Data Structures and Algorithms

Assessment-1

Sahaj Gupta

24BBS0166

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

Algorithm:

Initialize Stack

- Create an array `stack[MAX]` and set `top = -1`.

PUSH Operation

- Input `item` to be pushed.
- If `top == MAX - 1`, output "Stack Overflow".
- Else, increment `top` by 1 and assign `stack[top] = item`.

POP Operation

- If `top == -1`, output "Stack Underflow".
- Else, output `stack[top]` and decrement `top` by 1.

Display Operation

- If `top == -1`, output "Stack is empty".
- Else, loop from `i = top` to `i = 0` and print `stack[i]`.

Menu Loop

- Display a menu with PUSH, POP, and Display options.
- Take the user's choice and perform the corresponding operation.
- Continue until the user chooses to exit.

Code:

```c
#include <stdio.h>
int n=5;
int s[130];
int top=-1;
int isempty();
int isfull();
void push(int x)
{
   if (isfull()) {
       printf("Stack overflow");
   }
   else {
       top++;
       s[top]=x;
   }
}
int pop()
{
   if (isempty()) {
       printf("Stack underflow");
       return 0;
   }
   int k=s[top];
   top--;
   return k;
}
int isfull()
{
   if (top==n-1)
       return 1;
   return 0;
}
int isempty()
{
   if (top==-1)
       return 1;
   return 0;
}
int main()
{
   int r,x,choice,c;
```

```
    printf("Enter range:");
    scanf("%d",&r);
    while (1) {
        printf("\n1. Push element into stack\n");
        printf("2. Pop element from stack\n");
        printf("3. Display the stack\n");
        scanf("%d",&choice);
        switch (choice) {
            case 1:
                printf("Enter element:");
            scanf("%d",&x);
            push(x);
            break;
            case 2:
                c=pop();
            printf("Popped element:%d\n",c);
            break;
            case 3:
                if (top==-1) {
                    printf("Stack underflow");
                }
                for (int i=top;i>-1;i--) {
                    printf("%d\n", s[i]);
                }
        }
    }
}
```

Output:

```
1. Push element into stack
2. Pop element from stack
3. Display the stack
1
Enter element:2

1. Push element into stack
2. Pop element from stack
3. Display the stack
3
2
1

1. Push element into stack
2. Pop element from stack
3. Display the stack
2
Popped element:2
1. Push element into stack
2. Pop element from stack
3. Display the stack
3
1
```

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

Algorithm:

1. Initialize Queue

- Create an array `queue[MAX]` and initialize `front = -1` and `rear = -1`.

2. Enqueue Operation

- Input `item` to be added.
- If `(rear == MAX - 1)`, output "Queue Overflow".
- Else:
  - If `front == -1`, set `front = 0`.
  - Increment `rear` by 1 and assign `queue[rear] = item`.

3. Dequeue Operation

- If `front == -1 || front > rear`, output "Queue Underflow".
- Else:
  - Output `queue[front]`.
  - Increment `front` by 1.
  - If `front > rear`, reset `front = -1` and `rear = -1` (queue becomes empty).

4. Display Operation

- If `front == -1`, output "Queue is empty".
- Else:
  - Loop from `i = front` to `i = rear` and print `queue[i]`.

5. Menu Loop

- Display a menu with Enqueue, Dequeue, and Display options.
- Take the user's choice and perform the corresponding operation.
- Continue until the user chooses to exit.

Code:

```c
#define N 5
#include <stdio.h>
#include <stdlib.h>
int queue[N];
int front=-1,rear=-1;
void Enqueue(int x)
{
   if (rear == N-1) {
       printf("Queue Overflow");
   }
   else if (rear==-1 && front ==-1) {
       front++;
       rear++;
       queue[rear]=x;
   }
   else {
       rear++;
       queue[rear]=x;
   }
}
void Dequeue()
{
   if ((rear==-1 && front ==-1) || front>rear) {
       printf("Underflow");
   }
   else {
       front++;
   }
}
int Peak() {
   return queue[front];
}
void Display() {
   if (front == -1 || front > rear) {
       printf("Queue is empty\n");
   }
   else {
       for (int i = front; i <= rear; i++) {
           printf("%d\n", queue[i]);
       }
       printf("\n");
   }
}
```

```c
int main()
{
   int choice,y,p;
   while (1)
   {
       printf("\n1. Enqueue\n");
       printf("2. Dequeue\n");
       printf("3. Peek\n");
       printf("4. Display the Queue\n");
       printf("5. Quit\n\n");
       scanf("%d",&choice);
       switch (choice) {
           case 1:
               printf("Enter number you want to enqueue:");
           scanf("%d",&y);
           Enqueue(y);
           break;
           case 2:
               Dequeue();
           break;
           case 3:
               p = Peak();
           printf("%d",p);
           break;
           case 4:
               Display();
           case 5:
               exit(1);
       }

   }
}
```

Output:

```
1. Enqueue
2. Dequeue
3. Peek
4. Display the Queue
5. Quit

1
Enter number you want to enqueue:1

1. Enqueue
2. Dequeue
3. Peek
4. Display the Queue
5. Quit

1
```

```
Enter number you want to enqueue:2

1. Enqueue
2. Dequeue
3. Peek
4. Display the Queue
5. Quit

4
1
2


1. Enqueue
2. Dequeue
3. Peek
4. Display the Queue
5. Quit

2
```

```
1. Enqueue
2. Dequeue
3. Peek
4. Display the Queue
5. Quit

5


Process finished with exit code 1
```

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

Algorithm:

1. Enqueue Operation

Input: Take the element `x` to be inserted.
Check Overflow:

- If `(rear + 1) % N == front`, the queue is full.
  - **Output**: Print "Queue Overflow" and exit.

If the queue is empty:

- If `front == -1` and `rear == -1`:
  - Set `front = 0` and `rear = 0`.
  - Insert `x` at `queue[rear]`.

Otherwise:

- Increment `rear` in a circular manner: `rear = (rear + 1) % N`.
- Insert `x` at `queue[rear]`.

Output: Print "Element enqueued successfully."

2. Dequeue Operation
Check Underflow:

- If `front == -1`, the queue is empty.
  - **Output**: Print "Queue Underflow" and exit.

Retrieve the Element:

- Access `queue[front]` and store it as the element to be dequeued.

Check if the queue has only one element:

- If `front == rear`, set `front = -1` and `rear = -1` to reset the queue.

Otherwise:

- Increment `front` in a circular manner: `front = (front + 1) % N`.

Output: Print the dequeued element.

3. Display Operation
Check if the queue is empty:

- If `front == -1`, the queue is empty.
    - **Output**: Print "Queue is empty" and exit.

Print Elements:

- Start from `i = front` and print `queue[i]` until `i == rear`:
    - Print `queue[i]`.
    - Update `i = (i + 1) % N`.
- Print `queue[rear]` (last element).

Output: Display all elements from `front` to `rear`.

4. Menu Loop
Start the Program.
Display the menu with the following options:

- **1. Enqueue**
- **2. Dequeue**
- **3. Display**
- **4. Quit**

Input the user's choice.
Based on the user's choice:

- **If choice == 1**: Read the element `x` and call `Enqueue(x)`.
- **If choice == 2**: Call `Dequeue()`.
- **If choice == 3**: Call `Display()`.
- **If choice == 4**: Exit the program.

Repeat steps 2–4 until the user chooses to quit.
End the program.

