DATA STRUCTURES AND ALGORITHMS

ASSESSMENT 1

Q1] Write a menu driven program to implement the following operations on stack. a. PUSH() b. POP() c. Display()

Algorithm

PUSH

If top == SIZE - 1, print "Stack Overflow."
Else, increment top and store the element at stack[top].

POP

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

DISPLAY

If top == -1, print "Stack is empty."
Else, print elements from stack[top] to stack[0].

Menu-Driven Program

Repeat:
Show menu: Push, Pop, Display, Exit.
Perform the operation based on the user's choice.
Exit when the user selects "Exit."

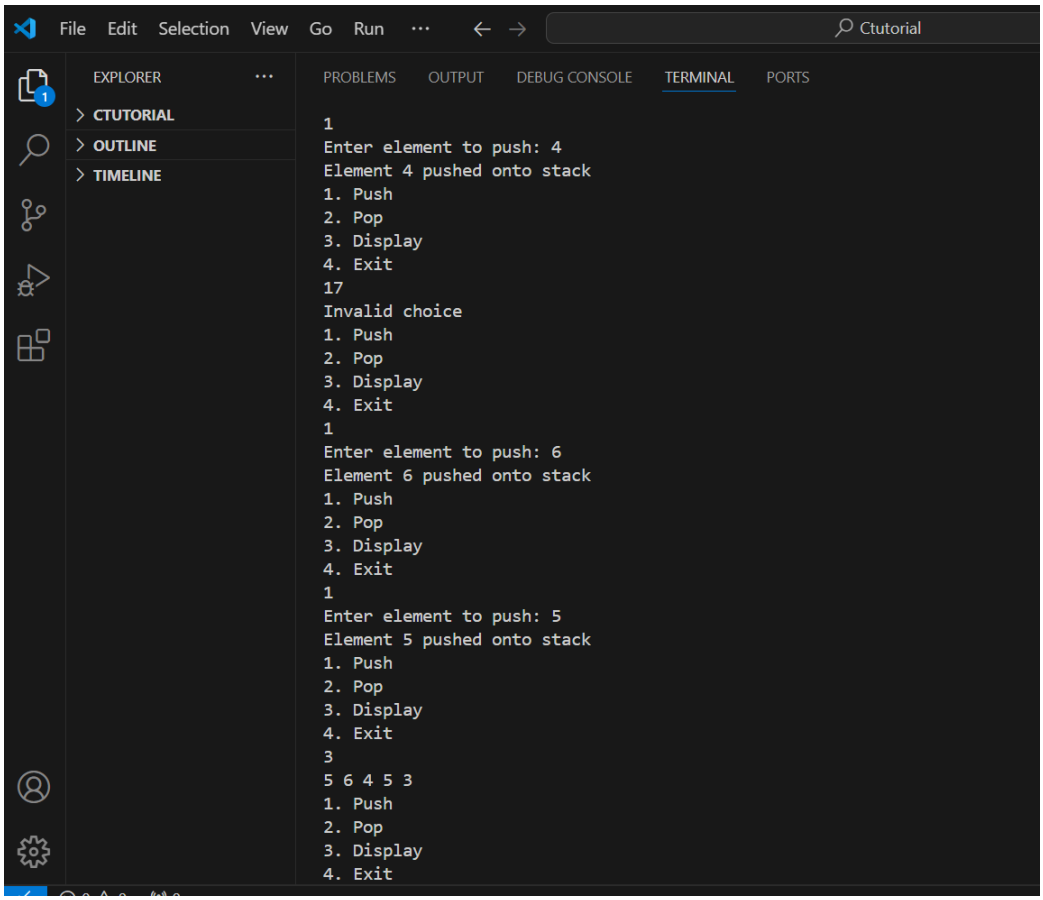
CODE

```
#include<stdio.h>

#include<stdlib.h>

#define SIZE 100

int stack[SIZE], top = -1;

void push(int element) {
   if(top == SIZE - 1) {
      printf("Stack Overflow\n");
      return;}
   stack[++top] = element;
   printf("Element %d pushed onto stack\n", element);
}

void pop() {
   if(top == -1) {
```

```
        printf("Stack Underflow\n");
        return;
    }
    printf("Element %d popped from stack\n", stack[top--]);}
void display() {
    if(top == -1) {
        printf("Stack is empty\n");
        return;
    }
    for(int i = top; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}
int main() {
    int choice, element;
    while(1) {
        printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
        scanf("%d", &choice);
        switch(choice) {
            case 1:
                printf("Enter element to push: ");
                scanf("%d", &element);
                push(element);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
```

