DATA STRUCTURES AND ALGORITHMS

<u>DIGITAL ASSIGNMENT-1</u>

Name - Kavya Gupta

Registration no. : 24BBS0208

Course Code : CBS1003

Q1. Write a menu driven program to implement the following operations on stack.

a. PUSH ()

b. POP ()

c. Display ()

Algorithm:

Define a Stack:

- Impletening stack using array
    - Define max(maximum size of stack)
    - Initialize top=-1
    - Define an array with size max

Push Operation:

- **Input:** Stack s, element data.
- **Steps:**
    1. Check if the stack is full (top == max - 1).
        - If yes, print "Stack Overflow" and return.
    2. Increment top.
    3. Add data to s.data[top].

Pop Operation:

- **Input:** Stack s.
- **Steps:**
    1. Check if the stack is empty (top == -1).
        - If yes, print "Stack Underflow" and return.
    2. Retrieve the element at s.data[top].
    3. Decrement top.
    4. Return the retrieved element.

Display Stack:

- **Input:** Stack s.
- **Steps:**
    1. If the stack is empty (top == -1), print "Stack is empty".
    2. Otherwise, print elements from s.data[top] to s.data[0].

Main Function:

1. Initialize a stack s with max.
2. Provide menu options:
    - 1: Push an element.
    - 2: Pop an element.
    - 3: Display the stack.
    - 4: Exit
3. Perform the chosen operation and display the updated stack.

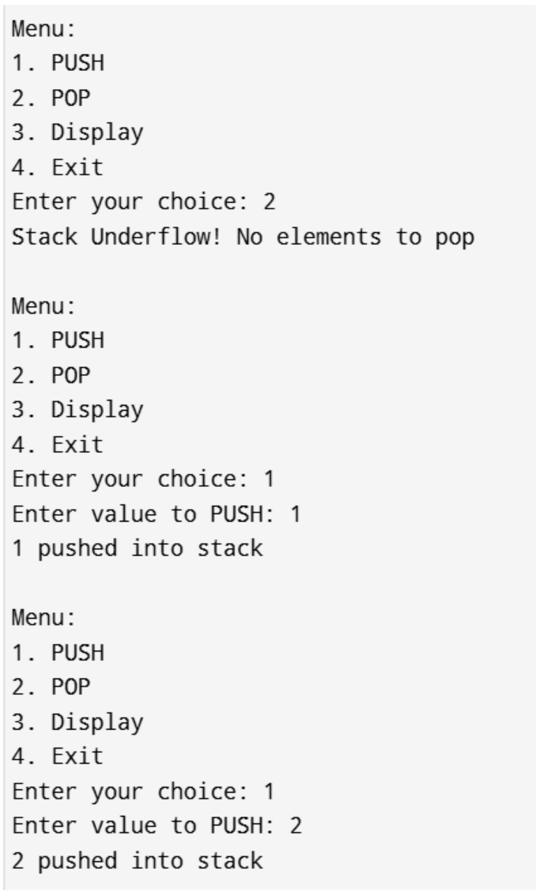
Program:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 3

int stack[MAX];

int top = -1;

int isFull() {

   return top == MAX - 1;}
```

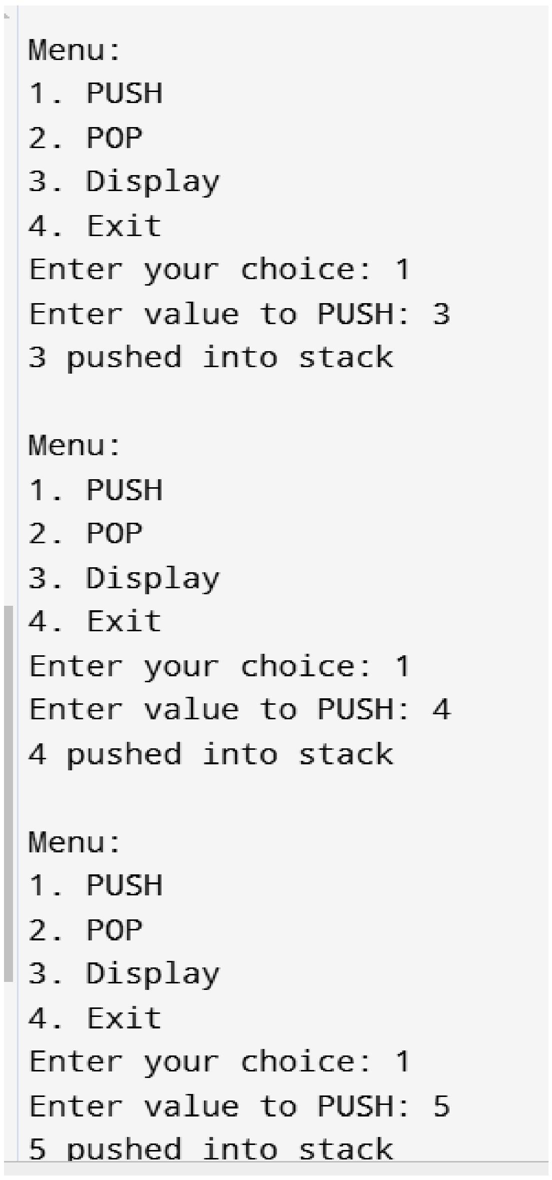
```
int isEmpty() {
    return top == -1;
}
void PUSH(int value) {
    if (isFull()) {
        printf("Stack Overflow! Cannot push %d\n", value);
    }
else {
        top++;
        stack[top] = value;
        printf("%d pushed into stack\n", value);
    }
}
int POP() {
    if (isEmpty()) {
        printf("Stack Underflow! No elements to pop\n");
        return -1; // Indicating stack underflow
    }
else {
        int value = stack[top];
        top--;
        return value;
    }
}
void Display() {
    if (isEmpty()) {
        printf("Stack is empty!\n");
    }
```

```
else {
        printf("Stack elements are:\n");
        for (int i = top; i >= 0; i--) {
            printf("%d ", stack[i]);
        }
        printf("\n");
    }
}

int main()
 {
    int choice, value;
    while (1) {
        printf("\nMenu:\n");
        printf("1. PUSH\n");
        printf("2. POP\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to PUSH: ");
                scanf("%d", &value);
                PUSH(value);
                break;
            case 2:
                value = POP();
                if (value != -1) {
                    printf("%d popped from stack\n", value);
                }
                break;
            case 3:
                Display();
                break;
            case 4:
                printf("Exiting the program.\n");
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

