Name- DEBOLINA PAUL

Reg no:- 24BBS0053

1. **Algorithm**

1. **Initialise:**
    - **Define an empty stack with a fixed size.**
    - **Set the `top` pointer to `-1` (indicating an empty stack).**
2. **Display Menu:**
    - **Show the menu with the options:**
        - 1. `PUSH`
        - 2. `POP`
        - 3. `Display`
        - 4. **Exit**
3. **Choice Input:**
    - **Take the user's input to choose an operation.**
4. **Perform Operation:**
    - **PUSH:**
        - **Check if the stack is full (`top == maxSize - 1`).**
        - **If full, print "Stack Overflow".**
        - **Otherwise, increment `top` and add the new element at `stack[top]`.**
    - **POP:**
        - **Check if the stack is empty (`top == -1`).**
        - **If empty, print "Stack Underflow".**
        - **Otherwise, print and remove the element at `stack[top]`, and decrement `top`.**
    - **Display:**
        - **Check if the stack is empty (`top == -1`).**
        - **If empty, print "Stack is empty".**
        - **Otherwise, print all elements from `stack[0]` to `stack[top]`.**

5. **Repeat:**
    - **Loop back to step 2 until the user chooses to exit.**

Pseudocode

```
BEGIN
  Initialize stack[MAX], top ← -1, maxSize ← MAX

  WHILE true DO
    PRINT "Menu:"
    PRINT "1. PUSH"
    PRINT "2. POP"
    PRINT "3. Display"
    PRINT "4. Exit"
    PRINT "Enter your choice: "
    READ choice

    SWITCH choice DO
      CASE 1:
        IF top == maxSize - 1 THEN
          PRINT "Stack Overflow"
        ELSE
          PRINT "Enter element to PUSH: "
          READ element
          top ← top + 1
          stack[top] ← element
          PRINT "Element pushed"
        ENDIF
        BREAK
```

```
    CASE 2:
        IF top == -1 THEN
            PRINT "Stack Underflow"
        ELSE
            PRINT "Popped element: ", stack[top]
            top ← top - 1
        ENDIF
        BREAK

    CASE 3:
        IF top == -1 THEN
            PRINT "Stack is empty"
        ELSE
            PRINT "Stack elements are: "
            FOR i ← 0 TO top DO
                PRINT stack[i]
            ENDFOR
        ENDIF
        BREAK

    CASE 4:
        PRINT "Exiting..."
        EXIT

    DEFAULT:
        PRINT "Invalid choice, please try again."
ENDSWITCH
```

