DSA LAB ASSESSMENT 1

NAME : Ankur Sinha

REG NO. : 24BBS0102

Q1. Write a menu driven program to implement the following operations on stack.

a. PUSH()

b. POP()

c. Display()

Algorithm:

Define a Stack:

- A structure Stack with:
    - data: An array to store stack elements.
    - top: An integer to track the index of the top element.
    - maxSize: The maximum capacity of the stack.

Push Operation:

- Input: Stack s, element data.
- Steps:
    1. Check if the stack is full (top == maxSize - 1).
        - If yes, print "Stack Overflow" and return.
    2. Increment top.
    3. Add data to s.data[top].

Pop Operation:

- Input: Stack s.
- Steps:
    1. Check if the stack is empty (top == -1).
        - If yes, print "Stack Underflow" and return.
    2. Retrieve the element at s.data[top].
    3. Decrement top.
    4. Return the retrieved element.

Display Stack:

- Input: Stack s.
- Steps:
    1. If the stack is empty (top == -1), print "Stack is empty".

2. Otherwise, print elements from s.data[top] to s.data[0].

Main Function:

1. Initialize a stack s with maxSize.
2. Provide menu options:
    - 1: Push an element.
    - 2: Pop an element.
    - 3: Display the stack.
3. Perform the chosen operation and display the updated stack.

Code:

```
#include <stdio.h>
#define MAX 100

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

void push(int item) {
   if (top == MAX - 1) {
      printf("Stack overflow. Cannot push %d\n", item);
      return;
   }
   stack[++top] = item;
   printf("%d pushed to stack.\n", item);
}

void pop() {
   if (top == -1) {
      printf("Stack underflow. Cannot pop.\n");
```

```
        return;
    }
    printf("%d popped from stack.\n", stack[top--]);
}

void display() {
    if (top == -1) {
        printf("Stack is empty.\n");
        return;
    }
    printf("Stack elements are:\n");
    for (int i = top; i >= 0; i--) {
        printf("%d\n", stack[i]);
    }
}

int main() {
    int choice, item;
    while (1) {
        printf("\nMenu:\n1. PUSH\n2. POP\n3. Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the item to push: ");
                scanf("%d", &item);
                push(item);
                break;
            case 2:
                pop();
```

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

Output:

```
Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter the item to push: 100
100 pushed to stack.

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter the item to push: 98
98 pushed to stack.

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter the item to push: 94
94 pushed to stack.

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 3
Stack elements are:
94
98
100
```

Q2. Write a menu driven program to implement the following operations on Queue:

a. Enqueue()

b. Dequeue()

c. Display()

ALGORITHM:

Define a Queue:

- A structure Queue with:
    - data: An array to store queue elements.
    - front: An integer to track the index of the first element.
    - rear: An integer to track the index of the last element.
    - maxSize: The maximum capacity of the queue.

Enqueue Operation:

- **Input:** Queue q, element data.
- **Steps:**
    1. Check if the queue is full (rear == maxSize - 1).
        - If yes, print "Queue Overflow" and return.
    2. If front == -1, set front = 0.
    3. Increment rear.
    4. Add data to q.data[rear].

Dequeue Operation:

- **Input:** Queue q.
- **Steps:**
    1. Check if the queue is empty (front == -1 or front > rear).
        - If yes, print "Queue Underflow" and return.

2. Retrieve the element at q.data[front].
3. Increment front.
4. If front > rear, reset front and rear to -1.
5. Return the retrieved element.

Display Queue:

- **Input:** Queue q.
- **Steps:**
    1. If the queue is empty (front == -1), print "Queue is empty".
    2. Otherwise, print elements from q.data[front] to q.data[rear].

Main Function:

1. Initialize a queue q with maxSize.
2. Provide menu options:
    - 1: Enqueue an element.
    - 2: Dequeue an element.
    - 3: Display the queue.
3. Perform the chosen operation and display the updated queue.

Code:

```
#include <stdio.h>
#define MAX 100

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

void enqueue(int item) {
   if (rear == MAX - 1) {
      printf("Queue overflow. Cannot enqueue %d\n", item);
      return;
   }
```

```
    if (front == -1) {
        front = 0;
    }
    queue[++rear] = item;
    printf("%d enqueued to queue.\n", item);
}

void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue underflow. Cannot dequeue.\n");
        return;
    }
    printf("%d dequeued from queue.\n", queue[front++]);
    if (front > rear) {
        front = rear = -1;
    }
}

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

```
int main() {
    int choice, item;
    while (1) {
        printf("\nMenu:\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the item to enqueue: ");
                scanf("%d", &item);
                enqueue(item);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}
```

Output :

```
Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the item to enqueue: 12
12 enqueued to queue.

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the item to enqueue: 54
54 enqueued to queue.

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the item to enqueue: 65
65 enqueued to queue.