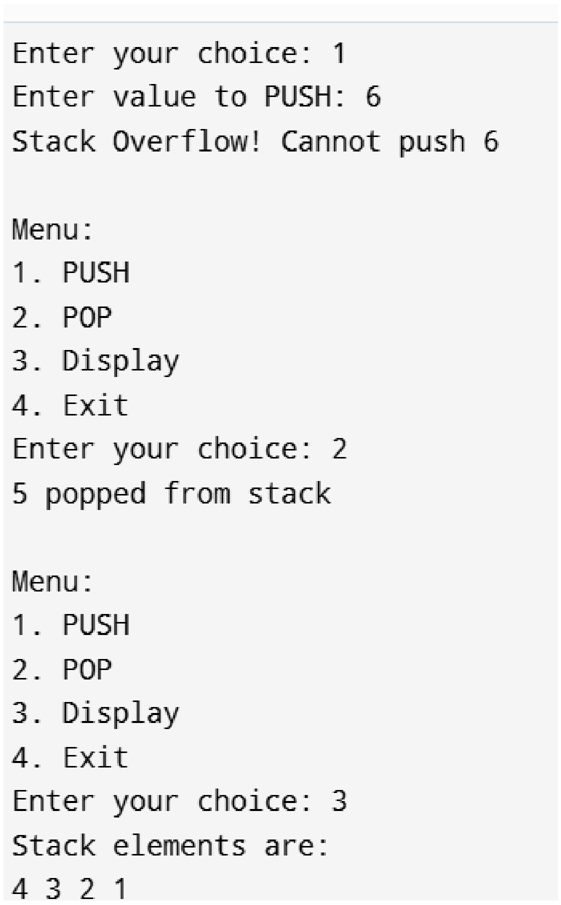
OUTPUT:



```
Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 2
Stack Underflow! No elements to pop

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to PUSH: 1
1 pushed into stack

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to PUSH: 2
2 pushed into stack
```



```
Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to PUSH: 3
3 pushed into stack

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to PUSH: 4
4 pushed into stack

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 1
Enter value to PUSH: 5
5 pushed into stack
```



```
Enter your choice: 1
Enter value to PUSH: 6
Stack Overflow! Cannot push 6

Menu:
1. PUSH
2. POP
3. Display
4. Exit
Enter your choice: 2
5 popped from stack

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

Q2. Write a menu driven program to implement the following operations on Queue:

a. Enqueue ()

b. Dequeue ()

c. Display ()

ALGORITHM:

Define a Queue:

- A structure Queue with:
    - data: An array to store queue elements.
    - front: An integer to track the index of the first element.
    - rear: An integer to track the index of the last element.
    - max: The maximum capacity of the queue.

Enqueue Operation:

- **Input:** Queue q, element data.
- **Steps:**
    1. Check if the queue is full (rear == max - 1).
        - If yes, print "Queue Overflow" and return.
    2. If front == -1, set front = 0.
    3. Increment rear.
    4. Add data to q.data[rear].

Dequeue Operation:

- **Input:** Queue q.
- **Steps:**
    1. Check if the queue is empty (front == -1 or front > rear).
        - If yes, print "Queue Underflow" and return.
    2. Retrieve the element at q.data[front].
    3. Increment front.
    4. If front > rear, reset front and rear to -1.
    5. Return the retrieved element.

Display Queue:

- **Input:** Queue q.
- **Steps:**
    1. If the queue is empty (front == -1), print "Queue is empty".
    2. Otherwise, print elements from q.data[front] to q.data[rear].

Main Function:

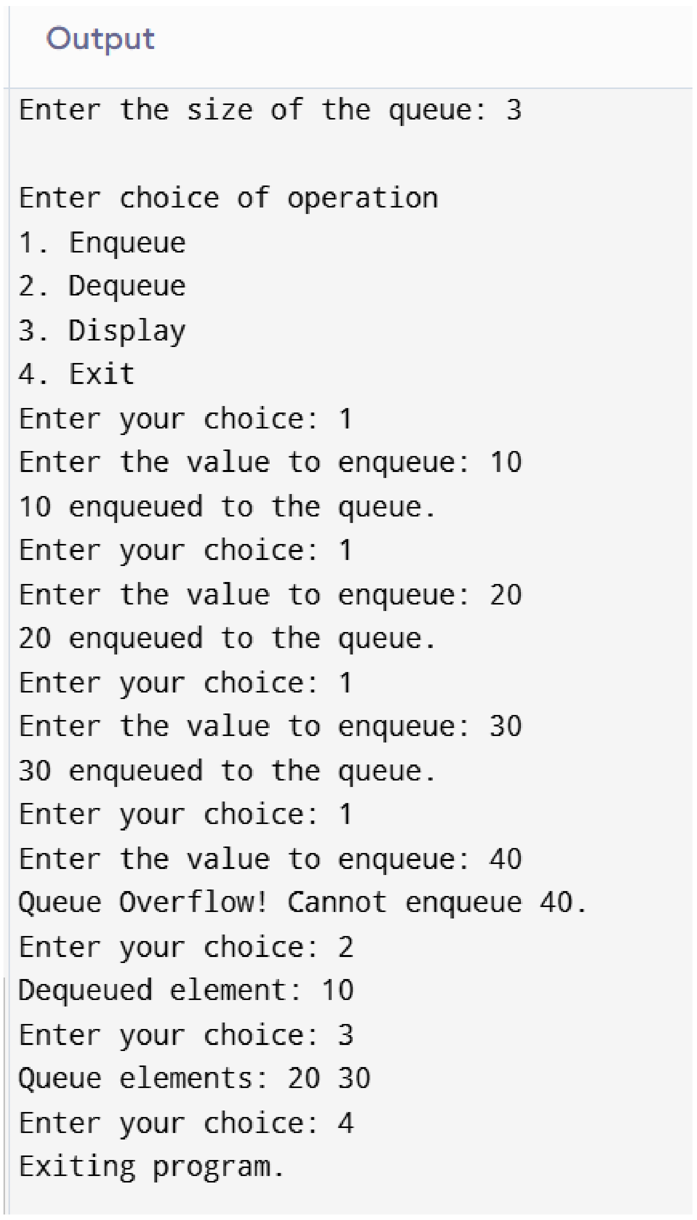
1. Initialize a queue q with maxSize.
2. Provide menu options:
    - 1: Enqueue an element.
    - 2: Dequeue an element.
    - 3: Display the queue.
3. Perform the chosen operation and display the updated queue.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
   int *items;
   int front, rear;
   int max;
}Queue;
void initializeQueue(Queue *q, int size) {
   q->max = size;
   q->items = (int *)malloc(size * sizeof(int));
   q->front = -1;
   q->rear = -1;
}
int isFull(Queue *q) {
   return (q->rear + 1) % q->max == 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->max;
   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->max;
   }
   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->max;
   }
   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) {
            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;
}
```

OUTPUT:


Output

```
Enter the size of the queue: 3

Enter choice of operation
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 10
10 enqueued to the queue.
Enter your choice: 1
Enter the value to enqueue: 20
20 enqueued to the queue.
Enter your choice: 1
Enter the value to enqueue: 30
30 enqueued to the queue.
Enter your choice: 1
Enter the value to enqueue: 40
Queue Overflow! Cannot enqueue 40.
Enter your choice: 2
Dequeued element: 10
Enter your choice: 3
Queue elements: 20 30
Enter your choice: 4
Exiting program.
```

Q3. Write a menu driven program to implement the following operations on circular Queue:

a. Enqueue ()

b. Dequeue ()

c. Display ()

ALGORITHM:

Define a Circular Queue:

- A structure CircularQueue with:
  - data: An array to store queue elements.
  - front: An integer to track the index of the first element.
  - rear: An integer to track the index of the last element.
  - maxSize: The maximum capacity of the queue.