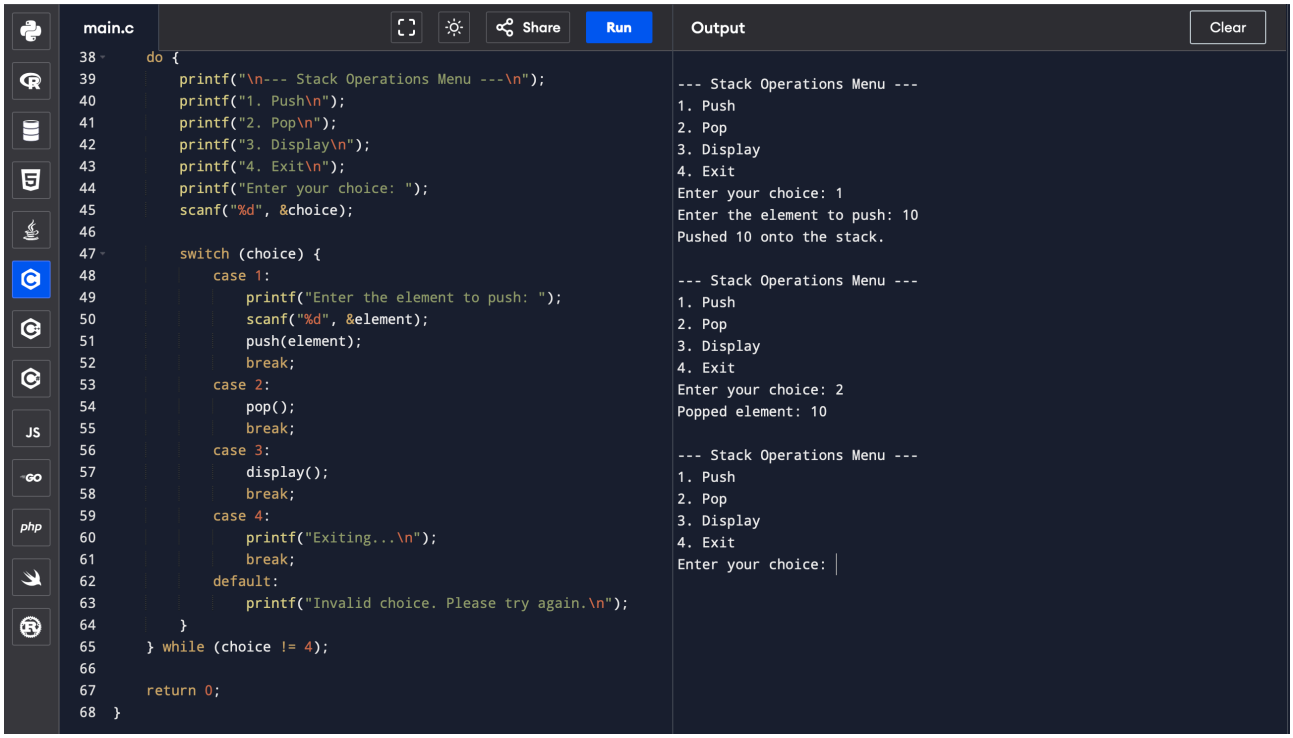
```
   ENDWHILE
END
```

C code

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int stack[MAX];
int top = -1;
void push(int element) {
   if (top == MAX - 1) {
      printf("Stack Overflow! Cannot push %d onto the stack.\n", element);
      return;
   }
   top++;
   stack[top] = element;
   printf("Pushed %d onto the stack.\n", element);
}
void pop() {
   if (top == -1) {
      printf("Stack Underflow! No elements to pop.\n");
      return;
   }
   int poppedElement = stack[top];
   top--;
   printf("Popped element: %d\n", poppedElement);
}
void display() {
   if (top == -1) {
```

```
        printf("Stack is empty.\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = top; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}
int main() {
    int choice, element;

    do {
        printf("\n--- Stack Operations 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 the element to push: ");
                scanf("%d", &element);
                push(element);
                break;
            case 2:
```

```
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }
} while (choice != 4);

return 0;
}
```



2. Algorithm

1. **Initialise:**
   - **Define a `queue` array, `front`, `rear`, and `size` to represent the queue structure.**
   - **Set `front = -1` and `rear = -1` to indicate an empty queue.**
2. **Enqueue (Insert):**
   - **Check if the queue is full:**
     - **If `(rear + 1) % size == front`, display "Queue is full."**
   - **Otherwise:**
     - **If the queue is empty (`front == -1`), set `front = 0`.**
     - **Increment `rear` using `(rear + 1) % size` to maintain circular indexing.**
     - **Add the new element at the `rear` position.**
3. **Dequeue (Delete):**
   - **Check if the queue is empty:**
     - **If `front == -1`, display "Queue is empty."**
   - **Otherwise:**
     - **Retrieve the element at the `front` position.**
     - **If `front == rear`, reset `front` and `rear` to `-1` (queue becomes empty).**
     - **Otherwise, increment `front` using `(front + 1) % size`.**
4. **Display:**
   - **Check if the queue is empty:**
     - **If `front == -1`, display "Queue is empty."**
   - **Otherwise:**
     - **Traverse the queue from `front` to `rear` using circular indexing.**
5. **Menu-Driven Program:**
   - **Display a menu with options for enqueue, dequeue, display, and exit.**
   - **Accept the user's choice and perform the corresponding operation.**
   - **Repeat until the user chooses to exit.**

Pseudocode

Start

```
Initialise queue[MAX], front = -1, rear = -1
size = MAX (define maximum size of the queue)

