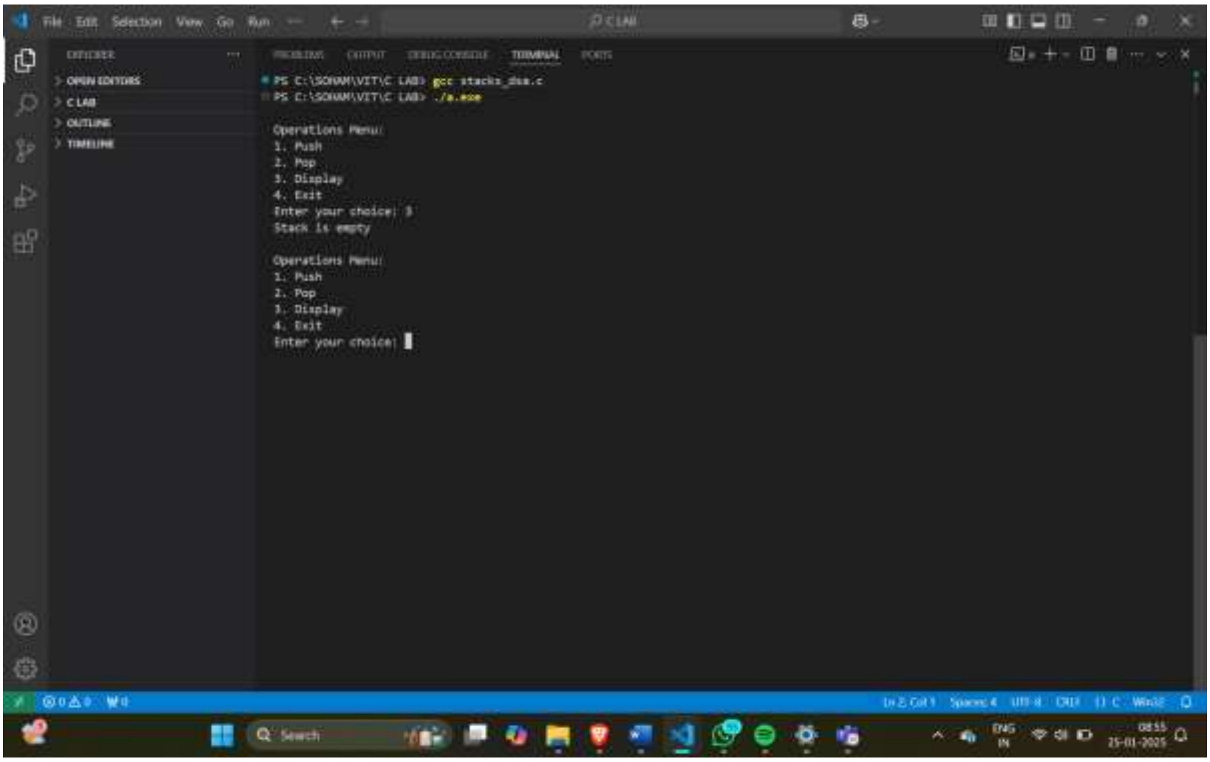
Regular push and display.



Stack overflow



Stack underflow



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

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

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_SIZE 5


typedef struct {

    int items[MAX_SIZE];

    int front;

    int rear;

} Queue;


void initQueue(Queue *q);

int isFull(Queue *q);
```

```
int isEmpty(Queue *q);
void enqueue(Queue *q, int value);
int dequeue(Queue *q);
void display(Queue *q);

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

int isFull(Queue *q) {
    return (q->rear == MAX_SIZE - 1);
}

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

void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full\n");
        return;
    }

    if (q->front == -1)
        q->front = 0;

    q->rear++;
```

```
    q->items[q->rear] = value;
    printf("%d added to the queue.\n", value);
}

int dequeue(Queue *q) {
    int item;

    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    }

    item = q->items[q->front];
    q->front++;

    printf("%d removed from the queue\n", item);
    return item;
}

void display(Queue *q) {
    int i;

    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }

    printf("Queue elements: ");
```

```
    for (i = q->front; i <= q->rear; i++) {
        printf("%d ", q->items[i]);
    }
    printf("\n");
}

int main() {
    Queue q;
    int choice, value;

    initQueue(&q);

    while (1) {
        printf("\nQueue Operations Menu:\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 value to enqueue: ");
                scanf("%d", &value);
                enqueue(&q, value);
                break;
```

```
        case 2:
          dequeue(&q);
          break;

        case 3:
          display(&q);
          break;

