DATA STRUCTURES AND ALGORITHMS

DIGITAL ASSIGNMENT-1

NAME: Madhuja Deb Roy

REGISTRATION NO: 24BBS0128

COURSE CODE: CBS1003

1.

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

Algorithm:

1. **Initialize**:

- Set top <- -1
- Set MAX as the maximum size of the stack.

2. **PUSH(value)**:

1.
if top = MAX - 1
do Print "Stack Overflow"
2.
else
do top <- top + 1
stack[top] <- value

3. **POP()**:

1.
if top = -1
Do print "Stack Underflow"
2.
else
do Print stack[top]
top <- top - 1

1. **Display()**:

3.
if top = -1
do Print "Stack is Empty"

4.
else
do for i <- 0 to top
Print stack[i]

CODE:-

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
int *items;
int top;
int maxSize;
}Stack;
void initializeStack(Stack *s, int n) {
s->maxSize =n;
s->items =(int *)malloc(n * sizeof(int));
s->top =-1;
}
int isFull(Stack *s) {return s->top==s->maxSize - 1;
}
int isEmpty(Stack *s) {
return s->top == -1;
}
void push(Stack *s, int element) {
if (isFull(s)) {
printf("Stack Overflow! Cannot push %d.\n", element);
return;
}
s->items[++(s->top)] = element;
printf("%d pushed onto the stack.\n", element);
}
int pop(Stack *s) {
if (isEmpty(s)) {
printf("Stack Underflow! No elements to pop.\n");
return -1;
}
return s->items[(s->top)--];
}
void display(Stack *s) {
if (isEmpty(s)) {
printf("Stack is empty.\n");
return;
```

```
}
printf("Stack elements: ");
for (int i = s->top; i >= 0; i--) {
printf("%d ", s->items[i]);
}
printf("\n");
}
void freeStack(Stack *s) {
free(s->items);
}
int main() {
Stack s;
int n, choice, value;
printf("Size of stack= ");
scanf("%d", &n);
initializeStack(&s, n);
printf("\nEnter choice of operation\n");
printf("1. Push\n");
printf("2. Pop\n");
printf("3. Display\n");
printf("4. Exit\n");
while (1) {
printf("Enter your choice: ");scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter the value to push: ");
scanf("%d", &value);
push(&s, value);
break;
case 2:
value = pop(&s);
if (value != -1) {
printf("Popped element: %d\n", value);
}
break;
case 3:
display(&s);
break;
case 4:
printf("Exiting program.\n");
freeStack(&s);
exit(0);
default:
printf("Invalid choice. Please try again.\n");
}
}
```

```
return 0;
}
```

```
Size of stack= 4

Enter choice of operation
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to push: 45
45 pushed onto the stack.
Enter your choice: 3
Stack elements: 45
Enter your choice: 1
Enter the value to push: 72
72 pushed onto the stack.
Enter your choice: 1
Enter the value to push: 48
48 pushed onto the stack.
Enter your choice: 3
Stack elements: 48 72 45
Enter your choice: 1
Enter the value to push: 400
```

```
400 pushed onto the stack.
Enter your choice: 1
Enter the value to push: 67
Stack Overflow! Cannot push 67.
Enter your choice: 2
Popped element: 400
Enter your choice: 2
Popped element: 48
Enter your choice: 2
Popped element: 72
Enter your choice: 2
Popped element: 45
Enter your choice: 2
Stack Underflow! No elements to pop.
shot your choice: 4
Exiting program.
```

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

Algorithm:

1. **Initialize**:

○ Set front <- -1 and rear <- -1.

o Set MAX as the maximum size of the queue.

2. **ENQUEUE(value)**:

1.
if (rear + 1) % MAX = front do Print "Queue is full"
2.
else if front = -1 do front <- 0
3.
rear <- (rear + 1) % MAX queue[rear] <- value

3. **DEQUEUE()**:

1.
if front = -1 do Print "Queue is empty"
2.
else if front = rear do Print queue[front] front <- -1, rear <- -1
3.
else do Print queue[front] front <- (front + 1) % MAX

4. **Display()**:

1.
if front = -1 do Print "Queue is empty"
2.
else do for i <- front to rear Print queue[i]

CODE:-

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
int *items;
int front, rear;
int maxSize;
}Queue;
void initializeQueue(Queue *q, int size) {
q->maxSize = size;
q->items = (int *)malloc(size * sizeof(int));
q->front = -1;
q->rear = -1;
}
int isFull(Queue *q) {
return (q->rear + 1) % q->maxSize == q->front;
}
int isEmpty(Queue *q) {
return q->front == -1;
}
void enqueue(Queue *q, int element) {
if (isFull(q)) {
printf("Queue Overflow! Cannot enqueue %d.\n", element);
return;
}
if (isEmpty(q)) q->front = 0;
```

```
q->rear = (q->rear + 1) % q->maxSize;q->items[q->rear] = element;
printf("%d enqueued to the queue.\n", element);
}
int dequeue(Queue *q) {
if (isEmpty(q)) {
printf("Queue Underflow! No elements to dequeue.\n");
return -1;
}
int element = q->items[q->front];
if (q->front == q->rear) {
q->front = -1;
q->rear = -1;
} else {
q->front = (q->front + 1) % q->maxSize;
}
return element;
}
void display(Queue *q) {
if (isEmpty(q)) {
printf("Queue is empty.\n");
return;
}
printf("Queue elements: ");
int i = q->front;
while (1) {
printf("%d ", q->items[i]);
if (i == q->rear) break;
i = (i + 1) % q->maxSize;
}
printf("\n");
}
void freeQueue(Queue *q) {
free(q->items);
}
int main() {
Queue q;
int size, choice, value;
printf("Enter the size of the queue: ");
scanf("%d", &size);
initializeQueue(&q, size);
printf("\nEnter choice of operation\n");
printf("1. Enqueue\n");
printf("2. Dequeue\n");
printf("3. Display\n");
printf("4. Exit\n");
while (1) {
printf("Enter your choice: ");scanf("%d", &choice);
switch (choice) {
```

```c
case 1:
printf("Enter the value to enqueue: ");
scanf("%d", &value);
enqueue(&q, value);
break;
case 2:
value = dequeue(&q);
if (value != -1) {
printf("Dequeued element: %d\n", value);
}
break;
case 3:
display(&q);
break;
case 4:
printf("Exiting program.\n");
freeQueue(&q);
exit(0);
default:
printf("Invalid choice. Please try again.\n");
}
}
return 0;
}
```

```
Enter the size of the queue: 4

Enter choice of operation
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 34
34 enqueued to the queue.
Enter your choice: 1
Enter the value to enqueue: 67
67 enqueued to the queue.
Enter your choice: 1
Enter the value to enqueue: 98
98 enqueued to the queue.
Enter your choice: 1
Enter the value to enqueue: 2
2 enqueued to the queue.
Enter your choice: 1
Enter the value to enqueue: 45
Queue Overflow! Cannot enqueue 45.
```

```
Enter your choice: 2
Dequeued element: 34
Enter your choice: 2
Dequeued element: 67
Enter your choice: 2
Dequeued element: 98
Enter your choice: 2
Dequeued element: 2
Enter your choice: 2
Queue Underflow! No elements to dequeue.
Enter your choice: 3
Queue is empty.
Enter your choice: 4
Exiting program.
```

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

Algorithm: Circular Queue

1. **Initialize**:

- Set front <- -1 and rear <- -1.
- Set MAX as the maximum size of the queue.

2. **ENQUEUE(value)**:

1.
if (rear + 1) % MAX = front do Print "Queue is full"
2.
else if front = -1 do front <- 0
3.
rear <- (rear + 1) % MAX queue[rear] <- value

3. **DEQUEUE()**:

1.
if front = -1 do Print "Queue is empty"
2.
else if front = rear do Print queue[front] front <- -1, rear <- -1
3.
else do Print queue[front] front <- (front + 1) % MAX

4. **Display()**:

1.
if front = -1 do Print "Queue is empty"
2.
else do for i <- front to rear Print queue[i]