Enqueue Operation:

- **Input:** CircularQueue cq, element data.
- **Steps:**
  1. Check if the queue is full ((rear + 1) % maxSize == front).
     - If yes, print "Queue Overflow" and return.
  2. If front == -1, set front = 0.
  3. Increment rear using rear = (rear + 1) % maxSize.
  4. Add data to cq.data[rear].

Dequeue Operation:

- **Input:** CircularQueue cq.
- **Steps:**
  1. Check if the queue is empty (front == -1).
     - If yes, print "Queue Underflow" and return.
  2. Retrieve the element at cq.data[front].
  3. If front == rear, reset front and rear to -1.
  4. Otherwise, increment front using front = (front + 1) % maxSize.
  5. Return the retrieved element.

Display Circular Queue:

- **Input:** CircularQueue cq.
- **Steps:**
    1. If the queue is empty (front == -1),

       print "Queue is empty".
    2. Otherwise:
        - Start from cq.data[front] and traverse circularly until rear.

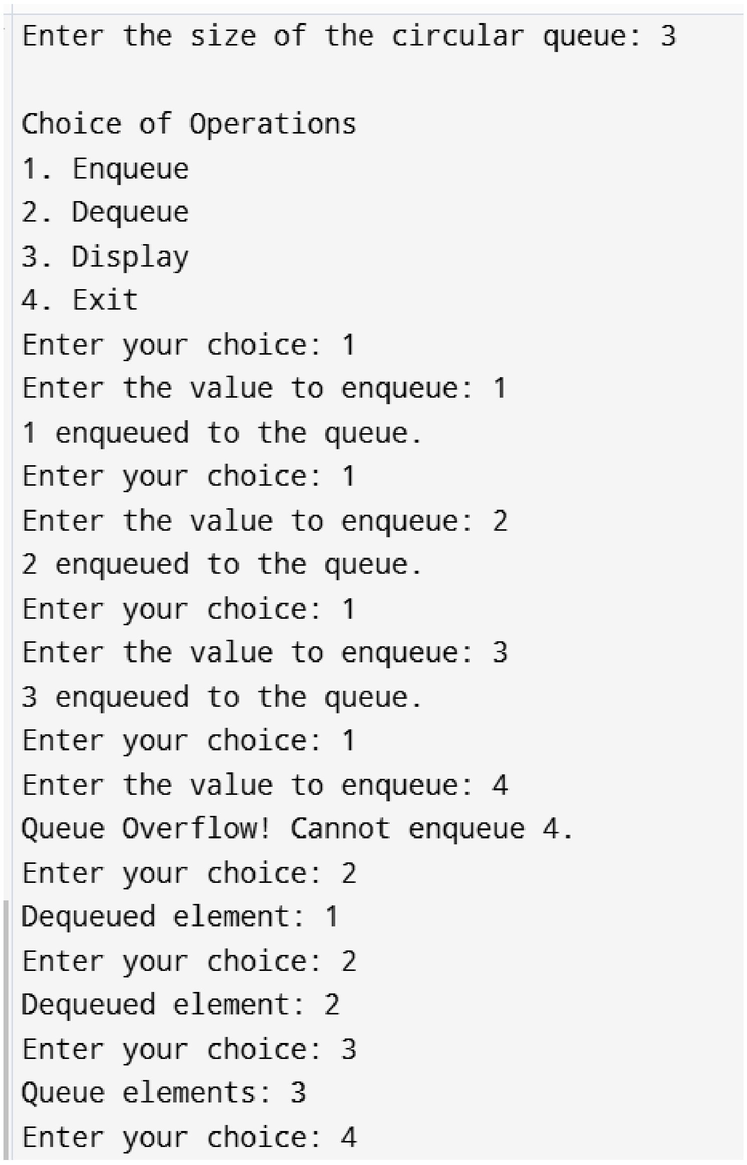
Main Function:

1. Initialize a circular queue cq with maxSize.
2. Provide menu options:
    - 1: Enqueue an element.
    - 2: Dequeue an element.
    - 3: Display the queue.
3. Perform the chosen operation and display the updated queue.

PROGRAM:

```
#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;
}
```

OUTPUT:



```
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: 1
1 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: 3
3 enqueued to the queue.
Enter your choice: 1
Enter the value to enqueue: 4
Queue Overflow! Cannot enqueue 4.
Enter your choice: 2
Dequeued element: 1
Enter your choice: 2
Dequeued element: 2
Enter your choice: 3
Queue elements: 3
Enter your choice: 4
```

Q4. 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:

Define a Node:

- Create a structure node with two fields:
  - data: Stores the value of the node.
  - next: Points to the next node in the list.

Insert at Beginning (insertAtBeg):

- Input: start (head of the list), data (data to be inserted).
- Allocate memory for a new node.
- Set the new node's data to the input data.
- Set the new node's next to point to the current start.
- Update start to point to the new node.
- Return the updated start.

Insert at End (insertAtEnd):

- Input: start (head of the list), data (data to be inserted).
- Create a new node with the input data.
- Traverse the list to find the last node (node where next is NULL).
- Set the last node's next to the new node.
- Set the new node's next to NULL.
- Return the updated start.

Insert at Position (insertAtPos):

- Input: start (head of the list), data (data to be inserted), pos (position).
- If the position is 0, call insertAtBeg to insert at the beginning and return.
- Traverse the list to find the node at position pos-1.
- If the node is NULL, print an error message and return.
- Insert the new node after the node at pos-1 by updating the pointers.

- Return the updated start.

Delete at Beginning (deleteAtBeg):

- Input: start (head of the list).
- If the list is empty (start == NULL), print an error message and return.
- Store the current start in a temporary pointer.
- Set start to the next node (start->next).
- Free the memory of the temporary node.
- Return the updated start.

Delete at End (deleteAtEnd):

- Input: start (head of the list).
- If the list is empty (start == NULL), print an error message and return.
- If the list has only one node (start->next == NULL), free the node and set start to NULL.
- Otherwise, traverse the list to find the second last node.
- Set the second last node's next to NULL and free the last node.
- Return the updated start.