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

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

  return 0;
}
```

ALGORITHM:

Step 1: Initialize the Queue

1. Define MAX_SIZE as the maximum size of the queue.
2. Create a Queue structure with:
    - An array items of size MAX_SIZE.
    - Two integer variables front and rear to track the front and rear of the queue.
3. Initialize the queue:

   - Set front = -1 and rear = -1.

Step 2: Define Supporting Functions

2.1 isFull(Queue *q)

- **Input**: Pointer to the queue (q).
- **Logic**:
  - If rear == MAX_SIZE - 1, the queue is full.
  - Return true if full, otherwise false.

2.2 isEmpty(Queue *q)

- **Input**: Pointer to the queue (q).
- **Logic**:
  - If front == -1 or front > rear, the queue is empty.
  - Return true if empty, otherwise false.

Step 3: Define Queue Operations

3.1 enqueue(Queue *q, int value)

- **Input**: Pointer to the queue (q), value to be added (value).
- **Logic**:
  - If the queue is full (isFull()), print "Queue is full" and exit.
  - Otherwise:
    - If front == -1, set front = 0.
    - Increment rear by 1.
    - Add value to items[rear].
    - Print a message indicating that the value was added to the queue.

3.2 dequeue(Queue *q)

- **Input**: Pointer to the queue (q).
- **Logic**:
  - If the queue is empty (isEmpty()), print "Queue is empty" and return -1.

- Otherwise:
  - Retrieve the value at items[front].
  - Increment front by 1.
  - Print a message indicating that the value was removed from the queue.
  - Return the retrieved value.

3.3 display(Queue *q)

- **Input**: Pointer to the queue (q).
- **Logic**:
  - If the queue is empty (isEmpty()), print "Queue is empty" and exit.
  - Otherwise:
    - Print all elements in items from front to rear.

Step 4: Main Program Logic

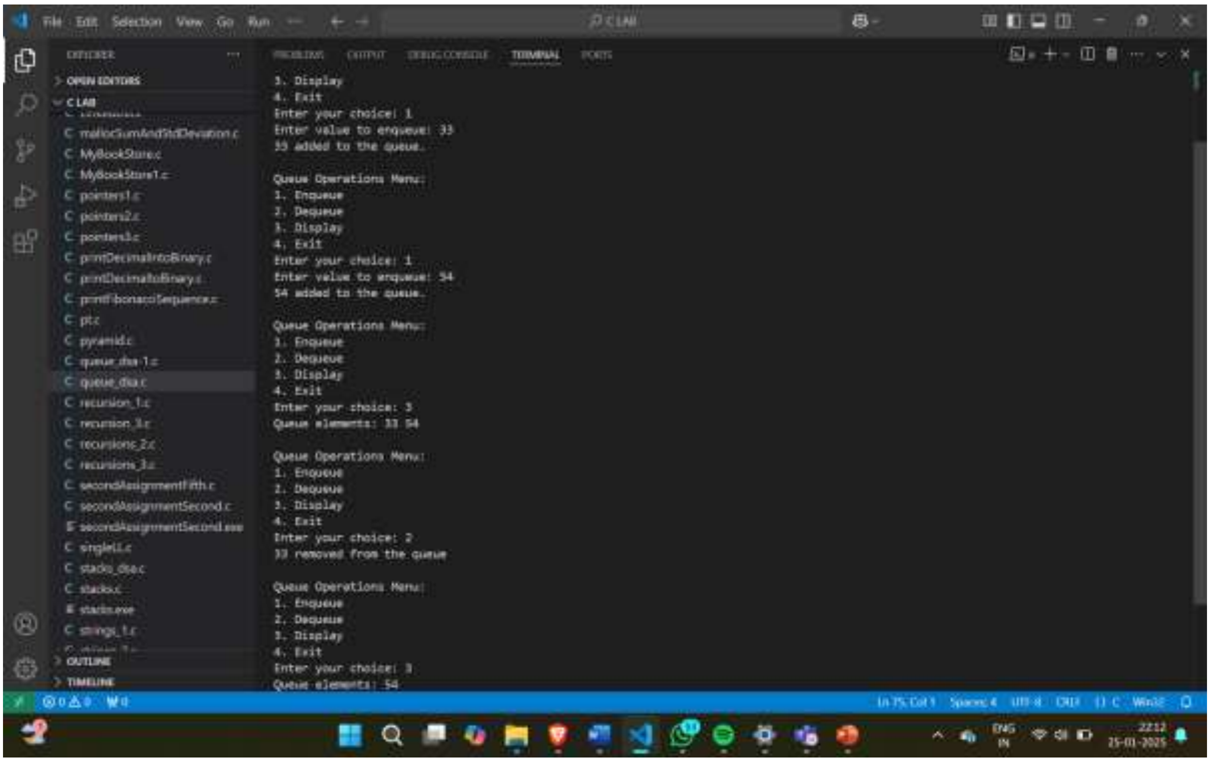
1. Initialize the queue by calling initQueue().
2. Start an infinite loop to present the menu of operations

1. Accept the user's choice (choice) and perform the corresponding operation:
   - **If choice == 1**:
     - Prompt the user for a value to enqueue.
     - Call enqueue() with the input value.
   - **If choice == 2**:
     - Call dequeue().
   - **If choice == 3**:
     - Call display().
   - **If choice == 4**:
     - Print "Exiting program" and terminate the program.
   - **For any other choice**:
     - Print "Invalid choice."

2. End the loop and exit the program when the user selects the "Exit" option.

Step 5: End

- Exit the program.



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

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_SIZE 5


typedef struct {

   int items[MAX_SIZE];

   int front;

   int rear;

   int size;

} CircularQueue;


void initQueue(CircularQueue *q);

int isFull(CircularQueue *q);

int isEmpty(CircularQueue *q);

void enqueue(CircularQueue *q, int value);

int dequeue(CircularQueue *q);

void display(CircularQueue *q);


void initQueue(CircularQueue *q) {

   q->front = -1;
```

```c
    q->rear = -1;
    q->size = 0;
}

int isFull(CircularQueue *q) {
    return (q->size == MAX_SIZE);
}

int isEmpty(CircularQueue *q) {
    return (q->size == 0);
}

void enqueue(CircularQueue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }

    if (isEmpty(q)) {
        q->front = 0;
    }

    q->rear = (q->rear + 1) % MAX_SIZE;
    q->items[q->rear] = value;
    q->size++;

    printf("%d added to the queue.\n", value);
}
```

```c
int dequeue(CircularQueue *q) {
    int item;

    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }

    item = q->items[q->front];
    q->front = (q->front + 1) % MAX_SIZE;
    q->size--;

    if (isEmpty(q)) {
        q->front = -1;
        q->rear = -1;
    }

    printf("%d removed from the queue.\n", item);
    return item;
}

void display(CircularQueue *q) {
    int i, count;

    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
```

```
    }

    printf("Queue elements: ");
    for (i = 0, count = 0; count < q->size; i = (i + 1) % MAX_SIZE, count++) {
        printf("%d ", q->items[i]);
    }
    printf("\n");
}

int main() {
    CircularQueue q;
    int choice, value;

    initQueue(&q);

    while (1) {

        printf("\nCircular Queue Operations Menu:\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 value to enqueue: ");
            scanf("%d", &value);
            enqueue(&q, value);
            break;

        case 2:
            dequeue(&q);
            break;
        case 3:
            display(&q);
            break;

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

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

    return 0;
}
```

ALGORITHM:

 Data Structure

- Fixed-size array of integers
- Front index

- Rear index
- Current size tracker

� **Initialize Queue**

- Set front = -1
- Set rear = -1
- Set size = 0

� **Enqueue Operation**

- Check if queue is full
- If empty, set front to 0
- Increment rear circularly using modulo
- Insert element at rear
- Increment size
- Display success message

� **Dequeue Operation**

- Check if queue is empty
- Remove element from front
- Increment front circularly using modulo
- Decrement size
- If queue becomes empty, reset front and rear to -1
- Return removed element

