CBS1003 – Data Structures and Algorithms

Assessment – I

Name: Devika Radhakrishna Pillai

Reg No.: 24BBS0152

1. **Write a menu driven program to implement the following operations on stack:**
   a. **PUSH( )**
   b. **POP( )**
   c. **Display( )**

Ans:

PSEUDOCODE

Initialize n size of stack

Create a stack structure with an array and a top pointer

Top_Stack( ):

top = -1 (stack is empty)

isFull( ):

Return true if top is equal to n - 1

Otherwise, return false

isEmpty( ):

Return true if top is -1

Otherwise, return false

PUSH(value):

If stack is full, print "Stack Overflow" and return

Otherwise, increment top, assign value to stack[top], and print "Pushed value"

POP( ):

If stack is empty, print "Stack Underflow" and return

Else, print the value at stack[top] and decrement top

Display( ):

If stack is empty, print "Stack is empty"

Else, loop from top to 0 and print each element in the stack

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#define n 5
struct Stack {
    int arr[n];
    int top;
};
void top_Stack(struct Stack* stack) {
    stack->top = -1;
}
int isFull(struct Stack* stack) {
    return stack->top == n- 1;
}
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}
void PUSH(struct Stack* stack, int value) {
    if (isFull(stack)) {
        printf("Stack is full and insertion is not possible!");
```

```
    } else {
        stack->arr[++(stack->top)] = value;
        printf("Pushed %d onto the stack.\n", value);
    }
}
void POP(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty and deletion is not possible!");
    } else {
        int poppedValue = stack->arr[(stack->top)--];
        printf("Popped %d from the stack.\n", poppedValue);
    }
}
void Display(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty!\n");
    } else {
        printf("Stack elements: ");
        for (int i = stack->top; i >= 0; i--) {
            printf("%d ", stack->arr[i]);
        }
        printf("\n");
    }
}
int main() {
```

```
struct Stack stack;
top_Stack(&stack);
int choice, value, t=1;
while (t==1) {
   printf("\nMenu:\n1. PUSH\n2. POP\n3. Display\n4. Exit\nEnter your choice: ");
   scanf("%d", &choice);

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

```
    }
    return 0;
}
```

OUTPUT

```
Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 2
Pushed 2 onto the stack.

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 4
Pushed 4 onto the stack.

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 6
Pushed 6 onto the stack.
```

```
Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 8
Pushed 8 onto the stack.

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 10
Pushed 10 onto the stack.

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 12
Stack is full and insertion is not possible!
```

```
Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 2
Popped 10 from the stack.

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 3
Stack elements: 8 6 4 2

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 4
```

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

PSEUDOCODE

Initialize n size of queue

Create a queue structure with an array, front pointer, and rear pointer

f_r_Queue( ):

front = -1 (queue is empty)

rear = -1 (queue is empty)

isFull( ):

If rear is equal to n – 1, return true

Else, return false

isEmpty( )

If front is -1 or front is greater than rear, return true

Else, return false

Enqueue(value):

If queue is full, print "Queue Overflow" and return 1

If queue is empty (front == -1), front = 0

Increment rear, assign value to queue[rear], and print "Enqueued value"

Dequeue( ):

If queue is empty, print "Queue Underflow" and return

Else, print the value at queue[front] and increment front

If front > rear, reset front and rear to -1 (queue is empty)

Display( ):

If queue is empty, print "Queue is empty"

Else, loop from front to rear and print each element in the queue

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#define n 5
struct Queue {
    int arr[n];
    int front;
    int rear;
};
void f_r_Queue(struct Queue* queue) {
    queue->front = -1;
    queue->rear = -1;
}
int isFull(struct Queue* queue) {
    return queue->rear == n - 1;
}
int isEmpty(struct Queue* queue) {
    return queue->front == -1 || queue->front > queue->rear;
}
void Enqueue(struct Queue* queue, int value) {
    if (isFull(queue)) {
        printf("Queue Overflow! Cannot enqueue %d\n", value);
    } else {
```

```
        if (queue->front == -1) {
            queue->front = 0;
        }
        queue->arr[++(queue->rear)] = value;
        printf("Enqueued %d to the queue.\n", value);
    }
}
void Dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue Underflow! Cannot dequeue.\n");
    } else {
        int dequeuedValue = queue->arr[(queue->front)++];
        printf("Dequeued %d from the queue.\n", dequeuedValue);
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
    }
}
void Display(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty!\n");
    } else {
        printf("Queue elements: ");
        for (int i = queue->front; i <= queue->rear; i++) {
            printf("%d ", queue->arr[i]);
```

```
        }
        printf("\n");
    }
}
int main() {
    struct Queue queue;
    initQueue(&queue);
    int choice, value,t=1;
    while (t==1) {
        printf("\nMenu:\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                Enqueue(&queue, value);
                break;
            case 2:
                Dequeue(&queue);
                break;
            case 3:
                Display(&queue);
                break;
            case 4:
```

```
            t==0;
            break;
        default:
            printf("Invalid choice! Please try again.\n");
    }
  }
  return 0;
}
```

OUTPUT

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

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 6
Enqueued 6 to the queue.

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 9
Enqueued 9 to the queue.
```

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

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 15
Enqueued 15 to the queue.

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 18
Queue Overflow! Cannot enqueue 18
```

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

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue elements: 6 9 12 15

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 4
```

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

PSEUDOCODE

Initialize n size of the queue

Create a queue structure with an array, front pointer, and rear pointer

F_r_Queue( ):

front = -1 (queue is empty)

rear = -1 (queue is empty)

isFull( ):

If (rear + 1) % n is equal to front, return true

Else, return false

isEmpty( ):

If front is -1 (queue is empty), return true

Else, return false

Enqueue(value):

If queue is full, print "Queue Overflow" and return 1

If queue is empty (front == -1), front = 0

Increment rear using (rear + 1) % n, assign value to queue[rear], and print "Enqueued value"

Dequeue( ):

If queue is empty, print "Queue Underflow" and return 1

Otherwise, print the value at queue[front] and increment front using (front + 1) % n

If front is equal to rear, reset both front and rear to -1 (queue is empty)

Display( ):

If queue is empty, print "Queue is empty"

Otherwise, loop from front to rear, printing each element in the queue, taking care of wrapping around using modulus (%)

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#define n 5
struct CircularQueue {
   int arr[n];
   int front;
   int rear;
};

void f_r_Queue(struct CircularQueue* queue) {
   queue->front = -1;
   queue->rear = -1;
}
int isFull(struct CircularQueue* queue) {
   return (queue->rear + 1) % n == queue->front;
}
int isEmpty(struct CircularQueue* queue) {
   return queue->front == -1;
}
void Enqueue(struct CircularQueue* queue, int value) {
   if (isFull(queue)) {
```

```
        printf("Queue Overflow! Cannot enqueue %d\n", value);
    } else {
        if (queue->front == -1) {
            queue->front = 0;
        }
        queue->rear = (queue->rear + 1) % n;
        queue->arr[queue->rear] = value;
        printf("Enqueued %d to the queue.\n", value);
    }
}
void Dequeue(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue Underflow! Cannot dequeue.\n");
    } else {
        int dequeuedValue = queue->arr[queue->front];
        if (queue->front == queue->rear) {
            queue->front = queue->rear = -1;
        } else {
            queue->front = (queue->front + 1) % n;
        }
        printf("Dequeued %d from the queue.\n", dequeuedValue);
    }
}
void Display(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
```

```
        printf("Queue is empty!\n");
    } else {
        printf("Queue elements: ");
        int i = queue->front;
        while (i != queue->rear) {
            printf("%d ", queue->arr[i]);
            i = (i + 1) % n;
        }
        printf("%d\n", queue->arr[queue->rear]);
        }
}

int main() {
    struct CircularQueue queue;
    f_r_Queue(&queue);

    int choice, value,t=1;
    while (t==1) {
        printf("\nMenu:\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                Enqueue(&queue, value);
```

```
            break;
        case 2:
            Dequeue(&queue);
            break;
        case 3:
            Display(&queue);
            break;
        case 4:
            t=0;
            break;
        default:
            printf("Invalid choice! Please try again.\n");
      }
   }
   return 0;
}
```

OUTPUT

```
Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 5
Enqueued 5 to the queue.