Delete at Position (deleteAtPos):

- Input: start (head of the list), pos (position of the node to delete).
- If the list is empty (start == NULL), print an error message and return.
- If the position is 0, call deleteAtBeg to delete the first node and return.
- Traverse the list to find the node at position pos.
- If the node is NULL, print an error message and return.
- Update the next pointer of the previous node to skip the node to be deleted.
- Free the memory of the deleted node.
- Return the updated start.

Search Operation:

- Input: start (head of the list), data (data to search).
- Traverse the list, checking each node's data.
- If the data is found, print the position and return.
- If the end of the list is reached without finding the data, print an error message.

Display the List:

- Input: start (head of the list).
- Traverse the list, printing each node's data.
- End the display with NULL to indicate the end of the list.

Main Function:

- Initialize start as NULL.
- Provide a menu of operations:
  - 1: Insert at beginning.
  - 2: Insert at end.
  - 3: Insert at a specific position.
  - 4: Delete the first node.
  - 5: Delete the last node.
  - 6: Delete at a specific position.
  - 7: Search for an element.
- Execute the corresponding function based on user input.
- Display the updated list after each operation.
- Allow the user to continue or exit based on input.

PROGRAM:

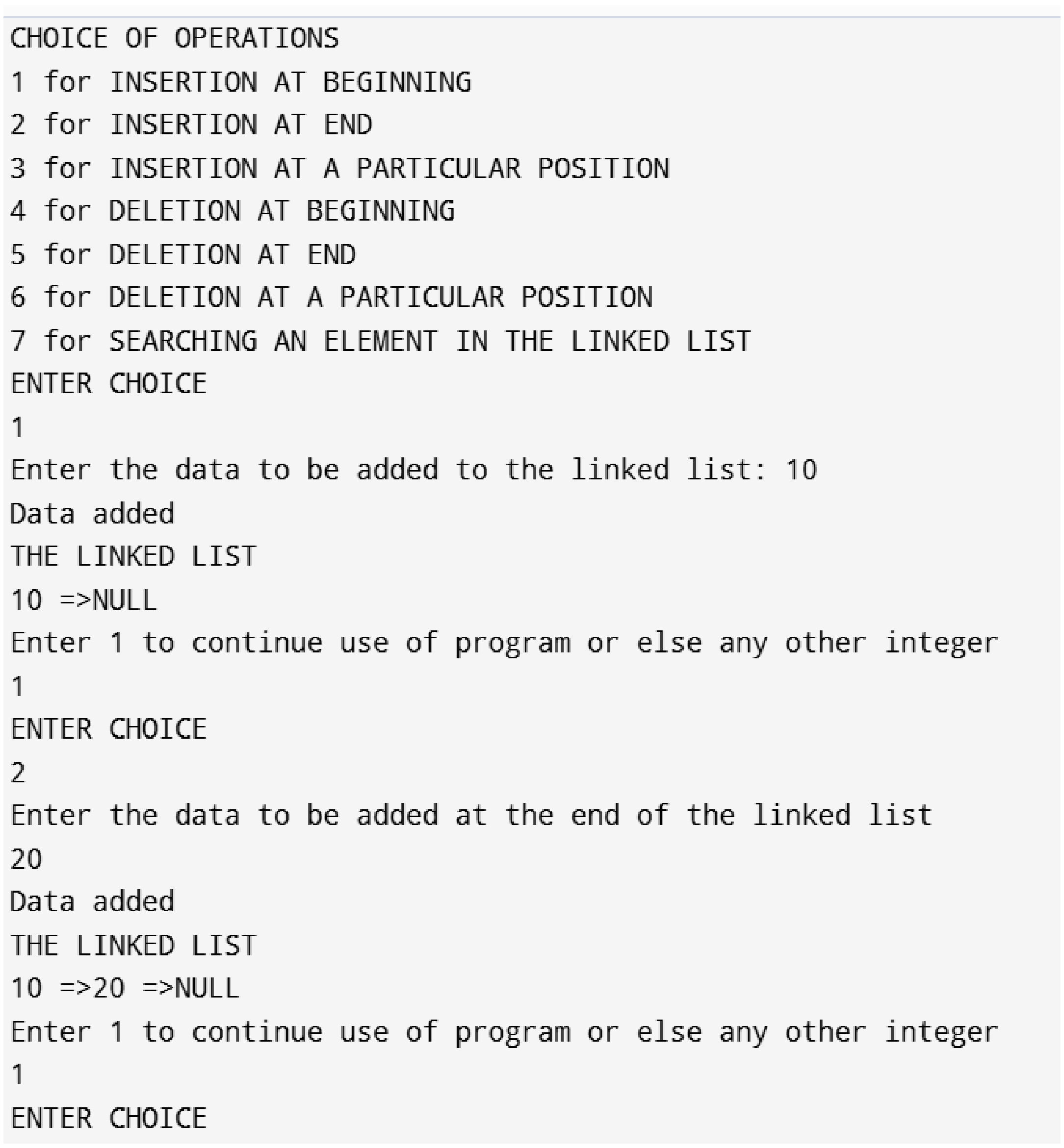
```
#include<stdio.h>
#include<stdlib.h>
struct node{
   int data;
   struct node* next;
};
struct node* insertAtBeg(struct node* start, int data)
{
   struct node* temp=(struct node*)malloc(sizeof(struct node));
   temp->data=data;
   temp->next=start;
```