CODE:-

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
```

```
int *items;
int front, rear, maxSize;
} CircularQueue;
void initializeQueue(CircularQueue *q, int size) {
q->maxSize = size;
q->items = (int *)malloc(size * sizeof(int));
q->front = q->rear = -1;
}
int isFull(CircularQueue *q) {
return (q->rear + 1) % q->maxSize == q->front;
}
int isEmpty(CircularQueue *q) {
return q->front == -1;
}
void enqueue(CircularQueue *q, int element) {
if (isFull(q)) {
printf("Queue Overflow! Cannot enqueue %d.\n", element);
return;
}
if (isEmpty(q))
q->front = 0;
q->rear = (q->rear + 1) % q->maxSize;q->items[q->rear] = element;
printf("%d enqueued to the queue.\n", element);
}
int dequeue(CircularQueue *q) {
if (isEmpty(q)) {
printf("Queue Underflow! No elements to dequeue.\n");
return -1;
}
int element = q->items[q->front];
if (q->front == q->rear)
q->front = q->rear = -1;
else
q->front = (q->front + 1) % q->maxSize;
return element;
}
void display(CircularQueue *q) {
if (isEmpty(q)) {
printf("Queue is empty.\n");
return;
}
printf("Queue elements: ");
```

```
for (int i = q->front;; i = (i + 1) % q->maxSize) {
printf("%d ", q->items[i]);
if (i == q->rear)
break;
}
printf("\n");
}
void freeQueue(CircularQueue *q) {
free(q->items);
}
int main() {
CircularQueue q;
int size, choice, value;
printf("Enter the size of the circular queue: ");
scanf("%d", &size);
initializeQueue(&q, size);
printf("\nChoice of Operations\n");
printf("1. Enqueue\n");
printf("2. Dequeue\n");
printf("3. Display\n");
printf("4. Exit\n");
while (1) {
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:
value = dequeue(&q);
if (value != -1)
printf("Dequeued element: %d\n", value);
break;
case 3:
display(&q);
break;
case 4:
freeQueue(&q);
printf("Exiting program.\n");
return 0;
default:
printf("Invalid choice. Please try again.\n");
}
}
return 0;
```

}

```
Enter the size of the circular queue: 3

Choice of Operations
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 65
65 enqueued to the queue.
Enter your choice: 1
Enter the value to enqueue: 444
444 enqueued to the queue.
Enter your choice: 1
Enter the value to enqueue: 89
89 enqueued to the queue.
```

```
Enter your choice: 1
Enter the value to enqueue: 23
Queue Overflow! Cannot enqueue 23.
Enter your choice: 2
Dequeued element: 65
Enter your choice: 2
Dequeued element: 444
Enter your choice: 3
Queue elements: 89
Enter your choice: 2
Dequeued element: 89
Enter your choice: 4
Exiting program.
```

3) 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

Algorithm: Singly Linked List

1. **Initialize**

head <- NULL

2. **Menu Loop**

Repeat until choice = 9

○ Print the menu options.

- Read choice.
- Perform the operation based on the value of choice.

3. **Insert at Beginning**

1. Create a new node.
2. Set new_node->data <- value.
3. Set new_node->next <- head.
4. Update head <- new_node.

4. **Insert at End**

1. Create a new node.
2. Set new_node->data <- value and new_node->next <- NULL.
3. If head = NULL, update head <- new_node.
4. Else, traverse to the last node and set last_node->next <- new_node.

5. **Insert at Position**

1. If position = 1, perform "Insert at Beginning".
2. Else:
   - Create a new node and set new_node->data <- value.
   - Traverse to the (position - 1) node.
   - Set new_node->next <- current->next.
   - Update current->next <- new_node.

6. **Delete from Beginning**

1. If head = NULL, print "List is empty".
2. Else:
   - Set temp <- head.
   - Update head <- head->next.
   - Free temp.

7. **Delete from End**

1. If head = NULL, print "List is empty".
2. Else, if head->next = NULL, free head and update head <- NULL.
3. Else:
   - Traverse to the second last node.
   - Set second_last->next <- NULL.
   - Free the last node.

8. **Delete from Position**

1.
If position = 1, perform "Delete from Beginning".
2.
Else:
- Traverse to the (position - 1) node.
- Set temp <- current->next.
-
Update current->next <- temp->next.
-
Free temp.

9. **Search**

1.
Traverse the list while current ≠ NULL.
2.
If current->data = value, print "Element found".
3.
If not found, print "Element not found".

10. **Display**

1.
If head = NULL, print "List is empty".
2.
Else, traverse the list and print each node's data.

11. **Exit**

If choice = 9, terminate the program.

CODE:-

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

struct node {
   int data;
   struct node* next;
};

struct node* insertAtBeg(struct node* start, int data) {
   struct node* temp = (struct node*)malloc(sizeof(struct node));
   temp->data = data;
   temp->next = start;
   start = temp;
   return start;
}

struct node* insertAtEnd(struct node* start, int data) {
   struct node* temp = (struct node*)malloc(sizeof(struct node));
   struct node* p = start;

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