Code:

```c
#include <stdio.h>
#define N 5

int queue[N];
int front = -1, rear = -1;
void Enqueue(int x) {
   if ((rear + 1) % N == front) {
       printf("Queue Overflow\n");
   } else if (front == -1 && rear == -1) {
       front = rear = 0;
       queue[rear] = x;
       printf("%d enqueued.\n", x);
   } else {
       rear = (rear + 1) % N;
       queue[rear] = x;
       printf("%d enqueued.\n", x);
   }
}
void Dequeue() {
   if (front == -1) {
       printf("Queue Underflow\n");
   } else {
       printf("Dequeued: %d\n", queue[front]);
       if (front == rear) {
           front = rear = -1;
       } else {
           front = (front + 1) % N;
       }
   }
}
void Display() {
   if (front == -1) {
       printf("Queue is empty\n");
   } else {
       int i = front;
       printf("Queue elements: ");
       while (i != rear) {
           printf("%d ", queue[i]);
           i = (i + 1) % N;
       }
       printf("%d\n", queue[rear]);
   }
}
int main() {
   int choice, value;
   while (1) {
       printf("\nMenu:\n");
```

```
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display the Queue\n");
        printf("4. Quit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element to enqueue: ");
                scanf("%d", &value);
                Enqueue(value);
                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 the Queue
4. Quit
Enter your choice: 1
Enter the element to enqueue: 1
1 enqueued.

Menu:
1. Enqueue
2. Dequeue
3. Display the Queue
4. Quit
Enter your choice: 1
Enter the element to enqueue: 2
2 enqueued.
```

```
Menu:
1. Enqueue
2. Dequeue
3. Display the Queue
4. Quit
Enter your choice: 1
Enter the element to enqueue: 3
3 enqueued.

Menu:
1. Enqueue
2. Dequeue
3. Display the Queue
4. Quit
Enter your choice: 1
Enter the element to enqueue: 4
4 enqueued.

Menu:
1. Enqueue
2. Dequeue
3. Display the Queue
4. Quit
Enter your choice: 1
Enter the element to enqueue: 5
5 enqueued.
```

```
Menu:
1. Enqueue
2. Dequeue
3. Display the Queue
4. Quit
Enter your choice: 2
Dequeued: 3

Menu:
1. Enqueue
2. Dequeue
3. Display the Queue
4. Quit
Enter your choice: 1
Enter the element to enqueue: 3
3 enqueued.

Menu:
1. Enqueue
2. Dequeue
3. Display the Queue
4. Quit
Enter your choice: 3
Queue elements: 4 5 3
```

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
   c. Search(): search for the given element on the list

Algorithm:

1. Insertion

Insertion at Beginning:

- **Input**: Take the value `x` to be inserted.
- Create a new node with the value `x`.
- Point the new node's `next` to the current head.
- Update the head of the list to the new node.

Insertion at End:

- **Input**: Take the value `x` to be inserted.
- Create a new node with the value `x`.
- If the list is empty (head is NULL), make the new node the head.
- Otherwise, traverse the list to the last node and update its `next` to the new node.

Insertion at a Given Position:

- **Input**: Take the value `x` and position `pos` where the node should be inserted.
- Traverse the list to find the node at position `pos - 1`.
- Create a new node with the value `x`.
- Make the previous node's `next` point to the new node, and the new node's `next` point to the next node in the list.

2. Deletion

Deletion at Beginning:

- If the list is empty, print "List is empty".
- Otherwise, update the head to point to the second node, effectively removing the first node.

Deletion at End:

- If the list is empty, print "List is empty".
- Traverse the list to find the second-to-last node.
- Update its `next` to NULL, removing the last node.

Deletion at a Given Position:

- **Input**: Take the position `pos` where the node should be deleted.
- Traverse the list to find the node at position `pos - 1`.
- Update the previous node's `next` to point to the node after the one to be deleted.

3. Search

- **Input**: Take the value `x` to search for.
- Traverse the list from the head.
- If a node with value `x` is found, print "Element found".
- If the list is traversed completely without finding `x`, print "Element not found".