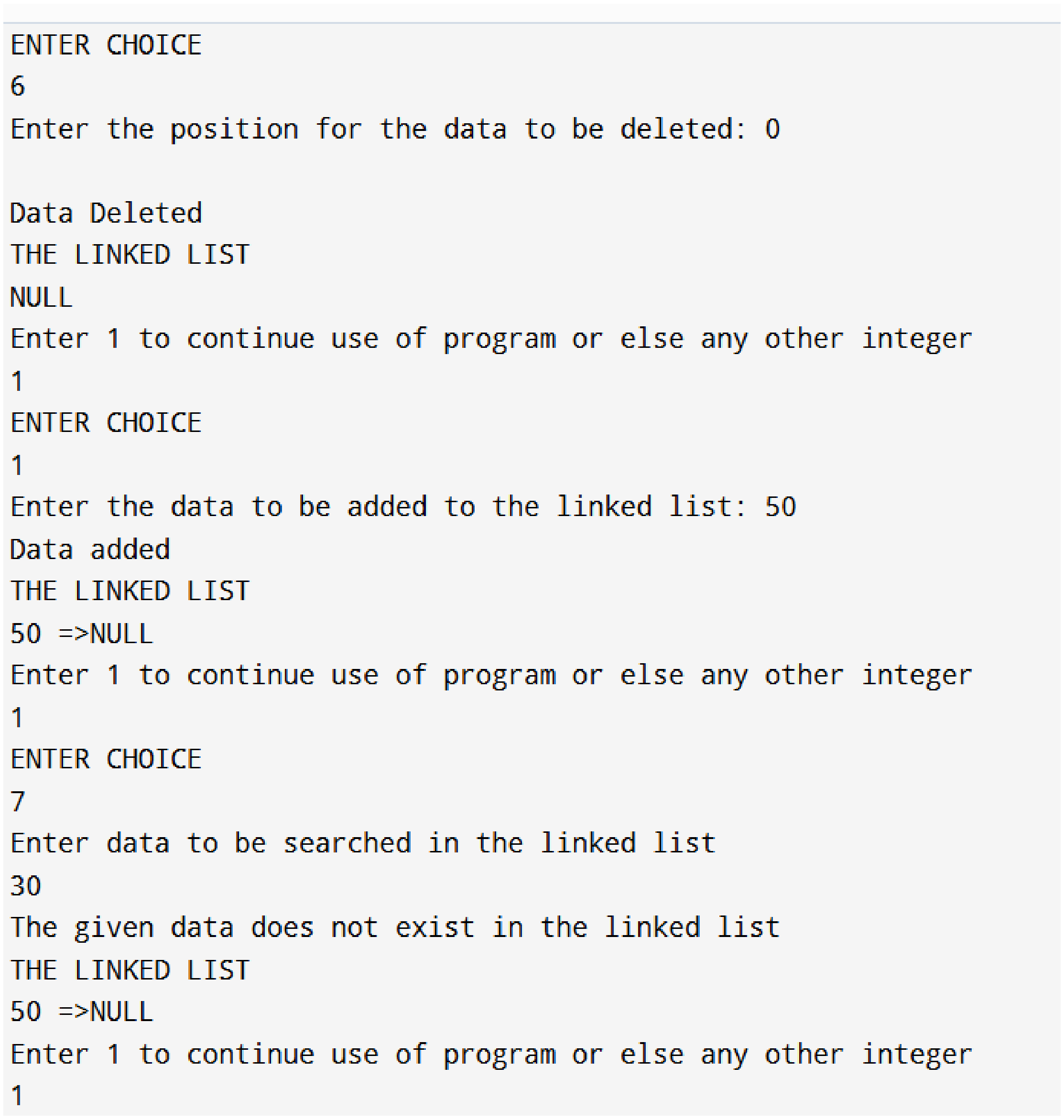
```
    start=temp;
    return start;
}
struct node* insertAtEnd(struct node* start, int data)
{
    struct node* temp=(struct node*)malloc(sizeof(struct node));
    struct node* p=start;
    while(p->next!=NULL)
    {
        p=p->next;
    }
    temp->data=data;
    p->next=temp;
    temp->next=NULL;
    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 %d position\n",counter);
            return;
        }
        p=p->next;
    }
    printf("The given data does not exist in the linked list\n");
    return;
}
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");
    return;
}
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:{
```

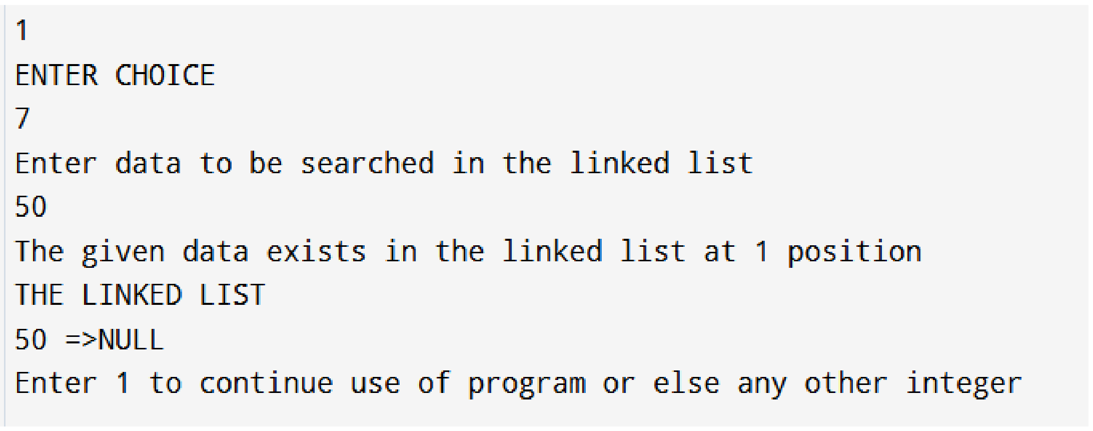
```
                start=deleteAtEnd(start);
                printf("Data deleted\n");
                break;
            }
            case 6:{
                int pos;
                printf("Enter the position for the data to be deleted: ");
                scanf("%d",&pos);
                start=deleteAtPos(start,pos);
                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;
}
```

OUTPUT:


```
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: 10
Data added
THE LINKED LIST
10 =>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
20
Data added
THE LINKED LIST
10 =>20 =>NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
ENTER CHOICE
3
Enter the data to be added
30

Enter the position at which data has to be added
2
Data added
THE LINKED LIST
10 =>30 =>20 =>NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
4
Data deleted
THE LINKED LIST
30 =>20 =>NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
5
Data deleted
THE LINKED LIST
30 =>NULL
Enter 1 to continue use of program or else any other integer
1
```



```
ENTER CHOICE
6
Enter the position for the data to be deleted: 0

Data Deleted
THE LINKED LIST
NULL
Enter 1 to continue use of program or else any other integer
1
ENTER CHOICE
1
Enter the data to be added to the linked list: 50
Data added
THE LINKED LIST
50 =>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
30
The given data does not exist in the linked list
THE LINKED LIST
50 =>NULL
Enter 1 to continue use of program or else any other integer
1
```



```
1
ENTER CHOICE
7
Enter data to be searched in the linked list
50
The given data exists in the linked list at 1 position
THE LINKED LIST
50 =>NULL
Enter 1 to continue use of program or else any other integer
```

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:

Define a Node:

- A node structure with three fields:
  - data: Stores the value of the node.
  - prev: Points to the previous node in the list.
  - next: Points to the next node in the list.

Create a New Node (createNewNode):

- Allocate memory for a new node.
- Set the node's data, prev, and next to the provided values (prev = NULL, next = NULL).
- Return the new node.

Insert at Beginning (insertAtBeg):

- Input: start (head of the list), tail (tail of the list), data (data to insert).
- Allocate memory for a new node.
- If the list is empty (start == NULL), set both start and tail to the new node.
- Otherwise, set the new node's next to start, and update the prev pointer of the current start to point to the new node.

- Set start to the new node.

Insert at End (insertAtEnd):

- Input: start (head of the list), tail (tail of the list), data (data to insert).
- Allocate memory for a new node.
- If the list is empty (start == NULL), set both start and tail to the new node.
- Otherwise, set the current tail's next to the new node and the new node's prev to the current tail.
- Set tail to the new node.