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue elements are:
12
54
65

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
12 dequeued from queue.
```

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

a. Enqueue()

b. Dequeue()

c. Disaply()

ALGORITHM:

Define a Circular Queue:

- A structure CircularQueue with:
    - data: An array to store queue elements.
    - front: An integer to track the index of the first element.
    - rear: An integer to track the index of the last element.
    - maxSize: The maximum capacity of the queue.

Enqueue Operation:

- **Input:** CircularQueue cq, element data.
- **Steps:**
    1. Check if the queue is full ((rear + 1) % maxSize == front).
        - If yes, print "Queue Overflow" and return.
    2. If front == -1, set front = 0.
    3. Increment rear using rear = (rear + 1) % maxSize.
    4. Add data to cq.data[rear].

Dequeue Operation:

- **Input:** CircularQueue cq.
- **Steps:**
    1. Check if the queue is empty (front == -1).
        - If yes, print "Queue Underflow" and return.
    2. Retrieve the element at cq.data[front].
    3. If front == rear, reset front and rear to -1.
    4. Otherwise, increment front using front = (front + 1) % maxSize.
    5. Return the retrieved element.

Display Circular Queue:

- **Input:** CircularQueue cq.

- **Steps:**
    1. If the queue is empty (front == -1), print "Queue is empty".
    2. Otherwise:
        - Start from cq.data[front] and traverse circularly until rear.

Main Function:

1. Initialize a circular queue cq with maxSize.
2. Provide menu options:
    - 1: Enqueue an element.
    - 2: Dequeue an element.
    - 3: Display the queue.
3. Perform the chosen operation and display the updated queue.

Code :

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

#define SIZE 5

typedef struct {
   int items[SIZE];
   int front, rear;
} CircularQueue;

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

int isFull(CircularQueue *q) {
   return (q->front == 0 && q->rear == SIZE - 1) || (q->rear == (q->front - 1) % (SIZE - 1));
```

```
}

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

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

    if (q->front == -1) {
        q->front = 0;
        q->rear = 0;
    } else if (q->rear == SIZE - 1 && q->front != 0) {
        q->rear = 0;
    } else {
        q->rear++;
    }

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

void dequeue(CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return;
```

```
    }

    int data = q->items[q->front];
    if (q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    } else if (q->front == SIZE - 1) {
        q->front = 0;
    } else {
        q->front++;
    }

    printf("%d dequeued from the queue.\n", data);
}

void display(CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Circular Queue elements: ");
    if (q->rear >= q->front) {
        for (int i = q->front; i <= q->rear; i++) {
            printf("%d ", q->items[i]);
        }
    } else {
        for (int i = q->front; i < SIZE; i++) {
            printf("%d ", q->items[i]);
```

```
        }
        for (int i = 0; i <= q->rear; i++) {
            printf("%d ", q->items[i]);
        }
    }
    printf("\n");
}

int main() {
    CircularQueue q;
    initQueue(&q);

    int choice, value;

    while (1) {
        printf("\nCircular 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 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! Please try again.\n");
        }
    }

    return 0;
}
```

Output :

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 1
1 enqueued to the queue.

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 2
2 enqueued to the queue.

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 3
3 enqueued to the queue.

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 4
4 enqueued to the queue.

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

Q4. 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

Define a Node:

- Create a structure node with two fields:
  - data: Stores the value of the node.

- next: Points to the next node in the list.

Insert at Beginning (insertAtBeg):

- Input: start (head of the list), data (data to be inserted).
- Allocate memory for a new node.
- Set the new node's data to the input data.
- Set the new node's next to point to the current start.
- Update start to point to the new node.
- Return the updated start.

Insert at End (insertAtEnd):

- Input: start (head of the list), data (data to be inserted).
- Create a new node with the input data.
- Traverse the list to find the last node (node where next is NULL).
- Set the last node's next to the new node.
- Set the new node's next to NULL.
- Return the updated start.

Insert at Position (insertAtPos):

- Input: start (head of the list), data (data to be inserted), pos (position).
- If the position is 0, call insertAtBeg to insert at the beginning and return.
- Traverse the list to find the node at position pos-1.
- If the node is NULL, print an error message and return.
- Insert the new node after the node at pos-1 by updating the pointers.
- Return the updated start.

Delete at Beginning (deleteAtBeg):

- Input: start (head of the list).
- If the list is empty (start == NULL), print an error message and return.
- Store the current start in a temporary pointer.
- Set start to the next node (start->next).
- Free the memory of the temporary node.
- Return the updated start.

Delete at End (deleteAtEnd):

- Input: start (head of the list).
- If the list is empty (start == NULL), print an error message and return.
- If the list has only one node (start->next == NULL), free the node and set start to NULL.
- Otherwise, traverse the list to find the second last node.
- Set the second last node's next to NULL and free the last node.
- Return the updated start.

Delete at Position (deleteAtPos):

- Input: start (head of the list), pos (position of the node to delete).
- If the list is empty (start == NULL), print an error message and return.
- If the position is 0, call deleteAtBeg to delete the first node and return.
- Traverse the list to find the node at position pos.
- If the node is NULL, print an error message and return.
- Update the next pointer of the previous node to skip the node to be deleted.
- Free the memory of the deleted node.
- Return the updated start.

Search Operation:

- Input: start (head of the list), data (data to search).
- Traverse the list, checking each node's data.
- If the data is found, print the position and return.
- If the end of the list is reached without finding the data, print an error message.

Display the List:

- Input: start (head of the list).
- Traverse the list, printing each node's data.
- End the display with NULL to indicate the end of the list.

Main Function:

- Initialize start as NULL.
- Provide a menu of operations:
    - 1: Insert at beginning.
    - 2: Insert at end.
    - 3: Insert at a specific position.
    - 4: Delete the first node.
    - 5: Delete the last node.
    - 6: Delete at a specific position.
    - 7: Search for an element.
- Execute the corresponding function based on user input.
- Display the updated list after each operation.
- Allow the user to continue or exit based on input.

Code :

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

   int data;

   struct Node* next;

};




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

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

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

void deleteAtBeginning(struct Node** head);

void deleteAtEnd(struct Node** head);
```

```
void deleteAtPosition(struct Node** head, int position);
void searchElement(struct Node* head, int key);
void displayList(struct Node* head);

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

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

        switch (choice) {
            case 1:
                printf("Enter data to insert at the beginning: ");
                scanf("%d", &data);
                insertAtBeginning(&head, data);
                break;
            case 2:
```

```
            printf("Enter data to insert at the end: ");
            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:
            deleteAtBeginning(&head);
            break;
        case 5:
            deleteAtEnd(&head);
            break;
        case 6:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(&head, position);
            break;
        case 7:
            printf("Enter element to search: ");
            scanf("%d", &data);
            searchElement(head, data);
            break;
        case 8:
            displayList(head);
```

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

   return 0;
}

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

void insertAtEnd(struct Node** head, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   if (*head == NULL) {
      *head = newNode;
   } else {
      struct Node* temp = *head;
      while (temp->next != NULL) {
         temp = temp->next;
```

```
        }
        temp->next = newNode;
    }
    printf("Node inserted at the end.\n");
}

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

void deleteAtBeginning(struct Node** head) {
```

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

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

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

void searchElement(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);
}

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

Output :

```
Menu:
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 Element
8. Display List
9. Exit
Enter your choice: 1
Enter data to insert at the beginning: 67
Node inserted at the beginning.

Menu:
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 Element
8. Display List
9. Exit
Enter your choice: 1
Enter data to insert at the beginning: 87
Node inserted at the beginning.

Menu:
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 Element
8. Display List
9. Exit
Enter your choice: 8
Linked List: 87 -> 67 -> NULL
```

Q 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:

Define a Node:

- A node structure with three fields:
  - data: Stores the value of the node.
  - prev: Points to the previous node in the list.
  - next: Points to the next node in the list.

Create a New Node (createNewNode):

- Allocate memory for a new node.
- Set the node's data, prev, and next to the provided values (prev = NULL, next = NULL).
- Return the new node.

Insert at Beginning (insertAtBeg):

- Input: start (head of the list), tail (tail of the list), data (data to insert).
- Allocate memory for a new node.
- If the list is empty (start == NULL), set both start and tail to the new node.
- Otherwise, set the new node's next to start, and update the prev pointer of the current start to point to the new node.
- Set start to the new node.

Insert at End (insertAtEnd):

- Input: start (head of the list), tail (tail of the list), data (data to insert).
- Allocate memory for a new node.
- If the list is empty (start == NULL), set both start and tail to the new node.
- Otherwise, set the current tail's next to the new node and the new node's prev to the current tail.

- Set tail to the new node.

Insert at Position (insertAtPos):

- Input: start (head of the list), tail (tail of the list), data (data to insert), pos (position).
- If the position is 0, call insertAtBeg to insert at the beginning and return.
- Traverse the list until reaching the node at position pos-1.
- If the node is NULL, print an error and free the new node.
- Insert the new node after the node at position pos-1, updating the prev and next pointers of adjacent nodes.

Delete at Beginning (deleteAtBeg):

- Input: start (head of the list), tail (tail of the list).
- If the list is empty (start == NULL), print an error and return.
- Otherwise, set start to the next node, and update the prev pointer of the new start to NULL.
- If the list becomes empty (start == NULL), set tail to NULL.
- Free the old start node.