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

OUTPUT



Q2] Write a menu driven program to implement the following operations on Queue: a. Enqueue() b. Dequeue() c. Disaply()

ALGORITHM

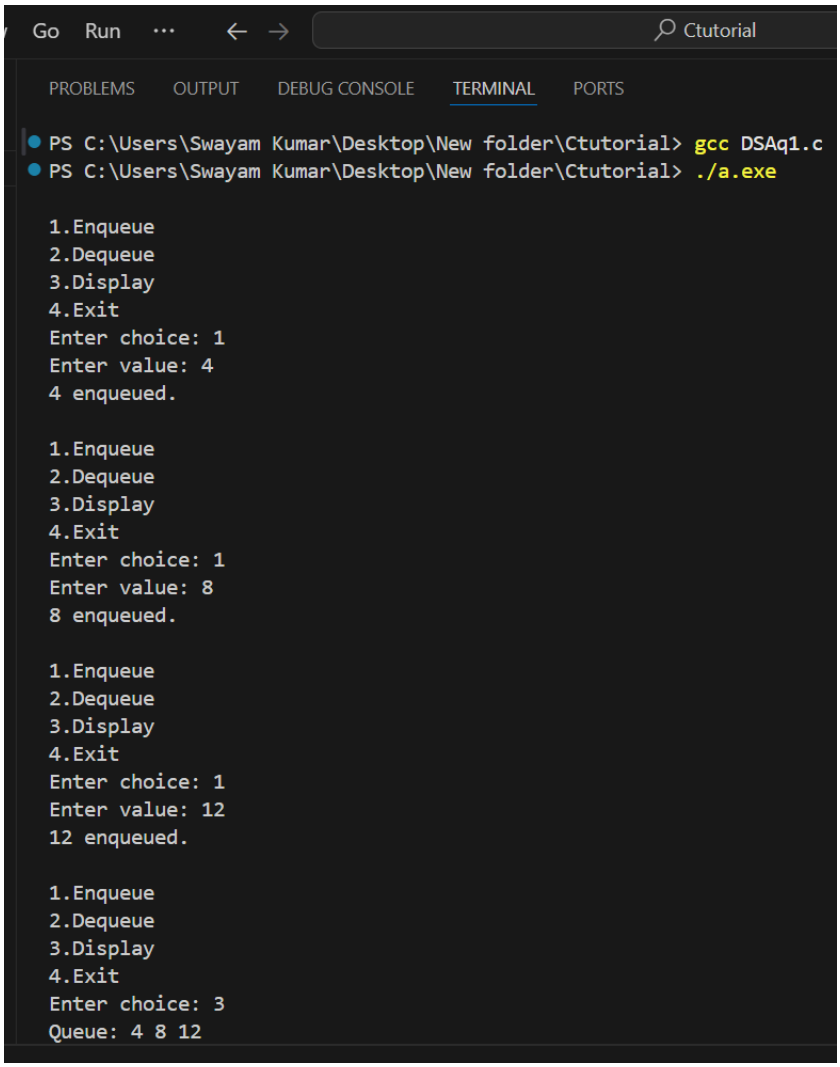
Start Initialize the queue with an array and variables front and rear to -1.
Display Menu: a. Enqueue b. Dequeue c. Display d.
Exit Input choice from the user. Based on the choice: a. Enqueue: Check if the queue is full (rear == size - 1).
If not, increment rear and insert the element at queue[rear].
b. Dequeue: Check if the queue is empty (front == rear). If not, increment front and remove the element from queue[front].
c. Display: Print all elements from front + 1 to rear.
d. Exit the program. Repeat steps 4-5 until the user selects "Exit".
End