� **Display Operation**

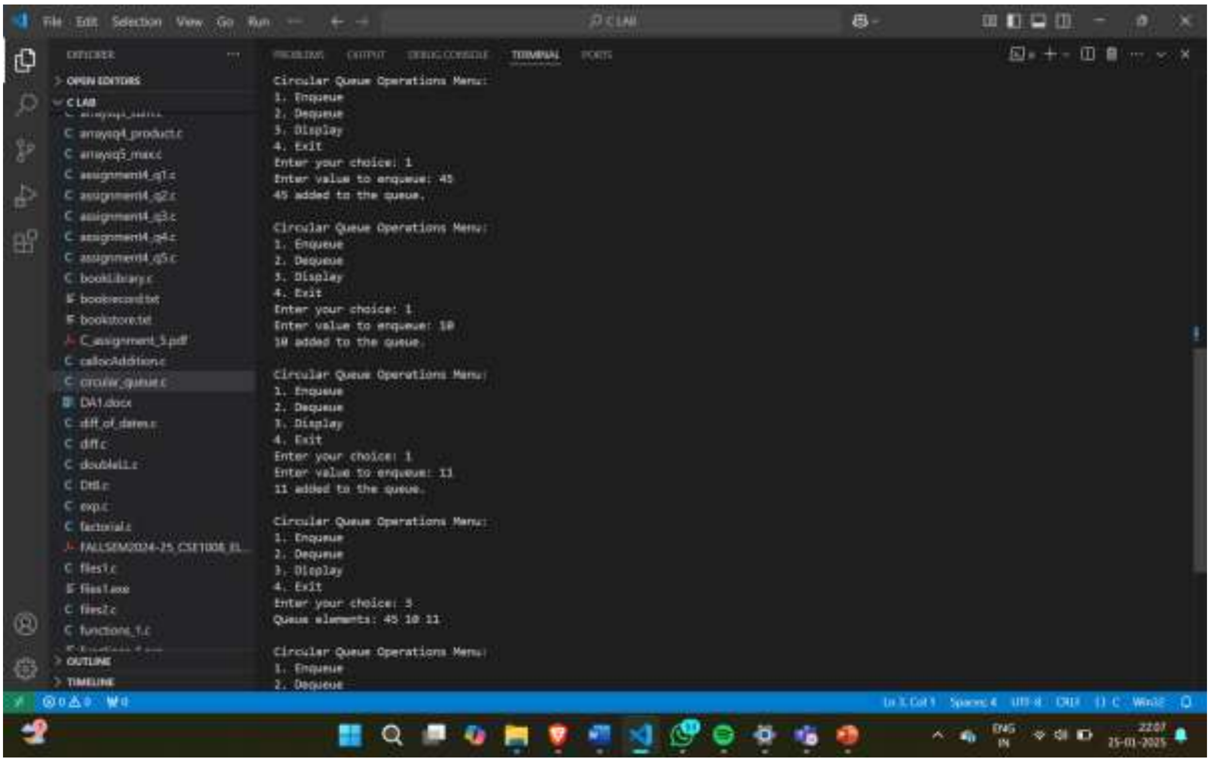
- Check if queue is empty
- Traverse elements from front to rear
- Use circular traversal with modulo arithmetic
- Print elements

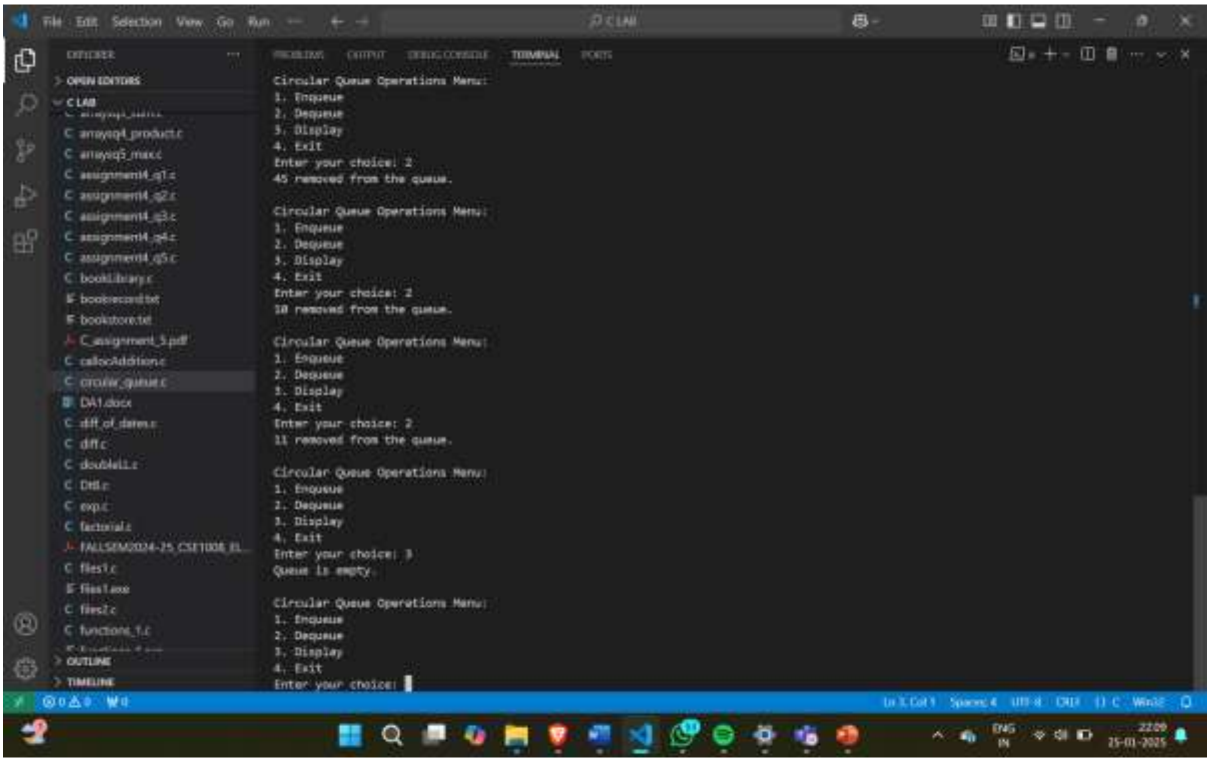
� **Helper Functions**

- isFull(): Check if size equals MAX_SIZE
- isEmpty(): Check if size is 0

Main Menu

- Provide options:
  - Enqueue
  - Dequeue
  - Display
  - Exit
- Handle user input through switch statement