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

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 15
Enqueued 15 to the queue.
```

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

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 25
Enqueued 25 to the queue.

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 30
Queue Overflow! Cannot enqueue 30
```

```
Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
Dequeued 5 from the queue.

Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue elements: 10 15 20 25
```

```
Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 4
```

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**
   - **d. Search(): search for the given element on the list**

PSEUDOCODE

Create a Node structure with two fields: data (integer) and next (pointer to the next node).

Create_node(value):

Allocate memory for a new node

Set the data of the new node to value

Set next of the new node to NULL

Return the new node

Insert_at_beginning(head, value):

Create a new node with the given value

Set the next pointer of the new node to head

Update head to point to the new node

Insert_at_end(head, value):

Create a new node with the given value

If the list is empty, set head to the new node

Else, traverse the list to the last node and set its next pointer to the new node

Insert_at_position(head, value, position):

Create a new node with the given value

If position is 1, insert the node at the beginning

Else, traverse the list to position - 1 and insert the node at the given position

Delete_from_beginning(head):

If the list is empty, return

Remove the node from the beginning and update head to the next node

Delete_from_end(head):

If the list is empty, return

Traverse the list to the second last node and set its next pointer to NULL

Free the memory of the last node

Delete_at_position(head, position):

If the list is empty, return

If position is 1, delete the first node

Else, traverse the list to position - 1 and delete the node at the given position

Search(head, value):

Traverse the list, comparing each node's data with the given value

If a match is found, print the position of the element

If the value is not found, print "Element not found"

Display(head):

If the list is empty, print "List is empty"

Else, traverse and print all the elements in the list

PROGRAM

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

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

```
    printf("Inserted %d at the end.\n", value);
}
void insert_at_position(struct Node** head, int value, int position) {
    struct Node* newNode = createNode(value);
    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
    } else {
        struct Node* temp = *head;
        for (int i = 1; i < position - 1; i++) {
            if (temp == NULL) {
                printf("Position out of bounds.\n");
                return;
            }
            temp = temp->next;
        }
        newNode->next = temp->next;
        temp->next = newNode;
    }
    printf("Inserted %d at position %d.\n", value, position);
}
void delete_from_beginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
```

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

```
void delete_at_position(struct Node** head, int position) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if (position == 1) {
        struct Node* temp = *head;
        *head = (*head)->next;
        free(temp);
        printf("Deleted from position %d.\n", position);
    } else {
        struct Node* temp = *head;
        for (int i = 1; i < position - 1; i++) {
            if (temp == NULL || temp->next == NULL) {
                printf("Position out of bounds.\n");
                return;
            }
            temp = temp->next;
        }
        struct Node* nodeToDelete = temp->next;
        temp->next = nodeToDelete->next;
        free(nodeToDelete);
        printf("Deleted from position %d.\n", position);
    }
}
```

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

```
}
int main() {
    struct Node* head = NULL;
    int choice, value, position,t=1;

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

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

```
case 2:
    printf("Enter value to insert at end: ");
    scanf("%d", &value);
    insert_at_end(&head, value);
    break;
case 3:
    printf("Enter value and position to insert: ");
    scanf("%d %d", &value, &position);
    insert_at_position(&head, value, position);
    break;
case 4:
    delete_from_beginning(&head);
    break;
case 5:
    delete_from_end(&head);
    break;
case 6:
    printf("Enter position to delete: ");
    scanf("%d", &position);
    delete_at_position(&head, position);
    break;
case 7:
    printf("Enter value to search for: ");
    scanf("%d", &value);
    search(head, value);
```

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

OUTPUT

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

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete at Given Position
7. Search for an Element
8. Display List
9. Exit
Enter your choice: 2
Enter value to insert at end: 14
Inserted 14 at the end.
```

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

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete at Given Position
7. Search for an Element
8. Display List
9. Exit
Enter your choice: 8
List elements: 7 8 14
```

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

PSEUDOCODE

Create a Node structure with three fields: data (integer), next (pointer to next node), and prev (pointer to previous node).

Create_Node(value):

Allocate memory for a new node

Set the data of the new node to value

Set next and prev to NULL

Return the new node

Insert_at_beginning(head, value):

Create a new node with the given value

If the list is empty, set head to the new node

Else, update the new node's next to point to the current head

Set the current head's prev to the new node

Update head to the new node

Insert_at_end(head, value):

Create a new node with the given value

If the list is empty, set head to the new node

Else, traverse to the end of the list and add the new node at the end

Update the new node's prev pointer to point to the last node

Insert_at_position(head, value, position):

Create a new node with the given value

If position is 1, insert at the beginning

Otherwise, traverse the list to position - 1

Insert the new node at the given position, updating the pointers

Delete_from_beginning(head):

If the list is empty, print "List is empty"

Remove the node from the beginning, update head, and free the node

Delete_from_end(head):

If the list is empty, print "List is empty"

Traverse the list to the last node and remove it, updating the previous node's next pointer

Delete_at_position(head, position):

If the list is empty, print "List is empty"

If position is 1, delete the first node

Else, traverse to the given position and remove the node, updating the appropriate pointers

Search(head, value):

Traverse the list and compare each node's data with the given value

If a match is found, print the position and return

If the value is not found, print "Element not found"

Display(head):

If the list is empty, print "List is empty"

Else, traverse and print all elements from head to the last node

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}
void insert_at_beginning(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    } else {
        newNode->next = *head;
        (*head)->prev = newNode;
        *head = newNode;
```

```
    }
    printf("Inserted %d at the beginning.\n", value);
}
void insert_at_end(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
    printf("Inserted %d at the end.\n", value);
}
void insert_at_position(struct Node** head, int value, int position) {
    struct Node* newNode = createNode(value);
    if (position == 1) {
        insert_at_beginning(head, value);
    } else {
        struct Node* temp = *head;
        for (int i = 1; i < position - 1 && temp != NULL; i++) {
            temp = temp->next;
```

```
        }
        if (temp == NULL) {
            printf("Position out of bounds.\n");
        } else {
            newNode->next = temp->next;
            if (temp->next != NULL) {
                temp->next->prev = newNode;
            }
            temp->next = newNode;
            newNode->prev = temp;
            printf("Inserted %d at position %d.\n", value, position);
        }
    }
}
void delete_from_beginning(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("Deleted from the beginning.\n");
}
void delete_from_end(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    if (temp->prev != NULL) {
        temp->prev->next = NULL;
    } else {
        *head = NULL;
    }
    free(temp);
    printf("Deleted from the end.\n");
}
void delete_at_position(struct Node** head, int position) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = *head;
```

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

```
            printf("Element %d found at position %d.\n", value, position);
            return;
        }
        temp = temp->next;
        position++;
    }
    printf("Element %d not found in the list.\n", value);
}
void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    printf("List elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    struct Node* head = NULL;
    int choice, value, position,t=1;
```

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

    switch (choice) {
        case 1:
            printf("Enter value to insert at beginning: ");
            scanf("%d", &value);
            insert_at_beginning(&head, value);
            break;
        case 2:
            printf("Enter value to insert at end: ");
            scanf("%d", &value);
            insert_at_end(&head, value);
            break;
```

```
case 3:
    printf("Enter value and position to insert: ");
    scanf("%d %d", &value, &position);
    insert_at_position(&head, value, position);
    break;
case 4:
    delete_from_beginning(&head);
    break;
case 5:
    delete_from_end(&head);
    break;
case 6:
    printf("Enter position to delete: ");
    scanf("%d", &position);
    delete_at_position(&head, position);
    break;
case 7:
    printf("Enter value to search for: ");
    scanf("%d", &value);
    search(head, value);
    break;
case 8:
    display(head);
    break;
case 9:
```

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

  return 0;
}
```

OUTPUT

```
Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete at Given Position
7. Search for an Element
8. Display List
9. Exit
Enter your choice: 1
Enter value to insert at beginning: 12
Inserted 12 at the beginning.

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete at Given Position
7. Search for an Element
8. Display List
9. Exit
Enter your choice: 2
Enter value to insert at end: 24
Inserted 24 at the end.
```

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

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete at Given Position
7. Search for an Element
8. Display List
9. Exit
Enter your choice: 8
List elements: 2 4 7 24
```