Delete at End (deleteAtEnd):

- Input: start (head of the list), tail (tail of the list).
- If the list is empty (tail == NULL), print an error and return.
- If the list has only one node (start == tail), set both start and tail to NULL.
- Otherwise, set tail to the previous node and set its next pointer to NULL.
- Free the old tail node.

Delete at Position (deleteAtPos):

- Input: start (head of the list), tail (tail of the list), pos (position).
- If the list is empty (start == NULL), print an error and return.
- If the position is 0, call deleteAtBeg to delete the first node and return.
- Traverse the list until reaching the node at position pos.
- If the node is NULL, print an error.
- If the node is the first (start), delete it using deleteAtBeg.
- If the node is the last (tail), delete it using deleteAtEnd.
- Otherwise, update the prev and next pointers of adjacent nodes and free the current node.

Search Operation (search):

- Input: start (head of the list), data (data to search).
- Traverse the list, checking each node's data.
- If the data is found, print the position and return.
- If the end of the list is reached without finding the data, print an error.

Display the List (display):

- Input: start (head of the list).
- Traverse the list and print each node's data from start to tail.
- Also display the list in reverse order, starting from tail to start.

Main Function:

- Initialize start and tail as NULL.
- Provide a menu with options:
    - 1: Insert at beginning.
    - 2: Insert at end.
    - 3: Insert at a specific position.
    - 4: Delete the first node.

    - 5: Delete the last node.
    - 6: Delete at a specific position.
    - 7: Search for an element.
- Execute the corresponding function based on user input.
- Display the updated list after each operation.
- Allow the user to continue or exit based on input.

Code :

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

   int data;

   struct Node* prev;

   struct Node* next;

};


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

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

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

void deleteAtBeginning(struct Node** head);

void deleteAtEnd(struct Node** head);

void deleteAtPosition(struct Node** head, int position);

void searchElement(struct Node* head, int key);

void displayList(struct Node* head);


int main() {

   struct Node* head = NULL;

   int choice, data, position;


   while (1) {
```

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

switch (choice) {
   case 1:
      printf("Enter data to insert at the beginning: ");
      scanf("%d", &data);
      insertAtBeginning(&head, data);
      break;
   case 2:
      printf("Enter data to insert at the end: ");
      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:
            deleteAtBeginning(&head);
            break;
        case 5:
            deleteAtEnd(&head);
            break;
        case 6:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(&head, position);
            break;
        case 7:
            printf("Enter element to search: ");
            scanf("%d", &data);
            searchElement(head, data);
            break;
        case 8:
            displayList(head);
            break;
        case 9:
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
    }
}

return 0;
```

```
}

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

void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
```

```
    printf("Node inserted at the end.\n");
}

void insertAtPosition(struct Node** head, int data, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    if (position == 1) {
        newNode->next = *head;
        newNode->prev = NULL;
        if (*head != NULL) {
            (*head)->prev = newNode;
        }
        *head = newNode;
    } else {
        struct Node* temp = *head;
        for (int i = 1; i < position - 1 && temp != NULL; i++) {
            temp = temp->next;
        }
        if (temp == NULL) {
            printf("Invalid position!\n");
            free(newNode);
            return;
        }
        newNode->next = temp->next;
        newNode->prev = temp;
        if (temp->next != NULL) {
            temp->next->prev = newNode;
        }
        temp->next = newNode;
```

```
    }
    printf("Node inserted at position %d.\n", position);
}

void deleteAtBeginning(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);
    printf("Node deleted from the beginning.\n");
}

void deleteAtEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = *head;
    if (temp->next == NULL) {
        *head = NULL;
    } else {
        while (temp->next != NULL) {
            temp = temp->next;
```

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

void deleteAtPosition(struct Node** head, int position) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = *head;
    if (position == 1) {
        *head = (*head)->next;
        if (*head != NULL) {
            (*head)->prev = NULL;
        }
        free(temp);
    } else {
        for (int i = 1; i < position && temp != NULL; i++) {
            temp = temp->next;
        }
        if (temp == NULL) {
            printf("Invalid position!\n");
            return;
        }
        temp->prev->next = temp->next;
        if (temp->next != NULL) {
```

```
            temp->next->prev = temp->prev;
        }
        free(temp);
    }
    printf("Node deleted at position %d.\n", position);
}

void searchElement(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);
}

void displayList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = head;
    printf("Doubly Linked List: ");
    while (temp != NULL) {
```

```
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Output :

```
Menu:
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 Element
8. Display List
9. Exit
Enter your choice: 1
Enter data to insert at the beginning: 67
Node inserted at the beginning.

Menu:
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 Element
8. Display List
9. Exit
Enter your choice: 1
Enter data to insert at the beginning: 9876
Node inserted at the beginning.

Menu:
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 Element
8. Display List
9. Exit
Enter your choice: 8
Doubly Linked List: 9876 <-> 67 <-> NULL
```