Circular Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
45 removed from the queue.
Circular Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
10 removed from the queue.
Circular Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
11 removed from the queue.
Circular Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue is empty.
Circular Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice:

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**

```
#include <stdio.h>

#include <stdlib.h>


typedef struct Node {

    int data;

    struct Node* next;

} Node;


Node* createNode(int data);

void insertAtBeginning(Node** head, int data);

void insertAtEnd(Node** head, int data);

void insertAtPosition(Node** head, int data, int position);

void deleteFromBeginning(Node** head);

void deleteFromEnd(Node** head);

void deleteFromPosition(Node** head, int position);

void display(Node* head);

int getLength(Node* head);


Node* createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed!\n");

        exit(1);

    }
```

```
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void insertAtBeginning(Node** head, int data) {
    Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
    printf("Inserted %d at the beginning.\n", data);
}

void insertAtEnd(Node** head, int data) {
    Node* newNode = createNode(data);

    if (*head == NULL) {
        *head = newNode;
        printf("Inserted %d at the end.\n", data);
        return;
    }

    Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    printf("Inserted %d at the end.\n", data);
}
```

```
void insertAtPosition(Node** head, int data, int position) {
    int len = getLength(*head);

    if (position < 1 || position > len + 1) {
        printf("Invalid position!\n");
        return;
    }

    if (position == 1) {
        insertAtBeginning(head, data);
        return;
    }

    Node* newNode = createNode(data);
    Node* temp = *head;

    for (int i = 1; i < position - 1; i++) {
        temp = temp->next;
    }

    newNode->next = temp->next;
    temp->next = newNode;
    printf("Inserted %d at position %d.\n", data, position);
}

void deleteFromBeginning(Node** head) {
    if (*head == NULL) {
```

```
        printf("List is empty!\n");
        return;
    }

    Node* temp = *head;
    *head = (*head)->next;
    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);
}

void deleteFromEnd(Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

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

    Node* temp = *head;
    Node* prev = NULL;

    while (temp->next != NULL) {
```

```
        prev = temp;
        temp = temp->next;
    }

    printf("Deleted %d from the end.\n", temp->data);
    prev->next = NULL;
    free(temp);
}

void deleteFromPosition(Node** head, int position) {
    int len = getLength(*head);

    if (position < 1 || position > len) {
        printf("Invalid position!\n");
        return;
    }

    if (position == 1) {
        deleteFromBeginning(head);
        return;
    }

    Node* temp = *head;
    Node* prev = NULL;

    for (int i = 1; i < position; i++) {
        prev = temp;
        temp = temp->next;
```

```
    }

    prev->next = temp->next;
    printf("Deleted %d from position %d.\n", temp->data, position);
    free(temp);
}

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

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

int getLength(Node* head) {
    int count = 0;
    Node* temp = head;

    while (temp != NULL) {
        count++;
```

```c
        temp = temp->next;
    }

    return count;
}

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

    while (1) {
        printf("\nSingly Linked List Operations:\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. Display\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
```

```
    insertAtBeginning(&head, data);
    break;

case 2:
    printf("Enter data to insert: ");
    scanf("%d", &data);
    insertAtEnd(&head, data);
    break;

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

```
            deleteFromPosition(&head, position);
            break;

        case 7:
            display(head);
            break;

        case 8:
            printf("Exiting program.\n");
            exit(0);