Insert at Position (insertAtPos):

- Input: start (head of the list), tail (tail of the list), data (data to insert), pos (position).
- If the position is 0, call insertAtBeg to insert at the beginning and return.
- Traverse the list until reaching the node at position pos-1.
- If the node is NULL, print an error and free the new node.
- Insert the new node after the node at position pos-1, updating the prev and next pointers of adjacent nodes.

Delete at Beginning (deleteAtBeg):

- Input: start (head of the list), tail (tail of the list).
- If the list is empty (start == NULL), print an error and return.
- Otherwise, set start to the next node, and update the prev pointer of the new start to NULL.
- If the list becomes empty (start == NULL), set tail to NULL.
- Free the old start node.

Delete at End (deleteAtEnd):

- Input: start (head of the list), tail (tail of the list).
- If the list is empty (tail == NULL), print an error and return.
- If the list has only one node (start == tail), set both start and tail to NULL.
- Otherwise, set tail to the previous node and set its next pointer to NULL.
- Free the old tail node.

Delete at Position (deleteAtPos):

- Input: start (head of the list), tail (tail of the list), pos (position).
- If the list is empty (start == NULL), print an error and return.
- If the position is 0, call deleteAtBeg to delete the first node and return.
- Traverse the list until reaching the node at position pos.
- If the node is NULL, print an error.
- If the node is the first (start), delete it using deleteAtBeg.
- If the node is the last (tail), delete it using deleteAtEnd.
- Otherwise, update the prev and next pointers of adjacent nodes and free the current node.

Search Operation (search):

- Input: start (head of the list), data (data to search).
- Traverse the list, checking each node's data.
- If the data is found, print the position and return.
- If the end of the list is reached without finding the data, print an error.

Display the List (display):

- Input: start (head of the list).
- Traverse the list and print each node's data from start to tail.
- Also display the list in reverse order, starting from tail to start.

Main Function:

- Initialize start and tail as NULL.
- Provide a menu with options:
    - 1: Insert at beginning.
    - 2: Insert at end.
    - 3: Insert at a specific position.
    - 4: Delete the first node.
    - 5: Delete the last node.
    - 6: Delete at a specific position.
    - 7: Search for an element.

- Execute the corresponding function based on user input.
- Display the updated list after each operation.
- Allow the user to continue or exit based on input.

PROGRAM:

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

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

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

// Insert at the beginning
void insertBeginning(struct Node **head, int data) {
    struct Node *newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        newNode->next = *head;
        (*head)->prev = newNode;
        *head = newNode;
    }
}

// Insert at the end
void insertEnd(struct Node **head, int data) {
    struct Node *newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node *last = *head;
        while (last->next != NULL) {
            last = last->next;
        }
        last->next = newNode;
        newNode->prev = last;
    }
}

// Insert at a given position
void insertAtPosition(struct Node **head, int data, int position) {
    if (position < 1) {
        printf("Position should be greater than or equal to 1.\n");
        return;
    }
    struct Node *newNode = createNode(data);
    if (position == 1) {
        insertBeginning(head, data);
        return;
    }

    struct Node *current = *head;
    int count = 1;
    while (current != NULL && count < position - 1) {
        current = current->next;
        count++;
    }

    if (current == NULL) {
        printf("Position out of range.\n");
        return;
    }

    newNode->next = current->next;
    newNode->prev = current;
    if (current->next != NULL) {
        current->next->prev = newNode;
    }
    current->next = newNode;
}

// Delete from the beginning
```

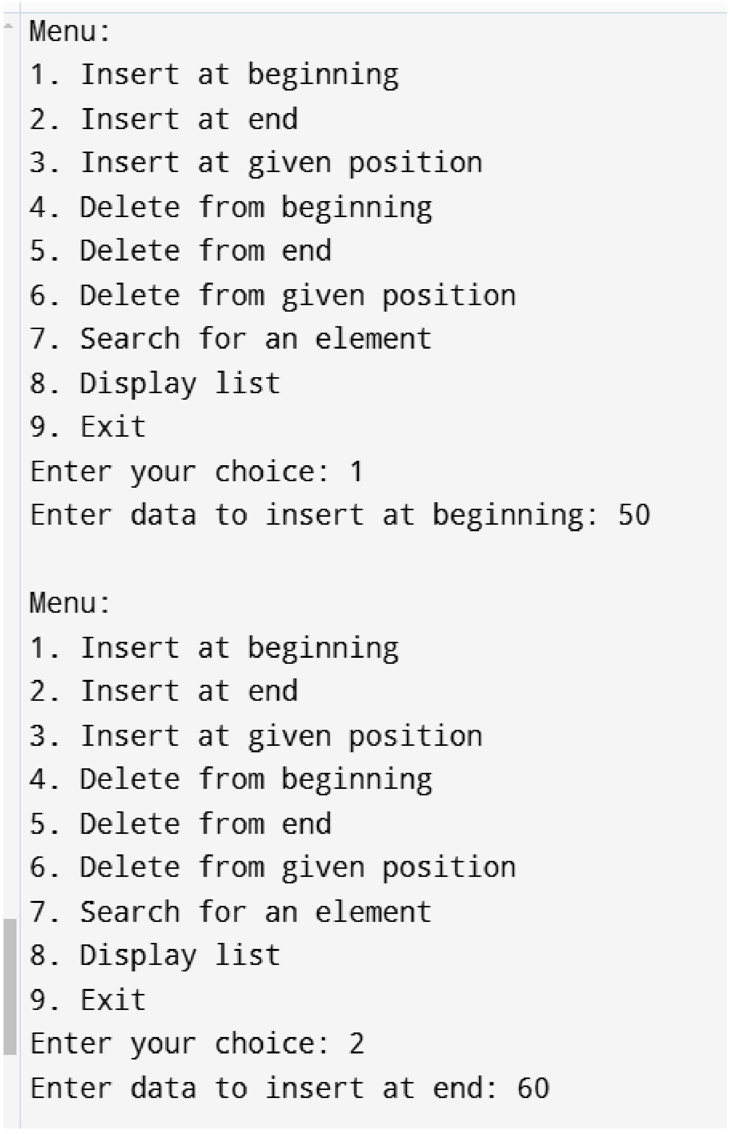
```
void deleteBeginning(struct Node **head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node *temp = *head;
    *head = (*head)->next;
    if (*head != NULL) {
        (*head)->prev = NULL;
    }
    free(temp);
}

// Delete from the end
void deleteEnd(struct Node **head) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node *last = *head;
    while (last->next != NULL) {
        last = last->next;
    }

    if (last->prev != NULL) {
        last->prev->next = NULL;
    } else {
        *head = NULL;
    }
    free(last);
}

// Delete at a given position
void deleteAtPosition(struct Node **head, int position) {
    if (position < 1) {
        printf("Position should be greater than or equal to 1.\n");
        return;
    }

