# DSA: ASSESMENT-1

Mohammad Umar Qureshi

24BBS0154

1.ALGORITHM

1. Initialization

Declaration of an array stack and storing elements in it.

Initializing a variable top to -1 which indicates an empty stack

2. Menu-Driven Loop

Repeat until the user chooses to exit

Display the menu with options:

1. Push

2.Pop

3.Display

4.Exit

Takes user choice as an input

According to the choice:

1.Push:

* If the stack is full (overflow), display an error message.

* Otherwise, increment top, store the data in stack[top], and display a success message.

2. Pop:

* If the stack is empty (underflow), display an error message.

* Otherwise, retrieve the data from stack[top], decrement top, and display the popped data.

3.Display:

* If the stack is empty, display an error message.

* Otherwise, iterate through the stack from top to 0 and display each element.

* Exit:

* Terminate the program.

3. End

CODE:

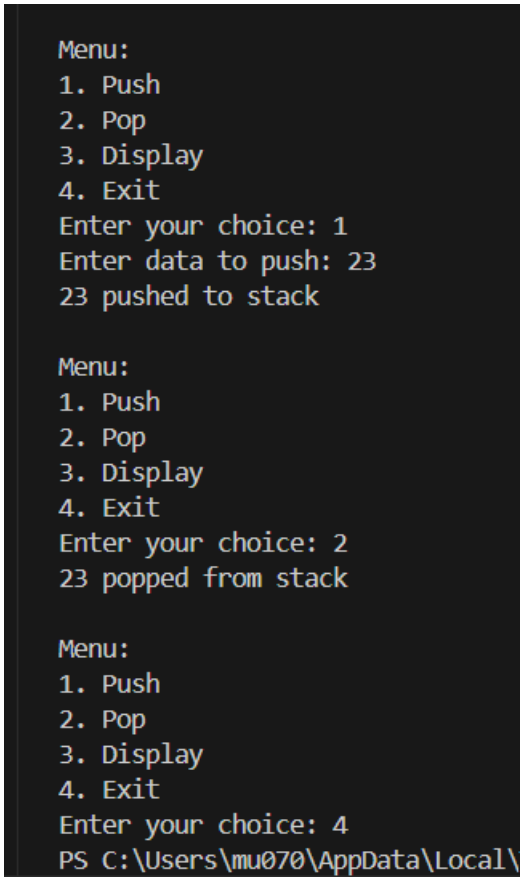
```
#include <stdio.h>
#include <stdlib.h>
#define max 100
int stack[max];
int top = -1;
int isEmpty() {
    return (top == -1);
}
int isFull() {
    return (top == max - 1);
}
void push(int data) {
    if (isFull()) {
        printf("Stack overflow!\n");
    } else {
        stack[++top] = data;
        printf("%d pushed to stack\n", data);
    }
}
int pop() {
    if (isEmpty()) {
        printf("Stack underflow!\n");
        return -1;
    } else {
        return stack[top--];
    }
}
```

```
void display() {
   if (isEmpty()) {
       printf("Stack is empty!\n");
    } else {
       printf("Stack elements:\n");
       for (int i = top; i >= 0; i--) {
           printf("%d ", stack[i]);
       }
       printf("\n");
    }
}
int main() {
    int choice, data;
    while (1) {
       printf("\nMenu:\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 data to push: ");
               scanf("%d", &data);
               push(data);
               break;
           case 2:
               data = pop();
```

```
        if (data != -1) {
                printf("%d popped from stack\n", data);
            }
            break;
        case 3:
            display();
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice!\n");
      }
   }

   return 0;
}
```

1.OUTPUT:




2. ALGORITHM:

1. Initialization

* Declare an array queue to store elements of the queue.

* Initialize variables front and rear to -1, indicating an empty queue.

2. Menu-Driven Loop

* Repeat until the user chooses to exit:

  * Display the menu with options:

    * * Enqueue

    * * Dequeue

    * * Display

    * * Exit

  * Get the user's choice.

  * Based on the choice:

    * Enqueue:

* If the queue is full (overflow), display an error message.

* Otherwise, if the queue is empty, set front to 0.

* Increment rear, store the data in queue[rear], and display a success message.