Function Enqueue(element):
    If (rear + 1) % size == front:
        Print "Queue is full"
        Return
    If front == -1:
        front = 0
    rear = (rear + 1) % size
    queue[rear] = element
    Print "Element inserted"

Function Dequeue():
    If front == -1:
        Print "Queue is empty"
        Return
    element = queue[front]
    If front == rear:
        front = -1
        rear = -1
    Else:
        front = (front + 1) % size
    Print "Element dequeued:", element

Function Display():
    If front == -1:
        Print "Queue is empty"
        Return
    i = front
    While True:
        Print queue[i]
        If i == rear:
            Break
        i = (i + 1) % size

Main:
    Repeat:
        Print "1. Enqueue 2. Dequeue 3. Display 4. Exit"
        Input choice
```

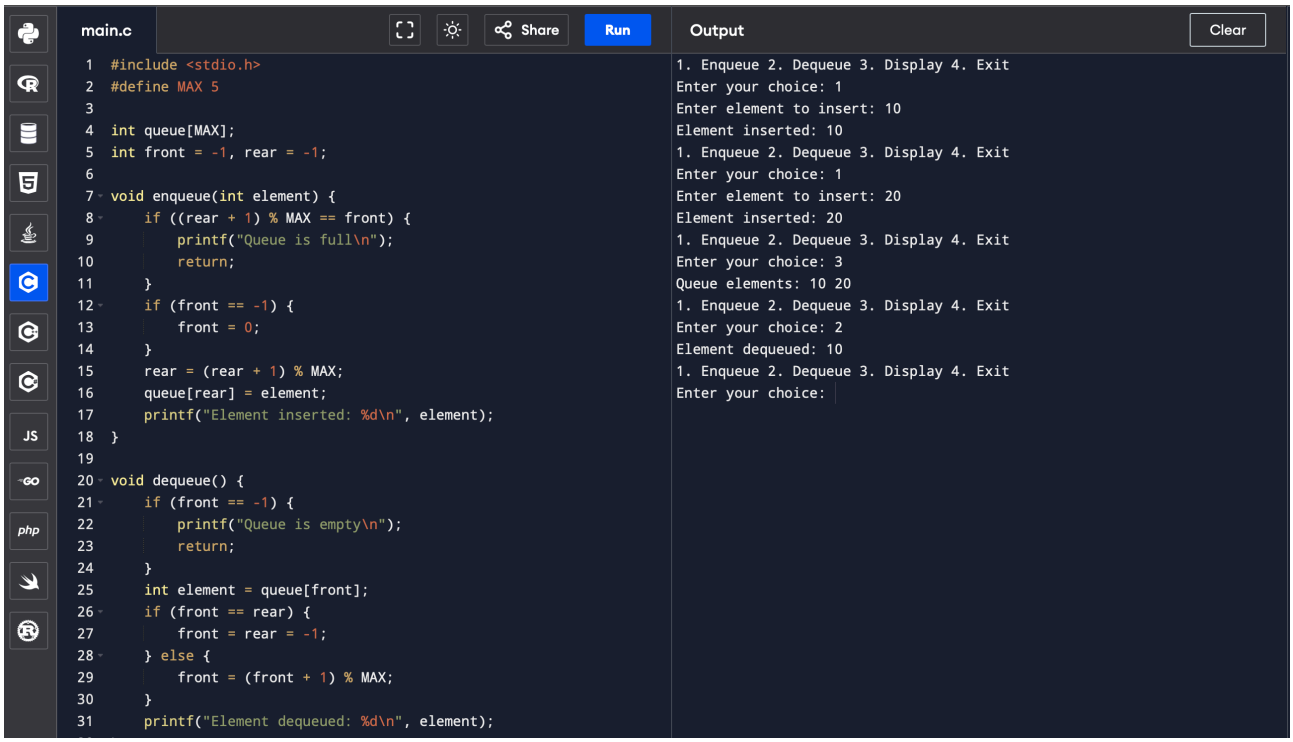
```
        If choice == 1:
            Input element
            Call Enqueue(element)
        Else If choice == 2:
            Call Dequeue()
        Else If choice == 3:
            Call Display()
        Else If choice == 4:
            Exit
        Else:
            Print "Invalid choice"
    Until choice == 4
End
```

C Code

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

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

void enqueue(int element) {
    if ((rear + 1) % MAX == front) {
        printf("Queue is full\n");
        return;
    }
    if (front == -1) {
        front = 0;
    }
    rear = (rear + 1) % MAX;
    queue[rear] = element;
    printf("Element inserted: %d\n", element);
}

void dequeue() {
    if (front == -1) {
        printf("Queue is empty\n");
        return;
    }
    int element = queue[front];
```

```
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % MAX;
    }
    printf("Element dequeued: %d\n", element);
}

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

int main() {
    int choice, element;
    do {
        printf("1. Enqueue 2. Dequeue 3. Display 4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter element to insert: ");
                scanf("%d", &element);
                enqueue(element);
                break;
            case 2:
                dequeue();
                break;
```

```
        case 3:
            display();
            break;
        case 4:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice\n");
    }
} while (choice != 4);
return 0;
```