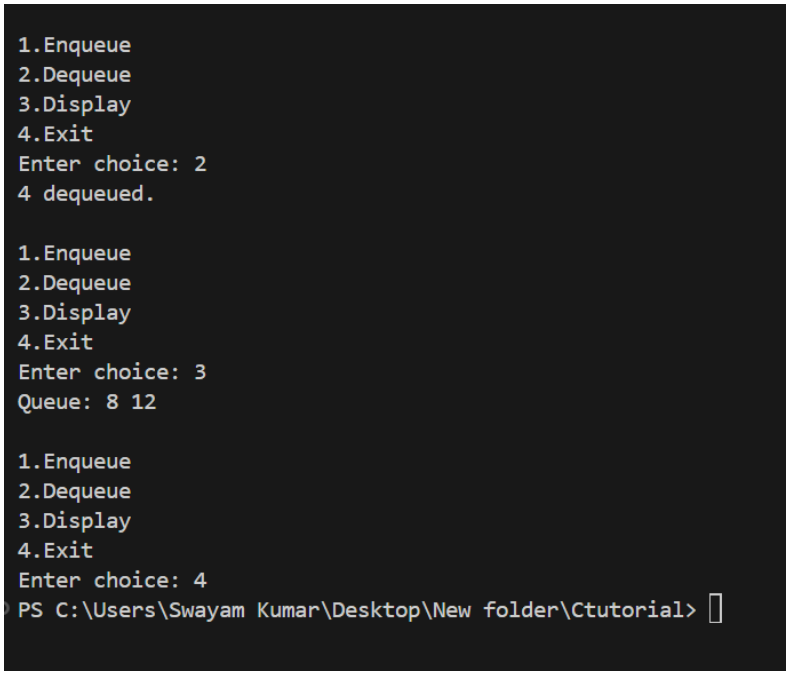
CODE

```c
#include <stdio.h>
#define SIZE 5
int queue[SIZE], front = -1, rear = -1;
void enqueue() {
    int value;
    if (rear == SIZE - 1) {
        printf("Queue is Full!\n");
    } else {
        printf("Enter value: ");
        scanf("%d", &value);
        rear++;
        queue[rear] = value;
        printf("%d enqueued.\n", value);
    }
}
void dequeue() {
    if (front == rear) {
        printf("Queue is Empty!\n");
    } else {
        front++;
        printf("%d dequeued.\n", queue[front]);
    }
}
void display() {
    if (front == rear) {
        printf("Queue is Empty!\n");
    } else {
        printf("Queue: ");
        for (int i = front + 1; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}
int main() {
```

```c
    int choice;
    while (1) {
        printf("\n1.Enqueue\n2.Dequeue\n3.Display\n4.Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: enqueue(); break;
            case 2: dequeue(); break;
            case 3: display(); break;
            case 4: return 0;
            default: printf("Invalid choice!\n");
        }
    }
}
```

OUTPUT




1.Enqueue
2.Dequeue
3.Display
4.Exit
Enter choice: 2
4 dequeued.
1.Enqueue
2.Dequeue
3.Display
4.Exit
Enter choice: 3
Queue: 8 12
1.Enqueue
2.Dequeue
3.Display
4.Exit
Enter choice: 4
PS C:\Users\Swayam Kumar\Desktop\New folder\Ctutorial>

Q3] Write a menu driven program to implement the following operations on circular Queue: a. Enqueue() b. Dequeue() c. Disaply()

ALGORITHM

Start
Initialize the circular queue with an array, variables front = -1 and rear = -1.
Display Menu with the following options:
a. Enqueue

b. Dequeue

c. Display

d. Exit
Input the user's choice.
Based on the choice, perform the following operations:
a. Enqueue:
Check if the queue is full using the condition: (rear + 1) % SIZE == front.
If the queue is full, display "Queue is Full".
Otherwise, insert the element at rear:
If front == -1, set front = 0 (first insertion).
Increment rear using rear = (rear + 1) % SIZE.
Store the value in queue[rear].
b. Dequeue:
Check if the queue is empty using the condition: front == -1.
If the queue is empty, display "Queue is Empty".
Otherwise:
Retrieve the element at queue[front].
If front == rear, reset front and rear to -1 (last element dequeued).