    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node *current = *head;
    int count = 1;
    while (current != NULL && count < position) {
        current = current->next;
        count++;
    }

    if (current == NULL) {
        printf("Position out of range.\n");
        return;
    }

    if (current->prev != NULL) {
        current->prev->next = current->next;
    } else {
        *head = current->next;
    }

    if (current->next != NULL) {
        current->next->prev = current->prev;
    }

    free(current);
}

// Search for an element
void search(struct Node *head, int key) {
    struct Node *current = head;
    int position = 1;
    while (current != NULL) {
        if (current->data == key) {
            printf("Element %d found at position %d.\n", key, position);
            return;
        }
        current = current->next;
        position++;
    }
    printf("Element %d not found in the list.\n", key);
```

```c
}

// Display the list
void display(struct Node *head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node *current = head;
    while (current != NULL) {
        printf("%d <-> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

// Menu function
void menu() {
    struct Node *head = NULL;
    int choice, data, position;

    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 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 data to insert at beginning: ");
            scanf("%d", &data);
            insertBeginning(&head, data);
            break;
          case 2:
            printf("Enter data to insert at end: ");
            scanf("%d", &data);
            insertEnd(&head, data);
            break;
          case 3:
            printf("Enter data to insert: ");
            scanf("%d", &data);
            printf("Enter position to insert: ");
            scanf("%d", &position);
            insertAtPosition(&head, data, position);
            break;
          case 4:
```

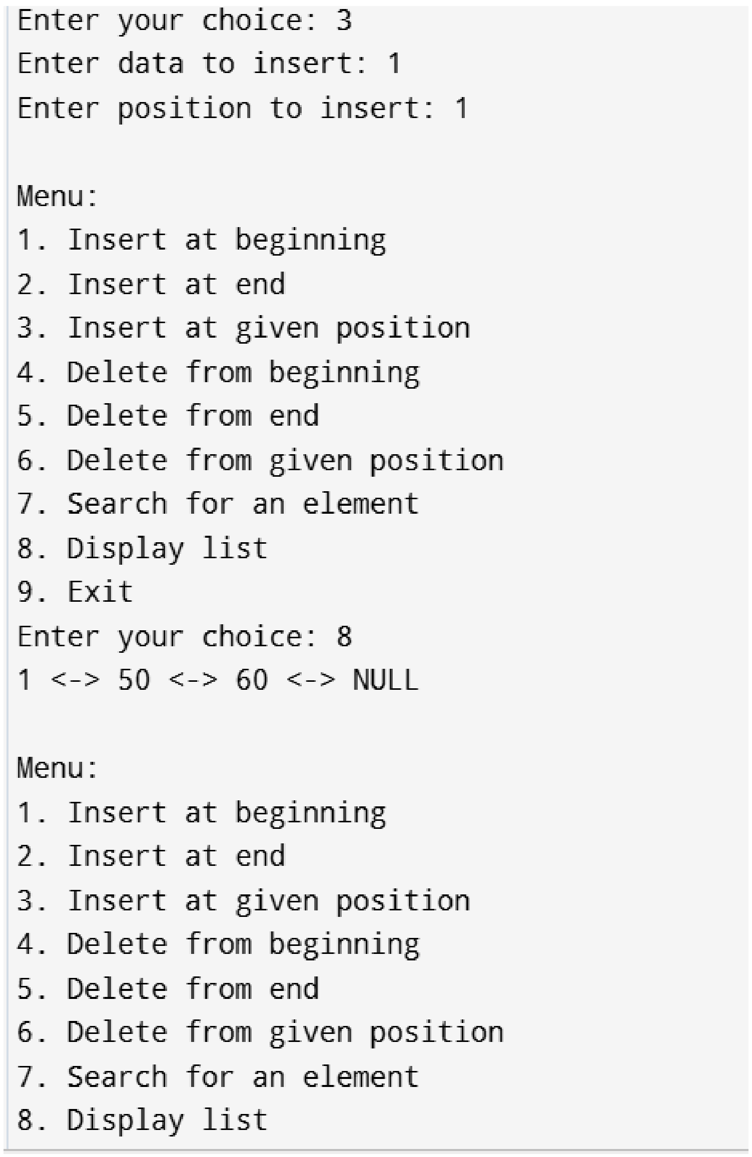
```
            deleteBeginning(&head);
            break;
        case 5:
            deleteEnd(&head);
            break;
        case 6:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(&head, position);
            break;
        case 7:
            printf("Enter element to search: ");
            scanf("%d", &data);
            search(head, data);
            break;
        case 8:
            display(head);
            break;
        case 9:
            printf("Exiting...\n");
            return;
        default:
            printf("Invalid choice, please try again.\n");
      }
   }
}