* Dequeue:

* If the queue is empty (underflow), display an error message.

* Otherwise, retrieve the data from queue[front].

* If front is equal to rear, set both to -1 to indicate an empty queue.

* Otherwise, increment front.

* Display the dequeued data.

* Display:

* If the queue is empty, display an error message.

* Otherwise, iterate through the queue from front to rear and display each element.

* Exit:

* Terminate the program.

2.

CODE:

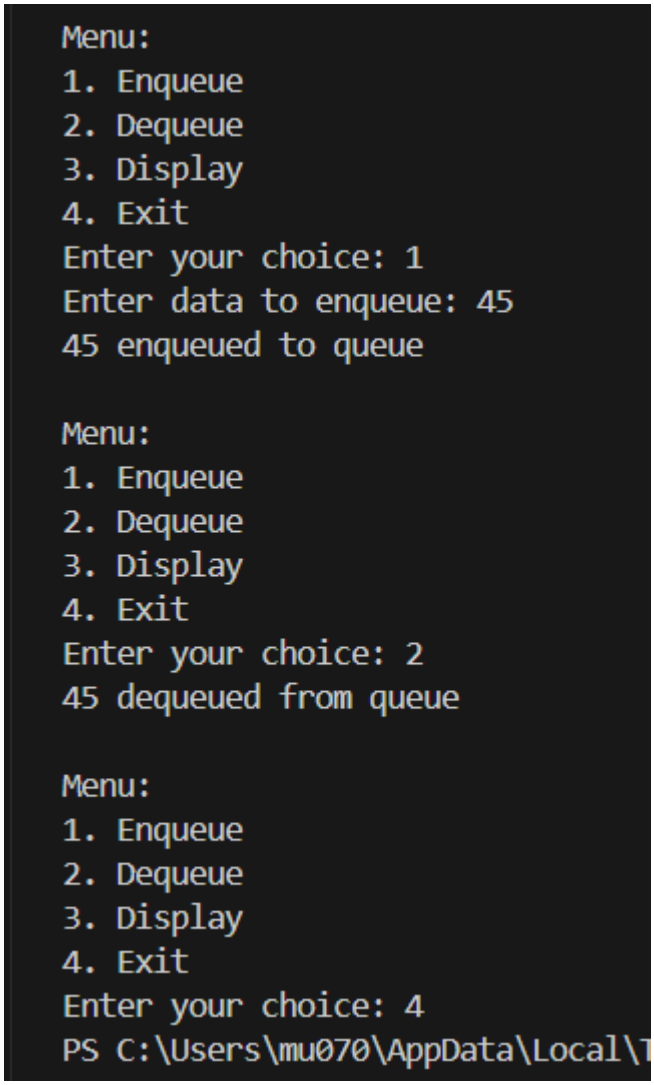
```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
int queue[MAX_SIZE];
int front = -1;
int rear = -1;
int isEmpty() {
    return (front == -1 && rear == -1);
}
int isFull() {
    return (rear == MAX_SIZE - 1);
}
```

```
void enqueue(int data) {
    if (isFull()) {
        printf("Queue Overflow!\n");
    } else {
        if (isEmpty()) {
            front = 0;
        }
        rear++;
        queue[rear] = data;
        printf("%d enqueued to queue\n", data);
    }
}
int dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow!\n");
        return -1;
    } else {
        int data = queue[front];
        if (front == rear) {
            front = rear = -1;
        } else {
            front++;
        }
 return data;
    }
}
void display() {
    if (isEmpty()) {
        printf("Queue is empty!\n");
```

```
    } else {
        printf("Queue elements:\n");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}
int main() {
    int choice, data;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to enqueue: ");
                scanf("%d", &data);
                enqueue(data);
                break;
            case 2:
                data = dequeue();
                if (data != -1) {
                    printf("%d dequeued from queue\n", data);
                }
```

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

2.OUTPUT:




3.ALGORITHM:

1. Initialization

 * Declare an array queue to store elements of the queue.

 * Initialize variables front and rear to -1, indicating an empty queue.

2. Menu-Driven Loop

 * Repeat until the user chooses to exit:

  * Display the menu with options:

   * * Enqueue

   * * Dequeue

   * * Display

   * * Exit

* Get the user's choice.