Otherwise, increment front using front = (front + 1) % SIZE.

c. Display:
Check if the queue is empty using the condition: front == -1.
If empty, display "Queue is Empty".
Otherwise:
Traverse the queue from front to rear, using a loop and the formula (index + 1) % SIZE to wrap around.

d. Exit:
Terminate the program.
Repeat steps 4 and 5 until the user chooses "Exit"
End.

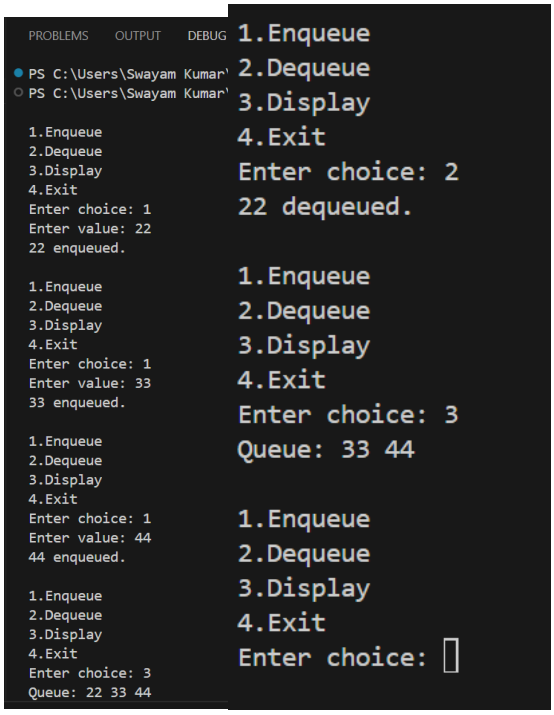
CODE

```c
#include <stdio.h>
#define SIZE 5
int queue[SIZE], front = -1, rear = -1;
void enqueue() {
```

```c
    int value;
    if ((rear + 1) % SIZE == front) {
        printf("Queue is Full!\n");
    } else {
        printf("Enter value: ");
        scanf("%d", &value);
        if (front == -1) front = 0;
        rear = (rear + 1) % SIZE;
        queue[rear] = value;
        printf("%d enqueued.\n", value);
    }
}
void dequeue() {
    if (front == -1) {
        printf("Queue is Empty!\n");
    } else {
        printf("%d dequeued.\n", queue[front]);
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % SIZE;
        }
    }
}
void display() {
    if (front == -1) {
        printf("Queue is Empty!\n");
    } else {
        printf("Queue: ");
        int i = front;
        while (1) {
            printf("%d ", queue[i]);
            if (i == rear) break;
            i = (i + 1) % SIZE;
        }
        printf("\n");
    }
}
int main() {
    int choice;
    while (1) {
        printf("\n1.Enqueue\n2.Dequeue\n3.Display\n4.Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: enqueue(); break;
            case 2: dequeue(); break;
            case 3: display(); break;
            case 4: return 0;
```

```
            default: printf("Invalid choice!\n");
        }
    }
}
```

OUTPUT



Q4]

ALGORITHM

Start
Initialize a pointer head = NULL to represent the start of the linked list.
Display Menu:
a. Insertion:
i. At the beginning
ii. At the end
iii. At a given position
b. Deletion:
i. At the beginning
ii. At the end
iii. At a given position
c. Search
d. Exit
Input user choice.
Perform the selected operation:
Insertion:
At the beginning: Create a new node, link it to head, and update head.
At the end: Traverse the list, create a new node, and update the last node's link.
At a position: Traverse to the desired position, insert the new node, and adjust links.
Deletion:
At the beginning: Update head to the next node and free the first node.
At the end: Traverse to the second last node, free the last node, and update the link.
At a position: Traverse to the node before the target, adjust links, and free the target.
Search: Traverse the list, compare each node's value, and display the position if found.
Repeat until Exit.
End