// Main function
```

```
int main() {
    menu();
    return 0;
}
```

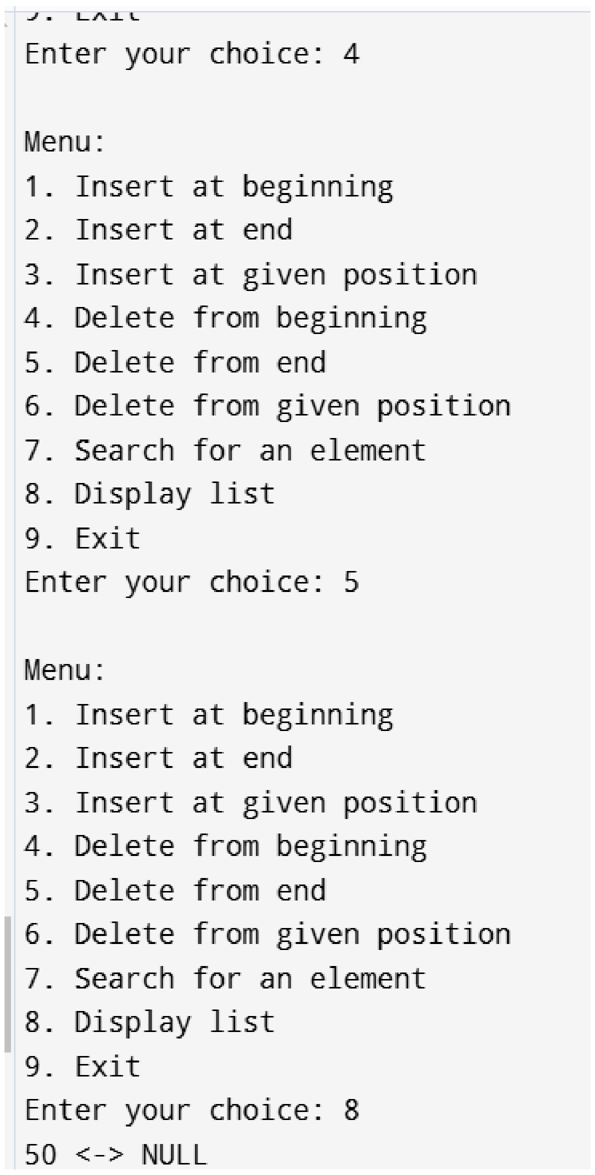
OUTPUT:



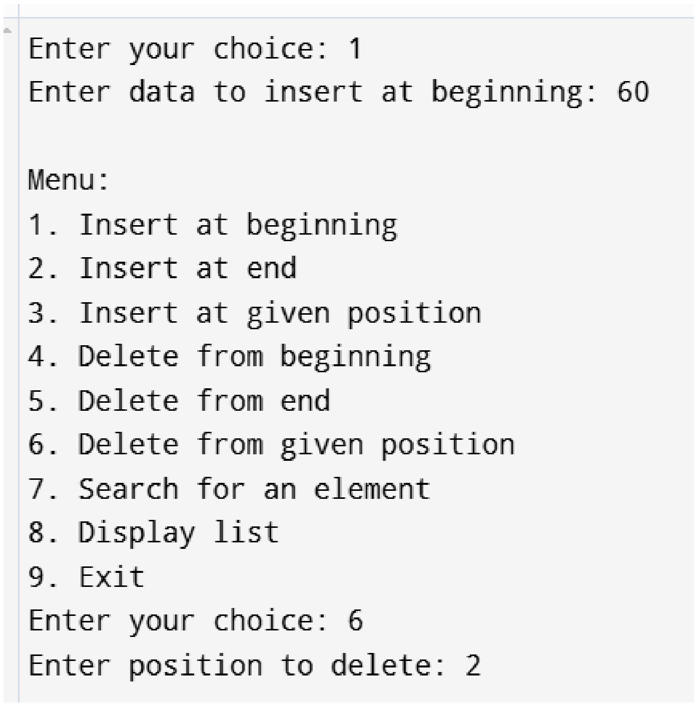
```
Menu:
1. Insert at beginning
2. Insert at end
3. Insert at given position
4. Delete from beginning
5. Delete from end
6. Delete from given position
7. Search for an element
8. Display list
9. Exit
Enter your choice: 1
Enter data to insert at beginning: 50

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 for an element
8. Display list
9. Exit
Enter your choice: 2
Enter data to insert at end: 60
```



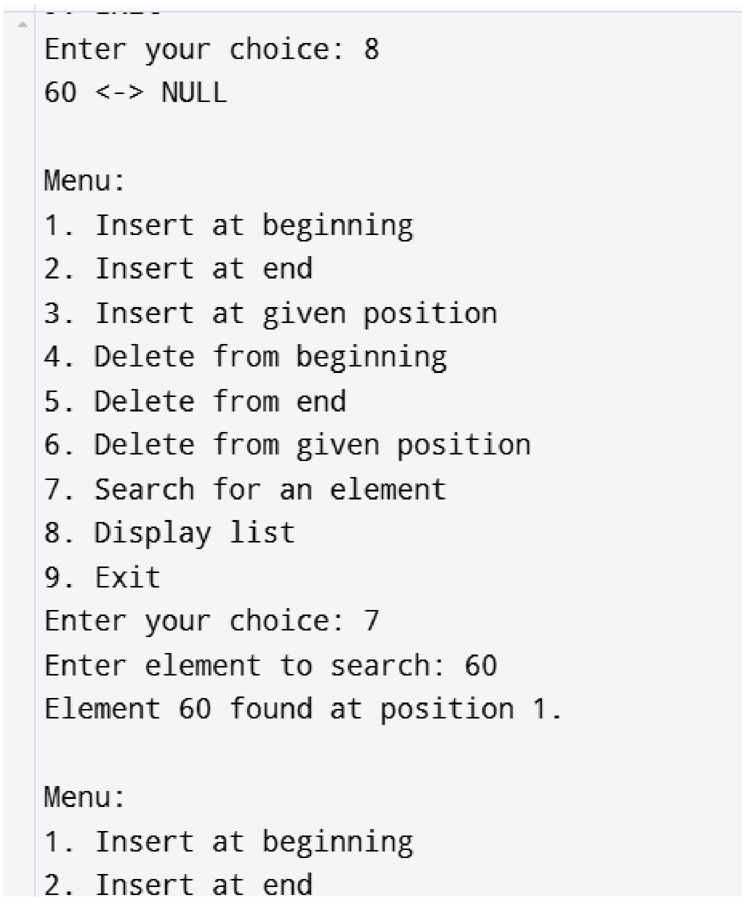
```
Enter your choice: 3
Enter data to insert: 1
Enter position to insert: 1

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 for an element
8. Display list
9. Exit
Enter your choice: 8
1 <-> 50 <-> 60 <-> NULL

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 for an element
8. Display list
```



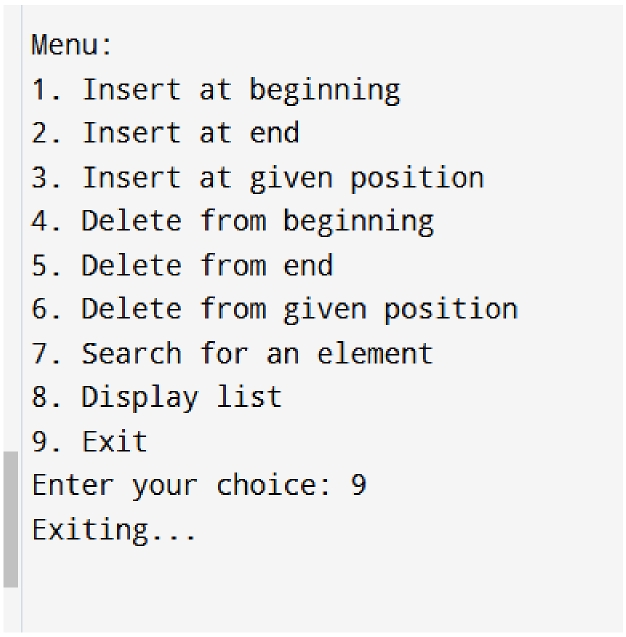
```
9. Exit
Enter your choice: 4

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 for an element
8. Display list
9. Exit
Enter your choice: 5

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 for an element
8. Display list
9. Exit
Enter your choice: 8
50 <-> NULL
```



```
Enter your choice: 1
Enter data to insert at beginning: 60

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 for an element
8. Display list
9. Exit
Enter your choice: 6
Enter position to delete: 2
```



```
Enter your choice: 8
60 <-> NULL

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 for an element
8. Display list
9. Exit
Enter your choice: 7
Enter element to search: 60
Element 60 found at position 1.

Menu:
1. Insert at beginning
2. Insert at 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 for an element
8. Display list
9. Exit
Enter your choice: 9
Exiting...
```