}

3.Algorithm

1. Initialization:

- Define an array `queue[MAX]` to store queue elements.
- Initialize `front` and `rear` pointers to `-1` to represent an empty queue.

2. Enqueue Operation:

- Check if the queue is full: `(rear + 1) % MAX == front`.
    - If full, print "Queue Overflow."

- **Otherwise:**
    - **If the queue is initially empty (`front == -1`), set `front = rear = 0`.**
    - **Else, update `rear` to `(rear + 1) % MAX`.**
    - **Insert the new element at `queue[rear]`.**

3. Dequeue Operation:

- **Check if the queue is empty: `front == -1`.**
    - **If empty, print "Queue Underflow."**
- **Otherwise:**
    - **Retrieve the element at `queue[front]`.**
    - **If the queue has only one element (`front == rear`), set `front = rear = -1` (queue becomes empty).**
    - **Otherwise, update `front` to `(front + 1) % MAX`.**

4. Display Operation:

- **Check if the queue is empty: `front == -1`.**
    - **If empty, print "Queue is empty."**
- **Otherwise:**
    - **Start from `front` and iterate through the queue using `(index + 1) % MAX` until `index == rear`.**

5. Menu and User Interaction:

- **Display menu options:**
    - **1. Enqueue**
    - **2. Dequeue**
    - **3. Display**
    - **4. Exit**
- **Repeat operations until the user chooses to exit**

Pseudocode

BEGIN

```
Initialize queue[MAX], front ← -1, rear ← -1

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

  SWITCH choice DO
    CASE 1:
      IF (rear + 1) % MAX == front THEN
        PRINT "Queue Overflow"
      ELSE
        PRINT "Enter the element to enqueue: "
        READ element
        IF front == -1 THEN
          front ← 0
        ENDIF
        rear ← (rear + 1) % MAX
        queue[rear] ← element
        PRINT "Element enqueued."
      ENDIF
      BREAK

    CASE 2:
      IF front == -1 THEN
        PRINT "Queue Underflow"
      ELSE
        PRINT "Dequeued element: ", queue[front]
```

```
        IF front == rear THEN
            front ← -1
            rear ← -1
        ELSE
            front ← (front + 1) % MAX
        ENDIF
    ENDIF
    BREAK

  CASE 3:
    IF front == -1 THEN
        PRINT "Queue is empty"
    ELSE
        PRINT "Queue elements are: "
        index ← front
        WHILE true DO
            PRINT queue[index]
            IF index == rear THEN
                BREAK
            ENDIF
            index ← (index + 1) % MAX
        ENDWHILE
    ENDIF
    BREAK

  CASE 4:
    PRINT "Exiting program..."
    EXIT

  DEFAULT:
    PRINT "Invalid choice. Please try again."
ENDSWITCH
```

```
   ENDWHILE
END
```

C code

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int queue[MAX];
int front = -1, rear = -1;
void enqueue(int element) {
   if ((rear + 1) % MAX == front) {
      printf("Queue is full! Cannot enqueue %d.\n", element);
      return;
   }
   if (front == -1) { // First element to be added
      front = 0;
   }
   rear = (rear + 1) % MAX;
   queue[rear] = element;
   printf("Enqueued %d.\n", element);
}
void dequeue() {
   if (front == -1) {
      printf("Queue is empty! Cannot dequeue.\n");
      return;
   }
   int removedElement = queue[front];
   if (front == rear) { // Queue becomes empty after removal
      front = rear = -1;
   } else {
      front = (front + 1) % MAX;
   }
   printf("Dequeued element: %d.\n", removedElement);
```

```
}
void display() {
   if (front == -1) {
      printf("Queue is empty.\n");
      return;
   }
   printf("Queue elements: ");
   int i = front;
   while (1) {
      printf("%d ", queue[i]);
      if (i == rear) {
         break;
      }
      i = (i + 1) % MAX;
   }
   printf("\n");
}
int main() {
   int choice, element;

   do {
      printf("\n--- Circular Queue Operations ---\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 element to enqueue: ");
```

```
            scanf("%d", &element);
            enqueue(element);
            break;
        case 2:
            dequeue();
            break;
        case 3:
            display();
            break;
        case 4:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice! Please try again.\n");
    }
  } while (choice != 4);

  return 0;
}
```

main.c

Share Run

```
51          printf("\n--- Circular Queue Operations ---\n");
52          printf("1. Enqueue\n");
53          printf("2. Dequeue\n");
54          printf("3. Display\n");
55          printf("4. Exit\n");
56          printf("Enter your choice: ");
57          scanf("%d", &choice);
58
59          switch (choice) {
60              case 1:
61                  printf("Enter the element to enqueue: ");
62                  scanf("%d", &element);
63                  enqueue(element);
64                  break;
65              case 2:
66                  dequeue();
67                  break;
68              case 3:
69                  display();
70                  break;
71              case 4:
72                  printf("Exiting...\n");
73                  break;
74              default:
75                  printf("Invalid choice! Please try again.\n");
76          }
77      } while (choice != 4);
78
79      return 0;
80  }
81
```

Output

Clear

```
--- Circular Queue Operations ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the element to enqueue: 10
Enqueued 10.

--- Circular Queue Operations ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue elements: 10

--- Circular Queue Operations ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
Dequeued element: 10.

--- Circular Queue Operations ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
```

4. Algorithm

1. Insertion

1. **At the Beginning:**
    - **Create a new node.**
    - **Set the new node's `next` pointer to the current head.**
    - **Update the head to point to the new node.**
2. **At the End:**
    - **Create a new node.**
    - **Traverse the list to find the last node.**
    - **Set the last node's `next` pointer to the new node.**
3. **At a Given Position:**
    - **Create a new node.**
    - **Traverse the list to the `(position - 1)` node.**
    - **Set the new node's `next` pointer to the next node of the `(position - 1)` node.**
    - **Update the `(position - 1)` node's `next` pointer to the new node.**

2. Deletion

1. **At the Beginning:**
    - **Check if the list is empty.**
    - **Update the head pointer to the next node of the current head.**
    - **Free the memory of the deleted node.**
2. **At the End:**
    - **Check if the list is empty.**
    - **Traverse to the second-last node.**
    - **Update its `next` pointer to `NULL`.**
    - **Free the memory of the last node.**
3. **At a Given Position:**
    - **Traverse the list to the `(position - 1)` node.**
    - **Update its `next` pointer to skip the next node.**
    - **Free the memory of the deleted node.**

3. Search

- **Traverse the list while comparing each node's data with the target element.**
- **If found, return the position; otherwise, return a "not found" message.**

4. Display

- **Traverse the list and print the data of each node.**

Pseudocode

```
Start
Define a Node structure with data and next pointer.

Function InsertAtBeginning(data):
    Create a new node with the given data.
    Set newNode->next = head.
    Update head = newNode.

Function InsertAtEnd(data):
    Create a new node with the given data.
    If head is NULL:
        Set head = newNode.
        Return.
    Traverse to the last node.
    Set lastNode->next = newNode.

Function InsertAtPosition(data, position):
    Create a new node with the given data.
    Traverse to the (position - 1) node.
    Set newNode->next = currentNode->next.
    Update currentNode->next = newNode.

Function DeleteAtBeginning():
    If head is NULL:
        Print "List is empty."
```

Return.
Update head = head->next.
Free the deleted node.

Function DeleteAtEnd():
If head is NULL:
Print "List is empty."
Return.
Traverse to the second-last node.
Update secondLastNode->next = NULL.
Free the last node.

Function DeleteAtPosition(position):
Traverse to the (position - 1) node.
Update currentNode->next = currentNode->next->next.
Free the deleted node.

Function Search(data):
Traverse the list and compare each node's data with the target.
If found, return the position.
Else, print "Element not found."

Function Display():
If head is NULL:
Print "List is empty."
Return.
Traverse the list and print each node's data.

Main:
Repeat:
Display menu for insertion, deletion, search, and display.
Input choice.
Call the corresponding function based on choice.
Until choice is Exit.
End

C Code

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

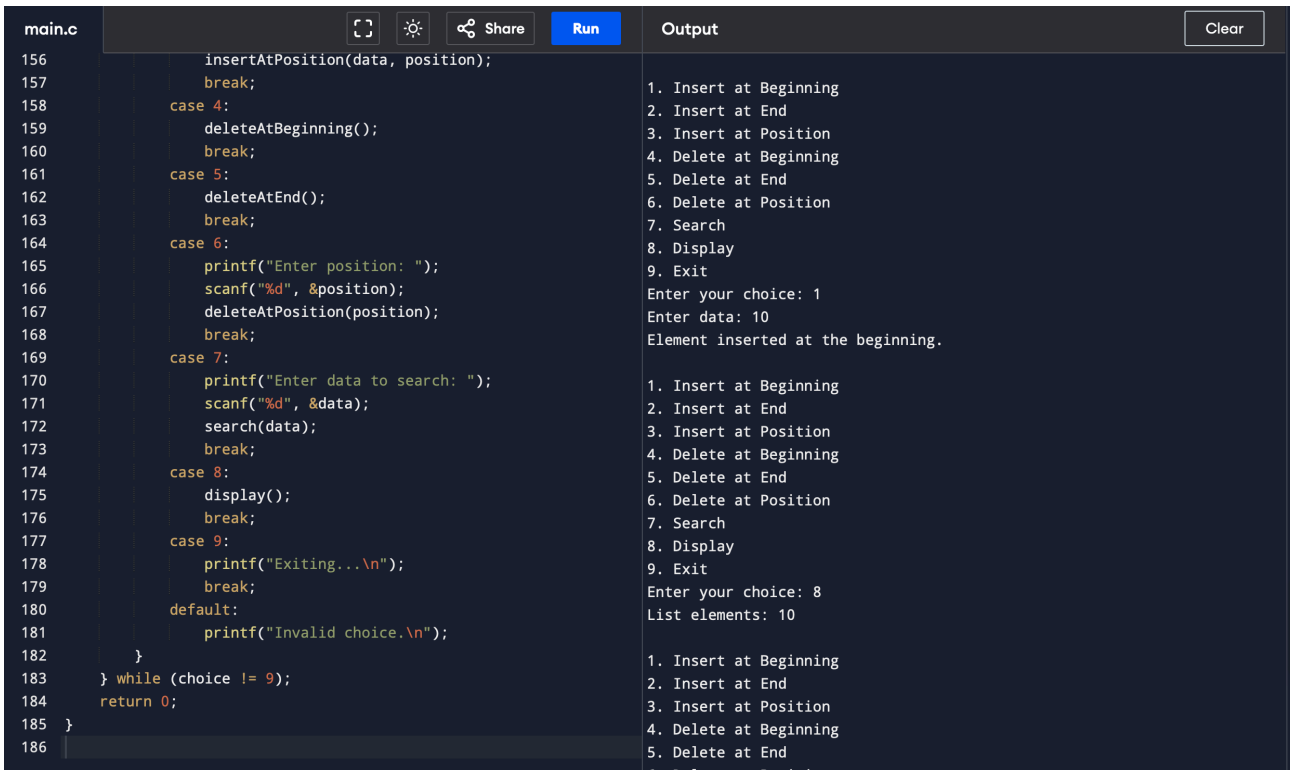
```
struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL;

void insertAtBeginning(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = head;
    head = newNode;
    printf("Element inserted at the beginning.\n");
}

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

void insertAtPosition(int data, int position) {
    if (position == 1) {
        insertAtBeginning(data);
        return;
    }
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    struct Node* temp = head;
```

```
    for (int i = 1; i < position - 1; i++) {
        if (temp == NULL) {
            printf("Invalid position.\n");
            free(newNode);
            return;
        }
        temp = temp->next;
    }
    newNode->next = temp->next;
    temp->next = newNode;
    printf("Element inserted at position %d.\n", position);
}

void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;
    free(temp);
    printf("Element deleted from the beginning.\n");
}

void deleteAtEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if (head->next == NULL) {
        free(head);
        head = NULL;
        printf("Element deleted from the end.\n");
        return;
    }
    struct Node* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
```

```
    temp->next = NULL;
    printf("Element deleted from the end.\n");
}

void deleteAtPosition(int position) {
    if (position == 1) {
        deleteAtBeginning();
        return;
    }
    struct Node* temp = head;
    for (int i = 1; i < position - 1; i++) {
        if (temp == NULL || temp->next == NULL) {
            printf("Invalid position.\n");
            return;
        }
        temp = temp->next;
    }
    struct Node* nodeToDelete = temp->next;
    temp->next = nodeToDelete->next;
    free(nodeToDelete);
    printf("Element deleted from position %d.\n", position);
}

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

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

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

int main() {
    int choice, data, position;
    do {
        printf("\n1. Insert at Beginning\n2. Insert at End\n3. Insert at Position\n");
        printf("4. Delete at Beginning\n5. Delete at End\n6. Delete at Position\n");
        printf("7. Search\n8. Display\n9. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data: ");
                scanf("%d", &data);
                insertAtBeginning(data);
                break;
            case 2:
                printf("Enter data: ");
                scanf("%d", &data);
                insertAtEnd(data);
                break;
            case 3:
                printf("Enter data and position: ");
                scanf("%d %d", &data, &position);
                insertAtPosition(data, position);
                break;
            case 4:
                deleteAtBeginning();
                break;
            case 5:
                deleteAtEnd();
                break;
```

```
            case 6:
                printf("Enter position: ");
                scanf("%d", &position);
                deleteAtPosition(position);
                break;
            case 7:
                printf("Enter data to search: ");
                scanf("%d", &data);
                search(data);
                break;
            case 8:
                display();
                break;
            case 9:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (choice != 9);
    return 0;
}
```



5. Algorithm

1. Initialisation:

- **Define a structure `Node` with fields:**
  - **`data` to store the value.**
  - **`prev` to store the address of the previous node.**
  - **`next` to store the address of the next node.**
- **Initialise `head = NULL`.**

2. Insertion Operations:

- **Beginning:**
  - **Create a new node.**
  - **Set its `next` to the current head and `prev` to NULL.**
  - **Update the `prev` of the old head (if it exists) to point to the new node.**
  - **Update `head` to the new node.**
- **End:**
  - **Create a new node.**
  - **Traverse the list to the last node.**
  - **Update the `next` of the last node and set the new node's `prev` to the last node.**
- **At a Given Position:**
  - **Traverse to the specified position.**
  - **Insert the new node by updating the `next` and `prev` pointers of the neighbouring nodes.**

3. Deletion Operations:

- **Beginning:**
  - **If `head` is NULL, print "List is empty."**
  - **Update `head` to `head->next`.**
  - **Set the `prev` of the new head to NULL.**
- **End:**
  - **Traverse to the last node.**
  - **Update the `next` of the second last node to NULL.**
- **At a Given Position:**

- Traverse to the specified position.
- Update the `next` and `prev` pointers of the neighbouring nodes to bypass the node to be deleted.

4. Search Operation:

- Traverse the list, comparing each node's `data` with the search key.
- If a match is found, print its position; otherwise, print "Element not found."

5. Menu and User Interaction:

- Display the menu with options for insertion, deletion, search, and exit.
- Repeat operations until the user chooses to exit.

PSEUDOCODE