CODE

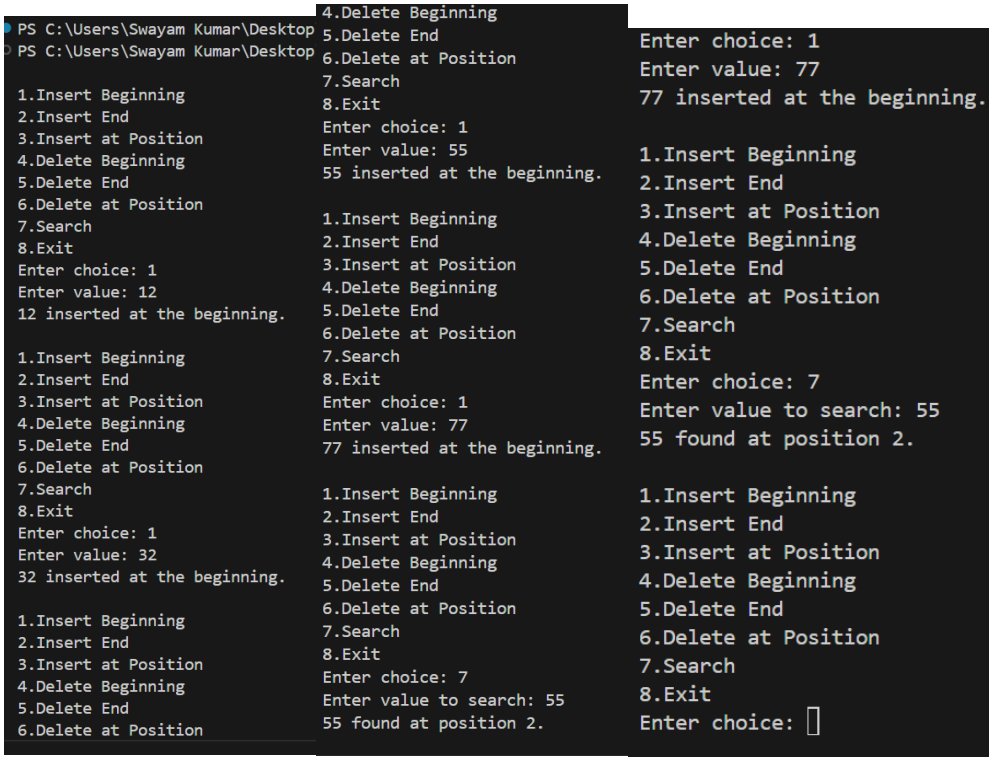
```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* head = NULL;
void insertAtBeginning(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
    head = newNode;
    printf("%d inserted at the beginning.\n", value);
}
void insertAtEnd(int value) {
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    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", value);
}
void insertAtPosition(int value, int pos) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if (pos == 1) {
        newNode->next = head;
        head = newNode;
    } else {
        struct Node* temp = head;
        for (int i = 1; i < pos - 1 && temp != NULL; i++) temp = temp->next;
        if (temp == NULL) {
            printf("Invalid position.\n");
            free(newNode);
            return;
        }
        newNode->next = temp->next;
        temp->next = newNode;
    }
    printf("%d inserted at position %d.\n", value, pos);
}
void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty.\n");
    } else {
        struct Node* temp = head;
        head = head->next;
        printf("%d deleted from the beginning.\n", temp->data);
        free(temp);
    }
}
void deleteAtEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
    } else if (head->next == NULL) {
        printf("%d deleted from the end.\n", head->data);
        free(head);
        head = NULL;
```

```c
    } else {
        struct Node* temp = head;
        while (temp->next->next != NULL) temp = temp->next;
        printf("%d deleted from the end.\n", temp->next->data);
        free(temp->next);
        temp->next = NULL;
    }
}
void deleteAtPosition(int pos) {
    if (head == NULL) {
        printf("List is empty.\n");
    } else if (pos == 1) {
        struct Node* temp = head;
        head = head->next;
        printf("%d deleted from position %d.\n", temp->data, pos);
        free(temp);
    } else {
        struct Node* temp = head;
        for (int i = 1; i < pos - 1 && temp != NULL; i++) temp = temp->next;
        if (temp == NULL || temp->next == NULL) {
            printf("Invalid position.\n");
        } else {
            struct Node* toDelete = temp->next;
            temp->next = toDelete->next;
            printf("%d deleted from position %d.\n", toDelete->data, pos);
            free(toDelete);
        }
    }
}
void search(int value) {
    struct Node* temp = head;
    int pos = 1;
    while (temp != NULL) {
        if (temp->data == value) {
            printf("%d found at position %d.\n", value, pos);
            return;
        }
        temp = temp->next;
        pos++;
    }
    printf("%d not found in the list.\n", value);
}
int main() {
    int choice, value, pos;
    while (1) {
        printf("\n1.Insert Beginning\n2.Insert End\n3.Insert at Position\n");
        printf("4.Delete Beginning\n5.Delete End\n6.Delete at Position\n");
        printf("7.Search\n8.Exit\nEnter choice: ");
```

```
        scanf("%d", &choice);
        switch (choice) {
            case 1: printf("Enter value: "); scanf("%d", &value);
insertAtBeginning(value); break;
            case 2: printf("Enter value: "); scanf("%d", &value);
insertAtEnd(value); break;
            case 3: printf("Enter value and position: "); scanf("%d%d",
&value, &pos); insertAtPosition(value, pos); break;
            case 4: deleteAtBeginning(); break;
            case 5: deleteAtEnd(); break;
            case 6: printf("Enter position: "); scanf("%d", &pos);
deleteAtPosition(pos); break;
            case 7: printf("Enter value to search: "); scanf("%d", &value);
search(value); break;
            case 8: return 0;
            default: printf("Invalid choice!\n");
        }
    }
}
```