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

    return 0;
}
```

ALGORITHM:

Step 1: Define Node Structure

Create a structure Node with:

data (to store the value of the node).

next (pointer to the next node).

Step 2: Define Operations

2.1 createNode(int data)

Input: Data for the new node.

Logic:

Allocate memory for a new node.

If memory allocation fails, print "Memory allocation failed!" and exit.

Assign data to the node's data field and set next to NULL.

Output: Return the newly created node.

2.2 insertAtBeginning(Node** head, int data)

Input: Pointer to the head of the list and data to insert.

Logic:

Create a new node using createNode(data).

Set newNode->next to the current head.

Update head to point to the new node.

Print a message confirming the insertion.

2.3 insertAtEnd(Node** head, int data)

Input: Pointer to the head of the list and data to insert.

Logic:

Create a new node using createNode(data).

If the list is empty (*head == NULL):

Set head to the new node.

Print a message confirming the insertion.

Otherwise:

Traverse the list to find the last node.

Set the last node's next to the new node.

Print a message confirming the insertion.

2.4 insertAtPosition(Node** head, int data, int position)

Input: Pointer to the head, data to insert, and position.

Logic:

Check if the position is valid (1 ≤ position ≤ getLength(head) + 1).

If position is invalid, print "Invalid position!" and exit.

If position is 1, call insertAtBeginning.

Otherwise:

Create a new node using createNode(data).

Traverse the list to the node just before the target position.

Adjust pointers to insert the new node at the specified position.

Print a message confirming the insertion.

2.5 deleteFromBeginning(Node** head)

Input: Pointer to the head of the list.

Logic:

If the list is empty, print "List is empty!" and exit.

Store the current head in a temporary variable.

Update head to point to the next node.

Free the memory of the temporary node and print a message confirming the deletion.

2.6 deleteFromEnd(Node** head)

Input: Pointer to the head of the list.

Logic:

If the list is empty, print "List is empty!" and exit.

If the list has only one node:

Free the node and set head = NULL.

Print a message confirming the deletion.

Otherwise:

Traverse the list to find the second-last node.

Free the memory of the last node and set the second-last node's next to NULL.

Print a message confirming the deletion.

2.7 deleteFromPosition(Node** head, int position)

Input: Pointer to the head and position of the node to delete.

Logic:

Check if the position is valid (1 ≤ position ≤ getLength(head)).

If position is invalid, print "Invalid position!" and exit.

If position is 1, call deleteFromBeginning.

Otherwise:

Traverse the list to the node just before the target position.

Adjust pointers to remove the target node.

Free the memory of the target node and print a message confirming the deletion.

2.8 display(Node* head)

Input: Head pointer of the list.

Logic:

If the list is empty, print "List is empty!" and exit.

Traverse the list and print the data of each node.

2.9 getLength(Node* head)

Input: Head pointer of the list.

Logic:

Initialize a counter to 0.

Traverse the list, incrementing the counter for each node.

Return the counter.

Step 3: Main Program

Initialize head to NULL.

Display the operations menu:

Accept the user's choice (choice):

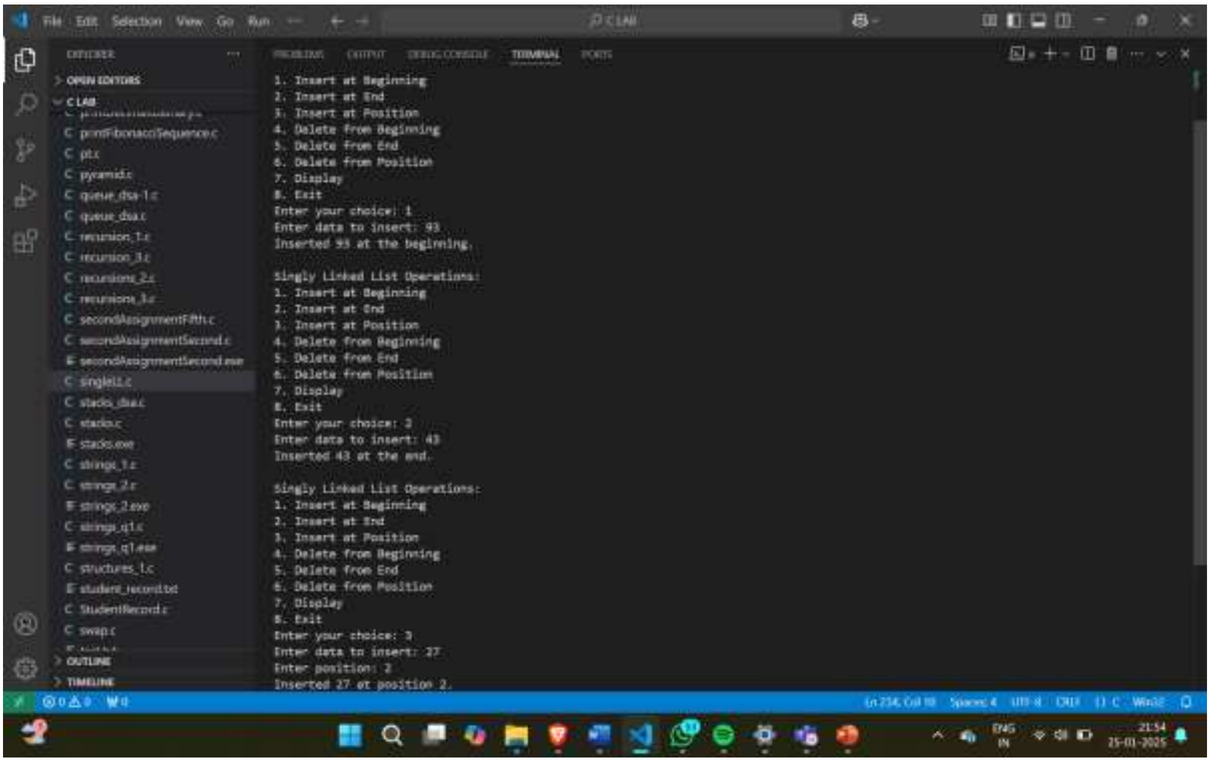
Call the corresponding function based on the choice.

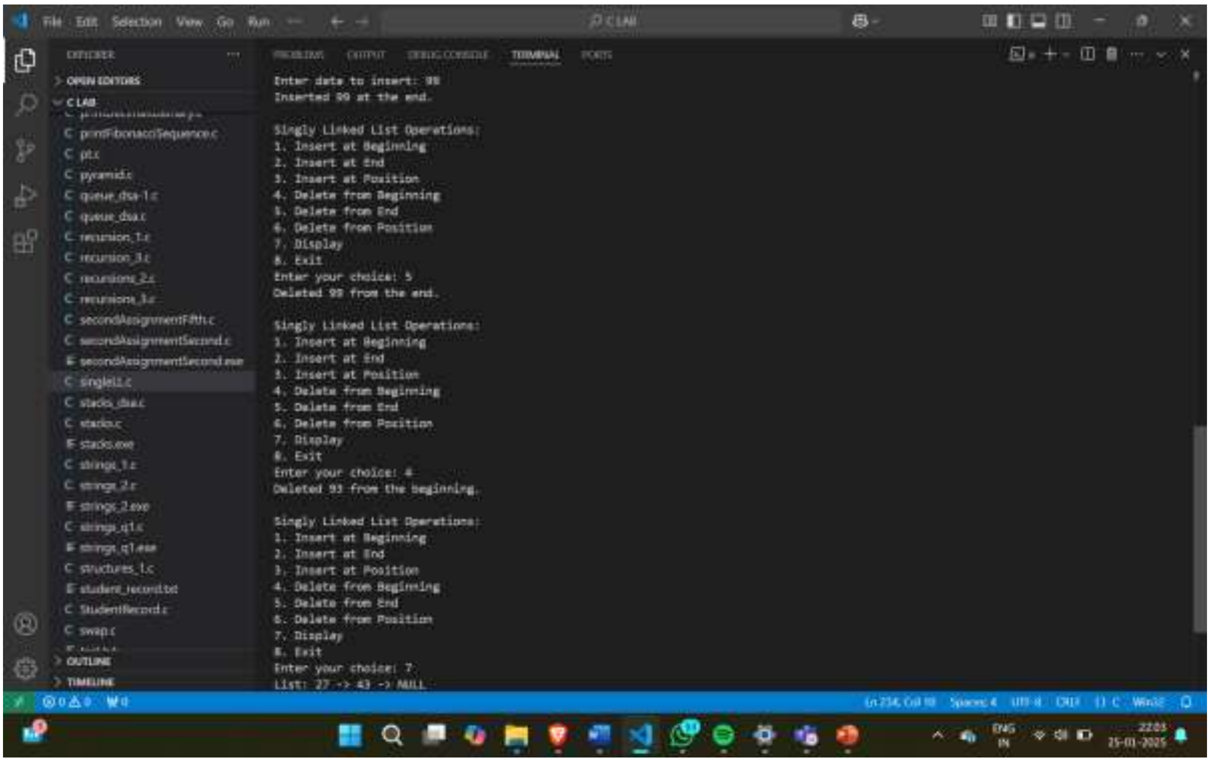
Handle invalid input by printing "Invalid choice. Try again."

Repeat the process until the user selects "Exit."

Step 4: End

Exit the program




Enter data to insert: 99
Inserted 99 at the end.
Singly Linked List Operations:
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. Display
8. Exit
Enter your choice: 5
Deleted 99 from the end.
Singly Linked List Operations:
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. Display
8. Exit
Enter your choice: 4
Deleted 93 from the beginning.
Singly Linked List Operations:
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. Display
8. Exit
Enter your choice: 7
List: 27 -> 43 -> NULL

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.

Algorithm for Doubly Linked List Operations:

1. Node Structure
   - Contains: data, prev pointer, next pointer
2. Insertion Operations a. Insert at Beginning
   - Create new node
   - If list empty, set head to new node
   - Else:
     - Set new node's next to current head
     - Set current head's prev to new node
     - Update head to new node

b. Insert at End

   - Create new node
   - If list empty, set head to new node
   - Else:
     - Traverse to last node
     - Set last node's next to new node
     - Set new node's prev to last node

c. Insert at Position

   - Validate position
   - If position is first, use insert at beginning
   - Traverse to position
   - Update links:
     - New node's next = current node's next
     - New node's prev = current node

        - Current node's next's prev = new node (if exists)
        - Current node's next = new node

3. Deletion Operations a. Delete from Beginning
    - If list empty, return
    - Move head to next node
    - Set new head's prev to NULL
    - Free previous head node

b. Delete from End

    - If list empty, return
    - If single node, set head to NULL
    - Else:
        - Traverse to last node
        - Update second last node's next to NULL
        - Free last node

c. Delete from Position

    - Validate position
    - If first position, use delete from beginning
    - Traverse to node
    - Update links:
        - Previous node's next = current node's next
        - Next node's prev = current node's prev (if exists)
    - Free current node

4. Search Operation
    - Traverse list
    - Compare each node's data with target
    - Return position if found
    - Return -1 if not found
5. Display Operation

- Traverse from head to end
- Print each node's data
- Show NULL at start and end

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

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

Node* createNode(int data);
void insertAtBeginning(Node** head, int data);
void insertAtEnd(Node** head, int data);
void insertAtPosition(Node** head, int data, int position);
void deleteFromBeginning(Node** head);
void deleteFromEnd(Node** head);
void deleteFromPosition(Node** head, int position);
void display(Node* head);
int search(Node* head, int key);
int getLength(Node* head);

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
```

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

void insertAtBeginning(Node** head, int data) {
    Node* newNode = createNode(data);

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

    printf("Inserted %d at the beginning.\n", data);
}

void insertAtEnd(Node** head, int data) {
    Node* newNode = createNode(data);

    if (*head == NULL) {
        *head = newNode;
```

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

    printf("Inserted %d at the end.\n", data);
}

void insertAtPosition(Node** head, int data, int position) {
    int len = getLength(*head);

    if (position < 1 || position > len + 1) {
        printf("Invalid position!\n");
        return;
    }

    if (position == 1) {
        insertAtBeginning(head, data);
        return;
    }

    Node* newNode = createNode(data);
    Node* temp = *head;
```

```
    for (int i = 1; i < position - 1; i++) {
        temp = temp->next;
    }

    newNode->next = temp->next;
    newNode->prev = temp;

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

    temp->next = newNode;

    printf("Inserted %d at position %d.\n", data, position);
}

void deleteFromBeginning(Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

    Node* temp = *head;
    *head = (*head)->next;

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

```
    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);
}

void deleteFromEnd(Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

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

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

    temp->prev->next = NULL;
    printf("Deleted %d from the end.\n", temp->data);
    free(temp);
}
```

```
void deleteFromPosition(Node** head, int position) {
    int len = getLength(*head);

    if (position < 1 || position > len) {
        printf("Invalid position!\n");
        return;
    }

    if (position == 1) {
        deleteFromBeginning(head);
        return;
    }

    Node* temp = *head;

    for (int i = 1; i < position; i++) {
        temp = temp->next;
    }

    temp->prev->next = temp->next;

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

    printf("Deleted %d from position %d.\n", temp->data, position);
    free(temp);
```

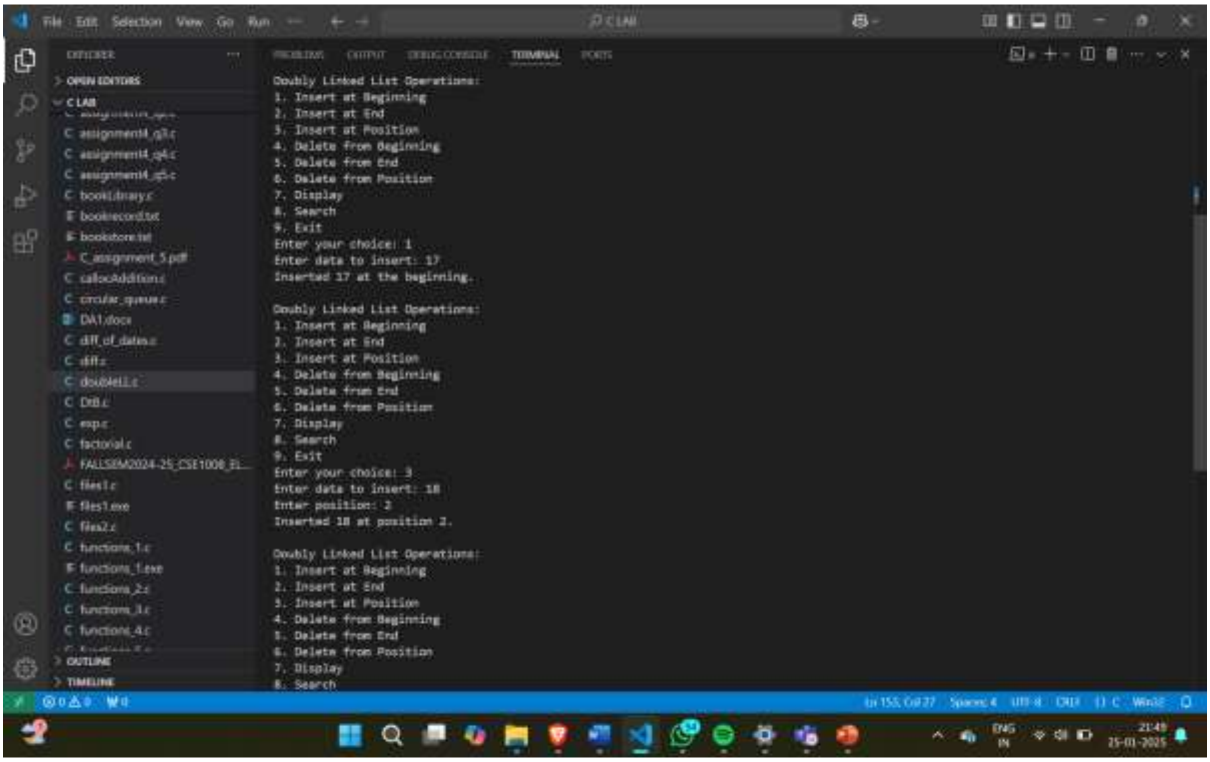
```
}

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

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

int search(Node* head, int key) {
    int position = 1;
    Node* temp = head;

    while (temp != NULL) {
        if (temp->data == key) {
            return position;
        }
        temp = temp->next;
        position++;
    }
```

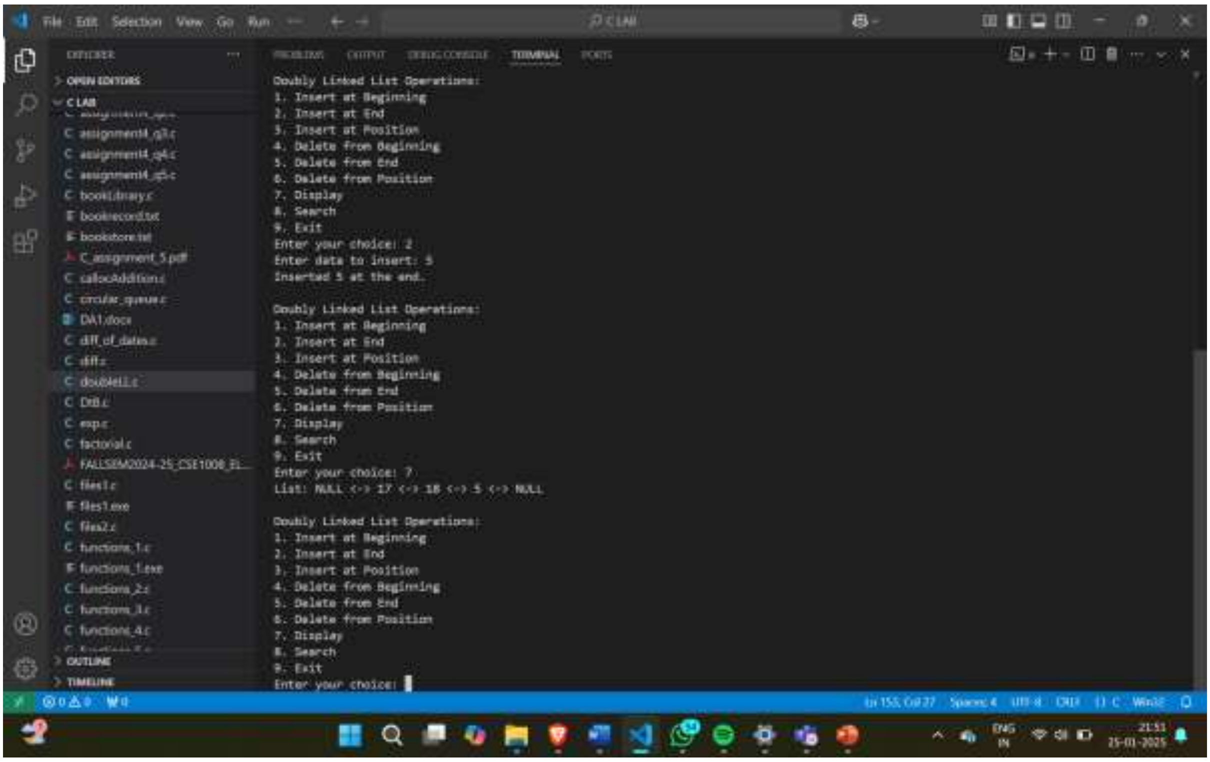
```
    return -1;
}

int getLength(Node* head) {
    int count = 0;
    Node* temp = head;

    while (temp != NULL) {
        count++;
        temp = temp->next;
    }

    return count;
}

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

    while (1) {
        printf("\nDoubly Linked List Operations:\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");
```

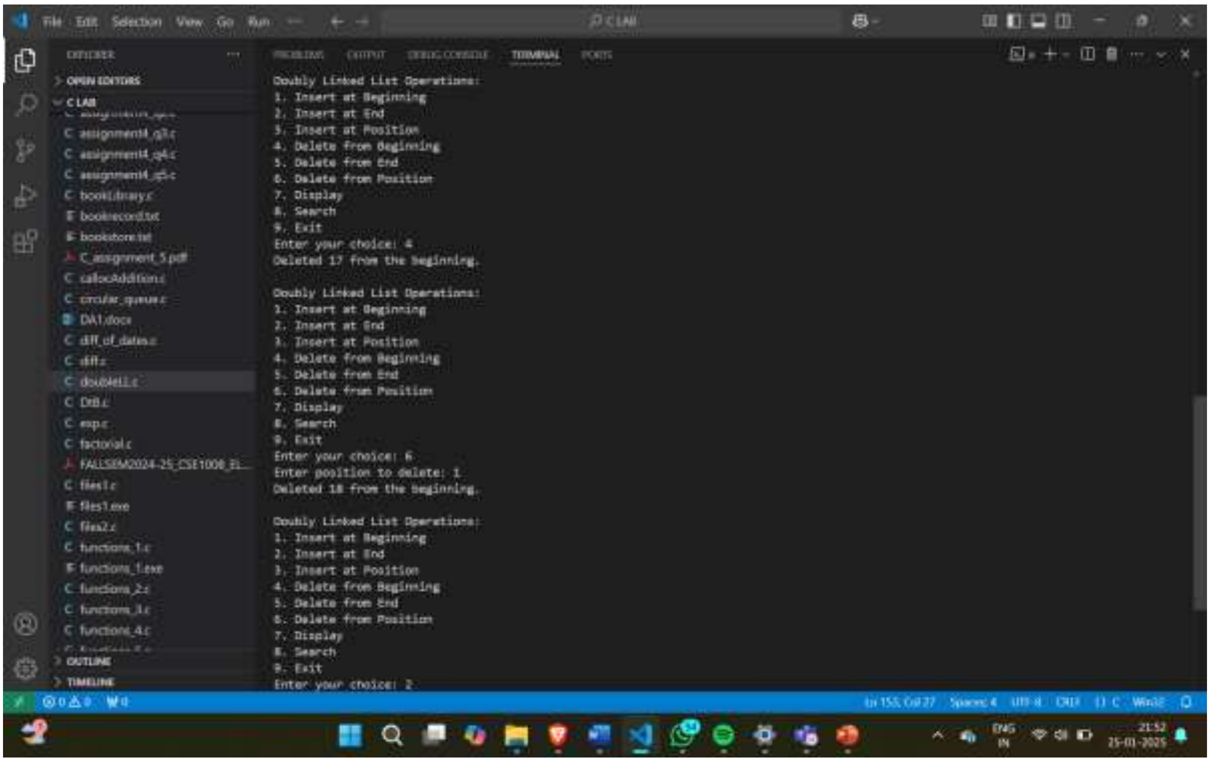
```
printf("7. Display\n");
printf("8. Search\n");
printf("9. Exit\n");
printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter data to insert: ");
        scanf("%d", &data);
        insertAtBeginning(&head, data);
        break;

    case 2:
        printf("Enter data to insert: ");
        scanf("%d", &data);
        insertAtEnd(&head, data);
        break;

    case 3:
        printf("Enter data to insert: ");
        scanf("%d", &data);
        printf("Enter position: ");
        scanf("%d", &position);
        insertAtPosition(&head, data, position);
        break;
```

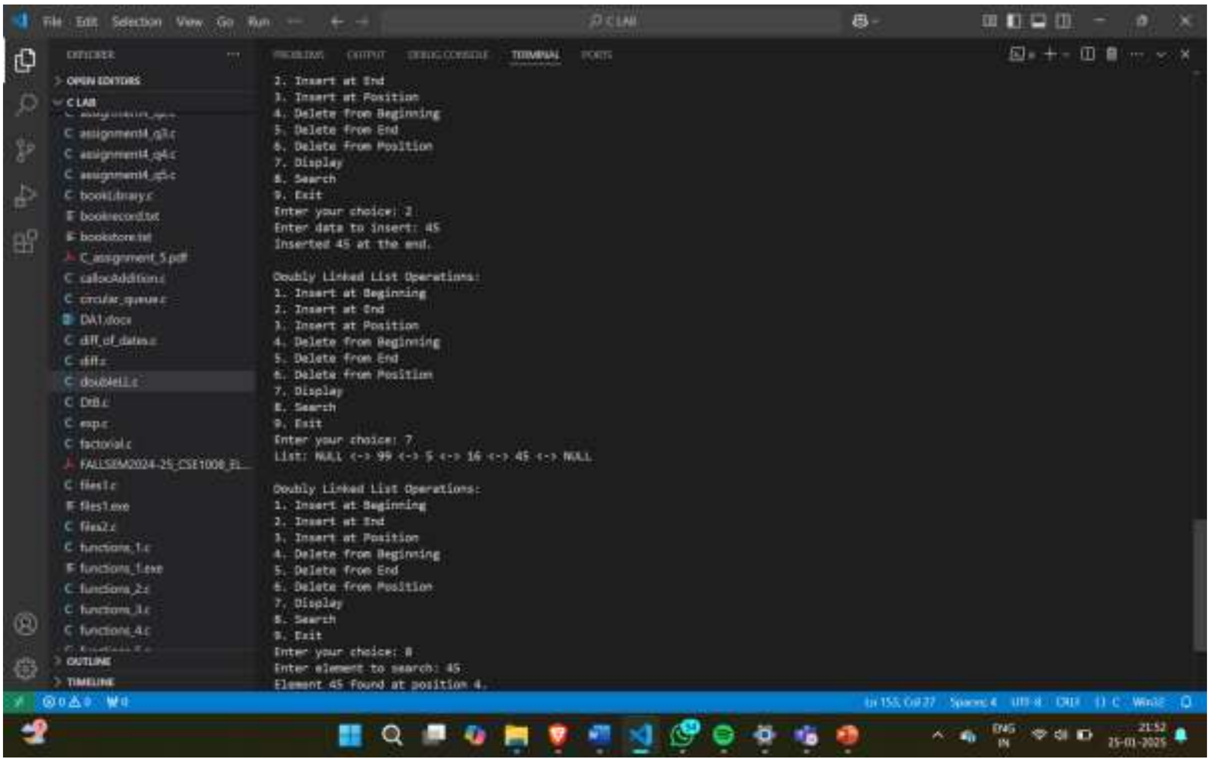
```
case 4:
    deleteFromBeginning(&head);
    break;

case 5:
    deleteFromEnd(&head);
    break;

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

case 7:
    display(head);
    break;

case 8:
    printf("Enter element to search: ");
    scanf("%d", &data);
    result = search(head, data);
    if (result != -1) {
        printf("Element %d found at position %d.\n", data, result);
    } else {
        printf("Element %d not found in the list.\n", data);
    }
    break;
```

```
        case 9:
          printf("Exiting program.\n");
          exit(0);

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

    return 0;
}
```




Doubly Linked List Operations:
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. Display
8. Search
9. Exit
Enter your choice: 2
Enter data to insert: 5
Inserted 5 at the end.
Doubly Linked List Operations:
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. Display
8. Search
9. Exit
Enter your choice: 7
List: NULL <-> 17 <-> 18 <-> 5 <-> NULL
Doubly Linked List Operations:
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. Display
8. Search
9. Exit
Enter your choice:


Doubly Linked List Operations:
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. Display
8. Search
9. Exit
Enter your choice: 4
Deleted 17 from the beginning.
Doubly Linked List Operations:
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. Display
8. Search
9. Exit
Enter your choice: 6
Enter position to delete: 1
Deleted 18 from the beginning.
Doubly Linked List Operations:
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. Display
8. Search
9. Exit
Enter your choice: 2


2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Display
8. Search
9. Exit
Enter your choice: 2
Enter data to insert: 45
Inserted 45 at the end.
Doubly Linked List Operations:
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. Display
8. Search
9. Exit
Enter your choice: 7
List: NULL <-> 99 <-> 5 <-> 16 <-> 45 <-> NULL
Doubly Linked List Operations:
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. Display
8. Search
9. Exit
Enter your choice: 8
Enter element to search: 45
Element 45 found at position 4.