Code:

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

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

struct Node* head = NULL;
void InsertAtBeginning(int x) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = x;
   newNode->next = head;
   head = newNode;
   printf("%d inserted at the beginning.\n", x);
}
void InsertAtEnd(int x) {
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    printf("%d inserted at the end.\n", x);
}
void InsertAtPosition(int x, int pos) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    if (pos == 1) {
        newNode->next = head;
        head = newNode;
        return;
    }
    struct Node* temp = head;
    for (int i = 1; i < pos - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of bounds.\n");
    } else {
        newNode->next = temp->next;
        temp->next = newNode;
        printf("%d inserted at position %d.\n", x, pos);
    }
}
void DeleteAtBeginning() {
    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() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
```

```
    struct Node* temp = head;
    if (temp->next == NULL) {
        free(temp);
        head = NULL;
        printf("Node deleted from the end.\n");
        return;
    }
    while (temp->next != NULL && temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
    printf("Node deleted from the end.\n");
}
void DeleteAtPosition(int pos) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if (pos == 1) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
        printf("Node deleted from position %d.\n", pos);
        return;
    }
    struct Node* temp = head;
    for (int i = 1; i < pos - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL) {
        printf("Position out of bounds.\n");
    } else {
        struct Node* nodeToDelete = temp->next;
        temp->next = nodeToDelete->next;
        free(nodeToDelete);
        printf("Node deleted from position %d.\n", pos);
    }
}
void Search(int x) {
    struct Node* temp = head;
    int pos = 1;
    while (temp != NULL) {
        if (temp->data == x) {
            printf("Element %d found at position %d.\n", x, pos);
            return;
        }
        temp = temp->next;
        pos++;
```

```
    }
    printf("Element %d not found.\n", x);
}
void Display() {
    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("\n");
}
int main() {
    int choice, value, pos;
    while (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 from Given Position\n");
        printf("7. Search Element\n");
        printf("8. Display the List\n");
        printf("9. Quit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value to insert: ");
                scanf("%d", &value);
                InsertAtBeginning(value);
                break;
            case 2:
                printf("Enter the value to insert: ");
                scanf("%d", &value);
                InsertAtEnd(value);
                break;
            case 3:
                printf("Enter the value to insert: ");
                scanf("%d", &value);
                printf("Enter the position: ");
                scanf("%d", &pos);
                InsertAtPosition(value, pos);
```

```
                break;
            case 4:
                DeleteAtBeginning();
                break;
            case 5:
                DeleteAtEnd();
                break;
            case 6:
                printf("Enter the position: ");
                scanf("%d", &pos);
                DeleteAtPosition(pos);
                break;
            case 7:
                printf("Enter the value to search: ");
                scanf("%d", &value);
                Search(value);
                break;
            case 8:
                Display();
                break;
            case 9:
                printf("Exiting program.\n");
                return 0;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
}
```

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 from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 1
Enter the value to insert: 1
1 inserted 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 from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 1
Enter the value to insert: 2
2 inserted 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 from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 8
Linked List: 2 1

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 2
Enter the value to insert: 3
3 inserted 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 from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 8
Linked List: 1

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 7
Enter the value to search: 1
Element 1 found at position 1.
```

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:

1. Insertion

Insertion at the Beginning:

1. Create a new node.
2. Set the `next` of the new node to the current `head`.
3. Set the `prev` of the old `head` (if it exists) to the new node.
4. Set the `head` to the new node.

Insertion at the End:

1. Create a new node.
2. Traverse the list to find the last node.
3. Set the `next` of the last node to the new node.
4. Set the `prev` of the new node to the last node.
5. Set the `next` of the new node to `NULL`.

Insertion at a Given Position:

1. Create a new node.
2. Traverse the list to find the node at the given position.
3. Set the `next` of the node at the given position to the new node.
4. Set the `prev` of the new node to the node at the given position.
5. Set the `next` of the new node to the next node of the given position node.

6. Set the `prev` of the next node to the new node (if it exists).

2. Deletion

Deletion from the Beginning:

1. If the list is empty, print "Underflow" and return.
2. Set the head to the next node of the current head.
3. Set the prev of the new head node to NULL.
4. Free the old head node.

Deletion from the End:

1. If the list is empty, print "Underflow" and return.
2. Traverse the list to find the last node.
3. Set the prev of the second last node to NULL.
4. Free the last node.

Deletion at a Given Position:

1. If the list is empty, print "Underflow" and return.
2. Traverse the list to find the node at the given position.
3. Set the next of the previous node to the node after the given node.
4. Set the prev of the next node (if it exists) to the previous node.
5. Free the node at the given position.

3. Search

- Traverse the list from the head.
- If the node's data matches the search element, print the position of the element.
- If the element is not found, print "Element not found".

Code:

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

struct Node {
   int data;
   struct Node* next;
   struct Node* prev;
};
struct Node* head = NULL;
void InsertAtBeginning(int x) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = x;
```

```
    newNode->next = head;
    newNode->prev = NULL;
    if (head != NULL) {
        head->prev = newNode;
    }
    head = newNode;
    printf("%d inserted at the beginning.\n", x);
}
void InsertAtEnd(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    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("%d inserted at the end.\n", x);
}
void InsertAtPosition(int x, int pos) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    if (pos == 1) {
        newNode->next = head;
        newNode->prev = NULL;
        if (head != NULL) {
            head->prev = newNode;
        }
        head = newNode;
        return;
    }
    struct Node* temp = head;
    for (int i = 1; i < pos - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of bounds.\n");
    } else {
        newNode->next = temp->next;
        newNode->prev = temp;
        if (temp->next != NULL) {
            temp->next->prev = newNode;
        }
```

```
        temp->next = newNode;
        printf("%d inserted at position %d.\n", x, pos);
    }
}
void DeleteAtBeginning() {
    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() {
    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("Node deleted from the end.\n");
}
void DeleteAtPosition(int pos) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    for (int i = 1; i < pos && 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("Node deleted from position %d.\n", pos);
    }
}
void Search(int x) {
    struct Node* temp = head;
    int pos = 1;
    while (temp != NULL) {
        if (temp->data == x) {
            printf("Element %d found at position %d.\n", x, pos);
            return;
        }
        temp = temp->next;
        pos++;
    }
    printf("Element %d not found.\n", x);
}
void Display() {
    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("\n");
}
int main() {
    int choice, value, pos;
    while (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 from Given Position\n");
        printf("7. Search Element\n");
        printf("8. Display the List\n");
        printf("9. Quit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
                printf("Enter the value to insert: ");
                scanf("%d", &value);
                InsertAtBeginning(value);
                break;
            case 2:
                printf("Enter the value to insert: ");
                scanf("%d", &value);
                InsertAtEnd(value);
                break;
            case 3:
                printf("Enter the value to insert: ");
                scanf("%d", &value);
                printf("Enter the position: ");
                scanf("%d", &pos);
                InsertAtPosition(value, pos);
                break;
            case 4:
                DeleteAtBeginning();
                break;
            case 5:
                DeleteAtEnd();
                break;
            case 6:
                printf("Enter the position: ");
                scanf("%d", &pos);
                DeleteAtPosition(pos);
                break;
            case 7:
                printf("Enter the value to search: ");
                scanf("%d", &value);
                Search(value);
                break;
            case 8:
                Display();
                break;
            case 9:
                printf("Exiting program.\n");
                return 0;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
}
```

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 from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 1
Enter the value to insert: 1
1 inserted 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 from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 1
Enter the value to insert: 2
2 inserted 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 from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 8
Linked List: 2 1

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 2
Enter the value to insert: 3
3 inserted 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 from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 8
Linked List: 1

Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Given Position
4. Delete from Beginning
5. Delete from End
6. Delete from Given Position
7. Search Element
8. Display the List
9. Quit
Enter your choice: 7
Enter the value to search: 1
Element 1 found at position 1.
```