OUTPUT



Q5] 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

Insertion:
Beginning:
Create a new node, set its next to head, and update head.
End:
Traverse to the last node, set its next to the new node, and link back.
At Position:
Traverse to the desired position, update prev and next pointers to insert the new node.
Deletion:
Beginning:
Update head to the next node and free the current node.
End:
Traverse to the last node, unlink it, and free the node.
At Position:
Traverse to the node at the position, unlink it, and free it.
Search:
Traverse the list, compare each node's data with the target value, and return the position if found.
Display:
Traverse the list and print each node's data.
Repeat Menu:
Display options, take user input, and perform the corresponding operation until the user exits.

CODE

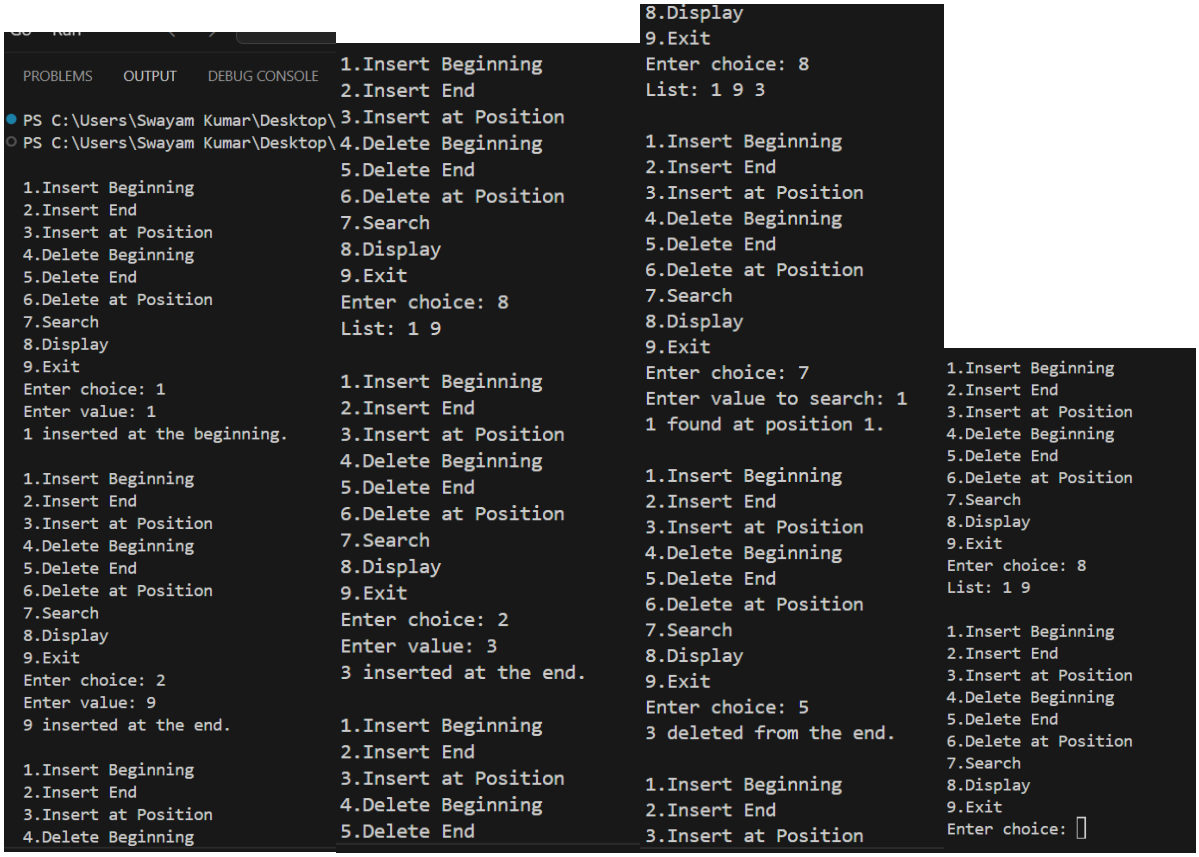
```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node *prev, *next;
};
struct Node *head = NULL;
void insertAtBeginning(int value) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = head;
    if (head != NULL) head->prev = newNode;
    head = newNode;
    printf("%d inserted at the beginning.\n", value);
}
void insertAtEnd(int value) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    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", value);
}
void insertAtPosition(int value, int pos) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    if (pos == 1) {
        newNode->prev = NULL;
        newNode->next = head;
        if (head != NULL) head->prev = newNode;
        head = newNode;
    } else {
        struct Node *temp = head;
        for (int i = 1; i < pos - 1 && temp != NULL; i++) temp = temp->next;
        if (temp == NULL) {
            printf("Invalid position.\n");
            free(newNode);
            return;
        }
        newNode->next = temp->next;
        newNode->prev = temp;
        if (temp->next != NULL) temp->next->prev = newNode;
        temp->next = newNode;
    }
    printf("%d inserted at position %d.\n", value, pos);
}
void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty.\n");
    } else {
        struct Node *temp = head;
        head = head->next;
        if (head != NULL) head->prev = NULL;
        printf("%d deleted from the beginning.\n", temp->data);
        free(temp);
    }
}
void deleteAtEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
```

```
    } else if (head->next == NULL) {
        printf("%d deleted from the end.\n", head->data);
        free(head);
        head = NULL;
    } else {