* Based on the choice:

  * Enqueue:

    * If the queue is full (overflow), display an error message.

    * Otherwise, if the queue is empty, set front and rear to 0.

    * Increment rear circularly using rear = (rear + 1) % MAX_SIZE.

    * Store the data in queue[rear].

    * Display a success message.

  * Dequeue:

    * If the queue is empty (underflow), display an error message.

    * Otherwise, retrieve the data from queue[front].

    * If front is equal to rear, set both to -1 to indicate an empty queue.

    * Otherwise, increment front circularly using front = (front + 1) % MAX_SIZE.

    * Display the dequeued data.

  * Display:

    * If the queue is empty, display an error message.

    * Otherwise, iterate through the queue from front to rear circularly and display each element.

  * Exit:

    * Terminate the program.

3. End

CODE:

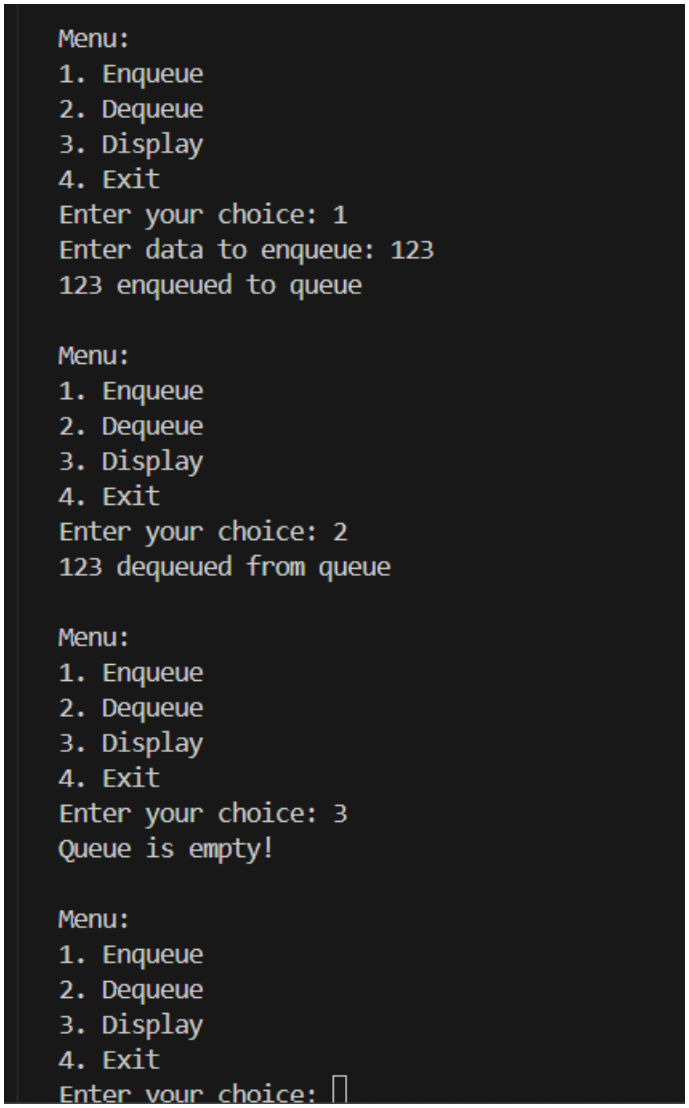
```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
int queue[MAX_SIZE];
int front = -1;
int rear = -1;
int isEmpty() {
```

```
    return (front == -1 && rear == -1);
}
int isFull() {
    return ((rear + 1) % MAX_SIZE == front);
}
void enqueue(int data) {
    if (isFull()) {
        printf("Queue Overflow!\n");
    } else {
        if (isEmpty()) {
            front = rear = 0;
        } else {
            rear = (rear + 1) % MAX_SIZE;
        }
        queue[rear] = data;
        printf("%d enqueued to queue\n", data);
    }
}
int dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow!\n");
        return -1;
    } else {
        int data = queue[front];
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % MAX_SIZE;
        }
```

```
        return data;
    }
}
void display() {
    if (isEmpty()) {
        printf("Queue is empty!\n");
    } else {
        printf("Queue elements:\n");
        int i = front;
        do {
            printf("%d ", queue[i]);
            i = (i + 1) % MAX_SIZE;
        } while (i != (rear + 1) % MAX_SIZE);
        printf("\n");
    }
}
int main() {
    int choice, data;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to enqueue: ");
```

```
            scanf("%d", &data);
            enqueue(data);
            break;
        case 2:
            data = dequeue();
            if (data != -1) {
                printf("%d dequeued from queue\n", data);
            }
            break;
        case 3:
            display();
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice!\n");
      }
   }
   return 0;
}
```