```
        p = p->next;
    }
    temp->data = data;
    p->next = temp;
    temp->next = NULL;
    return start;
}

struct node* insertAtPos(struct node* start, int data, int pos) {
    struct node* temp = (struct node*)malloc(sizeof(struct node));
    if(pos == 0) {
        start = insertAtBeg(start, data);
        return start;
    }

    struct node* p = start;
    for(int i = 0; i < pos - 1 && p != NULL; i++) {
        p = p->next;
    }

    if(p == NULL) {
        printf("The list is not big enough to insert the element at the given
position\n");
        return start;
    }

    temp->data = data;
    temp->next = p->next;
    p->next = temp;
    return start;
}

struct node* deleteAtBeg(struct node* start) {
    if(start == NULL) {
        printf("The given linked list is empty. No element deleted\n");
        return start;
    }
    struct node* temp = start;
    start = start->next;
    free(temp);
    return start;
}

struct node* deleteAtEnd(struct node* start) {
    if(start == NULL) {
        printf("The given linked list is empty. No element deleted\n");
        return start;
    }
```

```
    if(start->next == NULL) {
        free(start);
        start = NULL;
        return start;
    }

    struct node* p = start;
    while(p->next->next != NULL) {
        p = p->next;
    }
    free(p->next);
    p->next = NULL;
    return start;
}

struct node* deleteAtPos(struct node* start, int pos) {
    if(start == NULL) {
        printf("The given linked list is empty. No element deleted\n");
        return start;
    }
    struct node* p = start;
    struct node* prev = NULL;

    if(pos == 0) {
        start = deleteAtBeg(start);
        return start;
    }

    for(int i = 0; i < pos && p != NULL; i++) {
        prev = p;
        p = p->next;
    }

    if(p == NULL) {
        printf("The given linked list is not long enough to delete the element at
the given position\n");
        return start;
    }

    prev->next = p->next;
    free(p);
    return start;
}

void search(struct node* start, int data) {
    struct node* p = start;
    int counter = 0;
    while(p != NULL) {
```

```
        counter++;
        if(p->data == data) {
            printf("The given data exists in the linked list at position %d\n",
counter);
            return;
        }
        p = p->next;
    }
    printf("The given data does not exist in the linked list\n");
}

void display(struct node* start) {
    struct node* p = start;
    printf("THE LINKED LIST\n");
    while(p != NULL) {
        printf("%d => ", p->data);
        p = p->next;
    }
    printf("NULL\n");
}

int main() {
    struct node* start = NULL;
    int choice = 1;
    printf("CHOICE OF OPERATIONS \n");
    printf("1 for INSERTION AT BEGINNING \n");
    printf("2 for INSERTION AT END \n");
    printf("3 for INSERTION AT A PARTICULAR POSITION \n");
    printf("4 for DELETION AT BEGINNING \n");
    printf("5 for DELETION AT END \n");
    printf("6 for DELETION AT A PARTICULAR POSITION \n");
    printf("7 for SEARCHING AN ELEMENT IN THE LINKED LIST \n");

    do {
        int x;
        printf("ENTER CHOICE \n");
        scanf("%d", &x);
        switch(x) {
            case 1: {
                int data;
                printf("Enter the data to be added to the linked list: ");
                scanf("%d", &data);
                start = insertAtBeg(start, data);
                printf("Data added\n");
                break;
            }
            case 2: {
                int data;
```

```
            printf("Enter the data to be added at the end of the linked list\n");
            scanf("%d", &data);
            start = insertAtEnd(start, data);
            printf("Data added\n");
            break;
        }
        case 3: {
            int data, pos;
            printf("Enter the data to be added \n");
            scanf("%d", &data);
            printf("\nEnter the position at which data has to be added\n");
            scanf("%d", &pos);
            start = insertAtPos(start, data, pos - 1);
            printf("Data added\n");
            break;
        }
        case 4: {
            start = deleteAtBeg(start);
            printf("Data deleted\n");
            break;
        }
        case 5: {
            start = deleteAtEnd(start);
            printf("Data deleted\n");
            break;
        }
        case 6: {
            int pos;
            printf("Enter the position for the data to be deleted: ");
            scanf("%d", &pos);
            start = deleteAtPos(start, pos);
            printf("\nData Deleted\n");
            break;
        }
        case 7: {
            int data;
            printf("Enter data to be searched in the linked list\n");
            scanf("%d", &data);
            search(start, data);
            break;
        }
        default:
            printf("Invalid choice.\n");
    }
    display(start);
    printf("Enter 1 to continue use of program or else any other integer\n");
    scanf("%d", &choice);
} while(choice == 1);
```

```
    return 0;
}
```

```
CHOICE OF OPERATIONS
1 for INSERTION AT BEGINNING
2 for INSERTION AT END
3 for INSERTION AT A PARTICULAR POSITION
4 for DELETION AT BEGINNING
5 for DELETION AT END
6 for DELETION AT A PARTICULAR POSITION
7 for SEARCHING AN ELEMENT IN THE LINKED LIST
ENTER CHOICE
1
Enter the data to be added to the linked list: 4
Data added
THE LINKED LIST
4 => NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
2
Enter the data to be added at the end of the linked list
45
Data added
THE LINKED LIST
4 => 45 => NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
3
Enter the data to be added
67

Enter the position at which data has to be added
1
Data added
THE LINKED LIST
67 => 4 => 45 => NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
4
Data deleted
THE LINKED LIST
4 => 45 => NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
7
Enter data to be searched in the linked list
45
```

```
The given data exists in the linked list at position 2
THE LINKED LIST
4 => 45 => NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
5
Data deleted
THE LINKED LIST
4 => NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
4
Data deleted
THE LINKED LIST
NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
5
The given linked list is empty. No element deleted
Data deleted
THE LINKED LIST
shot
Enter 1 to continue use of program or else any other integer
```

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: Doubly Linked List

1. **Initialize**

head <- NULL

2. **Menu Loop**

Repeat until choice = 9

- Print the menu options.
- Read choice.
- Perform the operation based on the value of choice.

3. **Insert at Beginning**

1.
Create a new node.
2.
Set new_node->data <- value.
3.
Set new_node->prev <- NULL.
4.
Set new_node->next <- head.
5.
If head ≠ NULL, set head->prev <- new_node.
6.
Update head <- new_node.

4. **Insert at End**

1. Create a new node.
2. Set new_node->data <- value and new_node->next <- NULL.
3. If head = NULL, update head <- new_node.
4. Else, traverse to the last node and:
   - Set last_node->next <- new_node.
   - Set new_node->prev <- last_node.

5. **Insert at Position**

1. If position = 1, perform "Insert at Beginning".
2. Else:
   - Create a new node and set new_node->data <- value.
   - Traverse to the (position - 1) node.
   - Set new_node->next <- current->next.
   - Set new_node->prev <- current.
   - If current->next ≠ NULL, set current->next->prev <- new_node.
   - Update current->next <- new_node.

6. **Delete from Beginning**

1. If head = NULL, print "List is empty".
2. Else:
   - Set temp <- head.
   - Update head <- head->next.
   - If head ≠ NULL, set head->prev <- NULL.
   - Free temp.

7. **Delete from End**

1. If head = NULL, print "List is empty".
2. Else, if head->next = NULL, free head and update head <- NULL.
3. Else:
   - Traverse to the last node.
   - Set last_node->prev->next <- NULL.
   - Free last_node.

8. **Delete from Position**

1. If position = 1, perform "Delete from Beginning".
2.

Else:
- Traverse to the (position - 1) node.
- Set temp <- current->next.
-

Update current->next <- temp->next.
-

If temp->next ≠ NULL, set temp->next->prev <- current.
-

Free temp.

9. **Search**

1.
Traverse the list while current ≠ NULL.
2.
If current->data = value, print "Element found".
3.
If not found, print "Element not found".

10. **Display**

1.
If head = NULL, print "List is empty".
2.
Else, traverse the list and print each node's data.11. **Exit**

If choice = 9, terminate the program.

CODE:-

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

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

// Function to create a new node
struct node* createNewNode(int data) {
   struct node* newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = data;
   newNode->prev = NULL;
   newNode->next = NULL;
   return newNode;
}

// Function to insert at the beginning
void insertAtBeg(struct node** start, struct node** tail, int data) {
   struct node* newNode = createNewNode(data);
   if (*start == NULL) {
      *start = newNode;
      *tail = newNode;
```

```
        return;
    }
    newNode->next = *start;
    (*start)->prev = newNode;
    *start = newNode;
}

// Function to insert at the end
void insertAtEnd(struct node** start, struct node** tail, int data) {
    struct node* newNode = createNewNode(data);
    if (*start == NULL) {
        *start = newNode;
        *tail = newNode;
        return;
    }
    (*tail)->next = newNode;
    newNode->prev = *tail;
    *tail = newNode;
}

// Function to insert at a specific position
void insertAtPos(struct node** start, struct node** tail, int data, int pos) {
    struct node* newNode = createNewNode(data);
    if (pos == 0) {
        insertAtBeg(start, tail, data);
        return;
    }
    struct node* p = *start;
    for (int i = 0; i < pos - 1 && p != NULL; i++) {
        p = p->next;
    }
    if (p == NULL) {
        printf("There is not enough space in the linked list.\n\n");
        free(newNode);
        return;
    }
    newNode->next = p->next;
    newNode->prev = p;
    if (p->next != NULL) {
        p->next->prev = newNode;
    } else {
        *tail = newNode;
    }
    p->next = newNode;
}

// Function to delete from the beginning
void deleteAtBeg(struct node** start, struct node** tail) {
```

```
    if (*start == NULL) {
        printf("List is empty. Nothing to delete.\n\n");
        return;
    }
    struct node* temp = *start;
    *start = (*start)->next;
    if (*start != NULL) {
        (*start)->prev = NULL;
    } else {
        *tail = NULL;
    }
    free(temp);
    printf("First node deleted.\n\n");
}

// Function to delete from the end
void deleteAtEnd(struct node** start, struct node** tail) {
    if (*tail == NULL) {
        printf("List is empty. Nothing to delete.\n\n");
        return;
    }
    struct node* temp = *tail;
    if (*tail == *start) {
        *start = NULL;
        *tail = NULL;
    } else {
        *tail = (*tail)->prev;
        (*tail)->next = NULL;
    }
    free(temp);
    printf("Last node deleted.\n\n");
}

// Function to delete at a specific position
void deleteAtPos(struct node** start, struct node** tail, int pos) {
    if (*start == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }
    struct node* temp = *start;
    int curr = 1;
    while (temp != NULL && curr < pos) {
        temp = temp->next;
        curr++;
    }
    if (temp == NULL) {
        printf("Invalid position %d. No node found.\n", pos);
        return;
```

```
    }
    if (temp == *start) {
        deleteAtBeg(start, tail);
    } else if (temp == *tail) {
        deleteAtEnd(start, tail);
    } else {
        temp->prev->next = temp->next;
        if (temp->next != NULL) {
            temp->next->prev = temp->prev;
        }
        free(temp);
        printf("Node at position %d deleted.\n", pos);
    }
}

// Function to search an element
void search(struct node* start, int data) {
    struct node* p = start;
    int counter = 0;
    while (p != NULL) {
        counter++;
        if (p->data == data) {
            printf("The provided data exists in the linked list at %d position.\n\n",
counter);
            return;
        }
        p = p->next;
    }
    printf("The given data does not exist in the linked list.\n\n");
}

// Function to display the list
void display(struct node* start) {
    struct node* p = start;
    printf("\n\nLIST : NULL -> ");
    while (p != NULL) {
        printf("%d -> ", p->data);
        p = p->next;
    }
    printf("NULL\n");
    p = start;
    printf(" NULL <- ");
    while (p != NULL) {
        printf("%d <- ", p->data);
        p = p->next;
    }
    printf("NULL\n\n");
}
```

```
int main() {
    struct node* start = NULL;
    struct node* tail = NULL;
    int choice = 1;

    printf("Enter 1 for insertion at the beginning.\n");
    printf("Enter 2 for insertion at the end.\n");
    printf("Enter 3 for insertion at a particular position.\n");
    printf("4 for deleting the first node.\n");
    printf("Enter 5 for deleting the last node.\n");
    printf("Enter 6 for deleting the node at a particular position.\n");
    printf("Enter 7 for searching an element in the linked list.\n");

    do {
        int x;
        printf("Enter Choice \n");
        scanf("%d", &x);
        printf("\n\n");

        switch(x) {
            case 1: {
                int data;
                printf("Enter the data to be added : ");
                scanf("%d", &data);
                insertAtBeg(&start, &tail, data);
                printf("Data added.\n\n");
                break;
            }
            case 2: {
                int data;
                printf("Enter the data to be added : ");
                scanf("%d", &data);
                insertAtEnd(&start, &tail, data);
                printf("Data added.\n\n");
                break;
            }
            case 3: {
                int data, pos;
                printf("Enter the data to be added : ");
                scanf("%d", &data);
                printf("Enter the position for the data to be added : ");
                scanf("%d", &pos);
                insertAtPos(&start, &tail, data, pos-1);
                printf("Data added.\n\n");
                break;
            }
            case 4: {
```

```
            deleteAtBeg(&start, &tail);
            printf("Data deleted.\n\n");
            break;
        }
        case 5: {
            deleteAtEnd(&start, &tail);
            printf("Data deleted.\n\n");
            break;
        }
        case 6: {
            int pos;
            printf("Enter the position for the data to be deleted : ");
            scanf("%d", &pos);
            deleteAtPos(&start, &tail, pos);
            printf("Data deleted.\n\n");
            break;
        }
        case 7: {
            int data;
            printf("Enter the data to be searched for : ");
            scanf("%d", &data);
            search(start, data);
            break;
        }
        default: {
            printf("Invalid choice.\n\n");
            break;
        }
      }
      display(start);
      printf("Enter 1 to continue use of the program.\nEnter any other integer
to exit.\nCHOICE : ");
      scanf("%d", &choice);
   } while(choice == 1);

   return 0;
}
```

```
Enter 1 for insertion at the beginning.
Enter 2 for insertion at the end.
Enter 3 for insertion at a particular position.
4 for deleting the first node.
Enter 5 for deleting the last node.
Enter 6 for deleting the node at a particular position.
Enter 7 for searching an element in the linked list.
Enter Choice
1


Enter the data to be added : 59
Data added.



LIST : NULL -> 59 -> NULL
 NULL <- 59 <- NULL

Enter 1 to continue use of the program.
Enter any other integer to exit.
CHOICE : 1
Enter Choice
1
```

```
Enter the data to be added : 22
Data added.



LIST : NULL -> 22 -> 59 -> NULL
 NULL <- 22 <- 59 <- NULL

Enter 1 to continue use of the program.
Enter any other integer to exit.
CHOICE : 1
Enter Choice
2


Enter the data to be added : 44
Data added.



LIST : NULL -> 22 -> 59 -> 44 -> NULL
 NULL <- 22 <- 59 <- 44 <- NULL

Enter 1 to continue use of the program.
Enter any other integer to exit.
CHOICE : 1
Enter Choice
```

```
Enter 1 to continue use of the program.
Enter any other integer to exit.
CHOICE : 1
Enter Choice
3


Enter the data to be added : 66
Enter the position for the data to be added : 2
Data added.



LIST : NULL -> 22 -> 66 -> 59 -> 44 -> NULL
 NULL <- 22 <- 66 <- 59 <- 44 <- NULL

Enter 1 to continue use of the program.
Enter any other integer to exit.
CHOICE : 1
Enter Choice
4


First node deleted.

Data deleted.
```

```
Data deleted.



LIST : NULL -> 66 -> 59 -> 44 -> NULL
 NULL <- 66 <- 59 <- 44 <- NULL

Enter 1 to continue use of the program.
Enter any other integer to exit.
CHOICE : 1
Enter Choice
5


Last node deleted.

Data deleted.



LIST : NULL -> 66 -> 59 -> NULL
 NULL <- 66 <- 59 <- NULL

Enter 1 to continue use of the program.
Enter any other integer to exit.
CHOICE : 7
```