        struct Node *temp = head;
        while (temp->next != NULL) temp = temp->next;
        printf("%d deleted from the end.\n", temp->data);
        temp->prev->next = NULL;
        free(temp);
    }
}
void deleteAtPosition(int pos) {
    if (head == NULL) {
        printf("List is empty.\n");
    } else if (pos == 1) {
        struct Node *temp = head;
        head = head->next;
        if (head != NULL) head->prev = NULL;
        printf("%d deleted from position %d.\n", temp->data, pos);
        free(temp);
    } else {
        struct Node *temp = head;
        for (int i = 1; i < pos && temp != NULL; i++) temp = temp->next;
        if (temp == NULL) {
            printf("Invalid position.\n");
        } else {
            printf("%d deleted from position %d.\n", temp->data, pos);
            if (temp->next != NULL) temp->next->prev = temp->prev;
            if (temp->prev != NULL) temp->prev->next = temp->next;
            free(temp);
        }
    }
}
void search(int value) {
    struct Node *temp = head;
    int pos = 1;
    while (temp != NULL) {
        if (temp->data == value) {
            printf("%d found at position %d.\n", value, pos);
            return;
        }
        temp = temp->next;
        pos++;
    }
    printf("%d not found in the list.\n", value);
}
void display() {
```

```
    if (head == NULL) {
        printf("List is empty.\n");
    } else {
        struct Node *temp = head;
        printf("List: ");
        while (temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}
int main() {
    int choice, value, pos;
    while (1) {
        printf("\n1.Insert Beginning\n2.Insert End\n3.Insert at Position\n");
        printf("4.Delete Beginning\n5.Delete End\n6.Delete at Position\n");
        printf("7.Search\n8.Display\n9.Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: printf("Enter value: "); scanf("%d", &value);
insertAtBeginning(value); break;
            case 2: printf("Enter value: "); scanf("%d", &value);
insertAtEnd(value); break;
            case 3: printf("Enter value and position: "); scanf("%d%d",
&value, &pos); insertAtPosition(value, pos); break;
            case 4: deleteAtBeginning(); break;
            case 5: deleteAtEnd(); break;
            case 6: printf("Enter position: "); scanf("%d", &pos);
deleteAtPosition(pos); break;
            case 7: printf("Enter value to search: "); scanf("%d", &value);
search(value); break;
            case 8: display(); break;
            case 9: return 0;
            default: printf("Invalid choice!\n");
        }
    }
}
```