3.OUTPUT




4.ALGORITHM:

1. Initialization

* Create a Node structure to represent each node in the linked list:
  * data: To store the value of the node.
  * next: A pointer to the next node in the list.
* Initialize a global pointer head to NULL, indicating an empty list.

2. Menu-Driven Loop

* Repeat until the user chooses to exit:
  * Display the menu with options:
    * * Insert
    * * Delete

*  * Search

*  * Display

*  * Exit

* Get the user's choice.

* Based on the choice:

* Insert:

* Display options for insertion: Beginning, End, or At a given position.

* Get the user's choice.

* Based on the insertion choice:

* Beginning:

* Create a new node.

* Set the next pointer of the new node to the current head.

* Make the new node the new head.

* End:

* Create a new node.

* If the list is empty, make the new node the head.

* Otherwise, traverse to the last node in the list and set its next pointer to the new node.

* At a given position:

* Create a new node.

* If the position is 0, insert at the beginning.

* Otherwise, traverse to the node before the desired position.

* Insert the new node after the current node.

* Delete:

* Display options for deletion: Beginning, End, or At a given position.

* Get the user's choice.

* Based on the deletion choice:

* Beginning:

* If the list is empty, display an error message.

* Otherwise, store the current head in a temporary pointer, update head to point to the next node, and free the memory of the temporary pointer.

* End:

* If the list is empty or has only one node, display an error message or delete the only node, respectively.

* Otherwise, traverse to the second-to-last node in the list and set its next pointer to NULL.

* At a given position:

* If the list is empty or the position is invalid, display an error message.

* Otherwise, traverse to the node before the desired position.

* Update the next pointer of the previous node to skip the node to be deleted.

* Free the memory of the node to be deleted.

* Search:

* Get the data to search for.

* Traverse the list and compare each node's data with the search data.

* If a match is found, return the position of the node.

* If the end of the list is reached without a match, return -1 to indicate that the element was not found.

* Display:

* If the list is empty, display an error message.

* Otherwise, traverse the list and print the data of each node.

* Exit:

* Terminate the program.

4.

CODE:

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

```
};
struct Node* head = NULL;
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
     printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
void insertAtBeginning(int data) {
    struct Node* newNode = createNode(data);
    newNode->next = head;
    head = newNode;
}
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;
    }
}
```

```
void insertAtPosition(int data, int position) {
    if (position == 0) {
        insertAtBeginning(data);
    } else {
        struct Node* newNode = createNode(data);
        struct Node* temp = head;
        for (int i = 0; i < position - 1 && temp != NULL; i++) {
            temp = temp->next;
        }
        if (temp == NULL) {
            printf("Invalid position!\n");
        } else {
            newNode->next = temp->next;
            temp->next = newNode;
        }
    }
}
void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty!\n");
    } else {
        struct Node* temp = head;
        head = head->next;
        free(temp);
    }
}
void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty!\n");
```

```
    } else if (head->next == NULL) {
        free(head);
        head = NULL;
    } else {
        struct Node* temp = head;
        while (temp->next->next != NULL) {
            temp = temp->next;
        }
        free(temp->next);
        temp->next = NULL;
    }
}
void deleteAtPosition(int position) {
    if (head == NULL) {
        printf("List is empty!\n");
    } else if (position == 0) {
        deleteFromBeginning();
    } else {
        struct Node* temp = head;
        struct Node* prev = NULL;
        for (int i = 0; i < position && temp != NULL; i++) {
            prev = temp;
            temp = temp->next;
        }
        if (temp == NULL) {
            printf("Invalid position!\n");
        } else {
            if (prev != NULL) {
                prev->next = temp->next;
```

```
            } else {
                head = temp->next;
            }
            free(temp);
        }
    }
}
int search(int data) {
    struct Node* temp = head;
    int position = 0;
    while (temp != NULL) {
        if (temp->data == data) {
            return position;
        }
        temp = temp->next;
        position++;
    }
    return -1; // Element not found
}
void display() {
    if (head == NULL) {
        printf("List is empty!\n");
    } else {
        struct Node* temp = head;
        printf("List elements:\n");
        while (temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
```

```
        printf("\n");
    }
}

int main() {
    int choice, data, position;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("1. Beginning\n");
                printf("2. End\n");
                printf("3. At a given position\n");
                printf("Enter your choice: ");
                scanf("%d", &choice);
                switch (choice) {
                    case 1:
                        insertAtBeginning(data);
                        break;
                    case 2:
```

```
            insertAtEnd(data);
            break;
        case 3:
            printf("Enter position: ");
            scanf("%d", &position);
            insertAtPosition(data, position);
            break;
        default:
            printf("Invalid choice!\n");
    }
    break;
case 2:
    printf("1. Beginning\n");
    printf("2. End\n");
    printf("3. At a given position\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            deleteFromBeginning();
            break;
        case 2:
            deleteFromEnd();
            break;
        case 3:
            printf("Enter position: ");
            scanf("%d", &position);
            deleteAtPosition(position);
            break;
```

```
                default:
                    printf("Invalid choice!\n");
            }
            break;
        case 3:
            printf("Enter data to search: ");
            scanf("%d", &data);
            position = search(data);
            if (position == -1) {
                printf("Element not found!\n");
            } else {
                printf("Element found at position %d\n", position);
            }
            break;
        case 4:
            display();
            break;
        case 5:
            exit(0);
        default:
            printf("Invalid choice!\n");
      }
   }

   return 0;
}
```

4.OUTPUT

```
Menu:
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 1
Enter data to insert: 90
1. Beginning
2. End
3. At a given position
Enter your choice: 1

Menu:
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 3
Enter data to search: 90
Element found at position 0

Menu:
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 4
List elements:
90

Menu:
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 5
PS C:\Users\mu070\AppData\Local\
```

5.ALGORITHM:

1. Initialization

* Create a Node structure to represent each node in the doubly linked list:

  * data: To store the value of the node.

  * prev: A pointer to the previous node.

* next: A pointer to the next node.

* Initialize a global pointer head to NULL, indicating an empty list.

2. Menu-Driven Loop

* Repeat until the user chooses to exit:

* Display the menu with options:

* * Insert

* * Delete

* * Search

* * Display

* * Exit

* Get the user's choice.

* Based on the choice:

* Insert:

* Display options for insertion: Beginning, End, or At a given position.

* Get the user's choice.

* Based on the insertion choice:

* Beginning:

* Create a new node.

* If the list is empty, make the new node the head.

* Otherwise, set the next pointer of the new node to the current head, set the prev pointer of the current head to the new node, and make the new node the head.

* End:

* Create a new node.

* If the list is empty, make the new node the head.

* Otherwise, traverse to the last node in the list.

* Set the next pointer of the last node to the new node.

* Set the prev pointer of the new node to the last node.

* At a given position:

* Create a new node.

* If the position is 0, insert at the beginning.

* Otherwise, traverse to the node before the desired position.

* Update the pointers of the new node and its neighbors to insert it at the specified position.

* Delete:

* Display options for deletion: Beginning, End, or At a given position.

* Get the user's choice.

* Based on the deletion choice:

* Beginning:

* If the list is empty, display an error message.

* Otherwise, store the current head in a temporary pointer, update head to point to the next node, update the prev pointer of the new head to NULL, and free the memory of the temporary pointer.

* End:

* If the list is empty or has only one node, display an error message or delete the only node, respectively.

* Otherwise, traverse to the last node in the list.

* Set the next pointer of the second-to-last node to NULL.

* Free the memory of the last node.

* At a given position:

* If the list is empty or the position is invalid, display an error message.

* Otherwise, traverse to the node to be deleted.

* Update the pointers of the neighboring nodes to bypass the node to be deleted.

* Free the memory of the node to be deleted.

* Search:

* Get the data to search for.

* Traverse the list and compare each node's data with the search data.

* If a match is found, return the position of the node.

* If the end of the list is reached without a match, return -1 to indicate that the element was not found.

* Display:
* If the list is empty, display an error message.
* Otherwise, traverse the list and print the data of each node.
* Exit:
* Terminate the program.

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));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    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;
    }
}
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;
    }
}
void insertAtPosition(int data, int position) {
    if (position == 0) {
        insertAtBeginning(data);
    } else {
        struct Node* newNode = createNode(data);
        struct Node* temp = head;
        for (int i = 0; i < position - 1 && temp != NULL; i++) {
            temp = temp->next;
        }
```

```
        if (temp == NULL) {
            printf("Invalid position!\n");
        } else {
            newNode->next = temp->next;
            newNode->prev = temp;
            if (temp->next != NULL) {
                temp->next->prev = newNode;
            }
            temp->next = newNode;
        }
    }
}
void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty!\n");
    } else {
        struct Node* temp = head;
        head = head->next;
        if (head != NULL) {
            head->prev = NULL;
        }
        free(temp);
    }
}
void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty!\n");
    } else if (head->next == NULL) {
        free(head);
```

```
        head = NULL;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->prev->next = NULL;
        free(temp);
    }
}
void deleteAtPosition(int position) {
    if (head == NULL) {
        printf("List is empty!\n");
    } else if (position == 0) {
        deleteFromBeginning();
    } else {
        struct Node* temp = head;
        for (int i = 0; i < position && temp != NULL; i++) {
            temp = temp->next;
        }
        if (temp == NULL) {
            printf("Invalid position!\n");
        } else {
            if (temp->prev != NULL) {
                temp->prev->next = temp->next;
            } else {
                head = temp->next;
            }
            if (temp->next != NULL) {
```

```
            temp->next->prev = temp->prev;
        }
        free(temp);
    }
  }
}
int search(int data) {
  struct Node* temp = head;
  int position = 0;
  while (temp != NULL) {
    if (temp->data == data) {
      return position;
    }
    temp = temp->next;
    position++;
  }
  return -1; // Element not found
}
void display() {
  if (head == NULL) {
    printf("List is empty!\n");
  } else {
    struct Node* temp = head;
    printf("List elements:\n");
    while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->next;
    }
    printf("\n");
```

```
    }
}
int main() {
    int choice, data, position;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("1. Beginning\n");
                printf("2. End\n");
                printf("3. At a given position\n");
                printf("Enter your choice: ");
                scanf("%d", &choice);
                switch (choice) {
                    case 1:
                        insertAtBeginning(data);
                        break;
                    case 2:
                        insertAtEnd(data);
                        break;
```

```
            case 3:
                printf("Enter position: ");
                scanf("%d", &position);
                insertAtPosition(data, position);
                break;
            default:
                printf("Invalid choice!\n");
        }
        break;
    case 2:
        printf("1. Beginning\n");
        printf("2. End\n");
        printf("3. At a given position\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                deleteFromBeginning();
                break;
            case 2:
                deleteFromEnd();
                break;
            case 3:
                printf("Enter position: ");
                scanf("%d", &position);
                deleteAtPosition(position);
                break;
            default:
                printf("Invalid choice!\n");
```

```
            }
            break;
        case 3:
            printf("Enter data to search: ");
            scanf("%d", &data);
            position = search(data);
            if (position == -1) {
                printf("Element not found!\n");
            } else {
                printf("Element found at position %d\n", position);
            }
            break;
        case 4:
            display();
            break;
        case 5:
            exit(0);
        default:
            printf("Invalid choice!\n");
      }
   }

   return 0;
}
```

5.OUTPUT

```
Menu:
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 1
Enter data to insert: 76
1. Beginning
2. End
3. At a given position
Enter your choice: 2

Menu:
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 1
Enter data to insert: 56
1. Beginning
2. End
3. At a given position
Enter your choice: 1

Menu:
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 4
List elements:
56 76

Menu:
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 4
```