```
BEGIN
Initialize head ← NULL

WHILE true DO
  PRINT "Menu:"
  PRINT "1. Insert at Beginning"
  PRINT "2. Insert at End"
  PRINT "3. Insert at a Position"
  PRINT "4. Delete from Beginning"
  PRINT "5. Delete from End"
  PRINT "6. Delete from a Position"
  PRINT "7. Search for an Element"
  PRINT "8. Exit"
  PRINT "Enter your choice: "
  READ choice

  SWITCH choice DO
    CASE 1:
      CALL insertBeginning()
      BREAK
```

```
CASE 2:
   CALL insertEnd()
   BREAK

CASE 3:
   PRINT "Enter position: "
   READ pos
   CALL insertAtPosition(pos)
   BREAK

CASE 4:
   CALL deleteBeginning()
   BREAK

CASE 5:
   CALL deleteEnd()
   BREAK

CASE 6:
   PRINT "Enter position: "
   READ pos
   CALL deleteAtPosition(pos)
   BREAK

CASE 7:
   PRINT "Enter element to search: "
   READ element
   CALL search(element)
   BREAK

CASE 8:
   PRINT "Exiting program..."
```

```
            EXIT

        DEFAULT:
            PRINT "Invalid choice. Please try again."
    ENDSWITCH
  ENDWHILE
END
```

C code

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

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

struct Node* head = NULL;
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
void insertAtBeginning(int data) {
    struct 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(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;
    }
    printf("Inserted %d at the end.\n", data);
}
void insertAtPosition(int data, int position) {
    struct Node* newNode = createNode(data);
    if (position == 1) {
        insertAtBeginning(data);
        return;
    }
    struct Node* temp = head;
    for (int i = 1; i < position - 1; i++) {
        if (temp == NULL) {
            printf("Invalid position.\n");
            free(newNode);
            return;
        }
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Invalid position.\n");
        free(newNode);
        return;
    }
    newNode->next = temp->next;
    if (temp->next != NULL) {
        temp->next->prev = newNode;
    }
    temp->next = newNode;
    newNode->prev = temp;
    printf("Inserted %d at position %d.\n", data, position);
}
```

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

void deleteAtEnd() {
    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("Deleted %d from the end.\n", temp->data);
    free(temp);
}
void deleteAtPosition(int position) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if (position == 1) {
        deleteAtBeginning();
        return;
    }
```

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

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

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

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

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

    do {
        printf("\n--- Doubly 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 at Beginning\n");
        printf("5. Delete at End\n");
        printf("6. Delete at 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 data to insert at beginning: ");
                scanf("%d", &data);
                insertAtBeginning(data);
                break;
            case 2:
                printf("Enter data to insert at end: ");
                scanf("%d", &data);
                insertAtEnd(data);
                break;
            case 3:
                printf("Enter data and position: ");
                scanf("%d %d", &data, &position);
                insertAtPosition(data, position);
```

```
            break;
        case 4:
            deleteAtBeginning();
            break;
        case 5:
            deleteAtEnd();
            break;
        case 6:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(position);
            break;
        case 7:
            printf("Enter element to search: ");
            scanf("%d", &data);
            search(data);
            break;
        case 8:
            display();
            break;
        case 9:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }
  } while (choice != 9);

  return 0;
}
```

main.c

Share Run

```c

void deleteAtEnd() {
    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("Deleted %d from the end.\n", temp->data);
    free(temp);
}
void deleteAtPosition(int position) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if (position == 1) {
        deleteAtBeginning();
        return;
    }
    struct Node* temp = head;
    for (int i = 1; i < position; i++) {
        if (temp == NULL) {
            printf("Invalid position.\n");
```

Output

Clear

```
--- Doubly Linked List Operations ---
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 1
Enter data to insert at beginning: 10
Inserted 10 at the beginning.

--- Doubly Linked List Operations ---
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 8
List elements: 10

--- Doubly Linked List Operations ---
1. Insert at Beginning
2. Insert at End
```