OUTPUT


PROBLEMS
OUTPUT
DEBUG CONSOLE
PS C:\Users\Swayam Kumar\Desktop\
PS C:\Users\Swayam Kumar\Desktop\
1.Insert Beginning
2.Insert End
3.Insert at Position
4.Delete Beginning
5.Delete End
6.Delete at Position
7.Search
8.Display
9.Exit
Enter choice: 1
Enter value: 1
1 inserted at the beginning.
1.Insert Beginning
2.Insert End
3.Insert at Position
4.Delete Beginning
5.Delete End
6.Delete at Position
7.Search
8.Display
9.Exit
Enter choice: 2
Enter value: 9
9 inserted at the end.
1.Insert Beginning
2.Insert End
3.Insert at Position
4.Delete Beginning
1.Insert Beginning
2.Insert End
3.Insert at Position
4.Delete Beginning
5.Delete End
6.Delete at Position
7.Search
8.Display
9.Exit
Enter choice: 8
List: 1 9
1.Insert Beginning
2.Insert End
3.Insert at Position
4.Delete Beginning
5.Delete End
6.Delete at Position
7.Search
8.Display
9.Exit
Enter choice: 2
Enter value: 3
3 inserted at the end.
1.Insert Beginning
2.Insert End
3.Insert at Position
4.Delete Beginning
5.Delete End
8.Display
9.Exit
Enter choice: 8
List: 1 9 3
1.Insert Beginning
2.Insert End
3.Insert at Position
4.Delete Beginning
5.Delete End
6.Delete at Position
7.Search
8.Display
9.Exit
Enter choice: 7
Enter value to search: 1
1 found at position 1.
1.Insert Beginning
2.Insert End
3.Insert at Position
4.Delete Beginning
5.Delete End
6.Delete at Position
7.Search
8.Display
9.Exit
Enter choice: 5
3 deleted from the end.
1.Insert Beginning
2.Insert End
3.Insert at Position
1.Insert Beginning
2.Insert End
3.Insert at Position
4.Delete Beginning
5.Delete End
6.Delete at Position
7.Search
8.Display
9.Exit
Enter choice: 8
List: 1 9
1.Insert Beginning
2.Insert End
3.Insert at Position
4.Delete Beginning
5.Delete End
6.Delete at Position
7.Search
8.Display
9.Exit
Enter choice: